Scalable and Efficient Graph Algorithms and Analysis Techniques for Modern Machines

by

Quanquan C. Liu

S.B., Massachusetts Institute of Technology (2015)

M.Eng., Massachusetts Institute of Technology (2016)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2021

Author . . . . . .

Department of Electrical Engineering and Computer Science

August 27, 2021

Certified by . . . . . .

Erik D. Demaine

Professor of Electrical Engineering and Computer Science

Thesis Supervisor

Certified by . . . . . .

Julian Shun

Douglas T. Ross Career Development Professor of Software Technology

Associate Professor of Electrical Engineering and Computer Science

Thesis Supervisor

Accepted by . . . . . .

Leslie A. Kolodziejski

Professor of Electrical Engineering and Computer Science

Chair, Department Committee on Graduate Students

To my family.

Scalable and Efficient Graph Algorithms and Analysis Techniques for Modern Machines

by
Quanquan C. Liu

Submitted to the Department of Electrical Engineering and Computer Science
on August 27, 2021, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Rapidly growing real-world networks, with billions of vertices, call for scalable, fast, and efficient graph algorithms. Luckily, commercial multi-core, multi-processor, and multi-machine environments can handle such volumes of data. Unfortunately, despite the availability of such resources, many current graph algorithms do not take full advantage of these parallel and distributed environments or have non-optimal theoretical guarantees, translating to slower and less efficient algorithms in practice. The purpose of this thesis is to theoretically improve previous graph algorithms in modern machines. We demonstrate through experiments that such theoretical improvements also translate to practical gains.

Towards this goal, this thesis takes a two-pronged approach. First, we formulate algorithms in computation models that mimic large-scale data processing environments. Algorithms in such models take advantage of clusters of machines and a machine's multiple cores and processors. Second, we use specific properties of real-world networks when designing our algorithms. The degeneracy is one such characteristic; while a network may have billions of vertices, its degeneracy may only be a few hundred.

This thesis consists of three parts. The first part presents static graph algorithms. We first introduce a set of new editing algorithms for a framework that approximates solutions to hard-to-solve optimization problems via editing a graph into a desired structured class. Then, we present novel small subgraph counting algorithms, with better theoretical space and round guarantees, in the massively parallel computation model; our experiments corroborate our theoretical gains and show improvements in number of rounds and approximation factor, compared to the previous state-of-the-art, in real-world graphs. We conclude this part with a near-linear time scheduling algorithm for scheduling on identical machines with communication delay where precedence constrained jobs are modeled as directed acyclic graphs.

The second part focuses on dynamic graph algorithms. We first show a $O(1)$ amortized time, with high probability, dynamic algorithm for $(\Delta + 1)$-vertex coloring. Then, we provide a new parallel level data structure for the k-core decomposition problem under batch-dynamic updates (where dynamic edge updates are applied in batches). We show that our data structure provably provides a $(2+\varepsilon)$-approximation on the coreness of each vertex, improving on the previously best-known bound of $(4 + \varepsilon)$. We conclude with new parallel, work-efficient batch-dynamic algorithms for triangle and clique counting. Our extensive experiments for our batch-dynamic algorithms show orders of magnitude

improvements in performance over the best previous multi-core implementations in real-world networks.

The last part concludes with lower bounds. We show via hard instances the hardness of obtaining an optimal computation schedule on directed acyclic computation graphs in the external-memory model. We then demonstrate that such graphs can be used to construct static-memory-hard hash functions that use disk memory to deter large-scale password-cracking attacks.

Thesis Supervisor: Erik D. Demaine
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Julian Shun
Title: Douglas T. Ross Career Development Professor of Software Technology
Associate Professor of Electrical Engineering and Computer Science

Acknowledgements

I would not be here today without having met the people who inspired and supported me along the way. I'd like to take the chance here to thank all of you.

First and foremost, I want to thank my PhD advisors, Erik D. Demaine and Julian Shun, for their guidance, mentorship, inspiration, and research prowess. I look forward to collaborations with them as colleagues for many more years in the future.

I met Erik during my junior year as an undergrad at MIT when I took his 6.851 Advanced Data Structures course. This class, as well as Erik's engaging lectures, was one of the key reasons I chose to pursue my PhD in theoretical computer science. Since then, Erik has been an incredibly supportive advisor, providing guidance on both my research and career through our weekly meetings, and also helping me establish collaborations with various collaborators around the world through the Barbados workshops and research visits. One of the most memorable aspects of Erik's mentorship philosophy are his "open problem sessions." Participating in these sessions early on in my career (and even during my undergrad) have taught me how to communicate my ideas clearly and to have the confidence to voice my thoughts. I have no doubt these skills will be valuable throughout my life.

Julian became my official advisor this past year, but he's been an unofficial mentor for much longer. The projects we've worked on since several years ago eventually evolved into the theme of this thesis, a theme that I hope to continue way into the future. Julian has been incredibly supportive in my research and career–illustrative examples include staying up until the last minute on our AoE deadlines, being available to schedule meetings on short notice to discuss projects/thesis/jobs, and being rapidly fast in responding to emails and Slack messages. In addition to the research aspects, Julian has also fostered a fun and inclusive lab environment through our weekly group lunches, game nights, and Krunker.io memes. I was able to collaborate and also play games with many members of the lab; and I hope to also be able to create such a close-knit research group environment in my future career.

I would also like to thank the final member of my thesis committee, Ronitt Rubinfeld for her years of support. I've also had the opportunity to work with Ronitt on a project that is included in this thesis and I particularly enjoy Ronitt's enthusiasm and optimistic attitude in research. Through our multi-hour research meetings, we talked through each and every aspect of the project and such level of detail allowed us to make various improvements along the way. Ronitt not only appreciates a student's research abilities but also sees their non-academic strengths, making us feel appreciated for our unique characters and personalities.

I would like to thank the other mentors I've had that had big impacts on my career. My undergraduate UROP advisor, David Karger, really taught me how to think outside the box. Although I didn't work on a theory project with him, I could tell that David approached all projects with a TCS mind-set–analyzing the problem logically and discussing solutions in detail before implementing code. I thank Neha Narula and Tadge Dryja for hosting me for a summer internship at the MIT DCI. I learned a lot about cryptocurrency– both the theoretical and practical aspects–and I am thankful for all the friendships I made with those in the lab. I thank Josh Wang for hosting me (virtually) at Google last summer as part of the IOR group. Josh met with me every day for 3 entire months while providing feedback on my projects; I owe a large part of the success of these projects to these daily meetings. Furthermore, I'd like to thank the others in the group, Erik Vee, Manish Purohit, Ravi Kumar, and Zoya Svitkina for working on our paper intensely with me and for the fun weekly lunches and puzzle hunts–helping me feel like I was part of the group even though I was working remotely.

I'd like to thank all the professors who taught courses for which I was fortunate to be a TA:

Aleksander Madry, Bruce Tidor, Dina Katabi, Erik Demaine, Michael Carbin, Nancy Lynch, Nir Shavit, Piotr Indyk, Ron Rivest, and the other TAs and writing instructors who I had the pleasure of working with. I learned a lot about how to teach and motivate students during these experiences and I hope to take these skills into my future positions.

I'd like to thank my collaborators whose work appears in this thesis: Sayan Bhattacharya, Amartya Shankha Biswas, Erik D. Demaine, Laxman Dhulipala, Thaddeus Dryja, Talya Eden, Timothy D. Goodrich, Fabrizio Grandoni, Kyle Kloster, Janardhan Kulkarni, Brian Lavallee, Slobodan Mitrović, Sunoo Park, Manish Purohit, Ronitt Rubinfeld, Jessica Shi, Julian Shun, Shay Solomon, Blair D. Sullivan, Zoya Svitkina, Ali Vakilian, Andrew van der Poel, Erik Vee, Joshua R. Wang, and Shangdi Yu. Without you these works would not be possible and I am deeply thankful for your collaborations and friendship.

I would also like to thank my other collaborators and others who I have discussed research throughout the years: Aviv Adler, Hugo A. Akitaya, Shiri Antaki, Michael A. Bender, Jeffrey Bosboom, Rezaul Alam Chowdhury, Rathish Das, Martin L. Demaine, Cory Fields, Varun Ganesan, Ofir Geri, Adam Hesterberg, Rob Johnson, Tim Kaler, David R. Karger, Vladislav Kontsevoi, Justin Kopinsky, William Kuszmaul, Andrea Lincoln, Qipeng Liu, Yang P. Liu, Jayson Lynch, Fermi Ma, Aleksandar Makelov, Ofir Nachum, Neha Narula, Anne Ouyang, Shwetark Patel, Richard Peng, Aaron Potechin, Nicholas Schiefer, Adam Sealfon, Aaron Sidford, Lili Su, Prashant Vasudevan, Madars Virza, Erik Waingarten, Di Wang, Nicole Wein, Virginia Vassilevska Williams, Helen Xu, Adam Yedidia, and Daniel Ziegler. Though these collaborations, I have been able to become very good friends with many of you and I enjoyed not only all the intellectual discussions but also the good humor and fun each one of you brought to the meetings.

Thank you to Aleksander Madry, Nancy Lynch, and Ronitt Rubinfeld for hosting the various reading groups that I participated in throughout my time as a student. These groups have greatly broadened my horizons, made me a more diverse researcher, and I also appreciate the various research guidance you provided outside of these groups.

I'd like to thank USACO, in particular, Brian Dean, for allowing me to be a coach at the annual USACO summer training camp for the past five years. This has been the highlight of my summer each year. I am also thankful to NAPC, in particular, Richard Peng, for allowing me to be a trainer this year. Despite being virtual, it was really exciting to meet and train students at the ICPC level. To all my fellow coaches and trainers of both camps, you all inspire me.

I'd like to take a moment to thank the theory administrators without whom MIT theory would not be as close a community as it currently is: Debbie Goodwin, Joanne Hanley, Patrice Macaluso, and Rebecca Yadegar. You all truly make the MIT theory group a welcoming place from hosting art days to Corn Fest, to just randomly running into you in the hallways and having fun conversations. Along this vein, much thanks to Ryan Williams and Virginia Vassilevska Williams for hosting weekly music nights. A particular shoutout to Debbie for putting up with all my silly antics, including stealing her name during Codenames. Thanks also to Janet Fischer and Alicia Duarte for help with preparing this thesis for publication, and also to Anne Hunter for the many years of interesting conversations and support.

I'd also like to thank a few additional individuals, throughout the years, of the MIT theory group who I have not yet mentioned for making my graduate school life enjoyable. I am thankful, including but certainly not limited to: Alex Lombardi, Brynmor Chapman, Changwan Hong, Dylan McKay, Frederic Koehler, Gautam Kamath, Govind Ramnarayan, Greg Bodwin, Ilya Razenshteyn, Jennifer Tang, Jerry Li, John Wright, Josh Alman, Justin Holmgren, Kai Xiao, Lisa Yang, Luke Schaeffer, Madalina Persu, Maryam Aliakbarpour, Michael Cohen, Mina Dalirrooyfard, Rio LaVigne, Robin Hui, Saleet Mossel, Sam Park, Siddhartha Jayanti, Sitan Chen, Soya Park, Tianren

Liu, Will Leiserson, Yiqiu Wang, and others who I may have missed. I am glad that despite everything being virtual, I am able to talk to and hang out with some of you virtually. Also, thanks to my friends who have stayed in touch from undergrad, particularly Jenny, Laura, Leila, Leo, and Sophia, for actively contacting me even though I am terrible at online communication.

Finally, I'd like to thank my parents and extended family for the support they've given me ever since I was a little child. I thank my parents especially for putting up with me and for graciously allowing me to stay at their place (rent-free) for the last year and a half. My parents have been with me through all the ups and downs in my life, and I would not be nearly the person I am today without them.

Contents

List of Figures

List of Tables

Part I

Introduction and Preliminaries

Chapter 1

Introduction

Graphs are ubiquitous in computer science. They are fundamental objects that have been studied and analyzed for hundreds of years beginning with the Königsberg Bridge problem, famously studied by Euler in 1736. Fundamentally, graphs are mathematical representations of relationships between people, objects, and ideas. Each *vertex* in the graph represents an object and each *edge* represents some relationship between a pair of objects. It is thus no wonder that analyzing the properties of graphs can provide us with better insight into many different types of structures. When a graph is used to model a real system, it is often colloquially called a *network* with *nodes* and *links*. Common examples of real-world networks include road networks, brain networks, communication networks, molecular networks, disease transmission networks, and circuits. These networks could vastly differ in their characteristics. Some may be sparse while others are dense; some may have high diameter and others low diameter. Analyzing the different properties of graphs provides us with answers to a myriad of problems in network analysis, optimization, scheduling, data organization, security, and many more [Bón06, Dal17, Deo17, EK10, KK16, VS10].

Consider a large social network graph such as the Friendster friendship network or the densely connected neuronal network of the human brain. What are some things researchers might want to learn about these graphs? Although these two networks represent vastly different types of relationships, there are some common characteristics of both types of networks that are valuable to researchers. For one, it may be useful to determine the nodes which are well-connected in both networks. Well-connected nodes in the Friendster graph could indicate popular social influencers; well-connected nodes in the brain network could indicate an area of the brain that participates in many different aspects of thinking. A set of nodes which are all mutually connected to each other could indicate a close-knit community in the Friendster graph or an assembly of neurons representing a thought or memory in the neuronal network. It is, however, not possible to use *any* algorithm to determine these properties due to the immense size of today's networks; this algorithm would also need to be *efficient* in order to be useful.

The public graphs that we study nowadays can have up to billions of vertices and hundreds of billions of edges. The publicly available Friendster graph has more than 65 million vertices and almost 2 billion edges [LS16]. The neuronal network provided by NeuroData is incredibly dense with $784,262$ vertices and $267,844,669$ edges [RA15]. Furthermore, many of these networks are also rapidly changing and evolving, with new

edges and connections being added each second. For example, in the past second,[1] over 9 thousand Tweets were made, 100,000 Google searches were conducted, and 3 million emails were sent [ILS]. In previous decades, the state-of-the-art algorithms were considered efficient if they are able to process such graphs in time that is linear in the size of the graph. However, such methods are too slow in today's rapidly evolving world. If thousands of edges change in the course of a few seconds, then we cannot hope for even linear time algorithms to provide us with accurate values if we were to rerun these algorithms every time the graph changes. Luckily, such graphs can still be processed in commercial multi-core, multi-processor, and multi-machine environments using *dynamic* algorithms that provide accurate values without processing the entire graph each time one part changes. Such machines can even be rented by the hour from Google Cloud Platform [GCP] or Amazon Cloud Services [AWS]. However, it is often not the case that classical static algorithms in the sequential model will necessarily (1) be easily made dynamic and (2) take advantage of the computational advantages of modern machines. Thus, the goal of this thesis is the following:

We seek to develop fast, scalable, and provably efficient graph algorithms and algorithms for performing computation on graphs in models that mimic modern computing systems.

We take a *two-pronged approach* in working toward this goal. First, we formulate algorithms in parallel and distributed computation models that represent today's large-scale data processing environments. Algorithms in such models take advantage of clusters of machines and a particular machine's multiple cores and processors. Second, we make use of some specific properties of real-world networks when designing our algorithms. It may be the case that most real-world networks have small *average* degree or are close (via some number of edge deletions) to a structured graph class. One can design faster and more efficient algorithms that take advantage of these characteristics.

Models of Computation The standard model of computation used by most previous works in the past few decades is the *sequential model* of computation. In this model, each step of the algorithm occurs sequentially, one after the other. This model of computation accurately models the setting when all computation is performed on a *single thread* where no two steps of the algorithm can be performed simultaneously or in parallel. Such a model is somewhat prohibitive in today's computing environments as even our basic laptops have more than one core and can maintain multiple threads.

In this thesis, we consider four additional models of computation, designed to mimic modern machines: the *shared-memory work-depth model*, the *massively parallel computation (MPC) model*, and the *external-memory model*. Each model captures a different aspect of the modern computing environment:

- The ***shared-memory work-depth model*** assumes a shared common memory between one or more processors which can process the input in *parallel*.
- The ***massively parallel computation (MPC) model*** assumes memory is distributed across *multiple machines* which perform computation and then share their results with each other via one or more rounds of communication.

[1] Calculated at the time of the writing of this thesis.

- The ***external-memory model or I/O model*** separates memory into a fast, but small *cache* and a slow, but large *disk*, where data can be transferred between the two.

The formal definitions of the above models are provided in Section 2.2. There are various challenges in finding provably efficient algorithms in each of these models such that one cannot trivially perform some number of basic conversions to transform an algorithm in the sequential model to one that is efficient in each of these models. In shared-memory settings, algorithm designers need to figure out which steps of the algorithm can be performed simultaneously without causing *race conditions* or *deadlock*. In distributed memory settings, one often needs to compute global values–such as the set of edges in the induced subgraph of a set of vertices–without using too many communication rounds. Finally, in the external-memory model, the cache is often too small to fit the entire input and one needs to design an algorithm that minimizes the number of times data is transferred between the cache and disk.

In each of the following chapters of this thesis, we describe the specific challenges we face for each result we obtain in one of these models.

Graph Properties and Problems Certain graph properties allow algorithm designers to design simpler and faster algorithms. For example, one can efficiently find the *minimum vertex cover* in bipartite graphs even though the problem is NP-hard in general graphs. Other examples of graph classes that allow for better algorithms to hard problems are bounded degree graphs and planar graphs. The drawback for designing algorithms for these special classes of graphs is that such classes of graphs may not represent real networks. Both of the aforementioned Friendster and neuronal graphs are not bipartite, do not have degree bounded by a small constant, and are not planar.

However, there are certain properties of graphs that are simultaneously realistic and allow algorithm designers to design better algorithms. The well-known *degeneracy* property (and the related *arboricity* property) is one such example. Not only do many real-life networks exhibit small degeneracy (e.g., the Friendster graph has a degeneracy of 304 compared to 2 billion edges) but also the degeneracy of a graph is closely related to its k-core decomposition, which is used in many applications in computer science. We define *degeneracy*, *arboricity*, and *k-core decomposition* in what follows.

In this thesis, we mainly study algorithms for undirected graphs, although a few results we include are algorithms on directed graphs. Unless otherwise specified, we define a graph $G = (V, E)$ to be an *undirected, unweighted* graph with a set of $n = |V|$ vertices and $m = |E|$ edges.

Degeneracy and Arboricity The degeneracy of a graph is one measure of a graph's sparsity. The *degeneracy* of a graph $G = (V, E)$ is defined as the minimum value k such that every induced subgraph of G has a vertex of degree at most k. It is also equivalent to a number of other notions which we give in Lemma 2.1.5. The classic algorithm of Matula and Beck [MB83] finds the degeneracy of a graph in $O(n+m)$ time; the algorithm works by repeatedly removing the vertex with smallest degree (resolving ties arbitrarily). The degeneracy of the graph is the largest degree of a vertex removed by this procedure. The order by which vertices are removed in this procedure is known as the *degeneracy*

ordering.

To see why the degeneracy ordering may be useful, consider the problem of enumerating the set of triangles in a graph. A classical algorithm for this problem by Chiba and Nishizeki [CN85] first finds a degeneracy ordering, then, in order, starting with the first vertex in the ordering, they consider all possible wedges incident to the vertex and check whether the wedge forms a triangle. Checking all such wedges can be done in $O(km)$ time in a graph with degeneracy k and finding the degeneracy ordering can be done in $O(m+n)$ time to result in a $O(km+n)$ algorithm. The degeneracy ordering is also used by many other efficient algorithms that solve different types of graph problems.

By the Nash-Williams theorem, the degeneracy of the graph is within a factor of two of another measure of sparsity known as the *arboricity* of the graph. The *arboricity* of the graph is formally defined as the minimum number of forests into which the edges of the graph can be partitioned. Given a graph with arboricity α, the degeneracy, k, of the graph is bounded by: $\alpha \leq k \leq 2\alpha - 1$.

Finally, a graph with degeneracy k has average degree at most k. The proof for this is quite simple: given a graph with degeneracy k, it has at most nk edges. Then, the average degree is upper bounded by $nk/n = k$. Several of the algorithms in this thesis use the degeneracy/arboricity of the input graph as well as these related properties.

We study a number of problems in this thesis and among the problems we study, we consider the following three key problems.

k-Core Decomposition The k-core of a graph $G = (V, E)$ is the maximal subgraph of G where the degree of every vertex in the subgraph is at least k [Sei83]. The *coreness* of a vertex $v \in V$ in the graph is the largest k where v is part of a k-core. The *k-core decomposition* of G is a partition of the vertices V into layers based on their coreness, inducing a natural hierarchical clustering of the graph. Vertices with larger coreness values clearly have larger degree and are more central to a network's structure. In a social network, one can think of such nodes as the popular social influencers who are friends with other famous social influencers. On the other hand, nodes with smaller coreness either have small degree or are not central to the network's structure. In our social network analogy, such nodes may have few friends or may have many friends, but none of whom are particularly famous. Thus, one can see that the k-core decomposition tells us something more profound than how many friends a user has; it tells us which users are fundamental to a social network's function. Namely, it tells us which individuals have the ability to influence a large portion of the network.

The k-core decomposition of a graph is an incredibly useful graph statistic. The k-core decomposition and its variants have been widely studied in the past and also more recently in the machine learning [AHDBV05, ELM18, GLM19], database [CZL+20, LZZ+19, ESTW19, BGKV14, MMSS20], graph analytics [KBST15, KM17a, DBS17, DBS18b], and other communities [GBGL20, KBST15, LYL+19, SGJS+13].

Triangle and k-Clique Counting Given an input graph $G = (V, E)$, the triangle count of G is the number of triangles that are contained in any subgraph of G. Likewise, the

k-clique count is defined as the number of subsets, S, of k vertices such that there exists an edge between every pair of vertices in the induced subgraph given by S. Triangle and clique counting are widely used to measure the cohesiveness of communities in social network analysis. In a graph of the web where edges consist of hyperlinks between pages, triangles and cliques can reveal the sets of webpages that share a common topic. The triangle and clique counts of a graph are also instrumental in numerous other social and network science measurements [FM95, Gra77, GLM12, MFC+20, NG04, RAK18, SSPc17, Tso15, TPM17, WS98, YBL18].

$(\Delta+1)$-Vertex Coloring Given an input graph $G = (V, E)$, the $(\Delta+1)$-vertex coloring of G is defined as the problem where provided $\Delta+1$ colors where Δ is the maximum degree in the graph, color all vertices such that no two adjacent vertices are colored the same color. Server scheduling is one longstanding application of vertex coloring. Suppose some company wants to take down some number of their servers for a software update. However, they do not want to simultaneously take down any two adjacent servers at the same time. To figure out a schedule to take down the servers, the company can perform a $(\Delta+1)$-vertex coloring of the server network; then the set of servers with the same color can be taken down at the same time.

Similar to the previous problems mentioned in this section, $(\Delta+1)$-coloring also has numerous applications in scheduling, register and resource allocation, designing seating plans, data mining, clustering, image segmentation, and even Sudoku [Akm08, Cha04, CM84, GMP05, JP94, KHSL16, Lei79, Lew15, Mar04, MSZ15, Saa96].

In particular, we study these problems in dynamically changing graphs. For the k-core decomposition and triangle/clique counting algorithms, we also provide extensive experiments on real-world networks.

Dynamic Graph Algorithms Many of the results in this thesis are dynamic graph algorithms which are meant to provide accurate graph statistics when the graph changes due to vertex or edge updates. Provided an initial graph, $G = (V, E)$, updates can be vertex/edge insertions/deletions. In this thesis, we only consider *edge updates* although several of our algorithms can be adapted to handle vertex updates.[2]

In the sequential setting, an edge update is applied *one at a time.* After each update, our dynamic graph algorithm updates the desired graph statistic.[3] Such a setting is the most fundamental setting to consider and is the setting that has received the most attention in the sequential model. However, in the parallel and distributed models, we can leverage the ability to perform computation simultaneously to obtain dynamic algorithms that can handle *multiple, simultaneous* updates. This model is formally defined as the *batch-dynamic model* [BW09, DDK+20].

[2]Specifically, vertex updates involving adding and deleting vertices with 0 degree.

[3]Some versions of dynamic algorithms allow for separate *update* and *query* operations. An *update* operation in such a setting only affects the internal data structures maintained by the dynamic algorithm (and the algorithm does not have to report accurate graph statistics after each update). After a query operation, the algorithm reports the updated graph statistic. In this thesis, we only consider the model where accurate graph statistics are provided after each graph update.

Batch-Dynamic Model Provided an initial graph, $G = (V, E)$, updates are applied in ***batches***, $\mathcal{B}$, of updates to be processed in parallel. A ***batch-dynamic algorithm*** provides accurate graph statistics after each batch of updates is applied.

1.1 Summary of Contributions

In the following, we give a high-level introduction to the contents of this thesis as well as the motivation behind the topics we chose to study. In Parts II to IV, we provide the technical components of the thesis which is split into three parts:

1. Part II first discusses static graph algorithms. Chapter 3 provides approximation algorithms for hard-to-solve graph problems in graphs that are near an algorithmically tractable graph class. Chapter 4 discusses algorithms in the MPC model for small subgraph counting and demonstrates via experiments that our algorithms also empirically provides better approximations than the state-of-the-art on all tested real-world graphs. Chapter 5 shows a near-linear time scheduling algorithm for scheduling with communication delay where precedence constrained jobs are modeled as directed acyclic graphs.
2. Part III then expands on new efficient and scalable dynamic graph algorithms. Chapter 7 shows a $O(1)$ amortized time, *whp*, dynamic algorithm for $(\Delta + 1)$-vertex coloring. Chapter 6 provides a new parallel, batch-dynamic k-core decomposition algorithm. Chapter 8 provides new parallel, work-efficient batch-dynamic algorithms for triangle and clique counting. Both of these chapters also include extensive experiments showing orders of magnitude improvement in performance on real-world networks.
3. Finally, Part IV discusses hardness results from pebbling and cryptographic constructions using such hard instances. Chapter 9 presents a set of results showing the hardness of obtaining optimal computation schedules on directed acyclic computation graphs in the external-memory model. Chapter 10 concludes with constructions using such hard instance graphs to construct static-memory-hard hash functions that use disk memory to deter large-scale password-cracking attacks, even against an adversary with unlimited parallel processing power.

Each of these parts begins with its own technically more in-depth overview of the background and results therein. In Chapter 2, we provide all definitions, notations, and terminology used throughout the thesis.

The following sections provide details on the results mentioned in the outline above.

1.1.1 Part II: Static Graph Algorithms

Chapter 3: Approximation Algorithms for Graphs Near an Algorithmically Tractable Class Real-world networks often do not have underlying graphs that fall under a structured graph class such as bounded degree, bounded degeneracy, or bounded treewidth. Such networks could result from a variety of different reasons including natural variations in the underlying model or measurement error. We consider real-world

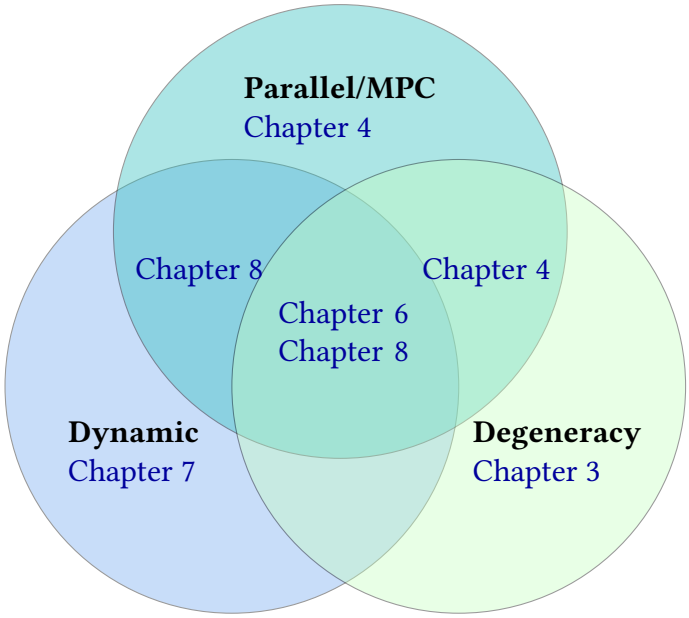


Figure 1-1: Pictorial representation of some of the graph algorithm results in this thesis.

networks that require some number of γ vertex deletions, edge deletions, or edge contractions to become a member of a structured graph class to be γ-close to this graph class; this leads to a natural procedure for solving hard problems in graphs that are γ-close to structured graph classes.

First, edit the graph by deleting some number of vertices and/or edges so that the graph becomes structured. Then, solve the hard-to-compute problem on the new graph. Finally, *lift* the solution S obtained for the problem on the structured graph by adding the set of vertices/edges that were deleted back into S. For any problem Π that satisfy our *stability* and *lifting* conditions [DGK+19], we show that a $\left(1 + O\left(\delta \log^{1.5} n\right)\right)$-approximation exists for Π for graphs which are $(\delta \cdot |\mathsf{OPT}_\Pi(G)|)$-close to having treewidth w via vertex deletions. Problems which fall under this domain include Vertex Cover, Feedback Vertex Set, Minimum Maximal Matching, and Chromatic Number. We also show that a $\left(\frac{1-4\delta}{4k+1}\right)$-approximation exists for Independent Set on graphs $(\delta \cdot |\mathsf{OPT}_\Pi(G)|)$-close to having degeneracy k.

Given this framework, it seems natural to study approximation editing algorithms for deleting vertices and edges to obtain a graph in a desired class. For parameterized graph classes, the natural bicriteria approximation is an approximation on the edit set combined with an approximation of the parameter of the parameterized graph class. So, in addition, we also show a set of new editing results spanning from editing algorithms that obtain constant or polylogarithmic bicriteria approximations to hardness of approximation results for computing the optimal number of edits to the exact parameterized graph class. Our results for editing to parameterized graph classes include, most prominently, an $\left(O\left(\log^{1.5}(n)\right), O\left(\sqrt{\log w}\right)\right)$-approximation on vertex deletion to treewidth w where the first parameter is the approximation factor on the vertex deletion set and the second is the

approximation factor on the treewidth. Our results also include a $\left(\frac{1}{\varepsilon}, \frac{4}{1-\varepsilon}\right)$-approximation for any $\varepsilon < 1$ to bounded degeneracy and a $O(\log n)$-approximation for editing to constant degeneracy k, which is tight up to constant factors with our lower bounds [DGK+19].

Chapter 4: Static Small Subgraph Counting In this chapter, we present results for triangle and clique counting in the massively parallel computation (MPC) model [BEL+20]. With the growing popularity of frameworks and programming models, such as MapReduce, Hadoop, Spark, and Dryad, that distribute computation across many machines and then perform such computation in parallel, the MPC model has become the theoretical standard for studying algorithms in such massively parallel frameworks. In this chapter, we show that there exists an MPC algorithm that gives the exact count of triangles in an input graph in $O(\log \log n)$ rounds, $O(n^\delta)$ space per machine for any constant $\delta > 0$, and $O(m\alpha)$ total space where α is the arboricity of the graph [BEL+20]. To the best of our knowledge, there are no known MPC triangle counting algorithms for bounded arboricity graphs beyond the trivial algorithm of computing a degeneracy orientation and brute-force counting the triangles by putting all out-neighbors in the same machine and trying all pairs. This naïve algorithm requires at least $O(\log n)$ rounds and total space $O(n\alpha^2)$, both of which are more costly than what we obtain in our work.

Approximate triangle counting on the one hand requires less time and space than exact counting. However, to obtain good estimates on the subgraph count and to provide approximations that concentrate well oftentimes require lower bounds on the count of the subgraphs in an input graph. In fact, these lower bounds could be quite large, on the order of n, where n is the number of vertices in the input graph. We show a $(1+\varepsilon)$-approximation algorithm for the number of triangles that uses near-linear total space and space per machine and with a lower bound of square root of the average degree, $T = \Omega\left(\sqrt{d_{avg}}\right)$. All of this is done in a constant number of MPC rounds. The best previous results for this problem required the lower bound on T to be $\Omega(d_{avg})$ [PT11, KPS13]. Thus, compared to the best previous bound on the number of triangles, our algorithm achieves a quadratic improvement over the previous lower bound and also improves on the space per machine. We show an extension of this algorithm to any K-vertex subgraph where $K \geq 1$ is constant.

We further extend our results to counting k-cliques. In the domain of graphs with bounded arboricity α, we obtain an exact k-clique counting algorithm in $O\left(m\alpha^{k-2}\right)$ total space and $O(\log \log n)$ rounds given machines with space at most $O(n^\delta)$ for any constant $\delta > 0$. In order to achieve this bound, we formulated several novel MPC primitives that allows us to duplicate and aggregate counts across multiple machines.

We tested[4] our triangle counting algorithms using graphs from the Stanford SNAP database [LS16]. For our simulation of the exact triangle counting algorithm on bounded arboricity graphs, we picked different values of space per machine and computed the number of rounds necessary to find the number of triangles. We tested the algorithm against the trivial folklore algorithm of just sending all the vertices for the out-degree neighborhood of a vertex to the same machine. We saw that our algorithm performs better

[4]Code can be found here: https://github.com/qqliu/mpc-triangle-count-exact-approx-simulations.git.

than the trivial algorithm for nearly all cases even when we give the trivial algorithm more memory.

For our implementation of the approximation algorithm, we looked at the approximation factors returned by our implementation of [PT11] with our implementation of our approximate algorithm. As expected, we obtain a better approximation for all cases than [PT11] for the same space per machine because we require a smaller theoretical bound on the number of triangles.

Chapter 5: Efficient Algorithms for Scheduling In this chapter, we consider the problem of efficiently scheduling precedence-constrained jobs on a set of identical machines in the presence of uniform communication delay [LPS+21]. Assuming a communication delay of ρ-time units where $\rho > 1$, all jobs are unit size, and allowing duplication, we provide the first efficient approximation algorithm that runs in near-linear time for this problem and whose approximation matches the best-known approximation algorithm by [LR02]. A naïve implementation of the Lepere and Rapine algorithm [LR02] requires $O(m\rho + n \ln M)$ time on a directed acyclic graph with n vertices and m edges. For a large communication delay, such a schedule is not efficient to compute.

The main bottleneck in the original algorithm proposed by Lepere and Rapine is a graph theoretic one: how does one obtain the sizes of ancestor sets of each vertex and determine sets that *do not overlap too much* in near-linear time? We solve this issue and others via a sampling-based approach to obtain our near-linear time algorithm where the crux of the algorithm is a new way of determining the fraction of ancestor sets that overlap between any two nodes in a DAG. Namely, we provide an $O\left(\frac{\ln \rho}{\ln \ln \rho}\right)$-approximation algorithm that runs in $O(n \ln M + m \cdot \operatorname{poly} \log n)$ time, where M is the total number of machines.

1.1.2 Part III: Dynamic Graph Algorithms

Chapter 6: Dynamic k-Core Decompositions Maintaining a k-core decomposition quickly in a dynamic graph is an important problem in many machine learning applications and has been cited as one problem that bridges theory and practice [HHS21]. The k-core decomposition of the graph is defined as a partition of the graph into layers such that a vertex v is in layer k if it belongs to a k-core but not a $(k+1)$-core. This induces a natural hierarchical clustering on the input graph. Many well-known algorithms for k-core decomposition are inherently sequential. The classic algorithm for finding such a decomposition is to iteratively select and remove all vertices v with smallest degree from the graph until the graph is empty [MB83]. The length of the sequential dependencies (the depth) of such a process could be $\Omega(n)$ given a graph with n vertices. Due to the potentially large depth, this algorithm cannot fully take advantage of parallelism on modern machines, and can therefore be too costly to run on large graphs.

In this chapter, we present the first parallel batch-dynamic algorithm for maintaining an approximate k-core decomposition that is efficient in both theory and practice [LSY+21]. Provided an input graph with m edges and a batch of B updates, our algorithm maintains a $(2+\varepsilon)$-approximation of the coreness values for all vertices (for any

constant $\varepsilon > 0$) in $O(B \log^2 n)$ amortized work, $O(\log^2 n \log\log n)$ depth with high probability, using $O(n \log^2 n + m)$ space. Theoretically, our proof of our $(2+\varepsilon)$-approximation bound improves upon the $(4+\varepsilon)$-approximation guarantee of the most recent dynamic algorithm for this problem by [SCS20] while maintaining polylogarithmic amortized work. Practically, we also implemented our algorithm on a 30-core machine with two-way hyperthreading. Our code-optimized implementations achieve up to 497.63× speedup over Sun et al. [SCS20] and 114.52× speedup over the state-of-the-art exact, multi-core implementation [HSY+20].[5]

Chapter 7: Dynamic $(\Delta+1)$-Vertex Coloring The problem of vertex coloring with $(\Delta+1)$ colors is an extremely well-studied problem under various computational models and settings. The problem seeks to find a set of colors to assign to each vertex in an input graph such that no two vertices which share an edge use the same color. The fully dynamic version of this problem seeks to maintain a valid $(\Delta+1)$-coloring of the graph under edge insertions and deletions. Our work [BGK+19] improves upon the previously best known result via an optimal $O(1)$ amortized update time algorithm with high probability, improving on the time bound of [BCHN18] and providing a result with high probability compared to the concurrent result of [HP20] which only provides the bound *in expectation*.

This result makes use of a data structure known as a *level data structure*. Different types of level data structures are used for a variety of dynamic graph problems. The result provided in Chapter 6 is based on our new parallel variant of the level data structure which we simply call a *parallel level data structure*. Such techniques used in our parallel level data structure may be applicable to the level data structure in this chapter. If such techniques can be applied, an interesting future direction would be to see whether our $(\Delta+1)$-vertex coloring algorithm would perform well, experimentally, on highly dynamic networks.

Chapter 8: Dynamic Triangle and Clique Counting Algorithms Triangle, k-clique, and other subgraph counting algorithms have wide practical importance. It is thus important to come up with fast triangle counting algorithms that have very good theoretical guarantees as well as being implementable and fast in practice. Furthermore, because graphs are continuously changing in real-world networks, it is also important to study the dynamic version of the problem. We investigated algorithms for such problems in two practical models of real-world systems.

Our algorithms in this chapter are algorithms in the shared-memory work-depth model. They also handle batch-dynamic updates. Given an input of a batch $\mathcal{B}$ of edge insertions and/or deletions into a graph, our goal is to formulate an algorithm such that the amortized work is at most the work of applying the update one at a time, but the depth is $O(\log n)$. Building on the sequential triangle counting algorithm presented in [KNN+18], we obtain a batch-dynamic triangle counting algorithm that returns the triangle count in $O(\log n)$ depth, $O(\sqrt{m})$ work per update, and $O(m)$ total space in the batch-dynamic setting [DLSY20]. A trivial implementation of the best sequential algorithm by Kara et. al. [KNN+18] instead results in at least $\Omega(|B|)$ depth, much too large for large batches. We also present a parallel batch-dynamic algorithm for k-clique counting in $O(\log(m+|B|))$

[5] Code can be found here: https://github.com/qqliu/batch-dynamic-kcore-decomposition.

depth, $O\left(\min(|B|m^{0.469k-0.704}, (m+|B|)^{0.469k})\right)$ work, and $O\left((m+|B|)^{0.469k}\right)$ space using our parallel implementation of the currently best-known matrix multiplication algorithm. This result also achieves the theoretically best bounds for k-clique counting in dense graphs in the *sequential model* when $k > 9$ [DLSY20].

We tested our batch-dynamic triangle counting algorithm on a 72-core machine on a number of datasets obtained from the Stanford SNAP database [LS16] and benchmarked against the state-of-the-art batch-dynamic triangle counting algorithm of [MBG17]. We found a 10× improvement for all tested batch sizes on the Twitter network graph which has billions of edges as well as improvements for smaller batch sizes on other graphs.

1.1.3 Part IV: Hardness from Pebbling

Chapter 9: Hardness of Computation via the Red-Blue Pebble Game Pebble games were originally used to model various forms of computation including register allocation, reversible computation, and input/output complexity in the external memory model. The hardness of computing the optimal strategy for pebbling a directed acyclic graph (DAG) using the rules of these games have been well-studied in the past (see e.g. [GLT80, HP10, CLNV15]). We proved the best current bound on the inapproximability of the standard pebble game in [DL17]. We showed that the standard pebble game is PSPACE-hard to approximate to an additive term of $O(n^{1/3})$ [DL17] (where n is the number of vertices in the DAG) whereas the previously best known result only proved that it is PSPACE-hard to approximate to an additive *constant* term of approximation [CLNV15].

In this chapter, we show additional hardness results for obtaining optimal schedules in the external-memory model. We show that determining the optimal strategy in the red-blue pebble game is PSPACE-complete and is fixed-parameter intractable when considering the number of input/outputs from disk to cache and vice versa [DL18]. The complexity of pebble games has long been studied [GLT80, HP10, WAPL14, CLNV15, DL17], because of the implications for proof complexity and computation. Furthermore, more recently, the hardness of the red-blue pebble game has been used to prove properties about bandwidth-hard hash functions [RD17, BRZ18]. Thus, our hardness result for the red-blue pebble game implies that these bandwidth-hard functions provide additional security.

Most recently, pebble games have been used to study rematerialization in the computation performed by neural networks [KPS+19]; thus, our hardness results also imply that such research on finding efficient algorithms to perform neural network computation in the area of deep learning should focus on specific graph classes.

Chapter 10: Static-Memory-Hard Hash Functions Defending against large-scale password cracking attacks have always been a fundamental problem in theory and in practice. The evolution of password hashing functions that are resistant to large-scale attacks has been something of an arms race for decades, with memory-hard functions as the recent bulwark against adversarial advantage. In this chapter, we first provide two significant and practical considerations not analyzed by existing models of memory-hardness, propose and analyze new models to capture them, and, finally, provide novel hash function constructions based on hard-to-pebble graphs that achieve the desired

memory-hardness [DLP18]. Our cryptographic constructions use ideas from Chapter 9, specifically those related to the construction of the hard instances in that chapter.

Most notably, we introduce a new cryptographic primitive–static-memory-hard hash functions–that can be combined with current memory-hard hash functions to obtain greater security. The main idea behind static-memory-hard hash functions is to use static/preprocessed (disk) memory as the memory-hard component of the function. Previous functions used only dynamic memory (RAM) such that the memory requirement of the function is bounded by the runtime of computing the function; this bound is due to the fact that honest evaluators need to be able to compute the function using standard hardware (e.g., a laptop).

Thus, by allowing the use of static memory (generated through a one-time preprocessing phase) we can guarantee that adversaries who try to cheat can suffer a penalty of potentially having to use gigabytes of RAM at runtime. We also instantiate our static-memory-hard function with two constructions and created an initial prototype in silico of our simpler construction. Our work can potentially have broad impact in password security. Furthermore, the ideas we introduce can be used in proofs-of-space protocols for cryptocurrencies (like Bitcoin) that require some proof-of-work or space.

1.2 Research Works Presented in This Thesis

This thesis contains the following research works performed by the thesis author and her collaborators (in the order they are presented in this thesis):

- [DGK+19] Erik D. Demaine, Timothy D. Goodrich, Kyle Kloster, Brian Lavallee, Quanquan C. Liu, Blair D. Sullivan, Ali Vakilian, and Andrew van der Poel. Structural rounding: Approximation algorithms for graphs near an algorithmically-tractable class. (Chapter 3)
- [BEL+20] Amartya Shankha Biswas, Talya Eden, Quanquan C. Liu, Slobodan Mitrović, and Ronitt Rubinfeld. Parallel algorithms for small subgraph counting. (Chapter 4)
- [LPS+21] Quanquan C. Liu, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua R. Wang. Scheduling with communication delay in near-linear time. (Chapter 5)
- [LSY+21] Quanquan C. Liu, Jessica Shi, Shangdi Yu, Laxman Dhulipala, and Julian Shun. Parallel batch-dynamic k-core decomposition. (Chapter 6)
- [BGK+19] Sayan Bhattacharya, Fabrizio Grandoni, Janardhan Kulkarni, Quanquan C. Liu, and Shay Solomon. Fully dynamic $(\Delta+1)$-coloring in constant update time. (Chapter 7)
- [DLSY21] Laxman Dhulipala, Quanquan C. Liu, Julian Shun, and Shangdi Yu. Parallel batch-dynamic k-clique counting. (Chapter 8)
- [DL18] Erik D. Demaine and Quanquan C. Liu. Red-blue pebble game: Complexity of computing the trade-off between cache size and memory transfers. (Chapter 9)
- [DLP18] Thaddeus Dryja, Quanquan C. Liu, and Sunoo Park. Static-memory-hard functions, and modeling the cost of space vs. time. (Chapter 10)

Other Research and Works Several other works not included in this thesis but studied by the author of this thesis and her coauthors also reflect a somewhat different aspect of the theme of this thesis. We include a brief description of them here for completeness. The first several works are algorithms and lower bounds for problems in the external-memory and cache-adaptive models. The last work studies important symmetry-breaking problems in graphs but in a somewhat different model where the input graph itself represents the communication network.

We studied the notion of fine-grained complexity in the external-memory (I/O) model in [DLL+18]. Several algorithms for very fundamental problems are slow in the I/O-model, particularly graph algorithms in sparse graphs, such as the algorithms for triangle detection and finding the diameter and radius in sparse graphs. We explain the reason for this lack of progress by using ideas from fine-grained complexity [DLL+18] to show that solving these problems in the I/O-model is just as hard as solving the all-pairs shortest paths problem (APSP), the 3-SUM problem, or orthogonal vectors problem (OV) which are commonly conjectured to not have linear time algorithms in both the I/O and RAM models.

Our work in cache-adaptive algorithms includes two important features: scan-hiding for adaptivity and smoothed analysis of non-cache-adaptive algorithms. Cache-adaptivity is a concept developed to account for multiple threads or processes running on a shared common cache. This behavior is modeled via the cache-adaptive model [BEF+14, BDE+16], where the performance of an I/O algorithm is evaluated on a changing memory profile. In this model, we developed a scan-hiding technique that introduced the first cache-adaptive matrix multiplication algorithm that uses less I/Os than the number of I/Os used by the naïve matrix multiplication algorithm [LLLX18]. Furthermore, we generalize this scan-hiding procedure to any (a, b, c)-regular recursive algorithm which scans at most the size of the input at each level of the recursion.

More recently, we showed that non-cache-adaptive algorithms are cache-adaptive beyond the worst-case [BCD+20]. We performed a smoothed analysis of common cache-oblivious but non-cache-adaptive algorithms and showed that even simple, natural forms of smoothing the worst-case input lead to the algorithm being cache-adaptive in expectation. Most notably, we show that provided any memory profile, if we randomly shuffle the parts of the profile (where a part is defined by some "significant event"), then our algorithm is optimally cache-adaptive on the shuffled profile in expectation. This shows that in real-world systems where it is difficult to tailor a memory sequence to a particular algorithmic structure, cache-obliviousness is a good proxy for cache-adaptiveness.

Finally, in a work currently under submission [ALS20], we consider a number of symmetry-breaking problems in the distributed Congest setting. In the Congest model, vertices in the input graph compose the communication network over which messages, of size $O(\log n)$ bits, are sent in a point-to-point fashion. The main difference between this model and a sequential dynamic model is that vertices in the graph only know about a topological change in the graph if they are adjacent to the edge update or if another vertex informs them about a change. Vertices in the graph are *asleep* and cannot send messages to other vertices in the graph unless an edge update incident to the vertex occurs or another adjacent vertex wakes them up. Only after a vertex wakes up can the vertex send messages to its neighbors. Due to the lack of knowledge of the current topology of the

graph, the key challenge we face when formulating algorithms in this model is for each vertex to maintain an accurate estimate of the number of edges in the graph. In the case when some number of vertices are present in disconnected subgraphs, oftentimes, vertices in such a disconnected component would not know about the number of edge insertions and deletions in other parts of the graph. We solve this issue by presenting a general framework for *restarting* the graph by updating the edge counts stored in vertices when a certain number of edge insertions and deletions have occurred. Using this framework, we obtain *deterministic* algorithms that are robust against a strongly adaptive adversary for maximal matching, $(3/2)$-approximate maximum matching, $(\Delta + 1)$-coloring, and maximal independent set [ALS20], which improve upon previous algorithms in their message complexity.

1.2.1 Summary and Thesis Statement

The goal of this thesis is to present efficient graph algorithms, in both the static and dynamic settings, and also lower bounds for computation on graphs in models that closely mimic current real-world systems. The next parts of the thesis detail the advances in the aforementioned three categories: static graph algorithms, dynamic graph algorithms, and hardness from pebbling.

In this thesis, we present novel theoretical results in the work-depth, MPC/distributed, and external-memory models. While the focus of this thesis is on the upper bound results, we also provide some complementary lower bound results. For many of our results, we also provide implementations of the algorithms/constructions and show the efficiency of these implementations on real-world data. The code for all of our implementation are publicly available (with links included in this thesis). We hope that the theoretical results in this thesis that do not currently have accompanying experiments will prove to be practically useful should future practitioners choose to implement them. We also hope that algorithm designers would be convinced that there are a variety of problems in this space that will lead to interesting theoretical contributions.

We conclude with the following thesis statement.

Thesis Statement *Novel static and dynamic graph theoretic techniques and lower bounds in the work-depth, massively parallel computation and external-memory models can lead to algorithms and constructions that are better suited for modern computing environments.*

Chapter 2

Preliminaries

2.1 Graph Definitions and Notations

In this thesis, we consider simple, finite graphs with no self-loops or multi-edges. Unless otherwise specified, we define a graph $G = (V, E)$ to be an *undirected, unweighted* graph with a set of $n = |V|$ vertices and $m = |E|$ edges. We denote an edge between two vertices $u, v \in V$ by (u, v). The set of neighbors of a vertex v is denoted by $N(v) = \{u \,|\, (u, v) \in E\}$ and the degree of v by $\deg(v) = |N(v)|$. For a subset of vertices, $S \subseteq V$, $N(S) = \bigcup_{v \in S} N(v)$. Given some subset $E' \subseteq E$ of the edges in G, we define $G[E']$ to be the subgraph of G induced on the edge set E'. Note that if every edge adjacent to some vertex v is in $E \setminus E'$, then v does not appear in the vertex set of $G[E']$.

For directed graphs, we denote the in-neighbors of a vertex v by $N^-(v)$ and the out-neighbors of v by $N^+(v)$. We then let the in-degree and out-degree of v be $\deg^-(v)$ and $\deg^+(v)$, respectively. We let $\Delta(G)$ be the maximum degree of G; when the context is clear, we simply use Δ.

Graph Classes and Properties We first define the related properties of *degeneracy* and *arboricity* which are used in several results in this thesis.

Definition 2.1.1 (Degeneracy)**.** *A graph $G = (V, E)$ is **k-degenerate** if every subgraph of G contains a vertex of degree at most k; the **degeneracy** is the smallest $k \in \mathbb{N}$ so that G is k-degenerate. Let $\text{degen}(G) = k$ denote the degeneracy of G.*

The concept of *k-cores* is directly related to degeneracy and much literature center around k-cores and k-core decomposition.

Definition 2.1.2 (k-Core and k-Shell)**.** *For a graph $G = (V, E)$ and positive integer k, the **k-core** of G, $\text{core}_k(G)$, is the maximal subgraph of G with minimum degree k. The **k-shell** of G is $\text{core}_k(G) \setminus \text{core}_{k+1}(G)$.*

Using the definition of k-core, we can now define a k-core decomposition.

Definition 2.1.3 (k-Core Decomposition)**.** *A **k-core decomposition** is a partition of vertices into layers such that a vertex v is in layer k if it belongs to a k-core but not to a $(k+1)$-core. $k(v)$ denotes the layer that vertex v is in, and is called the **coreness** of v.*

The above defines an *exact* k-core decomposition. A *c-approximate* k-core decomposition is defined as follows.

Definition 2.1.4 (c-Approximate k-Core Decomposition)**.** *A **c-approximate k-core decomposition** is a partition of vertices into layers such that a vertex v is in layer k' only if $\frac{k(v)}{c} \leq k' \leq ck(v)$ where $k(v)$ is the coreness of v.*

Throughout, we let $\hat{k}(v)$ denote the estimate of v's coreness. Fig. 2-1 shows a k-core decomposition and a (3/2)-approximate k-core decomposition.

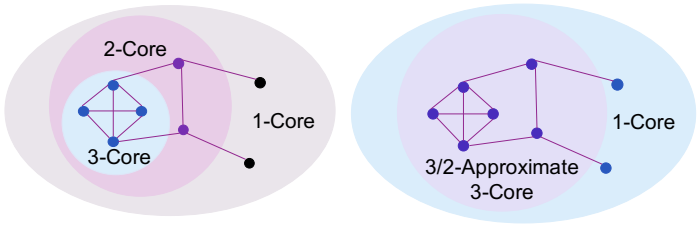


Figure 2-1: Exact k-core decomposition (left) and (3/2)-approximate k-core decomposition (right).

The degeneracy of a graph is equivalent to several other properties.

Lemma 2.1.5 ([CE91, LW70, MB83])**.** *Given a graph $G = (V, E)$, the following are equivalent:*

1. *The degeneracy of G is k.*
2. *The $(k+1)$-core of G is empty and the k-core of G is non-empty.*
3. *There exists an orientation of the edges in E so that it forms a directed acyclic graph and $\deg^+(u) \leq k$ for all $u \in V$.*
4. *There exists an ordering $v_1, \ldots, v_n$ of V so that the degree of v_j in $G[\{v_1, \ldots, v_j\}]$ is at most k.*

We formally define low out-degree orientation mentioned in Lemma 2.1.5.

Definition 2.1.6 (Low Out-degree Orientation)**.** *A **low out-degree orientation** of an undirected graph is an orientation of its edges such that the out-degree of every vertex is $O(k)$ where k is the degeneracy of the graph.*

Given a graph $G = (V, E)$, the size of the largest clique that is a subgraph of G is called the *clique number* of G and denoted $\omega(G)$. The *coloring number* of a graph G is the minimum number of colors needed to color G so that no two adjacent vertices have the same color. It is well-known that a graph which has bounded degeneracy k immediately implies bounded clique and chromatic numbers $\omega(G), \chi(G) \leq \text{degen}(G) + 1 = k + 1$.

Theorem 2.1.7 (Degeneracy Ordering Algorithm [MB83])**.** *Given $G = (V, E)$, there exists a sequential algorithm that runs in $O(m+n)$ time that finds the degeneracy of G by repeatedly finding and removing a vertex of minimum degree. The order by which vertices are removed is called the **degeneracy ordering**.*

Degeneracy is closely related to another notion of graph sparsity called *arboricity*. This thesis uses the ***arboricity*** of the graph as a parameter in several of our analyses. We define the property below.

Definition 2.1.8 (Arboricity)**.** *Let α denote the arboricity. Provided an input graph, $G = (V, E)$, the **arboricity** of the graph, α, is the minimum number of forests into which E can be partitioned.*

The arboricity and the degeneracy are both upper bounded by $O(\sqrt{m})$ [CN85]. Notably, by the Nash-Williams theorem, given a graph with arboricity α, the degeneracy, k, of the graph is bounded by: $\alpha \leq k \leq 2\alpha - 1$. The Nash-Williams theorem also relates arboricity to another quantity known as the *densest subgraph*.

Definition 2.1.9 (Densest Subgraph)**.** *A **densest subgraph** S of a graph $G = (V, E)$ has density $\lceil |E(S)|/|S| \rceil = \max_{S' \subseteq V} (\lceil |E(S')|/|S'| \rceil)$.*

The Nash-Williams theorem specifically shows that the arboricity is equal to $\max_{S' \subseteq V} (\lceil |E(S')|/(|S'| - 1) \rceil)$. This means that the density of the densest subgraph is upper bounded by the arboricity (and is approximately equal to the arboricity).

2.2 Models of Computation

In this section, we define the models of computation we consider in this thesis.

2.2.1 Shared-Memory Work-Depth Model

We analyze the theoretical efficiency of our parallel algorithms in the *work-depth model*. The work-depth model is a fundamental tool in analyzing parallel algorithms, e.g., see [BFS16, DBS18b, GSSB15, SBLP19, WGS20] for a sample of recent practical uses of this model. An algorithm in the work-depth model is characterized by two complexity measures, ***work*** and ***depth***, which are standard measures for analyzing shared-memory algorithms [Jaj92, CLRS09]. The ***work*** is the total number of operations performed by the algorithm. No parallel algorithm can perform less work than the best-known sequential algorithm since any parallel algorithm can be made sequential by performing each parallel step sequentially. Thus, the gold standard for parallel algorithms are algorithms whose work matches the running time of the best-known sequential algorithm. We call such algorithms ***work-efficient*** algorithms. The ***depth*** of the algorithm is the longest chain of sequential dependencies in the algorithm (or the computation time given an infinite number of processors) [Jaj92]. For sequential algorithms, the depth is equal to the work. However, for parallel algorithms, the depth of the algorithm is often much smaller than the work of the algorithm. Provided a graph $G = (V, E)$, an efficient algorithm in this model is work-efficient and operates in $\text{poly}\log n$ depth.

Our algorithms can run in the nested-parallel model or the PRAM model. We use the concurrent-read concurrent-write (CRCW) model, where reads and writes to a memory location can happen concurrently. We assume that concurrent reads and writes are supported in $O(1)$ work/depth. We also assume either that concurrent writes are resolved arbitrarily, or are reduced together (i.e., fetch-and-add PRAM).

Parallel Primitives We use the following primitives throughout this thesis. **Approximate compaction** takes a set of m objects in the range $[1, n]$ and allocates them unique IDs in the range $[1, O(m)]$. The primitive is useful for filtering (i.e. removing) out a set of obsolete elements from an array of size n, and mapping the remaining m live elements to a sparse array of size $O(m)$. Approximate compaction can be implemented in $O(n)$ work and $O(\log^* n)$ depth [GMV91]. We also use a **parallel hash table** which supports n operations (insertions, deletions) in $O(n)$ work and $O(\log^* n)$ depth with high probability, and n lookup operations in $O(n)$ work and $O(1)$ depth [GMV91].

Our algorithms in this thesis make use of the widely used **fetch-and-add** instruction. A fetch-and-add instruction takes a memory location and atomically increments the value stored at the location. In this thesis, we assume that the fetch-and-add instruction can be implemented in $O(1)$ work and depth. Existing simulation results show that the CRCW PRAM augmented with a fetch-and-add instruction can be simulated work-efficiently at the cost of a space increase proportional to the number of fetch-and-adds done, and a multiplicative $O(\log n)$ factor increase in the depth [MV91].

We use a parallel ***reduce-add*** in our algorithms, which takes as input a sequence A of length n, and returns the sum of the entries in A using $O(n)$ work and $O(\log n)$ depth [Jaj92].

Parallel Batch-Dynamic Algorithms All batch-dynamic [BW09, DDK+20] algorithms in this thesis operate in the shared-memory parallel setting. A ***batch-dynamic*** algorithm processes updates (vertex or edge insertions/deletions) in batches, $\mathcal{B}$, of size $|\mathcal{B}|$. A ***batch-dynamic algorithm*** provides accurate graph statistics after each batch of updates is applied. For simplicity, since we can reprocess the graph using an efficient parallel static algorithm when $|\mathcal{B}| \geq m$, we consider $1 \leq |\mathcal{B}| < m$ for our bounds.

2.2.2 Massively Parallel Computation (MPC) Model

The *massively parallel computation (MPC) model* assumes memory is distributed across multiple machines. Initially, the input is distributed in some organized fashion across more than one machine. This means that machines know how to access the relevant information via communication with other machines. Then, computation is performed by each individual machine and data is shared via one or more synchronous rounds of communication. During each round, the machines first perform computation locally without communicating with other machines. The computation done locally can be unbounded (although the machines have limited space so any reasonable program will not do an absurdly large amount of computation). At the end of the round, the machines exchange messages to inform the computation for the next round. The total size of all messages that can be received by a machine is upper bounded by the size of its local memory, and each machine outputs messages of sufficiently small size that can fit into its memory.

The purpose of such a model is to mimic distributed systems that process data that cannot all fit on one machine. The memory per machine is fixed and the complexity of an algorithm in this model is measured by the ***total space*** used and the ***number of rounds of communication***. Provided a graph $G = (V, E)$, efficient algorithms in this model use $O(n + m)$ total space and $O(\log n)$ or $O(1)$ rounds of communication.

Notation There exist $\mathcal{M}$ ***machines*** that communicate with each other in synchronous rounds. If N is the total size of the data and each machine has S words of space, we are interested in settings where S is sublinear in N. We use ***total space*** to refer to $\mathcal{M} \cdot S$, which represent the joined available space across all the machines.

2.2.3 External-Memory Model

In this model, memory is divided between a fast, but small internal memory (***cache***), and a slow, but large external memory (***disk***). Computation is performed in the cache. Sometimes not all data required in a computation can fit in cache, thus, the disk is needed to store the extra data necessary in the computation. Given a fixed cache size, the goal of cache-efficient algorithms is to minimize the number of reads and writes from disk. A read/write into disk is known as an ***I/O operation***. Then, the number of read/writes into disk is generally called the ***I/O complexity***. The general assumption made in this model is that the I/O complexity takes up the majority of the computation time. In other words, the time required to perform computations in cache is negligible compared to the time required to perform read/writes into disk. Thus, the complexity measure used to determine the efficiency of an algorithm in this model is the number of I/Os or I/O complexity.

Notation The cache has size M and is connected to a disk of unbounded size. Both the internal and external memory are divided into blocks of size B. One I/O consists of moving one block of B contiguous elements from cache to disk or vice versa.

2.3 Problem Definitions

2.3.1 $(\Delta + 1)$-Vertex Coloring

Given an input graph $G = (V, E)$, a valid $\mathbf{(\Delta + 1)}$***-vertex coloring*** exists if and only if no two adjacent vertices (two vertices u and v are adjacent if $(u, v) \in E$) are colored the same color and at most $(\Delta + 1)$ colors are used.

2.3.2 k-Clique Counting

Given an undirected graph $G = (V, E)$ with n vertices and m edges, and an integer k, a ***k*-clique** is defined as a set of k vertices $v_0, \ldots, v_k$ such that for all $i \neq j$, $(v_i, v_j) \in E$. The ***k*-clique count** is the total number of k-cliques in the graph. The **dynamic *k*-clique problem** maintains the number of k-cliques in the graph upon edge insertions and deletions, given individually or in a batch.

A quick note on notation: in this thesis, we choose to refer to k-core decomposition and k-clique counting both using the variable k. We made this choice since previous literature tend to use this variable when discussing these two topics. To avoid confusion, we preface each usage of k with the quantity it refers to.

2.3.3 Pebble Games

A ***pebble game*** is a one-player game played on a directed acyclic graph (DAG) where the goal of the player is to place pebbles on a set of one or more ***target nodes*** in the DAG. A pebble can be placed and moved according to some movement rules that are different for each type of pebble game. Given a directed acyclic computation graph, the number of pebbles used at any time represents the maximum *space* used in the computation. The number of pebble placements and moves represents the *time* necessary to perform the computation. The sequence of pebble placements and moves is also sometimes referred to as a *schedule* for the given computation graph.

Given an input DAG, the goal is to minimize the maximum number of pebbles on the graph at any time or the number of pebble moves/placements, or both.

2.4 Probability Bounds

Throughout the thesis, when we say ***high probability***, we mean with probability $1 - \frac{1}{n^c}$ for any constant $c \geq 1$. We also use the following probability bounds in this thesis.

Definition 2.4.1 (Chebyshev's Inequality)**.** *Let X denote a real-valued random variable with finite expected value μ and finite, non-zero variance, σ^2. For any $c > 0$,*

$$\mathbb{P}[|X - \mu| \geq c\sigma] \leq \frac{1}{c^2}.$$

Definition 2.4.2 (Chernoff Bound)**.** *Let $Y_1, \ldots, Y_m$ be m independent random variables that take on values in $[0,1]$ where $\mathbb{E}[Y_i] = p_i$ and $\sum_{i=1}^{m} p_i = P$. For any $\gamma \in (0,1]$, the multiplicative **Chernoff bound** gives*

$$\mathbb{P}\left[\sum_{i=1}^{m} Y_i > (1+\gamma)P\right] < \exp\left(-\gamma^2 P/3\right)$$

and

$$\mathbb{P}\left[\sum_{i=1}^{m} Y_i < (1-\gamma)P\right] < \exp\left(-\gamma^2 P/2\right).$$

Part II

Static Graph Algorithms

Overview

A teacher announces that a test will be given on one of the five week days of next week, but tells the class, "You will not know which day it is until you are informed at 8a.m. of your 1p.m. test that day. Why isn't the test going to be given? The Joy of Mathematics [Pap89].[1]

This part of the thesis presents novel static graph algorithms designed for real-world networks.

Structural Rounding Chapter 3 introduces a framework called *structural rounding* that provides approximation algorithms for hard-to-solve graph problems in classes of graphs that are close to some structured graph class. The framework consists of three steps: *edit* to a target class of graphs, apply an existing approximation algorithm to the edited graph, and then *lift* the solution to the original graph. An *edit* operation consists of the removal of an edge or vertex. The motivation behind such a line of study is that we hypothesize many real-world networks, social networks, communication networks, road networks...etc., are close to structured graph classes. Namely, these real-world networks will become part of a structured graph class after a small number of edits compared to the size of the optimum solution in the graph. This framework can be applied to any problem that satisfies a combinatorial property called *stability* and an algorithmic property called *structural lifting*. The first property ensures that the optimum solution does not change by much more than the number of edits that occur in the graph. The second property ensures that any solution in the edited graph can be converted to a solution in the original graph without too much change to the cost of the solution. We show a number of approximation algorithms for well-known NP-hard graph optimization problems including maximum independent set, minimum dominating set, minimum vertex cover, and minimum maximal matching using this framework. The best approximation algorithms resulting from the framework are for classes of graphs that are close to bounded degeneracy and bounded treewidth graphs.

Specifically, in Chapter 3, we show:

- A general framework for obtaining approximation algorithms for problems such as Independent Set, Vertex Cover, Feedback Vertex Set, Minimum Maximal Matching, Chromatic Number, (ℓ-)Dominating Set, Edge (ℓ-)Dominating Set, and Connected Dominating Set in graphs that are $O(\delta \cdot \mathrm{OPT}(G))$-close to bounded degeneracy and bounded treewidth. [Corollary 3.5.7, Corollary 3.5.10,

[1] Rather than quotes from literature (as is traditionally used), each part of this thesis will instead be prefaced with a math puzzle (loosely) related to at least one chapter in the part.

Corollary 3.5.11, Theorem 3.5.12, Corollary 3.5.15, Corollary 3.5.18, Corollary 3.5.19, Corollary 3.5.22, Corollary 3.5.25, Corollary 3.5.26, Corollary 3.5.29]

- A bicriteria $(4,4)$-approximation algorithm for vertex editing to bounded degeneracy that extends to a smoother bicriteria trade-off. [Theorem 3.6.1]
- A bicriteria $(5,5)$-approximation algorithm for edge editing to bounded degeneracy. [Corollary 3.6.16]
- A $O(k \log n)$-approximation algorithm for vertex and edge editing to degeneracy k. This approximation factor is *tight up to constant factors* when k is constant. [Corollary 3.6.24]
- A bicriteria $(O(\log^{1.5} n), O(\sqrt{\log w}))$-approximation algorithm for vertex editing to bounded treewidth w. [Theorem 3.6.31, Theorem 3.6.32]
- A bicriteria $(O(\log^{1.5} n), O(\sqrt{\log w} \cdot \log n))$-approximation algorithm for vertex editing to bounded pathwidth w. [Corollary 3.6.34]

Experimental Followup Work Since the publication of [DGK+19], a recent work empirically evaluated the structural rounding framework. This work showed that the structural rounding framework may lead to faster and more efficient practical implementations of approximation algorithms for graph optimization problems. Specifically, Lavallee, Russell, Sullivan, and van der Poel [LRSvdP20] show that structural rounding combined with various lifting strategies provides better approximations for vertex cover than traditional approaches on a large number of graphs with varying characteristics. It is an interesting open question whether the structural rounding framework is amenable to either the work-depth model or the MPC model.

Small Subgraph Counting in the MPC Model Chapter 4 presents static algorithms for small subgraph counting in the massively parallel computation (MPC) model. Such a model is used in practice at commercial data centers to process large graphs (with hundreds of billions of edges) that cannot fit on a single machine. Suppose the average degree of an input graph $G = (V, E)$ is d_{avg}. We present a $(1+\varepsilon)$-approximation algorithm (for any constant $\varepsilon > 0$) for counting the number of triangles in the input graph in $O(n^\delta)$ space per machine (for any constant $\delta > 0$), $O(1)$ rounds, and $\widetilde{O}(n+m)$ total space [2] when the number of triangles T is lower bounded $T \geq \sqrt{d_{avg}}$. This quadratically improves on the best-known previous lower bound on T by [PT11]. We also present an exact algorithm for triangle counting in graphs of arboricity α, that works in $O(n^\delta)$ space per machine (for any constant $\delta > 0$), $O(m\alpha)$ total space, and $O(\log \log n)$ rounds. To the best of our knowledge, this is the first algorithm that achieves $o(\log n)$ rounds for exact triangle counting in graphs with arboricity bounded by α. All of our results can be extended to cliques with a slight increase in total space and/or number of rounds. Furthermore, the approximation algorithm can be extended to any subgraph with k vertices for any constant k.

Specifically, in Chapter 4, we show:

- An $(1+\varepsilon)$-approximation algorithm for T, the number of triangles, *whp*, when $T \geq \sqrt{d_{avg}}$ where $d_{avg} = m/n$. The algorithm uses $O(n^\delta)$ space per machine for

[2] $\widetilde{O}$ is typically used in theoretical computer science to hide $\operatorname{poly}\log n$ factors.

any constant $\delta > 0$, $\widetilde{O}(n+m)$ total space, and $O(1)$ rounds. [Theorem 4.1.1, Corollary 4.1.2]

- An $(1+\varepsilon)$-approximation algorithm for counting the number of occurrences of H, a subgraph of size K (where K is constant), *whp*, when $B \geq d_{avg}^{K/2-1}$ where B is the number of occurrences of H in the graph and $d_{avg} = m/n$. The algorithm uses $O(n^\delta)$ space per machine for any constant $\delta > 0$, $\widetilde{O}(n+m)$ total space, and $O(1)$ rounds. [Theorem 4.5.13, Corollary 4.5.14]
- An exact algorithm for counting the number of triangles given $O(n^\delta)$ space per machine (for constant $\delta > 0$) that uses $O(m\alpha)$ total space, and $O(\log\log n)$ rounds where α is the arboricity of the graph. [Theorem 4.1.3]
- An exact algorithm for counting the number of k-cliques given $O(n^\delta)$ space per machine (for constant $\delta > 0$) that uses $O(m\alpha^{k-2})$ total space and $O(\log\log n)$ rounds where α is the arboricity of the graph. [Theorem 4.6.4]
- An exact algorithm for counting the number of k-cliques that requires $\Omega(\alpha^2)$ space per machine and uses $O(n\alpha^2)$ total space and $O(\log\log n)$ rounds where α is the arboricity of the graph. [Theorem 4.6.5]

Scheduling with Communication Delay in Near-Linear Time Chapter 5 considers the problem of efficiently scheduling jobs with precedence constraints on a set of identical machines in the presence of a uniform communication delay. Such precedence-constrained jobs can be modeled as a directed acyclic graph, $G = (V, E)$. In this setting, if two precedence-constrained jobs u and v, with v dependent on u ($u \prec v$), are scheduled on different machines, then v must start at least ρ time units after u completes. The scheduling objective is to minimize makespan, i.e. the total time from when the first job starts to when the last job finishes. The focus of this chapter is to provide an efficient approximation algorithm with near-linear running time. We build on the algorithm of Lepere and Rapine [STACS 2002] for this problem to give an $O\left(\frac{\ln\rho}{\ln\ln\rho}\right)$-approximation algorithm that runs in $\tilde{O}(|V|+|E|)$ time. Specifically, we show:

- There is an $O(\ln\rho/\ln\ln\rho)$-approximation algorithm for scheduling jobs with precedence constraints on a set of identical machines in the presence of a uniform communication delay that runs in $O\left(n\ln M + \frac{m\ln^3 n\ln\rho}{\ln\ln\rho}\right)$ time, assuming that the optimal solution has cost at least ρ. [Theorem 5.1.1]

Bibliographic Information The results in Part II are based off the following works:

1. [DGK+19] Erik D. Demaine, Timothy D. Goodrich, Kyle Kloster, Brian Lavallee, Quanquan C. Liu, Blair D. Sullivan, Ali Vakilian, and Andrew van der Poel. Structural rounding: Approximation algorithms for graphs near an algorithmically-tractable class. (Chapter 3)
2. [BEL+20] Amartya Shankha Biswas, Talya Eden, Quanquan C. Liu, Slobodan Mitrović, and Ronitt Rubinfeld. Parallel algorithms for small subgraph counting. (Chapter 4)
3. [LPS+21] Quanquan C. Liu, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua R. Wang. Scheduling with communication delay in near-linear time. (Chapter 5)

Chapter 3

Approximation Algorithms for Graphs Near an Algorithmically Tractable Class

This chapter presents results from the paper titled, "Structural rounding: Approximation algorithms for graphs near an algorithmically tractable class" that the thesis author coauthored with Erik D. Demaine, Timothy D. Goodrich, Kyle Kloster, Brian Lavallee, Blair D. Sullivan, Ali Vakilian and Andrew van der Poel [DGK+19]. This paper appeared in the European Symposium on Algorithms (ESA), 2019.

3.1 Introduction

Network science has empirically established that real-world networks (social, biological, computer, etc.) exhibit significant sparse structure. Theoretical computer science has shown that graphs with certain structural properties enable significantly better approximation algorithms for hard problems. Unfortunately, the experimentally observed structures and the theoretically required structures are generally not the same: mathematical graph classes are rigidly defined, while real-world data is noisy and full of exceptions. This chapter provides a framework for extending approximation guarantees from existing rigid classes to broader, more flexible graph families that are more likely to include real-world networks.

Specifically, we hypothesize that most real-world networks are in fact *small perturbations of graphs from a structural class.* Intuitively, these perturbations may be exceptions caused by unusual/atypical behavior (e.g., weak links rarely expressing themselves), natural variation from an underlying model, or noise caused by measurement error or uncertainty. Formally, a graph is γ*-close* to a structural class $\mathcal{C}$, where $\gamma \in \mathbb{N}$, if some γ edits (e.g., vertex deletions, edge deletions, or edge contractions) bring the graph into class $\mathcal{C}$.

Our goal is to extend existing approximation algorithms for a structural class $\mathcal{C}$ to apply more broadly to graphs γ-close to $\mathcal{C}$. To achieve this goal, we need two algorithmic ingredients:

1. **Editing algorithms.** Given a graph G that is γ-close to a structural class $\mathcal{C}$, find a

sequence of $f(\gamma)$ edits that edit G into $\mathcal{C}$. When the structural class is parameterized (e.g., treewidth $\leq w$), we may also approximate those parameters.

2. **Structural rounding algorithms.** Develop approximation algorithms for optimization problems on graphs γ-close to a structural class $\mathcal{C}$ by converting ρ-approximate solutions on an edited graph in class $\mathcal{C}$ into $g(\rho, \gamma)$-approximate solutions on the original graph.

3.1.1 Our Results: Structural Rounding

In Section 3.5, we present a general metatheorem giving sufficient conditions for an optimization problem to be amenable to the structural rounding framework. Specifically, if a problem Π has an approximation algorithm in structural class $\mathcal{C}$, the problem and its solutions are "stable" under an edit operation, and there is an α-approximate algorithm for editing to $\mathcal{C}$, then we get an approximation algorithm for solving Π on graphs γ-close to $\mathcal{C}$. The new approximation algorithm incurs an additive error of $O(\gamma)$, so we preserve PTAS-like $(1+\varepsilon)$ approximation factors provided $\gamma \leq \delta \mathrm{OPT}_\Pi$ for a suitable constant $\delta = \delta(\varepsilon, \alpha) > 0$.

For example, we obtain $(1 + O(\delta \log^{1.5} n))$-approximation algorithms for Vertex Cover, Feedback Vertex Set, Minimum Maximal Matching, and Chromatic Number on graphs $(\delta \cdot \mathrm{OPT}_\Pi(G))$-close to having treewidth w via vertex deletions (generalizing exact algorithms for bounded treewidth graphs); and we obtain a $(1-4\delta)/(4k+1)$-approximation algorithm for Independent Set on graphs $(\delta \cdot \mathrm{OPT}_\Pi(G))$-close to having degeneracy k (generalizing a $1/k$-approximation for degeneracy-k graphs). These results use our new algorithms for editing to treewidth-w and degeneracy-k graph classes as summarized next.

3.1.2 Our Results: Editing

We develop editing approximation algorithms and/or hardness-of-approximation results for six well-studied graph classes: bounded clique number, bounded degeneracy, bounded treewidth and pathwidth, bounded treedepth, bounded weak c-coloring number, and

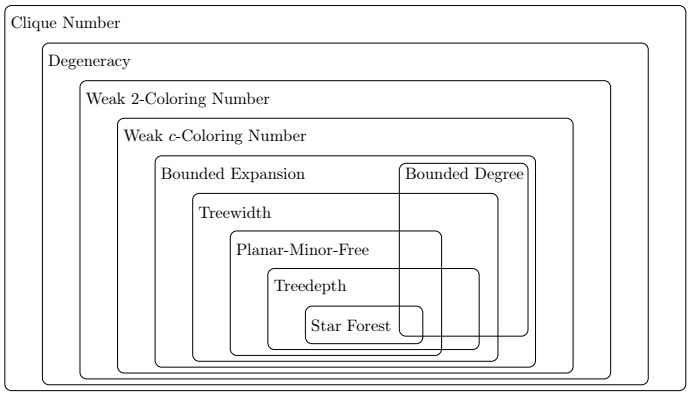


Figure 3-1: Illustration of hierarchy of structural graph classes used in this chapter.

bounded degree. Figure 3-1 summarizes the relationships among these classes, and Table 3.1 summarizes our results for each class.

Graph Family $\mathcal{C}_\lambda$	**Edit Operation ψ**	
	Vertex Deletion	**Edge Deletion**
Bounded Degree (d)	**d-BDD-V** $O(\log d)$-approx. [EKS] $(\ln d - C \cdot \ln\ln d)$-inapprox.	**d-BDD-E** Polynomial time [HP17]
Bounded Degeneracy (k)	**k-DE-V** $O(k \log n)$-approx. $\left(\frac{4m-\beta kn}{m-kn}, \beta\right)$-approx. $\left(\frac{1}{\varepsilon}, \frac{4}{1-2\varepsilon}\right)$-approx. ($\varepsilon < 1/2$) $o(\log(n/k))$-inapprox.	**k-DE-E** $O(k \log n)$-approx. – $\left(\frac{1}{\varepsilon}, \frac{4}{1-\varepsilon}\right)$-approx. ($\varepsilon < 1$) $o(\log(n/k))$-inapprox.
Bounded Weak c-Coloring Number (t)	**t-BWE-V-c** – $o(t)$-inapprox. for $t \in o(\log n)$	**t-BWE-E-c** – $o(t)$-inapprox. for $t \in o(\log n)$
Bounded Treewidth (w)	**w-TW-V** $(O(\log^{1.5} n), O(\sqrt{\log w}))$-approx. $o(\log n)$-inapprox. for $w \in \Omega(n^{1/2})$	**w-TW-E** $(O(\log n \log\log n), O(\log w))$-approx. [BRU17] –
Bounded Pathwidth (w)	**w-PW-V** $(O(\log^{1.5} n), O(\sqrt{\log w} \cdot \log n))$-approx. –	**w-PW-E** $(O(\log n \log\log n), O(\log w \cdot \log n))$-approx. [BRU17] –
Star Forest	**SF-V** 4-approx. $(2-\varepsilon)$-inapprox. (**UGC**)	**SF-E** 3-approx. APX-complete

Table 3.1: Summary of results for $(\mathcal{C}_\lambda, \psi)$-Edit problems (including abbreviations and standard parameter notation). "Approx." denotes a polynomial-time approximation or bicriteria approximation algorithm; "inapprox." denotes inapproximability assuming P $\neq$ NP unless otherwise specified.

This chapter only presents the approximation algorithms given in [DGK+19]; we refer readers to our full paper [DGK+18] for proofs of our lower bound results. In this chapter, we present two bicriteria approximation algorithms for k-DE-V, one using the local ratio theorem and another using LP-rounding. While both approximations can be tuned with error values, they yield constant $(4,4)$- and $(6,6)$-approximations for vertex editing, respectively. Note that the LP-rounding algorithm also gives a $(5,5)$-approximation for k-DE-E. We also give a $O(k \log n)$-approximation algorithm for k-DE-E and k-DE-V. Finally, using vertex separators, we show a $(O(\log^{1.5} n), O(\sqrt{\log w}))$-bicriteria approximation for vertex editing to bounded treewidth and pathwidth.

3.1.3 Related Work

Editing to approximate optimization problems. While there is extensive work on editing graphs into a desired graph class (summarized below), there is little prior work on how editing affects the quality of approximation algorithms (when applied to the edited

graph, but we desire a solution to the original graph). The most closely related results of this type are *parameterized approximation* results, meaning that they run in polynomial time only when the number of edits is very small (constant or logarithmic input size). This research direction was initiated by Cai [Cai03]; see the survey and results of Marx [Mar08, Section 3.2] and e.g. [GHN04, Mar06]. An example of one such result is a $\frac{7}{3}$-approximation algorithm to Chromatic Number in graphs that become planar after γ vertex edits, with a running time of $f(\gamma) \cdot O(n^2)$, where $f(\gamma)$ is at least $2^{2^{2^{2^{\Omega(\gamma)}}}}$ (from the use of Courcelle's Theorem), limiting its polynomial-time use to when the number of edits satisfies $\gamma = O(\log\log\log\log n)$. In contrast, our algorithms allow up to $\delta \mathrm{OPT}_\Pi$ edits.

Another body of related work is the "noisy setting" introduced by Magen and Moharrami [MM09], which imagines that the "true" graph lies in the structural graph class that we want, and any extra edges observed in the given graph are "noise" and thus can be ignored when solving the optimization problem. This approach effectively avoids analyzing the effect of the edge edits on the approximation factor, by asking for a solution to the edited graph instead of the given graph. In this simpler model, Magen and Moharrami [MM09] developed a PTAS for estimating the size of Independent Set (IS) in graphs that are δn edits away from a minor-closed graph family (for sufficiently small values of δ). Later, Chan and Har-Peled [CHP12] developed a PTAS that returns a $(1+\varepsilon)$-approximation to IS in noisy planar graphs. Recently, Bansal et al. [BRU17] developed an LP-based approach for noisy minor-closed IS whose runtime and approximation factor achieve better dependence on δ. Moreover, they provide a similar guarantee for noisy Max k-CSPs. Unlike our work, none of these algorithms bound the approximation ratio for a solution on the original graph.

Editing algorithms. Editing graphs into a desired graph class is an active field of research and has various applications outside of graph theory, including computer vision and pattern matching [GXTL10]. In general, the editing problem is to delete a minimum set X of vertices (or edges) in an input graph G such that the result $G[V \setminus X]$ has a specific property. Previous work studied this problem from the perspective of identifying the maximum induced subgraph of G that satisfies a desired "nontrivial, hereditary" property [KD79, Lew78, LY80, Yan78]. A graph property π is nontrivial if and only if infinitely many graphs satisfy π and infinitely many do not, and π is hereditary if G satisfying π implies that every induced subgraph of G satisfies π. The vertex-deletion problem for any nontrivial, hereditary property has been shown to be NP-complete [LY80] and even requires exponential time to solve, assuming the ETH [Kom18]. Approximation algorithms for such problems have also been studied somewhat [Fuj98, LY93, OB03] in this domain, but in general this problem requires additional restrictions on the input graph and/or output graph properties in order to develop fast algorithms [DGvH$^+$15, Dra15, DDLS15, HKN15, KKO16, Mat10, MS08, Xia16].

Much past work on editing is on parameterized algorithms. For example, Dabrowski et al. [DGvH$^+$15] found that editing a graph to have a given degree sequence is W[1]-complete, but if one additionally requires that the final graph be planar, the problem becomes Fixed Parameter Tractable (FPT). Mathieson [Mat10] showed that editing to de-

generacy k is W[P]-hard (even if the original graph has degeneracy $k+1$ or maximum degree $2k+1$), but suggests that classes which offer a balance between the overly rigid restrictions of bounded degree and the overly global condition of bounded degeneracy (e.g., structurally sparse classes such as H-minor-free and bounded expansion [NdM12]) may still be FPT. Some positive results on the parameterized complexity of editing to classes can be found in Drange's 2015 PhD thesis [Dra15]; in particular, the results mentioned include parameterized algorithms for a variety of NP-complete editing problems such as editing to threshold and chain graphs [DDLS15], star forests [DDLS15], multipartite cluster graphs [FKP+14], and $\mathcal{H}$-free graphs given finite $\mathcal{H}$ and bounded indegree [DDS16].

Our approach differs from this prior work in that we focus on approximations of edit distance that are *polynomial-time approximation algorithms*. There are previous results about approximate edit distance by Fomin et al. [FLMS12] and, in a very recent result regarding approximate edit distance to bounded treewidth graphs, by Gupta et al. [GLL+18]. Fomin et al. [FLMS12] provided two types of algorithms for vertex editing to planar $\mathcal{F}$-minor-free graphs: a randomized algorithm that runs in $O(f(\mathcal{F})\cdot mn)$ time with an approximation constant $c_{\mathcal{F}}$ that depends on $\mathcal{F}$, as well as a fixed-parameter algorithm parameterized by the size of the edit set whose running time thus has an exponential dependence on the size of this edit set.

Gupta et al. [GLL+18] strengthen the results in [FLMS12] but only in the context of *parameterized approximation algorithms*. Namely, they give a deterministic fixed-parameter algorithm for Planar $\mathcal{F}$-Deletion that runs in $f(\mathcal{F})\cdot n\log n + n^{O(1)}$ time and an $O(\log k)$-approximation where k is the maximum number of vertices in any planar graph in $\mathcal{F}$; this implies a fixed-parameter $O(\log w)$-approximation algorithm with running time $2^{O(w^2\log w)}\cdot n\log n + n^{O(1)}$ for w-TW-V and w-PW-V. They also show that w-TW-E and w-PW-E have parameterized algorithms that give an absolute constant factor approximation but with running times parameterized by w and the maximum degree of the graph [GLL+18]. Finally, they show that when $\mathcal{F}$ is the set of all connected graphs with three vertices, deleting the minimum number of edges to exclude $\mathcal{F}$ as a subgraph, minor, or immersion is APX-hard for bounded degree graphs [GLL+18]. Again, these running times are weaker than our results, which give bicriteria approximation algorithms that are polynomial without any parameterization on the treewidth or pathwidth of the target graphs.

In a similar regime, Bansal et al. [BRU17] studied w-TW-E (which implies an algorithm for w-PW-E) and designed an LP-based bicriteria approximation for this problem. For a slightly different set of problems in which the goal is to exclude a single graph H of size s as a subgraph (H-Vertex-Deletion), there exists a simple s-approximation algorithm. On the hardness side, Guruswami and Lee [GL17] proved that whenever H is 2-vertex-connected, it is NP-hard to approximate H-Vertex-Deletion within a factor of $(|V(H)|-1-\varepsilon)$ for any $\varepsilon>0$ ($|V(H)|-\varepsilon$ assuming UGC). Moreover, when H is a star or simple path with s vertices, $O(\log s)$-approximation algorithms with running time $2^{O(s^3\log s)}\cdot n^{O(1)}$ are known [GL17, Lee17].

An important special case of the problem of editing graphs into a desired class is the *minimum planarization* problem, in which the target class is planar graphs, and the related application is approximating the well-known *crossing number* problem [CMS11]. Refer

to [BCDM17, CS13, Chu11, JLS14, KS17, Kaw09, MS12, Sch13] for the recent developments on minimum planarization and crossing number.

3.2 Techniques

This section provides a quick summary of the main techniques, ideas, and contributions in the rest of the chapter. This chapter discusses the structural rounding framework and the editing algorithms to bounded degeneracy and bounded treewidth. For our lower bound results and our editing algorithms for treedepth and bounded degree, please refer to our full paper [DGK+18].

Structural Rounding Framework The main contribution of our structural rounding framework (Section 3.5) is establishing the right definitions that make for a broadly applicable framework with precise approximation guarantees. Our framework supports arbitrary graph edit operations and both minimization and maximization problems, provided they jointly satisfy two properties: a combinatorial property called "stability" and an algorithmic property called "structural lifting". Roughly, these properties bound the amount of change that OPT can undergo from each edit operation, but they are also parameterized to enable us to derive tighter bounds when the problem has additional structure. With the right definitions in place, the framework is simple: edit to the target class, apply an existing approximation algorithm, and lift.

The rest of Section 3.5 shows that this framework applies to many different graph optimization problems. In particular, we verify the stability and structural lifting properties, and combine all the necessary pieces, including our editing algorithms from Section 3.6 and existing approximation algorithms for structural graph classes. We summarize all of these results in Table 3.2 and formally define the framework in Section 3.5.1.

Editing to Bounded Degeneracy and Degree We present two constant-factor bicriteria approximation algorithms for finding the fewest vertex or edge deletions to reduce the *degeneracy* to a target threshold k. The first approach (Section 3.6.1) uses the local ratio technique by Bar-Yehuda et al. [BYBFR04] to establish that good-enough local choices result in a guaranteed approximation. The second approach (Section 3.6.2) is based on rounding a linear-programming relaxation of an integer linear program. Finally, we present a greedy combinatorial algorithm in Section 3.6.4 that uses the degeneracy ordering obtained by Theorem 2.1.7. This algorithm gives a $O(k \log n)$ approximation to an optimum edit set where k is the degeneracy of the *target* class. When k is constant, this approximation matches our lower bound given in our full paper [DGK+18].

Editing to Bounded Treewidth In Section 3.6.5, we present a bicriteria approximation algorithm for finding the fewest vertex edits to reduce the *treewidth* to a target threshold w. Our approach builds on the deep separator structure inherent in treewidth. We combine ideas from Bodlaender's $O(\log n)$-approximation algorithm for treewidth with Feige et al.'s $O(\sqrt{\log w})$-approximation algorithm for vertex separators [FHL08] (where

Problem	**Edit type ψ**	c'	c	**Class $\mathcal{C}_\lambda$**	$\rho(\lambda)$	**runtime**
Independent Set (IS)	vertex deletion	1	0	degeneracy k	$\frac{1}{r+1}$	polytime
Annotated Dominating Set (ADS)	vertex* deletion	0	1	degeneracy k	$O(r)$	polytime [BU17][1]
Independent Set (IS)	vertex deletion	1	0	treewidth w	1	$O(2^w n)$ [AN01]
Annotated Dominating Set (ADS)	vertex* deletion	0	1	treewidth w	1	$O(3^w n)$
Annotated (ℓ-)Dominating Set (ADS)	vertex* deletion	0	1	treewidth w	1	$O((2\ell+1)^w n)$ [BL17]
Connected Dominating Set (CDS)	vertex* deletion	0	3	treewidth w	1	$O(n^w)$[2]
Vertex Cover (VC)	vertex deletion	0	1	treewidth w	1	$O(2^w n)$ [AN01]
Feedback Vertex Set (FVS)	vertex deletion	0	1	treewidth w	1	$2^{O(w)} n^{O(1)}$ [CNP+11]
Minimum Maximal Matching (MMM)	vertex deletion	0	1	treewidth w	1	$O(3^w n)$[3]
Chromatic Number (CRN)	vertex deletion	0	1	treewidth w	1	$w^{O(w)} n^{O(1)}$
Independent Set (IS)	edge deletion	0	1	degeneracy k	$\frac{1}{r+1}$	polytime
Dominating Set (DS)	edge deletion	1	0	degeneracy k	$O(k)$	polytime [BU17]
(ℓ-)Dominating Set (DS)	edge deletion	1	0	treewidth w	1	$O((2\ell+1)^w n)$ [BL17]
Edge (ℓ-)Dominating Set (EDS)	edge deletion	1	1	treewidth w	1	$O((2\ell+1)^w n)$ [BL17]
Max-Cut (MC)	edge deletion	1	0	treewidth w	1	$O(2^w n)$ [DF13]

Table 3.2: Problems for which structural rounding (Theorem 3.5.4) results in approximation algorithms for graphs near the structural class $\mathcal{C}$, where the problem has a $\rho(\lambda)$-approximation algorithm. We also give the associated stability (c') and lifting (c) constants, which are class-independent. The last column shows the running time of the $\rho(\lambda)$-approximation algorithm for each problem provided an input graph from class $\mathcal{C}_\lambda$. We remark that vertex* is used to emphasize the rounding process has to pick the set of annotated vertices in the edited set carefully to achieve the associated stability and lifting constants.

w is the target treewidth). In the end, we obtain a bicriteria $(O(\log^{1.5} n), O(\sqrt{\log w}))$-approximation that runs in polynomial time on all graphs (in contrast to many previous treewidth algorithms). The tree decompositions that we generate are guaranteed to have $O(\log n)$ height. As a result, we also show a bicriteria $(O(\log^{1.5} n), O(\sqrt{\log w} \cdot \log n))$-approximation result for pathwidth, based on the fact that the *pathwidth* is at most the width times the height of a tree decomposition.

3.3 Treedwith and Pathwidth

In this section, we provide the additional necessary definitions for treewidth and pathwidth which are two structural graph classes used in this chapter (illustrated in Figure 3-1). This chapter also uses the definition of degeneracy given in Definition 2.1.1.

Perhaps the most heavily studied structural graph class is that of *bounded treewidth*; in this subsection we provide the necessary definitions for treewidth and pathwidth.

Definition 3.3.1 ([RS86]). *Given a graph G, a* tree decomposition *of G consists of a collection $\mathcal{Y}$ of subsets (called* bags*) of vertices in $V(G)$ together with a tree $T = (\mathcal{Y}, \mathcal{E})$ whose nodes $\mathcal{Y}$ correspond to bags which satisfy the following properties:*

1. *Every $v \in V(G)$ is contained in a bag $B \in \mathcal{Y}$ (i.e. $\bigcup_{B \in \mathcal{Y}} B = V$).*
2. *For all edges $(u, v) \in E(G)$ there is a bag $B \in \mathcal{Y}$ that contains both endpoints u, v.*
3. *For each $v \in V(G)$, the set of bags containing v form a connected subtree of T (i.e. $\{B | v \in B, B \in \mathcal{Y}\}$ forms a subtree of T).*

The width *of a tree decomposition is $\max_{B \in \mathcal{Y}} |B| - 1$, and the* treewidth *of a graph G, denoted $\mathsf{tw}(G)$, is the minimum width of any tree decomposition of G.*

All graphs that exclude a simple fixed planar minor H have bounded treewidth, indeed, treewidth $|V(H)|^{O(1)}$ [CC16]. Thus, every planar-H-minor-free graph class is a subclass of some bounded treewidth graph class.

Definition 3.3.2 ([RS86]). *A* path decomposition *is a tree decomposition in which the tree T is a path. The* pathwidth *of G, $\mathsf{pw}(G)$, is the minimum width of any path decomposition of G.*

3.4 Editing and Optimization Problems

3.4.1 Editing Problems

This chapter is concerned with algorithms that edit graphs into a desired structural class, while guaranteeing an approximation ratio on the size of the edit set. Besides its own importance, editing graphs into structural classes plays a key role in our structural rounding framework for approximating optimization problems on graphs that are "close" to structural graph classes (see Section 3.5). The basic editing problem is defined as follows relative to an edit operation ψ such as vertex deletion, edge deletion, or edge contraction:

$(\mathcal{C}, \psi)$-Edit

Input: An input graph $G = (V, E)$, family $\mathcal{C}$ of graphs, edit operation ψ
Problem: Find k edits $\psi_1, \psi_2, \ldots, \psi_k$ such that $\psi_k \circ \psi_{k-1} \circ \cdots \circ \psi_2 \circ \psi_1(G) \in \mathcal{C}$.
Objective: Minimize k

The literature has limited examples of approximation algorithms for specific edit operations and graph classes. Most notably, Fomin et al. [FLMS12] studies $(\mathcal{C}, \psi)$-Edit for vertex deletions into the class of *planar-H-minor-free graphs* (graphs excluding a fixed planar graph H).[4]

In addition to fixed-parameter algorithms (for when k is small), they give a c_H-approximation algorithm for $(\mathcal{C}, \psi)$-Edit where the constant $c_H = \Omega\left(2^{2^{|V(H)|^3}}\right)$ is rather large.

Most of the graph classes we consider consist of graphs where some parameter λ (clique number, maximum degree, degeneracy, weak c-coloring number, or treewidth) is bounded. Thus we can think of the graph class $\mathcal{C}$ as in fact being a parameterized family $\mathcal{C}_\lambda$. (For planar-H-minor-free, λ could be $|V(H)|$.) We can also loosen the graph class we are aiming for, and approximate the parameter value λ for the family $\mathcal{C}_\lambda$. Thus we obtain a *bicriteria problem* which can be formalized as follows:

$(\mathcal{C}_\lambda, \psi)$-Edit

Input: An input graph $G = (V, E)$, parameterized family $\mathcal{C}_\lambda$ of graphs, a target parameter value λ^*, edit operation ψ
Problem: Find k edits $\psi_1, \psi_2, \ldots, \psi_k$ such that $\psi_k \circ \psi_{k-1} \circ \cdots \circ \psi_2 \circ \psi_1(G) \in \mathcal{C}_\lambda$ where $\lambda \geq \lambda^*$.
Objective: Minimize k.

Definition 3.4.1. *An algorithm for $(\mathcal{C}_\lambda, \psi)$-Edit is a* (bicriteria) (α, β)-approximation *if it guarantees that the number of edits is at most α times the optimal number of edits into $\mathcal{C}_\lambda$, and that $\lambda \leq \beta \cdot \lambda^*$.*

See Table 3.1 for a complete list of the problems considered, along with their abbreviations. Recall that $\rho(\lambda)$ is the approximation factor for a problem in class $\mathcal{C}$. We assume that $\mathcal{C}_i \subseteq \mathcal{C}_j$ for $i \leq j$, or equivalently, that $\rho(\lambda)$ is monotonically increasing in λ.

3.4.2 Optimization Problems

We conclude this section with formal definitions of several additional optimization problems for which we give new approximation algorithms via structural rounding in Section 3.5.

[4] More generally, Fomin et al. [FLMS12] consider editing to the class of graphs excluding a finite family $\mathcal{F}$ of graphs at least one of which is planar, but as we just want the fewest edits to put the graph in some structural class, we focus on the case $|\mathcal{F}| = 1$.

ℓ-Dominating Set (ℓ-DS)

Input: An undirected graph $G = (V, E)$ and a positive integer ℓ.
Problem: Find a minimum size set of vertices $C \subseteq V$ s.t. every vertex in V is either in C or is connected by a path of length at most ℓ to a vertex in C.

Edge ℓ-Dominating Set (ℓ-EDS)

Input: An undirected graph $G = (V, E)$ and a positive integer ℓ.
Problem: Find a minimum size set of edges $C \subseteq E$ s.t. every edge in E is either in C or is connected by a path of length at most ℓ to an edge in C.

When $\ell = 1$, these are Dominating Set (DS) and Edge Dominating Set (EDS).

Annotated (ℓ-)Dominating Set (ADS)

Input: An undirected graph $G = (V, E)$, a subset of vertices $B \subseteq V$ and a positive integer.
Problem: Find a minimum size set of vertices $C \subseteq V$ s.t. every vertex in B is either in C or is connected by a path of length at most ℓ to a vertex in C.

Note that when $B = V$, Annotated (ℓ-)Dominating Set becomes (ℓ-)Dominating Set[5].

ℓ-Independent Set (ℓ-IS)

Input: A graph $G = (V, E)$.
Problem: Find a maximum size set of vertices $X \subseteq V$ s.t. no two vertices in X are connected by a path of length $\leq \ell$.

When $\ell = 1$, we call this Independent Set (IS).

Feedback Vertex Set (FVS)

Input: A graph $G = (V, E)$.
Problem: Find a minimum size set of vertices $X \subseteq V$ s.t. $G \setminus X$ has no cycles.

Minimum Maximal Matching (MMM)

Input: A graph $G = (V, E)$.
Problem: Find a minimum size set of edges $X \subseteq E$ s.t. X is a maximal matching.

[5]The Annotated Dominating Set problem has also been studied in the literature as *subset dominating set problem* in [GK96, HQ17].

Chromatic Number (CRN)

Input: A graph $G = (V, E)$.
Problem: Find a minimum size coloring of G s.t. adjacent vertices are different colors.

Max-Cut (MC)

Input: A graph $G = (V, E)$.
Problem: Find a partition of the nodes of G into sets S and $V \setminus S$ such that the number of edges from S to $V \setminus S$ is greatest.

3.5 Structural Rounding

In this section, we show how approximation algorithms for a structural graph class can be extended to graphs that are near that class, provided we can find a certificate of being near the class. These results thus motivate our results in later sections about editing to structural graph classes. Our general approach, which we call *structural rounding*, is to apply existing approximation algorithms on the edited ("rounded") graph in the class, then "lift" that solution to solve the original graph, while bounding the loss in solution quality throughout.

3.5.1 General Framework

First we define our notion of "closeness" in terms of a general family ψ of allowable graph edit operations (e.g., vertex deletion, edge deletion, edge contraction):

Definition 3.5.1. *A graph G' is* γ-editable *from a graph G under edit operation ψ if there is a sequence of $k \leq \gamma$ edits $\psi_1, \psi_2, \ldots, \psi_k$ of type ψ such that $G' = \psi_k \circ \psi_{k-1} \circ \cdots \circ \psi_2 \circ \psi_1(G)$. A graph G is* γ-close *to a graph class $\mathcal{C}$ under ψ if some $G' \in \mathcal{C}$ is γ-editable from G under ψ.*

To transform an approximation algorithm for a graph class $\mathcal{C}$ into an approximation algorithm for graphs γ-close to $\mathcal{C}$, we will need two properties relating the optimization problem and the type of edits:[6]

Definition 3.5.2. *A graph minimization (resp. maximization) problem Π is* stable *under an edit operation ψ with constant c' if $\mathrm{OPT}_\Pi(G') \leq \mathrm{OPT}_\Pi(G) + c'\gamma$ (resp. $\mathrm{OPT}_\Pi(G') \geq \mathrm{OPT}_\Pi(G) - c'\gamma$) for any graph G' that is γ-editable from G under ψ. In the special case where $c' = 0$, we call Π* closed *under ψ. When ψ is vertex deletion, closure is equivalent to the graph class defined by $\mathrm{OPT}_\Pi(G) \leq \lambda$ (resp. $\mathrm{OPT}_\Pi(G) \geq \lambda$) being* hereditary*; we also call Π* hereditary*.*

[6] These conditions are related to, but significantly generalize, the "separation property" from the bidimensionality framework for PTASs [DH05].

Definition 3.5.3. *A minimization (resp. maximization) problem* Π *can be* structurally lifted *with respect to an edit operation* ψ *with constant* c *if, given any graph* G' *that is* γ*-editable from* G *under* ψ*, and given the corresponding edit sequence* $\psi_1, \psi_2, \ldots, \psi_k$ *with* $k \leq \gamma$*, a solution* S' *for* G' *can be converted in polynomial time to a solution* S *for* G *such that* $\text{Cost}_\Pi(S) \leq \text{Cost}_\Pi(S') + c \cdot k$ *(resp.* $\text{Cost}_\Pi(S) \geq \text{Cost}_\Pi(S') - c \cdot k$*).*

Now we can state the main result of structural rounding:

Theorem 3.5.4 (Structural Rounding Approximation)**.** *Let* Π *be a minimization (resp. maximization) problem that is stable under the edit operation* ψ *with constant* c' *and that can be structurally lifted with respect to* ψ *with constant* c*. If* Π *has a polynomial-time* $\rho(\lambda)$*-approximation algorithm in the graph class* $\mathcal{C}_\lambda$*, and* $(\mathcal{C}_\lambda, \psi)$*-*Edit *has a polynomial-time* (α, β)*-approximation algorithm, then there is a polynomial-time* $((1 + c'\alpha\delta) \cdot \rho(\beta\lambda) + c\alpha\delta)$*-approximation (resp.* $((1 - c'\alpha\delta) \cdot \rho(\beta\lambda) - c\alpha\delta)$*-approximation) algorithm for* Π *on any graph that is* $(\delta \cdot \text{OPT}_\Pi(G))$*-close to the class* $\mathcal{C}_\lambda$*.*

Proof. We write $\text{OPT}(G)$ for $\text{OPT}_\Pi(G)$. Let G be a graph that is $(\delta \cdot \text{OPT}(G))$-close to the class $\mathcal{C}_\lambda$. By Definition 3.4.1, the polynomial-time (α, β)-approximation algorithm finds edit operations $\psi_1, \psi_2, \ldots, \psi_k$ where $k \leq \alpha\delta \cdot \text{OPT}(G)$ such that $G' = \psi_k \circ \psi_{k-1} \circ \cdots \circ \psi_2 \circ \psi_1(G) \in \mathcal{C}_{\beta\lambda}$.[7] Let $\rho = \rho(\beta\lambda)$ be the approximation factor we can attain on the graph $G' \in \mathcal{C}_{\beta\lambda}$.

First we prove the case when Π is a minimization problem. Because Π has a ρ-approximation in $\mathcal{C}_{\beta\lambda}$ (where $\rho > 1$), we can obtain a solution S' with cost at most $\rho \cdot \text{OPT}(G')$ in polynomial time. Applying structural lifting (Definition 3.5.3), we can use S' to obtain a solution S for G with $\text{Cost}(S) \leq \text{Cost}(S') + ck \leq \text{Cost}(S') + c\alpha\delta \cdot \text{OPT}(G)$ in polynomial time. Because Π is stable under ψ with constant c',

$$\text{OPT}(G') \leq \text{OPT}(G) + c'k \leq \text{OPT}(G) + c'\alpha\delta \cdot \text{OPT}(G) = (1 + c'\alpha\delta)\,\text{OPT}(G),$$

and we have

$$\begin{aligned} \text{Cost}(S) &\leq \rho \cdot \text{OPT}(G') + c\alpha\delta \cdot \text{OPT}(G) \\ &\leq \rho(1 + c'\alpha\delta)\,\text{OPT}(G) + c\alpha\delta \cdot \text{OPT}(G) \\ &= (\rho + \rho c'\alpha\delta + c\alpha\delta)\,\text{OPT}(G), \end{aligned}$$

proving that we have a polynomial time $(\rho + (c + c'\rho)\alpha\delta)$-approximation algorithm as required.

Next we prove the case when Π is a maximization problem. Because Π has a ρ-approximation in $\mathcal{C}$ (where $\rho < 1$), we can obtain a solution S' with cost at least $\rho \cdot \text{OPT}(G')$ in polynomial time. Applying structural lifting (Definition 3.5.3), we can use S' to obtain a solution S for G with $\text{Cost}(S) \geq \text{Cost}(S') - ck \geq \text{Cost}(S') - c\alpha\delta \cdot \text{OPT}(G)$ in polynomial time. Because Π is stable under ψ with constant c',

$$\text{OPT}(G') \geq \text{OPT}(G) - c'k \geq \text{OPT}(G) - c'\alpha\delta \cdot \text{OPT}(G) = (1 - c'\alpha\delta)\,\text{OPT}(G),$$

[7] We assume that $C_i \subseteq C_j$ for $i \leq j$, or equivalently, that $\rho(\lambda)$ is monotonically increasing in λ.

and we have

$$\begin{aligned}\mathrm{Cost}(S) &\geq \rho \cdot \mathrm{OPT}(G') - c\alpha\delta \cdot \mathrm{OPT}(G) \\ &\geq \rho(1 - c'\alpha\delta)\mathrm{OPT}(G) - c\alpha\delta \cdot \mathrm{OPT}(G) \\ &= (\rho - (c + c'\rho)\alpha\delta)\mathrm{OPT}(G),\end{aligned}$$

proving that we have a polynomial-time $(\rho - (c + c'\rho)\alpha\delta)$-approximation algorithm as required. Note that this approximation is meaningful only when $\rho > (c + c'\rho)\alpha\delta$. □

To apply Theorem 3.5.4, we need four ingredients: (a) a proof that the problem of interest is stable under some edit operation (Definition 3.5.2); (b) a polynomial-time (α, β)-approximation algorithm for editing under this operation (Definition 3.4.1); (c) a structural lifting algorithm (Definition 3.5.3); and (d) an approximation algorithm for the target class $\mathcal{C}$.

In the remainder of this section, we show how this framework applies to many problems and graph classes, as summarized in Table 3.2 on page 57. Most of our approximation algorithms depend on our editing algorithms described in Section 3.6. We present the problems ordered by edit type, as listed in Table 3.2.

Structural rounding for annotated problems. We refer to graph optimization problems where the input consists of both a graph and subset of annotated vertices/edges as *annotated* problems (see Annotated Dominating Set in Section 3.4.2). Hence, in our rounding framework, we have to carefully choose the set of annotated vertices/edges in the edited graph to guarantee small *lifting* and *stability* constants. To emphasize the difference compared to "standard" structural rounding, we denote the edit operations as vertex* and edge* in the annotated cases. Moreover, we show that we can further leverage the flexibility of annotated rounding to solve *non-annotated* problems that cannot normally be solved via structural rounding. In Section 3.5.4, we consider applications of annotated rounding for both annotated problems such as Annotated Dominating Set and non-annotated problems such as Connected Dominating Set.

3.5.2 Vertex Deletions

For each problem, we show stability and structural liftability, and use these to conclude approximation algorithms. Because IS is the only maximization problem we first consider in this section, we consider it separately.

Lemma 3.5.5. *Independent Set is stable under vertex deletion with constant $c' = 1$.*

Proof. Given a graph G and any set $X \subseteq V(G)$ with $|X| \leq \gamma$, let $G' = G[V \setminus X]$. For any independent set $Y \subset V(G)$, $Y' = Y \setminus X$ is also an independent set in G' with size $|Y'| \geq |Y| - |X|$, which is bounded below by $|Y| - \gamma$. In particular, for Y optimal in G we have $|Y'| \geq \mathrm{OPT}(G) - \gamma$, and so $\mathrm{OPT}(G') \geq \mathrm{OPT}(G) - \gamma$. □

Lemma 3.5.6. *Independent Set can be structurally lifted with respect to vertex deletion with constant $c = 0$.*

Proof. An independent set in $G' = G \setminus X$ is also an independent set in G. Thus, a solution S' for G' yields a solution S for G such that $\text{Cost}_{\text{IS}}(S') = \text{Cost}_{\text{IS}}(S)$. □

Corollary 3.5.7. *For graphs $(\delta \cdot \text{OPT}(G))$-close to a graph class $\mathcal{C}_\lambda$ via vertex deletions, Independent Set has the following approximations. For degeneracy k, IS has a $(1-4\delta)/(4k+1)$-approximation, for treewidth w such that $w\sqrt{\log w} = O(\log n)$, IS has a $(1 - O(\delta \log^{1.5} n))$-approximation, and for planar-H-minor-free, IS has a $(1 - c_H\delta)$-approximation.*

Proof. We apply Theorem 3.5.4 using stability with $c' = 1$ (Lemma 3.5.5) and structural lifting with $c = 0$ (Lemma 3.5.6). The independent-set approximation algorithm and the editing approximation algorithm depend on the class $\mathcal{C}_\lambda$.

For degeneracy k, we use our $(4, 4)$-approximate editing algorithm (Section 3.6.2) and a simple $1/(k+1)$-approximation algorithm for independent set: the k-degeneracy ordering on the vertices of a graph gives a canonical $(k+1)$-coloring, and the pigeonhole principle guarantees an independent set of size at least $|V|/(k+1)$, which is at least $1/(k+1)$ times the maximum independent set. Thus $\alpha = \beta = 4$ and $\rho(\beta k) = 1/(\beta k + 1)$, resulting in an approximation factor of $(1 - 4\delta)/(4k + 1)$.

For treewidth w such that $w\sqrt{\log w} = O(\log n)$, we use our $(O(\log^{1.5} n), O(\sqrt{\log w}))$-approximate editing algorithm (Section 3.6.5) and an exact algorithm for independent set [Bod88, AN01] given a tree decomposition of width $O(\log n)$ of the edited graph. Thus $\alpha = O(\log^{1.5} n)$ and $\rho = 1$, resulting in an approximation factor of $1 - O(\log^{1.5} n)\delta$.

For planar-H-minor-free, we use Fomin's c_H-approximate editing algorithm [FLMS12] and the same exact algorithm for IS in bounded treewidth (as any planar-H-minor-free graph has bounded treewidth [CC16]). Thus $\alpha = c_H$ and $\rho = 1$, resulting in an approximation factor of $1 - c_H\delta$. □

Lemma 3.5.8. *The problems Vertex Cover, Feedback Vertex Set, Minimum Maximal Matching, and Chromatic Number are hereditary (closed under vertex deletion).*

Proof. Let G be a graph, and $G' = G \setminus X$ where $X \subseteq V(G)$. Any vertex cover in G remains a cover in G' because $E(G') \subseteq E(G)$, so VC is hereditary.

Let S bs a feedback vertex set in G and $S' = S \setminus X$. For FVS, we observe that removing vertices can only decrease the number of cycles in the graph. Deleting a vertex in S breaks all cycles it is a part of and, thus, the cycles no longer need to be covered by a vertex in the feedback vertex set of G'. Deleting a vertex not in S can only decrease the number of cycles, and, thus, all cycles in G' are still covered by S'. Hence, FVS is hereditary.

For MMM, deleting vertices with adjacent edges not in the matching only decreases the number of edges; thus, the original matching is a still a matching in the edited graph. Deleting vertices adjacent to an edge in the matching means that at most one edge in the matching per deleted vertex is deleted. For each edge in the matching with one of its two endpoints deleted, at most one additional edge (an edge adjacent to its other endpoint) needs to be added to maintain the maximal matching. Thus, the size of the maximal matching does not increase and MMM is hereditary.

CRN is trivially hereditary because deleting vertices can only decrease the number of colors necessary to color the graph. □

Lemma 3.5.9. *The problems Vertex Cover, Feedback Vertex Set, Minimum Maximal Matching, and Chromatic Number can be structurally lifted with respect to vertex deletion with constant $c = 1$.*

Proof. Let G be a graph, and $G' = G \setminus X$ where $X \subseteq V(G)$. Let S' be a solution to optimization problem Π on G'. We will show that $S \subseteq S' \cup X$ is a valid solution to Π on G for each Π listed in the Lemma.

Given a solution S' to VC for the graph G', the only edges not covered by S' in G' are edges between X and G' and between two vertices in X. Both sets of such edges are covered by X. Thus, $S = S' \cup X$ is a valid cover for G.

Given a solution S' to FVS for the graph G', the only cycles not covered by S' in G' are cycles that include a vertex in X. Thus, $S = S' \cup X$ is a valid feedback vertex set for G since X covers all newly introduced cycles in G.

Given a solution S' to MMM for the graph G', the only edges not in the matching and not adjacent to edges in the matching are edges between X and G' and edges between two vertices in X. Thus, any additional edges added to the maximal matching will come from X, and $S \subseteq S' \cup X$ (by picking edges to add to the maximal matching greedily for example) is a valid solution.

Given a solution S' to CRN for the graph G', the only vertices that could violate the coloring of the graph G' are vertices in X. Making each vertex in X a different color from each other as well as the colors in G' creates a valid coloring of G. Thus, $S = S' \cup X$ is a valid coloring. □

Corollary 3.5.10. *The problems Vertex Cover, and Feedback Vertex Set have $(1 + O(\delta \log^{1.5} n))$-approximations for graphs $(\delta \cdot \mathrm{OPT}(G))$-close to treewidth w via vertex deletions where $w\sqrt{log w} = O(\log n)$; and $(1 + c_H\delta)$-approximations for graphs $(\delta \cdot \mathrm{OPT}(G))$-close to planar-$H$-minor-free via vertex deletions.*

Proof. We apply Theorem 3.5.4 using stability with constant $c' = 0$ (Lemma 3.5.8) and structural lifting with constant $c = 1$ (Lemma 3.5.9).

For treewidth w, we use our $(O(\log^{1.5} n), O(\sqrt{\log w}))$-approximate editing algorithm (Section 3.6.5) and an exact polynomial-time algorithm for the problem of interest [Bod88, AN01, CNP+11] given the tree-decomposition of width $O(w\sqrt{\log w})$ of the edited graph. Thus $\alpha = O(\log^{1.5} n)$ and $c = 1$, resulting in an approximation factor of $1 + O(\log^{1.5} n)\delta$. Note that since the edited graph has treewidth $O(w\sqrt{\log w}) = O(\log n)$, the exact algorithm runs in polynomial-time. For planar-H-minor-free graphs, we use Fomin's c_H-approximate editing algorithm [FLMS12] and the same exact algorithm for bounded treewidth (as any planar-H-minor-free graph has bounded treewidth [CC16]). Thus $\alpha = c_H$ and $c = 1$, resulting in an approximation factor of $1 + c_H\delta$. □

Corollary 3.5.11. *The problems Minimum Maximal Matching, and Chromatic Number have $(1 + O(\delta \log^{1.5} n))$-approximations for graphs $(\delta \cdot \mathrm{OPT}(G))$-close to treewidth w via vertex deletions where $w \log^{1.5} w = O(\log n)$; and $(1 + c_H\delta)$-approximations for graphs $(\delta \cdot \mathrm{OPT}(G))$-close to planar-$H$-minor-free via vertex deletions.*

Proof. We apply Theorem 3.5.4 using stability with constant $c' = 0$ (Lemma 3.5.8) and structural lifting with constant $c = 1$ (Lemma 3.5.9).

For treewidth w, we use our $(O(\log^{1.5} n), O(\sqrt{\log w}))$-approximate editing algorithm (Section 3.6.5) and an exact algorithm for the problem of interest [Bod88] given a tree-decomposition of width $O(w\sqrt{\log w})$ of the edited graph. Thus $\alpha = O(\log^{1.5} n)$ and $c = 1$, resulting in an approximation factor of $1 + O(\log^{1.5} n)\delta$. Note that since the edited graph has treewidth $O(w \log^{1.5} w) = O(\log n)$, the exact algorithm runs in polynomial-time. For planar-H-minor-free graphs, we use Fomin's c_H-approximate editing algorithm [FLMS12] and the same exact algorithm for bounded treewidth (as any planar-H-minor-free graph has bounded treewidth [CC16]). Thus $\alpha = c_H$ and $c = 1$, resulting in an approximation factor of $1 + c_H\delta$. □

3.5.3 Edge Deletions

Theorem 3.5.12. *For graphs $(\delta \cdot \mathrm{OPT}(G))$-close to degeneracy k via edge deletions:*

- *Independent Set has a $(1/(3k+1) - 3\delta)$-approximation.*
- *Dominating Set has an $O((1+\delta)k)$-approximation.*

For graphs $(\delta \cdot \mathrm{OPT}(G))$-close to treewidth w via edge deletions:

- *(ℓ-)Dominating Set and Edge (ℓ-)Dominating Set have $(1 + O(\delta \log n \log\log n))$-approximations when $w \log w = O(\log_\ell n)$.*
- *Max-Cut has a $(1 - O(\delta \log n \log\log n))$-approximation when $w \log w = O(\log n)$.*

We now consider the edit operation of edge deletion. For each problem, we show stability and structural liftability, and use these to conclude approximation algorithms.

Lemma 3.5.13. *For $\ell \geq 1$, (ℓ-)Independent Set is stable under edge deletion with constant $c' = 0$.*

Proof. Given G and any set $X \subseteq E(G)$ with $|X| \leq \gamma$, let $G' = G[E \setminus X]$. For any (ℓ-)independent set $Y \subseteq V(G)$, $Y' = Y$ is also an (ℓ-)independent set in G'. Then $\mathrm{OPT}(G') \geq |Y'| = |Y|$, and so for optimal Y, $\mathrm{OPT}(G') \geq \mathrm{OPT}(G)$. □

Lemma 3.5.14. *For $\ell \geq 1$, (ℓ-)Independent Set can be structurally lifted with respect to edge deletion with constant $c = 1$.*

Proof. Given a graph G and $X \subseteq E(G)$, let $G' = G[E \setminus X]$. Let $Y' \subseteq V(G')$ be an (ℓ-)independent set in G', and consider the same vertex set Y' in G. Assume that the edit set is a single edge, $X = \{(u, v)\}$. We claim there exists a subset of Y' with size at least $|Y'| - 1$ which is still an (ℓ-)independent set in G.

For convenience, we let $d(\cdot,\cdot) := d_G(\cdot,\cdot)$ for the remainder of this proof. Suppose there are four distinct nodes $a, b, f, g \in Y'$ such that $d(a,b) \leq \ell$ and $d(f,g) \leq \ell$ in G. Since these nodes are in Y', we know $d_{G'}(a,b), d_{G'}(f,g) \geq \ell + 1$, hence, any shortest path from a to b in G must use the edge (u,v) in order to have length $\leq \ell$. WLOG we can assume the a-b path goes from a to u to v to b, and so $d(a,u) + 1 + d(v,b) \leq \ell$. Similarly we can assume the f-g path goes from f to v to u to g, and so $d(f,v) + 1 + d(u,g) \leq \ell$. We now argue that the shortest paths in G from a to u, v to b, f to v, and u to g do not use the edge (u,v) and are therefore also paths in G'. Suppose not and consider WLOG the case when a shortest path from a to u contains (u,v). Then concatenating the subpath from a to v with a shortest path from v to b gives an a,b-path of length $d(a,u) - 1 + d(v,b) < \ell$, which

does not use the edge (u, v) (and is thus a path in G', contradicting ($\ell-$)independence of Y').

Now consider the paths (a to u to g) and (f to v to b). Let $\ell_A = d(a,u) + d(u,g)$ and $\ell_F = d(f,v) + d(v,b)$, and note that $\ell_A + \ell_F = d(a,u) + d(v,b) + d(f,v) + d(u,g)$, which is $\leq 2\ell - 2$. So at least one of ℓ_A or ℓ_F must be $\leq \ell - 1$, a contradiction.

Three cases remain: (1) Y' contains exactly two vertices connected by a path of length $\leq \ell$ in G; (2) Y' contains three distinct vertices pair-wise connected by paths of length $\leq \ell$ in G; or (3) Y' contains one vertex, a, connected to two or more other vertices of Y' by paths of length $\leq \ell$ in G. In the first case, Y' contains a, b with $d(a,b) \leq \ell$; then removing either endpoint from Y' yields an ($\ell-$)independent set of size $|Y'| - 1$ in G.

We now show the second case cannot occur. Suppose that $d(b,c), d(a,b), d(a,c) \leq \ell$ for $a, b, c \in Y'$. Note that each vertex is within distance $\ell/2$ of at least one of the vertices u or v. By the pigeonhole principle, some two of a, b, c must be within $\ell/2$ of the same endpoint of (u, v); say vertices a and b are within $\ell/2$ of u WLOG; this implies $d_{G'}(a,b) \leq \ell$, a contradiction.

Finally, in the third case, Y' contains a node a and a subset S so that $|S| \geq 2$, $d(a,s) \leq \ell$ for all $s \in S$ and $d(s_1, s_2) > \ell$ for all $s_1 \neq s_2$ in S. Further, we know no other pair of nodes in Y' is at distance at most ℓ in G (since then we would have two disjoint pairs at distance at most ℓ, a case we already handled). In this setting, $Y' \setminus \{a\}$ is an ($\ell-$)independent set of size $|Y'| - 1$ in G. This proves that adding a single edge to G' will reduce the size of the (ℓ-)independent set Y' by no more than one, so by induction the lemma holds. □

Corollary 3.5.15. *Independent Set has a $(1/(3k+1) - 3\delta)$-approximation for graphs $(\delta \cdot \mathrm{OPT}(G))$-close to degeneracy k via edge deletions.*

Proof. We apply Theorem 3.5.4 using stability with constant $c' = 0$ (Lemma 3.5.5) and structural lifting with constant $c = 1$ (Lemma 3.5.6). We use our $(3, 3)$-approximate editing algorithm (Corollary 3.6.16) and the $1/(k+1)$-approximation algorithm for independent set described in in the proof of Corollary 3.5.7. Thus $\alpha = \beta = 3$ and $\rho(\beta k) = 1/(\beta k + 1)$, resulting in an approximation factor of $1/(3k+1) - 3\delta$. □

Note that Corollary 3.5.15 only applies to IS and not ℓ-IS.

Lemma 3.5.16. *The problems (ℓ-)Dominating Set and Edge (ℓ-)Dominating Set are stable under edge deletion with constant $c' = 1$.*

Proof. Given G and any set $X \subseteq E(G)$ with $|X| \leq \gamma$, let $G' = G[E \setminus X]$, and let Y be a minimum (ℓ-)dominating set on G. Each vertex v may be (ℓ-)dominated by multiple vertices on multiple paths, which we refer to as v's *dominating paths*.

Consider all vertices for which a specific edge (u, v) is on all of their dominating paths in G. We refer to each of these vertices as (u, v)-dependent. Note that if we traverse all dominating paths from each (u, v)-dependent vertex, (u, v) is traversed in the same direction each time. Assume WLOG (u, v) is traversed with u before v, implying u is not (u, v)-dependent but v may be. Now if (u, v) is deleted, then $Y \cup \{v\}$ is a (ℓ-)dominating set on the new graph. Therefore for each edge (u, v) in X we must add at most one vertex to the (ℓ-)dominating set. Thus if Y' is a minimum (ℓ-)dominating set on G' then $|Y'| \leq |Y| + \gamma$ and DS is stable under edge deletion with constant $c' = 1$.

Now let Z be a minimum edge (ℓ-)dominating set on G. The proof for EDS follows similarly as in the above case when a deleted edge (u, v) is not in Z (though an edge incident to v would be picked to become part of the dominating set instead of v itself). However if (u, v) is in the minimum edge (ℓ-)dominating set then it is possible that there are edges which are strictly (u, v)-dependent through only u or v and no single edge is within distance ℓ of both. In this case we add an edge adjacent to u and an edge adjacent to v to Z, which also increases Z's size by one with the deletion of (u, v). Thus if Z' is a minimum edge (ℓ-)dominating set on G' then $|Z'| \leq |Z| + \gamma$ and Edge (ℓ-)Dominating Set is stable under edge deletion with constant $c' = 1$. □

Lemma 3.5.17. *(ℓ-)Dominating Set and Edge (ℓ-)Dominating Set can be structurally lifted with respect to edge deletion with constants $c = 0$ and $c = 1$ respectively.*

Proof. Given G and any set $X \subseteq E(G)$ with $|X| \leq \gamma$, let $G' = G[E \setminus X]$. A (ℓ-)dominating set in G' is also a (ℓ-)dominating set in G. Therefore, a solution S' in G' yields a solution S in G such that $\text{Cost}_{\text{DS}}(S') = \text{Cost}_{\text{DS}}(S)$.

An edge (ℓ-)dominating set Y' in G' may not be an edge (ℓ-)dominating set in G, as there may be edges in X which are not (ℓ-)dominated by Y'. However $Y' \cup X$ is an edge (ℓ-)dominating set in G and $|Y' \cup X| \leq |Y'| + |X|$. □

Corollary 3.5.18. *Dominating Set has an $O((1 + \delta)k)$-approximation for graphs $(\delta \cdot \text{OPT}(G))$-close to degeneracy k via edge deletions.*

Proof. We apply Theorem 3.5.4 using stability with constant $c' = 1$ (Lemma 3.5.16) and structural lifting with constant $c = 0$ (Lemma 3.5.17). We use our $(3, 3)$-approximate editing algorithm (Section 3.6.3) and a known $O(k^2)$-approximation algorithm for DS [LW10]. Thus $\alpha = \beta = 3$ and $\rho(\beta k) = \beta^2 k^2$, resulting in an approximation factor of $9(1+3\delta)k^2$. □

Corollary 3.5.19. *(ℓ-)Dominating Set and Edge (ℓ-)Dominating Set have $(1 + O(\delta \log n \log\log n))$-approximations for graphs $(\delta \cdot \text{OPT}(G))$-close to treewidth w via edge deletions where $w \log w = O(\log_\ell n)$.*

Proof. We apply Theorem 3.5.4 using stability with constant $c' = 1$ (Lemma 3.5.16) and structural lifting with constant $c = 0$ for DS and constant $c = 1$ for EDS (Lemma 3.5.17). For treewidth w, we use the $(O(\log n \log\log n), O(\log w))$-approximate editing algorithm of Bansal et al. [BRU17] and an exact algorithm for DS and EDS [BL17] given a tree-decomposition of width $O(w \log w)$ of the edited graph.

Thus $\alpha = O(\log n \log\log n)$ and $c' = 1$ for DS and $c' = c = 1$ for EDS, resulting in an approximation factor of $1 + O(\log n \log\log n)\delta$. Note that since the edited graph has treewidth $O(w \log w) = O(\log_\ell n)$, the exact algorithm runs in polynomial-time. □

Lemma 3.5.20. *The problem Max-Cut is stable under edge deletion with constant $c' = 1$.*

Proof. Given G and any set $X \subseteq E(G)$ with $|X| \leq \gamma$, let $G' = G[E \setminus X]$, and let Y be a maximum cut in G. Then, $Y' := Y \setminus X$ is a cut in G' of size at least $|Y| - |X|$; hence, $c' = 1$. □

Lemma 3.5.21. *Max-Cut can be structurally lifted with respect to edge deletion with constant $c = 0$.*

Proof. Given G and any set $X \subseteq E(G)$ with $|X| \leq \gamma$, let $G' = G[E \setminus X]$. A cut $Y \subseteq E(G')$ is trivially a valid cut in G and consequently $c = 0$. □

Corollary 3.5.22. *Max-Cut has $(1 - O(\delta \log n \log\log n))$-approximations for graphs $(\delta \cdot \mathrm{OPT}(G))$-close to treewidth w via edge deletions where $w \log w = O(\log n)$.*

Proof. We apply Theorem 3.5.4 using stability with constant $c' = 1$ (Lemma 3.5.20) and structural lifting with constant $c = 0$ for MC (Lemma 3.5.21). For treewidth w, we use the $(O(\log n \log\log n), O(\log w))$-approximate editing algorithm of Bansal et al. [BRU17] and an exact algorithm for MC given a tree-decomposition of width $O(w \log w)$ of the edited graph.

Thus $\alpha = O(\log n \log\log n)$ and $c' = 1$ for MC, resulting in an approximation factor of $1 + O(\log n \log\log n)\delta$. Note that since the edited graph has treewidth $O(w \log w) = O(\log n)$, the exact algorithm runs in polynomial-time. □

3.5.4 Vertex Deletion for Annotated Problems (Vertex* Deletion)

In this section, we show that several important variants of *annotated* Dominating Set (ADS) (which include their non-annotated variants as special cases) are closed under a relaxed version of vertex deletion, denoted by vertex* deletion, which is sufficient to apply the structural rounding framework. Given an instance of Annotated ℓ-Dominating Set with input graph $G = (V, E)$ and a subset of vertices B, the resulting ADS instance (G', B') after deleting the set $X \subset V$ is defined as follows: $G' = (V \setminus X, E[V \setminus X])$ and $B' = B \setminus N_\ell[X]$ where $N_\ell[X]$ denotes the set of all vertices at distance at most ℓ from X in G.

Lemma 3.5.23. *For $\ell \geq 1$, Annotated ℓ-Dominating Set is stable under vertex* deletion with $c' = 0$.*

Proof. Note that Annotated ℓ-Dominating Set with $B = V$ reduces to ℓ-Dominating Set and in particular ℓ-Dominating Set is stable under vertex* deletion with constant $c' = 0$.

Let (G', B') denote the ADS instance after performing vertex* deletion with edit set X; $G' = (V \setminus X, E[V \setminus X])$ and $B' = B \setminus N_\ell[X]$ where $N_\ell[X]$ denotes the set of all vertices at distance at most ℓ from X in G. Moreover, let $\mathrm{OPT}(G, B)$ denote an optimal solution of $\mathrm{ADS}(G, B)$. We show that $\mathrm{OPT}(G, B) \setminus X$ is a feasible solution of $\mathrm{ADS}(G', B')$. Since X ℓ-dominates $N_\ell[X]$, the set $B \setminus N_\ell[X]$ is ℓ-dominated by $\mathrm{OPT}(G, S) \setminus X$; hence, $\mathrm{OPT}(G, S) \setminus X$ is a feasible solution of $\mathrm{ADS}(G', S')$. Thus $|\mathrm{OPT}(G', S')| \leq |\mathrm{OPT}(G, S) \setminus X| \leq |\mathrm{OPT}(G, S)|$. □

Lemma 3.5.24. *For $\ell \geq 1$, Annotated ℓ-Dominating Set can be structurally lifted with respect to vertex* deletion with constant $c = 1$.*

Proof. Note that Annotated ℓ-Dominating Set with $B = V$ reduces to ℓ-Dominating Set and in particular ℓ-Dominating Set can be structurally lifted with respect to vertex* deletion with constant $c = 1$.

Let $(G', B') = ((V \setminus X, E[V \setminus X]), B \setminus N_\ell[X])$ denote the ADS instance after performing vertex* deletion with edit set X on $\mathrm{ADS}(G, B)$ and let $\mathrm{OPT}(G', B')$ denote an optimal solution of $\mathrm{ADS}(G', B')$ instance. Since the set X ℓ-dominates $N_\ell[X]$, $\mathrm{OPT}(G', B') \cup X$ ℓ-dominates $B' \cup N_\ell[X] = B$. Hence, $|\mathrm{OPT}(G, B)| \leq |\mathrm{OPT}(G', B')| + |X|$. □

Corollary 3.5.25. *Annotated Dominating Set has an $O(k+\delta)$-approximation for graphs $(\delta \cdot \mathrm{OPT}(G))$-close to degeneracy k via vertex deletion.*

Proof. We apply Theorem 3.5.4 using stability with constant $c' = 0$ (Lemma 3.5.23) and structural lifting with constant $c = 1$ (Lemma 3.5.24).

We use a $(O(1), O(1))$-approximate editing algorithm (Section 3.6.1/ 3.6.2) and $O(k)$-approximation algorithm for the problem of interest [BU17] in k-degenerate graphs. Note that although the algorithm of [BU17] is for Dominating Set, it can easily be modified to work for the annotated variant. Thus, $\alpha = O(1)$ and $c = 1$, resulting in an $O(k+\delta)$-approximation algorithm. □

Corollary 3.5.26. *Annotated Dominating Set has an $O(1 + O(\delta \log^{1.5} n))$-approximation for graphs $(\delta \cdot \mathrm{OPT}(G))$-close to treewidth w via vertex deletion where $w\sqrt{\log w} = O(\log_\ell n)$.*

Proof. We apply Theorem 3.5.4 using stability with constant $c' = 0$ (Lemma 3.5.23) and structural lifting with constant $c = 1$ (Lemma 3.5.24).

We use our $(O(\log^{1.5} n), O(\sqrt{\log w}))$-approximate editing algorithm (Section 3.6.5) and an exact polynomial-time algorithm for the problem of interest [BL17] given the tree-decomposition of width $O(w\sqrt{\log w})$ of the edited graph. Note that the algorithm of [BL17] is presented for ℓ-DS; however, by slightly modifying the dynamic programming approach it works for the annotated version as well. Thus $\alpha = O(\log^{1.5} n)$ and $c = 1$, resulting in an approximation factor of $(1 + O(\log^{1.5} n)\delta)$. Moreover, since the edited graph has treewidth $O(w\sqrt{\log w}) = O(\log n)$, the exact algorithm runs in polynomial-time. □

Smarter Vertex* Deletion. The idea of applying edit operations on annotated problems can also be used for non-annotated problems. More precisely, for several optimization problems that fail to satisfy the required conditions of the standard structural rounding under vertex deletion, we can still apply our structural rounding framework with a more careful choice of the subproblem that we need to solve on the edited graph. An exemplary problem in this category is Connected Dominating Set (CDS). Note that Connected Dominating Set is not stable under vertex deletion and the standard structural rounding framework fails to work for this problem. Besides the stability issue, it is also non-trivial how to handle the connectivity constraint under vertex or edge deletions. However, in what follows we show that if we instead solve a *slightly different problem* (i.e. annotated variant of Connected Dominating Set) on the edited graph, then we can guarantee an improved approximation factor for CDS on the graphs close to a structural class.

Let $G = (V, E)$ be an input graph that is $(\delta \cdot \mathrm{OPT}(G))$-close to the class $\mathcal{C}$ and let $X \subset V$ be a set of vertices so that $G \setminus X \in \mathcal{C}$. For a subset of vertices X, $N_G(X)$ is defined to be the set of all neighbors of X excluding the set X itself; $N_G(X) := \{u \mid uv \in E(G), v \in X \text{ and } u \notin X\}$[8]. Let $G' = G[V \setminus X]$ be the resulting graph after removing the edit set X. The problem that we have to solve on G' is an *annotated* variant of CDS which is defined as follows:

[8]We drop the G in N_G when it is clear from the context.

Annotated Connected Dominating Set

Input: An undirected graph $G = (V, E)$, a subset of vertices $B \subset V$ and ℓ vertex-disjoint cliques $K_1 = (V_1, E_1), \cdots, K_\ell = (V_\ell, E_\ell)$ where for each i, $V_i \subset V$.

Problem: Find a minimum size set of vertices $S \subseteq V$ s.t. S dominates all vertices in B and S induces a connected subgraph in $G \cup (\bigcup_{i \in [\ell]} K_i)$.

To specify the instance of Annotated Connected Dominating Set that we need to solve on the edited graph G', we construct an auxiliary graph $\bar{G} = (N_G(X), \bar{E})$ as follows: $uv \in \bar{E}$ if there exists a uv-path in G whose intermediate vertices are all in X.

First, we show that CDS is stable under vertex* deletion with constant $c' = 0$: the size of an optimal solution of ACDS$(G', B', K_1, \cdots, K_\ell)$ is not more than the size of an optimal solution of CDS(G) where $\{K_1, \cdots, K_\ell\}$ are the connected components of $\bar{G}$. Note that due to the transitivity of connectivity for each $i \in [\ell]$, K_i is a clique.

Lemma 3.5.27. *Connected Dominating Set is stable under vertex* deletion with $c' = 0$.*

Proof. Let OPT be an optimal solution of CDS(G). Here, we show that OPT $\setminus X$ is a feasible solution of ACDS$(G' = G[V \setminus X], B' = V \setminus N_G(X), K_1, \cdots, K_\ell)$ where $K_1, \cdots, K_\ell$ are connected the components of $\bar{G}$ as constructed above. This in particular implies that

$$\text{OPT}(G', B', K_1, \cdots, K_\ell) \leq |\text{OPT} \setminus X| \leq |\text{OPT}| = \text{OPT}(G).$$

Since OPT dominates V, it is straightforward to verify that OPT $\setminus X$ dominates B' in G'. Next, we show that OPT $\setminus X$ is connected in G' when for each i, all edges between the vertices of K_i are added to G'. Suppose that there exists a pair of vertices $u, v \in$ OPT $\setminus X$ that are not connected in G'. However, since OPT is connected, there exists a uv-path P_{uv} in OPT. If P_{uv} does not contain any vertices in X, then P_{uv} is contained in OPT $\setminus X$ as well and it is a contradiction. Now consider all occurrences of the vertices of X in P_{uv}. We show that each of them can be replaced by an edge in one of the K_is: for each subpath $v_0, x_1, \cdots, x_q, v_1$ of P_{uv} where $x_i \in X$ for all $i \in [q]$ and $v_0, v_1 \in N_G(X)$, $v_0 v_1$ belongs to the same connected component of $\bar{G}$. Hence, given P_{uv}, we can construct a path P'_{uv} in $G' \cup (\bigcup_{i \in [\ell]} K_i)$. Thus, OPT $\setminus X$ is a feasible solution of ACDS$(G', B', K_1, \cdots, K_\ell)$. □

Next, we show that a solution of the Annotated Connected Dominating Set instance we solve on the edited graph can be structurally lifted to a solution for Connected Dominating Set on the original graph with constant $c = 3$.

Lemma 3.5.28. *Connected Dominating Set can be structurally lifted under vertex* deletion with constant $c = 3$.*

Proof. Let OPT be an optimal solution of ACDS$(G' = G[V \setminus X], B' = V \setminus N_G(X), K_1, \cdots, K_\ell)$ where $K_1, \cdots, K_\ell$ are the connected components of $\bar{G}$ as constructed above. Here, we show that OPT $\cup X \cup Y$ is a feasible solution of CDS(G) where Y is a subset of $V \setminus X$ such that $|Y| \leq 2|X|$. First, it is easy to see that since X dominates $N_G(X) \cup X$ in G, OPT $\cup X$ is a dominating set of G. Next, we show that in polynomial time we can

find a subset of vertices Y of size at most $2|X|$ such that $\mathrm{OPT} \cup X \cup Y$ is a connected dominating set in G.

Note that if the subgraph induced by the vertex set OPT on $G' \cup (\bigcup_{i \in [\ell]} K_i)$ contains an edge uv which is not in $E(G')$, the edge can be replaced by a uv-path in G whose intermediate vertices are all in X. Hence, we can replace all such edges in OPT by including a subset of vertices $X' \subseteq X$ and the set $\mathrm{OPT} \cup X'$ remains connected in G. At this point, if $X = X'$, we are done: $\mathrm{OPT} \cup X$ is a connected dominating set in G. Suppose this is not the case and let $X_1 := X \setminus X'$ and $Y_1 := N_G(X_1) \setminus N_G(X')$. Since G is connected, there exists a path from X_1 to $\mathrm{OPT} \cup X'$. Moreover, we claim that there exists a path of length at most 4 from X_1 to $\mathrm{OPT} \cup X'$. Recall that $\mathrm{OPT} \cup X'$ dominates $V \setminus (X_1 \cup Y_1)$. Hence, the shortest path from of X_1 to $\mathrm{OPT} \cup X'$ has length at most 4. We add the vertices on the shortest path which are in $X \setminus X'$ to X' and the vertices in $V \setminus (X \cup \mathrm{OPT} \cup Y)$ to Y, and update the sets X_1 and Y_1 accordingly. Thus we reduce the size of X_1 and as we repeat this process it eventually becomes zero. At this point $X = X'$ and $\mathrm{OPT} \cup X \cup Y$ is a connected dominating set in G. Since, we pick up at most three vertices per each $x \in X_1$ and at least one is in X, the set $X \cup Y$ has size at most $3|X|$. □

Corollary 3.5.29. *Connected Dominating Set has $O(1 + O(\delta \log^{1.5} n))$-approximation for graphs $(\delta \cdot \mathrm{OPT}(G))$-close to treewidth w via vertex deletion where w is a fixed constant.*

Proof. We apply Theorem 3.5.4 using stability with constant $c' = 0$ (Lemma 3.5.27) and structural lifting with constant $c = 3$ (Lemma 3.5.28).

We use our $(O(\log^{1.5} n), O(\sqrt{\log w}))$-approximate editing algorithm (Section 3.6.5) and an exact polynomial-time algorithm for ACDS given the tree-decomposition of width $O(w\sqrt{\log w})$ of the edited graph. The FPT algorithm modifies the $w^{O(w)} \cdot n^{O(1)}$ dynamic-programming approach of DS such that it incorporates the annotated sets and cliques $K_1, \cdots, K_\ell$ which then runs in $(w+\ell)^{O(w)} \cdot n^{O(1)} = n^{O(w)}$. Thus $w = O(1)$, $\alpha = O(\log^{1.5} n)$ and $c = 3$, resulting in an algorithm that runs in polynomial time and constructs a $(1 + O(\log^{1.5} n)\delta)$-approximate solution. □

Although we do not present any editing algorithms for edge contractions, we point out that such an editing algorithm would enable our framework to apply to additional problems such as (Weighted) TSP Tour (which is closed under edge contractions and can be structurally lifted with constant $c = 2$ [DHK11]), and to apply more efficiently to other problems such as Dominating Set (reducing c' from 1 to 0).

