Staged Self-Assembly and Polyomino Context-Free Grammars

A dissertation submitted by

Andrew Winslow

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

Tufts University

February 2014

Advisors: Diane Souvaine, Erik Demaine

Abstract

Self-assembly is the programmatic design of particle systems that coalesce into complex superstructures according to simple, local rules. Such systems of square tiles attaching edgewise are capable of Turing-universal computation and efficient construction of arbitrary shapes. In this work we study the *staged tile assembly model* in which sequences of reactions are used to encode complex shapes using fewer tile species than otherwise possible. Our main contribution is the analysis of these systems by comparison with context-free grammars (CFGs), a standard model in formal language theory. Considering goal shapes as strings (one dimension) and labeled polyominoes (two dimensions), we perform a fine-grained comparison between the smallest CFGs and staged assembly systems (SASs) with the same language.

In one dimension, we show that SASs and CFGs are equivalent for a small number of tile types, and that SASs can be significantly smaller when more tile types are permitted. In two dimensions, we give a new definition of generalized context-free grammars we call *polyomino context-free grammars (PCFGs)* that simultaneously retains multiple aspects of CFGs not found in existing definitions. We then show a one-sided equivalence: for every PCFG deriving some labeled polyomino, there is a SAS of similar size that assembles an equivalent assembly. In the other direction, we give increasingly restrictive classes of shapes for which the smallest SAS assembling the shape is exponentially smaller than any PCFG deriving the shape.

Dedicated to my mom.

Acknowledgments

They were two, they spoke little, wanted to repeat the calm, take time to nap. They had a clear idea of what they wanted, what arrangement, which room, what sound, and showed impassive faces to our attempts to impose our rhythm. We used to run the Soirées de Poche, pushing and motivating artists, warming up the audience, writing an outline. But then, no, we were not the masters.

– Vincent Moon, *Soirée de Poche*

Funding for my research was provided by NSF grants CCF-0830734, CBET-0941538, and a Dean's Fellowship from Tufts University. Additional travel funding was provided by Caltech and the Molecular Programming Project for a visit during Summer 2012, and a number of government and industry sources to attend the Canadian Conference on Computational Geometry, Fall Workshop on Computational Geometry, and the International Conference on DNA Computing and Molecular Programming. I also thank Tufts University and the Department of Computer Science for providing accommodations throughout my time in Medford, and the Bellairs Research Institute of McGill University for accommodations in Barbados.

Even more than funding sources, I am deeply grateful to a large number of researchers and collaborators who provided me with mentorship, encouragement, and assistance. Foremost, my advisors Diane Souvaine and Erik

Demaine, along with my other committee members Lenore Cowen, Benjamin Hescott, and Hyunmin Yi. The Tufts-MIT Computational Geometry meeting and its many participants were instrumental in shaping my research experience, and I thank the many participants, including Martin Demaine, Sarah Eisenstat, David Charlton, Zachary Abel, Anna Lubiw, and too many others to list. I also thank the students and visiting faculty at Tufts University for many productive collaborations, including Gill Barequet, Godfried Toussaint, Greg Aloupis, Sarah Cannon, Mashhood Ishaque, and Eli Fox-Epstein. Not least of all, I thank tile self-assembly friends and collaborators who invited me into their circle: Damien Woods, Matthew Patitz, Robert Schweller, and Scott Summers – may the tiles forever assemble in their favor.

They say that the best art comes from misery. I thank Christina Birch for helping to make the first two years of graduate school possibly the most productive time in my life. I also specifically thank Erik and Marty Demaine for providing crucial support at several key times during my study, and being a source of much needed inspiration and perspective. Finally, I thank Megan Strait for being a core part of my life and time at Tufts, for playing an irreplaceable role in the days that produced the work contained herein, and for showing me what real perseverance is.

Contents

List of Figures

1

Introduction

1.1 Tile assembly

The original *abstract tile assembly model* or *aTAM* was introduced by Erik Winfree [Win98] in his Ph.D. thesis in the mid-1990s. In this model, a set of square particles (called *tiles*) are mixed in a container (called a *bin*) and can attach along edges with matching bonding sites or *glues* (each with an integer *strength*) to form a multi-tile *assembly*. The design of a *tile set* consists of specifying the glue on each of the four sides of each *tile type*. A *mixing* takes place by combining an infinite number of copies of each tile type in a bin. Once multi-tile assemblies have formed, they may attach to other assemblies (of which tiles are a special case) to form ever-larger *superassemblies*. Two *subassemblies* can attach to form a superassembly if there exists a translation of the assemblies such that the matching glues on coincident edges of the two assemblies have a total strength that exceeds the fixed *temperature* of the

For a complete exploration of the many models and results in tile assembly, see the surveys of Doty [Dot12], Patitz [Pat12], and Woods [Woo13].

mixing. The set of assemblies assembled that are not subassemblies of some other assembly are the *products* of the mixing. The specification of the tile set and temperature defines an *assembly system*, and the behavior of the system follows from these specifications. An example of an assembly system is seen in Figure 1.1. The system pictured forms a unique 2×2 square assembly, but only one of many possible assembly processes is shown. For the most part, only systems producing a unique assembly are considered.

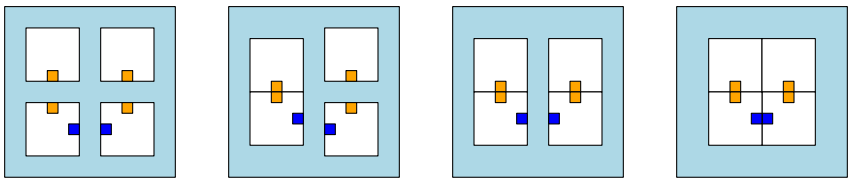


Figure 1.1: A temperature-1 tile assembly system with three tile types and two glues. Each color denotes a unique glue, and each glue forms a bond of strength 1.

As it turns out, the formation of multi-tile subassemblies that form superassemblies is a behavior *not* permitted by the original aTAM defined by Winfree. In Winfree's model, the tile system also has a special *seed tile* to which tiles attach to form a growing *seed assembly*, and the seed assembly is the only multi-tile assembly permitted to form (see Figure 1.2). Assembly in these systems has a more restricted form of crystalline-like growth, and this restriction (as we will describe) is both useful and limiting. The first work in seedless growth was the Ph.D. work of Rebecca Schulman [BSRW09, Sch98, SW05], who studied the theoretical and experimental problem of "spurious nucleation" where multi-tile assemblies form without the seed tile. Her work was focused on preventing growth not originating at the seed tile (or a preassembled seed assembly) by designing tile sets where assemblies not containing the seed assembly are kinetically unfavorable.

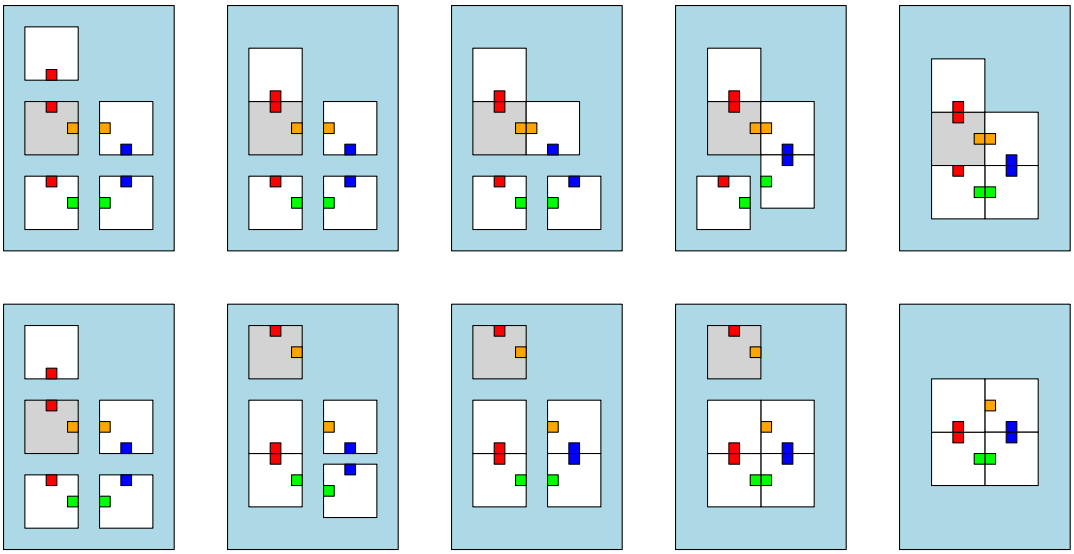


Figure 1.2: Top: a seeded (aTAM) temperature-1 tile assembly system that grows from a seed tile (gray) to produce a 5-tile assembly. Bottom: the same tile set without seeded growth produces a second, 4-tile assembly.