3.6 Editing Algorithms

3.6.1 Degeneracy: Density-Based Bicriteria Approximation

In this section we prove the following:

Theorem 3.6.1. *k-DE-V has a $\left(\frac{4m-\beta kn}{m-kn}, \beta\right)$-approximation algorithm.*

Observe that this yields a $(4, 4)$-approximation when $\beta = 4$. The algorithm is defined in Algorithm 1, and the analysis is based on the local ratio theorem from Bar-Yehuda et al. [BYBFR04].

Analysis overview and the local ratio theorem

Fundamentally, the local ratio theorem [BYBFR04] is machinery for showing that "good enough" local choices accumulate into a global approximation bound. This bookkeeping is done by maintaining *weight vectors* that encode the choices made. The local ratio theorem applies to optimization problems of the following form: given a weight vector $w \in \mathbb{R}^n$ and a set of feasibility constraints $\mathcal{C}$, find a solution vector $x \in \mathbb{R}^n$ satisfying the constraints $\mathcal{C}$ and minimizing $w^T x$ (for maximization problems see [BYBFR04]). We say a solution x to such a problem is *α-approximate with respect to w* if $w^T x \leq \alpha \cdot \min_{z \in \mathcal{C}}(w^T z)$.

Theorem 3.6.2 (Local Ratio Theorem [BYBFR04])**.** *Let $\mathcal{C}$ be a set of feasibility constraints on vectors in $\mathbb{R}^n$. Let $w, w_1, w_2 \in \mathbb{R}^n$ be such that $w = w_1 + w_2$. Let $x \in \mathbb{R}^n$ be a feasible solution (with respect to $\mathcal{C}$) that is α-approximate with respect to w_1, and with respect to w_2. Then x is α-approximate with respect to w as well.*

In our case, an instance of (βk)-Degenerate Vertex Deletion (abbreviated (βk)-DE-V) is represented with (G, w, k, β), where G is the graph, w is a weight vector on the vertices (where w is the all-ones vector, $\overrightarrow{1}$, when G is unweighted), k is our target degeneracy, and β is a multiplicative error on the target degeneracy. Our bicriteria approximation algorithm will yield an edit set to a (βk)-degenerate graph, using at most $\alpha \cdot \mathrm{OPT}_{(\beta k)\text{-DE-V}}(G, w, k, \beta)$ edits. This (weighted) cost function is encoded as an input vector of vertex weights w, which is evaluated with an indicator function $\mathcal{I}_X$ on a feasible solution X, such that the objective is to minimize $w^T \mathcal{I}_X$. Note that while the local ratio theorem can allow all feasible solutions, we require *minimal* feasible solutions for stronger structural guarantees.

Algorithm 1 Approximation for k-Degenerate Vertex Deletion

```
1:  procedure LocalRatioRecursion(Graph G, weights w, target degeneracy k, error β)
2:      if V(G) = ∅ then
3:          return ∅.
4:      else if ∃ v ∈ V(G) where deg_G(v) ≤ βk then
5:          return LocalRatioRecursion(G \ {v}, w, k, β)
6:      else if ∃ v ∈ V(G) where w(v) = 0 then
7:          X ← LocalRatioRecursion(G \ {v}, w, k, β)
8:          if G \ X has degeneracy βk then
9:              return X.
10:         else
11:             return MinimalSolution(G, X ∪ {v}, k, β).
12:     else
13:         Let ε := min_{v∈V(G)} w(v)/deg_G(v).
14:         Define w_1(u) := ε · deg_G(u) for all u ∈ V.
15:         Define w_2 := w − w_1.
16:         return LocalRatioRecursion(G, w_2, k, β).
```

To utilize the local ratio theorem, our strategy is to define a recursive function that decomposes the weight vector into $w = w_1 + w_2$ and then recurses on (G, w_2, k, β). By showing that the choices made in this recursive function lead to an (α, β)-approximation for the instances (G, w_1, k, β) and (G, w_2, k, β), by the local ratio theorem, these choices also sum to an (α, β)-approximation for (G, w, k, β).

As outlined in [BYBFR04, Section 5.2], the standard algorithm template for this recursive method handles the following cases: if a zero-cost minimal solution can be found, output this optimal solution, else if the problem contains a zero-cost element, do a problem size reduction, and otherwise do a weight decomposition.

Algorithm 2 Subroutine for guaranteeing minimal solutions

```
1: procedure MinimalSolution(Graph G, edit set X, target degeneracy k, error β)
2:     for vertex v ∈ V(G) do
3:         if G \ (X \ {v}) has degeneracy βk then
4:             return MinimalSolution(G, X \ {v}, k, β).
5:     return X.
```

Algorithm 1 follows this structure: Lines 2-3 are the first case, Lines 4-12 are the second case, and Lines 13-18 are the third case. The first two cases are typically straightforward, and the crucial step is the weight decomposition of $w = w_1 + w_2$. Note that the first case guarantees that all vertices in G have degree at least $\beta k+1$ before a weight decomposition is executed, so we may assume WLOG that the original input graph also has minimum degree $\beta k + 1$.

In the following subsections we show that Algorithm 1 returns a minimal, feasible solution (Lemma 3.6.3), that the algorithm returns an (α, β)-approximate solution with respect to w_1 (Theorem 3.6.4), and finally that the algorithm returns an (α, β)-approximate solution with respect to w (Theorem 3.6.1).

$\left(\frac{4m-\beta kn}{m-kn}, \beta\right)$-approximation for vertex deletion

Lemma 3.6.3. *Algorithm 1 returns minimal, feasible solutions for (βk)-DE-V.*

Proof. We proceed by induction on the number of recursive calls. In the base case, only Lines 2-3 will execute, and the empty set is trivially a minimal, feasible solution. In the inductive step, we show feasibility by constructing the degeneracy ordering. We consider each of the three branching cases not covered by the base case:

- Lines 4-5: Given an instance (G, w, k, β), if a vertex v has degree at most βk, add v to the degeneracy ordering and remove it from the graph. By the induction hypothesis, the algorithm will return a minimal, feasible solution $X_{\beta k}$ for $(G - \{v\}, w, k, \beta)$. By definition, v has at most βk neighbors later in the ordering (e.g. neighbors in $G-\{v\}$), so the returned $X_{\beta k}$ is still a feasible, minimal solution.
- Lines 6-12: Given an instance (G, w, k, β), if a vertex v has weight 0, remove v from the graph. By the induction hypothesis, the algorithm will return a minimal, feasible solution $X_{\beta k}$ for $(G - \{v\}, w, k, \beta)$. If $X_{\beta k}$ is a feasible solution on the instance

(G, w, k, β), then $X_{\beta k}$ will be returned as the minimal, feasible solution for this instance. Otherwise the solution $X_{\beta k} \cup \{v\}$ is feasible, and can be made minimal with a straightforward greedy subroutine (Algorithm 2).

- Lines 13-18: In this case, no modifications are made to the graph, therefore the recursive call's minimal, feasible solution $X_{\beta k}$ remains both minimal and feasible.

In all cases, a minimal, feasible solution is returned. □

We now show that a minimal, feasible solution is (α, β)-approximate with respect to the instance defined by weight function w_1:

Theorem 3.6.4. *Any minimal, feasible solution $X_{\beta k}$ is a $\left(\frac{4m-\beta rn}{m-kn}, \beta\right)$-approximation to the instance (G, w_1, k, β).*

Given a minimal, feasible solution $X_{\beta k}$, note that $w_1^T \mathcal{I}_{X_{\beta k}} = \varepsilon \sum_{v \in X_{\beta k}} \deg_G(v)$. Therefore it suffices to show that $b \leq \sum_{v \in X_k} \deg_G(v)$ and $\sum_{v \in X_{\beta k}} \deg_G(v) \leq \alpha b$, for some bound b, any minimal, feasible edit set X_k to degeneracy k, and any minimal, feasible edit set $X_{\beta k}$ to degeneracy βk. We prove these two bounds for $b = m - kn$ in Lemmas 3.6.5 and 3.6.8, respectively.

Lemma 3.6.5. *For any minimal feasible solution X_k for editing to degeneracy k,*

$$m - kn \leq \sum_{v \in X_k} \deg_G(v).$$

Proof. Since $G \setminus X_k$ has degeneracy k, it has at most kn edges, so at least $m - kn$ edges were deleted. Each deleted edge had at least one endpoint in X_k, therefore $m - kn \leq \sum_{v \in X_k} \deg_G(v)$. □

Before proving the upper bound, we define some notation. Let $X_{\beta k}$ be a minimal, feasible solution to (βk)-DE-V and let $Y = V(G) \setminus X_{\beta k}$ be the vertices in the (βk)-degenerate graph. Denote by m_X, m_Y, and m_{XY} the number of edges with both endpoints in $X_{\beta k}$, both endpoints in Y, and one endpoint in each set, respectively. We begin by bounding m_{XY}:

Lemma 3.6.6. *For any $X_{\beta k}$, it holds that $m_{XY} \leq 2m_Y + 2m_{XY} - \beta k|Y|$.*

Proof. Recall that we may assume WLOG that every vertex in G has degree at least $\beta k + 1$. Therefore $\beta k|Y| \leq \sum_{v \in Y} \deg_G(v) \leq 2m_Y + m_{XY}$, and so $m_{XY} \leq 2m_Y + 2m_{XY} - \beta k|Y|$. □

Corollary 3.6.7. *For any $X_{\beta k}$, it holds that $-\beta k|X_{\beta k}| \geq -2m_Y - 2m_{XY} + \beta k|Y|$.*

Proof. Because $X_{\beta k}$ is minimal, every vertex in $X_{\beta k}$ will induce a $(\beta k+1)$-core with vertices in Y if not removed. Therefore each such vertex has at least $(\beta k + 1)$-neighbors in Y, and $\beta k|X_{\beta k}| \leq m_{XY}$. Substituting into Lemma 3.6.6, we find that $-\beta k|X_{\beta k}| \geq -2m_Y - m_{XY} + \beta k|Y|$. □

We now prove the upper bound:

Lemma 3.6.8. *For any minimal, feasible solution $X_{\beta k}$ to (βk)-DE-V,*

$$\sum_{v \in X_{\beta k}} \deg_G(v) \leq 4m - \beta k n.$$

Proof. By using substitutions from Lemmas 3.6.6 and Corollary 3.6.7, we know that

$$\begin{aligned}
\sum_{v \in X_{\beta k}} \deg_G(v) &= 2m_X + m_{XY} \\
&\leq 2m_X + 2m_Y + 2m_{XY} - \beta k |Y| \\
&= 2m - \beta k |Y| \\
&= 2m + 2m_Y + 2m_{XY} - 2m_Y - 2m_{XY} + \beta k |Y| - 2\beta k |Y| \\
&\leq 2m + 2m_Y + 2m_{XY} - \beta k |X_{\beta k}| - 2\beta k |Y| \\
&\leq 4m - \beta k n.
\end{aligned}$$

□

Proof of Theorem 3.6.4. Let $X_{\beta k}$ be any minimal, feasible solution for editing to a graph of degeneracy βk. By definition of w_1 in Algorithm 1, it holds that $w_1^T \mathcal{I}_{X_{\beta k}} = \varepsilon \sum_{v \in X_{\beta k}} \deg_G(v)$, and because ε is a constant computed independently of the optimal solution, it suffices to show that $\sum_{v \in X_{\beta k}} \deg_G(v)$ has an α-approximation.

By Lemma 3.6.5, any minimal, feasible edit set to a degeneracy-r graph has a degree sum of at least $m - kn$. If an edit set is allowed to leave a degeneracy-(βk) graph, then by Lemma 3.6.8, at most $4m - \beta k n$ degrees are added to the degree sum of $X_{\beta k}$. Therefore $X_{\beta k}$ is $\left(\frac{4m - \beta k n}{m - kn}, \beta\right)$-approximate with respect to (G, w_1, k, β). □

We now prove the main result stated at the beginning of this section, Theorem 3.6.1.

Proof. For clarity, let $\alpha := \left(\frac{4m - \beta k n}{m - kn}\right)$; we prove that Algorithm 1 is an (α, β)-approximation. We proceed by induction on the number of recursive calls to Algorithm 1. In the base case (Lines 2-3), the solution returned is the empty set, which is trivially optimal. In the induction step, we examine the three recursive calls:

- Lines 4-5: Given an instance (G, w, k, β), if a vertex v has degree at most βk, add v to the degeneracy ordering and remove it from the graph. By the induction hypothesis, the algorithm will return an (α, β)-approximate solution $X_{\beta k}$ for $(G - \{v\}, w, k, \beta)$. Since v will not be added to $X_{\beta k}$, then $X_{\beta k}$ is also an (α, β)-approximation for (G, w, k, β).
- Line 6-12: Given an instance (G, w, k, β), if a vertex v has weight 0, remove v from the graph. By the induction hypothesis, the algorithm returns an (α, β)-approximate solution $X_{\beta k}$ for $(G - \{v\}, w, k, \beta)$. Regardless of whether v is added to $X_{\beta k}$ or not, it contributes exactly zero to the cost of the solution, therefore an (α, β)-approximation is returned.
- Line 13-18: In this case, the weight vector is decomposed into w_1 and $w_2 = w - w_1$. By induction, the algorithm will return an (α, β)-approximate solution $X_{\beta k}$ for

$(G, w - w_1, k, \beta)$. By Theorem 3.6.4, $w_1^T \mathcal{I}_{X_{\beta k}}$ is also (α, β)-approximate. Therefore, by Theorem 3.6.2, $w^T \mathcal{I}_{X_{\beta k}}$ must be (α, β)-approximate.

□

3.6.2 Degeneracy: LP-based Bicriteria Approximation

In this section, we design a bicriteria approximation for the problem of minimizing the number of required edits (edge/vertex deletions) to the family of k-degenerate graphs. Consider an instance of k-Degenerate Edge Deletion(G, k) and let OPT denote an optimal solution. The algorithm we describe here works even when the input graph is *weighted* (both vertices and edges are weighted) and the goal is to minimize the total weight of the edit set.

(6, 6)-approximation for vertex deletion

In what follows we formulate an LP-relaxation for the problem of minimizing the number of required vertex deletions to the family of k-degenerate graphs. For each edge $uv \in E$, x_{uv} variable denotes the orientation of uv; $x_{\overrightarrow{uv}} = 1$, $x_{\overrightarrow{vu}} = 0$ if uv is oriented from u to v and $x_{\overrightarrow{vu}} = 1$, $x_{\overrightarrow{uv}} = 0$ otherwise. Moreover, for each vertex $v \in V$ we define y_v to denote whether v is part of the edit set X ($y_v = 1$ if $v \in X$ and zero otherwise).

DegenVertexEdit-LP

Input: $G = (V, E), w, k$

$$\begin{aligned}
\text{Minimize} \quad & \sum_{v \in V} y_v w_v & \\
\text{s.t.} \quad & x_{\overrightarrow{vu}} + x_{\overrightarrow{uv}} \geq 1 - y_u - y_v & \forall uv \in E \\
& \sum_{u \in N(v)} x_{\overrightarrow{vu}} \leq k & \forall v \in V \\
& x_{\overrightarrow{uv}} \geq 0 & \forall uv \in E
\end{aligned}$$

The first set of constraints in the LP-relaxation DegenVertexEdit-LP guarantees that for each edge uv whose none of its endpoints is in X, it is oriented either from v to u or from u to v. The third set of the constraints ensure that for all $v \in V$, $\deg^+(v) \leq k$. Note that if $v \in X$ and thus $y_v = 1$, then WLOG we can assume that both $x_{\overrightarrow{uv}}$ and $x_{\overrightarrow{vu}}$ are set to zero.

Lemma 3.6.9. *DegenVertexEdit-LP(G, w, k) is a valid LP-relaxation of k-DE-V(G, w).*

Proof. Let X be a feasible edit set of k-DE-V(G, w). Let D be an k-degenerate ordering of $V \setminus X$. We define vectors (x, y) corresponding to X as follows: for each $v \in V$, $y(v) = 0$ if $v \in X$ and zero otherwise. Moreover, $x_{\overrightarrow{uv}} = 1$ if $u, v \in V \setminus X$ and u comes before v in the ordering D; otherwise, $x_{\overrightarrow{uv}}$ is set to zero.

Next, we show that the constructed solution (x, y) satisfies all constraints in DegenVertexEdit-LP. Since x only obtains non-negative values, for the first set of constraints we can only consider the set of survived edges after removing set X, $E[V \setminus X]$. For these edges, since one of u and v comes first in D, exactly one of $x_{\overrightarrow{uv}}, x_{\overrightarrow{vu}}$ is one and the constraint is satisfied. Lastly, since D is an k-degenerate ordering of $V \setminus X$, for each vertex $v \in V \setminus X$, the out-degree is at most k. Moreover, for each $v \in X$, the LHS in the second set of constraints is zero. □

Similarly to our approach for k-Degenerate Edge Deletion, first we find an optimal solution (x, y) of DegenVertexEdit-LP in polynomial time.

Rounding scheme. We prove that the following rounding scheme of DegenVertexEdit-LP gives a $(\frac{1}{\varepsilon}, \frac{4}{1-2\varepsilon})$-bicriteria approximation for k-DE-V.

$$\hat{y}_v = \begin{cases} 1 & \text{if } y_v \geq \varepsilon, \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

$$\hat{x}_{uv} = \begin{cases} 1 & \text{if } x_{uv} \geq (1-2\varepsilon)/2, \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

Lemma 3.6.10. *If (x, y) is an optimal solution to DegenVertexEdit-LP, then $(\hat{x}, \hat{y})$ as given by Eq. (3.1) and Eq. (3.2) is an integral $(\frac{1}{\varepsilon}, \frac{2}{1-2\varepsilon})$-bicriteria approximate solution of DegenVertexEdit-LP(G, w, r).*

Proof. First we show that $(\hat{x}, \hat{y})$ satisfies the first set of constraints: for each $uv \in E$, $\hat{x}_{\overrightarrow{vu}} + \hat{x}_{\overrightarrow{uv}} \geq 1 - \hat{y}_v - \hat{y}_u$. Note that if either $\hat{y}_v$ or $\hat{y}_u$ is one then the constraint trivially holds. Hence, we assume that both $\hat{y}_v$ and $\hat{y}_u$ are zero. By Eq. (3.1), this implies that both y_v and y_u have value less than ε. Hence, by feasibility of (x, y),

$$x_{\overrightarrow{vu}} + x_{\overrightarrow{uv}} \geq 1 - y_v - y_u \geq 1 - 2\varepsilon,$$

and in particular, $\max(x_{\overrightarrow{vu}}, x_{\overrightarrow{vu}}) \geq \frac{1-2\varepsilon}{2}$. Then, by Eq. (3.2), $\max(\hat{x}_{\overrightarrow{uv}}, \hat{x}_{\overrightarrow{vu}}) = 1$ and the constraint is satisfied: $\hat{x}_{uv} + \hat{x}_{vu} \geq 1 \geq 1 - \hat{y}_v - \hat{y}_u$. Note that if both of $\hat{x}_{uv}$ and $\hat{x}_{vu}$ are set to one, we can arbitrarily set one of them to zero.

Moreover, since for each arc $\overrightarrow{uv}$, $\hat{x}_{\overrightarrow{uv}} \leq \frac{2}{1-2\varepsilon} \cdot x_{\overrightarrow{uv}}$, for each $v \in V$:

$$\sum_{u \in N(v)} \hat{x}_{\overrightarrow{vu}} \leq \frac{2}{1-2\varepsilon} \cdot \sum_{u \in N(v)} x_{\overrightarrow{vu}} \leq \frac{2k}{1-2\varepsilon},$$

where the first inequality follows from Eq. (3.2) and the second from the feasibility of (x, z).

Finally, since for each $v \in V$, $\hat{y}_v \leq y_v/\varepsilon$, the cost of the rounded solution $(\hat{x}, \hat{y})$ is at

most

$$\sum_{u \in V} \hat{y}_v w_v \leq \frac{1}{\varepsilon} \cdot \sum_{u \in V} y_v w_v \leq \mathrm{OPT}_{k\text{-DE-V}}(G, w, k)/\varepsilon,$$

where the first inequality follows from Eq. (3.1) and the second directly from the optimality of (x, y). Hence, $(\hat{x}, \hat{y})$ is an integral $(\frac{1}{\varepsilon}, \frac{2}{1-2\varepsilon})$-bicriteria approximate solution of DegenVertexEdit-LP(G, w, k). □

Note that, the integral solution $(\hat{x}, \hat{y})$ specifies an edit set $X := \{v \in V | \hat{y}(v) = 1\}$ and orientation of edges $D := \{\overrightarrow{uv} | x_{\overrightarrow{uv}} = 1\}$ such that for each $v \in V \setminus X$, $\deg^+(v) \leq \frac{2r}{1-2\varepsilon}$. Hence, together with Lemma 2.1.5, we have the following result.

Corollary 3.6.11. *There exists a $(\frac{1}{\varepsilon}, \frac{4}{1-2\varepsilon})$-bicriteria approximation for k-DE-V.*

In particular, by setting $\varepsilon = 1/6$, there exists a $(6, 6)$-bicriteria approximation algorithm for the k-Degenerate Vertex Deletion problem.

3.6.3 $(5, 5)$-approximation for edge deletion

In what follows we formulate an LP-relaxation for the problem of minimizing the number of required edge edits (deletions) to the family of k-degenerate graphs. For each edge $uv \in E$, x variables denote the orientation of uv; $x_{\overrightarrow{uv}} = 1$, $x_{\overrightarrow{vu}} = 0$ if uv is oriented from u to v and $x_{\overrightarrow{vu}} = 1$, $x_{\overrightarrow{uv}} = 0$ if e is oriented from v to u. Moreover, for each uv we define z_{uv} to denote whether the edge uv is part of the edit set X ($z_{uv} = 1$ if the edge $uv \in X$ and zero otherwise).

DegenEdgeEdit-LP

Input: $G = (V, E), w, k$

$$\begin{aligned}
\text{Minimize} \quad & \sum_{uv \in E} z_{uv} w_{uv} & \\
\text{s.t.} \quad & x_{\overrightarrow{vu}} + x_{\overrightarrow{uv}} \geq 1 - z_{uv} & \forall uv \in E \\
& \sum_{u \in N(v)} x_{\overrightarrow{vu}} \leq k & \forall v \in V \\
& x_{\overrightarrow{uv}} \geq 0 & \forall uv \in V \times V
\end{aligned}$$

The first set of constraints in the LP-relaxation DegenEdgeEdit-LP guarantee that for each edge $uv \notin X$, it is oriented either from v to u or from u to v. The second set of the constraints ensure that for all $v \in V$, $\deg^+(v) \leq k$. Note that if an edge $uv \in X$ and thus $z_{uv} = 1$, then WLOG we can assume that both $x_{\overrightarrow{uv}}$ and $x_{\overrightarrow{vu}}$ are set to zero.

Lemma 3.6.12. *DegenEdgeEdit-LP(G, w, k) is a valid LP-relaxation of k-DE-E(G, w).*

Next, we propose a *two-phase* rounding scheme for the DegenEdgeEdit-LP.

First phase. Let (x, z) be an optimal solution of DegenEdgeEdit-LP. Note that since the DegenEdgeEdit-LP has polynomial size, we can find its optimal solution efficiently. Consider the following *semi-integral* solution $(x, \hat{z})$ of DegenEdgeEdit-LP:

$$\hat{z}_{uv} = \begin{cases} 1 & \text{if } z_{uv} \geq \varepsilon, \\ 0 & \text{otherwise.} \end{cases} \tag{3.3}$$

Claim 3.6.13. *$(\frac{x}{1-\varepsilon}, \hat{z})$ as given by Eq. (3.3) is a $(\frac{1}{\varepsilon}, \frac{1}{1-\varepsilon})$-bicriteria approximate solution of DegenEdgeEdit-LP(G, w, k).*

Proof. First, we show that $(\frac{1}{1-\varepsilon}x, \hat{z})$ satisfies the first set of constraints. For each edge uv,

$$\frac{x_{\overrightarrow{uv}}}{1-\varepsilon} + \frac{x_{\overrightarrow{vu}}}{1-\varepsilon} = \frac{1}{1-\varepsilon}(x_{\overrightarrow{uv}} + x_{\overrightarrow{vu}}) \geq \frac{1}{1-\varepsilon}(1 - z_{uv}) \geq 1 - \hat{z}_{uv},$$

where the first inequality follows from the feasibility of (x, z) and the second inequality follows from Eq. (3.3). Moreover, it is straightforward to check that as we multiply each x_{vu} by a factor of $1/(1-\varepsilon)$, the second set of constraints are off by the same factor; that is, $\forall v \in V, \sum_{u \in V} x_{\overrightarrow{vu}}/(1-\varepsilon) \leq k/(1-\varepsilon)$. Finally, since for each edge uv, $\hat{z}_{uv} \leq z_{uv}/\varepsilon$, the cost of the edit set increases by at most a factor of $1/\varepsilon$; that is, $\sum_{uv \in E} \hat{z}_{uv} w_{uv} \leq \frac{1}{\varepsilon} \sum_{uv \in E} z_{uv} w_{uv}$. □

Second phase. Next, we prune the fractional solution further to get an *integral* approximate *nearly feasible* solution of DegenEdgeEdit-LP. Let $\hat{x}$ denote the orientation of the surviving edges (edges uv such that $\hat{z}_{uv} = 0$) given by:

$$\hat{x}_{\overrightarrow{uv}} = \begin{cases} 1 & \text{if } x_{\overrightarrow{uv}} \geq (1-\varepsilon)/2, \\ 0 & \text{otherwise.} \end{cases} \tag{3.4}$$

We say an orientation is *valid* if each surviving edge (u, v) is oriented from u to v or v to u.

Lemma 3.6.14. *$\hat{x}$ as given by Eq. (3.4) is a valid orientation of the set of surviving edges.*

Proof. We need to show that for each $uv \in E$ with $\hat{z}_{uv} = 0$ at least one of $\hat{x}_{\overrightarrow{uv}}$ or $\hat{x}_{\overrightarrow{vu}}$ is one. Note that if both are one, we can arbitrarily set one of them to zero.

For an edge uv, by Eq. (3.3), $\hat{z}_{uv} = 0$ iff $z_{uv} \leq \varepsilon$. Then, using the fact that (x, z) is a feasible solution of DegenEdgeEdit-LP, $x_{\overrightarrow{uv}} + x_{\overrightarrow{vu}} \geq 1 - z_{uv} \geq 1 - \varepsilon$. Hence, $\max(x_{\overrightarrow{uv}}, x_{\overrightarrow{vu}}) \geq (1-\varepsilon)/2$ which implies that $\max(\hat{x}_{\overrightarrow{uv}}, \hat{x}_{\overrightarrow{vu}}) = 1$. Hence, for any surviving edge uv, at least one of $\hat{x}_{\overrightarrow{uv}}$ or $\hat{x}_{\overrightarrow{vu}}$ will be set to one. □

Lemma 3.6.15. *$(\hat{x}, \hat{z})$ as given by Eq. (3.3) and Eq. (3.4) is an integral $(\frac{1}{\varepsilon}, \frac{2}{1-\varepsilon})$-bicriteria approximate solution of DegenEdgeEdit-LP(G, w, r).*

Proof. As we showed in Lemma 3.6.14, $\hat{x}$ is a valid orientation of the surviving edges with respect to $\hat{z}$. Moreover, by Eq. (3.4), for each $uv \in E$, $\hat{x}_{\overrightarrow{uv}} \leq 2x_{\overrightarrow{uv}}/(1-\varepsilon)$. Hence, for each vertex $v \in V$, $\deg^+(v) \leq 2k/(1-\varepsilon)$. Finally, as we proved in Claim 3.6.13, the total weight

of the edit set defined by $\hat{z}$ is at most $\frac{1}{\varepsilon}$ times the total weight of the optimal solution (x, z). □

Hence, together with Lemma 2.1.5, we have the following result.

Corollary 3.6.16. *There exists a $(\frac{1}{\varepsilon}, \frac{4}{1-\varepsilon})$-bicriteria approximation algorithm for k-DE-E.*

In particular, by setting $\varepsilon = 1/5$, there exists a $(5, 5)$-bicriteria approximation algorithm for the k-Degenerate Edge Deletion problem.

We note that our approach also works in the general setting when both vertices and edges are weighted, and we consider an edit operation which includes both vertex and edge deletion.

Integrality gap of DegenEdgeEdit-LP and DegenVertexEdit-LP

A natural open question is if we can obtain "purely multiplicative" approximation guarantees for k-DE-E and k-DE-V via LP-based approaches. In this section, we show that the existing LP-relaxation of editing to bounded degeneracy cannot achieve $o(n)$-approximation. These results are particularly important because they show that the best we can hope for are bicriteria approximations.

Theorem 3.6.17. *The integrality gap of DegenEdgeEdit-LP is $\Omega(n)$.*

Proof. Consider an instance of k-DE-E(G) where G is an unweighted complete graph of size $2n$ and $k = n-2$. First, we show that DegenEdgeEdit-LP(G, r) admits a fractional solution of cost/size $O(n)$ and then we show that the size of any feasible edit set of k-DE-E(G) is $\Omega(n^2)$.

Consider the following fractional solution of DegenEdgeEdit-LP(G, r): for all $uv \in V \times V$ and $u \neq v$, $x_{\overrightarrow{uv}} = 1/2 - 1/n$ and for all edges $uv \in E$, $z_{uv} = 2/n$. Note that x and z satisfy the first set of constraints in DegenEdgeEdit-LP(G, k):

$$\forall uv \in E, \quad x_{\overrightarrow{uv}} + x_{\overrightarrow{vu}} = 1 - 2/n = 1 - z_{uv}.$$

Moreover, x satisfies the second set of the constraints in DegenEdgeEdit-LP(G, k)

$$\forall v \in V, \quad \sum_{u \in V} x_{\overrightarrow{vu}} = (2n-1)(1/2 - 1/n) < n - 2.$$

Finally, $\text{Cost}(x, z) = \sum_{uv \in E} z_{uv} = n(2n-1) \cdot (2/n) = 4n - 2$ which implies that the cost of an optimal solution of DegenEdgeEdit-LP(G, k) is $O(n)$.

Next, we show that any integral solution of k-DE-E(G, k) has size $\Omega(n^2)$. Let X be a solution of k-DE-E(G, k). Then, there exits an ordering of the vertices in G, $v_1, \cdots, v_{2n}$ such that $\deg(v_i)$ in $G[v_i, \ldots, v_{2n}]$ is at most $r \leq n - 2$. This implies that for $i \leq n - 2$, $|\delta(v_i) \cap X| \geq n + 2 - i$, where $\delta(v)$ denotes the set of edges incident to a vertex v. Thus,

$$|X| \geq \frac{1}{2} \sum_{i \leq n-2} |\delta(v_i) \cap X| \geq \frac{1}{2} \sum_{i \leq n-2} n + 2 - i \geq \frac{1}{2}(n^2 - 4 - \frac{(n-2)(n-3)}{2}) \geq n^2/4.$$

Hence, the integrality gap of DegenEdgeEdit-LP is $\Omega(n)$. □

Theorem 3.6.18. *The integrality gap of* DegenVertexEdit-LP *is* $\Omega(n)$.

Proof. Consider an instance of k-DE-V(G) where G is an unweighted complete graph of size $2n$ and $k = n-2$. First, we show that DegenVertexEdit-LP(G, r) admits a constant size fractional solution and then we show that the size of any feasible edit set of k-DE-E(G) is $\Omega(n)$.

Consider the following fractional solution of DegenVertexEdit-LP(G, r): for each $uv \in V^2$ and $u \neq v$, $x_{\overrightarrow{uv}} = 1/2 - 1/n$ and for each vertex $v \in V$, $z_v = 1/n$. First, we show that x and z satisfy the first set of constraints in DegenVertexEdit-LP(G, k):

$$\forall uv \in E, \quad x_{\overrightarrow{uv}} + x_{\overrightarrow{vu}} = 1 - 2/n = 1 - z_u - z_v.$$

Moreover, x satisfies the second set of the constraints in DegenVertexEdit-LP(G, k)

$$\forall v \in V, \quad \sum_{u \in V} x_{\overrightarrow{vu}} = (2n-1)(1/2 - 1/n) < n - 2.$$

Finally, $\text{Cost}(x, z) = \sum_{uv \in E} z_{uv} = 2n \cdot (1/n) = 2$ which implies that the cost of an optimal solution of DegenVertexEdit-LP(G, k) is at most 2.

Next, we show that any integral solution of k-DE-V(G, k) has size $\Omega(n)$. Let X be a solution of k-DE-E(G, k). Since $G \setminus X$ is a complete graph of size $2n - |X|$, in order to get degeneracy $n-2$, $|X| \geq n+2$.

Hence, the integrality gap of DegenVertexEdit-LP is $\Omega(n)$. □

3.6.4 Degeneracy: $O(\log n)$ Greedy Approximation

In this section, we give a polytime $O(\log n)$-approximation for reducing the degeneracy of a graph by one using either vertex deletions or edge deletions. More specifically, given a graph $G = (V, E)$ with degeneracy k, we produce an edit set X such that $G' = G \setminus X$ has degeneracy $k-1$ and $|X|$ is at most $O(\log |V|)$ times the size of an optimal edit set. Note that this complements an $O(\log \frac{n}{k})$-approximation hardness result for the same problem.

In general, the algorithm works by computing a vertex ordering and greedily choosing an edit to perform based on that ordering. In our algorithm, we use the *min-degree ordering* of a graph. The *min-degree ordering* is computed via the classic greedy algorithm given by Matula and Beck [MB83] that computes the degeneracy of the graph by repeatedly removing a minimum degree vertex from the graph. The degeneracy of G, $\text{degen}(G)$, is the maximum degree of a vertex when it is removed. In the following proofs, we make use of the observation that given a min-degree ordering L of the vertices in $G = (V, E)$ and assuming the edges are oriented from smaller to larger indices in L, $\deg^+(u) \leq \text{degen}(G)$[9] for any $u \in L$.

The first ordering L_0 is constructed by taking a min-degree ordering on the vertices of G where ties may be broken arbitrarily. Using L_0, an edit is greedily chosen to be added to X. Each subsequent ordering L_i is constructed by taking a min-degree ordering on the vertices of $G \setminus X$ where ties are broken based on L_{i-1}. Specifically, if the vertices u and v

[9] For notational reminders for $\deg^+(u)$, please refer to Section 2.1.

have equal degree at the time of removal in the process of computing L_i, then $L_i(u) < L_i(v)$ if and only if $L_{i-1}(u) < L_{i-1}(v)$. The algorithm terminates when the min-degree ordering L_j produces a witness that the degeneracy of $G \setminus X$ is $k-1$.

In order to determine which edit to make at step i, the algorithm first computes the forward degree of each vertex u based on the ordering L_i (equivalently, $\deg^+(u)$ when edges are oriented from smaller to larger index in L_i). Each vertex with forward degree k is marked, and similarly, each edge that has a marked left endpoint is also marked. The algorithm selects the edit that *resolves* the largest number of marked edges. We say that a marked edge is *resolved* if it will not be marked in the subsequent ordering L_{i+1}.

We observe that given an optimal edit set (of size k), removing the elements of the set in any order will resolve every marked edge after k rounds (assuming that at most one element from the optimal edit set is removed in each round). If it does not, then the final ordering L_k must have a vertex with forward degree k, a contradiction. Let m_i be the number of marked edges based on the ordering L_i. We show that we can always resolve at least $\frac{m_i}{k}$ marked edges in each round, giving our desired approximation.

Lemma 3.6.19. *A vertex that is unmarked in L_i cannot become marked in L_j for any $j > i$.*

Proof. For an unmarked vertex v to become marked, its forward degree must increase from $d \leq k-1$ to k when going from L_i to L_j for some $j > i$. In other words, $\deg^+{}_{L_i}(v) < k$ whereas $\deg^+{}_{L_j}(v) = k$. Since edges are not added to G, this can only occur if a backward neighbor u of v becomes a forward neighbor. Let $\{u, v\}$ be an *inversion* if $L_i(u) > L_i(v)$ but $L_j(u) < L_j(v)$[10]. An inversion can occur between neighbors u' and v', in which case u' and v' are connected by an edge. We call this a *positive inversion* for u and a *negative inversion* for v. If the number of positive inversions for u of u's neighbors is greater than the number of negative inversions of u's neighbors between L_i and L_j, then $\deg^+{}_{L_j}(u) > \deg^+{}_{L_i}(u)$.

By our previous observation, an unmarked vertex can only become a marked vertex through inversions. Let u be a vertex that was unmarked in L_i but becomes marked in L_j. Let u and v be the first positive inversion for u in L_i (i.e. there is not a w such that u and w form a positive inversion for u and $L_i(w) < L_i(v)$). Because the algorithm breaks ties when constructing L_j based on L_{j-1}, if u and v form a positive inversion for u and u becomes marked in L_j, then $\deg^+{}_{L_j}(u) < \deg_{L_j[i_u,n]}(v)$[11] and $\deg^+{}_{L_j}(u) = k$. (If, instead, $\deg^+{}_{L_j}(u) > \deg_{L_j[i_u,n]}(v)$, then v would have been removed first according to L_j.) Then, either (1) $\deg_{L_j[i_u,n]}(v) \leq \deg^+{}_{L_i}(v)$ or (2) $\deg_{L_j[i_u,n]}(v) > \deg^+{}_{L_i}(v)$.

If (1) occurs, then $\deg^+{}_{L_j}(u)$ cannot be k since this would imply $\deg^+{}_{L_i}(v) > k$, a contradiction. However, (2) can only occur through positive inversions of v (in fact, through positive inversions of v's neighbors) since we chose v to be the first positive inversion of u. (Hence, v cannot gain additional edges in the range $L_j[i_u, i_v]$ when going from L_i to L_j.) Let w be the first positive inversion of v in L_j. Given that v must have at least one positive inversion with one of its neighbors, w must exist (it can either be the neighbor of

[10]Note that u and v do *not* have to be connected by an edge.

[11]Let $\deg_{L_j[i_u,n]}(v)$ be the degree of v restricted to vertices between indices i_u and n in L_j. Here, i_u is the index of u.

v or another node.) The same case analysis applies to v and w, implying that w has a positive inversion and so on, eventually leading to a contradiction due to a lack of additional vertices to form a positive inversion. □

Using Lemma 3.6.19, we are able to prove a similar statement about marked edges.

Lemma 3.6.20. *An edge that is unmarked in L_i cannot become marked in L_j for any $j > i$.*

Proof. Suppose wlog that the edge $e = (u, v)$ is unmarked in L_i and that $L_i(u) < L_i(v)$. This implies that u is unmarked in L_i. By Lemma 3.6.19, u cannot become marked in L_j. Thus, in order for e to become marked, $L_j(v) < L_j(u)$ and $\deg^+_{L_j}(v) = k$. Since u is unmarked in L_j, we know that $\deg^+_{L_j}(u) \leq k-1$. So for $L_j(v) < L_j(u)$ where u and v are an inversion, the forward degree of v including u must be less than $k-1$. Thus, v must be unmarked in L_j, and so e is also unmarked. □

Lemmas 3.6.19 and 3.6.20 allow us to make a claim about the number of marked edges that any one edit resolves.

Lemma 3.6.21. *For a given edit x, the number of marked edges that it resolves is monotonically non-increasing from L_0. In other words, for any $i < j$, the number of marked edges x resolves in L_i is at least as many as the number of marked edges x resolves in L_j.*

Proof. By Lemma 3.6.20, we know that an unmarked edge cannot become marked. Thus, for the number of marked edges that an edit resolves in L_i to increase in L_{i+1}, an existing marked edge must become resolvable by making a different edit. Note that edits resolve edges by either deleting them or reducing the forward degree of marked vertices. Since the back neighbors of an edit x cannot become marked by Lemma 3.6.19 and any vertices that form a negative inversion with x via another edit must have forward degree at most $k-1$, it is not possible for x to gain marked edges that are resolvable by deletion.

Instead, any new resolvable marked edges must be resolved by the deletion of x reducing the degree of a back neighbor by one. Note that changes to the set of resolvable edges can only occur if the relative ordering of the neighbors of x changes. First, we will consider the case where a forward neighbor v forms an inversion with neighbor x. After the inversion, the forward degree of v will be one less than the original forward degree of x. Furthermore, we note that v can now only form an inversion with its last back neighbor b assuming that the forward degree of b is exactly one greater than the forward degree of v. Thus, the forward edges of b are resolvable by x if they are marked. Note that there are exactly k of these edges if this is the case. However, this implies that the forward degree of v is $k-1$, and so the original forward degree of x must have been k. These k edges must have been resolved by the inversion of v and x, so x resolves at most the same number of edges as it did originally. Note that if multiple forward neighbors form inversions with x simultaneously, only one of them can have forward degree $k-1$.

Next, we consider the case where a back neighbor b of x forms an inversion with some other neighboring vertex v (potentially of no relation to x). Again, we rely on the fact that b must have forward degree $k-1$ after the inversion in order to be able to make a second inversion that resolves additional marked edges. However, this implies that v originally had forward degree k, and so when b inverted with v, k marked edges were resolved.

Since b can only form an inversion with one of its back neighbors per edit, there are at most k new marked edges that could be resolved by editing x. Thus, x resolves at most the same number of edges as it did originally.

Finally, we consider the case where v forms an inversion with a back neighbor b. In order for b to be able to form another inversion with a different marked neighbor, it must have forward degree $k-1$. However, it must first form an inversion with v again which has already been considered. □

Theorem 3.6.22. *There exists an $O(\log n)$-approximation for finding the minimum size edit set to reduce the degeneracy of a graph from k to $k-1$.*

Proof. Let m_0 be the number of marked edges in the first min-degree ordering. Since the optimal edit set (of size k) resolves every marked edge in k steps, the k edits must on average resolve $\frac{m_0}{k}$ marked edges. Thus, the largest number of marked edges resolved by an edit from the optimal edit set must be at least $\frac{m_0}{k}$. Fix a sequence of the optimal edit set. By Lemma 3.6.21, the number of marked edges that this edit resolves at the current step must be at least as large as when it appears in the optimal sequence. Thus, there must exist an edit which resolves at least $\frac{m_0}{k}$ marked edges, and so the edit that resolves the most marked edges must resolve at least $\frac{m_0}{k}$ as well. After one iteration then, there are $m_1 \leq m_0(1-\frac{1}{k})$ marked edges remaining. Since the edited graph is a subgraph of the original, the optimal sized edit set must still be of size at most k and so the same analysis applies. Thus, after t steps, there are at most $m_0(1-\frac{1}{k})^t$ marked edges remaining. If we set $t = k\ln m_0$, then there at most $m_0(1-\frac{1}{k})^{k\ln m_0} \leq m_0 \cdot \frac{1}{e}^{\ln m_0} = 1$ marked edges remaining. Thus, we need at most $k\ln m_0 + 1$ iterations to resolve every marked edge. Since we add one edit at each iteration, we produce an edit set of size at most $k\ln m_0 + 1$. Because m_0 is at most n^2, the size of the edit set is $k\ln m_0 + 1 \leq k\ln n^2 + 1 = O(k\log n)$. □

Corollary 3.6.23. *There exists an $O(d\cdot\log n)$-approximation for finding the minimum size edit set to reduce the degeneracy of a graph from k to $k-d$.*

Proof. Apply the above algorithm d times. Let G_i be the edited graph after i applications. Note that G_i has degeneracy $k-i$. The optimally sized edit set OPT to reduce the degeneracy of G_i from $k-i$ to $k-i-1$ is at most the size of the smallest edit set to reduce the degeneracy of G_0 from k to $k-d$. Thus, at each iteration, we add at most $O(|\mathrm{OPT}|\log n)$ edits to our edit set. After d iterations then, we have a graph with degeneracy $k-d$ and an edit set of size $O(d|\mathrm{OPT}|\log n)$. □

Corollary 3.6.24. *There exists an $O(k\cdot\log n)$-approximation for finding the minimum size edit set to reduce the degeneracy of a graph to k.*

Proof. Apply an algorithm from Section 3.6.1 or 3.6.2. This yields a graph with degeneracy $O(k)$ and an edit set of size $O(|\mathrm{OPT}|)$. Apply the algorithm from Corollary 3.6.23 to reduce the degeneracy by the remaining $O(k)$ steps. The final size of the edit set is $O(|\mathrm{OPT}|) + O(k|\mathrm{OPT}|\log n) = O(k|\mathrm{OPT}|\log n)$. □

3.6.5 Treewidth/Pathwidth: Bicriteria Approximation for Vertex Editing

In this section, we design a polynomial-time algorithm that constructs a $(O(\log^{1.5} n), O(\sqrt{\log w}))$-bicriteria approximate solution to w-Treewidth Vertex Deletion: the size of the edit set is at most $O(\log^{1.5} n)$ times the optimum ($\mathrm{OPT}_{w\text{-TW-V}}(G)$) and the resulting subgraph has treewidth $O(w\sqrt{\log w})$. We also give a $(O(\log^{1.5} n), O(\sqrt{\log w} \cdot \log n))$-bicriteria approximation for editing to pathwidth w.

Our approach relies on known results for *vertex c-separators*, structures which are used extensively in many other algorithms for finding an approximate tree decomposition.

Definition 3.6.25. *For a subset of vertices W, a set of vertices $S \subseteq V(G)$ is a* vertex c-separator *of W in G if each component of $G[V \setminus S]$ contains at most $c|W|$ vertices of W. The minimum size vertex c-separator of a graph, denoted $\mathsf{sep}_c(G)$, is the minimum integer k such that for any subset $W \subseteq V$ there exists a vertex c-separator of W in G of size k.*

The size of a minimum size vertex c-separator of a graph is a parameter of interest and has applications in bounding treewidth and finding an approximate tree decomposition. Our algorithms in this section use vertex $\left(\frac{3}{4}\right)$-separators.

Lemma 3.6.26 ([FHL08, Section 6.2])**.** *There exist polynomial time algorithms that find a vertex $\left(\frac{3}{4}\right)$-separator of a graph G of size $c_1 \cdot \mathsf{sep}_{2/3}(G)\sqrt{\log \mathsf{sep}_{2/3}(G)}$, for a sufficiently large constant c_1.*

The following bounds relating the treewidth of G, $\mathsf{tw}(G)$, and the minimum size vertex $(\frac{2}{3})$-separator of G, $\mathsf{sep}_{2/3}(G)$, are useful in the analysis of our proposed algorithm for the problem of editing to treewidth w.

Lemma 3.6.27 ([RS86, RS95, HW17, Lemma 7])**.** *For any graph G, $\mathsf{sep}_{2/3}(G) \leq \mathsf{tw}(G)+1 \leq 4\mathsf{sep}_{2/3}(G)$.*

Lemma 3.6.28. *For any graph $G = (V, E)$, and integer $w \leq \frac{3}{4} \cdot \mathsf{tw}(G)$, $\mathsf{sep}_{2/3}(G) \leq 6 \cdot \mathrm{OPT}_{w\text{-TW-V}}(G)$.*

Proof. It is straightforward to verify that for any $X \subseteq V$, if $\mathsf{tw}(G[V \setminus X]) = w$, then $|X| \geq \mathsf{tw}(G) - w$. Suppose not. Then, we can add X to all bags in the tree decomposition of $G[V \setminus X]$ and the resulting tree decomposition of G has treewidth less than $\mathsf{tw}(G)$ which is a contradiction.

If we assume $w \leq \frac{3}{4} \cdot \mathsf{tw}(G)$, then $\mathrm{OPT}_{w\text{-TW-V}}(G) \geq \mathsf{tw}(G) - w \geq \mathsf{tw}(G)/4$, which, together with Lemma 3.6.27, implies that $\mathsf{sep}_{2/3}(G) \leq \frac{3}{2} \cdot \mathsf{tw}(G) \leq 6 \cdot \mathrm{OPT}_{w\text{-TW-V}}(G)$. □

Treewidth: $(O(\log^{1.5} n), O(\sqrt{\log w}))$-approximation for vertex deletion

Our method exploits the general recursive approach of the approximation algorithms for constructing a tree decomposition [Ami10, BGHK95, FHL08, Ree92]. Our algorithm iteratively subdivides the graph, considering $G[V_i]$ in iteration i. We first apply the result of [BGHK95, FHL08] to determine if $G[V_i]$ has a tree decomposition with "small"

width; if yes, the algorithm removes nothing and terminates. Otherwise, we compute an approximate vertex (3/4)-separator S of $G[V_i]$ (applying the algorithm of [FHL08]), remove it from the graph, and recurse on the connected components of $G[V_i \setminus S]$. We show that the total number of vertices removed from G in our algorithm is not more than $O(\mathrm{OPT}_{w\text{-TW-V}}(G)\log^{1.5} n)$ in Theorem 3.6.31 and the treewidth of the resulting graph is $O(w\sqrt{\log w})$ in Theorem 3.6.32.

Algorithm 3 Approximation for Vertex Editing to Bounded Treewidth Graphs

1: **procedure** TreeWidthNodeEdit($G = (V, E)$, w)
2: $t \leftarrow$ compute $\mathrm{tw}(G)$ by invoking the algorithm of [BGHK95] together with [FHL08]
3: **if** $t \leq 32c_1 \cdot w\sqrt{\log w}$ **then**
4: **return** $\varnothing$
5: **else**
6: $S \leftarrow$ compute a vertex $(\frac{3}{4})$-separator of G by invoking the algorithm of [FHL08]
7: **let** $G[V_1], \cdots, G[V_\ell]$ be the connected components of $G[V \setminus S]$.
8: **return** $\left(\bigcup_{i\leq\ell} \textsc{TreeWidthNodeEdit}(G[V_i], w)\right) \cup S$

The key observation in our approach is the following lemma:

Lemma 3.6.29. *Suppose that the vertex set of $G = (V, E)$ is partitioned into $V_1, \cdots, V_\ell$. The minimum edit set of G to treewidth w has size at least $\sum_{i\leq\ell} \max(0, \mathrm{tw}(G[V_i]) - w)$.*

Proof. This directly follows from the straightforward observation that if $\mathrm{tw}(G[V \setminus X]) = w$, then $|X| \geq \mathrm{tw}(G) - w$. Since the sets of vertices are disjoint, then the lower bound on the number of vertices that must be deleted is the summation of the lower bound of the number of vertices that must be deleted in each disjoint set. □

In Algorithm 3, we use the approach of Bodlaender et al. [BGHK95] together with the $O(\sqrt{\log \mathrm{tw}(G)})$-approximation algorithm of [FHL08] for computing treewidth of G in Line 2.

Theorem 3.6.30 ([BGHK95, FHL08])**.** *There exists an algorithm that, given an input graph G, in polynomial time returns a tree decomposition of G of width at most $c_2 \cdot \mathrm{tw}(G)\sqrt{\log \mathrm{tw}(G)}$ and height $O(\log |V(G)|)$ for a sufficiently large constant c_2.*

For the sake of completeness, we provide the proof of Theorem 3.6.30 in Section 3.6.5. Next, we analyze the performance of Algorithm 3.

Theorem 3.6.31. *Algorithm 3 removes at most $O(\log^{1.5} n)\mathrm{OPT}_{w\text{-TW-V}}(G)$ vertices from any n-vertex graph G.*

Proof. The proof is by induction on the number of vertices in the given induced subgraph of G: we show that for any subset $V' \subseteq V$, the number of vertices removed by Algorithm 3 is at most $(c \cdot \log_{4/3}|V'| \cdot \sqrt{\log n})\mathrm{OPT}_{w\text{-TW-V}}(G[V'])$ where $c \geq 6$ is a fixed constant. We remark that c must also be greater than the constant c_2 in Theorem 3.6.30 for the width guarantee.

By the condition in Line 3 of the algorithm, the claim trivially holds for the case $|V| = O(w\sqrt{\log w})$. Assume that the claim holds for all induced subgraphs of G containing at most $n'-1$ vertices. Next, we show that the claim holds for any n'-vertex induced subgraph of G, $G[V']$, too. If $t \leq 32c_1 \cdot w\sqrt{\log w}$, then no vertex is deleted and the claim holds. Otherwise,

$$\begin{aligned} 32c_1 \cdot w\sqrt{\log w} < t &\leq c_1 \cdot \mathsf{sep}_{2/3}(G[V'])\sqrt{\log \mathsf{sep}_{2/3}(G[V'])} && \text{by Lemma 3.6.26,} \\ &\leq c_1 \cdot (2\mathsf{tw}(G[V']))\sqrt{2\log \mathsf{tw}(G[V'])} && \text{by Lemma 3.6.27,} \end{aligned}$$

which implies that $\mathsf{tw}(G[V']) \geq 4w$. By Lemma 3.6.28, $\mathsf{sep}_{2/3}(G[V']) \leq 6\,\mathrm{OPT}_{w\text{-TW-V}}(G[V'])$ which implies that $|S'| \leq c_1\sqrt{\log \mathsf{sep}_{2/3}(G[V'])} \cdot \mathsf{sep}_{2/3}(G[V']) \leq 6c_1\sqrt{\log n}\cdot\mathrm{OPT}_{w\text{-TW-V}}(G[V'])$ where S' is a vertex $(\frac{3}{4})$-separator of V' computed in Line 6 of the algorithm.

Let $V'_1, \cdots, V'_\ell$ be the disjoint components in $G[V' \setminus S']$. Then, $\mathrm{OPT}_{w\text{-TW-V}}(G[V']) \geq \sum_{i\leq\ell}\mathrm{OPT}_{w\text{-TW-V}}(G[V'_i])$, by Lemma 3.6.29. Further, by the induction assumption for each $i \leq \ell$, the number of vertices removed by TreeWidthNodeEdit$(G[V'_i], w)$ is at most $(c\log_{4/3}|V'_i| \cdot \sqrt{\log n})\mathrm{OPT}_{w\text{-TW-V}}(G[V'_i])$. Hence, the vertices removed by TreeWidthNodeEdit$(G[V'], w)$, satisfy

$$\begin{aligned} |X| &\leq 6c_1\sqrt{\log n} \cdot \operatorname*{OPT}_{w\text{-TW-V}}(G[V']) + \sum_{i\leq\ell}(c\log_{4/3}|V'_i| \cdot \sqrt{\log n}) \cdot \operatorname*{OPT}_{w\text{-TW-V}}(G[V'_i]) \\ &\leq 6c_1\sqrt{\log n} \cdot \operatorname*{OPT}_{w\text{-TW-V}}(G[V']) + (c\log_{4/3}\frac{3|V'|}{4} \cdot \sqrt{\log n})\sum_{i\leq\ell}\operatorname*{OPT}_{w\text{-TW-V}}(G[V'_i]) \\ &\leq c\sqrt{\log n} \cdot (1+\log_{4/3}|V'|-1)\operatorname*{OPT}_{w\text{-TW-V}}(G[V']) && \text{with } c > 6c_1. \\ &\leq (c\log_{4/3}|V'| \cdot \sqrt{\log n})\operatorname*{OPT}_{w\text{-TW-V}}(G[V']). \end{aligned}$$

□

Theorem 3.6.32. *The treewidth of the subgraph of G returned by Algorithm 3 is $O(w \cdot \sqrt{\log w})$.*

Proof. This follows immediately from the condition in Line 3 of the algorithm. □

Pathwidth: $(O(\log^{1.5} n), O(\sqrt{\log w} \cdot \log n))$-approximation for vertex deletion

Our algorithm in this section builds on the reduction of Bodleander et al. [BGHK95] from tree decomposition to path decomposition and Algorithm 3 described in Section 3.6.5 for finding a minimum size edit set to treewidth w. The main component of the reduction approach of [BGHK95] is the following.

Lemma 3.6.33 ([BGHK95])**.** *Given a tree decomposition of G with width at most w and height at most h, we can find a path decomposition of G with width at most $w \cdot h$ efficiently.*

Corollary 3.6.34. *Given an input graph $G = (V, E)$ and a target pathwidth w, Algorithm 3 removes $O(\log^{1.5} n)\cdot\mathrm{OPT}_{w\text{-PW-V}}(G)$ vertices X such that $\mathsf{pw}(G[V \setminus X]) \leq (\sqrt{\log w}\cdot\log n)\cdot w$.*

Proof. By Theorem 3.6.31, $|X| \leq (\log^{1.5} n) \cdot \mathrm{OPT}_{w\text{-TW-V}}(G)$. Since $\mathrm{tw}(G) \leq \mathrm{pw}(G)$ for all G, we have $\mathrm{OPT}_{w\text{-TW-V}}(G) \leq \mathrm{OPT}_{w\text{-PW-V}}(G)$. Hence, $|X| \leq \log^{1.5} n \cdot \mathrm{OPT}_{w\text{-PW-V}}(G)$. Further, by Lemma 3.6.33 and Theorem 3.6.30, $\mathrm{pw}(G[V \setminus X]) \leq (w\sqrt{\log w}) \cdot \log n$. □

Proof of Theorem 3.6.30

In this section, we provide the proof of Theorem 3.6.30 which is essentially via the tree decomposition of Bodleander et al. [BGHK95] by plugging in the $O(\sqrt{\log \mathrm{OPT}})$-approximation algorithm of [FHL08] for vertex separators. Algorithm 4 is the recursive approach of [BGHK95] for approximating treewidth (and constructing its tree decomposition).

Algorithm 4 Approximation Algorithm for Tree Decomposition (From [BGHK95, FHL08])

1: **procedure** TreeDecomposition(G, Z, W) ⟨⟨$Z \cap W = \varnothing$, *output contains* W *in root bag*⟩⟩
2: **if** $8|Z| \leq |W|$ **then**
3: **return** a tree decomposition with a single node containing $Z \cap W$
4: **else**
5: $S \leftarrow$ a vertex $(\frac{3}{4})$-separator of W in $G[Z \cup W]$ by invoking the algorithm of [FHL08]
6: $T \leftarrow$ a vertex $(\frac{3}{4})$-separator of $Z \cup W$ in $G[Z \cup W]$ by invoking the algorithm of [FHL08]
7: **let** $G[V_1], \cdots, G[V_\ell]$ be the connected components of $G[(W \cup Z) \setminus (S \cup T)]$.
8: **for** $i = 1$ to ℓ **do**
9: $Z_i \leftarrow Z \cap V_i$
10: $W_i \leftarrow W \cap V_i$
11: $T_i \leftarrow$ TreeDecomposition$(G, Z_i, W_i \cup S \cup T)$
12: **return** the tree decomposition with $(W \cup S \cup T)$ as its root and $T_1, \cdots, T_\ell$ as its children

Claim 3.6.35. *If W and Z are disjoint sets of vertices of G and $|W| \leq 32c_1 \cdot \mathrm{tw}(G)\sqrt{\log \mathrm{tw}(G)}$, then the solution produced by the algorithm is a tree decomposition of $G[W \cup Z]$ of width at most $36c_1 \cdot \mathrm{tw}(G)\sqrt{\log \mathrm{tw}(G)}$.*

Proof. First we show that the output is a valid tree decomposition of $G[W \cup Z]$.

- **All edges of $G[Z \cup W]$ are covered in T.** The proof is by an induction on the recursive structure of the algorithm. For a leaf bag in the tree decomposition ($|Z| \leq |W|/8$), a single bag contains all vertices of $Z \cup W$; hence, the claim holds. Now, suppose that this property holds for all subtrees rooted at children of the tree decomposition constructed by TreeDecomposition(G, Z, W). Consider an edge $uv \in E$. If $u, v \in W$, then u and v are both contained in the root bag and it is covered in the tree decomposition. Otherwise, v and u both belong to $V_i \cup S \cup T$ for

an $i \in [\ell]$ and by the induction hypothesis, uv is covered in the subtree T_i corresponding to TreeDecomposition$(G, Z_i, W_i \cup S \cup T)$. Thus, the property holds for T as well.

- **Bags containing each vertex are connected in tree structure T.** The proof is by induction on the recursive structure of TreeDecomposition$(\cdot)$. More precisely, we show that for each pair (Z', W'), the bags containing a vertex $v \in Z' \cup W'$ are connected in TreeDecomposition(G, Z', W'). The property trivially holds for the leaves of T (the case $|Z'| \leq |W'|/8$). Suppose that this property holds for all subtrees rooted at children of the tree decomposition $T =$ TreeDecomposition(G, Z, W). Then, we show that the property holds for T as well. If $v \in W \cup S \cup T$, then by the induction hypothesis on children of T, the property holds for T as well. Otherwise, v is only contained in one of the children of T (i.e., $v \in Z_i \cup W_i$) and by the induction hypothesis the property holds.

Next, we show that the width of the tree decomposition constructed by TreeDecomposition(G, Z, W) is at most $36c_1 \cdot \mathsf{tw}(G)\sqrt{\log \mathsf{tw}(G)}$. By induction on the size of Z, it suffices to show that $|W \cup S \cup T| \leq 36c_1 \cdot \mathsf{tw}(G)\sqrt{\log \mathsf{tw}(G)}$ and for each i, $|W_i \cup S \cup T| \leq 32c_1 \cdot \mathsf{tw}(G)\sqrt{\log \mathsf{tw}(G)}$. Note that, by Line 3, if $|Z| \leq |W|/8$, then the returned tree decomposition has width at most $36c_1 \cdot \mathsf{tw}(G)\sqrt{\log \mathsf{tw}(G)}$. Suppose that the claim holds for all (Z', W') where $|Z'| < |Z|$. We bound the size of S and T as follows:

$$\begin{aligned} |S|, |T| &\leq c_1 \mathsf{sep}_{2/3}(G)\sqrt{\log \mathsf{sep}_{2/3}(G)} && \text{by Lemma 3.6.26,} \\ &< 4c_1 \cdot \mathsf{tw}(G)\sqrt{\log \mathsf{tw}(G)} && \text{by Lemma 3.6.27.} \end{aligned}$$

Since $|W| \leq 32c_1 \cdot \mathsf{tw}(G)\sqrt{\log \mathsf{tw}(G)}$, the root bag has size at most $(32c_1 + 2 \cdot 4c_1) \cdot \mathsf{tw}(G)\sqrt{\log \mathsf{tw}(G)}$. Moreover, since S and T are respectively a $(\frac{3}{4})$-vertex separator of W and $W \cup Z$ in $G[W \cup Z]$, for each $i \in [\ell]$,

$$\begin{aligned} |Z_i| &\leq \frac{3}{4}|Z| < |Z|, \text{ and} \\ |W_i \cup S \cup T| &\leq \left(\frac{3}{4}32c_1 + 8c_1\right) \cdot \mathsf{tw}(G)\sqrt{\log \mathsf{tw}(G)} \leq 32c_1 \cdot \mathsf{tw}(G)\sqrt{\log \mathsf{tw}(G)}. \end{aligned}$$

Thus, it follows from the induction hypothesis that the width of each subtree T_i is at most $36c_1 \cdot \mathsf{tw}(G)\sqrt{\log \mathsf{tw}(G)}$. □

Proof of Theorem 3.6.30. It follows from Claim 3.6.35 that TreeDecomposition$(G = (V, E), \varnothing, V)$ constructs a tree decomposition of G of width at most $36c_1 \cdot \mathsf{tw}(G)\sqrt{\log \mathsf{tw}(G)}$. Moreover, since for each $i \in [\ell]$, $|Z_i| \leq 3|Z|/4$, the returned tree decomposition has height $O(\log |V(G)|)$. □

3.7 Open Problems

We hope that our framework for extending approximation algorithms from structural graph classes to graphs near those classes, by editing to the class and lifting the resulting solution, can be applied to many more contexts. Specific challenges raised by this work include the following:

1. Editing via edge contractions. Approximation algorithms for this type of editing would enable the framework to apply to the many optimization problems closed under just contraction, such as TSP Tour and Connected Vertex Cover.
2. Editing to H-minor-free graphs. Existing results apply only when H is planar [FLMS12]. According to Graph Minor Theory, the natural next steps are when H can be drawn with a single crossing, when H is an apex graph (removal of one vertex leaves a planar graph), and when H is an arbitrary graph (say, a clique). H-minor-free graphs have many PTASs (e.g., [DH05, DHK11]) that would be exciting to extend via structural rounding.
3. Editing to bounded clique number and bounded weak c-coloring number. While we have lower bounds on approximability, we lack good approximation algorithms.

Acknowledgments

We thank Abida Haque and Adam Hesterberg for helpful initial discussions, Nicole Wein for providing helpful references on bounded degeneracy problems, and Michael O'Brien for helpful comments on the manuscript.

Chapter 4

Massively Parallel Algorithms for Small Subgraph Counting

This chapter presents results from the paper titled, "Massively Parallel Algorithms for Small Subgraph Counting" that the thesis author coauthored with Amartya Shankha Biswas, Talya Eden, Slobodan Mitrović and Ronitt Rubinfeld [BEL+20]. This paper is currently under submission at the time of the writing of this thesis.

4.1 Introduction

Estimating the number of small subgraphs, cliques in particular, is a fundamental problem in computer science, and has been extensively studied both theoretically and from an applied perspective. Given its importance, the task of counting subgraphs has been explored in various computational settings, e.g., sequential [AYZ97, Vas09, BHKK09], distributed and parallel [SV11, PT12, KPP+14, PSKP14, LQLC15], streaming [BYKS02, KMSS12, BC17, MVV16], and sublinear-time [ELRS17, ABG+18, AKK19, ERS20]. There are usually two perspectives from which subgraph counting is studied: first, optimizing the running time (especially relevant in the sequential and sublinear-time settings) and, second, optimizing the space or query requirement (relevant in the streaming, parallel, and distributed settings). In each of these perspectives, there are two, somewhat orthogonal, directions that one can take. The first is *exact* counting. However, in most scenarios, algorithms that perform exact counting are prohibitive, e.g., they require too much space or too many parallel rounds to be implementable in practice.

Hence, the second direction of obtaining an *estimate/approximation* on the number of small subgraphs is both an interesting theoretical problem and of practical importance. If $H_\#$ is the number of subgraphs isomorphic to H, the main question in approximate counting is whether we can design algorithms that, under given resource constraints, provide approximations that concentrate well. This concentration is usually parametrized by $H_\#$ (and potentially some other parameters). In particular, most known results do not provide a strong approximation guarantee when $H_\#$ is very small, e.g., $H_\# \in O(1)$. So, the main attempts in this line of work is to provide an estimation that concentrates well while imposing as small a lower bound on $H_\#$ as possible.

Due to ever increasing sizes of data stores, there has been an increasing interest in designing scalable algorithms. The *Massively Parallel Computation* (MPC) model is a theoretical abstraction of popular frameworks for large-scale computation, such as MapReduce [DG08], Hadoop [Whi12], Spark [ZCF+10] and Dryad [IBY+07]. MPC gained significant interest recently, most prominently in building algorithmic toolkits for graph processing [GSZ11, LMSV11, BKS13, ANOY14, BKS14, HP15, AG15, RVW16, IMS17, CLM+18, Ass17, ABB+19, GGK+18, HLL18, BFU18, ASW18, BEG+18, BDH+19, BBD+19, BHH19, ASZ19, ASW19, GLM19, GKMS19, GU19, LMOS19, ILMP19, CFG+19, GKU19, GNT20]. Efficiency of an algorithm in MPC is characterized by three parameters: round complexity, the space per machine in the system, and the number of machines/total memory used. Our work aims to design efficient algorithms with respect to all three parameters and is guided by the following question:

How does one design efficient massively parallel algorithms for small subgraph counting?

In this chapter, we are working in the Massively Parallel Computation (MPC) model introduced by [KSV10, GSZ11, BKS13] and defined by Section 2.2.2.

4.1.1 Our Contributions

In this chapter, let $G = (V, E)$ be a graph that has T triangles.

First we study the question of approximately counting the number of triangles under the restriction that the round and total space complexities are essentially *optimal*, i.e., $O(1)$ and $\widetilde{O}(m)$, where $\tilde{O}$ hides $O(\operatorname{poly}\log n)$ factors, respectively.

Theorem 4.1.1. *Let $G = (V, E)$ be a graph over n vertices, m edges, and let T be the number of triangles in G. Assuming*

1. $T = \widetilde{\Omega}\left(\sqrt{\frac{m}{S}}\right)$,
2. $S = \widetilde{\Omega}\left(\max\left\{\frac{\sqrt{m}}{\varepsilon}, \frac{n^2}{m}\right\}\right)$,

there exists an MPC *algorithm, using $\mathcal{M}$ machines, each with local space S, and total space $\mathcal{M}S = \tilde{O}_\varepsilon(m)$, that outputs a $(1 \pm \varepsilon)$-approximation of T, with high probability, in $O(1)$ rounds.*

For $S = \Theta(n \log n)$ (specifically, $S > 100n \log n$) in Theorem 4.1.1, we derive the following corollary.

Corollary 4.1.2. *Let G be a graph and T be the number of triangles it contains. If $T \geq \sqrt{d_{avg}}$, then there exists an MPC algorithm that in $O(1)$ rounds with high probability outputs a $(1+\varepsilon)$-approximation of T. This algorithm uses a total space of $\widetilde{O}(m)$ and space $\widetilde{\Theta}(n)$ per machine. d_{avg} is the average degree of the vertices in the graph.*

The necessity of a lower bound in the triangle/subgraph count when obtaining good approximations for the triangle/subgraph count in any sampling/streaming based algorithm has been demonstrated in numerous prior papers in the streaming [BYKS02, KMSS12, BC17, MVV16] and sublinear query-based models [ELRS17, ABG+18, AKK19,

ERS20]; in the MPC setting, we improve the lower bound provided by [PT12] by a quadratic factor. We extend these results to approximately counting *any* K-subgraph where K is a constant in Theorem 4.5.13 and Corollary 4.5.14.

There is a long line of work on computing approximate triangle counting in parallel computation [Coh09, TKMF09, SV11, YK11, PT12, KMPT12, PC13, SPK13, AKM13, PSKP14, KPP+14, JS17a] and references therein. Despite this progress, and to the best of our knowledge, on one hand, each MPC algorithm for exact triangle counting either requires strictly super-polynomial in m total space, or the number of rounds is super-constant. On the other hand, each algorithm for approximate triangle counting requires $T \geq d_{avg}$ even when the space per machine is $\Theta(n)$. We design an algorithm that has essentially optimal total space and round complexity, while at least quadratically improving the requirement on T.

Furthermore, since the amount of messages sent and received by each machine is bounded by $O(n)$, by [BDH18], our algorithm directly implies an $O(1)$-rounds algorithm in the CONGESTED-CLIQUE model under the same restriction $T = \Omega(\sqrt{m/n})$. The best known (to our knowledge) triangle approximation algorithm for general graphs in this model, is an $O(n^{1/3}/t^{2/3})$-rounds algorithm by [DLP12]. This bound only results in a constant round complexity when $T = \Omega(\sqrt{n})$.

The second question we consider is the question of exact counting, for which we present an algorithm whose total space depends on the arboricity of the input graph.

The arboricity of a graph (roughly) equals the average degree of its densest subgraph. Specifically, the *arboricity*, α, of a graph, $G = (V, E)$, is equal to the minimum number of forests into which E can be partitioned (Definition 2.1.8). The class of graphs with bounded arboricity includes many important graph families such as planar graphs, bounded degree graphs and randomly generated preferential attachment graphs. Also many real-world graphs exhibit bounded arboricity [GG06, ELS13, SERF18], making this property important also in practical settings. For many problems, a bound on the arboricity of the graph allows for much more efficient algorithms and/or better approximation ratios [AG09, ELS13].

Specifically for the task of subgraph counting, in a seminal paper, Chiba and Nishizeki [CN85] prove that triangle enumeration can be performed in $O(m\alpha)$ time, and assuming 3SUM-hypothesis[1] this result is optimal up to dependencies in $O(\text{poly}\log n)$ [Pat10, KPP16]. Many applied algorithms also rely on the property of having bounded arboricity in order to achieve better space and time bounds, e.g., [SW05, CC11, Lat08]. Our main theorem with respect to this question is the following. Here and throughout, we use O_δ to hide constant factors of δ.

Theorem 4.1.3. *Let $G = (V, E)$ be a graph over n vertices, m edges and arboricity α. Count-Triangles(G) takes $O_\delta(\log\log n)$ rounds, $O\left(n^\delta\right)$ space per machine for any $\delta > 0$, and $O(m\alpha)$ total space.*

It is interesting to note that our total space complexity matches the time complex-

[1]The 3SUM-hypothesis is a fined-grained complexity hypothesis that states that the 3SUM problem cannot be solved in $O(n^{2-\varepsilon})$ for any constant $\varepsilon > 0$.

ity (both upper and conditional lower bounds) of combinatorial [2] triangle counting algorithms in the sequential model [CN85, Pat10, KPP16].

We prove the above theorem in Section 4.6, and in Section 4.8, we prove that this result can be extended to exactly counting k-cliques for any constant k:

Theorem 4.1.4. *Let $G = (V, E)$ be a graph over n vertices, m edges and arboricity α. Count-Cliques(G) takes $O_\delta(\log\log n)$ rounds, $O\left(n^\delta\right)$ space per machine for any $\delta > 0$, and $O\left(m\alpha^{k-2}\right)$ total space.*

We can improve on the total space usage if we are given machines where the memory for each individual machine satisfies $\alpha < n^{\delta'/2}$ where $\delta' < \delta$. In this case, we obtain an algorithm that counts the number of k-cliques in G using $O(n\alpha^2)$ total space and $O_\delta(\log\log n)$ communication rounds.

Finally, in Section 4.9, we consider the problem of exactly counting subgraphs of size at most 5, and show that the recent result of [BPS20] for this question in the sequential model, can be implemented in the MPC model. Here too, our total space complexity matches the time complexity of the sequential model algorithm.

Theorem 4.1.5. *Let $G = (V, E)$ be a graph over n vertices, m edges, and arboricity α. The algorithm of BPS for counting the number of occurrences of a subgraph H over $k \leq 5$ vertices in G can be implemented in the MPC model with high probability and round complexity $O_\delta(\sqrt{\log n}\log\log m)$ for any $\delta > 0$. The space requirement per machine is $O(n^{2\delta})$ and the total space is $O(m\alpha^3)$.*

4.1.2 Related Work

There has been a long line of work on small subgraph counting in networks. [FFF15] design an algorithm for clique counting, but their approach requires the total space of $O(m^{3/2})$. Another work, [AFU13], shows how to count small subgraphs by using b^3 machines, each requiring $O(m/b^2)$ space per machine. Hence, it uses a total space of $O(mb)$. Therefore, this approach either requires super-linear total space or almost $O(m)$ space per machine.

[SV11] studied triangle counting in MPC, where they design two algorithms. The first of those algorithms, that runs in 2 rounds, requires $O(\sqrt{m})$ space per machine and total space $O(m^{3/2})$. Their second algorithm requires only one round for exact triangle counting, total space $O(\rho m)$ and space per machine $O(m/\rho^2)$. Therefore, for this algorithm to work with polynomially less than space m per machine, it has to allow for a total space that is polynomially larger than m. [CC11] focus on algorithms that require a total space of $O(m)$. In the worst case, their algorithm performs $O(|E|/S)$ MPC rounds to output the exact count where S is the maximum space per machine.

[TKMF09, AKM13] design a randomized algorithm for approximate triangle counting. Their approach first sparsifies the input graph by sampling a subset of edges, and executes some of the known algorithms for triangle counting on the sampled subgraph.

[2] Combinatorial algorithms, usually, refer to algorithms that do not rely on fast matrix multiplication.

Denoting by p their sampling probability, their approach outputs a $(1+\varepsilon)$-approximate triangle count with probability at most $1 - 1/(\varepsilon^2 p^3 T)$. [3] To contrast this result with our approach, consider a graph G where $m = \Theta(n^2)$. Let G' be the edge-sparsified graph as explained above. To be able to execute the first algorithm of [SV11] on G' such that the total space requirement is $O(m)$, one can verify that it is needed to set $p = \Theta(n^{-2/3})$. This in turn implies that the result in [TKMF09, AKM13] outputs the correct approximation with constant probability only if $T = \Omega(n^2)$. An improved lower-bound can be obtained be using the second algorithm of [SV11]. By balancing out ρ and p and for $S = O(n)$, one can show that the sparsification results in a constant probability of success for $T = O(n)$. On the other hand, for $S = O(n)$, our approach obtains the same guarantee even when $T = \Theta(\sqrt{d_{avg}(G)}) = \Theta(\sqrt{n})$.

[PT12] also gives a randomized algorithm for approximate triangle counting, which is based on graph partitioning. The graph is partitioned into $1/p$ pieces, where p is at least the ratio of the maximum number of triangles sharing an edge and T. When all the triangles share one edge, then $p \geq 1$, and hence such an approach would require the space per machine to be $\Omega(m)$. Furthermore, this approach requires the number of triangles to be lower bounded by $T = \Omega\left(d_{avg}\right)$. The techniques of [SPK13], that rely on wedge sampling, provide a $(1+\varepsilon)$-approximation of the triangle count when T is a constant fraction of the sum of squares of degrees.

Other related work Subgraph counting (primarily triangles) was also extensively studied in the streaming model, see [BYKS02, KMSS12, BOV13, JSP13, MVV16, BC17, AKK19] and references therein. This culminated in a result that requires space $\widetilde{O}\left(m^{3/2}/(T\varepsilon^2)\right)$ to estimate the number of triangles within a $(1+\varepsilon)$-factor. In the semi-streaming setting it is assumed that one has $\widetilde{O}(n)$ space at their disposal. This result fits in this regime for $T \geq m^{3/2}/n = d_{avg} \cdot m^{1/2}$. As a reminder, our MPC result requires $T \geq \sqrt{d_{avg}}$ when $S \in \Theta(n)$. Adapting known streaming results to the MPC setting is met with the key challenge of sending the correct subgraphs to each individual machine when each machine has limited memory; without solving this challenge, known streaming algorithms do not produce very good estimations of the triangle count in the MPC model.

In a celebrated result, [AYZ97] designed an algorithm for triangle counting in the sequential settings that runs in $O(m^{2\omega/(\omega+1)})$ time, where ω is the best known exponent of matrix multiplication. Since then, several important works have extended this result to k-clique counting [EG04, Vas09]. In the work-depth (shared-memory parallel processors) model, several results are known for this problem. There has been significant work on practical parallel algorithms for the case of triangle counting (e.g. [AKM13, SV11, PC13, PSKP14, ST15] among others). There is even an annual competition for parallel triangle counting algorithms [Gra]. For counting $k = 4$ and $k = 5$ cliques, efficient practical solutions have also been developed [ANR+17, DAH17, ESBD16, HD14, PSV17]. [DBS18a] recently implemented the Chiba Nishizeki algorithm [CN85] for k-cliques in the parallel setting; although, their work does not achieve polylogarithmic depth. Even more recently, [SS20] enumerated k-cliques in the work-depth model in $O\left(m\alpha^{k-2}\right)$ expected

[3] The actual probability is even smaller and also depends on pairs of triangles that share an edge.

work and $O\left(\log^{k-2} n\right)$ depth with high probability, using $O(m)$ space. Among other distinctions from our setting, the CRCW PRAM model assumes a shared, common memory.

In the CONGESTED-CLIQUE model, [CHKK+19], present an $\tilde{O}(n^{1-2/\omega}) = \tilde{O}(n^{0.158})$ rounds algorithm for matrix multiplication, implying the same complexity for exact triangle counting. [DLP12] present an algorithm for approximate triangle counting in general graphs whose expected running time is $O(n^{2/3}/T^{1/3})$. They also present an $O(\alpha^2/n)$-rounds algorithm for bounded arboricity graphs.

Comparison with Other MPC Algorithm There exists a few related papers in the MPC model [BHH19, GLM19] which *partition* the vertices in the graph to separate machines and compute the desired quantities on the partitions. As we emphasized previously, our algorithm *does not* partition the graph. Instead, we *sample with replacement* the vertices of the graph into machines. Thus, the subgraphs within each machine may overlap. The key difference is that whereas a partition-based algorithm never exceeds the total space of $O(n+m)$, a sampling-based approach *may* exceed this total space if the sampling probabilities are not correctly chosen. In fact, the previous algorithm of [PT12] partitions the vertices into machines, but we are able to achieve a quadratic improvement in the lower bound on the number of triangles with our sampling-based approach.

4.2 Useful MPC Primitives and Notation

In this chapter, we use the notation $\widetilde{O}_\delta$ to hide $\operatorname{poly}\log n$ and δ factors. We are able to do this since $\delta > 0$ is generally assumed to be constant. We use the definition and notation of the MPC model given in Section 2.2.2 and the definition of arboricity and its related properties given in Definition 2.1.8.

Counting Duplicates We make use of a new MPC algorithm for certain parts of this chapter to count the number of repeating elements in a sorted list, given bounded space per machine. We use an algorithm similar to the MPC sorting algorithm given by [GSZ11] to obtain our count duplicates data structure implementation in the MPC model. We prove the following theorem in the MPC model regarding our count duplicates algorithm.

Theorem 4.2.1. *Given a sorted list of N elements implemented on processors where the space per processor is S and the total space among all processors is $O(N)$, for each unique element in the list, we can compute the number of times it repeats in $O\left(\log_S N\right)$ communication rounds.*

Proof. Using a similar construction to the interval tree algorithm defined in [GSZ11] that has branching factor $d = \mathcal{M}/2$, we perform the following to count the number of times each element repeats in our sorted list of N elements. To initialize the tree, each leaf of the tree contains exactly one of the elements in the sorted list of elements where leaf v_i contains element x_i of the list. Let the height of the tree be L, the leaves of the tree be at level $L-1$ and the root be at level 0. Then, the rest of the algorithm proceeds in two phases:

1. **Bottom-up phase:** For each level $\ell = L - 1$ up to 0:
   (a) For each node v on level ℓ:
      i. If v is a leaf, it sends its value x_i to its parent $p(v)$.
      ii. If v is a vertex in level $L-2$, let $(x_i, x_{i+1}, \ldots, x_{i+j})$ where $j < d$ be values obtained from its leaf children from left to right. Let $c(x_i)$ be the count of element x_i among the values obtained from the children of v. The counts are computed locally on the machine storing v. Then, v sends x_i, $c(x_i)$, x_{i+j}, $c(x_{i+j})$ to its parent $p(v)$.
      iii. If v is a non-leaf node on level $\ell < L-2$, let $x_a, c(a), x_b, c(b), \ldots$ be the values of elements obtained from its children and their counts. v updates the counts of all elements received. For example, if $x_a = x_b$, v updates $c(a)$ and $c(b)$ to be $c(a) + c(b)$. Let x_{left} be the first element received from v's leftmost leaf and x_{right} be the second element received from v's rightmost leaf. Then, send these elements and their updated counts, x_{left}, $c(x_{left})$, x_{right}, and $c(x_{right})$, to its parent $p(v)$.
2. **Top-down phase:** For each level $\ell = 0$ down to $\ell = L - 1$:
   (a) For each node v at level ℓ:
      i. If v is the root, then it computed and stored in its memory new repeating counts for the values it received from its children: $x_a, c(x_a), x_b, c(x_b), \ldots$. It sends the new counts and values to its respective child that sent it the value originally (e.g. $x_{left}, c(x_{left})$ to v_{left}). Intuitively, this updates the child's count of values with values that are not in its subtree.
      ii. If v is not the root and is a non-leaf node, it receives the values from its parents for its leftmost and rightmost child counts. Given the set of values it stored from its children it updates the counts with counts of values received from its parents. This allows for the counts to reflect values not in its subtree. Then, it sends the updated counts to its children.
      iii. If v is a leaf, it receives values x_i, $c(x_i)$ from its parent. $c(x_i)$ is then the number of times x_i occurs in the sorted list.

The above procedure uses $O(d)$ space per processor and $O(L)$ rounds of communication. Since $L = O(\log_d(N))$ and $d = \mathcal{M}/2$, the number of rounds of communication that is necessary is $O\left(\log_{\mathcal{M}} N\right)$. □

Subset Containment and Copying We also present MPC algorithms for determining the intersection between two subsets of tuples and for copying the data in one machine onto multiple machines. Although such algorithms are trivial in shared-memory settings, they are somewhat tricky to do while using linear total space and $O_\delta(1)$ rounds in the MPC model.

Lemma 4.2.2. *Given two sets of tuples Q and C (both of which may contain duplicates), for each tuple $q \in Q$, we return whether $q \in C$ in $O(|Q \cup C|)$ total space and $O_\delta(1)$ rounds given machines with space $O(n^\delta)$ for any $\delta > 0$.*

Proof. We first create the following tuples in parallel to represent tuples in Q and C, respectively. For each tuple $q \in Q$, we create the tuple $(q, 1)$. For each tuple $c \in C$, we

create the tuple $(c, 0)$. Let F denote the set of tuples $(c, 0)$ and $(q, 1)$. First, we sort the tuples in F lexicographically (where 0 comes before 1) [GSZ11]. Then, we use the predecessor primitive given in (e.g. [GSZ11, ASS+18], Appendix A of [BDE+19]) to determine the queries $q \in Q$ that are in C. Given the sorted F, we use the predecessor algorithm of [BDE+19] to determine for each $(q, 1)$ tuple, the first tuple that appears before it that has value 0. Suppose this tuple is $(c, 0)$. Then, if $q = c$, then the queried tuple q is in C. For all tuples $q \in Q$, we can then return in parallel whether $q \in C$ also. Both the sorting and the predecessor queries take $O(|Q \cup C|)$ total space and $O_\delta(1)$ rounds. □

Lemma 4.2.3. *Given a machine M that has space $O(n^{2\delta})$ for any $\delta > 0$ and contains data of $O(n^\delta)$ words, we can generate x copies of M, each holding the same data as M, using $O(M \cdot x)$ machines with $O(n^\delta)$ space each in $O(\log_{n^\delta} x)$ rounds.*

Proof. Let M be some machine with n^δ information and $O(n^{2\delta})$ space. We create the x duplicates by repeatedly duplicating each machine M_j^i to n^δ machines $M_{n^\delta \cdot j}^{i+1}, \ldots, M_{n^\delta \cdot j + n^\delta - 1}^{i+1}$, starting with $M_0^0 = M$. Therefore, after $\ell = \log_{n^\delta} x$ rounds this process terminates, and the required duplicates is the set of machines M_1^ℓ to M_x^ℓ. □

4.3 Overview of Our Techniques

4.3.1 Approximate Triangle Counting

Our work reduces approximate triangle counting to exact triangle counting in multiple induced subgraphs of the original graph. In our work, and in contrast to prior approaches (e.g., [PT12]), the induced sugraphs on different machines might *overlap*. This enables us to obtain better concentration bounds compared to prior work, but also brings many challenges.

The high level idea is that each machine M_i samples a subset of vertices V_i by including each vertex in V_i with probability $\hat{p}$. Then each machine computes the induced subgraph $G[V_i]$ and the number of triangles in that subgraph. The total number of triangles seen across all the machines is used as an estimator. We repeat in parallel this sampling process $O(\log n)$ times and return the median of the estimates. The main challenge this approach raises is: *How do we efficiently collect* overlapping *induced subgraphs?* (Indeed, approximate triangle counting, even when the number of triangles is $O(1)$, can be reduced to counting the number of edges in *sparse* induced subgraphs with the total size of subgraphs being $\widetilde{O}(m)$.) We now describe how to handle this task in our case.

Computing induced overlapping subgraphs It is unclear how to compute the induced subgraph on each machine in $O(1)$ rounds without exceeding the total allowed space of $\tilde{O}(m)$. This task becomes easier if the subgraphs are disjoint. The trivial strategy of sampling vertices into the machines and querying for all possible edges between any pair of two vertices takes total space at least $\sum_{i=1}^{\mathcal{M}} X^2$ where X is the number of vertices sampled to each machine. In general, this approach requires much larger than $\widetilde{O}(m)$ space. We tackle this challenge by using a *globally known* hash function $h : V \times [\mathcal{M}] \to \{0, 1\}$,

to indicate whether vertex v is sampled in the i^{th} machine. By requiring that the hash function is known to all machines, we can efficiently compute which edges to send to each machine, i.e., which edges belong to the subgraph $G[V_i]$. However, in order for all machines to be able to compute the hash function, it has to be of limited space. Hence, we cannot hope for a fully independent function, rather we can only use an $(S/\log n)$-wise independent hash function. Still, we manage to show, that we are able to handle the dependencies introduced by the above, even if we allow as little as $O(\log n)$-independence.

4.3.2 Exact Triangle Counting

Let G be a graph over n vertices, m edges and with arboricity at most α. We tackle the task of exactly counting the number of triangles in G in $O_\delta(\log\log n)$ rounds using the following ideas. In each round i, we partition the vertices into low-degree vertices A_i and high-degree vertices, according to a degree threshold γ_i, which grows doubly exponentially in the number of rounds. We then count the number of triangles incident to the set of low degree vertices A_i: Each low-degree vertex $v \in A_i$ sends a list of its neighbors to all its neighbors. Then, any neighbor u of v that detects a common neighbor w to u and v, adds the triangle (u, v, w) to the list of discovered triangles.

Once all triangles incident to the vertices in A_i are processed, we remove this set from the graph and continue with the now smaller graph. This removal of the already processed vertices, allows us to handle larger and larger degrees from step to step, while using a total space of $O(m\alpha)$. This behavior also leads to the $O_\delta(\log\log n)$ round complexity, as after this many rounds all vertices are processed. Ideas similar to some of the above were used in [ASS+18, BDE+19] with respect to connectivity, but ours is the first to achieve such number of rounds for triangle counting.

4.3.3 Counting k-cliques and 5-subgraphs.

We use a similar technique for both problems of exactly counting the number of k-cliques and of subgraphs up to size 5. See Section 4.8.1 for details on the former task, and Section 4.9 for details on the latter. Let H denote the subgraph of interest. We say that a subgraph that can be mapped to a subset of vertices and edges of H of size i is a *i-subcopy of H*. Our main contribution in this section is a procedure that in each round, tries to extend i-subcopies of H to $(i+1)$-subcopies of H, by increasing the total space by a factor of at most α. This is possible by ordering the vertices in H such that each vertex has at most $O(\alpha)$ outgoing neighbors, so that in each iteration only α possible extensions should be considered per each previously discovered subcopy.

Challenges The major challenge we face here is dealing with *finding* and *storing* copies of small (constant-sized) subgraphs. This is a challenge due to the fact that an entire neighborhood of a vertex v may not fit on one machine (recall that we have no restrictions on the δ in $O(n^\delta)$ machine size). Thus, we cannot compute all such small subgraphs on one machine. However, if not done carefully, computing small subgraphs across many machines could potentially result in many rounds of computation (since we potentially have

to try all combinations of vertices in a neighborhood). We solve this issue by formulating a procedure (Lemma 4.2.3) in which we carefully duplicate neighborhoods of vertices across machines.

4.4 Approximate Triangle Counting in General Graphs

In this section we provide our algorithm for estimating the number of triangles in general graphs (see Algorithms 5 and 7) and hence prove Theorem 4.1.1.

Theorem 4.1.1. *Let $G = (V, E)$ be a graph over n vertices, m edges, and let T be the number of triangles in G. Assuming*

1. $T = \widetilde{\Omega}\left(\sqrt{\frac{m}{S}}\right)$,
2. $S = \widetilde{\Omega}\left(\max\left\{\frac{\sqrt{m}}{\varepsilon}, \frac{n^2}{m}\right\}\right)$,

there exists an MPC *algorithm, using $\mathcal{M}$ machines, each with local space S, and total space $\mathcal{M}S = \tilde{O}_\varepsilon(m)$, that outputs a $(1 \pm \varepsilon)$-approximation of T, with high probability, in $O(1)$ rounds.*

The rationale behind the lower bound constraints in Theorem 4.1.1 will become clear when we discuss the challenges and analysis (formally presented in Sections 4.5.1 and 4.5.2).

4.4.1 Overview of the Algorithm and Challenges

Our approach is to use the collection of machines to repeat the following experiment multiple times in parallel. Each machine M_i samples a subset of vertices V_i, and then count the number of triangles $\hat{T}_i$ seen in each induced graph $G[V_i]$. We then use the sum $\hat{T}$ of all $\hat{T}_i$'s as an unbiased estimator (after appropriate scaling) for the number of triangles T in the original graph.

Algorithm 5 Approximate-Triangle-Counting($G = (V, E)$)

$R \leftarrow 0$
parfor $i \leftarrow 1 \dots \mathcal{M}$ **do**
  Let V_i be a random subset of V ⟨⟨*See Section 4.4.1 for details about the sampling*⟩⟩
  if size of $G[V_i]$ exceeds S **then**
    Ignore this sample and set $\hat{T}_i \leftarrow 0$
  else
    Let $\hat{T}_i$ be the number of triangles in $G[V_i]$
    $R \leftarrow R + 1$
Let $\hat{T} = \sum_{i=1}^{\mathcal{M}} \hat{T}_i$
return $\frac{1}{\hat{p}^3 R}\hat{T}$

Moving forwards, for the most part, we will focus on a specific machine M_i containing V_i (a single experiment). We list the main challenges in the analysis of this algorithm, along with the sections that describe them.

1. **Section 4.4.1:** The induced subgraph $G[V_i]$ fits into the memory S of M_i (thus allowing us to count the number of triangles in $G[V_i]$ in one round).
2. **Section 4.4.1:** We can efficiently (in one round) collect all the edges in the induced subgraph $G[V_i]$. This involves presenting an MPC protocol such that the number of messages *sent and received* by any machine is at most the *space per machine* S.
3. **Section 4.4.1** With high constant probability $\geq 9/10$, the number of messages sent and received by each machine M_i is at most S.
4. **Section 4.4.1:** With high *constant* probability $\geq 9/10$, the sum of triangles across all machines, $\hat{T}$, is close to its expected value. Then, repeating the algorithm polylogarithmic number of times with only a polylogarithmic increase in total space, and by using the median trick, allows us to get a high probability bound. The specifics are discussed in Section 4.5.2.

Challenge (1): Ensuring That $G[V_i]$ Fits on a Single Machine

Ensuring that *edges* fit on a machine: Our algorithm constructs V_i by including each $v \in V$ with probability $\hat{p}$, which implies that the expected number of edges in $G[V_i]$ is $\hat{p}^2 m$. Since we have to ensure that each induced subgraph $G[V_i]$ fits on a single machine, we obtain the constraint $\hat{p}^2 m = O(S)$. Concretely, we achieve this by defining:

$$\hat{p} \stackrel{\text{def}}{=} \frac{1}{10} \cdot \sqrt{\frac{S}{mk}}, \tag{4.1}$$

where the parameter $k = O(\log n)$ will be exactly determined later (See Section 4.4.1).

Ensuring that *vertices* fit on a machine: In certain regimes of values of n and m, the *expected number of vertices* ending up in an induced subgraph – $\hat{p}n$, may exceed the space limit S. Avoiding this scenario introduces an additional constraint $\hat{p}n = O(S) \iff S = \Omega(n^2/km)$.

Getting a high probability guarantee: As discussed above, the value of $\hat{p} = \widetilde{\Theta}_\varepsilon(\sqrt{S/m})$ is chosen specifically so that the *expected number of edges* in the induced subgraphs $G[V_i]$ is $\hat{p}^2 m \leq \Theta(S)$, thus using all the available space (asymptotically). In order to guarantee that this bound holds *with high probability* (see Section 4.5.1), we require additional constraints on the space per machine $S = \widetilde{\Omega}_\varepsilon(\sqrt{m})$. We remark that this lower bound $S = \widetilde{\Omega}_\varepsilon(\sqrt{m})$ is essentially saying that $\mathcal{M} = \widetilde{O}_\varepsilon(\sqrt{m})$, i.e. the *space per machine* is much larger than the *number of machines*. This is a realistic assumption as in practice we can have machines with 10^{11} words of local random access memory, however, it is unlikely that we also have as many machines in our cluster.

Lower Bound on *space per machine*: Combining the above two constraints, we get:

$$S > \max\left\{15\frac{\sqrt{mk}}{\varepsilon}, \frac{100n^2}{km}\right\} \implies S = \widetilde{\Omega}_\varepsilon\left(\max\left\{\sqrt{m}, \frac{n^2}{m}\right\}\right) \tag{4.2}$$

Note that Eq. (4.2) always allows *linear space per machine*, as long as $m = \Omega(n)$. Sections 4.5.1 and 4.5.1 present a detailed analysis, showing that the number of vertices and edges in each subgraph is at most S *with high probability*.

Challenge (2): Using k-wise Independence to Compute the Induced Subgraph $G[V_i]$ in MPC

For each sub-sampled set of vertices V_i, we need to compute $G[V_i]$, i.e. we need to send all the edges in the induced subgraph $G[V_i]$ to the machine M_i. Let Q_u denote the set of all machines containing u. Each edge (u, w) then needs to be sent to all machines that contain both u and w, $Q_u \cap Q_w$. Naively, one could try to send the sets Q_u and Q_w to the edge $e = (u, w)$, for all $e \in E$. However, this strategy could result in Q_v being replicated $d(v)$ times. Since the expected size of Q_v is $|Q_v| = \hat{p}\mathcal{M}$ the total expected memory usage of this strategy would be $\sum_{v \in V} |Q_v| \cdot d(v) = \widetilde{\Theta}_\varepsilon(m \cdot \hat{p}\mathcal{M}) = \widetilde{\omega}_\varepsilon(m)$, since $\hat{p} = \widetilde{\Theta}(1/\sqrt{\mathcal{M}})$. This defies our goal of optimal total memory.

Instead, we address this challenge by using globally known hash functions to sample the vertices on each machine. That is, we let $h : V \times [\mathcal{M}] \to \{0, 1\}$ (formally presented in Definition 4.4.1) be a hash function known globally to all the machines. Then we can compute the induced subgraphs $G[V_i]$ as follows.

Algorithm 6 Compute-Induced-Subgraphs

1: $Q_v \leftarrow \{i \in [\mathcal{M}] \mid h(v, i) = 1\}$.
2: $Q_w \leftarrow \{i \in [\mathcal{M}] \mid h(w, i) = 1\}$.
3: **parfor** $i \in Q_v \cap Q_w$ **do**
4: Send e to machine M_i, containing V_i.

Definition 4.4.1. *The hash function $h(v, i)$ indicates whether vertex v is sampled in V_i or not. Specifically, $h : V \times [\mathcal{M}] \to \{0, 1\}$ such that $\mathbb{P}[h(v, i) = 1] = \hat{p}$ for all $v \in V$ and $i \in [\mathcal{M}]$. Recall that $\mathcal{M}$ is the number of machines, and $\hat{p} = \frac{1}{10} \cdot \sqrt{\frac{S}{mk}}$ is the sampling probability set in Eq. (4.1).*

Using limited independence Ideally, we would want a perfect hash function, which would allow us to sample the V_i's i.i.d. from the uniform distribution on V. However, since the hash function needs to be known globally, it must fit into each of the machines. This implies that we *cannot* use a fully independent perfect hash function. Rather, we *can* use one that has a high level of independence. Specifically, given that the space per machine is S, we can have a globally known hash function h that is k-wise independent[4] for any $k < \Theta(S/\log n)$. In fact, we can get away with as little as $(6 \log n)$-wise independence (i.e., $k = 6 \log n$). Recalling Eq. (4.1), this also fixes the sampling probability to be $\hat{p} = \sqrt{S/600m \log n}$.

[4] A k-wise independent hash function is one where the hashes of *any* k distinct keys are guaranteed to be independent random variables (see [WC81]).

Challenge (3): Showing that, with high constant probability, the size of the sent/received messages is bounded.

We need to show that the number of edges sent and received by any machine M_i is at most S *with high constant probability*. To this end, we partition the vertex set V into V_{light} and V_{heavy} by picking a threshold degree τ for the vertices. Following this, we define *light edges* as ones that have both end-points in V_{light}, and conversely, any edge with at least one end-point in V_{heavy} is designated as *heavy*. In order for the protocol to succeed, the following must hold:

1. The number of *light edges* concentrates (see Section 4.5.1).
2. The number of *heavy edges* concentrates (see Section 4.5.1).
3. The number of sent messages is at most S (see Section 4.5.1).

The first two items ensure that each machine M_i *receives* at most S messages, and the last item ensures that each machine *sends* at most S messages. Given the above, we proceed to address the last challenge.

Challenge (4): $\hat{T}$ is close to its expected value

In this section, we provide merely a brief discussion of this challenge for intuition, and we fully analyze the approximation guarantees of our algorithm in Section 4.5.2. That analysis also makes clear the source of our advertised lower-bound on T for which an estimated count concentrates well.

Lower Bound on Number of Triangles In order to output *any* approximation (note that we are ignoring all factors of ε and $O(\text{poly}\log n)$ here) to the triangle count, we must see $\Omega(1)$ triangles amongst all of the induced subgraphs on all the machines. The expected number of triangles in a specific induced $G[V_i]$ is $\hat{p}^3 T$, and therefore, the expected number of triangles overall is $\hat{p}^3 T\mathcal{M}$ which must be $\Omega(1)$ for some setting of T. Since we set $\hat{p}$ such that $\hat{p}^2 m = \Theta(S)$, this gives that $\hat{p}^2 = O(S/m)$ which implies $\hat{p}^2 \cdot \mathcal{M} = \hat{p}^2 \cdot (m/S) = \Theta(1)$. This then immmediately implies that to show that $\hat{p}^3 T$ is $\Omega(1)$, we need only show that $\hat{p} \cdot T$ is $\Omega(1)$. Specifically, we show in Lemma 4.5.9 that when $T > 1/\hat{p}$, we can obtain a $(1 \pm \varepsilon)$-approximation. To get some intuition for this lower bound on T, note that, in the linear memory regime, when $S = \Theta(n)$, this translates to $T > \sqrt{d_{avg}}$, where d_{avg} is the average degree of G.

$$T > \frac{1}{\hat{p}} = \widetilde{\Theta}\left(\sqrt{\frac{m}{S}}\right) \xRightarrow{\text{for } S=\widetilde{\Theta}(n)} T > \widetilde{\Theta}\left(\sqrt{d_{avg}}\right).$$

We present our complete proofs solving the above challenges in Section 4.5.

4.5 Approximate Counting Detailed Analysis

4.5.1 Bounding the Number of Messages Sent/Received by a Machine

This section analyzes the estimation algorithm from Section 4.4. Recall, from Section 4.4.1, that we use a k-wise independent hash function, to compute the induced sub-graphs $G[V_i]$, where $k = 6\log n$. In the subsequent proofs, we will use the following assumptions from within Theorem 4.1.1 (note that we added specific constants).

$$T \geq 10\sqrt{\frac{mk}{S}} \qquad S \geq \max\left\{15\frac{\sqrt{mk}}{\varepsilon}, \frac{100kn^2}{m}\right\} \qquad \mathcal{M} = \frac{2000mk}{\varepsilon^2 S} \tag{4.3}$$

Note that we set the number of machines to a specific value, instead of lower bounding it. This is acceptable, because we can just ignore some of the machines. We will now bound the probability that any of the induced subgraphs *does not fit* on a machine. To that end, we set a degree threshold $\tau = \frac{k}{\hat{p}}$, and define the set of *light* vertices V_{light} to be the ones with degree less than τ. All other vertices are *heavy*, and we let them comprise the set V_{heavy}.

Fix a machine M_i. We prove that, with probability at least 9/10, the number of edges in $G[V_i]$ is upper bounded by S.

We start with analyzing the contribution of the light vertices to the induced sub-graphs.

Bounding the Number of Light Edges *Received* by a Machine

Fix a machine M_i. We first consider the simpler case of bounding the number of edges in $G[V_i]$ that have both end-points in V_{light}. We refer to such edges as *light edges* and denote them by E_{light}. For every edge $e \in E_{light}$, we define a random variable $Z_e^{(i)}$ as follows.

$$Z_e^{(i)} = \begin{cases} 1 & \text{if } e \in G[V_i], \\ 0 & \text{otherwise.} \end{cases}$$

We let $Z^{(i)}$ be the sum over all random variables Z_e^i, $Z^i = \sum_{e \in E_{light}} Z_e^i$, and we let m_ℓ denote the total number of edges with *light* endpoints in the original graph G, i.e., $m_\ell = |E_{light}|$.

We prove the following lemma.

Lemma 4.5.1. *With probability at least* 9/10*, for every* $i \in [\mathcal{M}]$*,* $G[V_i]$ *contains at most* $\frac{1}{4}S$ *light edges.*

Proof. Fix a machine M_i, and let $Z = Z^i$ be as defined in the previous paragraph.

$$\mathbb{E}[Z] = \mathbb{E}\left[\sum_{e \in E_{light}} Z_e\right] = m_\ell \hat{p}^2 \leq m \cdot \frac{S}{100mk} = \frac{S}{100k} \leq \frac{S}{100}.$$

As Z_e are $\{0,1\}$ random variables, we also have $\mathbb{E}[Z] = \mathbb{E}\left[\sum_{e \in E_{light}} Z_e^2\right]$. Now we upper-bound the variance.

$$\begin{aligned}
\mathrm{Var}[Z] &= \mathbb{E}\left[\left(\sum_{e \in E_{light}} Z_e\right)^2\right] - \mathbb{E}\left[\sum_{e \in E_{light}} Z_e\right]^2 \\
&\leq \sum_{e \in E_{light}} \mathbb{E}[Z_e^2] + \sum_{\substack{e_1, e_2 \in E_{light} \\ e_1 \neq e_2}} 2 \cdot \mathbb{E}\left[Z_{e_1} Z_{e_2}\right] \\
&\quad - \sum_{\substack{e_1, e_2 \in E_{light} \\ e_1 \neq e_2}} 2 \cdot \mathbb{E}[Z_{e_1}]\mathbb{E}[Z_{e_2}] \\
&= m_\ell \cdot \hat{p}^2 + \sum_{\substack{e_1, e_2 \in E_{light} \\ e_1 \neq e_2}} 2 \cdot \mathbb{E}\left[Z_{e_1} Z_{e_2}\right] \\
&\quad - \sum_{\substack{e_1, e_2 \in E_{light} \\ e_1 \neq e_2}} 2 \cdot \mathbb{E}[Z_{e_1}]\mathbb{E}[Z_{e_2}] \\
&\leq m_\ell \cdot \hat{p}^2 + \sum_{e_1 \text{ and } e_2 \text{ intersect}} 2 \cdot \mathbb{E}\left[Z_{e_1} Z_{e_2}\right] \\
&\leq m_\ell \cdot \hat{p}^2 + \left(\sum_{v \in V_{light}} d(v)^2\right) \cdot \hat{p}^3 \\
&\leq m_\ell \cdot \hat{p}^2 + \left(\sum_{v \in V_{light}} d(v)\right) \cdot \frac{k}{\hat{p}} \cdot \hat{p}^3 \\
&\leq 3m_\ell \cdot \hat{p}^2 \cdot k \leq 3m \cdot \frac{S}{100mk} \cdot k < \frac{S}{30}
\end{aligned}$$

□

We can now use Chebyshev's inequality to conclude that

$$\begin{aligned}
&\mathbb{P}\left[|Z^{(i)} - \mathbb{E}[Z^{(i)}]| > S/\sqrt{3}\right] \leq \frac{\mathrm{Var}\left[Z^{(i)}\right]}{S^2/3} \\
&\implies \mathbb{P}\left[Z^{(i)} > 3S/4\right] \leq \frac{3}{30S} = \frac{1}{10S}
\end{aligned}$$

Finally, we can use union bound over all $\mathcal{M}$ machines to upper bound the probability that, *any* of the $Z^{(i)}$ values exceeds $3S/4$ (using the the constraints descrbed in Eq. (4.3) to simplify).

$$\frac{\mathcal{M}}{10S} = \frac{2000mk}{\varepsilon^2 S} \cdot \frac{1}{10S} \leq \frac{200mk}{\varepsilon^2} \cdot \frac{1}{(15\sqrt{mk}/\varepsilon)^2} = \frac{200mk}{\varepsilon^2 S^2},$$

Therefore, with probability at least 9/10, none of the induced subgraphs $G[V_i]$ will contain more than $3S/4$ light edges.

Bounding the Number of Heavy Edges *Received* by a Machine

Next, we turn our attention to the edges that have at least one endpoint in V_{heavy} (we call such edges *heavy*). We will show that for each $v \in V_{heavy} \cap V_i$, the number of edges contributed by v concentrates around its expectation.[5] In this section, we will use $2m_h$ to denote the total degree of all the heavy vertices i.e. $2m_h = \sum_{v \in V_{heavy}} d(v)$.

Let $Z_w^{(v)}$ be the $\{0,1\}$ indicator random variable for $w \in V_i$ conditioned on the event that $v \in V_i \cap V_{heavy}$. We use this conditioning on v being present, because, in its absence, the number of edges contributed by v, can be *zero* with probability $(1-\hat{p})$, i.e. this naive estimator would not concentrate around its expectation.

Let $Z^{(v)}$ be the sum of all $Z_w^{(v)}$ for $w \in N(v)$. For a particular v, the $Z_w^{(v)}$ variables are k-wise independent, which allows us to use the following lemma to bound $Z^{(v)}$. In what follows, we will omit the super-script (v) for the sake of convenience.

Lemma 4.5.2. *If $Z_1, Z_2, \cdots, Z_n$ are k-wise independent $\{0,1\}$ random variables with $\mathbb{E}[Z_i] = p$ and $k \leq np$, then for $Z = \sum_i Z_i$ we have*

$$\mathbb{P}[Z > 3np] \leq 2^{-k}.$$

Proof. To prove the claim, we will re-write $\mathbb{P}[\sum Z_i > 3np]$, as the probability that the number of size k subsets of $\{Z_1, Z_2, \cdots, Z_n\}$ that are all equal to 1 is larger than $\binom{3np}{k}$.

$$\begin{aligned}
&\mathbb{P}[Z > 3np] \\
&= \mathbb{P}\left[|\{T \ : \ T \subseteq [n], |T| = k, \text{ and } Z_i = 1 \ \forall i \in T\}| > \binom{3np}{k}\right] \\
&\leq \frac{\mathbb{E}[|\{T \ : \ T \subseteq [n], |T| = k, \text{ and } Z_i = 1 \ \forall i \in T\}|]}{\binom{3np}{k}} \\
&= \frac{\binom{n}{k} \cdot p^k}{\binom{3np}{k}} \leq \left(\frac{n}{3np-k} \cdot p\right)^k \leq \left(\frac{np}{2np}\right)^k = 2^{-k}
\end{aligned}$$

where to obtain $3np - k \geq 2np$ we used our assumption that $k \leq np$. □

[5] Intuitively, this is because v has high degree, and therefore the number of its sampled neighbors ($|N(v) \cap V_i|$) will concentrate.

Since v is heavy, there are at least τ variables in the sum $Z^{(v)} = \sum_{w \in N(v)} Z_w^{(v)}$. Additionally, we know that $\mathbb{E}[Z_w^{(v)}] = \hat{p}$ and $k \leq \tau \hat{p}$. Thus, we obtain the following corollary from Lemma 4.5.2:

Corollary 4.5.3. *For any vertex $v \in V_{heavy} \cap V_i$, we get $\mathbb{P}\left[Z^{(v)} > 3d(v) \cdot \hat{p}\right] < 2^{-k}$, or explicitly*

$$\mathbb{P}[N(v) \cap V_i > 3d(v)\hat{p} \mid v \in V_i \text{ and } d(v) > \tau] < 2^{-k} = \frac{1}{n^6}$$

Corollary 4.5.4. *With high probability $1 - \frac{1}{n^5}$, we ensure that for all $v \in V_{heavy}$, $Z^{(v)} \leq 3 \cdot \mathbb{E}\left[Z^{(v)}\right]$*

The important point is that the sum of $Z^{(v)}$ (over all $v \in V_i$) is an upper bound on m_h – the number of heavy edges in $G[V_i]$. In order to bound this sum, we define random variables W_v for each $v \in V_{heavy}$ as follows:

$$W_v = \begin{cases} \frac{d(v)}{n} & \text{if } v \in V_i \\ 0 & \text{otherwise} \end{cases}$$

We also define W to be the sum of all W_v, thus implying $\mu = \mathbb{E}[W] = \sum_{v \in V_{heavy}} \hat{p} \cdot \frac{d(v)}{n} \leq \frac{2\hat{p}m_h}{n}$.

Theorem 4.5.5. *(Theorem 5 from [SSS95]) If W is the sum of k-wise independent random variables, each of which takes values in the interval $[0,1]$, and $\delta \geq 1$, then:*

$$k < \left\lfloor \delta \mu e^{-1/3} \right\rfloor \implies \mathbb{P}[|W - \mu| > \delta\mu] \leq e^{\lfloor k/2 \rfloor}$$

Corollary 4.5.6. $\mathbb{P}\left[W > \frac{4\hat{p}mk}{n}\right] \leq e^{-\lfloor k/2 \rfloor}$

Proof. We can use the fact the random variables W_v are k-wise independent to apply Theorem 4.5.5. First, we ensure that $k < \left\lfloor \delta \mu e^{-1/3} \right\rfloor$, that we achieve by setting $\delta = \frac{mk}{m_h}$.

Recall that m_h is the number of heavy edges (ones with at least one heavy end-point), and m is the total number of edges in the original graph G.

$$\delta = \frac{mk}{m_h} \implies \delta \mu e^{-1/3} = \frac{mk \cdot 2\hat{p}m_h}{m_h \cdot n} \cdot e^{-1/3} > \frac{\hat{p}mk}{n} \implies \delta \mu e^{-1/3} > k$$

In the last step, we used the fact that $S > 100kn^2/m$ from Eq. (4.3), to imply that $\hat{p}m/n > 1$. Therefore, we can now apply Theorem 4.5.5 to conclude:

$$\begin{aligned} & \mathbb{P}[|W - \mu| > \delta\mu] \leq e^{-\lfloor k/2 \rfloor} \\ \implies & \mathbb{P}\left[W > \mu + \frac{2\hat{p}mk}{n}\right] \leq e^{-\lfloor k/2 \rfloor} \\ \implies & \mathbb{P}\left[W > \frac{4\hat{p}mk}{n}\right] \leq e^{-\lfloor k/2 \rfloor} \end{aligned}$$

In the second step, we used the fact that $\mu = \mathbb{E}[W] = \sum_{v \in V_{heavy}} \hat{p} \cdot \frac{d(v)}{n} \leq \frac{2m\hat{p}}{n}$. □

Now we are finally ready to upper bound the number of heavy edges in $G[V_i]$. With high probability (using Corollary 4.5.3), the following holds:

$$\begin{aligned}
\#(\text{heavy edges in } G[V_i]) &\leq \sum_{v \in V_{heavy}} \mathbb{P}[v \in V_i] \cdot (3d(v)\hat{p}) \\
&\leq \sum_{v \in V_{heavy}} W_v \cdot n \cdot (3\hat{p}) = 3n\hat{p} \cdot W \\
&\leq 12\hat{p}^2 mk = \frac{12S}{100} < \frac{S}{8}
\end{aligned}$$

Theorem 4.5.7 (Heavy edges)**.** *With high probability, the number of edges in $G[V_i]$ that have some endpoint with degree larger than τ is at most $S/8$.*

Combining this result with Theorem 4.5.7, we conclude the following:

Theorem 4.5.8. *With probability at least $9/10$, the maximum number of edges in any of the $G[V_i]$s (where $i \in [R]$) does not exceed S, and hence Algorithm 5 does not terminate on Line 4.*

Upper-Bounding the Number of Messages *Sent* by any Machine

Recalling Algorithm 6, we note that the number of messages *received* by the machine containing V_i, is equal to the number of edges in $G[V_i]$. Therefore, the last section essentially proved that the number of messages (edges) *received* by a particular machine is upper-bounded by S. Conversely, in this section, we will justify that the number of messages *sent* by any machine is $O(S)$. Since the number of edges stored in a machine is $\leq S$, it suffices to to show that for each edge e, Algorithm 6 sends only $O(1)$ messages (each message is a copy of the edge e).

Let $Z_i^{(e)}$ be the $\{0,1\}$ indicator random variable for $e \in G[V_i]$, and let $Z^{(e)}$ be the sum of $Z_i^{(e)}$ for all $i \in [\mathcal{M}]$. Here, $Z^{(e)}$ represents the number of messages that are created by edge e. Additionally we make $r = S\mathcal{M}/m = O_\varepsilon(\log n)$ copies of each edge e, and ensure that all replicates reside on the same machine. We distribute the $Z^{(e)}$ messages evenly amongst the replicates, so that each replica is only responsible for $Z^{(e)}/r$ messages.

Since all replicates are on the same machine, this last step is purely conceptual, but it will simplify our arguemnt, by allowing us to charge the outgoing messages to each replicate (as opposed to each edge). Our goal will be show that each replicate is responsible for only $O(1)$ messages, which is the same as showing that w.h.p. $Z^{(e)}/r = O(1)$.

Clearly $\mu = \mathbb{E}[Z^{(e)}] = \hat{p}^2 \cdot \mathcal{M} = \frac{S\mathcal{M}}{100mk}$. This allows us to apply Lemma 4.5.2 with $\delta = \frac{100e^{1/3}mk^2}{S\mathcal{M}}$

$$\mathbb{P}\left[Z^{(e)} > \delta\mu\right] \leq e^{-\lfloor k/2 \rfloor} = \frac{1}{n^3} \implies \mathbb{P}\left[\frac{Z^{(e)}}{r} > \frac{e^{1/3}k}{r}\right] \leq \frac{1}{n^3}$$

Using the assumption (from Eq. (4.3)) that $\mathcal{M} > 2000mk/S \implies r > 2000k$, we see that with high probability, the number of messages sent by any replicate is bounded above by $e^{1/3}/2000 \leq 1$. So, the number of messages sent from any machine is bounded by S with high probability.

4.5.2 Showing that the Estimate Concentrates

Showing Concentration for the Triangle Count

Algorithm 5 outputs an estimate on the number of triangles in G (Line 10). It is not hard to show that in expectation this output equals T. The main challenge is to show that this output also concentrates well around its expectation. Specifically, we show the following claim.

Lemma 4.5.9. *Ignore Line 4 of Algorithm 5. Let $\hat{T}$ be as defined on Line 9 and $\mathcal{M} = \frac{20}{\varepsilon^2 \hat{p}^2}$ be as defined in Eq. (4.3), and assume that $T \geq 1/\hat{p}$. Then, the following hold:*

1. $\mathbb{E}\left[\hat{T}\right] = \hat{p}^3 \cdot R \cdot T$, *and*
2. $\mathbb{P}\left[|\hat{T} - \mathbb{E}\left[\hat{T}\right]| > \varepsilon \mathbb{E}\left[\hat{T}\right]\right] < \frac{1}{10}$.

We will prove Property (2) of the claim by applying Chebyshev's inequality, for which we need to compute $\mathrm{Var}\left[\hat{T}\right]$. Let $\Delta(G)$ be the set of all triangles in G. For a triangle $t \in \Delta(G)$, let $\hat{T}_{i,t} = 1$ if $t \in V[G_i]$, and $\hat{T}_{i,t} = 0$ otherwise. Hence, $\hat{T}_i = \sum_{t \in \Delta(G)} \hat{T}_{i,t}$. We begin by deriving $\mathbb{E}\left[\hat{T}\right]$ and then proceed to showing that $\mathrm{Var}\left[\hat{T}\right] = \sum_{i=1}^{R} \mathrm{Var}\left[\hat{T}_i\right]$. After that we upper-bound $\mathrm{Var}\left[\hat{T}_i\right]$ and conclude the proof by applying Chebyshev's inequality.

Deriving $\mathbb{E}\left[\hat{T}\right]$. Let t be a triangle in G. Let $\hat{T}_t$ be a random variable denoting the total number of times t appears in $G[V_i]$, for all $i = 1 \ldots R$. Given that $\mathbb{P}[u \in V_i] = \hat{p}$, we have that $\mathbb{P}[t \in G[V_i]] = \hat{p}^3$. Therefore, $\mathbb{E}\left[\hat{T}_t\right] = R \cdot \hat{p}^3$.

Since $\hat{T} = \sum_{t \in \Delta(G)} \hat{T}_t$, we have

$$\mathbb{E}\left[\hat{T}\right] = \sum_{t \in \Delta(G)} \mathbb{E}\left[\hat{T}_t\right] = \hat{p}^3 \cdot R \cdot T. \tag{4.4}$$

This proves Property (1) of this claim.

Decoupling $\mathrm{Var}\left[\hat{T}\right]$. To compute variance, one considers the second moment of a given random variable. So, to compute $\mathrm{Var}\left[\hat{T}\right]$, we will consider products $\hat{T}_{i,t_1} \cdot \hat{T}_{j,t_2}$. Each of those products depend on at most 6 vertices. Now, given that we used a 6-wise independent function (see Section 4.4.1) to sample vertices in each V_i, one could expect that $\mathrm{Var}\left[\hat{T}_i\right]$ and $\mathrm{Var}\left[\hat{T}_j\right]$ for $i \neq j$ behave like they are independent, i.e., one could expect that

it holds $\mathrm{Var}\left[\hat{T}\right] = \sum_{i=1}^{R} \mathrm{Var}\left[\hat{T}_i\right]$. As we show next, it is indeed the case. We have

$$\begin{aligned}
\mathrm{Var}\left[\hat{T}\right] &= \mathbb{E}\left[\hat{T}^2\right] - \mathbb{E}\left[\hat{T}\right]^2 \\
&= \mathbb{E}\left[\left(\sum_{i=1}^{R}\sum_{t\in\Delta(G)} \hat{T}_{i,t}\right)^2\right] - \left(\sum_{i=1}^{R}\sum_{t\in\Delta(G)} \mathbb{E}\left[\hat{T}_{i,t}\right]\right)^2
\end{aligned} \tag{4.5}$$

Consider now $\hat{T}_{i,t_1}$ and $\hat{T}_{j,t_2}$ for $i \neq j$ and some $t_1, t_2 \in \Delta(G)$ not necessarily distinct. In the first summand of (4.5), we will have $\mathbb{E}\left[2\hat{T}_{i,t_1} \cdot \hat{T}_{j,t_2}\right]$. The vertices constituting t_1 and t_2 are 6 *distinct* copies of some (not necessarily all distinct) vertices of V. Since they are chosen by applying a 6-wise independent function, we have $\mathbb{E}\left[2\hat{T}_{i,t_1} \cdot \hat{T}_{j,t_2}\right] = 2\mathbb{E}\left[\hat{T}_{i,t_1}\right] \cdot \mathbb{E}\left[\hat{T}_{j,t_2}\right]$. On the other hand, the second summand of (4.5) also contains $2\mathbb{E}\left[\hat{T}_{i,t_1}\right] \cdot \mathbb{E}\left[\hat{T}_{j,t_2}\right]$, which follows by direct expansion of the sum. Therefore, all the terms $\mathbb{E}\left[2\hat{T}_{i,t_1} \cdot \hat{T}_{j,t_2}\right]$ in $\mathrm{Var}\left[\hat{T}\right]$ for $i \neq j$ cancel each other. So, we can also write $\mathrm{Var}\left[\hat{T}\right]$ as

$$\begin{aligned}
\mathrm{Var}\left[\hat{T}\right] &= \sum_{i=1}^{R} \mathbb{E}\left[\left(\sum_{t\in\Delta(G)} \hat{T}_{i,t}\right)^2\right] - \sum_{i=1}^{R}\left(\sum_{t\in\Delta(G)} \mathbb{E}\left[\hat{T}_{i,t}\right]\right)^2 \\
&= \sum_{i=1}^{R} \mathrm{Var}\left[\hat{T}_i\right].
\end{aligned} \tag{4.6}$$

Therefore, to upper-bound $\mathrm{Var}\left[\hat{T}\right]$ it suffices to upper-bound $\mathrm{Var}\left[\hat{T}_i\right]$.

Upper-bounding $\mathrm{Var}\left[\hat{T}_i\right]$. We have

$$\begin{aligned}
\mathrm{Var}\left[\hat{T}_i\right] &= \mathbb{E}\left[\left(\sum_{t\in\Delta(G)} \hat{T}_{i,t}\right)^2\right] - \left(\sum_{t\in\Delta(G)} \mathbb{E}\left[\hat{T}_{i,t}\right]\right)^2 \\
&\leq \mathbb{E}\left[\left(\sum_{t\in\Delta(G)} \hat{T}_{i,t}\right)^2\right] \\
&= \mathbb{E}\left[\sum_{t\in\Delta(G)} \hat{T}_{i,t}^2\right] + \mathbb{E}\left[\sum_{t_1,t_2\in\Delta(G);t_1\neq t_2} \hat{T}_{i,t_1} \cdot \hat{T}_{i,t_2}\right].
\end{aligned} \tag{4.7}$$

Since each $\hat{T}_{i,t}$ is a 0/1 random variables, $\hat{T}_{i,t}^2 = \hat{T}_{i,t}$. Let $t_1 \neq t_2$ be two triangles in $\Delta(G)$. Let k be the number of distinct vertices they are consisted of, which implies $4 \leq k \leq 6$. Then, observe that $\mathbb{E}\left[\hat{T}_{i,t_1} \cdot \hat{T}_{i,t_2}\right] = \hat{p}^k \leq \hat{p}^4$. We now have all ingredients to upper-bound

$\mathrm{Var}\left[\hat{T}_i\right]$. From (4.7) and our discussion it follows

$$\mathrm{Var}\left[\hat{T}_i\right] \leq T\hat{p}^3 + T^2\hat{p}^4 \leq 2T^2\hat{p}^4, \tag{4.8}$$

where we used our assumption that $T \geq 1/\hat{p}$.

Finalizing the proof. From (4.6) and (4.8) we have

$$\mathrm{Var}\left[\hat{T}\right] \leq 2RT^2\hat{p}^4.$$

So, from Chebyshev's inequality and (4.4) we derive

$$\begin{aligned}\mathbb{P}\left[|\hat{T} - \mathbb{E}\left[\hat{T}\right]| > \varepsilon\mathbb{E}\left[\hat{T}\right]\right] &< \frac{\mathrm{Var}\left[\hat{T}\right]}{\varepsilon^2\mathbb{E}\left[\hat{T}\right]^2} \\ &\leq \frac{2RT^2\hat{p}^4}{\varepsilon^2\hat{p}^6R^2T^2} \\ &= \frac{2}{\varepsilon^2\hat{p}^2R}.\end{aligned}$$

Hence, for $R \geq \frac{20}{\varepsilon^2\hat{p}^2}$ we get the desired bound.

Getting the High Probability Bound

By building on Lemma 4.5.9 and Algorithm 5, we design Algorithm 7 that outputs an approximate triangle counting with high probability, as opposed with only constant success probability. We have the following guarantee for Algorithm 7.

Algorithm 7 Approximate Triangle Counting

1: **function** Approx-Triangles-Main($G = (V, E)$)
2: Let $I \leftarrow 100 \cdot \log n$.
3: **parfor** $i \leftarrow 1 \dots I$ **do**
4: Let Y_i be the output of Algorithm 5 invoked on G. We assume that each invocation of Algorithm 5 uses fresh randomness compared to previous runs.
5: Let $\mathcal{Y}$ be the list of all Y_i, for $i = 1 \dots I$.
6: Sort $\mathcal{Y}$ in non-decreasing order.
7: **return** the median of $\mathcal{Y}$

Theorem 4.5.10. *Let Y be the output of Algorithm 7. Then, with high probability it holds*

$$|Y - T| \leq \varepsilon T.$$

In the proof of this theorem we use the following concentration bound.

Theorem 4.5.11 (Chernoff bound). *Let $X_1, \ldots, X_k$ be independent random variables taking values in $[0,1]$. Let $X \stackrel{\text{def}}{=} \sum_{i=1}^{k} X_i$ and $\mu \stackrel{\text{def}}{=} \mathbb{E}[X]$. Then, or any $\delta \in [0,1]$ it holds $\mathbb{P}[X \leq (1-\delta)\mu] \leq \exp\left(-\delta^2 \mu/2\right)$.*

Proof of Theorem 4.5.10. The proof of this theorem is essentially the so-called "Median trick". We provide full proof here for completeness.

Let Y_i be as defined on Line 4 of Algorithm 7. By Theorem 4.5.8, with probability at most 1/10 Algorithm 5 terminates due to creating too big subgraphs. If we ignore Line 4 of Algorithm 5, then by Property (1) of Lemma 4.5.9 we have $\mathbb{E}[Y_i] = T$. Y_i significantly deviates from its expectation if Algorithm 5 terminates on Line 4 or if the estimate Y_i is simply off. Define a 0/1 variable Z_i which equals 1 iff $|Y_i - T| \leq \varepsilon T$. By union bound on Property (2) of Lemma 4.5.9 and Theorem 4.5.8, we have $\mathbb{P}[Z_i = 1] \geq 1 - 1/10 - 1/10 = 4/5$. Also, following Line 4 of Algorithm 7 we have that all Z_i are independent.

Let $Z = \sum_{i=1}^{I} Z_i$. We have that $\mathbb{E}[Z] \geq \frac{4}{5} I$, implying that in expectation at least 4/5 fraction on Z-variables are 1. We now bound the probability that at least 2/5 of these variables equal 0, i.e, at most 3/5 of them equal 1. Since Z-variables are independent, for this we can use Theorem 4.5.11, obtaining

$$\mathbb{P}\left[Z \leq \frac{3}{5} I\right] \leq \mathbb{P}\left[Z \leq \left(1 - \frac{1}{5}\right)\mathbb{E}[Z]\right] \leq \exp\left(-\mathbb{E}[Z]/50\right).$$

Given that $I = 100 \cdot \log n$ (see Line 2 of Algorithm 7), we derive that $\mathbb{P}\left[Z \leq \frac{3}{5} I\right] < n^{-1}$. This now implies that with probability at least n^{-1} the output of Algorithm 7 is some Y_j such that $Z_j = 1$. This completes the analysis. □

4.5.3 Showing Concentration for the K-Subgraph Count

Using similar analysis to the previous section, in this section, we show the expectation and concentration bound of our subgraph counting algorithm for any subgraphs consisting of K nodes where K is constant. Let this subgraph be H. In what follows, let B be the actual count of the K-subgraphs and $\hat{B}$ be the estimate. We only require a (slightly) modified Algorithm 5; the rest of the algorithm follows that of Algorithm 6 and Algorithm 7.

Algorithm 8 Approximate-K-Subgraph-Counting($G = (V, E)$)

1: $R \leftarrow 0$
2: **parfor** $i \leftarrow 1 \ldots \mathcal{M}$ **do**
3: Let V_i be a random subset of V ⟨⟨*See Section 4.4.1 for details about the sampling*⟩⟩
4: **if** size of $G[V_i]$ exceeds S **then**
5: Ignore this sample and set $\hat{B}_i \leftarrow 0$
6: **else**
7: Let $\hat{B}_i$ be the number of the desired K-subgraphs in $G[V_i]$
8: $R \leftarrow R + 1$
9: Let $\hat{B} = \sum_{i=1}^{\mathcal{M}} \hat{B}_i$
10: **return** $\frac{1}{p^K R} \hat{B}$

Lemma 4.5.12. *Let $\hat{B}$ be the count of subgraph H (with K vertices) in $G[V_i]$ and $\mathcal{M} = \frac{20}{\varepsilon^2 \hat{p}^{K-1}}$ be as defined in Eq. (4.3), and assume that $B \geq 1/\hat{p}$. Then, the following hold:*

1. $\mathbb{E}\left[\hat{B}\right] = \hat{p}^K \cdot R \cdot B$, *and*
2. $\mathbb{P}\left[|\hat{B} - \mathbb{E}\left[\hat{B}\right]| > \varepsilon \mathbb{E}\left[\hat{B}\right]\right] < \frac{1}{10}$.

Proof. We first prove Item 1. The probability that a particular K-vertex occurrence h of H appears in machine M_i is $\mathbb{P}[h \in G[V_i]] = \hat{p}^K$. There are B number of occurrences of H in G. Thus, the expected number of occurrences of H in machine M_i is $\mathbb{E}\left[\hat{B}_{M_i}\right] = \hat{p}^K \cdot B$. Since there are R machines (which did not exceed the memory limit), the expected number of occurrences of H in all R machines is $\mathbb{E}\left[\hat{B}\right] = \sum_{i \leq R} \mathbb{E}\left[\hat{B}_{M_i}\right] = \hat{p}^K RB$.

We now prove Item 2. Let $H(G)$ be the set of occurrences of H in G. Let $\hat{B}_{i,h}$ be a random variable where $\hat{B}_{i,h} = 1$ if h, a particular occurrence of H in G, is in machine i; otherwise, $\hat{B}_{i,h} = 0$. Then,

$$\begin{aligned} \mathrm{Var}\left[\hat{B}\right] &= \mathbb{E}\left[\hat{B}^2\right] - \mathbb{E}\left[\hat{B}\right]^2 && (4.9) \\ &= \mathbb{E}\left[\left(\sum_{i=1}^{R} \sum_{h \in H(G)} \hat{B}_{i,h}\right)^2\right] - \left(\sum_{i=1}^{R} \sum_{h \in H(G)} \mathbb{E}\left[\hat{B}_{i,h}\right]\right)^2. && (4.10) \end{aligned}$$

First, consider random variables $\hat{B}_{i,h_1}$ and $\hat{B}_{j,h_2}$; for each $i \neq j \in [R]$ and each $h_1, h_2 \in H(G)$, there exists a term in the first summand of Section 4.5.3 containing $\mathbb{E}\left[2\hat{B}_{i,h_1}\hat{B}_{j,h_2}\right]$. The vertices constituting h_1 and h_2 are $2K$ distinct copies of some not necessarily distinct copies of vertices in V. Suppose we use a $2k$-wise independent hash function, then we have $\mathbb{E}\left[2\hat{B}_{i,h_1} \cdot \hat{B}_{j,h_2}\right] = 2\mathbb{E}\left[\hat{B}_{i,h_1}\right] \cdot \mathbb{E}\left[\hat{B}_{j,h_2}\right]$. We see that this term also shows up in the second summand of Section 4.5.3. Hence, the terms cancel for each $i \neq j$ and we can simplify Section 4.5.3 to the following.

$$\mathrm{Var}\left[\hat{B}\right] = \sum_{i=1}^{R} \mathbb{E}\left[\left(\sum_{h \in H(G)} \hat{B}_{i,h}\right)^2\right] - \sum_{i=1}^{R} \left(\sum_{h \in H(G)} \mathbb{E}\left[\hat{B}_{i,h}\right]\right)^2 = \sum_{i=1}^{R} \mathrm{Var}\left[\hat{B}_i\right]. \tag{4.11}$$

Now, what remains is to upper bound $\mathrm{Var}\left[\hat{B}_i\right]$. Using the same approach as in the previous section with the observation that any two distince occurrences h_1 and h_2 must contain $K + 1 \leq k \leq 2K$ distinct vertices. This means that $\mathbb{E}\left[\hat{B}_{i,h_1} \cdot \hat{B}_{i,h_2}\right] = \hat{p}^k \leq \hat{p}^{K+1}$. Then, we can bound

$$\mathrm{Var}\left[\hat{B}_i\right] \leq 2B^2 \hat{p}^{K+1} \tag{4.12}$$

(assuming $B \geq 1/\hat{p}$).

Then, from Section 4.5.3 and Section 4.5.3, we get the bound on the variance to be $\mathrm{Var}\left[\hat{B}\right] \leq 2RB^2 \hat{p}^{K+1}$.

By Chebyshev's inequality and Item 1, we compute

$$\mathbb{P}\left[|\hat{B} - \mathbb{E}\left[\hat{B}\right]| > \varepsilon \mathbb{E}\left[\hat{B}\right]\right] < \frac{\mathrm{Var}\left[\hat{B}\right]}{\varepsilon^2 \mathbb{E}\left[\hat{B}\right]^2} \leq \frac{2B^2\hat{p}^{K+1}}{\varepsilon^2 \hat{p}^{2K} R^2 B^2} = \frac{2}{\varepsilon^2 \hat{p}^{K-1} R^2}. \tag{4.13}$$

When setting $R \geq \frac{20}{\varepsilon^2 \hat{p}^{K-1}}$, we obtain Item 2. □

Using Lemma 4.5.12 and the median trick (used in an identical way to Theorem 4.5.10), we can obtain the following theorem about counting occurrences of *any* subgraph H with K vertices.

Theorem 4.5.13. *Let $G = (V, E)$ be a graph over n vertices, m edges, and let B be the number of occurrrences of subgraph H with K vertices in G. Assuming*

1. $B = \widetilde{\Omega}\left(\left(\frac{m}{S}\right)^{K/2-1}\right)$,
2. $S = \widetilde{\Omega}\left(\max\left\{\frac{\sqrt{m}}{\varepsilon}, \frac{n^2}{m}\right\}\right)$,

there exists an MPC *algorithm, using $\mathcal{M}$ machines, each with local space S, and total space $\mathcal{M}S = \tilde{O}_\varepsilon(m)$, that outputs a $(1 \pm \varepsilon)$-approximation of T, with high probability, in $O(1)$ rounds.*

Corollary 4.5.14. *Let $G = (V, E)$ be an input graph and B be the number of occurrences of subgraph H with K vertices in G. If $B \geq d_{avg}^{K/2-1}$, then there exists an MPC algorithm that in $O(1)$ rounds with high probability outputs a $(1 + \varepsilon)$-approximation of B. This algorithm uses a total space of $\tilde{O}(m)$ and space $\tilde{\Theta}(n)$ per machine. d_{avg} is the average degree of the vertices in the graph.*

4.6 Exact Triangle Counting in $O(m\alpha)$ Total Space

In this section we describe our algorithm for (exactly) counting the number of triangles in graphs $G = (V, E)$ of arboricity α and prove Theorem 4.1.3, restated here, in Section 4.7.

Theorem 4.6.1. *Let $G = (V, E)$ be a graph over n vertices, m edges and arboricity α. Count-Triangles(G) takes $O_\delta(\log\log n)$ rounds, $O\left(n^\delta\right)$ space per machine for some constant $0 < \delta < 1$, and $O(m\alpha)$ total space.*

Importantly, unlike previous methods, we *do not* need to assume knowledge of the arboricity of the graph α as input into our algorithm. The arboricity only shows up in our space bound as a property of the graph but we do not need to have knowledge of its value as we run the algorithm.

In this section, we assume that individual machines have space $\Theta(n^\delta)$ where δ is some constant $0 < \delta < 1$. Given this setting, there are several challenges associated with this problem.

Challenge 4.6.2. *The entire subgraph neighborhood of a vertex may not fit on a single machine. This means that all triangles incident to a particular vertex cannot be counted on*

one machine. Even if we are considering vertices with degree at most α, it is possible that $\alpha > n^{\delta}$. Thus, we need to have a way to count triangles efficiently when the neighborhood of a vertex is spread across multiple machines.

The second challenge is to avoid over-counting.

Challenge 4.6.3. *When counting triangles across different machines, over-counting the triangles might occur, e.g., if two different machines count the same triangle. We need some way to deal with duplicate counting of the triangles to obtain the exact count of the triangles.*

We deal with the above challenges in our procedures below. We assume in our algorithm that each vertex can access its neighbors in $O(1)$ rounds of communication; such can be ensured via standard MPC techniques. Let $d_Q(v)$ be the degree of v in the subgraph induced by vertex set Q, i.e. in $G[Q]$. Our main algorithm consists of the following Count-Triangles(G) procedure.

Algorithm 9 Count-Triangles($G = (V, E)$)

Let Q_i be the set of vertices not yet processed by iteration i. Initially set $Q_0 \leftarrow V$.
Let T be the current count of triangles. Set $T \leftarrow 0$.
for $i = 0$ to $i = \left\lceil \log_{3/2}(\log_2(n)) \right\rceil$ **do**
    $\gamma_i \leftarrow 2^{(3/2)^i}$.
    Let A_i be the list of vertices $v \in Q_i$ where $d_{Q_i}(v) \leq \gamma_i$. Set $Q_{i+1} \leftarrow Q_i \setminus A_i$.
    parfor $v \in A_i$ **do**
        Retrieve the list of neighbors of v and denote it by L_v.
        Send each of v's neighbors a copy of L_v.
    parfor $w \in Q_i$ **do**
        Let $\mathcal{L}_w = \bigcup_{v \in (N(w) \cap A_i)} L_v$ be the union of neighbor lists received by w.
        Set $T \leftarrow T +$ Find-Triangles($w, \mathcal{L}_w$). ⟨⟨*Algorithm 10*⟩⟩
Return T.

4.6.1 MPC Implementation Details

In order to implement Count-Triangles(G) in the MPC model, we define our Find-Triangles($w, \mathcal{L}$) procedure and provide additional details on sending and storing neighbor lists across different machines. We define *high-degree* vertices to be the set of vertices whose degree is $> \gamma$ and *low-degree* vertices to be ones whose degree is $\leq \gamma$ (for some γ defined in our algorithm). We now define the function Find-Triangles($w, \mathcal{L}$) used in the above procedure:

Allocating machines for sorting Since each $v \in Q_i$ could have multiple neighbors whose degrees are $\leq \gamma$, the total size of all neighbor lists v receives could exceed their allowed space $\Theta\left(n^{\delta}\right)$. Thus, we allocate $O\left(\frac{\gamma d_{Q_i}(v)}{n^{\delta}}\right)$ machines for each vertex $v \in Q_i$ to store all neighbor lists that v receives.

Algorithm 10 Find-Triangles$(w, \mathcal{L}_w)$

1: Sort all elements in $(\mathcal{L}_w \cup (N(w) \cap Q_i))$ lexicographically, using the procedure given in Lemma 4.3 of [GSZ11]. Let this sorted list of all elements be S.
2: Let T denote the corrected[6] number of duplicates in S using Theorem 4.2.1.
3: Return T.

The complete analysis for Theorem 4.6.1 is given in Section 4.7.

We provide two additional extensions of our triangle counting algorithm to counting k-cliques:

Theorem 4.6.4. *Given a graph $G = (V, E)$ with arboricity α, we can count all k-cliques in $O(m\alpha^{k-2})$ total space, $O_\delta(\log\log n)$ rounds, on machines with $O(n^{2\delta})$ space for any $0 < \delta < 1$.*

We can prove a stronger result when we have some bound on the arboricity of our input graph. Namely, if $\alpha = O(n^{\delta'/2})$ for any $\delta' < \delta$, then we obtain the following result:

Theorem 4.6.5. *Given a graph $G = (V, E)$ with arboricity α where $\alpha = O(n^{\frac{\delta'}{2}})$ for any $\delta' < \delta$, we can count all k-cliques in $O\left(n\alpha^2\right)$ total space and $O_\delta(\log\log n)$ rounds, on machines with $O(n^\delta)$ space for any $0 < \delta < 1$.*

The proofs of these theorems are provided in Section 4.8.

4.7 Exact Triangle Counting in $O(m\alpha)$ Total Space Analysis

In this section we give the full details and analysis of algorithm Algorithm 9, , given in Section 4.6, for exactly counting the number of triangles in the graph.

We first provide a detailed version of Algorithm 10 that also takes into account over counting due to the fact that each triangle might be counted by several endpoints, and then continue to prove the main theorem of this section, Theorem 4.1.3.

4.7.1 Details about finding duplicate elements using Theorem 4.2.1

Find-Triangles$(w, \mathcal{L}_w)$ finds triangles by counting the number of duplicates that occur between elements in lists. Theorem 4.2.1 provides a MPC implementation for finding the count of all occurrences of every element in a sorted list. Provided a sorted list of neighbors of $v \in Q_i$ and neighbor lists in $\mathcal{L}_v$, this function counts the number of intersections between a neighbor list sent to v and the neighbors of v. Every intersection indicates the existence of a triangle. As given, Find-Triangles$(w, \mathcal{L}_w)$ (see v Algorithm 10) returns a 6-approximation of the number of triangles in any graph. We provide a detailed and somewhat more complicated algorithm Find-Triangles-Exact$(w, \mathcal{L}_w)$ that accounts for over-counting of triangles and returns the exact number of triangles.

Since Theorem 4.2.1 returns the total count of each element, we subtract the value returned by 1 to obtain the number of intersections. Finally, each triangle containing one low-degree vertex will be counted twice, each containing two low-degree vertices will be counted 4 times, and each containing three low-degree vertices will be counted 6 times. Thus, we need to divide the counts by 2, 4, and 6, respectively, to obtain the exact count of unique triangles.

Algorithm 11 Find-Triangles-Exact($w, \mathcal{L}_w$)

Set the number of triangles $T_i \leftarrow 0$.
Sort all elements in $(\mathcal{L}_w \cup (N(w) \cap Q_i))$ lexicographically using the procedure given in Lemma 4.3 of [GSZ11]. Let this sorted list of all elements be S.
Count the duplicates in S using Theorem 4.2.1.
parfor all $v \in N(w)$ **do**
    Let R be the number of duplicates of v returned by Theorem 4.2.1.
    if $d_{Q_i}(v) > \gamma_i$ and $d_{Q_i}(w) > \gamma_i$ **then**
        Increment $T_i \leftarrow T_i + \frac{R-1}{2}$.
    else if $(d_{Q_i}(v) > \gamma_i$ and $d_{Q_i}(w) \leq \gamma_i)$ or $(d_{Q_i}(v) \leq \gamma_i$ and $d_{Q_i}(w) > \gamma_i)$ **then**
        Increment $T_i \leftarrow T_i + \frac{R-1}{4}$.
    else
        Increment $T_i \leftarrow T_i + \frac{R-1}{6}$.
Return T_i.

Substituting Find-Triangles-Exact in Count-Triangles finds the exact count of triangles in graphs with arboricity α using $O(m\alpha)$ total space.

4.7.2 Proof of Theorem 4.1.3

First, all proofs below assume we start at a cutoff of $\gamma = 4\alpha$. Because we increase the cutoff bound doubly exponentially, we can reach such a bound in $O(\log\log\alpha)$ rounds. Thus, in the following proofs, we ignore all rounds before we get to a round where $\gamma \geq 4\alpha$. Before proving the theorem, we provide several useful lemmas stating that the number of vertices and edges remaining at the beginning of each iteration is bounded.

Lemma 4.7.1. *At the beginning of iteration i of Count-Triangles, given $\gamma_i = 2^{(3/2)^i} \cdot (2\alpha)$ as stated in Algorithm 9, the number of remaining vertices $N_i = |Q_i|$ is at most $\frac{n}{2^{2\cdot((3/2)^i-1)}}$.*

Proof. Let N_i be the number of vertices in Q_i at the beginning of iteration i. Since the subgraph induced by Q_i must have arboricity bounded by α, we can bound the total degree of Q_i,

$$\sum_{v \in Q_i} d_{Q_i}(v) < 2\alpha|Q_i| = 2N_i\alpha.$$

At the end of the iteration, we only keep the vertices in $Q_{i+1} = \{v \in Q_i \mid d_{Q_i}(v) > \gamma_i\}$.

If we assume that $|Q_{i+1}| > \frac{N_i}{\gamma_i/(2\alpha)}$, then we obtain a contradiction since this implies that

$$\sum_{v \in Q_{i+1}} d_{Q_i}(v) > |Q_{i+1}| \cdot \gamma_i > 2N_i\alpha > \sum_{v \in Q_i} d_{Q_i}(v).$$

Then, the number of remaining vertices follows directly from the above by induction on i with base case $N_1 = n$,

$$N_i \leq \frac{N_{i-1}}{\gamma_i/(2\alpha)} = \frac{N_{i-1}}{2^{(3/2)^{i-1}}} \leq \frac{n}{\prod_{j=0}^{i-1} 2^{(3/2)^j}} = \frac{n}{2^{2\cdot\left((3/2)^i - 1\right)}}.$$

□

We can show a similar statement for the number of edges that remain at the start of the i^{th} iteration.

Lemma 4.7.2. *At the beginning of iteration i of Count-Triangles, given γ_i, the number of remaining edges m_i is at most $m_i \leq \frac{m}{2^{2\cdot((3/2)^{i-1}-1)}}$.*

Proof. The number of vertices remaining at the beginning of iteration i is given by $|Q_i|$. Thus, because the arboricity of our graph is α, we can upper bound m_i by

$$m_i \leq |Q_i|\alpha.$$

Then, we can also lower bound the number of edges at the beginning of iteration $i-1$ since the vertices that remain at the beginning of round i are ones which have greater than γ_{i-1} degree,

$$m_{i-1} \geq \frac{1}{2} \sum_{v \in Q_{i-1}} d_{Q_{i-1}}(v) \geq \frac{1}{2}|Q_i|\gamma_{i-1}.$$

Thus, we conclude that $m_i \leq \frac{2\alpha m_{i-1}}{\gamma_{i-1}}$. By induction on i with base case $m_0 = m$, we obtain,

$$m_i \leq 2\alpha\left(\frac{m_{i-1}}{\gamma_{i-1}}\right) \leq \frac{m}{\prod_{j=0}^{i-2} 2^{(3/2)^j}} = \frac{m}{2^{2\cdot((3/2)^{i-1}-1)}}.$$

□

The above lemmas allows us to bound the total space used by the algorithm.

Lemma 4.7.3. *Count-Triangles(G) uses $O(m\alpha)$ total space when run on a graph G with arboricity α.*

Proof. The total space the algorithm requires is the sum of the space necessary for storing the neighbor lists sent by all vertices with degree $\leq \gamma_i$ and the space necessary for all vertices to store their own neighbor lists. The total space necessary for each vertex to store its own neighbor list is $O(m)$.

Now we compute the total space used by the algorithm during iteration i. The number of vertices in Q_i at the beginning of this iteration is at most $N_i \leq \frac{n}{2^{2\cdot\left((3/2)^i-1\right)}}$ by Lemma 4.7.1. Each vertex v with $d_{Q_i}(v) \leq \gamma_i$, makes $d_{Q_i}(v)$ copies of its neighbor list $(N(v) \cap Q_i)$ and sends each neighbor in $N(v) \cap Q_i$ a copy of the list. Thus, the total space required by the messages sent by v is $d_{Q_i}(v)^2 \leq \gamma_i^2$. v sends at most one message of size $d_{Q_i}(v) \leq \gamma_i$ along each edge (v, w) for $w \in N(v) \cap Q_i$. Then, by Lemma 4.7.2 the total space required by all the low-degree vertices in round i is at most (as at most two messages are sent along each edge):

$$2m_i \cdot \gamma_i < \frac{m}{2^{2\cdot\left((3/2)^{i-1}-1\right)}} \cdot \left[2^{(3/2)^i}(2\alpha)\right] = 16m\alpha.$$

□

We are now ready to prove Theorem 4.1.3.

Proof of Theorem 4.1.3. By Lemma 4.7.1, the number of vertices remaining in Q_i at the beginning of iteration i is $\frac{n}{2^{2\cdot\left((3/2)^i-1\right)}}$. This means that the procedure runs for $O(\log\log n)$ iterations before there will be no vertices. For each of the $O(\log\log n)$ iterations, Count-Triangles(G) uses $O_\delta(1)$ rounds of communication for the low-degree vertices to send their neighbor lists to their neighbors. The algorithm then calls Find-Triangles-Exact$(w, \mathcal{L}_w)$ on each vertex $w \in Q_i$ (in parallel) to find the number of triangles incident to w and vertices in $A_i \subseteq Q_i$. Find-Triangles-Exact$(w, \mathcal{L}_w)$ requires $O\left(\log_{n^\delta}(m\alpha)\right) = O(1/\delta)$ rounds by Lemma 4.3 of [GSZ11] and Theorem 4.2.1. Therefore, the total number of rounds required by Count-Triangles(G) is $O\left(\frac{\log\log n}{\delta}\right) = O_\delta(\log\log n)$. □

4.8 Extensions to Exact k-Clique Counting in Graphs with Arboricity α

In this section, we briefly provide two algorithms for exact counting of k-cliques (where k is constant) in graphs with arboricity α. The first is an extension of our exact triangle counting result given in Section 4.6. The second is a query-based algorithm where the neighborbood of a low-degree vertex is constructed on a single machine via edge queries. In this case, the triangles incident to any given low-degree vertex can be counted on the same machine.

4.8.1 Exact k-Clique Counting

Exact k-Clique Counting in $O\left(m\alpha^{k-2}\right)$ Total Space and $O_\delta(\log\log n)$ Rounds We extend our algorithm given in Section 4.6 to exactly count k-cliques (where k is constant) in $O\left(m\alpha^{k-2}\right)$ total space and $O_\delta(\log\log n)$ rounds. Given a graph $G = (V, E)$ with arboricity α, the idea behind the algorithm is the following: let $G_i = (V_i \cup V, E_k \cup E)$ be a graph where each vertex $v \in V_i$ corresponds to an i-clique in G. Let $K(u)$ denote the $K_i \in G$ represented by $u \in V_k$. An edge (u, v) exists in E_i iff $u \in V_i$, $v \in V$ and $K(u) \cup \{v\}$ is

an $(i+1)$-clique in G. We construct the G_k graphs iteratively, starting with $G_1 = G$. Then, given G_{i-1}, we recursively construct G_i by using our exact triangle counting algorithm. Once we have G_{k-2}, we obtain our final count of the number of k-cliques by running our exact triangle counting algorithm one last time. The total space used is dominated by running the triangle counting algorithm on G_{k-2}, which uses $O\left(m\alpha^{k-2}\right)$ total space. Since we run the triangle counting algorithm $O(k)$ times and k is a constant, the total number of rounds of communication necessary is $O_\delta(\log\log n)$ rounds. This detailed algorithm is given below.

Below, we describe our $O(n\alpha^{k-1})$ total space, $O(\log\log n)$ rounds exact k-clique counting algorithm that can be run on machines with space $O(n^\delta)$. Calling Count-k-Cliques(G,k,k) for any given graph $G=(V,E)$ returns the number of k-cliques in G.

Algorithm 12 ***k*-Clique-Counting$(G=(V,E),k,k')$**

if $k \leq 1$ **then**
  Return $(|N|, G)$
else
  $(x, G_{k-1}) \leftarrow$ Count-k-Cliques$(G, k-1, k')$
  $T \leftarrow$ Enumerate-Triangles(G_{k-1}). Let T be the set of all enumerated triangles.
  Initialize sets $V_k \leftarrow \varnothing$ and $E_k \leftarrow \varnothing$.
  parfor $t \in T$ **do**
    Let $K(t)$ represent the set of vertices in V composing the clique represented by $t \in T$.
    parfor $v \in K(t)$ **do**
      Let $v'(S)$ be a vertex v representing a set of vertices S. In other words, $K(v) = K(v'(S)) = S$.
      $V_k \leftarrow V_k \cup v'(K(t) \setminus v)$.
      $E_k \leftarrow E_k \cup (v, v'(K(t) \setminus v))$.
  if $k = k' - 2$ **then**
    Return $|T|$.
  else
    Return $(|V_k|, G_k(V \cup V_k, E \cup E_k))$.

4.8.2 MPC Implementation

To implement Count-k-Cliques in the MPC model, we must be able to create the graph $G_2, \ldots, G_{k-1}$ efficiently in our given space and rounds. The crux of this algorithm is the procedure for enumerating all triangles given a set A of vertices in G where $d(v) \leq \gamma$ for all $v \in A$. To do the triangle enumeration, we prove Lemma 4.8.1 which can enumerate all such triangles incident to A in $O(m\gamma)$ total space, $O_\delta(1)$ rounds given machines with space $O\left(n^{2\delta}\right)$.

Algorithm 13 Triangle-Enumeration($G = (V, E)$)

1: Let the set of enumerated triangles to be $T \leftarrow \varnothing$.
2: Let Q_i be the set of vertices that have not yet been processed by iteration i. Initially set $Q_0 \leftarrow V$.
3: **parfor** $i = 0$ to $i = \left\lceil \log_{3/2}(\log_2(n)) \right\rceil$ **do**
4: $\quad \gamma_i \leftarrow 2^{(3/2)^i} \cdot 2\alpha$.
5: $\quad$ Let A_i be the list of vertices $v \in Q_i$ where $d_{Q_i}(v) \leq \gamma_i$. Set $Q_{i+1} \leftarrow Q_i \setminus A_i$.
6: $\quad$ Use Lemma 4.8.1 to enumerate the set of triangles incident to A_i. Let this set be T_i.
7: $\quad T \leftarrow T \cup T_i$.
8: Return T.

Lemma 4.8.1. *Given a graph G, a constant integer $k \geq 2$, and a subset $A \subseteq G$ of vertices such that for every $v \in A$, $d(v) \leq \gamma$, we can generate all triangles in G that are incident to vertices in A in $O_\delta(1)$ rounds, $O(n^{2\delta})$ space per machine, and $O(m\gamma)$ total space.*

Proof. Let R be the set of machines holding the edges incident to A. Here too, similarly to the proof of Lemma 4.9.4, it will be easier to think of each machine M as a set of n^δ parts, so that each edge, incident to a vertex in A, resides on a single part. We duplicate each such part, holding some neighbor of A, α times, using Lemma 4.2.3. (We will actually duplicate machines, but, again, think of the duplicated machines as a collection of duplicated parts.) By Lemma 4.2.3, this takes $O(\log_{n^\delta} \alpha) = O_\delta(1)$ rounds. Fix some vertex $v \in A$ and assume that $u \in N(v)$ resides on part $P_i(v)$. After the duplication step, there are α copies of each part. We denote these copies $P_{i,1}(v), \ldots, P_{i,\alpha}(v)$. All parts $P_{i,j}(v)$ where $j \in [\alpha]$ and $v \in A$ then asks for v's i-th neighbor in $O(1)$ rounds of communication. Now, each part $P_{i,j}(v)$ creates $O(1)$ edge queries to check whether its vertices form a triangle. All of the queries generated by all parts can be answered in parallel using Lemma 4.2.2 in $O_\delta(1)$ rounds. Then each part that discovered a triangle incident to v adds it to a list K. Now we sort the list K and remove any duplicated triangles, so that the list only holds a single copy of every clique incident to some vertex in A. The total round complexity is $O_\delta(1)$ due to the duplications, sorting, and answering the queries. The space per machine is $O(n^{2\delta})$ and the total memory is $O(m\alpha)$ as each machine was duplicated α times. □

Using Lemma 4.8.1, we can now prove the space usage and round complexity of Enumerate-Triangles.

Lemma 4.8.2. *Given a graph $G = (V, E)$ with arboricity α, Enumerate-Triangles(G) uses $O(n\alpha^2)$ total space, $O_\delta(\log \log n)$ rounds on machines with $O(n^{2\delta})$ space.*

Proof. By Lemma 4.7.1, the number of vertices remaining in Q_i at the beginning of the i-th iteration of Enumerate-Triangles is at most $\frac{n}{2^{2\cdot\left((3/2)^i-1\right)}}$. By Lemma 4.8.1, the total space usage of enumerating all triangles incident to A_i is $O(m\gamma_i) = O\left(m \cdot \left(2^{(3/2)^i} \cdot 2\alpha\right)\right)$.

The summation of the space used for all i is then:

$$\sum_{i=0}^{\lceil \log_{3/2}(\log_2(n)) \rceil} \left(\frac{n}{2^{2\cdot((3/2)^i - 1)}} \right) \cdot \left(2^{(3/2)^i} \cdot 2\alpha^2 \right) = O(n\alpha^2).$$

The number of rounds required by this algorithm is $O(\log\log n) \cdot O_\delta(1) = O_\delta(\log\log n)$. □

Given the total space usage and number of rounds required by Enumerate-Triangles, we can now prove the total space usage and number of rounds required by Count-k-Cliques. But first, we show that for any graph $G = (V, E)$ with arboricity α, all graphs $G_1, \ldots, G_{k-1}$ created by Count-k-Cliques has arboricity $O(\alpha)$ for constant k.

Lemma 4.8.3. *Given a graph $G = (V, E)$ with arboricity α as input to Enumerate-Triangles, all graphs $G_1, \ldots, G_{k-1}$ generated by the procedure have arboricity $O(\alpha)$ for constant k.*

Proof. We prove this lemma via induction. In the base case, $G_1 = G$ and so G_1 has arboricity α. Now we assume that G_i for $i \in [k-1]$ has arboricity $O(\alpha)$ (for constant i) and show that G_{i+1} has $O(\alpha)$ arboricity. Suppose that G_i has arboricity $c\alpha$ for some constant c. We prove via contradiction that the arboricity of G_{i+1} is upper bounded by $3(i+1)c\alpha$. Suppose for the sake of contradiction that the arboricity of G_{i+1} is greater than $3(i+1)c\alpha$. Then, there must exist a subgraph, $G_{i+1}[V']$ for some vertex set, V', of G_{i+1} that contains greater than $3(i+1)c\alpha|V'|$ edges (by definition of arboricity). We now convert this subgraph $G_{i+1}[V']$ to a subgraph in G_i. Every vertex in V' maps to at most i pairs of vertices in G_i connected by an edge. Every edge in $G_{i+1}[V']$ maps to at least 1 edge. Thus, the subgraph in G_i that $G_{i+1}[V']$ maps to contains at most $2i|V'|$ vertices and at least $3(i+1)c\alpha|V'|$ edges. This implies, by the definition of arboricity, that the arboricity of G_i is $\geq \frac{3(i+1)c\alpha|V'|}{2i|V'|} > c\alpha$, a contradiction. Hence, the arboricity of G_{i+1} is at most $3(i+1)c\alpha$. And we have proven that the arboricity of G_{i+1} is $O(\alpha)$ for constant k. By induction, all graphs $G_1, \ldots, G_{k-1}$ have arboricity $O(\alpha)$. □

Now we prove our final theorem of the space and round complexity of Count-k-Cliques.

Proof of Theorem 4.6.4. The number of i-cliques in a graph with arboricity α is at most $O(m\alpha^{i-2})$. Thus, by Lemma 4.8.2 and Lemma 4.8.3, Count-k-Cliques during the i-th call uses $O(m\alpha^i)$ total space, $O_\delta(\log\log n)$ rounds. Thus, Count-k-Cliques uses $O(m\alpha^{k-2})$ space, $O_\delta(\log\log n)$ rounds given machines with $O(n^{2\delta})$ space to count k-cliques given that the procedure terminates on the $(k-2)$-th iteration. □

4.8.3 Exact k-Clique Counting in $O\left(n\alpha^2\right)$ Total Space and $O_\delta(\log\log n)$ Rounds

We can improve on the total space usage if we are given machines where the memory for each individual machine satisfies $\alpha < n^{\delta'/2}$ where $\delta' < \delta$. In this case, we obtain

an algorithm that counts the number of k-cliques in G using $O(n\alpha^2)$ total space and $O_\delta(\log\log n)$ communication rounds.

The entire neighborhood of any vertex with degree $\leq n^{\delta/2}$ can fit on one machine. Suppose that $\alpha < n^{\delta'/2}$ where $\delta' < \delta$, then, there will always exist vertices that have degree $\leq n^{\delta/2}$. Our algorithm proceeds as follows:

Algorithm 14 Count-Cliques$(G = (V, E))$

1: Let Q_i be the set of vertices that have not yet been processed by iteration i. Initially set $Q_0 \leftarrow V$.
2: Let C be the current count of cliques. Set $C \leftarrow 0$.
3: **for** $i = 0$ to $i = \left\lceil \log_{3/2}(\log_2(n)) \right\rceil$ **do**
4: $\quad \gamma_i \leftarrow 2^{(3/2)^i} \cdot 2\alpha$.
5: $\quad$ Let A_i be the list of vertices where $d_{Q_i}(v) \leq \min(cn^{\delta/2}, \gamma_i)$ for some constant c.
6: $\quad$ Set $Q_{i+1} \leftarrow Q_i \setminus A_i$.
7: $\quad$ **parfor** $v \in A_i$ **do**
8: $\quad\quad$ Retrieve all neighbors of v. Let this list of v's neighbors be L_v.
9: $\quad\quad$ Query for all pairs $u, v \in L_v$ to determine whether edge (u, v) exist. Retrieve all edges that exist.
10: $\quad\quad$ Count the number of triangles T_v incident to v, accounting for duplicates.
11: $\quad\quad$ $T \leftarrow T + T_v$.

4.8.4 MPC Implementation Details

Accounting for Duplicates We account for duplicates by counting for each iteration i how many triangles on each machine contains 1, 2 or 3 vertices which have degree $\leq \min(cn^{\delta/2}, \gamma_i)$ (again we call these vertices low-degree). We multiply the count of triangles which have $t \geq 2$ low-degree vertices by $\frac{1}{t}$ to correct for over-counting due to multiple low-degree vertices performing the count on the same triangle. Each machine can retrieve the degrees of vertices in it in $O_\delta(1)$ rounds and such information can be stored on the machine given sufficiently small constant c in Count-Clique.

Proof of Theorem 4.6.5. Since we are considering vertices with degree at most $\min(cn^{\delta/2}, \gamma_i)$, by Lemma 4.7.3, the total space used by our algorithm during any iteration i is

$$N_i \cdot \left(\min\left(cn^{\delta/2}, \gamma_i\right)\right)^2 < 16n\alpha^2.$$

By Lemma 4.2.2, we query for whether each of the $\min\left(cn^{\delta/2}, \gamma_i\right)^2$ potential edges on each machine is an edge in G in parallel using $O(n\alpha^2)$ total space and $O_\delta(1)$ rounds.

If $\gamma_i < cn^{\delta/2}$ for all iterations i, then by Theorem 4.6.1, the number of communication rounds required by Count-Cliques is $O_\delta(\log\log n)$. If, on the other hand, $cn^{\delta/2} < \gamma_i$, then the number of vertices remaining in Q_i decreases by a factor of $cn^{\delta/2}$ every round. Thus, the number of rounds required in this case is $O\left(\frac{2+\delta'}{\delta}\right)$. Since we assume

δ' and δ are constants, the number of communication rounds needed by this algorithm is $O_\delta(\log\log n)$. □

4.9 Counting Subgraphs of Size at Most 5 in Bounded Arboricity Graphs

In this section, we present a procedure that for every subgraph H such that $|H| \leq 5$, counts the exact number of occurrences of H in G in $O(\sqrt{\log n})$ rounds and $O(m\alpha^3)$ total memory, where as before, α is an upper bound on the arboricity of G [7]. The procedure is based on a recent paper by Bera, Pashanasangi and Seshadhri [BPS20] (henceforth BPS) which presented an $O(m\alpha^3)$ time and space algorithm for the same task in the sequential model. We will start by a short description of the BPS result, and then continue to explain how to implement it in the MPC model.

4.9.1 The BPS algorithm

BPS generalize the ideas of Chiba and Nisheziki [CN85] for counting constant-size-cliques and 4-cycles in the classical sequential model to counting all subgraphs of up to 5 nodes in $O(n + m\alpha^3)$ time. Let H be the subgraph in question. The main idea of BPS is as follows. The algorithm starts by computing a degeneracy ordering of G, which is an acyclic orientation of G, denoted $\overrightarrow{G}$, where each vertex has at most $O(\alpha)$ outgoing neighbors. The idea is then to consider all acyclic orientations of H (up to isomorphisms), and for each such acyclic orientation $\overrightarrow{H}$, count the number of occurrences of $\overrightarrow{H}$ in $\overrightarrow{G}$, as described next. The algorithm computes what is referred to as a largest directed rooted tree subgraph (DRTS) of $\overrightarrow{H}$, denoted $\overrightarrow{T}$. That is, the DRTS $\overrightarrow{T}$ is a largest (in number of vertices) tree that is contained in $\overrightarrow{H}$ such that all of the edges are directed away from the root of $\overrightarrow{T}$. Given a DRTS $\overrightarrow{T}$, proceed by looking for all copies of $\overrightarrow{T}$ in $\overrightarrow{G}$. Once a copy of $\overrightarrow{T}$ is found, it needs to be verified whether it can be extended to a copy of $\overrightarrow{H}$ in $\overrightarrow{G}$. This verification is based on the observation that for any directed subgraph $\overrightarrow{H}$ on at most 5 vertices, and for every largest directed rooted tree $\overrightarrow{T}$ of $\overrightarrow{H}$, the complement of $\overrightarrow{T}$ in $\overrightarrow{H}$ is a collection of rooted paths and stars[8]. Therefore, all potential completions of a copy of $\overrightarrow{T}$ to $\overrightarrow{H}$ in $\overrightarrow{G}$ can be computed and hashed in time $O(m \cdot poly(\alpha))$. See figure below for an illustration of a possible $\overrightarrow{H}$ and its DRTS $\overrightarrow{T}$ (adapted from [BPS20]). Hence, whenever a copy of $\overrightarrow{T}$ is discovered in $\overrightarrow{G}$, it can be verified in $O(1/\delta)$ rounds whether this copy can be extended to $\overrightarrow{H}$. Since all copies of $\overrightarrow{T}$ can be enumerated in $O(m\alpha^3)$ time, the overall algorithm takes $O(m\alpha^3)$ time.

[7] Strictly speaking, we will have $\alpha \leq 5\alpha(G)$ but as this does not affect the asymptotic bounds, it is easier to just relate to it as the exact arboricity.

[8] This does not hold for subgraphs H that are stars, but stars can be dealt with differently.

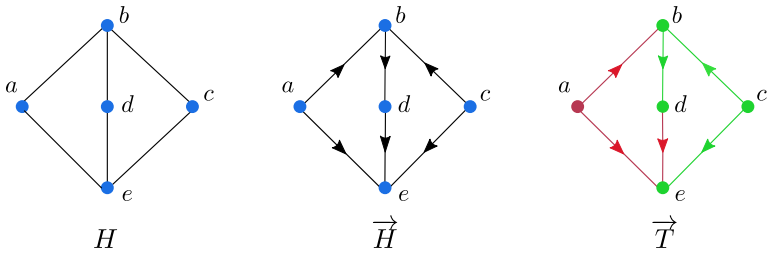


Figure 4-1: From left to right: A subgraph H; a possible directed copy of H; the DRTS in green, and its complement with respect to H in red. Based on a figure from BPS [BPS20].

4.9.2 Implementation in the MPC model

Notation 4.9.1 (Outgoing neighbors and out-degree). *Let $\overrightarrow{G} = (V, \overrightarrow{E})$ be a directed graph. For a vertex $v \in V$, We denote by $N^+(v)$ its set of* outgoing neighbors, *and by $d^+(v) = |N^+(v)|$ its* outgoing degree *or* out-degree.

Definition 4.9.2 (Degeneracy and degeneracy ordering.). *A* degeneracy ordering *of a graph G, is an ordering obtained by repeatedly removing a minimum degree vertex and all the edges incident to this vertex. A vertex u precedes a vertex v in this ordering, $u \prec v$, if u was removed before v. The* degeneracy *of a graph G is then the maximum outgoing degree over all vertices in a degeneracy ordering of G.*

Theorem 4.9.3 (Thm 2 in [GLM19].). *Given a graph G with arboricity α, it outputs, with high probability, an orientation of G, $\overrightarrow{G}$, where each vertex in $\overrightarrow{G}$ has out-degree at most $O(\alpha)$. The algorithm performs $O(\sqrt{\log n} \cdot \log\log n)$ rounds, uses $\widetilde{O}(n^\delta)$ space per machine, for an arbitrary constant $\delta \in (0,1)$, and the total memory is $O(\max\{m, n^{1+\delta}\})$.*

The following is a key lemma.

Lemma 4.9.4. *Let $\overrightarrow{G}$ be a directed graph over m edges such that each vertex has out-degree at most α. Let $\overrightarrow{T}$ be a directed rooted tree of size $t \geq 2$. We can list all copies of $\overrightarrow{T}$ in G in $O(1/\delta)$ rounds, $O(n^{2\delta})$ space per machine, and $O(m \cdot \alpha^{t-2})$ total memory.*

Proof. Let $a_1, \ldots, a_t$ denote the vertices of $\overrightarrow{T}$, where a_1 is the root, and a_i is the i^{th} vertex with respect to the BFS ordering of $\overrightarrow{T}$. Let $\overrightarrow{T}_i$ denote $\overrightarrow{T}[\{a_0, \ldots, a_i\}]$.

We prove the claim by induction on t. For $t = 2$, all edges in G are copies of $\overrightarrow{T}$, so the claim holds trivially.

Assume that the claim holds for i, and we now prove it for $i+1$. By the assumption, in $O(1/\delta)$ rounds and $O(m\alpha^{i-2})$ total memory, all copies of $\overrightarrow{T}_i$ can be listed. We will show that we can use these copies to find all copies of $\overrightarrow{T}_{i+1}$ in $O(1/\delta)$ rounds and $O(m\alpha^{i-1})$ memory. Recall that we have machines with $O(n^{2\delta})$ memory. We will divide the copies among the machines, so that each machine only holds $O(n^\delta)$ copies. Let M be some machine containing copies $\tau_1, \ldots, \tau_{n^\delta}$ of $\overrightarrow{T}_i$. It will be easier to think of M as a collection of n^δ constant memory parts, each holding a single copy of $\overrightarrow{T}_i$. Consider a specific copy τ

of $\overrightarrow{T}_i$ and let P_τ denote the part storing that copy. Let a_p denote the vertex in $\overrightarrow{T}$ that is the parent of a_{i+1}, and let u denote the vertex in τ that is mapped to a_p. We would like to create all tuples (τ, w), where $w \in N^+(u) \setminus \tau$ and w can be mapped to a_{i+1}. In order to achieve this we duplicate P_τ for α times, to get copies $P_{\tau,1}, \ldots, P_{\tau,\alpha}$. Each part $P_{(\tau,i)}$ then asks u for its i^{th} neighbor w, and then checks if τ can be extended to $\overrightarrow{T}_{t_1+1}$ using w. If (τ, w) is a copy of $\overrightarrow{T}_{i+1}$, then the part creates the tuple (τ, w). All the the duplications above can be done in parallel to all copies of $\overrightarrow{T}$ residing on a single machine, so that in total each machine is duplicated α time. Since each machine has $O(n^\delta)$ information, and $O(n^{2\delta})$ space, by Lemma 4.2.3, this process takes $O(\log_{n^\delta} \alpha) = O(1/\delta)$ rounds. Furthermore, as each machine is duplicated α times, the amount of total memory increases by a factor of α.

Hence, at the end of the process, all copies of $\overrightarrow{T}$ are generated, the round complexity is $O(1/\delta)$, and the total memory is $O(m\alpha^{t-2})$. □

For a directed graph $\overrightarrow{G}$, we consider the following lists of key-value pairs, as described in Lemma 15 in [BPS20].

- $\mathcal{HM}_1$: $((u,v),1)$ for all $(u,v) \in E(\overrightarrow{G})$.
- $\mathcal{HM}_2$: (S,ℓ) $\forall S \subseteq V(\overrightarrow{G})$ such that $1 \leq |S| \leq 4$ and ℓ is the number of vertices u such that $S \subseteq N^+(u)$.
- $\mathcal{HM}_3$: $\Big((S_1,S_2,\ell)\Big)$ $\forall S_1, S_2 \subseteq V(\overrightarrow{G})$, where $1 \leq |S_1 \cup S_2| \leq 3$, and ℓ is the number of edges $e = (u,v) \in E(\overrightarrow{G})$ such that $S_1 \subseteq N^+(u)$ and $S_2 \subseteq N^+(v)$.

Lemma 4.9.5. *Let $\overrightarrow{G}$ be a directed graph with m edges, such that for every $v \in V(\overrightarrow{G})$, $d^+(v) \leq \alpha$. The lists $\mathcal{HM}_1, \mathcal{HM}_2$ and $\mathcal{HM}_3$ can be computed in $O(1/\delta)$ rounds and $O(m\alpha^3)$ total memory.*

Proof. In order to create $\mathcal{HM}_1$, each vertex u simply adds for each $v \in N^+(u)$ the pair $((u,v),1)$ to the list. Clearly this can be done in $O(1)$ rounds, and $O(m)$ total memory.

We now consider $\mathcal{HM}_2$. Let $s = |S|$ denote the size of the requested set. Fix s, and let $\overrightarrow{T}$ be a DRT which consists of a root and s outgoing neighbors. By Lemma 4.9.4, we can generate all copies of $\overrightarrow{T}$ in $O(1/\delta)$ rounds, and $O(m \cdot \alpha^{s-2}) = O(m \cdot \alpha^2)$ total memory. From each copy $(v, u_1, \ldots, u_s)$ of $\overrightarrow{T}$, we create a tuple $(\{u_1, \ldots, u_s\}, 1)$ and add it to a temporary list $\mathcal{HM}_2'$. Finally, we use Theorem 4.2.1 to sort this list and aggregate the counts of each set $S = \{u_1, \ldots, u_s\}$, so that for every S we create the tuple (S,ℓ) and add it to $\mathcal{HM}_2$, where ℓ is the number of occurrences of the tuple $(S,1)$ in $\mathcal{HM}_2'$. By Theorem 4.2.1, this process takes $O(\log_{n^\delta} m \cdot \alpha^2) = O(1/\delta)$ rounds.

$\mathcal{HM}_3$ is constructed similarly. Fix some s_1 and s_2 such that $1 \leq s_1 + s_2 \leq 3$, and consider the corresponding DRT $\overrightarrow{T}$. That is, $\overrightarrow{T}$ is a DRT with a vertex u with s_1 outgoing neighbors, where one of the neighbors has s_2 additional outgoing neighbors. This is a DRT over $|S_1| + |S_2| + 1 \leq 4$ vertices, so by Lemma 4.9.4, we can generate all copies in $O(1/\delta)$ rounds, and $O(m \cdot \alpha^2)$ total memory. From the list of all copies we can generate $\mathcal{HM}_3$, similarly to as described for $\mathcal{HM}_2$, in $O(1/\delta)$ rounds. □

Theorem 4.9.6. *Let $G = (V, E)$ be a graph with $n = |V|$ and $m = |E|$. There is an algorithm for counting the number of occurrences of any given subgraph H over $k \leq 5$ vertices in G with high probability, with round complexity $O(\sqrt{\log n} + 1/\delta)$, $O(n^{2\delta})$ memory per machine, and $O(m\alpha^3)$ total memory.*

Proof. If H is a k-star, then the number of occurrence of H in G is simply $\sum_{v \in V} \binom{d(v)}{k}$ where $\binom{d(v)}{k} = 0$ for $k > d(v)$, which can be computed in $O(1)$ rounds. Hence, we assume that H is not a star.

The first step in the algorithm of BPS is to direct the graph G according to the degeneracy ordering (see Definition 4.9.2). We achieve this using the algorithm of [GLM19] described in Theorem 4.9.3. Note that the algorithm of [GLM19] returns an approximate degeneracy ordering, but as the degeneracy of a graph is at most twice the arboricity, it holds that each vertex has out-degree $O(\alpha)$.

Given the ordering of $\overrightarrow{G}$, the algorithm continues by considering all orientations $\overrightarrow{H}$ of H (up to isomorphisms). For each $\overrightarrow{H}$ it computes the maximal rooted directed tree, DRT, of $\overrightarrow{H}$, denoted $\overrightarrow{T}$. As H is of constant size, this can be computed in $O(1)$ rounds on a single machine.

The next step is to find all copies of $\overrightarrow{T}$ in $\overrightarrow{G}$. By Lemma 4.9.4, this can be implemented in $O(1/\delta)$ rounds, $O(n^{2\delta})$ space per machine, and $O(m\alpha^2)$ total memory.

Now, for each copy of $\overrightarrow{T}$ in $\overrightarrow{G}$ it needs to be verified if the copy can be completed to a copy of $\overrightarrow{H}$ in $\overrightarrow{G}$. By Lemma 16 in [BPS20], this can be computed in if given query access to $\mathcal{HM}_1, \mathcal{HM}_2$ and $\mathcal{HM}_3$, as defined in Section 4.9.2. That is, it can be determined if a copy τ of $\overrightarrow{T}$ using $O(|H|^2) = O(1)$ queries to the lists $\mathcal{HM}_1$, $\mathcal{HM}_2$ and $\mathcal{HM}_3$. By Lemma 4.9.5, these lists can be generated in $O(1/\delta)$ rounds, and $O(m\alpha^2)$ total memory. For $i \in [1..3]$, let Q_i denote the set of all queries to list $\mathcal{HM}_i$. By [GSZ11], all queries Q_i to $\mathcal{HM}_i$ can be answered in time $O(1/\delta)$.

Finally, by Lemma 16 in [BPS20], each v can use the answers to its queries to compute the number of copies of $\overrightarrow{H}$ it can be extended to. Therefore, by summing over all vertices and over all possible orientations of H, and taking into account isomorphisms, we can compute the number of occurrences of H in $\overrightarrow{G}$. The total round complexity is dominated by computing the approximate arboricity orientation of G and the sorting operations. Therefore the round complexity is $O(\sqrt{\log n} \log\log n + 1/\delta)$. The space per machine is $O(n^{2\delta})$, and the total memory over all machines is $O(m\alpha^3)$. □

4.10 Experiments

We performed experiments using our algorithms given in Sections 4.4 and 4.6. Our code [exp20] simulates the algorithms described in these sections as well as the MPC procedures we use as subroutines. In the implementation of the algorithm of Section 4.4 we output the approximation factor we achieve using our algorithm versus the amount of space per machine. In the implementation of the exact algorithm of Section 4.6 we output the number of MPC rounds necessary to execute the algorithm versus the arboricity

bound we pass into it. In the implementation of the algorithm of Section 4.4, our algorithm achieves a better approximation *on all tested graphs* than the best-known previous algorithm. In the implementation of the exact algorithm of Section 4.6, our algorithm achieves fewer number of MPC rounds than the baseline algorithm. We include these experiments in this chapter only as a proof-of-concept. We leave as interesting future directions implementing and testing our algorithms in massively parallel software framworks, such as Apache Hadoop and others, on much larger graphs.

All real-world graphs on which we performed our experiments can be found in the Stanford Large Network Dataset Collection (SNAP) [LK14].

We tested our algorithms against datasets described in Table 4.1.

File	Number of Vertices (n)	Number of Edges (m)	Number of Triangles (T)
email-Eu-core	1005	25571	105461
ego-Facebook	4039	88234	1612010
feather-lastfm-social	7624	27806	40433
ca-GrQc	5242	14496	48260
musae-twitch (DE)	9498	153138	603088
ca-HepTh	9877	25998	28339
oregon1_010519	11051	22724	17677
ca-HepPh	12008	118521	3358499
email-Enron	36692	183831	727044

Table 4.1: All datasets can be found in the Stanford Large Network Dataset (SNAP) Collection [LK14]. This table shows the number of vertices, edges, and exact number of triangles in each of these graphs.

Results for Section 4.6 The first set of experiments were performed using our new exact triangle count algorithm provided in Section 4.6. The results are shown in Fig. 4-2. Our experiments were performed on five datasets: oregon1_010519, email-Enron, ca-HepTh, ca-GrQc, email-Eu-core. We compare against the baseline algorithm (labeled "-base" in the figures) of removing (and counting) the vertices with degree at most the degeneracy of each graph during each round. We measure the number of rounds our algorithm takes against the amount of space per machine (indicated by the different colors of the bars) and our *initial* setting of our degree bound. Recall that since the degree bound of our algorithm for removal of vertices grows doubly exponentially, we can set our initial degree bound to be smaller than the degeneracy of the graph. Note that for the baseline algorithm, we cannot do this since the degree bound remains the same (and hence, a smaller degree bound than the degeneracy will cause the algorithm to terminate at some point with vertices still left in the graph with degree greater than the bound). Our initial degree bound settings are shown on the x-axes while the y-axes shows the number of rounds. The machine space bounds are in terms of the *number of nodes* of the graph that can fit in each machine. These numbers are shown in the legend indicating the colors of the bars. When setting the cutoff in the baseline algorithm, we consider the degeneracy

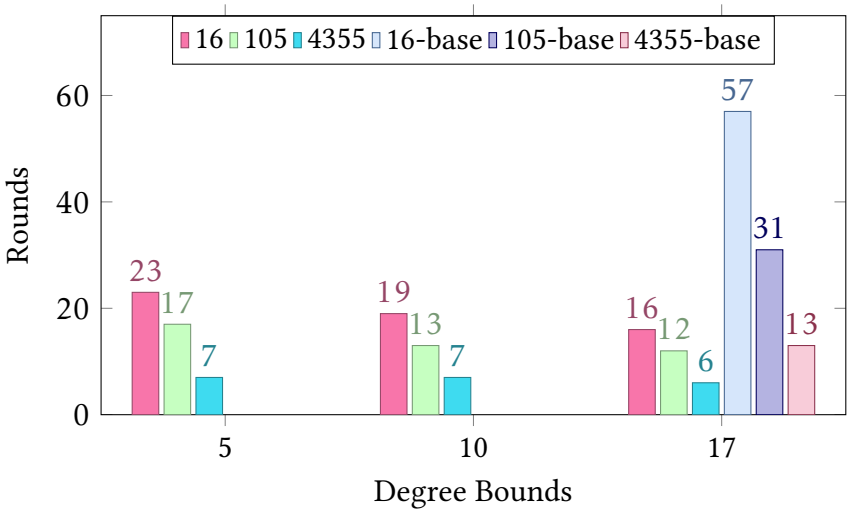
of the graphs instead of the arboricity. But as we noted previously, the degeneracy of the graph is at most within a constant factor of 2 of the arboricity of the graph.

We see in Fig. 4-2 that our algorithms result in less (or the same number of) rounds than the baseline algorithm for all cases given the same space and degree bound except the 2225 case for the email-Enron dataset. This confirms our theoretical analysis of the asymptotic number of rounds of our algorithm, taking $O_\delta(\log\log n)$ compared to the $O_\delta(\log n)$ rounds we expect the baseline algorithm to take. The anomaly with the single 2225 case might be due to the larger constant factor (derived from MPC sort and find duplicate) of having to sort more items per round in our case compared to the baseline case. The experiments confirm that our algorithm can count triangles using much lower initial degree bounds than the degeneracy of the graph. Even for such degree bounds, the number of rounds necessary is still often (much) less than the number of rounds necessary for the baseline algorithm. This provides the advantage of being able to use our algorithm for real-world graphs without first needing to determine their degeneracy value.

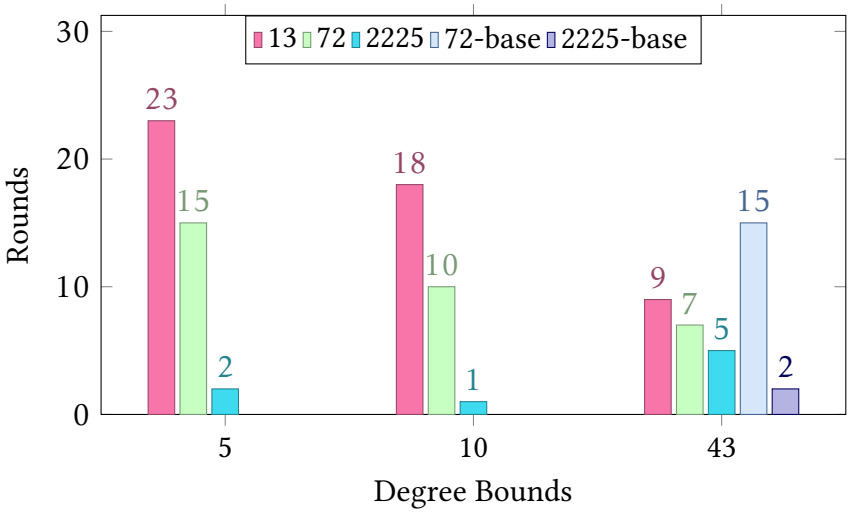
File	δ	Partition Approximation ([PT12])	Our Approximation
ego-Facebook	0.5	0.62	1.31
feather-lastfm-social	0.5	5.41	1.08
ca-GrQc	0.5	4.53	1.64
ca-HepPh	0.5	0.66	1.22
ego-Facebook	0.75	0.75	0.97
ca-GrQc	0.75	5.82	0.82
ca-HepPh	0.75	5.90	0.86
musae-twitch (DE)	0.75	0.74	0.95
oregon1_010519	0.75	0.60	0.71

Table 4.2: The approximation factors obtained when running our algorithm given in Section 4.4 against our implementation of the partition algorithm given in Algorithm 1 and Algorithm 2 of [PT12]. We perform the algorithms on machines of size $2m^\delta \cdot \log n$. The approximation factor is calculated by the equation C/T where C is the triangle count returned by either algorithm and T is the actual count of the triangles in the graph.

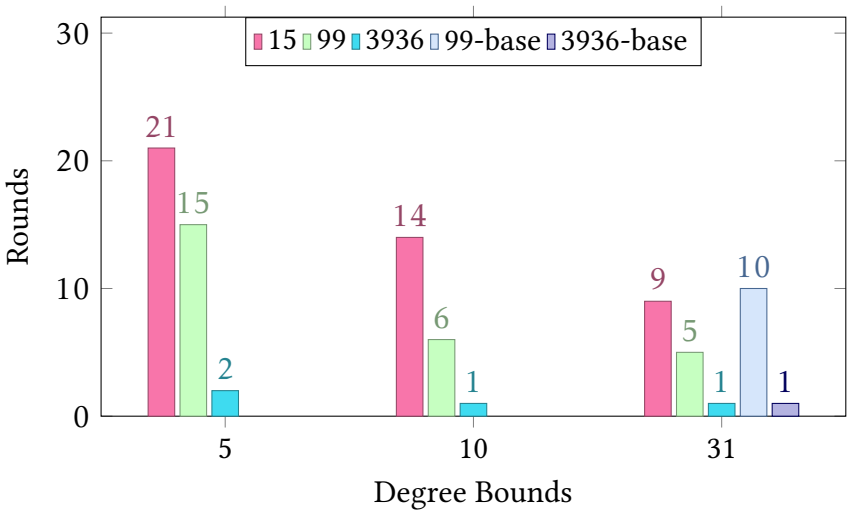
Results for Section 4.4. The results of experiments for our approximation algorithm described in Section 4.4 are given in Table 4.2. We further compare our approximations against our implementation of the partition algorithm given in Algorithms 1 and 2 of [PT12]. In the implementation of our algorithm, due to the (sequential) time constraints of simulating our algorithms, we do not use a k-wise independent hash function, as such functions require too much time to compute. Hence, for our experiments, we use standard pseudorandom functions given in programming packages (specifically numpy in Python3). For each of the experiments the space per machine is $2m^\delta \cdot \log n$, where δ is



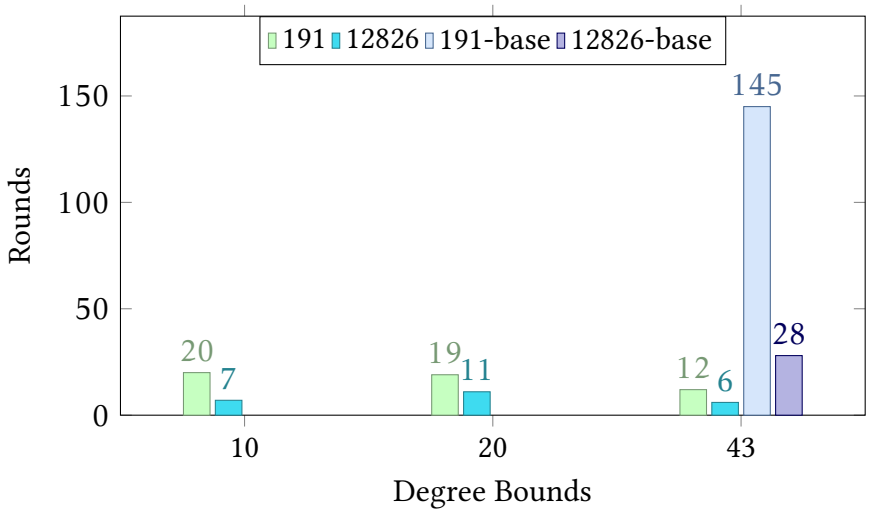
(a) oregon1_010519. Degeneracy is 17.



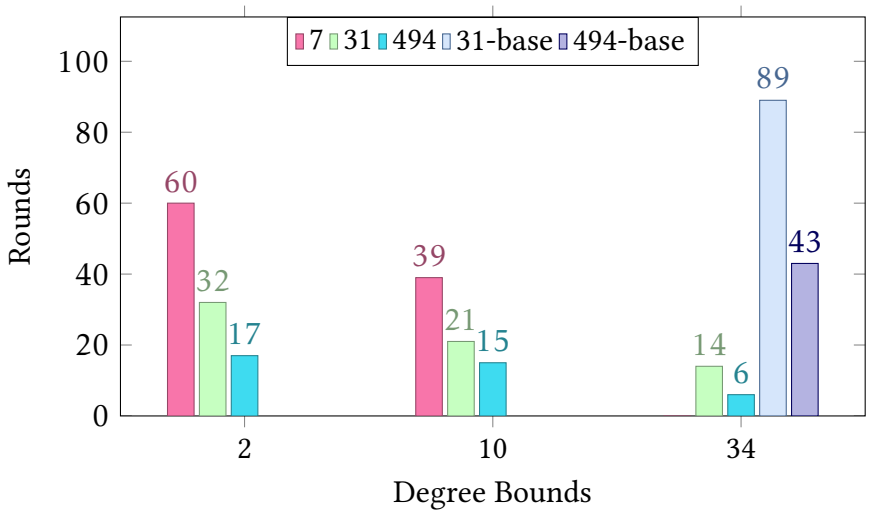
(b) email-Enron. Degeneracy is 43.



(c) ca-HepTh. Degeneracy is 31.



(d) ca-GrQc. Degeneracy is 43.



(e) email-Eu-core. Degeneracy is 34.

Figure 4-2: This set of graphs shows the results of our experiments using our exact counting algorithm described in Section 4.6. We test on five datasets labeled under each plot. In each of these graphs, we compare against the number of rounds required by the MPC algorithm that removes, in each round, only vertices with degree at most the degeneracy of the graph α. Each color represents a different space per machine, which is represented in terms of the number of nodes that can fit in each machine. The colors (green, red, yellow) labeled with "-base" represent our baseline algorithm results.

specified in Table 4.2. The same space per machine is used for both algorithms. The total space used for both algorithms is $2m \cdot \log m$. For the implementation of our algorithm, we set the probability of sampling to be $\frac{1}{5} \cdot \sqrt{\frac{S}{M \cdot k}}$ where we set $k = \max(2, \lfloor 6.5 \cdot \log n \rfloor)$. We chose to test these algorithms on these specific δ values because $\delta = 0.5, 0.7$ represent $\tilde{O}(n)$ and $\tilde{o}(m)$, respectively. Because the theoretical guarantees of our algorithm relies on some specific contraints on T and S, we wanted to see how our algorithm performs on real-world networks. We use the median-of-means trick for the concentration for both algorithms.

As Table 4.2 shows, compared to the partition algorithm, our algorithm obtains a better approximation ratio for all datasets and for all machine spaces. This follows from our theoretical analysis as we ensure $(1+\varepsilon)$-approximations on the number of triangles in each graph using $\tilde{O}(n)$ space and with a quadratically smaller constraint on the number of actual triangles in the graph than all other previous work. Thus, we show that practically, on real-world graphs, our algorithm obtains better approximations on the number of triangles even when given smaller space per machine compared to the state-of-the-art algorithm of [PT11].

4.11 Open Questions

There are a number of key open questions that result from our work:

1. Can we obtain a better bound on the number of triangles, T, while guaranteeing a $(1+\varepsilon)$-approximation, $O(n^{\delta})$ space per machine (for any constant $\delta > 0$), $\widetilde{O}(n+m)$ total space, and $O(1)$ rounds? The main challenge for us was obtaining the induced subgraph for each set of sampled vertices in a machine; perhaps with a different MPC procedure for doing this, one can obtain a better bound on T.
2. Our exact triangle counting algorithm uses $O(\log \log n)$ rounds. Is it possible to obtain $O(1)$ rounds while using $O(n^{\delta})$ space per machine (for constant $\delta > 0$) and $O(m\alpha)$ total space?
3. The best-known algorithm for computing a $(2+\varepsilon)$-approximate k-core decomposition in $O(n^{\delta})$ space per machine (for constant $\delta > 0$) requires $O(\sqrt{\log n} \cdot \log \log n)$ rounds, *whp*, and total memory $\widetilde{O}\left(\max\{m, n^{1+\delta}\}\right)$ [GLM19]. Is it possible to reduce the number of rounds or the total memory for this problem using ideas from our exact triangle counting algorithm (for graphs with bounded arboricity)?

Chapter 5

Scheduling with Communication Delay in Near-Linear Time

This chapter presents results from the paper titled, "Scheduling with Communication Delay in Near-Linear Time" that the thesis author coauthored with Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua R. Wang [LPS+21]. This paper is currently under submission at the time of the writing of this thesis.

5.1 Introduction

The problem of efficiently scheduling a set of jobs over a number of machines is a fundamental optimization problem in computer science that becomes ever more relevant as computational workloads become larger and more complex. Furthermore, in real-world data centers, there exists non-trivial *communication delay* when data is transferred between different machines. There is a variety of very recent literature devoted to the theoretical study of this topic [DKR+20, DKR+21, MRS+20]. However, all such literature to date focuses on obtaining algorithms with good approximation factors for the schedule length, but these algorithms require $\omega(n^2)$ time (and potentially polynomially more) to compute the schedule. In this chapter, we instead focus on efficient, near-linear time algorithms for scheduling while maintaining an approximation factor equal to that obtained by the best-known algorithm for our setting [LR02].

Even simplistic formulations of the scheduling problem (e.g. precedence-constrained jobs with unit length to be scheduled on M machines) are typically NP-hard, and there is a rich body of literature on designing good approximation algorithms for the many variations of multiprocessor scheduling (refer to [Bru10] for a comprehensive history of such problems). Motivated by a desire to better understand the computational complexity of scheduling problems and to tackle rapidly growing input sizes, we ask the following research question:

How computationally expensive is it to perform approximately-optimal scheduling?

In this chapter, we focus on the classical problem of multiprocessor scheduling with communication delays on identical machines where all jobs have unit size. The jobs that

need to be scheduled have data dependencies between them, where the output of one job acts as the input to another. These dependencies are represented using a directed acyclic graph (DAG) $G = (V, E)$ where each vertex $v \in V$ corresponds to a job and an edge $(u, v) \in E$ indicates that job u must be scheduled before v. In our multiprocessor environment, if these two jobs are scheduled on different machines, then some additional time must be spent to transfer data between them. We consider the problem with *uniform communication delay*; in this setting, a uniform delay of ρ is incurred for transferring data between any two machines. Thus for any edge $(u, v) \in E$, if the jobs u and v are scheduled on different machines, then v must be scheduled at least ρ units of time after u finishes. Since the communication delay ρ may be large, it may actually be more efficient for a machine to *recompute* some jobs rather than wait for the results to be communicated. Such duplication of work can reduce schedule length by up to a logarithmic factor [MRS+20] and has been shown to be effective in minimizing latency in schedulers for grid computing and cloud environments [BOC08, CTR+17]. Our scheduling objective is to minimize the makespan of the schedule, i.e., the completion time of the last job. In the standard three field notation for scheduling problems, this problem is denoted "$P \mid \text{duplication}, \text{prec}, p_j = 1, c \mid C_{\max}$", [1] where c indicates uniform communication delay.

This problem was studied by Lepere and Rapine, who devised an $O(\ln \rho / \ln \ln \rho)$-approximation algorithm for it [LR02], under the assumption that the optimal solution takes at least ρ time. However, their analysis was primarily concerned with getting a good quality solution and less with optimizing the running time of their polynomial-time algorithm. A naïve implementation of their algorithm takes roughly $O(m\rho + n \ln M)$ time, where n and m are the numbers of vertices and edges in the DAG, respectively, and M is the number of machines. This runtime is based on two bottlenecks, (i) the computation of ancestor sets, which can be done in $O(m\rho)$ time via propagating in topological order plus merging and (ii) list scheduling, which can be done in $O(n \ln M)$ time by using a priority queue to look up the least loaded machine when scheduling a set of jobs.

However, with growing input sizes, it is highly desirable to obtain a scheduling algorithm whose running time is linear in the size of the input. Our primary contribution is to design a *near-linear time* approximation algorithm while preserving the approximation ratio of the Lepere-Rapine algorithm:

Theorem 5.1.1. *There is an $O(\ln \rho / \ln \ln \rho)$-approximation algorithm for scheduling jobs with precedence constraints on a set of identical machines in the presence of a uniform communication delay that runs in $O\left(n \ln M + \frac{m \ln^3 n \ln \rho}{\ln \ln \rho}\right)$ time, assuming that the optimal solution has cost at least ρ.*

Of course, this is tight, up to log factors, because any algorithm for this problem must respect the precedence constraints, which require $\Omega(n + m)$ time to read in.

[1] The fields denote the following. **Identical machine information:** P: number, M, of machines is provided as input to the algorithm; **Job properties:** duplication: duplication is allowed; prec: precedence constraints; $p_j = 1$: unit size jobs; c: there is non-zero communication delay; **Objective:** $C_{\max}$: minimize makespan.

5.1.1 Related Work

Algorithms for scheduling problems under different models have been studied for decades, and there is a rich literature on the topic (refer to [Bru10] for a comprehensive look). Here we review work on theoretical aspects of scheduling with communication delay, which is most relevant to our results.

Without duplication, scheduling a DAG of unit-length jobs with unit communication delay was shown to be NP-hard by Rayward-Smith [RS87], who also gave a 3-approximation for this problem. Munier and König gave a 4/3-approximation for an unbounded number of machines [MK97], and Hanen and Munier gave a 7/3-approximation for a bounded number of machines [HM01]. Hardness of approximation results were shown in [BGK96, HLV94, Pic95]. In recent results, Kulkarni et al. [KLTY20] gave a quasi-polynomial time approximation scheme for a constant number of machines and a constant communication delay, whereas Davies et al. [DKR+20] gave an $O(\log \rho \log M)$ approximation for general delay and number of machines. Even more recently, Davies et al. [DKR+21] presented a $O(\log^4 n)$-approximation algorithm for the problem of minimizing the weighted sum of completion times on *related machines* in the presence of communication delays. They also obtained a $O(\log^3 n)$-approximation algorithm under the same model but for the problem of minimizing makespan under communication delay. Notably, *none* of the aforementioned algorithms consider duplication.

Allowing the duplication of jobs was first studied by Papadimitriou and Yannakakis [PY90], who obtained a 2-approximation algorithm for scheduling a DAG of identical jobs on an unlimited number of identical machines. A number of papers have improved the results for this setting [AK98, DA98, PLW96]. With a finite number of machines, Munier and Hanen [MH97] proposed a 2-approximation algorithm for the case of unit communication delay, and Munier [Mun99] gave a constant approximation for the case of tree precedence graphs. For a general DAG and a fixed delay ρ, Lepere and Rapine [LR02] gave an algorithm that finds a solution of cost $O(\log \rho / \log \log \rho) \cdot (OPT + \rho)$, which is a true approximation if one assumes that $OPT \geq \rho$. This is the main result that this chapter builds on. It applies to a set of identical machines and a set of jobs with unit processing times. Recently, an $O(\log M \log \rho / \log \log \rho)$ approximation has been obtained for a more general setting of M machines that run at different speeds and jobs of different lengths [MRS+20], also under the assumption that $OPT \geq \rho$. However, the running time of this algorithm is a large polynomial, as it requires solving an LP with $O(Mn^2)$ variables.

5.1.2 Technical Contributions

A naïve implementation of the Lepere-Rapine algorithm is bottlenecked by the need to determine the set of all ancestors of a vertex v in the graph, as well as the intersection of this set with a set of already scheduled vertices. Since the ancestor sets may significantly overlap with each other, trying to compute them explicitly (e.g., using DFS to write them down) results in superlinear work. We use a variety of technical ideas to only compute the essential size information that the algorithm needs to make decisions about these ancestor sets.

- **Size estimation via sketching.** We use streaming techniques to quickly estimate

the sizes of all ancestor sets simultaneously. It costs $O((|V|+|E|)\log^2 n)$ time to make such an estimate once, so we are careful to do so sparingly.

- **Work charging argument.** Since we cannot compute our size estimates too often, we still need to perform some DFS for ancestor sets. We control the amount of work spent doing so by carefully charging the edges searched to the edges we manage to schedule.
- **Sampling and pruning.** Because we cannot brute-force search all ancestor sets, we randomly sample vertices, using a consecutive run of unscheduleable vertices as evidence that many vertices are not schedulable. This allows us to pay for an expensive size-estimator computation to prune many ancestor sets simultaneously.

5.1.3 Organization

The main contribution of this chapter is our algorithm for scheduling small subgraphs in near-linear time. We provide a detailed description and analysis of this algorithm in Section 5.5. Then, we proceed with our algorithm for scheduling general graphs in Section 5.6.

5.2 Problem Definition and Preliminaries

An instance of scheduling with communication delay is specified by a directed acyclic graph $G = (V, E)$, a quantity $M \geq 1$ of identical machines, and an integer communication delay $\rho > 1$. We assume that time is slotted and let $T = \{1, 2, \ldots\}$ denote the set of integer times. Each vertex $v \in V$ corresponds to a job with processing time 1 and a directed edge $(u, v) \in E$ represents the precedence constraint that job v depends on job u. In total, there are $n = |V|$ vertices (representing jobs) and $m = |E|$ precedence constraints. The parameter ρ indicates the amount of time required to communicate the result of a job computed on one machine to another. In other words, a job v can be scheduled on a machine at time t only if all jobs u with $(u, v) \in E$ have either completed on the same machine before time t or on another machine before time $t - \rho$. We allow for a job to be *duplicated*, i.e., copies of the same vertex $v \in V$ may be processed on different machines. Let $\mathcal{M}$ be the set of machines available to schedule the jobs. A schedule σ is represented by a set of triples $\{(m, v, t)\} \subset \mathcal{M} \times V \times T$ where each triple represents that job v is scheduled on machine m at time t. The goal is to obtain a feasible schedule that minimizes the makespan, i.e., the completion time of the last job. Let OPT denote the makespan of an optimal schedule. Since ρ represents the amount of time required to communicate between machines, and in practice, any schedule must communicate the results of the computation, we assume that $\mathrm{OPT} \geq \rho$ as is standard in literature [LR02, MRS+20].

We now set up some notation to help us better discuss dependencies arising from the precedence constraints of G. For any vertex $v \in V$, let $\mathsf{Pred}(v) \triangleq \{u \in V \mid (u, v) \in E\}$ be the set of (immediate) predecessors of v in the graph G, and similarly let $\mathsf{Succ}(v) \triangleq \{w \in V \mid (v, w) \in E\}$ be the set of (immediate) successors. For $H = (V_H, E_H)$, a subgraph of G, we use $\mathcal{A}_H(v) \triangleq \{u \in V_H \mid \exists \text{ a directed path from } u \text{ to } v \text{ in } H\} \cup \{v\}$ to denote the set of (indirect) ancestors of v, including v itself. Similarly, for $S \subseteq V$, we use $\mathcal{A}_H(S) \triangleq \bigcup_{v \in S} \mathcal{A}_H(v)$ to denote the indirect ancestors of the entire set S. We use $\mathcal{E}_H(S)$ to denote

Symbol	Meaning
$G = (V, E)$	main input graph
$n = \|V\|, m = \|E\|$	number of vertices / edges
$H = (V_H, E_H)$	subgraph to be scheduled in each phase
ρ	communication delay
u, v	vertices
$\mathcal{A}_H(v)$	set of ancestors of vertex v in graph H including v
$\mathcal{A}_H(S)$	$\mathcal{A}_H(S) = \bigcup_{v \in S} \mathcal{A}_H(v)$ in graph H
$\mathcal{E}_H(v)$ [resp., $\mathcal{E}_H(S)$]	edges induced by $\mathcal{A}_H(v)$ [resp., $\mathcal{A}_H(S)$] in graph H
$\hat{a}_H(v), \hat{e}_H(v)$	estimated size of $\mathcal{A}_H(v)$ and $\mathcal{E}_H(v)$
M	number of machines
γ	threshold for fresh vs. stale vertices

Table 5.1: Table of Symbols

the edges of the subgraph induced by $\mathcal{A}_H(S)$. We drop the subscript H when the subgraph H is clear from context. Throughout, we use the phrase *with high probability* to indicate with probability at least $1 - \frac{1}{n^c}$ for any constant $c \geq 1$.

For convenience, we summarize the notation we use throughout the chapter in Table 5.1.

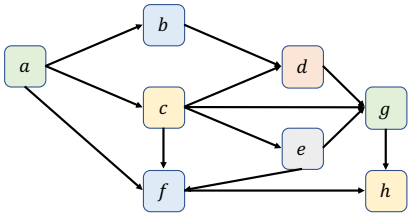
5.3 Technical Overview

We start by reviewing the algorithm of Lepere and Rapine [LR02], shown in Algorithm 15, as our algorithm follows a similar outline. Then we describe the technical improvements of our algorithm to achieve near-linear running time.

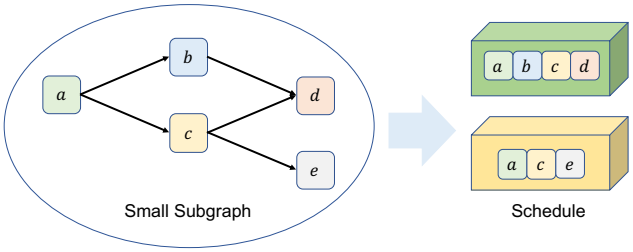
Algorithm 15 Outline of Lepere Rapine Scheduling Algorithm [LR02]

```
1: while G is non-empty do
2:   Let H be a subgraph of G induced by vertices with at most ρ ancestors
3:   while H is non-empty do
4:     for each vertex v in H do
5:       if greater than γ fraction of A_H(v) is unscheduled then
6:         Add A_H(v), in topological order, to a machine with earliest end time
7:     Insert a delay until C + ρ on all machines, where C is the latest end time
8:     Remove scheduled vertices from H
9:   Delete vertices in H from G
```

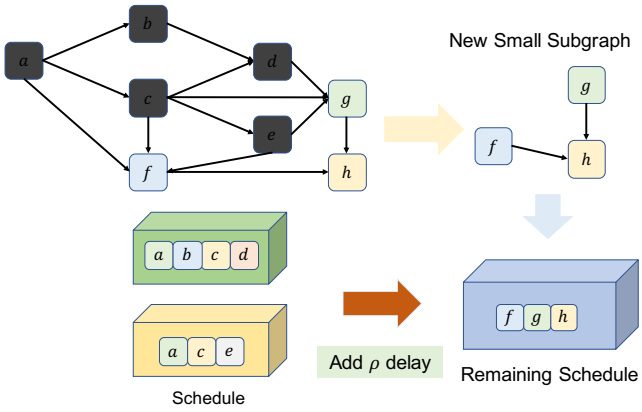
Description of Lepere-Rapine [LR02] The outer loop (Line 1) iteratively finds *small subgraphs* of G which consist of vertices that have *height* at most $\rho + 1$. We show in this chapter that instead of considering their definition of *height*, it is sufficient to consider small subgraphs to be those with at most 2ρ ancestors. We call one iteration of this loop



(a) Initial input DAG.



(b) Find a small subgraph in the input graph and schedule the small subgraph.



(c) Remove scheduled vertices from the graph. Add a ρ communication delay to the schedule after the previously scheduled jobs. Find a new small subgraph and schedule it.

Figure 5-1: Overview of the Lepere-Rapine algorithm for scheduling general graphs.

a *phase*. Within the phase, H is fully scheduled, after which the algorithm goes on to the next "slice" of G. However, H is not scheduled all at once, but instead each iteration of the inner `while` loop (Line 3) schedules a subset of H, which we call a *batch*. To determine which vertices of H make it into a batch, the algorithm checks the fraction of ancestors of each vertex that have already been scheduled in the same batch. If this fraction for a vertex v is low (we call v *fresh* in that case), then its ancestor set $\mathcal{A}_H(v)$ is list-scheduled as a unit, i.e. topologically sorted and placed on one machine. If the fraction of scheduled ancestors is high (in which case we call v *stale*), v is skipped in this iteration. This avoids excessive duplication which would create too much load on the machines. After each batch is placed on the machines, a delay of ρ is added to the end of the schedule to allow all the results to propagate. This allows the scheduled jobs to be deleted from H. This algorithm is illustrated pictorially in Fig. 5-1.

Runtime Challenges with Lepere-Rapine Naively, both finding the small subgraphs as well as determining each batch takes $\Omega(n\rho^2)$ time. Determining which nodes belong in the current small subgraph is a matter of whether their ancestor counts are more than ρ or at most ρ. A standard procedure would be to apply DFS and merge ancestor sets, but that can easily run in $\Omega(\rho^2)$ time per node (a node may have $\Omega(\rho)$ direct parents, each

with an ancestor size of $\Omega(\rho)$ that needs to get merged in).

The other technical hurdle is in determining the batches to schedule. We would like to schedule vertices whose ancestors do not *overlap too much*. To illustrate the difficulty of applying sketching-based methods (e.g. min-hash), consider the following example. Suppose that ρ^2 elements have already been scheduled in this batch. Now, we want to find the number of ancestors of vertex v, $\mathcal{A}(v)$, that intersect with the currently scheduled batch, where $|\mathcal{A}(v)| \leq \rho$ by construction. By the lower bound given in [PSW14], even estimating (up to $1 \pm \varepsilon$ relative error with constant probability) the size of this intersection would require sketches of size at least $\varepsilon^{-2}(\rho^2/\rho) = \varepsilon^{-2}\rho$. Using such ρ-sized sketches over all batches and all small subgraphs requires $\Omega(n\rho)$ time in total.

Since ρ may be super-logarithmic, these naive implementations don't quite meet our goal of a near-linear time algorithm. To summarize, the two main technical challenges for our setting are the following:

Challenge 5.3.1. *We must be able to find the small subgraphs in near-linear time.*

Challenge 5.3.2. *We must be able to find the vertices to add to each batch $\mathcal{B}$ in near-linear time.*

We solve Challenge 5.3.1 by relaxing the definition of small subgraph and using *count-distinct* estimators (discussed in Section 5.4). The majority of this chapter focuses on solving Challenge 5.3.2 which requires several new techniques for the problem outlined in the rest of this section (Section 5.3.1 and Section 5.3.2). The below procedures run on a small subgraph, $H = (V_H, E_H)$, where the number of ancestors of each vertex is bounded by 2ρ. Note the factor of 2 results from our count-distinct estimator. This is described in Section 5.5.

5.3.1 Sampling Vertices to Add to the Batch

We first partition the set of unscheduled vertices in V_H into buckets based on the estimated number of edges in the subgraph induced by their ancestors. (We place vertex v–if it has no ancestors–into the smallest bucket.) More formally, let S_i be the set of vertices not yet scheduled in iteration i (Line 3, Algorithm 15). We partition S_i into $k = O(\log \rho)$ buckets $K_1, \ldots, K_k$ such that bucket K_j contains all vertices $w \in S_i$ where $\hat{e}(w) \in [2^j, 2^{j+1})$; $\hat{e}(w)$ denotes the estimated number of edges in the subgraph induced by ancestors of w.

From each bucket C_j, in decreasing order of j, we sample vertices, sequentially, without replacement. For each sampled vertex v, we enumerate its ancestors and determine how many are in the current batch $\mathcal{B}$. If at least a γ-fraction of the vertices *are not in* $\mathcal{B}$ *and* at least a γ-fraction of the edges in the induced subgraph $G_{H_i}(v)$ are *not in* $\mathcal{B}$, then add v to $\mathcal{B}$. We call such a vertex v ***fresh***. Otherwise, we do not add v to $\mathcal{B}$ and label this vertex as ***stale***. For our algorithms, we set $\gamma = \frac{1}{\sqrt{\rho}}$ but γ can be set to any value $\gamma < 1/2$. Lepere-Rapine did not consider edges in their algorithm because the number of edges in the induced subgraph does not affect the schedule length; however, considering edges is crucial for our algorithm to run in near-linear time.

For each bucket sequentially, we sample vertices uniformly at random, until we have sampled $O(\log n)$ consecutive vertices that are *stale* (or we have run out of vertices and

the bucket is empty). Then, the key intuition is that for every v that we add to $\mathcal{B}$, we can afford to *charge the cost of enumerating the ancestor set* for $O(\log n)$ additional vertices in the same bucket as well as $O(\log n)$ additional vertices in each bucket with smaller j to it. Because we are looking at buckets with decreasing size, we can charge the additional vertices found in future buckets to the most recently found fresh vertex. To see why we can charge the samples from buckets with smaller i, suppose that one vertex v in bucket i was added to B and no vertices in buckets $(i, \log \rho]$ were added to B. Then, the cost charged to v of enumerating the $O(\log n)$ ancestor sets in buckets $[i, \log \rho]$ is at most $\sum_{j=i}^{\log \rho} (2^{\log \rho - j}) \log n = O(2^{\log \rho - i} \log n)$, asymptotically the same cost as charging the sampled vertices from bucket i.

5.3.2 Pruning All Stale Vertices from Buckets

After we have performed the sampling procedure, we are still not done. Our goal is to make sure that *all vertices which are not included* in $\mathcal{B}$ are approximately stale. This means that we must remove the stale vertices so that we can perform our sampling procedure again in a smaller sample space in order to find additional fresh vertices. To accomplish this, we perform a ***pruning*** procedure involving re-estimating the ancestor sets consisting of vertices that have not been added to the batch. Using these estimates, we remove *all* stale vertices from our buckets. Note that we *do not rebucket the vertices* because none of the ancestor sets of the vertices changed sizes. Then, we perform our sampling procedure above (again) to find more fresh vertices. The key is that since we removed all stale vertices, *the first sampled vertex from the largest non-empty bucket is fresh.*

We perform the above sampling and pruning procedures until each bucket is empty. Then, we schedule the batch and remove all scheduled vertices from H and proceed again with the procedure until the graph is empty. We perform a standard simple greedy list scheduling algorithm (Appendix A.2) on our batch on M machines.

5.4 Estimating Number of Ancestors

Let $\tilde{G} = (\tilde{V}, \tilde{E})$ be an arbitrary directed, acyclic graph. We first present our algorithm to estimate the number of ancestors of any vertex $v \in \tilde{V}$. Consider any vertex $v \in \tilde{V}$ and let $p_1, p_2, \ldots p_\ell$ be the predecessors of v in $\tilde{G}$. Then we have $\mathcal{A}_{\tilde{G}}(v) = \cup_{i=1}^{\ell} \mathcal{A}_{\tilde{G}}(p_i) \cup \{v\}$ and hence $|\mathcal{A}_{\tilde{G}}(v)|$ is the number of distinct elements in the multiset $\cup_{i=1}^{\ell} \mathcal{A}_{\tilde{G}}(p_i) \cup \{v\}$. In order to estimate $|\mathcal{A}_{\tilde{G}}(v)|$ efficiently, we use a procedure to estimate the number of distinct elements in a data stream. This problem, known as the *count-distinct problem*, is well studied and many efficient estimators exist [AMS96, BYJK+02, WVZT90, FFGM07, KNW10]. Since we need to estimate $|\mathcal{A}_{\tilde{G}}(v)|$ for all vertices $v \in \tilde{V}$ in near-linear time, we require an additional *mergeable* property to ensure that we can efficiently obtain an estimate for $|\mathcal{A}_{\tilde{G}}(v)|$ from the estimates of the parent ancestor set sizes $\{|\mathcal{A}_{\tilde{G}}(p_1)|, \ldots, |\mathcal{A}_{\tilde{G}}(p_\ell)|\}$.

We formally define the notion of a *count-distinct estimator* and the mergeable property.

Definition 5.4.1. *For any multiset $\mathcal{S}$, let $|\mathcal{S}|$ denote the number of distinct elements in $\mathcal{S}$. We say T is an (ε, δ, D)-*CountDistinctEstimator *for $\mathcal{S}$ if it uses space D and returns a value $\hat{s}$ such that $(1-\varepsilon)|\mathcal{S}| \leq \hat{s} \leq (1+\varepsilon)|\mathcal{S}|$ with probability at least $(1-\delta)$.*

Definition 5.4.2 (Mergeable Property)**.** *An (ε, δ, D)-*CountDistinctEstimator *exhibits the* mergeable property *if estimator T_1 for multiset $\mathcal{S}_1$ and estimator T_2 for multiset $\mathcal{S}_2$ can be merged in $O(D)$ time using $O(D)$ space into an (ε, δ, D)-estimator for $\mathcal{S}_1 \cup \mathcal{S}_2$.*

We note that the *count-distinct estimator* in [BYJK+02] satisfies the mergeable property and suffices for our purposes. We include a description of the procedure and a proof of the mergeable property in Appendix A.1.

Lemma 5.4.3 ([BYJK+02])**.** *For any constant $\varepsilon > 0$ and $d \geq 1$, there exists an $\left(\varepsilon, \frac{1}{n^d}, O\left(\frac{1}{\varepsilon^2}\log^2 n\right)\right)$-*CountDistinctEstimator *that satisfies the mergeable property where n denotes an upper bound on the number of distinct elements.*

Given such an estimator, one can readily estimate the number of ancestors of each vertex $v \in \tilde{V}$ in near-linear time by traversing the vertices of the graph in topological order. An estimator for vertex v can be obtained by *merging* the estimators for each predecessor of v. Similarly, we can also estimate the number of edges $|\mathcal{E}(v)|$ in the subgraph induced by ancestors of any vertex v in near-linear time. We defer a detailed description of these procedures to Algorithm 36 and Algorithm 37 in Appendix A.1.

5.4.1 Estimator Efficiency

Lemma 5.4.4. *Given any input graph $\tilde{G} = (\tilde{V}, \tilde{E})$ and constants $\varepsilon > 0, d \geq 1$, there exists an algorithm that runs in $O\left((|\tilde{V}| + |\tilde{E}|)\log^2 n\right)$ time and returns estimates $\hat{a}(v)$ and $\hat{e}(v)$ for each $v \in \tilde{V}$ such that $(1-\varepsilon)|\mathcal{A}_{\tilde{G}}(v)| \leq \hat{a}(v) \leq (1+\varepsilon)|\mathcal{A}_{\tilde{G}}(v)|$ and $(1-\varepsilon)|\mathcal{E}_{\tilde{G}}(v)| \leq \hat{e}(v) \leq (1+\varepsilon)|\mathcal{E}_{\tilde{G}}(v)|$ with probability at least $1 - \frac{1}{n^d}$.*

Proof. Lemma A.1.1 provides us with our desired approximation. Now, all that remains to show is that Algorithm 36 and Algorithm 37 runs within our desired time bounds. Algorithm 37 visits each vertex exactly once. For each vertex, it merges the estimators of each of its immediate predecessors. By Lemma A.1.2, each merge takes $O\left(\frac{1}{\varepsilon^2}\log^2 n\right)$ time. Because we visit each vertex exactly once, we also visit each predecessor edge exactly once. This means that in total we perform $O\left(\frac{m}{\varepsilon^2}\log^2 n\right)$ merges. Since ε is constant, this algorithm requires $O(m\log^2 n)$ time. The same proof follows for Algorithm 37. □

Throughout the remaining parts of the chapter, we assume that $\varepsilon = 1/3$ in our estimation procedures and do not explicitly give our results in terms of ε.

5.5 Scheduling Small Subgraphs in Near-Linear Time

Here, we consider subgraphs $H = (V, E)$ such that every vertex in the graph has a bounded number of ancestors and obtain a schedule for such *small subgraphs* in near-linear time.

Definition 5.5.1. *A* small subgraph *is a graph $H = (V_H, E_H)$ where each vertex $v \in V_H$ has at most 2ρ ancestors.*

Our main algorithm schedules a small graph in batches using Algorithm 17. After scheduling a batch of vertices, we insert a communication delay of ρ time units so that results of the computation from the previous batch are shared with all machines (similar to Lepere-Rapine). Then, we remove all vertices that we scheduled and compute the next batch from the smaller graph. We present this algorithm in Algorithm 16.

Algorithm 16 ScheduleSmallSubgraph(H, γ)

Input $H = (V_H, E_H)$ where $|\mathcal{A}_H(v)| \leq 2\rho$ for all $v \in V_H$ and parameter $0 < \gamma < 1/2$.
Output A schedule of small subgraph H on M processors.

1: **while** $H \neq \varnothing$ **do**
2: $\quad\mathcal{B} \leftarrow$ FindBatch(H, γ). [Algorithm 17]
3: $\quad$List schedule $\mathcal{A}(v)$ for all $v \in \mathcal{B}$. (Appendix A.2)
4: $\quad$Insert communication delay of ρ time units into the schedule.
5: $\quad$Remove each $v \in \mathcal{A}(\mathcal{B})$ and all edges adjacent to v from H.

Our algorithm for scheduling small subgraphs relies on two key building blocks – estimating the sizes of the ancestor sets (and ancestor edges) of each vertex (Section 5.4), and using these estimates to find a *batch* of vertices that can be scheduled without any communication (possibly by duplicating some vertices). We show how to find a batch in Section 5.5.1.

5.5.1 Batching Algorithm

Recall that the plan is for our algorithm to schedule a small subgraph by successively scheduling maximal subsets of vertices in the graph whose ancestors do not *overlap too much*; we call such a set of vertices a ***batch***. After scheduling each batch, we remove all the scheduled vertices from the graph and iterate on the remaining subgraph.

A detailed description of this procedure is given in Algorithm 17. For each vertex $v \in V_H$, let $\hat{a}(v)$ and $\hat{e}(v)$ denote the estimated sizes of $\mathcal{A}_H(v)$ and $\mathcal{E}_H(v)$ respectively (henceforth referred to as $\mathcal{A}(v)$ and $\mathcal{E}(v)$). Then the i-th bucket, C_i, is defined as $C_i = \{v \in V_H \mid 2^i \leq \hat{e}(v) < 2^{i+1}\}$. Since every node $v \in V_H$ has at most $O(\rho)$ ancestors, there are only $k = O(\log \rho)$ such buckets. Recall that from Lemma 5.4.4, this estimation can be performed in near-linear time. The algorithm maintains a batch $\mathcal{B}$ of vertices that is initially empty. For each non-empty bucket C_i (processed in decreasing order of size), we repeatedly sample nodes uniformly at random from the bucket (without replacement).

For each sampled node $v \in C_i$, we explicitly enumerate the ancestor sets $\mathcal{A}(v)$ and $\mathcal{E}(v)$ and also compute $\mathcal{A}(v) \setminus \mathcal{A}(B)$ and $\mathcal{E}(v) \setminus \mathcal{E}(B)$. Since we can maintain the ancestor sets of the current batch $\mathcal{B}$ in a hash table, this enumeration takes $O(|\mathcal{E}(v)|)$ time. A sampled node v is said to be *fresh* if $|\mathcal{A}(v) \setminus \mathcal{A}(B)| > \gamma|\mathcal{A}(v)|$ *and* $|\mathcal{E}(v) \setminus \mathcal{E}(B)| > \gamma|\mathcal{E}(v)|$; and said to be *stale* otherwise. The algorithm adds all fresh nodes to the batch $\mathcal{B}$ and continues sampling from the bucket until it samples $\Theta(\log n)$ consecutive stale nodes. Once all the buckets have been processed, we *prune* the buckets to remove all stale nodes and then repeat the

sampling procedure until all buckets are empty. The pruning procedure is presented in Algorithm 18. In this step, we again estimate the sizes of ancestor sets of all vertices in the graph $H \setminus \mathcal{A}(B)$ to determine whether a vertex is stale.

Algorithm 17 FindBatch(H, γ)

Input A subgraph $H = (V_H, E_H)$ such that $|\mathcal{A}_H(v)| \leq 2\rho$ for all $v \in V_H$; $0 < \gamma < 1/2$.
Output Returns batch $\mathcal{B}$, the batch of vertices to schedule.

1: Let $N = \Theta(\log n)$.
2: Initially, $\mathcal{B} \leftarrow \varnothing$ and all nodes are unmarked.
3: Obtain estimates $\hat{a}(v)$ and $\hat{e}(v)$ for all $v \in V_H$.
4: Let bucket $C_i = \{v \in V_H : 2^i \leq \hat{e}(v) < 2^{i+1}\}$.
5: **while** at least one bucket is non-empty **do**
6: **for** $i = k$ to 1 **do**
7: Let $s = 0$.
8: **while** $s < N$ and $|C_i| > 0$ **do**
9: Let v be a uniformly sampled node in bucket C_i.
10: Find $\mathcal{A}(v)$ and $\mathcal{A}(v) \setminus \mathcal{A}(\mathcal{B})$ as well as $\mathcal{E}(v)$ and $\mathcal{E}(v) \setminus \mathcal{E}(\mathcal{B})$.
11: **if** $|\mathcal{A}(v) \setminus \mathcal{A}(\mathcal{B})| > \gamma|\mathcal{A}(v)|$ and $|\mathcal{E}(v) \setminus \mathcal{E}(\mathcal{B})| > \gamma|\mathcal{E}(v)|$ **then**
12: Mark v as fresh, add v to $\mathcal{B}$, and remove v from K_i.
13: Set $s = 0$.
14: **else**
15: Mark v as stale and remove v from K_i. $s = s + 1$.
16: $C_1, \ldots, C_k \leftarrow$ Prune(H, B, $C_1, \ldots, C_k$) [Algorithm 18].
17: Return $\mathcal{B}$.

5.5.2 Analysis

We first provide two key properties of the batch $\mathcal{B}$ of vertices found by Algorithm 17 that are crucial for our final approximation factor and then analyze the running time of the algorithm.

Quality of the Schedule We show that $\mathcal{B}$ comprises of vertices whose ancestor sets do not overlap significantly, and further that it is the "maximal" such set.

5.5.3 Charateristics of Batches

Lemma 5.5.2. *The batch $\mathcal{B}$ returned by Algorithm 17 satisfies $|\mathcal{A}(\mathcal{B})| > \gamma \sum_{v \in \mathcal{B}} |\mathcal{A}(v)|$ and $|\mathcal{E}(\mathcal{B})| > \gamma \sum_{v \in \mathcal{B}} |\mathcal{E}(v)|$.*

Proof. Let $\mathcal{B}^{(\ell)} \subseteq B$ denote the set containing the first ℓ vertices added to B by the algorithm. We prove the lemma via induction. In the base case, $\mathcal{B}^{(1)}$ consists of a single vertex and trivially satisfies the claim. Now suppose that the claim is true for some $\ell \geq 1$ and let v be the $(\ell + 1)$-th vertex to be added to B. By Line 11 of Algorithm 17, we add a vertex v

Algorithm 18 Prune$(H, B, C_1, \ldots, C_k)$

Input A graph $H = (V, E)$, a batch $\mathcal{B} \subseteq V$, and buckets $C_1, \ldots, C_k$.
Output New buckets $C_1, \ldots, C_k$.

1: Obtain estimates $\hat{a}_H(v)$ and $\hat{e}_H(v)$ for all nodes $v \in \cup_{i=1}^{k} C_i$ in the graph H.
2: Let $H' \leftarrow H \setminus \mathcal{A}(B)$
3: Obtain estimates $\hat{a}_{H'}(v)$ and $\hat{e}_{H'}(v)$ for all nodes $v \in \cup_{i=1}^{k} C_i$ in the graph H'.
4: **for** $i = k$ to 1 **do**
5: **for** each node v in bucket C_i **do**
6: $X \leftarrow \dfrac{\hat{a}_{H'}(v)}{\hat{a}_H(v)}$.
7: $Y \leftarrow \dfrac{\hat{e}_{H'}(v)}{\hat{e}_H(v)}$.
8: **if** $X \leq 2\gamma$ or $Y \leq 2\gamma$ **then**
9: Remove v from C_i.
10: Return the new buckets $C_1, \ldots, C_k$.

into $\mathcal{B}^{(\ell)}$ if and only if $|\mathcal{A}(v) \setminus \mathcal{A}(\mathcal{B}^{(\ell)})| > \gamma|\mathcal{A}(v)|$ and $|\mathcal{E}(v) \setminus \mathcal{E}(\mathcal{B}^{(\ell)})| > \gamma|\mathcal{E}(v)|$. Furthermore, since we enumerate $\mathcal{A}(v)$ via DFS, our calculation of the cardinality of each of these sets is exact. We now have, $|\mathcal{A}(\mathcal{B}^{(\ell+1)})| = |\mathcal{A}(\mathcal{B}^{(\ell)})| + |\mathcal{A}(v) \setminus \mathcal{A}(\mathcal{B})| > |\mathcal{A}(\mathcal{B}^{(\ell)})| + \gamma|\mathcal{A}(v)|$. By the induction hypothesis, we now have $|\mathcal{A}(\mathcal{B}^{(\ell+1)})| > \gamma \sum_{w \in \mathcal{B}^{(\ell)}} \mathcal{A}(w) + \gamma\mathcal{A}(v) = \gamma \sum_{w \in \mathcal{B}^{(\ell+1)}} \mathcal{A}(w)$. The same proof also holds for $\mathcal{E}(\mathcal{B})$ and the lemma follows. □

Lemma 5.5.3. *If a vertex w was not added to B, it is pruned by Algorithm 18, with high probability. If a vertex v is pruned by Algorithm 18, then $|\mathcal{A}(v) \setminus \mathcal{A}(\mathcal{B})| \leq 4\gamma|\mathcal{A}(v)|$ or $|\mathcal{E}(v) \setminus \mathcal{E}(\mathcal{B})| \leq 4\gamma|\mathcal{E}(v)|$, with high probability.*

Proof. We first prove that any vertex v that is not added to B must be removed from its bucket by Algorithm 18. Any vertex not added to B must have $|\mathcal{A}(v) \setminus \mathcal{A}(B)| \leq \gamma|\mathcal{A}(v)|$. By Lemma 5.4.4, $\hat{a}_{H'}(v) \leq 4/3|\mathcal{A}(v) \setminus \mathcal{A}(B)|$ and $\hat{a}_H(v) \geq 2/3|\mathcal{A}(v)|$, with high probability. This must mean that $\frac{\hat{a}_{H'}(v)}{\hat{a}_H(v)} \leq \frac{4/3|\mathcal{A}(v)\setminus\mathcal{A}(B)|}{2/3|\mathcal{A}(v)|} \leq \frac{4/3\gamma|\mathcal{A}(v)|}{2/3|\mathcal{A}(v)|} \leq 2\gamma$. Thus, v will be pruned. The same proof holds for $\hat{e}_{H'}(v)$.

We now prove that the pruning procedure successfully prunes vertices with not too many unique ancestors. In Algorithm 18, by Lemma 5.4.4 (setting $\varepsilon = 1/3$), we have with high probability, $\hat{a}_{H'}(v) \geq 2/3|\mathcal{A}_{H'}(v)|$. Similarly, with high probability, $\hat{a}_H(v) \leq 4/3|\mathcal{A}_H(v)|$. This means $X = \frac{\hat{a}_{H'}(v)}{\hat{a}_H(v)} \geq \frac{2/3|\mathcal{A}_{H'}(v)|}{4/3|\mathcal{A}_H(v)|} = \frac{1}{2}\left(\frac{|\mathcal{A}_{H'}(v)|}{|\mathcal{A}_H(v)|}\right)$. By the same argument, we also have $Y = \frac{\hat{e}_{H'}(v)}{\hat{e}_H(v)} \geq \frac{1}{2}\left(\frac{|\mathcal{E}_{H'}(v)|}{|\mathcal{E}_H(v)|}\right)$ with high probability.

By Line 8 of Algorithm 18, when we remove a vertex v we have either $X \leq 2\gamma$ or $Y \leq 2\gamma$. By the above, $X, Y \geq \frac{1}{2}(4\gamma) = 2\gamma$. Thus, the largest that $\left(\frac{|\mathcal{A}_{H'}(v)|}{|\mathcal{A}_H(v)|}\right)$ or $\left(\frac{|\mathcal{E}_{H'}(v)|}{|\mathcal{E}_H(v)|}\right)$ can be while still being pruned is 4γ. Thus, with high probability, we have either $\frac{|\mathcal{A}_{H'}(v)|}{|\mathcal{A}_H(v)|} \leq 4\gamma$ or $\frac{|\mathcal{E}_{H'}(v)|}{|\mathcal{E}_H(v)|} \leq 4\gamma$. Since $\mathcal{A}_{H'}(v) = \mathcal{A}(v) \setminus \mathcal{A}(\mathcal{B})$, the claim follows. □

The above two lemmas tell us that there are enough unique elements in each batch B, any vertex not added to B will be pruned w.h.p., and the pruning procedure only prunes

vertices with a large enough overlap with B w.h.p. This allows us to show the following lemma on the length of the schedule produced by Algorithm 16 for small subgraph H. We first show that we only call Algorithm 17 at most $O\left(\log_{1/\gamma}(\rho)\right)$ times from Line 2 of Algorithm 16.

5.5.4 Number of Batches and Small Subgraph Schedule Length

Lemma 5.5.4. *The number of batches needed to be scheduled before all vertices in H are scheduled is at most $4\log_{1/4\gamma}(2\rho)$, with high probability.*

Proof. By Lemma 5.5.3, each vertex v we do not schedule in a batch $\mathcal{B}$ has at least $(1-4\gamma)|\mathcal{A}(v)|$ vertices in $\mathcal{A}(\mathcal{B})$ or at least $(1-4\gamma)|\mathcal{E}(v)|$ edges in $\mathcal{E}(\mathcal{B})$. Since we assumed that all vertices in H have $\leq 2\rho$ ancestors, this means that v can only remain unscheduled for at most $2\log_{1/4\gamma}(4\rho^2)$ batches until $\mathcal{A}(v)$ and $\mathcal{E}(v)$ both become empty (G_v can have at most $4\rho^2$ edges). □

Using Lemma 5.5.4, we can prove the length of the schedule for H using Algorithm 16. The proof of this lemma is similar to the proof of schedule length of small subgraphs in [LR02].

Lemma 5.5.5. *With high probability, the schedule obtained from Algorithm 16 has size at most $\frac{|V_H|}{\gamma M} + 12\rho\log_{1/4\gamma}(2\rho)$ on M processors.*

Proof. By definition of the input, each $\mathcal{A}(v)$ for $v \in V_H$ has at most 2ρ elements. Recall that we schedule all elements in each batch $\mathcal{B}$ by duplicating the common shared ancestors such that we obtain a set of independent ancestor sets to schedule. Then, we use a standard list scheduling algorithm to schedule these lists; see Appendix A.2 for a classic list scheduling algorithm. Each vertex in H gets scheduled in exactly one batch since we remove all scheduled vertices from the subgraph used to compute the next batch. Let $B_1, B_2, \ldots, B_k$ denote the batches scheduled by Algorithm 16. Let H_i be the subgraph obtained from H by removing batches $B_0, \ldots, B_{i-1}$ and adjacent edges. (B_0 is empty.) By Lemma 5.5.2, with high probability, for each batch $\mathcal{B}_i$, we have $\sum_{v\in\mathcal{B}_i}|\mathcal{A}_{H_i}(v)| \leq \frac{1}{\gamma}|\mathcal{A}_{H_i}(\mathcal{B}_i)|$. Let $Z_i = \frac{1}{\gamma}|\mathcal{A}_{H_i}(\mathcal{B}_i)|$.

Graham's list scheduling algorithm [Gra69] for independent jobs is known to produce a schedule whose length is at most the total length of jobs divided by the number of machines, plus the length of the longest job. In our case, we treat each ancestor set as one big independent job, and thus for each batch $\mathcal{B}_i$, this bound becomes $Z_i/M + 2\rho$.

Finally Algorithm 16 inserts an idle time of ρ between two successive batches. The total length of the schedule is thus upper bounded by (where k is the number of batches):

$$\sum_{i=1}^{k}\left(\frac{Z_i}{M}+2\rho+\rho\right)\leq 3\rho\cdot 4\log_{1/4\gamma}(2\rho)\sum_{i=1}^{k}\frac{Z_i}{M} \quad \text{(by Lemma 5.5.4)}$$

$$\leq 3\rho\cdot 4\log_{1/4\gamma}(2\rho)+\frac{1}{\gamma M}\cdot\sum_{i=1}^{k}|\mathcal{A}_{H_i}(B_i)|$$

$$=\frac{|V_H|}{\gamma M}+12\rho\log_{1/4\gamma}(2\rho)$$ □

Running Time In order to analyze the running time of Algorithm 17, we need a couple of technical lemmas. The key observation is that although computing the ancestor sets $\mathcal{A}(v)$ and $\mathcal{E}(v)$ (in Line 10) of a vertex v takes $O(|\mathcal{E}(v)|)$ time in the worst case, we can bound the total amount of time spent computing these ancestor sets by the size of the ancestor sets scheduled in the batch. There are two main components to the analysis. First, we show that after every iteration of the *pruning* step, the number of vertices in each bucket reduces by at least a constant fraction and hence the sampling procedure is repeated at most $O(\ln n)$ times per batch. Secondly, we use a charging argument to upper bound the amount of time spent enumerating the ancestor sets of sampled vertices.
Finding Stale Vertices. We first argue that with high probability, there are at most $O(\ln n)$ iterations of the while loop in Line 5 of Algorithm 17. Intuitively, in each iteration of the while loop, the number of vertices in any bucket C_i reduces by at least a constant fraction.

5.5.5 Runtime of Scheduling Small Subgraphs

Lemma 5.5.6. *For any constant $d \geq 1$ and $\psi > 0$, there is a constant $c \geq 1$ such that, with probability at least $1-\frac{1}{n^d}$, at most a ψ-fraction of remaining nodes in each bucket are fresh after sampling $c\ln n$ stale vertices consecutively.*

Proof. Algorithm 17 samples the vertices in each bucket C_i consecutively, uniformly at random without replacement, until a new fresh vertex is found or at least $c\ln n$ stale vertices are sampled consecutively. Let F be the set of fresh and stale vertices sampled (and removed) so far from bucket K_i before the most recent time a fresh vertex was sampled from K_i (i.e. F includes all vertices sampled including and up to the most recent fresh vertex sampled from K_i). Let f be some fraction $0 < f < 1$. Suppose at most a f-fraction of the vertices in bucket C_i are fresh after removing the previously sampled F vertices and $c\ln n - 1$ additional stale vertices. Such an f exists for every bucket with at least one fresh vertex and one stale vertex after doing such removal. (In the case when all vertices in the bucket are fresh, all vertices from that bucket will be sampled and added to B. If all vertices in the bucket are stale, then $c\ln n$ stale vertices will be sampled immediately.)

From here on out, we assume the bucket K_i only contains the remaining vertices after the previously sampled F vertices were removed. We assume the number of vertices in K_i is more than $c\ln n$. The expected number of fresh vertices in the $c\ln n$ samples from C_i is upper bounded by:

$$\sum_{i=1}^{c\ln n}\left(i\cdot\binom{c\ln n}{i}f^i(1-f)^{c\ln n-i}\right)=fc\ln n.$$

The above is an upper bound on the expected number of fresh vertices in K_i since f is the fraction of fresh vertices after removing $c\ln n-1$ stale vertices, so f upper bounds the current fraction of fresh vertices in K_i.

By the Chernoff bound, the probability that we sample less than $(1-\varepsilon)fc\ln n$ fresh vertices is less than $\exp\left(-\frac{\varepsilon^2 fc\ln n}{2}\right)$. However, we are instead sampling vertices until the next $c\ln n$ consecutive vertices are all stale.

When $c>\frac{1}{(1-\varepsilon)f}$, $(1-\varepsilon)fc\ln n\geq 1$ for any $0<\varepsilon<1$ and $0<f<1$. Then, the probability that no fresh vertices are sampled is less than $\exp\left(-\frac{\varepsilon^2 fc\ln n}{3}\right)=n^{\frac{-\varepsilon^2 fc}{3}}$. We replace f in this expression by a constant $\varphi\in(0,1)$. If $f=o(1)$, then there exists a constant φ for which at most a φ-fraction of the vertices in K_i are fresh. If $f=\omega(1)$, then the probability becomes super-polynomially small. We can sample $c\ln n$ vertices for large enough constant $c\geq\frac{3d}{\varepsilon^2\psi}$ such that with probability at least $1-\frac{1}{n^d}$ for any constant $d\geq 1$, there exists less than ψ-fraction of vertices in the bucket that are fresh if the next $c\ln n$ sampled vertices are stale. □

Lemma 5.5.7. *We perform $O(\ln n)$ iterations of sampling and pruning, with high probability, before all buckets are empty. In other words, with high probability, Line 5 of Algorithm 17 runs for $O(\ln n)$ iterations.*

Proof. We prove the lemma for one bucket C_i and by the union bound, the lemma holds for all buckets. First, any sampled vertex which is fresh is added to $\mathcal{B}$. Furthermore, we showed in Lemma 5.5.3 that any vertex which is stale is removed from C_i by Algorithm 18. Since the estimates $\hat{a}(v)$ and $\hat{e}(v)$ are within a $\frac{1}{3}$-factor of $|\mathcal{A}(v)|$ and $|\mathcal{E}(v)|$, respectively, we can upper bound $\frac{\hat{a}_{H'}(v)}{\hat{a}_H(v)}\leq 2\cdot\frac{|\mathcal{A}(v)\setminus\mathcal{A}(B)|}{|\mathcal{A}(v)|}$ (same holds for $\hat{e}(v)$). If a vertex v is stale, then with high probability, we have either $\frac{\hat{a}_{H'}(v)}{\hat{a}_H(v)}\leq\frac{2|\mathcal{A}(v)\setminus\mathcal{A}(B)|}{|\mathcal{A}(v)|}\leq 2\gamma$ or $\frac{\hat{e}_{H'}(v)}{\hat{e}_H(v)}\leq 2\gamma$, and it is removed by Line 8 of Algorithm 18. Since any fresh vertices that are sampled gets added into $\mathcal{B}$ and all stale vertices are pruned at the end of each iteration, it only remains to show a large enough number of stale vertices are pruned.

Lemma 5.5.6 guarantees that, with high probability, at least $(1-\psi)$-fraction of the vertices in C_i are stale for any constant $\psi\in(0,1)$. Then, Algorithm 18 removes at least $(1-\psi)|C_i|$ vertices in C_i in each iteration. The number of iterations needed is then $\log_{1/(1-\psi)}(|C_i|)=O(\ln n)$.

Since there exists $O(\log\rho)$ buckets and $O(m)$ estimates, we can take the union bound on the probability of success over all buckets and estimates. We obtain, with high probability, $O(\log n)$ iterations are necessary before all buckets are empty. □

Charging the Cost of Examining Stale Sets. Here we describe our charging argument that allows us to explictly enumerate the ancestor set of each sampled vertex. Computing the

ancestor set of a vertex v takes time $O(|\mathcal{E}(v)|)$ using DFS. Since a fresh vertex gets added to the batch, the cost of computing the ancestor set of a fresh vertex can be easily bounded by the set of edges in $\mathcal{E}(B)$, achieving a total cost, specifically, of $O\left(\frac{1}{\gamma}|\mathcal{E}(B)|\right)$. Our charging argument allows us to bound the cost of computing ancestor sets of sampled stale vertices by charging it to the most recently sampled fresh vertex. Using the above, we provide the runtime of Algorithm 17 below and then the runtime of Algorithm 16.

Lemma 5.5.8. *With high probability, the total runtime of enumerating the ancestor sets of all sampled vertices in Algorithm 17 is $O\left(\frac{1}{\gamma}|\mathcal{E}(\mathcal{B})|\ln\rho\ln n\right)$. In other words, the total runtime of performing all iterations of Line 6 of Algorithm 17 is $O\left(\frac{1}{\gamma}|\mathcal{E}(\mathcal{B})|\ln\rho\ln n\right)$, with high probability.*

Proof. First, we calculate the runtime of enumerating the ancestor sets of each element of $\mathcal{B}$. By Lemma 5.5.2, $|\mathcal{E}(B)| \geq \gamma\sum_{v\in\mathcal{B}}|\mathcal{E}(v)|$. Hence, the amount of time to enumerate all ancestor sets of every vertex in B is at most $\frac{1}{\gamma}|\mathcal{E}(B)|$.

We employ the following charging scheme to calculate the total time necessary to enumerate the ancestor sets of all sampled stale vertices. Let u be the most recent vertex added to $\mathcal{B}$ from some bucket C_i. We charge the cost of enumerating the ancestor sets of all stale vertices sampled after u to the cost of enumerating the ancestor set of u. Since we sample at most $O(\log n)$ consecutive stale vertices from each bucket before moving to the next bucket, u gets charged with at most the work of enumerating $O(\log\rho\log n)$ vertices from the same or smaller buckets. With high probability, the largest ancestor set in bucket C_i has a size at most four times the smallest ancestor set size. Since we sample vertices in decreasing bucket size, we charge at most $O(|\mathcal{E}(v)|\log\rho\log n)$ work to v.

By our bound on the cost of enumerating all ancestor sets of vertices in $\mathcal{B}$, the additional charged cost results in a total cost of $\sum_{v\in B}|\mathcal{E}(v)|\cdot O(\log\rho\log n) = O(\frac{1}{\gamma}|\mathcal{E}(B)|\log\rho\log n)$. □

Lemma 5.5.9. *Algorithm 17 runs in $O\left(\frac{1}{\gamma}|\mathcal{E}_H(\mathcal{B})|\ln\rho\ln n + |E_H|\ln^3 n\right)$ time, with high probability.*

Proof. The runtime of Algorithm 17 consists of three parts: the time to sample and enumerate ancestor sets, the time to prune stale vertices, and the time to list schedule all vertices in $\mathcal{B}$.

By Lemma 5.5.8, the time it takes to enumerate all sampled ancestor sets is $O\left(\frac{1}{\gamma}|\mathcal{E}(\mathcal{B})|\log\rho\log n\right)$ over all iterations of the loops on Line 5 and Line 6 of Algorithm 17.

The time it takes to run Algorithm 18 is $O(|E_H|\ln^2 n)$ since obtaining the estimates for each node (by Lemma 5.4.4), creating graph H', and calculating X and Y for each node in the bucket can be done in that time. By Lemma 5.5.7, we perform $O(\ln n)$ iterations of pruning, with high probability. Thus, the total time to prune the graph is $O(|E_H|\ln^3 n)$. □

Lemma 5.5.10. *Given a graph $H = (V_H, E_H)$ where $|\mathcal{A}(v)| \leq 2\rho$ for each $v \in V_H$ and parameter $\gamma \in (0, 1/2)$, the time it takes to compute the schedule of H using Algorithm 16 is, with high probability, $O\left(\frac{1}{\gamma}|E_H|\ln\rho\ln n + |E_H|\ln_{1/4\gamma}\rho\ln^3 n + |V_H|\ln M\right)$.*

Proof. By Lemma 5.5.4, we perform $O(\log_{1/4\gamma} \rho)$ calls to Algorithm 17. Each call to Algorithm 17 requires $O\left(\frac{1}{\gamma}|\mathcal{E}_H(\mathcal{B})| \ln \rho \ln n + |E_H| \ln^3 n\right)$ time by Lemma 5.5.9. However, we know that each vertex (and edges adjacent to it) is scheduled in exactly one batch.

For each batch $\mathcal{B}$, our greedy list scheduling procedure schedules each $\mathcal{A}(v)$ for $v \in \mathcal{B}$ greedily and independently by duplicating vertices that appear in more than one ancestor set. Thus, enumerating all the ancestor sets require $O\left(\frac{1}{\gamma}|\mathcal{E}(\mathcal{B})|\right)$ time by Lemma 5.5.2. When $M > |\mathcal{B}|$, we easily schedule each list on a separate machine in $O\left(\frac{1}{\gamma}|\mathcal{E}(\mathcal{B})|\right)$ time. Otherwise, to schedule the lists, we maintain a priority queue of the machine finishing times. For each list, we greedily assign it to the machine that has the smallest finishing time. We can perform this procedure using $O(M \ln M)$ time. Since $M \leq |\mathcal{B}|$, this results in $O(|\mathcal{B}| \ln |\mathcal{B}|)$ time to assign ancestor sets to machines.

Thus, the total runtime of all calls to Algorithm 17 is

$$\begin{aligned}
&\sum_{i=1}^{\log_{1/4\gamma} \rho} O\left(\frac{1}{\gamma}|\mathcal{E}_H(\mathcal{B}_i)| \ln \rho \ln n + |E_H| \ln^3 n + \frac{1}{\gamma}|\mathcal{E}(B_i)| + |B| \ln |B|\right) \\
&= O\left(\frac{1}{\gamma}|E_H| \ln \rho \ln n + |E_H| \ln_{1/4\gamma} \rho \ln^3 n\right).
\end{aligned}$$

Then, the time it takes to perform Line 4 of Algorithm 16 is $O(1)$ per iteration. Scheduling vertices with no adjacent edges requires $|B| \ln |B| = O(|V_H| \ln M)$ time. Finally, the time it takes to remove each $v \in \mathcal{A}(\mathcal{B})$ and all edges adjacent to v from H for each batch $\mathcal{B}$ is $O(|V_H| + |E_H|)$. Doing this for $O(\ln_{1/4\gamma} \rho)$ iterations results in $O(|V_H| + |E_H| \ln_{1/4\gamma} \rho)$ time. □

5.6 Scheduling General Graphs

We now present our main scheduling algorithm for scheduling any DAG $G = (V, E)$ (Algorithm 38 in Appendix A.3). This algorithm also uses as a subroutine the procedure for estimating the number of ancestors of each vertex in G as described in Section 5.4. We use the estimates to compute the small subgraphs which we pass into Algorithm 16 to schedule. We produce the small subgraphs by setting the cutoff for the estimates to be $\frac{4}{3}\rho$. This produces small graphs where the number of ancestors of each vertex is upper bounded by 2ρ, with high probability. We present a simplified algorithm below in Algorithm 19.

Quality of the Schedule and Running Time Let OPT be the length of the optimal schedule. We first give two bounds on OPT, and then relate them to the length of the schedule found by our algorithm. A detailed set of proofs is provided in Section 5.6.1.

Our main algorithm, Algorithm 19, partitions the vertices of G into small subgraphs $H \in \mathcal{H}$. It does so based on estimates of ancestor set sizes. We first lower bound OPT by working with exact ancestor set sizes. Since the schedule produced by our algorithm cannot have length smaller than OPT, this process also provides a lower bound on our

Algorithm 19 ScheduleGeneralGraph(G)

Input A directed acyclic task graph $G = (V, E)$.
Output A schedule of the input graph $G = (V, E)$ on M processors.

Let $\mathcal{H} \leftarrow \varnothing$ represent a list of small subgraphs that we will build.
while G is not empty **do**
    Let V_H be the set of vertices in G where $\hat{a}(v) \leq \frac{4}{3}\rho$ for each $v \in V_H$.
    Compute edge set E_H to be all edges induced by V_H.
    Add $H = (V_H, E_H)$ to $\mathcal{H}$.
    Remove V_H and all incident edges from G.
for $H \in \mathcal{H}$ in the order they were added **do**
    Call ScheduleSmallSubgraph(H) to obtain a schedule of H. [Algorithm 16]

schedule length. Then, we show that Algorithm 19 does not output more small subgraphs than the number of subgraphs produced by working with exact ancestor set sizes, with high probability.

The crucial fact in obtaining our final runtime is that producing the estimates of the number of ancestors of each vertex requires $\widetilde{O}(|V| + |E|)$ time *in total* over the course of finding all small subgraphs. Together, these facts allow us to obtain Theorem 5.6.5.

5.6.1 Quality of Schedule Produced by Main Algorithm

Assuming we are working with exact ancestor set sizes, we would wind up with vertex sets $V_1 \triangleq \{v \in V : |\mathcal{A}(v)| \leq \rho\}$ and, inductively, for $i > 1$, $V_i \triangleq \{v \in V \setminus \bigcup_{j=1}^{i-1} V_j : |\mathcal{A}(v) \setminus \bigcup_{j=1}^{i-1} V_j| \leq \rho\}$. Let L be the maximum index such that V_L is nonempty.

The following lemma follows a similar argument as that found in Lepere-Rapine [LR02] (although we have simplified the analysis). We repeat it here for completeness.

Lemma 5.6.1. $\mathrm{OPT} \geq (L-1)\rho$.

Proof. We show by induction on i that in any valid schedule, there exists a job $v \in V_i$ that cannot start earlier than time $(i-1)\rho$. Given that, the job in V_L starts at time at least $(L-1)\rho$ in OPT, proving the lemma.

The base case of $i = 1$ is trivial. For the induction step, consider a job $v \in V_{i+1}$. This job has at least ρ ancestors in V_i (call this set $A = \mathcal{A}(v) \cap V_i$), since if it had less, v would be in V_i itself. All jobs in A start no earlier than $(i-1)\rho$ by the induction hypothesis. There are two cases. If all of the jobs in A are executed on the same machine as v, then it would take at least ρ units of time for them to finish before v can start. If at least one job in A is executed on a different machine than v, then it would take ρ units of time to communicate the result. In either case, v would start later than the first job in A by at least ρ, and thus no earlier than $i \cdot \rho$. □

Lemma 5.6.2. $\mathrm{OPT} \geq |V|/M$.

Proof. Every job has to be scheduled on at least one machine, and the makespan is at least the average load on any machine. □

We show that our general algorithm only calls the schedule small subgraph procedure at most L times, w.h.p.

Lemma 5.6.3. *With high probability, Algorithm 38 calls Algorithm 16 at most L times on input graph $G = (V, E)$.*

Proof. By construction, the V_i's are inductively defined by stripping all vertices with ancestor sets at most ρ in size. With high probability, our estimates $\hat{a}(v)$ are at most $\frac{4}{3}|\mathcal{A}(v)|$. Line 7 of Algorithm 38 only takes vertex v into the subgraph H if $\hat{a}(v) \leq \frac{4}{3}\rho$. By Lemma 5.4.4, $\frac{2}{3}|\mathcal{A}(v)| \leq \hat{a}(v) \leq \frac{4}{3}|\mathcal{A}(v)|$. Then, $|\mathcal{A}(v)| \leq \frac{3}{2}\hat{a}(v) \leq \frac{3}{2} \cdot \frac{4}{3}\rho = 2\rho$. Furthermore, since $|\mathcal{A}(v)| \geq \frac{3}{4} \cdot \hat{a}(v)$, if $\hat{a}(v) = \frac{4}{3}\rho$, then $|\mathcal{A}(v)| \geq \rho$. Hence, all vertices with height ρ are added into H, with high probability. Taken together, this means that all vertices of V_i (even if their ancestor sets are maximally overestimated) are contained in the small graphs H produced by iterations one through i of Line 7 of Algorithm 38. Since V_L was chosen to be the last non-empty set, we know our algorithm runs for at most L iterations, with high probability. □

Theorem 5.6.4. *Algorithm 38 produces a schedule of length at most $O\left(\frac{\ln \rho}{\ln \ln \rho}\right) \cdot (\mathrm{OPT} + \rho)$.*

Proof. In Algorithm 16, by Lemma 5.5.5, the schedule length obtained from any small subgraph H is $\frac{|V_H|}{\gamma M} + 12\rho \log_{1/4\gamma}(2\rho)$.

Let $\mathcal{H}$ be the set of all small subgraphs Algorithm 38 sends to Algorithm 16 to be scheduled. By Lemma 5.6.3, there are at most L of them. Each vertex trivially appears in at most one subgraph. Then the total length of our schedule is given by

$$\begin{aligned}\sum_{H \in \mathcal{H}} \left(\frac{|V_H|}{\gamma M} + 12\rho \log_{1/4\gamma}(2\rho) \right) &= \sum_{H \in \mathcal{H}} \frac{|V_H|}{\gamma M} + \sum_{H \in \mathcal{H}} 12\rho \log_{1/4\gamma}(2\rho) \\ &\leq \frac{|V|}{\gamma M} + L\left(12\rho \log_{1/4\gamma}(2\rho)\right).\end{aligned}$$

By Lemmas 5.6.1 and 5.6.2, this last quantity is upper bounded by

$$OPT \cdot \left(\frac{1}{\gamma} + 12 \log_{1/4\gamma}(2\rho) \right) + \rho \cdot 12 \log_{1/4\gamma}(2\rho)$$

Setting $\gamma = 1/\sqrt{\ln \rho}$ gives our bound of $(OPT + \rho) \cdot O(\frac{\ln \rho}{\ln \ln \rho})$. □

5.6.2 Running Time of the Main Algorithm

We prove the following theorem regarding the runtime of Algorithm 38 which uses Algorithm 16 as a subroutine.

Theorem 5.6.5. *On input graph $G = (V, E)$, Algorithm 38 produces a schedule of length at most $O\left(\frac{\ln \rho}{\ln \ln \rho} \cdot (\mathrm{OPT} + \rho)\right)$ and runs in time $O\left(n \ln M + \frac{m \ln^3 n \ln \rho}{\ln \ln \rho}\right)$, with high probability.*

Proof. By Lemma 5.6.3, Algorithm 17 is called at most L times. Then, since each vertex is in at most one small subgraph (and hence each edge is in at most one small subgraph), the total runtime for all the calls (by Lemma 5.5.10) is

$$\sum_{i=1}^{L} O\left(\frac{1}{\gamma}|E_{H_i}|\ln\rho\ln n + |E_{H_i}|\ln_{1/4\gamma}\rho\ln^3 n + |V_{H_i}|\ln M\right)$$
$$= O\left(\frac{1}{\gamma}|E|\ln\rho\ln n + |E|\ln_{1/4\gamma}\rho\ln^3 n + |V|\ln M\right).$$

Furthermore, each iteration of Line 7 requires estimating $\hat{a}(v)$ for a set of vertices v, adding v to H, and checking all successors of v. First, we show that $\hat{a}(v)$ is computed at most twice for each vertex in V, and then, we show that the rest of the steps are efficient.

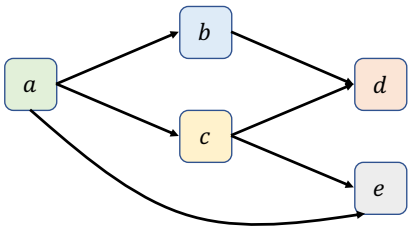
Each vertex contained in the queue, Q, in Algorithm 17, either does not have any ancestors, or all of its ancestors are in H (the current subgraph). If a vertex $v \in Q$ was not added to H during iteration i, then it must have at least one ancestor in iteration i and no ancestors in iteration $i+1$. Since v has no ancestors in iteration $i+1$, it must be added to H_{i+1}. The time it takes to compute the estimate for one vertex is $O(\ln^2 n)$. Thus, the total time it takes to compute the estimate of the number of ancestors of all vertices is $O\left(m\ln^2 n\right)$. Adding v to H and checking all successors can be done in $O(m)$ time. Finally removing each $v \in H$ from G can be done in $O(m)$ time.

As earlier, we use $\gamma = \sqrt{\ln\rho}$, so the total runtime summing the above can be upper bounded by $O\left(n\ln M + \frac{m\ln^3 n\ln\rho}{\ln\ln\rho}\right)$. Thus, the algorithm produces a schedule of length $O\left(\frac{\ln\rho}{\ln\ln\rho}\cdot(\mathrm{OPT}+\rho)\right)$ and the total runtime of the algorithm is $O\left(n\ln M + \frac{m\ln^3 n\ln\rho}{\ln\ln\rho}\right)$, with high probability. □

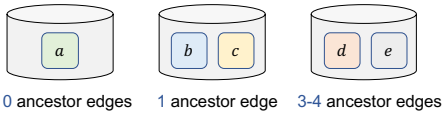
Theorem 5.6.5 gives the main result stated informally in Theorem 5.1.1 of the introduction.

5.7 Open Questions

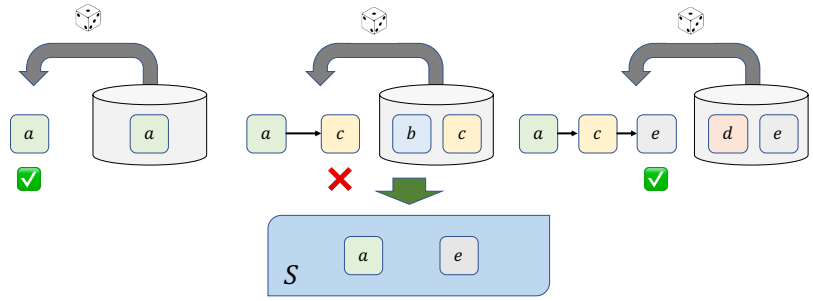
Our results so far only apply to scheduling with duplication. In [MRS+20], a polynomial-time reduction is presented that transforms a schedule with duplication into one without duplication (with a polylogarithmic increase in makespan). However, this reduction involves constructing an auxiliary graph of possibly $\Omega(\rho^2)$ size, and thus does not lend itself easily to a near-linear time algorithm. It would be interesting to see if a near-linear time reduction could be found.



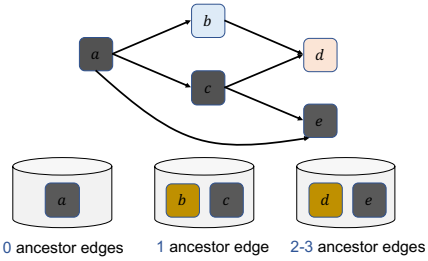
(a) Input small subgraph.



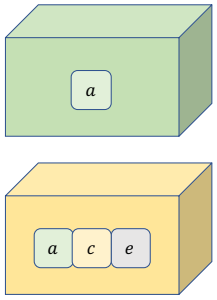
(b) Vertices are bucketed according to the estimate of the number of edges in the induced subgraph of its ancestors.



(c) Vertices are uniformly at random sampled from buckets. Then, vertices which have sufficiently many ancestors and ancestor edges not in S are added to S.



(d) Vertices which are in S or have a large proportion of ancestors or ancestor edges in S are pruned from buckets. b and d are pruned in this example.



(e) Vertices in S and all ancestors are scheduled by duplicating ancestors and list scheduling.

Figure 5-2: Overview of our scheduling small subgraphs algorithm. We choose $\gamma = 2/3$ here for illustration purposes but in our algorithms $\gamma < 1/2$.

Part III

Dynamic Graph Algorithms

Overview

Six Hollywood stars form a social group that has very special characteristics. Every two stars in the group either mutually love each other or mutually hate each other. There is no set of three individuals who mutually love one another. Prove that there is at least one set of three individuals who mutually hate each other.
The Colossal Book of Short Puzzles and Problems [Gar06]

This part of the thesis presents new dynamic and batch-dynamic graph algorithms in the sequential, parallel, and distributed models.

Parallel Batch-Dynamic k-Core Decomposition Chapter 6 presents a parallel batch-dynamic k-core decomposition algorithm that maintains the coreness of each vertex on batches of updates. Maintaining a k-core decomposition quickly in a dynamic graph is an important problem in many applications, including social network analytics, graph visualization, centrality measure computations, and community detection algorithms. The main challenge for designing efficient k-core decomposition algorithms is that a single change to the graph can cause the decomposition to change significantly.

We present the first parallel batch-dynamic algorithm for maintaining an approximate k-core decomposition that is efficient in both theory and practice. Given an initial graph with m edges, and a batch of B updates, our algorithm maintains a $(2+\varepsilon)$-approximation of the coreness values for all vertices (for any constant $\varepsilon > 0$) in $O(|\mathcal{B}|\log^2 m)$ amortized work and $O(\log^2 m \log\log m)$ depth (parallel time) with high probability. Our algorithm also maintains a low out-degree orientation of the graph in the same bounds.

Specifically, we show:

- Provided an input graph with m edges, and a batch of updates $\mathcal{B}$, our algorithm maintains a $(2+\varepsilon)$-approximation of the coreness values for all vertices (for any constant $\varepsilon > 0$) in $O(|\mathcal{B}|\log^2 m)$ amortized work and $O(\log^2 m \log\log m)$ depth with high probability, using $O(n\log^2 m + m)$ space. [Theorem 6.1.2]
- For a graph with m edges, there is a *static* algorithm that finds an $(2+\varepsilon)$-approximate k-core decomposition in $O(m)$ expected work and $O(\log^3 n)$ depth, *whp*. [Theorem 6.1.1]

We implemented and experimentally evaluated our algorithm on a 30-core machine with two-way hyper-threading on 11 graphs of varying densities and sizes. Compared to the state-of-the-art algorithms, our algorithm achieves up to a 114.52x speedup against the best exact, multicore implementation and up to a 497.63x speedup against the best sequential, approximate algorithm, obtaining results for graphs that are orders-of-magnitude larger than those used in previous studies.

In addition, we present the first approximate static k-core algorithm with linear work and polylogarithmic depth. We show that on a 30-core machine with two-way hyper-threading, our implementation achieves up to a 3.9x speedup in the static case over the previous state-of-the-art parallel algorithm.

Fully Dynamic $(\Delta+1)$-Vertex Coloring Chapter 7 presents an improved algorithm for dynamic $(\Delta+1)$-vertex coloring in the sequential model. The problem of $(\Delta+1)$-vertex coloring a graph of maximum degree Δ has been extremely well-studied over the years in various settings and models including the static, distributed, and parallel models and for special classes of graphs. Surprisingly, for the dynamic setting, almost nothing was known until recently. In SODA'18, Bhattacharya, Chakrabarty, Henzinger and Nanongkai devised a randomized algorithm for maintaining a $(\Delta+1)$-coloring with $O(\log\Delta)$ expected amortized update time [BCHN18]. This chapter presents an improved randomized algorithm for $(\Delta+1)$-coloring that achieves $O(1)$ amortized update time and show that this bound holds not only in expectation but also with high probability. Specifically, in this chapter, we prove the following:

- There is a randomized algorithm for maintaining a $(\Delta+1)$-vertex coloring in a dynamic graph that, given any sequence of t edge updates, takes total time $O(n\log n+n\Delta+t)$ in expectation and with high probability. For $t=\Omega(n\log n+n\Delta)$, we obtain $O(1)$ amortized update time in expectation and with high probability. [Theorem 7.1.2]

Our starting point is the state-of-the-art randomized algorithm for maintaining a maximal matching (Solomon, FOCS'16). We carefully build on the approach of Solomon, but, due to inherent differences between the maximal matching and $(\Delta+1)$-coloring problems, we need to deviate significantly from it in several crucial and highly nontrivial points.

Although this algorithm is in the sequential model, one may perhaps use the techniques for parallelizing level data structures (presented in Chapter 6) to parallelize this algorithm for practical implementations.

Independent concurrent work: Independently of our work, Henzinger and Peng [HP19] have obtained an algorithm for $(\Delta+1)$-vertex coloring with $O(1)$ *expected amortized update time*. Note that our work achieves $(\Delta+1)$-vertex coloring with $O(1)$ amortized update time not only in expectation, but also with *with high probability*, and uses very different techniques.

Parallel Batch-Dynamic k-Clique Counting Chapter 8 presents new batch-dynamic algorithms for the k-clique counting problem for constant k. We study this problem in the parallel setting, where the goal is to obtain algorithms with low (polylogarithmic) depth. Our first result is a new parallel batch-dynamic triangle counting algorithm with $O(|\mathcal{B}|\sqrt{|\mathcal{B}|+m})$ amortized work and $O(\log^*(|\mathcal{B}|+m))$ depth with high probability, and $O(|\mathcal{B}|+m)$ space for a batch of $|\mathcal{B}|$ edge insertions or deletions. Our second result is an algebraic algorithm based on parallel fast matrix multiplication. Assuming that a parallel fast matrix multiplication algorithm exists with parallel matrix multiplication constant ω_p, the same algorithm solves dynamic k-clique counting with

$O\left(\min\left(|\mathcal{B}|m^{\frac{(2k-1)\omega_p}{3(\omega_p+1)}}, (|\mathcal{B}|+m)^{\frac{2(k+1)\omega_p}{3(\omega_p+1)}}\right)\right)$ amortized work and $O(\log(|\mathcal{B}|+m))$ depth with high probability, and $O\left((|\mathcal{B}|+m)^{\frac{2(k+1)\omega_p}{3(\omega_p+1)}}\right)$ space. Using a recently developed parallel k-clique counting algorithm, we also obtain a simple batch-dynamic algorithm for k-clique counting on graphs with arboricity α running in $O(|\mathcal{B}|(m+|\mathcal{B}|)\alpha^{k-4})$ expected work and $O(\log^2 n)$ depth with high probability, and $O(m+\Delta)$ space. Finally, we present a multi-core CPU implementation of our parallel batch-dynamic triangle counting algorithm. On a 72-core machine with two-way hyper-threading, our implementation achieves 36.54–74.73x parallel speedup, and in certain cases achieves significant speedups over existing parallel algorithms for the problem, which are not theoretically-efficient.

Specifically, in Chapter 8, we show:

- A parallel batch-dynamic algorithm that takes $O(|\mathcal{B}|\sqrt{|\mathcal{B}|+m})$ amortized work and $O(\log^*(|\mathcal{B}|+m))$ depth, *whp*. [Theorem 8.3.1]
- Using the best currently known parallel matrix multiplication algorithm [Wil12, LG14, AW21], our algorithm dynamically maintains the number of k-cliques in $O\left(\min\left(|\mathcal{B}|m^{0.469k-0.235}, (|\mathcal{B}|+m)^{0.469k+0.469}\right)\right)$ amortized work and $O(\log(|\mathcal{B}|+m))$ depth, *whp*. This is also faster than the best-known sequential algorithm when $k > 9$. [Corollary 8.5.2]
- A parallel batch-dynamic algorithm that runs in $O(|\mathcal{B}|(m+|\mathcal{B}|)\alpha^{k-4})$ expected work, $O(\log^2 n)$ depth w.h.p., and $O(m+|\mathcal{B}|)$ space. [Theorem 8.2.1]

Bibliographic Information The results in Part III are based off the following works:

1. [LSY+21] Quanquan C. Liu, Jessica Shi, Shangdi Yu, Laxman Dhulipala, and Julian Shun. Parallel batch-dynamic k-core decomposition. (Chapter 6)
2. [BGK+19] Sayan Bhattacharya, Fabrizio Grandoni, Janardhan Kulkarni, Quanquan C. Liu, and Shay Solomon. Fully dynamic $(\Delta+1)$-coloring in constant update time. (Chapter 7)
3. [DLSY21] Laxman Dhulipala, Quanquan C. Liu, Julian Shun, and Shangdi Yu. Parallel batch-dynamic k-clique counting. (Chapter 8)

Chapter 6

Parallel Batch-Dynamic k-Core Decomposition

This chapter presents results from the paper titled, "Parallel Batch-Dynamic k-Core Decomposition" that the thesis author coauthored with Jessica Shi, Shangdi Yu, Laxman Dhulipala, and Julian Shun [LSY+21]. This paper is currently under submission at the time of the writing of this thesis.

6.1 Introduction

Discovering the structure of large-scale networks is a fundamental problem for many areas of computing, including social network analysis, computational biology, and spam and fraud detection, among many others. One of the key challenges is to detect communicaties in which individuals (or vertices) have close ties with one another, and also to understand how well-connected a particular individual is to the community. For example, a social network provider may need this information to determine how many friends of a user needs to drop out before that user also leaves. The well-connectedness of a node or a group of nodes is naturally captured by the concept of a k-core or, more generally, the k-core decomposition; hence, this particular problem and its variants have been widely studied in the past and also more recently in the machine learning [AHDBV05, ELM18, GLM19], database [CZL+20, LZZ+19, ESTW19, BGKV14, MMSS20], graph analytics [KBST15, KM17a, DBS17, DBS18b], and other communities [GBGL20, KBST15, LYL+19, SGJS+13].

Given an undirected graph G, with n vertices and m edges, the k-core of the graph is the maximal subgraph $H \subseteq G$ such that the induced degree of every vertex in H is at least k. The k-core decomposition of the graph is defined as a partition of the graph into layers such that a vertex v is in layer k if it belongs to a k-core but not a $(k+1)$-core, which induces a natural hierarchical clustering on the input graph. Many well-known algorithms for k-core decomposition are inherently sequential. The classic algorithm for finding such a decomposition is to iteratively select and remove all vertices v with smallest degree from the graph until the graph is empty. Unfortunately, the length of the sequential dependencies (the *depth*) of such a process can be $\Omega(n)$ given a graph with n vertices. Due to the potentially large depth, this algorithm cannot fully take advantage of paral-

lelism on modern machines, and can therefore be too costly to run on large graphs. As k-core decomposition is a P-complete problem [AM84], it is unlikely that there is a parallel algorithm with polylogarithmic depth for this problem. To obtain parallel methods for this problem, we relax the condition of obtaining an *exact* k-core decomposition to one of obtaining a close *approximate* k-core decomposition.

Parallel Static Approximate k-Core Decomposition In this chapter, we present a novel parallel static approximate k-core decomposition algorithm based on a relaxed version of the peeling algorithm of [DBS17] that achieves both optimal linear work and has polylogarithmic depth. In contrast to existing parallel approximate k-core algorithms [GLM19, ELM18] which are designed for distributed memory, our algorithm is designed for shared-memory multicore machines, which have been shown to be able to process the largest graphs with hundreds of billions of edges efficiently [DBS18b, MIM15]. To the best of our knowledge, ours is the first parallel algorithm to achieve work-efficiency and polylogarithmic depth. Our bounds are summarized in Theorem 6.1.1.

Theorem 6.1.1. *For a graph with m edges,[1] for any constant $\varepsilon > 0$, there is an algorithm that finds an $(2+\varepsilon)$-approximate k-core decomposition in $O(m)$ expected work and $O(\log^3 n)$ depth with high probability,[2] using $O(m)$ space.*

Parallel Batch-Dynamic Approximate k-Core Decomposition Because of the rapidly changing nature of today's graphs, it is inefficient to recompute the k-core decomposition of the graph from scratch on every update. For this purpose, dynamic k-core decompositions are especially useful. In this chapter, we design an approximate k-core algorithm with strong theoretical guarantees for the *batch-dynamic* setting, where updates to the graph are provided in *batches* that can be processed in parallel. Our batch-dynamic algorithm efficiently maintains the k-core decomposition in parallel given batches of updates, and uses a level data structure inspired by [BHNT15, HNW20] to maintain a partition of the vertices satisfying certain degree properties. Our algorithm takes $O(\log^2 m)$ amortized work per update, matching the best sequential algorithm [SCS20], while achieving polylogarithmic depth at the same time. The bounds for our algorithms are as follows.

Theorem 6.1.2. *Provided an input graph with m edges, and a batch of updates $\mathcal{B}$, our algorithm maintains a $(2+\varepsilon)$-approximation of the coreness values for all vertices (for any constant $\varepsilon > 0$) in $O(|\mathcal{B}|\log^2 m)$ amortized work and $O(\log^2 m \log\log m)$ depth with high probability, using $O(n\log^2 m + m)$ space.*

Moreover, our parallel batch-dynamic algorithm can be used to maintain low out-degree orientations and approximate densest subgraphs in the same complexity bounds.

Experimental Evaluation In addition to our theoretical contributions, we also provide optimized multicore implementations of both our static and batch-dynamic algorithms. We compare the performance of our algorithms with state-of-the-art algorithms

[1] Our bounds in this chapter assume $m = \Omega(n)$ for simplicity, although our algorithms work even if $m = o(n)$.

[2] ***With high probability (whp)*** is defined as with probability at least $1 - 1/n^c$ for any $c \geq 1$.

on a variety of real-world graphs using a 30-core machine with two-way hyper-threading. Our parallel static approximate k-core algorithm achieves a 2.8–3.9x speedup over the fastest parallel exact k-core algorithm [DBS17], and achieves a 14.76–36.07x self-relative speedup. We show that our parallel batch-dynamic k-core algorithm achieves up to a 497.63x speedup over the state-of-the-art sequential dynamic k-core algorithm [SCS20], while achieving comparable accuracy. Furthermore, our batch-dynamic algorithm is able to outperform our static approximate k-core algorithm by up to 75.94x on small batch sizes. We also achieve up to a 114.52x speedup over the state-of-the-art parallel batch-dynamic k-core algorithm of Hua et al. [HSY+20].

Related Work The k-core decomposition of a graph and its related concepts of arboricity, low out-degree orientation, and densest subgraph are widely used in machine learning applications, including community detection [MPP+15, BZ11], analyzing social network dynamics [BKL+15], visualizing large-scale complex networks [AHDBV05], analyzing protein networks [ASM+06], approximating network centrality measures [HJMA07], and many more. In addition, low out-degree orientations can lead to efficient graph algorithms for sparse graphs [NS15, SW20b].

Many parallel k-core algorithms have been designed for the static setting (e.g., [PKT14, EM13, MPM13, DDZ14, DBS17, KM17a, MCT20, THCG18]). The algorithms of [GLM19, ELM18] are approximate algorithms that achieve a logarithmic or sub-logarithmic number of rounds in the distributed MPC model. However, the local computation in their algorithms is free in the MPC model, but have linear depth in the shared-memory setting, which is worse than the polylogarithmic depth bound that we achieve. [GLM19, ELM18] do not report running times in their paper, and hence it is difficult to compare actual performance. We note that we are able to process graphs used in their papers using a single commodity multicore machine.

Several generalizations of the k-core problem, such as k-truss, and the (r, s) nucleus decomposition problem have been intensely studied in recent years [WC12, CCC14, Zou16, KM17b, SLA+17, CLS+20, SSPc17, SSP18, ESTW20]. Another recent line of work studies k-core-like computations in bipartite graphs [SS20, WLQ+20, LKPR20, SP18, LYL+20].

There has been significant interest in obtaining fast and practical dynamic k-core algorithms. Dynamic algorithms have been developed for both the sequential [LYM14, SGJS+16, ZYZQ17, WQZ+19, LYM14, SCS20, LZL+21] and parallel [HSY+20, JWY+18, ABMV16] settings. However, to the best of our knowledge, there are no parallel dynamic k-core algorithms with polylogarithmic depth, which our algorithm achieves. Several dynamic algorithms have also been developed for the closely-related k-truss problem [HCQ+14, AZ17, ZY19, LYS+21].

A recent paper by Sun et. al. [SCS20] provides a *sequential*, dynamic k-core decomposition algorithm that gives a $(4+\delta)$-approximation in $\text{poly}\log n$ time. Their algorithm is inherently sequential and parallelizing their algorithm requires non-trivial changes in theory and in practice. The most recent, state-of-the-art parallel batch-dynamic algorithm by Hua et al. [HSY+20] relies on the concept of a *joint edge set* whose insertion and removal determines the core numbers of the nodes; however, their algorithm does not achieve polylogarithmic depth.

Related dynamic problems to k-core decompositions have been recently studied in the theory community [BHNT15, HNW20, SW20b]. Such problems include densest subgraph [BHNT15, SW20a] and low out-degree orientations [HNW20, SW20b]. None of these aforementioned works proved guarantees regarding the k-core decomposition that can be maintained via a level data structure. Furthermore, no prior work have provided implementations of their structures or performed experiments on the practical efficiency of these structures.

6.2 Preliminaries

In this chapter, we use the definition of k-core decomposition presented in Definition 2.1.3 and its related properties. The problem we study is the following:

Problem Definition. Given a graph $G = (V, E)$ and a sequence of batches of edge insertions and deletions, $\mathcal{B}_1, \ldots, \mathcal{B}_N$, where $\mathcal{B}_i = (E^i_{delete}, E^i_{insert})$, the goal is to efficiently maintain a $(2+\varepsilon)$-approximate k-core decomposition (for any constant $\varepsilon > 0$) after applying each batch $\mathcal{B}_i$ (in order) on G. In other words, let $G_i = (V, E_i)$ be the graph after applying batches $\mathcal{B}_1, \ldots, \mathcal{B}_i$ and suppose we have a $(2+\varepsilon)$-approximate k-core decomposition on G_i; then, for $\mathcal{B}_{i+1}$, our goal is to efficiently find a $(2+\varepsilon)$-approximate k-core decomposition of $G_{i+1} = \left(V, (E_i \cup E^{i+1}_{insert}) \setminus E^{i+1}_{delete}\right)$.

All notations used in this chapter are summarized in Table 6.1.

6.3 Batch-Dynamic $(2+\varepsilon)$-Approximate k-Core Decomposition

In this section, we describe our parallel, batch-dynamic algorithm for maintaining an $(2+\varepsilon)$-approximate k-core decomposition (for any constant $\varepsilon > 0$) and prove its theoretical efficiency.

6.3.1 Algorithm Overview

We provide a *parallel level data structure (PLDS)* that maintains a $(2+\varepsilon)$-approximate k-core decomposition and low out-degree orientation that builds off the sequential level data structures (LDS) of [BHNT15, HNW20]. Our algorithm achieves $O(\log^2 m)$ amortized work per update and $O(\log^2 m \log\log m)$ depth *whp*. We also present a deterministic version of our algorithm that achieves the same work bound with $O(\log^3 m)$ depth. Our data structure can also handle batches of vertex insertions/deletions, which we discuss in Section 6.3.4.

As in the case of [HNW20], our data structure can handle *changing arboricity* that is not known a priori to the algorithm. This means that our data structure can handle the case when the arboricity α of the underlying graph changes throughout the execution

Symbol	Meaning
$G = (V, E)$	undirected/unweighted graph
n, m	number of vertices, edges, resp.
α	current arboricity of graph
K	number of levels in PLDS
$N(v)$ (resp. $N(S)$)	set of neighbors of vertex v (resp. vertices S)
$\mathsf{dl}(v)$	*desire-level* of vertex v
ℓ	a level (starting with level $\ell = 0$)
$\ell(v)$	current level of vertex v
g_i	set of levels in group i (starting with g_0)
V_ℓ	set of vertices in level ℓ
Z_ℓ	set of vertices in levels $\geq \ell$
$g(v)$	*group number* of vertex v
$gn(\ell)$	index i where level $\ell \in g_i$
$k(v)$	coreness of v
$\hat{k}(v)$	estimate of the coreness of v
$\mathsf{up}(v)$	*up-degree* of v
$\mathsf{up}^*(v)$	*up*-degree* of v
λ	a constant where $\lambda > 0$
δ	a constant where $\delta > 0$

Table 6.1: Table of notations used in this chapter.

of batches of updates and successfully maintains approximations of the coreness of each node with respect to the *current* arboricity.

The level data structure consists of a partition of the vertices of the graph into $K = O(\log^2 m)$ ***levels***. [3] The levels are partitioned into equal-sized ***groups*** of consecutive levels. Vertices move up and down levels depending on the type of edge update incident to the vertex. Rules governing the induced degrees of vertices to neighbors in different levels determine whether a vertex moves up or down. Using information about the assigned level of a vertex, we obtain a $(2+\varepsilon)$-approximation on the coreness of the vertex.

Fig. 6-1 shows an illustration of this data structure. There are $\Theta(\log n)$ groups, each with $\Theta(\log n)$ levels (labeled on the right). Each vertex is in exactly one level of the structure and moves up and down by some movement rules. We describe our parallel level data structure in more detail in Section 6.3.3.

6.3.2 Low Out-Degree Orientations and LDS

Low out-degree orientations have been used in many efficient static and dynamic algorithms for graphs (e.g., [SW20b, ELS10, BE10, GP11]). Our work is based on the sequential

[3] A more refined analysis shows that we only need $O(\log \Delta \log m)$ levels, where Δ is the current maximum degree in the graph.

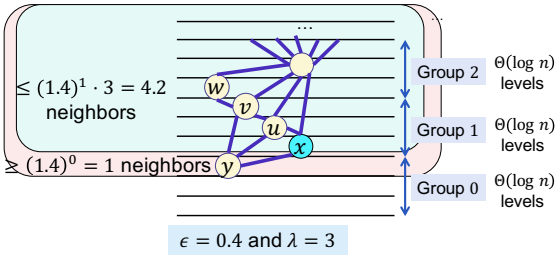


Figure 6-1: Example of invariants maintained by the PLDS for $\delta = 0.4$ and $\lambda = 3$.

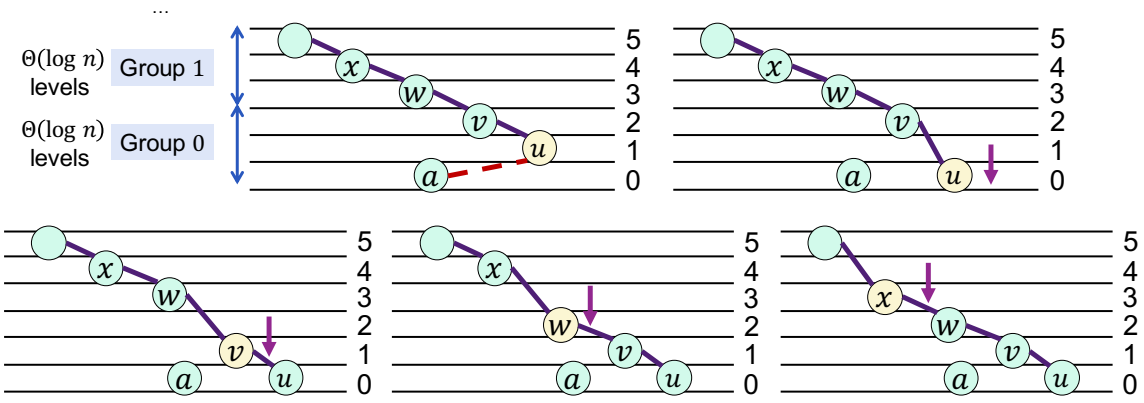


Figure 6-2: Example of a cascade of vertex movements caused by edge deletion.

level data structures (LDS) of [BHNT15, HNW20], which maintain a low out-degree orientation under dynamic updates. Within their LDS, a vertex moves up or down levels one by one, meaning that a vertex v (incident to an edge update) first checks whether an invariant is violated, and then may move up or down one level. Then, the vertex checks the invariants and repeats.