1.2 Two-handed assembly

After the study of kinetic seedless assembly was initiated by Schulman et al., the study of seedless tile assembly in the kinetics-less setting (similar to the aTAM) began. Various notions of seedless assembly restricted to linear assemblies [Adl00, ACG+01, CGR12] were studied first, and the "seedless aTAM", referred to as the *hierarchical aTAM* [CD12, PLS12], *polyomino tile assembly model (pTAM)* [Luh09, Luh10], or *two-handed assembly model (2HAM)* [ABD+10, CDD+13, DDF+08a, DPR+10] have since been studied more extensively. From now on we refer to this model as the 2HAM for consistency with prior work most closely linked to our results.

The 2HAM and aTAM lie on opposite ends of the parameterized *q-tile model* of Aggarwal et al. [ACG+05] in which assemblies of at most q (a fixed integer) may form and attach to the seed assembly. The *two-handed planar geometric tile assembly model (2GAM)* of Fu et al. [FPSS12] is a modification of the 2HAM in which individual tiles are replaced by preassembled assemblies

called *geometric tiles*. Geometric tiles attach using the same rules, but utilize their geometry and the constraint that they must meet and attach by planar motion to achieve more efficient assembly.

Because the 2HAM permits seedless assembly behavior seemingly disallowed in the aTAM, comparative study of the two models has been done to better understand their relationship. Chen and Doty [CD12] considered assembly time of the two models: the expected time spent for an assembly to form, where the growth rate is proportional to the number of possible ways two assemblies can attach. They achieved assembly of an infinite class of $n \times m$ rectangles with $n > m$ in time $O(n^{4/5} \log n)$ in the 2HAM, beating the $\Omega(n)$ bound for aTAM systems.

Recently it was shown by Doty et al. [DLP$^+$10, DLP$^+$12] that the aTAM at temperature 2 (where assemblies may need to bind using multiple glues at once) is *intrinsically universal*. An intrinsically universal model has a single tile set that, given an appropriate seed assembly, simulates any system in the model. In the case of the aTAM, this implies that temperatures above 2 have no fundamentally distinct behaviors. For the 2HAM, Demaine et al. [DPR$^+$13] showed that for each temperature $\tau \geq 2$, there is no system that is intrinsically universal for the model at temperature $\tau + 1$. That is, the 2HAM gains new power and behavior as the temperature is increased, while the aTAM does not. Cannon et al. [CDD$^+$13] showed that any aTAM system at $\tau \geq 2$ can be simulated by a 2HAM system at τ. Combining this knowledge with the prior results, the 2HAM at $\tau \geq 3$ or more exhibit behaviors that cannot be simulated by *any* aTAM system at *any* temperature.

Because tile systems are geometric in nature and the applications often involve the construction of a specified structure rather than a "Yes" or "No" answer, merely proving bounds on the *computational* power of various mod-

els is not sufficiently informative to understand how models relate. Recent results, including those reviewed here, have demonstrated that the ability to simulate more accurately captures the power of a model, a realization that occurred earlier in the field of cellular automata [Oll08]. A recent survey by Woods [Woo13] explores intrinsic universality and inter-model simulation results thoroughly, and includes a hierarchy diagram of all known tile assembly models and their ability to simulate each other.

1.3 Staged self-assembly

One the challenging aspects of implementing tile assembly models with real molecules is the engineering of glues (chemical bonding sites) that form sufficiently strong bonds with matching glues, and sufficiently weak bonding with non-matching glues. For k distinct glues, $\binom{k}{2} = k(k-1)/2$ interactions between these glues must be considered and engineered. Also, since systems are entirely specified by their tile sets, only a constant number of systems exist for any fixed k and the complexity of the shapes assembled by these systems also have constant, bounded complexity.

Various tile assembly models with other aspects that can encode arbitrary amounts of information have been proposed, including control of concentrations of each tile type [KS08, Dot10], adjustment of temperature during mixing [KS06, Sum12], and tile shape [DDF$^+$12]. The *staged tile assembly model* introduced by Demaine et al. [DDF$^+$08a, DDF$^+$08b] uses a sequence of 2HAM mixings to encode shape, where the product assemblies of the previous mixing are used (in place of single tiles) as reagents of the next mixing. The graph describing the mixings can grow without bound, and is used to encode arbitrarily complex assemblies using a constant number of glues.

For instance, Rothemund and Winfree [RW00] prove that all $n \times n$ squares can be produced with an aTAM system of $O(\log n)$ tile types while Demaine et al. [DDF+08a] show that assembly is possible by a staged assembly system with a constant number of tiles and $O(\log n)$ mixings. For measuring the size of an aTAM system, the number of tile types is used, whereas the size of a staged assembly system is determined by the number of mixings. So under these distinct measures of size, assembling an $n \times n$ square is possible using systems of size $O(\log n)$ in both models. Since these measures roughly correspond to the information content of aTAM and staged systems, a common information-theoretic argument can be used to show that a lower bound of $\Omega(\log n / \log \log n)$ exists for the size of any system assembling an $n \times n$ square.

1.4 Combinatorial optimization

Shortly after Rothemund and Winfree achieved $O(\log n)$ systems for $n \times n$ squares, Adleman, Cheng, Goel, and Huang [ACGH01] found an optimal family of $O(\log n / \log \log n)$-sized tile sets. With the case of squares completely solved, these four authors, along with Kempe, de Espanés, and Rothemund [ACG+02] studied the general *minimum tile set problem*: find the smallest temperature-2 tile set producing a unique assembly with a given input shape. They showed that the minimum tile set problem is NP-complete, but is solvable in polynomial-time for trees (shapes with no 2×2 subshape) and squares. If the restriction of producing a unique assembly is relaxed to allow possibly multiple assemblies that share the input shape, then Bryans et al. [BCD+11] showed that this problem is $\mathrm{NP}^{\mathrm{NP}}$-complete, harder under standard complexity-theoretic assumptions. The additional complexity introduced by allowing a system to produce multiple assemblies has kept most study in

tile assembly restricted to systems producing a unique assembly.

Note that finding the smallest tile set uniquely assembling an $n \times n$ assembly follows trivially from the $O(\log n)$ upper bound on the solution tile set size, as there are only a polynomial number of possible solution tile sets of this size. This led to the study of a number of variants of the minimum tile set problem. Chen, Doty, and Seki [CDS11] showed that the minimum tile set problem extended to arbitrary temperatures remains polynomial-time solvable.

Ma and Lombardi [ML08] introduced the *patterned self-assembly tile set synthesis (PATS)* problem in which the goal is to find a small tile set that produces an $n \times m$ rectangular assembly in which each tile has an assigned *label* or *color*. They also modifed the assembly process to start with an L-shaped seed of $n + m - 1$ tiles forming the west and south boundary of the shape, and require that all glues have strength 1 and the system has temperature 2. These restrictions enforce that each tile bonds to the seed assembly using both west and south edges. Despite being a heavily-constrained model of tile assembly, the problem remains interesting and difficult. Czeizler and Popa [CP12] were the first to make significant headway[1], proving that the problem is NP-complete if the number of distinct labels in the pattern is allowed to grow unbounded with the pattern dimensions n and m. Seki [Sek13] showed that the problem remains NP-complete if the number of labels is fixed and at most 60, and Kari, Kopecki, and Seki [KKS13] have shown that if upper limits are placed on the number of tile types with each label, then the problem is NP-complete for patterns with only 3 labels.

[1] See the discussion in [CP12] for an account of the history of the PATS problem.

1.5 Our work

We study the *smallest staged assembly system (SAS) problem*: given a labeled shape, find the smallest staged assembly system that produces this shape. The problem lies at the center of several areas of interest: seedless and 2HAM models, combinatorial optimization, assembly of labeled shapes, and alternate methods of information encoding. We approach the smallest SAS problem by drawing parallels between staged assembly systems and *context-free grammars (CFGs)*, a natural model of structured languages (sets of strings) developed by Noam Chomsky [Cho56, Cho59]. Though understanding staged self-assembly is the primary goal of this work, context-free grammars serve as a tool, a reference, and an object of study.

In Chapter 2, we introduce context-free grammars and the *smallest grammar problem*: given a string s, find the smallest CFG whose language is $\{s\}$. As shown in Chapter 3, the smallest CFG problem is closely linked to the smallest SAS problem. We briefly review the proof by Lehman et al. [Leh02] that the smallest grammar problem is NP-hard, and extend this result to well-known restricted classes of *normal form* context-free grammars.

In Chapter 3, we formally define staged assembly systems and give several results on the smallest SAS problem restricted to one-dimensional assemblies of labeled $n \times 1$ polyominoes. We first show that the smallest SAS problem is NP-hard using a reduction reminiscent of the one used by Lehman et al. [Leh02] for the smallest grammar problem. Then we compare smallest grammars and SASs for strings, where the string is *derived* by the CFG and found on an assembly *produced* by the SAS. We show that if the number of the glues on the SAS is constant, then the smallest CFG and SAS for any string are equivalent in size and structure. This result combined with prior work on the smallest grammar problem yields additional results on approximation algorithms for

the smallest SAS problem. In the case that the number of glues is allowed to grow with the size of the input assembly, we show that the equivalence no longer holds and that the smallest SAS may by significantly smaller. Finally, we examine whether a similar idea can be used to construct large unlabeled assemblies with SASs more efficiently than with CFGs and show that the answer is "No".

Before extending the ideas of Chapter 3 to general two-dimensional shapes, Chapter 4 is used to examine the state of context-free grammars in two dimensions. We review the existing generalizations of context-free grammars from strings to polyominoes and show that each fails to maintain at least one desirable property of context-free grammars used in the comparison of CFGs and SASs. As a solution, we introduce a new generalization of CFGs called *polyomino context-free grammmars (PCFGs)* that retains the necessary properties of CFGs, and we give a collection of results related to the smallest PCFG problem.

In Chapter 5 we compare PCFGs and SASs in full two-dimensional glory. We start by showing that the smallest SAS problem is in the polynomial hierarchy, and explain why achieving an NP-completeness result appears difficult. Next, we show that any PCFG deriving a shape can be converted into a slightly larger SAS that assembles the same shape at a scale factor. In the other direction, we show that smallest SASs and PCFGs may differ by a nearly linear factor, even for squares with a constant number of labels and general shapes with only one label. Taken together, these results prove that only a one-sided equivalence between PCFGs and SASs is possible: a small PCFG implies a small SAS, but not vice versa.

2

Context-Free Grammars

Context-free grammars were first developed by Noam Chomsky in the 1950s [Cho56, Cho59] as a formal model of languages (sets of strings), intended to capture some of the interesting structure of natural language. Context-free grammars lie in the Chomsky hierarchy of formal language models, along with finite automata, context-sensitive grammars, and Turing machines. The expressive power of context-free grammars strikes a balance between the capability to describe interesting structure and the tractability of key problems on grammars and their languages. Commonly studied problems for context-free grammars include deciding whether a string belongs to a given language (parsing), listing the set of strings in a language (enumeration), and deciding whether a language is context-free.

In this chapter, we consider the *smallest grammar problem*: given a string s, find the smallest context-free grammar whose language is $\{s\}$. Heuristic algorithms for the smallest grammar problem have existed for several decades, but the first breakthrough in understanding the complexity of the problem occurred in the early 2000s with the Ph.D. thesis work of Eric Lehman [CLL+02, CLL+05, Leh02, Las02] who showed, with his coauthors, that the problem

is NP-complete to approximate within some small constant factor of optimal. They also created a polynomial-time algorithm that produces a grammar within a $O(\log n)$-factor of optimal, where n is the length of s. Curiously, two other $O(\log n)$-approximation algorithms were developed by Rytter [Ryt02] and Sakamoto [Sak05] around the same time.[1] The algorithms of Lehman et al., Rytter, and Sakamoto use a common tool of the *LZ77 decomposition* of the input string. LZ77 is an abbreviation of "Lempel-Ziv 1977" from the compression algorithm of Ziv and Lempel [ZL77]. Recently, Jeż [Jeż13] has given a new $O(\log n)$-approximation using a simplified approach similar to Sakamoto but without using the LZ77 decomposition.

In this work, we begin by defining context-free grammars and reviewing the proof of Lehman et al. [CLL+05] that the smallest grammar problem is NP-hard to approximate. Then we consider the smallest grammar problem for seven commonly used context-free grammar *normal forms*. Normal form grammars have extra restrictions on how the grammar is structured, often making them easier to analyze. We show that the smallest grammar problem restricted to these normal forms remains NP-hard to approximate within factors that are *larger* than the best known for unrestricted grammars achieved by Lehman et al., even when using the same inapproximability results by Berman and Karpinski [BK98] in the analysis. Such results are unexpected, as the restricted grammars have a smaller search space, and thus appear to be easier problems.

[1] See the discussion in Jeż [Jeż13] for concerns about the approximation factor of the algorithm by Sakamoto [Sak05].

2.1 Definitions

A string s is a sequence of symbols $a_1 a_2 \dots a_n$. We call the set of symbols in s the *alphabet* of s, denoted $\Sigma(s)$, with $\Sigma(s) = \{a_1, a_2, \dots, a_n\}$. The *size* of s is n, the number of symbols in the sequence, and is denoted $|s|$. Similarly, the size of the alphabet is denoted $|\Sigma(s)|$.

A *context-free grammar*, abbreviated *CFG* or simply *grammar*, is a 4-tuple $(\Sigma, \Gamma, S, \Delta)$. The set Σ is a set of *terminal symbols* and Γ is a set of *non-terminal symbols*. The symbol $S \in \Gamma$ is a special *start symbol*. Finally, the set Δ consists of *production rules*, each of the form $A \to B_1 B_2 \dots B_j$, with $A \in \Gamma$ and $B_i \in \Sigma \cup \Gamma$. These rules form the building blocks of the *derivation* process described next.

A string s can be derived by starting with S, the start symbol of G, and repeatedly replacing a non-terminal symbol with a sequence of non-terminal and terminal symbols. The set of valid replacements is Δ, the production rules of G, where a non-terminal symbol A can be replaced with a sequence of symbols $B_1 B_2 \dots B_j$ if there exists a rule $A \to B_1 B_2 \dots B_j$ in Δ.

The set of all strings that can be derived using a grammar G is called the *language of* G, denoted $L(G)$. A language is *singleton* if it contains a single string, and a grammar is singleton if its language is singleton.