Unfortunately, such a procedure can be slow in practice. Specifically, a vertex that moves one level could cause a cascade of vertices to move one level (illustrated in Example 6.3.1). Then, if the vertex moves again, the same cascade of movements may occur.

Example 6.3.1. *In Fig. 6-2, suppose a vertex u at level 1 is incident to an edge deletion (a, u) (dashed edge in red) and must move down one level. This in turn could cause another one of its neighbors v at level 2 to move down one level, leading to a cascade of vertices which must move down (w at level 3, then x at level 4, etc.). If u could move down one additional level, it may cause another cascade of movements.*

Furthermore, any trivial parallelization of the LDS to support a batch of updates will run into race conditions and other issues, requiring the use of locks which blows up the runtime in practice. We must also be careful not to perform unnecessary work; one example is a vertex that moves up due to edge insertions but then later moves down due to edge deletions (from the same batch).

Thus, our PLDS algorithm solves several challenges posed by the sequential algorithm.

Provided a batch $\mathcal{B}$ of edge insertions and deletions: **(1)** our algorithm processes the levels in a careful order that yields provably low depth for batches of updates; **(2)** our algorithm processes insertions and deletions in separate batches to avoid excess work; **(3)** our insertion algorithm processes vertices on each level at most once, which is key to bounding the depth–after a vertex moves up from level ℓ, no future step in the algorithm requires a vertex to move up from level ℓ; and **(4)** our deletion algorithm moves vertices to their final level in one step. In other words, a vertex moves at most once in a batch of deletions before a new batch arrives.

6.3.3 Detailed PLDS Algorithm

As mentioned previously, the vertices of the input graph $G = (V, E)$ are partitioned across K ***levels***. For each level $\ell = 0, \ldots, K-1$, let V_ℓ be the set of vertices that are currently assigned to level ℓ. Let Z_ℓ be the set of vertices in levels $\geq \ell$. Provided a constant $\delta > 0$, the levels are partitioned into ***groups*** $g_0, \ldots, g_{\lceil \log_{(1+\delta)} m \rceil}$, where each group contains $\lceil \log_{(1+\delta)} m \rceil$ consecutive levels. Each $\ell \in \left[i \lceil \log_{(1+\delta)} m \rceil, (i+1) \lceil \log_{(1+\delta)} m \rceil - 1 \right]$ is a level in group i. Our data structure consists of $K = O(\log^2 m)$ levels. The PLDS satisfies the following invariants, which also govern how the data structure is maintained. The invariants assume a given constant $\delta > 0$ and a constant $\lambda > 0$. [4] The below invariants follow invariants defined in [HNW20, BHNT15].

Invariant 1 (Degree Upper Bound)**.** *If vertex $v \in V_\ell$, level $\ell < K$ and $\ell \in g_i$, then v has at most $(2 + 3/\lambda)(1+\delta)^i$ neighbors in Z_ℓ.*

Invariant 2 (Degree Lower Bound)**.** *If vertex $v \in V_\ell$, level $\ell > 0$, and $\ell - 1 \in g_i$, then v has at least $(1+\delta)^i$ neighbors in $Z_{\ell-1}$.*

All vertices with no neighbors are placed in level 0. An example partitioning of vertices and the maintained invariants is shown in Example 6.3.2.

Example 6.3.2. *In Fig. 6-1, where $\delta = 0.4$ and $\lambda = 3$, vertex x (blue) is on level 3 in group 1. This means that by Invariant 1, it has at most $(2 + 3/\lambda)(1+\delta)^1 = 4.2$ neighbors at level 3 and above. In Fig. 6-1, the green box highlights all levels ≥ 3; indeed x has 2 neighbors in levels ≥ 3, satisfying Invariant 1. By Invariant 2, x has more than $(1+\delta)^0 = 1$ neighbor in levels ≥ 2 (note that level 2 is in group 0). In the example, x has 3 neighbors, satisfying the invariant (levels ≥ 2 are highlighted by the pink box).*

Let $\ell(v)$ be the level that v is currently on. We define the ***group number***, $g(v)$, of a vertex v to be the index i of the group g_i where $\ell(v) \in g_i$. Similarly, we define $gn(\ell) = i$ to be the group number for level ℓ where $\ell \in g_i$. We define the ***up-degree***, $\mathsf{up}(v)$, of a vertex v to be the number of its neighbors in $Z_{\ell(v)}$ (up-neighbors), and ***up*-degree***, $\mathsf{up}^*(v)$, to be the number of its neighbors in $Z_{\ell(v)-1}$ (up^*-neighbors). These two notions of induced degree correspond to the requirements of the two invariants that our data structure maintains. Lastly, the ***desire-level*** $\mathsf{dl}(v)$ of a vertex v is the *closest level to the current level of the vertex* that satisfies both Invariant 1 and Invariant 2.

[4] The magnitude of both δ and λ impact the approximation factor and the work, practically.

Algorithm 20 Update(B)

Input: A batch of edge updates B.

1: Let B_{ins} = all edge insertions in B, and B_{del} = all edge deletions in B.
2: Call RebalanceInsertions(B_{ins}). [Algorithm 21]
3: Call RebalanceDeletions(B_{del}). [Algorithm 22]

Definition 6.3.3 (Desire-level)**.** *The* desire-level, $\mathsf{dl}(v)$, *of vertex* v *is the level* ℓ' *which minimizes* $|\ell(v) - \ell'|$, *and where* $\mathsf{up}^*(v) \geq (1+\delta)^{i'}$ *and* $\mathsf{up}(v) \leq (2+3/\lambda)(1+\delta)^i$ *where* $\ell' - 1 \in g_{i'}$, $\ell' \in g_i$ *and* $i' \leq i$. *In other words, the desire-level of* v *is the closest level* $\ell(v)$ *where both Invariant 1 and Invariant 2 are satisfied.*

We show that the invariants above are always maintained except for a period of time when processing a new batch of insertions/deletions. During this period, the data structure undergoes a *rebalance procedure*, during which the invariants may be violated. The main update procedure is shown in Algorithm 20. It separates the updates into insertions and deletions (Line 1), and then calls RebalanceInsertions (Line 2: Algorithm 21) and RebalanceDeletions (Line 3: Algorithm 22). We make note of two crucial observations that we prove in the analysis: when processing a batch of insertions, Invariant 2 is never violated; and similarly, when processing a batch of deletions, Invariant 1 is not violated. This means that no vertex needs to move *down* when processing a batch of insertions and no vertex needs to move *up* when processing a batch of deletions. We first describe the data structures that we maintain and then the RebalanceInsertions and RebalanceDeletions procedures below.

Data Structures Each vertex v keeps track of its set of neighbors in two structures. U keeps track of the neighbors at v's level and above. We denote this set of v's neighbors by $U[v]$. L_v keeps track of neighbors of v for every level below $\ell(v)$—in particular, $L_v[j]$ contains the neighbors of v at level $j < \ell(v)$. We describe specific data structure implementation details in Section 6.3.6 and Section 6.6.

RebalanceInsertions(B_{ins}) Algorithm 21 shows the pseudocode. Provided a batch of insertions B_{ins}, we iterate through the K levels from the lowest level $\ell = 0$ to the highest level $\ell = K - 1$ (Algorithm 21, Line 5). For each level, in parallel we check the vertices incident to edge insertions in B_{ins} (or is marked) to see if they violate Invariant 1 (Line 6). If a vertex v in the current level l violates Invariant 1, we move v to level $l+1$ (Line 7). After moving v, we update structures $U[v], L_v$, and the structures of $w \in N(v)$ where $\ell(w) \in [l, l+1]$. First, we create $L_v[l]$ to store the neighbors of v in level l (Line 7). If v moved to level $l+1$ and w stayed in level l, then we delete w from $U[v]$ and instead insert w into $L_v[l]$ (Line 9). We do not need to make any data structure modifications for w since v stays in $U[w]$. Similarly, no data structure modifications to v and w are necessary when both v and w move to level $l+1$. For each neighbor of v on level $l+1$, we need to check whether it now violates Invariant 1. If it does, then we mark the vertex (Line 11). We process any such marked vertices when we process level $l+1$. We also update the U and L arrays of every neighbor of v on level $l+1$ (Line 12). Specifically,

Algorithm 21 RebalanceInsertions(B_{ins})

Input: A batch of edge insertions B_{ins}.

1: Let U contain all up-neighbors of each vertex, keyed by vertex. So $U[v]$ contains all up-neighbors of v.
2: Let L_v contain all neighbors of v in levels $[0, \ell(v)-1]$, keyed by level number.
3: **parfor** each edge insertion $e = (u, v) \in \mathcal{B}_{ins}$ **do**
4: Insert e into the graph.
5: **for** each level $l \in [0, K-1]$ starting with $l = 0$ **do**
6: **parfor** each vertex v incident to B_{ins} or is marked, where $\ell(v) = l \cap \mathsf{up}(v) > (2+3/\lambda)(1+\delta)^{gn(l)}$ **do**
7: Mark and move v to level $l+1$ and create $L_v[l]$ to store v's neighbors at level l.
8: **parfor** each $w \in N(v)$ of a vertex v that moved to level $l+1$ and w stayed in level l **do**
9: $U[v] \leftarrow U[v] \setminus \{w\}, L_v[l] \leftarrow L_v[l] \cup \{w\}$.
10: **parfor** each $u \in N(v)$ of a vertex v that moved to level $l+1$ and u is in level $l+1$ **do**
11: Mark u if $\mathsf{up}(u) > (2+3/\lambda)(1+\delta)^{gn(l+1)}$.
12: $U[u] \leftarrow U[u] \cup \{v\}$, $L_u[l] \leftarrow L_u[l] \setminus \{v\}$.
13: **parfor** each $x \in N(v)$ of a vertex v that moved to level $l+1$ and x is in level $\ell(x) \geq l+2$ **do**
14: $L_x[l] \leftarrow L_x[l] \setminus \{v\}, L_x[l+1] \leftarrow L_x[l+1] \cup \{v\}$.
15: Unmark v if $\mathsf{up}(v) \leq (2+3/\lambda)(1+\delta)^{gn(l+1)}$. Otherwise, leave v marked.

let u be one such neighbor, we add v to $U[u]$ and remove v from $L_u[l]$. We conclude by making appropriate modifications to L for each neighbor on levels $\geq l+2$ (Line 13–Line 14). Specifically, let x be one such neighbor, we remove v from $L_x[l]$ and add v to $L_x[l+1]$. All neighbors of vertices that moved can be checked and processed in parallel. Finally, v becomes unmarked if it satisfies all invariants; otherwise, it remains marked and must move again in a future step (Line 15).

A detailed example of this procedure is below.

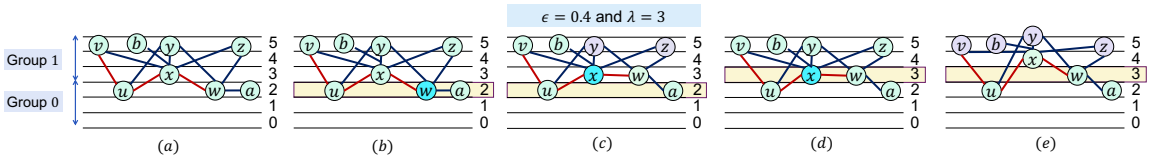


Figure 6-3: Example of RebalanceInsertions described in Example 6.3.4 for $\delta = 0.4$ and $\lambda = 3$.

Example 6.3.4. *Fig. 6-3 shows an example of our entire insertion procedure described in Algorithm 21. The red lines in the example represent the batch of edge insertions. Thus, in* (a)*, the newly inserted edges are the edges* (u, v)*,* (u, x)*, and* (x, w)*. We iterate from the bottommost level (starting with level* 0*) to the topmost level (level* $K-1$*).*

The first level where we encounter vertices that are marked or are adjacent to an edge insertion is level 2*. Since level* 2 *is part of group* 0*, the cutoff for Invariant 1 is* $(2+3/\lambda)(1+\delta)^0 = 3$ *provided* $\lambda = 3$ *and* $\delta = 0.4$*. In level* 2*, only* w *violates Invariant 1 since the number of its neighbors on levels* ≥ 2 *is* 4 *(*x*,* y*,* z*, and* a*), so* $\mathsf{up}(w) = 4 > 3$ *(shown in* (b)*). Then, in*

Algorithm 22 RebalanceDeletions(B_{del})

Input: A batch of edge deletions B_{del}.

1: Let U contain all up-neighbors of each vertex, keyed by vertex. So $U[v]$ contains all up-neighbors of v.
2: Let L_v contain all neighbors of v in levels $[0, \ell(v)-1]$, keyed by level number.
3: **parfor** each edge deletion $e = (u, v) \in B_{del}$ **do**
4: Remove e from the graph.
5: **parfor** each vertex v where $\mathsf{up}^*(v) < (1+\delta)^{gn(\ell(v)-1)}$ **do**
6: Calculate $\mathsf{dl}(v)$ using CalculateDesireLevel(v).
7: **for** each level $l \in [0, K-1]$ starting with level $l = 0$ **do**
8: **parfor** each vertex v where $\mathsf{dl}(v) = l$ **do**
9: Move v to level l.
10: **parfor** each neighbor w of a vertex v that moved to level l where $\ell(w) \geq \ell(v)$ **do**
11: Let p_v and p_w be the previous levels of v and w, respectively, before the move.
12: **if** $\ell(w) = \ell(v) = l$ **then**
13: $L_w[p_v] \leftarrow L_w[p_v] \setminus \{v\}$, $L_v[p_w] \leftarrow L_v[p_w] \setminus \{w\}$.
14: $U[w] \leftarrow U[w] \cup \{v\}$, $U[v] \leftarrow U[v] \cup \{w\}$.
15: **else**
16: **if** $p_v > \ell(w)$ **then**
17: $U[w] \leftarrow U[w] \setminus \{v\}$, $L_v[\ell(w)] \leftarrow L_v[\ell(w)] \setminus \{v\}$.
18: **else if** $p_v = \ell(w)$ **then**
19: $U[w] \leftarrow U[w] \setminus \{v\}$.
20: **else** $L_w[p_v] \leftarrow L_w[p_v] \setminus \{v\}$.
21: $L_w[l] \leftarrow L_w[l] \cup \{v\}$, $U[v] \leftarrow U[v] \cup \{w\}$.
22: **if** $\mathsf{up}^*(w) < (1+\delta)^{gn(\ell(w)-1)}$ **then**
23: Recalculate $\mathsf{dl}(w)$ using Algorithm 23.

(c), we move w up to level 3. We need to update the data structures for neighbors of w at level 3 and above (as well as w's own data structures); the vertices with data structure updates are x, w, y, and z. After the move, x becomes marked because it now violates Invariant 1 (the cutoff for level 3 is $3 \cdot (1.4) = 4.2$ since level 3 is in group 1); w becomes unmarked because it no longer violates Invariant 1.

In (d), we move on to process level 3. The only vertex that is marked or violates Invariant 1 is x. Therefore, we move x up one level (shown in (e)) and update relevant data structures (of x, v, y, z, and b).

RebalanceDeletions(B_{del}) Deletions are handled in a similar way to insertions, except for one major difference. Instead of moving vertices down one level at a time, we ensure that when we move a vertex, we move it to its final level (i.e., we will not move this vertex again during this procedure). As we show in the analysis, this guarantee is *crucial to obtaining low depth.*

Algorithm 22 shows the pseudocode. For each vertex v incident to an edge deletion, first we check whether it violates Invariant 2 (Algorithm 22, Line 5). In Line 5, $gn(\ell(v)-1)$ gives the group number i where $\ell(v)-1 \in g_i$. If it violates Invariant 2, we calculate its

desire-level, $\mathsf{dl}(v)$, using CalculateDesireLevel (Algorithm 22, Line 6). We describe how to do this at the end of this subsection. As in the insertion procedure, we iterate through the levels from $l = 0$ to $l = K - 1$ (Line 7). Then, in parallel for each vertex whose desire-level is l, we move the vertex to level l (Line 8–Line 9). As with insertions, we update the data structures of v and $w \in N(v)$ where $\ell(w) \geq l$ (Line 10–Line 21). Finally, we update the desire-level of neighbors of v that no longer satisfy Invariant 2 (Line 22–Line 23). We process all vertices that move as well as their neighbors in parallel.

An example of the RebalanceDeletions procedure is given below.

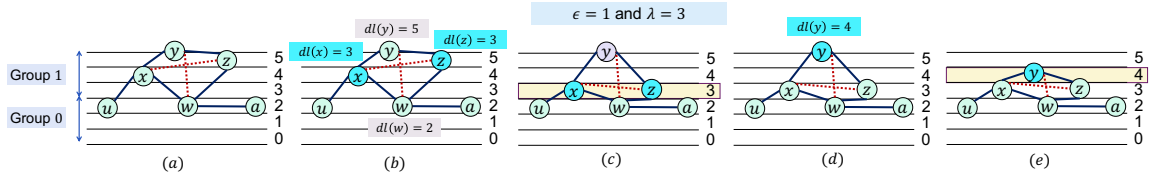


Figure 6-4: Example of RebalanceDeletions described in Example 6.3.5 for $\delta = 1$ and $\lambda = 3$.

Example 6.3.5. *Fig. 6-4 shows an example of our entire deletion procedure described in Algorithm 22 for $\delta = 1$ and $\lambda = 3$. The red dotted lines in the example represent the batch of edge deletions. Thus, in (a), the newly deleted edges are the edges (x, z) and (y, w). For each vertex adjacent to an edge deletion, we calculate its desire-level, or the closest level to its current level that satisfies Invariant 2.*

In this example, shown in (b), only x and z violate Invariant 2. The lower bound on the number of neighbors that must be at or above level 3 for x and level 4 for z is $(1+\delta)^1 = 2$ since $\delta = 1$ and levels 3 and 4 are in group 1. (Recall that the lower bound is calculated with respect to the level below *x and z.) We calculated that the desire levels of x and z are both 3. The desire levels of y and w are their current levels because they do not violate the invariant. Then, we iterate from the bottommost level (starting with level 0) to the topmost level (level $K - 1$).*

Level 3 is the first level where vertices want to move. Then, we move x and z to level 3 (shown in (c)). We only need to update the data structures of neighbors at or above x and z so we only update the data structures of x, y, and z. Invariant 2 is no longer violated for x and z. In fact, our algorithm guarantees that each vertex moves at most once. *We check whether any of x or z's up-neighbors violate Invariant 2. Indeed, in this example, y now violates the invariant. We recompute the desire-level of y and its desire-level is now 4 (shown in (d)). We proceed to process level 4 and move y to that level (shown in (e)).*

CalculateDesireLevel(v) Algorithm 23 shows the procedure for calculating the desire-level, $\mathsf{dl}(v)$, of vertex v, which is used in Algorithm 22. Let $gn(\ell)$ be the index i where level $\ell \in g_i$. We use a doubling procedure followed by a binary search to calculate our desire-level. We initialize a variable d to $\mathsf{up}^*(v)$ (number of neighbors at or above $\ell(v) - 1$). Starting with level $\ell(v) - 2$, we add the number of neighbors in level $\ell(v) - 2$ to d (Algorithm 23, Line 3). This procedure checks whether moving v to $\ell(v) - 1$ satisfies Invariant 2 (Line 6). If it passes the check, then we are done and we move v to $\ell(v) - 1$. Otherwise, we double the number of levels from which we count neighbors (Line 6). In our example,

Algorithm 23 CalculateDesireLevel(v)

Input: A vertex v that needs to move to a level $j < \ell(v)$.
Output: The desire-level $\mathsf{dl}(v)$ of vertex v.

1: $d \leftarrow \mathsf{up}^*(v), p \leftarrow 1, i \leftarrow 2$
2: **while** $d < (1+\delta)^{gn(\ell(v)-p)}$ and $\ell(v) - p > 0$ **do**
3: $\quad d \leftarrow d + \sum_{j=p}^{i-1} \left|L_v[\ell(v) - j - 1]\right|$
4: $\quad$ **if** $d \geq (1+\delta)^{gn(\ell(v)-i)}$ **then**
5: $\quad\quad$ Binary search within levels $[\ell(v) - i + 1, \ell(v) - p]$ to find the closest level to $\ell(v)$ that satisfies Invariants 1 and 2.
6: $\quad p \leftarrow i, i \leftarrow \min(2 \cdot i, \ell(v))$.

in the next iteration, we sum the number of neighbors in levels $[\ell(v) - 4, \ell(v) - 3]$. We continue until we find a level where Invariant 2 is satisfied. Let this level be ℓ' and the previous cutoff be ℓ_{prev}. Finally, we perform a binary search within the range $[\ell', \ell_{prev}]$ to find the *closest* level to $\ell(v)$ that satisfies Invariant 2 (Line 5). The sum on Line 3 is done using a parallel reduce.

6.3.4 Vertex Insertions and Deletions

We can handle vertex insertions and deletions by inserting vertices which have zero degree and considering deletions of vertices to be a batch of edge deletions of all edges adjacent to the deleted vertex. When we insert a vertex with zero degree, it automatically gets added to level 0 and remains in level 0 until edges incident to the vertex are inserted. For a vertex deletion, we add all edges incident to the deleted vertex to a batch of edge deletions. Note, first, that all vertices which have 0 degree will remain in level 0. Thus, there are at most $O(m)$ vertices which have non-zero degree.

Because we have $O(\log^2 m)$ levels in our data structure, we rebuild the data structure once we have made $\frac{m}{2}$ edge updates (including edge updates from edges incident to deleted vertices). Rebuilding the data structure requires $O(m \log^2 n)$ total work which we can amortize to the $\frac{m}{2}$ edge updates to $O(\log^2 n)$ amortized work. Running Algorithm 21 and Algorithm 22 on the entire set of $O(m)$ edges requires $\tilde{O}(\log^2 n)$ depth.

6.3.5 Coreness and Low-Outdegree Orientation

To obtain the coreness estimates from our PLDS, we only need to maintain the current level of each vertex and the number of levels in a group (recall that all groups have equal numbers of levels). Then, we calculate the estimate of the coreness of v to be $\hat{k}(v) = (1+\delta)^{\max(\lfloor \ell(v)/\lceil \log_{1+\delta} m \rceil \rfloor - 1, 0)}$, where the number of levels in a group is $\left\lceil \log_{1+\delta} m \right\rceil$. In other words, our estimate of the coreness of v is $(1+\delta)^i$, where i is the group index of the *highest level* that is the last level in a group *and* is equal to or lower than $\ell(v)$. We need to perform this calculation in order to obtain our optimum approximation ratios both in theory (Lemma 6.3.16) and in practice. To see an example, consider vertex y in Fig. 6-4 (e). We estimate $\hat{k}(y) = 1$ since the highest level that is the last level of a group and is equal

to or below level $\ell(y) = 4$ is level 2. Level 2 is part of group 0 so our coreness estimate for y is $(1+\delta)^0 = 1$. This is a 2-approximation of its actual coreness 2.

To obtain a low-outdegree orientation we maintain an orientation of each edge from the endpoint at a lower level to the endpoint with a higher level. So edges are directed from lower to higher levels. For any two vertices on the same level, we direct the edge from the vertex with higher ID to the vertex with lower ID. Performing this procedure results in an 8α outdegree orientation, where α is the arboricity of the graph.

6.3.6 Data Structure Implementations

Due to space constraints, we only specify the data structures used to obtain our randomized (high probability) bounds. We implement these data structures in our experiments. Our data structures presented here result in $O(n \log^2 m + m)$ space usage. However, we present two additional sets of data structures in Section 6.6: one that achieves $O(m)$ space with an $O(\log^2 m)$ factor increase in depth and another that results in a *deterministic* PLDS.

Our data structure uses parallel hash tables, which support x concurrent insertions, deletions, or finds in $O(x)$ amortized work and $O(\log^* x)$ depth *whp* [GMV91]. We use dynamic arrays, which support adding or deleting x elements from the end in $O(x)$ amortized work and $O(1)$ depth.

We first assign each vertex a unique ID in $[n]$. We also maintain an array U of size n keyed by vertex ID that returns a parallel hash table containing neighbors of v on levels $\geq \ell(v)$. For each vertex v, we maintain a dynamic array L_v keyed by indices $i \in [0, \ell(v)-1]$. The i'th entry of the array contains a pointer to a parallel hash table containing the neighbors of v in level i. Appropriate pointers exist that allow $O(1)$ work to access elements in structures. Furthermore, we maintain a hash table which contains pointers to vertices v where $\mathsf{dl}(v) \neq \ell(v)$, partitioned by their levels. This allows us to quickly determine which vertices to move up (in Algorithm 21) or move down (in Algorithm 22).

6.3.7 Analysis

We now present the lemmas used to analyze our algorithms, as well as our overall bounds and approximation guarantee.

Depth and Work Bound First, it is easy to show that there exists a level where both invariants are satisfied. This allows our PLDS data structure to assign each vertex to a single level.

Lemma 6.3.6. *If a vertex v violates Invariant 1, then there exists a level $l > \ell(v)$ where v satisfies both Invariant 1 and Invariant 2. If a vertex w violates Invariant 2, then there exists a level $l < \ell(w)$ where w satisfies both invariants or $l = 0$ (it is in the bottommost level).*

Proof. First note that no vertex can simultaneously violate both Invariant 1 and Invariant 2. Thus, suppose first that v violates Invariant 1. Then, this means that the number of neighbors of v on levels $\geq \ell(v)$ is more than $(2 + 3/\lambda)(1+\delta)^{g(v)}$ where $g(v)$ is the group

number of v. If v still violates Invariant 1 on level $\ell(v)+1$, then we keep moving v to the next level.

Otherwise, v does not violate Invariant 1 on level $\ell(v)+1$. Since we know that v violated Invariant 1 on level $\ell(v)$, then after we move v to $\ell(v)+1$, v's up*-degree is greater than $(1+\delta)^{gn(\ell(v))}$; hence, v also does not violate Invariant 2. The very last level of the K levels has up-degree bound $(2+3/\lambda)(1+\delta)^{\lceil \log_{1+\delta}(m)\rceil} \geq 2n'$ where n' is the number of vertices with at least one adjacent edge. Hence, there must exist a level at or below the last level where both invariants are satisfied. A similar argument holds for Invariant 2. □

Then, we show that the batch of insertions never violates Invariant 2 and a batch of deletions never violates Invariant 1.

Lemma 6.3.7 (Batch Insertions)**.** *Given a batch of insertions, $\mathcal{B}_{ins}$, Invariant 2 is never violated at any point during the rebalance procedure given by Algorithm 21.*

Proof. The first part of the algorithm inserts the edges into the data structure. Since no edges are removed from the data structures, the degrees of all the vertices after the insertion of edges cannot decrease. Invariant 2 was satisfied before the insertion of the edges, and hence, it remains satisfied after the insertion of edges because no vertices lose neighbors. We prove that the lemma holds for the remaining part of Algorithm 21 via induction on the level i processed by the procedure. In the base case, when $i = 0$, all vertices v in the level which violate Invariant 1 are moved up to a level $\mathsf{dl}(v) > 0$. By definition of desire-level, v is moved to a level where Invariant 2 is still satisfied, by Lemma 6.3.6. Vertices from level 0 which move to levels $k \geq 1$ cannot decrease the up*-degree for neighbors in all levels j where $j > 1$. Thus, Invariant 2 cannot be violated for these vertices. Vertices not adjacent to v are not affected by the move.

We assume that Invariant 2 was not violated up to level i and prove it is not violated while processing vertices on level $i+1$. By our induction hypothesis, no vertices violate Invariant 2 before we process level $i+1$. Then, when we process level $i+1$, no vertices move down to a lower level than $i+1$ by construction of our algorithm because Invariant 2 is not violated for any vertex on level $i+1$ and if Invariant 1 is violated for any vertex w, w must move up to a higher level. Any vertex w which moves up to a higher level cannot decrease the up*-degree of neighbors of w. Hence, no vertex on levels $\geq i+1$ can violate Invariant 2. The up*-degree of vertices on levels $< i+1$ are not affected by the move. Hence, no vertices on levels $< i+1$ violate Invariant 2. Finally, if a vertex on level $i+1$ violates Invariant 1, it will move to a level $j > i+1$ where both invariants are satisfied by Lemma 6.3.6. □

Lemma 6.3.8 (Batch Deletions)**.** *Given a batch of deletions, $\mathcal{B}_{del}$, Invariant 1 is never violated while $\mathcal{B}_{del}$ is applied.*

Proof. Algorithm 22 first applies all the edge deletions in the batch. Edge deletions cannot make the up-degree of any vertex greater; thus, no vertex violates Invariant 1 after applying the edge deletions. We prove that the rest of the algorithm does not violate Invariant 1 via induction over the levels i. Specifically, we use as our induction hypothesis that after processing the i'th level, no vertices violate Invariant 1. In the base case, when $i = 0$,

no vertices violate Invariant 1 at the beginning, and vertices from levels $i > 0$ move to level 0. This means that during the processing of level $i = 0$, vertices only move to level 0 from a higher level. Thus, all such vertices that move will move to a lower level. Since vertices which move to lower levels do not increase the up-degree of any other vertices, no vertex can violate Invariant 1 at the end of processing level 0. We now prove the case for processing level $i + 1$. In this case, we assume by our induction hypothesis that no vertices violate Invariant 1 after we finish processing level i. Thus, all vertices that want to move to level $i + 1$ and violate Invariant 2 are at levels $j > i + 1$. Such vertices move down and thus cannot increase the up-degree of any vertex. This means that after moving all vertices that want to move to level $i + 1$, no vertices violate Invariant 1. □

Batch Insertion Depth Bound The depth of our batch insertion algorithm (Algorithm 21) depends on the following lemma which states that once we have *processed* a level (after finishing the corresponding iteration of Line 5), no vertex will want to move from any level lower than that level. This means that each level is processed exactly once, resulting in at most $O(\log^2 m)$ levels to be processed sequentially.

Lemma 6.3.9. *After processing level i in Algorithm 21, no vertex v in levels $\ell(v) \leq i$ will violate Invariant 1. Furthermore, no vertex w on levels $\ell(w) > i$ will have $\mathsf{dl}(w) \leq i$.*

Proof. We prove this via induction. For the base case $i = 0$, all vertices on level 0 are part of each other's up-degree; then, no vertices which move up from $i = 0$ can cause the up-degree of any vertices remaining in level 0 to increase. We now assume the induction hypothesis for $i-1$ and prove the case for i. Vertices on level $j \leq i$ already contain vertices on levels $\geq i$ in its up-degree. Such vertices on levels $\geq i$ when moved to a higher level are still part of the up-degree of vertices on levels $j \leq i$. Hence, no vertices on levels $j \leq i$ will violate Invariant 1 due to vertices in levels $\geq i$ moving up to a level $l > i$. Then, in order for a vertex w with $\ell(w) > i$ to have $\mathsf{dl}(w) \leq i$, some neighbors of w must have moved to a level $\leq i$. By Lemma 6.3.7, no vertices move down during Algorithm 21, so this is not possible. □

Batch Deletion Depth Bound For the batch deletion algorithm (Algorithm 22), we prove that after all vertices which have $\mathsf{dl}(w) = i$ are moved to the i'th level, no vertex will have $\mathsf{dl}(v) \leq i$. As in the insertion case, this means that each level is processed exactly once, resulting in at most $O(\log^2 m)$ levels to be processed sequentially.

Lemma 6.3.10. *After processing all vertices that move to level i in Algorithm 22, no vertex needs to be moved to any level $j \leq i$ in a future iteration of Line 7; in other words, no vertex v has $\mathsf{dl}(v) \leq i$ after level i has been processed.*

Proof. We prove this invariant via induction. In the base case when $i = 0$, all vertices with $\mathsf{dl}(v) = 0$ are moved to level 0. All vertices which have $\mathsf{dl}(v) = 0$ are vertices which have degree 0. Thus, all vertices that do not have $\mathsf{dl}(v) = 0$ have degree ≥ 1 and have $\mathsf{dl}(w) \geq 1$. Hence, after moving all vertices with $\mathsf{dl}(v) = 0$ to level 0, no additional vertices need to be moved to level 0. Assuming our induction hypothesis, we now show our lemma holds for level $i + 1$. All vertices that move to level $i + 1$ violated Invariant 2 and hence have

up*-degree $< (1+\delta)^{gn(j-1)}$ at level $j > i+1$ and up*-degree $\geq (1+\delta)^{gn(i)}$ at level $i+1$. After moving all vertices with $\mathsf{dl}(v) = i+1$ to level $i+1$, no vertices on levels $k \leq i+1$ have their up*-degree decreased by the move. We conclude the proof with vertices at levels $l > i+1$. Suppose for the sake of contradiction that there exists some vertex w on level $l > i+1$ which has $\mathsf{dl}(w) \leq i+1$ after the move. In order for $\mathsf{dl}(w) \leq i+1$, some neighbor(s) of w must move below level i, a contradiction. Finally, due to Lemma 6.3.8, no vertices below level $i+1$ will move up. □

The depth of our algorithm follows almost immediately from the above lemmas. Before we present the final lemmas for the depth of our algorithm, we discuss the depth incurred from our data structures and also briefly the deterministic and space-efficient settings.

Deterministic Setting In the deterministic setting, we maintain the list of neighbors using dynamic arrays, which also means that we maintain and access the sizes of these arrays in $O(1)$ work and depth. Because we are using dynamic arrays, we need to occasionally resize the arrays in $O(1)$ amortized work and $O(1)$ depth. Finally, we can modify the arrays in $O(1)$ work and depth (not counting the depth for resizing).

Randomized Setting In the randomized setting, we maintain the list of neighbors using parallel hash tables keyed by level. Each vertex has one hash table which contains their neighbors at each level below them as well as all its neighbors at the same or higher levels. Then, vertices themselves are contained in separate hash tables for each level. Parallel lookups into the hash tables each require $O(1)$ work and depth, *whp*. Then, modifying (inserting and deleting) elements within the hash tables require $O(\log\log m)$ depth and work proportional to the number of inserted elements, *whp*.

Space-Efficient Setting In the space-efficient setting, we replace the structure used to represent L_{v_i} with a linked list. Inserting and deleting nodes from the linked list requires $O(1)$ work and depth (assuming we are given a pointer to the node). Then, resizing the dynamic arrays (pointed to by the linked lists to maintain the set of elements in each level) require $O(1)$ amortized work and $O(1)$ depth.

The only additional depth we need to consider is the depth acquired from Algorithm 23. Both the doubling search and the binary search require $O(\log K) = O(\log\log m)$ depth. All other depth comes from concurrently modifying and accessing dynamic arrays and hash tables, which can be done in $O(\log^* m)$ depth *whp*.

Using the above, we prove the depth of Algorithm 20.

Lemma 6.3.11. *Algorithm 20 returns a deterministic level data structure that maintains Invariant 1 and Invariant 2 and has $O(\log^3 m)$ worst-case depth and $O(n\log^2 m + m)$ space.*

Proof. All edge updates can be partitioned into $\mathcal{B}_{ins}$ and $\mathcal{B}_{del}$ in parallel in $O(1)$ depth. Then, it remains to bound the depth of Algorithm 21 and Algorithm 22.

Algorithm 21 iterates through all $K = O\left(\log^2 m\right)$ levels sequentially. By Lemma 6.3.9, no vertices on level $\leq i$ will violate Invariant 1 after processing level i. Thus, by the

end of the procedure no vertices violate Invariant 1. By Lemma 6.3.7, Invariant 2 was never violated during Algorithm 21. Thus, both invariants are maintained at the end of the algorithm. Since we iterate through $O(\log^2 m)$ levels and, in each level, we require checking the neighbors at one additional level which can be done in parallel in $O(1)$ depth, the total depth of this procedure is $O(\log^2 m)$. To resize the dynamic array, we require $O(\log m)$ depth whenever the array becomes too large or too small to compute the offsets by which to insert the new elements. For each level, an additional depth of $O(\log m)$ might be necessary to compute the element offsets and then resize the arrays in $O(1)$ depth. Then, Algorithm 21 requires $O(\log^3 m)$ worst-case depth.

Algorithm 22 iterates through all $K = O\left(\log^2 m\right)$ levels sequentially. By Lemma 6.3.8 and Lemma 6.3.10, after processing level i, no vertices in a level higher than $i+1$ will have $\mathsf{dl}(v) \leq i+1$ and no vertices on levels $\leq i$ will violate Invariant 1. Thus, by the end of the procedure all vertices satisfy Invariant 2. Furthermore, Invariant 1 was never violated due to Lemma 6.3.8. There are $O(\log^2 m)$ levels and for each level we require running Algorithm 23 to obtain the $\mathsf{dl}(v)$ of each affected vertex v that should be moved to each level.

Running Algorithm 23 requires $O(\log\log m)$ depth to obtain the first level that satisfies invariants for each affected vertex v and $O(\log\log m)$ depth for the final binary search that determines the closest level to $\ell(v)$ that satisfies the invariants. In conclusion, Algorithm 22 requires $O(\log^3 m)$ worst-case depth.

In general, Algorithm 20 requires $O(\log^3 m)$ worst-case depth. □

Corollary 6.3.12. *Algorithm 20 returns a randomized level data structure that maintains Invariant 1 and Invariant 2 and has $O(\log^2 m \log\log m)$ depth, whp, and $O(n\log^2 m + m)$ space.*

Proof. The proof is the same as the proof of Lemma 6.3.11 except we replace dynamic arrays with parallel hash tables. Simultaneously changing the values within the hash table require $O(\log^* m)$ depth *whp*. Then, the depth per level of the structure is dominated by Algorithm 23. Thus, the total depth of our randomized algorithm is $O(\log^2 m \log\log m)$ *whp*, and $O(n\log^2 m + m)$ space. □

The additional $n\log^2 m$ space comes from needing to store the L_v dynamic arrays for each vertex v in the graph. We show that with slightly more complicated data structures involving linked lists, we can obtain *space-efficient* structures.

Corollary 6.3.13. *Algorithm 20 returns a space-efficient, deterministic, level data structure that maintains Invariant 1 and Invariant 2 and has $O(\log^4 m)$ worst-case depth.*

Proof. The proof is the same as the proof of Lemma 6.3.11 except we replace Algorithm 23 with an $O(\log^2 m)$ linear in number of levels search. The specific data structure we use for each vertex v is a linked list with each node of the linked list representing a level $\leq \ell(v) - 1$ which contains one or more neighbors of v. Then, each node in the linked list contains a pointer to a dynamic array containing the neighbors in that level. The linked list has size at most $O(\log^2 m)$. Thus, the total depth is $O(\log^4 m)$. □

Our work bound uses potential functions similar to those presented in Section 4 of [BHNT15]. We show our parallel algorithm serializes to a set of sequential steps that can be analyzed using these potential functions. Because the potential functions are very similar to those in [BHNT15], we leave the discussion to Section 6.7. Together, we obtain the work and depth bounds shown in Theorem 6.1.2.

6.3.8 $(2+\varepsilon)$-Approximation of Coreness

The *coreness estimate*, $\hat{k}(v)$, is an estimate of the coreness of a vertex v. As mentioned previously in Section 6.3.5, we compute an estimate of the coreness using information about the level of the vertex v. Here, we show that such an estimate provides us with a $(2+\varepsilon)$-approximation to the actual coreness of v for any constant $\varepsilon > 0$. (We can find an approximation for any fixed ε by appropriately setting our parameters δ and λ.) To calculate $\hat{k}(v)$, we first find the largest index i of a group g_i, where $\ell(v)$ is at least as high as the highest level in g_i.

Definition 6.3.14 (Coreness Estimate)**.** *The* coreness estimate $\hat{k}(v)$ *of vertex* v *is* $(1+\delta)^{\max(\lfloor \ell(v)/\lceil \log_{1+\delta} n \rceil \rfloor - 1, 0)}$.

We prove that our PLDS maintains a $(2+3/\lambda)(1+\delta)$ approximation of the coreness value of each vertex, for any constants $\lambda > 0$ and $\delta > 0$, if we use $\hat{k}(v)$ as computed in Definition 6.3.14. Therefore, we obtain the following lemma giving the desired $(2+\varepsilon)$-approximation, a 2-factor improvement on the previous approximation bounds. Previous works were only able to obtain a $(4+\varepsilon)$-approximation theoretical bound [SCS20]. We see in our experimental analysis that such a bound improves the maximum error of our algorithm compared to previous algorithms that obtain similar average errors. Given fixed δ and λ, the maximal error of our algorithm is given by $(2+3/\lambda)(1+\delta)$.

We first show some properties of $\hat{k}(v)$ and then we show that we can obtain an approximate coreness number by looking at $\hat{k}(v)$.

Lemma 6.3.15. *Let* $\hat{k}(v)$ *be the coreness estimate and* $k(v)$ *be the coreness of* v*, respectively. If* $k(v) > (2+3/\lambda)(1+\delta)^{g'}$*, then* $\hat{k}(v) \geq (1+\delta)^{g'}$*. Otherwise, if* $k(v) < \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$*, then* $\hat{k}(v) < (1+\delta)^{g'}$.

Proof. For simplicity, we assume the number of levels per group is $2\left\lceil \log_{(1+\delta)} m \right\rceil$ (a tighter analysis can accomodate the case when the number of levels per group is $\left\lceil \log_{(1+\delta)} m \right\rceil$). Let $T(g')$ be the topmost level of group g'. In the first case, we show that if $k(v) > (2+3/\lambda)(1+\delta)^{g'}$, then v would be in a level higher than $T(g')$ in our level data structure. This would also imply that $\hat{k}(v) \geq (1+\delta)^{g'}$. Suppose for the sake of contradiction that v is located at some level $\ell(v)$ where $\ell(v) \leq T(g')$. This means that $\mathsf{up}(v) \leq (2+3/\lambda)(1+\delta)^{g'}$ at level $\ell(v)$. Furthermore, by the invariants of our level data structure, each vertex w at the same or lower level has $\mathsf{up}(w) \leq (2+3/\lambda)(1+\delta)^{g'}$. This means that when we remove all vertices starting at level 0 sequentially up to and including $\ell(v)$, all vertices removed have degree $\leq (2+3/\lambda)(1+\delta)^{g'}$ when removed. Thus, when we reach $\ell(v)$, v also has

degree $\leq (2+3/\lambda)(1+\delta)^{g'}$. This is a contradiction with $k(v) > (2+3/\lambda)(1+\delta)^{g'}$. It must then be the case that v is at a level higher than $T(g')$ and $\hat{k}(v) \geq (1+\delta)^{g'}$.

Now we prove that if $k(v) < \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$, then $\hat{k}(v) < (1+\delta)^{g'}$. We assume for sake of contradiction that $k(v) < \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$ and $\hat{k}(v) \geq (1+\delta)^{g'}$. To prove this case, we consider the following process, which we call the *pruning* process. Pruning is done on a subgraph $S \subseteq G$. We use the notation $d_S(v)$ to denote the degree of v in the subgraph induced by S. For a given subgraph S, we *prune* S by repeatedly removing all vertices v in S whose $d_S(v) < \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$. Note that in this argument, we need only consider levels from the same group g' before we reach a contradiction, so we assume that all levels are in the group g'. Let j represent the number of levels below level $T(g')$. (Recall that because $\hat{k}(v) \geq (1+\delta)^{g'}$, $\ell(v) \geq T(g')$. If we consider a level $\ell(v) > T(g')$, then the up*-degree cannot decrease due to Invariant 2 becoming stricter. This only makes our proof easier, and so for simplicity, we consider $\ell(v) = T(g')$.) We prove via induction that the number of vertices pruned from the subgraph induced by $Z_{T(g')-j}$ *must* be at least

$$\left(\frac{(2+3/\lambda)(1+\delta)}{2}\right)^{j-1}\left((1+\delta)^{g'} - \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}\right)$$

or otherwise, $k(v) \geq \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$. We first prove the base case when $j = 1$. In the base case, we know that $d_{Z_{T(g')-1}}(v) \geq (1+\delta)^{g'}$ by Invariant 2. Then, if fewer than $(1+\delta)^{g'} - \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$ neighbors of v are pruned from the graph, then v is part of a $\geq \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$-core and $k(v) \geq \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$, a contradiction.

Thus, at least $(1+\delta)^{g'} - \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$ vertices must be pruned in this case. We now assume the induction hypothesis for j and prove that this is true for step $j+1$. By Invariant 2, each vertex on level $T(g') - j$ and above has degree *at least* $(1+\delta)^{g'}$ in graph $Z_{T(g')-j-1}$. Then, in order to prune all X vertices from the previous induction step, we must prune at least $\frac{(1+\delta)^{g'}X}{2}$ edges. Each vertex that is pruned can remove *at most* $\frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$ edges when it is pruned, by definition of our pruning procedure. Thus, the *minimum* number of vertices we must prune in order to prune the $X = \left(\frac{(2+3/\lambda)(1+\delta)}{2}\right)^{j-1}\left((1+\delta)^{g'} - \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}\right)$ vertices from the previous step is

$$\begin{aligned}\frac{(1+\delta)^{g'}X}{2\frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}} &= \frac{(2+3/\lambda)(1+\delta)}{2}X \\ &= \left(\frac{(2+3/\lambda)(1+\delta)}{2}\right)^{j}\left((1+\delta)^{g'} - \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}\right).\end{aligned}$$

Thus, we have proven our argument for the $(j+1)$-st induction step. Note that for $j = \left\lceil \log_{(2+3/\lambda)(1+\delta)/2}(2m+1)\right\rceil$, we have $j \leq 2\left\lceil \log_{(1+\delta)}(m)\right\rceil$. This is because, since we

pick λ to be a constant, $2 + 3/\lambda > 2$ and for large enough m, $\log_{(2+3/\lambda)(1+\delta)/2}(2m+1) \leq 2\left\lceil \log_{(1+\delta)}(m) \right\rceil$. Then, by our induction, we must prune at least $2m+1$ vertices at this step, which we cannot because there are at most $2m$ vertices. This last step holds because all vertices with degree 0 must be on the first level. Hence, all vertices not on level 0 must be adjacent to at least one edge, and $n \leq 2m$. Thus, our assumption is incorrect and we have proven our desired property. □

Lemma 6.3.16. *The coreness estimate $\hat{k}(v)$ of a vertex v satisfies $\frac{k(v)}{(2+\varepsilon)} \leq \hat{k}(v) \leq (2+\varepsilon)k(v)$ for any constant $\varepsilon > 0$.*

Proof. Suppose $\hat{k}(v) = (1+\delta)^g$. Then, by Lemma 6.3.15,

$$\frac{(1+\delta)^g}{(2+3/\lambda)(1+\delta)} \leq k(v) \leq (2+3/\lambda)(1+\delta)^{g+1}.$$

Then, solving the above bounds, $\frac{k(v)}{(2+3/\lambda)(1+\delta)} \leq \hat{k}(v) \leq (2+3/\lambda)(1+\delta)k(v)$. For any constant $\varepsilon > 0$, there exists constants $\lambda, \delta > 0$ where $\frac{k(v)}{2(1+\varepsilon)} \leq \hat{k}(v) \leq 2(1+\varepsilon)k(v)$. □

By Lemma 6.3.16, it suffices to return $\hat{k}(v)$ as the estimate of the core number of v. Lemma 6.3.16 then proves the approximation factor in Theorem 6.1.2.

Using our deterministic and space-efficient data structures, we can obtain the following additional results.

Corollary 6.3.17. *Provided an input graph with m edges, and a batch of updates $\mathcal{B}$, our algorithm maintains a $(2+\varepsilon)$-approximation of the coreness values for all vertices (for any constant $\varepsilon > 0$) in $O(|\mathcal{B}|\log^2 m)$ amortized work and $O(\log^3 m)$ depth worst-case, using $O(n\log^2 m + m)$ space.*

Corollary 6.3.18. *Provided an input graph with m edges, and a batch of updates $\mathcal{B}$, our algorithm maintains a $(2+\varepsilon)$-approximation of the coreness values for all vertices (for any constant $\varepsilon > 0$) in $O(|\mathcal{B}|\log^2 m)$ amortized work and $O(\log^4 m)$ depth worst-case, using $O(n+m)$ space.*

Using this data structure, we show a bound for the outdegree if we orient all edges towards neighbors at higher levels and orient edges from higher ID to lower ID in the same level.

Corollary 6.3.19. *The PLDS maintained by Algorithm 20 provides a low out-degree orientation of out-degree at most $(8+\varepsilon)\alpha$ where α is the arboricity, if all edges are oriented from vertices in lower levels to vertices in higher levels and edges between vertices in the same level are oriented from higher ID to lower ID.*

Proof. Let the degeneracy of the graph be d. As is well-known, the degeneracy of the graph is equal to k_{max} where k_{max} is the maximum k-core of the graph. Furthermore, it is well-known that $d \leq 2\alpha$. By Lemma 6.3.16, the vertices in the largest k-core in the graph are in a level with group number at most $\log_{(1+\delta)}((2+3/\lambda)d) + 1$. This means that the

Algorithm 24 Static Approximate k-core Decomposition

Input: An undirected graph $G(V, E)$.
Output: An array of $(2+\varepsilon)$-approximate coreness values.

$\forall v \in V$, let $C[v] = |N(v)|$
Let M be a bucketing structure formed by initially assigning each $v \in V$ to the $\left\lceil \log_{1+\varepsilon} C[v] \right\rceil$'th bucket.
$finished \leftarrow 0, t \leftarrow 0$.
while $(finished < |V|)$ **do**
  $(I, bkt) \leftarrow$ Vertex IDs and bucket ID of next (peeled) bucket in M.
  if $> \log_{1+\delta}(n)$ iterations with $t = bkt$ **then** $t \leftarrow t + 1$
  else if $bkt \neq t$ **then** $t \leftarrow bkt$
  $R \leftarrow \{(v, r_v) \mid v \in N(I), r_v = |\{(u, v) \in E \mid u \in I\}|\}$
  $U \leftarrow$ Array of length $|R|$.
  parfor $R[i] = (v, r_v)$, $i \in [0, |R|)$ **do**
    $inducedDeg = C[v] - r_v$
    $C[v] = \max(inducedDeg, \left\lceil (1+\varepsilon)^{t-1} \right\rceil)$
    $newbkt = \max(\left\lceil \log_{1+\varepsilon} C[v] \right\rceil, t)$
    $U[i] = (v, newbkt)$
  Update M for each $(u, newbkt)$ in U.

up-degree of each vertex in that group is at most $(2+3/\lambda)(1+\delta)^{\log_{(1+\delta)}((2+3/\lambda)d)} = (4+\varepsilon)d$ for any constant $\varepsilon > 0$ for an appropriate setting of $\lambda > 0$. We then obtain an $(8+\varepsilon)\alpha$ out-degree orientation where α is the arboricity of the graph. □

Our data structure also provides an approximation of the densest subgraph within the input graph by the Nash-Williams theorem.

Corollary 6.3.20. *The PLDS maintained by Algorithm 20 provides an $(8+\varepsilon)$-approximation on the density of the densest subgraph within the input graph.*

6.4 Static $(2+\varepsilon)$-Approximate k-core

Due to the P-completeness of k-core decomposition for $k \geq 3$ [AM84], all known static exact k-core algorithms do not achieve polylogarithmic depth. We introduce a linear work and polylogarithmic depth $(2+\varepsilon)$-approximate k-core decomposition algorithm based on the parallel bucketing-based peeling algorithm for static *exact* k-core decomposition of Dhulipala et al. [DBS17]. The algorithm maintains a mapping M from $v \in V$ to a set of *buckets*, with the bucket for a vertex $M(v)$ changing over the course of the algorithm. The algorithm starts at $k = 0$, peels all vertices with degree at most k, increments k, and repeats until the graph becomes empty. The k-core value of v is the value of k when v is peeled. We observe that the dynamic algorithm in this chapter can be combined with a peeling algorithm like the above to yield a linear-work approximate k-core algorithm with polylogarithmic depth.

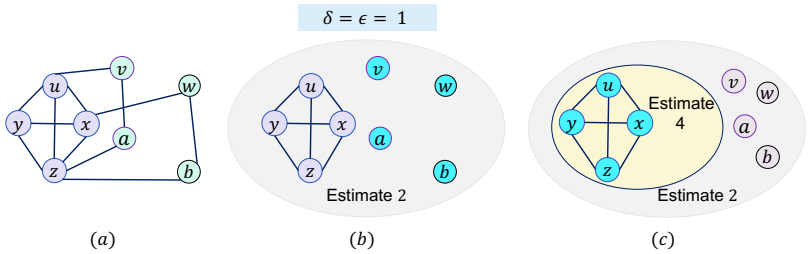


Figure 6-5: Example of a run of Algorithm 24 described in Example 6.4.1.

Algorithm 24 shows pseudocode for our approximate k-core algorithm, which computes an approximate coreness value for each vertex. The algorithm sets the initial coreness estimates, $C[v]$, of each vertex to its degree (Line 1). Then, it maintains a parallel bucketing data structure M, which maps each vertex to the $\lceil \log_{1+\varepsilon} C[v] \rceil$'th bucket (Line 2). It initializes a variable *finished* $= 0$ to keep track of the number of vertices peeled and a variable $t = 0$ used to compute the approximate core values (Line 3). The rest of the algorithm performs peeling, where the peeling thresholds are powers of $(1+\varepsilon)$. The peeling loop (Line 4–Line 15) first extracts the lowest non-empty bucket from M (Line 5), which consists of I, a set of vertex IDs of vertices that are being peeled, and the bucket number bkt. If more than $\log_{1+\delta}(n)$ rounds of peeling have occured at the threshold $(1+\varepsilon)^t$, the algorithm increments t (Line 6). Next, the algorithm computes in parallel an array R of pairs (v, r_v), where v is a neighbor of some vertex in I and r_v is the number of neighbors of v in I (Line 8). Finally, the algorithm computes in parallel the new buckets for the affected neighbors v (Line 10–Line 14). The coreness estimate is updated to the maximum of the peeling threshold of the previous level and the current induced degree of v after r_v of its neighbors are removed. Finally, the algorithm updates the buckets using the new coreness estimates for the updated vertices (Line 15), which can be done in parallel using our bucketing data structure.

We provide an example of this algorithm below.

Example 6.4.1. *Fig. 6-5 shows a run of Algorithm 24 on an example graph. Given the parameters $\varepsilon = \delta = 1$, the two buckets that the vertices of the input graph (shown in (a)) are partitioned into are bucket index 1 (green vertices) and bucket index 2 (purple vertices). Vertices v, w, a, and b have degree 2 so they are put into the bucket with index $\lceil \log_2(2) \rceil = 1$. Since u, x, y, and z have degree ≥ 3, they are put into the bucket with index $\lceil \log_2(3) \rceil = 2$.*

Since the bucket with index 1 has the smaller bucket index, we peel off all the vertices in that bucket (the green vertices) and we assign the core estimate of $(1+\varepsilon)^1 = 2$ to all vertices in that bucket (shown in (b)). We update the buckets of all neighbors of the peeled vertices; however, since u, x, y, and z all still have degree ≥ 3, they remain in the bucket with index 2. Finally, we peel bucket index 2 and assign all vertices in that bucket an estimate of $(1+\varepsilon)^2 = 4$ (shown in (c)). In this example, the estimates produced are 3-approximations of the real coreness values.

We prove below that Algorithm 24 finds an $(2+\varepsilon)$-approximate k-core decomposition in $O(m)$ expected work and $O(\log^3 m)$ depth *whp*, using $O(m)$ space, as stated in Theorem 6.1.1. Our proof uses the quality guarantees in Lemma 8 of [GLM19], as well as an

efficient parallel semisort implementation [GSSB15].

Proof of Theorem 6.1.1. Our approximation guarantee is given by Lemma 8 of [GLM19]. Our algorithm uses a number of data structures that we use to obtain our work, depth, and space bounds. Our parallel bucketing data structure (Line 2) can be maintained via a sparse set (hash map), or by using the bucketing data structure from [DBS17]. The outer loop iterates for $O(\log n)$ times (Line 4). Within each iteration of the outer loop, we iterate for $O(\log_{(1+\delta)} n) = O(\log n)$ rounds for constant δ. After obtaining a set of vertices, we update the buckets using semisort in $O(\log n)$ depth *whp* [DBS17]. Thus the overall depth of the algorithm is $O(\log^3 m)$ for any constant $\delta > 0$.

The work of the algorithm can be bounded as follows. We charge the work for moving a vertex from its current bucket to a lower bucket within a given round to one of the edges that was peeled from the vertex in the round. Thus the total number of bucket moves done by the algorithm is $O(m)$. Each round of the algorithm also peels a number of edges and aggregates, for each vertex that has a neighbor in the current bucket, the number of edges incident to this vertex that are peeled (the r_v variable in the algorithm). We implement this step using a randomized semisort [GSSB15]. Since $2m$ edges are peeled in total, the overall work is $O(m)$ in expectation.

Lastly, we bound the space used by the algorithm. There are a total of $O(\log_{1+\varepsilon} n) = O(\log n)$ buckets for any constant $\varepsilon > 0$. Each vertex appears in exactly one bucket, and thus the overall space of the bucketing structure is $O(n)$. The algorithm also semisorts the edges peeled from the graph in each step. Since all m edges could be peeled and removed within a single step, and thus semisorted the overall space used by the algorithm is $O(m)$. □

The approximation guarantees provided by our algorithm are essentially the best possible, under widely believed conjectures. Specifically, Anderson and Mayr [AM84] show that the optimization version of the High-Degree Subgraph problem, namely to compute the largest core number, or *degeneracy* of a graph cannot be done better than a factor of 2. Thus, obtaining a polynomial work and polylogarithmic depth $(2-\varepsilon)$-approximation to the coreness value of each vertex would yield a $2-\varepsilon$ approximation to the optimization version of the High-Degree Subgraph problem, and show that P = NC, contradicting a widely-believed conjecture in parallel complexity theory.

In recent years, several results have given parallel algorithms that obtain a $(1+\delta)$-approximation to the coreness values in distributed models of computation such as the Massively Parallel Computation model [ELM18, GLM19]. These results work by performing a *random sparsification* of the graph into a subgraph that approximately preserves the coreness values. They then send this subgraph to a single machine, which runs the sequential peeling algorithm on the subgraph to find approximate coreness values. Crucially, this second peeling step on a single machine can have $\Theta(n)$ depth, and thus, this approach does not yield a polylogarithmic depth algorithm in the work-depth model of computation.

Table 6.2: Sizes of graph inputs.

Graph Dataset	Num. Vertices	Num. Edges
dblp	425,957	2,099,732
brain	784,262	267,844,669
wiki	1,140,149	2,787,967
youtube	1,138,499	5,980,886
stackoverflow	2,601,977	28,183,518
livejournal	4,847,571	85,702,474
orkut	3,072,627	234,370,166
ctr	14,081,816	16,933,413
usa	23,072,627	28,854,312
twitter	41,652,231	2,405,026,092
friendster	65,608,366	3,612,134,270

6.5 Experimental Evaluation

Setup We use `c2-standard-60` Google Cloud instances, which have 30 cores with two-way hyper-threading (3.1 GHz Intel Xeon Cascade Lake) and 236 GiB memory, and `m1-megamem-96` Google Cloud instances, which have 48 cores with two-way hyper-threading (2.0 GHz Intel Xeon Skylake) and 1433.6 GB memory. Our programs use a work-stealing scheduler, and are compiled using `g++` (version 7.5.0) with the `-O3` flag. We terminate experiments that take over 3 hours. We test our algorithms on real-world undirected graphs from SNAP [LS16], the DIMACS Shortest Paths challenge road networks [DGJ08], and Network Repository [RA15], shown in Table 6.2, namely ***dblp***, ***brain***, ***wiki***, ***orkut***, ***friendster***, ***stackoverflow***, ***usa***, ***ctr***, ***youtube***, and ***livejournal***. We also used ***twitter***, a symmetrized version of the Twitter network [KLPM10]. We remove duplicate edges as well as zero degree edges and self-loops. The table reflects the size of each graph *after* this preprocessing. Both *stackoverflow* and *wiki* are temporal networks obtained from the SNAP database. For these two networks, we maintain the same order of edge insertions and deletions in the order that they occur temporally as provided by SNAP. *usa* and *ctr* are two high diameter road network graphs and *brain* is a highly dense network of the human brain where the largest k-core has size ≥ 1200.

All experiments are run on the `c2-standard-60` instances, except for *twitter* and *friendster*, which are run on the `m1-megamem-96` instances as they require more memory. The edge updates for the dynamic algorithms (for all except the temporal networks) are generated by taking a random permutation of a list containing two copies of each edge, where the first appearance of an edge is an insertion, and the second appearance is a deletion. Batches are generated by taking regular intervals of the list. For the experiments on insertion-only batches, we ignore the second appearance of each edge and, likewise, the first for deletion-only batches.

For the static algorithms in the batch-dynamic setting, within each batch we order all insertions in the batch before all deletions. Then, we generate two static graphs per batch, one following all insertions, and the other following all deletions. We re-run the

static algorithms on each static graph generated in this manner, to obtain comparable per-batch running times. [5]

Algorithms Unless otherwise noted, we run our parallel algorithms using all available cores with two-way hyper-threading. We make one modification in our parallel implementation of our insertion procedure from our theoretical algorithm which is instead of moving vertices up level-by-level, we perform a parallel filter and sort that calculates the desire-level of vertices we move up. This results in more work theoretically, but we find that, practically, it results in faster runtimes. Also, notably, in practice, we optimized the performance of our PLDS by considering $\lceil \log_{(1+\delta)} n \rceil / 50$ levels per group instead of $\lceil \log_{(1+\delta)} n \rceil$. We also implemented a version of our structure that exactly follows our theoretical algorithm and compared the performance of both structures. We see that even such a simple optimization resulted in massive gains in performance.

We test the following implementations: ***LDS***: the *sequential dynamic* approximation algorithm using the level data structures of Bhattacharya et al. and Henzinger et al. [BHNT15, HNW20]; ***PLDS***: our *parallel batch-dynamic* approximation algorithm; ***PLDSOpt***: our optimized *parallel batch-dynamic* approximation algorithm (that uses less levels per group); ***ApproxKCore***: our *parallel static* approximation algorithm; ***ExactKCore***: the *parallel static* exact algorithm of Dhulipala et al. [DBS17]; ***Hua***: the *parallel, batch-dynamic* exact algorithm of Hua et al. [HSY+20]; and ***Sun***: the *sequential dynamic* approximation algorithm of Sun et al. [SCS20].

We implemented our algorithms using the Graph Based Benchmark Suite [DBS18b], and we use the atomic compare-and-swap and fetch-and-add instructions. We also used a concurrent hash table with linear probing [SB14] for the level sets $L[v][j]$ and $U[v]$. For deletions, we used the folklore *tombstone* method: when an element is deleted, we mark the slot in the table as a tombstone, which can be reused, or cleared during a table resize.

Accuracy vs. Running Time We evaluated the empirical error ratio of the per-vertex core estimates given by our algorithms (LDS, PLDS, and PLDSOpt) and Sun on *dblp* and *livejournal*, using batches of size 10^5 and 10^6, respectively, consisting of both insertions and deletions. The results are shown in Fig. 6-6. The figure shows the average batch time against the average and maximum per-vertex core estimate error ratio.[6] The parameters that we use for LDS, PLDS, and PLDSOpt are $\delta = \{0.1, 0.2, 0.4, 0.8, 1.6, 3.2\}$ and $\lambda = \{3, 6, 12, 24, 48, 96\}$. For Sun, we use the parameters $\delta_{sun} = \lambda_{sun} = \{0.1, 0.2, 0.4, 0.8, 1.6, 3.2\}$, and $\alpha_{sun} = \{2(1 + 3\delta_{sun})\}$ (these are parameters used in their algorithm [SCS20]). We also tested $\alpha_{sun} = 1.1$, as done in Sun et al.'s code [SCS20]. In this setting, the theoretical bounds given by Sun et al. [SCS20] no longer hold, but it gives better estimates empirically. We compare this heuristic setting to a similar heuristic in our PLDS and LDS algorithms, where we pick a value of $\lambda < 3$ such that $(2 + 3/\lambda) = 1.1$ for $\delta = \{0.4, 0.8, 1.6, 3.2\}$.

[5] The insertions and deletions are processed separately in the static algorithms, because in large batches many of the insertions and deletions in the generated updates cancel each other out.

[6] This error ratio is computed as $\max(\frac{approx}{exact}, \frac{exact}{approx})$. If the exact core number is 0, we ignore the vertex in our error ratio; for vertices of non-zero degree, the lowest estimated core number is 1.

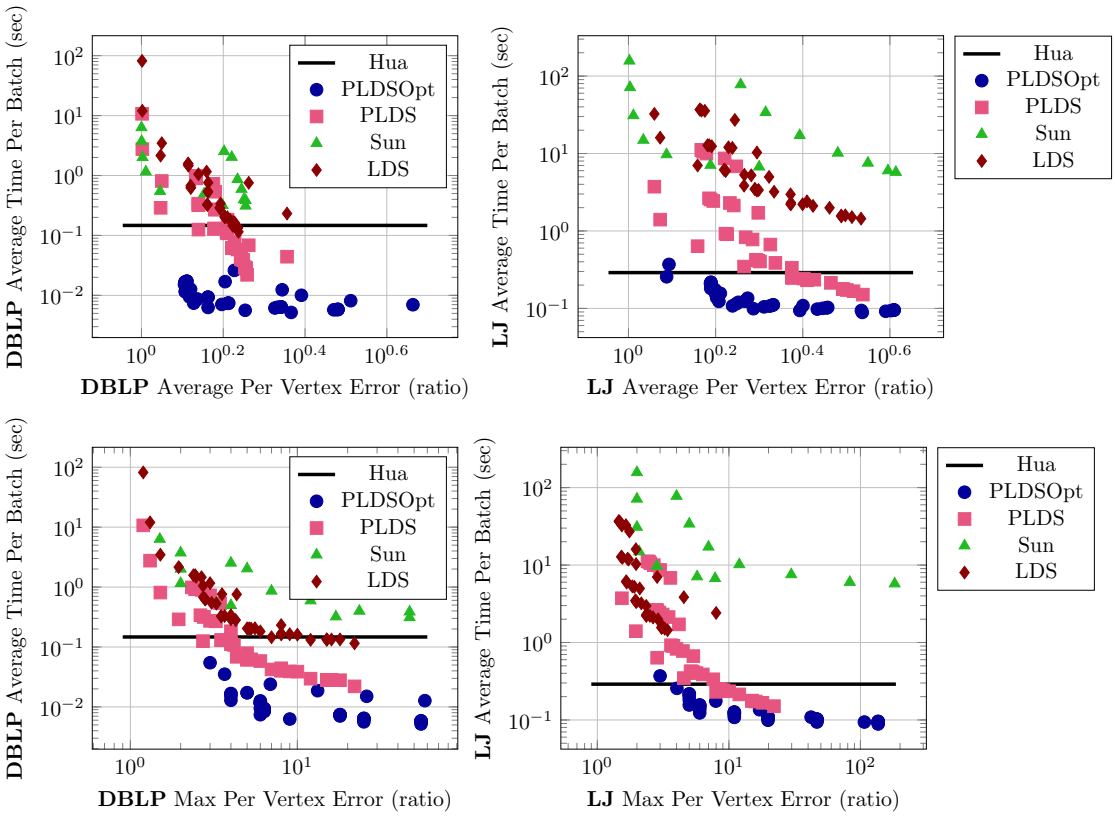


Figure 6-6: Comparison of the average per-batch time versus the average (top row) and maximum (bottom row) per-vertex core estimate error ratio of LDS, PLDS, PLDSOpt and Sun, using varying parameters, on the *dblp* and *livejournal* graphs, with a batch size of 10^5 and 10^6, respectively. The data uses theoretically efficient parameters as well as those using $(2 + 3/\lambda) = \alpha_{sun} = 1.1$. The runtime for Hua is shown as a black horizontal line.

Overall, we see that using theoretically efficient parameters, our PLDSOpt, PLDS and LDS algorithms are faster than Sun, for parameters that give similar average and maximum per-vertex core estimate error ratios. For *dblp*, PLDSOpt achieves 22.35–195.82x and PLDS achieves 11.15–25.02x speedups over Sun, and for *livejournal*, PLDSOpt achieves 27.64–497.63x and PLDS achieves 36.44–57.43x speedups over Sun. Using the same parameters, for *dblp*, PLDSOpt achieves a 3.43–57.124x speedups over PLDS; for *livejournal*, the speedups are 1.70–51.36x. Moreover, PLDS achieves 1.41–5.23x speedups over LDS on *dblp*, and 3.30–9.57x speedups over LDS on *livejournal*, using the same parameters; PLDS does not show as much speedup over LDS on *dblp* because it is a small graph. PLDS also has much lower maximum error ratios compared to Sun and PLDSOpt on parameters that give similar average error ratios. PLDSOpt gives similar maximum errors to Sun with maximum error factors of 4–58 compared to Sun's maximum error factors of 1.5–47 for *dblp*; for *livejournal*, PLDSOpt gives maximum error factors of 3–136.3 compared to Sun's maximum error of 4–182.

The heuristic settings in both our algorithms and Sun give better approximations at the cost of speed. We achieve similar error ratios with up to 3.96x speedup using PLDS

on *dblp*, and up to 22.64x speedup using PLDS on *livejournal*. Also, PLDS achieves 4.28–11.01x speedup over LDS on *dblp*, and 8.71–11.43x speedup over LDS on *livejournal*. Unfortunately, PLDSOpt was not able to achieve comparable error ratios for these settings.

For the rest of the experiments, we fix $\delta = 0.4$ and $\lambda = 3$; these parameters offer a reasonable tradeoff between approximation error and speed, as shown in Fig. 6-6. We also focus on insertion-only and deletion-only batches since inputs are initially filtered anyways into such batches.

Comparison with Hua et al We also compare with a parallel batch-dynamic algorithm for exact k-core by Hua et al. [HSY+20]. We obtained a multicore implementation of their code and tested their code under the same experimental conditions as our code. Although the experiments in [HSY+20] tested the runtime of randomly sampling a batch of insertions or deletions, we also tested our code under such settings and found that it obtained similar speedups to the experiments run under our setting; thus, we present here the results under our experimental setting. Hua et al. included a timing function in their code which we use to time their code. However, this timing function does not include the time to process the graph and maintain their data structures; we include all such times (for updating the graph and maintaining our data structures) in our code. If we include this time in Hua et al's implementation, their running times increase by up to 8x for some experiments. But for our experiments, we use the original timing functionality in Hua et al.'s code *without* this additional time.

6.5.1 Experiments on Insertions

Batch Size vs. Running Time Fig. 6-7 (first row) shows the average insertion-only per-batch running times on varying batch sizes for LDS, PLDS, PLDSOpt, ExactKCore, and ApproxKCore on *dblp* and *livejournal*. ExactKCore and ApproxKCore recompute the k-core decomposition on the current batch as well as all previously inserted edges. We see that PLDSOpt is 16.55–6205.19x and PLDS is 1.76–51.81x faster than LDS, and ApproxKCore is 1.37–1.67x faster than ExactKCore overall. PLDSOpt is also 8.47-18.91x faster than PLDS. Moreover, for *livejournal*, ExactKCore and ApproxKCore both time out for small batch sizes. For *dblp*, we see that PLDSOpt is up to 75.94x faster, PLDS is up to 7x faster than ApproxKCore for small batch sizes, and even our sequential LDS is up to 3.98x faster than ApproxKCore. Against Hua, PLDSOpt achieves a speedup over all batches from 5.17–16.43x for *dblp* and 15.97–114.52x for *livejournal*. PLDS achieves speedups over Hua for the smaller batches, a speedup of 1.51x for *dblp* and up to 13.51x speedup for *livejournal*. Finally, ApproxKCore achieves speedups of up to 33.90x for *dblp* and 42.02x for *livejournal* over Hua for the larger batch sizes.

Thread Count vs. Running Time We focus on the parallel speedups of our dynamic algorithms since the parallel speedups of static algorithms have been well-studied. Fig. 6-8 (top row) shows the scalability of PLDSOpt, PLDS, and Hua with respect to their single-thread running times on *dblp* and *livejournal* for insertion-only batches. We display the scalability of the algorithms on dynamic inputs, of insertion-only batches of size 10^6.

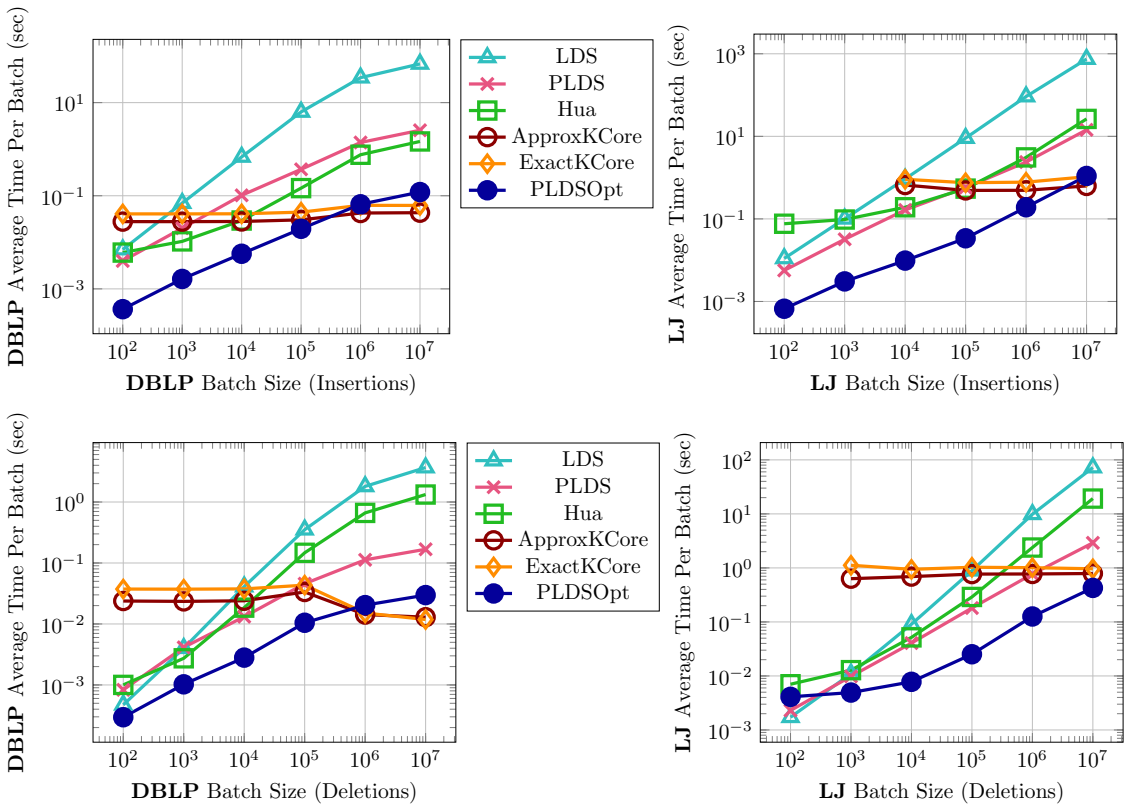


Figure 6-7: Average insertion-only (top row) and deletion-only (bottom row) per-batch running times on varying batch sizes for LDS, PLDS, PLDSOpt, ExactKCore, and ApproxKCore on *dblp* and *livejournal*. The missing batch sizes for ApproxKCore and ExactKCore timed out at 3 hours.

PLDSOpt and PLDS achieve up to 30.28x and 26.46x self-relative speedup, respectively. Hua achieves up to a 2.07x self-relative speedup. We see that our PLDS algorithms achieve greater self-relative speedups than Hua.

Additional Graphs Fig. 6-9 (top) shows the runtimes of PLDSOpt, Hua, PLDS, ExactKCore, and ApproxKCore on additional graphs, using insertion-only batches, all of size 10^6. Hua, ExactKCore and ApproxKCore timed out on *twitter* and *friendster*. For the remaining graphs, ApproxKCore is up to 1.85x faster than ExactKCore on average per batch. For the smaller graphs (*dblp*, *youtube*, and *orkut*), ApproxKCore is up to 4.55x faster than PLDSOpt and up to 48.29x faster than PLDS on average per batch, because 10^6 is a relatively large batch size for these graphs, so it is faster to re-run our static algorithm compared to our batch-dynamic algorithm. For the larger graphs (except for the graphs where ApproxKCore times out), PLDSOpt achieves up to a 13.09x speedup and PLDS achieves up to a 7.41x speedup against ApproxKCore. We see that among these graphs both PLDSOpt and PLDS perform particularly well in the high-diameter road networks suggesting that the dynamic algorithms perform many less changes to the data structures than the

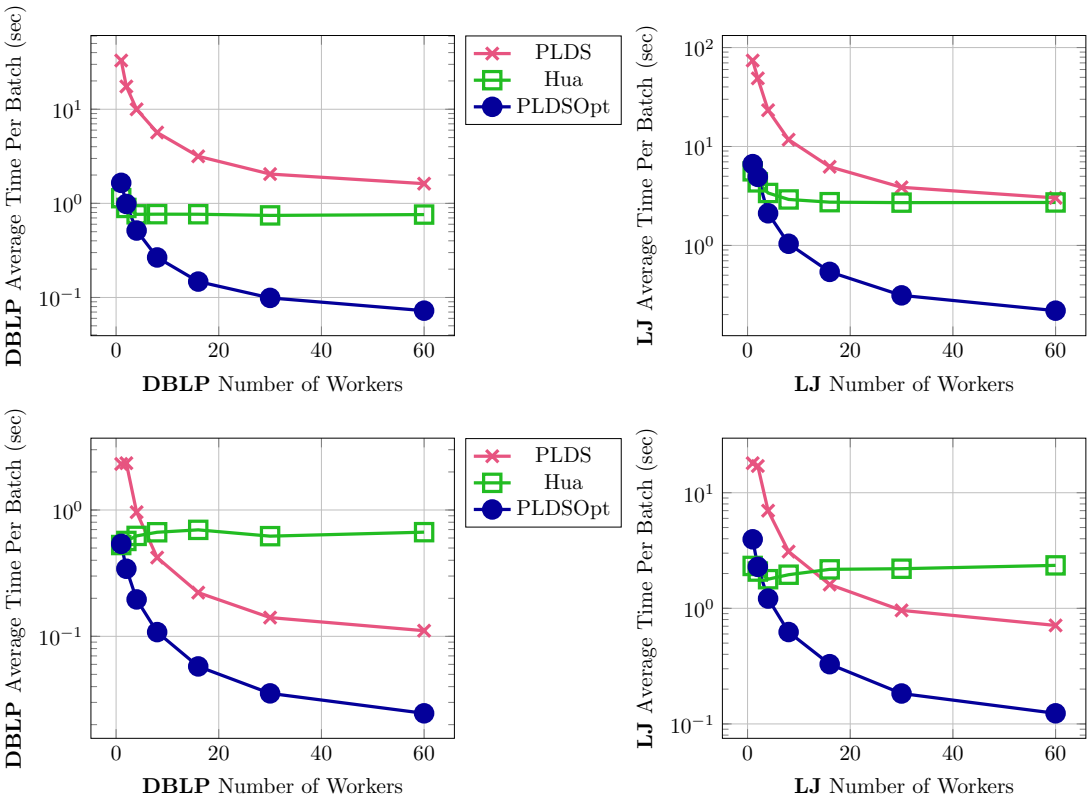


Figure 6-8: Parallel speedup of PLDSOpt, PLDS and Hua, with respect to their single-threaded running times on *dblp* and *livejournal*, using insertion-only (top row) and deletion-only (bottom row) batches of size 10^6 for all algorithms. The last "60" on the x-axis indicates 30 cores with hyper-threading.

static algorithm for high-diameter (sparser) networks. For the largest graphs (*twitter* and *friendster*), PLDSOpt and PLDS are orders of magnitude faster as ApproxKCore times out.

Compared against Hua, PLDSOpt achieves speedups of 6.20–58.66x on all graphs that Hua did not time out for and is orders of magnitude faster on *twitter* and *friendster* since Hua times out. Except for the two smallest graphs, *dblp* and *wiki*, PLDS achieves speedups of 1.30–49.02x for all graphs that Hua did not time out for and is orders of magnitude faster on the two largest graphs. On *brain*, *twitter*, and *friendster*, PLDSOpt achieves 13.83x, 5.82x, and 12.40x speedups, respectively, over PLDS, suggesting the optimization is most beneficial for dense graphs.

We also compared the performance of ApproxKCore and ExactKCore using the full graph on all datasets. We found ApproxKCore to be 1.7–3.9x faster than ExactKCore, with an average coreness error ratio of 1.044–1.172.

Accuracy of Approximation Algorithms We also computed the average and maximum errors of all of our approximation algorithms for our experiments shown in Fig. 6-9. The data for these error ratios are shown in Table 6.3. We tested the errors for $\delta = 0.4$

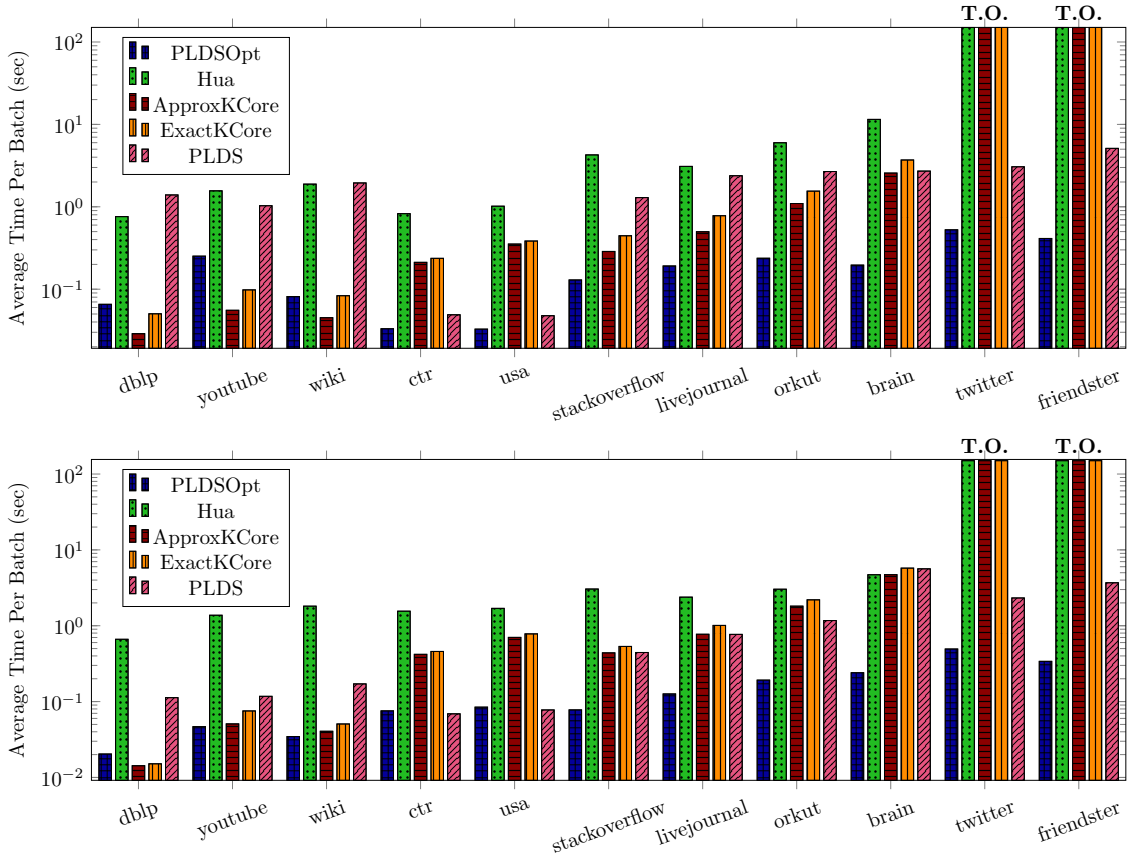


Figure 6-9: Average per-batch running times for PLDSOpt, Hua, PLDS, ApproxKCore, and ExactKCore, on *dblp*, *youtube*, *wiki*, *ctr*, *usa*, *stackoverflow*, *livejournal*, *orkut*, *brain*, *twitter*, and *friendster* with batches of size 10^6 (and approximation settings $\delta = 0.4$ and $\lambda = 3$ for PLDSOpt and PLDS). Hua, ApproxKCore, and ExactKCore timed out (T.O.) at 3 hours for *twitter* and *friendster*. The top graph shows insertion-only batch runtimes and the bottom graph shows deletion-only batch runtimes.

and $\lambda = 3$. According to our theoretical proofs, the maximum error (for PLDS) should be $(2 + 3/3) \cdot (1 + 0.4) = 4.2$. We see that indeed the maximum empirical error for PLDS falls under this hard constraint. PLDSOpt achieves an average error of 1.359–2.118 on insertion batches (left half) of Table 6.3 compared to errors of 1.593–3.363x for PLDS and 1.010–4.175 for ApproxKCore. For max error, PLDSOpt achieves a max error of 3 (less than the theoretical optimal) compared to 3–4.193 for PLDS and 3–4.305 for ApproxKCore. For *twitter* and *friendster*, we sampled error counts uniformly at random with probability 1/10 due to our timeout constraints. PLDS and ApproxKCore computations timed out on both of these datasets (even with sampling). We see that decreasing the number of levels improves the error ratio for insertion-only batches.

Table 6.3: Average and maximum errors of PLDSOpt, PLDS, and ApproxKCore on insertion-only and deletion-only batches of size 10^6. Insertion-only batches errors are shown on the left and deletion-only batches are shown on the right. * indicates that the error was obtained via sampling the error probability with 1/10 probability. T.O. indicates that the program timed out at 3 hours (even when performing the sampling with 1/10 probability).

Graph Dataset	PLDSOpt Avg.	PLDSOpt Max	PLDS Avg.	PLDS Max	Approx KCore Avg.	Approx KCore Max
dblp	1.9345	3	2.635	4	1.15	3.875
brain	1.834	3	3.363	4.193	1.315	4.305
wiki	1.590	3	1.780	4.172	1.010	3
youtube	1.359	3	1.593	4	1.1283	3.75
stackoverflow	1.826	3	2.272	4.067	1.048	3.875
livejournal	1.660	3	2.321	4.175	4.175	1.165
orkut	1.926	3	3.115	4.175	1.204	4.2
ctr	1.601	3	1.683	3	1.374	3
usa	1.826	3	1.683	3	1.379	3
twitter	2.118*	3*	T.O.	T.O.	T.O.	T.O.
friendster	1.851*	3*	T.O.	T.O.	T.O.	T.O.

Graph Dataset	PLDSOpt Avg.	PLDSOpt Max	PLDS Avg.	PLDS Max	Approx KCore Avg.	Approx KCore Max
dblp	1.187	6	1.507	2	1.236	3.0
brain	1.575	6	1.943	4.186	1.315	5.0
wiki	1.423	4	1.494	4	1.013	3.875
youtube	1.268	4	1.317	4	1.137	3.706
stackoverflow	1.630	6	1.792	4.172	1.045	3.908
livejournal	1.613	6	1.704	4.14	1.167	3.984
orkut	1.681	6	1.913	4.175	1.205	4.2
ctr	1.243	3	1.257	3	1.524	3.0
usa	1.253	3	1.278	3	1.522	3.0
twitter	1.893*	4*	T.O.	T.O.	T.O.	T.O.
friendster	1.685*	3*	T.O.	T.O.	T.O.	T.O.

Space Usage Finally, we tested the space usage of our parallel batch-dynamic programs against the space usage needed by Hua. We implemented functions that counted the space usage of the entire level data structure used in our programs and the data structures used in the Hua code using the `sizeof` operator. Fig. 6-10 (top row) shows the results of our experiment. As expected, since our data structures require $O(n \log^2 m + m)$ asymptotic space, our space usage for PLDS is up to 63.87x more and 62.50x more than Hua for *dblp* and *livejournal*, respectively. However, our PLDSOpt uses less memory than Hua in most settings (up to 1.53x factor less memory) for *dblp* and up to 1.08x additional space in a few cases; for *livejournal*, it uses up to 1.72x additional space.

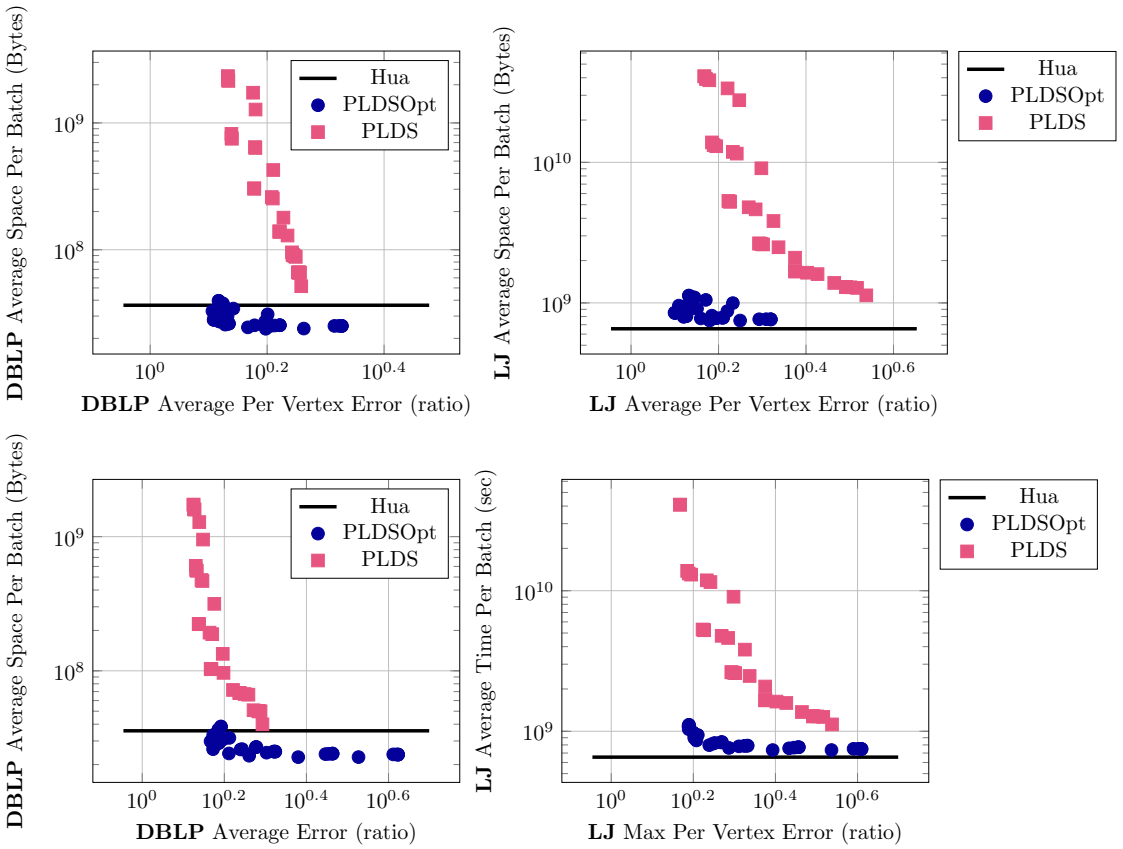


Figure 6-10: Average space usage in bytes for PLDSOpt, Hua, and PLDS in terms of the average error. We varied δ and λ and computed the error ratio and space usage for the programs on *dblp* and *livejournal*. We tested against 10^5 insertion-only (top row) and deletion-only (bottom row) batches for *dblp* and 10^6 batch sizes for *livejournal*.

6.5.2 Experiments on Deletions

In this subsection, we present deletion-only experimental results of our LDS, PLDSOpt, PLDS, and ApproxKCore algorithms using the same environment as our insertion-only ex-

periments. In general, for our batch-dynamic implementations, deletions are faster than insertions due to the fact that we move each vertex at most once. Specifically, when determining vertices to move up a level, in order to check the requisite invariants, each vertex must maintain its neighbors in the levels below, keyed by level number. This takes more work than the symmetric computation for deletions, where each vertex simply maintains the set of neighbors in the levels above and needs not key these neighbors by their level numbers. Furthermore, PLDSOpt results in slightly greater error for the deletion case because we decrease the number of levels.

Batch Size vs. Running Time Fig. 6-7 (bottom row) shows the average deletion-only per-batch running times on varying batch sizes for LDS, PLDSOpt, PLDS, ExactKCore, and ApproxKCore on *dblp* and *livejournal*. ExactKCore and ApproxKCore recompute the k-core decomposition on the graph after removing the current batch, as well as all edges in previous batches. We see that PLDSOpt is up to 7.17x faster than PLDS, PLDS is up to 24.75x faster than LDS, and ApproxKCore is up to 1.59x faster than ExactKCore overall. Compared to using insertion-only batches, PLDS is slower than LDS on smaller batches, and the speedup of PLDS over LDS is also smaller. We believe that this is due to the lower amount of work in the desire level computations for deletion-only batches, where LDS is faster at processing deletion-only batches compared to insertion-only batches. However, the speedups of PLDSOpt and PLDS over Hua are 3.39–44.58x and 1.2–7.89x, respectively, for *dblp* and 1.71–45.01x and 1.27–6.59x, respectively, for *livejournal*. The additional speedups of PLDS over Hua is due to our more efficient deletion procedure that does less work than our insertion procedure.

Moreover, for *livejournal*, ExactKCore and ApproxKCore both time out for the small batch sizes. For *dblp*, we see that PLDS is up to 26.61x faster than ApproxKCore for small batch sizes, and even our sequential LDS is up to 50.38x faster than ApproxKCore. For ApproxKCore and ExactKCore, the running times are similar to the insertion-only case, except for large batch sizes on *dblp*, where the deletion-only batches are faster to process. This is because *dblp* is small, with fewer than 3×10^6 edges, and so in the deletion-only setting, we are left with a relatively small graph (or an empty graph in the 10^7 batch size case) after the first batch. On the other hand, in the insertion-only setting, we have almost half of the graph (or the full graph in the 10^7 batch size case) after the first batch. Thus, there is greater processing required overall in the insertion-only setting, for small graphs with large batch sizes.

Thread Count vs. Running Time Fig. 6-8 (bottom row) shows the scalability of PLDS with respect to its single-thread running times on *dblp* and *livejournal* for deletion-only batches of size 10^6. PLDSOpt achieves up to a 32.02x self-relative speedup and PLDS achieves up to 25.33x self-relative speedup, similar to the speedups obtained in the insertion-only setting. Unlike the case with insertions, Hua achieves no speedup on *dblp* and up to a 1.30x self-relative speedup on *livejournal*.

Additional Graphs Fig. 6-9 (bottom graph) shows the runtimes of PLDSOpt, Hua, PLDS, ExactKCore, and ApproxKCore on the remaining graphs, using deletion-only

batches of size 10^6. Hua, ExactKCore and ApproxKCore timed out at 3 hours on *twitter* and *friendster*. PLDSOpt is uniformly faster than all other programs for all graphs except *dblp*, achieving a maximum speedup of 23.45x against PLDS, 19.64x against ApproxKCore, and 52.36x against Hua (on all graphs that did not time out). These results are similar to those obtained for insertions, although the slight speedup could be due to fewer levels causing deletions to achieve greater speedup than insertions. For *youtube* and *orkut*, ApproxKCore is up to 1.4x faster than ExactKCore on average per batch. Additionally, for these graphs, ApproxKCore is up to 2.26x faster than PLDS on average per batch, for the same reason as discussed earlier (10^6 is a relatively large batch size for these graphs, and so it is faster to re-run our static algorithm compared to our batch-dynamic algorithm). For the larger graphs (*twitter* and *friendster*), PLDS is orders of magnitude faster, as ApproxKCore times out.

Accuracy and Space Usage Table 6.3 shows a slight increase in the maximum error for PLDSOpt, with a max error of 6. This is expected theoretically; recall the theoretical upper bound on the error is 4.2 (followed by PLDS). PLDS and ApproxKCore do not show noticeable differences in error. Fig. 6-10 (bottom row) shows the space usage of PLDS, PLDSOpt, and Hua for deletions. Similar to insertions, PLDSOpt uses less space than Hua for most cases for *dblp*, up to 1.57x factor less space. PLDSOpt has a maximum space usage of 1.07x and 1.69x for *dblp* and *livejournal* over Hua, and PLDS a 48.50x and 21.15x factor over Hua.

6.6 Additional Data Structure Implementations

In addition to the randomized data structures presented in Section 3.4, we present two additional sets of data structures that we can use to obtain a *deterministic* and a *space-efficient* $(2+\varepsilon)$-approximate k-core algorithms.

The work of all of our randomized, deterministic, and space-efficient algorithms are the same; however, using randomization allows us to obtain a better depth with slightly less complicated data structures.

Deterministic Data Structures We initialize an array U, of size n. Each vertex is assigned a unique index in U. The entry for the i'th vertex, $U[i]$, contains a pointer to a dynamic array that stores the neighbors of vertex v_i at levels $\geq \ell(v_i)$. Each vertex v_i also stores another dynamic array, L_{v_i}, that contains pointers to a set of dynamic arrays storing the neighbors of v_i partitioned by their levels j where $j < \ell(v_i)$. Specifically, we maintain a separate dynamic array for each level from level 0 to level $\ell(v_i)-1$ storing the neighbors of v_i at each respective level. We also maintain the current level of each vertex in an array.

To perform a batch of insertions into a dynamic array, we insert the elements at the end of the array. The array is resized and doubles in size if too many elements are inserted into the array (and it exceeds its current size). For a batch of deletions, the deletions are initially marked with a "deleted" marker indicating that the element in the slot has been deleted. A counter is used to maintain how many slots contain "deleted." Then, once a

constant fraction of elements (e.g. 1/2) has "deleted" marked in their slots, the array is cleaned up by reassigning vertices to new slots and resizing the array.

$O(n+m)$ Total Space Data Structures Here we describe how to reduce the total space usage of our data structures to $O(m)$. All of our previous data structures use $O(n\log^2 m + m)$ space, which means that when $m = O(n)$, we use space that is superlinear in the size of the graph. To reduce the total space to $O(m)$, we maintain two structures for L_{v_i}. We can use either the deterministic or randomized structures for the other structures. Each L_{v_i} is maintained as a linked list. The j'th node in the linked list maintains the number of neighbors of v_i at the j'th non-empty level (a non-empty level is one where v_i has neighbors at that level) that is less than $\ell(v_i)$. The node representing a level is removed from the linked list when the level becomes empty. Each node in L_{v_i} contains pointers to vertices at the level represented by the node. Each vertex then contains pointers to every edge it is adjacent to and every edge contains pointers to the two nodes in the two linked lists representing the levels in which endpoints of the edge reside. Using either dynamic arrays or hash tables for the lists of neighbors allow us to maintain these data structures in $O(m)$ space. Since we only maintain a node in our linked list for every non-empty level, our linked list contains precisely the number of elements equal to $2m$.

6.7 Potential Argument for Work Bound

Our work bound uses the potential functions presented in Section 4 of [BHNT15]. We show that we can analyze our algorithm using these potential functions and our parallel algorithm serializes to a set of sequential steps that obey the potential function. We obtain the following lemma by the potential argument provided in this section.

Lemma 6.7.1. *For a batch of $\mathcal{B} < m$ updates, Algorithm 20 returns a PLDS that maintains Invariant 1 and Invariant 2 in $O(\mathcal{B}\log^2 m)$ amortized work and $O(\log^2 m \log\log m)$ depth whp, using $O(n\log^2 m + m)$ space.*

6.7.1 Proof of Work Bound

Unlike the algorithm presented in [HNW20, BHNT15], in each round, to handle deletions, we recompute the $\mathsf{dl}(v)$ of any vertex v that we want to move to a lower level. Specifically, we compute and move v to the closest level that satisfies both Invariant 1 and Invariant 2. This is a different algorithm from the algorithm presented in [HNW20, BHNT15], and so we present for completeness a work argument for our modified algorithm. The work bound we present accounts for the work of any one movement up or down levels using the potential function argument of [BHNT15]. Note that this potential function also gives us the amortized work per edge update since there exists a corresponding set of sequential updates that cannot do less work than the set of parallel updates.

Charging the Cost of Moving Levels The strategy behind our potential function is use the *increase* in our potential function due to edge updates to pay for the *decrease* in

potential due to vertices moving up or down levels. We can then charge our costs to the increase in potential due to edge updates. Below, we bound the increase in potential due to edge updates and the decrease in potential due to vertex movements.

We use the following potential function to calculate our potential. First, recall some notation. Let Z_i be the set of vertices in levels i to $K-1$. In other words, $Z_i = \bigcup_{j=i}^{K-1} V_j$. Let $N(u, Z_i)$ be the set of neighbors of u in the induced subgraph given by Z_i. Let $\ell(u)$ be the current level that u is on. Finally, let $gn(\ell)$ be the group number of level ℓ; in other words, $\ell \in g_{gn(\ell)}$. Let $f : [n] \times [n] \to \{0, 1\}$ be a function where $f(u, v) = 1$ when $\ell(u) = \ell(v)$ and $f(u, v) = 0$ when $\ell(u) \neq \ell(v)$. Using the potential functions defined in [BHNT15], for some constant $\lambda > 0$:

$$\Pi = \sum_{v \in V} \Phi(v) + \sum_{e \in E} \Psi(e) \tag{6.1}$$

$$\Phi(v) = \lambda \sum_{i=0}^{\ell(v)-1} \max(0, (2 + 3/\lambda)(1 + \delta)^{gn(i)} - N(v, Z_i)) \tag{6.2}$$

$$\Psi(u, v) = 2(K - \min(\ell(u), \ell(v))) + f(u, v) \tag{6.3}$$

We first calculate the potential changes for insertions and deletions of edges.

Insertion The insertion of an edge (u, v) creates a new edge with potential $\Psi(u, v)$. The new potential has value at most $2K + 1$. With an edge insertion $\Phi(u)$ and $\Phi(v)$ cannot increase. Thus, the potential increases by at most $2K + 1$.

Deletion The deletion of edge (u, v) increases potentials $\Phi(u)$ and $\Phi(v)$ by at most $2\lambda K$. It does not increase any other potential since the potential of edge (u, v) is eliminated.

First it is easy to see that the potential Π is always non-negative. Thus, we can use the positive gain in potential over edge insertions and deletions to pay for the decrease in potential caused by moving vertices to different levels.

Now we discuss the change in potential given a movement of a vertex to a higher or lower level. Moving such a vertex decreases the potential and we show that this decrease in potential is enough to pay for the cost of moving the vertex to a higher or lower level.

A vertex v moves from level i to level $\mathsf{dl}(v) < i$ due to Algorithm 22 Since vertex v moved down at least one level, this means that prior to the move, its up*-degree is $\mathsf{up}^*(v) < (1 + \delta)^{gn(\ell(v)-1)}$. It is moved to a level $\mathsf{dl}(v)$ where its up*-degree is at least $(1 + \delta)^{gn(\mathsf{dl}(v)-1)}$ and its up-degree is at most $(2 + 3/\lambda)(1 + \delta)^{gn(\mathsf{dl}(v))}$ (or it is moved to level 0).

The potential before the move is at least

$$\lambda \sum_{i=0}^{\mathsf{dl}(v)-1} \max\Big(0, (2+3/\lambda)(1+\delta)^{gn(i)} - N(v, Z_i)\Big) + \sum_{i=\mathsf{dl}(v)}^{\ell(v)-1} (\lambda+3)(1+\delta)^{gn(i)}$$

since we only move a vertex to a lower level if $\mathsf{up}^*(v) < (1+\delta)^{gn(\ell(v)-1)}$ and we move it to the closest level $\mathsf{dl}(v)$ where Invariant 2 is no longer violated. To derive the second term, since we moved vertex v to level $\mathsf{dl}(v)$, we know that its degree $|N(v, Z_{\mathsf{dl}(v)})| < (1+\delta)^{gn(\mathsf{dl}(v))}$ (otherwise, we could've moved v to level $\mathsf{dl}(v)+1$). Then, substituting $(1+\delta)^{gn(i)}$ for all levels $i \geq \mathsf{dl}(v)$ into $\Phi(v)$ allows us to obtain $\sum_{i=\mathsf{dl}(v)}^{\ell(v)-1}(\lambda+3)(1+\delta)^{gn(i)}$. Then, when it reaches its final level, we know that it is at the highest level it can move to or at level 0. In both cases,

$$\Phi(v) = \lambda \sum_{i=0}^{\mathsf{dl}(v)-1} \max\Big(0, (2+3/\lambda)(1+\delta)^{gn(i)} - N(v, Z_i)\Big)$$

after the move. In this case, $\Phi(v)$ decreases by at least $\sum_{i=\mathsf{dl}(v)}^{\ell(v)-1}(\lambda+3)(1+\delta)^{gn(i)}$.

We need to account for two potential increases: the increase in Ψ and the increase in Φ from neighbors of v. We first consider the increase in Ψ. The potential increase in $\Psi(u,v)$ for every edge (u,v) where $\ell(u) \geq \mathsf{dl}(v)$ is at most $2(\ell(v) - \mathsf{dl}(v))(1+\delta)^{gn(\mathsf{dl}(v))}$, since v has up-degree at most $(1+\delta)^{gn(\mathsf{dl}(v))}$ at level $\mathsf{dl}(v)$ (otherwise, we can increase its $\mathsf{dl}(v)$) and each of these edge's potential gain is upper bounded by 2 for every level in $[\mathsf{dl}(v), \ell(v)-1]$.

Furthermore, we need to account for the increase in potential of every neighbor whose edge is flipped by the move. The total increase in Φ is at most $\lambda(\ell(v) - \mathsf{dl}(v))(1+\delta)^{gn(\mathsf{dl}(v))}$ for every neighbor on levels $> \mathsf{dl}(v)+1$, since we move v to the *highest* level that satisfies the invariants and $N(v, \mathsf{dl}(v)) < (1+\delta)^{gn(\mathsf{dl}(v))}$. Decreasing the degree of each neighbor by one for each of $N(v, \mathsf{dl}(v)) < (1+\delta)^{gn(\mathsf{dl}(v))}$ results in the total increase in Φ.

Then, in total, the potential decrease is at least

$$\left(\sum_{i=\mathsf{dl}(v)}^{\ell(v)-1} (\lambda+3)(1+\delta)^{gn(i)}\right) - (\lambda+2)(\ell(v) - \mathsf{dl}(v))(1+\delta)^{gn(\mathsf{dl}(v))}$$
$$\geq (\ell(v) - \mathsf{dl}(v))(1+\delta)^{gn(\mathsf{dl}(v))}$$

which is enough to pay for the at most $(1+\delta)^{gn(\mathsf{dl}(v))}$ edge flips as well as the $O(\ell(v) - \mathsf{dl}(v))$ work for computing the desire-level. The total number of edge flips is upper bounded by $N(v, \mathsf{dl}(v))$. Since we moved v to $\mathsf{dl}(v)$ and not $\mathsf{dl}(v)+1$, we know that v satisfies Invariant 2 at $\mathsf{dl}(v)$ and not at $\mathsf{dl}(v)+1$. Then, this means that $|N(v, \mathsf{dl}(v)))| < (1+\delta)^{gn(\mathsf{dl}(v))}$. Hence, our number of edge flips is also bounded by $(1+\delta)^{gn(\mathsf{dl}(v))}$.

A vertex v moves from level i to level $i+1$ due to Algorithm 21 In order for Algorithm 21 to move a vertex from level i to $i+1$, it must have violated Invariant 1 and that $\mathsf{up}(v) > (2+3/\lambda)(1+\delta)^{gn(i)}$ before the move. Before and after the move, $\Phi(v) = 0$, since in these cases $\mathsf{up}^*(v) > (2+3/\lambda)(1+\delta)^{gn(i-1)}$ and $\mathsf{up}^*(v) > (2+3/\lambda)(1+\delta)^{gn(i)}$, respectively. Thus, $\Phi(v)$ does not change in value. Furthermore, the $\Phi(w)$ of its neighbors w cannot increase. Then, this leaves us with the potential change in $\Psi(v, w)$.

Let Z_i be the set of neighbors that v has to iterate through within its data structures if v goes up a level. The potential decrease for every neighbor of v on $i = \ell(v)$ is 1. The potential decrease for every neighbor on level $i+1$ is 1. Finally, the potential decrease for every neighbor in levels $> \mathsf{dl}(v)$ is 2. Then, the potential decrease for every neighbor in Z_i is at least 1 and is enough to pay for the $O(|Z_i|)$ cost of iterating and moving the neighbors of v in its data structures.

Parallel Amortized Work The last part of the proof that needs to be shown is that any set of parallel level data structure operations that is undertaken by Algorithm 21 or Algorithm 22 has a sequential set of operations of the form detailed above (i.e., moving v to $\mathsf{dl}(v)$ or moving v from level i to $i+1$) that consists of the same or strictly larger set of operations.

Lemma 6.7.2. *For any set of operations performed in parallel by Algorithm 21 or Algorithm 22, there exists an identical set of sequential operations to the set of parallel operations.*

Proof. In Algorithm 21, the parallel set of operations consists of moving all vertices that violate Invariant 1 in the same level i up to level $i+1$. Again, suppose we choose an arbitrary order to move the vertices in level i to level $i+1$. Given two neighbors in the order v and w, if v moves to level $i+1$, the up-degree of w still includes v; since the up-degree of any vertex w is not affected by the previous vertices that moved to level $i+1$, w moves to $i+1$ on its turn. This order provides a sequential set of operations that is equivalent to the parallel set of operations.

In Algorithm 22, the parallel set of operations consists of moving a set of vertices down from arbitrary levels to the same level i. We show that there exists an identical set of sequential operations to the parallel operations. First, any vertex whose $\mathsf{dl}(v) = i$ considered all vertices in levels $\geq i-1$ in its calculation of $\mathsf{dl}(v)$. Thus, any other vertex w moving from a level $j > i$ to level i is included in calculating the desire-level of vertex v. Suppose we pick an arbitrary order to move the vertices that have $\mathsf{dl}(v) = i$ to level i. Then, the desire-level of any vertex w whose $\mathsf{dl}(w) = i$ does not change after v is moved to level i. Hence, when it is w's turn in the order, w moves to level i. This arbitrary order is a sequential set of operations that is identical to the parallel set of operations.

□

Lemma 6.7.3. *For a batch of $\mathcal{B} < m$ updates, Algorithm 20 requires $O(\mathcal{B}\log^2 m)$ amortized work with high probability. The required space is $O(n\log^2 m + m)$ using the randomized data structures.*

Proof. Our potential argument handles the cost of moving neighbors of a vertex v between different levels. Namely, our potential argument shows that such costs of updating

neighbor lists of nodes require $O(\log^2 m)$ amortized work per edge update to the structure.

Then, it remains to calculate the amount of work of Algorithm 23. We can obtain the size of each neighbor list in $O(1)$ work and depth. If we show that the work of running Algorithm 23 is asymptotically bounded by to the work of calculating the set of neighbor vertices that need to be moved between neighbor lists for a vertex, then we can also charge this work to the potential. To compute the first lower bound on $\mathsf{dl}(v)$, we maintain a cumulative sum of the total number of neighbors for each vertex at or below the current level $\ell(v)$. Then, we sequentially double the number of elements we use to compute the next level. We use $O((\ell - \mathsf{dl}(v)))$ work to compute $\mathsf{dl}(v)$.

Finally, we also bound the work of the final binary search. Let R be the size of the range of values in which we perform our binary search. The size of the number of possible levels becomes smaller as we decrease our range of values to search. Whenever we go right in the binary search, we perform $R/2$ work. Whenever we go left in the binary search, we also perform at most $R/2$ work. Thus, the total amount of work we perform while doing the binary search is $O(R)$. And by the argument above, the amount of work is $O\left(|Z_{\mathsf{dl}(v)} \setminus Z_{\ell(v)}|\right)$.

The total work of Algorithm 23 is $O(|Z_{\mathsf{dl}(v)} \setminus Z_{\ell(v)}| + (\ell - \mathsf{dl}(v)))$ which we can successfully charge to the potential. We conclude that the amount of work per update is $O(\log^2 m)$. □

6.7.2 Overall Work and Depth Bounds

Our deterministic and space-efficient structures also give the following corollary using our above work-bound arguments.

Corollary 6.7.4. *There exists a set of data structures where Algorithm 20 requires $O(\log^2 m)$ amortized work per update, deterministically, using $O(m+n)$ space.*

Using Corollary 6.3.12 and Lemma 6.7.3, we obtain Lemma 6.7.1. Similarly, combining Lemma 6.3.11 and Corollary 6.7.4 and Corollary 6.3.13 and Corollary 6.7.4, we obtain the following two corollaries.

Corollary 6.7.5. *For a batch of $\mathcal{B} < m$ updates, Algorithm 20 returns a PLDS that maintains Invariant 1 and Invariant 2 in $O(\mathcal{B}\log^2 m)$ amortized work and $O(\log^3 m)$ worst-case depth, using $O(n\log^2 m + m)$ space.*

Corollary 6.7.6. *For a batch of $\mathcal{B} < m$ updates, Algorithm 20 returns a PLDS that maintains Invariant 1 and Invariant 2 in $O(\mathcal{B}\log^2 m)$ amortized work and $O(\log^4 m)$ worst-case depth, using $O(n+m)$ space.*

6.8 Handling Vertex Insertions and Deletions

We can handle vertex insertions and deletions by inserting vertices that have zero degree and considering deletions of vertices to be a batch of edge deletions of all edges adjacent

to the deleted vertex. When we insert a vertex with zero degree, it automatically gets added to level 0 and remains in level 0 until edges incident to the vertex are inserted. For a vertex deletion, we add all edges incident to the deleted vertex to a batch of edge deletions. Note, first, that all vertices which have 0 degree will remain in level 0. Thus, there are at most $O(m)$ vertices which have non-zero degree.

Because we have $O(\log^2 m)$ levels in our data structure, we rebuild the data structure once we have made $m/2$ edge updates (including edge updates from edges incident to deleted vertices). Rebuilding the data structure requires $O(m \log^2 n)$ total work which we can amortize to the $m/2$ edge updates to obtain $O(\log^2 n)$ amortized work *whp*. Running Algorithm 21 and Algorithm 22 on the entire set of $O(m)$ edges requires $O(\text{poly} \log n)$ depth *whp* depending on the specific set of data structures we use.

Lastly, in order to obtain a set of vertices which are re-numbered consecutively (in order to maintain our space bounds), we perform parallel integer sort or hashing.

6.9 Conclusion

We presented a work-efficient parallel batch-dynamic level data structure that gives a $(2+\varepsilon)$-approximate k-core decomposition. Our approach also gave a static approximate k-core algorithm that is to the best of our knowledge the first work-efficient algorithm with polylogarithmic depth for this problem. We studied shared-memory implementations of all of our algorithms and confirmed the practical applicability of our approach.

Chapter 7

Fully Dynamic $(\Delta+1)$-Vertex Coloring in $O(1)$ Update Time

This chapter presents results from the paper titled, "Fully Dynamic $(\Delta+1)$-Coloring in Constant Update Time" that the thesis author coauthored with Sayan Bhattacharya, Fabrizio Grandoni, Janardhan Kulkarni, and Shay Solomon [BGK+19]. This paper is currently under submission at the time of the writing of this thesis.

7.1 Introduction

Vertex coloring is one of the most fundamental and most well-studied graph problems. Consider any integral parameter $\lambda > 0$, an undirected graph $G = (V, E)$ with n nodes and m edges, and a *palette* $\mathcal{C} = \{1, \ldots, \lambda\}$ of λ colors. A *λ-coloring* in G is simply a function $\chi : V \rightarrow \mathcal{C}$ which assigns a color $\chi(v) \in \mathcal{C}$ to each vertex $v \in V$. Such a coloring is called *proper* iff no two neighboring nodes in G get the same color. The main goal is to compute a proper λ-coloring in the input graph $G = (V, E)$ such that λ is as small as possible. Unfortunately, this problem is NP-hard and even extremely hard to approximate: for any constant $\varepsilon > 0$, there is no polynomial-time approximation algorithm with approximation factor $n^{1-\varepsilon}$ unless $P \neq NP$ [FK98, KP06, Zuc07]. Vertex coloring remains NP-hard even in graphs of small chromatic number. In particular, recognizing 3-colorable graphs is a classic NP-hard problem [GJS74], and there is a deep line of work on coloring 3-colorable graphs in polynomial time with as few colors as possible [KT17].

Since the problem is computationally hard in general, much of the work on vertex coloring has focused on restricted families of graphs in different settings. In the static case, results have included settings such as bounded arboricity, with the classic paper by Matula and Beck [MB83] in the sequential setting, and more recent results in a variety of models including the streaming [BCG20], Congested Clique [BCG20, GS19], MPC [BCG20], the general graph query [BCG20], and Local models [BE10, BE11, BCG20, KP11]; other graph classes in which various vertex coloring problems were explored in the static case include bounded treewidth graphs [AP89, JS97, FGK11], bounded clique-width graphs [FGLS09, KR03], n-uniform hypergraphs [RSV15], bounded diameter [CdCMGI+21, MPS19, MPS21], and bounded degree graphs [DDJP19]. In the dynamic

case, past results on restricted families of graphs have focused on the class of bounded arboricity graphs [HNW20, SW20b].

Obviously it is always possible to $\Delta+1$ color a graph $G = (V, E)$ with maximum degree Δ, and a simple linear time algorithm can achieve this goal in the classical centralized offline setting. Achieving the same goal in different computational models is however not trivial. For example this problem was extensively studied in the distributed literature, both as a classical symmetry breaking problem, and due to its intimate connections with other fundamental distributed problems such as maximal matching and maximal independent set (MIS) [Lub86].

In this work we study the $(\Delta+1)$-coloring problem in the *fully dynamic* setting. Here, the input graph $G = (V, E)$ changes via a sequence of *updates*, where each update consists of the insertion or deletion of an edge in G. There is a *fixed* parameter $\Delta > 0$ such that the maximum degree in G remains upper bounded by Δ throughout this update sequence. We want to design an algorithm that is capable of maintaining a proper $(\Delta+1)$-coloring in such a dynamic graph G. The time taken by the algorithm to handle an update is called its *update time*. We say that an algorithm has an *amortized* update time of $O(\gamma)$ iff starting from an empty graph,[1] it takes at most $O(t \cdot \gamma)$ time to handle any sequence of t updates. Our goal is to ensure that the amortized update time of our algorithm is as small as possible. Our focus is on amortized time bounds, and we henceforth use the phrase "update time" to refer to "amortized update time".

There is a naive dynamic algorithm for this problem that has $O(\Delta)$ update time, which works as follows. Suppose that we are maintaining a proper $\Delta+1$-coloring $\chi : V \to \mathcal{C}$ in G. At this point, if an edge gets deleted from the graph, then we do nothing, as the coloring χ continues to remain proper provided that Δ remains fixed throughout the update sequence. Otherwise, if an edge uv gets inserted into G, then we first check if $\chi(u) = \chi(v)$. If not, we do nothing. If yes, then we pick an arbitrary endpoint $x \in \{u, v\}$, and by scanning all its neighbors we identify a *blank* color $c' \in \mathcal{C}$ for x (one that is not assigned to any of its neighbors). Such a blank color is guaranteed to exist, since x has at most Δ neighbors and the palette $\mathcal{C}$ consists of $\Delta+1$ colors. We now *recolor* the node x by assigning it the color c'. This results in a proper $(\Delta+1)$-coloring in the current graph. The time taken to implement this procedure is proportional to the degree of x, hence it is at most $O(\Delta)$. Bhattacharya et al. [BCHN18] significantly improved the $O(\Delta)$ time bound, obtaining the following result.

Theorem 7.1.1. *[BCHN18] There is a randomized dynamic algorithm that can maintain a $\Delta+1$-coloring in a dynamic graph with $O(\log \Delta)$ update time in expectation.*

A fundamental open question left by [BCHN18] is whether one can improve the update time to constant. A constant update time is the holy grail for any graph problem that admits a linear time (static) algorithm, and thus far was obtained only for a handful of problems. Building on the algorithm of Baswana et al. [BGS15], in FOCS'16 Solomon [Sol16] presented a randomized algorithm for maintaining a maximal matching with constant

[1] In this chapter, when we refer to an *empty graph*, we mean a graph that has n vertices and no edges. However, our algorithm can be modified to handle insertions and deletions of vertices with 0 degree. Such vertices are inserted into and deleted from the bottommost level of our structure.

update time. Remarkably, Solomon's algorithm was the first to achieve a constant update time *for any nontrivial problem in general dynamic graphs*. A maximal matching provides a 2-approximation for both the maximum matching and the minimum vertex cover. An alternative deterministic primal-dual approach, introduced by Bhattacharya et al. [BHI15], maintains a fractional "almost-maximal" matching, and thus a $(2+\varepsilon)$-approximation for the maximum matching size, and also an integral $(2+\varepsilon)$-approximation for the minimum vertex cover. This line of work culminates in the SODA'19 paper of Bhattacharya and Kulkarni [BK19], which achieves an update time $O(1/\varepsilon^2)$. There is an intimate connection between the two approaches, which is hard to formalize, but at a very high level, the randomness encapsulated within the maximal matching algorithms of [BGS15, Sol16] naturally correspond to the fractional deterministic almost-maximal solutions of [BHI15, BK19, GKKP17, BCH17].

Our main result is summarized in Theorem 7.1.2 below. We design a randomized algorithm for $(\Delta+1)$-coloring with $O(1)$ update time in expectation and with high probability (for a sufficiently long update sequence). This constitutes a *dramatic* improvement over the update time of [BCHN18] as stated in Theorem 7.1.1. As with most existing randomized dynamic algorithms, both Theorems 7.1.1 and 7.1.2 hold only when the adversary deciding the next update is *oblivious* to the past random choices made by the algorithm. We emphasize that, unlike several related results in the literature–including the previous result for $(\Delta+1)$-coloring [BCHN18], our bound holds also with high probability.

Theorem 7.1.2. *There is a randomized algorithm for maintaining a $(\Delta+1)$-coloring in a dynamic graph that, given any sequence of t updates, takes total time $O(n\log n + n\Delta + t)$ in expectation and with high probability. The space usage is $O(n\Delta + m)$, where m is the maximum number of edges present at any time. For $t = \Omega(n\log n + n\Delta)$, we obtain $O(1)$ amortized update time in expectation and with high probability.*

To provide a very quick explanation of our bound: the factor of $O(n\Delta)$ comes from our data structure for maintaining free colors in Lemma 7.2.1, and the factor of $O(n\log n)$ comes from Lemma 7.3.13 and Lemma 7.3.14 in our analysis.

Previous work: We start with a high level overview of the dynamic algorithm in [BCHN18]. They maintain a *hierarchical partition* of the node-set V into $O(\log\Delta)$ levels. Let $\ell(v) \in \{1,\ldots,\log\Delta\}$ denote the *level* of a node $v \in V$. For every edge $(u,v) \in E$, say that u is a *same-level-neighbor*, *down-neighbor* and *up-neighbor* of v respectively iff $\ell(u) = \ell(v)$, $\ell(u) < \ell(v)$ and $\ell(u) \geq \ell(v)$. The following invariant is maintained.

Invariant 3. *Each node $v \in V$ has $\Omega(2^{\ell(v)})$ down-neighbors and $O(2^{\ell(v)})$ same-level neighbors.*

In order to ensure that Invariant 3 holds, the nodes need to keep changing their levels as the input graph keeps getting updated via a sequence of edge insertions/deletions. It is important to note that the subroutine in charge of maintaining this invariant is *deterministic* and has $O(\log\Delta)$ amortized update time.

The algorithm in [BCHN18] uses a separate (randomized) subroutine to maintain a proper $(\Delta+1)$-coloring in the input graph, on top of the hierarchical partition. To appreciate the main intuition behind this *recoloring subroutine*, consider the insertion of an

edge (u, v) at some time-step τ, and suppose that both u and v had the same color just before this insertion. Pick any arbitrary endpoint $x \in \{u, v\}$. The algorithm picks a new color for x as follows. Let $\mathcal{C}_x \subseteq \mathcal{C}$ denote the subset of colors that satisfy the following property at time-step τ: A color $c \in \mathcal{C}$ belongs to $\mathcal{C}_x$ iff (a) no up-neighbor of x has color c, and (b) at most one down-neighbor of x has color c. Since the node x has at most Δ neighbors and the palette $\mathcal{C}$ consists of $\Delta + 1$ colors, a simple counting argument (see the proof of Lemma 7.3.1) along with Invariant 3 implies that the size of the set $\mathcal{C}_x$ is at least $\Omega(2^{\ell(x)})$. Furthermore, using appropriate data structures, the set $\mathcal{C}_x$ can be computed in time proportional to the number of down-neighbors and same-level neighbors of x, which is at most $O(2^{\ell(x)})$ by Invariant 3. The algorithm picks a color c' uniformly at random from the set $\mathcal{C}_x$, and then recolors x by assigning it the color c'. By definition of the set $\mathcal{C}_x$, at most one neighbor (say, y) of x has the color c', and, furthermore, if such a neighbor y exists then $\ell(y) < \ell(x)$. If the down-neighbor y exists, then we recursively recolor y in the same manner. Note that this entire procedure leads to a *chain* of recolorings. However, the levels of the nodes involved in these successive recolorings form a strictly decreasing sequence. Thus, the total time taken by the subroutine to handle the edge insertion is at most $\sum_{\ell=1}^{\ell(x)} O(2^{\ell}) = O(2^{\ell(x)})$.

Now comes the most crucial observation. Note that each time the algorithm recolors a node x, it picks a new color uniformly at random from a set of size $\Omega(2^{\ell(x)})$. Thus, intuitively, if the adversary deciding the update sequence is oblivious to the random choices made by the algorithm, then in expectation at least $\Omega(2^{\ell(x)}/2) = \Omega(2^{\ell(x)})$ edge insertions incident on x should take place before we encounter a *bad event* (where the other endpoint of the edge being inserted has the same color as x). The discussion in the preceding paragraph implies that we need $O(2^{\ell(x)})$ time to handle the bad event. Thus, overall we get an amortized update time of $O(1)$ in expectation.

Our contribution: To summarize, the algorithm in [BCHN18] has two components – (1) a deterministic subroutine for maintaining the hierarchical partition which takes $O(\log \Delta)$ amortized update time, and (2) a randomized subroutine for maintaining a proper $(\Delta+1)$-coloring which takes $O(1)$ amortized update time. The analysis of the amortized update time of the first subroutine is done via an intricate potential function, and it is not clear if it is possible to improve the update time of this subroutine to $O(1)$.

To get an overall update time of $O(1)$, our algorithm merges these two components together in a very careful manner. Our starting point is to build on the high-level strategy used for maximal matching in [Sol16]. Suppose that we decide to recolor a node x during the course of our algorithm (either due to the insertion of an edge incident on it, or because one of its up-neighbors took up the same color as x while recoloring itself). Let $\ell(x)$ be the current level of x. We first check if the number of down-neighbors of x is $\Omega(3^{\ell(x)})$. If the answer is yes, then we move up the node x to the minimum level $\ell'(x) > \ell(x)$ where the number of its down-neighbors becomes $\Theta(3^{\ell'(x)})$, following which we recolor the node x in the same manner as in [BCHN18]. In contrast, if the answer is no, then we find a new color for x that does not conflict with any of its neighbors and move the node x down to the smallest possible level. Thus, in our algorithm, the hierarchical partition itself is determined by the random choices made by the nodes while they recolor themselves. This makes the analysis of our algorithm significantly more challenging than

that of [BCHN18], as Invariant 3 is no longer satisfied all the time.

Furthermore, our analysis is more challenging than that of [Sol16] in one central aspect, which we discuss next. As mentioned, due to the oblivious adversary assumption, at least $\Omega(3^{\ell(x)}/2) = \Omega(3^{\ell(x)})$ edge insertions incident on node x are expected to occur before a *bad event* is encountered, i.e., when the other endpoint x' of the edge being inserted has the same color as x. Importantly, the color of that other endpoint x' at the time of that edge insertion (x, x') might have been chosen *after* the color of x was chosen, which may create dependencies between the (random variables corresponding to the) colors of x and x'. A similar reasoning was applied to the maximal matching problem [BGS15, Sol16]; if the matched edge incident on a node x, denoted by (x, x'), was sampled uniformly at random among $\Omega(3^{\ell(x)})$ edges incident on x, then $\Omega(3^{\ell(x)})$ edge deletions incident on x are expected to occur (among the sampled ones) before a bad event is encountered, where the bad event here is that the deleted edge on x is its matched edge (x, x'). There is an inherent difference, however, between these two bad events. In the maximal matching problem, the time step of the bad event is fully determined by the adversarial updates that occur after the creation of that matched edge (under the oblivious adversary assumption), and in particular it is independent of future random choices made by the algorithm. On the other hand, in our coloring problem the time step of the bad event may depend on random choices made by the algorithm after the random color of x has been chosen, due to nodes that become neighbors of x in the future and whose colors are chosen after x's color has been chosen. Thus, we must cope with subtle conditional probability issues that did not effect the analysis in [BGS15, Sol16]. Note that in our analysis, the value of Δ is the maximum Δ over the course of the edge updates. The main difficulty with getting our running time for Δ that is the maximum degree of the current graph is that a single update may decrease the maximum degree of the current graph by 1; and so every vertex which is colored with the $(\Delta+1)$-th color needs to be recolored in our algorithm and recoloring all such nodes may be expensive in total. Specifically, we think it may be possible to modify our algorithm to obtain a $(\deg(v)+1)$-coloring where $\deg(v)$ is the current degree of vertex v, in which case, the returned coloring will trivially be a $(\Delta_{current}+1)$-coloring. So far no work has obtained such a dynamic $(\deg(v)+1)$-coloring in $O(1)$ amortized running time; it is an interesting open question whether there exists a dynamic algorithm that maintains a $(\deg(v)+1)$-coloring for all vertices $v \in V$ in the input graph in $O(1)$ amortized time per update. Furthermore, it is an open question to obtain $(\Delta_{current}+1)$-coloring *with high probability* in $O(1)$ amortized running time.

Independent work: Independently of our work, Henzinger and Peng [HP19] have obtained an algorithm for $(\Delta+1)$-vertex coloring with $O(1)$ *expected amortized update time*. Note that our work achieves $(\Delta+1)$-vertex coloring with $O(1)$ amortized update time not only in expectation, but also with *with high probability*.

7.2 Our Algorithm

Consider a graph $G = (V, E)$ with $|V| = n$ nodes that is changing via a sequence of *updates* (edge insertions and deletions). The graph initially starts off as empty (containing n vertices and no edges). Let $\Delta > 0$ be a fixed integer such that the maximum degree of any

node in the dynamic graph G is always upper bounded by Δ. In other words, Δ represents the maximum degree that any vertex can take throughout the update sequence. Let $\mathcal{C} = \{1, \ldots, \Delta + 1\}$ denote a palette of $\Delta + 1$ colors. Our algorithm will maintain a proper $\Delta + 1$-coloring $\chi : V \to \mathcal{C}$ in the dynamic graph G.

A hierarchical partition of the node-set V: Fix a parameter $L = \lceil \log_3(n-1) \rceil - 1$. Our dynamic algorithm will maintain a hierarchical partition of the node-set V into $L+2$ distinct *levels* $\{-1, 0, \ldots, L\}$. We let $\ell(v) \in \{-1, 0, \ldots, L\}$ denote the level of a given node $v \in V$. The levels of the nodes will vary over time. Consider any edge $(u, v) \in E$ in the dynamic graph G at any given point in time: We say that u is an *up-neighbor* of v iff $\ell(u) \geq \ell(v)$, and a *down-neighbor* of v iff $\ell(u) < \ell(v)$.

Notations: Fix any node $v \in V$. Let $\mathcal{N}_v = \{u \in V : uv \in E\}$ denote the set of neighbors of v. Furthermore, let $\mathcal{C}_v^+ = \{c \in \mathcal{C} : c = \chi(u) \text{ for some } u \in \mathcal{N}_v \text{ with } \ell(u) \geq \ell(v)\}$ denote the set of colors assigned to the up-neighbors of v. We say that $c \in \mathcal{C}$ is a *blank* color for v iff no neighbor of v currently has the color c. Similarly, we say that $c \in \mathcal{C}$ is a *unique* color for v iff $c \notin \mathcal{C}_v^+$ *and* exactly one down-neighbor of v currently has the color c. C_v, as defined before,[2] then, consists of the blank and unique colors of v. Finally, for every $\ell \in \{-1, \ldots, L\}$, we let $\varphi_v(\ell) = |\{u \in \mathcal{N}_v : \ell(u) < \ell\}|$ denote the number of neighbors of v that currently lie below level ℓ. We are now ready to describe our dynamic algorithm.

Preprocessing: In the beginning, the input graph $G = (V, E)$ has an empty edge-set, i.e., $E = \varnothing$, and the algorithm starts with any arbitrary coloring $\chi : V \to \mathcal{C}$. All the relevant data structures are initialized. Subsequently, the algorithm handles the sequence of updates to the input graph in the following manner.

Handling the deletion of an edge: Suppose that an edge (u, v) gets deleted from G. Just before this deletion, the coloring $\chi : V \to \mathcal{C}$ maintained by the algorithm was proper (no two adjacent nodes had the same color). So the coloring χ continues to remain proper even after the deletion of the edge. So the deletion of an edge does *not* lead to any change in the levels of the nodes and the coloring maintained by the algorithm.

Handling the insertion of an edge: This procedure is described in Algorithm 25. Suppose that an edge (u, v) gets inserted into G. If, just before this insertion, we had $\chi(u) \neq \chi(v)$, then we call this insertion *conflict-less*, and otherwise *conflicting*. In case of a conflict-less insertion, the coloring χ continues to remain proper even after insertion of the edge. In this case, the insertion does *not* lead to any change in the levels of the nodes or the colors assigned to them. Otherwise, we pick the endpoint $x \in \{u, v\}$ that was most recently recolored and call the subroutine `recolor`(x). Such a choice of which vertex to recolor is crucial for our proof of the running time. This call to `recolor`(x) changes the color assigned to x and it might also change the level of x. However, there is a possibility that the new color assigned to x might be the same as the color of (at most one) down-neighbor of x. If this is the case, then we go to that neighbor of x it conflicts with, and keep repeating the same process until we end up with a proper coloring in G.

Procedure `recolor`(x) (see Algorithm 26), depending on whether $\varphi_x(\ell(x) + 1) < 3^{\ell(x)+2}$ or not, calls one of the procedures `det-color`(x) and `rand-color`(x).

[2] C_v denotes the subset of colors that satisfy the following property at timestep τ: A color $c \in \mathcal{C}$ belongs to $\mathcal{C}_v$ iff (a) no up-neighbor of v has color c and (b) at most one down-neighbor of v has color c.

`det-color`(x): This subroutine first picks a *blank* color (say) c for the node x. By definition no neighbor of x has the color c. It now recolors the node x by setting $\chi(x) \leftarrow c$. Finally, it moves the node x down to level -1, by setting $\ell(x) \leftarrow -1$. It then updates all the relevant data structures.

`rand-color`(x): This subroutine works as follows. Let $\ell = \ell(x)$ be the level of the node x when this subroutine is called. Step 04 in Algorithm 26 implies that at that time we have $\varphi_x(\ell+1) \geq 3^{\ell+2}$. It identifies the *minimum* level $\ell' > \ell$ where $\varphi_x(\ell'+1) < 3^{\ell'+2}$. Such a level ℓ' must exist because $\varphi_x(L+1) \leq (n-1) < 3^{L+2}$. The subroutine then moves the node x up to level ℓ', by setting $\ell(x) \leftarrow \ell'$, and updates all the relevant data structures. After this step, the subroutine computes the set $\mathcal{C}_x \subseteq \mathcal{C}$ of colors that are either blank or unique for x, next called *palette*. It picks a color $c \in \mathcal{C}_x$ uniformly at random, and recolors the node x with color c, by setting $\chi(x) \leftarrow c$. It then updates all the relevant data structures. If c happens to be a blank color for x, then no neighbor of x has the same color as c. In other words, this recoloring of x does *not* lead to any new conflict. Accordingly, in this case the subroutine returns NULL. Otherwise, if c happens to be an unique color for x, then by definition exactly one down-neighbor (and zero up-neighbors) of x also has color c. Let this down-neighbor be y. In other words, the recoloring of x creates a new *conflict* along the edge (x, y), and we need to recolor y to ensure a proper coloring. Thus, in this case the subroutine returns the node y.

Algorithm 25 **`handle-insertion`$((u,v))$**

```
1: if χ(u) = χ(v) then
2:     Let x ∈ {u, v} be the endpoint that was most recently recolored.
3:     while x ≠ NULL do
4:         x ← recolor(x).
```

Algorithm 26 **`recolor`(x)**

```
1: if φ_x(ℓ(x) + 1) < 3^{ℓ(x)+2} then
2:     det-color(x).
3: else
4:     y ← rand-color(x).
5:     return y.
```

It is not difficult to come up with suitable data structures for the algorithm described above such that the following result holds (more details in Section 7.4). Due to the complexity of the data structures from the need to maintain many low-level details, we defer the full details of such structures to Section 7.4 so as not to interrupt the core ideas and analysis in this section. However, we describe the main functionalities of the data structures here as is necessary in our main analysis.

Lemma 7.2.1. *There is an implementation of the above dynamic algorithm such that:*

1. *The preprocessing time is $O(\Delta n)$;*

2. *The space usage is $O(\Delta n + m)$, where m is the maximum number of edges present at any time;*
3. *Each deletion and conflict-less insertion takes $O(1)$ time deterministically;*
4. *Procedure* `det-color`(x) *takes time $O(3^{\ell(x)})$;*
5. *Procedure* `rand-color`(x) *takes time $O(3^{\ell'(x)})$ where $\ell'(x) > \ell(x)$ is the new level of node x at the end of the procedure.*

Proof. We now justify the five claims made in the statement of the lemma. A full, detailed implementation section of the data structures can be found in Section 7.4.1.

1. We initialize a *dynamic* array $\mathcal{U}_v$ for each vertex v that contains $O(\log n)$ entries (specifically, let L be the set of non-empty levels for v, the dynamic array contains $O(L)$ entries) that stores the up-neighbors of v. Each index of the array $\mathcal{U}_v$ contains a pointer to a linked list containing the up-neighbors of v at that level. For example, suppose that w is an up-neighbor of v at level i. Then, the i-th entry of $\mathcal{U}_v$ contains a linked list which contains w. We initialize another linked list $\mathcal{D}_v$ which contains the down-neighbors of v. Furthermore, we initialize two linked lists, $\mathcal{C}_v^+$ and $\mathcal{C}_v$. $\mathcal{C}_v^+$ contains exactly one copy of each color held by up-neighbors stored in $\mathcal{U}_v$. $\mathcal{C}\backslash\mathcal{C}_v^+$ then represents the colors of the down-neighbors stored in $\mathcal{D}_v$ that are not in $\mathcal{C}_v^+$ and the blank colors. The palette $\mathcal{C}_v$ containing the unique and blank colors of v can thus be computed from $\mathcal{C}\backslash\mathcal{C}_v^+$. Each $\mathcal{U}_v$ has size $O(1)$ initially when there are no edges (see Section 7.4 for details); $\mathcal{C}_v^+$ and $\mathcal{C}_v$ each has size $O(\Delta)$; and $\mathcal{D}_v$ is initially empty. Thus, the preprocessing time necessary to initialize these structures is $O(\Delta n)$. [3] More details on these structures can be found in Section 7.4.1.
2. The total space used by $\mathcal{D}_v$ for all v is $O(m)$ since $\mathcal{D}_v$ for vertex v stores at most the number of neighbors of v. All other data structures are initialized during preprocessing. Therefore, the space cost of the other data structures is $O(\Delta n)$. For each edge $e = (u, v)$, we maintain pointers that represent e between copies of u and v in the various data structures. Refer to Section 7.4.1 for a detailed description of the pointer management.
3. Deleting an edge uv requires deleting u from $\mathcal{U}_v$ and v from $\mathcal{D}_u$ (or vice versa). Inserting an edge uv requires inserting u into $\mathcal{U}_v$ and v into $\mathcal{D}_u$ (or vice versa). The colors for u and v can be moved in between $\mathcal{C}_v^+$ and $\mathcal{C}_v$ and between $\mathcal{C}_u^+$ and $\mathcal{C}_u$ via a set of pointers connecting the colors to the vertices. Refer to Fig. 7-1, Fig. 7-2 and Section 7.4.3 for a detailed description of these elementary operations. The total cost of these operations is then $O(1)$.
4. In this procedure `det-color`(v), the level of v is set deterministically to -1 and the color for v is chosen deterministically from its set of blank colors. In this case, all the data structures of vertices in levels $[-1, \ell(v)]$ (where $\ell(v)$ is the old level of v) must be updated with the new level of v. Due to the existence of pointers in between vertices and its neighbors in $\mathcal{D}_v$ and $\mathcal{U}_v$ in all the data structures, the cost of updating each individual neighbor is $O(1)$. To update the colors of the data structures requires following $O(1)$ pointers for each $w \in \mathcal{D}_v$. By the definition of `recolor`(v) (which calls `det-color`(v)), $\varphi_v(\ell(v) + 1) < 3^{\ell(v)+2}$. Hence, there are

[3] We require such a list of blank and unique colors for each vertex v in order to ensure our running time. It is an interesting open question whether we can remove the need for such lists of blank and unique colors.

$O(3^{\ell(v)})$ neighbors in levels $[\ell'(v), \ell(v)]$ to update and the cost of the procedure is $O(3^{\ell(v)})$. Refer to Section 7.4.1 and Fig. 7-5 for a complete description of this procedure.

5. Since $\ell'(v) > \ell(v)$, all the data structures of vertices in levels $[\ell(v), \ell'(v)]$ must be updated with the new level of v. The data structures can be updated in the same way as given above. Since $\ell'(v) > -1$ (it must be, by definition of the procedure), then, $\varphi_v(\ell'(v)+1) < 3^{\ell'(v)+2}$. Hence, this procedure takes $O(3^{\ell'(v)})$ time. Refer to Section 7.4.1 and Fig. 7-6 for a complete description of this procedure.

□

Again, a complete, detailed description of our data structures (with pseudocode) can be found in Section 7.4.

7.3 Analysis

We assume that our graph is empty at the end, meaning no edges exist on the graph after we perform all the updates in our update sequence. To ensure we end with an empty graph, we append additional edge deletions at the end of the original update sequence. Since we begin with an empty graph, this at most doubles the number of updates in our update sequence, but simplifies our analysis. Because edge deletions will never cause a recoloring of any vertex and the number of updates increases by at most a factor of 2, an amortized runtime bound of our algorithm with respect to the new update sequence will imply the same (up to a factor of 2) amortized bound with respect to the original sequence. We now show that our dynamic algorithm maintains the following invariant.

Invariant 4. *Consider a vertex v at level $\ell(v) \geq 0$ at a given point of time τ. When v was most recently recolored prior to τ, it chose a color uniformly at random from a palette of size at least $3^{\ell(v)+1}/2 + 1$. Furthermore, at that time v has at least $3^{\ell(v)+1}$ down-neighbors. For $\ell(v) = -1$, the color of v is set deterministically.*

Lemma 7.3.1. *Invariant 4 holds for all vertices at the beginning of each update.*

Proof. During the preprocessing step the color of each node v is set deterministically to some arbitrary color and $\ell(v) = -1$. Hence the claim holds initially. The color of v changes only due to a call to `recolor`(v). Let $\ell(v)$ and $\ell'(v)$ denote the level of v at the beginning and end of this call. If `recolor`(v) calls `det-color`(v), the color of v is set determinstically and $\ell'(v) = -1$. Hence the invariant holds. Otherwise, `recolor`(v) invokes `rand-color`(v). The latter procedure sets $\ell'(v)$ to the smallest value (larger than $\ell(v)$) such that $\varphi_v(\ell'(v)+1) < 3^{\ell'(v)+2}$. Recall that $\varphi_v(\ell)$ is the number of neighbors of v of level smaller than ℓ. This implies that the number of down-neighbors of v (at level $\ell'(v)$) are $\varphi := \varphi_v(\ell'(v)) \geq 3^{\ell'(v)+1}$.