For some grammars it is the case that a non-terminal symbol $N_1 \in \Gamma$ appears on the left-hand side (l.h.s.) of multiple production rules in Δ. In other words, two rules $A \to B_1 B_2 \dots B_j$ and $A \to C_1 C_2 \dots C_k$ are found in Δ. If a grammar G is not singleton then two such rules must exist, as the derivation process starting with S must be *non-deterministic* to yield multiple strings. However, a non-deterministic grammar may still be singleton as seen by the grammar $G = (\{a\}, \{S, A, B\}, S, \{S \to A, S \to B, A \to a, B \to a\})$. On the other hand, every deterministic grammar *is* singleton, as the derivation

process always carries out the same set of replacements.

This chapter considers representing strings and other sequences as grammars. Given a string s, any grammar G with $L(G) = \{s\}$ is a representation of s as a grammar, including $(\Sigma(s), \{S\}, S, \{S \to s\})$. We define the size of a grammar $|G|$ as the total number of right-hand side (r.h.s.) symbols appearing in all production rules, including repetition. Clearly $|G| \geq |\Sigma|$, $|G| \geq |\Gamma|$, and $|G| \geq |\Delta|$, so $|G|$ is a good approximation of the "bigness" of the entire grammar.

2.2 The smallest grammar problem

In Chapter 3 we show the equivalence between an optimization problem in self-assembly and finding the smallest grammar whose language is a given string, the well-studied *smallest grammar problem*:

Problem 2.2.1 (Smallest CFG)**.** *Given a string s, find the smallest CFG G such that $L(G) = \{s\}$.*

For optimization problems, including the smallest grammar problem, approximation algorithms are often developed. We define an algorithm to be a *c-approximation* if it always returns a solution with size at most $c \cdot OPT$ (for minimization problems) or at least $c \cdot OPT$ (for maximization problems), where OPT is the size of the optimal solution. Alternatively, a problem is *c-approximable* if it admits a c-approximation. A problem is said to be *c-inapproximable* if a c-approximation exists only if P = NP. The shorthand *$(c-\varepsilon)$-inapproximable* is used to indicate that the problem is inapproximable for any value less than c, i.e. for all $\varepsilon > 0$.

Eric Lehman and coauthors [CLL+05] showed that the smallest grammar problem is NP-complete and 8579/8578-inapproximable. Here we review this

proof, and in Sections 2.3 through 2.6 use similar reductions to show that the smallest grammar problem remains inapproximable when the grammars are restricted to belong to one of several normal forms. We start with an easy proof showing that the smallest grammar problem is in NP:

Lemma 2.2.2. *The smallest grammar is in* NP*.*

Proof. Every string s admits a grammar of size $|s|$: the grammar $(\Sigma(s), \{S\}, S, \{S \to s\})$ has language $\{s\}$ and a single rule with n r.h.s. symbols. The smallest grammar for s is also deterministic, as any non-deterministic grammar with language $\{s\}$ and production rules $A \to B_1 B_2 \dots B_j$ and $A \to C_1 C_2 \dots C_k$ yields a smaller grammar with language $\{s\}$ by removing the second rule. Trivially, the smallest grammar for s contains no rules of the form $A \to \varepsilon$.

Combining these three facts yields a polynomial-time verifier for an input s, k, and G. First, check that $|G| \leq k$, G is deterministic, and G has no rules of the form $A \to \varepsilon$, otherwise reject. Second, perform the (deterministic) derivation of the single string in $L(G)$. If the derivation process ever yields a sequence of terminals and non-terminals of length more than $|s|$, reject. Otherwise, check that the string is s and accept if so, otherwise reject. □

The proof of NP-hardness is, as usual, more difficult. Lehman et al. use a reduction from the minimum vertex cover problem on degree-three graphs, shown to be inapproximable within better than a 145/144-factor by Berman and Karpinski:

Problem 2.2.3 (Degree-k vertex cover)**.** *Given a degree-k graph $G = (V, E)$ find the smallest set of vertices $C \subseteq V$ such that for every edge $(u, v) \in E$, $\{u, v\} \cap C \neq \emptyset$.*

Lemma 2.2.4. *The degree-3 vertex cover problem is* $(145/144-\varepsilon)$*-inapproximable [BK98].*

The reduction encodes the edges of the input graph as a sequence of constant-length substrings. Ample use of unique terminal symbols and repeated substrings are used to rigidly fix the structure of any smallest grammar to consist of three levels: the root node labeled S, a set of non-terminals corresponding to vertices in the cover, and a sequence of terminal symbols equal to the input. A grammar encoding a vertex cover of size k for the input string generated from a graph $G = (V, E)$ has size $15|V| + 3|E| + k$. Since the graph is degree-three, $3/2|V| \geq |E|$ and $k \geq |E|/3$. These facts yield a c-inapproximability result for the smallest grammar problem:

$$\begin{aligned} c &= \frac{15|V| + 3|E| + 145/144k}{15|V| + 3|E| + k} \\ &\geq \frac{15|V| + 3(3/2|V|) + 145/144(1/3|V|)}{15|V| + 3(3/2|V|) + 1/3|V|} \\ &\geq \frac{8569}{8568} \end{aligned}$$

Theorem 2.2.5. *The smallest grammar problem is* $(8569/8568-\varepsilon)$*-inapproximable. [CLL+05]*

Next, we give a slight improvement to this result by replacing the inapproximability result of Berman and Karpinski on the degree-3 vertex cover problem with a similar result on the degree-4 vertex cover problem.

Lemma 2.2.6. *The degree-4 vertex cover problem is* $(79/78-\varepsilon)$*-inapproximable [BK98].*

Then we proceed with the same reduction and analysis as before, but with $2|V| \geq |E|$ and $k \geq |E|/4$:

$$
\begin{aligned}
c &= \frac{15|V| + 3|E| + 79/78k}{15|V| + 3|E| + k} \\
&\geq \frac{15|V| + 3(2|V|) + 79/78(1/4|V|)}{15|V| + 3(2|V|) + 1/4|V|} \\
&\geq \frac{6631}{6630}
\end{aligned}
$$

Theorem 2.2.7. *The smallest grammar problem is* $(6631/6630-\varepsilon)$*-inapproximable.*

Surprisingly, this improvement does not appear to be previously known.

The reduction

Let the symbol # denote a unique symbol at every use, and let $\prod$ denote concatenation. Then given a degree-three graph $G = (V, E)$, the string computed by the reduction is:

$$
s = \prod_{v_i \in V} (0v_i \# v_i 0 \#)^2 \prod_{v_i \in V} (0v_i 0 \#) \prod_{(v_i, v_j) \in E} (0v_i 0 v_j 0 \#)
$$

Consider decomposition the string into three parts, so $s = s_1 s_2 s_3$:

$$
s_1 = \prod_{v_i \in V} (0v_i \# v_i 0 \#) \qquad s_2 = \prod_{v_i \in V} (0v_i 0 \#) \qquad s_3 = \prod_{(v_i, v_j) \in E} (0v_i 0 v_j 0 \#)
$$

Some smallest grammar for s has rules $L_i \to 0v_i$ and $R_i \to v_i 0$ for each v_i (appearing in s_1). Such a grammar also has a rule $B_i \to L_i 0$ or $B_j \to L_j 0$ for each string $0v_i 0v_j 0$ in s_3. As a result, every edge incident to v_i (appearing as a string in s_3) can then be encoded as a rule $E_i \to B_i R_j$ or $E_i \to L_i B_j$. A smallest grammar then selects the fewest vertices $v_i \in V$ to have rules

$B_i \to L_i 0$, equivalent to selecting the smallest set of vertices for a vertex cover.

Recall that the size of a grammar is the total number of symbols on the right-hand sides of all rules. Then converting a rule $A \to bcd$ into a pair of rules $A \to Ed$, $E \to bc$ increases the size of the grammar by 1, so starting with a single rule $S \to s$, additional rules result in a smaller grammar only if the rules derive substrings that repeat. Because a unique # symbol appears at least once in every substring of s with length 5 or greater, no strings of length greater than 5 repeat. As a result, *every* smallest grammar consists of a large start rule and a collection of short rules $L_i \to 0v_i$, $R_i \to v_i 0$, $B_i \to L_i 0$, and $E_i \to B_i R_i$ or $L_i B_i$.

2.3 Grammar normal forms

Grammar normal forms are restricted classes of context-free grammars that obey constraints on the form of production rules, but remain capable of encoding any language encoded by a general context-free grammar. Normal forms have found a variety of applications, primarily in simplifying algorithms involving context-free grammars. For instance, the problem of deciding whether an input string belongs to the language of an input grammar, also known as parsing, is a critical step of compiling computer programs.

The popular CYK algorithm [CS70] for parsing requires that the input grammar be given in Chomsky normal form, where the r.h.s. of each production rule is either two non-terminal symbols, or a single terminal symbol. Only recently Lange and Leiß [LL09] were able to show that a simple modification to the CYK algorithm enables its use on unrestricted grammars without an asymptotic performance penalty.

Name	Rule format in [LL09]	Minimal rule format	Inapproximability
CNF^-	$A \to BC \mid a$	$A \to BC \mid a$	$3667/3666 - \varepsilon$
CNF	$A \to BC \mid a, S \to \varepsilon$		
CNF_ε	$A \to BC \mid a \mid \varepsilon$		
C2F	$A \to BC \mid B \mid a, S \to \varepsilon$		
S2F	$A \to \alpha$ where $\lvert\alpha\rvert = 2$	$A \to \alpha$ where $\lvert\alpha\rvert = 2$	3353/3352
2NF	$A \to \alpha$ where $\lvert\alpha\rvert \leq 2$		
2LF	$A \to uBvCw \mid uBv \mid u$	$A \to uBvCw \mid u$	$8191/8190 - \varepsilon$

Table 2.1: Common grammar normal forms and their inapproximability ratios proved in Section 2.3. The minimal rule format contains all rule formats possibly found in some smallest grammar, where symbols A, B, C are non-terminals, S is the start symbol, a is a terminal symbol, α is a string of terminals and non-terminals, and u, v, w are strings of terminals.

In this section we consider the smallest grammar problem for seven normal forms, seen in Table 2.1 (partially reproduced from [LL09]). If only smallest grammars for singleton languages of positive-length strings are considered, the seven normal forms reduce to three classes normal forms. We consider three canonical normal forms, one from each class: CNF^- (Chomsky normal form with no empty string), 2NF (2-normal form), and 2LF (bilinear form). For each canonical normal form, and thus all seven normal forms, we prove that finding the smallest grammar in the form is NP-complete and inapproximable within some small constant factor.

It is clear that for any given string, the smallest normal form grammar is at least the size of the smallest unrestricted grammar. However, this does not imply that the smallest grammar problem for normal forms is NP-hard. Consider the smallest grammar for reduction string s under the restriction that every rule has at most two right-hand-side symbols. In this case, a rule $B_i \to L_i 0$ must be created for every $v_i \in V$, so every vertex is in the cover "for free" and the reduction fails.

2.4 Chomsky normal form (CNF)

Given a graph $G = (V, E)$, create the string $s = s_1 s_2$, with:

$$s_1 = \prod_{v_i \in V} (0 v_i v_i 0) \qquad s_2 = \prod_{(v_i, v_j) \in E} (0 v_i 0 v_j 0 \#)$$

Consider the various ways in which each substring $0v_iv_i0$ can be derived. Using a single global rule $P \to 00$ enables the use of a minimal 2-rule derivation $V_i \to PB_i$, $B_i \to v_iv_i$ for each v_i, as v_iv_i appears nowhere else in s. On the other hand, the 3-rule encoding $V_i \to L_iR_i$, $L_i \to 0v_i$, $R_i \to v_i0$ is also possible, and contains rules L_i and R_i reusable in s_2.

Now consider rules for s_2. Some smallest grammar must produce a non-terminal B_i deriving $0v_i0$ or $0v_j0$ for each substring $0v_i0v_j0$, i.e. must select a cover vertex v_i or v_j. Moreover, if rules $L_i \to 0v_i$ and $R_i \to v_i0$ were not created to derive a substring $0v_iv_i0$ in s_1, they must be created for a substring $0v_i0v_j0$ of s_2. So some smallest grammar creates them for all vertices v_i. So some smallest grammar uses a rule set $E_i^f \to E_i\#$, and $E_i \to B_iR_j$ or $E_i \to L_iB_j$ for each substring $0v_i0v_j0$.

In total, deriving s_1 uses $3|V| + |V| - 1$ rules, s_2 uses $2|E| + |E| - 1$ rules, and both use k additional B_i rules, where k is the smallest vertex cover for G. An additional set of $|V| + |E| + 1$ rules of the form $A \to a$ are needed for the symbols found in s and a start rule deriving the two non-terminals for s_1 and s_2. Pushing these values through the analysis in [CLL+05], the smallest CNF grammar problem is not approximable within better than a c factor:

$$
\begin{aligned}
c &= \frac{2(4|V| - 1 + 3|E| - 1 + 79/78k) + |V| + |E| + 2}{2(4|V| - 1 + 3|E| - 1 + k) + |V| + |E| + 2} \\
&\geq \frac{9|V| + 7|E| - 2 + 158/78k}{9|V| + 7|E| - 2 + 2k} \\
&\geq \frac{9|V| + 7(2|V|) - 2 + 158/78(1/4|V|))}{9|V| + 7(2|V|) - 2 + 2(1/4|V|)} \\
&\geq \frac{3667}{3666}
\end{aligned}
$$

Theorem 2.4.1. *The smallest CNF grammar problem is* $(3667/3666 - \varepsilon)$*-inapproximable.*

2.5 2-normal form (2NF)

Rather than do a reduction from scratch, we prove a small set of results that tightly link the smallest CNF and smallest 2NF grammar problems.

Lemma 2.5.1. *For any string s there exists a 2NF grammar G deriving s if and only if there exists a CNF grammar G' deriving s with $|G| + |\Sigma(s)| = |G'|$.*

Proof. First, start with a CNF grammar G' and create a 2NF grammar G deriving s by eliminating rules of the form $A \to a$ by replacing every occurrence of A with a. The new grammar has two r.h.s. symbols in every rule (which may be terminal or non-terminal) and so is a 2NF grammar with size $|G'| - |\Sigma(s)|$.

Second, start with a 2NF grammar G and add a set of rules $A_i \to a_i$ for each $a_i \in \Sigma(s)$. For each occurrence of any a_i on the r.h.s. of a rule, replace a_i with A_i. This new grammar is a CNF grammar and has size $|G| + \Sigma(s)$. □

Note that this lemma immediately implies that the smallest 2NF grammar problem is NP-hard.

Lemma 2.5.2. *Let $0 < d \leq 1$ and $\Sigma(s)$ be the set of symbols in s. If the smallest CNF grammar problem is $(1 + c')$-inapproximable for an infinite set of strings s_i such that $|\Sigma(s_i)| \geq d|s_i|$ for all s_i, then the smallest 2NF grammar problem is $(1 + c' + c'd/2)$-inapproximable.*

Proof. For a given input string s_i, call the smallest 2NF grammar G and smallest CNF grammar G', with $|G| + |\Sigma(s_i)| = |G'|$. Then computing a CNF$^-$ grammar of size at most $c|G'| = c|G| + c|\Sigma(s_i)|$ for all s_i is NP-hard. So by Lemma 2.5.1, computing a S2F grammar of size at most $c|G|+c|\Sigma(s_i)|-|\Sigma(s_i)| = c|G| + (c-1)|\Sigma(s_i)|$ is NP-hard. By assumption, $|\Sigma(s_i)| \geq d|s_i|$. Moreover, $|s_i| \geq |G|/2$ and thus $|\Sigma(s_i)| \geq d|G|/2$. So computing a 2NF grammar of size at most $c|G|+(c-1)|\Sigma(s_i)| \geq c|G|+(c-1)d|G|/2$ is NP-hard. Letting $c = (1 + c')$, the smallest 2NF grammar problem is $(1 + c' + c'd/2)$-inapproximable. □