It is then sufficient to argue that the palette used by `rand-color`(v) has size at least $\varphi/2+1$. We use exactly the same argument as in [BCHN18]. One has $|\mathcal{C} \setminus \mathcal{C}_v^+| = (\Delta+1) - |\mathcal{C}_v^+|$ since $\mathcal{C}_v^+$ contains exactly one copy of each color occupied by an up-neighbor of v. Since the degree of any vertex is at most Δ and the number of down-neighbors of v is φ, $|\mathcal{C}_v^+| \leq (\Delta - \varphi)$ since the number of colors occupied by the up-neighbors is at most the number

of up-neighbors. Then, $|\mathcal{C} \setminus \mathcal{C}_v^+| = (\Delta+1) - |\mathcal{C}_v^+| \geq (\Delta+1) - (\Delta - \varphi) = \varphi + 1$, where equality holds when v has degree Δ and up-neighbors of v all have distinct colors. For any color $c \in \mathcal{C}_v$ that is occupied by at most one down-neighbor of v, c is a blank or unique color. Let x be the number of down-neighbors of v that occupy a unique color. Then, the size of v's palette is at least $|\mathcal{C}_v| \geq |\mathcal{C} \setminus \mathcal{C}_v^+| - (\varphi - x)/2$; this is due to the fact that there can be at most $(\varphi - x)/2$ colors that are occupied by at least *two* down-neighbors of v. Then, $|\mathcal{C}_v| \geq |\mathcal{C} \setminus \mathcal{C}_v^+| - (\varphi - x)/2 \geq 1 + |\varphi| - (\varphi - x)/2 \geq \varphi/2 + 1$. □

Let t be the total number of updates. Excluding the preprocessing time, the running time of our algorithm is given by the cost of handling insertions and deletions. By Lemma 7.2.1-Item 3, the total cost of deletions and insertions that do not cause conflicts is $O(t)$. We thus focus on insertions that cause conflicts. Modulo $O(1)$ factors, the total cost of the latter insertions is bounded by the total cost of the calls to `recolor`$(\cdot)$.

Epochs: It remains to bound the total cost of the calls to `recolor`$(\cdot)$. To that aim, and inspired by [BGS15], we introduce the following notion of epochs. An epoch $\mathcal{E}$ is associated with a node $v = v(\mathcal{E})$, and consists of any maximal time interval in which v does not get recolored. So $\mathcal{E}$ starts with a call to `recolor`(v), and ends immediately before the next call to `recolor`(v) is executed. Note that even if v gets recolored with the same color that it occupied before, the epoch still ends and a new epoch begins. Observe that there are potentially multiple epochs associated with the same node v. Notice that by construction, during an epoch $\mathcal{E}$ the level and color of $v(\mathcal{E})$ does not change: we refer to that level and color as $\ell(\mathcal{E})$ and $\chi(\mathcal{E})$, resp. By $\mathcal{E}_\ell$ we denote the set of epochs at level ℓ. We define the *cost* $c(\mathcal{E})$ of an epoch $\mathcal{E}$ as the time spent by the call to `recolor`$(v(\mathcal{E}))$ that starts it, and then we charge the cost of every epoch $\mathcal{E}$ at level $\ell(\mathcal{E}) = -1$ to the previous epoch involving the same node $v(\mathcal{E})$.

Lemma 7.3.2. *Excluding the preprocessing time, the total running time of the dynamic algorithm is given by:* $O(\sum_\ell \sum_{\mathcal{E} \in \mathcal{E}_\ell} c(\mathcal{E})) = O(\sum_\ell |\mathcal{E}_\ell| \cdot 3^{\ell(\mathcal{E})})$.

Proof. By the above discussion and Lemma 7.2.1 (Item 4-Item 5), the cost of any epoch $\mathcal{E}$ is given by $c(\mathcal{E}) = O\left(3^{\ell(\mathcal{E})}\right)$. The claim follows. □

A classification of epochs: It will be convenient to classify epochs as follows. An epoch $\mathcal{E}$ is *final* if it is not concluded by a call to `recolor`$(v(\mathcal{E}))$. Thus, for a final epoch $\mathcal{E}$, $v(\mathcal{E})$ keeps the same color from the beginning of $\mathcal{E}$ till the end of all the updates. Otherwise $\mathcal{E}$ is *terminated*. A terminated epoch $\mathcal{E}$, $v = v(\mathcal{E})$, terminates for one of the following possible events: (1) some edge (u, v) is inserted, with $\chi(u) = \chi(v)$, hence leading to a call to `recolor`(v); (2) a call to `recolor`(w) for some up-neighbor w of v forces a call to `recolor`(v) (without the insertion of any edge incident to v). We call the epochs of the first and second type *original* and *induced*, resp. In the second case, we say that the epoch $\mathcal{E}'$ that starts with the recoloring of w *induces* $\mathcal{E}$.

Lemma 7.3.3. *The total cost of induced epochs is (deterministically) at most* $O(1)$ *times the total cost of original and final epochs.*

Proof. Let us construct a directed *epoch graph*, with node set the set of epochs, and a directed edge $(\mathcal{E}, \mathcal{E}')$ iff $\mathcal{E}'$ induced $\mathcal{E}$. Notice that, for any edge $(\mathcal{E}, \mathcal{E}')$ in the epoch graph,

$\ell(\mathcal{E}') > \ell(\mathcal{E})$. Observe also that this graph consists of a collection of disjoint, directed paths starting at original, induced, or final epochs and ending at original and final epochs. Let us charge the cost of each induced epoch $\mathcal{E}$ to the root $r(\mathcal{E})$ of the corresponding path in the epoch graph. All the cost is charged to original and final epochs, and the cost charged to one epoch $\mathcal{E}$ of the latter type is at most $\sum_{\ell<\ell(\mathcal{E})} O(3^\ell) = O(3^{\ell(\mathcal{E})})$. The claim follows. □

Lemma 7.3.4. *Given any sequence of t updates, the total cost of final epochs is (deterministically) $O(t)$.*

Proof. By Invariant 4, for any final epoch $\mathcal{E}$, $v = v(\mathcal{E})$ and $\ell = \ell(\mathcal{E})$, v must have at least $3^{\ell+1}$ down-neighbors at the beginning of $\mathcal{E}$. Since by assumption at the end of the process the graph is empty, there must be at least $3^{\ell+1}$ deletions with one endpoint being v during $\mathcal{E}$. By charging the $O(3^\ell)$ cost of $\mathcal{E}$ to the later deletions, and considering that each deletion is charged at most twice, we achieve a average cost per deletion in $O(1)$, hence a total cost in $O(t)$. □

A classification of levels: Recall that $\mathcal{E}_\ell$ denotes the set of epochs at level ℓ. We now classify the levels into 3 types, as defined below.

- A level ℓ is *induced-heavy* iff at least 1/2-fraction of the epochs in $\mathcal{E}_\ell$ are induced.
- A level ℓ is *final-heavy* iff (a) it is not induced-heavy *and* (b) at least 1/8-fraction of the epochs in $\mathcal{E}_\ell$ are final.
- A level ℓ is *original-heavy* iff it is neither induced-heavy nor final-heavy. Note that if a level ℓ is original-heavy, then $\geq 3/8$-fraction of the epochs in $\mathcal{E}_\ell$ are original.

Henceforth, we say that an epoch is induced-heavy, final-heavy and original-heavy if it respectively belongs to an induced-heavy, final-heavy and original-heavy level. We use the term "*cost of a level ℓ*" to refer to the total cost of all the epochs at level ℓ.

Lemma 7.3.5. *The total cost of all the induced-heavy levels is (deterministically) at most $O(1)$ times the total cost of all the original-heavy and final-heavy levels.*

Proof. We perform charging level by level, starting from the lowest level -1. Given a level ℓ, if it is either original-heavy or final-heavy then we do nothing. Otherwise, we match each epoch $\mathcal{E} \in \mathcal{E}_\ell$ that is either original or final with some distinct induced epoch $\mathcal{E}' \in \mathcal{E}_\ell$. We next charge the cost of $\mathcal{E}$ (as obtained from the proof of Lemma 7.3.3) to $\mathcal{E}'$. Finally, we charge the cost of $\mathcal{E}'$ to some original or final epoch $\mathcal{E}''$ at a higher level following the same scheme as in the proof of Lemma 7.3.3. At the end of this process, only original and final epochs at the original-heavy and final-heavy levels are charged. By an easy induction, when we start processing level ℓ the total charge on an original or final epoch at level ℓ coming from the lower levels is at most $\sum_{\ell'<\ell} O(3^{\ell'}) = O(3^{\ell-1})$. The lemma follows. □

Lemma 7.3.6. *Given any sequence of t updates, the total cost of all the final-heavy levels is (deterministically) at most $O(t)$.*

Proof. Note that at each final-heavy level at least 1/8-fraction of the epochs are final. Thus, the cost of the other epochs given by Lemma 7.3.2 can be charged to the final epochs in the layer. There is already $O(3^\ell)$ cost charged to the final epoch; thus, the additional cost of other epochs in the same level only increases this cost by an 8-factor. The proof now follows from Lemma 7.3.4. □

Corollary 7.3.7. *The total cost of the dynamic algorithm, excluding the preprocessing time and a term $O(t)$, is $O(1)$ times the total cost of the original-heavy levels.*

Proof. It follows from the above discussion and Lemma 7.3.5, Lemma 7.3.6. □

Bounding the Cost of the Original-Heavy Levels: It now remains to bound the total cost of the original-heavy levels. Recall that at each original-heavy level, at least 3/8-fraction of the epochs are original. Thus, using a simple charging scheme, the task of bounding the total cost of all the original-heavy levels reduces to bounding the total cost of all the original epochs in these levels. At this point, it is tempting to use the following argument. By Invariant 4, for each epoch $\mathcal{E}$, $\ell = \ell(\mathcal{E})$, the corresponding color $\chi(\mathcal{E})$ is chosen uniformly at random in a palette of size at least $3^{\ell(\mathcal{E})}/2+1$. Therefore, if $\mathcal{E}$ is original, we expect to see at least $\Omega(3^{\ell})$ edge insertions having $v(\mathcal{E})$ as one endpoint before one such insertion causes a conflict with $v(\mathcal{E})$. This would imply an $O(1)$ amortized cost per edge insertion. The problem with this argument is that, conditioning on an epoch $\mathcal{E}$ being original, modifies a posteriori the distribution of colors taken at the beginning of $\mathcal{E}$. For example, the choice of certain colors might make more likely that the considered epoch is induced rather than original. To circumvent this issue, we need a more sophisticated argument that exploits the fact that we are considering original epochs in original-heavy levels only.

We define the *duration* $dur(\mathcal{E})$ of an epoch $\mathcal{E}$, $v = v(\mathcal{E})$, as the number of edge insertions of type (u,v) that happen during $\mathcal{E}$, plus possibly the final insertion that causes the termination of $\mathcal{E}$ (if $\mathcal{E}$ is original). We also define a critical notion of *pseudo-duration* $psdur(\mathcal{E})$ of $\mathcal{E}$ as follows. Let $(v,u_1),\ldots,(v,u_q)$ be the subsequence of insertions of edges incident to v in the input sequence after the creation of $\mathcal{E}$. For each u_i in the sequence of updates, let $\chi(u_i)$ represent the color of u_i *right before the creation* of $\mathcal{E}$. Consider the sequence of colors $\chi(u_1),\ldots,\chi(u_q)$. Remove from this sequence all colors not in the palette C used by $\mathcal{E}$ to sample $\chi(\mathcal{E})$, and then leave only the first occurrence of each duplicated color. Let $\chi(1),\ldots,\chi(k)$ be the obtained subsequence of (distinct) colors. We assume that $\chi(1),\ldots,\chi(k)$ is a permutation of C (so that $k=|C|$), and otherwise extend it arbitrarily to enforce this property. We define $psdur(\mathcal{E})$ to be the index i such that $\chi(i)=\chi(\mathcal{E})$. In other words, $psdur(\mathcal{E})$ is equal to the number of distinct colors in $\chi(u_1),\ldots,\chi(u_j)$ that are also in C where $\chi(u_j)$ is the first occurrence of the color $\chi(\mathcal{E})$. Such a j exists since $j=i$ if the first occurrence of $\chi(\mathcal{E})$ is at index i.

Lemma 7.3.8. *For an original epoch $\mathcal{E}$, $psdur(\mathcal{E}) \le dur(\mathcal{E})$ deterministically.*

Proof. Let (v,u_i) be the edge insertion that causes the termination of $\mathcal{E}$, so that $dur(\mathcal{E}) = i$. Let $j \le i$ be the smallest index with $\chi(u_j)=\chi(u_i)$. Let C be the palette used by v to sample $\chi(u_i)$. The value of $psdur(\mathcal{E})$ equals the number of distinct colors in the set $\chi(u_1),\ldots,\chi(u_j)$ that are also in C. The latter number is clearly at most $j \le i$. (In this proof we crucially used the following property: If the insertion of an edge (x,y) creates a conflict, in the sense that both x and y have the same color, then our algorithm changes the color of the node $z \in \{x,y\}$ that was *most recently recolored*.) □

We say that an epoch $\mathcal{E}$ is *short* if $psdur(\mathcal{E}) \le \frac{1}{32e}3^{\ell(\mathcal{E})}$, and *long* otherwise. The following critical technical lemma upper bounds the probability that an epoch is short.

Lemma 7.3.9. *An epoch $\mathcal{E}$ is short with probability at most $\frac{1}{16e}$, independently from the random bits used by the algorithm other than the ones used to sample $\chi(\mathcal{E})$.*

Proof. Let C be the palette from which $v = v(\mathcal{E})$ took its color $c = \chi(\mathcal{E})$ uniformly at random. Let us condition on all the random bits used by the algorithm prior to the ones used to sample $\chi(\mathcal{E})$. Notice that this fixes C and the permutation $\chi(1),\ldots,\chi(|C|)$ of C used for the definition of $psdur(\mathcal{E})$ (see the paragraph before Lemma 7.3.8). The random bits used after the sampling of $\chi(\mathcal{E})$ clearly do not affect $psdur(\mathcal{E})$. The probability that $psdur(\mathcal{E}) = i$, i.e. $\chi(i) = \chi(\mathcal{E})$, is precisely $1/|C|$. The latter probability is deterministically at most $\frac{2}{3^{\ell(\mathcal{E})}}$ by Invariant 4. In particular, this upper bound holds independently from the random bits on which we conditioned earlier. The claim then follows since

$$\mathbb{P}[\mathcal{E} \text{ is short}] = \mathbb{P}\left[psdur(\mathcal{E}) \leq \frac{3^{\ell(\mathcal{E})}}{32e}\right] = \frac{3^{\ell(\mathcal{E})}}{32e} \cdot \frac{1}{|C|} \leq \frac{1}{16e}.$$

□

We next define some *bad* events, that happen with very small probability. Recall that $\mathcal{E}_\ell$ is the set of epochs at level ℓ. We define $\mathcal{E}_\ell^{short}$ (resp., $\mathcal{E}_\ell^{long}$) as the collection of all epochs in $\mathcal{E}_\ell$ that are short (resp., long).

Lemma 7.3.10. *Consider any $x \geq 0$, and let A_ℓ^x be the event that $|\mathcal{E}_\ell| > x$ and $|\mathcal{E}_\ell^{short}| \geq \frac{|\mathcal{E}_\ell|}{4}$. Then $\mathbb{P}(A_\ell^x) \leq \frac{4}{2^{x/2}}$.*

Proof. Fix two parameters q and j, with $j \geq q/4$, and consider any q level-ℓ epochs $\mathcal{E}^1,\ldots,\mathcal{E}^q$, ordered by their creation time. We argue that the probability that precisely j particular epochs $\mathcal{E}^{(1)},\ldots,\mathcal{E}^{(j)}$ among these q are short is at most $\left(\frac{1}{16e}\right)^j$. Let $B^{(i)}$ be the event that $\mathcal{E}^{(i)}$ is short, $1 \leq i \leq j$. By a simple induction and Lemma 7.3.9, we have that $\mathbb{P}(B^{(i)} \mid B^{(1)} \cap B^{(2)} \cap \ldots B^{(i-1)}) \leq \frac{1}{16e}$. Consequently, we get: $\mathbb{P}(B^{(1)} \cap B^{(2)} \cap \ldots \cap B^{(j)}) = \mathbb{P}(B^{(1)}) \cdot \mathbb{P}(B^{(2)} \mid B^{(1)}) \cdot \ldots \cdot \mathbb{P}(B^{(j)} \mid B^{(1)} \cap B^{(2)} \cap \ldots \cap B^{(j-1)}) \leq \left(\frac{1}{16e}\right)^j$.

There are $\binom{q}{j}$ choices for the subsequence $\mathcal{E}^{(1)}\ldots\mathcal{E}^{(j)}$. Thus, we get: $\mathbb{P}[|\mathcal{E}_\ell| = q \cap |\mathcal{E}_\ell^{short}| = j] \leq \binom{q}{j}\left(\frac{1}{16e}\right)^j$. Since $\binom{q}{j} \leq (\frac{eq}{j})^j \leq (4e)^j$, we have $\binom{q}{j}\left(\frac{1}{16e}\right)^j \leq \frac{1}{4^j}$. Hence, $\mathbb{P}(A_\ell^x) = \sum_{q>x}\sum_{j=q/4}^{q}\mathbb{P}[|\mathcal{E}_\ell| = q \cap |\mathcal{E}_\ell^{short}| = j] \leq \sum_{q>x}\sum_{j=q/4}^{q}\frac{1}{4^j} \leq \sum_{q>x}\frac{4}{3}\cdot\frac{1}{2^{q/2}} \leq \frac{4}{2^{x/2}}$. □

Corollary 7.3.11. *For a large enough constant $a > 0$ and $x = 2a\log_2 n$, let A denote the event that A_ℓ^x happens for* some *level ℓ. Then $\mathbb{P}(A) = O(\frac{\log n}{n^a})$.*

Proof. It follows from Lemma 7.3.10 and the union bound over all levels ℓ. □

Lemma 7.3.12. *Let g be the number of level-ℓ epochs with duration $\geq \delta$, and IN_ℓ be the set of input insertions of edges incident to vertices at level ℓ. Then $g \leq 2|IN_\ell|/\delta$.*

Proof. Observe that for the duration of the concerned epochs, we consider only insertions in IN_ℓ. Furthermore, each such insertion can influence the duration of at most 2 such epochs. The claim follows by the pigeon-hole principle. □

Let $c(\mathcal{E}_\ell) = O(3^\ell \cdot |\mathcal{E}_\ell|)$ be the total cost of the epochs in level ℓ. We next relate the occurrence of event $\neg A_\ell^x$ to the value of the random variable $c(\mathcal{E}_\ell)$ for original epochs.

Lemma 7.3.13. *If $\neg A_\ell^x$ occurs and level ℓ is original-heavy, then $c(\mathcal{E}_\ell) = O(|IN_\ell| + 3^\ell x)$.*

Proof. If $|\mathcal{E}_\ell| \leq x$, then we clearly have $c(\mathcal{E}_\ell) = O(3^\ell x)$. For the rest of the proof, we assume that $|\mathcal{E}_\ell^{short}| < \frac{|\mathcal{E}_\ell|}{4}$, or equivalently: $|\mathcal{E}_\ell^{long}| \geq \frac{3}{4} \cdot |\mathcal{E}_\ell|$. Let $\mathcal{E}_\ell^* \subseteq \mathcal{E}_\ell$ be the set of original epochs at level ℓ. As the level ℓ is original-heavy, we have: $|\mathcal{E}_\ell^*| \geq \frac{3}{8} \cdot |\mathcal{E}_\ell|$. Since $|\mathcal{E}_\ell^{long}| \geq \frac{3}{4} \cdot |\mathcal{E}_\ell|$, applying the pigeon-hole principle we infer that at least $q \geq \frac{|\mathcal{E}_\ell|}{8}$ level-ℓ epochs are original and long at the same time. Specifically, we get: $|\mathcal{E}_\ell^* \cap \mathcal{E}_\ell^{long}| \geq \frac{1}{8} \cdot |\mathcal{E}_\ell|$, or equivalently: $|\mathcal{E}_\ell| \leq 8 \cdot |\mathcal{E}_\ell^* \cap \mathcal{E}_\ell^{long}|$.

Any epoch $\mathcal{E} \in \mathcal{E}_\ell^* \cap \mathcal{E}_\ell^{long}$ has duration $dur(\mathcal{E}) \geq psdur(\mathcal{E}) \geq \frac{3^\ell}{32e}$ by Lemma 7.3.8 and the definition of long epochs. Hence by applying Lemma 7.3.12 with $\delta = \frac{3^\ell}{32e}$, we can conclude that the number of such epochs is at most $\frac{2|IN_\ell|}{3^\ell/(32e)} = \frac{64e|IN_\ell|}{3^\ell}$. Hence, we get: $|\mathcal{E}_\ell| \leq 8 \cdot |\mathcal{E}_\ell^* \cap \mathcal{E}_\ell^{long}| \leq 8 \cdot \frac{64e}{3^\ell} \cdot |IN_\ell|$. The lemma follows if we multiply both sides of this inequality by the $O(3^\ell)$ cost charged to each epoch in $\mathcal{E}_\ell$. □

We are now ready to bound the amortized update time of our dynamic algorithm. Recall that t denotes the total number of updates.

Lemma 7.3.14. *For any fixed sequence of t updates, with probability $1 - O(\log n/n^a)$, the total running time of our algorithm is $O(t + an \log n + \Delta n)$.*

Proof. By Lemma 7.2.1-Item 1, the preprocessing time is $O(\Delta n)$. The total cost of deletion and conflict-less insertions is $O(t)$, due to Lemma 7.2.1-Item 3. Let us condition on the event $\neg A$, which happens with probability $1 - O(\log n/n^a)$ by Corollary 7.3.11. Then the total cost of the original-heavy levels is $O(\sum_\ell (|IN_\ell| + 3^\ell a \log n)) = O(t + n \log n)$ by Lemma 7.3.13. The lemma now follows from Corollary 7.3.7. □

In order to prove that the amortized update time of our algorithm is $O(1)$ in expectation, we also need the following upper bound on its worst-case total running time.

Lemma 7.3.15. *Our algorithm's total runtime is deterministically $O(tn^2 + n \log n + \Delta n)$.*

Proof. The preprocessing time is $O(n \log n + \Delta n)$ and the total cost of deletions and conflict-less insertions is $O(t)$ deterministically. Each conflicting insertion starts a sequence of calls to `recolor`$(\cdot)$ involving some nodes $w_1, \ldots, w_q$. A given node w can appear multiple times in the latter sequence. However, the sequence ends when some node w^* is moved to level -1, and in all other cases the level of w is increased by at least one. This means that the total cost associated with node w is $O(\sum_\ell 3^\ell) = O(n)$. The lemma follows by summing over the n nodes and the $O(t)$ insertions. □

Hence we can conclude:

Lemma 7.3.16. *The total expected running time of our algorithm is $O(t + n \log n + \Delta n)$.*

Proof of Lemma 7.3.16. When the event $\neg A$ happens, the total cost of the algorithm is $O(t + \Delta n)$ by Lemma 7.3.14. If instead the event A happens, then the cost is $O(tn^2 + \Delta n)$ by Lemma 7.3.15. However the latter event happens with probability at most $O(\frac{\log n}{n^a})$ by Corollary 7.3.11. So this second case adds $o(t)$ to the total expected cost for $a > 2$. □

We now have all the ingredients to prove the main theorem of this chapter.

Proof of Theorem 7.1.2. Consider the dynamic algorithm described above. The space usage follows from Lemma 7.2.1-Item 2 and the update time from Lemmas 7.3.14 and 7.3.16. □

7.4 $(\Delta + 1)$-Coloring Update Data Structures

In this section, we give a full detailed description of the data structures used by our dynamic algorithm.

The update algorithm is applied following edge insertions and deletions to and from the graph. In this section, we provide a complete description of the update data structures and algorithm. The pseudocode of this algorithm can be found in Section 7.5. We begin with a description of the data structures and invariants that will be maintained by our algorithm. Throughout, we use the phrase *mutual pointers* between two elements a and b (i.e. specifically, we use the phrase "mutual pointers between a and b") to mean pointers from a to b and from b to a (hence the pointers are *mutual*).

7.4.1 Hierarchical Partitioning and Coloring Data Structures

Our algorithm maintains the following set of data structures which we divide into two groups: the data structures responsible for maintaining our hierarchical partitioning and the data structures used to maintain the set of colors associated with each vertex. Let $\mathcal{C}$ be the set of all $\Delta+1$ colors. The first group of data structures is a hierarchical partitioning of the vertices of the graph into different *levels* according to some procedures that maintain a set of invariants. A vertex at a level has some number of neighbors in other levels of the hierarchical partitioning structure. We refer to neighbors at the same or higher levels of the hierarchical partitioning structure as the *up-neighbors*. We refer to neighbors at lower levels of the hierarchical partitioning as the *down-neighbors*. Different data structures will be used to maintain the colors of the down-neighbors and the colors of the up-neighbors of a vertex. We can obtain the palette, $\mathcal{C}_v$, defined to consist of the blank and unique colors, by scanning through the list $\mathcal{C} \setminus \mathcal{C}_v^+$.

The second group of data structures deals with maintaining the colors of the vertices, inspired by the structures given in [BCHN18]. For the following data structures, we use logarithms in base 3 unless stated otherwise.

Let $\mathcal{C}$ be the set of all $\Delta + 1$ colors:

- **Hierarchical Partitioning:** We maintain the following data structures necessary for our hierarchical partitioning.
  1. For each vertex v:
     (a) $\mathcal{N}_v$: a linked list containing all neighbors of v.
     (b) $\mathcal{D}_v$: a linked list containing all down-neighbors of v.

(c) $\mathcal{U}_v$: a dynamic array where each index corresponds to a distinct level $\ell \in \{0, \ldots, \log_3(n-1)-1\}$. $\mathcal{U}_v[\ell]$ holds a level number, a pointer to the head of a non-empty doubly linked list containing all up-neighbors of v at level ℓ, and the size of the non-empty doubly linked list of neighbors. If this list is empty, then the corresponding pointer is not stored.

2. For any vertex v and any neighbor u in $\mathcal{D}_v$, let $u_{\mathcal{D}_v}$ represent the copy of $u \in \mathcal{D}_v$, $v_{\mathcal{U}_u[\ell(v)]}$ be the copy of $v \in \mathcal{U}_u[\ell(v)]$, $v_{\mathcal{N}_u}$ be the copy of $v \in \mathcal{N}_u$, and, finally $u_{\mathcal{N}_v}$ be the copy of $u \in \mathcal{N}_v$. We maintain the following pairs of pointers, where for each pair, there exists a pointer from the first element in the pair to the second and vice versa: $(u_{\mathcal{D}_v}, v_{\mathcal{U}_u[\ell(v)]})$, $(u_{\mathcal{D}_v}, v_{\mathcal{N}_u})$, $(u_{\mathcal{D}_v}, u_{\mathcal{N}_v})$, $(v_{\mathcal{U}_u[\ell(v)]}, v_{\mathcal{N}_u})$, $(v_{\mathcal{U}_u[\ell(v)]}, u_{\mathcal{N}_v})$, and $(v_{\mathcal{N}_u}, u_{\mathcal{N}_v})$. In other words, there exists two pointers (one forwards and one backwards) between every pair of elements in $\{u_{\mathcal{D}_v}, v_{\mathcal{U}_u[\ell(v)]}, v_{\mathcal{N}_u}, u_{\mathcal{N}_v}\}$. The set of pointers means that given an edge insertion or deletion, we are able to quickly access the endpoints of the edge in each data structure once we locate one copy of an endpoint in memory.
3. We define $\varphi_v(\ell)$ to be the number of neighbors of v with level strictly lower than ℓ. We calculate the appropriate values for $\varphi_v(\ell)$ as follows. For any level, $\ell' < \ell(v)$, we look through all neighbors stored in $\mathcal{D}_v$ (defined above) to calculate $\varphi_v(\ell')$. For levels $\ell' \geq \ell(v)$, we use the sizes of the linked lists in $\mathcal{U}_v$ (defined above) to calculate $\varphi_v(\ell')$.

- **Coloring:** We maintain the following data structures for our coloring procedures. These structures are similar to the structures used in [BCHN18].
  1. A static array χ of size $O(n)$ where $\chi[i]$ stores the *current* color of the i-th vertex.
  2. For each vertex v:
     (a) $\mathcal{C}_v^+$: a doubly linked list of exactly one copy of each color occupied by vertices in $\mathcal{U}_v$. Each color contains a counter $\mu_v^+(c)$ counting the number of vertices in $\mathcal{U}_v$ that is colored color c.
     (b) The counters $\mu_v^+(c)$ are stored in a static array of size $\Delta + 1$ where index i contains the number of vertices in $\mathcal{U}_v$ that is colored with color i.
     (c) $\mathcal{C}_v$: a doubly linked list of colors in $\mathcal{C} \backslash \mathcal{C}_v^+$ that are blank or unique.
     (d) A static array $\mathcal{P}_v$ of size $\Delta + 1$ containing mutual pointers (i.e. the pair of pointers from element a to element b and from element b to element a) to each color c in $\mathcal{C}_v$ or $\mathcal{C}_v^+$ and to each of two additional nodes representing each color in $\mathcal{C}$. Let i_c be the index of color c in $\mathcal{P}_v$. Suppose that $c \in \mathcal{C}_v$. Let p_c and p_c^+ be the two additional nodes representing c. Then $\mathcal{P}_v[i_c]$ contains pointers to $c \in \mathcal{C}_v$, p_c, and p_c^+. In addition, if $c \in \mathcal{C}_v$, then it has mutual pointers to p_c. If, instead, $c \in \mathcal{C}_v^+$, then it has mutual pointers to p_c^+ instead. In other words, p_c receives pointers from nodes in $\mathcal{C}_v$ (and has outgoing pointers to nodes in $\mathcal{C}_v$) and p_c^+ receives pointers from nodes in $\mathcal{C}_v^+$ (and has outgoing pointers to nodes in $\mathcal{C}_v^+$).

We define the set of *blank colors* for v to be colors in $\mathcal{C}_v$ which are not occupied by any vertex in $\mathcal{D}_v$. We define the set of *unique colors* of v to be colors in $\mathcal{C}_v$ which are occupied by at most one vertex in $\mathcal{D}_v$.

We now describe the pointers from the hierarchical partitioning structures to the col-

oring structures and vice versa.

- Each color c in $\mathcal{C}_v^+$ has a pointer to p_c^+ and vice versa.
- Each color c' in $\mathcal{C}_v$ has a pointer to $p_{c'}$ and vice versa.
- Each vertex $u \in \mathcal{U}_v$ contains mutual pointers to the node p_c^+ representing its color in $\mathcal{P}_v$ that it is currently colored with. The color c is also in $\mathcal{C}_v^+$ and has mutual pointers to p_c^+.
- Each vertex $u \in \mathcal{D}_v$ contains mutual pointers to p_c representing its corresponding color in $\mathcal{P}_v$. If its color is in $\mathcal{C}_v$, then mutual pointers also exist between p_c and c in $\mathcal{C}_v$.
- Each edge (u, v) contains two pointers, one to $u \in \mathcal{N}_v$ and one to $v \in \mathcal{N}_u$. u and v also contain pointers to edge (u, v).

Initial Data Structure Configuration, Time Cost, and Space Usage There exist no edges in the graph initially; thus all vertices can be colored the same color. Such an arbitrary starting color is chosen. Before any edge updates are made, we assume that all vertices are on level -1, colored with the arbitrary starting color. Thus, all colors are also initially in $\mathcal{C}_v$.

Before any edge insertions, the only structures that we initialize are an empty dynamic array $\mathcal{U}_v$ for each vertex v, the list of all colors $\mathcal{C}$, and χ. When the first edge that contains vertex v as an endpoint is inserted, we initialize $\mathcal{N}_v, \mathcal{D}_v, \mathcal{U}_v, \mathcal{C}_v^+, \mathcal{C}_v, \mu_v^+$ for v (as well as the associated pointers). The time for initializing these structures is $O(n\Delta)$ which means that the preprocessing time will result in $O(1)$ amortized time per update assuming $\Omega(n\Delta)$ updates.

The dynamic arrays in our data structure are implemented as follows. When the array contains no elements, we set the default size of the array to 8. Whenever the array is more than half full, we double the size of the array. Similarly, whenever the array is less than 1/4 full, we shrink the size of the array by half. Then, suppose we have an array which just shrank in size or doubled in size and the size of the array is L. We require at least either $L/4$ insertions or deletions to double or halve the size of the array again. Thus, the $O(L)$ cost of resizing the array can be amortized over the $L/4$ updates to $O(1)$ cost per update. Given that our algorithm is run on a graph which is initially empty, all dynamic arrays in our structure are initially empty and initialized to size 8.

We note a particular choice in constructing our data structures. In the case of $\mathcal{U}_v$, given our assumption of the number of updates, we can also implement $\mathcal{U}_v$ as a static array instead of a dynamic array. The maximum number of levels is bounded by $\log_3(n-1)+1$. Thus, if we instead implemented $\mathcal{U}_v$ as static arrays instead of dynamic arrays, the total space usage (and initialization cost) would be $O(n \log n)$, amortizing to $O(1)$ per update given $\Omega(n \log n)$ updates. There may be reasons to implement $\mathcal{U}_v$ as static arrays instead of dynamic dynamic arrays such as easier implementation of basic functions. However, we choose to use a dynamic array implementation for potential future work for the cases when the number of updates is $o(n \log n + n\Delta)$. The key property we can potentially take advantage of in using the dynamic array implementation is that the total space used (and the total time spent in initializing the data structure) is within a constant factor of the number of edges in the graph at any particular time.

Usefulness of the Pointers Pointers between the various data structures used for the hierarchical partitioning and for maintaining the coloring allows for us to quickly update the state following an edge insertion or deletion. For example, when an edge uv is inserted or deleted, we get pointers to $v \in \mathcal{N}_u$ and $u \in \mathcal{N}_v$, and through these pointers we delete all elements $u \in \mathcal{D}_v, v \in \mathcal{U}_u[\ell(v)], v \in \mathcal{N}_u, u \in \mathcal{N}_v$ and potentially move a color from $\mathcal{C}_u^+$ to $\mathcal{C}_u$. The exact procedure for handling edge deletions is described later.

7.4.2 Invariants

Our update algorithm and data structures maintain the following invariant (reproduced again here for readability).

Invariant 5. *The following hold for all vertices:*

1. *A vertex in level ℓ was last colored using a palette of size at least $(1/2)\,3^{\ell+1} + 1$. As a special case, a vertex in level -1 was last colored using a palette of size 1 (in other words, it was colored deterministically).*
2. *The level of a vertex remains unchanged until the vertex is recolored.*

7.4.3 Edge Update Algorithm

We now describe the update algorithm in detail. The data structures are initialized as described in Section 7.4.1. Then, edge updates are applied to the graph. Following an edge insertion or deletion, the procedure `handle-insertion`(u,v) or `handle-deletion`(u,v), respectively, is called. The descriptions of the insertion and deletion procedures are given below.

Procedures `handle-insertion`(u,v) and `handle-deletion`(u,v).
`handle-deletion`(u,v) is called on an edge deletion uv. This case would not result in any need to recolor any vertices since a conflict will never be created. Thus, we update the relevant data structures in the obvious way (by deleting all relevant entries in all relevant structures); details of this set of deletions are given in the pseudocode in Fig. 7-2.

Procedure `handle-insertion`(u,v) is called on an edge insertion `handle-insertion`(u,v). The pseudocode for this procedure is given in Fig. 7-1. If edge uv does not connect two vertices that are colored the same color (i.e. if the insertion is *conflict-less*), then we only need to update the relevant data structures with the inserted edge. Namely the vertices are added to the structures maintaining the neighbors of u and v. If u is on a higher level than v, then u is added to $\mathcal{U}_v$ and v is added to $\mathcal{D}_u$ (and vice versa). If u and v are on the same level, then u is added to $\mathcal{U}_v$ and v is added to $\mathcal{U}_u$. Furthermore, the colors that are associated with the vertices are moved in between the lists $\mathcal{C}_v^+$ and $\mathcal{C}_v$ as necessary. See the pseudocode in Fig. 7-1 for exact details of these straightforward data structure updates.

In the case that edge uv connects two vertices of the same color (i.e. if the insertion is *conflicting*), we need to recolor at least one of these two vertices. We arbitrarily recolor *one* of the vertices u or v using procedure `recolor` (i.e. `recolor`(u) as given in the pseudocode in Fig. 7-4). Procedure `recolor` is the crux of the update algorithm and is

described next.

Whenever a conflict is created following an edge insertion uv, procedure `recolor`(u) is called on one of the two endpoints. This procedure is described below.

Procedure `recolor`(v). The pseudocode for this procedure can be found in Fig. 7-4. The procedure `recolor`(v) makes use of the level of v as well as the number of its down-neighbors to either choose a blank color deterministically to recolor v or to determine the palette from which to select a random color to recolor v. Recall that all vertices start in level -1 before any edges are inserted into the graph.

The procedure `recolor`(v) considers two cases:

- *Case 1:* $\varphi_v(\ell(v)+1) < 3^{\ell(v)+2}$. In other words, the first case is when the number of down-neighbors and vertices on the same level as v is not much greater than $3^{\ell(v)+1}$. We show in the analysis that in this case, we can find the colors of all the neighbors in $\mathcal{D}_v$ and pick a color in $\mathcal{C}_v$ that does not conflict with any such neighbors (or the color that it currently has). Thus, we deterministically choose a blank color to recolor v, creating no further conflicts. The procedure to choose a blank color for v, `det-color`(v), is described in the following.
- *Case 2:* $\varphi(\ell(v)+1) \geq 3^{\ell(v)+2}$. In this case, the number of down-neighbors and vertices on the same level as v is at least $3^{\ell(v)+2}$ and it will be too expensive to look for a blank color since we need to look at all neighbors in $\mathcal{D}_v$ to determine such a color and the size of $\mathcal{D}_v$ could be very large. Thus, we need to pick a random color from $\mathcal{C}_v$ to recolor v by running Procedure `rand-color`(v) as described below.

Procedures `det-color`(v) and `rand-color`(v). When called, the procedure `det-color`(v) starts by scanning the list $\mathcal{C}_v$ to find at least one blank color that we can use to color v. By the definition of $(\Delta+1)$-coloring, there must exist at least one blank color with which we can use to color v. We can deterministically find a blank color in the following way. The elements in $\mathcal{C}_v$ are stored in a doubly linked list. We start with the first element at the front of the list and scan through the list until we reach an element that does not have a pointer to a vertex in $\mathcal{D}_v$. We can determine whether a color $c \in \mathcal{C}_v$ has a pointer to a vertex in $\mathcal{D}_v$ by following the pointer from c to p_c. From p_c, we can then determine whether any vertices in $\mathcal{D}_v$ are colored with c.

Let this first blank color be c_b. We assign color c_b to v, update $\chi(i_v)$ to indicate that the color of v is c_b, and update the lists $\mathcal{C}_w^+$ and/or $\mathcal{C}_w$ of all $w \in \mathcal{D}_v$. To update all $\mathcal{C}_w^+$ and $\mathcal{C}_w$, we follow the following set of pointers:

1. From $w \in \mathcal{D}_v$, follow pointers to reach $w \in \mathcal{N}_v$.
2. From $w \in \mathcal{N}_v$, follow pointers to reach $v \in \mathcal{N}_w$.
3. From $v \in \mathcal{N}_w$, follow pointers to reach $v \in \mathcal{U}_w[\ell(v)]$.
4. Let c be the previous color of v as recorded in $\mathcal{C}_w^+$. From $v \in \mathcal{U}_w[\ell(v)]$, follow pointers to reach $c \in \mathcal{C}_w^+$.
5. Decrement $\mu_w^+(c)$ by 1. Delete mutual pointers between v and p_c^+. If now $\mu_w^+(c) = 0$, remove c from $\mathcal{C}_w^+$, append c to the end of $\mathcal{C}_w$, delete mutual pointers between c and p_c^+, and add mutual pointers between c and p_c.

6. Use $\mathcal{P}_w$ to find c_b in either $\mathcal{C}_w^+$ or $\mathcal{C}_w$. If $c_b \in \mathcal{C}_w^+$, increment $\mu_w^+(c_b)$ by 1. Otherwise, if $c_b \in \mathcal{C}_w$, remove c_b from $\mathcal{C}_w$, append c_b to the end of $\mathcal{C}_w^+$, increment $\mu_w^+(c_b)$ by 1, delete mutual pointers between c_b and p_{c_b}, and create mutual pointers between c_b and $p_{c_b}^+$. Create mutual pointers between $v \in \mathcal{U}_w[\ell(v)]$ and $p_{c_b}^+$.

After the above is done in terms of recoloring the vertex v, `set-level`$(v,-1)$ is called to bring the level of v down to -1. The description of `set-level`$(v,-1)$ is given in the following. See the pseudocode for `det-color`(v) in Fig. 7-5 for concrete details of this procedure.

The procedure `rand-color`(v) employs a *level-rising mechanism.* We mentioned before the concept of partitioning vertices into levels. Each level bounds the down-neighbbors of the vertices at that level, providing both an upper and lower bound on the number of down-neighbors of the vertex. Because there are at most $\log_3(n-1)$ levels, the number of vertices in each level is thus exponentially increasing. The procedure `rand-color`(v) takes advantage of this bound on the number of down-neighbors of the vertex v to find a level to recolor v with a color randomly chosen from its $\mathcal{C}_v$. Specifically, `rand-color`(v) recolors v at some level ℓ^* higher than $\ell(v)$, with a random blank or unique color occupied by vertices of levels *strictly lower* than ℓ^*. At level ℓ^*, it attempts to select a color c within time $O(3^{\ell^*})$; this can only occur if $|\mathcal{D}_v| = O(3^{\ell^*})$. Upon failure, it calls itself recursively to color v at yet a higher level. Again, `set-level`(v,ℓ^*) is called every time v moves to a high level.

Procedure `set-level`(v,ℓ). Procedures `det-color`(v) and `rand-color`(v) may set the level of v to a different level, in which case the procedure `set-level`(v,ℓ) is called with the new level ℓ as input. Let $\ell(v)$ be the previous level of v. The procedure does nothing if $\ell = \ell(v)$. Otherwise:

If v is set to a lower level $\ell < \ell(v)$: we need to update the data structures of vertices in levels $[\ell+1, \ell(v)]$. For each vertex $w \in \mathcal{D}_v$ where $\ell + 1 \le \ell(w) \le \ell(v)$, we make the following data structure updates:

1. Delete w from $\mathcal{D}_v$. Delete the mutual pointers between w and p_c. Let w's color be c. Move w's color, c, in $\mathcal{C}_v$ to $\mathcal{C}_v^+$ if c is currently in $\mathcal{C}_v$. Delete the mutual pointers between c and p_c. Create mutual pointers between c and p_c^+. Increment w's color count $\mu_v^+(c)$ by 1.
2. Add w to $\mathcal{U}_v[\ell(w)]$. Add mutual pointers between w and p_c^+ where c is w's color.
3. Delete v from $\mathcal{U}_w[\ell(v)]$. Let v's color be c'. Delete the mutual pointers between v and $p_{c'}^+$. Decrement v's color count $\mu_w^+(c')$ by 1. If $\mu_w^+(c')$ is now 0, move c' from $\mathcal{C}_w^+$ to $\mathcal{C}_w$, delete the mutual pointers between c' and $p_{c'}^+$, and create mutual pointers between c' and $p_{c'}$.
4. Add v to $\mathcal{D}_w$. Add mutual pointers between v and $p_{c'}$ where c' is v's color if c' was moved to $\mathcal{C}_w$.
5. Add mutual pointers between all elements $v \in \mathcal{D}_w, w \in \mathcal{U}_v[\ell(w)], v \in \mathcal{N}_w, w \in \mathcal{N}_v$.
6. Add mutual pointers between all copies of the same element: i.e. $w \in \mathcal{D}_v, w \in \mathcal{N}_v$, and/or $w \in \mathcal{U}_v[\ell(w)]$.
7. Maintain mutual pointers between $\mathcal{P}_v[i_c]$, p_c, and p_c^+. Maintain mutual pointers

between $\mathcal{P}_w[i_{c'}]$, $p^+_{c'}$, and p_c.

If v is set to a higher level $\ell > \ell(v)$: we need to update the data structures of vertices in levels $[\ell(v), \ell - 1]$. Specifically, for each non-empty list $\mathcal{U}_v[i]$, with $\ell(v) \le i \le \ell - 1$, and for each vertex $w \in \mathcal{U}_v[i]$, we perform the following operations:

1. Delete w from $\mathcal{U}_v[i]$. Let c be the color of w. Delete the mutual pointers between w and p^+_c. Decrement $\mu^+_v(c)$ by 1. If $\mu^+_v(c) = 0$, then move c from $\mathcal{C}^+_v$ to $\mathcal{C}_v$, delete the mutual pointers between p^+_c and c, and add mutual pointers between p_c and c.
2. Add w to $\mathcal{D}_v$, create mutual pointers between w and p_c (where c is w's color), delete v from $\mathcal{D}_w$, and add v to $\mathcal{U}_w[\ell]$. Let v's color be c'. Delete the mutual pointers between v and $p_{c'}$. Add mutual pointers between v and $p^+_{c'}$. If c' is currently in $\mathcal{C}_w$, move v's color, c', in $\mathcal{C}_w$ to $\mathcal{C}^+_w$, delete mutual pointers between c' and $p_{c'}$, and add mutual pointers between c' and $p^+_{c'}$. Increment $\mu^+_w(c')$ by 1.
3. Add mutual pointers between all elements $w \in \mathcal{D}_v, v \in \mathcal{U}_w[\ell], v \in N_w, w \in N_v$.
4. Maintain mutual pointers between $\mathcal{P}_v[i_c]$, p_c, and p^+_c. Maintain mutual pointers between $\mathcal{P}_w[i_{c'}]$, $p_{c'}$, and $p^+_{c'}$.

The full pseudocode of this procedure can be found in Fig. 7-3.

7.5 Pseudocode

In the below pseudocode, we do not describe (most of) the straightforward but tedious pointer creation procedures. We assume that the corresponding pointers are created according to the procedure described in Section 7.4.1. In the cases where the pointer change is significant, we describe it in the pseudocode.

```
handle-insertion(u, v):
  1. N_v ← N_v ∪ {u};
  2. N_u ← N_u ∪ {v};
  3. If ℓ(u) > ℓ(v):
       (a) D_u ← D_u ∪ {v};
       (b) U_v[ℓ(u)] ← U_v[ℓ(u)] ∪ {u};
  4. Else if ℓ(u) = ℓ(v):
       (a) U_v[ℓ(u)] ← U_v[ℓ(u)] ∪ {u};
       (b) U_u[ℓ(v)] ← U_u[ℓ(v)] ∪ {v};
  5. Else:
       (a) D_v ← D_v ∪ {u};
       (b) U_u[ℓ(v)] ← U_u[ℓ(v)] ∪ {v};
  6. update-color-edge-insertion(u, v, c_u, c_v);
  7. If color(u) = color(v): /* if u and v have the same color */
       (a) recolor(u); /* assuming u is the most recently colored */
```

Figure 7-1: Handling edge insertion (u, v).

`handle-deletion`(u, v):

1. $\mathcal{N}_v \leftarrow \mathcal{N}_v \setminus \{u\}$;
2. $\mathcal{N}_u \leftarrow \mathcal{N}_u \setminus \{v\}$;
3. If $v \in \mathcal{D}_u$:
   (a) $\mathcal{D}_u \leftarrow \mathcal{D}_u \setminus \{v\}$;
   (b) $\mathcal{U}_v[\ell(u)] \leftarrow \mathcal{U}_v[\ell(u)] \setminus \{u\}$;
4. Else if $u \in \mathcal{D}_v$:
   (a) $\mathcal{D}_v \leftarrow \mathcal{D}_v \setminus \{u\}$;
   (b) $\mathcal{U}_u[\ell(v)] \leftarrow \mathcal{U}_u[\ell(v)] \setminus \{v\}$;
5. Else:
   (a) $\mathcal{U}_u[\ell(v)] \leftarrow \mathcal{U}_u[\ell(v)] \setminus \{v\}$;
   (b) $\mathcal{U}_v[\ell(u)] \leftarrow \mathcal{U}_v[\ell(u)] \setminus \{u\}$;
6. Remove all associate color pointers and shift colors between $\mathcal{C}_u$, $\mathcal{C}_u^+$ and $\mathcal{C}_w$, $\mathcal{C}_w^+$ as necessary;

Figure 7-2: Handling edge deletion (u, v).

`set-level`(v, ℓ):

1. For all $w \in \mathcal{D}_v$: /* update $\mathcal{U}_w$ regarding v's new level */
   (a) $\mathcal{U}_w[\ell(v)] \leftarrow \mathcal{U}_w[\ell(v)] \setminus \{v\}$;
   (b) $\mathcal{U}_w[\ell] \leftarrow \mathcal{U}_w[\ell] \cup \{v\}$;
2. If $\ell < \ell(v)$: /* in this case the level of v is decreased by at least one */
   (a) For all $w \in \mathcal{D}_v$ such that $\ell \leq \ell(w) < \ell(v)$: /* reassign color pointers*/
      i. $\mathcal{D}_v \leftarrow \mathcal{D}_v \setminus \{w\}$;
      ii. Delete mutual pointers between w and $p_{\text{color}(w)}$;
      iii. If $\text{color}(w) \in \mathcal{C}_v$:
         A. Move $\text{color}(w) \in \mathcal{C}_v$ to $\mathcal{C}_v^+$;
         B. Delete mutual pointers between $\text{color}(w)$ and $p_{\text{color}(w)}$;
         C. Create mutual pointers between $\text{color}(w)$ and $p^+_{\text{color}(w)}$;
      iv. Increment $\mu_v^+(\text{color}(w))$ by 1;
      v. $\mathcal{U}_v[\ell(w)] \leftarrow \mathcal{U}_v[\ell(w)] \cup \{w\}$;
      vi. Create mutual pointers between w and $p^+_{\text{color}(w)}$ if such pointers do not already exist;
      vii. $\mathcal{U}_w[\ell] \leftarrow \mathcal{U}_w[\ell] \setminus \{v\}$;
      viii. Delete mutual pointers between $\text{color}(v)$ and $p^+_{\text{color}(v)}$;
      ix. Decrement $\mu_w^+(\text{color}(v))$ by 1;
      x. If $\mu_w^+(\text{color}(v)) = 0$:
         A. Move $\text{color}(v) \in \mathcal{C}_w^+$ to $\mathcal{C}_w$;
         B. Delete mutual pointers between $\text{color}(v)$ and $p^+_{\text{color}(v)}$;
         C. Create mutual pointers between $\text{color}(v)$ and $p_{\text{color}(v)}$;
      xi. $\mathcal{D}_w \leftarrow \mathcal{D}_w \cup \{v\}$;
      xii. Create mutual pointers between v and $p_{\text{color}(v)}$ if such pointers do not already exist;
3. If $\ell > \ell(v)$: /* in this case the level of v is increased by at least one */ [a]
   (a) For all $i = \ell(v), \ldots, \ell - 1$ and all $w \in \mathcal{U}_v[i]$:
      i. $\mathcal{U}_v[i] \leftarrow \mathcal{U}_v[i] \setminus \{w\}$;
      ii. Decrement $\mu_v^+(\text{color}(w))$ by 1;
      iii. If $\mu_v^+(\text{color}(w)) = 0$: Move $\text{color}(w)$ from $\mathcal{C}_v^+$ to $\mathcal{C}_v$;
      iv. $\mathcal{D}_v \leftarrow \mathcal{D}_v \cup \{w\}$;
      v. $\mathcal{D}_w \leftarrow \mathcal{D}_w \setminus \{v\}$;
      vi. $\mathcal{U}_w[\ell] \leftarrow \mathcal{U}_w[\ell] \cup \{v\}$;
      vii. If $\text{color}(v) \in \mathcal{C}_w$: Move $\text{color}(v)$ from $\mathcal{C}_w$ to $\mathcal{C}_w^+$;
      viii. Increment $\mu_w^+(\text{color}(v))$ by 1;
4. $\ell(v) \leftarrow \ell$;

[a] For the sake of clarity and brevity, we do not describe the pointer deletions, creations, and changes in the case where $\ell > \ell(v)$ because these changes are almost identical to the changes given above for the case $\ell < \ell(v)$.

Figure 7-3: Setting the old level $\ell(v)$ of v to ℓ.

`recolor`(v):
1. If $\varphi_v(\ell(v)+1) < 3^{\ell(v)+2}$: `det-color`($v$);
2. Else `rand-color`(v);

Figure 7-4: Recoloring a vertex that collides with the color of an adjacent vertex after an edge insertion.

`det-color`(v):
1. For all $c \in \mathcal{C}_v$:
   (a) If c is not occupied by any vertex $w \in \mathcal{D}_v$ and $c \in \mathcal{C}_v$: /* if c is a blank color, color v with c */
      i. Set $\chi(i_v) = c$;
      ii. For all $w \in \mathcal{D}_v$:
         A. `update-color`(v, w, c).
      iii. `set-level`($v, -1$);
      iv. terminate the procedure; /* Note that the procedure will always terminate within this if statement because a blank color always exists by definition of $(\Delta+1)$-coloring. */

Figure 7-5: Coloring v deterministically with a blank color. It is assumed that $\varphi_v(\ell(v)+1) < 3^{\ell(v)+2}$.

`rand-color`(v):
1. $\ell^* \leftarrow \ell(v)$;
2. while $\varphi_v(\ell^*+1) \geq 3^{\ell^*+2}$: $\ell^* \leftarrow \ell^*+1$;
   /* ℓ^* is the minimum level after $\ell(v)$ with $\varphi_v(\ell^*+1) < 3^{\ell^*+2}$ */
3. `set-level`(v, ℓ^*); /* after this call $\ell(v) = \ell^*$ and $3^{\ell^*+1} \leq d_{\mathtt{out}}(v) = \varphi_v(\ell^*) < 3^{\ell^*+2}$ */
4. Pick a blank or unique color c from $\mathcal{C}_v$ uniformly at random;
   /* c is chosen with probability at most $2/3^{\ell^*+1}$ and $\ell(w) \leq \ell^*-1$ */
5. If $c \neq$ color(v): /* If c is not the previous color of v. */
   (a) Set $\chi(i_v) = c$;
   (b) For all $z \in \mathcal{D}_v$:
      i. `update-color`(v, z, c).
6. If c is a unique color (let $w \in \varphi_v(\ell^*)$ be the vertex that is colored with c):
   (a) `recolor`(w);

Figure 7-6: Coloring v at level ℓ^* higher than $\ell(v)$, with a random blank or unique color of level lower than ℓ^*. If the procedure chose a unique color, it calls `recolor` (which may call itself recursively) to color w. It is assumed that $\varphi_v(\ell(v)+1) \geq 3^{\ell(v)+2}$.

`update-color-edge-insertion`(v, w, c_v, c_w):

1. If $\ell(v) > \ell(w)$:
   (a) Locate $v \in \mathcal{U}_w[\ell(v)]$;
   (b) Delete the mutual pointers (if they exist) between v and $p^+_{c'}$ where c' is v's previous color; /* Note that v's previous color could be located by following pointers from v. */
   (c) Decrement $\mu^+_w(c')$ by 1 if pointers were deleted in the previous step; /* If no pointers were deleted, then w had no knowledge of v's previous color and we do not need to decrement */
   (d) If $\mu^+_w(c') = 0$:
      i. Move c' from $\mathcal{C}^+_w$ to $\mathcal{C}_w$ by appending c' to the end of the linked list representing $\mathcal{C}_w$;
   (e) Locate $p^+_{c_v}$ by following pointers from $\mathcal{P}_w$;
   (f) Create mutual pointers between v and $p^+_{c_v}$;
   (g) Increment $\mu^+_w(c_v)$ by 1;
   (h) If c_v is in $\mathcal{C}_w$:
      i. Move c_v from $\mathcal{C}_w$ to $\mathcal{C}^+_w$ by appending c_v to the end of the linked list representing $\mathcal{C}^+_w$;
   (i) Locate $w \in \mathcal{D}_v$.
   (j) Delete the mutual pointers (if they exist) between w and $p_{c''}$ where c'' is w's previous color;
   (k) Locate p_{c_v} by following pointers from $\mathcal{P}_v$;
   (l) Create mutual pointers between $w \in \mathcal{D}_v$ and p_{c_w};
2. Else if $\ell(v) < \ell(w)$:
   (a) /* Do the above except switch the roles of v and w as well as c_v and c_w. */
3. Else:
   (a) Locate $v \in \mathcal{U}_w[\ell(v)]$ and $w \in \mathcal{U}_v[\ell(w)]$;
   (b) /* Do the above procedure given in the case when $\ell(v) > \ell(w)$ for $v \in \mathcal{U}_w[\ell(v)]$ for both $v \in \mathcal{U}_w[\ell(v)]$ and $w \in \mathcal{U}_v[\ell(w)]$.*/

Figure 7-7: Updates the data structures with v and w's colors when an edge is inserted between v and w.

`update-color`(v, c_v):

1. For $w \in \mathcal{D}_v$:
   (a) Locate $v \in \mathcal{U}_w[\ell(v)]$;
   (b) Delete the mutual pointers (if they exist) between v and $p^+_{c'}$ where c' is v's previous color; /* Note that v's previous color could be located by following pointers from v. */
   (c) Decrement $\mu^+_w(c')$ by 1 if pointers were deleted in the previous step; /* If no pointers were deleted, then w had no knowledge of v's previous color and we do not need to decrement */
   (d) If $\mu^+_w(c') = 0$:
      i. Move c' from $\mathcal{C}^+_w$ to $\mathcal{C}_w$ by appending c' to the end of the linked list representing $\mathcal{C}_w$;
   (e) Locate $p^+_{c_v}$ by following pointers from $\mathcal{P}_w$;
   (f) Create mutual pointers between v and $p^+_{c_v}$;
   (g) Increment $\mu^+_w(c_v)$ by 1;
   (h) If c_v is in $\mathcal{C}_w$:
      i. Move c_v from $\mathcal{C}_w$ to $\mathcal{C}^+_w$ by appending c_v to the end of the linked list representing $\mathcal{C}^+_w$;

Figure 7-8: Updates the color pointers of v of all of v's down-neighbors when v changes color.

Chapter 8

Parallel Batch-Dynamic k-Clique Counting

This chapter presents results from the paper titled, "Parallel Batch-Dynamic k-Clique Counting" that the thesis author coauthored with Laxman Dhulipala, Julian Shun, and Shangdi Yu [DLSY20]. This paper appeared in the Symposium on Algorithmic Principles of Computer Systems (APOCS) 2021.

8.1 Introduction

Subgraph counting algorithms are fundamental graph analysis tools, with numerous applications in network classification in domains including social network analysis and bioinformatics. A particularly important type of subgraph for these applications is the triangle, or 3-clique—three vertices that are all mutually connected [New03]. Counting the number of triangles is a basic and fundamental task that is used in numerous social and network science measurements [Gra77, WS98].

In this chapter, we study the triangle counting problem and its generalization to higher cliques from the perspective of dynamic algorithms. A k-clique consists of k vertices and all $\binom{k}{2}$ possible edges among them (for applications of k-cliques, see, e.g., [HR05]). As many real-world graphs change rapidly in real-time, it is crucial to design dynamic algorithms that efficiently maintain k-cliques upon updates, since the cost of re-computation from scratch can be prohibitive. Furthermore, due to the fact that dynamic updates can occur at a rapid rate in practice, it is increasingly important to design ***batch-dynamic*** algorithms which can take arbitrarily large batches of updates (edge insertions or deletions) as their input. Finally, since the batches, and corresponding update complexity can be large, it is also desirable to use parallelism to speed-up maintenance and design algorithms that map to modern parallel architectures.

Due to the broad applicability of k-clique counting in practice and the fact that k-clique counting is a fundamental theoretical problem of its own right, there has been a large body of prior work on the problem. Theoretically, the fastest static algorithm for arbitrary graphs uses fast matrix multiplication, and counts 3ℓ cliques in $O(n^{\ell\omega})$ time where ω is the matrix multiplication exponent [NP85]. Considerable effort has also been devoted

to efficient combinatorial algorithms. Chiba and Nishizeki [CN85] show how to compute k-cliques in $O(\alpha^{k-2}m)$ work, where m is the number of edges in the graph and α is the arboricity of the graph. This algorithm was recently parallelized by Danisch et al. [DBS18a] (although not in polylogarithmic depth). Worst-case optimal join algorithms can perform k-clique counting in $O(m^{k/2})$ work as a special case [NPRR18, ALT+17]. Alon, Yuster, and Zwick [AYZ97] design an algorithm for triangle counting in the sequential model, based on fast matrix multiplication. Eisenbrand and Grandoni [EG04] then extend this result to k-clique counting based on fast matrix multiplication. Vassilevska designs a space-efficient combinatorial algorithm for k-clique counting [Vas09]. Finocchi et al. give clique counting algorithms for MapReduce [FFF15]. Jain and Seshadri provide probabilistic algorithms for estimating clique counts [JS17b]. The k-clique problem is also a classical problem in parameterized-complexity, and is known to be $W[1]$-complete [DF95a].

The problem of maintaining k-cliques under dynamic updates began more recently. Eppstein et al. [ES09, EGST12] design sequential dynamic algorithms for maintaining size-3 subgraphs in $O(h)$ amortized time and $O(mh)$ space and size-4 subgraphs in $O(h^2)$ amortized time and $O(mh^2)$ space, where h is the h-index of the graph ($h = O(\sqrt{m})$). Ammar et al. extend the worst-case optimal join algorithms to the parallel and dynamic setting [AMSJ18]. However, their update time is not better than the static worst-case optimal join algorithm. Recently, Kara et al. [KNN+19] present a sequential dynamic algorithm for maintaining triangles in $O(\sqrt{m})$ amortized time and $O(m)$ space. Dvorak and Tuma [DT13] present a dynamic algorithm that maintains k-cliques as a special case in $O(\alpha^{k-2}\log n)$ amortized time and $O(\alpha^{k-2}m)$ space by using low out-degree orientations for graphs with arboricity α.

Designing Parallel Batch-Dynamic Algorithms Traditional dynamic algorithms receive and apply updates one at a time. However, in the ***parallel batch-dynamic*** setting, the algorithm receives *batches of updates* one after the other, where each batch contains a mix of edge insertions and deletions. Unlike traditional dynamic algorithms, a parallel batch-dynamic algorithm can apply *all* of the updates together, and also take advantage of parallelism while processing the batch. We note that the edges inside of a batch may also be ordered (e.g., by a timestamp). If there are duplicate edge insertions within a batch, or an insertion of an edge followed by its deletion, a batch-dynamic algorithm can easily remove such redundant or nullifying updates.

The key challenge is to design the algorithm so that updates can be processed in parallel while ensuring low work and depth bounds. The only existing parallel batch-dynamic algorithms for k-clique counting are triangle counting algorithms by Ediger et al. [EJRB10] and Makkar et al. [MBG17], which take linear work per update in the worst case. The algorithms in this chapter make use of efficient data structures such as parallel hash tables, which let us perform parallel batches of edge insertions and deletions with better work and (polylogarithmic) depth bounds. To the best of our knowledge, no prior work has designed dynamic algorithms for the problem that support parallel batch updates with non-trivial theoretical guarantees.

Theoretically-efficient parallel dynamic (and batch-dynamic) algorithms have been designed for a variety of other graph problems, including minimum spanning tree [KPR18,

FL94, DF94], Euler tour trees [TDB19], connectivity [STTW18, AABD19, FL94], tree contraction [RT94, AAW17], and depth-first search [Kha17]. Very recently, parallel dynamic algorithms were also designed for the Massively Parallel Computation (MPC) setting [ILMP19, DDK+20].

Other Related Work There has been significant amount of work on practical parallel algorithms for the case of static 3-clique counting, also known as triangle counting. (e.g., [SV11, AKM13, PC13, PSKP14, ST15], among many others). Due to the importance of the problem, there is even an annual competition for parallel triangle counting solutions [Gra]. Practical static counting algorithms for the special cases of $k = 4$ and $k = 5$ have also been developed [HD14, ESBD16, PSV17, ANR+17, DAH17].

Dynamic algorithms have been studied in distributed models of computation under the framework of *self-stabilization* [Sch93]. In this setting, the system undergoes various changes, for example topology changes, and must quickly converge to a stable state. Most of the existing work in this setting focuses on a single change per round [CHHK16, BCH19, AOSS19], although algorithms studying multiple changes per round have been considered very recently [BKM19, CHDK+19]. Understanding how these algorithms relate to parallel batch-dynamic algorithms is an interesting question for future work.

Summary of Our Contributions In this chapter, we design parallel algorithms in the batch-dynamic setting, where the algorithm receives a batch of $|\mathcal{B}| \geq 1$ edge updates that can be processed in parallel. Our focus is on parallel batch-dynamic algorithms that admit strong theoretical bounds on their work and have polylogarithmic depth with high probability. Note that although our work bounds may be amortized, our depth will be polylogarithmic with high probability, leading to efficient RNC algorithms. As a special case of our results, we obtain algorithms for parallelizing single updates ($|\mathcal{B}| = 1$). We first design a parallel batch-dynamic triangle counting algorithm based on the sequential algorithm of Kara et al. [KNN+19]. For triangle counting, we obtain an algorithm that takes $O(|\mathcal{B}|\sqrt{|\mathcal{B}| + m})$ amortized work and $O(\log^*(|\mathcal{B}|+m))$ depth w.h.p.[1] assuming a fetch-and-add instruction that runs in $O(1)$ work and depth, and runs in $O(|\mathcal{B}|+m)$ space. The work of our parallel algorithm matches that of the sequential algorithm of performing one update at a time (i.e., it is work-efficient), and we can perform all updates in parallel with low depth.

We then present a new parallel batch-dynamic algorithm based on fast matrix multiplication. Using the best currently known parallel matrix multiplication [Wil12, LG14], our algorithm dynamically maintains the number of k-cliques in $O\left(\min\left(\mathcal{B}m^{0.469k-0.235}, (\mathcal{B}+m)^{0.469k+0.469}\right)\right)$ amortized work w.h.p. per batch of $\mathcal{B}$ updates where m is defined as the maximum number of edges in the graph before and after all updates in the batch are applied. Our approach is based on the algorithm of [AYZ97, EG04, NP85], and maintains triples of $k/3$-cliques that together form k-cliques. The depth is $O(\log(|\mathcal{B}| + m))$ w.h.p. and the space is $O\left((\mathcal{B}+m)^{0.469k+0.469}\right)$. Our results also imply an amortized time bound of $O\left(m^{0.469k-0.235}\right)$ per update for dense graphs

[1] We use "with high probability" (w.h.p.) to mean with probability at least $1 - 1/n^c$ for any constant $c > 0$.

in the sequential setting. Of potential independent interest, we present the first proof of logarithmic depth in the parallelization of any tensor-based fast matrix multiplication algorithms. We also give a simple batch-dynamic k-clique listing algorithm, based on enumerating smaller cliques and intersecting them with edges in the batch. The algorithm runs in $O(|\mathcal{B}|(m+|\mathcal{B}|)\alpha^{k-4})$ expected work, $O(\log^2 n)$ depth w.h.p., and $O(m+\mathcal{B})$ space for constant k.

Finally, we implement our new parallel batch-dynamic triangle counting algorithm for multicore CPUs, and present some experimental results on large graphs and with varying batch sizes using a 72-core machine with two-way hyper-threading. We found our parallel implementation to be much faster than the multicore implementation of Ediger et al. [EJRB10]. We also developed an optimized multicore implementation of the GPU algorithm by Makkar et al. [MBG17]. We found that our new algorithm is up to an order of magnitude faster than our CPU implementation of the Makkar et al. algorithm, and our new algorithm achieves 36.54–74.73x parallel speedup on 72 cores with hyper-threading. Our code is publicly available at `https://github.com/ParAlg/gbbs`.

8.2 Technical Overview

In this section, we present a high-level technical overview of our approach in this chapter.

Parallel Batch-Dynamic Triangle Counting

Our parallel batch-dynamic triangle counting algorithm is based on a recently proposed sequential dynamic algorithm due to Kara et al. [KNN+19]. They describe their algorithm in the database setting, in the context of dynamically maintaining the result of a database join. We provide a self-contained description of their sequential algorithm in Appendix C.1.

High-Level Approach The basic idea of the algorithm from [KNN+19] is to partition the vertex set using degree-based thresholding. Roughly, they specify a threshold $t = \Theta(\sqrt{m})$, and classify all vertices with degree less than t to be low-degree, and all vertices with degree larger than t to be high-degree. This thresholding technique is widely used in the design of fast static triangle-counting and k-clique counting algorithms, (e.g., [NP85, AYZ97]). Observe that if we insert an edge (u,v) incident to a low-degree vertex, u, we can enumerate all w in $N(u)$ in $O(\sqrt{m})$ expected time and check if (u,v,w) forms a triangle (checking if the (v,w) edge is present in G can be done by storing all edges in a hash table). In this way, edge updates incident to low-degree vertices are handled relatively simply. The more interesting case is how to handle edge updates between high-degree vertices. The main problem is that a single edge insertion (u,v) between two high-degree vertices can cause up to $O(n)$ triangles to appear in G, and enumerating all of these would require $O(n)$ work—potentially much more than $O(\sqrt{m})$. Therefore, the algorithm maintains an auxiliary data structure, $\mathcal{T}$, over wedges (2-paths). $\mathcal{T}$ stores for every pair of high-degree vertices (v,w), the number of low-degree vertices u that are connected to both v and w (i.e., (u,v) and (u,w) are both in E). Given this structure, the number of triangles formed

by the insertion of the edge (v, w) going between two high-degree vertices can be found in $O(1)$ time by checking the count for (v, w) in $\mathcal{T}$. Updates to $\mathcal{T}$ can be handled in $O(\sqrt{m})$ time, since $\mathcal{T}$ need only be updated when a low-degree vertex inserts/deletes a neighbor, and the number of entries in $\mathcal{T}$ that are affected is at most $t = O(\sqrt{m})$. Some additional care needs to be taken when specifying the threshold t to handle re-classifying vertices (going from low-degree to high-degree, or vice versa), and also to handle rebuilding the data structures, which leads to a bound of $O(\sqrt{m})$ amortized work per update for the algorithm.

Incorporating Batching and Parallelism The input to the parallel batch-dynamic algorithm is a batch containing (possibly) a mix of edge insertions and deletions (vertex insertions and deletions can be handled by inserting or deleting its incident edges). For simplicity, and without any loss in our asymptotic bounds, our algorithm handles insertions and deletions separately. The algorithm first removes all *nullifying* updates, which are updates that have no effect after applying the entire batch (i.e., an insertion which is subsequently deleted within the same batch, an insertion of an edge that already exists or a deletion of an edge that doesn't exist). This can easily be done within the bounds using basic parallel primitives. The algorithm then updates tables representing the adjacency information of both low-degree and high-degree vertices in parallel. To obtain strong parallel bounds, we represent these sets using parallel hash tables. For each insertion (deletion), we then determine the number of new triangles that are created (deleted). Since a given triangle could incorporate multiple edges within the same batch of insertions (deletions), our algorithm must carefully ensure that the triangle is counted only once, assigning each new inserted (deleted) triangle uniquely to one of the updates forming it. We then update the overall triangle count with the number of distinct triangles inserted (deleted) into the graph by the current batch of insertions (deletions). The remaining work of the algorithm cleans up mutable state such as marking of edges contained in the current update in the hash tables, and also migrating vertices between low-degree and high-degree states.

Worst-Case Optimality We note that the Kara et al. algorithm which is the basis for our parallel batch-dynamic triangle counting algorithm is conditionally optimal under the Online Matrix-Vector Multiplication (OMv) conjecture [HKNS15, KNN+19]. The same result in the sequential setting implies that our parallel work-bounds, which are work-efficient with respect to the Kara et al. algorithm [KNN+19], are conditionally optimal. It is an interesting question whether our depth bounds are conditionally optimal on the CRCW PRAM.

Dynamic k-Clique Counting via Fast Static Parallel Algorithms

Next, we present a very simple, and potentially practical algorithm for dynamically maintaining the number of k-cliques based on statically enumerating smaller cliques in the graph, and intersecting the enumerated cliques with the edge updates in the input batch. The algorithm is space-efficient, and is asymptotically more efficient than other methods for sparse graphs. Our algorithm is based on a recent and concurrent work proposing a

work-efficient parallel algorithm for counting k-cliques in work $O(m\alpha^{k-2})$ and polylogarithmic depth [SDS20]. Using this algorithm, we show that updating the k-clique count for a batch of $|\mathcal{B}|$ updates can be done in $O(|\mathcal{B}|(m+|\mathcal{B}|)\alpha^{k-4})$ work, and polylogarithmic depth, using $O(m+|\mathcal{B}|)$ space by using the static algorithm to (i) enumerate all $(k-2)$-cliques, and (ii) checking whether each $(k-2)$-clique forms a k-clique with an edge in the batch. This algorithm strictly outperforms re-computation using the new static parallel algorithm for $|\mathcal{B}| < \alpha^2$.

Theorem 8.2.1. *Given a collection of updates, $\mathcal{B}$, there is a batch-dynamic k-clique counting algorithm that updates the k-clique counts running in $O(|\mathcal{B}|(m+|\mathcal{B}|)\alpha^{k-4})$ expected work and $O(\log^2 n)$ depth w.h.p., using $O(m+|\mathcal{B}|)$ space for constant k.*

Dynamic k-Clique via Fast Matrix Multiplication

We then present a parallel batch-dynamic k-clique counting algorithm using parallel fast matrix multiplication (MM). Our algorithm is inspired by the static triangle counting algorithm of Alon, Yuster, and Zwick (AYZ) [AYZ97] and the static k-clique counting algorithm of [EG04] that uses MM-based triangle counting. We present a new dynamic algorithm that obtains better bounds than the simple algorithm based on static k-clique enumeration above for larger values of k. Specifically, assuming a parallel matrix multiplication exponent of ω_p, our algorithm handles batches of $|\mathcal{B}|$ edge insertions/deletions using $O\left(\min\left(|\mathcal{B}|m^{\frac{(2k-3)\omega_p}{3(1+\omega_p)}}, (m+\mathcal{B})^{\frac{2k\omega_p}{3(1+\omega_p)}}\right)\right)$ work and $O(\log m)$ depth w.h.p., in $O\left((m+\mathcal{B})^{\frac{2k\omega_p}{3(1+\omega_p)}}\right)$ space where m is the maximum number of edges in the graph before and after applying the batch of updates. To the best of our knowledge, the sequential (batch-dynamic) version of our algorithm also provides the best bounds for dynamic triangle counting in the sequential model for dense graphs for such values of k (assuming we use the best currently known matrix multiplication algorithm) [DT13].

High-Level Approach and Techniques For a given graph $G = (V, E)$, we create an auxiliary graph $G' = (V', E')$ with vertices and edges representing cliques of various sizes in G. For a given k-clique problem, vertices in V' represent cliques of size $k/3$ in G and edges (u, v) between vertices $u, v \in V'$ represent cliques of size $2k/3$ in G. Thus, a triangle in G' represents a k-clique in G. Specifically, there exist exactly $\binom{k}{k/3}\binom{2k/3}{k/3}$ different triangles in G' for each clique in G.

Given a batch of edge insertions and deletions to G, we create a set of edge insertions and deletions to G'. An edge is inserted in G' when a new $2k/3$-clique is created in G and an edge is deleted in G' when a $2k/3$-clique is destroyed in G. Suppose, for now, that we have a dynamic algorithm for processing the edge insertions/deletions into G'. Counting the number of triangles in G' after processing all edge insertions/deletions and dividing by $\binom{k}{k/3}\binom{2k/3}{k}$ provides us with the exact number of cliques in G.

There are a number of challenges that we must deal with when formulating our dynamic triangle counting algorithm for counting the triangles in G':

1. We cannot simply count all the triangles in G' after inserting/deleting the new edges as this does not perform better than a trivial static algorithm.
2. Any trivial dynamization of the AYZ algorithm will not be able to detect all new triangles in G'. Specifically, because the AYZ algorithm counts all triangles containing a low-degree vertex separately from all triangles containing only high-degree vertices, if an edge update only occurs between high-degree vertices, a trivial dynamization of the algorithm will not be able to detect any triangle that the two high-degree endpoints make with low-degree vertices.

To solve the first challenge, we dynamically count *low-degree* and *high-degree* vertices in different ways. Let $\ell = k/3$ and $M = 2m+1$. For some value of $0 < t < 1$, we define *low-degree* vertices to be vertices that have degree less than $M^{t\ell}/2$ and *high-degree* vertices to have degree greater than $3M^{t\ell}/2$. Vertices with degrees in the range $[M^{t\ell}/2, 3M^{t\ell}/2]$ can be classified as either low-degree or high-degree. We determine the specific value for t in Lemma 8.5.12. We perform rebalancing of the data structures as needed as they handle more updates. For low-degree vertices, we only count the triangles that include at least one newly inserted/deleted edge, at least one of whose endpoints is low-degree. This means that we do not need to count any pre-existing triangles that contain at least one low-degree vertex. For the high-degree vertices, because there is an upper bound on the maximum number of such vertices in the graph, we update an adjacency matrix A containing edges only between high-degree vertices. At the end of all of the edge updates, computing A^3 gives us a count of all of the triangles that contain three high-degree vertices.

This procedure immediately then leads to our second challenge. To solve this second challenge, we make the observation (proven in Lemma 8.5.3) that if there exists an edge update between two high-degree vertices that creates or destroys a triangle that contains a low-degree vertex in G', then there *must* exist at least one new edge insertion/deletion *that creates or destroys a triangle representing the same clique* to that low-degree vertex in the same batch of updates to G'. Thus, we can use one of those edge insertions/deletions to determine the new clique that was created and, through this method, find all triangles containing at least one low-degree vertex and at least one new edge update. Some care must be observed in implementing this procedure in order to not increase the runtime or space usage; such details can be found in Section 8.5.2.

Incorporating Batching and Parallelism When dealing with a batch of updates containing both edge insertions and deletions, we must be careful when vertices switch from being high-degree to being low-degree and vice versa. If we intersperse the edge insertions with the edge deletions, there is the possibility that a vertex switches between low and high-degree multiple times in a single batch. Thus, we batch all edge deletions together and perform these updates first before handling the edge insertions. After processing the batch of edge deletions, we must subsequently move any high-degree vertices that become low-degree to their correct data structures. After dealing with the edge insertions, we must similarly move any low-degree vertices that become high-degree to the correct data structures. Finally, for triangles that contain more than one edge update, we must account for potential double counting by different updates happening in parallel.

Such challenges are described and dealt with in Section 8.5.2 and Algorithm 31.

8.3 Parallel Batch-Dynamic Triangle Counting

We now present our parallel batch-dynamic triangle counting algorithm, which is based on the $O(m)$ space and $O(\sqrt{m})$ amortized update, sequential, dynamic algorithm of Kara et al. [KNN+19]. Theorem 8.3.1 summarizes the guarantees of our algorithm.

Theorem 8.3.1. *There exists a parallel batch-dynamic triangle counting algorithm that requires $O(|\mathcal{B}|(\sqrt{|\mathcal{B}|+m}))$ amortized work and $O(\log^*(\mathcal{B}+m))$ depth with high probability, and $O(|\mathcal{B}|+m)$ space for a batch of $|\mathcal{B}|$ edge updates.*

Our algorithm is work-efficient and achieves a significantly lower depth for a batch of updates than applying the updates one at a time using the sequential algorithm of [KNN+19]. We provide a detailed description of the fully dynamic sequential algorithm of [KNN+19] in Appendix C.1 for reference,[2] and a brief high-level overview of that algorithm in this section.

8.3.1 Sequential Algorithm Overview

Given a graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, let $M = 2m + 1$, $t_1 = \sqrt{M}/2$, and $t_2 = 3\sqrt{M}/2$. We classify a vertex as ***low-degree*** if its degree is at most t_1 and ***high-degree*** if its degree is at least t_2. Vertices with degree in between t_1 and t_2 can be classified either way.

Data Structures The algorithm partitions the edges into four edge-stores $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ based on a degree-based partitioning of the vertices. $\mathcal{HH}$ stores all of the edges (u, v), where both u and v are high-degree. $\mathcal{HL}$ stores edges (u, v), where u is high-degree and v is low-degree. $\mathcal{LH}$ stores the edges (u, v), where u is low-degree and v is high-degree. Finally, $\mathcal{LL}$ stores edges (u, v), where both u and v are low-degree.

The algorithm also maintains a wedge-store $\mathcal{T}$ (a wedge is a triple of distinct vertices (x, y, z) where both (x, y) and (y, z) are edges in E). For each pair of high-degree vertices u and v, the wedge-store $\mathcal{T}$ stores the number of wedges (u, w, v), where w is a low-degree vertex. $\mathcal{T}$ has the property that given an edge insertion (resp. deletion) (u, v) where both u and v are high-degree vertices, it returns the number of wedges (u, w, v), where w is low-degree, that u and v are part of in $O(1)$ expected time. $\mathcal{T}$ is implemented via a hash table indexed by pairs of high-degree vertices that stores the number of wedges for each pair.

Finally, we have an array containing the degrees of each vertex, $\mathcal{D}$.

[2] Kara et al. [KNN+19] described their algorithm for counting directed 3-cycles in relational databases, where each triangle edge is drawn from a different relation, and we simplified it for the case of undirected graphs.

Initialization Given a graph with m edges, the algorithm first initializes the triangle count C using a static triangle counting algorithm in $O(\alpha m) = O(m^{3/2})$ work and $O(m)$ space [Lat08]. The $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ tables are created by scanning all edges in the input graph and inserting them into the appropriate hash tables. $\mathcal{T}$ can be initialized by iterating over edges (u, w) in $\mathcal{HL}$ and for each w, iterating over all edges (w, v) in $\mathcal{LH}$ to find pairs of high-degree vertices u and v, and then incrementing $\mathcal{T}(u, v)$.

The Kara et al. Algorithm [KNN+19] Given an edge insertion (u, v) (deletions are handled similarly, and for simplicity assume that the edge does not already exist in G), the update algorithm must identify all tuples (u, w, v) where (u, w) and (v, w) already exist in G, since such triples correspond to new triangles formed by the edge insertion. The algorithm proceeds by considering how a triangle's edges can reside in the data structures. For example, if all of u, v, and w are high-degree, then the algorithm will enumerate these triangles by checking $\mathcal{HH}$ and finding all neighbors w of u that are also high-degree (there are at most $O(\sqrt{m})$ such neighbors), checking if the (v, w) edge exists in constant time. On the other hand, if u is low-degree, then checking its $O(\sqrt{m})$ many neighbors suffices to enumerate all new triangles. The interesting case is if both u and v are high-degree, but w is low-degree, since there can be much more than $O(\sqrt{m})$ such w's. This case is handled using $\mathcal{T}$, which stores for a given (u, v) edge in $\mathcal{HH}$ all w such that (w, u) and (w, v) both exist in $\mathcal{LH}$.

Finally, the algorithm updates the data structures, first inserting the new edge into the appropriate edge-store. The algorithm updates $\mathcal{T}$ as follows. If u and v are both low-degree or both high-degree, then no update is needed to $\mathcal{T}$. Otherwise, without loss of generality suppose u is low-degree and v is high-degree. Then, the algorithm enumerates all high-degree vertices w that are neighbors of u and increments the value of (v, w) in $\mathcal{T}$.

8.3.2 Parallel Batch-Dynamic Update Algorithm

We present a high-level overview of our parallel algorithm in this section, and a more detailed description in Section 8.3.3. We consider batches of $|\mathcal{B}|$ edge insertions and/or deletions. Let insert(u, v) represent the update corresponding to inserting an edge between vertices u and v, and delete(u, v) represent deleting the edge between u and v. We first preprocess the batch to account for updates that *nullify* each other. For example, an insert(u, v) update followed chronologically by a delete(u, v) update nullify each other because the (u, v) edge that is inserted is immediately deleted, resulting in no change to the graph. To process the batch consisting of nullifying updates, we claim that the only update that is not nullifying for any pair of vertices is the chronologically last update in the batch for that edge. Since all updates contain a timestamp, to account for nullifying updates we first find all updates on the same edge by hashing the updates by the edge that it is being performed on. Then, we run the parallel maximum-finding algorithm given in [Vis08] on each set of updates for each edge in parallel. This maximum-finding algorithm then returns the update with the largest timestamp (the most recent update) from the set of updates for each edge. This set of returned updates then composes a batch of non-nullifying updates.

Before we go into the details of our parallel batch-dynamic triangle counting algorithm, we first describe some challenges that must be solved in using Kara et al. [KNN+19] for the parallel batch-dynamic setting.

Challenges Because Kara et al. [KNN+19] only considers one update at a time in their algorithm, they do not deal with cases where a set of two or more updates creates a new triangle. Since, in our setting, we must account for batches of multiple updates, we encounter the following set of challenges:

(**1**) We must be able to efficient find new triangles that are created via two or more edge insertions.

(**2**) We must be able to handle insertions and deletions simultaneously meaning that a triangle with one inserted edge and one deleted edge should not be counted as a new triangle.

(**3**) We must account for over-counting of triangles due to multiple updates occurring simultaneously.

For the rest of this section, we assume that $|\mathcal{B}| \leq m$, as otherwise we can re-initialize our data structure using the static parallel triangle-counting algorithm [ST15][3] to get the count in $O(|\mathcal{B}|^{3/2})$ work, $O(\log^* |\mathcal{B}|)$ depth, and $O(|\mathcal{B}|)$ space (assuming atomic-add), which is within the bounds of Theorem 8.3.1.

Parallel Initialization Given a graph with m edges, we initialize the triangle count C using a static parallel triangle counting algorithm in $O(\alpha m) = O(m^{3/2})$ work, $O(\log^* m)$ depth, and $O(m)$ space [ST15], using atomic-add. We initialize $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ by scanning the edges in parallel and inserting them into the appropriate parallel hash tables. We initialize the degree array $\mathcal{D}$ by scanning the vertices. Both steps take $O(m)$ work and $O(\log^* m)$ depth w.h.p. $\mathcal{T}$ can be initialized by iterating over edges (u, w) in $\mathcal{HL}$ in parallel and for each w, iterating over all edges (w, v) in $\mathcal{LH}$ in parallel to find pairs of high-degree vertices u and v, and then incrementing $\mathcal{T}(u, v)$. The number of entries in $\mathcal{HL}$ is $O(m)$ and each w has $O(\sqrt{m})$ neighbors in $\mathcal{LH}$, giving a total of $O(m^{3/2})$ work and $O(\log^* m)$ depth w.h.p. for the hash table insertions. The amortized work per edge update is $O(\sqrt{m})$.

Data Structure Modifications We now describe additional information that is stored in $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, $\mathcal{LL}$, and $\mathcal{T}$, which is used by the batch-dynamic update algorithm:

(**1**) Every edge stored in $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ stores an associated state, indicating whether it is an ***old edge***, a ***new insertion*** or a ***new deletion***, which correspond to the values of 0, 1, and 2, respectively.

(**2**) $\mathcal{T}(u, v)$ stores a tuple with 5 values instead of a single value for each index (u, v). Specifically, a 5-tuple entry of $\mathcal{T}(u, v) = (t_1^{(u,v)}, t_2^{(u,v)}, t_3^{(u,v)}, t_4^{(u,v)}, t_5^{(u,v)})$ represents the following:

[3] The hashing-based version of the algorithm given in [ST15] can be modified to obtain the stated bounds if it does not do ranking and when using the $O(\log^* n)$ depth w.h.p. parallel hash table and uses atomic-add.

- $t_1^{(u,v)}$ represents the number of wedges with endpoints u and v that include only old edges.
- $t_2^{(u,v)}$ and $t_3^{(u,v)}$ represent the number of wedges with endpoints u and v containing one or two newly inserted edges, respectively.
- $t_4^{(u,v)}$ and $t_5^{(u,v)}$ represent the number of wedges with endpoints u and v containing one or two newly deleted edges, respectively. In other words, they are wedges that do not exist anymore due to one or two edge deletions.

Algorithm Overview We first remove updates in the batch that either insert edges already in the graph or delete edges not in the graph by using approximate compaction to filter. Next, we update the tables $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ with the new edge insertions. Recall that we must update the tables with both (u, v) and (v, u) (and similarly when we update these tables with edge deletions). We also mark these edges as newly inserted. Next, we update $\mathcal{D}$ with the new degrees of all vertices due to edge insertions. Since the degrees of some vertices have now increased, for low-degree vertices whose degree exceeds t_2, in parallel, we promote them to high-degree vertices, which involves updating the tables $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, $\mathcal{LL}$, and $\mathcal{T}$. Next, we update the tables $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ with new edge deletions, and mark these edges as newly deleted. We then call the procedures `update_table_insertions` and `update_table_deletions`, which update the wedge-table $\mathcal{T}$ based on all new insertions and all new deletions, respectively. At this point, our auxiliary data structures contain both new triangles formed by edge insertions, and triangles deleted due to edge deletions.

For each update in the batch, we then determine the number of new triangles that are created by counting different types of triangles that the edge appears in (based on the number of other updates forming the triangle). We then aggregate these per-update counts to update the overall triangle count.

Now that the count is updated, the remaining steps of the algorithm handle unmarking the edges and restoring the data structures so that they can be used by the next batch. We unmark all newly inserted edges from the tables, and delete all edges marked as deletes in this batch. Finally, we handle updating $\mathcal{T}$, the wedge-table for all insertions and deletions of edges incident to low-degree vertices. The last steps in our algorithm are to update the degrees in response to the newly inserted edges and the now truly deleted edges. Then, since the degrees of some high-degree vertices may drop below t_1 (and vice versa), we convert them to low-degree vertices and update the tables $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, $\mathcal{LL}$, and $\mathcal{T}$ (and vice versa). This step is called ***minor rebalancing***. Finally, if the number of edges in the graph becomes less than $M/4$ or greater than M we reset the values of M, t_1, and t_2, and re-initialize all of the data structures. This step is called ***major rebalancing***.

Algorithm Description A simplified version of our algorithm is shown below. The following COUNT-TRIANGLE procedure takes as input a batch of $|\mathcal{B}|$ updates $\mathcal{B}$ and returns the count of the updated number of triangles in the graph (assuming the initialization process has already been run on the input graph and all associated data structures are up-to-date).

Algorithm 27 Simplified parallel batch-dynamic triangle counting algorithm.

```
1: function COUNT-TRIANGLES(B)
2:   parfor insert(u, v) ∈ B do
3:     Update and label edges (u, v) and (v, u) in HH,
         HL, LH, and LL as inserted edges.
4:   parfor delete(u, v) ∈ B do
5:     Update and label edges (u, v) and (v, u) in HH,
         HL, LH, and LL as deleted edges.
6:   parfor insert(u, v) ∈ B or delete(u, v) ∈ B do
7:     Update T with (u, v). T records the number of
         wedges that have 0, 1, or 2 edge updates.
8:   parfor insert(u, v) ∈ B or delete(u, v) ∈ B do
9:     Count the number of new triangles and deleted
         triangles incident to edge (u, v), and account for
         duplicates.
10:  Rebalance data structures if necessary.
```

Small Example Batch Updates Here we provide a small example of processing a batch of updates. We assume that no rebalancing occurs. Suppose we have a batch of updates containing an edge insertion (u, v) with timestamp 3, an edge deletion (w, x) with timestamp 1, and an edge deletion (u, v) with timestamp 2. Since the edge insertion (u, v) has the later timestamp, it is the update that remains. After removing nullifying updates, the two updates that remain are insertion of (u, v) and deletion of (w, x). The algorithm first looks in $\mathcal{D}$ to find the degrees of u, v, w, and x in parallel. Suppose u, v, and w are high-degree and x is low-degree. We need to first update our data structures with the new edge updates. To update the data structure, we first update the edge table $\mathcal{HH}$ with (u, v) marked as an edge insertion. Then, we update the edge tables $\mathcal{HL}$ and $\mathcal{LH}$ with (w, x) as an edge deletion. Finally, we update the counts of wedges in $\mathcal{T}$ with (w, x)'s deletion. Specifically, for each of x's neighbors y in $\mathcal{LH}$, we update $\mathcal{T}(w, y)$ by incrementing $t_4^{(w,y)}$ (since (x, y) is not a new update).

After updating the data structures, we can count the changes to the total number of triangles in the graph. All of the following actions can be performed in parallel. Suppose that u comes lexicographically before v. We count the number of neighbors of u in $\mathcal{HH}$ and this will be the number of new triangles containing three high-degree vertices. To avoid overcounting, we do not count the number of high-degree neighbors of v. Since we are counting the number of triangles containing updates, we also do not count the number of high-degree neighbors of w since (w, x) cannot be part of any new triangles containing three high-degree vertices. Then, in parallel, we count the number of neighbors of x in $\mathcal{LL}$ and $\mathcal{LH}$; this is the number of deleted triangles containing one and two high-degree vertices, respectively. We use $\mathcal{T}$ to count the number of triangles containing one low-degree vertex and (u, v). To count the number of inserted triangles containing (u, v) and a low-degree vertex, we look up $t_1^{(u,v)}$ in $\mathcal{T}$ and add it to our final triangle count; all other

stored count values for (u, v) in $\mathcal{T}$ are 0 since there are no other new updates incident to u or v.

8.3.3 Parallel Batch-Dynamic Triangle Counting Detailed Algorithm

The detailed pseudocode of our parallel batch-dynamic triangle counting algorithm are shown below. Recall that the update procedure for a set of $|\mathcal{B}| \leq m$ non-nullifying updates is as follows (the subroutines used in the following steps are described afterward).

Algorithm 28 Detailed parallel batch-dynamic triangle counting procedure.

(**1**) Remove updates that insert edges already in the graph or delete edges not in the graph as well as nullifying updates using approximate compaction.

(**2**) Update tables $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ with the new edge insertions using insert(u, v) and insert(v, u). Mark these edges as newly inserted by running `mark_inserted_edges`$(\mathcal{B})$ on the batch of updates $\mathcal{B}$.

(**3**) Update tables $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ with new edge deletions using delete(u, v) and delete(v, u). Mark these edges as newly deleted using `mark_deleted_edges`$(\mathcal{B})$ on $\mathcal{B}$.

(**4**) Call `update_table_insertions`$(\mathcal{B})$ for the set $\mathcal{B}$ of all edge insertions insert(u, w), where either u or w is low-degree and the other is high-degree.

(**5**) Call `update_table_deletions`$(\mathcal{B})$ for the set $\mathcal{B}$ of all edge deletions delete(u, w) where either u or w is low-degree and the other is high-degree.

(**6**) For each update in the batch, determine the number of new triangles that are created by counting 6 values. Count the values using a 6-tuple, $(c_1, c_2, c_3, c_4, c_5, c_6)$ based on the number of other updates contained in a triangle:

(a) For each edge insertion insert(u, v) resulting in a triangle containing only one newly inserted edge (and no deleted edges), increment c_1 by `count_triangles`$(1, 0,$ insert$(u, v))$.

(b) For each edge insertion insert(u, v) resulting in a triangle containing two newly inserted edges (and no deleted edges), increment c_2 by `count_triangles`$(2, 0,$ insert$(u, v))$.

(c) For each edge insertion insert(u, v) resulting in a triangle containing three newly inserted edges, increment c_3 by `count_triangles`$(3, 0,$ insert$(u, v))$.

(d) For each edge deletion delete(u, v) resulting in a deleted triangle with one newly deleted edge, increment c_4 by `count_triangles`$(0, 1,$ delete$(u, v))$.

(e) For each edge deletion delete(u, v) resulting in a deleted triangle with two newly deleted edges, increment c_5 by `count_triangles`$(0, 2,$ delete$(u, v))$.

(f) For each edge deletion delete(u, v) resulting in a deleted triangle with three newly deleted edges, increment c_6 by `count_triangles`$(0, 3,$ delete$(u, v))$.

Let C be the previous count of the number of triangles. Update C to be $C + c_1 + (1/2)c_2 + (1/3)c_3 - c_4 - (1/2)c_5 - (1/3)c_6$, which becomes the new count.

(**7**) Scan through updates again. For each update, if the value stored in $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and/or $\mathcal{LL}$ is 2 (a deleted edge), remove this edge. If stored value is 1 (an inserted

edge), change the value to 0. For all updates where the endpoints are both high-degree or both low-degree, we are done. For each update (u, w) where either u or w is low-degree (assume without loss of generality that w is) and the other is high-degree, look for all high-degree neighbors v of w and update $\mathcal{T}(u, v)$ by summing all c_1, c_2,, and c_3 of the tuple and subtracting c_4 and c_5.

(**8**) Update $\mathcal{D}$ with the new degrees.

(**9**) Perform minor rebalancing for all vertices v that exceed t_2 in degree or fall under t_1 in parallel using `minor_rebalance(v)`. This makes a formerly low-degree vertex high-degree (and vice versa) and updates relevant structures.

(**10**) Perform major rebalancing if necessary (i.e., the total number of edges in the graph is less than $M/4$ or greater than M). Major rebalancing re-initializes all structures.

Procedure `mark_inserted_edges(`$\mathcal{B}$`)`. We scan through each of the insert(u, v) updates in $\mathcal{B}$ and mark (u, v) and (v, u) as newly inserted edges in $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and/or $\mathcal{LL}$ by storing a value of 1 associated with the edge.

Procedure `mark_deleted_edges(`$\mathcal{B}$`)`. Because we removed all nullifying updates before $\mathcal{B}$ is passed into the procedure, none of the deletion updates in $\mathcal{B}$ should delete newly inserted edges. For all edge deletions delete(u, v), we change the values stored under (u, v) and (v, u) from 0 to 2 in the tables $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and/or $\mathcal{LL}$.

Procedure `update_table_insertions(`$\mathcal{B}$`)`. For each $(u, w) \in \mathcal{B}$, assume without loss of generality that w is the low-degree vertex and do the following. We first find all of w's neighbors, v, in $\mathcal{LH}$ in parallel. Then, we determine for each neighbor v if (w, v) is new (marked as 1). If the edge (w, v) is not new, then increment the second value stored in the tuple with index $\mathcal{T}(u, v)$. If (w, v) is newly inserted, then increment the third value stored in $\mathcal{T}(u, v)$. The first, fourth, and fifth values stored in $\mathcal{T}(u, v)$ do not change in this step. The first, second, and third values count the number of edge insertions contained in the wedge keyed by (u, v). The first value counts all wedges with endpoints u and v that do not contain any edge update, the second count the number of wedges containing one edge insertion, and the third counts the number of wedges containing two edge insertions. Then, intuitively, the first, second, and third values will tell us later for edge insertion (u, v) between two high-degree vertices whether newly created triangles containing (u, v) have one (the only update being (u, v)), two, or three, respectively, new edge insertions from the batch update. We do not update the edge insertion counts of wedges which contain a mix of edge insertion updates and edge deletion updates.

Procedure `update_table_deletions(`$\mathcal{B}$`)`. For each $(u, w) \in \mathcal{B}$, assume without loss of generality that w is the low-degree vertex and do the following. We first find all of w's neighbors, v, in $\mathcal{LH}$ in parallel. Then, we determine for each neighbor v if (w, v) is a newly deleted edge (marked as 2). If (w, v) is not a newly deleted edge, increment the fourth value in the tuple stored in $\mathcal{T}(u, v)$ and decrement the first value. Otherwise, if (w, v) is a newly deleted edge, increment the fifth value of $\mathcal{T}(u, v)$ and decrement the first value. The second and third values in $\mathcal{T}(u, v)$ do not change in this step. For any key (u, v), the first, fourth, and fifth values gives the number of wedges with endpoints u and v that contain zero, one, or two edge deletions, respectively. Intuitively, the first, fourth, and fifth values tell us later whether newly deleted triangles have one (where the

only edge deletion is (u, v)), two, or three, respectively, new edge deletions from the batch update.

Procedure `count_triangles(i,d,update)`. This procedure returns the number of triangles containing the update insert(u, v) or delete(u, v) and exactly i newly inserted edges or exactly d newly deleted edges (the update itself counts as one newly inserted edge or one newly deleted edge). If at least one of u or v is low-degree, we search in the tables, $\mathcal{LH}$, and $\mathcal{LL}$ for neighbors of the low-degree vertex and the number of marked edges per triangle: edges marked as 1 for insertion updates and edges marked as 2 for deletion updates. If both u and v are high-degree, we first look through all high-degree vertices using $\mathcal{HH}$ to see if any form a triangle with both high-degree endpoints u and v of the update. This allows us to find all newly updated triangles containing only high-degree vertices. Then, we confirm the existence of a triangle for each neighbor found in the tables by checking for the third edge in $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, or $\mathcal{LL}$. We return only the counts containing the correct number of updates of the correct type. To avoid double counting for each update we do the following. Suppose all vertices are ordered lexicographically in some order. For any edge which contains two high-degree or two low-degree vertices, we search in $\mathcal{LL}$, $\mathcal{HH}$, and $\mathcal{LH}$ for exactly one of the two endpoints, the one that is lexicographically smaller.

Then, we return a tuple in $\mathcal{T}(u, v)$ based on the values of i and d to determine the count of triangles containing u and v and one low-degree vertex:

- Return the first value $t_1^{(u,v)}$ if either $i = 1$ or $d = 1$.
- Return the second value $t_2^{(u,v)}$ if $i = 2$.
- Return the third value $t_3^{(u,v)}$ if $i = 3$.
- Return the fourth value $t_4^{(u,v)}$ if $d = 2$.
- Return the fifth value $t_5^{(u,v)}$ if $d = 3$.

Note that we ignore all triangles that include more than one insertion update *and* more than one deletion update.

Procedure `minor_rebalance(u)`. This procedure performs a minor rebalance when either the degree of u decreases below t_1 or increases above t_2. We move all edges in $\mathcal{HH}$ and $\mathcal{HL}$ to $\mathcal{LH}$ and $\mathcal{LL}$ and vice versa. We also update $\mathcal{T}$ with new pairs of vertices that became high-degree and delete pairs that are no longer both high-degree.

8.3.4 Analysis

We prove the correctness of our algorithm in the following theorem. The proof is based on accounting for the contributions of an edge to each triangle that it participates in based on the number of other updated edges found in the triangle.

Theorem 8.3.2. *Our parallel batch-dynamic algorithm maintains the number of triangles in the graph.*

Proof. All triangles containing at least one low-degree vertex can be found either in $\mathcal{T}$ or by searching through $\mathcal{LH}$ and $\mathcal{LL}$. All triangles containing all high-degree vertices can be found by searching $\mathcal{HH}$. Suppose that an edge update insert(u, v) (resp. delete(u, v))

is part of $I_{(u,v)}$ (resp. $D_{(u,v)}$) triangles. We need to add or subtract from the total count of triangles $I_{(u,v)}$ or $D_{(u,v)}$, respectively. However, some of the triangles will be counted twice or three times if they contain more than one edge update. By dividing each triangle count by the number of updated edges they contain, each triangle is counted exactly once for the total count C. □

Overall Bound We now prove that our parallel batch-dynamic algorithm runs in $O(\mathcal{B}\sqrt{\mathcal{B}+m})$ work, $O(\log^*(\mathcal{B}+m))$ depth, and uses $O(\mathcal{B}+m)$ space. Henceforth, we assume that our algorithm uses the atomic-add instruction (see Section 4.2). Removing nullifying updates takes $O(\mathcal{B})$ total work, $O(\log^* \mathcal{B})$ depth w.h.p., and $O(\mathcal{B})$ space for hashing and the find-maximum procedure outlined in Section 8.3.2. In Item (1), we perform table lookups for the updates into $\mathcal{D}$ and in $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, or $\mathcal{LL}$, followed by approximate compaction to filter. The hash table lookups take $O(\mathcal{B})$ work and $O(\log^* m)$ depth with high probability and $O(m)$ space. Approximate compaction [GMV91] takes $O(\mathcal{B})$ work, $O(\log^* |\mathcal{B}|)$ depth, and $O(\mathcal{B})$ space. Item (2), Item (3), and Item (8) perform hash table insertions and updates on the batch of $O(\mathcal{B})$ edges, which takes $O(\mathcal{B})$ amortized work and $O(\log^* m)$ depth with high probability.

The next lemma shows that updating the tables based on the edges in the update (Item (4) and Item (5)) can be done in $O(\mathcal{B}\sqrt{m})$ work and $O(\log^* m)$ depth w.h.p., and $O(m)$ space.

Lemma 8.3.3. `update_table_insertions`$(\mathcal{B})$ *and* `update_table_deletions`$(\mathcal{B})$ *on a batch $\mathcal{B}$ of size $\mathcal{B}$ takes $O(\mathcal{B}\sqrt{m})$ work and $O(\log^*(\mathcal{B}+m))$ depth w.h.p., and $O(\mathcal{B}+m)$ space.*

Proof. For each w, we find all of its high-degree neighbors in $\mathcal{LH}$ and perform the increment or decrement in the corresponding entry in $\mathcal{T}$ in parallel (at this point, the vertices are still classified based on their original degrees). The total number of new neighbors gained across all vertices is $O(\mathcal{B})$ since there are $\mathcal{B}$ updates. Therefore, across all updates, this takes $O(\mathcal{B}\sqrt{m}+\mathcal{B})$ work and $O(\log^*(\mathcal{B}+m))$ depth w.h.p. due to hash table lookup and updates. Then, for all high-degree neighbors found, we perform the increments or decrements on the corresponding entries in $\mathcal{T}$ in parallel, taking the same bounds. All vertices can be processed in parallel, giving a total of $O(\mathcal{B}\sqrt{m}+\mathcal{B})$ work and $O(\log^*(\mathcal{B}+m))$ depth w.h.p. □

The next lemma bounds the complexity of updating the triangle count in Item (6).

Lemma 8.3.4. *Updating the triangle count takes $O(\mathcal{B}\sqrt{m})$ work and $O(\log^*(\mathcal{B}+m))$ depth w.h.p., and $O(\mathcal{B}+m)$ space.*

Proof. We initialize $c_1,\ldots,c_6$ to 0. For each edge update in $\mathcal{B}$ where both endpoints are high-degree, we perform lookups in $\mathcal{T}$ and $\mathcal{HH}$ for the relevant values in parallel and increment the appropriate c_i. Finding all triangles containing the edge update and containing only high-degree vertices takes $O(\mathcal{B}\sqrt{m})$ work and $O(\log^*(\mathcal{B}+m))$ depth w.h.p. This is because there are $O(\sqrt{m})$ high-degree vertices in total, and for each we check whether it appears in the $\mathcal{HH}$ table for both endpoints of each update. Performing lookups in $\mathcal{T}$ takes $O(\mathcal{B})$ work and $O(\log^*(\mathcal{B}+m))$ depth w.h.p.

For each update containing at least one endpoint with low-degree, we perform lookups in the tables $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ to find all triangles containing the update and increment the appropriate c_i. This takes $O(\mathcal{B}\sqrt{m}+\mathcal{B})$ work and $O(\log^*(\mathcal{B}+m))$ depth w.h.p. Incrementing all c_i's for all newly updated triangles takes $O(\mathcal{B})$ work and $O(1)$ depth. We then apply the equation in Item (6) to update C, which takes $O(1)$ work and depth. □

The following lemma bounds the cost for minor rebalancing, where a low-degree vertex becomes high-degree or vice versa (Item (9)).

Lemma 8.3.5. *Minor rebalancing for edge updates takes $O(\mathcal{B}\sqrt{m})$ amortized work and $O(\log^*(\mathcal{B}+m))$ depth w.h.p., and $O(\mathcal{B}+m)$ space.*

Proof. We describe the case of edge insertions, and the case for edge deletions is similar. Using approximate compaction to perform the filtering, we first find the set S of low-degree vertices exceeding t_2 in degree. This step takes $O(\mathcal{B})$ work and $O(\log^*\mathcal{B})$ depth w.h.p. For vertices in S, we then delete the edges from their old hash tables and move the edges to their new hash tables. The work for each vertex is proportional to its current degree, giving a total work of $O(\sum_{v\in S}\deg(v)) = O(\mathcal{B}\sqrt{m}+\mathcal{B})$ w.h.p. since the original degree of low-degree vertices is $O(\sqrt{m})$ and each edge in the batch could have caused at most 2 such vertices to have their degree increase by 1 (the w.h.p. is for parallel hash table operations).

In addition to moving the edges into new hash tables, we also have to update $\mathcal{T}$ with new pairs of vertices that became high-degree and delete pairs of vertices that are no longer both high-degree. To update these tables, we need to find all new pairs of high-degree vertices. There are at most $O(|\mathcal{B}|\sqrt{m+\mathcal{B}})$ such new pairs, which can be found by filtering neighbors using approximate compaction of vertices in S in $O(|\mathcal{B}|\sqrt{m+\mathcal{B}})$ work and $O(\log^*(\mathcal{B}+m))$ depth w.h.p. For each pair (u,v), we check all neighbors of an endpoint that just became high-degree and increment the entry $\mathcal{T}(u,v)$ for each low-degree neighbor w found that has edges (u,w) and (w,v). Low-degree neighbors have degree $O(\sqrt{m+\mathcal{B}})$, and so the total work is $O(\mathcal{B}(m+\mathcal{B}))$ and depth is $O(\log^*(\mathcal{B}+m))$ w.h.p. using atomic-add. There must have been $\Omega(\sqrt{m})$ updates on a vertex before minor rebalancing is triggered, and so the amortized work per update is $O(|\mathcal{B}|\sqrt{m})$ and the depth is $O(\log^* m)$ w.h.p. The space for filtering is $O(m+\mathcal{B})$. □

We now finish showing Theorem 8.3.1. Theorem 8.3.2 shows that our algorithm maintains the correct count of triangles. Lemma 8.3.3, Lemma 8.3.4, and Lemma 8.3.5 show that the cost of updating tables to reflect the batch, updating the triangle counts, and minor rebalancing is $O(\mathcal{B}\sqrt{m}+\mathcal{B})$ amortized work and $O(\log^*(\mathcal{B}+m))$ depth w.h.p., and $O(\mathcal{B}+m)$ space.

Item (7) can be done in $O(\mathcal{B}\sqrt{m})$ work and $O(\log^* m)$ depth as follows. We scan through the batch $\mathcal{B}$ in parallel and update the hash tables $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ in $O(\mathcal{B})$ work and $O(\log^*(\mathcal{B}+m))$ depth w.h.p. For all updates in $\mathcal{B}$ containing one high-degree vertex and one low-degree vertex, we update the table $\mathcal{T}$ in parallel by scanning the neighbors in $\mathcal{LH}$ of the low-degree vertex. This step takes $O(\mathcal{B}\sqrt{m}+\mathcal{B})$ work and $O(\log^*(\mathcal{B}+m))$ depth w.h.p. Major rebalancing (Item (10)) takes $O((\mathcal{B}+m)^{3/2})$ work and $O(\log^*(\mathcal{B}+m))$ depth by re-initializing the data structures. The rebalancing happens

every $\Omega(m)$ updates, and so the amortized work per update is $O(\sqrt{\mathcal{B}+m})$ and depth is $O(\log^*(\mathcal{B}+m))$ w.h.p.

Therefore, our update algorithm takes $O(|\mathcal{B}|\sqrt{|\mathcal{B}|+m})$ amortized work and $O(\log^*(\mathcal{B}+m))$ depth w.h.p., and $O(\mathcal{B}+m)$ space overall using atomic-add as stated in Theorem 8.3.1.

Bounds without Atomic-Add Without the atomic-add instruction, we can use a parallel reduction [Jaj92] to sum over values when needed. This is work-efficient and takes logarithmic depth, but uses space proportional to the number of values summed over in parallel. For updates, this is bounded by $O(|\mathcal{B}|\sqrt{m}+|\mathcal{B}|)$, and for initialization and major rebalancing, this is bounded by $O(\alpha m)$ [ST15]. This would give an overall bound of $O(|\mathcal{B}|(\sqrt{|\mathcal{B}|+m}))$ work and $O(\log(|\mathcal{B}|+m))$ depth w.h.p., and $O(\alpha m+|\mathcal{B}|\sqrt{m})$ space.

8.4 Dynamic k-Clique Counting via Fast Static Parallel Algorithms

In this section, we present a very simple algorithm for dynamically maintaining the number of k-cliques for $k > 3$ based on statically enumerating a number of smaller cliques in the graph, and intersecting the enumerated cliques with the edge updates in the input batch. Importantly, the algorithm is space-efficient, and only relies on simple primitives such as clique enumeration of cliques of size smaller than k, for which there are highly efficient algorithms both in theory and practice.

Fast Static Parallel k-Clique Enumeration The main tool used by algorithm is the following theorem, which is presented in concurrent and independent work [SDS20]:

Theorem 8.4.1 (Theorem 4.2 of [SDS20])**.** *There is a parallel algorithm that given a graph G can enumerate all k-cliques in G in $O(m\alpha^{k-2})$ expected work and $O(\log^2 n)$ depth w.h.p., using $O(m)$ space for constant k.*

Theorem 8.4.1 is proven by modifying the Chiba-Nishizeki (CN) algorithm in the parallel setting, and combining the CN algorithm with parallel low-outdegree orientation algorithms [BE10, GP11].

A Dynamic k-Clique Counting Algorithm Given Theorem 8.4.1, one approach to maintain the number of k-cliques in G upon receiving a batch of insertions or deletions $\mathcal{B}$ is to have each edge e in the batch simply enumerate all $(k-2)$-cliques, check whether e forms a k-clique with any of these $(k-2)$-cliques, and update the clique counts based on the newly discovered (or deleted) cliques.

Algorithm 29 presents a formalized version of this idea. The algorithm first removes all nullifying updates from $\mathcal{B}$. It then checks whether the batch is large ($|\mathcal{B}| \geq m$), and if so simply recomputes the overall k-clique count by re-running the static enumeration algorithm. Otherwise, the algorithm inserts the edge insertions in the batch into G, and stores them in a static parallel hash table $\mathcal{H}$ that maps each edge in the batch to a value indicating whether the edge is an insertion or deletion in $\mathcal{B}$.

Algorithm 29 Dynamic k-Clique Counting

```
1:  function k-Clique-Count(G = (V, E), B)
2:      Let N be the current count of cliques before processing the current batch.
3:      Remove nullifying updates from B.
4:      if |B| ≥ m then
5:          Rerun the static k-clique counting algorithm.
6:      else
7:          Insert all updates that are edge insertions in B into G.
8:          Let H be a static parallel hash table representing B.
9:          parfor e = {u, v} ∈ B do
10:             Enumerate all (k − 2)-cliques in G in parallel using the Algorithm from Theorem 8.4.1.
11:             parfor each enumerated (k − 2)-clique, C do
12:                 if C forms a newly inserted or newly deleted k-clique with e then
13:                     if e = (u, v) is the lexicographically-first edge in C in the batch then
14:                         Atomically update the k-clique count with C ∪ {u, v}: N ← N + 1.
15:         Delete all updates that are edge deletions in B from G.
```

Then, in parallel, for each edge $e = (u, v)$ in the batch, it enumerates all $(k-2)$-cliques in the graph. For each $(k-2)$-clique, C, the algorithm checks whether this clique forms a newly inserted or newly deleted k-clique with e. A newly inserted k-clique is one where at least one edge is an edge insertion in $\mathcal{B}$ and all other edges are not deleted in $\mathcal{B}$. Similarly a newly deleted k-clique is one where at least one edge is an edge deletion in $\mathcal{B}$ and all other edges are not edge insertions in $\mathcal{B}$. This step is done by querying the static parallel hash table $\mathcal{H}$ for each edge in the clique to check whether it is an insertion or deletion in $\mathcal{B}$. Cliques consisting of a mix of edge insertions and deletions are cliques that are not previously present before the batch, and will not be present after the batch, and are thus ignored.

For a newly inserted or deleted clique, the algorithm then checks whether e is the *lexicographically-first edge in the batch* inside of this clique formed by $C \cup \{u, v\}$ (otherwise, a different edge update from the batch will find and handle the processing of this clique).[4] Checking whether e is the lexicographically-first edge in a clique C is done by querying the static parallel hash table $\mathcal{H}$. For each clique where e is the lexicographically-first edge in the batch in the clique, we either atomically increment, or decrement the count, based on whether this clique is newly inserted or newly deleted. After the clique count has been updated, the algorithm updates G by performing the edge deletions from $\mathcal{B}$.

We note that we could just as well enumerate all of the $(k-2)$-cliques a single time, and then for each $(k-2)$-clique we discover, check whether it forms a k-clique with each edge in the batch. A practical optimization of this idea may store edges in a batch incident to

[4] An edge $e = (u, v)$ is the lexicographically first edge in the batch in a clique C if, $\forall e' = (u', v') \in C$ such that $(u', v') \in \mathcal{B}$, e is lexicographically smaller than e'. Note that we are working over an undirected graph without self-loops. By convention, when discussing lexicographic comparison, we have that for any $e = (u, v)$ that $u < v$; in other words, the order in the tuple representing the edge is based on the lexicographical order of the two endpoints.

their corresponding endpoints, and so vertices in the discovered $(k-2)$-clique would only need to check updates incident to the vertices in this clique. The asymptotic complexity of both ideas—joining cliques with edges, instead of edges with cliques, and pruning edges from the batch to consider—is the same in the worst case.

Correctness and Bounds If a k-clique in the graph is not incident to any edges in the batch, then its count is unaffected (since we only perform modifications to the count for cliques containing edges in $\mathcal{B}$). For cliques incident to edges in $\mathcal{B}$, we consider two cases. If the clique C is deleted after applying $\mathcal{B}$, observe that by decomposing C into a $(k-2)$-clique and the lexicographically-first marked edge e in C, C will be found and counted by e. The argument that a newly inserted clique, C, will be found is similar. Lastly, cliques consisting of both edge insertions and deletions in $\mathcal{B}$ will be correctly ignored by the check on Line 12. In other words, we check in parallel whether any enumerated k-clique $C \cup \{u, v\}$ contains both an edge deletion and an edge insertion (by checking in the hash table representing $\mathcal{B}$); if so, the k-clique composed of $C \cup \{u, v\}$ is not counted. This argument proves the following theorem:

Theorem 8.4.2. *Algorithm 29 correctly maintains the number of k-cliques in the graph.*

Theorem 8.4.3. *Given a collection of updates, $\mathcal{B}$, there is a batch-dynamic k-clique counting algorithm that updates the k-clique counts running in $O(|\mathcal{B}|(m+|\mathcal{B}|)\alpha^{k-4})$ expected work and $O(\log^2 n)$ depth w.h.p., using $O(m+|\mathcal{B}|)$ space for constant k.*

Proof. We analyze Algorithm 29. First, updating the graph, assuming that the edges incident to each vertex are represented sparsely using a parallel hash table, requires $O(|\mathcal{B}|)$ work and $O(\log^* n)$ depth w.h.p.

If $|\mathcal{B}| \geq m$, the algorithm calls the static k-clique counting algorithm, which takes $O((m+|\mathcal{B}|)\alpha^{k-2})$ expected work. Since $m = O(|\mathcal{B}|)$ and $\alpha^2 = O(m+|\mathcal{B}|)$, the work of calling the static algorithm is upper-bounded by $O(|\mathcal{B}|(m+|\mathcal{B}|)\alpha^{k-4})$ as required. Finally, the depth bound is $O(\log^{k-2} n)$ w.h.p. as required.

Otherwise, $|\mathcal{B}| < m$. Then, the algorithm first inserts and marks the batch in the graph. It also stores the edges in the batch in a parallel hash table. Creating the parallel hash table takes $O(|\mathcal{B}|)$ work and $O(\log^* n)$ depth w.h.p., which are both subsumed by the overall work and depth for the relevant setting of $k > 2$. For each update, we list all $(k-2)$-cliques using the algorithm from Theorem 8.4.1. This step can be done in $O((m+|\mathcal{B}|)\alpha^{k-4})$ expected work and $O(\log^{k-4} n)$ depth w.h.p. If the $(k-2)$-clique C forms a k-clique with e, then the cost of checking whether the clique is newly inserted or newly deleted using $\mathcal{H}$ costs $O(k)$ work, which is a constant, and $O(1)$ depth. The cost of checking whether e is the lexicographically first edge in $\mathcal{B}$ is also constant. Multiplying the cost of enumeration by the number of edges in the batch completes the proof. □

Our batch-dynamic algorithm outperforms re-computation using the static parallel k-clique counting algorithm for $|\mathcal{B}| = o(\alpha^2)$.

It is an interesting open question whether our dependence on m could be entirely removed from the update bound. Existing work has provided efficient sequential dynamic algorithms maintaining the k-clique count in $\tilde{O}(\alpha^{k-2})$ work per update using dynamic low

out-degree orientations [DT13]. It would be interesting to understand whether such an algorithm can be work-efficiently parallelized in the parallel batch-dynamic setting, which would allow the dynamic algorithm to match the work of static parallel recomputation up to logarithmic factors.

8.5 Dynamic k-Clique via Fast Matrix Multiplication

In this section, we present our final result which is a parallel batch-dynamic algorithm for counting k-cliques based on fast matrix multiplication in general graphs (which may be dense). For bounded arboricity graphs, we can also count cliques in $O(|\mathcal{B}|(m+|\mathcal{B}|)\alpha^{k-4})$ expected work and $O(\log^2 n)$ depth w.h.p., using $O(m+|\mathcal{B}|)$ space as explained in the previous section.

Using parallel matrix multiplication (discussed in Section 8.5.6), we achieve a better work bound (in terms of m) for large values of k than our bound of $O(|\mathcal{B}|(|\mathcal{B}|+m)\alpha^{k-4})$ obtained from the simple algorithm presented in Section 8.4. To the best of our knowledge, our algorithm (when made sequential) also achieves the best runtime for any sequential dynamic k-clique counting algorithm on dense graphs for large k when using the best currently known matrix multiplication algorithm [Wil12, LG14]. For values of $k > 9$, our MM based algorithm achieves $o(m^{k/2-1})$ amortized time compared to the arboricity-based algorithm of [DT13] that dynamically counts cliques in $\tilde{O}(\alpha^{k-2})$ amortized time where α is the arboricity of the graph (or $\tilde{O}\left(m^{k/2-1}\right)$ amortized time when $\alpha = \Omega\left(\sqrt{m}\right)$) or the trivial $O\left(m^{k/2-1}\right)$ algorithm of choosing all $k/2-1$ combinations of edges containing neighbors of the incident vertices of the inserted edge.

Our dynamic algorithm modifies the algorithm of [AYZ97] for counting triangles based on fast matrix multiplication and combines it with a dynamic version of the static k-clique counting algorithm of [EG04] to count the number of k-cliques under edge updates in batches of size $|\mathcal{B}|$. Section 8.5.1–Section 8.5.4 proves the following theorem for the case when $k \bmod 3 = 0$. Section 8.5.5 describes the changes needed for the case when $k \bmod 3 \neq 0$.

Theorem 8.5.1. *There exists a parallel batch-dynamic algorithm for counting the number of k-cliques, where $k \bmod 3 = 0$, that takes $O\left(\min\left(|\mathcal{B}|m^{\frac{(2k-3)\omega_p}{3(1+\omega_p)}}, (m+\mathcal{B})^{\frac{2k\omega_p}{3(1+\omega_p)}}\right)\right)$ amortized work and $O(\log(m+\mathcal{B}))$ depth w.h.p., in $O\left((m+\mathcal{B})^{\frac{2k\omega_p}{3(1+\omega_p)}}\right)$ space, given a parallel matrix multiplication algorithm with exponent ω_p.*

Using the best currently known matrix multiplication algorithms with exponent $\omega_p = 2.373$, we obtain the following work and space bounds.

Corollary 8.5.2. *There exists a parallel batch-dynamic algorithm for counting the number of k-cliques, where $k \bmod 3 = 0$, which takes $O\left(\min(\mathcal{B}m^{0.469k-0.704}, (m+\mathcal{B})^{0.469k})\right)$ work and $O(\log(m+\mathcal{B}))$ depth w.h.p., in $O\left((m+\mathcal{B})^{0.469k}\right)$ space by Corollary 8.5.19.*

Specifically, when amortized over the total number of edge updates $\mathcal{B}$, we obtain an amortized work bound of $O(m^{0.469k-0.704})$ per edge update which is asymptotically better than the combinatorial bound of $O\left(m^{k/2-1}\right)$ per update for $k > 9$. To the best of our knowledge, this is also the best known worst-case bound for dense graphs in the sequential setting.

Observe that our update algorithm only needs to handle batches of size $0 < \mathcal{B} \leq m^{\omega_p/(1+\omega_p)}$. For batches which have size $\mathcal{B} > m^{\omega_p/(1+\omega_p)}$, we can reinitialize our data structures in $O((m+\mathcal{B})^{0.469k})$ work ($O\left(m^{0.469k-0.704}\right)$ amortized work per update in the batch), $O(\log \mathcal{B})$ depth, and $O((m+\mathcal{B})^{0.469k})$ space using our initialization algorithm described in Lemma 8.5.5 and the fast parallel matrix multiplication of Corollary 8.5.19, which is faster than using the update algorithm (in general, we can use any fast matrix multiplication algorithm that has low depth, but the cutoff for when to reinitialize would be different). The analysis of the reinitialization procedure (similar to the static case presented by Alon, Yuster, and Zwick [AYZ97]) is provided in Section 8.5.4. Thus, in the following sections, we only describe our dynamic update procedures for batches of size $0 < \mathcal{B} \leq m^{\omega_p/(1+\omega_p)}$.

8.5.1 Our Algorithm

In what follows, we assume that $k \bmod 3 = 0$ (please refer to Section 8.5.5 for $k \bmod 3 \neq 0$). We use a batch-dynamic triangle counting algorithm as a subroutine for our batch-dynamic k-clique algorithm. Our algorithm for maintaining triangles is a batch-dynamic version of the triangle counting algorithm by Alon, Yuster, and Zwick (AYZ) [AYZ97]. However, our dynamic algorithm cannot directly be used for the case of $k = 3$ (and only applies for cases $k > 3$) due to the following challenge which we resolve in Section 8.5.2. Furthermore, our analysis also assumes $k > 6$ for greater simplicity and since for smaller k, our algorithm from Section 8.4 is also faster.

Adapting the Static Algorithm We face a major challenge when adapting the algorithm of Alon, Yuster, and Zwick [AYZ97] for our setting as well as for the sequential setting. Because the AYZ algorithm is meant to count cliques in the static setting, it is fine to consider two different types of triangles and count the triangles of each type separately. The two different types of triangles considered are triangles which contain at least one low-degree vertex and triangles which contain only high-degree vertices. In the static case, we can find all low-degree vertices, but in the dynamic case, we cannot afford to look at all low-degree vertices. If we only look at low-degree vertices incident to edge updates, then the following case may occur: an edge update between two high-degree nodes forms a new triangle incident to a low-degree node. In such a case, only looking at the vertices adjacent to this edge update will not find this triangle. We resolve this issue for $k > 3$ via Lemma 8.5.3 in Section 8.5.2.

Definitions and Data Structures Given a graph G, we construct an auxiliary graph G' consisting of vertices where each vertex represents a clique of size $\ell = k/3$ in G.[5] An edge (u, v) between two vertices in G' exists if and only if the cliques represented by u and v form a clique of size 2ℓ in G. Our algorithm maintains a dynamic total triangle count C on G'. Let $M = 2m + 1$ and let a ***low-degree*** vertex in G' be a vertex with degree less than $M^{t\ell}/2$ (for some $0 < t < 1$ to be determined later) and a ***high-degree*** vertex in G' be a vertex with degree greater than $3M^{t\ell}/2$. The vertices with degree in the range $[M^{t\ell}/2, 3M^{t\ell}/2]$ can be classified as either low-degree or high-degree. In addition to the total triangle count, we maintain a count, $C_{\mathcal{L}}$, of all triangles involving a low-degree vertex. Using the algorithm of AYZ [AYZ97], we assume we have a two-level hash table, $\mathcal{L}$, representing the neighbors of low-degree vertices in G' (a table mapping a low-degree vertex to another hash table containing its incident edges). We also maintain the adjacency matrix A of high-degree vertices in G' used in AYZ as a two-level hash table for easy insertion and deletion of additional high-degree vertices. Finally, we maintain another hash table $\mathcal{D}$ which dynamically maintains the degrees of the vertices.

An simplified version of the algorithm is given in Algorithm 30.

Algorithm 30 Simplified matrix multiplication k-clique counting algorithm.

function COUNT-CLIQUES($\mathcal{B}$)
  Update graph G' with $\mathcal{B}$ by inserting new ℓ- and 2ℓ-cliques.
  Find batch of insertions into G', $\mathcal{B}'_I$, and batch of deletions, $\mathcal{B}'_D$.
  Determine the final degrees of every vertex in G' after performing updates $\mathcal{B}'_I$ and $\mathcal{B}'_D$.
  parfor insert$(u, v) \in \mathcal{B}'_I$, `handle-deletion`$(u, v) \in \mathcal{B}'_D$ **do**[6]
    if either u or v is low-degree: $d(u) \leq \delta$ or $d(v) \leq \delta$ **then**
      Enumerate all triangles containing (u, v). Let this set be T.
      By Lemma 8.5.3, find all possible triangles representing the same triangle $t \in T$.
      Correct for duplicate counting of triangles.
    else
      Update A (adjacency list for high-degree vertices).
  Compute A^3. The diagonal provides the triangle counts for all triangles containing only high-degree vertices.
  Sum the counts of all triangles.
  Correct for duplicate counting of cliques.

[5] We use a hash table $\mathcal{Q}$ that stores each vertex in G' as an index to a set of vertices in G and also stores each set of vertices composing an ℓ-clique in G (lexicographically sort the vertices and turn into a string) as an index to a vertex in G'.

[6] Some care must be taken to ensure that rebalancing does not incur too much work. The details of how to deal with rebalancing are given in the full implementation, Algorithm 31.

8.5.2 Overview

Our algorithm proceeds as follows. Each edge in an update in the batch (edges in G) can either create at most $O(m^{k/3-1})$ new $(2k/3)$-cliques or disrupt $O(m^{k/3-1})$ existing $(2k/3)$-cliques in G. We treat each of these newly created or destroyed cliques as an edge insertion or deletion in G'. Since we preprocess the updates to G such that there are no duplicate or nullifying updates, a destroyed clique cannot be created again or vice versa. This means that the set of updates to G' will also contain no nullifying updates.

Importantly, the AYZ algorithm does not take into account edge insertions and deletions between two high-degree vertices that create or destroy triangles containing at least one low-degree vertex.[7] Thus, we must prove the following lemma for any edge insertion/deletion in G that results in an edge insertion in G' between two high-degree vertices which creates or destroys a triangle containing a low-degree vertex. This lemma is crucial for our algorithm, since it ensures that a triangle formed by two high-degree vertices and a low-degree vertex will be discovered by enumerating all triangles formed or deleted by an edge update incident to the low-degree vertex, and its current edges. Furthermore, this lemma is the reason why our algorithm does not work for $k = 3$ cliques.

Lemma 8.5.3. *Given a graph $G = (V, E)$, the corresponding $G' = (V', E')$, and for $k > 3$, suppose an edge insertion (resp. deletion) between two high-degree vertices in G' creates a new triangle, (u_H, w_H, x_L), in G' which contains a low-degree vertex x_L. Let $R(y)$ denote the set of vertices in V represented by a vertex $y \in V'$. Then, there exists a new edge insertion (resp. deletion) in G' that is incident to x_L and creates a new triangle (u', w', x_L) such that $R(u') \cup R(w') = R(u_H) \cup R(w_H)$.*

Proof. We prove this lemma for edge insertions in G. The proof can be easily modified to account for the case of edge deletions in G. Suppose an edge insertion (y, z) in G leads to an edge insertion in G' between the two high-degree vertices u_H and w_H that creates the new triangle (u_H, w_H, x_L). The creation of the new triangle signifies that a new clique was created in G consisting of vertices $R(u_H) \cup R(w_H) \cup R(x_L)$. Then, the edge insertion (y, z) created a new $2k/3$-clique in G consisting of the vertices in $R(u_H) \cup R(w_H)$. Since the edge (y, z) between $y, z \in V$ did not exist previously but now exists, $\binom{2k/3-2}{k/3-2}$ new cliques were created using the set of vertices in $R(u_H) \cup R(w_H)$. Each of these new cliques corresponds to a new vertex in G'. Suppose u' is one such new vertex representing vertex set $R(u') \subseteq R(u_H) \cup R(w_H)$ and w' represents vertex set $R(w') = (R(u_H) \cup R(w_H)) \setminus R(u')$. Then, new edges are inserted between u' and w' and between u' and x_L (the edge (w', x_L) might be a newly inserted edge or it is already present in the graph) since all triangles representing the clique of vertices (u_H, w_H, x_L) must be present in G'. Thus, the new triangle (u', w', x_L) is created in G'. □

We now describe our dynamic clique counting algorithm that combines the AYZ algorithm [AYZ97] with the clique counting algorithm of [EG04]. Given the batch of edge insertions/deletions into G, we first compute the duplicate and nullifying updates and remove them. Then, for a set of insertions/deletions into G', we form two batches, one containing the edge insertions and one containing the edge deletions. Given the batch

[7] Note that this is fine for the static case but not for the dynamic case.

of updates to G', we now formulate a dynamic version of the AYZ algorithm [AYZ97] on the updates to G'. For the batch of updates, we first look at the updates pertaining to the low-degree vertices. For every update (u, v) that contains at least one low-degree vertex (without loss of generality, let v be a low-degree vertex), we search all of v's $O\left(3M^{t\ell}/2\right)$ neighbors and check whether a triangle is formed (resp. deleted). For each triangle formed (resp. deleted), we update the total triangle count of the graph G'. For high-degree vertices, we update our adjacency matrix A containing vertices with high-degree. To compute the triangles containing high-degree vertices, we need only compute A^3 (the diagonal will then provide us with the triangle counts). Lastly, one clique results in many different copies of triangles. We must obtain the correct clique count by dividing the number of triangles by the number of ways we can partition the vertices in a k-clique into triples of subcliques of size $k/3$.

8.5.3 Detailed Parallel Batch-Dynamic Matrix Multiplication Based Algorithm

The analysis we perform in Section 8.5.4 on the efficiency of our algorithm is with respect to the detailed implementation. We provide the detailed description and implementation of our algorithm below in Algorithm 31.

Algorithm 31 Detailed matrix multiplication based parallel batch-dynamic k-clique counting algorithm.

(**1**) Given a batch $\mathcal{B}$ of non-nullifying edge updates,[8] first update the graph G'. If the update is an insertion, `insert`(u, v), add all new ℓ-cliques created by it into G'. If the update is a deletion, `handle-deletion`(u, v), mark all ℓ-cliques destroyed by it in G'.[9] For each update, `insert`(u, v) or `handle-deletion`(u, v), determine all 2ℓ-cliques that include it. This will determine the set of edge insertions/deletions into G'. Let all edge updates that destroy 2ℓ-cliques be a batch $\mathcal{B}'_D$ of edge deletions in G'. Then, let all 2ℓ-cliques formed by edge updates be a batch of edge insertions $\mathcal{B}'_I$ into G'. Note that edge insertions in the batch could be edges for newly created vertices; for each such newly created vertex, we also add the vertex into G' and its associated data structures.

(**2**) Determine the final degree of each vertex after all insertions in $\mathcal{B}'_I$ and all deletions in $\mathcal{B}'_D$. (We do not perform the updates yet–only compute the final degrees.) For all vertices, X, which become low-degree after the set of all updates (and were originally high-degree), we create a batch of updates $\mathcal{B}'_{I,L}$ consisting of old edges (not update edges) that are adjacent to vertices in X and were not deleted by the batches of updates. For all vertices, Y, which become high-degree after the set of updates (and were originally low-degree), we create a batch of updates $\mathcal{B}'_{D,H}$ consisting of old edges adjacent to vertices in Y that were not deleted after the batches of updates. [10]

[8] Recall that we can always remove nullifying edge updates as given in Section 8.3.2.

[9] We check in our hash table $\mathcal{Q}$ whether each newly created (deleted) ℓ-clique is already represented (non-existent) in the graph G'. If not, we insert the new clique and/or remove an old clique from $\mathcal{Q}$.

[10] The batch of updates $\mathcal{B}'_{I,L}$ is used to rebalance the data structures when vertices need to be removed

(**3**) Let the edges in $\mathcal{B}'_D \cup \mathcal{B}'_{D,H}$ be the batch of edge deletions to G'. For each of the edges in $\mathcal{B}'_D \cup \mathcal{B}'_{D,H}$, we first count the number of triangles it is a part of that contain at least one low-degree vertex. We call this the set of deleted triangles. Let this number of deleted triangles be T_D (initially set $T_D = 0$).
   (a) To count the number of triangles that contain at least one low-degree vertex, we first check for each edge whether one of its endpoints is low-degree. Let this set of edge deletions be $D'_L \subseteq \mathcal{B}'_D \cup \mathcal{B}'_{D,H}$.
   (b) For every edge $(u', v') \in D'_L$, without loss of generality let u' be the lexicographically[11] first low-degree vertex. For every edge (u', w') incident to u', check whether (u', v') forms a triangle with (u', w').
   (c) For every (u', v', w') triangle deleted (where (u', v', w') is sorted lexicographically), call
   $t \leftarrow$ `count_updated_low_degree_triangles`$((u', v', w'), (u', v'))$, and atomically update $T_D \leftarrow T_D + t$.

(**4**) Update $C_{\mathcal{L}} \leftarrow C_{\mathcal{L}} - T_D$.

(**5**) Update the data structures using the batches of edges insertions and deletions, $\mathcal{B}'_D$ and $\mathcal{B}'_I$:
   (a) Using $\mathcal{B}'_D$, delete the relevant edges in $\mathcal{L}$ (containing neighbors of low-degree vertices) and then change the relevant values in A to 0. We also update $\mathcal{D}$ with the new degrees of the vertices for which an adjacent edge was deleted.
   (b) For the batch of edge insertions into G', $\mathcal{B}'_I$, we first insert the relevant edges into $\mathcal{L}$. Then, we change the relevant entries in A from 0 to 1. Finally, we update $\mathcal{D}$ with the new degrees of the vertices following the edge insertions.
   (c) Remove all vertices which are no longer high-degree (i.e. their degree is now less than $M^{t\ell}/2$) from A. Create entries in $\mathcal{L}$ for all edges adjacent to each vertex that was removed from A.
   (d) Remove the edges of all vertices which are no longer low-degree (i.e. their degree is now greater than $3M^{t\ell}/2$) from $\mathcal{L}$ and create new entries in A with the new high-degree vertices. Set the relevant entries in A corresponding to edges adjacent to the new high-degree vertices to 1.

(**6**) Let the edges in $\mathcal{B}'_I \cup \mathcal{B}'_{I,L}$ be the batch of edge insertions to G'. For each of the edges in $\mathcal{B}'_I \cup \mathcal{B}'_{I,L}$, we first count the number of triangles it is a part of that contain at least one low-degree vertex. We call this the set of inserted triangles. Let this value be T_I ($T_I = 0$ initially).
   (a) To count the number of triangles that contain at least one low-degree vertex, we first check for each edge whether one of its endpoints is low-degree. Let this set of edge insertions be $I'_L \subseteq \mathcal{B}'_I \cup \mathcal{B}'_{I,L}$.
   (b) For every edge $(u', v') \in I'_L$, without loss of generality let u' be the lexicographically first low-degree vertex. For every edge (u', w') of u', check whether (u', v')

from A after becoming low-degree. Because the edges adjacent to these vertices need to be inserted into the structures maintaining low-degree vertices, $\mathcal{B}'_{I,L}$, then, can be thought of as a set of edge insertions to update low-degree data structures. Similarly, vertices which become high-degree need to be deleted from low-degree structures, and hence, $\mathcal{B}'_{D,H}$ can be thought of as a set of edge deletions from low-degree structures.

[11] The specific lexicographical order for the vertices in G' is fixed but can be arbitrary.

forms a triangle with (u', w').

(c) For every newly inserted triangle (u', v', w') (where (u', v', w') is sorted lexicographically), call
$t = \texttt{count\_updated\_low\_degree\_triangles}((u', v', w'), (u', v'))$, and atomically update $T_I \leftarrow T_I + t$.

(7) Update $C_{\mathcal{L}} \leftarrow C_{\mathcal{L}} + T_I$.

(8) We perform parallel matrix multiplication after all entries in A have been modified to calculate $S = A^3$. Then, $C_{\mathcal{H}} = \frac{1}{2} \sum_{i \in n} S_{i,i}$.

(9) Update $C \leftarrow C_{\mathcal{L}} + C_{\mathcal{H}}$.

(10) Compute the number of k-cliques by dividing C by $\binom{k}{k/3}\binom{2k/3}{k/3}$.

(11) If m falls outside the range $[M/4, M]$, then reinitialize the degree thresholds and data structures.

Algorithm 31 uses a subroutine defined below in Algorithm 32.

Algorithm 32 Subroutine used in our detailed matrix multiplication k-clique counting algorithm that counts the number of unique triangles containing an edge.

(1) Let $u', v', w' \in V'$ represent the sets of vertices $U', X', W' \subseteq V$, respectively.

(2) Enumerate all possible triangles that represent the clique containing vertices $U' \cup X' \cup W'$.

(3) Sort the vertices of each triangle lexicographically to obtain tuples of vertices representing the triangles. Let $\text{ID}(u', v')$ be the ID of edge (u', v').[12]

(4) For each enumerated tuple (x', y', z'), create a label containing the tuple representing the triangle concatenated with all labels (sorted lexicographically) of edges that are updates in the triangle. Thus, each label can have 4 to 6 entries consisting of the three vertices of a triangle tuple and at most 3 edge labels. For example, suppose that (x', y') is the only edge that is an updated edge in triangle (x', y', z'). Then, the label representing this triangle is $(x', y', z', \text{ID}(x', y'))$ where the ID of the edge is given by $\text{ID}(x', y')$. The IDs of all deleted or inserted edges are appended to the end of the label in the order $\text{ID}(x', y'), \text{ID}(y', z'), \text{ID}(z', x')$.

(5) Sort all labels lexicographically.

(6) Without loss of generality, let $L = (x', y', z', \text{ID}(x', y'))$ be the lexicographically-first of these triangle labels which contains at least one edge deletion (resp. edge insertion) of an edge that is incident to at least one low-degree vertex.

(7) If (u', v', w') corresponds to the lexicographically-first label L *and* $\text{ID}(u', v')$ is the first edge ID in the label that contains a low-degree vertex, then (u', v') performs the following steps:

(a) Count the number of unique triangles (using the labels, one can count the unique triangles) containing at least one edge deletion (resp. insertion) and at least one low-degree vertex as T_D (resp. T_I). We count using the generated labels for the triangles enumerated in Item (2) of this procedure.

(b) Return T_D (resp. T_I).

(8) If (u', v', w') is not equal to L *or* $\text{ID}(u', v')$ is not the first edge ID that contains a low-degree vertex in the label, return 0.

8.5.4 Analysis

In Theorem 8.5.4, we prove that the procedure correctly returns the exact number of k-cliques in G. The proof is similar to AYZ except that each ℓ-clique can appear multiple times in G' so we need to normalize by the constant stated in Item (**10**) of Algorithm 31.

Theorem 8.5.4. *Algorithm 31 correctly computes the exact number of cliques in a graph $G = (V, E)$ when $k \bmod 3 = 0$.*

Proof. We first show that all triangles in G' represent a k-clique in G. A vertex exists in G' if and only if it is a $(k/3)$-clique in G. Similarly, an edge exists in G' if and only if it connects two vertices in G' that form a $(2k/3)$-clique in G. Thus, a triangle connects 3 pairs of 3 distinct $(k/3)$-cliques. This implies that each pair represents a complete subgraph, which necessarily means by the pigeonhole principle that the triangle represents a k-clique. Now we show that for each unique k-clique in G, there exist exactly $\binom{k}{k/3}\binom{2k/3}{k/3}$ triangles representing it in G'. For each k-clique in G, there are $\binom{k}{k/3}$ distinct $(k/3)$-subcliques. Each of these subcliques is represented by a vertex in G'. Each distinct triple of subcliques will be a triangle in G'. There are $\binom{k}{k/3}$ ways to choose the first subclique, $\binom{2k/3}{k/3}$ ways to choose the second subclique, and $\binom{k/3}{k/3}$ ways to choose the third subclique in the triple. Thus, the total number of duplicate triangles is $\binom{k}{k/3}\binom{2k/3}{k/3}$.