Theorem 2.5.3. *The smallest 2NF grammar problem is* $(3353/3352)$*-inapproximable.*

Proof. For each string s used in Section 2.4, $|\Sigma(s)| = |V| + |E| + 1$ and $|s| = 4|V| + 6|E|$. So $|\Sigma(s)|/(|s|) = (|V| + |E| + 1)/(4|V| + 6|E|) \geq (|V| + 2|V| + 1)/(4|V| + 6(2|V|)) \geq 3/16$. Invoking Lemma 2.5.2 with $c' = \frac{1}{3666}$ and $d = \frac{3}{16}$ yields an inapproximability ratio of $1 + 1/3666 + 1/3666 \cdot 3/16 \cdot 1/2 - \varepsilon > 3353/3352$ for the smallest 2NF grammar problem. □

2.6 Bilinear form (2LF)

This normal form shares properties of both 2NF (only two non-terminals are allowed per right-hand side) and general CFGs (an arbitrary number of terminal symbols are allowed per right-hand side). Here we use a string s with many duplicated substrings, effectively forcing each such substring to be derived from a distinct non-terminal symbol. Consider $s = s_1 s_2 s_3$, with:

$$s_1 = \prod_{v_i \in V} (0v_i\overline{v_i}v_i0\overline{v_i})^2 \quad s_2 = \prod_{v_i \in V} (0v_i0c_i)^2 \quad s_3 = \prod_{(v_i,v_j) \in E} (0v_i0v_j0e_{ij})^2$$

Each substring $(0v_i\overline{v_i}v_i0\overline{v_i})^2$ has identical halves that should reuse an identical set of rules. By a similar argument to that used in previous reductions, a set of rules $L_i \to 0v_i$, $R_i \to v_i0$, $V_i^h \to L_i\overline{v_i}R_i\overline{v_i}$, $V_i \to V_i^hV_i^h$ are found in some smallest 2LF grammar for s. Similarly, each substring $(0v_i0v_j0e_{ij})^2$ should have both occurrences of its duplicated half $(0v_i0v_j0e_{ij})$ derived using a common non-terminal symbol, and the entire string derived with a unique non-terminal symbol. Deriving each half costs 3 (e.g. $E_i^h \to B_iR_je_{ij}$) or 4 r.h.s. symbols (e.g. $E_i^h \to L_iL_j0e_{ij}$).

Finally, each substring $(0v_i0c_i)^2$ also has two duplicate halves that should be derived identically. Each half of the substring can be encoded using 5 ($C_i^h \to L_i0c_i$ and $C_i \to C_i^hC_i^h$) or 6 r.h.s. symbols ($C_i^h \to B_ic_i$ and $B_i \to L_i0$). Because the symbol-cost for using B_i (one) is equal to the symbol-cost of failing to have either vertex covering a particular edge (i.e. not using B_i or B_j in deriving a particular substring $0v_i0v_j0e_{ij}$), some smallest grammar corresponds to a vertex cover. So there is a smallest grammar with the following production rules:

1. For each $(0v_i\overline{v_i}v_i0\overline{v_i})^2$ in s_1: $L_i \to 0v_i$, $R_i \to v_i0$, $V_i^h \to L_i\overline{v_i}R_i\overline{v_i}$, $V_i \to V_i^hV_i^h$.

2. For each $(0v_i0c_i)^2$ in s_2: $C_i \to B_ic_iB_ic_i$, $B_i \to L_i0$ if v_i in cover, $C_i \to C_i^hC_i^h$, $C_i^h \to L_i0c_i$ otherwise.

3. For each $(0v_i0v_j0e_{ij})^2$ in s_3: $E_i \to E_i^hE_i^h$ with $E_i^h \to B_iR_je_{ij}$ or $E_i^h \to L_jB_je_{ij}$.

Moreover, deriving the sequence of V_i, C_i, and E_i symbols requires an additional $2|V|+|E|-2$ r.h.s. symbols in a set of branching production rules. So if there is a 2LF grammar of size $(10|V|+5|V|+k+5|E|)+(2|V|+|E|-2)$ with singleton language $\{s\}$ then there is a vertex cover of size k. So the smallest 2LF grammar problem is inapproximable within better than a c factor:

$$\begin{aligned} c &= \frac{17|V|+6|E|+79/78k-2}{17|V|+6|E|+k-2} \\ &\geq \frac{17|V|+6(2|V|)+79/78(1/4|V|)-2}{17|V|+6(2|V|)+(1/4|V|)-2} \\ &\geq \frac{26|V|+79/78(1/4|V|)}{26|V|+1/4|V|} \\ &\geq \frac{8191}{8190} \end{aligned}$$

Theorem 2.6.1. *The smallest 2LF grammar problem is* $(8191/8190-\varepsilon)$*-inapproximable.*

2.7 APX-hardness of the smallest grammar problem

The complexity class APX consists of the set of optimization problems that admit constant-factor approximation algorithms. A subset of these problems form the class PTAS of problems that admit *polynomial-time approximation schemes*: an infinite sequence of approximation algorithms with arbitrarily-good approximation ratios. A problem is APX-hard if a polynomial-time approximation scheme for the problem implies such schemes for all problems in APX.

Sakamoto et al. [SKS04] and Maruyama et al. [MMS06] cite Lehman and

shelat [Las02] as showing the smallest grammar problem to be APX-hard. Lehman and shelat do not explicitly mention the APX-hardness result in their work, nor does Lehman explicitly mention such a result in his thesis [Leh02] or his other coauthored papers containing the result [CLL+02, CLL+05]. On the other hand, the explicit inapproximability bound given by Lehman et al. immediately implies the APX-hardness of the problem: a PTAS (polynomial-time approximation scheme) for the smallest grammar problem implies P = NP and thus polynomial-time algorithms solving all problems in APX exactly!

3

One-Dimensional Staged Self-Assembly

In Chapter 2 we reviewed work on *context-free grammars (CFGs)*, a model of formal languages, and in particular the *smallest grammar problem*, finding the smallest context-free grammar whose language is a single given string. With context-free grammars in mind, we are able to begin the analysis of staged assembly systems, abbreviated *SASs*, and in particular the *smallest SAS problem*.

A theme in Chapters 3 and 5 is examining the correspondence between CFGs and SASs, and the smallest CFG and smallest SAS problems. At first, it may be unclear how these two models may be compared, as SASs manipulate and describe assemblies, not strings. However, extending the staged assembly model to use systems of *labeled* tiles, as done in the aTAM and reviewed in Section 1.4, yields a model in which assemblies are arrangements of labeled tiles and can be described by the arrangements of labels. In the special case of

Portions of this chapter have been published as [DEIW11] and [DEIW12] with coauthors Erik Demaine, Sarah Eisenstat, and Mashhood Ishaque.

one-dimensional assemblies consisting of a single row of tiles, the labels form a string.

Since purity is of importance in laboratory synthesis of compounds, we define the set of assemblies *produced* by a SAS to be those assemblies appearing as the lone product of some mixing in the system. Extracting the *label string* of these assemblies yields the "language" of the SAS. In practice, label strings may correspond to particle functionalities or other properties, and an assembly of a given label string may carry out a diverse set of behaviors or complex pipeline of interactions.

Beyond the simple correspondence of labeled one-dimensional assemblies and strings, context-free grammars are chosen for their hierarchical derivation of larger strings by composing smaller strings, akin to mixing smaller assemblies to produce larger assemblies. The direct correspondence of CFG production rules and SAS mixings is explored in Sections 3.3 and 3.4, and is shown to be surprisingly intricate. In Sections 3.3 we show that CFGs with language $\{s\}$ and SASs producing an assembly with label string s are equivalent up to some constant factor in size, under the assumption that each mixing of the SAS produces a single assembly (which we call *singleton staged assembly systems (SSASs)*. This result justifies the comparison of CFGs and SASs, and yields a number of corollaries about the smallest SAS problem by applying known results from prior work on the smallest CFG problem. These include a $O(\log n)$-approximation algorithm for the smallest SSAS problem, and strong evidence that the problem is $o(\log n / \log\log n)$-inapproximable.

After this result, we compare CFGs and SASs with multi-product mixings (Section 3.4) and show that for some strings, the smallest CFG is significantly larger than the smallest SAS. We show that the ratio of the smallest CFG over the smallest SAS, which we call *separation*, can be $\Omega(\sqrt{n/\log n})$. On

the other hand, if the number of glues k used in the system is parameterized, then the separation is $\Omega(k)$ and $O(k^2)$. Since the number of glues is small and constant in practice[1], this implies that the results achieved for SSASs apply to practical systems as well.

Finally, in Section 3.5 we answer a question posed by Robert Schweller [Sch13] about whether additional glues can be used to assemble larger assemblies more quickly, and give a nearly complete negative answer. This negative result, when combined with the separation results of Section 3.4, shows that additional glues are only useful for efficient production of assemblies with complex label strings, and do not enable for efficient assembly in general.

3.1 Definitions

An instance of the staged tile assembly model is called a *staged assembly system* or *system*, abbreviated *SAS*. A SAS $\mathcal{S} = (T, G, \tau, M, B)$ is specified by five parts: a *tile set* T of square *tiles*, a *glue function* $G : \Sigma(G)^2 \to \{0, 1, \ldots, \tau\}$, a *temperature* $\tau \in \mathbb{N}$, a directed acyclic *mix graph* $M = (V, E)$, and a *start bin function* $B : V_L \to T$ from the *root vertices* $V_L \subseteq V$ of M with no incoming edges.

Each tile $t \in T$ is specified by a 5-tuple (l, g_n, g_e, g_s, g_w) consisting of a label l taken from an alphabet $\Sigma(T)$ (denoted $l(t)$) and a set of four non-negative integers in $\Sigma(G) = \{0, 1, \ldots, k\}$ specifying the *glues* on the sides of t with normal vectors $\langle 0, 1\rangle$ (north), $\langle 1, 0\rangle$ (east), $\langle 0, -1\rangle$ (south), and $\langle -1, 0\rangle$ (west), respectively, denoted $g_{\vec{u}}(t)$. In this work we only consider glue functions with the constraints that if $G(g_i, g_j) > 0$ then $g_i = g_j$, and $G(0, 0) = 0$. A *configuration* is a partial function $C : \mathbb{Z}^2 \to T$ mapping locations on the integer lattice to tiles. Any two locations $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ in the domain of C (de-

[1]This is one of the motivations for the staged self-assembly model.

noted $\mathrm{dom}(C)$) are *adjacent* if $||p_2 - p_1|| = 1$ and the *bond strength* between any pair of adjacent tiles $C(p_1)$ and $C(p_2)$ is $G(g_{p_2-p_1}(C(p_1)), g_{p_1-p_2}(C(p_2)))$. A configuration is a *$\tau$-stable assembly* or an *assembly at temperature τ* if $\mathrm{dom}(C)$ is connected on the lattice and, for any partition of $\mathrm{dom}(C)$ into two subconfigurations C_1, C_2, the sum of the bond strengths between tiles at pairs of locations $p_1 \in \mathrm{dom}(C_1)$, $p_2 \in \mathrm{dom}(C_2)$ is at least τ. Any pair of configurations C_1, C_2 are equivalent if there exists a vector $\vec{v} = \langle x, y \rangle$ such that $\mathrm{dom}(C_1) = \{p + \vec{v} \mid p \in \mathrm{dom}(C_2)\}$ and $C_1(p) = C_2(p + \vec{v})$ for all $p \in \mathrm{dom}(C_1)$. The *size* of an assembly A is $|\mathrm{dom}(A)|$, the the number of lattice locations that map to tiles in T.

Two τ-stable assemblies A_1, A_2 are said to *assemble* into a *superassembly* A_3 if there exists a translation vector $\vec{v} = \langle x, y \rangle$ such that $\mathrm{dom}(A_1) \cap \{p + \vec{v} \mid p \in A_2\} = \emptyset$ and A_3 defined by the partial functions A_1 and A_2' with $A_2'(p) = A_2(p + \vec{v})$ is a τ-stable assembly. Similarly, an assembly A_1 is a *subassembly* of A_2, denoted $A_1 \subseteq A_2$, if there exists a translation vector $\vec{v} = \langle x, y \rangle$ such that $\mathrm{dom}(A_1) \subseteq \{p + \vec{v} \mid p \in A_2\}$.

Each vertex of the mix graph M describes a *two-handed assembly process*. This process starts with a set of τ-stable *reagent assemblies* or *reagents* I. The set of *assemblable assemblies* Q is defined recursively as $I \subseteq Q$, and for any pair of assemblies $A_1, A_2 \in Q$ with superassembly A_3, $A_3 \in Q$. Finally, the set of *product assemblies* or *products* $P \subseteq Q$ is the set of assemblies A such that for any assembly A', no superassembly of A and A' is in Q.

The mix graph $M = (V, E)$ of $\mathcal{S}$ defines a set of two-handed assembly processes (called *mixings*) for the non-root vertices of M (called *bins*). The reagents of the bin v is the union of all products of mixings at vertices v' with $(v', v) \in E$. The start bin function B defines the lone single-tile product of each mixing at a root bin. The system $\mathcal{S}$ is said to *produce* an assembly

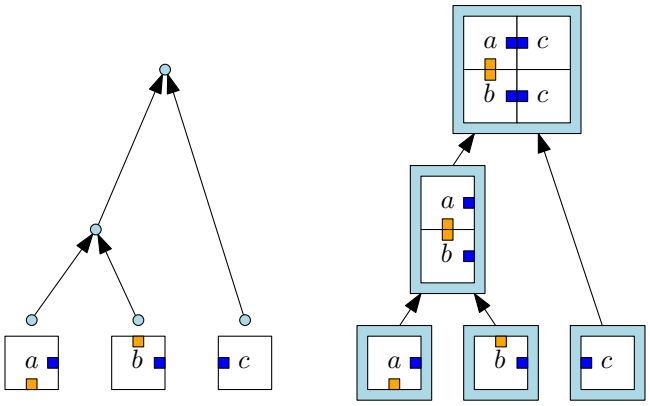


Figure 3.1: A $\tau = 1$ self-assembly system (SAS) defined by its mix graph and tile set (left), and the products of the system (right). Tile sides lacking a glue denote the presence of glue 0, which does not form positive-strength bonds.

A if some mixing of $\mathcal{S}$ has a single product, A. We define the *size* of $\mathcal{S}$ or alternatively, the amount of *work* done by $\mathcal{S}$, to be $|E|$ and denote it by $|\mathcal{S}|$. If every mixing in $\mathcal{S}$ has a single product, then $\mathcal{S}$ is a *singleton self-assembly system (SSAS)*.

1D notation In Chapter 5 we consider general staged assembly systems producing assemblies on the integer lattice. However, in this chapter we only consider one-dimensional staged self-assembly: assemblies limited to a single horizontal row of tiles (see Figure 3.2).

Because these assemblies never attach vertically, their north and south glues can all be replaced with the null glue with no change in the assemblies produced. As 5-tuples, tiles in one-dimensional systems have the form $(l, 0, 0, g_e, g_w)$. In the remainder of the chapter we use the in-line shorthand $g_w[l]g_e$ to represent such a tile and denote multi-tile assemblies as $g_w[l_1 l_2 \ldots l_n]g_e$, where $l_1, l_2, \ldots, l_n$ is the sequence of tile labels as they appear from west to east in the assembly. We call this sequence the *label string* of the assembly. Because distinct assemblies can have identical west/glue pair and label string, we are careful to avoid using this information to define assemblies and limit

its use to *describing* assemblies.