We conclude by proving that our algorithm finds the exact number of triangles in G'. All triangles containing edge updates where at least one of its endpoints is low-degree can be found by searching all of the neighbors of the low-degree vertex. All such neighbors will be in $\mathcal{L}$, thus, searching through the entries in $\mathcal{L}$ is enough to find all triangles containing at least one low-degree vertex and an edge update to a low-degree vertex. By Lemma 8.5.3, all triangles with a low-degree vertex, containing a single edge update between high-degree vertices can be found via the `count_new_low_degree_triangles` procedure. The same logic handles vertices that change status from high-degree to low-degree, since we treat edges incident to these vertices as new edge insertions. Finally, the procedure ensures that no duplicate triangles are added to the update triangle count because the lexicographically first triangle counts all possible triangles representing the same clique (and no others increment the count). Table A is used to compute (via transitive closure) the number of triangles that contain no low-degree vertices. Thus, by computing A^3, we find the remaining triangles which only contain high-degree vertices. Finally, dividing by the total number of different triangles that are created per unique clique gives us the precise count of the number of k-cliques in G. □

Cost We now analyze the work, depth, and space of the dynamic algorithm. Our analysis assumes that $m^{\omega_p/(1+\omega_p)} = O(m^{t\ell})$ so that the $O(m^{t\ell})$ terms in our analysis are only affected by a constant factor for our batch size of $\mathcal{B} \leq m^{\omega_p/(1+\omega_p)}$. This is true for $k > 6$ because $t \geq 1/3$ and $\ell \geq 3\omega_p/(1+\omega_p)$. For small ℓ we use the combinatorial algorithm from Section 8.4, which is also faster.

First, we compute the work and depth bound of performing preprocessing on an initial graph $G = (V, E)$ with m edges. We can also apply this preprocessing directly without running the update algorithm whenever we receive a batch of size $\mathcal{B} > m^{\omega_p/(1+\omega_p)}$.

For preprocessing, we use a different threshold $m^{t'\ell}$ for low-degree and high-degree vertices. Searching for all the triangles containing at least one low-degree vertex takes $O\left(m^{(1+t')\ell}\right)$ work by a similar calculation as in Lemma 8.5.9 and searching for triangles containing all high-degree vertices takes $O\left(m^{(1-t')\ell\omega_p}\right)$ work by Lemma 8.5.10. Thus, the optimal value t' is when $m^{(1+t')\ell} = m^{(1-t')\ell\omega_p}$, which gives $t' = \frac{\omega_p - 1}{\omega_p + 1}$ as in [AYZ97].

Lemma 8.5.5. *Preprocessing the graph $G = (V, E)$ with m edges into G', creating the data structures $\mathcal{L}$, A, and $\mathcal{D}$, and counting the number of k-cliques takes $O\left(m^{\frac{2k\omega_p}{3(1+\omega_p)}}\right)$ work and $O(\log m)$ depth w.h.p., and $O\left(m^{\frac{2k\omega_p}{3(1+\omega_p)}}\right)$ space assuming a parallel matrix multiplication algorithm with coefficient ω_p. Using the fastest parallel matrix multiplication currently known ([LG14], Corollary 8.5.19), preprocessing takes $O\left(m^{0.469k}\right)$ work and $O(\log m)$ depth w.h.p., and $O(m^{0.469k})$ space.*

Proof. The graph G' has size $O(m^\ell)$ by Lemma 8.5.6. We can find all ℓ-cliques using $O(m^{\ell/2})$ work and $O(1)$ depth and all 2ℓ-cliques using $O(m^\ell)$ work and $O(1)$ depth. Initializing the data structures $\mathcal{L}$ and $\mathcal{D}$ with $O(m^\ell)$ entries requires insertions into two parallel hash tables. This takes $O(m^\ell)$ work and $O(\log^* m)$ depth w.h.p., and $O(m^\ell)$ space. There are $O\left(m^{\frac{2\ell}{(1+\omega_p)}}\right)$ high-degree vertices which means that initializing A, the adjacency matrix, requires creating a 2-level hash table with $O\left(m^{\frac{4\ell}{(1+\omega_p)}}\right)$ entries. This takes $O\left(m^{\frac{4\ell}{(1+\omega_p)}}\right)$ work and $O(\log^* m)$ depth w.h.p., and $O\left(m^{\frac{4\ell}{(1+\omega_p)}}\right)$ space. Computing A^3 requires $O\left(m^{\frac{2\ell\omega_p}{(1+\omega_p)}}\right)$ work, $O(\log m)$ depth, and $O\left(m^{\frac{2\ell\omega_p}{(1+\omega_p)}}\right)$ space. Finally, counting all the triangles with at least one low-degree vertex requires $O\left(m^{\frac{2\ell\omega_p}{(1+\omega_p)}}\right)$ work and $O(1)$ depth (by performing $O\left(m^{(1+t)\ell}\right)$ lookups in $\mathcal{L}$). By Corollary 8.5.19, $\omega_p = 2.373$, and since $\ell = k/3$, preprocessing takes $O\left(m^{0.469k}\right)$ work, $O(\log m)$ depth, and $O(m^{0.469k})$ space. □

Next, we analyze the update procedure of our dynamic algorithm. To start, we bound the number of vertices and edges in G' (representing the number of ℓ and 2ℓ cliques in G, respectively) in terms of m (the number of edges in G) below.

Lemma 8.5.6 ([CN85])**.** *Given a graph $G = (V, E)$ with m edges, the number of k-cliques that G can have is bounded by $O(m^{k/2})$.*

Lemma 8.5.7. *G' uses $O(m^\ell)$ space.*

Proof. Each vertex in G' represents an ℓ-clique. By Lemma 8.5.6, G' has $O(m^{\ell/2})$ vertices and thus $O(m^\ell)$ edges. □

Before we compute the number of triangles in G', we must update G' and the data structures associated with G' with our batch of updates.

Lemma 8.5.8. *Updating G' and the associated data structures $\mathcal{L}$ and A after a batch of $\mathcal{B}$ edge updates in G takes $O(\mathcal{B}m^{\ell-1} + \mathcal{B}m^{(2-2t)\ell-1})$ amortized work and $O(\log^* m)$ depth w.h.p., and $O\left(m^\ell + m^{(2-2t)\ell}\right)$ space.*

Proof. In Item (1) we first add and/or delete vertices in G'. Since each vertex in G' represents a different clique of size ℓ, one edge update in G can result in $O(m^{(\ell/2)-1})$ new vertices (or vertex deletions) since given two vertices (the endpoints of the edge update) that must be in the ℓ-clique, we only need to look for all $(\ell-2)$-cliques in G. For a batch of size $\mathcal{B}$, the total number of vertices added or deleted in G' is $O(\mathcal{B}m^{(\ell/2)-1})$.

In Item (5)a and Item (5)b, updating the data structures $\mathcal{L}$, A, and $\mathcal{D}$ by insertions/deletions into parallel hash tables requires $O(\mathcal{B}m^{\ell-1})$ amortized work and $O(\log^* m)$ depth w.h.p. Recall that the number of edges in G' is determined by the total number of 2ℓ-cliques in G. One edge update can affect at most $O(m^{\ell-1})$ 2ℓ-cliques in G, thus, given a $\mathcal{B}$-batch of edge updates in G, there will be $O(\mathcal{B}m^{\ell-1})$ edge updates in G', separated into a deletion batch $\mathcal{B}'_D$ and an insertion batch $\mathcal{B}'_I$.

We now analyze the cost for Item (5)c and Item (5)d. Adding/removing a row and column from A takes $O(m^{(1-t)\ell})$ amortized work. Since there are $O(m^{\ell-1})$ edge updates in G' per update in G, the total work for resizing is $O(m^{(2-t)\ell-1})$ per edge update in G. The work for adding/removing a vertex from $\mathcal{L}$ is $O(m^{t\ell})$, and since there are $O(m^{\ell-1})$ edge updates per update in G, the total work is $O(m^{(1+t)\ell-1})$ per update in G. We must have $\Omega(m^{t\ell})$ updates in G' before a vertex changes statuses (becomes high-degree if it originally was low-degree and vice versa) and needs to update A and $\mathcal{L}$. Therefore, we can charge the work of updating A and $\mathcal{L}$ against $\Omega(m^{t\ell})$ updates in G'. Thus, the amortized work for updating A and $\mathcal{L}$ given a batch of $\mathcal{B}$ updates in G is $O\left(\mathcal{B}\left(m^{(2-2t)\ell-1} + m^{\ell-1}\right)\right)$ for Item (1) and Item (5). The depth is $O(\log^* m)$ w.h.p. due to hash table operations.

The data structures $\mathcal{L}$, $\mathcal{D}$, and A use a combined $O(m^\ell + m^{(2-2t)\ell})$ space because there are $O(m^\ell)$ edges in the graph and A contains $O(m^{(2-2t)\ell})$ entries. □

By Lemma 8.5.8, Item (2) takes $O\left(\mathcal{B}m^{\ell-1}\right)$ amortized work to determine the final degrees and $O(\mathcal{B}m^{\ell-1} + \mathcal{B}m^{(2-2t)\ell-1})$ amortized work to compute $B'_{I,L}$ and $B'_{D,H}$. In total, Item (2) takes $O(\mathcal{B}m^{\ell-1} + \mathcal{B}m^{(2-2t)\ell-1})$ amortized work, $O(\log m)$ depth (dominated by computing the final degrees), and $O(m^\ell + m^{(2-2t)\ell})$ space by Lemma 8.5.8. Item (4), Item (7), Item (9), and Item (10) of the algorithm take $O(1)$ work. The following lemmas bound the cost for the remaining steps.

Lemma 8.5.9 below bounds the cost for Item (3) and Item (6). The proof is based on counting the number of new edge updates necessary in G'.

Lemma 8.5.9. *Computing all new k-cliques represented by triangles that contain at least one low-degree vertex in G' takes $O(\mathcal{B}m^{(t+1)\ell-1})$ work and $O(\log^* m)$ depth w.h.p., and $O(m^\ell)$ space.*

Proof. We first bound the work necessary to perform Item (3) and Item (6) for new edge insertions and deletions. Given one edge update in G, there can be at most $O(m^{\ell-1})$ edge

updates necessary in G' by Lemma 8.5.6. For each of these edge updates, we consider whether each edge update in G' contains a low-degree vertex. By Lemma 8.5.3 and Theorem 8.5.4, to find all updated triangles containing at least one low-degree vertex, it is only necessary to consider edge updates to low-degree vertices. For every edge update to a low-degree vertex, we search the neighbors of that low-degree vertex to see if new triangles are formed/destroyed. Since each low-degree vertex has degree $O(m^{t\ell})$, this results in a total of $O(m^{(t+1)\ell-1})$ work per update in G to perform the search. For each triangle found that contains the low-degree vertex, we need to perform the additional work of computing every triangle that contains the set of vertices represented by the triangle, sort the labels, and determine which triangle is responsible for incrementing the count of triangles by all $\binom{k}{k/3}\binom{2k/3}{k/3}$ triangles representing the same clique. This additional work is done by calling `count_updated_low_degree_triangles`$((u',v',w'),(u',v'))$ on each triangle (u',v',w') and each edge update (u',v'). The total amount of additional work done for each triangle that is passed into `count_updated_low_degree_triangles` is then $O\left(k(3e^2)^k\right)$, where the number of triangles corresponding to the same k-clique is given by $O\left((3e^2)^k\right)$ and an additional $O(k(3e^2)^k)$ work is required to sort all the labels. Since we assume that k is constant, this results in $O(1)$ additional work per call to `count_updated_low_degree_triangles`. The depth is $O(\log^* m)$ w.h.p. due to hash table lookups.

Now we bound the work of performing Item (3) and Item (6) for edges that are 'inserted' or 'deleted' due to rebalancing. Suppose there are X vertices that must be rebalanced in this way. Each of these X vertices must have degree $O(m^{t\ell})$ at the time of rebalancing. Thus, the total work performed for these updates is $O(Xm^{2t\ell})$. However, in order for a rebalancing on a vertex to happen, there must be $\Omega(m^{t\ell})$ updates. Thus, if X vertices are rebalanced, then there must be $\Omega(Xm^{t\ell})$ updates. Hence, we can charge the work of rebalancing to the $\Omega(Xm^{t\ell})$ updates to obtain $O(m^{t\ell})$ amortized work per update in G'. Then, we obtain $O(\mathcal{B}m^{(t+1)\ell-1})$ amortized work for a $\mathcal{B}$ batch updates to G. Rebalancing requires $O(\log^* m)$ depth w.h.p. due to hash table operations and $O(m^\ell)$ space (the total number of edges in the graph). □

Lemma 8.5.10 bounds the cost for Item (8) by using the matrix multiplication bounds for the adjacency matrix containing high-degree vertices.

Lemma 8.5.10. *Computing A^3 using parallel matrix multiplication takes $O(m^{(1-t)\ell\omega_p})$ work, where ω_p is the parallel matrix multiplication constant, $O(\log m)$ depth, and $O(m^{\omega_p(1-t)\ell})$ space, assuming that there exists a parallel matrix multiplication algorithm with coefficient ω_p and using $O(\log n)$ depth and $O(n^{\omega_p})$ space given $n \times n$ matrices.*

Proof. There are $O(m^{(1-t)\ell})$ high-degree vertices because each high-degree vertex has degree $\Omega(m^{t\ell})$ and there are $O(m^\ell)$ edges in G'. Since the table A is an adjacency matrix on the high-degree vertices, by Corollary 8.5.19, parallel matrix multiplication can be done in $O(m^{(1-t)\ell\omega_p})$ work.

□

Lemma 8.5.11 bounds the cost for Item (11). The proof is based on amortizing the cost for reconstruction over $\Omega(m)$ updates.

Lemma 8.5.11. *Item (11) requires $O(|\mathcal{B}|m^{(2-2t)\ell-1} + |\mathcal{B}|m^{\ell-1})$ amortized work and $O(\log^* m)$ depth w.h.p., and $O(m^{(2-2t)\ell} + m^{\ell})$ space.*

Proof. We reconstruct A from scratch, which has one entry for every pair of high-degree vertices, which takes $O(m^{2(1-t)\ell}) = O(m^{(2-2t)\ell})$ work and space. However, this is amortized against $\Omega(m)$ updates, and so the amortized work is $O(m^{(2-2t)\ell-1})$ per update. The work and space for creating $\mathcal{L}$ can be bounded by $O(m^{\ell})$, the number of edges in G'. Amortized against $\Omega(m)$ updates gives $O(m^{\ell-1})$ work per update. The depth is $O(\log^* m)$ w.h.p. using parallel hash table operations. □

Given these costs, we can now compute the optimal value of t in terms of ω_p that minimizes the work. Note that here we compute for t assuming $\mathcal{B} = 1$ because to adaptively change our threshold requires too much work in terms of rebalancing the data structures. However, if we have a fixed batch size, $\mathcal{B}$, we can further optimize our threshold t to take into account the fixed batch size.

Lemma 8.5.12. *$t = \frac{3-k+k\omega_p}{k+k\omega_p}$ gives us an optimal work bound assuming $\mathcal{B} = 1$.*

Proof. From Lemma 8.5.8, Lemma 8.5.9, Lemma 8.5.10, and Lemma 8.5.11, we have that the work is $O(\mathcal{B}m^{(t+1)\frac{k}{3}-1} + m^{\frac{(1-t)k\omega_p}{3}})$ w.h.p. (the $O(\mathcal{B}m^{(2-2t)l-1})$ term is dominated by the $O(\mathcal{B}m^{(1+t)l-1})$ term since $\omega_p \geq 2$ implies $t \geq 1/3$). Assuming $\mathcal{B} = 1$, balancing the two sides of the equation yields:

$$m^{\frac{(1-t)k\omega_p}{3}} = m^{(t+1)\frac{k}{3}-1}.$$

Solving for t gives

$$t = \frac{3-k+k\omega_p}{k+k\omega_p}.$$

□

Plugging in our value for t from Lemma 8.5.12, we prove Theorem 8.5.1 and Corollary 8.5.2 for the cost of our algorithm when $0 < m \leq m^{\omega_p/(1+\omega_p)}$.

8.5.5 Accounting for $k \bmod 3 \neq 0$

We now modify the algorithm above to account for all values k following the algorithm presented in [EG04]. This requires several changes to how we construct our graph G' from a graph $G = (V, E)$, resulting in changes to our data structures which we detail below. We recall the notation $R(x)$ for vertex $x \in G'$ to denote the vertices in G that x represents.

Construction of G'

For $k \bmod 3 \neq 0$, the fundamental problem we face in this case in constructing the graph G' is that triangles in the graph G' representing cliques of size $\lfloor \frac{k}{3} \rfloor$ no longer create k-cliques. In fact, they now create $(k-1)$-cliques or $(k-2)$-cliques for $k \bmod 3 = 1$ and $k \bmod 3 = 2$, respectively. We modify the creation of G' in the two following ways to account for this issue:

$k \bmod 3 = 1$: In this case, we create two sets of vertices. One set, A, of vertices represents all $\left(\frac{k-1}{3}\right)$-cliques in the graph G. Edges exist between $v_1, v_2 \in A$ if and only if the vertices, $R(v_1)$ and $R(v_2)$, in the $\left(\frac{k-1}{3}\right)$-cliques represented by v_1 and v_2 form a $\frac{2(k-1)}{3}$ clique and there are no duplicate vertices, i.e., $R(v_1) \cap R(v_2) = \varnothing$. We create a second set of vertices B which contains vertices which represent cliques of size $\frac{k+2}{3}$. Edges exist between $v \in A$ and $w \in B$ if and only if $R(v)$ and $R(w)$ form a $\left(\frac{2k+1}{3}\right)$-clique and $R(v) \cap R(w) = \varnothing$.

$k \bmod 3 = 2$: In this case, we still create two sets of vertices but A instead represents $\left(\frac{k+1}{3}\right)$-cliques in the graph G. Edges exist between $v_1, v_2 \in A$ if and only if $R(v_1) \cup R(v_2)$ form a $\left(\frac{2(k+1)}{3}\right)$-clique and $R(v_1) \cap R(v_2) = \varnothing$. We create a second set of vertices B which contains vertices which represent cliques of size $\frac{k-2}{3}$. Edges exist between $v \in A$ and $w \in B$ if and only if $R(v)$ and $R(w)$ form a $\left(\frac{2k-1}{3}\right)$-clique and $R(v) \cap R(w) = \varnothing$.

We first prove the properties the new graph G' has, namely the number of vertices it contains as well as the number of edges in the graph.

Lemma 8.5.13. *G' constructed as in Section 8.5.5 contains $O\left(m^{\frac{k+2}{6}}\right)$ vertices and $O\left(m^{\frac{2k+1}{6}}\right)$ edges if $k \bmod 3 = 1$. G' contains $O\left(m^{\frac{k+1}{6}}\right)$ vertices and $O\left(m^{\frac{k+1}{3}}\right)$ edges if $k \bmod 3 = 2$.*

Proof. When $k \bmod 3 = 1$, the number of vertices is upper bounded (asymptotically) by the number of $\left(\frac{k+2}{3}\right)$-cliques in the graph. By Lemma 8.5.6, the number of vertices is then bounded by $O\left(m^{\frac{k+2}{6}}\right)$. The number of edges is bounded by the number of $\left(\frac{2k+1}{3}\right)$-cliques in the graph which is $O\left(m^{\frac{2k+1}{6}}\right)$. Similarly, when $k \bmod 3 = 2$, by Lemma 8.5.6, the number of vertices and edges are bounded by $O\left(m^{\frac{k+1}{6}}\right)$ and $O\left(m^{\frac{k+1}{3}}\right)$, respectively. □

Data Structure and Algorithm Changes

The major data structure change is to redefine the high-degree and low-degree vertices in terms of the number of edges in the graph. This means that low-degree is defined as having a degree less than $\frac{M^t\left(\frac{2k+1}{6}\right)}{2}$ and high-degree as greater than $\frac{3M^t\left(\frac{2k+1}{6}\right)}{2}$ for the $k \bmod 3 = 1$ case; similarly we define low-degree to be less than $\frac{M^t\left(\frac{k+1}{3}\right)}{2}$ and high-degree to be greater than $\frac{3M^t\left(\frac{k+1}{3}\right)}{2}$ for the $k \bmod 3 = 2$ case.

Another key difference between this case and the case when k is divisible by 3 is that the number of duplicate cliques is different for these two cases. For the $k \bmod 3 = 1$ case, each k-clique in G will be represented by $\binom{k}{(k+2)/3}\binom{(2k-2)/3}{(k-1)/3}$ triangles found by the algorithm. For the $k \bmod 3 = 2$ case, each k-clique in G will be represented by $\binom{k}{(k-2)/3}\binom{(2k+2)/3}{(k+1)/3}$ triangles. Thus, at the end of our algorithm, we must divide the count of the triangles by their respective number of duplicates.

The rest of the algorithm remains the same as before, except that we solve for different values of t depending on the case. Since the proofs for obtaining the following results are

nearly identical to the ones for $k \bmod 3 = 0$, we do not restate the proofs and only give our results.

Lemma 8.5.14. *For the case when $k \bmod 3 = 1$, there exists $O\left(m^{\frac{2k+1}{6}}\right)$ edges in the graph and solving for the optimal value of t (assuming $\mathcal{B} = 1$) gives $t = \frac{2k\omega_p - 2k + \omega_p + 5}{2k\omega_p + 2k + \omega_p + 1}$. For the case when $k \bmod 3 = 2$, there exists $O\left(m^{\frac{k+1}{3}}\right)$ edges in the graph and solving for the optimal value of t gives $t = \frac{k\omega_p - k + \omega_p + 2}{k\omega_p + k + \omega_p + 1}$.*

Using our values for t, we can obtain our final theorem, Theorem 8.5.15, for the work and depth bounds for these two cases.

Theorem 8.5.15. *Our fast matrix multiplication based k-clique algorithm takes $O\left(\min\left(\mathcal{B}m^{\frac{2(k-1)\omega_p}{3(\omega_p+1)}}, (\mathcal{B}+m)^{\frac{(2k+1)\omega_p}{3(\omega_p+1)}}\right)\right)$ work and $O(\log(m+\mathcal{B}))$ depth w.h.p., and $O\left((\mathcal{B}+m)^{\frac{(2k+1)\omega_p}{3(\omega_p+1)}}\right)$ space assuming a parallel matrix multiplication algorithm with co-efficient ω_p when $k \bmod 3 = 1$, and $O\left(\min\left(\mathcal{B}m^{\frac{(2k-1)\omega_p}{3(\omega_p+1)}}, (\mathcal{B}+m)^{\frac{2(k+1)\omega_p}{3(\omega_p+1)}}\right)\right)$ work and $O(\log(m+\mathcal{B}))$ depth w.h.p., and $O\left((\mathcal{B}+m)^{\frac{2(k+1)\omega_p}{3(\omega_p+1)}}\right)$ space when $k \bmod 3 = 2$.*

Corollary 8.5.16. *Using Corollary 8.5.19 with $\omega_p = 2.373$, we obtain a parallel fast matrix multiplication k-clique algorithm that takes $O\left(\min\left(\mathcal{B}m^{0.469k-0.469}, (\mathcal{B}+m)^{0.469k+0.235}\right)\right)$ work and $O(\log m)$ depth w.h.p., and $O\left((\mathcal{B}+m)^{0.469k+0.235}\right)$ space when $k \bmod 3 = 1$, and $O\left(\min\left(\mathcal{B}m^{0.469k-0.235}, (\mathcal{B}+m)^{0.469k+0.469}\right)\right)$ work and $O(\log m)$ depth w.h.p., and $O\left((\mathcal{B}+m)^{0.469k+0.469}\right)$ space when $k \bmod 3 = 2$.*

8.5.6 Parallel Fast Matrix Multiplication

In this section, we show that tensor-based matrix multiplication algorithms (including Strassen's algorithm) can be parallelized in $O(\log n)$ depth and $O(n^\omega)$ work. Such techniques are used for algorithms that achieve the best currently known matrix multiplication exponents [Wil12, LG14]. We assume, as is common in models such as the arithmetic circuit model, that field operations can be performed in constant work. We refer readers interested in learning more about current techniques in fast matrix multiplication to [Blä13, Alm19].

Before we prove our main parallel result in this section, we first define the *matrix multiplication tensor* as used in previous literature.

Definition 8.5.17 (Matrix Multiplication Tensor (see, e.g., [Alm19]))**.** *For positive integers a, b, c, the matrix multiplication tensor $\langle a, b, c\rangle$ is a tensor over*

$\left\{x_{ij}\right\}_{i\in[a],j\in[b]}, \left\{y_{jk}\right\}_{j\in[b],k\in[c]}, \{z_{ki}\}_{k\in[c],i\in[a]}$, *where*

$$\langle a,b,c\rangle = \sum_{i=1}^{a}\sum_{j=1}^{b}\sum_{k=1}^{c} x_{ij}y_{jk}z_{ki}.$$

The matrix multiplication tensor can be seen as a generating function for $A\times B$ multiplication where the coefficients of the z_{ki} terms are exactly the (i,k) entries in the matrix product $A\times B$ where $A=\begin{pmatrix} x_{11} & \dots & x_{1b} \\ \dots & \dots & \dots \\ x_{a1} & \dots & x_{ab}\end{pmatrix}$ and $B=\begin{pmatrix} y_{11} & \dots & y_{1c} \\ \dots & \dots & \dots \\ y_{b1} & \dots & y_{bc}\end{pmatrix}$.

Current matrix multiplications algorithms use this fact to obtain the best known exponents. The proof of the following lemma closely follows the proof of Proposition 4.1 given in [Alm19].

Lemma 8.5.18. *Let $R(\langle q,q,q\rangle)\le r$ (over a field $\mathbb{F}$) be the rank of the matrix multiplication tensor $\langle q,q,q\rangle$. Assuming that field operations take $O(1)$ work, then, there exists a parallel matrix multiplication algorithm that performs $A\times B$ matrix multiplication (where $A,B\in\mathbb{F}^{n\times n}$) over $\mathbb{F}$ using $O\left(n^{\log_q(r)}\right)$ work and $O((\log r+\log q)\log_q n)$ depth using $O\left(n^{\log_q(r)}\right)$ space.*

Proof. By definition of rank, since $R(\langle q,q,q\rangle)\le r$,

$$\langle q,q,q\rangle = \sum_{\ell=1}^{r}\left(\sum_{i,j\in[q]} a_{ij\ell}x_{ij}\right)\left(\sum_{j,k\in[q]} b_{jk\ell}y_{jk}\right)\left(\sum_{k,i\in[q]} c_{ki\ell}z_{ki}\right)$$

for some coefficients $a_{ij\ell}, b_{jk\ell}, c_{ki\ell}\in\mathbb{F}$. Computing this matrix multiplication tensor requires at most $O\left(rq^2\right)$ field operations.

Using this information, we perform parallel matrix multiplication via the following recursive algorithm. We assume that n is a power of q; otherwise, we can pad A and B with 0's until such a condition is satisfied–this would increase the dimensions by at most a factor of q.

Partition the padded matrices A and B into $q\times q$ block matrices where each block has size $n/q\times n/q$. This algorithm performs, in parallel, the following linear combinations for each ℓ,

$$A'_\ell = \sum_{i,j\in[q]} a_{ij\ell}A_{ij}$$
$$B'_\ell = \sum_{j,k\in[q]} b_{jk\ell}B_{jk}$$

where A_{ij} and B_{jk} are the $n/q\times n/q$ blocks in A and B, respectively. Such operations require $O(rq^2)$ operations to perform; however, all such multiplication operations can be

done in parallel, and the summation of the results can be done in $O(\log q)$ depth, resulting in $O(\log q)$ depth.

Then, for each $\ell \in [r]$, we compute $C'_\ell = A'_\ell \times B'_\ell$ by performing parallel $n/q \times n/q$ matrix multiplication recursively on A'_ℓ and B'_ℓ where the base case is $q \times q$ matrix multiplication. All field operations in the same level of the recursion can be performed in parallel. There are $O(\log_q n)$ levels of recursion. Each level of recursion computes a number of field operations in parallel in $O(\log q)$ depth as in the top level.

Finally, after obtaining the results C'_ℓ of the recursive calls, we compute

$$C_{ki} = \sum_{\ell \in [r]} c_{ki\ell} C'_{\ell,ki}$$

for all $k, i \in [q]$ where $C'_{\ell,ki}$ are the results we obtain from our recursive calls. The blocks C_{ki} for all $k, i \in [q]$ are the results of our matrix multiplication $A \times B$.

This final step can compute in parallel the blocks C_{ki} for all $k, i \in [q]$ in $O(\log r)$ depth (assuming that we have the results $C'_{\ell,ki}$) since the multiplication operations can be done in parallel and the summation of the elements in the resulting matrices can be done in $O(\log r)$ depth.

Thus, the depth required for this algorithm is $O((\log r + \log q) \log_q n)$.

To compute the work and space usage, we compute the total number of field operations performed, which is $O(n^2)$ per level of the recursion. For each level of recursion, there are r calls per subproblem of the recursion. Since we assume that each field operation is $O(1)$ work, this results in total work given by

$$W(n) = r \cdot W(n/q) + O(n^2).$$

Solving the recurrence gives $W(n) = O\left(n^{\log_q r}\right)$ work for the entire algorithm. The space usage is also $O\left(n^{\log_q r}\right)$. □

Using Lemma 8.5.18, we obtain the following parallel matrix multiplication bounds:

Corollary 8.5.19. *There exists a parallel matrix multiplication algorithm based on [Wil12, LG14] that multiplies two $n \times n$ matrices with $O\left(n^{2.373}\right)$ work and $O(\log n)$ depth, using $O\left(n^{2.373}\right)$ space.*

8.6 Experimental Results

Experimental Setup Our experiments are performed on a 72-core Dell PowerEdge R930 (with two-way hyper-threading) with 4 × 2.4GHz Intel 18-core E7-8867 v4 Xeon processors (with a 4800MHz bus and 45MB L3 cache) and 1TB of main memory. Our programs use a work-stealing scheduler that we implemented [BAD20]. The scheduler is implemented similarly to Cilk for parallelism. Our programs are compiled using `g++` (version 7.3.0) with the `-O3` flag.

Graph Dataset	Num. Vertices	Num. Edges
Orkut	3,072,627	234,370,166
Twitter	41,652,231	2,405,026,092
rMAT	16,384	121,362,232

Table 8.1: Graph inputs, including number of vertices and edges.

m	unique edges	m	unique edges
2×10^6	1,569,454	4×10^8	55,395,676
2×10^7	9,689,644	8×10^8	74,698,492
1×10^8	27,089,362	3.2×10^9	121,362,232
2×10^8	39,510,764		

Table 8.2: Number of unique edges in the first m edges from the *rMAT* generator.

Graph Data Table 8.1 lists the graphs that we use. ***com-Orkut*** is an undirected graph of the Orkut social network [LS16]. ***Twitter*** is a directed graph of the Twitter network [KLPM10]. We symmetrize the Twitter graph for our experiments. For some of our experiments which ingest a stream of edge updates, we sample edges from an rMAT generator [CZF04] with $a = 0.5, b = c = 0.1, d = 0.1$ to perform the updates. The update stream can have duplicate edges, and Table 8.2 reports the number of unique edges found in prefixes of various sizes of the rMAT stream that we generate. The unique edges in the full stream represents the rMAT graph described in Table 8.1.

8.6.1 Our Implementation

Parallel Primitives We implemented a multicore CPU version of our algorithm using the Graph Based Benchmark Suite (GBBS) [DBS18b], which includes a number of useful parallel primitives, including high-performance parallel sorting, and primitives such as prefix sum, reduce, and filter [Jaj92]. In what follows, a ***filter*** takes an array A and a predicate function f, and returns a new array containing $a \in A$ for which $f(a)$ is true, in the same order that they appear in A. Our implementations use the atomic compare-and-swap and atomic-add instructions available on modern CPUs.

Implementation For $\mathcal{T}$, we used the concurrent linear probing hash table by Shun and Blelloch [SB14]. For each of the data structures $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$, we created an array of size n, storing (possibly null) pointers to hash tables [SB14]. For an edge (u, v) in one of the data structures, the value v will be stored in the hash table pointed to by the u'th slot in the array. We also tried using hash tables for both levels, but found it to be slower in practice. For deletions, we used the folklore *tombstone* method. In this method, when an element is deleted, we mark the slot in the table as a tombstone, which is a special value. When inserting, we can insert into a tombstone, but we have to first check until seeing an empty slot to make sure that we are not inserting a duplicate key. In the preprocessing phase of the algorithm, instead of using approximate compaction, we used filter. To find the last update for duplicate updates, we use a parallel sample sort [SBF+12]

Algorithm	Graph	Batch Size 2×10^3	2×10^4	2×10^5	2×10^6	m
Ours (INS)	Orkut	1.90e-3	4.76e-3	0.0235	0.168	–
	Twitter	**2.11e-3**	**7.10e-3**	**0.0430**	**0.366**	–
	rMAT	**6.42e-4**	**2.09e-3**	**8.62e-3**	0.0618	–
Makkar et al. (INS) [MBG17]	Orkut	**9.76e-4**	**2.69e-3**	**0.0143**	**0.0830**	–
	Twitter	time-out	0.0644	0.437	3.88	–
	rMAT	1.98e-3	6.90e-3	0.012	**0.0335**	–
Ours (DEL)	Orkut	1.80e-3	4.37e-3	0.0189	0.124	–
	Twitter	**2.14e-3**	**7.76e-3**	**0.0486**	**0.385**	–
	rMAT	6.48e-4	2.23e-3	9.21e-3	0.0723	–
Makkar et al. (DEL) [MBG17]	Orkut	**4.63e-4**	**1.46e-3**	**8.12e-3**	**0.0499**	–
	Twitter	time-out	0.0597	0.401	3.64	–
	rMAT	**4.47e-4**	**1.81e-3**	**5.12e-3**	**0.027**	–
Static [ST15]	Orkut	–	–	–	–	1.027
	Twitter	–	–	–	–	32.1
	rMAT	–	–	–	–	14.7

Table 8.3: Running times (seconds) for our parallel batch-dynamic triangle counting algorithm and Makkar et al. [MBG17]'s algorithm on 72 cores with hyper-threading. We apply the edges in each graph as batches of edge insertions (INS) or deletions (DEL) of varying sizes, ranging from 2×10^3 to 2×10^6, and report the average time for each batch size. The update time of Makkar et al. algorithm for Twitter batch size 2×10^3 is missing because the expriment timed out. We also report the update time for the state-of-the-art static triangle counting algorithm of Shun and Tangwongsan [ST15], which processes a single batch of size m. Note that for the Twitter and Orkut datasets, all of the edges are unique. However, for the rMAT dataset, batches can have duplicate edges. For each batch size of each dataset, we list the fastest time in bold.

to sort the edges first by both endpoints, and then by timestamp. Then we use filter to remove duplicate updates. When we initialize the dynamic data structures, a vertex is considered high-degree if it has degree greater than $2t_1$ and low-degree otherwise.

During minor rebalancing, a vertex only changes its status if its degree drops below t_1 or increases above t_2 due to the batch update. In major rebalancing, we merge our dynamic data structure and the updated edges into a compressed sparse row (CSR) format graph and use the static parallel triangle counting algorithm by Shun and Tangwongsan [ST15] to recompute the triangle count. We then build a new dynamic data structure from the CSR graph. We also implement several natural optimizations which improve performance. To reduce the overhead of using hash tables, we use an array to store the neighbors of vertices with degree less than a certain threshold (we used 128 in our experiments). Moreover, we only keep a single entry for (u,v) and (v,u) in the wedges table $\mathcal{T}$.

Experiments Table 8.3 report the parallel running times on varying insertion and deletion batch sizes for our implementation of our new parallel batch-dynamic triangle counting algorithm designed. For the two graphs based on static graph inputs (Orkut and Twitter), we generate updates for the algorithm by representing the edges of the graph as an

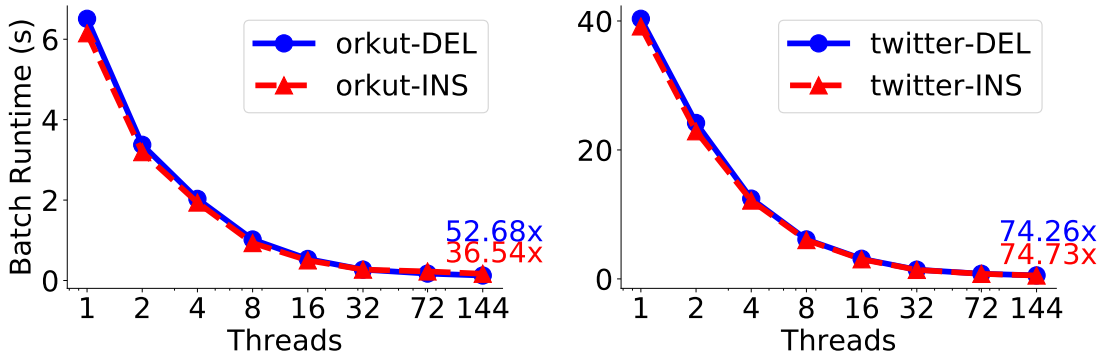


Figure 8-1: Running times of our parallel batch-dynamic triangle counting algorithm with respect to thread count (the x-axis is in log-scale) on the Orkut (average time across all batches) and Twitter (running time for the 6th batch) graph for both insertion (red dashed line) and deletion (blue solid line). "144" indicates 72 cores with hyper-threading. The experiment is run with a batch size of 2×10^6. The parallel speedup on 144 threads over a single thread is displayed.

array, and randomly permuting them. The algorithm is then run using batches of the specified size. For insertions, we start with an empty graph and apply batches from the beginning to the end of the permuted array. For deletions, we start with the full graph and apply batches from the end to the beginning of the permuted array. The table also reports the running time for the GBBS implementation of the state-of-the-art static triangle counting algorithm of Shun and Tangwongsan [ST15, DBS18b].

Across varying batch sizes, our algorithm achieves throughputs between 1.05–16.2 million edges per second for the Orkut graph, 0.935–5.46 million edges per second for the Twitter graph, and 3.08–32.4 million edges per second for the rMAT graph. We obtain much higher throughput for the rMAT graph due to the large number of duplicate edges found in this graph stream, as illustrated in Table 8.2. We observe that in all cases, the average time for processing a batch is smaller than the running time of the static algorithm. The maximum speedup of our algorithm over the static algorithm is 22709× for the rMAT graph with a deletion batch of size 2×10^3, but in general our algorithm achieves good speedups across the entire range of batches that we evaluate.

Lastly, Fig. 8-1 shows the parallel speedup of our algorithm with varying thread-count on the Orkut and Twitter graph, for a fixed batch size of 2×10^6. Our algorithm achieves a maximum of 74.73× speedup using 72 cores with hyper-threading for this experiment.

8.6.2 Comparison with Existing Algorithms

Comparison with Ediger et al We compared our implementation with a shared-memory implementation of the Ediger et al. algorithm [EJRB10], which is implemented as part of the STINGER dynamic graph processing system [EMRB12]. Unfortunately, we found that their implementation is much slower than ours due to bottlenecks in the update time for the underlying dynamic graph data structure. We note that recent work

on streaming graph processing observed similar results for using STINGER [DBS19]. To obtain a fair comparison, we chose to focus on implementing a more recent GPU batch-dynamic triangle counting algorithm ourselves, which we discuss next.

Comparison with Makkar et al The Makkar et al. algorithm [MBG17] is a state-of-the-art parallel batch-dynamic triangle counting implementation designed for GPUs. To the best of our knowledge, there is no multicore implementation of this algorithm, and so in this chapter we implement an optimized multicore version of their algorithm. The algorithm works as follows. First, their algorithm separates the batch of updates into batches for insertions and deletions. Then, for each batch of updates, it creates an *update graph*, $\hat{G}$, for each batch consisting of only the updates within each batch. Then, it merges the updates from each batch with the original edges in the graph to create an updated graph for each of the batches, G'. Note that this graph contains both the edges previously in the graph, as well as the new edges.

The merging process to construct G' first sorts the batch to obtain sorted lists of neighbors to add/delete from the adjacency lists of vertices in the graph. Then, the algorithm performs a simple linear-work procedure to merge each existing adjacency list with the sorted updates. In particular, doing t edge updates on a vertex with degree d takes $O(d+t)$ work. Finally, the algorithm counts the triangles by intersecting the adjacency lists of the endpoints of each edge in the batch. For each edge (u, v), they intersect $G'(u)$ with $G'(v)$, $G'(u)$ with $\hat{G}(v)$, and $\hat{G}(u)$ with $\hat{G}(v)$. The count of the number of triangles can be obtained from the number of intersections obtained from each of these cases using a simple inclusion-exclusion formula. They provide a further optimization by only intersecting *truncated* adjacency lists in some of the cases where a truncated adjacency list is one where the list only contains vertices with IDs less than the ID of the vertex that the adjacency list belongs to. Their algorithm has a worst case work bound of $O(n^2)$.

Implementation We developed a new multicore implementation of the Makkar et al. algorithm using the same parallel primitives and framework described earlier for the implementation of our algorithm. We implemented several optimizations that improved performance. First, we handle vertices with degree lower than 16 by storing their incident edges in a special array of size $16n$, and only allocate memory for vertices with larger degree. Second, we note that their algorithm does not specify how to handle redundant insertions that are already present in the graph. We remove these edge updates by modifying the merge algorithm that constructs G' from G. Specifically, during the merge, if we identify that a given edge is already present in G, we mark it in the sorted sequence of batch updates that we are merging in. Removing these marked updates to construct $\hat{G}$ without redundant updates is done by using a parallel filter.

Performance Comparison Table 8.3 shows the running times of the Makkar et al. algorithm on batches of insertions and deletions of different sizes. The data points for the Twitter graph are also plotted in Fig. 8-2. We observe that the Makkar et al. algorithm is faster than our algorithm on the Orkut graph, especially for large batches. On the other hand, for the Twitter graph, our algorithm is consistently faster for both insertions and

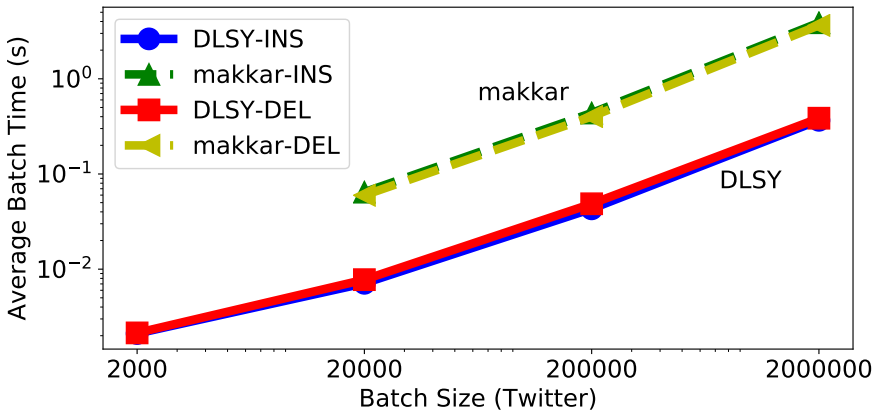


Figure 8-2: This figure plots the average insertion and deletion round times for each batch size (log-log scale) on Twitter using 72 cores with hyper-threading. The plot is in log-log scale. The lines for our algorithm are solid (blue for insertion and red for deletion) while the lines for Makkar et al. algorithm are dashed (green for insertion and yellow for deletion). The update time of Makkar et al. algorithm for Twitter batch size 2×10^3 is missing because the experiment timed out (due to cumulative runtime being too large).

deletions across all batch sizes. This is because there are no vertices with very high degree in the Orkut graph, and so the Makkar et al. algorithm does less work in merging adjacency lists with updates, while the Twitter graph has vertices with extremely high degree, which are costly to merge. Both algorithms are significantly faster than simply applying the static triangle counting algorithm for the range of batch sizes that we considered.

Next, we evaluate the performance of insertion batches in our algorithm and the Makkar et al. algorithm on the synthetic rMAT graph with 3.2 billion generated edges (which have duplicates). This synthetic experiment allows us to study how both algorithms perform as the graph becomes more dense. We evaluate the performance for different insertion batch sizes. The experiment uses prefixes of the rMAT graph (the number of unique edges per prefix is shown in Table 8.2) to control the density of the graph. The vertex set in this experiment is fixed, and thus a larger number of unique edges corresponds to a denser graph.

Fig. 8-3 plots the running time of both implementations for varying batch sizes as a function of the graph density. We observe that for small batch sizes, the performance of the Makkar et al. algorithm degrades significantly as the graph grows more dense and contains more high-degree vertices. On the other hand, our algorithm's performance generally does not degrade as the graph grows denser, across all batch sizes. We also significantly outperform the Makkar et al. algorithm for small batch sizes. Specifically, we obtain a maximum speedup of 3.31× for a batch of size 2×10^4. This is because the overhead of updating of high-degree vertices in the Makkar et al. algorithm becomes relatively higher, as work proportional to the vertex degree must be done regardless of the number of new incident edges.

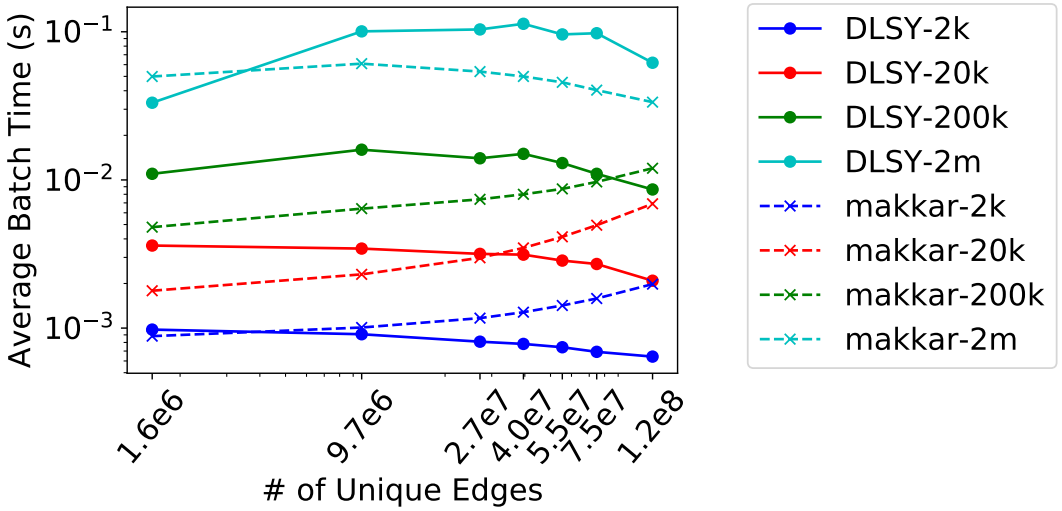


Figure 8-3: Comparison of the performance of our implementation (DLSY, solid line) and Makkar et al. algorithm [MBG17] (makkar, dotted line) for batches of insertions. The figure shows the average batch time for different batch sizes on the rMAT graph with varying prefixes of the generated edge stream to control density. The number of unique edges in the prefix is shown on the x-axis. The number of vertices is fixed at 16,384. The dark blue, red, green, and light blue lines are for batches of size 2×10^3, 2×10^4, 2×10^5, and 2×10^6, respectively. We see that our new algorithm is faster for small batches and on denser graphs.

8.7 Open Questions

There are a number of theoretical and practical questions resulting from our triangle and clique counting results:

1. For bounded arboricity graphs, we give a $O(|\mathcal{B}|(m+|\mathcal{B}|)\alpha^{k-4})$ expected work and $O(\log^2 n)$ depth *whp*, and $O(m+|\mathcal{B}|)$ space algorithm. Can we do better in terms of work and/or depth?
2. Can we apply the techniques used to obtain our batch-dynamic $O(\sqrt{m})$ amortized work, $O(1)$ depth triangle counting algorithm to larger cliques, to obtain better amortized work for e.g., 4-cliques or 5-cliques?
3. We did not implement our matrix multiplication based clique counting algorithm so we do not know how well it performs in real dense networks. It would be interesting to test whether it performs better than the trivial combinatorial algorithm for counting larger cliques dynamically in dense networks such as the neuronal network.

8.8 Conclusion

In this chapter, we have given new dynamic algorithms for the k-clique problem. We study this fundamental problem in the batch-dynamic setting, which is better suited for parallel hardware that is widely available today, and enables dynamic algorithms to scale to high-rate data streams. We have presented a work-efficient parallel batch-dynamic triangle counting algorithm. We also gave a simple, enumeration-based algorithm for maintaining the k-clique count. In addition, we have presented a novel parallel batch-dynamic k-clique counting algorithm based on fast matrix multiplication, which is asymptotically faster than existing dynamic approaches on dense graphs. Finally, we provide a multicore implementation of our parallel batch-dynamic triangle counting algorithm and compare it with state-of-the-art implementations that have weaker theoretical guarantees, showing that our algorithm is competitive in practice.

Acknowledgements.

We thank Josh Alman, Virginia Vassilevska Williams, and Nicole Wein for helpful discussions on various aspects of our paper.

Part IV

Hardness from Pebbling

Overview

You may have heard the following puzzle: what English word contains four consecutive letters that are consecutive letters of the alphabet? [What are words containing the following letters (consecutively)] RAOR, XS, DQ, HCR, XOP, and BEK?
Mathematical Mind-Benders [Win07] [13]

This part of the thesis presents lower bound results that are obtained using a combinatorial game on directed acyclic graphs, known as a *pebble game*.

Complexity of Computing the Trade-Off between Cache Size and Memory Transfers The red-blue pebble game was formulated in the 1980s [JWK81] to model the I/O complexity of algorithms on a two-level memory hierarchy. Given a directed acyclic graph representing computations (vertices) and their dependencies (edges), the red-blue pebble game allows sequentially adding, removing, and recoloring red or blue pebbles according to a few rules, where red pebbles represent data in cache (fast memory) and blue pebbles represent data on disk (slow, external memory). Specifically, a vertex can be newly pebbled red if and only if all of its predecessors currently have a red pebble; pebbles can always be removed; and pebbles can be recolored between red and blue (corresponding to reading or writing data between disk and cache, also called I/Os or memory transfers). Given an upper bound on the number of red pebbles at any time (the cache size), the goal is to compute a game execution with the fewest pebble recolorings (memory transfers) that finish with pebbles on a specified subset of nodes (outputs get computed).

In this chapter, we investigate the complexity of computing this trade-off between red-pebble limit (cache size) and number of recolorings (memory transfers) in general DAGs. First we prove this problem PSPACE-complete through an extension of the proof of the PSPACE-hardness of black pebbling [GLT80]. Second, we consider a natural restriction on the red-blue pebble game to forbid pebble deletions, or equivalently, forbid discarding data from cache without first writing it to disk. This assumption both simplifies the model and immediately places the trade-off computation problem within NP. Unfortunately, we show that even this restricted version is NP-complete. Finally, we show that the trade-off problem parameterized by the number of transitions is W[1]-hard, meaning that there is likely no algorithm running in a fixed polynomial for constant number of transitions.

Specifically, in Chapter 9, we show:

- Generalized red-blue no-deletion pebble game on a DAG with maximum in-degree 2 is NP-Complete. [Theorem 9.2.1]

[13] Arguably, linguistics puzzles are marginally related to cryptography.

- Determining the minimum pebbling cost and number of transitions is PSPACE-complete (even given constant number of transitions) to compute in the red-blue pebble game. [Theorem 9.2.4]
- The red-blue pebble game parameterized by the number of transitions k is W[1]-hard. [Theorem 9.4.12]

Follow-up Work Recently, a follow-up work by Papp and Wattenhofer [PW20] study a variety of additional variants of the red-blue pebble game and prove their hardness. Notably, they show the *one-shot* version as well as a novel variant which they call Compcost to be NP-hard. The Compcost model assumes that computation *within cache* also has some cost; thus, this model may be a more accurate representation of external-memory than a model which assumes computation in cache have no cost.

Static-Memory-Hard Hash Functions from Pebbling A series of recent research starting with (Alwen and Serbinenko, STOC 2015) has deepened our understanding of the notion of *memory-hardness* in cryptography — a useful property of hash functions for deterring large-scale password-cracking attacks — and has shown memory-hardness to have intricate connections with the theory of graph pebbling. Definitions of memory-hardness are not yet unified in the somewhat nascent field of memory-hardness, however, and the guarantees proven to date are with respect to a range of proposed definitions. In this chapter, we observe *two* significant and practical considerations that are not analyzed by existing models of memory-hardness, and propose new models to capture them, accompanied by constructions based on new hard-to-pebble graphs. Our contribution is two-fold, as follows.

First, existing measures of memory-hardness only account for *dynamic* memory usage (i.e., memory read/written at runtime), and do not consider *static* memory usage (e.g., memory on disk). Among other things, this means that memory requirements considered by prior models are inherently upper-bounded by a hash function's runtime; in contrast, counting static memory would potentially allow quantification of much larger memory requirements, decoupled from runtime. We propose a new definition of *static-memory-hard* function (SHF) which takes static memory into account: we model static memory usage by oracle access to a large preprocessed string, which may be considered part of the hash function description. Static memory requirements are *complementary* to dynamic memory requirements: neither can replace the other, and to deter large-scale password-cracking attacks, a hash function will benefit from being *both* dynamic-memory-hard and static-memory-hard. We give two SHF constructions based on pebbling. To prove static-memory-hardness, we define a new pebble game ("*black-magic pebble game*"), and new graph constructions with optimal complexity under our proposed measure. Moreover, we provide a prototype implementation of our first SHF construction (which is based on pebbling of a simple "cylinder" graph), providing an initial demonstration of practical feasibility for a limited range of parameter settings.

Secondly, existing memory-hardness models implicitly assume that the cost of space and time are more or less on par: they consider only *linear* ratios between the costs of time and space. We propose a new model to capture *nonlinear* time-space trade-offs: e.g.,

how is the adversary impacted when space is quadratically more expensive than time? We prove that nonlinear tradeoffs can in fact cause adversaries to employ different strategies from linear tradeoffs.

Finally, as an additional contribution of independent interest, we present an asymptotically tight graph construction that achieves the best possible space complexity up to $\log\log n$-factors for an existing memory-hardness measure called *cumulative complexity* in the sequential pebbling model.

Specially, in Chapter 10, we show:

- There exists a family of graphs where the cumulative complexity of any graph with n nodes in the family is $\Theta\left(\frac{n^2 \log\log n}{\log n}\right)$ which is asymptotically tight to the upper bound of $\Theta\left(\frac{n^2 \log\log n}{\log n}\right)$ given in [ABP17a, ABP17b] in the sequential pebbling model. Notably we can show a family of constant in-degree graphs exists that satisfy this property. [Theorem 10.6.23]
- The "cylinder graph" (Graph Construction 10.5.4) can be used to construct an SHF, $\mathcal{H} = (\mathcal{H}_1, \mathcal{H}_2)$, with static memory requirement $\Lambda \in \Theta((\kappa - \xi \log(q_2))\sqrt{n})$ (in bits) where n is the number of nodes in the graph, κ is a security parameter, q_2 is the number of oracles queries made by $\mathcal{H}_2$, and $\xi \in \omega(1)$. This means that any successful adversary using non-trivially less *static* memory than Λ must incur at least Λ *dynamic* memory usage for at least $\Theta(\sqrt{n})$ steps. [Theorem 10.5.28]
- Graph Construction 10.5.15 can be used to construct an SHF, $\mathcal{H} = (\mathcal{H}_1, \mathcal{H}_2)$, with static memory requirement $\Lambda \in \Theta((\kappa - \xi \log(q_2))\sqrt{n})$ (in bits) where n, κ, q_2, and ξ are as described above. This means that any successful adversary using non-trivially less *static* memory than Λ must incur at least Λ *dynamic* memory usage for at least $\Theta(n)$ steps. [Theorem 10.5.30]
- There exist graphs for which an adversary facing a linear space-time cost trade-off would in fact employ a *different pebbling strategy* from one facing a cubic trade-off. [Theorem 10.6.8]
- Given any graph construction $G = (V, E)$, there exists a pebbling strategy that is less expensive asymptotically than any strategy using a number of pebbles asymptotically equal to the number of nodes in the graph for any time-space tradeoff. [Theorem 10.6.13]

Bibliographic Information The results in Part IV are based off the following works:

1. [DL18] Erik D. Demaine and Quanquan C. Liu. Red-blue pebble game: Complexity of computing the trade-off between cache size and memory transfers. (Chapter 9)
2. [DLP18] Thaddeus Dryja, Quanquan C. Liu, and Sunoo Park. Static-memory-hard functions, and modeling the cost of space vs. time. (Chapter 10)

Chapter 9

Complexity of Computing the Trade-Off between Cache Size and Memory Transfers

This chapter presents results from the paper titled, "Red-Blue Pebble Game: Complexity of Computing the Trade-Off between Cache Size and Memory Transfers" that the thesis author coauthored with Erik D. Demaine [DL18]. This paper appeared in the Symposium on Parallelism in Algorithms and Architectures (SPAA) 2018.

9.1 Introduction

Pebble games were originally introduced to study compiler operations and programming languages. One example of such an application is when a directed acyclic graph (DAG) represents the computational dependencies between operations and the pebbles represent register or memory allocation. Minimizing the resources allocated to perform a computation is accomplished by minimizing the number of pebbles placed on the graph [Set75], and the time of computation is modeled by the number of pebbles moves the player makes in a strategy that ultimately pebbles the desired output vertices. In addition to the standard pebble game (also known as the black pebble game in the literature) that models register or memory allocation, there are several other pebble games that are useful for studying computation. The *red-blue pebble game* is used to study I/O complexity [JWK81], the *reversible pebble game* is used to model reversible computation [Ben89], and the *black-white pebble game* is used to model non-deterministic straight-line programs [CS74].

In this chapter, we study the *red-blue pebble game* used to model the cost of programs in a two-level memory hierarchy. The two-level memory hierarchy model of [JWK81] and the blocked version, the I/O model or external memory model [AV88], were introduced in the 1980s [JWK81, AV88] to capture the computational bottleneck of transferring data between a large but slow disk and a small but fast cache. (See [Dem17] for more about the history.) The *red-blue pebble game* models the number of such data transitions that are necessary between cache and disk, as well as the limit of the cache size. The rules of the game are as follows:

Red-Blue Pebble Game [JWK81]. Given a DAG $G = (V, E)$, the game works as follows:

1. At the start, every source node has a blue pebble, and no other nodes have pebbles.
2. The player can place a red pebble on a node if and only if all of its predecessors are currently pebbled with red pebbles.
3. The player can recolor a blue pebble to red, or a red pebble to blue.
4. The player can delete a pebble from a node at any time.

Goal: Pebble all sink nodes with blue pebbles.

Red pebbles represent data in cache and blue pebbles represent data in disk. We suppose (as in real systems) that there is a limited amount of cache and an unlimited amount of disk. In terms of the red-blue pebble game, this means that the number of red pebbles is limited by some upper bound r. Subject to this constraint, the goal is to pebble all target nodes while minimizing the number k of pebble recolorings between red and blue, i.e., minimizing the number k of memory transfers between cache and disk. Recoloring a pebble from red to blue corresponds to writing data out from cache to disk, while recoloring from blue to red corresponds to reading data in from disk to cache.

Much previous research has focused on proving lower and upper bounds on the pebbling cost (i.e., the number of pebbles and/or transitions used) of pebbling a given family of DAGs under the rules of the red-blue pebble game. Such upper and lower bounds are computed with respect to the number of transitions given a number of red pebbles, r, that are provided to pebble the graph. Such previous results include upper and lower bounds of $O(n \log n / \log r)$ and $\Omega(n \log n / \log r)$, respectively, on the minimum number of transitions needed to pebble an FFT graph. Upper and lower bounds on number of transitions for other graph classes such as r-pyramids, diamond graphs, butterfly graphs, and matrix multiplication graphs can be found in [ERP+15, JWK81, RSZ12]. More recently, the model of one-shot red-blue pebble games was introduced in [CRSS16]. This pebble game is used to model I/O-complexity *without recomputating any calculations in cache.* They also show how to extend this model to the multi-level memory hierarchy case.

Despite somewhat extensive research on the upper and lower bounds of optimally pebbling a DAG in pebble games, the complexity of finding a minimum solution has fewer results. In fact, it is not yet known whether it is hard to find the minimum number of pebbles within a constant or logarithmic multiplicative approximation factor [CLNV15, DL17]. It turns out that finding a strategy to optimally pebble a graph in the standard pebble game is computationally difficult even when each vertex is allowed to be pebbled only once. Specifically, finding the minimum number of black pebbles needed to pebble a DAG in the standard pebble game to within an additive $n^{1/3-\varepsilon}$ term is PSPACE-complete [DL17] and finding the minimum number of black pebbles needed in the one-shot case is NP-complete [Set75]. While much has been done in showing upper and lower bounds in pebbling price in terms of number of red pebbles and number of transitions of pebbling certain types of DAGs using the red-blue pebble game, the computational complexity of finding the exact number of minimum red pebbles used and the minimum number of transitions has not been studied in the past to the best of the authors' knowledge.

The purpose of this chapter is two-fold. We seek to answer the question of computational complexity of finding the optimal number of red pebbles and minimum number of transitions used. Secondly, we seek to motivate the study of finding lower bounds and optimal pebbling strategies for certain graph classes by showing that such hardness results even hold for the very restricted class of layered graphs. In this chapter, we discuss the following new results:

1. In Section 9.2, we show that the red-blue pebble game is PSPACE-complete, via a simple extension of the result of [GLT80].
2. In Section 9.3, we introduce a new red-blue pebble game and prove it NP-complete. Specifically, we consider the natural restriction to when all data must be kept somewhere, either in cache or on disk, or equivalently, the player cannot delete pebbles, only recolor them.
3. In Section 9.4, we analyze the complexity of the red-blue pebble game when parameterized by the allowed number t of transitions. We prove that this problem is W[1]-hard, so it does not have a fixed-parameter algorithm (with running time $f(t)n^{O(1)}$) unless FPT $= W[1]$. (Note that some PSPACE-complete problems are fixed-parameter tractable, so this result is not implied by our other results.)

9.2 Red-Blue Pebble Game

We begin this section with a short proof that the red-blue pebble game *with deletion* is PSPACE-complete. We do not expand too much into the proof since it relies heavily on the proof given in [GLT80] (and is almost identical to the proof provided there). Therefore we include this result first before our main result on red-blue pebbling *without deletions* whose proof we expand upon in more detail in the next section.

First, we define the *red-blue start-in-disk* game to be the version of the red-blue pebble game as defined in Section 9.1 (i.e. all source nodes contain blue pebbles at the beginning of the computation, i.e. at $t = 0$, and all inputs if deleted from cache must be obtained from disk and all outputs are written back to disk) and the *red-blue start-in-cache* game to be the version of the red-blue pebble game where we remove the condition that all source nodes contain blue pebbles at the beginning of the computation (i.e. essentially, all inputs start in cache) and red pebbles can be placed at any time on the source nodes– without the need of blue pebbles being on the source nodes first (i.e. inputs always stay in cache). Furthermore, for the *red-blue start-in-cache* game, we assume that all targets must be computed at least once in cache without the need to write the results back into disk (i.e. this is also known as a visiting pebbling in cache [Nor15] in that the red pebbles do not need to be turned into blue pebbles even if they are deleted in cache).

Before we dive into the proofs, we first show that any red-blue pebbling of a DAG G using the rules of the red-blue start-in-cache game that has minimum red pebble space usage r and minimum number of transitions k can be converted to a DAG G' with minimum pebbling space usage $r + 1$ and number of transitions $k + 1 + |T|$ (where T is the target set or output nodes) using the rules of the red-blue start-in-disk game.

Theorem 9.2.1 (Red-Blue Disk to Cache). *Given a DAG, $G = (V, E)$ with target set T and bounded in-degree 2 that uses a minimum of r red pebbles and k transitions to pebble using*

the rules of the red-blue start-in-cache game, we can convert it into a DAG, $G' = (V', E')$, that uses a minimum of $r+1$ red pebbles and $k+1+|T|$ transitions to pebble using the rules of the red-blue start-in-disk game.

Proof. First, create a node u and let $V' = V \cup \{u\}$. Then, create a set of directed edges U where we add an edge (u, v) to U for all $v \in V$. Let $E' = E \cup U$. Graph G' now potentially has vertices with in-degree up to 3. In the final step of creating G', we replace all vertices with in-degree 3 with pyramids of height 3. Note that a pyramid of height 3 functions in the same manner as a node with in-degree 3 by normality of pebbling strategies proven in [GLT80, Nor15].

Let status be the optimal strategy used to pebble G using the rules of the red-blue start-in-cache game that results in a minimum of r red pebbles and k transitions.

Now, we prove that a minimum of $r+1$ pebbles and $k+1+|T|$ transitions are necessary to pebble G' using the rules of the red-blue start-in-disk game. By construction, G' has one source (leaf) node where one blue pebble is placed on it at the beginning of the pebbling (at $t = 0$). Before any other pebbles can be placed on G', we must use exactly 1 transition to turn the blue pebble on the source to a red pebble. The red pebble remains on the source, and we use strategy status to pebble the remaining nodes of G'.

We now consider the minimum number of transitions that can be used with at most $r+1$ red pebbles and before all outputs are written back to disk. We show that in order to use a minimum of $k+1$ transitions, the red pebble must remain on the source during the entire computation of G'. Suppose for contradiction, the red pebble is turned into a blue pebble (recall that all leaves must contain a pebble at all times–otherwise they can never be pebbled again using the rules of the red-blue start-in-disk game), then, in any future pebbling of any other nodes in G', the blue pebble on the source must be turned into a red pebble, resulting in 2 additional transitions (a total of $k+3$ transitions) which exceeds the minimum allowed $k+1$ transitions (since status uses a minimum of k transitions and turning the pebble on u to red requires 1 transition). Thus, given that the red pebble remains on the source during the entire pebbling of G' (i.e. the red pebble on the source u is present at time $t = \arg\max_{P_t \in \text{status}} \{|P_t|\}$) where the maximum number of red pebbles are on the graph using strategy status) the number of red pebbles necessary to pebble G is then increased by 1, so a minimum of $r+1$ red pebbles are necessary to pebble G' given that a minimum of $k+1$ transitions are used before all outputs are written back into disk.

Finally, given the set T of targets, one transition must be spent to turn a red pebble into a blue pebble on each $v_T \in T$. Thus, at least $|T|$ transitions must be spent in this case, resulting in a total of at least $k+1+|T|$ transitions. □

In the remaining sections of this chapter, we prove all results with respect to the rules of the red-blue start-in-cache game, even if we do not explicitly state that we do so. Note that using Theorem 9.2.1, we can transform any graph G we use in our hardness reductions into a graph G' that can be used to show the corresponding hardness results for the red-blue start-in-disk game.

The red-blue start-in-cache pebble game as defined above is PSPACE-hard as a simple extension of the proof given in [GLT80]. The formal definition of the problem is given below.

Definition 9.2.2 (Red-Blue Pebble Game). *Given a DAG, $G(V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, find a pebbling of G following the red-blue start-in-cache pebbling rules as defined above such that at most r red pebbles are present on G at any time and the number of red-blue transitions, k, is minimized.*

The proof structure and the gadgets to show that the red-blue pebble game is PSPACE-hard can be constructed in the same way as the gadgets in the proof of the PSPACE-hardness of the standard pebble game as defined in [GLT80]. The reduction would specify the number of red pebbles necessary to be one greater than the number of pebbles necessary in the proof presented by Gilbert et al. [GLT80] and the number of transitions to be 0. We, thus, only need to show that the number of red pebbles necessary to pebble the gadgets in the construction is indeed one greater than the number necessary to pebble the construction provided in [GLT80]. We show that the construction can be pebbled with one greater pebble in the red-blue pebble game using 0 transitions if and only if the construction in [GLT80] can be pebbled using the rules of the standard pebble game; hence the red-blue pebble game is PSPACE-complete.

Lemma 9.2.3. *The proof construction provided in [GLT80] can be pebbled using s pebbles in the standard pebble game if and only if it can be pebbled using $s + 1$ red pebbles and 0 transitions in the red-blue start-in-cache pebble game.*

Proof. We first show that if the construction given in [GLT80] can be pebbled using s pebbles in the standard pebble game, then it can be pebbled using $s + 1$ red pebbles and 0 transitions in the red-blue pebble game. The only difference between the rules of the standard pebble game and the red-blue pebble game is that in the standard pebble game, a pebble can be moved from a node to a successor from a predecessor. However, in the red-blue pebble game, we are no longer allowed to move a pebble from a predecessor to a successor. However, the process of moving a pebble from predecessor to successor can be modeled using 2 red pebbles. Suppose that a pebble is moved from a predecessor to a successor in the standard pebble game. Let v be a node in the graph and p be the predecessor from which a black pebble was moved to v. Let $pred(v)$ indicate the set of nodes that are the predecessors of v. Since a pebble was moved from p to v, it means that all nodes in $pred(v)$ are pebbled with black pebbles. If the pebble movement from p to v is modeled using red pebbles, then at the time the black pebble was moved from p to v, all nodes in $pred(v)$ would be pebbled with red pebbles. Thus, one additional red pebble can be placed on v at the same time. In the construction provided by [GLT80], if the constructed DAG can be pebbled using s pebbles, then one additional pebble will be able to simulate all pebble movements from predecessor nodes to successor nodes. Thus, if the original construction can be pebbled using s pebbles in the standard pebble game, the construction can be pebbled using $s + 1$ pebbles in the red-blue start-in-cache pebble game.

Now we show that if the construction provided in [GLT80] can be pebbled using $s + 1$ red pebbles and 0 transitions in the red-blue pebble game, then it can be pebbled using s black pebbles in the standard pebble game. We need to show that having one additional pebble in the red-blue pebble game does not provide an advantage in any situations where a pebble is moved from a predecessor to a successor in the standard pebble game. Suppose

for the purposes of contradiction that there is an advantage if an extra pebble is provided in the red-blue pebble game for situations where a pebble is moved from a predecessor to successor node in the standard pebble game. Then, some node can be pebbled in the red-blue pebble game with one extra pebble that cannot be pebbled in the standard pebble game. Suppose that node v cannot be pebbled in the standard pebble game. This means that some node in $pred(v)$ is not pebbled. Otherwise, a pebble can be moved from a node in $pred(v)$ to v. With one extra pebble, the node in $pred(v)$ can be pebbled. However, in the red-blue pebble game, no red pebbles can be moved from a node in $pred(v)$ to v. Therefore, v cannot be pebbled even with one extra pebble, a contradiction.

In the proof construction provided in [GLT80], every pebbling of a node requires moving a pebble from a predecessor to a successor *except* pebbling the leaf nodes of the clause gadgets. Thus, having an extra pebble in all cases but the case of pebbling the leaf nodes of the clause gadgets does not provide an advantage. Now, we will show that having an extra pebble in the red-blue pebble game also does not provide an advantage in this case. In the proof provided by [GLT80], the clause gadgets can be pebbled using 3 additional pebbles. Suppose that in the red-blue pebble game construction, the clauses now have 4 additional pebbles available. In this case, the red-blue pebble game only provides an advantage if 4 pebbles can be used to pebble a clause gadget even if all variable gadgets connecting to it are in the false configuration. In this case, 4 pebbles must be used to pebble all the leaves of the clause gadget. Then, since the non-leaf nodes of the clause gadget requires a pebble to be moved from a predecessor to successor, they cannot be pebbled in the red-blue pebble game using 4 pebbles if all literals connecting to the clause are false. A simple case by case analysis shows that no other strategy can cause the clause gadget to be completely pebbled. □

Theorem 9.2.4. *Determining the minimum pebbling cost and number of transitions is PSPACE-complete (even given constant number of transitions) to compute in the red-blue pebble game.*

Proof. Containment in PSPACE is trivial and PSPACE-hardness of the red-blue pebble game follows immediately from Lemma 9.2.3. □

By noticing that we can transform the above hardness construction to a layered graph by topologically sorting the vertices used in the construction and replacing each edge that go between non-consecutive layers by a path of length equal to the number of layers that the edge go between, we result in a multiplicative factor increase of at most $O(n^2)$ in the number of nodes in the graph since the number of layers is at most $O(n)$. It is trivial to see that all pebbling constraints are still preserved by replacing edges with paths.

Corollary 9.2.5. *Determining the minimum pebbling cost and number of transitions is PSPACE-complete to compute in the red-blue pebble game even for layered graphs.*

9.3 Red-Blue Pebble Game with No Deletion

In this section, we introduce our model of the red-blue pebble game with *no deletion* and prove that it is NP-complete to determine the minimum number of red pebbles and tran-

sitions needed to pebble a given DAG under the rules of this game. The red-blue pebble game with no deletion is defined as follows:

1. A red pebble can be placed on any vertex that has a blue pebble. (Transition move.)
2. A blue pebble can be placed on any vertex that has a red pebble. (Transition move.)
3. A red pebble can be placed on a vertex where all predecessors of the vertex contain red pebbles. (The red pebble can override preexisting pebble placements without using any additional transitions, i.e. the vertex already contains a blue pebble.)
4. No pebbles can be deleted from a vertex.

As usual, red pebbles represent fast memory and blue pebbles represent slow memory; we assume that we have infinitely large slow memory, but only a bounded fast memory. The goal of this game is to pebble all vertices in G while minimizing the number of *transition* moves. The motivation of this game is to determine the added computational complexity of allowing deletions to occur in the RAM. Suppose that one would like to limit the number of deletions or to minimize the number of transitions as well as deletions. Another motivation is to always maintain computed data in memory. For example, for certain persistent data structures, one always want to keep some form of computed values in memory at all times. This chapter analyzes the computational complexity of such a model.

The formal statement of the game is almost identical to the definition of the red-blue pebble game and is the following:

Definition 9.3.1 (Red-Blue Pebble Game with No Deletions)**.** *Given a DAG, $G(V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, find a pebbling of G following the red-blue with no deletion pebbling rules (given above) such that at most r red pebbles are present on G at any time and at most k red-blue transitions are used.*

In the next few sections, we show that the Red-Blue Pebble Game is NP-complete.

9.3.1 Proof Overview

We provide a reduction broadly similar in concept to [GLT80] except we reduce from Positive 1-in-3 SAT to show that our problem is NP-complete. The definition of Positive 1-in-3 SAT is given below:

Definition 9.3.2 (Positive 1-in-3 SAT [GJ90])**.** *Given a set U of variables and a collection C of clauses over U such that each clause $c \in C$ has size $|c| = 3$ and all literals in c are positive, does there exist a truth assignment for U such that each clause has exactly one true literal?*

The proof of NP-completeness of the red-blue pebble game with no deletions proceeds as follows. We create a set of variable gadgets that are pebbled with a set of red pebbles that determine whether the variable is set to true or false. The variable gadgets are then connected to clause and anti-clause gadgets that enforce the 1-in-3 condition on the literal truth settings. The variable gadgets are also connected to a *pebble sink path* that ensures that all variables are pebbled and set to a truth configuration before the clause gadgets are pebbled. Finally, the clause gadgets and variable gadgets are connected to a *pebble hold*

path that ensures that all red pebbles are removed from these gadgets and are used to fill up the pebble hold path. Specifics about the gadgets and details of the proof construction will be given in the next few sections.

The pebbling of the gadgets occur in two phases: Phase 1 and Phase 2. During Phase 1 of the pebbling, all variables gadgets are set to a truth value. During Phase 2 of the pebbling, the portions of the variable gadgets that were not pebbled in Phase 1, leading to the pebbling of the target node. The gadgets are connected in the following way: all variable gadgets are connected to a pebble sink path (see g_i, g_i', and g_i'' in Fig. 9-2) where the end of the path is connected to the first clause gadget; all clause gadgets are connected via a path and the last clause is connected to another pebble sink path (see s_i, s_i', and s_i'' in Fig. 9-2) which is connected to the target node. Each variable gadget is also connected to the clause it appears in as well as the corresponding anti-clause. We define the pebbling of the clauses and anti-clauses to be the *clause verification phase*.

9.3.2 Gadgets

In this section, we introduce some gadget components that will be used in the proof that the red-blue pebble game with no deletions is NP-complete.

We define a *variable* gadget for every $x_i \in U$. The purpose of the variable gadget is to force a selection of variable assignments. In order to construct the variable gadget, we use a *pyramid* gadget introduced by previous work [GLT80] that is used to "trap" a certain minimum number of pebbles that must be used to pebble the gadget. Henceforth, for every gadget, g, we introduce, we will specify the minimum number of red pebbles, r_g, that can remain on the gadget after it has been pebbled once and t_g, the minimum number of red-blue transitions that must be performed on the gadget after it is pebbled each time.

The pyramid graph has been proven to use h pebbles where h is the height (where a single node has height 1) of the pyramid graph using standard pebbling (with sliding pebbles) [CS74]. Let Π_h be a pyramid graph with height h. It was proven in [Nor15] that the standard pebbling price with no sliding is $h+1$ for a pyramid with height h. Here we prove that using the red-blue pebbling strategy with no deletions, the minimum pebbling price of a pyramid with height h is $r_{\Pi_h} = h+1$ and $t_{\Pi_h} = \frac{h(h+1)}{2}$. Let $PebRBD(\Pi_h)$ be the minimum pebbling price of a pyramid graph using the red-blue strategy with no deletions. The ending state of the pyramid has no red pebbles. We use this property of the pyramid graph in our proof in Section 9.3.3.

Lemma 9.3.3. *Given a pyramid graph of height h, the $PebRBD(\Pi_h)$ is $r_{\Pi_h} = h+1$ and $t_{\Pi_h} = \frac{h(h+1)}{2}$ such that no red pebbles remain on the pyramid at the end of the pebbling.*

Proof. The standard pebbling lower bound (no sliding) for pyramids is $h+1$ given a pyramid of height h. The number of red pebbles necessary to pebble a pyramid of height h, however, is $h+1$ since the key component of the bound on standard pebbling of pyramids relies on the ability to slide pebbles, whereas in our pebbling game, no pebble slides are allowed. Otherwise, the rules for red pebble placement is the same as the rule for standard pebbling with no sliding. Therefore, by the proof of Theorem 4.8 in [Nor15], the lower bound for the number of red pebbles necessary to pebble Π_h is $r_{\Pi_h} = h+1$.

As stated in the proof of Theorem 4.8 in [Nor15], the strategy for achieving this pebbling is to pebble the bottom row (the sources) of the pyramid and use one extra pebble to move the pebbles up the levels of the pyramid.

To prove the transitions bound, since the only way to remove all the pebbles from the pyramid is to use transitions, the number of transitions is $\geq \frac{h(h+1)}{2}$. Since the pebbling strategy stated above for pyramids is linear (each vertex is pebbled by a red pebble at most once), each node is visited once and a red pebble is removed from each node once resulting in $t_{\Pi_h} = \frac{h(h+1)}{2}$. When a pebble is "moved up" a level of the pyramid, the pebble on the previous node is turned to blue. □

Fig. 9-1 shows the construction of the pyramid gadget and its associated symbol that will be used to denote it in all subsequent proofs in Section 9.3.3. One can see that the number of pebbles required to fill the pyramid gadget is also the number of leaves on the bottom layer plus one and the number of transitions is $\sum_{i=1}^{l} i$ where l is the number of leaves in the gadget.

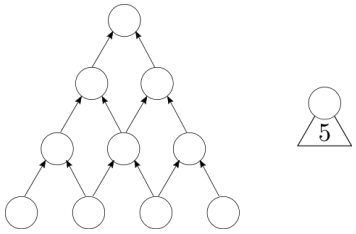


Figure 9-1: Example of a pyramid gadget with $r_{\Pi_4} = 5$ and $t_{\Pi_4} = 10$.

Using the pyramid gadget we can construct the variable gadget as in Fig. 9-2.

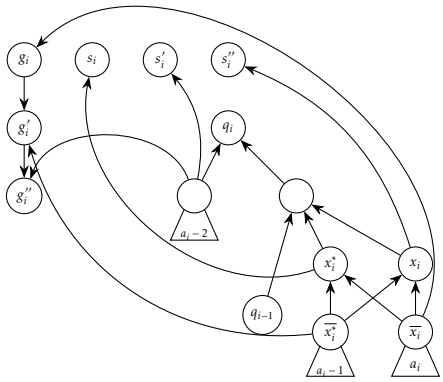


Figure 9-2: Example of a variable gadget, x_i, with pyramid costs a_i, pebble sink path connections g_i, g'_i, and g''_i. The corresponding pebble sinks that correspond with this gadget are s_i, s'_i, and s''_i.

We define a_i for all x_i shortly. Each x_i variable gadget requires a_i pebbles since in order to choose an assignment for x_i, a_i pebbles must be used to pebble the pyramid

gadget attached to the variable. Since each q_i is part of two variable gadgets, we define q_{i-1} to be part of variable gadget x_i and q_i to be part of the next variable gadget. We show that the gadget must be pebbled in the following way.

During Phase 1 of pebbling the variable gadget x_i, each pyramid gadget is pebbled. Then, the corresponding nodes in the pebble sink path can be pebbled in the order g_i, g_i', and, then, g_i''. All nodes along the pebble sink path must be pebbled in order to pebble the clauses and proceed to the clause verification phase. First, a_i pebbles must be used to pebble both $\overline{x_i}$ and $\overline{x_i^*}$ with red pebbles, converting all other pebbled vertices in each pyramid to contain blue pebbles. This then leaves $a_i - 2$ pebbles to pebble the other pyramid gadget, leaving one pebble at the apex of the gadget and converting all other pebbled vertices in the pyramid to contain blue pebbles. The apex of these pyramids are connected to *pyramid sink paths* which are paths of length n occuring after each set of clause and two anti-clauses triples. Then, either the pair of nodes x_i and x_i^* is pebbled with red pebbles and $\overline{x_i}$ and $\overline{x_i^*}$ are converted to blue pebbles or $\overline{x_i}$ and $\overline{x_i^*}$ contain red pebbles and x_i and x_i^* are not pebbled. We show that at most 3 red pebbles can remain on each variable gadget.

In Phase 2, q_{i-1} will be pebbled once all clauses are pebbled. Therefore, all other nodes of the variable gadget must be pebbled using the pebbles that remain on each pyramid gadget. If the red pebbles from Phase 1 are placed on $\overline{x_i^*}$ and $\overline{x_i}$, then x_i and x_i^* need to be pebbled in Phase 2. Furthermore, s_i, s_i' and s_i'' need to be pebbled with red pebbles in Phase 2 by moving the red pebbles from each corresponding pyramid. The red pebbles will remain on s_i, s_i', and s_i'' since no transitions are allowed to be spent on these nodes.

A *clause gadget* is created for each $c_j \in C$. The clause gadget is connected to every positive x_i literal that is present in its respective clause c_j. An example clause gadget with $r_{c_i} = 6$ and $t_{c_i} = 29$ (here r_{c_i} *does not* include the red pebble on the one true literal and the red pebble on p_{i-1} and t_{c_i} does not include the transition used to turn the pebble on p_i to blue) is shown below in Fig. 9-3. The order of the clauses is determined arbitrarily and all clauses are duplicated and occur in the same topological order in the two sets (the original clauses and the duplicate clauses).

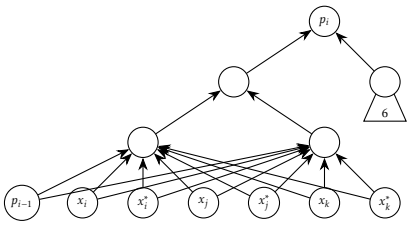


Figure 9-3: Example of a clause gadget with $r_{c_i} = 2$ and $t_{c_i} = 29$ for clause $c_i = (x_i \vee x_j \vee x_k)$. The number of red pebbles that is needed to fill this gadget is 6 (excluding the two red pebbles that are present on the true literal and the red pebble on p_{i-1}).

The clause gadget is accompanied by two *anti-clause* gadgets, $\overline{c_i}$ and $\overline{c_i}'$, that are used to enforce the exact 1-in-3SAT condition. The anti-clause gadgets should also be pebbled with $r_{\overline{c_i}} = 6$ and $t_{\overline{c_i}} = 64$. $\overline{c_i}$ contains all $\overline{x_i}$ literals and $\overline{c_i}'$ contains all $\overline{x_i^*}$ literals. First, the pyramids must be pebbled along the path leading up to p_i. Then, the one negative literal that is not pebbled must be pebbled with a red pebble by using 2 transitions. Finally, the

remaining nodes of the gadget are pebbled once using 62 transitions resulting in p_i being pebbled with a red pebble. An anti-clause gadget is shown in Fig. 9-5.

Each variable gadget is connected to a *pebble sink path* as shown in Fig. 9-4. The purpose of the pebble sink path is to ensure that all pyramids are pebbled by the end of Phase 1 of variable pebbling and that each variable only contains at most 3 red pebbles. The pebble sink path can be pebbled using $a_n - 3n$ red pebbles and the number of transitions needed to pebble this path is $3n + \frac{(a_n-3n+1)(a_n-3n)}{2}$. The number of transitions indicate that each node of the pebble sink path can only be pebbled once. Once the end of the pebble sink path is pebbled, the clause gadgets can be pebbled in the *clause verification phase*.

We now prove our claims above more formally. We begin by proving that the end of Phase 1, at most 3 red pebbles can remain on each of the variable gadgets.

Lemma 9.3.4. *At most $3n$ red pebbles can remain on the variable gadgets at the end of Phase 1 and the beginning of the clause verification phase where n is the number of variable gadgets.*

Proof. The number of nodes in the pebble sink path is $3n + \frac{(a_n-3n+1)(a_n-3n)}{2}$; therefore, in order to pebble the pebble sink path only once with red pebbles using exactly $3n + \frac{(a_n-3n-1)(a_n-3n)}{2}$ transitions, one can only pebble each of the pyramids along the path once. To pebble the last pyramid on the path requires $a_n - 3n$ pebbles; thus, one can only leave at most $3n$ red pebbles on the variable gadgets when the pyramid is pebbled at the end of Phase 1. □

We prove a matching lower bound for the number of red pebbles that should remain on the variable gadgets after Phase 1.

Lemma 9.3.5. *At least 3 red pebbles must remain on each variable gadget at the end of Phase 1 in order to be able to pebble the constructed graph using a minimum number of red pebbles and transitions.*

Proof. We prove this by contradiction. Consider the following cases where less than 3 red pebbles are placed on a particular variable gadget:

1. A red pebble is removed from variable gadget x_i from the apex of the pyramid with cost $a_i - 2$. This pebble can be used to save at most $2m/3$ transitions (where m the total number of clause and anti-clause gadgets) but incurs an additional $4m/3$ because of the pyramid sink line, resulting in an extra $2m/3$ transitions.
2. A red pebble is removed from node x_i, x_i^*, $\overline{x_i}$, or $\overline{x_i^*}$. This pebble can be used to satisfy an unsatisfied clause or anti-clause. However, this means that there exists an unsatisfied clause or anti-clause occurring later on (since we duplicate all clauses). In other words, there exists a clause where 6 pebbles are not enough to satisfy the clause. Thus, at least 4 additional transitions are used to remove the pebble, satisfy the unsatisfied clause or anti-clause, and repebble the node that originally contained the red pebble.

Thus, if a pebble is removed from a variable gadget that contains 3 pebbles, then extra transitions are incurred beyond our allowed number of transitions. Thus, each variable gadget must contain at least 3 red pebbles must remain on the variable gadgets at the end of Phase 1. □

Using Lemma 9.3.4, we can now prove the number of red pebbles and transitions necessary to pebble the variable gadgets.

Lemma 9.3.6. *The pebbling price of the variable gadget (not including s_i, s_i', s_i'', g_i, g_i', g_i'' or q_i) for variable x_i is $r_{x_i} = a_i$ and $t_{x_i} = \sum_{j=1}^{3} \frac{(a_i-j+1)(a_i-j)}{2} + 4$ assuming all red pebbles are removed from the gadget at the end of the pebbling. The transitions cost includes the cost of pebbling and removing red pebbles from all nodes as shown in Fig. 9-2 except s_i, s_i', s_i'', g_i, g_i', g_i'' and q_i. Assuming all variable gadgets are set to a truth value during Phase 1 and Phase 2 begins with the pebbling of q_0 and ends with red pebbles on s_i, s_i', s_i'', q_i, during Phase 1, the pebbling cost is $r_{x_i} = a_i$ and $\sum_{j=1}^{3} \frac{(a_i-j+1)(a_i-j)}{2} - 3 \leq t \leq \sum_{j=1}^{3} \frac{(a_i-j+1)(a_i-j)}{2} - 1$. Furthermore, during Phase 2 of the pebbling, the pebbling cost is $r_{x_i} = 4$ and $5 \leq t_{x_i} \leq 7$ and red pebbles remain only on s_i, s_i', s_i'', and q_i.*

Proof. Each of the three pyramids must be pebbled using the number of pebbles shown in Fig. 9-2 by 9.3.3. The number of nodes in the gadget is $\sum_{j=1}^{3} \frac{(a_i-j+1)(a_i-j)}{2} + 4$ (excluding s_i, s_i', s_i'', g_i, g_i', g_i'' and q_i). If we are limited to a total of $t_{x_i} = \sum_{j=1}^{3} \frac{(a_i-j+1)(a_i-j)}{2} + 4$ transitions divided between Phase 1 and Phase 2, then each vertex of the gadget can only be pebbled once among the two phases. In Phase 1, in order to set the values for all variables, we must use a_i red pebbles to pebble all three pyramids. By normality of red-blue pebbling strategies, none of the a_i pebbles can be used to pebble other gadgets when the pyramids are being pebbled. Recall that g_i, g_i', and g_i'' must be pebbled at the end of Phase 1 in order to begin pebbling the clauses since all g_i, g_i', and g_i'' are ancestors of the clause gadgets. In order to pebble g_i, g_i', and g_i'', all the pyramids in all variable gadgets x_i must be pebbled by the end of Phase 1.

In addition to pebbling the pyramids, one pair of x_i^* and x_i or $\overline{x_i^*}$ and $\overline{x_i}$ must be pebbled in Phase 1 due to our assumption that all variable gadgets are set to a truth value during Phase 1. To ensure that each vertex in the gadget is pebbled exactly once during Phase 1 and Phase 2, the variables must be selected in the indicated pairs. Suppose for contradiction that x_i and $\overline{x_i^*}$ are selected to each contain a red pebble, and no red pebbles remain on x_i^* and $\overline{x_i}$. This means that $\overline{x_i}$ must have been pebbled at some point in Phase 1 (and a transition was used to turn the pebble to blue). Now, in Phase 2, $\overline{x_i}$ must be pebbled again and use one more transition to turn the pebble to blue to pebble x_i^*, breaking our invariant. The same holds if the pair x_i^* and $\overline{x_i}$ was chosen initially (and no red pebbles remain on x_i and $\overline{x_i^*}$).

In Phase 2, the following set of pebblings must occur. If $\overline{x_i}$ and $\overline{x_i^*}$ have red pebbles remaining from Phase 1, then x_i^* and x_i must be pebbled in Phase 2. Since, q_{i-1} will be pebbled by the end of the clause verification phase, the remaining parts of the gadgets can be pebbled using 7 more transitions. If, instead, x_i and x_i^* have red pebbles remaining from Phase 1, then the remaining nodes of the gadget can be pebbled using 5 more transitions. Since s_i, s_i', s_i'', and q_i must be pebbled with red pebbles at the end of pebbling x_i in Phase 2, the number of pebbles necessary to pebble x_i in Phase 2 is 4. □

We prove the following lemma to help us prove that all variable gadgets must be set in Phase 1.

Lemma 9.3.7. *If n' variables do not have at least one of the two pairs, x_i^* and x_i or $\overline{x_i}$ and $\overline{x_i^*}$, pebbled at the end of the clause verification phase, then a total of at least $\left(\sum_{i=1}^{n}\sum_{j=1}^{3}\frac{(a_i-j+1)(a_i-j)}{2}+4\right)+n'$ transitions are needed to pebble all variable gadgets in Phase 1 and Phase 2.*

Proof. Suppose that without loss of generality, node $\overline{x_i}$ is pebbled and node $\overline{x_i^*}$ is not at time t in Phase 2. Suppose also that in accordance with the lemma, either node x_i^* or x_i is also not pebbled with a red pebble. Since all pyramids must be pebbled at the end of Phase 1, $\overline{x_i^*}$ must have been pebbled with a red pebble at some time $t' < t$ during Phase 1. Since $\overline{x_i^*}$ was pebbled with a red pebble at time t' and holds a blue pebble at time t in Phase 2, 1 transition must have been used to convert the red pebble to a blue pebble during Phase 1 by Lemma 9.3.4. In order to pebble either x_i^* or x_i in Phase 2, one transition must be used to convert the blue pebble on $\overline{x_i^*}$ to a red pebble. Therefore, one additional transition per each of the n' variables must be used to pebble the remaining portions of the variable gadgets. Therefore, the number of transitions necessary to pebble the variable gadgets in Phase 1 and Phase 2 is at least $\left(\sum_{i=1}^{n}\sum_{j=1}^{3}\frac{(a_i-j+1)(a_i-j)}{2}+4\right)+n'$.

□

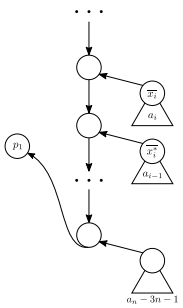


Figure 9-4: Example pebble sink path. Each node is connected to the root of each pyramid in each variable gadget.

Lemma 9.3.8. *Given $r_{c_i} = 6$ and $t_{c_i} = 29$, the clause gadget c_i (shown in Fig. 9-3) cannot be pebbled if all three variable gadgets incident on c_i are in the false configuration and no red pebbles remain on c_i after it is pebbled (i.e. when p_i is pebbled).*

Proof. The clause gadget must be pebbled by first pebbling the pyramid gadget that requires 6 red pebbles to pebble. Then, with the remaining 5 red pebbles the bottom layer of the gadget (i.e. the nodes representing the literals) are pebbled using 4 red pebbles plus the two pebbles that are present on the true positive literal (false literals are pebbled to become true). The final red pebble plus the red pebble from p_{i-1} are used to pebble the next layer. The rest of the pebbling follows directly from this initial pebbling. Once the gadget has been pebbled, one red pebble is left at the apex of the gadget, p_i, and the true literal still contains its red pebbles. Four transitions are used to convert the two pebbled false literals back to false and 29 transitions are used to convert all other vertices except p_i and the positive literal to blue.

The amount of transitions needed to pebble the clause gadget if all incident literals are in the false position is 31 if no red pebbles remain on the gadget after it is pebbled. Suppose that two red pebbles remain on the gadget, then the number of red pebbles available to the next clause is 4 which is not enough to ensure that the clause is successfully pebbled. Suppose without loss of generality that two red pebbles remain on x_i and x_i^*, then, two red pebbles need to be removed from $\overline{x_i}$ and $\overline{x_i^*}$ resulting in two extra transitions as before (essentially setting the literal to true–but this can be done at most once per variable). Therefore, a clause gadget cannot be pebbled under the conditions stated in the lemma unless at least one literal is true. □

Lemma 9.3.9. *Given $r_{\overline{c_i}} = 6$ and $t_{\overline{c_i}} = 61$, the anti-clause gadget cannot be pebbled if less than 2 negative literals are true.*

Proof. 4 red pebbles are necessary to pebble the pyramids with costs in the range $[3, 6]$. By Lemma 9.3.8, at least one of the x_i, x_j, and x_k variables must be set to true. Thus, at least one of $\overline{x_i}, \overline{x_j}, \overline{x_k}$ does not contain a red pebble. Thus, at least 2 red pebbles in addition to the red pebble on p_{i-1} are necessary to pebble the descendents of $\overline{x_i}, \overline{x_j}, \overline{x_k}$.

Without loss of generality, we will assume we are pebbling the anti-clause gadgets containing the $\overline{x_i}$ variables. At most 2 transitions are allocated to pebble and unpebble any $\overline{x_i}$ variables that are not pebbled with red pebbles from Phase 1. The remaining vertices need at most $t = 59$ transitions to pebble and unpebble. Therefore, the total number of transitions and red pebbles needed is $r = 6$ and $t = 61$. □

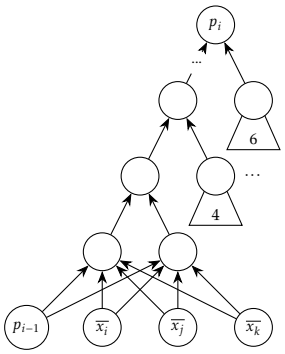


Figure 9-5: Example of an anti-clause gadget with $r = 6$ and $t = 64$. The number of red pebbles that is needed to fill this gadget is 6 (excluding the two red pebbles that are present on the true negative literals).

Lemma 9.3.10. *Each clause gadget must contain exactly one true literal pebbled with red pebbles and each anti-clause gadget must contain exactly two true literals pebbled with red pebbles at the end of Phase 1 before the clause verification phase.*

Proof. The following are the different possible ways red pebbles can reside on the nodes for each variable gadget:

1. A clause contains 0 gadgets set in the true configuration at time t during the clause verification phase. If a clause contains 0 true literals, then we must pebble the clause using either at least 7 red pebbles (in addition to the red pebble on p_{i-1}). Given that all variable gadgets must contain 3 red pebbles by the end of Phase 1 by Lemma 9.3.5, we must obtain the 2 extra pebbles from another variable gadget. Obtaining the 2 extra pebbles from the variable gadget results in 2 extra transitions during Phase 1 or at a previous time $t' < t$ in the clause verification phase. Furthermore, these two pebbles must be deleted and reinserted back into the other variable gadget resulting in two more transitions.
2. A clause contains 2 variable gadgets set in the true configuration. In this case, the clause gadget does not save any transitions since recomputation in memory is free. However, this also means that both corresponding anti-clause gadgets need one more red pebble placement to pebble them. This results in at least 2 additional transitions to turn the blue pebbles on the negative literals to red plus 2 additional transition from Phase 1 or some time t' before the current time to obtain the extra necessary red pebbles. This results in a net gain of 4 additional transitions.
3. A clause contains 3 true variables. This is the same case as 2 with net transitions change (i.e. number of transitions needed to pebble the anti-clauses minus the number of transitions saved) of 8 instead.
4. Without loss of generality, a clause contains the pair x_i and $\overline{x_i^*}$ that are pebbled with red pebbles and $\overline{x_i}$ is pebbled with a blue pebble and x_i^* is not pebbled. In order to pebble the corresponding anti-clause, one pebble must be removed from x_i using one transition and the blue pebble on $\overline{x_i}$ must be turned into a red pebble using an additional transition. There will be a net increase of at least one additional transition with each variable gadget that is set in this configuration.
5. A variable gadget contains more than 3 pebbles. (For instance, a variable gadget could contain red pebbles on $\overline{x_i}$, $\overline{x_i^*}$, and x_i.) Recall that at the end of Phase 1, all variables gadgets are each pebbled with 3 red pebbles. Therefore, in order for a literal to be pebbled with more than 2 red pebbles, at least one transition is used to delete the pebble from a variable gadget by the end of Phase 1 (assuming that the pebble is not one of the 6 red pebbles used to pebble the clause gadgets). If the red pebble is moved onto a positive literal node then one transition is used to delete the pebble from its previous node. If it is moved onto a negative literal node, then 2 transitions are used to delete the pebble from its previous node and place the pebble on its new node. If the pebble movement is unnecessary for satisfying a clause or anti-clause, then it would not occur. If the movement is necessary, then at least one other transition per variable that contains more than 3 pebbles is necessary resulting in more transitions than that allowed.
6. The variables switch value after some clauses are satisfied. If the variables switch from the true configuration to the false configuration to satisfy some anti-clause, then the switch would result in at least 4 transitions per switch exceeding our allowed bound on transitions. If the switch was from a false configuration to a true configuration, then an extra 2 transitions per switch is necessary. Suppose that some of these extra transitions are credited to the transitions necessary for pebbling a clause. In the case of the false to true switch, none of the switches can be cred-

ited to satisfying the corresponding previous clause since the switch needs to occur before the first clause or anti-clause is unsatisfiable. Therefore, no extra transitions can be saved from previous clauses or anti-clauses since all clauses were satisfied (and none of the previously mentioned cases occurred). The problem occurs when one of the previously false variables could be turned to true and a true variable can be turned to false during clause verification. However, this results in 2 extra transitions that cannot be shared in the next clause verification since two of the true literal red pebbles must be removed from the variable that was turned to false since all variables must be set by the conclusion of the clause verification phase. In the other case, if a variable needs to be switched from true to false, then one can only charge the transitions necessary for deleting the pebbles on the positive literals to satisfying the previous clause. However, the 2 transitions for turning the negative literals to true cannot be charged to clause satisfaction.
The proof of the lemma follows from the set of cases mentioned above.

□

Given these gadgets, we are ready to proceed with the reduction from Positive 1-in-3 SAT.

9.3.3 Reduction from Positive 1-in-3 SAT

Given a Positive 1-in-3 SAT expression, φ, we create a variable gadget for each of the n variables and a clause and two anti-clause gadgets (one for $\overline{x_i}$ and one for $\overline{x_i^*}$) for each of the m clauses. The gadgets are linked together as shown in Fig. 9-6. .

Each variable gadget is connected to the next by the set of vertices Q consisting of nodes $q_i \in Q$. Each variable gadget is also connected to the pebble sink path consisting of vertices $g_i \in G$ and to the pebble hold nodes s_i, s_i', and s_i''. For each clause gadget, we connect it with its corresponding anti-clause gadgets via the nodes in the set $p_i \in P$. The final anti-clause gadget in the chain of clause and anti-clause gadgets is connected to the bottom of the chain of variable gadgets. Finally, all variable gadgets are connected to pebble hold nodes, $s_i, s_i', s_i'' \in S$ that are also along a path and ensure that all red pebbles end on these set of nodes. There are no transitions allocated for these nodes; therefore, any red pebbles that are used to pebble these nodes must remain.

We let $a_n = 3n + 6$ and $a_i = a_{i-1} + 3$. Therefore, we set $r = 3n + 6$ and $t = 93m + 3n + 22 + \sum_{i=1}^{n} \sum_{j=1}^{3} \frac{(a_i - j + 1)(a_i - j)}{2}$ for the entirety of the construction.

We now provide an argument that red-blue pebbling with no deletions is in NP.

Lemma 9.3.11. *Given a DAG $G(V, E)$ where $n = |V|$, and parameters r and t and a pebbling strategy, we can check whether the strategy works in time $O(n^2)$.*

Proof. We can solve any red-blue pebbling game using $O(n^2)$ transitions given a reasonable number of red pebbles. We can achieve this by performing the pebbling greedily. If $r < \max(indegree(v))$ for some vertex v, then the pebbling cannot be done. Otherwise, all other pebblings can be completed using $(2d + 1)n$ transitions where $d = \max(indegree(v))$ for all vertices v in a graph G with n vertices. On the other hand, a pebbling can never be performed if $t < n - r$. Therefore, we seek to show that

$n - r \leq t \leq (2d + 1)n$ and $r \geq d$ are necessary conditions to valid strategies. To show that all pebblings can be completed using $(2d + 1)n$ transitions, first, topologically sort the vertices in the DAG, then perform pebbling according to the topological sort.

No transitions are used to pebble the predecessors of the first node in the topological sort and only 1 transition is used to remove a pebble from the node after it has been pebbled. Now for each of the outgoing edges of this node, the number of times this vertex will need to be pebbled and removed is at most the number of outgoing edges from the node. Since all nodes that are predecessors of the current node in the topological sort must be pebbled before the current node, the current node can be pebbled by using d transitions to make all predecessors contain red pebbles, pebble the current node, and then use $d + 1$ transitions to convert all red pebbles to blue pebbles. There are n nodes that need to be pebbled in this way, therefore, at most $(2d + 1)n$ transitions are needed to pebble the graph. Therefore, the maximum number of transitions needed to pebble a graph is $2dn = O(n^2)$.

Using only $O(n^2)$ transitions results in a time of pebbling that is $O(n^2)$. Therefore, checking the pebbling strategy should not take more than polynomial amount of time (if $t = \omega(n^2)$ and $r \geq d$, then the graph can always be pebbled). □

Theorem 9.3.12. *Generalized red-blue no-deletion pebble game on a DAG with maximum in degree 7 is NP-complete by reduction from Positive 1-in-3 SAT.*

Proof. We first show that given a solution to φ, we can construct a solution to our construction using the amount of red pebbles and transitions as described above. We first set the variables according to the assignment provided by the satisfying assignments to φ. Then, we pebble the remainder of the construction according to the steps provided in the previous section.

Now we prove that given a satisfying pebbling strategy to our construction, we also have a satisfying assignment for φ. First, during Phase 1 of pebbling the construction, the variable gadgets as described in Section 9.3.2 are pebbled using the number of red pebbles and transitions described in Lemma 9.3.6. By Lemma 9.3.7, at the end of Phase 1, the variable gadgets all have at least one pair of x_i and x_i^* or $\overline{x_i}$ and $\overline{x_i^*}$ pebbled with red pebbles. Each pyramid of the variable gadget needs to be pebbled in order to pebble the pebble sink path which is necessary to pebble before the clause verification phase. No more than 3 red pebbles can remain in each variable gadget since the next variable gadget uses the number of red pebbles minus the 3 that remain in the previous variable gadget. The remaining 6 pebbles after all variable gadgets are pebbled will be used to pebble the clause and anti-clause gadgets. In order for the clause and anti-clause gadgets to be pebbled, exactly 1 literal of each clause must be true as proven by Lemma 9.3.10.

Each clause and anti-clause gadget uses the number of red pebbles and transitions as given in Lemma 9.3.8 and Lemma 9.3.9 in order to be pebbled. After all clauses and anti-clauses have been pebbled, we can proceed with Phase 2 of pebbling the variables. The cost of pebbling the variables during Phase 2 is given in Lemma 9.3.6.

Since the number of transitions thus far cover the pebbling of the variable gadgets, the pebble sink path, and the clause/anti-clause gadgets, there does not remain any transitions for the pebble hold nodes. Therefore, all the red pebbles will be used to pebble these

pebble hold nodes. By Lemmas 9.3.7 and 9.3.10, if there exists a valid strategy to pebble the configuration, then there exists a valid variable assignment for φ.

The problem is in NP by Lemma 9.3.11. Therefore, the problem is NP-Complete by reduction from Positive 1-in-3SAT. □

Corollary 9.3.13. *Generalized red-blue no-deletion pebble game on a DAG with maximum in degree 2 is NP-Complete by reduction from Positive 1-in-3 SAT.*

Proof. This follows as an immediate corollary of Theorem 9.3.12 by using the gadgets described in [GLT80]. □

One can produce a reduction with indegree 2 graphs by replacing all nodes with in-degree greater than 2 with pyramids with height equal to the indegree minus 1. Similar proofs can prove hardness in this case.

9.4 Red-Blue Pebble Game Parameterized by Transitions

In this section, we prove that the red-blue pebble game with deletion where the number of red-to-blue or blue-to-red transitions is parameterized by k is W[1]-hard by reduction from the W[1]-complete problem, Weighted q-CNF Satisfiability. It was previously shown by [DF95a] that Weighted q-CNF Satisfiability is W[1]-complete for any fixed $q \geq 2$. In order to maximize the similarity to our previous reductions, we will be reducing from Weighted 3-CNF SAT via a parameterized reduction.

It has been noted that this result seems superfluous given the NP-hardness result for 0 transitions given in Section 9.2. However, we note that an NP-hardness result does not necessarily supersede a parameterized complexity result since they are different complexity domains. In fact, there exist NP-complete problems with natural parameterizations that have fixed-parameter tractable algorithms. Furthermore, the techniques presented in this section are techniques that could be important for future proofs of hardness or hard-to-pebble graph family constructions.

Definition 9.4.1 (Weighted q-CNF Satisfiability [DF95a, DF95b])**.** *Given a CNF formula, φ, a set U of variables ($n = |U|$), and a set C of clauses ($m = |C|$) where the number of literals per clause is at most q, determine whether there is a satisfying assignment for φ of truth values to the variables in U such that the number of variables that are true is k.*

Given a 3-CNF formula, we first create two clauses for each of the variables: for all $x_i \in U$ we add the clauses $(x_i \vee x_i \vee \overline{x_i}) \wedge (x_i \vee \overline{x_i} \vee \overline{x_i})$. Note that if a truth value is assigned to x_i, then both clauses must be true. The presence of these clauses is to ensure that each of the variable gadgets in the reduction are assigned a valid truth value.

The reduction transforms an instance of Weighted 3-CNF SAT with parameter k to an instance of red-blue pebble game *with deletion* (note that this is a different model from the one presented in Section 9.3) such that the reduced instance is allowed $r = 7n - 4k + 1$ pebbles and $2k$ red-blue transitions.

We first provide an overview of our proof techniques. Then, we describe the gadgets used in our proof. Finally, we provide the proof that the red-blue pebble game defined in Section 9.2 is W[1]-hard.

9.4.1 Proof Overview

Given a Weighted 3-CNF SAT expression, φ, we first duplicate all the variables and clauses until n' the number of new variables including duplicates follows the rule $\frac{3n'}{4} > k$. From here onwards, we refer to φ to be the new 3-CNF expression (with the duplications) and n to be the number of new variables.

As in the proof given in Section 9.3, we create a set of variable gadgets that are connected to a set of clause gadgets that check whether each clause is satisfied according to the truth settings of the variables. The k true variables conditions is enforced by the *All-False* and *k-True-Variables* gadgets which first force all variable gadgets to be set to false, then picks exactly k variables to set to true and uses exactly $2k$ transitions. The problem is parameterized by the number of transitions, $2k$, and the number of red pebbles is limited by some number that is polynomial in the number of variables in φ. All of the transitions will be used before the clause gadgets are pebbled. Therefore, all pebblings of all gadgets after the variable gadgets, the All-False gadget, and the k-True-Variables gadget are pebbled using only red pebbles and no transitions.

We will now describe the gadgets that are used in the reduction.

9.4.2 Gadgets

Variable Gadget

The variable gadgets are used to represent the variables that are in U and are present in φ (i.e. we do not create a variable gadget for variables that are not present in φ). We again categorize the complete pebbling of the variable gadgets into three phases: Phase 1, Phase 2, and Phase 3. During Phase 1, each variable must be pebbled in the following way. The pyramid gadgets within each variable gadget are first pebbled with red pebbles and one red pebble remains on the apex of each pyramid gadget. See Fig. 9-7.

After all variable gadgets have been pebbled once (i.e. both $\overline{x_i'}$ and $\overline{x_i}$ are pebbled), the x_i' nodes must be pebbled with red pebbles. In order to pebble the x_i' nodes, the remaining red pebbles will be used as well as the red pebbles on $\overline{x_i'}$. The corresponding red pebble on $\overline{x_i'}$ is either turned to blue or removed. At most k of these red pebbles may be turned to blue since we are given only $2k$ transitions and each of the blue pebbles must be reverted back to red at some point in the future (proof will be provided later). The three vertices representing x_i remain un-pebbled and a red pebble remains on each $\overline{x_i}$. Each vertex of x_i' as well as $\overline{x_i}$ are connected to the All False gadget (described below). The All False gadget must be pebbled after the variable gadgets since all other subsequent pebbling depends on the set of $2k + 1$ nodes that were pebbled during the pebbling of the All False gadget.

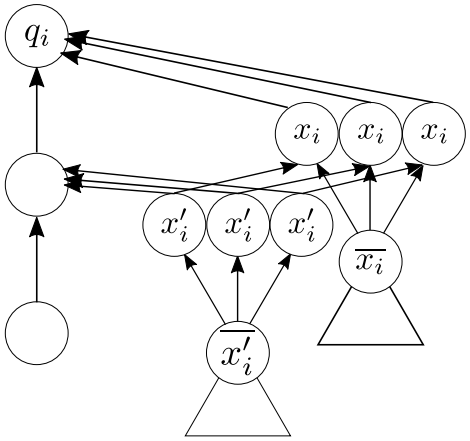


Figure 9-7: Variable gadget.

Lemma 9.4.2. *All variable gadgets must be in the false configuration after Phase 1.*

Proof. During Phase 1, the All False Gadget (see Fig. 9-8) must be pebbled. We are allowed at most a_n pebbles and the All False gadget consists of a set of $2k+1$ nodes each of which costs a_n, $a_n - 1, \ldots, a_n - 2k$ pebbles to pebble (i.e. have indegree a_n, $a_n - 1, \ldots, a_n - 2k$). Thus, the only possible pebbling configuration is the configuration that leads all variable gadgets to be in the false configuration and the remaining pebbles are used to pebble the other nodes that each of the $2k + 1$ nodes are dependent on. □

During Phase 3 of pebbling the variable gadgets, the other nodes within the gadget are pebbled using the red pebbles that are left on the gadget from Phase 2. This phase requires no transitions since the extra red pebbles from the clause and pebble sink path gadgets can be used to pebble the variable gadgets during this phase.

All False Gadget

The All False gadget is used to check that all variables are initially set to false. See Fig. 9-8. It consists of $2k + 1$ vertices with unbounded indegree with predecessors x'_i and $\overline{x_i}$ for all i. Furthermore, its predecessors also contain $a_n - 4n - i$ nodes for $i = \{0, \ldots, 2k\}$ to use up the other $a_n - 4n - i$ extra pebbles. The $2k + 1$ nodes from the All False gadget are then connected to the k-True-Variables gadget and all the clause gadgets.

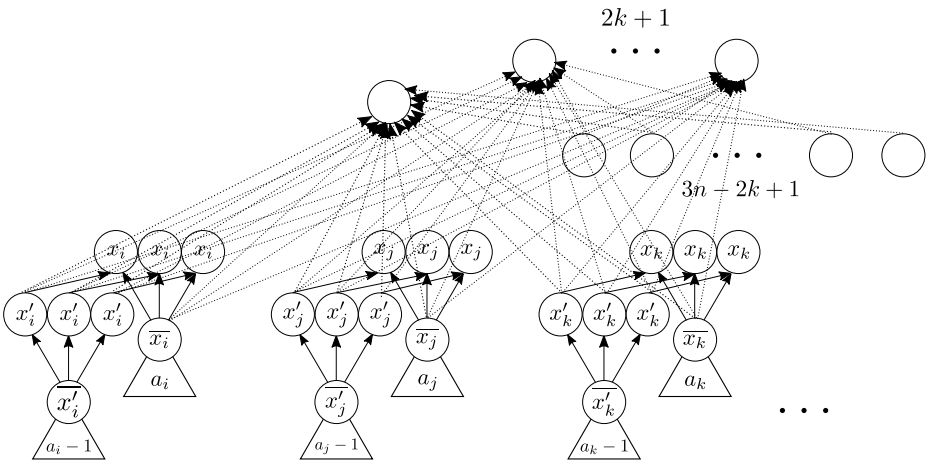


Figure 9-8: The All False gadget consists of $2k+1$ nodes that all have x'_i and $\overline{x_i}$ as predecessors. Each of these $2k+1$ nodes are connected to the k-True-Variables gadget and the clause gadgets.

k-True-Variables Gadget

Phase 2 of the variable pebbling phases consists of resetting a set of k variables to true. The k-True-Variables gadget is present to constrain the number of true variables to exactly k. The k-True-Variables gadget consists of a single unbounded indegree vertex with x_i as predecessors for all i. After passing through the All False gadget, k variable gadgets must be switched from the False position to the True position by moving $3k$ pebbles from x'_i to x_i and using k transitions to move k red pebbles to $\overline{x'_i}$. As we will show in the next few gadgets, k transitions must be used to pebble $\overline{x'_i}$ nodes with red pebbles. See Fig. 9-9 for an example.

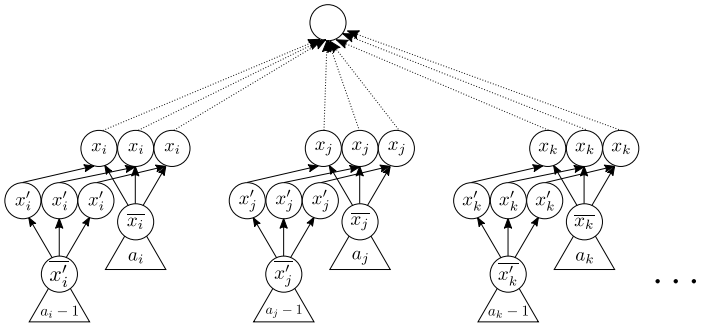


Figure 9-9: k-True gadget connects to all x_i for all i.

Lemma 9.4.3. *At the end of Phase 2, exactly k variable gadgets are set in the true configuration.*

Proof. In order to prove this lemma, we first prove the following two claims.

Claim 9.4.4. *At least $3k$ pebbles must be removed from x'_i and used to pebble x_i with red pebbles and k pebbles must be removed from $\overline{x_i}$.*

Proof. In order to pebble the k-True-Variables gadget, all the nodes that are incident to the central node must be pebbled. Given that we only have $3n - 4k$ pebbles remaining after pebbling all the variable gadgets in Phase 1 and the All-False gadget, we must use $4k$ pebbles from the variable gadgets in order to pebble the remaining $4k$ slots. The pebbling strategy proceeds as follows.

First, pebble $3n - 4k$ x_i's. Of these, remove $4k$ pebbles from the corresponding x_i' and $\overline{x_i}$. Using these removed pebbles, pebble the remaining k variables (i.e. the remaining $4k$ x_i's).

Since the $2k + 1$ pebbles from the All-False gadget are needed to pebble the k-True gadget, we cannot remove these pebbles. Thus, the smallest number of pebbles we need to remove is $4k$ from the variable gadgets. □

Claim 9.4.5. *No pyramid gadgets in the variable gadgets may be repebbled using $\leq 2k$ transitions in Phase 2.*

Proof. In order to pebble the k-True-Variables gadget, the $2k + 1$ nodes pebbled during the All-False phase must remain pebbled with red pebbles. Even if all red pebbles were removed from all x_i' nodes, the number of red pebbles available is not enough to pebble the pyramid with the smallest cost among all pyramids in variable gadgets. In order to obtain the $2k + 1$ red pebbles, at least $2k + 1$ transitions need to be used to turn the red pebbles on the All-False gadget to blue which exceeds our limit of $2k$ transitions. □

Claim 9.4.6. *At least k pebbles must be turned from blue to red on $\overline{x_i'}$ using k transitions.*

Proof. The Pebble Sink Path gadget uses $3n - 4k - 6$ pebbles and the clause gadgets use 5 pebbles each; therefore, $3n - 4k$ of the pebbles that are used to pebble the k-True-Variables gadget (or the variable gadgets) must be removed from the gadgets and used to pebble the pebble sink path. Therefore a total of $4n$ pebbles remain on the variable gadgets. By Lemma 9-10, each variable gadget must be set in either the true or false configuration (in other words, each variable gadget either has all x_i nodes pebbled with red pebbles or $\overline{x_i'}$ pebbled with a red pebble *and* all x_i' nodes pebbled with red pebbles or $\overline{x_i}$ pebbled with a red pebble). In order to transform a variable gadget from the false configuration to the true configuration, at least one transition must be used to pebble $\overline{x_i'}$ with a red pebble provided that the pyramid under $\overline{x_i'}$ cannot be repebbled. We showed this in Claim 9.4.5 that in order to repebble this pyramid, more than $2k$ transitions are needed.

Therefore, the only way to change the truth value of a variable gadget is to use 1 transition per gadget. Suppose that a red pebble is removed from a x_i' node. Then, the node $\overline{x_i'}$ must be pebbled by Lemma 9-10. Furthermore, if a node is removed from $\overline{x_i}$, then all x_i nodes must be pebbled which implies that $\overline{x_i'}$ must also be pebbled. Therefore, the minimum number of variables that need to be switched from false to true is k since $4k$ pebbles can be removed from k variables and switched from false to true using the strategy provided by Claim 9.4.4. The k transitions are used to turn blue pebbles on k $\overline{x_i'}$ nodes to red.

□

Claim 9.4.7. *The minimum number of transitions that are needed for Phase 2 is k.*

Proof. This follows directly from Claim 9.4.6. □

Therefore, since all variables either have x_i' and $\overline{x_i}$ or x_i and $\overline{x_i'}$ pebbled, all variables are set to either true or false. Furthermore, as the claims show, at most k variables are set to true. □

3-or-None Gadget

The 3-or-None gadgets are used to ensure that every variable either has 3 pebbles on each x_i or x_i' or none on them. This is to ensure that the player cannot cheat by using less than 3 pebbles to set either x_i' or x_i true. A 3-or-None gadget is created for each variable. The 3-or-None gadget consists of sets of 2 vertices one picked from x_i and the other picked from x_i'. All such pairings are connected to a path with vertices of indegree 5 (the other vertices are roots) so that only one pebble is allowed to go through the path. See Fig. 9-10 for an example of a 3-or-None gadget. This gadget can be pebbled using 5 pebbles. However, more pebbles will not ensure that the gadget can be pebbled if it does not satisfy the invariant as stated in Lemma 9.4.10. In other words, more pebbles does not guarantee that the gadget can be pebbled without using any transitions. Attached to each node of the 3-or-None gadget are $3n-4k-3$ roots that have outgoing edges to the nodes in the path in the gadget. The All-False termination node is also connected to every node in the path of the 3-or-None gadget.

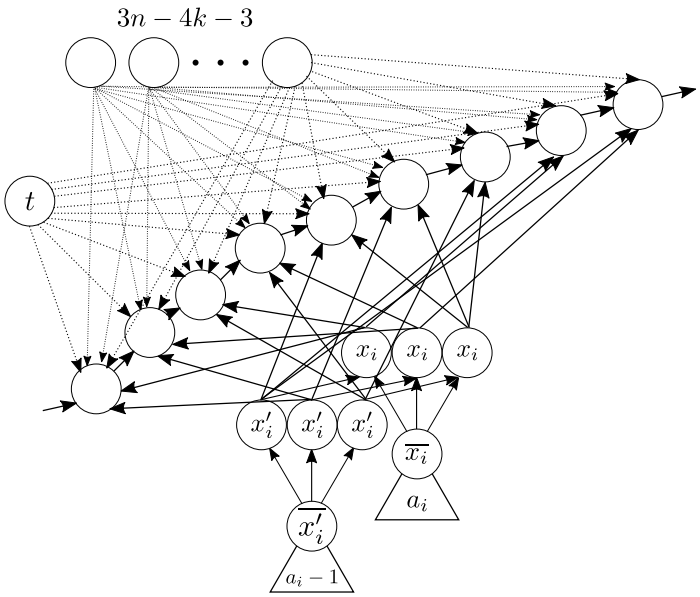


Figure 9-10: 3-or-None gadget. One is created for every variable.

Lemma 9.4.8. *For every variable gadget that does not follow the condition specified in Lemma 9.4.10 and contains less than 6 red pebbles, at least two transitions are required to satisfy the gadget.*

Proof. Suppose that less than 6 red pebbles are on a variable gadget and no pair of components x_i and $\overline{x_i'}$ or x_i' and $\overline{x_i}$ are pebbled with red pebbles, then the distribution of red

pebbles are ones that partially fill in some x_i or x_i' and does not pebble $\overline{x_i}$ or $\overline{x_i'}$. Therefore, in order to pebble the x_i or x_i' that does not contain a red pebble, we have to use two transitions to turn the blue pebble on $\overline{x_i}$ or $\overline{x_i'}$ to red. □

Lemma 9.4.9. *Suppose some variable gadgets are set in the configuration where $\overline{x_i}$ and $\overline{x_i'}$ are pebbled with red pebbles, then the number of transitions needed to pebble both the All-False gadget and the 3-or-None gadget is greater than $2k$.*

Proof. Suppose that b variable gadgets have red pebbles on $\overline{x_i}$ and $\overline{x_i'}$. During Phase 2, $4k$ red pebbles must be removed and moved to the x_i nodes. In order for this to occur, some number of red pebbles are removed. For any pebble placed during Phase 1 removed, at least 2 transitions must be used to reset the value of the variable unless only the red pebble on $\overline{x_i}$ is removed and no other red pebbles are removed from the gadget. For each of the variable gadgets where the red pebble on $\overline{x_i}$ is removed, in order for the variable gadget to have red pebbles on $\overline{x_i}$ and $\overline{x_i'}$, two transitions must be spent on placing a red pebble on $\overline{x_i}$ during Phase 2. We know that having a variable gadget in the configuration $\overline{x_i}$ and $\overline{x_i'}$ saves 2 pebbles. However, in order to achieve this configuration, we must spend 4 transitions per gadget in Phase 2 which does not make up for the amount that is saved by this configuration. The removal of any other red pebble from a variable gadget requires at least two transitions; therefore, the optimal removal number is 4 (instead of 3 which would lead to a $\overline{x_i}$ and $\overline{x_i'}$ configuration). □

Lemma 9.4.10. *In order to satisfy all 3-or-None gadgets using at most $2k$ transitions, the only possible configurations for all pebbles must be placed in one of the two pairs of components: $\overline{x_i'}$ and $\overline{x_i}$ or x_i or x_i'.*

Proof. By Lemma 9.4.9, pebbles cannot be placed in the configuration $\overline{x_i}$ and $\overline{x_i'}$ using at most $2k$ transitions. Therefore, in order for a red pebble placement to satisfy the corresponding 3-or-None gadget without using 2 additional transitions, the must be placed in the pairs given in the lemma or red pebbles must be on all nodes x_i and x_i'. Suppose that a variable gadget has this configuration. Then, two pebbles will be removed from some other variable gadget. Since no variables can be in the configuration $\overline{x_i}$ and $\overline{x_i'}$ by Lemma 9.4.9, if a variable is not set in a configuration, then two transitions are used. By Lemma 9.4.6, at least k variables must be set to false at the end of Phase 2. Therefore, each variable must be in the configurations as stated in this lemma in order to satisfy all 3-or-None gadgets. □

The remaining gadgets may be pebbled with red pebbles without using any transitions.

Pebble Sink Path Gadget

The Pebble Sink Path Gadget is used to take up $3n - 4k - 6$ pebbles that were used in the k-True-Variables gadget (and were left over after passing through the gadget) leaving only 5 pebbles for the remaining parts of the winning path. The Pebble Sink Path occurs directly after the clause gadgets path and must be pebbled before the clause gadgets are

pebbled. This sink path consists of $3n - 4k - 6$ pyramid gadgets of successively smaller value starting from $3n - 4k - 1$. See Fig. 9-11 for an example.

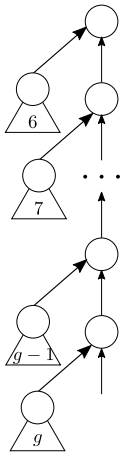


Figure 9-11: Pebble sink that captures $3n - 4k - 6$ pebbles leaving 5 pebbles to be used in the clauses. Here $g = 3n - 4k - 1$.

Clause Gadget

After the set of 3-or-None gadgets comes the Clause gadgets which are used to ensure that the 3SAT clauses are satisfied by the assignments. The clause gadget can only be pebbled with the 5 extra pebbles that remain after the pebble sink has been pebbled. Given that all $2k$ transitions are used in Phase 2 and/or the pebbling 3-or-None gadgets phase, no transitions can be spent in Phase 3 or pebbling the clause gadgets. The output of the clause path must be connected to each vertex of the Pebble Sink Path gadget. See Fig. 9-12.

Finally, the target vertex can be pebbled with a red pebble if and only if all previous gadgets are pebbled according to the necessary rules and conditions. The $2k + 1$ nodes from the All-False gadget are also predecessors of this target vertex.

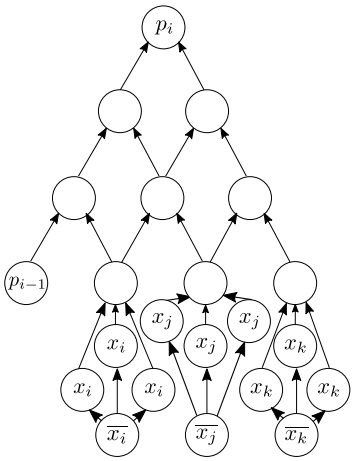


Figure 9-12: Clause gadget.

For an example reduction, see Fig. 9-13. The target vertex that must be pebbled is the one colored blue.

Lemma 9.4.11. *The clause gadget can be pebbled with* 5 *pebbles (not including the red pebble on p_{i-1}) if and only if at least one of the variable gadgets that connects to it is set in the true configuration.*

Proof. If at least one literal is true in the clause gadget, then we can pebble the gadget in the following way. First, pebble all the other literals that are not set to true. Pebbling the other literals requires at most 4 pebbles. Finally, pebble the bottom layer of the pyramid for the literal that is set to true. Then, the pyramid can be pebbled using the method described in Lemma 9.3.3 using five pebbles once the bottom layer of the pyramid has been pebbled.

Based on the proof provided in Lemma 9.3.3, the cost of pebbling the pyramid in each clause gadget is 5. In order for a red pebble to be placed on the pyramid in the clause gadget (aside from the red pebble on p_{i-1}), 3 red pebbles must be used on all literals that are not set in the true configuration. Suppose that 2 pebbles are currently on the pyramid at some time t, then, to pebble the last literal (since it is not set to true) takes at least 4 red pebbles. At the time the last literal is pebbled, as least 2 red pebbles must already be on the pyramid (otherwise, the literal is not the last literal to be pebbled in the gadget). Therefore, if the clause gadget can be pebbled using 5 red pebbles (not including the red pebble on p_{i-1}), then the clause is satisfiable. □

9.4.3 Red-Blue Pebbling is W[1]-hard

In this section, we prove that red-blue pebbling parameterized by the number of transitions is W[1]-hard using the gadgets as specified in Section 9.4.2. An example construction is shown in Fig. 9-13. The order of the pebbling is given as the following. First, the variable gadgets are pebbled during Phase 1 of the pebbling which pebbles each pyramid gadget in each variable with a red pebble. Then, the All-False gadget is pebbled which results in all x_i' and $\overline{x_i}$ nodes being pebbled with red pebbles. During Phase 2 of pebbling the variable gadgets, k variables are switched from the false configuration to the true configuration. This in total uses the entirety of the allowed $2k$ transitions. To ensure the $2k$ transitions are used in this phase, the 3-or-None gadgets are pebbled using only 5 red pebbles and no transitions. After the 3-or-None gadgets are pebbled, we pebble the Pebble Sink Path gadget which consumes $3n - 4k - 6$ pebbles. The clause gadgets are pebbled with the remaining 5 pebbles not used in the pebble sink path. Finally, the variable gadgets are pebbled completely using all the pebbles during Phase 3 and the target node as indicated in Fig. 9-13 pebbled with a red pebble. The total number of red pebbles necessary is $r = 7n - 2k + 1$ and the total number of transitions is $t = 2k$.

Theorem 9.4.12. *The red-blue pebble game parameterized by the number of transitions k is W[1]-hard.*

Proof. We first show that our reduction is a valid FPT reduction. As defined above, our reduction is a polynomial time reduction in terms of n, m, and k. The number of nodes

created is $O(n+m+k)$ and the number of edges is at most the square of this amount. The number of transitions is determined by the function $f(k) = 2k$ where k is the parameter in Weighted 3-CNF Satisfiability.

Now we will prove that a solution exists in our construction if and only if a solution exists for the expression φ. Suppose a solution exists for φ, then one can set the variables in the construction to have the truth value given by the solution to φ. We can set all the variables to their corresponding truth values using at most $2k$ transitions. Furthermore, we can pebble the remainder of the construction using the prescribed number of red pebbles.

As we proved in Lemma 9.4.10, the number of transitions that must be used after pebbling the 3-or-None gadgets is $2k$. Therefore, the remainder of the construction must be pebbled using red pebbles and no transitions. We proved in Lemma 9.4.11 that the clauses can only be pebbled using 5 red pebbles if they are satisfiable. Since the Pebble Sink Path gadget is pebbled after the 3-or-None gadgets are pebbled, they must be pebbled at the time when the clause gadgets are pebbled, leaving only 5 free red pebbles to pebble the clause gadgets. Finally, the path leading up to the blue pebble can be pebbled if all the clauses are successfully pebbled. Therefore, this is a valid reduction from Weighted 3-CNF SAT and the problem is W[1]-hard. □

9.5 Open Problems

We conclude this chapter with several open questions:

1. Are red pebbling number and minimum number of transitions hard to approximate?
2. Does there exist FPT algorithms for restricted class of graphs (such as bounded width graphs)?
3. Is finding the red pebbling number W[1]-hard? Recall in Section 9.4 that we proved W[1]-hardness only for transitions, not for number of red pebbles.

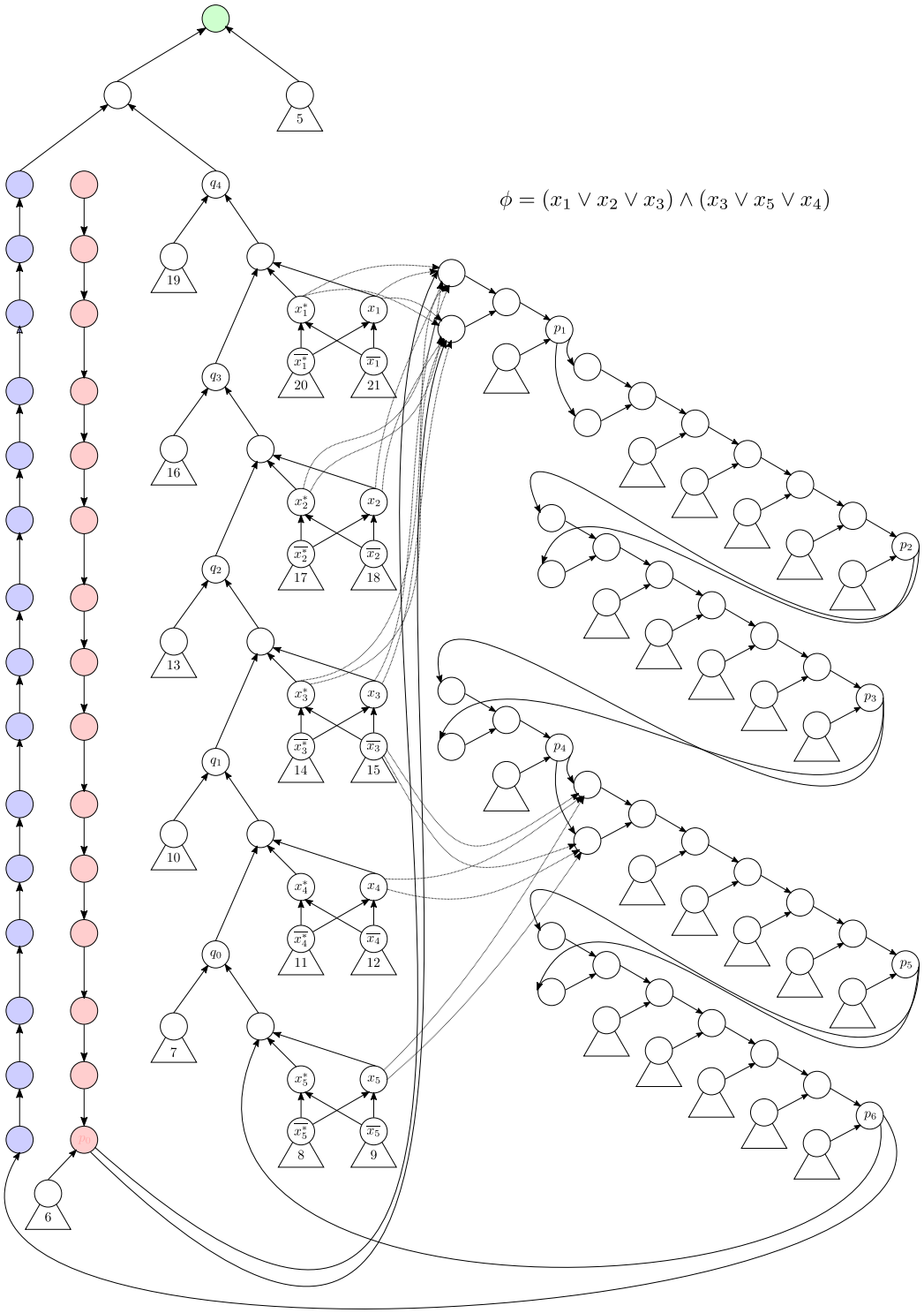


Figure 9-6: Example construction given $\varphi = (x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee x_5 \vee x_4)$. Blue nodes represent the pebble hold nodes and red nodes represent the pebble sink path. The green node is the target node that needs to be pebbled in the end. Note that many of the edges for variable nodes have been omitted for clarity.

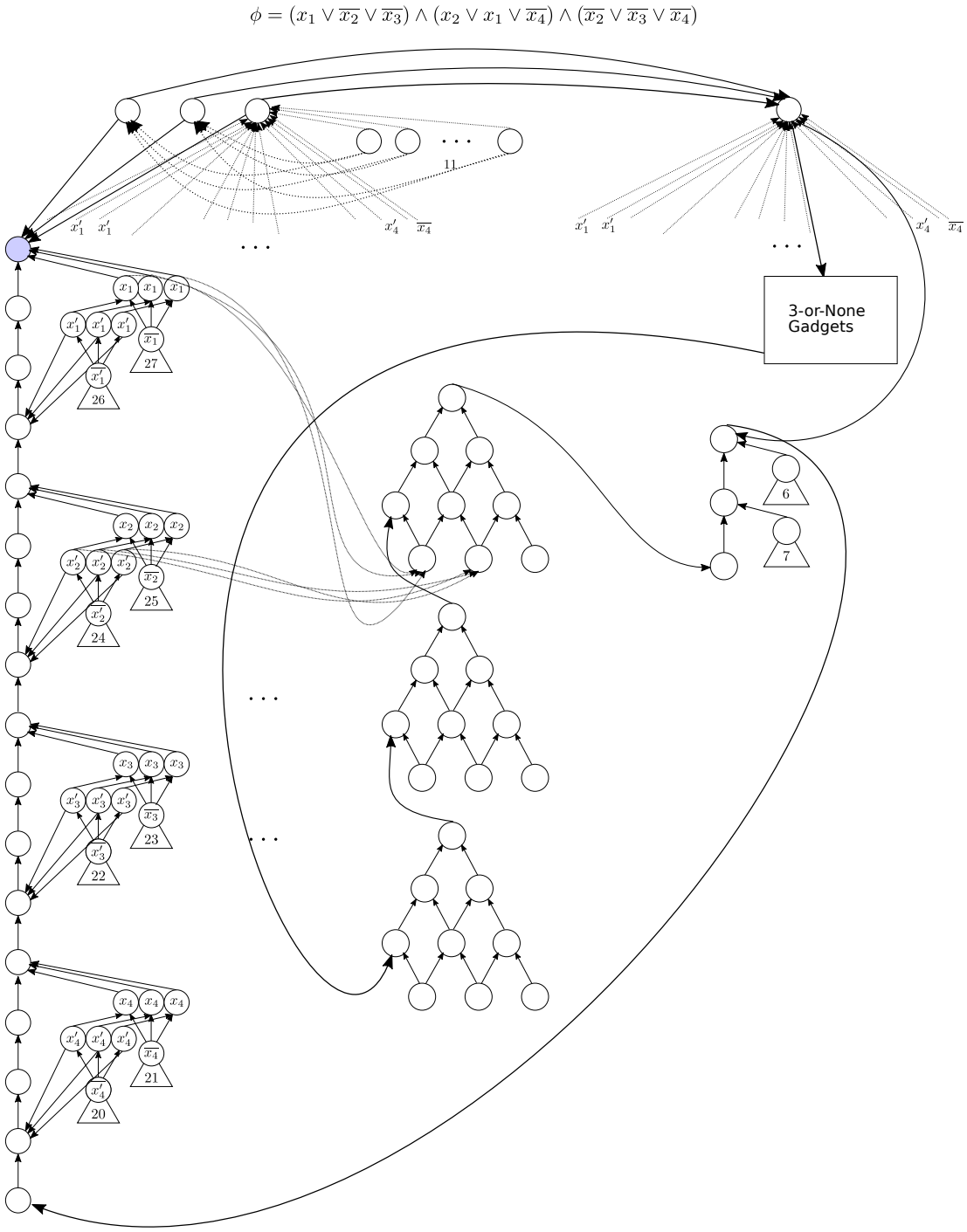


Figure 9-13: Example W[1]-hardness reduction for the red-blue pebble game. The vertex colored blue is the vertex that must be pebbled at the end and can only be pebbled if and only if the 3SAT instance has a solution that sets exactly k variables to True and uses at most $2k$ transitions.

Chapter 10

Static-Memory-Hard Hash Functions from Pebbling

This chapter presents results from the paper titled, "Static-Memory-Hard Functions, and Modeling the Cost of Space vs. Time" that the thesis author coauthored with Thaddeus Dryja and Sunoo Park [DLP18]. This paper appeared in the Theory of Cryptography Conference (TCC) 2018.

10.1 Introduction

Pebble games were originally formulated to model time-space tradeoffs by a game played on DAGs. Generally, a DAG can be thought to represent a computation graph where each node is associated with some computation and a pebble placed on a node represents performing the computation and saving the result of its computation in memory. Thus, the number of pebbles represents the amount of memory necessary to perform some set of computations. The natural complexity measures to optimize in this game is the minimum number of pebbles used, as well as the minimum amount of time it takes to finish pebbling all the nodes; these goals correspond with minimizing the amount of memory and time of computation.

Pebble games were first introduced to study programming languages and compiler construction [PH70] but have since then been used to study a much broader range of tasks such as register allocation [Set75], proof complexity [AdRNV17, Nor12], time-space tradeoffs in Turing machine computation [Coo73, HPV77], reversible computation [Ben89], circuit complexity [Pot17], and time-space tradeoffs in various algorithms such as FFT [Tom81], linear recursion [Cha73, SS79b], matrix multiplication [Tom81], and integer multiplication [SS79a] in the RAM as well as the external memory model [JWK81]. To see a more comprehensive survey of the results in pebbling up to the last couple of years, see [Pip82] up to the 1980s and [Nor15] up to 2015.

The relationship between pebbling and cryptography has been a subject of research interest for decades, which has enjoyed renewed activity in the last few years. A series of recent works [AB16, ABH17, ABP17a, ABP17b, AS15, AT17, ACP+16, AAC+17, BZ16, BZ17] has deepened our understanding of the notion of *memory-hardness* in cryptography, and

has shown memory-hardness to have intricate connections with the theory of graph pebbling.

Memory-hard functions (MHFs) have garnered substantial recent interest as a security measure against adversaries trying to perform attacks at scale, particularly in the ubiquitous context of password hashing. Consider the following scenario: hashes of user passwords are stored in a database,[1] and when a user enters a password p to log in, her computer sends $H(p)$ to the database server, and the server compares the received hash to its stored hash for that user's account. For a normal user, it would be no problem if hash evaluation were to take, say, one second. An attacker trying to guess the password by brute-force search, on the other hand, would try orders of magnitude more passwords, so a one-second hash evaluation could be prohibitively expensive for the attacker.

The evolution of password hashing functions has been something of an arms race for decades, starting with the ability to increase the number of rounds in the DES-based unix `crypt` function to increase its computation time—a feature that was used for exactly the above purpose of deterring large-scale password-cracking. Attackers responded by building special-purpose circuits for more efficient evaluation of `crypt`, resulting in a gap between the evaluation cost for an attacker and the cost for an honest user.[2]

A promising approach to mitigating this asymmetry in cost between hash evaluation on general- and special-purpose hardware is to increase the use of *memory* in the password hashing function. Memory is implemented in standardized ways which have been highly optimized, and memory chips are widely regarded to be an interchangeable commodity. Commonly used forms of memory — whether on-die SRAM cache, DRAM, or hard disks — are already optimized for the purpose of data I/O operations; and while there is active research in improving memory access times and costs, progress is and has been relatively incremental. This state of affairs sets up a relatively "even playing field," as the normal user and the attacker are likely to be using memory chips of similar memory access speed. While an attacker may choose to buy more memory, the cost of doing so scales linearly with the amount purchased.

The designs of several MHFs proposed to date (e.g., [Per09, AS15, AB16, ACP+16, ABP17a]) have proven memory-hardness guarantees by basing their hash function constructions on DAGs, and using space complexity bounds from graph pebbling. Definitions of memory-hardness are not yet unified in this somewhat nascent field, however — the first MHF candidate was proposed only in 2009 [Per09] — and the guarantees proven are with respect to a range of definitions. The "cumulative complexity"-based definitions of [AS15] have enjoyed notable popularity, but some of their shortcomings have been pointed out by subsequent work proposing alternative more expressive measures, in particular, [ABP17b, AT17].

Our contribution We observe *two* significant and practical considerations not analyzed by existing models of memory-hardness, and propose new models to capture them, accompanied by constructions based on new hard-to-pebble graphs. Our main contribution

[1] In practice, the password should first be concatenated with a random user-specific string called a *salt*, and then hashed. The salt is stored in the database alongside the hash to deter *dictionary attacks*.

[2] E.g., [CB02] discusses FPGA-based attacks on DES.

is two-fold, as described in (1) and (2) below. We also provide an additional contribution of separate interest, described in (3).

1. ***Static-memory-hardness.*** Existing measures of memory-hardness only account for *dynamic* memory usage (i.e., memory read/written at runtime), and do not consider *static* memory usage (e.g., memory on disk). Among other things, this means that memory requirements considered by prior models are inherently upper-bounded by a hash function's runtime; in contrast, counting static memory would potentially allow quantification of much larger memory requirements, decoupled from the honest evaluator's runtime.
   We propose a new definition of *static-memory-hard* function (SHF) (Definition 10.4.2), and present two SHF constructions based on pebbling. To prove static-memory-hardness, we define a new pebble game called the *black-magic pebble game* (Definition 10.2.3), and prove properties of the space complexity of this game for new graphs (Graph Constructions 10.5.4 and 10.5.15). Graph Construction 10.5.15 gives rise to an SHF with a better asymptotic guarantee (same space usage but sustained over more time), whereas Graph Construction 10.5.4 yields an SHF with the advantage of simplicity in practice. Informal theorems stating the constructions' static-memory-hardness guarantees are given in Section 10.1.3 and formal theorems are in Section 10.5. In Section 10.7, we discuss our prototype implementation based on Graph Construction 10.5.4.
   We emphasize that static memory requirements are *complementary* to dynamic memory requirements: neither can replace the other, and to deter large-scale password-cracking attacks, a hash function will benefit from being *both* dynamic-memory-hard and static-memory-hard.
2. ***Modeling nonlinear cost of space vs. time.*** Existing measures of memory-hardness implicitly assume a linear trade-off between the costs of space and time. This model precludes situations where the relative costs of space and time might be more unbalanced (e.g., quadratic or cubic). We demonstrate that this modeling limitation is significant, by giving an example where adversaries facing asymptotically different space-time cost tradeoffs would in fact employ *different strategies*. Then, to remedy this shortcoming, we define *graph-optimal* variants of memory-hardness measures (in Section 10.2) that *explicitly* model the relative cost of space and time. These can be seen as extending the main memory-hardness measures in the literature (namely, *cumulative complexity* and *sustained memory complexity*). We prove bounds on the new measure as elaborated in Section 10.1.3.
3. We give the first graph construction that is tight, up to $\log\log n$-factors, to the optimal cumulative complexity that can be achieved for any graph (upper bound due to [ABP17a, ABP17b]).

   Informal version of Theorem 10.6.23. There exists a family of graphs where the cumulative complexity of any graph with n nodes in the family is $\Theta\left(\frac{n^2 \log\log n}{\log n}\right)$ which is asymptotically tight to the upper bound of $\Theta\left(\frac{n^2 \log\log n}{\log n}\right)$ given in [ABP17a, ABP17b] in the sequential pebbling model. Notably we can show a family of constant in-degree graphs exists that satisfy this property.

Next, Section 10.1.1 gives a brief background on graph pebbling, Section 10.1.2 gives discussion on memory-hardness measures and related work, and Sections 10.1.3 and 10.1.3 give more detailed high-level overviews of our SHF contribution and nonlinear space-time tradeoff model (items (1) and (2) above), respectively.

10.1.1 Background on graph pebbling

The standard *black pebble game* is parametrized by a directed acyclic graph (DAG) and a special subset of its nodes (called the *target set*). In the game, an unlimited supply of "pebbles" is made available to a player, who must place and remove pebbles on the nodes of the DAG in a sequence of moves according to the following two rules.

1. A pebble may be placed or moved onto a node only if all of its predecessors have already been pebbled. (In particular, pebbles may be placed on source nodes at any time.)
2. Any pebble can be removed from the graph at any time.

The goal of the game is to arrive at a state where every node in the target set has been covered by a pebble at least once in the pebbling of the graph. Often, the target set is the set of the sink nodes.

The pebbling literature, starting with [PH70, Set75, Coo73, HPV77], has established a number of complexity measures describing the complexity of pebbling: e.g., measuring the minimum number of pebbles that must be used to achieve a complete pebbling, or the minimum number of moves needed. In the literature, there are several variants of the game, including sequential and parallel (depending on whether many pebbles can be placed in a single move), and versions where other different types of pebbles are used (such as the red-blue pebble game [JWK81] and the black-white pebble game [CS74]). In this work, our results are stated and proven in the context of constant in-degree graphs since most graphs in the real-world have constant in-degree; however, our results extend straightforwardly to non-constant in-degree graphs.

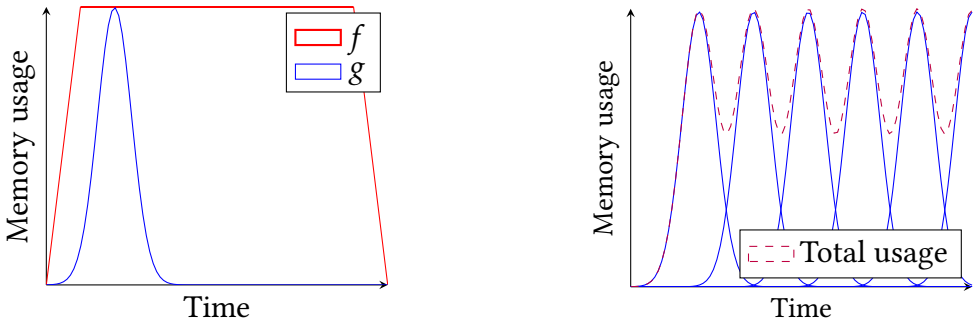
Graph pebbling and memory-hardness Graph pebbling algorithms can be used to construct hash functions in the (parallel) random oracle model. This paradigm has been used by prior constructions of memory-hard hashing [AS15] as well as other prior works [DKW11].

Informally, the idea to "convert" a graph into a hash function is to associate with each node v a string called a *label*, which is defined to be $\mathcal{O}(v, \mathsf{Pred}(v))$ where $\mathcal{O}$ is a random oracle and $\mathsf{Pred}(v)$ is the list of labels of predecessors of v. For source nodes, the label is instead defined to be $\mathcal{O}(v, \zeta)$ for a string ζ which is an input to the hash function. The output of the hash function is defined to be the list of labels of target nodes. Intuitively, since the label of a node cannot be computed without the "random" labels of all its predecessors, any algorithm computing this hash function must move through the nodes of the graph according to rules very similar to those prescribed by the pebbling game; and therefore, the memory requirement of computing the hash function roughly corresponds to the pebble requirement of the graph. Thus, proving lower bounds on the pebbling complexity of graph families has useful implications for constructing provably memory-hard functions.

In our setting, in contrast to previous work, we employ a variant of the above technique: the string ζ is a fixed parameter of our hash function, and the input to the hash function instead specifies the indices of the target nodes whose labels are to be outputted.

10.1.2 Discussion on memory-hardness measures and related work

The original paper proposing memory-hard functions [Per09] suggested a very simple measure: the minimum amount of memory necessary to compute the hash function. It was subsequently observed that a major drawback of this measure is that it does not distinguish between functions f and g with the same peak memory usage, even if the peak memory lasts a long time in evaluating f and is just fleeting in evaluating g (Figure 10-1a). This is significant as the latter type of function is much better for a password-cracking adversary. In particular, pipelining the evaluation of the latter type of function would allow reuse of the same memory for many function evaluations at once, effectively reducing the adversary's amortized memory requirement by a factor of the number of concurrent executions (Figure 10-1b).



(a) Functions with the same peak memory usage (b) Pipelined evaluations of g (reusing memory)

Figure 10-1: Limitations of peak memory usage as a memory-hardness measure

Cumulative complexity [AS15] put forward the notion of *cumulative complexity* (CC), a complexity measure on graphs. CC was adopted by several subsequent works as a canonical measure of memory-hardness. CC measures the *cumulative* memory usage of a graph pebbling function evaluation: that is, the sum of memory usage over all time-steps of computation. In other words, this is the area under a graph of memory usage against time. CC is designed to be very robust against amortization, and in particular, scales linearly when computing many copies of a function on different inputs. This is a great advantage compared to the simpler measure of [Per09], which does not account well for an amortizing adversary (as shown in Figure 10-1).

Depth-robust graphs More recently, [AB16, ABP17a] proved bounds on optimal CC of certain graph families. They showed that a particular graph property called *depth-robustness* suffices to attain optimal CC (up to polylog factors–the CC of any graph with bounded in-degree is upper bounded by $O\left(\frac{n^2 \log\log n}{\log n}\right)$ [AB16, ABP17b]). An (r, s)-depth-robust graph is one where there exists a path of length s even when any r vertices are removed. Intuitively, this captures the notion that storing any r vertices of the graph will not shortcut the pebbling in a significant way.

Sustained memory complexity Very recently, Alwen, Blocki, and Pietrzak [ABP17a] proposed a new measure of memory complexity, which captures not only the cumulative memory usage over time (as does CC), but goes further and captures the amount of time for which a particular level of memory usage is sustained. Our SHF definition also captures *sustained* memory usage: we propose a definition of capturing the duration for which a given amount of memory is required, designed to capture static as well as dynamic memory requirements. By the nature of static memory, it is especially appropriate in our setting to consider (and maximize) the amount of time for which a static memory requirement is *sustained*.

Bandwidth-hard function In a concurrent work, Blocki, Ren, and Zhou [BRZ18], building upon the previous work of Ren and Devadas [RD17], studied the bandwidth costs of different functions. In real computers, cache misses take up significantly more energy or bandwidth than computation within the cache. Given any computation that can be modeled as a DAG, the red-blue pebble game has been traditionally used to study the I/O-transition cost of the computation [JWK81]. In their work [BRZ18], they show an equivalence between lower bounds in the cost of the parallel red-blue pebble game and the bandwidth-hardness or energy cost of evaluation of functions. Furthermore, they analyze the bandwidth-hardness of some popular memory-hard functions and found that memory-hardness does indeed correlate with bandwidth-hardness. Their work differs from our work in that they do not look at static memory, instead looking at the dynamic memory generated at evaluation time of the function and the bandwidth-hardness of such a process.

Core-area memory ratio Previous works have considered certain hardware-dependent non-linearities in the ratio between the cost of memory and computation [BK15, AB16, RD17]. Such phenomena may incur a multiplicative factor increase in the memory cost that is dependent, in a possibly non-linear way, on specific hardware features. Note that *the non-linearity here is in the hardware-dependence*, rather than the space-time tradeoff itself. In contrast, our new models are more expressive, in that they make configurable the asymptotic tradeoff between space and time (by a parameter α which is in the exponent, as detailed in Definition 10.2.21) in an application-dependent way. This versatility of configuration targets applications where the trade-off may realistically depend on arbitrary and possibly exogenous space/time costs, and thus contrasts with metrics tailored for a specific hardware feature, such as core-memory ratio.

Towards a general theory of moderately hard functions Most recently, Alwen and Tackmann [AT17] proposed a more general (though not comprehensive) framework for defining desirable guarantees of "moderately hard functions," i.e., functions that are efficient to compute but somewhat hard to invert. Their work points out a number of drawbacks of prior measures such as those described above. Notably, many of the prior measures characterized the hardness of *computing* the function with an implicit assumption that this hardness would translate to the hardness of *inverting* the function (as it would indeed in the case of a brute-force approach to inversion). In other words, these measures implicitly assume that the hash function in question "behaves like a random oracle" in the sense that brute-force inversion is the optimal approach.

10.1.3 Our contributions in more detail

To prove static-memory-hardness, we define a new pebble game called the *black-magic pebble game* (Definition 10.2.3), and prove properties of the space complexity of this game for new graphs (Graph Constructions 10.5.4 and 10.5.15).

The black-magic pebble game may additionally be of independent interest for the pebbling literature. Indeed, a pebble game used to analyze security of *proofs of space* [DFKP15] can be viewed as a non-adaptive[3] version of the black-magic pebble game in which the target node set is sampled from a distribution by a challenger.

Based on our new graph constructions, we construct SHFs with provable guarantees on sustained memory usage, as follows. Graph Construction 10.5.15 gives a better asymptotic guarantee (same space usage but sustained over more time), whereas Graph Construction 10.5.4 has the advantage of simplicity in practice. In Section 10.7, we discuss our prototype implementation based on Graph Construction 10.5.4.

Static-memory-hard functions (SHFs)

Prior memory-hardness measures make a modeling assumption: namely, that the memory usage of interest is solely that of memory dynamically generated at run-time. However, static memory can be costly for the adversary too, and yet it is not taken into account by existing measures such as CC. Intuitively, it can be beneficial to design a function whose evaluation requires keeping a large amount of static memory on disk (which may be thought to be produced in a one-time initial setup phase). While not all the static memory might be accessed in any given evaluation, the "necessity" to maintain the data on disk can arise from the idea that an adversary attempting to evaluate the function on an arbitrary input while having stored a lesser amount of data would be forced to *dynamically* generate comparable amounts of memory. Note that the resulting *dynamic* memory requirements could be orders of magnitude larger (say, gigabytes) than the memory requirements of existing memory-hard function proposals, because unlike in prior memory-hardness models, here we have decoupled the memory requirement from the memory requirements of the honest evaluator.

[3]Here, "non-adaptive" means that all magic pebbles must be fixed at the start of the game rather than placed throughout the game.

We propose a new model and definitions for *static-memory-hard functions* (SHFs), in which we model static memory usage by oracle access to a large preprocessed string, which may be considered part of the hash function description. In particular, the preprocessed string can be public and known to the adversary — the important guarantee is that without storing (almost) all of it statically, the adversary will incur huge online memory requirements.

Definition (informal). We model a *static-memory-hard function family* as a two-part algorithm $\mathcal{H} = (\mathcal{H}_1, \mathcal{H}_2)$ in the parallel random oracle model, where $\mathcal{H}_1(1^\kappa)$ outputs a "large" string to which $\mathcal{H}_2$ has oracle access,[4] and $\mathcal{H}_2$ receives an input x and outputs a hash function output y. Informally, our hardness requirement is that with high probability, any *two-part* adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ must *either* have $\mathcal{A}_1$ output a large state (comparable to the output size of $\mathcal{H}_1$), *or* have $\mathcal{A}_2$ use large (dynamic) space.

We then give two constructions of SHFs based on graph pebbling. To prove static-memory-hardness, we define a new pebble game called the *black-magic pebble game* of which we give an overview in Section 10.1.3. Our simpler SHF construction is based on a family of tree-like "cylinder" graphs, which achieves memory usage proportional to the square root of the number of nodes, sustained over time proportional to the square root of the number of nodes. Furthermore, we give a better construction based on pebbling of a new graph family, that achieves better parameters: the same (square root) memory usage, but sustained over time proportional to the number of nodes.

Informal version of Theorem 10.5.29. The "cylinder graph" (Graph Construction 10.5.4) can be used to construct an SHF, $\mathcal{H} = (\mathcal{H}_1, \mathcal{H}_2)$, with static memory requirement $\Lambda \in \Theta((\kappa - \xi \log(q_2))\sqrt{n})$ (in bits) where n is the number of nodes in the graph, κ is a security parameter, q_2 is the number of oracles queries made by $\mathcal{H}_2$, and $\xi \in \omega(1)$. This means that any successful adversary using non-trivially less *static* memory than Λ must incur at least Λ *dynamic* memory usage for at least $\Theta(\sqrt{n})$ steps.

Informal version of Theorem 10.5.30. Graph Construction 10.5.15 can be used to construct an SHF, $\mathcal{H} = (\mathcal{H}_1, \mathcal{H}_2)$, with static memory requirement $\Lambda \in \Theta((\kappa - \xi \log(q_2))\sqrt{n})$ (in bits) where n, κ, q_2, and ξ are as described above. This means that any successful adversary using non-trivially less *static* memory than Λ must incur at least Λ *dynamic* memory usage for at least $\Theta(n)$ steps.

Static memory requirements are *complementary* to dynamic memory requirements: neither can replace the other, and to deter large-scale password-cracking attacks, a hash function will benefit from being *both* dynamic-memory-hard and static-memory-hard. In Section 10.4.1, we give a discussion of how, given a static-memory-hard function and a (dynamic-)memory-hard function, they can be concatenated to yield a "dynamic SHF" that inherits both the static memory requirement of the former and the dynamic memory requirement of the latter.

Implementation We have a prototype implementation of our "cylinder" SHF construction. The code is available on github at `https://github.com/adiabat/masshash`. A

[4] More precisely, $\mathcal{H}_2$ may adaptively query the value of $\mathcal{H}_1$'s output string at specific locations.

discussion of the implementation and its performance for different static memory sizes is given in Section 10.7.

Remarks about the static-memory model

On static vs. dynamic memory In the case of function performing lookups to a large, static memory table, the memory storing the table does not need to be writable. This may seem to point to an optimization for the attacker: produce a read-only memory chip which supports fast, random-access read queries, but omits the hardware needed for writing data, as it has been pre-programmed at the factory with the precomputed static table. However, in practice, this optimization seems implausible. In modern hardware, ROM chips have almost entirely disappeared; where they do still exist, they are used for their non-volatile storage properties (they retain data when power is lost, unlike most RAM), and are copied to RAM before being read from, due to the low speed of the ROM. Current development focuses almost exclusively on dynamic access memory which supports both reads and writes, so it is reasonable to believe that an attacker would need to use this type of hardware; switching to ROM would likely increase costs and slow down access to the static table.

Static and dynamic memory requirements are thus incomparable, and both are useful to deter a password-cracking adversary.

Alternative application: bounded retrieval ("big-key") model As already stated above, the preprocessed string in our setting is assumed to be public, and our static-memory-hardness guarantees hold assuming the adversary knows the string. This is useful as it allows defining a single hash function accessible to all parties in a system, like a random oracle: one could imagine a standards body like NIST simply publishing a set of parameters defining a fixed hash function with a fixed "preprocessed string." One informal way to think of this is that the preprocessed string is part of the description of a fixed hash function. A single hash function accessible to everyone is particularly useful for certain applications such as checksums, where many parties in a distributed network may need to compute the same hash function.

In some other applications, however, hash function *families* may suffice or be more appropriate, i.e., where each party samples a function from the family for her own use, rather than every party using exactly the same function. In such applications, the preprocessed string can be considered the *seed* of a particular hash function from the family defined by $(\mathcal{H}_1, \mathcal{H}_2)$, and generated on a per-application basis. We observe one potential advantage of such a setup, inspired by the bounded retrieval [Dzi06, CLW06, CDD+07, ADW09, ADN+10, ADW09] ("big-key" [BKR16]) model.[5]: to make hash function evaluation more difficult for (e.g., password-cracking) adversaries. If the party using the hash function decides to keep the preprocessed string *secret*, then an adversary would have to exfiltrate almost all of the large preprocessed string from the honest user in order to be able to evaluate the hash function. As observed in the bounded retrieval literature, exfil-

[5]We cite the seminal papers that coined these terms, and note that there has been a rich literature on the topic since.

trating large quantities of data (say, gigabytes) can be much more costly for adversaries than exfiltrating smaller data items (such as secret keys).

Black-magic pebble game

We introduce a new pebble game called the *black-magic pebble game*. This game bears some similarity to the standard (black) pebble game, with the main difference that the player has access to an additional set of pebbles called *magic pebbles*. Magic pebbles are subject to different rules from standard pebbles: they may be placed anywhere at any time, but cannot be removed once placed, and may be limited in supply. The pebbling space cost of this game is defined as the maximum of the number of pebbles (both black and magic pebbles) on the graph at any time-step and the total number of magic pebbles used throughout the computation. Observe that while the most time-efficient strategy in the black-magic pebble game is always to pebble all the target nodes with magic pebbles in the first step, the most space-efficient strategy is much less clear.

Lower-bounds on space usage can be non-trivially different between the standard and black-magic pebble games. For example, if a graph has a constant number of targets, then magic pebbling space usage will never exceed a constant number of pebbles[6], whereas the standard pebbling space usage can be super-constant. In particular, it is unclear, in the new setting of magic pebbling, whether known upper-bounds on minimum pebbling space usage in the standard pebble game[7] are transferable to the magic pebble game. We prove in Section 10.5 that for layered graphs,[8] the best possible upper-bound on minimum space usage for the black-magic pebble game is $\Theta(\sqrt{n})$ (i.e. in other words, there exists a pebbling strategy for *any layered graph* such that at most $\Theta(\sqrt{n})$ pebbles–magic and/or black–are needed to pebble the target nodes). We present a simple layered graph construction that meets this upper bound in Section 10.5.1.

We leave determining both the lower bound and upper bound for minimum magic pebbling space usage in general graphs as an open question. An answer to this open question would be useful towards constructing better static-memory-hard functions using the paradigm presented herein.

Our proof techniques rely on a close relationship between black-magic pebbling complexity and a new graph property which we define, called *local hardness*. Local hardness considers black-magic pebbling complexity in a variant model where *subsets* of target nodes are required to be pebbled (rather than *all* target nodes, as in the traditional pebbling game), and moreover, a "preprocessing phase" is allowed, wherein magic pebbles may be placed on the graph in advance of knowing which target nodes are to be produced. This "preprocessing" aspect bears some resemblance to the *black-white pebbling game* [CS74], a variant of the standard pebbling game in which some limited number of *white* pebbles can be placed "for free," and the black pebbles must be placed according to the standard rules. However, our setting differs from the black-white pebbling game: while preprocessing and storing magic pebbles in advance can be viewed as analogous to

[6] In this case, you just place magic pebbles on all the targets.

[7] E.g., $\Theta\left(\frac{n}{\log n}\right)$ space is necessary to pebble certain classes of graphs in the standard pebble game [LT82].

[8] "Layered graph" is a standard term in the pebbling literature that refers to graphs whose nodes can be partitioned into a sequence of "layers" such that edges only go between vertices in adjacent layers.

placing white pebbles for free, the black-white pebbling game imposes restrictions on the *removal* of white pebbles from the graph, which are not present in our setting.

Capturing relative cost of memory vs. time

Existing measures such as CC and sustained memory complexity trade off space against time at a linear ratio. In particular, CC measures the minimal area under a graph of memory usage against time, over all possible algorithms that evaluate a function.[9]

However, different applications may have different relative cost of space and time. We propose and define a variant of CC called CC^{α}, parametrized by α which determines the relative cost of space and time, and observe that CC^{α} may be meaningfully different from CC and more suitable for certain application scenarios. For example, when memory is "quadratically" more expensive than time, the measure of interest to an adversary may be the area under a graph of memory squared against time, as demonstrated by the following theorem.

Informal version of Theorem 10.6.8. There exist graphs for which an adversary facing a linear space-time cost trade-off would in fact employ a *different pebbling strategy* from one facing a cubic trade-off.

It follows that when the costs of space and time are not linearly related, the CC measure may be measuring the complexity of *the wrong algorithm*, i.e., not the algorithm that an adversary would in fact favor. We thus see that our CC^{α} measure is more appropriate in settings where space may be substantially more costly than time (or vice versa). Moreover, our parametrized approach generalizes naturally to sustained memory complexity. We show that our graph constructions are invariant across different values of α, a potentially desirable property for hash functions so that they are robust against different types of adversaries.

Informal version of Theorem 10.6.13. Given any graph construction $G = (V, E)$, there exists a pebbling strategy that is less expensive asymptotically than any strategy using a number of pebbles asymptotically equal to the number of nodes in the graph for any time-space tradeoff.

10.1.4 Organization

Section 10.2 introduces *standard and new* graph pebbling definitions, Section 10.3 introduces computation in the parallel random oracle model (PROM) and its relation to our new *black-magic* pebbling complexity measures, Section 10.4 introduces our definition of a static-memory-hard function (SHF), Section 10.5 gives our SHF constructions and

[9] Of course, in general, memory usage and time depend on the specific computational model in discussion. However, in the stylized parallel random oracle model (PROM), on which all analyses in this chapter (and previous literature on MHFs) are based, time-steps and memory usage are well-defined. We refer to Section 10.3 for a description of the PROM.

proofs. Then, Section 10.6 presents our modeling and motivation of nonlinear cost tradeoffs between space and time, with upper and lower bounds in the new model. Finally, Section 10.7 discusses our prototype SHF implementation.

10.2 Pebbling definitions

A *pebbling game* is a one-player game played on a DAG where the goal of the player is to place pebbles on a set of one or more *target nodes* in the DAG.

In Section 10.2.1, we formally define two variations of the sequential and parallel pebble games: the *standard (black) pebble game* and the *black-magic pebble game*, the latter of which we introduce in this work. We also give the definitions of valid strategies and moves in these games. Then in Section 10.2.2, we define measures for evaluating the sequential and parallel pebbling complexity on families of graphs.

10.2.1 Standard and magic pebbling definitions

Definition 10.2.1 (Standard (black) pebble game)**.**

- ***Input:*** *A DAG, $G = (V, E)$, and a* target set *$T \subseteq V$. Define $\mathsf{Pred}(v) = \{u \in V : (u, v) \in E\}$, and let $S \subseteq V$ be the set of sources of G.*
- ***Rules at move i:*** *At the start of the game, no node of G contains a pebble. The player has access to a supply of* black pebbles*. Game-play proceeds in discrete moves, and P_i (called a "pebble configuration") is defined as the set of nodes containing pebbles after the ith move. $P_0 = \varnothing$ represents the initial configuration where no pebbles have been placed. Each move may consist of multiple actions adhering to the following rules.*[10]
  1. *A pebble can be placed on any source, $s \in S$.*
  2. *A pebble can be removed from any vertex.*
  3. *A pebble can be placed on a non-source vertex, v, if and only if its direct predecessors were pebbled at time $i-1$ (i.e., $\mathsf{Pred}(v) \in P_{i-1}$).*
  4. *A pebble can be moved from vertex v to vertex w if and only if $(v, w) \in E$ and $\mathsf{Pred}(w) \in P_{i-1}$.*
- ***Goal:*** *Pebble all nodes in T at least once (i.e., $T \subseteq \bigcup_{i=0}^{t} P_i$).*[11]

Remark 10.2.2. *At first glance, it may seem that rule 4 in Definition 10.2.1 is redundant as a similar effect can be achieved by a combination of the other rules. However, the application of rule 4 can allow the usage of fewer pebbles. For example, a simple two-layer binary tree (with three nodes) could be pebbled with two pebbles using rule 4, but would require three pebbles otherwise. Nordström [Nor15] showed that in sequential strategies, it is always possible to use one fewer pebble by using rule 4.*

[10] Multiple applications of rules 1, 2, and 3 can occur in a single move. E.g., multiple sources can be pebbled in a single move. Rule 4 can also be applied multiple times in a single move *for different pebbles*, but cannot be applied more than once to the same pebble (since, naturally, a single pebble cannot move to multiple locations).

[11] This goal statement corresponds to the notion of a *visiting pebbling* as defined in [Nor15]. This chapter will use this *visiting pebbling* notion throughout; however, we remark that an alternative notion of pebbling exists in the literature, called *persistent pebbling*, which requires that all the nodes in T be pebbled in the final configuration (i.e., $T \subseteq P_t$).

We note for completeness that while rule 4 is standard in the pebbling literature, not all the papers in the MHF literature include rule 4.

Next, we define the *black-magic pebble game* which we will use to prove security properties of our static-memory-hard functions.

Definition 10.2.3 (Black-magic pebble game)**.**

- ***Input:*** *A DAG $G = (V, E)$, a target set $T \subseteq V$, and* magic pebble bound $\mathfrak{M} \in \cup\{\infty\}$.
- ***Rules:*** *At the start of the game, no node of G contains a pebble. The player has access to two types of pebbles:* black pebbles *and up to $\mathfrak{M}$* magic pebbles*. Game-play proceeds in discrete moves, and $P_i = (M_i, B_i)$ is the pebble configuration after the ith move, where M_i, B_i are the sets of nodes containing magic and black pebbles after the ith move, respectively. $P_0 = (\varnothing, \varnothing)$ represents the initial configuration where no black pebbles or magic pebbles have been placed. Each move may consist of multiple actions adhering to the following rules.*
  1. *Black pebbles can be placed and removed according to the rules of the standard pebble game.*[12]
  2. *A magic pebble can be placed on and removed from any node, subject to the constraint that at most $\mathfrak{M}$ magic pebbles are used throughout the game.*
  3. *Each magic pebble can be placed at most once: after a magic pebble is removed from a node, it disappears and can never be used again.*
- ***Goal:*** *Pebble all nodes in T at least once (i.e., $T \subseteq \bigcup_{i=0}^{t} (M_i \cup B_i)$).*

Remark 10.2.4. *In the black-magic pebble game, unlike in the standard pebble game, there is always the simple strategy of placing magic pebbles directly on all the target nodes. At first glance, this may seem to trivialize the black-magic game. When optimizing for space usage, however, this simple strategy may not be favorable for the player: by employing a different strategy, the player might be able to use much fewer than T pebbles overall.*

Next, we define valid sequential and parallel strategies in these games.

Definition 10.2.5 (Pebbling strategy)**.** *Let G be a graph and T be a target set. A* standard (resp., black-magic) pebbling strategy for (G, T) *is defined as a sequence of pebble configurations, status $= \{P_0, \ldots, P_t\}$, satisfying conditions 1 and 2 below. status is moreover* valid *if it satisfies condition 3, and* sequential *if it satisfies condition 4.*

1. *$P_0 = \varnothing$.*
2. *For each $i \in [t]$, P_i can be obtained from P_{i-1} by a legal move in the standard (resp., black-magic) pebble game.*
3. *status successfully pebbles all targets, i.e., $T \subseteq \bigcup_{i=0}^{t} P_i$.*
4. *For each $i \in [t]$, P_i contains at most one vertex not contained in P_{i-1} (i.e., $|P_i \setminus P_{i-1}| \leq 1$).*

A black-magic pebbling strategy must satisfy one additional condition to be considered valid*:*

[12] The rules of the standard pebble game are a standard definition in the pebbling literature. In the black-magic game, a predecessor node counts as "pebbled" if it contains either a black or a magic pebble. Where Definition 10.2.1 treats P_i as a set of nodes, Definition 10.2.3 treats P_i as equal to $M_i \cup B_i$.

5. *At most $\mathfrak{M}$ magic pebbles are used throughout the strategy, i.e., $|\bigcup_{i\in[t]} M_i| \leq \mathfrak{M}$ where M_i is the ith configuration of magic pebbles*[13].

10.2.2 Cost of pebbling

In this subsection, we give definitions of several cost measures of graph pebbling, applicable to the standard and black-magic pebbling games. While these definitions assume parallel strategies, we note that the sequential versions of the definitions are entirely analogous.

Space complexity in standard pebbling

We give a brief informal summary of the definitions in this subsection, before proceeding to the formal definitions.

Pebbling complexity measures We informally overview the pebbling complexity definitions, some of which are new to this work.

The *time complexity* of a pebbling strategy status is the number of steps, i.e., $\mathsf{Time}(\text{status}) = |\text{status}|$. The *time complexity* of a graph $G = (V, E)$ given that at most S pebbles can be used is $\mathsf{Time}(G, S) = \min_{\text{status}\in\mathbb{P}_{G,T,S}}(\mathsf{Time}(\text{status}))$. Next, we overview variants of space complexity.

1. **Space complexity** of a *pebbling strategy* status on a graph G, denoted by $\mathbf{P}_{\text{s}}(\text{status})$, is the minimum number of pebbles required to execute status. Space complexity of the *graph* G with target set T, written $\mathbf{P}_{\text{s}}(G, T)$, is the minimum space complexity of any valid pebbling strategy for G.
2. **Λ-sustained space complexity** [ABP17a][14] of a *pebbling strategy* status on a graph G, denoted by $\mathbf{P}_{\text{ss}}(\text{status}, \Lambda)$, is the number of time-steps during the execution of status, in which at least Λ pebbles are used. Λ-sustained space complexity of the *graph* G with target set T, written $\mathbf{P}_{\text{ss}}(G, \Lambda, T)$ is the minimum Λ-sustained space complexity of all valid pebbling strategies for G.
3. **Graph-optimal sustained complexity** of a *pebbling strategy* status, denoted by $\mathbf{P}_{\text{opt-ss}}(\text{status})$, is the number of time-steps during the execution of status, in which the number of pebbles in use is equal to the space complexity of G. Graph-optimal sustained complexity of the *graph* G with target set T, written $\mathbf{P}_{\text{opt-ss}}(G, T)$ is the minimum graph-optimal sustained complexity of all valid pebbling strategies for G.

[13] Note that here we assume that we can distinguish between a node when a magic pebble was placed on it and removed, and another magic pebble is placed on it later. In other words, if a magic pebble was placed on a node and removed, and then, a different magic pebble is placed on the node at a later time, this node counts as two distinct nodes in the union.

[14] We note that our notation diverges from that of [ABP17a], but our Definition 10.2.9 is equivalent to their definition of "s-sustained space complexity." (E.g., they write $\Pi_{ss}(\text{status}, \Lambda)$ instead of $\mathbf{P}_{\text{ss}}(G, \text{status}, \Lambda)$.) We gave this decision some consideration as inconsistent notation can add confusing clutter to a literature; we decided on our notation (1) in order to keep consistency with the pebbling literature, where the pyramid graphs that will be used in our SHF construction are traditionally denoted by Π; and (2) because our notation makes the graph G explicit where sometimes it is implicit in [ABP17a], and this is important for the new "graph-optimal sustained complexity" notion we introduce.

4. **Δ-suboptimal sustained complexity** of a *pebbling strategy* status is the number of time-steps, during the execution of status, in which the number of pebbles in use is at least the space complexity of G *minus* Δ. Δ-suboptimal sustained complexity of the *graph* G is the minimum Δ-suboptimal sustained complexity of all valid pebbling strategies for G.

A couple of remarks are in order.

Remark 10.2.6. *The third and fourth definitions are new to this chapter. They can be seen as special variants of Λ-sustained space complexity, i.e., with a special setting of Λ dependent on the specific graph family in question. They are useful to define in their own right, as unlike plain Λ-sustained space complexity, these measures express complexity for a given graph family relative to the best possible value of Λ at which sustained space usage could be hoped for. In the rest of this chapter, we prove guarantees on* graph-optimal sustained complexity *of our constructions, which have high sustained space usage at the optimal Λ-value. However, we also define* Δ-suboptimal sustained complexity *here for completeness, since it is more general*[15] *and preferable to graph-optimal complexity when evaluating graph families where the maximal space usage may not be sustained for very long.*

Remark 10.2.7. *We have found the term "Λ-sustained space complexity" can be slightly confusing, in that it measures a number of time-steps rather than an amount of space. We retain the original terminology as it was introduced, but include this remark to clarify this point.*

We now present the formal definitions of the complexity measures for the standard pebbling game. In all of the below definitions, $G = (V, E)$ is a graph, $T \subseteq V$ is a target set, $\mathcal{P} = (P_1, \ldots, P_t)$ is a standard pebbling strategy on (G, T), and $\mathbb{P}_{G,T}$ denotes the set of all valid standard pebbling strategies on (G, T).

Definition 10.2.8. *The* space complexity of pebbling strategy status *is:* $\mathbf{P}_{\mathrm{s}}(status) = \max_{P_i \in status}(|P_i|)$. *The* space complexity of G *is the minimal space complexity of any valid pebbling strategy that pebbles the target set $T \subseteq V$:* $\mathbf{P}_{\mathrm{s}}(G, T) = \min_{status' \in \mathbb{P}_{G,T}}(\mathbf{P}_{\mathrm{s}}(status'))$.

Definition 10.2.9. *The* Λ-sustained space complexity of status *is:* $\mathbf{P}_{\mathrm{ss}}(status, \Lambda) = |\{P_i : |P_i| \geq \Lambda\}|$. *The* Λ-sustained space complexity of G *is the minimal Λ-sustained space complexity of any valid pebbling strategy that pebbles the target set $T \subseteq V$:* $\mathbf{P}_{\mathrm{ss}}(G, \Lambda, T) = \min_{status' \in \mathbb{P}_{G,T}}(\mathbf{P}_{\mathrm{ss}}(status', \Lambda))$.

Definition 10.2.10. *The* graph-optimal sustained complexity of status *is:*

$\mathbf{P}_{\text{opt-ss}}(status) = \mathbf{P}_{\mathrm{ss}}(status, \mathbf{P}_{\mathrm{s}}(G, T))$. *The* graph-optimal sustained complexity of G *is the minimal graph-optimal sustained complexity of any valid pebbling strategy that pebbles the target set $T \subseteq V$:* $\mathbf{P}_{\text{opt-ss}}(G, T) = \min_{status' \in \mathbb{P}_{G,T}}\left(\mathbf{P}_{\text{opt-ss}}(status')\right)$.

Definition 10.2.11. *The* Δ-suboptimal sustained complexity of status *is:*

$$\mathbf{P}_{\text{opt-ss}}(status, \Delta) = \mathbf{P}_{\mathrm{ss}}(status, \mathbf{P}_{\mathrm{s}}(G, T) - \Delta).$$

[15] More specifically, graph-optimal sustained complexity is Δ-suboptimal sustained complexity for $\Delta = 0$.

The Δ-suboptimal sustained complexity of G *is the minimal graph-optimal sustained complexity of any valid pebbling strategy that pebbles the target set* $T \subseteq V$*:* $\mathbf{P}_{\text{opt-ss}}(G, \Delta, T) = \min_{status' \in \mathbb{P}_{G,T}} \left(\mathbf{P}_{\text{opt-ss}}(status', \Delta) \right)$.

Time complexity in standard pebbling

We present the following formal definitions for measuring the time complexity of strategies in the standard pebble game. In all the below definitions, $G = (V, E)$ is a graph, $T \subseteq V$ is a target set, status $= (P_1, \ldots, P_t)$ is a standard pebbling strategy on (G, T) where $\mathbb{P}_{G,T,S}$ denotes the set of all valid pebbling strategies on (G, T) that use at most S pebbles.

Definition 10.2.12. *The* time complexity *of a pebbling strategy status is* $\mathsf{Time}(status) = |status|$*. The* time complexity *of a graph* $G = (V, E)$ *given that at most* S *pebbles can be used is* $\mathsf{Time}(G, S) = \min_{status \in \mathbb{P}_{G,T,S}} (\mathsf{Time}(status))$.

Space complexity in black-magic pebbling

Next, we define the corresponding complexity notions for the black-magic pebbling game. As above, $G = (V, E)$ is a graph, $T \subseteq V$ is a target set, and $\mathfrak{M}$ is a magic pebble bound. In this subsection, $\mathcal{P} = (P_1, \ldots, P_t) = ((M_1, B_1), \ldots, (M_t, B_t))$ denotes a black-magic pebbling strategy on (G, T). Moreover, $\mathbb{M}_{G,T,\mathfrak{M}}$ denotes the set of all valid magic pebbling strategies on (G, T), and $m(\text{status})$ denotes the total number of magic pebbles used in the execution of status.

Definition 10.2.13. *The* (magic) space complexity *of* $\mathcal{P}$ *is:* $\mathbf{P}_{\text{s}}(status) = \max\left(m(status), \max_{P_i \in status}(|P_i|)\right)$. *The* (magic) space complexity of G w.r.t. $\mathfrak{M}$ *is the minimal space complexity of any valid magic pebbling strategy that pebbles the target set* $T \subseteq V$*:* $\mathbf{P}_{\text{s}}(G, \mathfrak{M}, T) = \min_{status \in \mathbb{P}_{G,T,\mathfrak{M}}} (\mathbf{P}_{\text{s}}(status))$.

Remark 10.2.14. *We briefly provide some intuition for the complexity measure defined above in Def. 10.2.13. If we consider all magic pebbles to be static memory objects that were saved from a previous evaluation of the hash function, then the total number of magic pebbles is the amount of memory that was used to save the results of a previous evaluation of the hash function. Because of this, it is natural to take the maximum of the memory used to store results from a previous evaluation of the function and the current memory that is used by our current pebbling strategy since that would represent how much memory was used to compute the results of the hash function during the current evaluation.*

Definition 10.2.15. *The* (magic) Λ-sustained space complexity *of* $\mathcal{P}$ *is:* $\mathbf{P}_{\text{ss}}(status, \Lambda) = |\{P_i : |P_i| \geq \Lambda\}|$. *The* Λ-sustained space complexity of G w.r.t. $\mathfrak{M}$ and $T \subseteq V$ *is:* $\mathbf{P}_{\text{ss}}(G, \Lambda, \mathfrak{M}, T) = \min_{status \in \mathbb{P}_{G,T,\mathfrak{M}}} (\mathbf{P}_{\text{ss}}(status, \Lambda))$.

Definition 10.2.16. *The* (magic) graph-optimal sustained complexity *of* $\mathcal{P}$ *is:* $\mathbf{P}_{\text{opt-ss}}(status) = \mathbf{P}_{\text{ss}}(status, \mathbf{P}_{\text{s}}(G, T))$*. The* graph-optimal sustained complexity of G w.r.t. $\mathfrak{M}$ and $T \subseteq V$ *is:* $\mathbf{P}_{\text{opt-ss}}(G, \mathfrak{M}, T) = \min_{status \in \mathbb{P}_{G,T,\mathfrak{M}}} \left(\mathbf{P}_{\text{opt-ss}}(status) \right)$.

Definition 10.2.17. *The* (magic) Δ-suboptimal sustained complexity *of* $\mathcal{P}$ *is:* $\mathbf{P}_{\text{opt-ss}}(status, \Delta) = \mathbf{P}_{\text{ss}}(status, \mathbf{P}_{\text{s}}(G, T) - \Delta)$. *The* Δ-suboptimal sustained complexity of G w.r.t. $\mathfrak{M}$ and $T \subseteq V$ *is:*

$$\mathbf{P}_{\text{opt-ss}}(G, \Delta, \mathfrak{M}, T) = \min_{status \in \mathbb{P}_{G,T,\mathfrak{M}}} \left(\mathbf{P}_{\text{opt-ss}}(status, \Delta) \right).$$

10.2.3 Incrementally hard graphs

We introduce the following definition for our notion of graphs which require $|T|$ pebbles to pebble regardless of the number of targets that are asked, given a constraint on the number of magic pebbles that can be used. This concept has not been previously analyzed in the pebbling literature; traditional pebbling complexity usually treats graphs with fixed target sets.

Definition 10.2.18 (Incremental Hardness)**.** *Given at most* $\mathfrak{M}$ *magic pebbles, for any subset of targets* $C \subseteq T$ *where* $|C| > \mathfrak{M}$*, the number of pebbles (magic and black pebbles) necessary in the black-magic pebble game to pebble* C *is at least* $|T|$ *where the number of magic pebbles used in this game is upper bounded by* $\mathfrak{M}$*:* $\mathbf{P}_{\text{s}}(G, |C| - 1, C) \geq |T|$.

α-tradeoff cumulative complexity

α-tradeoff cumulative complexity, or CC^{α}, is a new measure introduced in this chapter, which accounts for situations where space and time do not trade off linearly. Similar notions to this have been explored before e.g. [FLW13], [BK15, AB16, RD17]. A discussion of the *core-area memory ratio* [BK15, AB16, RD17] can be found in Section 10.1.2. They considered the notion of λ-memory-hardness to be applied to graphs where intuitively $S \cdot T = \Omega\left(G^{\lambda+1}\right)$[16] or rather the space-time cost is some exponential of the size of the stored graph [FLW13]. We note that this notion is very different from our notion of α-tradeoff complexity since they only consider the space-time cost (not cumulative complexity) and do not consider nonlinear tradeoffs between space and time (one can just consider $G^{\lambda+1}$ to be a constant in the tradeoff curve).

Here, we see the usefulness of defining sustained complexities in terms of the minimum required space (as opposed to being parametrized by Λ) since we can always obtain an upper bound on CC^{α}, for *any* α, of a graph directly from our proofs of the space complexity and sustained time complexity of a DAG.

Definition 10.2.19 (Standard pebbling α-space cumulative complexity)**.** *Given a valid parallel standard pebbling strategy, status, for pebbling a graph* $G = (V, E)$*, the* standard pebbling α-space cumulative complexity *is the following:*

$$\text{p-cc}_{\alpha}(G, status) = \sum_{P_i \in status} |P_i|^{\alpha} .$$

[16] Note that S is the maximum space used in the computation and T is the time.

Definition 10.2.20 (Black-magic pebbling α-space cumulative complexity)**.** *Given a valid parallel black-magic pebbling strategy, status, for pebbling a graph $G = (V, E)$, the* black-magic pebbling α-space cumulative complexity *is the following:*

$$\text{p-cc}_{\alpha}^{M}(G, status) = \max\left(m(status)^{\alpha}, \sum_{P_i \in status} |P_i|^{\alpha}\right) = \max\left(m(status)^{\alpha}, \sum_{P_i \in status} |B_i \cup M_i|^{\alpha}\right)$$

where $m(status)$ denotes the total number of magic pebbles used in the magic pebbling strategy status.

The following definition, CC^{α}, is an analogous definition to CC as defined by [AS15] (specifically, CC^{α} when $\alpha = 1$ is equivalent to CC) to account for varying costs of memory usage vs. time.

Definition 10.2.21 (CC^{α})**.** *Given a graph, $G \in \mathbb{G}$, and the family of all valid standard pebbling strategies, $\mathbb{P}$, we define the $CC^{\alpha}(G)$ to be*

$$CC^{\alpha}(G) = \min_{status \in \mathbb{P}} \left(\text{p-cc}_{\alpha}\left(G, status\right)\right),$$

and, given the family $\mathbb{P}^M$ of all valid black-magic pebbling strategies, we define $CC^{\alpha}(G)$ to be

$$CC^{\alpha}(G) = \min_{status^M \in \mathbb{P}^M} \left(\text{p-cc}_{\alpha}^{M}\left(G, status^M\right)\right).$$

10.3 Parallel random oracle model (PROM)

In this chapter, we consider two broad categories of computations: *pebbling strategies* and *PROM algorithms*. Specifically, we discussed above the pebbling models and pebble games we use to construct our static memory-hard functions. Now, we define our PROM algorithms.

Prior work has observed the close connections between these two types of computations as applied to DAGs, and our work brings out yet more connections between the two models. In this section, we give an overview of how PROM computations work and define the complexity measures that we apply to PROM algorithms. Some of the complexity measures were introduced by prior work, and others are new in this work.

10.3.1 Overview of PROM computation

The random oracle model was introduced by [BR93]. When we say random oracle, we always mean a *parallel* random oracle unless otherwise specified.

An *algorithm* in the PROM is a probabilistic algorithm $\mathcal{B}$ which has parallel access to a stateless oracle $\mathcal{O}$: that is, $\mathcal{B}$ may submit many queries in parallel to $\mathcal{O}$. We assume $\mathcal{O}$ is sampled uniformly from an oracle set $\mathbb{O}$ and that $\mathcal{B}$ may depend on $\mathbb{O}$ but not $\mathcal{O}$.

The algorithm proceeds in discrete time-steps called *iterations*, and may be thought to consist of a series of algorithms $(\mathcal{B}_i)_{i\in}$, indexed by the iteration i, where each $\mathcal{B}_i$ passes a

state $\sigma_i \in \{0,1\}^*$ to its successor $\mathcal{B}_{i+1}$. σ_0 is defined to contain the input to the algorithm. We write $|\sigma_i|$ to denote the size, in bits, of σ_i. We write $[\![\sigma_i]\!]$ to denote $\frac{|\sigma_i|}{w}$, where w is the output length of the oracle $\mathcal{O}$. In other words, $[\![\sigma_i]\!]$ is the size of σ_i when counting in words of size w. In each iteration, the algorithm $\mathcal{B}_i$ may make a *batch* $\mathbf{q}_i = (q_{i,1}, \ldots, q_{i,|\mathbf{q}_i|})$ of queries, consisting of $|\mathbf{q}_i|$ individual queries to $\mathcal{O}$, and instantly receive back from the oracle the evaluations of $\mathcal{O}$ on the individual queries, i.e., $(\mathcal{O}(q_{i,1}), \ldots, \mathcal{O}(q_{i,|\mathbf{q}_i|}))$.

At the end of any iteration, $\mathcal{B}$ can append values to a special output register, and it can end the computation by appending a special terminate symbol $\perp$ on that register. When this happens, the contents y of the output register, excluding the trailing $\perp$, is considered to be the output of the computation. To denote the process of sampling an output, y, provided input x, we write $y \leftarrow \mathcal{B}^{\mathcal{O}}(x)$.

Definition 10.3.1 (Oracle functions)**.** *An* oracle function *is a collection* $\mathfrak{f} = \{f^{\mathcal{O}} : D \to R\}_{\mathcal{O} \in \mathbb{O}}$ *of functions with domain* D *and outputs in* R *indexed by oracles* $\mathcal{O} \in \mathbb{O}$.

A family of oracle functions *is a set* $\mathcal{F} = \{\mathfrak{f}_\kappa : D_\kappa \to R_\kappa\}_{\kappa \in}$ *where each* $\mathfrak{f}_\kappa$ *is indexed by oracles from an oracle set* $\mathbb{O}_\kappa : \{0,1\}^\kappa \to \{0,1\}^\kappa$ *indexed by a security parameter* κ.[17]

Definition 10.3.2 (Memory complexity of PROM algorithms)**.** *The* memory complexity of $\mathcal{B}(x;\rho)$ *(i.e., the memory complexity of* $\mathcal{B}$ *on input* x *and randomness* ρ*) is defined as:*

$$\mathsf{mem}_{\mathbb{O}}(\mathcal{B}, x, \rho) = \max_{i \in} \{[\![\sigma_i]\!]\} \ . \tag{10.1}$$

Definition 10.3.3 (**Λ**-sustained memory complexity of PROM algorithms)**.** *The* Λ-sustained memory complexity of $\mathcal{B}(x;\rho)$ *is defined as:*

$$\text{s-}\mathsf{mem}_{\mathbb{O}}(\Lambda, \mathcal{B}, x, \rho) = |\{i \in : |\sigma_i| \geq \Lambda\}| \ . \tag{10.2}$$

Note that (10.1) and (10.2) are distributions over the choice of $\mathcal{O} \leftarrow \mathbb{O}$.

10.3.2 Functions defined by DAGs

We now describe how to translate a graph construction into a function family, whose evaluation involves a series of oracle calls in the PROM. Any family of DAGs induces a family of *oracle functions* in the PROM, whose complexity is related to the pebbling complexity of the DAG. We first define the syntax of *labeling* of DAG nodes, then define a *graph function family*.

Definition 10.3.4 (Labeling)**.** *Let* $G = (V, E)$ *be a DAG with maximum in-degree* δ*, let* $\mathfrak{L}$ *be an arbitrary "label set," and define* $\mathbb{O}(\delta, \mathfrak{L}) = \left(\left[V \times \bigcup_{\delta'=1}^{\delta} \mathfrak{L}^{\delta'}\right] \to \mathfrak{L}\right)$*. For any function* $\mathcal{O} \in \mathbb{O}(\delta, \mathfrak{L})$ *and any label* $\zeta \in \mathfrak{L}$*, the* $(\mathcal{O}, \zeta)$*-labeling of* G *is a mapping* $\mathsf{label}_{\mathcal{O},\zeta} : V \to \mathfrak{L}$

[17] For simplicity, we have the input and output domains of the oracles equal to $\{0,1\}^\kappa$, but this is not a necessary restriction: the sizes could be any polynomials in κ.

defined recursively as follows.[18]

$$\mathsf{label}_{\mathcal{O},\zeta}(v) = \begin{cases} \mathcal{O}(v,\zeta) & \text{if } \mathsf{indeg}(v) = 0 \\ \mathcal{O}(v, \mathsf{label}_{\mathcal{O},\zeta}(v_1), \ldots, \mathsf{label}_{\mathcal{O},\zeta}(v_{\mathsf{indeg}(v)})) & \text{if } \mathsf{indeg}(v) > 0 \end{cases}.$$

Definition 10.3.5 (Graph function family)**.** *Let $n = n(\kappa)$ and let $\mathbb{G}_\delta = \{G_{n,\delta} = (V_n, E_n)\}_{\kappa\in}$ be a graph family. We write $\mathbb{O}_{\delta,\kappa}$ to denote the set $\mathbb{O}(\delta, \{0,1\}^\kappa)$ as defined in Definition 10.3.4. The* graph function family *of $\mathbb{G}$ is the family of oracle functions $\mathcal{F}_\mathbb{G} = \{\mathfrak{f}_G\}_{\kappa\in}$ where $\mathfrak{f}_G = \{f_G^\mathcal{O} : \{0,1\}^\kappa \to (\{0,1\}^\kappa)^z\}_{\mathcal{O}\in\mathbb{O}_{\delta,\kappa}}$ and $z = z(\kappa)$ is the number of sink nodes in G. The output of $f_G^\mathcal{O}$ on input label $\zeta \in \{0,1\}^\kappa$ is defined to be*

$$f_G^\mathcal{O}(\zeta) = \mathsf{label}_{\mathcal{O},\zeta}(s_1), \ldots, \mathsf{label}_{\mathcal{O},\zeta}(s_i)\ \forall s_1, \ldots, s_i \in \mathsf{sink}(G),$$

where $\mathsf{sink}(G)$ is the set of sink nodes of G.

10.3.3 Relating complexity of PROM algorithms and pebbling strategies

Any PROM algorithm $\mathcal{B}$ and input x induce a black-magic pebbling strategy, $\mathsf{epf\text{-}magic}_\zeta(\mathcal{B}, \mathcal{O}, x, \$)$, called an *ex-post-facto black-magic pebbling strategy*. The way in which this strategy is induced is similar to *ex-post-facto pebbling* as originally defined by [AS15] in the context of the standard pebble game. Please refer to [AS15] for a detailed description of this proof technique. We adapt their technique for the black-magic game.

Definition 10.3.6 (Ex-post-facto black-magic pebbling)**.** *Let $n = n(\kappa)$ and let $\mathbb{G}_\delta = \{G_{n,\delta} = (V_n, E_n)\}_{\kappa\in}$ be a graph family. Let $\zeta = \zeta(\kappa) \in \{0,1\}^\kappa$ be an arbitrary input label for the graph function family $\mathcal{F}_\mathbb{G}$. For any $v \in V_n$, define*

$$\mathsf{pre\text{-}lab}_{\mathcal{O},\zeta}(v) = (v, \mathsf{label}_{\mathcal{O},\zeta}(\mathsf{Pred}(v)))\,.$$

Let $\mathcal{B}$ be a non-uniform PROM algorithm. Fix an implicit security parameter κ. Let x be an input to $\mathcal{B}$. We now define a magic pebbling strategy induced by any given execution of $\mathcal{B}^\mathcal{O}(x;\$)$, where $\$$ denotes the random coins of $\mathcal{B}$. Such an execution makes a sequence of batches of random oracle calls (as defined in Section 10.3.1), which we denote by

$$\mathbf{q}(\mathcal{B}, \mathcal{O}, x, \$) = (\mathbf{q}_1, \ldots, \mathbf{q}_t)\,.$$

The induced black-magic pebbling strategy,

$$\mathsf{epf\text{-}magic}_\zeta(\mathcal{B}, \mathcal{O}, x, \$) = ((B_0, M_0), \ldots, (B_t, M_t))\,, \tag{10.3}$$

is called an ex-post-facto black-magic pebbling*, and is defined by the following procedure.*

1. $B_0 = M_0 = \varnothing$.

[18] We abuse notation slightly and also invoke $\mathsf{label}_{\mathcal{O},\zeta}$ on *sets* of vertices, in which case the output is defined to be a tuple containing the labels of all the input vertices, arranged in lexicographic order of vertices.

2. *For* $i = 1, \ldots, t$*:*
   (a) $B_i = B_{i-1}$.
   (b) $M_i = M_{i-1}$.
   (c) *For each individual query* $q \in \mathbf{q}_i$*, if there is some* $v \in V_n$ *such that* $q = \mathsf{pre\text{-}lab}_{\mathcal{O},\zeta}(v)$ *and* $v \notin P_i$*, then "pebble* v*" by performing the following steps:*
      i. If $\mathsf{Pred}(v) \subseteq M_i \cup B_i$*:*[19]
         - $B_i = B_i \cup \{v\}$.

      ii. Else:
         - $V = \{v\}$.
         - *Let* V^* *be the transitive closure of* V *under the following operation:* $V = V \cup (\bigcup_{v' \in V} \mathsf{Pred}(v') \setminus (M_i \cup B_i))$.
         - $M_i = M_i \cup V^*$.
3. *For* $i = 1, \ldots, t$*:*
   (a) *A node* $v \in M_i \cup B_i$ *is said to be* necessary at time i *if*

$$\exists j \in [t], q \in \mathbf{q}_j, v' \in V_n \text{ s.t. } \quad j > i \wedge v \in \mathsf{Pred}(v') \wedge q = \mathsf{pre\text{-}lab}_{\mathcal{O},\zeta}(v') \\ \wedge \left(\nexists k \in [t], q' \in \mathbf{q}_k \text{ s.t. } i < k < j \wedge q' = \mathsf{pre\text{-}lab}_{\mathcal{O},\zeta}(v) \right).$$

In other words, a node is necessary if its label will be required in a future oracle call, but its label will not be obtained by any oracle query between now and that future oracle call.
Remove from B_i *and* M_i *all nodes that are not necessary at time* i.

10.3.4 Legality and space usage of ex-post-facto black-magic pebbling

The following theorems establish that the space usage of PROM algorithms is closely related to the space usage of the induced pebbling.

We will use the following supporting lemma, also used in prior work such as [AS15, DKW11] (see, e.g., [DKW10] for a proof).

Lemma 10.3.7. *Let* $B = b_1, \ldots, b_u$ *be a sequence of random bits and let* $\mathbb{H}$ *be a set. Let* $\mathcal{P}$ *be a randomized procedure that gets a hint* $h \in \mathbb{H}$*, and can adaptively query any of the bits of* B *by submitting an index* i *and receiving* b_i *as a response. At the end of its execution,* $\mathcal{P}$ *outputs a subset* $S \subseteq \{1, \ldots, u\}$ *of* $|S| = \varphi$ *indices which were not previously queried, along with guesses for the values of the bits* $\{b_i : i \in S\}$*. Then the probability (over the choice of* B *and the randomness of* $\mathcal{P}$*) that there exists some* $h \in \mathbb{H}$ *such that* $\mathcal{P}(h)$ *outputs* all *correct guesses is at most* $|\mathbb{H}|/2^{\varphi}$.

Lemma 10.3.8 (Legality and magic pebble usage of ex-post-facto black-magic pebbling)**.** *Let* $n = n(\kappa)$ *and let* $\mathbb{G}_\delta = \{G_{n,\delta} = (V_n, E_n)\}_{\kappa \in}$ *be a graph family. Let* $\zeta \in \{0,1\}^\kappa$ *be an arbitrary input label for* $\mathbb{G}_\delta$*. Fix any efficient PROM algorithm* $\mathcal{B}$ *and input* x*. With overwhelming probability over the choice of random oracle* $\mathcal{O} \leftarrow \mathbb{O}$ *and the random coins* \$ *of*

[19] Recall that $\mathsf{Pred}(v)$ returns the immediate predecessors of v (i.e. vertices $u \in V_n$ where $(u, v) \in E_n$).

$\mathcal{B}$, it holds that the ex-post-facto magic pebbling $\mathsf{epf\text{-}magic}_\zeta(\mathcal{B},\mathcal{O},x,\$)$ *consists of valid magic-pebbling moves, and uses fewer than* $\chi = \left\lfloor \frac{|x|}{\kappa - \log(q)} + 1 \right\rfloor$ *magic pebbles (i.e., for all i, $|M_i| \leq \chi$), where q is the number of oracle queries made by $\mathcal{B}(x)$.*

Proof. Fix an algorithm $\mathcal{B}$ and, for the sake of contradiction, suppose that there is an input x such that with non-negligible probability over $\mathcal{O}$ and \$, the induced pebbling $\mathsf{epf\text{-}magic}_\zeta(\mathcal{B},\mathcal{O},x,\$)$ uses at least χ magic pebbles or contains an invalid move. By definition, this means that the following event $\mathcal{E}$ occurs with non-negligible probability: on at least χ occasions, a (magic) pebble is placed on a node v although its parents were not all pebbled in the previous step. In turn, this means that a correct random-oracle query for the label of v is made by $\mathcal{B}$; and the correct query contains the label of some predecessor node v' which was not contained in the output of any previous oracle call.

Let us suppose that event $\mathcal{E}$ occurs with probability more than $p = \frac{q^\chi 2^{|x|}}{2^{\kappa\chi}}$. Note that this probability is negligible, since

$$\begin{aligned}
p = \frac{q^\chi 2^{|x|}}{2^{\kappa\chi}} &= 2^{\chi\log(q)+|x|-\kappa\chi} \\
\chi\log(q)+|x|-\kappa\chi &= \chi(\log(q)-\kappa)+|x| && \text{(analyzing the exponent)} \\
&= \left\lfloor \frac{|x|}{\kappa-\log(q)} + 1 \right\rfloor (\log(q)-\kappa)+|x| && \text{(substituting for } \chi) \\
&\leq \frac{|x|+\kappa-\log(q)}{\kappa-\log(q)}(\log(q)-\kappa)+|x| \\
&= -(|x|+\kappa-\log(q))+|x| && \text{(canceling denominator)} \\
&= -\kappa+\log(q)
\end{aligned}$$

and q is polynomial in κ. Based on this assumption, we construct a predictor that predicts χ output values of the random oracle with impossibly high probability (specifically, violating Lemma 10.3.7) as follows. The predictor $\mathcal{P}$ depends on input x and can query the random oracle on inputs of its choice, before outputting its prediction. Let $\hat{r}$ be an upper bound on the number of random bits used by $\mathcal{B}(x)$. The predictor also has access to a sequence $\hat{R}$ of $\hat{r}$ random bits, that it can use to simulate the random coins of $\mathcal{B}$.

- *Hint:* The predictor $\mathcal{P}$ receives as its hint[20] *either* $\perp$ if the induced pebbling $\mathsf{epf\text{-}magic}_\zeta(\mathcal{B},\mathcal{O},x,\$)$ is valid and uses no more than χ magic pebbles, *or* the following information otherwise:
  - the index $i^* \in [q]$ of the first oracle call causing the illegal event (inducing the χth placement of a magic pebble on some node v) to happen;
  - the indices $I \subset [i^*]$ of all oracle calls preceding the i^*th oracle call, that induce the placement of a magic pebble or pebbles; and
  - $\mathcal{B}$'s input x.

  The size of this hint is at most $\chi\log(q)+|x|$ bits.
- *Execution:* If the hint is $\perp$, then $\mathcal{P}$ halts and outputs nothing. Otherwise, $\mathcal{P}$ runs $\mathcal{B}(x;\hat{R})$, forwarding all oracle calls to the random oracle, until the i^*th query. By

[20] Note that the hint may depend both on the choice of random oracle, and on the randomness $\hat{R}$.

construction, for each $i' \in I \cup \{i^*\}$, the i'th query contains the labels of the parents of the node $v_{i'}$ whose pebbling is induced by the i'th query, and at least one of these labels (say, label $\ell_{w_{i'}}$ for parent node $w_{i'}$) was not the output of any previous query to the random oracle. For each $i' \in I \cup \{i\}$, our predictor recomputes the value $\tilde{w}_{i'} = \mathsf{pre\text{-}lab}_{\mathcal{O},\zeta}(w_{i'})$ which is the preimage under $\mathcal{O}$ of $\ell_{w_{i'}}$. Note that by definition of pre-lab, $\tilde{w}_{i'}$ can be computed without ever querying $\mathcal{O}$ on input $\tilde{w}_{i'}$. Finally, $\mathcal{P}$ outputs the following pairs:

$$\left\{(\tilde{w}_{i'}, \ell_{w_{i'}})\right\}_{i' \in I \cup \{i^*\}} .$$

Since by construction, each query $i' \in I \cup \{i^*\}$ induced the placement of a magic pebble, it follows that each pair $(\tilde{w}_{i'}, \ell_{w_{i'}})$ is a valid input-output pair of $\mathcal{O}$. Moreover, $\mathcal{P}$ never queried $\mathcal{O}$ on any $\tilde{w}_{i'}$.

The predictor's hint is $\perp$ with probability at most that of $\mathcal{E}$, and the predictor succeeds whenever the hint is not $\perp$. Hence, by our assumption about the probability p of the event $\mathcal{E}$, the predictor must succeed with probability greater than $p = \frac{q^\chi 2^{|x|}}{2^{\kappa\chi}}$. By construction, the size of the predictor's hint set is at most $q^\chi 2^{|x|}$, and the predictor's output is $\kappa\chi$ bits long. Thus Lemma 10.3.7 implies that the probability (over the choice of $\mathcal{O}$ and the randomness of $\mathcal{P}$) that there is some hint such that $\mathcal{P}$ outputs all correct guesses is at most $\frac{q^\chi 2^{|x|}}{2^{\kappa\chi}}$. (This is equal to p.) We have a contradiction, and the lemma follows. □

Lemma 10.3.9 (Space usage of ex-post-facto black-magic pebbling)**.** *Let $n, \mathbb{G}_\delta, \zeta$ be as in Lemma 10.3.8. Fix any PROM algorithm $\mathcal{B}$ and input x. Fix any $i \in [t]$, $\lambda \geq 0$, and define*

$$\mathsf{epf\text{-}magic}_\zeta(\mathcal{B}, \mathcal{O}, x, \$) = (P_1^{\mathcal{O}}, \ldots, P_t^{\mathcal{O}}) = ((B_1^{\mathcal{O}}, M_1^{\mathcal{O}}), \ldots, (B_t^{\mathcal{O}}, M_t^{\mathcal{O}}))$$

for oracle $\mathcal{O}$. We may omit the superscript $\mathcal{O}$ for notational simplicity. It holds for all large enough κ that the following probability is overwhelming:

$$\Pr[\forall i \in [t],\ |P_i| \leq \chi'] ,$$

where $\chi' = \left\lfloor \frac{|\sigma_i|}{\kappa - \log(q)} + 1 \right\rfloor$ (where σ_i is the state that $\mathcal{B}_i$ passes to $\mathcal{B}_{i+1}$), q is the number of oracle queries made by $\mathcal{B}$, and the probability is taken over $\mathcal{O} \leftarrow \mathbb{O}$ and the coins of $\mathcal{B}$.

Proof. This proof has a very similar structure to that of Lemma 10.3.8. Assume for contradiction that with non-negligible probability for some $i \in [t]$ it holds that $|P_i| > \chi'$. Let $\mathcal{E}$ denote the event that the induced pebbling $\mathsf{epf\text{-}magic}_\zeta(\mathcal{B}, \mathcal{O}, x, \$)$ satisfies $|P_i| > \chi'$, and suppose that $\mathcal{E}$ occurs with probability more than $p = \frac{q^{\chi'} 2^{|\sigma_i|}}{2^{\kappa\chi'}}$. Note that p is negligible,

since

$$
\begin{aligned}
p = \frac{q^{\chi'} 2^{|\sigma_i|}}{2^{\kappa\chi'}} &= 2^{\chi' \log(q) + |\sigma_i| - \kappa\chi'} \\
\chi' \log(q) + |\sigma_i| - \kappa\chi' &= \chi'(\log(q) - \kappa) + |\sigma_i| && \text{(analyzing the exponent)} \\
&= \left\lfloor \frac{|\sigma_i|}{\kappa - \log(q)} + 1 \right\rfloor (\log(q) - \kappa) + |\sigma_i| && \text{(substituting for } \chi') \\
&\leq \frac{|\sigma_i| + \kappa - \log(q)}{\kappa - \log(q)} (\log(q) - \kappa) + |\sigma_i| \\
&= -(|\sigma_i| + \kappa - \log(q)) + |\sigma_i| && \text{(canceling denominator)} \\
&= -\kappa + \log(q)
\end{aligned}
$$

We design a predictor $\mathcal{P}$ to predict the labels of all nodes in P_i with impossibly high probability, as follows. We refer to the oracle call that causes the ex-post-facto pebbling of a node $v \in P_i$ a *critical call*. (Critical calls encompass both black and magic pebble placements.) $\mathcal{P}$ depends on σ_i, $\mathcal{O}$, and a long enough sequence $\hat{R}$ of random bits used to simulate the coins of $\mathcal{B}$.

- *Hint:* The predictor $\mathcal{P}$ receives as its hint *either* $\perp$ if the induced pebbling $\mathsf{epf\text{-}magic}_{\zeta}(\mathcal{B}, \mathcal{O}, x, \$)$ satisfies $|P_i| \leq \chi'$, *or* the following information otherwise:
  - the indices $J = \{j_1, \ldots, j_c\} \in [q]^{|P_i|}$ of the critical calls made by $\mathcal{B}$, and
  - the state σ_i outputted by $\mathcal{B}$ at the end of iteration i, and

  The size of this hint is $|P_i| \log(q) + |\sigma_i|$ bits. By our assumption on $|P_i|$,[21] this is more than $\chi' \log(q) + |\sigma_i|$ bits.
- *Execution:* $\mathcal{P}$ runs $\mathcal{B}$ on input (z, σ_i), recording the labels of all input-nodes of the critical calls. To answer any oracle call Q with output-node v, the predictor does the following:
  - Determines if the call is correct. A call is correct iff it is a critical call or for each parent $w_{i'}$ of v, a correct call for $w_{i'}$ has already been made and Q matches the results of those calls. In particular, $Q = \mathsf{pre\text{-}lab}_{\mathcal{O},\zeta'}(w_{i'})$ and no new oracle calls need be made by the predictor to check this.
  - If the call is correct and the label of v has already been recorded then output the label. Otherwise query $\mathcal{O}$ to answer the call.

  Finally, $\mathcal{P}$ outputs predictions of all of the labels of the magic pebbles and all the labels associated with P_i, as follows.
  - The labels of the magic pebbles are determined as described in the proof of Lemma 10.3.8.
  - When $\mathcal{B}$ terminates, $\mathcal{P}$ checks the transcript to determine the set B_i.[22] It is easy to verify that their labels were never queried to $\mathcal{O}$ by $\mathcal{P}$. Then, for all $v \in B_i$ the predictor computes $\tilde{v} = \mathsf{pre\text{-}lab}_{\mathcal{O},\zeta'}(v)$ and outputs the pair $(\tilde{v}, \ell_v)$ where ℓ_v is the label of v (as specified in the input of the oracle call for associated critical call).

[21] Recall for the sake of contradiction, we assumed $|P_i| > \chi'$.

[22] Recall, this is the set of nodes containing black pebbles.

The predictor's hint is $\perp$ with probability at most that of $\mathcal{E}$, and the predictor succeeds whenever the hint is not $\perp$. Hence, by our assumption about the probability p of the event $\mathcal{E}$, the predictor must succeed with probability greater than p. The predictor's output is $\kappa|P_i| > \kappa\chi'$ bits long. From Lemma 10.3.7, it follows that the probability (over the choice of $\mathcal{O}$ and the randomness of $\mathcal{P}$) that there is some hint such that $\mathcal{P}$ outputs all correct guesses is at most $(q^{\chi'}2^{|\sigma_i|})/2^{\kappa\chi'}$. (This is equal to p.) We have a contradiction and the lemma follows. □

10.4 Static-memory-hard functions

We now define *static-memory-hard functions*. As mentioned above, prior notions of memory-hardness consider only dynamic memory usage. To model static memory usage, we consider a hash function with two parts $(\mathcal{H}_1, \mathcal{H}_2)$ where $\mathcal{H}_2(x)$ computes the output of the hash function $h(x)$ given oracle access to the output of $\mathcal{H}_1$. This design can be seen to reduce honest party computation time by limiting the hard work to one-off preprocessing phase, while maintaining a large space requirement for password-cracking adversaries. Informally, our guarantee says that unless the adversary stores a specified amount of *static* memory, he must use an equivalent amount of *dynamic* memory to compute h correctly on many outputs. Definition 10.4.1 is syntactic and Definition 10.4.2 formalizes the memory-hardness guarantee.

Notation PPT stands for "probabilistic polynomial time." For $\vec{b} \in \{0,1\}^*$, define $\mathsf{Seek}_{\vec{b}} : \{1,\ldots,|\vec{b}|\} \to \{0,1\}$ to be an oracle that on input ι returns the ιth bit of $\vec{b}$.

Definition 10.4.1 (Static-memory hash function family (SHF))**.** *A* static-memory hash function family $\mathcal{H}^{\mathcal{O}} = \{h_\kappa^{\mathcal{O}} : \{0,1\}^{w'} \to \{0,1\}^w\}_{\kappa\in}$ *mapping* $w' = w'(\kappa)$ *bits to* $w = w(\kappa)$ *bits is described by a pair of deterministic oracle algorithms* $(\mathcal{H}_1, \mathcal{H}_2)$ *such that for all* $\kappa \in$ *and* $x \in \{0,1\}^n$,

$$\mathcal{H}_2^{\mathsf{Seek}_R}(1^\kappa, x) = h_\kappa(x), \text{ where } R = \mathcal{H}_1(1^\kappa)\,.$$

(The superscript $\mathcal{O}$ *is left implicit.)*

The next definition presents a parametrized notion of $(\Lambda, \Delta, \tau, q)$-hardness of an SHF. Before delving into the formal definition, we give a brief intuition of the guarantee provided by Definition 10.4.2: any adversary who produces at least q correct input-output pairs of the hash function must *either* have used $\Lambda - \Delta$ static memory *or* incur a requirement of Λ dynamic memory *sustained over* τ time-steps at runtime.

The role of q. The parameter q in Definition 10.4.2 serves to capture the intuitive idea that an adversary that uses a certain amount of space could always use that space to directly store output values of h_κ. Clearly, an adversary with an arbitrary input R could very easily output up to $[\![|R|]\!]$ correct output values. Our goal is to lower bound the amount of space needed by an adversary who outputs nontrivially more correct values than that — and q, which is a function of $|R|$, captures how many more.

Definition 10.4.2 (**$(\Lambda, \Delta, \tau, q)$**-hardness of SHF)**.** *Let $\mathcal{H} = \{h_\kappa\}_{\kappa\in}$ be a static-memory hash function family described by algorithms $(\mathcal{H}_1, \mathcal{H}_2)$, mapping w' to w bits. $\mathcal{H}$ is $(\Lambda, \Delta, \tau, q)$-*hard *if for any large enough $\kappa \in$, any string $R \in \{0,1\}^{\Lambda-\Delta}$, and any PPT algorithm $\mathcal{A}$, for any set $X = \{x_1, \ldots, x_q\} \subseteq \{0,1\}^{w'}$, and for randomness ρ, there is a negligible ε such that*

$$\Pr_{\mathcal{O},\rho}\left[\left\{(x_1, h_\kappa(x_1)), \ldots, (x_q, h_\kappa(x_q))\right\} = \mathcal{A}(1^\kappa, R; \rho) \wedge \mathsf{s\text{-}mem}_{\mathbb{O}}(\Lambda, \mathcal{A}, R, \rho) < \tau\right] < \varepsilon\ .$$

For simplicity, we henceforth assume $w' = w = \kappa$ (i.e., the oracle's input and output sizes are equal to the security parameter) unless otherwise stated.

10.4.1 Dynamic SHFs

As discussed in detail in the introduction, static memory requirements are orthogonal and complementary to dynamic memory requirements of MHFs as formalized by [AS15]. Given a pebbling-based SHF and a pebbling-based MHF, they can be combined by simple concatenation into a "dynamic SHF," a function that inherits both the static memory requirement of the former and the dynamic memory requirement of the latter, as outlined (informally) next.

Let $\mathcal{H}^{\mathcal{O}}_{\mathsf{dyn}}$ be a dynamic MHF and $(\mathcal{H}^{\mathcal{O}}_1, \mathcal{H}^{\mathcal{O}}_2)$ be a SHF family, and the computation of both of these correspond to computing labels of nodes in a DAG as a function of a pebbling algorithm and a random oracle $\mathcal{O}$. We construct a dynamic SHF $\mathcal{H}^{\mathcal{O}}$ that is defined as follows: on input $(1^\kappa, x)$, output $\mathcal{H}^{\mathcal{O}(0,\cdot)}_2(1^\kappa, x) \| \mathcal{H}^{\mathcal{O}(1,\cdot)}_{\mathsf{dyn}}(1^\kappa, x)$. The resulting $\mathcal{H}^{\mathcal{O}}$ inherits both the MHF guarantees of $\mathcal{H}_{\mathsf{dyn}}$ and the SHF guarantees of $(\mathcal{H}_1, \mathcal{H}_2)$. Note that importantly, the labels of the nodes in the graphs corresponding to the MHF $\mathcal{H}^{\mathcal{O}(0,\cdot)}_{\mathsf{dyn}}$ and the SHF $(\mathcal{H}^{\mathcal{O}(1,\cdot)}_1, \mathcal{H}^{\mathcal{O}(1,\cdot)}_2)$ are independent as the MHF and the SHF use disjoint partitions of the random oracle domain.

Using this method, our SHF constructions can be combined with existing MHF constructions such as [AS15], [ABP17a], [ABP17b], yielding a "best of both worlds" dynamic SHF that enjoys both types of memory-hardness.

10.5 SHF constructions

A first attempt What if we pebble a hard-to-pebble graph, and then let $R_{k,i} = H(P(k), i)$ where $P(k)$ is the entire pebbling of the graph (on input k and iteration i is the i-th call to the hash function H)? This would in fact work in the random oracle model where the random oracle takes arbitrary-length input. However, in practice, hash functions do not take arbitrary-length input. While constructions like Merkle-Damgård [Mer79] and sponge [BDPA08] can transform a fixed-input-length hash function into one that takes arbitrary-length inputs, the resulting function does *not* behave like a random oracle even if the fixed-length hash function does.[23] Moreover, the computation graphs of known

[23] For example, both the constructions mentioned process the input sequentially in chunks. Evaluating the hash function on inputs that differ only in the final chunk will yield outputs that differ in a known

length-expanding transformations such as Merkle-Damgård and sponge functions require very little space to compute. For instance, the computation graph of the Merkle-Damgård construction is a binary tree and the computation graph of the sponge function is a caterpillar graph both of which take logarithmic and constant space, respectively, to compute. Thus, we have to use special constructions to achieve the local-hardness properties we need.

Recall from Definition 10.2.18 that the property we want is this "locally hard to access" notion, meaning that if an adversarial party chooses to not store the static part of our hash function which they obtain from performing the "preprocessing" computation associated with $\mathcal{H}_1$, then they must use the same memory and sustained time to recompute the function when our static-memory-hard function is called on *any subset of inputs* larger than the memory used to store the preprocessed computation. We achieve this desired property in our $\mathcal{H}_1$ functions using two novel DAG constructions, one of which is optimal for a specific graph class and the other we conjecture to be optimal for all general graph classes.

10.5.1 $\mathcal{H}_1$ constructions

We first note the differences between the graph constructions we present here and the constructions presented in previous literature [AS15, ACK+16, ABP17a, DFKP15]. Firstly, many of the constructions presented in previous work feature a single target node. This is reasonable in the context of memory-hard functions since both the honest party and the adversary must compute the hash function dynamically (obtaining a single label as the output of the function) on each input. However, in our context of static-memory-hard functions, single-target-node constructions do not make sense. Secondly, our constructions differ from even the multiple target node constructions presented in the literature (specifically, the constructions of [DFKP15]) since prior constructions mainly focused on finding graphs that have large memory vs. time tradeoffs.

Our constructions are designed with the goal that any adversary that does not store almost all the target labels must dynamically use *the same amount of space as needed to store all the labels* to compute the hash function (while *still incurring a cost in runtime*). Moreover, our constructions based on local hardness ensure a stronger guarantee than the constructions in [DFKP15]; in our case, one must use at least S space (for some definition of S) to compute *any* given subset of targets larger than one's current memory usage, whereas in their case, they use S space to compute some subset of targets chosen uniformly at random. Therefore, our specifications are stronger in that we provide a space bound as well as a time bound for adversaries; and moreover, for *honest* parties, the time cost is only a one-time setup cost. We prove our pebbling costs in terms of the black-magic pebble game (defined in Section 10.2) as opposed to the standard pebble game used in previous works. Most notably, this means that in all of our constructions, the pebbling number is upper bounded by the number of targets (since one can always just pebble the targets with magic pebbles).

way; this provides a way to distinguish these constructions from a random oracle even if the underlying fixed-length hash function is a random oracle.

We begin with some simple and clean constructions of $\mathcal{H}_1$ based on pebbling constructions that exist in the literature. We first prove a lemma regarding the minimum number of pebbles used in the PROM model and the minimum number of pebbles used in the sequential memory model. This is useful in more than one way: (1) it tells us that parallelization does not save the adversary in space so honest parties (who can only compute a constant number of labels at a time) and adversaries (who can compute an arbitrary number of labels at the same time) operate under the same space constraints and (2) it allows us to directly compare sustained time complexities between adversaries and honest parties with respect to space usage .

Lemma 10.5.1 (Standard Pebbling Sequential/Parallel Equivalence)**.** *Given a DAG $G = (V, E)$, $\mathbf{P}_s(G, T) = \mathbf{P}_s^{\|}(G, T)$ where $\mathbf{P}_s(G, T)$ is defined to be the minimum standard pebbling space complexity in the sequential model, and we define $\mathbf{P}_s^{\|}(G, T)$ to be the minimum standard pebbling space complexity in the parallel model.*

Proof. Any sequential pebbling strategy, status can be simulated by a parallel pebbling strategy, $\mathcal{P}^{\|}$ since $\mathcal{P}^{\|}$ can choose to place one pebble at a time. Therefore, $\mathbf{P}_s^{\|}(G, T) \leq \mathbf{P}_s(G, T)$. We now show that there exists a sequential pebbling strategy, status, that uses the same number of pebbles to pebble a graph as a parallel strategy $\mathcal{P}^{\|}$. Suppose that at time i, a set of pebbles are added to nodes in P_i in G under algorithm $\mathcal{P}^{\|}$. Then, $pred(P_i)$ must be pebbled at time $i - 1$. status can thus spend $|P_i \backslash P_{i-1}|$ pebbling steps to pebble the graph sequentially by adding pebbles on all vertices $v \in P_i \backslash P_{i-1}$ sequentially until the state of the graph is the same as the state of the graph at time i under strategy $\mathcal{P}^{\|}$. Similarly, if a set of pebbles D_i are deleted from the graph at time i, then status can choose to spend at most $|D_i|$ sequential pebbling steps to delete $|D_i|$ pebbles. If both strategies start on identical graphs with the same starting configuration P_0, then we have shown that $\mathbf{P}_s^{\|}(G, T) \geq \mathbf{P}_s(G, T)$. Thus, $\mathbf{P}_s^{\|}(G, T) = \mathbf{P}_s(G, T)$. □

We use Lemma 10.5.1 to prove an equivalent lemma for the black-magic pebble game below.

Lemma 10.5.2 (Black-Magic Pebbling Sequential/Parallel Equivalence)**.** *Given a DAG $G = (V, E)$, $\mathbf{P}_s(G, |T|, T) = \mathbf{P}_s{}^{\|}(G, |T|, T)$ where $\mathbf{P}_s(G, |T|, T)$ was defined to be the minimum black-magic pebbling space complexity in the sequential model, and we define $\mathbf{P}_s{}^{\|}(G, |T|, T)$ to be the minimum black-magic pebbling space complexity in the parallel model.*

Proof. Any placement of black pebbles can be translated from the sequential to the parallel pebbling strategy and vice versa using the techniques stated in the proof of Lemma 10.5.1. Any sequential pebbling placement of magic pebbles can be simulated trivially by a parallel pebbling strategy. Any parallel pebbling placement of M magic pebbles can be simulated via a sequential pebbling strategy using M additional steps. Thus, $\mathbf{P}_s(G, |T|, T) = \mathbf{P}_s{}^{\|}(G, |T|, T)$. □

Now, we jump into our constructions. We first provide a simple construction and show why this construction is not optimal. In addition, we define some subgraph components in the pebbling literature that are important subcomponents of our constructions.

A failed attempt at $\mathcal{H}_1$

We first provide a failed attempt at constructing $\mathcal{H}_1$ due to the large amount of time that is needed to compute the function (for the sequential honest party) with respect to the amount of memory needed to store the output of the function. In other words, this construction is problematic in the sense that an exponential number of steps is necessary to compute the stored results of the function from scratch for the honest party but the adversary with parallel processing time can compute the function from scratch in linear time. Although the honest party could obtain the results of the preprocessing (i.e. the static part of the hash function) from elsewhere, we must ensure that they can still feasibly compute $\mathcal{H}_1$ themselves in the event that they do not trust any of the sources from which they can obtain the static data.

Intuitively, our failed attempt at constructing $\mathcal{H}_1$ is a series of binary search trees. From here onwards, we describe all constructions of $\mathcal{H}_1$ as a directed acyclic graph with n nodes and later use our theorems above to prove static memory hardness from our constructed DAGs.

Graph Construction 10.5.3 (Composite Binary Tree DAG)**.** *Let B_h^C be a composite binary tree DAG with height h constructed in the following way where T is the intended set of targets of our DAG. Let $s = |T|$. In our intended construction $h = s$, also.*

1. *Let the set of nodes be V. Let the set of edges be E.*
2. *Create $(s+1)2^{h-1} + s$ nodes.*
3. *Create $s+1$ binary search trees using $(s+1)2^{h-1}$ nodes in total where edges are directed from children to parents in each binary tree. Let r_i for $i \in [1, s+1]$ be the roots of these binary search trees.*
4. *Order the remaining nodes in some arbitrary order, let s_j be the jth node in this order for $j \in [1, s]$.*
5. *Create directed edges (r_i, s_i) and $(r_{i+1 \bmod s}, s_i)$ for all $i \in [1, s]$.*

The set of target nodes consists of all nodes with 0 out-degree.

Given any binary search tree with height h, the minimum number of pebbles necessary to pebble the tree is h (assuming a 'tree' with one node has height 1) using the rules of the standard pebble game. Therefore, to ensure that the apex of the tree is pebbled and that both the honest party and the adversary both use h space to pebble the apex, the number of leaves necessary at the base of the tree is 2^{h-1}. If we suppose that the computationally weak honest party (who does not build special circuits) can only evaluate a constant number of random oracle calls at a time (place a constant number of pebbles), the number of sequential evaluations necessary for the honest party is $\geq \Omega(2^h)$ which is infeasible to accomplish. In constrast, the adversary only has to make $O(h)$ parallel random oracle calls, an exponential factor difference between the honest party and the adversary! Such a construction fails since it is clearly infeasible for the honest party since they would never be able to compute all target values of $\mathcal{H}_1$ from scratch (since this computation requires exponential time for the honest party). Thus, we would like a construction that has the same minimum space requirement but also small sequential evaluation time. We prove a better (but also simply defined) construction below.

Cylinder construction

We make use of what is defined in the pebbling literature as a *pyramid graph* [GLT80] in constructing our *cylinder graph*. The key characteristic of the pyramid graph we use is that the number of pebbles that is required to pebble the apex of the pyramid is equal to the height of the pyramid [GLT80] using the rules of the standard pebble game. Note that a pyramid by itself is not useful for our purposes since the black-magic pebbling space complexity of a pyramid with one apex is 1. Therefore, we need to be able to use the pyramid in a different construction that uses superconstant number of pebbles in the magic pebble game in order to successfully pebble all target nodes.

Graph Construction 10.5.4 (Illustrated in Fig. 10-2)**.** *Let Π_h^C be a* cylinder *graph with height h. We define Π_h^C as follows:*

1. *Create $2h^2$ nodes. Let this set of $2h^2$ nodes be V.*
2. *Arrange the nodes in V into $2h$ levels of h nodes each, ranging from level 0 to level $2h-1$. Let the j-th node in level i be v_i^j. Create directed edges $(v_i^{j \bmod h}, v_{i+1}^{j \bmod h})$ and $(v_i^{j \bmod h}, v_{i+1}^{(j+1) \bmod h})$ for all $i \in [0, 2h-2]$. Let this set of edges be E.*

The set of target nodes consists of all nodes with 0 out-degree.

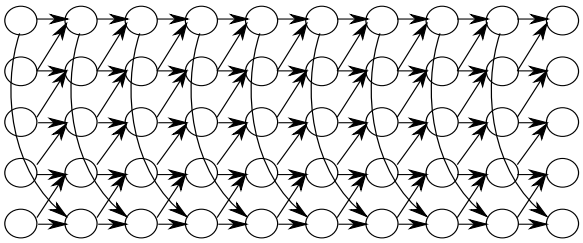


Figure 10-2: Cylinder construction (Def. 10.5.4) for $h = 5$.

Lemma 10.5.5. *Given a cylinder graph with height h, Π_h^C, $\mathbf{P}_s(\Pi_h^C, T) \geq h$.*

Proof. Let T be the target nodes of Π_h^C. Each target node is connected to a pyramid of height h. Therefore, by the proofs of minimum pebbling cost of pyramids given in [GLT80], the pyramid requires h pebbles to pebble using the rules of the standard pebble game. Therefore, to pebble any one target node $t \in T$ requires h pebbles, so pebbling all target nodes of Π_h^C, T, trivially requires h pebbles. □

Lemma 10.5.6. $\mathbf{P}_{\text{opt-ss}}(\Pi_h^C, T) \geq 2h$.

Proof. The depth of Π_h^C is $2h$ (i.e. the longest directed path in Π_h^C has length $2h$). Thus, the minimum number of parallel steps necessary to pebble any $v \in T$ is $2h$. Let L_i be the set of nodes at the i-th level of Π_h^C where T is at level $2h-1$ and S is at level 0. To pebble each target node requires that all vertices in L_{h-1} (v_{h-1}^i for all $i \in [1, h]$) be pebbled at some time step t simultaneously[24], $t \in [0, t_{\text{status}}]$, by normality of pebbing strategies[25]

[24] Whereby 'simultaneously', we mean there exists some time t' where all vertices in L_{h-1} are pebbled.

[25] See the definition of *frugal* and *normal* strategies in Definitions B.2.1 and B.2.2 [GLT80, DL17].

given any normal strategy status. Thus, at least h parallel time steps where h pebbles are on the graph simultaneously are necessary to pebble any target $v \in T$ because to pebble all nodes in L_{h-1} at time t requires h parallel time steps where h pebbles are used at each time step.

Suppose for contradiction that $\mathbf{P}_{\text{opt-ss}}(\Pi_h^C, T) < 2h$. We first prove that to pebble any k targets (where $k \leq h$) simultaneously require at least k time steps (where each time step is larger than t defined above) where h pebbles are on the graph simultaneously. Furthermore, there exists time steps $t_{l-1} > t_{l-2} > \cdots > t_1 > t$ where h pebbles are on all vertices in L_{h-1+j} (v^i_{h-1+j} for all $i \in [1,h]$) at time t_j. We prove this by induction. Let the base case be $k = 1$. In order to pebble any target $v \in T$ using a normal strategy status, there must be a time step $t_1 > t$ where h pebbles are on all vertices in L_h (v^i_h for all $i \in [1,h]$) by normality of pebbling strategies (see Theorem B.2.3 [GLT80]). We assume as our induction hypothesis that the statement is true for all $k \leq l-1$ where $l \leq h$. We now prove the statement for $k = l$. At time t_{l-1}, there exist h pebbles on all vertices in L_{h+l-2} by definition of t_{l-1} and by our induction hypothesis. By inspection, to pebble any subset of l targets requires all vertices in L_{h+l-1} to be pebbled at some point in the execution of the pebbling strategy. Suppose there exists a strategy that pebbles k targets using at most $k-1$ parallel moves where h pebbles are on the graph during each of the $k-1$ parallel moves. By our induction hypothesis, pebbling any $k-1$ sized subset of the k targets requires $k-1$ parallel moves where h pebbles are on the graph and all nodes in L_{h+k-1} for all $k < l$ are pebbled simultaneously at time t_k. If no more than $h-1$ pebbles can be on the vertices in L_{h+l-1}, this means that there exists a vertex in L_{h+l-1} that must be pebbled with at least l pebbles (given there exists a previous time step when h pebbles are on all vertices in L_{h+l-2} and no more than $h-1$ of these pebbles can be moved to the vertices in L_{h+l-1}). Let this vertex be u. If we continue strategy status without pebbling u, then there will exists a vertex at every level $h + l' - 1$ (for all $l' \geq l$) where l' pebbles are necessary to pebble the vertex. Thus, the lower bound on the minimum number of pebbles necessary to pebble k targets using strategy status is $h - 1 + l'$ at some time step $t_{l'} > t_{l-1}$, a contradiction since $l' \geq 1$.[26]

Given that to pebble any k targets requires at least k time steps (inaddition to the h timesteps necessary to pebble all nodes in L_{h-1}) where h pebbles are on the graph simultaneously. Thus, pebbling all targets using any strategy that pebbles sequentially subsets of targets $S_1, \ldots, S_d$ where $\bigcup_{i=1}^d S_i = T$ results in $\sum_{i=1}^d |S_i| \geq h$ steps where h pebbles are on the graph simultaneously. In all cases, we reach a contradiction with $\mathbf{P}_{\text{opt-ss}}(\Pi_h^C, T) < 2h$. Therefore, $\mathbf{P}_{\text{opt-ss}}(\Pi_h^C, T) \geq 2h$. □

Theorem 10.5.7. *Using the rules of the standard pebble game, h pebbles are necessary for at least h parallel steps to pebble* any *target of a height $2h$ cylinder graph, Π_h^C.*

Proof. To pebble any target of Π_h^C requires h pebbles on all nodes in level h by normality of pebbling strategies. Given at most h pebbles, to pebble any subset k of nodes in level h (by the normality of pebbling strategies) require h pebbles to be present on the graph

[26] Note that a simpler proof can be shown to state that at least h pebbles are needed to pebble u at level l' but we present the present proof to show that even for a cylinder with height h (instead of $2h$) our proof here still holds–i.e. h steps where h pebbles are on the cylinder are necessary to pebble all targets T.

for at least k parallel time steps as proven in the proof for Lemma 10.5.6. Thus, given a pebbling strategy that pebbles the following subsets of nodes in level h sequentially, $S_1, \ldots, S_d$ where $T = \bigcup_{i=1}^{d} S_i$, the number of time steps where h pebbles are on the graph is given by $\sum_{i=1}^{d} |S_i| \geq h$. Therefore, h pebbles are on the graph during at least h time steps when pebbling any target of Π_h^C, proving our theorem. □

Theorem 10.5.8. $\mathbf{P}_s(\Pi_h^C, |T|, T) \geq h$ *where* Π_h^C *is defined as in Def. 10.5.4 where* $|S| = |T| = h$.

Proof. Assume for the sake of contradiction that $s < h$ pebbles can be used to pebble all target nodes in T. By the rules of the black-magic pebble game, we can choose to use either magic pebbles or black pebbles at each time step in a valid strategy.

We first prove that given $s < |T|$ magic pebbles, one would choose to place the pebbles on s target nodes as opposed to any number of intermediate nodes. Let L_i be the set of nodes at the $i + h$-th level of Π_h^C (for $0 \leq i \leq h-1$) where T is at level $2h-1$ and S is at level 0. Given s adjacent pebble placements on nodes in L_i, we can pebble at most $j \leq \max(0, s + i - h + 1)$ target nodes by construction of Π_h^C without performing any repebbling of any nodes in S. (Note that we do not need to account for the case when $s < |T|$ pebbles are placed on levels 0 to $h-1$ since no targets can be pebbled if that is the case.) If repebbling of any node in L_i needs to be done (using black pebbles), then at least h total pebbles are necessary to pebble T. We now show this is true. Suppose that in order to pebble a target node $v \in T$, there exist at most $h-i-1$ magic pebbles on adjacent nodes in L_i. Then, at least 1 additional pebble is necessary at some node in L_i to pebble v. Let the node that needs to be pebble in L_i be w. Suppose that we use a black pebble to pebble w at level i (i.e. we wouldn't choose to use magic pebbles to pebble the ancestors of w since that would use more magic pebble than if we used a magic pebble to pebble w). Note that w is the apex of a pyramid of height at least $i+1$. Therefore, at least $i+1$ black pebbles are necessary to pebble w resulting in $i + 1 + h - i - 1 = h$ total pebbles necessary to pebble v, which is greater than the initial $h-i-1$ magic pebbles in total pebble count for all $i \in [0, h-1]$ (our desired range of values of i). Note that this argument applies recursively to any number $i' \leq i$ missing pebbles at level i.

Therefore, for any number of magic pebbles $s' \leq s$ that are *not* on target nodes, we can obtain at most $s' - 1$ target values without performing repebbling of any nodes in S. It is then strictly more efficient to pebble s' target nodes with magic pebbles instead of s' non-target nodes. We can have a total of $s < h$ magic pebbles which is not enough pebbles to pebble all the target nodes. To pebble the target node that is not pebbled by a magic pebble, we require h additional pebbles by pebbling price of pyramids [GLT80], contradicting our assumption. □

As a simple extension of our theorem and proof above, we get Corollary 10.5.9. Moreover, as an extension of the proof given for Theorem 10.5.8 that all magic pebbles are placed on targets and from Theorem 10.5.7, we obtain Corollary 10.5.10.

Corollary 10.5.9. *Given a cylinder* $G = (V, E)$ *as constructed in Graph Construction 10.5.4,* G *is incrementally hard:* $\mathbf{P}_s(G, |C| - 1, C) \geq |T|$ *for any subset* $C \subseteq T$.

Corollary 10.5.10. *Given a cylinder $G = (V, E)$ as constructed in Graph Construction 10.5.4, $\mathbf{P}_{\text{opt-ss}}(G, |C| - 1, C) = \Theta(|T|)$ for all subsets of $C \subseteq T$.*

A logical question to ask after constructing our very simple hash function based on a cylinder graph is whether such a construction is optimal in terms of graph-optimal sustained complexity *and* follows our requirements for a static-memory-hard hash function. As it turns out, the graph-optimal sustained complexity of a cylinder graph is optimal in the class of layered graphs. In other words, if we choose to use layered graphs in our constructions, then we cannot hope to get a better memory and time guarantee. From an implementation and practical standpoint, layered graphs are easier to implement and hence this result has potential practical applications (as more complicated constructions need to consider memory allocation factors in the real-life implementation, not considered in the theoretical model).

Theorem 10.5.11. *Given a* layered graph, *$G = (V, E)$, if the number of target nodes is $|T| = s$ and $\mathbf{P}_s(G, s, T) \geq s$, then $|V| = \Omega(s^2)$. A* layered graph *is one such that the vertices can be partitioned into layers and edges only go between vertices in consecutive layers.*

Proof. In order to satisfy $\mathbf{P}_s(G, s, T) \geq s$, the number of targets has to be at least s; if $|T| < s$, then T can be completely pebbled with less than s magic pebbles and $\mathbf{P}_s(G, s, T) < s$. Suppose the sources (the first level) are at level 0 and the targets (the last level) are at level $h - 1$ where h is the height of the layered graph. In any layered graph with in-degree 2, the cost of pebbling a vertex v_i in level i is at most $i + 1$ [Nor15]. Therefore, the height of G must be at least $s - 1$, in order for $\mathbf{P}_s(G, s, T) \geq s$. Let $h = s - 1$. In order for $\mathbf{P}_s(G, s, T) \geq s$, the width of the layered graph in layer j for all $j \in \left[\frac{h}{2}, h - 1\right]$ must be at least $\frac{h}{2}$ (where by width, we mean the number of nodes in layer j).

Suppose that a layer j where $j \in \left[\frac{h}{2}, h - 1\right]$ has width less than $\frac{h}{2}$. We can subsequently use less than $\frac{h}{2}$ magic pebbles to pebble layer j. Then, at most $\frac{h}{2}$ black pebbles are necessary to pebble all targets in T resulting in $\mathbf{P}_s(G, s, T) < h$ and $\mathbf{P}_s(G, s, T) < s$ (by our definition of h), a contradiction. The total number of nodes in layers $[\frac{h}{2}, h - 1]$ must then be at least $\frac{h^2}{4}$, and $|V| = \Omega(h^2) = \Omega(s^2)$. □

Thus, our construction of the cylinder graph is optimal in terms of amount of memory used in the asymptotic sense for the class of layered graphs. An open question is whether this is also optimal when we consider the larger class of all DAGs.

Open Question. *Does Thm 10.5.11 also hold for general graphs with bounded in-degree 2?*

Given the impossibility of providing a better space guarantee for layered graphs, we provide a general (non-layered) construction that transforms a graph from a certain class into another graph with the same space guarantee as in Theorem 10.5.11. Furthermore, we provide an example below that has the same space guarantees but a better time guarantee.

Layering *shortcut-free* graphs

We now show how to convert any *shortcut-free* DAG, $G = (V, E)$, with $\mathbf{P}_s(G, T) = s$ and one target node (i.e. $|T| = 1$) into a DAG, $G' = (V', E')$, with $|T'| = s$ targets and $\mathbf{P}_s(G', s, |T'|) = s$.

Definition 10.5.12 (Shortcut-Free Graphs). *Let $G = (V, E)$ be a DAG where $\mathbf{P}_s(G, T) \geq s$. Let t_s^{status} be the last time step that exactly s pebbles must be on G during any normal and regular pebbling strategy, status, (see Thms B.2.3 and B.2.5, [GLT80, DL17]) that uses s pebbles. More specifically, let*

$$t_s^{status} = \arg \max_{t' \in [t_{status}]} \{|P_{t'}| : |P_{t'}| \geq \mathbf{P}_s(G, T)\}$$

where $t_{status} = |status|$ and $P_{t'} \in status$ for all $t' \in [t_{status}]$. Let X be the union of the set of nodes that are pebbled at t_s^{status} for all normal and regular strategies status: $X = \bigcup_{status \in \mathbb{P}} P_{t_s^{status}}$. Let D be the set of descendants of nodes of X. A DAG is shortcut-free *if $|X| \leq s$ and given $s_1 < s$ pebbles placed on any subset $X_1 \subset X$, no normal and regular strategy uses less than $s - s_1$ pebbles to pebble $D \cup (X \backslash X_1)$.*

Graph Construction 10.5.13. *Given a shortcut-free DAG, $G = (V, E)$, with $\mathbf{P}_s(G, T) = s$ and $|T| = 1$, we create a DAG, $G' = (V', E')$, with the following vertices and edges and with the set of targets T' where $|T'| = s$. Let X be defined as in Definition 10.5.12.*

1. *V' is composed of the nodes in V and $s - 1$ copies of $X \cup D$. Let the i-th copy of X be X_i (the original is X_0) and let the i-th copy of $x \in X_i$ be x_i.*
2. *E' is composed of the edges in E and the following directed edges. If $(v, w) \in E$ and $v, w \in X$, then create edges $(v_i, w_i) \in E'$ for all $i \in [1, s-1]$. Create edges $(u, v_i) \in E'$ if $(u, v) \in E$ and $u \in V \setminus (X \cup D)$.*
3. *The set of targets T' is the union of the set of targets of the different copies: $T' = \bigcup_{i=0}^{s-1} T_i$.*

Using the above construction, we have created a graph $G' = (V', E')$ where $|V'| = |V| + (s-1)(|D| + |X|)$ and $|T'| = s$.

Theorem 10.5.14. *Given a shortcut-free DAG $G = (V, E)$ with $\mathbf{P}_s(G, T) = s$ and $|T| = 1$, the construction produced by Graph Construction 10.5.13 produces a DAG $G' = (V', E')$ such that $\mathbf{P}_s(G', s, |T|) = s$.*

Proof. We first prove that $\mathbf{P}_s(G', s, |T|) \leq s$. Since there are s different targets, $\mathbf{P}_s(G', s, |T|) \leq s$ trivially.

We now prove that $\mathbf{P}_s(G', s, |T|) \geq s$. If only black pebbles are used to pebble the targets in T', then s black pebbles must trivially be used provided $\mathbf{P}_s(G, T) = s$. Suppose some number of magic pebbles are used. Using the magic pebbles on any node in a copy of D (defined in Def. 10.5.13) that is not a target in T' is strictly worse than using a magic pebble on a target. Suppose the total number of pebbles used is less than s. We first prove that no magic pebbles are used on copies of D. If the total number of pebbles used is less than s, then not all of the s targets can be pebbled using magic pebbles. The remaining target that is not pebbled must be pebbled using s black pebbles since $\mathbf{P}_s(G, T) = s$ by definition. By the same logic, no magic pebbles are used on the nodes in the copies of X.

Therefore, if less than s magic pebbles are used to pebble the graph, all magic pebbles should be used to pebble the predecessors of X. No magic pebble can be removed and repebbled since such a magic pebble must be placed s times (once for each copy of X and D), exceeding the maximum number of magic pebbles we can have. Given that we can

use a total of less than s magic pebbles to pebble the predecessors of X, suppose some $s' < s$ pebbles are used, then less than $s - s'$ pebbles are left to pebble each copy of X and D; by incremental hardness, less than $s - s'$ cannot be used to pebble each copy of X and D. At least one magic pebble is used on the predecessors of X; by our definition of shortcut-free, less than $s - 1$ pebbles cannot be used to pebble X and D, a contradiction. Thus, $\mathbf{P}_s(G', s, T) \geq s$. □

If $D = \Theta(s)$ and $s = O(\sqrt{|V|})$, then $|V'| = \Theta(s^2 + |V|)$ which has a better sustained time guarantee than our cylinder construction.

We first note that the sustained memory graphs presented in [ABP17a] *do not* achieve optimal local memory hardness because $X \cup D$ (as defined in Definition 10.5.13) is $\Theta(n)$ (since the sources are the ones that remain pebbled in their construction). Thus, we would like to provide a construction of a shortcut-free DAG where $|X \cup D| = \Theta(s)$. Note that the size of $X \cup D$ will always be $\Omega(s)$, trivially. We now provide a definition of a shortcut-free graph class G that can be transformed using Definition 10.5.13.

Graph Construction 10.5.15 (Illustrated in Fig. 10-3)**.** *Let $G = (V, E)$ be a graph defined by parameter s and in-degree 2 with the following set of vertices and edges:*

1. *Create a height s pyramid. Let r_i be the root of a* subpyramid *(i.e. a pyramid that lies in the original height s pyramid) with height $i \in [2, s]$. One can pick any set of these subpyramids.*
2. *Topologically sort the vertices in each level and create a path through the vertices in each level (see Fig. 10-3). Replace any in-degree-3 nodes with a pyramid of height 3, with a 6-factor increase in the number of vertices.*
3. *Create $c_1 s$ additional nodes for some constant $c_1 \geq 2$ (in Fig. 10-3, $c_1 = 4$). Label these nodes v_j for all $j \in [1, c_1 s]$. Create edges (v_i, v_{i+1}) for all $i \in [1, c_1 s - 1]$.*
4. *Create directed edges (r_s, v_1) and $(r_i, v_{i-1+(k-1)(s-1)})$ for all $k \in [1, s]$.*
5. *Create $s - 1$ additional nodes. Let these nodes be w_l for all $l \in [1, s - 1]$.*
6. *Create directed edges $(v_{c_1 s}, w_1)$ and (r_i, w_{i-1}) for all $i \in [2, s]$.*
7. *The target node is w_{s-1}.*

Lemma 10.5.16. *Given a DAG $G = (V, E)$ and a parameter s where G is defined by Definition 10.5.15, $\mathbf{P}_s(G, T) = s$.*

Proof. In order to pebble the apex of the pyramid of height s, we must use at least s pebbles as proven in the proof for black pebbling cost of pyramids [GLT80]. □

Before we prove that $G = (V, E)$ created by Definition 10.5.15 with parameter s is shortcut-free, we first prove the following stronger lemma which will help us prove that G is shortcut-free.

Lemma 10.5.17. *Let $G = (V, E)$ be a graph created using Definition 10.5.15 with parameter s. Given a normal strategy status to pebble G, when v_q for $q \in [1, c_1 s]$ is pebbled at some time step, black pebbles equal in number to the number of nodes in $[r_i, r_s]$ are always present on the graph where $i = (q \bmod s - 1) + 1$ from the time when v_1 is pebbled to when v_q is pebbled.*

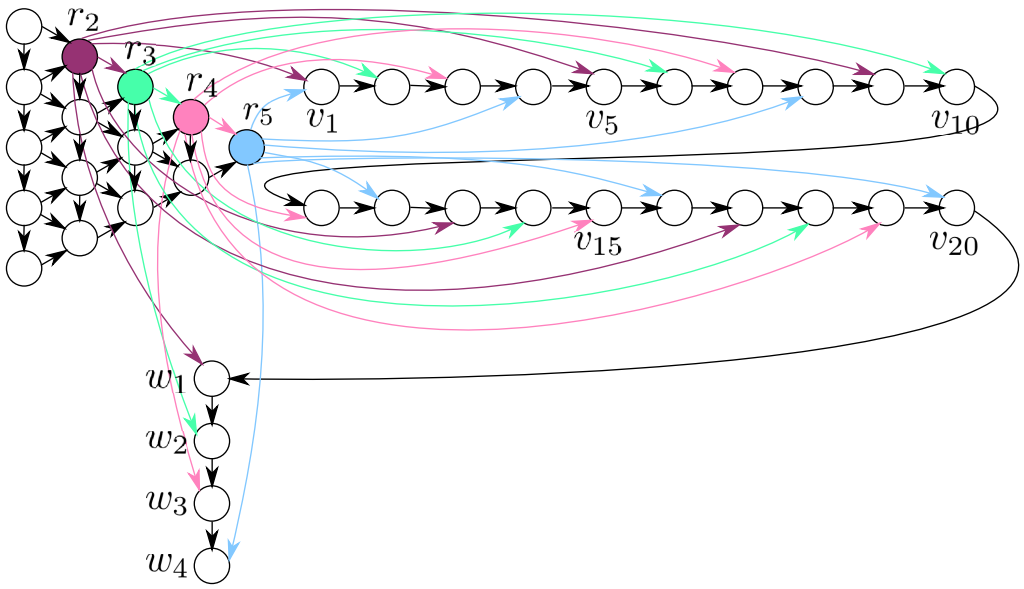


Figure 10-3: Example of a time optimal graph family construction as defined in Def. 10.5.15. Here, $s = 5$. Note that one can replace *any* constant in-degree node with in-degree c with a pyramid of size $\frac{c(c+1)}{2}$. For the sake of visual simplicity, such replacements are not shown in this figure.

Proof. We prove this lemma via induction.

In our base case when $i = s$, when the corresponding v_q is pebbled, a black pebble must be on r_s and v_{q-1} in the previous time step. Thus, a black pebble remains on r_s from the time v_1 is pebbled till the time that v_q is pebbled or a set of $s-j$ black pebbles remain on the j-th level of the pyramid for some $j \in [0, s-1]$ (in which case we can charge one of these pebbles to be "present on r_s"). Suppose neither of these conditions are met. Then, by the pebbling number of pyramids (see Thm B.2.6, [Nor15]), at least s pebbles must be used to pebble r_s, contradicting the frugality of status (since at most s pebbles are used to pebble G). In general, we make the observation that if there $s-j$ pebbles on some level $j \in [0, s-1]$, then we can charge these $s-j$ pebbles to be "on all nodes in $[r_j, r_s]$".

For our induction hypothesis, we assume that the theorem is true for j and prove the stratement for $i = j-1$. When $i = j-1$ and the corresponding v_q is pebbled, we assume by our induction hypothesis that there are $s-j$ black pebbles present on $[r_j, r_s]$ (or charged to be on $[r_j, r_s]$) from when v_1 is pebbled to when v_q is pebbled. In order to pebble v_q, there must be black pebbles on r_i and v_{q-1}. If there does not exist a black pebble on r_i (or on the predecessors of r_i) from when v_1 is pebbled to when v_q is pebbled, then at least one pebble must be removed from some $r \in [r_j, r_s]$ or from v_{q-1} since at least $j-1$ pebbles are necessary to pebble r_{j-1} ($s-j+2$ pebbles are currently in use–leaving not enough pebbles to pebble r_{j-1} unless a pebble is removed). If the black pebble is removed from v_{q-1}, the frugality of status is contradicted. If the black pebble is removed from some $r \in [r_j, r_s]$, then by observation, r_s will need to be repebbled sometime in the future, also a contradiction to the frugality of status. Thus, we prove our statement. □

Lemma 10.5.18. *Given a DAG $G = (V, E)$ and a parameter s where G is defined by Defi-*

nition 10.5.15, G is shortcut-free.

Proof. We first prove that any normal standard pebbling strategy status that pebbles G must contain pebbles on all r_i and $v_{c_1 s}$ at some time (say, t_X) during the execution of status.

Let X be the set of vertices containing black pebbles when $v_{c_1 s}$ is pebbled. Thus, a total of s pebbles must be on the graph (specifically on all nodes in X) at this time in any normal strategy by proof of Lemma 10.5.17. We now prove the incremental hardness of G. Let $s' < s$ pebbles be on X at time t_X. We prove that we cannot pebble $X \cup D$ using less than $s - s'$ pebbles.

Suppose for the purposes of contradiction, given $s' < s$, assume that s' pebbles are placed on X and less than $s - s'$ pebbles can be used to pebble $X \backslash X' \cup D$. Supose that $X \backslash X'$ includes either:

1. $v_{c_1 s}$ and some $s - s' - 1$ subset of vertices in $[r_2, r_s]$, or
2. some $s - s'$ subset of vertices in $[r_2, r_s]$.

In the first case, if no pebbles are on v_i for $i \in [1, c_1 s]$, then at least one pebble needs to be used to pebble v_i for $i \in [1, c_1 s]$. If $X \backslash X'$ includes some $s - s' - 1$ subset of vertices in $[r_2, r_s]$, then at least $s - s'$ pebbles are needed to pebble the vertices missing the pebbles.

In the second case, if some subset $s - s'$ of vertices in $[r_2, r_s]$ are in $X \backslash X'$, then at least $s - s' + 1$ pebbles are necessary to pebble the nodes missing pebbles in order to be able to pebble w_l for $l \in [1, s - 1]$.

In either case, at least $s - s'$ pebbles are necessary to pebble $X \backslash X' \cup D$, thus, this construction is shortcut-free. □

Theorem 10.5.19. *s pebbles are necessary for at least $\Theta(s^2)$ parallel steps to pebble any target of G'.*

Proof. To pebble v_j for all $j \in [1, c_1 s]$, we require pebbles on all r_i for $i \in [2, s]$ and one pebble on the path from v_1 to $v_{c_1 s}$; otherwise, the entire pyramid must be rebuilt, resulting in repebbling all nodes in the graph as we showed in the proof of Lemma 10.5.17. To pebble the pyramid requires s pebbles on the pyramid at all times and takes $\Theta(s^2)$. We show this is true.

Suppose that at some point before pebbling the apex of the pyramid that a pebble is removed from the graph, then, by our requirement that $s - 1$ pebbles must remain on r_i for $i \in [2, s]$ and that a pebble must be on the path from v_1 to $v_{c_1 s}$, the removed pebble cannot be used for either of these tasks. Thus, the entire pyramid must be rebuilt, contradicting the frugality of the strategy.

Thus, s nodes must remain on the graph for $\Theta(s^2 + c_1 s) = \Theta(s^2)$ parallel time steps, proving our theorem. □

We create $G' = (V', E')$ from G (as constructed using Definition 10.5.15) using Definition 10.5.13 , resulting in a graph with $\Theta(s^2)$ total nodes.

Theorem 10.5.20. $\mathbf{P}_s(G', s, T) = s$.

Proof. By Lemma 10.5.18 the graph is shortcut-free and by Lemma 10.5.16 $\mathbf{P}_s(G, T) = s$, therefore, we use Theorem 10.5.14 to prove that $\mathbf{P}_s(G', s, T) = s$. □

By the proof that G' is shortcut-free, we obtain the following corollary that G' is also incrementally hard. Moreover, Corollary 10.5.22 follows directly from the proof of Theorem 10.5.14.

Corollary 10.5.21. *Given a graph $G = (V, E)$ as constructed in Graph Construction 10.5.15, G is incrementally hard: $\mathbf{P}_s(G, |C| - 1, C) \geq |T|$ for any subset $C \subseteq T$.*

The following corollary about the graph-optimal sustained time complexity is proven directly from the proof of Lemma 10.5.17 and Theorem 10.5.19 that if less than $\frac{s}{2}$ magic pebbles are on the pyramid, then half the pyramid must be rebuilt resulting in $\Theta(s^2)$ time-steps in which s pebbles are on the graph; thus proving for the cases when $|C| - 1 < \frac{s}{2}$. We now prove the case when $|C| - 1 \geq \frac{s}{2}$.

Corollary 10.5.22. *Given a graph $G = (V, E)$ as constructed in Graph Construction 10.5.15, $\mathbf{P}_{\text{opt-ss}}(G, |C| - 1, C) = \Theta(|V|)$ for all subsets of $C \subseteq T$.*

Proof. If $|C| - 1 \geq \frac{s}{2}$ magic pebbles are not placed on r_i for all $i \in [2, s]$, then we have to rebuild at least half the pyramid, resulting in $\Theta(s^2) = \Theta(|V|)$ time being used. Thus, some $s' \geq \frac{s}{2}$ magic pebbles must be used on r_i for all $i \in [2, s]$. Then, to pebble all $|C| \geq \frac{s}{2}$ targets requires $\Theta(s^2) = \Theta(|V|)$ time using another black pebble since $s' \geq \frac{s}{2}$ pebbles are used on the pyramid. □

10.5.2 $\mathcal{H}_2$ construction

Our construction of $\mathcal{H}_2$ is presented in Algorithm 33.

Algorithm 33 $\mathcal{H}_2$

On input $(1^\kappa, x)$ and given oracle access to Seek_R (where R is the string outputted by $\mathcal{H}_1$):

1. Let $[\![R]\!] = |R|/w$ be the length of R in words.
2. Query the random oracle to obtain $\rho_0 = \mathcal{O}(x)$ and $\rho_1 = \mathcal{O}(x + 1)$.
3. Use ρ_0 to sample a random $\iota \in [[\![R]\!]]$.
4. Query the Seek_R oracle to obtain $y' = \mathsf{Seek}_R(\iota)$.
5. Output $y' \oplus \rho_1$.

Lemma 10.5.23. *For any R, the output distribution of $\mathcal{H}_2$ is uniform over the choice of random oracle $\mathcal{O} \leftarrow \mathbb{O}$.*

Proof. Over the choice of random oracle, the value ρ_1 computed in Step 2 is truly random, and y' is independent of ρ_1 by construction, so the output $y' \oplus \rho_1$ is also truly random. □

Remark 10.5.24. *Lemma 10.5.23 is important as an indication that our SHF construction "behaves like a random oracle." The memory-hardness guarantee alone does not assure that the hash function is suitable for cryptographic hashing: e.g., a modified version of $\mathcal{H}_2$ which directly outputted y' instead of $y' \oplus \rho_1$ would still satisfy memory-hardness, but would be an awful hash function (with polynomial size codomain). The inadequacy of existing memory-hardness definitions for assuring that a function "behaves like a hash function" is discussed by [AT17].*

10.5.3 Proofs of hardness of SHF Constructions

We now prove the hardness of our graph constructions given earlier in Section 10.5.

We begin by stating two supporting lemmata. The first is due to Erdős and Rényi [ER61], on the topic of the Coupon Collector's Problem.

Lemma 10.5.25 ([ER61]). *Let Z_n be a random variable denoting the number of samples required, when drawing uniformly from a set of n distinct objects with replacement, to draw each object at least once. Then for any c, $\lim_{n\to\infty} \Pr[Z_n < n\log n + cn] = e^{-e^{-c}}$.*

Corollary 10.5.26. *Let $Z_{n,k}$ be a random variable denoting the number of samples required, when drawing uniformly from a set of n distinct objects with replacement, to have drawn at least $k \in [n]$ distinct objects. Let $q \in \omega(k\log k)$. Then $\Pr[Z_{n,k} < q]$ is overwhelming (in k).*

Proof. For $m \in$ and $i \in [m-1]$, let $\mathcal{E}_{i,m}$ denote the event that after i elements out of a set of m elements have already been sampled uniformly with replacement, the $(i+1)$th sample will coincide with one of the elements already drawn. For any $i \le k \le n$, it holds that $\Pr[\mathcal{E}_{i,n}] \ge \Pr[\mathcal{E}_{i,k}]$. The desired event of drawing k distinct objects corresponds exactly to the conjunction of $\mathcal{E}_{i,m}$ for $i \in [k]$. Therefore, for all $k \in [n]$ and any c',

$$\Pr[Z_{n,k} < c'] \ge \Pr[Z_k < c']\,. \tag{10.4}$$

Hence, it suffices for our purposes to bound $\Pr[Z_k]$. From Lemma 10.5.25,

$$\lim_{k\to\infty} \Pr[Z_k < k\log k + ck] = \lim_{k\to\infty} e^{-e^{-c}}.$$

Applying a Taylor expansion, we get $\Pr[Z_k < k\log k + ck] \in O(1-e^{-c})$. This probability is overwhelming in k (i.e., e^{-c} is negligible) whenever $c \in \omega(\log(k))$. □

Theorems 10.5.27–10.5.30 state the static-memory-hardness of our SHF constructions based on Graph Constructions 10.5.4 and 10.5.15.

Theorem 10.5.27. *Define a static-memory hash function family $(\mathcal{H}_1, \mathcal{H}_2)$ as follows: let $\mathcal{H}_1$ be the graph function family $\mathcal{F}^{\mathsf{C}}_{\Pi_h}$ (Graph Construction 10.5.4), and let $\mathcal{H}_2$ be as defined in Algorithm 33. Let $\mathcal{H} = \{h_\kappa\}_{\kappa\in}$ be the static-memory hash function family described by $(\mathcal{H}_1, \mathcal{H}_2)$. Let q_2 be the number of oracle queries made by $\mathcal{H}_2$, let $\hat\kappa = \kappa - \xi\log(q_2)$ for any $\xi \in \omega(1)$, let $\hat\Lambda \in O(\sqrt{n}), \tau \in \Theta(\sqrt{n})$, and let $q \in \omega(\Lambda\log\Lambda)$. Then $(\mathcal{H}_1, \mathcal{H}_2)$ is $(\hat\kappa\hat\Lambda, \hat\kappa, \tau, q)$-hard.*

Proof. Suppose, for contradiction, that the theorem does not hold. Then by Definition 10.4.2, there exist: $\kappa \in$, a string $R \in \{0,1\}^{\hat\kappa(\hat\Lambda-1)}$, an algorithm $\mathcal{A}$, and a set $X = \{x_1,\ldots,x_q\}$ such that the following probability is non-negligible:

$$\Pr_{\mathcal{O},\rho}\left[\left\{(x_1,h_\kappa(x_1)),\ldots,(x_q,h_\kappa(x_q))\right\} = \mathcal{A}(1^\kappa, R;\rho) \wedge \mathsf{s\text{-}mem}_{\mathbb{O}}(\hat\kappa\hat\Lambda, \mathcal{A}, R, \rho) < \tau\right]. \tag{10.5}$$

We denote by $\mathcal{E}_\rho$ the event that

$$\left\{(x_1,h_\kappa(x_1)),\ldots,(x_q,h_\kappa(x_q))\right\} = \mathcal{A}(1^\kappa, R;\rho) \wedge \mathsf{s\text{-}mem}_{\mathbb{O}}(\hat\kappa\hat\Lambda, \mathcal{A}, R, \rho) < \tau\,.$$

Given a correct evaluation $y = h_\kappa(x)$ of $\mathcal{H}_2$ on a given input x, one can easily compute ρ_0, ρ_1 by evaluating $\mathcal{O}$ on $x, x+1$ respectively, and demask y to obtain the value $y'(x) = y \oplus \rho_1$ of the target label computed in Step 4 of Algorithm 33. Moreover, the index ι computed in Step 3 can be computed as a deterministic function of ρ_0. Define $\mathcal{B}'$ to be the deterministic algorithm that on input (x, y) computes ρ_0, ρ_1 and y' as described above, and outputs (ι, y').

Next, define $\mathcal{B}$ to be the algorithm that runs $\mathcal{A}$ and then applies $\mathcal{B}'$ on each pair (x_i, y_i) outputted by $\mathcal{A}$, and outputs the resulting set $J = \{(\iota_1, y'_1), \ldots, (\iota_q, y'_q)\}$ where each $(\iota_i, y'_i) = \mathcal{B}'(x_i, y_i)$. By construction, if $y_i = h_\kappa(x_i)$, each y'_i is the correct label of ι_ith target node of the cylinder graph. Notice that this means that for each value of ι, there is a unique value of y' such that $(\iota, y') = \mathcal{B}'(x, h_\kappa(x))$ for any x.

Let I denote $|\{\iota_i\}_{x_i \in X}|$. Since the set X is fixed before the random oracle, the locations I are distributed uniformly and independently (with replacement). Then by Corollary 10.5.26, the number of distinct locations $|I|$ is at least $\hat{\Lambda}$ with overwhelming probability. That is, there is a negligible function ε' such that $\Pr\left[|I| \geq \hat{\Lambda}\right] \geq 1 - \varepsilon'$. Conditioned on $\mathcal{E}_\rho$, all pairs (x_i, y_i) outputted by $\mathcal{A}$ are such that $y_i = h_\kappa(x_i)$, and we have already observed that each value of ι induces a unique value of y' outputted by $\mathcal{B}'$ on input pairs of the form $(x_i, h_\kappa(x_i))$. It follows that $\Pr[|J| \geq \hat{\Lambda} \mid \mathcal{E}_\rho] \geq 1 - \varepsilon'$.

Now consider the ex-post-facto magic pebbling strategy status induced by $\mathcal{B}$. By Lemma 10.3.8, with overwhelming probability over the random oracle and the coins of $\mathcal{B}$, status is legal and uses at most

$$\left\lfloor \frac{|R|}{\hat{\kappa}} \right\rfloor = \left\lfloor \frac{\hat{\kappa}(\hat{\Lambda} - 1)}{\hat{\kappa}} \right\rfloor \leq \hat{\Lambda} - 1 \tag{10.6}$$

magic pebbles; call this event $\mathcal{E}'_\rho$ (where ρ denotes the randomness of $\mathcal{B}$). By Lemma 10.3.9, with overwhelming probability over the same,

$$\forall i \in [t],\ |P_i| \leq \left\lfloor \frac{|\sigma_i|}{\hat{\kappa}} \right\rfloor, \tag{10.7}$$

where t is the length of status, P_i is the ith configuration of status, and σ_i is the ith state of the execution of $\mathcal{B}$. We denote by $\mathcal{E}''_\rho$ the event that (10.7) is satisfied (where ρ denotes the randomness of $\mathcal{B}$). By definition, event $\mathcal{E}_\rho$ implies that $|\sigma_i| \geq \hat{\kappa}\hat{\Lambda}$ for fewer than τ values of i. Combining this observation with (10.7), we have that whenever $\mathcal{E}_\rho$ occurs, $|P_i| \geq \hat{\Lambda}$ for fewer than τ values of i.

Finally, we observe that conditioned on $\mathcal{E}_\rho$, since we established above that $\mathcal{B}$ outputs a set of at least $\hat{\Lambda}$ correct target labels, the strategy status must successfully pebble the corresponding $\hat{\Lambda}$ target nodes. Since $\Pr[\mathcal{E}_\rho]$ is non-negligible and $\Pr[\mathcal{E}']$ and $\Pr[\mathcal{E}'']$ are overwhelming, $\Pr[\mathcal{E}' \wedge \mathcal{E}'' | \mathcal{E}]$ must be negligibly close to $\Pr[\mathcal{E}]$ (and thus, non-negligible). The occurrence of $\mathcal{E} \wedge \mathcal{E}' \wedge \mathcal{E}''$ implies the existence of a pebbling strategy status that is legal, uses at most $\hat{\Lambda} - 1$ magic pebbles, and for which the number of time-steps in which at least $\hat{\Lambda}$ total (i.e., black and magic) pebbles are used is less than τ. This contradicts Corollaries 10.5.9–10.5.10. □

Theorem 10.5.28. *Define a static-memory hash function family $(\mathcal{H}_1, \mathcal{H}_2)$ as follows: let $\mathcal{H}_1$ be the graph function family $\mathcal{F}_G$ (Graph Construction 10.5.15), and let $\mathcal{H}_2$ be as defined in Algorithm 33. Let q_2 be the number of oracle queries made by $\mathcal{H}_2$, let $\hat{\kappa} = \kappa - \xi \log(q_2)$ for any $\xi \in \omega(1)$, let $\hat{\Lambda} \in O(\sqrt{n})$, let $\tau \in \Theta(n)$, and let $q \in \omega(\Lambda \log \Lambda)$. Then $(\mathcal{H}_1, \mathcal{H}_2)$ is $(\hat{\kappa}\hat{\Lambda}, \hat{\kappa}, \tau, q)$-hard.*

Proof sketch. Identical proof structure to the proof of Theorem 10.5.27, except instead of invoking Corollaries 10.5.9–10.5.10 at the end, we derive a contradiction to Corollaries 10.5.21–10.5.22. □

The parameter q is suboptimal in Theorems 10.5.27 and 10.5.28. We can achieve optimality (i.e., $q = \lVert |R| \rVert$) by the following alternative construction of $\mathcal{H}_2$: make $q' = \omega(\log(\kappa))$ random calls instead of just one call to the Seek oracle in Step 4. To preserve the output size of h_κ, it may be useful to reduce the size of node labels by a corresponding factor of q'. This can be achieved by truncating the random oracle outputs used to compute labels in Definition 10.3.4. The description of this altered $\mathcal{H}_2^{q'}$ and the definition of graph function family $\mathcal{F}_G^{q'}$ with shorter labels are given in Appendix B.1.

Theorem 10.5.29. *Define a static-memory hash function family $(\mathcal{H}_1, \mathcal{H}_2)$ as follows: let $\mathcal{H}_1$ be the graph function family $\mathcal{F}_{\Pi_h^C}^{\kappa/q'}$ (Graph Construction 10.5.4), and let $\mathcal{H}_2$ be $\mathcal{H}_2^{q'}$ as defined in Algorithm 39 for some $q' \in \omega(\log \Lambda)$. Let q_2 be the number of oracle queries made by $\mathcal{H}_2$, let $\hat{\kappa} = \kappa - \xi \log(q_2)$ for any $\xi \in \omega(1)$, let $\hat{\Lambda} \in O(\sqrt{n})$, $\tau \in \Theta(\sqrt{n})$, and let $q = \Lambda$. Then $(\mathcal{H}_1, \mathcal{H}_2)$ is $(\hat{\kappa}\hat{\Lambda}, \hat{\kappa}, \tau, q)$-hard.*

Proof sketch. Identical proof structure to the proof of Theorem 10.5.27, except that when invoking Corollary 10.5.26, due to the design of $\mathcal{H}_2^{q'}$ which calls Seek more times than $\mathcal{H}_2$, we obtain the stronger statement that an adversary that successfully outputs q pairs $((x_1, h_\kappa(x_1)), \ldots, (x_q, h_\kappa(x_q)))$ must correctly guess q target labels of the graph. □

Theorem 10.5.30. *Define a static-memory hash function family $(\mathcal{H}_1, \mathcal{H}_2)$ as follows: let $\mathcal{H}_1$ be the graph function family $\mathcal{F}_G^{\kappa/q'}$ (Graph Construction 10.5.15), and let $\mathcal{H}_2$ be $\mathcal{H}_2^{q'}$ as defined in Algorithm 39 for some $q' \in \omega(\log \Lambda)$. Let q_2 be the number of oracle queries made by $\mathcal{H}_2$, let $\hat{\kappa} = \kappa - \xi \log(q_2)$ for any $\xi \in \omega(1)$, let $\hat{\Lambda} \in O(\sqrt{n})$, let $\tau \in \Theta(n)$, and let $q = \Lambda$. Then $(\mathcal{H}_1, \mathcal{H}_2)$ is $(\hat{\kappa}\hat{\Lambda}, \hat{\kappa}, \tau, q)$-hard.*

Proof sketch. Identical proof structure to the proof of Theorem 10.5.29, except instead of invoking Corollaries 10.5.9–10.5.10 at the end, we derive a contradiction to Corollaries 10.5.21–10.5.22. □

10.6 Capturing nonlinear space-time tradeoffs with CC^α

Next, we motivate our notion of CC^α (Definition 10.2.21). We show that both the honest party and the adversary may choose to use different pebbling strategies given different

values of α even when α is constant. Furthermore, we show that both of our pebbling constructions of $\mathcal{H}_1$ (given in Section 10.5) have the desirable feature that the honest party *and* the adversary use the same strategy regardless of the size of α.

10.6.1 CC and CC$^\alpha$ consider cumulative cost of *different strategies*

We present a graph family with in-degree-2 where the strategy that an adversary chooses to pebble an instance G in the graph family differs depending on the α parameter of the CC$^\alpha$ complexity measure. We show that in our case, for certain α, we would choose to use constant space, whereas for other α, using superconstant space is the preferred option. We define our graph family as follows:

Graph Construction 10.6.1. *We define a graph family $\mathbb{G}$ with bounded degree 2 and arbitrary $n \in \mathbb{N}$ nodes such that the time-space tradeoff of a graph with n nodes in the family is $T(S) \geq (\frac{n^c}{n^a})(n^a - (S-2))(n^b) + n$ (where S is the number of pebbles used to pebble the graph) where $0 \leq a, b, c < 1$, $b + c > a + 1$, $a < b, c$, and $n^c \approx n - n^{a+b}$.*

- *Given a graph $G = (V, E)$ with n vertices, partition the set of vertices, V, into 2 sets, A and B where $|A| = n^{a+b}$ and $|B| = n^c$ (since we know $n^c \approx n - n^{a+b}$, $n^c + n^{a+b} \approx n$).*
- *We arbitrarily order all vertices in B in some order, $[v_i, \ldots, v_n]$ and create edges $(v_j, v_{j+1}) \in E$ for all $j \in [i, n-1]$.*
- *We arbitrarily order all vertices in A in some order, $[v_1, \ldots, v_{i-1}]$ and create edges $(v_j, v_{j+1}) \in E$ for all $j \in [1, i-2]$.*
- *We create edge (v_{i-1}, v_i).*
- *Create edges $(v_k, v_l) \in E$ ($v_k \in A$ and $v_l \in B$) where $k \bmod n^b = 0$ and $l = n^{a+b} + \left(\frac{k}{n^b}\right) + (q-1)n^a$ for all integers $q \in \left[1, \frac{n^c}{n^a}\right]$.*

Fig. 10-4 illustrates Graph Construction 10.6.1.

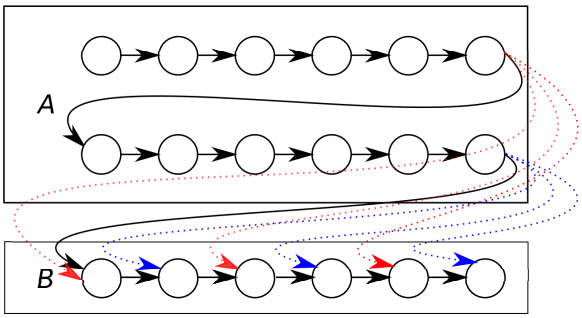


Figure 10-4: Graph Construction 10.6.1 with $n = 16$, $a = \frac{1}{4}$, $b = \frac{2}{3}$, $c = \frac{2}{3}$. For clarity, we depict $n^a = 2$, $n^b \approx 6$ and $n^c \approx 6$.

We show that there are at least two pebbling strategies, status$_1$ and status$_2$, where an adversary would differ in his preferred strategy depending on α when using the CC$^\alpha$ complexity measure when $\alpha > \alpha'$ where α' is calculated with respect to the parameters of the graph family constructed from Graph Construction 10.6.1.

Lemma 10.6.2. *Given a pebbling strategy $status_1$ that uses constant space S_1, $\mathsf{Time}(status_1) = \Theta(n^{b+c})$ where $G \in \mathbb{G}$ in defined by Graph Construction 10.6.1.*

Proof. Suppose that a constant S_1 pebbles can be on the graph at any particular time, then at most $S_1 - 1$ of the vertices in $A \subseteq V$ can be pebbled. It does not help to pebble the vertices in B since all vertices in B needs to be pebbled only once regardless of the pebbling strategy used. Since only the vertices $v_j \in A$ where $j \mod n^b = 0$ are connected to vertices in B, using the given S_1, the optimal placements are on vertices v_j in order to minimize pebbling time since any extra space needs to be used to pebble B and pebbling anywhere else results in greater pebbling time since the pebble needs to be moved to vertex v_j by the pigeonhole principle. Given constant S_1 pebbles, there exist v_j vertices that do not contain pebbles. Thus, each time one reaches a vertex in B with predecessor $v_j \in A$ without a pebble, at least n^b time must be spent to pebble it. Therefore, given S_1 pebbles, the total amount of time necessary to pebble G is $(\frac{n^c}{n^a})(n^a - S_1)(n^b) + n = \Theta(n^{b+c})$. □

Corollary 10.6.3. *Given a pebbling strategy, $status_1$, that uses constant space S_1, $\mathsf{p\text{-}cc}_\alpha(status_1) = \Theta(n^{b+c})$ where $G \in \mathbb{G}$ is constructed by Def. 10.6.1.*

Proof. This follows immediately from Lemma 10.6.2 since constant space is used throughout the pebbling. □

Lemma 10.6.4. *Given a pebbling strategy $status_2$ that uses space $S_2 = n^a + 1$, $\mathsf{Time}(status_2) = \Theta(n)$ where $G \in \mathbb{G}$ is constructed by Graph Construction 10.6.1.*

Proof. It is trivial to show that pebbling a line takes $\Omega(n)$ time since all nodes have to be pebbled at least once. We now show a strategy using $n^a + 1$ pebbles that uses $O(n)$ time.

We start with the vertices in A and pebble them in topological order, keeping pebbles on all $v_j \in A$ where $j \mod n^b = 0$. There exists exactly n^a vertices in A by definition that are predecessors of vertices in B. Therefore, as we pebble the vertices in A in topological order, we leave a pebble on each vertex v_j. When we pebble B all predecessors of vertices in B are either in B or are pebbled in A. Therefore, we only need to pebble all vertices in A and B once, resulting in $\mathsf{Time}(\mathrm{status}_2) = \Theta(n)$. □

The following corollary is directly proven by the proof of Lemma 10.6.2.

Corollary 10.6.5. *Given a pebbling strategy, $status_2$, that uses space $S_2 = n^a + 1$ and $\mathsf{Time}(status_2) = \Theta(n)$, $\mathsf{p\text{-}cc}_\alpha(status_2) = \Theta(n^{\alpha a+1})$ where $G \in \mathbb{G}$ is constructed by Graph Construction 10.6.1.*

Lemma 10.6.6. *When $\alpha = 1$, then $CC^\alpha(G) = \Theta(n^{a+1})$.*

Proof. Suppose in the case when $\alpha = 1$, we use a pebbling strategy, status, that uses nonconstant space $s \leq n^a + 1$. Then, for each pebble, we pebble one of the vertices $v_j \in A$ where $j \mod n^b = 0$. The resulting $\mathsf{p\text{-}cc}_\alpha(\mathrm{status}) = s(\frac{n^c}{n^a})(n^a - s)(n^b) + n$ which is minimized when $s = n^a + 1$ given $b + c > a + 1$ by definition of our graph family. □

Lemma 10.6.7. *For all a, b, c, there exists an α' such that for all constant $\alpha > \alpha'$, $CC^\alpha(G) = \Theta(n^{b+c})$.*

Proof. Given a pebbling strategy, status, that uses space $s = \omega(1)$, the pebbling cost is then $\text{p-cc}_\alpha(\text{status}) = s^\alpha(\frac{n^c}{n^a})(n^a - s)(n^b) + n$. When $\alpha > 1$, $\text{p-cc}_\alpha(\text{status}) = \Theta(\min(s^\alpha n^{b+c}, n^{\alpha a+1})) = \omega(n^{b+c})$ when $\alpha > \frac{b+c}{a}$ and $s = \omega(1)$. Therefore, only for $s = O(1)$, does the pebbling cost become $\text{p-cc}_\alpha(\text{status}) = \Theta(n^{b+c})$ when $\alpha' = \frac{b+c}{a}$. Since a, b, c are constants, for all $\alpha > \alpha'$, $\text{CC}^\alpha(G) = \Theta\left(n^{b+c}\right)$. □

From the above two lemmas, we immediately get the following theorem regarding the CC^α of the constructions given different constant values of α.

Theorem 10.6.8. *Given a graph $G = (V, E)$ as constructed by Graph Construction 10.6.1, when $\alpha = 1$, $CC^\alpha(G) = \Theta(n^{a+1})$ but when $\alpha > \alpha'$ for some constant α', $CC^\alpha(G) = \Theta(n^{b+c})$.*

As an immediate result of the above, there exists a point for constants a, b, c that the adversary chooses a different strategy to pebble a graph for different constant values of α (we can pick values of a, b, c such that α' can be reduced even down to $\alpha' \geq 3$).

10.6.2 Upper bounds for CC^α

We prove a tighter upper bound for CC^α when α is a constant than the trivial upper bound of $n^{\alpha+1}$. We first note that $n^{\alpha+1}$ is a trivial upper bound on the $\text{CC}^\alpha(G)$ of a graph, G, since at any timestep $\mathbf{P}_s(G, T) \leq n$ and the algorithm runs for $\text{Time}(G, |T|) \leq n$ given n space is used throughout. Therefore, $\text{CC}^\alpha(G) \leq n^{\alpha+1}$ for all graphs G. We now prove a tighter upper bound using the general pebbling algorithm described in [AB16] as $\text{GenPeb}(G, S, g, d)$.

We formulate a simplified version of the $\text{GenPeb}(G, S, g, d)$ procedure which we call the $\text{GenPeb}(G)$ procedure. At a high-level the $\text{GenPeb}(G)$ algorithm proceeds as follows (see [AB16] for more detail).

Definition 10.6.9 ($\text{GenPeb}(G)$:).

1. *There exists a subset S of $|S| \leq \frac{2\alpha n \log\log n}{\log n}$ vertices (for large enough n where $2\alpha \log\log n \leq \log n$) such that $\text{depth}(G - S) \leq \frac{n}{\log^\alpha n}$ (Lemma 6.1, 6.2 in [AB16], [Val77]).*
2. ***Balloon Phase:** Pebble all nodes up to depth $\frac{n}{\log^\alpha n}$ (depth measured from the last light phase) until all immediate descendants lie in S.*
3. ***Light Phase:** When all immediate descendants lie in S, remove all pebbles from nodes not in S and not on parents of the next nodes to be pebbled. Continue in the light phase until a node not in S must be pebbled.*
4. *Repeat the above until no more nodes need to be pebbled.*

Lemma 10.6.10. *Let s_Σ be the total number of pebbles used in the balloon phase (the sum of the number of pebbles used in all balloon phases) and $s_{\Sigma\setminus S}$ be the total number of pebbles used in the balloon phases on all nodes $v \notin S$. Then, $s_{\Sigma\setminus S} \leq n$.*

Proof. This proof is trivial since at most n pebbles can be the graph at any time. □

Lemma 10.6.11. *Let $\Sigma\setminus S$ be the subgraph of $G = (V, E)$ which is pebbled during the balloon phase and whose vertices are not in S. Then, $CC^\alpha(\Sigma\setminus S) \leq \frac{n^{\alpha+1}}{\log^\alpha n}$.*

Proof. By Lemma 10.6.10, the number of pebbles necessary to pebble $\Sigma \backslash S$ is at most n: $s_{\Sigma \backslash S} \leq n$. Therefore, we can compute $\mathrm{CC}^{\alpha}(\Sigma \backslash S) \leq \sum_{B_i \in \mathcal{B}} (|B_i|)^{\alpha} \leq n^{\alpha} (\frac{n}{\log^{\alpha} n}) = \frac{n^{\alpha+1}}{\log^{\alpha} n}$ given a series of balloon phase pebble configurations $\mathcal{B}$ where $\sum_{B_i \in \mathcal{B}} |B_i| = n$ and $B_0 \cup \cdots \cup B_{|\mathcal{B}|} = V$. □

Lemma 10.6.12. *Let $CC^{\alpha}(S)$ be the cost of pebbling S in both the light and the balloon phases. The $CC^{\alpha}(S)$ of the light and balloon phases is at most $O\left(\frac{n^{\alpha+1}(\log\log n)^{\alpha}}{\log^{\alpha} n}\right)$.*

Proof. The total amount of time that light and balloon phases last in which nodes in S are pebbled is at most n timesteps since a number greater than n implies that $|S| \geq n$ which is impossible since the number of nodes in the graph is n. In the light phases, at most $2|S| = \frac{4\alpha n \log\log n}{\log n}$ pebbles are kept on the graph since each node has bounded in-degree 2. Therefore, $\mathrm{CC}^{\alpha}(G) \leq \frac{4^{\alpha} \alpha^{\alpha} n^{\alpha+1} (\log\log n)^{\alpha}}{\log^{\alpha} n}$. □

Theorem 10.6.13. *For any bounded in-degree-2 graph, $CC^{\alpha}(G) = O\left(\frac{n^{\alpha+1}(\log\log n)^{\alpha}}{\log^{\alpha} n}\right)$ for constant $\alpha \geq 1$.*

Proof. This follows directly from Lemmas 10.6.11 and 10.6.12. □

10.6.3 Asymptotically tight sequential lower bound for $\alpha = 1$

We give an explicit construction of a graph that achieves asymptotically tight lower bound (up to $\log\log n$ factors) in CC^{α} that matches our upper bound provided in Section 10.6 for $\alpha = 1$ and in [AB16, ABP17b] *when considering the sequential pebbling model*[27] . Previous constructions [AB16, ABP17a] ignored $\log\log n$ factors and were not tight up to such factors in the parallel model. Because we consider the sequential pebbling model (and not the parallel model) in proving our lowerbound below, our results are incomparable to these previous lower bound results in the parallel model. Our graph constructions are new, and their tightness in the parallel pebbling model is an open question.

In our construction, we make use of the stacked superconcentrators constructed in [LT82, §4] except that the vertices are connected in some topological order (blowing up our graph by only a constant factor of 6 if we replace all degree 3 nodes with a height 3 pyramid).

Graph Construction 10.6.14. *Let $C(n,k)$ be a stacked superconcentrator with k layers where C_i is the i-th linear superconcentrator. We create the following edges between nodes. Let* T *be a topological sort order of the vertices in $C(n,k)$. Create edges (v_i, v_{i+1}) where v_i is the vertex immediately preceding v_{i+1} in* T*. Replace all degree* 3 *nodes with pyramids of height* 3*.*

It was proven in [LT82] (Theorem 4.2.6) that given $S \leq \frac{n}{20}$ pebbles, k layers, and n nodes in each linear superconcentrator per layer, the pebbling time, $T(n,k,S)$, of pebbling $C(n,k)$ is lower bounded by:

[27] Although our construction matches asymptotically the best lower bound construction in the pROM (see the footnote for Lemma 10.6.22).

$$T(n,k,S) = n\Omega\left(\left(\frac{nk}{64S}\right)^k\right).$$

In our construction defined by Def. 10.6.14, we first let $S = c_1(N \log\log N/\log N)$ (for some constant c_1), $n = 20S$, $k = \lfloor N/S \rfloor$, and we get a graph $C(n,k)$ with $\Theta(N)$ vertices. Thus, we obtain the following tradeoff for this graph given S pebbles:

$$T \geq S\Omega\left(\frac{N}{S}\right)^{\Omega(N/S)}$$

for $S \leq c_2\left(\frac{N \log\log N}{\log N}\right)$ for some constant c_2 where $c_2 < c_1$.

Thus, we notice two main characteristics of our graph. If $S \geq c_1\left(\frac{N \log\log N}{\log N}\right)$, then the time it takes to pebble the graph is $O(N)$ since the width of the graph is $\Theta\left(\frac{N \log\log N}{\log N}\right)$. Second, if $S \leq c_2\left(\frac{N \log\log N}{\log N}\right)$ then S pebbles are used to pebble the graph for $\omega(N)$ time by Theorem 4.2.6 of [LT82]. Note that if the tradeoff is sufficiently great, then we achieve our stated lower bound. To prove our stated lower bound, we modify the proof for Theorem 4.2.5 of [LT82] so that we account for CC^α instead of just the time-space tradeoff. Minimizing the equation for tradeoff in terms of $\alpha = 1$ and showing that the cost is greater than the cost of when $S \geq c_1\left(\frac{N \log\log N}{\log N}\right)$ and the cumulative complexity for when $S \geq c_1\left(\frac{N \log\log N}{\log N}\right)$ is $\Theta\left(\frac{N^2 \log\log N}{\log N}\right)$ then provides us with the lower bound we want.

We use the same notation as that used in the proof of Theorem 4.2.5 in [LT82]. Let n be the number of outputs of the superconcentrator $C(n,k)$ and k be the number of copies of the linear superconcentrators (number of levels in the stack of superconcentrators) in $C(n,k)$. We number the parts of $C(n,k)$ similarly to how they are numbered in the proof of Theorem 4.2.5, let C_i be the i-th copy of the linear superconcentrators that composes $C(n,k)$. We consider the outputs of C_k as numbered in the order in which they are first pebbled. Let z_i be the time that output i (where $1 \leq i \leq n$) is pebbled. Therefore, $z_0 = 0$ and $z_{n+1} = \mathsf{Time}(C(n,k), S)$. Then, let $[z'_i, z''_i]$ be the interval of time starting with the z'_i-th move and ending with the z''_i-th move where $z_{i-1} \leq z'_i \leq z''_i \leq z_i$. Let p_i be the minimum number of pebbles on C_k in the interval $[z_{i-1}, z_i]$ for $1 \leq i \leq n$ and where $p_0 = 0$, $p_{n+1} = 0$, and $p_i \leq S$ for all i in the valid range.

We first note that since we do not remove any vertices or edges (only add edges to the construction to maintain the topological order and to ensure that at most one additional pebble is added to the graph at each time step), all properties of the graph with respect to n as proven in [LT82] still hold (i.e. adding edges does not change the linear superconcentrator properties of the graphs). Hence, we restate some of the key theorems and lemmas in [LT82] that will allow us to prove the lower bound in CC^α when $\alpha = 1$ that we seek.

We restate the definition of a good interval given in [LT82] below:

Definition 10.6.15 (Good Intervals [LT82])**.** *An interval* $[i,j] \subset [1,n]$ *is* good *if it fulfills*

the following three requirements:

$$p_i \leq \frac{j-i}{2}, \tag{10.8}$$

$$p_{j+1} \leq \frac{j-i}{2}, \tag{10.9}$$

$$p_k > \frac{j-i}{8} \text{ for } i < k \leq j. \tag{10.10}$$

We also restate one key lemma relating to good intervals below:

Lemma 10.6.16 (Lemma 4.2.3 [LT82])**.** *During the good interval $[i, j]$ at least $n-2S$ different outputs of C_{k-1} are pebbled. Only $S-1-\left\lfloor \frac{j-i}{8} \right\rfloor$ pebbles are available to pebble the $n-2S$ different outputs of C_{k-1}.*

We also restate a combinatorial lemma proved in [LT82] that will allow us to prove a recursive relation on CC^{α} (which will subsequently allow us to provide a bound for our construction).

Lemma 10.6.17 (Lemma 4.2.4 [LT82])**.** *Let $r \leq n$. We can find a set of disjoint good intervals in $[1, r]$ that covers at least $\frac{r}{4} - S - p_{r+1}$ elements of $[1, r]$.*

Finally, we adapt a theorem based on a simple application of BLBA that provides a (not quite tight enough) lower bound on the time necessary to pebble our constructed graph given S pebbles and provide a proof for our construction defined in Graph Construction 10.6.14.

Theorem 10.6.18 (Theorem 4.2.1 [LT82])**.** *In order to pebble all outputs of $C(n,k)$ as defined in Graph Construction 10.6.14 using S black pebbles, $2 \leq S \leq \frac{n-1}{4}$ (starting with any configuration of pebbles on the graph), we need T placements where*

$$T \geq n\left(\frac{n}{10S}\right)^k.$$

Using these lemmas, we now write our final recursive theorem for the CC^{α} of our construction.

Theorem 10.6.19. *Let $CC^{\alpha}(N,k,S)$ be the CC^{α} (when $\alpha = 1$) necessary to pebble all the outputs of $C(n,k)$ (recall that the topological sort of the vertices requires that for the last output to be pebbled, all other outputs must be pebbled) with $S \leq \frac{n}{20}$ pebbles. Then,*

$$T(n,1,S) \geq \frac{n^2}{10S} \tag{10.11}$$

$$T(n,k,S) \geq \min_{(x_1,\ldots,x_m)\in D_k} \sum_{1\leq i\leq m} T\left(n,k-1,S-1-\left\lfloor\frac{x_i-1}{8}\right\rfloor\right) \text{ for } k>1, \tag{10.12}$$

$$CC^{\alpha}(N,k,S) \geq \min_{D_1,\ldots,D_k} \sum_{1\leq j\leq k} \sum_{(x_1,\ldots,x_m)\in D_j} \left\lfloor\frac{x_i-1}{8}\right\rfloor\left(T\left(n,j-1,S-1-\left\lfloor\frac{x_i-1}{8}\right\rfloor\right)\right) \tag{10.13}$$

$$\geq \min_{D} \sum_{(x_1,\ldots,x_m)\in D} \left\lfloor\frac{x_i-1}{8}\right\rfloor T\left(n,k-1,S-1-\left\lfloor\frac{x_i-1}{8}\right\rfloor\right). \tag{10.14}$$

where D_i is an index set that contains all the ways in which we can select a large number of good intervals. Specifically,

$$D_i = \left\{(x_1,\ldots,x_m) | m > \frac{n}{64S}, 1\leq x_i \leq 8S-6 \text{ for } 1\leq i\leq m, \text{ and } \sum_{1\leq i\leq m} x_i \geq \frac{n}{8}\right\}.$$

Proof. The proof for the expression for $T(n,k,S)$ follows directly from Theorem 4.2.5 in [LT82].

Now we prove the expression for CC^{α} of $C(n,k)$ for the case when $S \leq n/20$. For each good interval, at least $\left\lfloor\frac{x_i-1}{8}\right\rfloor$ pebbles must remain on C_k while $C_1,\ldots,C_{k-1}$ are pebbled with the remaining $S-1-\left\lfloor\frac{i-1}{8}\right\rfloor$ pebbles. Therefore, the CC^{α} when $\alpha=1$ of the good period with length x is $\left\lfloor\frac{x_i-1}{8}\right\rfloor T\left(n,k-1,S-1-\left\lfloor\frac{x_i-1}{8}\right\rfloor\right)$. By Lemma 10.6.17, we have that the total length of the disjoint good intervals is at least $n/8$ (since $p_{r+1}\leq S$ and $n/4-2S\geq n/8$). Thus, summing over the CC^{α} for all good intervals and minimizing over all possible allocations of good intervals gives a lower bound on the CC^{α} for C_k which is a lowerbound on the CC^{α} when $\alpha=1$ of the entire graph. □

Lemma 10.6.20. *When $S = c_1\left(\frac{N\log\log N}{\log N}\right)$ for some constant c_1, $n=20S$, $k=\lfloor N/S\rfloor$ and we create a graph according to Graph Construction 10.6.14, $C(n,k)$ with $\Theta(N)$ vertices,*

$$CC^{\alpha}(N,k,S) \geq \min_{D} \sum_{(x_1,\ldots,x_m)\in D} \left\lfloor\frac{x_i-1}{8}\right\rfloor\left(20S\left(\frac{20S(\lfloor N/S\rfloor-1)}{c\left(S-1-\left\lfloor\frac{x_i-1}{8}\right\rfloor\right)}\right)^{\lfloor N/S\rfloor-1}\right) \tag{10.15}$$

for $S\leq c_2\left(\frac{N\log\log N}{\log N}\right)$ for some constants c (specified in the proof) and $c_2<c_1$.

Proof. We know from [LT82] that the expression for $T(n,k,S)$ is lower bounded by $T(n,k,S)\geq n\left(\frac{nk}{cS}\right)^k$ for some constant $c\geq 10$. Therefore, we can substitute this expression into our Eq. 10.14 to obtain the following expression:

$$\mathrm{CC}^{\alpha}(N,k,S) \geq \min_{D} \sum_{(x_1,\ldots,x_m)\in D} \left\lfloor \frac{x_i-1}{8} \right\rfloor \left(n \left(\frac{n(k-1)}{c(S-1-\left\lfloor \frac{x_i-1}{8} \right\rfloor)} \right)^{k-1} \right).$$

Substituting our values as stated above then gives

$$\mathrm{CC}^{\alpha}(N,k,S) \geq \min_{D} \sum_{(x_1,\ldots,x_m)\in D} \left\lfloor \frac{x_i-1}{8} \right\rfloor \left(20S \left(\frac{20S(\lfloor N/S \rfloor - 1)}{c\left(S-1-\left\lfloor \frac{x_i-1}{8} \right\rfloor\right)} \right)^{\lfloor N/S \rfloor - 1} \right) \tag{10.16}$$

for some number of pebbles used that is less than $n/20$; or in other words, for some constant c_2, $S \leq c_2\left(\frac{N \log\log N}{\log N}\right)$ where we determine the exact values of c_1 and c_2 later on (since the exact values of c_1 and c_2 also depend on the types of linear superconcentrators used in each of the k layers of our construction). □

Lemma 10.6.21. *Given $S \leq c_2\left(\frac{N \log\log N}{\log N}\right)$ for some constant c_2 where $c_2\left(\frac{N \log\log N}{\log N}\right) < n/20$,*

$$CC^{\alpha}(N,k,S) \geq \frac{c_2}{8}\left(\frac{N \log\log N}{\log N}\right)\left(20S \left(\frac{20S\left(\left\lfloor \frac{N}{S} \right\rfloor - 1\right)}{c(S-1)} \right)^{\left\lfloor \frac{N}{S} \right\rfloor - 1} \right). \tag{10.17}$$

Proof. We assume for the sake of contradiction that there exists a closed formed lower-bound for the equation where some $x_i > 1$. Suppose there exists some good period with length $x_i > 1$, then the term

$$\frac{x_i}{8}\left(20S \left(\frac{20S\left(\left\lfloor \frac{N}{S} \right\rfloor - 1\right)}{c(S-1-\left\lfloor \frac{x_i-1}{8} \right\rfloor)} \right)^{\left\lfloor \frac{N}{S} \right\rfloor - 1} \right)$$

is in the summation of the calculation of $\mathrm{CC}^{\alpha}(N,k,S)$ (see Eq. 10.16). We can replace the term with the following:

$$x_i\left(\frac{1}{8}\left(20S \left(\frac{20S\left(\left\lfloor \frac{N}{S} \right\rfloor - 1\right)}{c(S-1)} \right)^{\left\lfloor \frac{N}{S} \right\rfloor - 1} \right)\right)$$

which results in a smaller $\mathrm{CC}^{\alpha}(N,k,S)$ a contradiction, therefore no values of x_i are greater than 1 and the closed form lower bound is that as stated in Eq. 10.17. □

Lemma 10.6.22. *Given $S \leq c_2\left(\frac{N \log\log N}{\log N}\right)$ for some constant c_2 where $c_2\left(\frac{N \log\log N}{\log N}\right) < n/20$, CC^{α} when $\alpha = 1$ is $\omega\left(\frac{N^2 \log\log N}{\log N}\right)$.*[28]

Proof. From Lemma 10.6.21, the CC^{α} when less than $c_2\left(\frac{N \log\log N}{\log N}\right)$ pebbles are used is lower bounded by the closed form expression,

$$CC^{\alpha}(N,k,S) \geq \frac{c_2}{64}\left(\frac{N \log\log N}{\log N}\right)\left(20S\left(\frac{20S\left(\left\lfloor \frac{N}{S} \right\rfloor - 1\right)}{c(S-1)}\right)^{\left\lfloor \frac{N}{S} \right\rfloor - 1}\right). \tag{10.18}$$

We know that the lower bound given in Eq. 10.18 is $\Theta\left(\frac{N \log\log N}{\log N}\left(S\left(\frac{N}{S}\right)^{\frac{N}{S}-1}\right)\right)$.

Given $S \leq \frac{N \log\log N}{\log N}$ pebbles, we now prove that the CC^{α} of our construction for $\alpha = 1$ is $\omega\left(\frac{N^2 \log\log N}{\log N}\right)$. We know that $S\left(\frac{N}{S}\right)^{\frac{N}{S}-1} = \omega(N)$ for all $S \leq c_2\left(\frac{N \log\log N}{\log N}\right)$. Therefore, $CC^{\alpha}(N,k,S) = \omega\left(\frac{N^2 \log\log N}{\log N}\right)$. □

Theorem 10.6.23. *Given $S > c_2\left(\frac{N \log\log N}{\log N}\right)$, CC^{α} when $\alpha = 1$ is $\Omega\left(\frac{N^2 \log\log N}{\log N}\right)$. Therefore, $CC^{\alpha}(G) = \Theta\left(\frac{N^2 \log\log N}{\log N}\right)$ in the sequential pebbling model where G is given by our Graph Construction 10.6.14 above.*

Proof. Let S be large enough that a single linear superconcentrator with n output nodes can be pebbled in almost linear time. In this case, we use the simple BLBA argument presented in Theorem 4.2.1 of [LT82] to prove that in this case, $CC^{\alpha}(N,k,S) = \Omega\left(\frac{N^2 \log\log N}{\log N}\right)$ since each C_i in the construction of $C(n,k)$ as defined in Graph Construction 10.6.14 along with the edges joining C_{k-1} with C_k is an n-superconcentrator.

The BLBA theorem as proven in [LT82] proves a tradeoff in time with respect to the number of pebbles in the starting and ending configuration of the graph. Let S_a be the starting number of pebbles on the graph and S_b be the ending number of pebbles on the graph. Suppose that $S_b = 0$ for the sake of lowerbounding our cumulative complexity. Then $c_2\left(\frac{N \log\log N}{\log N}\right) < \min_{1 \leq i \leq k}\left(S_a^i\right) \leq S$ by our theorem statement where S_a^i is the starting pebble configuration for level i. Suppose that $S_a^{c_i} \leq c_2\left(\frac{N \log\log N}{\log N}\right)$ for L levels (i.e. for some set of levels in $[c_1, \ldots, c_L]$), then $CC^{\alpha}(n,i,S)$ is given by Lemma 10.6.22 for the L values. Using Lemma 10.6.22, we see that in order for the bound from Lemma 10.6.22 to not hold, we must have $L = o(N/S)$. But, then, $N/S - o(N/S) = \Theta(N/S)$ layers are pebbled with $S_a^i > c_2\left(\frac{N \log\log N}{\log N}\right)$ pebbles. Therefore, we achieve the same asymptotic bound by considering $c_2\left(\frac{N \log\log N}{\log N}\right) < \min_{1 \leq i \leq k}\left(S_a^i\right) \leq S$.

[28] We can show for this case that CC^{α} is $\omega\left(\frac{N^2}{\log N}\right)$ in the parallel random oracle case since the runtime in Eq. 10.18 can be improved by at most a factor of $\frac{1}{S}$.

Thus, by BLBA, we know that

$$T(n,1,S) \geq \max\left(1, \frac{n-2S}{2S+1}\right) \tag{10.19}$$

$$T(n,i,S) \geq n\left(\max\left(1, \frac{n}{10S}\right)\right)^i \tag{10.20}$$

$$\mathrm{CC}^\alpha(N,k,S) \geq \sum_{1\leq i\leq k:S_a^i} S_a^i T(n,i-1,S-S_a^i)\max\left(1,\left(\frac{n-2S_a^i}{2S_a^i+1}\right)\right) \tag{10.21}$$

$$\geq n \min_{1\leq i\leq k:S_a^i}\left(S_a^i\right)\max\left(1,\frac{n-2S}{2S+1}\right)(k-1) \tag{10.22}$$

We can simplify in the last step since $T(n,i-1,S-S_a^i) \geq n$ for all $1 \leq i \leq k$. Furthermore, by our argument above, we know that $\min_{1\leq i\leq k:S_a^i}\left(S_a^i\right) = \Theta(S)$.

When $n = c_1\left(\frac{N\log\log N}{\log N}\right)$, $S > c_2\left(\frac{N\log\log N}{\log N}\right)$, and $k = \frac{\log N}{\log\log N}$, then Eq. 10.22 simplifies to $\Omega\left(\frac{N^2\log\log N}{\log N}\right)$ for some predefined c_2 and c_1. Otherwise, the time of pebbling is N using $c_1\left(\frac{N\log\log N}{\log N}\right)$ pebbles resulting in CC^α when $\alpha = 1$ to be $\Theta\left(\frac{N\log\log N}{\log N}\right)$. □

Case of $\alpha = 2$ We briefly note that the above construction does not asymptotically achieve tightness for $\alpha = 2$ by our current analysis. This is due to the fact that when $\alpha = 2$, Lemma 10.6.22 no longer holds due to the fact that $\left(\frac{N\log\log N}{\log N}\right)\cdot\left(\frac{\log N}{\log\log N}\right)^{\frac{\log N}{\log\log N}} = o(N^2)$.

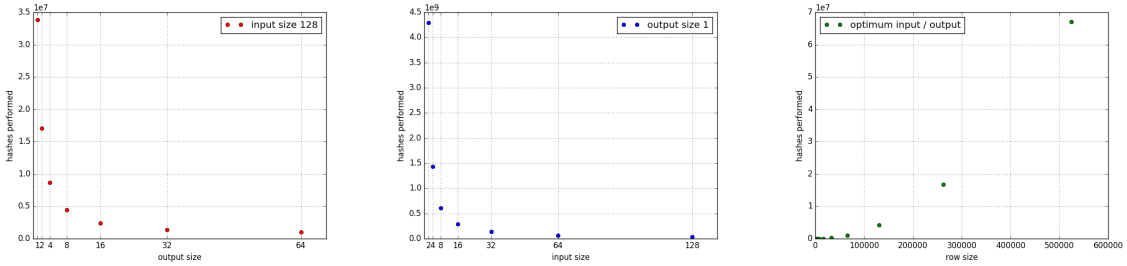
Open Question. *Does there exist a bounded in-degree graph family that has CC^α for $\alpha \geq 2$ that meets the upper bound?*

10.7 Cylinder-based SHF implementation

We implemented a prototype our *cylinder* construction defined in Def. 10.5.4. We choose to implement this construction because it is simplest of the constructions we present for $\mathcal{H}_1$, yet achieves memory and time bounds comparable to our more complicated construction. Our implementation seeks to give a preliminary demonstration of practical feasibility of the cylinder construction for certain parameter ranges; it is not a detailed evaluation of optimized performance.

In implementing the pebbling construction, we seek to minimize the runtime of $\mathcal{H}_1$ while maximizing its output size. This leads to some interesting tradeoffs as well as an observation about static-memory-hardness and the random oracle model in general.

Overview of implementation First, we map an entire row of labels (i.e., labels in a particular layer of our construction) in our cylinder construction defined in Definition 10.5.4 to an array of bits in memory of length l. We implement a serialized pebbling algorithm by iteratively reading n bits, starting at offset f, applying a hash function to the read bits,



(a) Runtime vs. output size, 128 B input, 65 KB row

(b) Runtime vs. input size, 1 B output, 65 KB row

(c) Runtime vs. row size, 64 B output, 128 B input

Figure 10-5: Evaluation of cylinder implementation

writing the n-bit output from the hash function at offset f, and finally incrementing f by additive n bits for the next round. This process is repeated until the end of the string, which constitutes one row of the cylinder construction. This procedure of processing the rows of the cylinder is repeated once for every row of the cylinder DAG.

Parameters Our configurable parameters are: the total size of the array l, the input size i and the output size n of the hash function. Other parameters depend on these as follows:

- The *label size* is n which is also the output length of the hash function. Every hash produces one label. The number of labels per row is then $\frac{l}{n}$. l should be a multiple of n so that there are no partial labels at the end of the array.
- The *indegree* of the wraparound pyramid is $\frac{i}{n}$. Here as well, i should be a multiple of n so that the degree is an integer and this maps cleanly to the pebbling model when we consider ingree $2n$ (ie constant indegree 2 in the pebbling model).
- The *height* needed for the wraparound pyramid is then the array size divided by the difference between input and output sizes, or $\frac{l}{i-n}$. The input size must be greater than the output size for the height to be defined. This corresponds with the requirement that the degree must be at least 2 for the pyramid construction to provide meaningful guarantees.

Instantiating the random oracle We used blake2b, a fast and well-known hash function. Blake2b has an internal state size of 1024 bits, so we were able to set i to 1024 bits while keeping the memory-hardness. n was set to 512 bits, giving an indegree-2 pebbling graph. Decreasing either i or n would lead to inefficient use of the function. It would seem that hash functions with a larger internal state size, capable of supporting a larger i would be faster for this usage, but it is not as clear as larger state sizes may correlate with slower evaluation of a single hash.

We measured the time taken by using a single core of an AMD Ryzen 7 1700 processor. The single threaded code was able to perform approximately 300 million hash operations per second on 1024 bit inputs. This rate could be increased by using multiple cores, but the 300 million hashes per second rate can be used with the above figures to see how many hash operations are being performed at the different settings.

10.7.1 Remarks on implementation and musings on random oracles in practice

Reducing number of hashes The runtime of evaluating $\mathcal{H}_1$ is determined by the number of hash transformations called as very little other computation is done. The number of hashes per row is $\frac{l}{n}$, and the number of rows is $\frac{l}{i-n}$, giving a total number of hash calls as

$$\left(\frac{l}{n}\right)\left(\frac{l}{i-n}\right) = \frac{l^2}{ni-n^2}\,.$$

l^2 indicates the expected time requirement proportional to the square of the output size. To optimize the time for a given l, we look at the denominator, $ni - n^2$, noting that $i > n$, keeping this positive. To reduce the time taken, increase the input size i. Graphically, this makes sense as descending from the top of the wraparound pyramid, the higher degree will quickly cover the entire width of a row. However, in practice we cannot increase i and maintain the memory-hard properties: this is an interesting divergence between the random oracle model and real-world hash functions.

Data busses to the random oracle One aspect which is rarely discussed in the random oracle model is the exact process by which one makes a call to the oracle. Does the query need to be sent to the oracle via a parallel bus, all bits at once, or is the query sent via a serial bus, one bit at a time? If serially, can we send some of the bits, then wait a while, and send the rest? We are not aware of literature dealing with these mechanics of data transmission to and from the oracle; however, in our case it is quite relevant. If serial transmission is allowed, i can be made arbitrarily large without needing to store the whole row of the wraparound pyramid in memory. For each bit of a label, as soon as it is computed it can be sent to all the oracles using that bit as an input, and promptly forgotten; the oracles act as a memory cache. The memory-hardness proofs implicity assume an oracle model where the entire query is handed over simultaneously to the oracle, and as such, any query to the oracle must exist in its entirety in memory before the query is made.

In practice, real-world hash functions resemble a serial-bus oracle much more closely than a parallel bus oracle. Whether we're referring to the Merkle-Damgård construction, the Sponge construction [BDPA11], or other methods, today's widely used hash functions are built out of fixed length one-way functions. The internal state of a hash function can thus act as a data cache for the purposes of the pebbling graph. For a high-degree node, the left predecessors can be fed to the hash function and forgotten before the right predecessors are known. Since the internal state of the hash function has a fixed size, this defeats the memory hardness promised by the pebbling construction.

Data-dependence and cache timing attacks We implement a data-dependent memory access pattern for $\mathcal{H}_2$. Other papers (e.g., Catena [LW15] and Balloon Hash [BCGS16]) have identified security vulnerabilities due to data-dependent memory access patterns which can leak information about the password to an attacker with incomplete access to the physical system evaluating the password hashing function. These attacks occur be-

cause of the variable time taken to evaluate the function based on the input data, primarily due to the automatic caching of data inside the CPU.

We believe that our $\mathcal{H}_2$ function, while implementing a data-dependent memory access pattern, is likely much more resistant to cache timing attacks than the examples mentioned above, based on the following analysis. In practical usage, the output of $\mathcal{H}_1$ will be very large with respect to the number of queries $\mathcal{H}_2$ performs on the data; for any single evaluation of $\mathcal{H}_2$, nearly all of the $\mathcal{H}_1$ data will go unread. Because of this sparse access, *data will be read once and not used again before being evicted from the cache.* The probability that an input to $\mathcal{H}_2$ results in a collision, and multiple reads from the same memory region, is thus modeled by $(\lfloor \Lambda \rfloor)^{-Q}$ where Λ is the size of the output of $\mathcal{H}_1$, and Q is the number of oracle calls made by $\mathcal{H}_2$.

Inputs resulting in cache hits should be rare, and knowledge of a cache hit during $\mathcal{H}_2$ evaluation give a bounded advantage to the attacker expressed by

$$\frac{\text{total number of access patterns with } n \text{ collisions}}{\text{total number of zero-collision access patterns}}.$$

However, we note that this could still be significant advantage in practice because attackers do not need to perform memory lookups into the set of $\mathcal{H}_1$ outputs in order to detect collisions. That is, an attacker still has to perform lots of hashes, but their memory requirement could go down significantly.

On memory allocation In order to implement the wraparound pyramid in the efficient way described above, memory usage needs to be slightly greater than that stated in theoretical model, due to necessary memory allocations in the hardware. Namely, the leftmost bits of the array need to be copied and appended to the right side, so that the lower level input values are available to the final hash interations which consume the wraparound inputs. This increases the memory needed by $i - n$.

Acknowledgements

We are grateful to Jeremiah Blocki, Krzysztof Pietrzak, and Joël Alwen for valuable feedback on earlier versions of this paper. We thank Ling Ren for helpful technical discussions. We also thank Erik D. Demaine and Shafi Goldwasser for their advice on this paper.

Part V

Conclusion

Chapter 11

Summary

This thesis presents a set of algorithms and lower bound results for computing environments in models that represent modern computing environments. Part II first discusses static graph algorithms. Chapter 3 provides approximation algorithms for hard-to-solve graph problems in graphs that are near an algorithmically tractable graph class. Chapter 4 discusses algorithms in the MPC model for small subgraph counting and demonstrates via experiments that our algorithms also empirically provides better approximations than the state-of-the-art on all tested real-world graphs. Chapter 5 shows a near-linear time scheduling algorithm for scheduling with communication delay where precedence constrained jobs are modeled as directed acyclic graphs.

Part III then expands on new efficient and scalable dynamic graph algorithms. Chapter 7 shows a $O(1)$ amortized time, *whp*, dynamic algorithm for $(\Delta + 1)$-vertex coloring. Chapter 6 provides a new parallel, batch-dynamic k-core decomposition algorithm. Chapter 8 provides new parallel, work-efficient batch-dynamic algorithms for triangle and clique counting. Both of these chapters also include extensive experiments showing orders of magnitude improvement in performance on real-world networks.

Finally, Part IV discusses hardness results from pebbling and cryptographic constructions using such hard instances. Chapter 9 presents a set of results showing the hardness of obtaining optimal computation schedules on directed acyclic computation graphs in the external-memory model. Chapter 10 concludes with constructions using such hard instance graphs to construct static-memory-hard hash functions that use disk memory to deter large-scale password-cracking attacks, even against an adversary with unlimited parallel processing power.

Chapter 12

Future Directions

There are a number of open questions that remain as a result of this thesis. We hope that those who are interested in working on the questions central to the theme of this thesis will take the time to explore these and other questions in the future.

Here we summarize the open questions stated at the end of each chapter for convenience.

12.1 Chapter 3: Structural Rounding

We hope that our framework for extending approximation algorithms from structural graph classes to graphs near those classes, by editing to the class and lifting the resulting solution, can be applied to many more contexts. Specific challenges raised by this work include the following:

1. Editing via edge contractions. Approximation algorithms for this type of editing would enable the framework to apply to the many optimization problems closed under just contraction, such as TSP Tour and Connected Vertex Cover.
2. Editing to H-minor-free graphs. Existing results apply only when H is planar [FLMS12]. According to Graph Minor Theory, the natural next steps are when H can be drawn with a single crossing, when H is an apex graph (removal of one vertex leaves a planar graph), and when H is an arbitrary graph (say, a clique). H-minor-free graphs have many PTASs (e.g., [DH05, DHK11]) that would be exciting to extend via structural rounding.
3. Editing to bounded clique number and bounded weak c-coloring number. While we have lower bounds on approximability, we lack good approximation algorithms.

It is also an interesting to see if this framework will lead to faster approximation algorithms in practice, on real-world networks.

12.2 Chapter 4: Massively Parallel Small Subgraph Counting

There are a number of key open questions that result from our work:

1. Can we obtain a better bound on the number of triangles, T, while guaranteeing a $(1+\varepsilon)$-approximation, $O(n^{\delta})$ space per machine (for any constant $\delta > 0$), $\widetilde{O}(n+m)$ total space, and $O(1)$ rounds? The main challenge for us was obtaining the induced subgraph for each set of sampled vertices in a machine; perhaps with a different MPC procedure for doing this, one can obtain a better bound on T.
2. Our exact triangle counting algorithm uses $O(\log \log n)$ rounds. Is it possible to obtain $O(1)$ rounds while using $O(n^{\delta})$ space per machine (for constant $\delta > 0$) and $O(m\alpha)$ total space?
3. The best-known algorithm for computing a $(2+\varepsilon)$-approximate k-core decomposition in $O(n^{\delta})$ space per machine (for constant $\delta > 0$) requires $O(\sqrt{\log n} \cdot \log \log n)$ rounds, *whp*, and total memory $\widetilde{O}\left(\max\{m, n^{1+\delta}\}\right)$ [GLM19]. Is it possible to reduce the number of rounds or the total memory for this problem using ideas from our exact triangle counting algorithm (for graphs with bounded arboricity)?

12.3 Chapter 5: Near-Linear Time Scheduling

Our results so far only apply to scheduling with duplication. In [MRS+20], a polynomial-time reduction is presented that transforms a schedule with duplication into one without duplication (with a polylogarithmic increase in makespan). However, this reduction involves constructing an auxiliary graph of possibly $\Omega(\rho^2)$ size, and thus does not lend itself easily to a near-linear time algorithm. It would be interesting to see if a near-linear time reduction could be found.

12.4 Chapter 8: Parallel Batch-Dynamic k-Clique Counting

There are a number of theoretical and practical questions resulting from our triangle and clique counting results:

1. For bounded arboricity graphs, we give a $O(|\mathcal{B}|(m+|\mathcal{B}|)\alpha^{k-4})$ expected work and $O(\log^2 n)$ depth *whp*, and $O(m+|\mathcal{B}|)$ space algorithm. Can we do better in terms of work and/or depth?
2. Can we apply the techniques used to obtain our batch-dynamic $O(\sqrt{m})$ amortized work, $O(1)$ depth triangle counting algorithm to larger cliques, to obtain better amortized work for e.g., 4-cliques or 5-cliques?
3. We did not implement our matrix multiplication based clique counting algorithm so we do not know how well it performs in real dense networks. It would be interesting to test whether it performs better than the trivial combinatorial algorithm for counting larger cliques dynamically in dense networks such as the neuronal network.

12.5 Chapter 9 Hardness of Red-Blue Pebbling

There are several remaining open questions resulting from our hardness results:

1. Are red pebbling number and minimum number of transitions hard to approximate?
2. Does there exist FPT algorithms for restricted class of graphs (such as bounded width graphs)?
3. Is finding the red pebbling number W[1]-hard? Recall in Section 9.4 that we proved W[1]-hardness only for transitions, not for number of red pebbles.

Part VI

Appendix

Appendix A

Scheduling Appendix

A.1 Count-Distinct Estimator [BYJK+02]

Algorithm 34 [BYJK+02] algorithm for estimating number of distinct elements in a set.

Input A multiset S of elements where $n = |S|$.
Output An estimate on the number of distinct elements in the input multiset.

Let $t \leftarrow \frac{c}{\varepsilon^2}$ for some fixed constant $c \geq 1$.
Let $\mathcal{H}$ be a 2-universal hash family mapping elements from $[n]$ to elements in $[N]$ where $N = n^3$.
Let $h_1, \ldots, h_{c' \log n} \in \mathcal{H}$ be a set of $c' \log n$ hash functions chosen uniformly at random from $\mathcal{H}$ where $c' \geq 1$ is a constant.
Maintain a different binary tree T_i of the smallest t values seen so far of the hash outputs of hash function h_i. Initially each T_i has no elements.
for $a_j \in S$ **do**
    for $h_i \in [h_1, \ldots, h_{c' \log n}]$ **do**
        Compute $h_i(a_j)$.
        if $h_i(a_j)$ is smaller than the largest element in T_i **then**
            Add $h_i(a_j)$ to T_i.
            if size of T_i is greater than t **then**
                Remove the largest element in T_i from T_i.
Let L be a list of hash values. Initially L is empty.
for each T_i **do**
    Let ℓ_i be the largest element in T_i.
    Insert ℓ_i into L.
Sort L.
Return tN/ℓ where ℓ is the median of L.

We provide the algorithm of Bar-Yossef et al. [BYJK+02] in Algorithm 34. The algorithm of [BYJK+02] works as follows. Provided a multiset S of elements where $n = |S|$, we pick $t = \frac{c}{\varepsilon^2}$ where c is some fixed constant $c \geq 1$ and $\mathcal{H}$, a 2-univesal hash family. Then, we choose $O(\log n)$ hash functions from $\mathcal{H}$ uniformly at random, without replacement.

For each hash function $h_i : [n] \to [n^3]$ ($i = O(\log n)$), we maintain a balanced binary tree T_i of the *smallest* t values seen so far from the hash outputs of h_i. Initially, all T_i are empty. We iterate through S and for each $a_j \in S$, we compute $h_i(a_j)$ using each h_i that we picked; we update T_i if $h_i(a_j)$ is smaller than the largest element in T_i or if the size of T_i is smaller than t. After iterating through all of S, for each T_i, we add the largest value of each tree T_i to a list L. Then, we sort L and find the median value ℓ (using the *median trick*). We return tn^3/ℓ as our estimate.

We now show how to use Algorithm 34 to get our desired mergeable estimator. Let $\mathcal{T}_X$ be the set of trees $T_i \in \mathcal{T}_X$ maintained for the estimator defined by Algorithm 34 for multiset X. Since each T_i has size at most $O(t) = O\left(\frac{1}{\varepsilon^2}\right)$, the total space required to store all T_i is $O\left(\frac{1}{\varepsilon^2}\log^2 n\right)$ in bits. We can initialize our estimator on input d by picking a set of random hash functions: $h_1, \ldots, h_{d\log n} \in \mathcal{H}$. Let H be the set of picked hash function. Then, for each set S, we initialize $d\log n$ trees $T_i \in \mathcal{T}_S$ (as used in Algorithm 34) and maintain $\mathcal{T}_S$ in memory. The elements of T_i are computed using $h_i \in H$. Let $\mathcal{D}_S$ denote the estimator for S. Using $\mathcal{T}_S$ for set S, we can implement the following functions (pseudocode for the three functions can be found in Algorithm 35):

- **CountDistinctEstimator.insert($\mathcal{D}_S, x$)**: Insert $h_i(x)$ into $T_i \in \mathcal{T}_S$ for each $i \in [d\log n]$. If T_i has size greater than t, delete the largest element of T_i.
- **CountDistinctEstimator.merge($\mathcal{D}_{S_1}$, $\mathcal{D}_{S_2}$)**: Here we assume that the same set of hash functions are used for both $\mathcal{D}_{S_1}$ and $\mathcal{D}_{S_2}$. For each pair of $T_{1,i} \in \mathcal{T}_{S_1}$ and $T_{2,i} \in \mathcal{T}_{S_2}$ for hash function h_i, build a new tree T_i by taking the t smallest elements from $T_{1,i} \cup T_{2,i}$.
- **CountDistinctEstimator.estimateCardinality($\mathcal{D}_S$)**: Let ℓ be the median value of the largest values of the trees $T_i \in \mathcal{T}_S$. Return tN/ℓ.

The estimator provided in Bar-Yossef et al. [BYJK+02] satisfies the following lemmas as proven in [CG06] (specifically it is proven that the estimator is unbiased):

Lemma A.1.1 ([BYJK+02])**.** *The Bar-Yossef et al. [BYJK+02] estimator is an $\left(\varepsilon, \frac{1}{n^d}, O\left(\frac{1}{\varepsilon^2}\log^2 n\right)\right)$-estimator for the count-distinct problem.*

Lemma A.1.2. *Furthermore, the insert, merge, and estimate cardinality functions of the Bar-Yossef et al. [BYJK+02] estimator can be implemented in $O\left(\frac{1}{\varepsilon^2}\log^2 n\right)$ time.*

Proof. **CountDistinctEstimator.insert** requires $O(\log t)$ time to insert $h_i(x)$ and $O(\log t)$ time to remove the largest element. Thus, this method requires $O\left(\log\left(\frac{1}{\varepsilon}\right)\right) = O\left(\frac{1}{\varepsilon^2}\right)$ time. **CountDistinctEstimator.merge** requires $O(t) = O\left(\frac{1}{\varepsilon^2}\right)$ time to merge $T_{1,i}$ and $T_{2,i}$ and also $O(t)$ time to build the new tree. Finally, **CountDistinctEstimator.estimateCardinality** requires $O(\log n)$ time to create the list L and $O(\log n \log\log n)$ time to sort and find the median. □

Estimating the Number of Ancestors and Edges Using the count-distinct estimator described above, we can provide our full algorithms for estimating the number of ancestors and the number of edges in the induced subgraph of every vertex in a given input graph.

Algorithm 35 Initialize New CountDistinctEstimator.

Input $\mathcal{D}_S, \mathcal{T}_S, t, h_i \in \mathcal{H}$ are as defined above for multiset S. Let $n = |S|$.
Output An estimator.

1: CountDistinctEstimator.**insert**($\mathcal{D}_S, x$):
2: **for** $T_i \in \mathcal{T}_S$ **do**
3: Compute $h_i(x)$.
4: **if** T_i has less than t elements or $h_i(x)$ is smaller than the largest value in T_i **then**
5: Insert $h_i(x)$ into T_i.
6: **if** T_i has more than t elements **then**
7: Remove the largest element in T_i.
8: CountDistinctEstimator.**merge**($\mathcal{D}_{S_1}, \mathcal{D}_{S_2}$):
9: **for** $T_{1,i} \in \mathcal{T}_{S_1}$, $T_{2,i} \in \mathcal{T}_{S_2}$ **do**
10: Perform inorder traversal of $T_{1,i}$ and $T_{2,i}$ to obtain non-decreasing lists of elements, $L_{1,i}$ and $L_{2,i}$.
11: Merge $L_{1,i}$ and $L_{2,i}$ to obtain a new non-decreasing list of elements, L.
12: Build a new balanced binary tree from the first t elements of L.
13: CountDistinctEstimator.**estimateCardinality**($\mathcal{D}_S$):
14: **for** $T_i \in \mathcal{T}_S$ **do**
15: Insert largest element of T_i into list L.
16: Sort L.
17: Let ℓ be the median of L.
18: Return tn^3/ℓ.

Our complete algorithm for estimating the number of ancestors $\hat{a}(v)$ of every vertex in an input graph is given in Algorithm 36.

Algorithm 36 Estimate Number of Ancestors

Input A graph $H = (V, E)$.
Output Estimate $\hat{a}_H(v)$ such that $(1-\varepsilon)|\mathcal{A}_H(v)| \leq \hat{a}_H(v) \leq (1+\varepsilon)|\mathcal{A}_H(v)|$, $\forall v \in V$.

1: Topologically sort all the vertices in H.
2: **for** vertex w in the topological order of vertices in H **do**
3: Let $\mathsf{Pred}(w)$ be the set of predecessors of w.
4: Let $\mathcal{D}_w \leftarrow$ New $\left(\varepsilon, \frac{1}{n^d}, O\left(\frac{1}{\varepsilon^2}\log^2 n\right)\right)$-CountDistinctEstimator for w.
5: CountDistinctEstimator.insert$(\mathcal{D}_w, w)$
6: **for** $v \in \mathsf{Pred}(w)$ **do**
7: $\mathcal{D}_w =$ CountDistinctEstimator.merge$(\mathcal{D}_w, \mathcal{D}_v)$.
8: $\hat{a}_H(w) =$ CountDistinctEstimator.estimateCardinality$(\mathcal{D}_w)$.

Our algorithm for finding $\hat{e}(v)$ for every vertex in the input graph is given in Algorithm 37.

Algorithm 37 Estimate Number of Ancestor Edges

Input A graph $H = (V, E)$.
Output Estimate $\hat{e}_H(v)$ such that $(1-\varepsilon)|\mathcal{E}_H(v)| \leq \hat{e}_H(v) \leq (1+\varepsilon)|\mathcal{E}_H(v)|$, $\forall v \in V$.

1: Topologically sort all the vertices in H.
2: **for** vertex w in the topological order of vertices in H **do**
3: Let $\mathsf{Pred}(w)$ be the set of predecessors of w.
4: Let $\mathcal{D}_w \leftarrow$ New $\left(\varepsilon, \frac{1}{n^d}, O\left(\frac{1}{\varepsilon^2}\log^2 n\right)\right)$-CountDistinctEstimator for w.
5: **for** $v \in \mathsf{Pred}(w)$ **do**
6: CountDistinctEstimator.insert$(\mathcal{D}_w, (v, w))$.
7: $\mathcal{D}_w =$ CountDistinctEstimator.merge$(\mathcal{D}_w, \mathcal{D}_v)$.
8: $\hat{e}_H(w) =$ CountDistinctEstimator.estimateCardinality$(\mathcal{D}_w)$.

A.2 List Scheduling

Here we provide a brief description of the classic Graham list scheduling algorithm [Gra71]. For our purposes, we are given a set of vertices and their ancestors. We duplicate the ancestors for each vertex v so that each vertex and its ancestors is scheduled as a *single* unit with job size equal to the *number of ancestors* of v. Then, we perform the following greedy procedure: for each vertex v, we sequentially assign v to the machine $M_i \in \mathcal{M}$ with *smallest load* (i.e. load is defined by the jobs lengths of all jobs assigned to it). We can maintain loads of the machines in a heap to determine the machine with the lowest load at any time. To schedule n jobs using this procedure requires $O(n \ln M)$ time.

A.3 Scheduling General Graphs Full Algorithm

Algorithm 19 is a shortened version of Algorithm 38 presented here.

Algorithm 38 ScheduleGeneralGraph(G)

Input A directed acyclic task graph $G = (V, E)$.
Output A schedule of the input graph $G = (V, E)$ on M processors.

Let $\mathcal{H} \leftarrow \varnothing$ represent a list of small subgraphs that we will build.
Initialize a data structure to keep track of marked vertices; all vertices are initially unmarked.
while G is not empty **do**
    Let $V_H \leftarrow \varnothing$ be the vertices of the current small subgraph we are building.
    Initialize queue Q with all sources (vertices with no ancestors) of G.
    while Q is not empty **do** Remove first element v of Q. Compute the size estimate for v's ancestor set as $\hat{a}(v)$.
        if $\hat{a}(v) \leq \frac{4}{3}\rho$ **then**
            Mark v and add it to V_H.
            for each successor w of v **do**
                if all predecessors of w are marked **then**
                    Enqueue w into queue Q.
    Compute edge set E_H to be all edges induced by V_H.
    Add $H = (V_H, E_H)$ to $\mathcal{H}$.
    Remove V_H and all incident edges from G.
for $H \in \mathcal{H}$ in the order they were added **do**
    Call ScheduleSmallSubgraph(H) to obtain a schedule of H. [Algorithm 16]

Appendix B

Static-Memory-Hard Hash Functions

B.1 Details of SHF construction with short labels

Algorithm 39 $\mathcal{H}_2^{q'}$

On input $(1^\kappa, x)$ and given oracle access to Seek_R (where R is the string outputted by $\mathcal{H}_1$):

1. Let $[\![R]\!] = |R|/w$ be the length of R in words.
2. Query the random oracle to obtain $\rho_0 = \mathcal{O}(x)$ and $\rho_1 = \mathcal{O}(x+1)$.
3. Use ρ_0 to sample randomly $\iota_1, \ldots, \iota_{q'} \in [[\![R]\!]]$.
4. Query the Seek_R oracle to obtain $\{y'_i = \mathsf{Seek}_R(\iota_i)\}_{i \in [q']}$.
5. Output $(y'_1 \| \ldots \| y'_{q'}) \oplus \rho_1$.

Definition B.1.1 (***q''***-labeling)**.** *Let $G = (V, E)$ be a DAG with maximum in-degree δ, let $\mathfrak{L}$ be an arbitrary "label set," and define $\mathbb{O}(\delta, \mathfrak{L}) = \left(V \times \bigcup_{\delta'=1}^{\delta} \mathfrak{L}^{\delta'} \to \mathfrak{L}\right)$. Let $\mathcal{O}|_{q''}$ be the function that outputs the first q' bits of the output of $\mathcal{O}$. For any function $\mathcal{O} \in \mathbb{O}(\delta, \mathfrak{L})$ and any label $\zeta \in \mathfrak{L}$, the $(\mathcal{O}, \zeta, q'')$-labeling of G is a mapping $\mathsf{label}_{\mathcal{O},\zeta} : V \to \mathfrak{L}$ defined recursively as follows.*[1]

$$\mathsf{label}_{\mathcal{O},\zeta}(v) = \begin{cases} \mathcal{O}|_{q''}(v, \zeta) & \text{if } \mathsf{indeg}(v) = 0 \\ \mathcal{O}|_{q''}(v, \mathsf{label}_{\mathcal{O},\zeta}(\mathsf{Pred}(v))) & \text{if } \mathsf{indeg}(v) > 0 \end{cases}.$$

Then, we define our family of random oracle functions defined from our hard to pebble graph family constructions.

Definition B.1.2 (***q''***-graph function family)**.** *Let $n = n(\kappa)$ and let $\mathbb{G}_\delta = \{G_{n,\delta} = (V_n, E_n)\}_{\kappa \in}$ be a graph family. We write $\mathbb{O}_{\delta,\kappa}$ to denote the set $\mathbb{O}(\delta, \{0,1\}^\kappa)$ as defined in Definition B.1.1. The q''-*graph function family *of $\mathbb{G}$ is the family of oracle functions $\mathcal{F}_{\mathbb{G}}^{q''} = \{\mathfrak{f}_G\}_{\kappa \in}$ where $\mathfrak{f}_G = \{f_G^{\mathcal{O}} : \{0,1\}^\kappa \to (\{0,1\}^\kappa)^z\}_{\mathcal{O} \in \mathbb{O}_{\delta,\kappa}}$ and $z = z(\kappa)$ is the number of*

[1] We abuse notation slightly and also invoke $\mathsf{label}_{\mathcal{O},\zeta}$ on *sets* of vertices, in which case the output is defined to be a tuple containing the labels of all the input vertices, arranged in lexicographic order of vertices.

sink nodes in G. The output of $f_G^{\mathcal{O}}$ on input label $\zeta \in \{0,1\}^\kappa$ is defined to be

$$f_G^{\mathcal{O}}(\zeta) = \mathsf{label}_{\mathcal{O},\zeta}(\mathsf{sink}(G)) \,,$$

where $\mathsf{sink}(G)$ is the set of sink nodes of G.

B.2 Regular and normal pebbling strategies

Here, we restate three theorems and prove briefly their equivalent formulation in the parallel model for the parallel model adapted from theorems in [GLT80, DL17] proven in the sequential model.

We first restate the definitions for normal and regular strategies:

Definition B.2.1 (Frugal Strategy [GLT80])**.** *Given a DAG $G = (V, E)$, a* frugal strategy *is a pebbling strategy with no unnecessary placements. In particular, the following are true of any frugal pebbling strategy:*

1. *At all times after the first placement on a vertex v, some path from v to the goal vertex contains a pebble.*
2. *At all times after the last placement on a vertex v, all paths from v to the goal vertex contain a pebble.*
3. *The number of placements on a nongoal vertex is bounded by the total number of placements on its successors.*

Definition B.2.2 (Normal Strategy [GLT80])**.** *A* normal strategy *is a standard pebbling strategy that is frugal and it pebbles each pyramid P in G as follows: after the first pebble is placed on P, no placement or removal of pebbles occurs outside P until the apex of P is pebbled and all other pebbles are removed from P. No new placement occurs on P until after the pebble on the apex of P is removed.*

Theorem B.2.3 (Normal Strategy Conversion [GLT80])**.** *If the goal vertex is not inside a pyramid, any standard pebbling strategy can be transformed into a normal pebbling strategy without increasing the number of pebbles used in both the sequential and parallel pebbling models.*

Proof. The proof of this statement in the sequential model is given in [GLT80]. We now prove this statement in the parallel model. By our proof of Lemma 10.5.1, any sequential strategy can be simulated trivially by a parallel strategy; therefore, if any pebbling strategy can be transformed into a sequential normal pebbling strategy, then any pebbling strategy can be transformed into a parallel normal pebbling strategy. □

We now define regular pebbling strategies:

Definition B.2.4 (Regular Strategy [DL17])**.** *Given a DAG $G = (V, E)$, a* regular strategy *is a standard pebbling strategy that is frugal and after the first pebble is placed on any road graph $R_w \in G$, no placements of pebbles occurs outisde R_w until the set of desired outputs of R_w all contain pebbles and all other pebbles are removed from R_w.*

By the same argument as given for the proof of Theorem B.2.3, we can prove the equivalent for parallel regular pebbling strategies.

Theorem B.2.5 (Regular Strategy Conversion [DL17])**.** *Given a DAG $G = (V, E)$, if each input, $i_j \in \{i_1, \ldots, i_w\}$, to a road graph has at most 1 predecessor, any standard pebbling strategy that pebbles a set of desired outputs, $O \subseteq \{o_1, \ldots, o_w\}$, at the same tiime can be transformed into a* regular strategy *without increasing the number of pebbles used.*

In addition, we prove this stronger theorem about the pebbling space complexity of pyramid graphs below than the theorems provided in [GLT80, Nor15] that will be useful for determining the pebbling space complexity of pyramids in the magic pebble game.

Theorem B.2.6. *Given a pyramid graph Π_h with h levels where level 1 has h nodes and level h has 1 node. Given S pebbles and if all S pebbles are placed on level i of the pyramid and $S < h + 1 - i$, then the apex of the pyramid cannot be pebbled using the rules of the standard pebble game.*

Proof. Given $S < h + 1 - i$ pebbles on the i-th layer of a height h pyramid, we know that the i-th level of the pyramid forms a height $h + 1 - i$ height pyramid with the apex. Thus, by the pebbling space complexity of pyramids, $h + 1 - i$ pebbles are necessary on level $h + 1 - i$ in order to pebble the apex. □

Appendix C

Parallel Batch-Dynamic k-Clique Counting Appendix

C.1 Sequential Fully Dynamic Triangle Counting Algorithm of [KNN+19]

Here, we present the sequential fully dynamic triangle counting algorithm of Kara et al. [KNN+19] that operates in $O(m)$ space, $O(\sqrt{m})$ amortized work per edge update, and $O(m^{3/2})$ work for preprocessing. This algorithm returns the exact count of the number of triangles in an undirected graph under both edge insertions and deletions. Kara et al. [KNN+19] present their algorithm for directed 3-cycles using relational database terminology (where each edge in the triangle may be drawn from a different relation), but we simplify their algorithm for the case of undirected graphs. Kara et al. [KNN+19] prove the following theorem.

Theorem C.1.1 (Fully Dynamic Triangle Counting [KNN+19])**.** *There exists a sequential algorithm to count the number of triangles in an undirected graph $G = (V, E)$ using $O(m^{3/2})$ preprocessing work that can handle an edge update in $O(\sqrt{m})$ amortized work and $O(m)$ space.*

We now explain the fully dynamic triangle counting algorithm of [KNN+19] in greater detail.

Given a graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, we initialize the following variables: $M = 2m + 1$, $t_1 = \sqrt{M}/2$, and $t_2 = 3\sqrt{M}/2$. We define a vertex to be ***low-degree*** if its degree is at most t_1 and ***high-degree*** if its degree is at least t_2. Vertices with degree in between t_1 and t_2 can be classified either way. Let C be the current count of the number of triangles in the graph. We compute the initial count of the number of triangles in the input graph G using a static triangle counting algorithm [IR77] in $O(m^{3/2})$ work and $O(m)$ space. Thus, we immediately have a preprocessing work of $O(m^{3/2})$.

We create four data structures $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$. $\mathcal{HH}$ stores all of the edges (u, v) where both u and v are high-degree, $\mathcal{HL}$ stores edges (u, v), where u is high-degree and v is low-degree, $\mathcal{LH}$ stores the edges (u, v) where u is low-degree and v is high-degree,

and $\mathcal{LL}$ stores edges where both u and v are low-degree. With our data structures, the following operations are supported:

1. Given a vertex v, determine whether it is low-degree or high-degree in $O(1)$ work.
2. Given an edge (u, v), check if it is in $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, or $\mathcal{LL}$ in $O(1)$ work.
3. Given a vertex v, return all neighbors of v in $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ in $O(\deg(v))$ work.
4. Given an edge (v, w) to insert or delete, update $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, or $\mathcal{LL}$ in $O(1)$ work.

We can implement $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ to support these operations by using a two-level hash table for each of these structures and an additional array $\mathcal{D}$. $\mathcal{D}$ is a dynamic hash table containing a key for each vertex that has non-zero degree and stores the degree of the vertex as the value. The data structures support insertions and deletions in $O(1)$ work. $\mathcal{D}$ can be initialized in $O(m)$ work by scanning over all vertices and computing their degree. $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ can be initialized in $O(m)$ work by scanning over all edges and inserting them into the right table based on the degrees of their endpoints.

We maintain one additional data structure $\mathcal{T}$ that counts the number of wedges (u, w, v), where u and v are high-degree vertices and w is a low-degree vertex. $\mathcal{T}$ has the property that given an edge insertion or deletion (u, v) where both u and v are high-degree vertices, it returns the number of such wedges (u, w, v) where w is low-degree that u and v are part of in $O(1)$ work. We can implement this via a hash table indexed by pairs of high-degree vertices that stores the number of wedges for each pair. $\mathcal{T}$ can be initialized in $O(m^{3/2})$ work by iterating over all edges (u, w) in $\mathcal{HL}$ and then for each w, iterating over all edges (w, v) in $\mathcal{LH}$ to determine whether v is high-degree, and if so then increment $T(u, v)$ by 1. There are $O(m)$ edges (u, w) in $\mathcal{HL}$, and for each w there are at most $O(\sqrt{m})$ edges (w, v) in $\mathcal{LH}$ since w is low-degree. Each lookup and increment takes $O(1)$ work, giving an overall work of $O(m^{3/2})$.

C.1.1 Update Procedure [KNN+19]

The procedure for handling single edge updates in the sequential setting given by [KNN+19] as follows:

For an edge insertion (resp. deletion) (u, v), we first find the degree of u and v in $\mathcal{D}$ and then look up the edge in their respective tables $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, or $\mathcal{LL}$. If the edge already exists (resp. does not exist) in the table, nothing else is done. Otherwise, we need to find all tuples (u, w, v) such that (v, u) and (u, w) already exist in the graph because for each such tuple, a new triangle will be formed (resp. an existing triangle will be deleted). We first update the triangle count, and then we update the data structures. For updating the triangle count C, there are 4 different cases for such tuples, and so we check each of the following cases:

1. **(u, w) is in $\mathcal{HH}$ and (w, v) is in $\mathcal{H}y$ where $y \in \{\mathcal{H}, \mathcal{L}\}$**: We extract all high-degree neighbors of u in $\mathcal{HH}$. Given that the degree of all high-degree vertices is $\Omega(\sqrt{m})$, there are at most $O(\sqrt{m})$ such vertices. For each of these neighbors, we can check in $O(1)$ work for each w whether (w, v) exists in $\mathcal{H}y$. This takes $O(\sqrt{m})$ work.
2. **(u, w) is in $\mathcal{HL}$ and (w, v) is in $\mathcal{LH}$ where $y \in \{\mathcal{H}, \mathcal{L}\}$**: Since both u and v are high-degree in this case, we perform an $O(1)$ work lookup in $\mathcal{T}$ for the count of the number of wedges (u, w, v) in this case.

3. **(u, w) is in $\mathcal{LH}$ and (w, v) is in $\mathcal{H}y$ where $y \in \{\mathcal{H}, \mathcal{L}\}$**: Scan through the neighbors of u in $\mathcal{LH}$. For each neighbors of u, check whether (w, v) exists in $\mathcal{H}y$. This takes $O(\sqrt{m})$ work since u has low-degree.
4. **(u, w) is in $\mathcal{LL}$ and (w, v) is in $\mathcal{L}y$ where $y \in \{\mathcal{L}, \mathcal{H}\}$**: Again, scan through the neighbors of u in $\mathcal{LH}$. For each neighbors of u, check whether (w, v) exists in $\mathcal{L}y$. This takes $O(\sqrt{m})$ work since u has low-degree.

After updating the triangle count, we proceed with updating the data structures with the edge insertion (resp. deletion).

We first update $\mathcal{T}$ given an edge insertion (resp. deletion) (u, v) as follows:

1. If u is high-degree and v is low-degree, then we find all of v's neighbors in $\mathcal{LH}$ and for each such neighbor x, we increment (resp. decrement) the entry $\mathcal{T}(u, x)$ by 1. It takes $O(\sqrt{m})$ work to perform this update since v is low-degree.
2. If u is low-degree and v is high-degree, then we scan through all vertices in $\mathcal{HL}$ and for each vertex x in $\mathcal{HL}$ that has u as a neighbor, we increment (resp. decrement) $\mathcal{T}(x, v)$ by 1. This takes $O(\sqrt{m})$ work since there are at most $O(\sqrt{m})$ high-degree vertices.

In addition to the updates to $\mathcal{T}$, we also insert (resp. delete) (u, v) into $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ depending on the degrees of u and v, and update $\mathcal{D}$. For a given edge (u, v) insertion (resp. deletion), we first determine whether u and v are low-degree or high-degree by looking in $\mathcal{D}$ for u and v in $O(1)$ work. $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ are constructed as hash tables keyed by first the first vertex in the edge tuple and then the second vertex in the edge tuple with pointers to second-level hash tables storing the neighbors of that particular vertex. If u is high-degree, then the edge is inserted (resp. deleted) into $\mathcal{HH}$ or $\mathcal{HL}$ (depending on whether v is low or high-degree) using u as the key and adding v to the second level hash table. Similarly, if u is low-degree, (u, v) is inserted (resp. deleted) into $\mathcal{LH}$ or $\mathcal{LL}$. Furthermore, (v, u) is also inserted into its respective table depending on whether v is low or high-degree. The entries for u and v in $\mathcal{D}$ are then incremented (resp. decremented) in $\mathcal{D}$. The updates to these data structures take $O(1)$ work.

We also have to deal with the cases where the degree classification of vertices have changed or the number of edges has changed by too much that the values of M, t_1, and t_2 need to be updated. This is described in the next section.

C.1.2 Rebalancing [KNN+19]

We now describe the rebalancing procedure given in [KNN+19] when a low-degree vertex becomes a high-degree vertex (or vice versa) and when too many updates have been applied (and all the data structures must be changed according to the new values of M, t_1, and t_2).

Minor rebalancing This type of rebalancing occurs if a vertex which was previously high-degree has its degree fall below t_1 or if a vertex that was previously low-degree has its degree increase above t_2. In the first case, we move the vertex and all its edges from $\mathcal{HH}$ to $\mathcal{HL}$, and from $\mathcal{LH}$ to $\mathcal{LL}$. In the second case, we move the vertex and all its edges from $\mathcal{HL}$ to $\mathcal{HH}$, and from $\mathcal{LL}$ to $\mathcal{LH}$. Since our data structures support additions and deletions of an edge in $O(1)$ work, and since the degree of v is $\Theta(\sqrt{m})$ at this point,

we perform $\Theta(\sqrt{m})$ updates. We showed in Appendix C.1.1 that updates take $O(\sqrt{m})$ work so we take $O(m)$ work overall for a minor rebalancing. However, $\Omega(\sqrt{m})$ updates must have occurred on this vertex before we have to perform minor rebalancing since $t_2 - t_1 = \Theta(\sqrt{m})$, and so we can amortize this cost over the $\Omega(\sqrt{m})$ updates, resulting in $O(\sqrt{m})$ amortized work per update.

Major rebalancing A major rebalancing occurs when m, the number of edges in the graph, falls outside the range $[M/4, M]$. We simply reinitialize the data structures as in the original algorithm. Major rebalancing can only occur after $\Omega(M)$ updates, and so we can afford to re-initialize our data structure and recompute the triangle count from scratch using an $O(m^{3/2})$ work triangle counting algorithm. The amortized work of major rebalancing over $\Omega(m)$ updates is then $O(\sqrt{m})$.

Bibliography

[AABD19] Umut A. Acar, Daniel Anderson, Guy E. Blelloch, and Laxman Dhulipala. Parallel batch-dynamic graph connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 381–392, 2019.

[AAC+17] Hamza Abusalah, Joël Alwen, Bram Cohen, Danylo Khilko, Krzysztof Pietrzak, and Leonid Reyzin. Beyond hellman's time-memory trade-offs with applications to proofs of space. *ASIACRYPT (2)*, 10625:357–379, 2017.

[AAW17] Umut A. Acar, Vitaly Aksenov, and Sam Westrick. Brief announcement: Parallel dynamic tree contraction via self-adjusting computation. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 275–277, 2017.

[AB16] Joël Alwen and Jeremiah Blocki. Efficiently computing data-independent memory-hard functions. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, pages 241–271, 2016.

[ABB+19] Sepehr Assadi, MohammadHossein Bateni, Aaron Bernstein, Vahab Mirrokni, and Cliff Stein. Coresets meet EDCS: algorithms for matching and vertex cover on massive graphs. In *Proceedings 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2019.

[ABG+18] Maryam Aliakbarpour, Amartya Shankha Biswas, Themis Gouleakis, John Peebles, Ronitt Rubinfeld, and Anak Yodpinyanee. Sublinear-time algorithms for counting star subgraphs via edge sampling. *Algorithmica*, 80(2):668–697, 2018.

[ABH17] Joël Alwen, Jeremiah Blocki, and Ben Harsha. Practical graphs for optimal side-channel resistant memory-hard functions. In *CCS*, pages 1001–1017. ACM, 2017.

[ABMV16] Sabeur Aridhi, Martin Brugnara, Alberto Montresor, and Yannis Velegrakis. Distributed k-core decomposition and maintenance in large dynamic graphs. In *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems*, pages 161–168, 2016.

[ABP17a] Joël Alwen, Jeremiah Blocki, and Krzysztof Pietrzak. Depth-robust graphs and their cumulative memory complexity. In *EUROCRYPT (3)*, volume 10212 of *Lecture Notes in Computer Science*, pages 3–32, 2017.

[ABP17b] Joël Alwen, Jeremiah Blocki, and Krzysztof Pietrzak. Sustained space complexity. *CoRR*, abs/1705.05313, 2017.

[ACK+16] Joël Alwen, Binyi Chen, Chethan Kamath, Vladimir Kolmogorov, Krzysztof Pietrzak, and Stefano Tessaro. On the complexity of scrypt and proofs of space in the parallel random oracle model. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, pages 358–387, 2016.

[ACP+16] Joël Alwen, Binyi Chen, Krzysztof Pietrzak, Leonid Reyzin, and Stefano Tessaro. Scrypt is maximally memory-hard. *IACR Cryptology ePrint Archive*, 2016:989, 2016.

[ADN+10] Joël Alwen, Yevgeniy Dodis, Moni Naor, Gil Segev, Shabsi Walfish, and Daniel Wichs. Public-key encryption in the bounded-retrieval model. In *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, pages 113–134, 2010.

[AdRNV17] Joël Alwen, Susanna F. de Rezende, Jakob Nordström, and Marc Vinyals. Cumulative space in black-white pebbling and resolution. In *ITCS*, volume 67 of *LIPIcs*, pages 38:1–38:21. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[ADW09] Joël Alwen, Yevgeniy Dodis, and Daniel Wichs. Survey: Leakage resilience and the bounded retrieval model. In *Information Theoretic Security, 4th International Conference, ICITS 2009, Shizuoka, Japan, December 3-6, 2009. Revised Selected Papers*, pages 1–18, 2009.

[AFU13] Foto N Afrati, Dimitris Fotakis, and Jeffrey D Ullman. Enumerating subgraph instances using map-reduce. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 62–73. IEEE, 2013.

[AG09] Noga Alon and Shai Gutner. Linear time algorithms for finding a dominating set of fixed size in degenerated graphs. *Algorithmica*, 54(4):544–556, 2009.

[AG15] Kook Jin Ahn and Sudipto Guha. Access to data and number of iterations: Dual primal algorithms for maximum matching under resource constraints. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 202–211, 2015.

[AHDBV05] J. Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In *Proceedings of the 18th International Conference on Neural Information Processing Systems*, 2005.

[AK98] Ishfaq Ahmad and Yu-Kwong Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 9(9):872–892, Sep. 1998.

[AKK19] Sepehr Assadi, Michael Kapralov, and Sanjeev Khanna. A simple sublinear-time algorithm for counting arbitrary subgraphs via edge sampling. In *ITCS*, volume 124 of *LIPIcs*, pages 6:1–6:20. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.

[Akm08] Fusun Akman. Partial chromatic polynomials and diagonally distinct sudoku squares. *arXiv preprint arXiv:0804.0284*, 2008.

[AKM13] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. PATRIC: A parallel algorithm for counting triangles in massive networks. In *ACM Conference on Information and Knowledge Management (CIKM)*, pages 529–538, 2013.

[Alm19] Josh Alman. *Linear Algebraic Techniques in Algorithms and Complexity*. PhD thesis, Massachusetts Institute of Technology, 2019.

[ALS20] Shiri Antaki, Quanquan C. Liu, and Shay Solomon. Near-optimal distributed implementations of dynamic algorithms for symmetry-breaking problems. *CoRR*, abs/2010.16177, 2020.

[ALT+17] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. EmptyHeaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 42(4):20:1–20:44, 2017.

[AM84] Richard Anderson and Ernst W. Mayr. A P-complete problem and approximations to it. Technical report, Stanford University, 1984.

[Ami10] Eyal Amir. Approximation algorithms for treewidth. *Algorithmica*, 56(4):448–479, 2010.

[AMS96] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 20–29, 1996.

[AMSJ18] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proc. VLDB Endow.*, 11(6):691–704, February 2018.

[AN01] Jochen Alber and Rolf Niedermeier. Improved tree decomposition based algorithms for domination-like problems. In *Proceedings of the 5th Latin American Symposium on Theoretical Informatics*, pages 613–627. Springer, 2001.

[ANOY14] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. In *STOC*, pages 574–583, 2014.

[ANR+17] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, Nick G. Duffield, and Theodore L. Willke. Graphlet decomposition: framework, algorithms, and applications. *Knowl. Inf. Syst.*, 50(3):689–722, 2017.

[AOSS19] Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. Fully dynamic maximal independent set with sublinear in n update time. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 1919–1936, 2019.

[AP89] Stefan Arnborg and Andrzej Proskurowski. Linear time algorithms for np-hard problems restricted to partial k-trees. *Discrete Applied Mathematics*, 23(1):11–24, 1989.

[AS15] Joël Alwen and Vladimir Serbinenko. High parallel complexity graphs and memory-hard functions. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 595–603, 2015.

[ASM+06] Altaf Amin, Yoko Shinbo, Kenji Mihara, Ken Kurokawa, and Shigehiko Kanaya. Development and implementation of an algorithm for detection of protein complexes in large interaction networks. *BMC bioinformatics*, 7:207, 02 2006.

[Ass17] Sepehr Assadi. Simple round compression for parallel vertex cover. *arXiv preprint arXiv:1709.04599*, 2017.

[ASS+18] A. Andoni, Z. Song, C. Stein, Z. Wang, and P. Zhong. Parallel graph connectivity in log diameter rounds. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 674–685, Oct 2018.

[ASW18] Sepehr Assadi, Xiaorui Sun, and Omri Weinstein. Massively parallel algorithms for finding well-connected components in sparse graphs. *arXiv preprint arXiv:1805.02974*, 2018.

[ASW19] Sepehr Assadi, Xiaorui Sun, and Omri Weinstein. Massively parallel algorithms for finding well-connected components in sparse graphs. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 461–470. ACM, 2019.

[ASZ19] Alexandr Andoni, Clifford Stein, and Peilin Zhong. Log diameter rounds algorithms for 2-vertex and 2-edge connectivity. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 14:1–14:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[AT17] Joël Alwen and Björn Tackmann. Moderately hard functions: Definition, instantiations, and applications. In *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part I*, pages 493–526, 2017.

[AV88] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, September 1988.

[AW21] Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In *SODA*, pages 522–539. SIAM, 2021.

[AWS] Amazon web services. https://aws.amazon.com/. Accessed: 2021-07-01.

[AYZ97] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, Mar 1997.

[AZ17] Esra Akbas and Peixiang Zhao. Truss-based community search: A truss-equivalence based indexing approach. *Proc. VLDB Endow.*, 10(11):1298–1309, August 2017.

[BAD20] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. Brief announcement: ParlayLib – a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020.

[BBD+19] Soheil Behnezhad, Sebastian Brandt, Mahsa Derakhshan, Manuela Fischer, MohammadTaghi Hajiaghayi, Richard M. Karp, and Jara Uitto. Massively parallel computation of matching and MIS in sparse graphs. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 481–490. ACM, 2019.

[BC17] Suman K Bera and Amit Chakrabarti. Towards tighter space bounds for counting triangles and other substructures in graph streams. In *34th Symposium on Theoretical Aspects of Computer Science*, 2017.

[BCD+20] Michael A. Bender, Rezaul Alam Chowdhury, Rathish Das, Rob Johnson, William Kuszmaul, Andrea Lincoln, Quanquan C. Liu, Jayson Lynch, and Helen Xu. Closing the gap between cache-oblivious and cache-adaptive

analysis. In Christian Scheideler and Michael Spear, editors, *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, pages 63–73. ACM, 2020.

[BCDM17] Therese Biedl, Markus Chimani, Martin Derka, and Petra Mutzel. Crossing number for graphs with bounded pathwidth. In *28th International Symposium on Algorithms and Computation, (ISAAC)*, pages 13:1–13:13, Phuket, Thailand, December 2017.

[BCG20] Suman K. Bera, Amit Chakrabarti, and Prantar Ghosh. Graph coloring via degeneracy in streaming and other space-conscious models. In *ICALP*, volume 168 of *LIPIcs*, pages 11:1–11:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[BCGS16] Dan Boneh, Henry Corrigan-Gibbs, and Stuart Schechter. *Balloon Hashing: A Memory-Hard Function Providing Provable Protection Against Sequential Attacks*, pages 220–248. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[BCH17] Sayan Bhattacharya, Deeparnab Chakrabarty, and Monika Henzinger. Deterministic fully dynamic approximate vertex cover and fractional matching in O(1) amortized update time. In *IPCO*, volume 10328 of *Lecture Notes in Computer Science*, pages 86–98. Springer, 2017.

[BCH19] Matthias Bonne and Keren Censor-Hillel. Distributed detection of cliques in dynamic networks. In *International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, pages 132:1–132:15, 2019.

[BCHN18] Sayan Bhattacharya, Deeparnab Chakrabarty, Monika Henzinger, and Danupon Nanongkai. Dynamic algorithms for graph coloring. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1–20. SIAM, 2018.

[BDE+16] Michael A. Bender, Erik D. Demaine, Roozbeh Ebrahimi, Jeremy T. Fineman, Rob Johnson, Andrea Lincoln, Jayson Lynch, and Samuel McCauley. Cache-adaptive analysis. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 135–144, New York, NY, USA, 2016. ACM.

[BDE+19] Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, and Vahab S. Mirrokni. Near-optimal massively parallel graph connectivity. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 1615–1636, 2019.

[BDH18] Soheil Behnezhad, Mahsa Derakhshan, and MohammadTaghi Hajiaghayi. Semi-mapreduce meets congested clique. *arXiv preprint arXiv:1802.10297*, 2018.

[BDH+19] Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Marina Knittel, and Hamed Saleh. Streaming and massively parallel algorithms for edge coloring. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*, volume 144 of *LIPIcs*, pages 15:1–15:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[BDPA08] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, pages 181–197, 2008.

[BDPA11] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Cryptographic sponge functions. In *CRYPTO 2009*, pages 1–93, 2011.

[BE10] Leonid Barenboim and Michael Elkin. Sublogarithmic distributed mis algorithm for sparse graphs using nash-williams decomposition. *Distributed Comput.*, 22(5-6):363–379, 2010.

[BE11] Leonid Barenboim and Michael Elkin. Deterministic distributed vertex coloring in polylogarithmic time. *J. ACM*, 58(5):23:1–23:25, 2011. Announced at PODC 2010.

[BEF+14] Michael A. Bender, Roozbeh Ebrahimi, Jeremy T. Fineman, Golnaz Ghasemiesfeh, Rob Johnson, and Samuel McCauley. Cache-adaptive algorithms. In *Proceedings of the Twenty-fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, pages 958–971, Philadelphia, PA, USA, 2014. Society for Industrial and Applied Mathematics.

[BEG+18] Mahdi Boroujeni, Soheil Ehsani, Mohammad Ghodsi, MohammadTaghi HajiAghayi, and Saeed Seddighin. Approximating edit distance in truly subquadratic time: quantum and MapReduce. In *SODA*, pages 1170–1189, 2018.

[BEL+20] Amartya Shankha Biswas, Talya Eden, Quanquan C. Liu, Slobodan Mitrovic, and Ronitt Rubinfeld. Parallel algorithms for small subgraph counting. *CoRR*, abs/2002.08299, 2020.

[Ben89] Charles H. Bennett. Time/space trade-offs for reversible computation. *SIAM J. Comput.*, 18(4):766–776, 1989.

[BFS16] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[BFU18] Sebastian Brandt, Manuela Fischer, and Jara Uitto. Breaking the linear-memory barrier in MPC: Fast MIS on trees with n^{ε} memory per machine. *arXiv preprint arXiv:1802.06748*, 2018.

[BGHK95] Hans L. Bodlaender, John R. Gilbert, Hjálmtyr Hafsteinsson, and Ton Kloks. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *Journal of Algorithms*, 18(2):238–255, 1995.

[BGK96] Evripidis Bampis, Aristotelis Giannakos, and Jean-Claude König. On the complexity of scheduling with large communication delays. *European Journal of Operational Research*, 94:252–260, 1996.

[BGK+19] Sayan Bhattacharya, Fabrizio Grandoni, Janardhan Kulkarni, Quanquan C. Liu, and Shay Solomon. Fully dynamic (Δ+1)-coloring in constant update time. *CoRR*, abs/1910.02063, 2019. In submission to Transactions on Algorithms (TALG).

[BGKV14] Francesco Bonchi, Francesco Gullo, Andreas Kaltenbrunner, and Yana Volkovich. Core decomposition of uncertain graphs. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1316–1325, 2014.

[BGS15] Surender Baswana, Manoj Gupta, and Sandeep Sen. Fully dynamic maximal matching in $O(\log n)$ update time. *SIAM J. Comput.*, 44(1):88–113, 2015.

[BHH19] Soheil Behnezhad, MohammadTaghi Hajiaghayi, and David G. Harris. Exponentially faster massively parallel maximal matching. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 1637–1649. IEEE Computer Society, 2019.

[BHI15] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Deterministic fully dynamic data structures for vertex cover and matching. In *SODA*, pages 785–804, 2015.

[BHKK09] Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Counting paths and packings in halves. *Algorithms - ESA 2009*, page 578–586, 2009.

[BHNT15] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos Tsourakakis. Space- and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *ACM Symposium on Theory of Computing (STOC)*, pages 173–182, 2015.

[BK15] Alex Biryukov and Dmitry Khovratovich. Tradeoff cryptanalysis of memory-hard functions. In *ASIACRYPT (2)*, volume 9453 of *Lecture Notes in Computer Science*, pages 633–657. Springer, 2015.

[BK19] Sayan Bhattacharya and Janardhan Kulkarni. Deterministically maintaining a $(2+\varepsilon)$-approximate minimum vertex cover in $o(1/\varepsilon^2)$ amortized update time. In *SODA*, 2019.

[BKL+15] Kshipra Bhawalkar, Jon Kleinberg, Kevin Lewi, Tim Roughgarden, and Aneesh Sharma. Preventing unraveling in social networks: The anchored k-core problem. *SIAM Journal on Discrete Mathematics*, 29(3):1452–1475, 2015.

[BKM19] Philipp Bamberger, Fabian Kuhn, and Yannic Maus. Local distributed algorithms in highly dynamic networks. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 33–42, 2019.

[BKR16] Mihir Bellare, Daniel Kane, and Phillip Rogaway. Big-key symmetric encryption: Resisting key exfiltration. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I*, pages 373–402, 2016.

[BKS13] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In *Proceedings of the 32Nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS)*, pages 273–284, 2013.

[BKS14] Paul Beame, Paraschos Koutris, and Dan Suciu. Skew in parallel query processing. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 212–223, 2014.

[BL17] Glencora Borradaile and Hung Le. Optimal Dynamic Program for r-Domination Problems over Tree Decompositions. In *Proceedings of the 11th International Symposium on Parameterized and Exact Computation*, volume 63 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:23, 2017.

[Blä13] Markus Bläser. Fast matrix multiplication. *Theory of Computing, Graduate Surveys*, 5:1–60, 2013.

[BOC08] Doruk Bozdag, Fusun Ozguner, and Umit V Catalyurek. Compaction of schedules and a two-stage approach for duplication-based DAG scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 20(6):857–871, 2008.

[Bod88] Hans L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In Timo Lepistö and Arto Salomaa, editors, *Automata, Languages and Programming*, pages 105–118, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[Bón06] Miklós Bóna. *A walk through combinatorics: an introduction to enumeration and graph theory*. World Scientific, 2006.

[BOV13] Vladimir Braverman, Rafail Ostrovsky, and Dan Vilenchik. How hard is counting triangles in the streaming model? In *International Colloquium on Automata, Languages, and Programming*, pages 244–254. Springer, 2013.

[BPS20] Suman K. Bera, Noujan Pashanasangi, and C. Seshadhri. Linear Time Subgraph Counting, Graph Degeneracy, and the Chasm at Size Six. In Thomas Vidick, editor, *11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*, volume 151 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:20, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993.*, pages 62–73, 1993.

[Bru10] Peter Brucker. *Scheduling Algorithms*. Springer Publishing Company, Incorporated, 5th edition, 2010.

[BRU17] Nikhil Bansal, Daniel Reichman, and Seeun William Umboh. LP-based robust algorithms for noisy minor-free and bounded treewidth graphs. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1964–1979. Society for Industrial and Applied Mathematics, 2017.

[BRZ18] Jeremiah Blocki, Ling Ren, and Samson Zhou. Bandwidth-hard functions: Reductions and lower bounds. *IACR Cryptology ePrint Archive*, 2018:221, 2018.

[BU17] Nikhil Bansal and Seeun William Umboh. Tight approximation bounds for dominating set on graphs of bounded arboricity. *Information Processing Letters*, 122:21–24, 2017.

[BW09] Reinhard Bauer and Dorothea Wagner. Batch dynamic single-source shortest-path algorithms: An experimental study. In Jan Vahrenhold, editor, *Experimental Algorithms*, pages 51–62, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[BYBFR04] Reuven Bar-Yehuda, Keren Bendel, Ari Freund, and Dror Rawitz. Local ratio: A unified framework for approximation algorithms. *ACM Computing Surveys*, 36(4):422–463, 2004.

[BYJK+02] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *Proceedings of the 6th International Workshop on Randomization and Approximation Techniques*, RANDOM '02, page 1–10, 2002.

[BYKS02] Ziv Bar-Yossef, Ravi Kumar, and D Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 623–632. Society for Industrial and Applied Mathematics, 2002.

[BZ11] Vladimir Batagelj and Matjaž Zaveršnik. Fast algorithms for determining (generalized) core groups in social networks. *Advances in Data Analysis and Classification*, 5(2):129–145, 2011.

[BZ16] Jeremiah Blocki and Samson Zhou. On the computational complexity of minimal cumulative cost graph pebbling. *CoRR*, abs/1609.04449, 2016.

[BZ17] Jeremiah Blocki and Samson Zhou. On the depth-robustness and cumulative pebbling cost of argon2i. In *TCC (1)*, volume 10677 of *Lecture Notes in Computer Science*, pages 445–465. Springer, 2017.

[Cai03] Leizhen Cai. Parameterized complexity of vertex colouring. *Discrete Applied Mathematics*, 127(3):415–429, 2003.

[CB02] Richard Clayton and Mike Bond. Experience using a low-cost FPGA design to crack DES keys. In *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, pages 579–592, 2002.

[CC11] Shumo Chu and James Cheng. Triangle listing in massive networks and its applications. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 672–680, 2011.

[CC16] Chandra Chekuri and Julia Chuzhoy. Polynomial bounds for the grid-minor theorem. *Journal of the ACM*, 63(5):40:1–40:65, December 2016.

[CCC14] Pei-Ling Chen, Chung-Kuang Chou, and Ming-Syan Chen. Distributed algorithms for k-truss decomposition. In *IEEE International Conference on Big Data (Big Data)*, pages 471–480, 2014.

[CdCMGI+21] Victor A. Campos, Guilherme de C. M. Gomes, Allen Ibiapina, Raul Lopes, Ignasi Sau, and Ana Silva. Coloring problems on bipartite graphs of small diameter. *Electron. J. Comb.*, 28(2):P2.14, 2021.

[CDD+07] David Cash, Yan Zong Ding, Yevgeniy Dodis, Wenke Lee, Richard J. Lipton, and Shabsi Walfish. Intrusion-resilient key exchange in the bounded retrieval model. In *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*, pages 479–498, 2007.

[CE91] Marek Chrobak and David Eppstein. Planar orientations with low out-degree and compaction of adjacency matrices. *Theoretical Computer Science*, 86(2):243–266, 1991.

[CFG+19] Yi-Jun Chang, Manuela Fischer, Mohsen Ghaffari, Jara Uitto, and Yufan Zheng. The complexity of (Δ+1) coloring in congested clique, massively parallel computation, and centralized local computation. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 471–480. ACM, 2019.

[CG06] P. Chassaing and L. Gerin. Efficient estimation of the cardinality of large data sets. *Discrete Mathematics & Theoretical Computer Science*, pages 419–422, 2006.

[Cha73] Ashok K. Chandra. Efficient compilation of linear recursive programs. In *SWAT (FOCS)*, pages 16–25. IEEE Computer Society, 1973.

[Cha04] Gregory Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Not.*, 39(4):66–74, April 2004.

[CHDK+19] Keren Censor-Hillel, Neta Dafni, Victor I Kolobov, Ami Paz, and Gregory Schwartzman. Fast and simple deterministic algorithms for highly-dynamic networks. *arXiv preprint arXiv:1901.04008*, 2019.

[CHHK16] Keren Censor-Hillel, Elad Haramaty, and Zohar Karnin. Optimal dynamic distributed MIS. In *ACM Symposium on Principles of Distributed Computing*, pages 217–226, 2016.

[CHKK+19] Keren Censor-Hillel, Petteri Kaski, Janne H Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic methods in the congested clique. *Distributed Computing*, 32(6):461–478, 2019.

[CHP12] Timothy M Chan and Sariel Har-Peled. Approximation algorithms for maximum independent set of pseudo-disks. *Discrete & Computational Geometry*, 48(2):373–392, 2012.

[Chu11] Julia Chuzhoy. An algorithm for the graph crossing number problem. In *Proceedings of the 43rd Annual ACM Symposium on Theory of Computing*, pages 303–312, 2011.

[CLM+18] Artur Czumaj, Jakub Lacki, Aleksander Madry, Slobodan Mitrovic, Krzysztof Onak, and Piotr Sankowski. Round compression for parallel matching algorithms. In *STOC*, pages 471–484, 2018.

[CLNV15] Siu Man Chan, Massimo Lauria, Jakob Nordström, and Marc Vinyals. Hardness of approximation in PSPACE and separation results for pebble games. In *Proceedings of the IEEE 56th Annual Symposium on Foundations of Computer Science*, FOCS '15, pages 466–485, 2015.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.).* MIT Press, 2009.

[CLS+20] Yulin Che, Zhuohang Lai, Shixuan Sun, Yue Wang, and Qiong Luo. Accelerating truss decomposition on heterogeneous processors. *Proc. VLDB Endow.*, 13(10):1751–1764, June 2020.

[CLW06] Giovanni Di Crescenzo, Richard J. Lipton, and Shabsi Walfish. Perfectly secure password protocols in the bounded retrieval model. In *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, pages 225–244, 2006.

[CM84] Thomas F Coleman and Jorge J Moré. Estimation of sparse hessian matrices and graph coloring problems. *Mathematical programming*, 28(3):243–270, 1984.

[CMS11] Julia Chuzhoy, Yury Makarychev, and Anastasios Sidiropoulos. On graph crossing number and edge planarization. In *Proceedings of the 22nd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1050–1069, 2011.

[CN85] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on computing*, 14(1):210–223, 1985.

[CNP+11] Marek Cygan, Jesper Nederlof, Marcin Pilipczuk, Michal Pilipczuk, Joham MM van Rooij, and Jakub Onufry Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. In *Foundations of computer science (focs), 2011 ieee 52nd annual symposium on*, pages 150–159. IEEE, 2011.

[Coh09] Jonathan Cohen. Graph twiddling in a mapreduce world. *Computing in Science & Engineering*, 11(4):29–41, 2009.

[Coo73] Stephen A. Cook. An observation on time-storage trade off. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, pages 29–33, New York, NY, USA, 1973. ACM.

[CRSS16] Timothy Carpenter, Fabrice Rastello, P. Sadayappan, and Anastasios Sidiropoulos. Brief announcement: Approximating the I/O complexity of one-shot red-blue pebbling. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 161–163, New York, NY, USA, 2016. ACM.

[CS74] Stephen Cook and Ravi Sethi. Storage requirements for deterministic / polynomial time recognizable languages. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, pages 33–39, New York, NY, USA, 1974. ACM.

[CS13] Chandra Chekuri and Anastasios Sidiropoulos. Approximation algorithms for Euler genus and related problems. In *Proceedings of the IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 167–176. IEEE, 2013.

[CTR+17] Israel Casas, Javid Taheri, Rajiv Ranjan, Lizhe Wang, and Albert Y Zomaya. A balanced scheduler with data reuse and replication for scientific workflows in cloud computing systems. *Future Generation Computer Systems*, 74:168–178, 2017.

[CZF04] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SIAM International Conference on Data Mining (SDM)*, pages 442–446, 2004.

[CZL+20] Deming Chu, Fan Zhang, Xuemin Lin, Wenjie Zhang, Ying Zhang, Yinglong Xia, and Chenyi Zhang. Finding the best k in core decomposition: A time and space optimal solution. In *IEEE 36th International Conference on Data Engineering (ICDE)*, pages 685–696, 2020.

[DA98] S. Darbha and D. P. Agrawal. Optimal scheduling algorithm for distributed-memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 9:87–95, 1998.

[DAH17] V. S. Dave, N. K. Ahmed, and M. Hasan. PE-CLoG: Counting edge-centric local graphlets. In *IEEE International Conference on Big Data*, pages 586–595, 2017.

[Dal17] Mark RT Dale. *Applying graph theory in ecological research*. Cambridge University Press, 2017.

[DBS17] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 293–304, 2017.

[DBS18a] Maximilien Danisch, Oana Balalau, and Mauro Sozio. Listing k-cliques in sparse real-world graphs*. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, page 589–598, Republic and Canton of Geneva, CHE, 2018. International World Wide Web Conferences Steering Committee.

[DBS18b] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[DBS19] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 918–934, 2019.

[DDJP19] Konrad K. Dabrowski, François Dross, Matthew Johnson, and Daniël Paulusma. Filling the complexity gaps for colouring planar and bounded degree graphs. *J. Graph Theory*, 92(4):377–393, 2019.

[DDK+20] Laxman Dhulipala, David Durfee, Janardhan Kulkarni, Richard Peng, Saurabh Sawlani, and Xiaorui Sun. Parallel batch-dynamic graphs: Algorithms and lower bounds. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1300–1319, 2020.

[DDLS15] Pål Grønås Drange, Markus Sortland Dregi, Daniel Lokshtanov, and Blair D. Sullivan. On the threshold of intractability. In *Proceedings of the 23rd Annual European Symposium on Algorithms*, pages 411–423, Patras, Greece, September 2015.

[DDS16] Pål Grønås Drange, Markus S. Dregi, and R. B. Sandeep. Compressing bounded degree graphs. In *LATIN*, volume 9644 of *Lecture Notes in Computer Science*, pages 362–375. Springer, 2016.

[DDZ14] N. S. Dasari, R. Desh, and M. Zubair. ParK: An efficient algorithm for k-core decomposition on multicore processors. In *IEEE International Conference on Big Data*, pages 9–16, 2014.

[Dem17] Erik D. Demaine. Lecture 22: History of memory models. In *6.851: Advanced Data Structures*. Massachusetts Institute of Technology, Fall 2017.

[Deo17] Narsingh Deo. *Graph theory with applications to engineering and computer science*. Courier Dover Publications, 2017.

[DF94] Sajal K. Das and Paolo Ferragina. An $o(n)$ work EREW parallel algorithm for updating MST. In *Annual European Symposium on Algorithms (ESA)*, pages 331–342, 1994.

[DF95a] Rod G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness I: Basic results. *SIAM Journal on Computing*, 24(4):873–921, 1995.

[DF95b] Rod G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness II: On completeness for $W[1]$. *Theor. Comput. Sci.*, 141(1-2):109–131, April 1995.

[DF13] Rodney G Downey and Michael R Fellows. *Fundamentals of parameterized complexity*, volume 4. Springer, 2013.

[DFKP15] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. *Proofs of Space*, pages 585–605. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DGJ08] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson. *Implementation Challenge for Shortest Paths*, pages 395–398. Springer US, Boston, MA, 2008.

[DGK+18] Erik D. Demaine, Timothy D. Goodrich, Kyle Kloster, Brian Lavallee, Quanquan C. Liu, Blair D. Sullivan, Ali Vakilian, and Andrew van der Poel. Structural rounding: Approximation algorithms for graphs near an algorithmically tractable class. *CoRR*, abs/1806.02771, 2018.

[DGK+19] Erik D. Demaine, Timothy D. Goodrich, Kyle Kloster, Brian Lavallee, Quanquan C. Liu, Blair D. Sullivan, Ali Vakilian, and Andrew van der Poel. Structural rounding: Approximation algorithms for graphs near an algorithmically tractable class. In *ESA*, volume 144 of *LIPIcs*, pages 37:1–37:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[DGvH+15] Konrad K Dabrowski, Petr A Golovach, Pim van't Hof, Daniël Paulusma, and Dimitrios M Thilikos. Editing to a planar graph of given degrees. In *Computer Science–Theory and Applications*, pages 143–156. Springer, 2015.

[DH05] Erik D. Demaine and MohammadTaghi Hajiaghayi. Bidimensionality: New connections between FPT algorithms and PTASs. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 590–601, 2005.

[DHK11] Erik D. Demaine, MohammadTaghi Hajiaghayi, and Ken-ichi Kawarabayashi. Contraction decomposition in H-minor-free graphs and algorithmic applications. In *Proceedings of the 43rd Annual ACM Symposium on Theory of Computing*, pages 441–450, 2011.

[DKR+20] Sami Davies, Janardhan Kulkarni, Thomas Rothvoss, Jakub Tarnawski, and Yihao Zhang. Scheduling with communication delays via LP hierarchies and clustering. In *FOCS*, 2020.

[DKR+21] Sami Davies, Janardhan Kulkarni, Thomas Rothvoss, Jakub Tarnawski, and Yihao Zhang. Scheduling with communication delays via LP hierarchies and clustering II: weighted completion times on related machines. In *SODA 2021*, pages 2958–2977. SIAM, 2021.

[DKW10] Stefan Dziembowski, Tomasz Kazana, and Daniel Wichs. One-time computable and uncomputable functions. *IACR Cryptology ePrint Archive*, 2010:541, 2010.

[DKW11] Stefan Dziembowski, Tomasz Kazana, and Daniel Wichs. One-time computable self-erasing functions. In *Theory of Cryptography - 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings*, pages 125–143, 2011.

[DL17] Erik D. Demaine and Quanquan C. Liu. Inapproximability of the standard pebble game and hard to pebble graphs. In *Proceedings of the 16th International Symposium on Algorithms and Data Structures, WADS '17*, volume 10389 of *Lecture Notes in Computer Science*, pages 313–324. Springer, 2017.

[DL18] Erik D. Demaine and Quanquan C. Liu. Red-blue pebble game: Complexity of computing the trade-off between cache size and memory transfers. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, pages 195–204, 2018.

[DLL+18] Erik D. Demaine, Andrea Lincoln, Quanquan C. Liu, Jayson Lynch, and Virginia Vassilevska Williams. Fine-grained I/O complexity via reductions: New lower bounds, faster algorithms, and a time hierarchy. In *9th Innovations in Theoretical Computer Science Conference, ITCS 2018, January 11-14, 2018, Cambridge, MA, USA*, pages 34:1–34:23, 2018.

[DLP12] Danny Dolev, Christoph Lenzen, and Shir Peled. "Tri, Tri again": Finding triangles and small subgraphs in a distributed setting. *Distributed Computing*, page 195–209, 2012.

[DLP18] Thaddeus Dryja, Quanquan C. Liu, and Sunoo Park. Static-memory-hard functions, and modeling the cost of space vs. time. In *TCC*, volume 11239 of *Lecture Notes in Computer Science*, pages 33–66. Springer, 2018.

[DLSY20] Laxman Dhulipala, Quanquan C. Liu, Julian Shun, and Shangdi Yu. Parallel batch-dynamic k-clique counting. *CoRR*, abs/2003.13585, 2020. To appear in APOCS 2021.

[DLSY21] Laxman Dhulipala, Quanquan C. Liu, Julian Shun, and Shangdi Yu. Parallel batch-dynamic k-clique counting. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 129–143, 2021.

[Dra15] Pål Grønås Drange. *Parameterized Graph Modification Algorithms*. PhD thesis, The University of Bergen, 2015.

[DT13] Zdeněk Dvořák and Vojtěch Tůma. A dynamic data structure for counting subgraphs in sparse graphs. In *Algorithms and Data Structures*, pages 304–315, 2013.

[Dzi06] Stefan Dziembowski. Intrusion-resilience via the bounded-storage model. In *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, pages 207–224, 2006.

[EG04] Friedrich Eisenbrand and Fabrizio Grandoni. On the complexity of fixed parameter clique and dominating set. *Theor. Comput. Sci.*, 326(1-3):57–67, October 2004.

[EGST12] David Eppstein, Michael T. Goodrich, Darren Strash, and Lowell Trott. Extended dynamic subgraph statistics using h-index parameterized data structures. *Theoretical Computer Science*, 447:44 – 52, 2012. Combinational Algorithms and Applications.

[EJRB10] D. Ediger, K. Jiang, J. Riedy, and D. A. Bader. Massive streaming data analytics: A case study with clustering coefficients. In *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, 2010.

[EK10] David Easley and Jon Kleinberg. *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge University Press, USA, 2010.

[EKS] Tomáš Ebenlendr, Petr Kolman, and Jirı Sgall. An approximation algorithm for bounded degree deletion. http://kam.mff.cuni.cz/~kolman/papers/star.pdf.

[ELM18] Hossein Esfandiari, Silvio Lattanzi, and Vahab Mirrokni. Parallel and streaming algorithms for k-core decomposition. In *Proceedings of the 35th International Conference on Machine Learning*, pages 1397–1406, 2018.

[ELRS17] Talya Eden, Amit Levi, Dana Ron, and C Seshadhri. Approximately counting triangles in sublinear time. *SIAM Journal on Computing*, 46(5):1603–1646, 2017.

[ELS10] David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In *Algorithms and Computation*, pages 403–414, 2010.

[ELS13] David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in large sparse real-world graphs. *ACM Journal of Experimental Algorithms*, 18(3):364–375, 2013.

[EM13] B. Elser and A. Montresor. An evaluation study of BigData frameworks for graph processing. In *IEEE International Conference on Big Data*, pages 60–67, 2013.

[EMRB12] David Ediger, Robert McColl, Jason Riedy, and David A Bader. Stinger: High performance data structure for streaming graphs. In *IEEE Conference on High Performance Extreme Computing (HPEC)*, pages 1–5, 2012.

[ER61] Paul Erdős and Alfréd Rényi. On a classical problem of probability theory. *Magyar Tudományos Akadémia Matematikai Kutató Intézetének Közleményei*, 6:215–220, 1961.

[ERP+15] Venmugil Elango, Fabrice Rastello, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. On characterizing the data access complexity of programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 567–580, 2015.

[ERS20] Talya Eden, Dana Ron, and C. Seshadhri. Faster sublinear approximation of the number of k-cliques in low-arboricity graphs. In *Proceedings of the*

2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020, pages 1467–1478, 2020.

[ES09] David Eppstein and Emma S. Spiro. The h-index of a graph and its application to dynamic subgraph statistics. In *Algorithms and Data Structures*, pages 278–289, 2009.

[ESBD16] Ethan R. Elenberg, Karthikeyan Shanmugam, Michael Borokhovich, and Alexandros G. Dimakis. Distributed estimation of graph 4-profiles. In *International Conference on World Wide Web (WWW)*, pages 483–493, 2016.

[ESTW19] Fatemeh Esfahani, Venkatesh Srinivasan, Alex Thomo, and Kui Wu. Efficient computation of probabilistic core decomposition at web-scale. In *International Conference on Extending Database Technology*, pages 325–336, 2019.

[ESTW20] Fatemeh Esfahani, Venkatesh Srinivasan, Alex Thomo, and Kui Wu. Nucleus decomposition in probabilistic graphs: Hardness and algorithms. *arXiv preprint arXiv:2006.01958*, 2020.

[exp20] https://github.com/qqliu/mpc-triangle-count-exact-approx-simulations, 2020.

[FFF15] Irene Finocchi, Marco Finocchi, and Emanuele G. Fusco. Clique counting in MapReduce: Algorithms and experiments. *J. Exp. Algorithmics*, 20:1.7:1–1.7:20, October 2015.

[FFGM07] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA: Analysis of Algorithms*, pages 137–156, 2007.

[FGK11] Jiří Fiala, Petr A Golovach, and Jan Kratochvíl. Parameterized complexity of coloring problems: Treewidth versus vertex cover. *Theoretical Computer Science*, 412(23):2513–2523, 2011.

[FGLS09] Fedor V Fomin, Petr A Golovach, Daniel Lokshtanov, and Saket Saurabh. Clique-width: on the price of generality. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 825–834. SIAM, 2009.

[FHL08] Uriel Feige, MohammadTaghi Hajiaghayi, and James R. Lee. Improved approximation algorithms for minimum weight vertex separators. *SIAM Journal on Computing*, 38(2):629–657, 2008.

[FK98] Uriel Feige and Joe Kilian. Zero knowledge and the chromatic number. *J. Comput. Syst. Sci.*, 57(2):187–199, 1998. Announced at CCC'96.

[FKP+14] Fedor V. Fomin, Stefan Kratsch, Marcin Pilipczuk, Michal Pilipczuk, and Yngve Villanger. Tight bounds for parameterized complexity of cluster editing with a small number of clusters. *Journal of Computer and System Sciences*, 80(7):1430–1447, 2014.

[FL94] Paolo Ferragina and Fabrizio Luccio. Batch dynamic algorithms for two graph problems. In *Parallel Architectures and Languages Europe (PARLE)*, pages 713–724, 1994.

[FLMS12] Fedor V. Fomin, Daniel Lokshtanov, Neeldhara Misra, and Saket Saurabh. Planar F-deletion: Approximation, kernelization and optimal FPT algorithms. In *Proceedings of the IEEE 53rd Annual Symposium on Foundations of Computer Science*, pages 470–479, 2012.

[FLW13] Christian Forler, Stefan Lucks, and Jakob Wenzel. Catena: A memory-consuming password scrambler. *IACR Cryptology ePrint Archive*, 2013:525, 2013.

[FM95] Tomás Feder and Rajeev Motwani. Clique partitions, graph compression and speeding-up algorithms. *Journal of Computer and System Sciences*, 51(2):261–272, 1995.

[Fuj98] Toshihiro Fujito. A unified approximation algorithm for node-deletion problems. *Discrete Applied Mathematics*, 86(2-3):213–231, 1998.

[Gar06] Martin Gardner. *The colossal book of short puzzles and problems : combinatorics, probability, algebra, geometry, topology, chess, logic, cryptarithms, wordplay, physics and other topics of recreational mathematics*. Norton, New York, 1st ed.. edition, 2006.

[GBGL20] Edoardo Galimberti, Francesco Bonchi, Francesco Gullo, and Tommaso Lanciano. Core decomposition in multilayer networks: Theory, algorithms, and applications. *ACM Trans. Knowl. Discov. Data*, 14(1), January 2020.

[GCP] Google cloud platform. https://cloud.google.com/. Accessed: 2021-07-01.

[GG06] Gaurav Goel and Jens Gustedt. Bounded arboricity to determine the local structure of sparse graphs. In *Graph-Theoretic Concepts in Computer Science, 32nd International Workshop, WG 2006, Bergen, Norway, June 22-24, 2006, Revised Papers*, pages 159–167, 2006.

[GGK+18] Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrović, and Ronitt Rubinfeld. Improved massively parallel computation algorithms for mis, matching, and vertex cover. In *PODC*. arXiv:1802.08237, 2018.

[GHN04] Jiong Guo, Falk Hüffner, and Rolf Niedermeier. A structural view on parameterizing problems: Distance from triviality. In *International Workshop on Parameterized and Exact Computation*, pages 162–173. Springer, 2004.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[GJS74] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, pages 47–63, New York, NY, USA, 1974. ACM.

[GK96] Sudipto Guha and Samir Khuller. Approximation algorithms for connected dominating sets. In *European symposium on algorithms*, pages 179–193. Springer, 1996.

[GKKP17] Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. Online and dynamic algorithms for set cover. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2017, page 537–550, New York, NY, USA, 2017. Association for Computing Machinery.

[GKMS19] Buddhima Gamlath, Sagar Kale, Slobodan Mitrovic, and Ola Svensson. Weighted matchings via unweighted augmentations. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 491–500. ACM, 2019.

[GKU19] Mohsen Ghaffari, Fabian Kuhn, and Jara Uitto. Conditional hardness results for massively parallel computation from distributed lower bounds. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 1650–1663. IEEE Computer Society, 2019.

[GL17] Venkatesan Guruswami and Euiwoong Lee. Inapproximability of H-transversal/packing. *SIAM Journal on Discrete Mathematics*, 31(3):1552–1571, 2017.

[GLL+18] Anupam Gupta, Euiwoong Lee, Jason Li, Pasin Manurangsi, and Michał Włodarczyk. Losing treewidth by separating subsets. *CoRR*, abs/1804.01366, 2018.

[GLM12] Enrico Gregori, Luciano Lenzini, and Simone Mainardi. Parallel k-clique community detection on large-scale networks. *IEEE Transactions on Parallel and Distributed Systems*, 24(8):1651–1660, 2012.

[GLM19] Mohsen Ghaffari, Silvio Lattanzi, and Slobodan Mitrovic. Improved parallel algorithms for density-based network clustering. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 2201–2210. PMLR, 2019.

[GLT80] John R. Gilbert, Thomas Lengauer, and Robert Endre Tarjan. The pebbling problem is complete in polynomial space. *SIAM Journal on Computing*, 9(3):513–524, 1980.

[GMP05] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. What color is your jacobian? graph coloring for computing derivatives. *SIAM review*, 47(4):629–705, 2005.

[GMV91] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 698–710, 1991.

[GNT20] Mohsen Ghaffari, Krzysztof Nowicki, and Mikkel Thorup. Faster algorithms for edge connectivity via random 2-out contractions. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 1260–1279. SIAM, 2020.

[GP11] Michael T. Goodrich and Paweł Pszona. External-memory network analysis algorithms for naturally sparse graphs. In *European Symposium on Algorithms*, page 664–676, 2011.

[Gra] GraphChallenge. http://graphchallenge.mit.edu/.

[Gra69] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17:416–429, 1969.

[Gra71] R. L. Graham. Bounds on multiprocessing anomalies and related packing algorithms. In *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference*, AFIPS '72 (Spring), page 205–217, New York, NY, USA, 1971. Association for Computing Machinery.

[Gra77] Mark S Granovetter. The strength of weak ties. In *Social networks*, pages 347–367. Elsevier, 1977.

[GS19] Mohsen Ghaffari and Ali Sayyadi. Distributed Arboricity-Dependent Graph Coloring via All-to-All Communication. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, volume 132 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 142:1–142:14, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[GSSB15] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. A top-down parallel semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 24–34, 2015.

[GSZ11] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *Proceedings of the 22Nd International Conference on Algorithms and Computation*, ISAAC'11, pages 374–383, Berlin, Heidelberg, 2011. Springer-Verlag.

[GU19] Mohsen Ghaffari and Jara Uitto. Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1636–1653. SIAM, 2019.

[GXTL10] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. A survey of graph edit distance. *Pattern Analysis and Applications*, 13(1):113–129, 2010.

[HCQ+14] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. Querying k-truss community in large and dynamic graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1311–1322, 2014.

[HD14] Tomaz Hocevar and Janez Demsar. A combinatorial approach to graphlet counting. *Bioinformatics*, pages 559–65, 2014.

[HHS21] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Recent advances in fully dynamic graph algorithms. *CoRR*, abs/2102.11169, 2021.

[HJMA07] John Healy, Jeannette Janssen, Evangelos Milios, and William Aiello. *Characterization of Graphs Using Degree Cores*, pages 137–148. 2007.

[HKN15] Falk Hüffner, Christian Komusiewicz, and André Nichterlein. Editing graphs into few cliques: Complexity, approximation, and kernelization schemes. In *Proceedings of the 14th International Symposium on Algorithms and Data Structures*, pages 410–421. Springer, 2015.

[HKNS15] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *ACM Symposium on Theory of Computing*, pages 21–30, 2015.

[HLL18] Nicholas J. A. Harvey, Christopher Liaw, and Paul Liu. Greedy and local ratio algorithms in the MapReduce model. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 43–52, New York, NY, USA, 2018. ACM.

[HLV94] J.A. Hoogeveen, J.K. Lenstra, and B. Veltman. Three, four, five, six, or the complexity of scheduling with communication delays. *Operations Research Letters*, 16(3):129 – 137, 1994.

[HM01] Claire Hanen and Alix Munier. An approximation algorithm for scheduling dependent tasks on m processors with small communication delays. *Discret. Appl. Math.*, 108(3):239–257, 2001.

[HNW20] Monika Henzinger, Stefan Neumann, and Andreas Wiese. Explicit and implicit dynamic coloring of graphs with bounded arboricity. *CoRR*, abs/2002.10142, 2020.

[HP10] Philipp Hertel and Toniann Pitassi. The pspace-completeness of black-white pebbling. *SIAM J. Comput.*, 39(6):2622–2682, 2010.

[HP15] James W Hegeman and Sriram V Pemmaraju. Lessons from the congested clique applied to MapReduce. *Theoretical Computer Science*, 608:268–281, 2015.

[HP17] Dawei Huang and Seth Pettie. Approximate generalized matching: f-factors and f-edge covers. *CoRR*, abs/1706.05761, 2017.

[HP19] Monika Henzinger and Pan Peng. Constant-time dynamic (Δ+1)-coloring and weight approximation for minimum spanning forest: Dynamic algorithms meet property testing. *CoRR*, abs/1907.04745, 2019.

[HP20] Monika Henzinger and Pan Peng. Constant-time dynamic (Δ+1)-coloring. In Christophe Paul and Markus Bläser, editors, *37th International Symposium on Theoretical Aspects of Computer Science, STACS 2020, March 10-13, 2020, Montpellier, France*, volume 154 of *LIPIcs*, pages 53:1–53:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[HPV77] John Hopcroft, Wolfgang Paul, and Leslie Valiant. On time versus space. *J. ACM*, 24(2):332–337, April 1977.

[HQ17] Sariel Har-Peled and Kent Quanrud. Approximation algorithms for polynomial-expansion and low-density graphs. *SIAM Journal on Computing*, 46(6):1712–1744, 2017.

[HR05] Robert A. Hanneman and Mark Riddle. *Introduction to social network methods*. University of California, Riverside, 2005.

[HSY+20] Q. Hua, Y. Shi, D. Yu, H. Jin, J. Yu, Z. Cai, X. Cheng, and H. Chen. Faster parallel core maintenance algorithms in dynamic graphs. *IEEE Transactions on Parallel and Distributed Systems*, 31(6):1287–1300, 2020.

[HW17] Daniel J. Harvey and David R. Wood. Parameters tied to treewidth. *Journal of Graph Theory*, 84(4):364–385, 2017.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, volume 41, pages 59–72. ACM, 2007.

[ILMP19] Giuseppe F. Italiano, Silvio Lattanzi, Vahab S. Mirrokni, and Nikos Parotsidis. Dynamic algorithms for the massively parallel computation model. In Christian Scheideler and Petra Berenbrink, editors, *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019*, pages 49–58. ACM, 2019.

[ILS] Internet live stats. https://www.internetlivestats.com/one-second/. Accessed: 2021-07-01.

[IMS17] Sungjin Im, Benjamin Moseley, and Xiaorui Sun. Efficient massively parallel methods for dynamic programming. In *STOC*, pages 798–811, 2017.

[IR77] Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. In *Annual ACM Symposium on Theory of Computing (STOC)*, pages 1–10, 1977.

[Jaj92] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[JLS14] Bart M. P. Jansen, Daniel Lokshtanov, and Saket Saurabh. A near-optimal planarization algorithm. In *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1802–1811, 2014.

[JP94] Mark T Jones and Paul E Plassmann. Scalable iterative solution of sparse linear systems. *Parallel Computing*, 20(5):753–773, 1994.

[JS97] Klaus Jansen and Petra Scheffler. Generalized coloring for tree-like graphs. *Discrete Applied Mathematics*, 75(2):135–155, 1997.

[JS17a] Shweta Jain and C Seshadhri. A fast and provable method for estimating clique counts using turán's theorem. In *Proceedings of the 26th International Conference on World Wide Web*, pages 441–449, 2017.

[JS17b] Shweta Jain and C. Seshadhri. A fast and provable method for estimating clique counts using Turán's theorem. In *International Conference on World Wide Web (WWW)*, pages 441–449, 2017.

[JSP13] Madhav Jha, Comandur Seshadhri, and Ali Pinar. A space efficient streaming algorithm for triangle counting using the birthday paradox. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 589–597, 2013.

[JWK81] Hong Jia-Wei and H. T. Kung. I/o complexity: The red-blue pebble game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, STOC '81, pages 326–333, 1981.

[JWY+18] H. Jin, N. Wang, D. Yu, Q. Hua, X. Shi, and X. Xie. Core maintenance in dynamic graphs: A parallel approach based on matching. *IEEE Transactions on Parallel and Distributed Systems*, 29(11):2416–2428, 2018.

[Kaw09] Ken-ichi Kawarabayashi. Planarity allowing few error vertices in linear time. In *Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 639–648. IEEE, 2009.

[KBST15] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. K-core decomposition of large networks on a single pc. *Proc. VLDB Endow.*, 9(1):13–23, September 2015.

[KD79] Mukkai S Krishnamoorthy and Narsingh Deo. Node-deletion NP-complete problems. *SIAM Journal on Computing*, 8(4):619–625, 1979.

[Kha17] Shahbaz Khan. Near optimal parallel algorithms for dynamic DFS in undirected graphs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 283–292, 2017.

[KHSL16] Tim Kaler, William Hasenplaugh, Tao B. Schardl, and Charles E. Leiserson. Executing dynamic data-graph computations deterministically using chromatic scheduling. *ACM Trans. Parallel Comput.*, 3(1), July 2016.

[KK16] William Kocay and Donald L Kreher. *Graphs, algorithms, and optimization*. Chapman and Hall/CRC, 2016.

[KKO16] Michal Kotrbčík, Rastislav Královič, and Sebastian Ordyniak. Edge-editing to a dense and a sparse graph class. In *Proceedings of the 12th Latin American Symposium on Theoretical Informatics*, pages 562–575. Springer, 2016.

[KLPM10] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proceedings of the International Conference on World Wide Web*, pages 591–600, 2010.

[KLTY20] Janardhan Kulkarni, Shi Li, Jakub Tarnawski, and Minwei Ye. Hierarchy-based algorithms for minimizing makespan under precedence and communication constraints. In *Proceedings of the Fortieth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2020.

[KM17a] H. Kabir and K. Madduri. Parallel k-core decomposition on multicore platforms. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1482–1491, 2017.

[KM17b] Humayun Kabir and Kamesh Madduri. Parallel k-truss decomposition on multicore systems. In *IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017.

[KMPT12] Mihail N Kolountzakis, Gary L Miller, Richard Peng, and Charalampos E Tsourakakis. Efficient triangle counting in large graphs via degree-based vertex partitioning. *Internet Mathematics*, 8(1-2):161–185, 2012.

[KMSS12] Daniel M Kane, Kurt Mehlhorn, Thomas Sauerwald, and He Sun. Counting arbitrary subgraphs in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 598–609. Springer, 2012.

[KNN+18] Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Counting triangles under updates. In *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018.*, 2018.

[KNN+19] Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Counting triangles under updates in worst-case optimal time. In *International Conference on Database Theory (ICDT)*, volume 127, pages 4:1–4:18, 2019.

[KNW10] Daniel M. Kane, Jelani Nelson, and David P. Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '10, page 41–52, 2010.

[Kom18] Christian Komusiewicz. Tight running time lower bounds for vertex deletion problems. *TOCT*, 10(2):6:1–6:18, 2018.

[KP06] Subhash Khot and Ashok Kumar Ponnuswami. Better inapproximability results for maxclique, chromatic number and min-3lin-deletion. In *ICALP*, pages 226–237, 2006.

[KP11] Kishore Kothapalli and Sriram Pemmaraju. Distributed graph coloring in a few rounds. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11, page 31–40, New York, NY, USA, 2011. Association for Computing Machinery.

[KPP+14] Tamara G Kolda, Ali Pinar, Todd Plantenga, C Seshadhri, and Christine Task. Counting triangles in massive graphs with mapreduce. *SIAM Journal on Scientific Computing*, 36(5):S48–S77, 2014.

[KPP16] Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3sum conjecture. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 1272–1287. SIAM, 2016.

[KPR18] Tsvi Kopelowitz, Ely Porat, and Yair Rosenmutter. Improved worst-case deterministic parallel dynamic minimum spanning forest. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 333–341, 2018.

[KPS13] Tamara G. Kolda, Ali Pinar, and C. Seshadhri. Triadic measures on graphs: The power of wedge sampling. In *Proceedings of the 13th SIAM International Conference on Data Mining, May 2-4, 2013. Austin, Texas, USA*, pages 10–18. SIAM, 2013.

[KPS+19] Ravi Kumar, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua R. Wang. Efficient rematerialization for deep networks. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 15146–15155, 2019.

[KR03] Daniel Kobler and Udi Rotics. Edge dominating set and colorings on graphs with fixed clique-width. *Discrete Applied Mathematics*, 126(2):197–221, 2003.

[KS17] Ken-ichi Kawarabayashi and Anastasios Sidiropoulos. Polylogarithmic approximation for minimum planarization (almost). In *Proceedings of the 58th IEEE Annual Symposium on Foundations of Computer Science*, pages 779–788, Berkeley, CA, October 2017.

[KSV10] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for MapReduce. In *SODA*, pages 938–948, 2010.

[KT17] Ken-Ichi Kawarabayashi and Mikkel Thorup. Coloring 3-colorable graphs with less than $n^{1/5}$ colors. *J. ACM*, 64(1):4:1–4:23, March 2017.

[Lat08] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical computer science*, 407(1-3):458–473, 2008.

[Lee17] Euiwoong Lee. Partitioning a graph into small pieces with applications to path transversal. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1546–1558. Society for Industrial and Applied Mathematics, 2017.

[Lei79] Frank Thomson Leighton. A graph coloring algorithm for large scheduling problems. *Journal of research of the national bureau of standards*, 84(6):489–506, 1979.

[Lew78] John M. Lewis. On the complexity of the maximum subgraph problem. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 265–274. ACM, 1978.

[Lew15] R.M. R. Lewis. *A Guide to Graph Colouring: Algorithms and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2015.

[LG14] François Le Gall. Powers of tensors and fast matrix multiplication. In *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 296–303, 2014.

[LK14] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[LKPR20] Kartik Lakhotia, Rajgopal Kannan, Viktor K. Prasanna, and César A. F. De Rose. RECEIPT: refine coarse-grained independent tasks for parallel tip decomposition of bipartite graphs. *Proc. VLDB Endow.*, 14(3):404–417, 2020.

[LLLX18] Andrea Lincoln, Quanquan C. Liu, Jayson Lynch, and Helen Xu. Cache-adaptive exploration: Experimental results and scan-hiding for adaptivity. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, pages 213–222, 2018.

[LMOS19] Jakub Lacki, Slobodan Mitrovic, Krzysztof Onak, and Piotr Sankowski. Walking randomly, massively, and efficiently. *CoRR*, abs/1907.05391, 2019.

[LMSV11] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in MapReduce. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 85–94, 2011.

[LPS+21] Quanquan C. Liu, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua R. Wang. Scheduling with communication delay in near-linear time. *CoRR*, abs/2108.02770, 2021.

[LQLC15] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. Scalable subgraph enumeration in mapreduce. *Proceedings of the VLDB Endowment*, 8(10):974–985, 2015.

[LR02] Renaud Lepère and Christophe Rapine. An asymptotic $O(\ln \rho / \ln \ln \rho)$-approximation algorithm for the scheduling problem with duplication on large communication delay graphs. In *STACS*, volume 2285 of *Lecture Notes in Computer Science*, pages 154–165, 2002.

[LRSvdP20] Brian Lavallee, Hayley Russell, Blair D Sullivan, and Andrew van der Poel. Approximating vertex cover using structural rounding. In *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 70–80. SIAM, 2020.

[LS16] Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

[LSY+21] Quanquan C. Liu, Jessica Shi, Shangdi Yu, Laxman Dhulipala, and Julian Shun. Parallel batch-dynamic k-core decomposition. *CoRR*, abs/2106.03824, 2021.

[LT82] Thomas Lengauer and Robert E. Tarjan. Asymptotically tight bounds on time-space trade-offs in a pebble game. *J. ACM*, 29(4):1087–1130, October 1982.

[Lub86] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.

[LW70] Don R. Lick and Arthur T. White. k-degenerate graphs. *Canadian Journal of Mathematics*, 22:1082–1096, 1970.

[LW10] Christoph Lenzen and Roger Wattenhofer. Minimum dominating set approximation in graphs of bounded arboricity. In *Proceedings of the International Symposium on Distributed Computing*, pages 510–524. Springer, 2010.

[LW15] Stefan Lucks and Jakob Wenzel. Catena variants - different instantiations for an extremely flexible password-hashing framework. In *Technology and Practice of Passwords - 9th International Conference, PASSWORDS 2015, Cambridge, UK, December 7-9, 2015, Proceedings*, pages 95–119, 2015.

[LY80] John M. Lewis and Mihalis Yannakakis. The node-deletion problem for hereditary properties is NP-complete. *Journal of Computer and System Sciences*, 20(2):219–230, 1980.

[LY93] Carsten Lund and Mihalis Yannakakis. The approximation of maximum subgraph problems. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*, pages 40–51. Springer, 1993.

[LYL+19] Qi Luo, Dongxiao Yu, Feng Li, Zhenhao Dou, Zhipeng Cai, Jiguo Yu, and Xiuzhen Cheng. Distributed core decomposition in probabilistic graphs. In *Computational Data and Social Networks*, pages 16–32, 2019.

[LYL+20] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. Efficient (α, β)-core computation in bipartite graphs. *VLDB J.*, 29(5):1075–1099, 2020.

[LYM14] R. Li, J. Yu, and R. Mao. Efficient core maintenance in large dynamic graphs. *IEEE Transactions on Knowledge & Data Engineering*, 26(10):2453–2465, oct 2014.

[LYS+21] Qi Luo, Dongxiao Yu, Hao Sheng, Jiguo Yu, and Xiuzhen Cheng. Distributed algorithm for truss maintenance in dynamic graphs. In *Parallel and Distributed Computing, Applications and Technologies*, pages 104–115, 2021.

[LZL+21] Zhe Lin, Fan Zhang, Xuemin Lin, Wenjie Zhang, and Zhihong Tian. Hierarchical core maintenance on large dynamic graphs. *Proc. VLDB Endow.*, 14(5):757–770, 2021.

[LZZ+19] Conggai Li, Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. Efficient progressive minimum k-core search. *Proc. VLDB Endow.*, 13(3):362–375, November 2019.

[Mar04] Dániel Marx. Graph colouring problems and their applications in scheduling. *Periodica Polytechnica Electrical Engineering (Archives)*, 48(1-2):11–16, 2004.

[Mar06] Dániel Marx. Parameterized coloring problems on chordal graphs. *Theoretical Computer Science*, 351(3):407–424, 2006.

[Mar08] Dániel Marx. Parameterized complexity and approximation algorithms. *The Computer Journal*, 51(1):60–78, 2008.

[Mat10] Luke Mathieson. The parameterized complexity of editing graphs for bounded degeneracy. *Theoretical Computer Science*, 411(34):3181–3187, 2010.

[MB83] David W. Matula and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417–427, July 1983.

[MBG17] Devavret Makkar, David A. Bader, and Oded Green. Exact and parallel triangle counting in dynamic graphs. In *24th IEEE International Conference on High Performance Computing, HiPC 2017, Jaipur, India, December 18-21, 2017*, pages 2–12, Los Alamitos, CA, 2017. IEEE Computer Society.

[MCT20] Amir Mehrafsa, Sean Chester, and Alex Thomo. Vectorising k-core decomposition for GPU acceleration. In *International Conference on Scientific and Statistical Database Management*, 2020.

[Mer79] Ralph Charles Merkle. *Secrecy, Authentication, and Public Key Systems.* PhD thesis, Stanford University, Stanford, CA, USA, 1979. AAI8001972.

[MFC+20] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks VS Lakshmanan, Wenjie Zhang, and Xuemin Lin. Efficient algorithms for densest subgraph discovery on large directed graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1051–1066, 2020.

[MH97] Alix Munier and Claire Hanen. Using duplication for scheduling unitary tasks on m processors with unit communication delays. *Theoretical Computer Science*, 178(1):119 – 127, 1997.

[MIM15] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what COST? In *USENIX Conference on Hot Topics in Operating Systems (HotOS)*, 2015.

[MK97] Alix Munier and Jean-Claude König. A heuristic for a scheduling problem with communi-cation delays. *Operations Research*, 45(1):145–147, 1997.

[MM09] Avner Magen and Mohammad Moharrami. Robust algorithms for on minor-free graphs based on the Sherali-Adams hierarchy. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 258–271. Springer, 2009.

[MMSS20] Sourav Medya, Tianyi Ma, Arlei Silva, and Ambuj Singh. A game theoretic approach for k-core minimization. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1922–1924, 2020.

[MPM13] A. Montresor, F. De Pellegrini, and D. Miorandi. Distributed k-core decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 24(2):288–300, 2013.

[MPP+15] Michael Mitzenmacher, Jakub Pachocki, Richard Peng, Charalampos Tsourakakis, and Shen Chen Xu. Scalable large near-clique detection in large-scale networks via sampling. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 815–824, 2015.

[MPS19] Barnaby Martin, Daniël Paulusma, and Siani Smith. Colouring H-Free Graphs of Bounded Diameter. In Peter Rossmanith, Pinar Heggernes, and Joost-Pieter Katoen, editors, *44th International Symposium on Mathematical Foundations of Computer Science (MFCS 2019)*, volume 138 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:14, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[MPS21] B. Martin, D. Paulusma, and S. Smith. Colouring graphs of bounded diameter in the absence of small cycles. In *CIAC 2021*, Lecture Notes in Computer Science. Springer, 2021.

[MRS+20] Biswaroop Maiti, Rajmohan Rajaraman, David Stalfa, Zoya Svitkina, and Aravindan Vijayaraghavan. Scheduling precedence-constrained jobs on related machines withcommunication delay. In *FOCS*, 2020.

[MS08] Luke Mathieson and Stefan Szeider. Parameterized graph editing with chosen vertex degrees. In *Combinatorial Optimization and Applications*, pages 13–22. Springer, 2008.

[MS12] Dániel Marx and Ildikó Schlotter. Obtaining a planar graph by vertex deletion. *Algorithmica*, 62(3-4):807–822, 2012.

[MSZ15] Michael McCourt, Barry Smith, and Hong Zhang. Sparse matrix-matrix products executed through coloring. *SIAM Journal on Matrix Analysis and Applications*, 36(1):90–109, 2015.

[Mun99] Alix Munier. Approximation algorithms for scheduling trees with general communication delays. *Parallel Computing*, 25(1):41–48, 1999.

[MV91] Yossi Matias and Uzi Vishkin. On parallel hashing and integer sorting. *Journal of Algorithms*, 12(4):573–606, 1991.

[MVV16] Andrew McGregor, Sofya Vorotnikova, and Hoa T Vu. Better algorithms for counting triangles in data streams. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 401–411, 2016.

[NdM12] J. Nešetřil and P. O. de Mendez. *Sparsity: Graphs, Structures, and Algorithms*. Algorithms and Combinatorics. Springer Berlin Heidelberg, 2012.

[New03] Mark EJ Newman. The structure and function of complex networks. *SIAM review*, 45(2):167–256, 2003.

[NG04] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[Nor12] Jakob Nordström. On the relative strength of pebbling and resolution. *ACM Trans. Comput. Log.*, 13(2):16:1–16:43, 2012.

[Nor15] Jakob Nordstrom. New wine into old wineskins: A survey of some pebbling classics with supplemental results. 2015.

[NP85] Jaroslav Nešetřil and Svatopluk Poljak. On the complexity of the subgraph problem. *Commentationes Mathematicae Universitatis Carolinae*, 026(2):415–419, 1985.

[NPRR18] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *J. ACM*, 65(3):16:1–16:40, March 2018.

[NS15] Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. *ACM Transactions on Algorithms (TALG)*, 12(1):1–15, 2015.

[OB03] Michael Okun and Amnon Barak. A new approach for approximating node deletion problems. *Information Processing Letters*, 88(5):231–236, 2003.

[Pap89] T. Pappas. *The Joy of Mathematics: Discovering Mathematics All Around You*. Wide World Pub./Tetra, 1989.

[Pat10] Mihai Patrascu. Towards polynomial lower bounds for dynamic problems. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 603–610, 2010.

[PC13] Ha-Myung Park and Chin-Wan Chung. An efficient MapReduce algorithm for counting triangles in a very large graph. In *ACM Conference on Information and Knowledge Management (CIKM)*, pages 539–548, 2013.

[Per09] Colin Percival. Stronger key derivation via sequential memory-hard functions, 2009. Presented at BSDCan 2009. Available online at: http://www.tarsnap.com/scrypt/scrypt.pdf.

[PH70] Michael S. Paterson and Carl E. Hewitt. Record of the project mac conference on concurrent systems and parallel computation. chapter Comparative Schematology, pages 119–127. ACM, New York, NY, USA, 1970.

[Pic95] Christophe Picouleau. New complexity results on scheduling with small communication delays. *Discret. Appl. Math.*, 60(1-3):331–342, 1995.

[Pip82] Nicholas Pippenger. Advances in pebbling (preliminary version). In *ICALP*, volume 140 of *Lecture Notes in Computer Science*, pages 407–417. Springer, 1982.

[PKT14] K. Pechlivanidou, D. Katsaros, and L. Tassiulas. Mapreduce-based distributed k-shell decomposition for online social networks. In *IEEE World Congress on Services*, pages 30–37, 2014.

[PLW96] Michael A. Palis, Jing-Chiou Liou, and David S. L. Wei. Task clustering and scheduling for distributed memory parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):46–55, 1996.

[Pot17] Aaron Potechin. Bounds on monotone switching networks for directed connectivity. *J. ACM*, 64(4):29:1–29:48, 2017.

[PSKP14] Ha-Myung Park, Francesco Silvestri, U Kang, and Rasmus Pagh. MapReduce triangle enumeration with guarantees. In *ACM Conference on Information and Knowledge Management (CIKM)*, 2014.

[PSV17] Ali Pinar, C. Seshadhri, and Vaidyanathan Vishal. ESCAPE: Efficiently counting all 5-vertex subgraphs. In *International Conference on World Wide Web (WWW)*, pages 1431–1440, 2017.

[PSW14] Rasmus Pagh, Morten Stöckel, and David P. Woodruff. Is min-wise hashing optimal for summarizing set intersection? In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '14, page 109–120, New York, NY, USA, 2014. Association for Computing Machinery.

[PT11] Rasmus Pagh and Charalampos E. Tsourakakis. Colorful triangle counting and a mapreduce implementation. *CoRR*, abs/1103.6073, 2011.

[PT12] Rasmus Pagh and Charalampos E Tsourakakis. Colorful triangle counting and a mapreduce implementation. *Information Processing Letters*, 112(7):277–281, 2012.

[PW20] Pál András Papp and Roger Wattenhofer. On the hardness of red-blue pebble games. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 419–429, 2020.

[PY90] Christos H. Papadimitriou and Mihalis Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM journal on computing*, 19(2):322–328, 1990.

[RA15] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[RAK18] Ryan A. Rossi, Nesreen K. Ahmed, and Eunyee Koh. Higher-order network representation learning. In *Companion Proceedings of the The Web Conference 2018*, WWW '18, page 3–4, Republic and Canton of Geneva, CHE, 2018. International World Wide Web Conferences Steering Committee.

[RD17] Ling Ren and Srinivas Devadas. Bandwidth hard functions for ASIC resistance. In *TCC (1)*, volume 10677 of *Lecture Notes in Computer Science*, pages 466–492. Springer, 2017.

[Ree92] Bruce A. Reed. Finding approximate separators and computing tree width quickly. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 221–228, 1992.

[RS86] Neil Robertson and Paul D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986.

[RS87] Victor J Rayward-Smith. UET scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics*, 18(1):55–71, 1987.

[RS95] Neil Robertson and Paul D. Seymour. Graph minors. XIII. The disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 63(1):65–110, 1995.

[RSV15] Jaikumar Radhakrishnan, Saswata Shannigrahi, and Rakesh Venkat. Hypergraph two-coloring in the streaming model. *CoRR*, abs/1512.04188, 2015.

[RSZ12] Desh Ranjan, John E. Savage, and Mohammad Zubair. Upper and lower I/O bounds for pebbling r-pyramids. *J. Discrete Algorithms*, 14:2–12, 2012.

[RT94] John H. Reif and Stephen R. Tate. Dynamic parallel tree contraction (extended abstract). In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 114–121, 1994.

[RVW16] Tim Roughgarden, Sergei Vassilvitskii, and Joshua R. Wang. Shuffles and circuits: (on lower bounds for modern parallel computation). In *SPAA*, pages 1–12, 2016.

[Saa96] Yousef Saad. Ilum: A multi-elimination ilu preconditioner for general sparse matrices. *SIAM Journal on Scientific Computing*, 17(4):830–847, 1996.

[SB14] Julian Shun and Guy E Blelloch. Phase-concurrent hash tables for determinism. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 96–107, 2014.

[SBF+12] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2012.

[SBLP19] Yihan Sun, Guy E Blelloch, Wan Shen Lim, and Andrew Pavlo. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *PVLDB*, 13(2):211–225, 2019.

[Sch93] Marco Schneider. Self-stabilization. *ACM Computing Surveys (CSUR)*, 25(1):45–67, 1993.

[Sch13] Marcus Schaefer. The graph crossing number and its variants: A survey. *The Electronic Journal of Combinatorics*, 1000:21–22, 2013.

[SCS20] Bintao Sun, T.-H. Hubert Chan, and Mauro Sozio. Fully dynamic approximate k-core decomposition in hypergraphs. *ACM Trans. Knowl. Discov. Data*, 14(4), May 2020.

[SDS20] Jessica Shi, Laxman Dhulipala, and Julian Shun. Parallel clique counting and peeling algorithms, 2020.

[Sei83] Stephen B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269–287, 1983.

[SERF18] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. Patterns and anomalies in k-cores of real-world graphs with applications. *Knowledge and Information Systems*, 54(3):677–710, 2018.

[Set75] Ravi Sethi. Complete register allocation problems. *SIAM J. Comput.*, 4(3):226–248, 1975.

[SGJS+13] Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. Streaming algorithms for k-core decomposition. *Proc. VLDB Endow.*, 6(6):433–444, April 2013.

[SGJS+16] Ahmet Erdem Sariyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. Incremental k-core decomposition: algorithms and evaluation. *The VLDB Journal*, 25(3):425–447, 2016.

[SLA+17] Shaden Smith, Xing Liu, Nesreen K Ahmed, Ancy Sarah Tom, Fabrizio Petrini, and George Karypis. Truss decomposition on shared-memory parallel systems. In *IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2017.

[Sol16] Shay Solomon. Fully dynamic maximal matching in constant update time. In *FOCS*, pages 325–334, 2016.

[SP18] Ahmet Erdem Sariyüce and Ali Pinar. Peeling bipartite networks for dense subgraph discovery. In *ACM International Conference on Web Search and Data Mining (WSDM)*, pages 504–512, 2018.

[SPK13] Comandur Seshadhri, Ali Pinar, and Tamara G Kolda. Triadic measures on graphs: The power of wedge sampling. In *Proceedings of the 2013 SIAM International Conference on Data Mining*, pages 10–18. SIAM, 2013.

[SS79a] John E. Savage and Sowmitri Swamy. Space-time tradeoffs for oblivious integer multiplication. In Hermann A. Maurer, editor, *Automata, Languages and Programming*, pages 498–504, Berlin, Heidelberg, 1979. Springer Berlin Heidelberg.

[SS79b] Sowmitri Swamy and John E. Savage. Space-time tradeoffs for linear recursion. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 135–142, New York, NY, USA, 1979. ACM.

[SS20] Jessica Shi and Julian Shun. Parallel algorithms for butterfly computations. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APoCS)*, pages 16–30, 2020.

[SSP18] Ahmet Erdem Sariyüce, C. Seshadhri, and Ali Pinar. Local algorithms for hierarchical dense subgraph discovery. *Proc. VLDB Endow.*, 12(1):43–56, 2018.

[SSPc17] Ahmet Erdem Sariyüce, C. Seshadhri, Ali Pinar, and Ümit V. Çatalyürek. Nucleus decompositions for identifying hierarchy of dense subgraphs. *ACM Trans. Web*, 11(3):16:1–16:27, July 2017.

[SSS95] Jeanette P Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff–hoeffding bounds for applications with limited independence. *SIAM Journal on Discrete Mathematics*, 8(2):223–250, 1995.

[ST15] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *2015 IEEE 31st International Conference on Data Engineering*, pages 149–160, April 2015.

[STTW18] Natcha Simsiri, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Work-efficient parallel union-find. *Concurrency and Computation: Practice and Experience*, 30(4), 2018.

[SV11] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *International Conference on World Wide Web (WWW)*, pages 607–614, 2011.

[SW05] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Experimental and Efficient Algorithms*, pages 606–609. 2005.

[SW20a] Saurabh Sawlani and Junxing Wang. Near-optimal fully dynamic densest subgraph. In *Proceedings of the ACM SIGACT Symposium on Theory of Computing*, pages 181–193, 2020.

[SW20b] Shay Solomon and Nicole Wein. Improved dynamic graph coloring. *ACM Trans. Algorithms*, 16(3), June 2020.

[TDB19] Thomas Tseng, Laxman Dhulipala, and Guy E. Blelloch. Batch-parallel Euler tour trees. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 92–106, 2019.

[THCG18] A. Tripathy, F. Hohman, D. H. Chau, and O. Green. Scalable k-core decomposition for static graphs using a dynamic graph data structure. In *IEEE International Conference on Big Data (Big Data)*, pages 1134–1141, 2018.

[TKMF09] Charalampos E Tsourakakis, U Kang, Gary L Miller, and Christos Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 837–846, 2009.

[Tom81] Martin Tompa. Corrigendum: Time-space tradeoffs for computing functions, using connectivity properties of their circuits. *J. Comput. Syst. Sci.*, 23(1):106, 1981.

[TPM17] Charalampos E. Tsourakakis, Jakub Pachocki, and Michael Mitzenmacher. Scalable motif-aware graph clustering. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 1451–1460, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee.

[Tso15] Charalampos Tsourakakis. The k-clique densest subgraph problem. In *Proceedings of the 24th international conference on world wide web*, pages 1122–1132, 2015.

[Val77] Leslie G. Valiant. Graph-theoretic arguments in low-level complexity. In *Mathematical Foundations of Computer Science 1977, 6th Symposium, Tatranska Lomnica, Czechoslovakia, September 5-9, 1977, Proceedings*, pages 162–176, 1977.

[Vas09] Virginia Vassilevska. Efficient algorithms for clique problems. *Information Processing Letters*, 109(4):254–257, 2009.

[Vis08] Uzi Vishkin. Thinking in parallel: Some basic data-parallel algorithms and techniques. 2008.

[VS10] Maarten Van Steen. Graph theory and complex networks. *An introduction*, 144, 2010.

[WAPL14] Yu Wu, Per Austrin, Toniann Pitassi, and David Liu. Inapproximability of treewidth and related problems. *J. Artif. Intell. Res.*, 49:569–600, 2014.

[WC81] Mark N Wegman and J Lawrence Carter. New hash functions and their use in authentication and set equality. *Journal of computer and system sciences*, 22(3):265–279, 1981.

[WC12] Jia Wang and James Cheng. Truss decomposition in massive networks. *Proc. VLDB Endow.*, 5(9):812–823, May 2012.

[WGS20] Yiqiu Wang, Yan Gu, and Julian Shun. Theoretically-efficient and practical parallel DBSCAN. In *ACM SIGMOD*, 2020.

[Whi12] Tom White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[Wil12] Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *ACM Symposium on Theory of Computing Conference*, pages 887–898, 2012.

[Win07] Peter Winkler. *Mathematical Mind-Benders.* A K Peters/CRC Press, Wellesley, MA, 1st ed.. edition, 2007.

[WLQ+20] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. Efficient bitruss decomposition for large-scale bipartite graphs. In *IEEE International Conference on Data Engineering (ICDE)*, pages 661–672, 2020.

[WQZ+19] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. Yu. I/O efficient core graph decomposition: Application to degeneracy ordering. *IEEE Transactions on Knowledge & Data Engineering*, 31(01):75–90, jan 2019.

[WS98] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440, 1998.

[WVZT90] Kyu-Young Whang, Brad T. Vander-Zanden, and Howard M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Syst.*, 15(2):208–229, June 1990.

[Xia16] Mingyu Xiao. A parameterized algorithm for bounded-degree vertex deletion. In *Proceedings of the 22nd International Conference on Computing and Combinatorics*, pages 79–91, 2016.

[Yan78] Mihalis Yannakakis. Node-and edge-deletion NP-complete problems. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 253–264, 1978.

[YBL18] Hao Yin, Austin R Benson, and Jure Leskovec. Higher-order clustering in networks. *Physical Review E*, 97(5):052306, 2018.

[YK11] Jin-Hyun Yoon and Sung-Ryul Kim. Improved sampling for triangle counting with mapreduce. In *International Conference on Hybrid Information Technology*, pages 685–689. Springer, 2011.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[Zou16] Zhaonian Zou. Bitruss decomposition of bipartite graphs. In *Database Systems for Advanced Applications*, pages 218–233, 2016.

[Zuc07] David Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. *Theory of Computing*, 3(1):103–128, 2007. Announced at STOC'06.

[ZY19] Yikai Zhang and Jeffrey Xu Yu. Unboundedness and efficiency of truss maintenance in evolving graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1024–1041, 2019.

[ZYZQ17] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin. A fast order-based approach for core maintenance. In *IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 337–348, 2017.