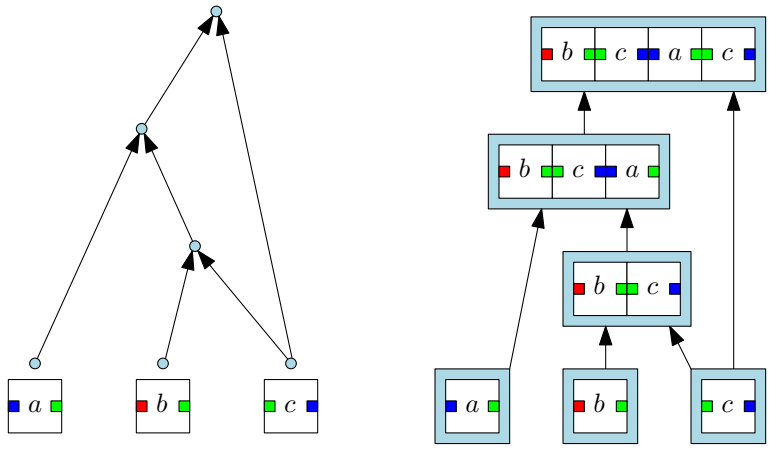


Figure 3.2: A one-dimensional $\tau = 1$ self-assembly system in which only east and west glues are non-null.

3.2 The Smallest SAS Problem

As with the smallest grammar problem for CFGs, the smallest SAS problem is an essential optimization problem for staged assembly systems, and is a primary problem of study in this work.

Problem 3.2.1 (Smallest SAS)**.** *Given an input string s, find the smallest SAS with at most k glues that produces an assembly with label string s.*

Before giving the NP-completeness proof, we prove some helpful results about SASs.

Lemma 3.2.2. *If a SAS mixing v has no infinite products, then each product of v contains at most one copy of each reagent.*

Proof. Let A be a product with two copies of a common subassembly $g_1[s_1]g_2$. Then A has a sequence of subassemblies $g_1[s_1]g_2$, $g_2[s_2]g_1$, $g_1[s_1]g_2$ and $g_1[s_1s_2]g_1$ is an assemblable assembly of the mixing. So $g_1[s_1s_2s_1s_2 \ldots s_1s_2]g_1$, an infinite assembly, is a product of the mixing. □

Corollary 3.2.3. *Let $\mathcal{S}$ be a smallest SAS with a bin v with two products $A_1 = g_1[s_1]g_2$ and $A_2 = g_3[s_2]g_4$. Then either $g_1 \neq g_3$ or $g_2 \neq g_4$.*

Lemma 3.2.4. *Let $\mathcal{S}$ be a smallest SAS using k glues. Then each bin in $\mathcal{S}$ has at most $\lfloor k/2 \rfloor \cdot \lceil k/2 \rceil$ products.*

Proof. Each glue appears exclusively on east or west sides of products, otherwise two products can combine. Without loss of generality, let the first $i \leq k$ glues appear on the east side of products, and the remainder of the glues appear on the west side of products. By Corollary 3.2.3, each glue pair is found on at most one product. So the number of products is $i \cdot (k - i)$ and is maximized when i and $k - i$ are as close as possible and integer. This occurs when $i = \lfloor k/2 \rfloor$ and thus $k - i = \lceil k/2 \rceil$. □

Lemma 3.2.5. *Let $\mathcal{S}$ be a smallest SAS for an assembly A. Then every product of every bin in $\mathcal{S}$ is a subassembly of A and $\mathcal{S}$ has a single leaf bin with product set $\{A\}$.*

Proof. By contradiction, assume $\mathcal{S}$ has a bin v with product A' not a subassembly of A. Then some product of every descendant of v must contain A', since A' is either a product or was combined with other assemblies to form a product. So any single-product descendant of v cannot produce A, as its product contains A'. Thus, removing v and its descendants results in a second SAS $\mathcal{S}'$ that also produces A, with $|\mathcal{S}'| < |\mathcal{S}|$, a contradiction. So every product of every bin is a subassembly of A.

Next, assume $\mathcal{S}$ has a leaf bin with a product set other than $\{A\}$ or two or more leaf bins with product sets $\{A\}$. Removing one of the leaf bins while keeping a leaf bin with product set $\{A\}$ yields a second SAS that is smaller and produces A, a contradiction. So $\mathcal{S}$ must have exactly one leaf bin, and this bin has product set $\{A\}$. □

Lemma 3.2.6. *Let $\mathcal{S}$ be a smallest SAS with $k = 3$ glues. Then $\mathcal{S}$ is a SSAS and each non-root bin has two parent bins.*

Proof. Suppose, by contradiction, that $\mathcal{S}$ has a bin v with multiple products $A_1 = g_1[s_1]g_2$ and $A_2 = g_3[s_2]g_4$. Without loss of generality and by Corollary 3.2.3 and the pigeonhole principle, $A_1 = 1[s_1]2$ and $A_2 = 1[s_2]3$.We will show that A_1 and A_2 are products of all descendant bins of v (including the single leaf bin guaranteed by Lemma 3.2.5) and so $\mathcal{S}$ is not a smallest SAS.

First, suppose A_1 and A_2 are reagents of some mixing and one attaches to some other assembly $A_3 = g_5[s_3]g_6$ in the mixing. If A_3 attaches to the west side of either assembly, then $g_6 = 1$ and $g_5 \in \{2, 3\}$. Otherwise, if A_3 attaches to the east side of A_1, then $g_5 = g_2$ and $g_6 \in \{1, 3\}$. So in either case, A_3 forms an infinite assembly and by Lemma 3.2.5 $\mathcal{S}$ is not a smallest SAS. So any mixing with A_1 and A_2 as reagents has A_1 and A_2 as products. Then by induction, all descendant bins of v has A_1 and A_2 as products and $\mathcal{S}$ is not smallest, a contradiction. So $\mathcal{S}$ has no mixing with multiple products and is a SSAS.

Suppose, by contradiction, that $\mathcal{S}$ has a non-root bin v with some number of parent bins other than two. If v has one parent bin, then the product of this parent bin is the product of v, and $\mathcal{S}$ is not smallest by Lemma 3.2.5. If v has three or more parent bins, then the product of v has four glues (east, west, and two interior locations where the three reagents attached) and one repeating glue, so the product is infinite and $\mathcal{S}$ is not smallest by Lemma 3.2.5. So every non-root bin must have exactly two parents. □

Now we show that finding the smallest SAS is NP-complete. Unlike the smallest grammar problem, containment of the smallest SAS problem in NP is not trivial: the number of products of each mixing in a non-deterministically chosen SAS could be exponential. Proving that this is not the case, then

efficiently computing the product sets are the two main steps of the proof.

Lemma 3.2.7. *The smallest SAS problem is in* NP*.*

Proof. For an input label string s, glue count k, and integer SAS size n, non-deterministically select a SAS $\mathcal{S} = (T, G, \tau, M, B)$ with $|\Sigma(G)| < k$, $|S| \leq \min(|s|, n)$, and a single leaf bin, along with an assembly A with label string s. Next, "fill in" the products for each bin in $\mathcal{S}$, starting at the roots. During this process, a bin may be encountered with a product that is not a subassembly of A. Then by Lemma 3.2.5, $\mathcal{S}$ is not a smallest SAS and the machine rejects. So going forward, we may assume that $\mathcal{S}$ has no such bins, an each bin contains a polynomial number ($O(|s|(|s|-1)/2)$) of products, each with polynomial ($O(|s|)$) size.

To compute the set of products of a bin, produce the following graph: create a vertex v_i for each reagent A_i and a directed edge (v_i, v_j) for each pair of reagents A_i, A_j such that the east glue of A_i matches the west glue of A_j. Products are then maximal paths in this graph, starting at vertices with in-degree 0 and ending at vertices with out-degree 0. The set of all such paths can be enumerated by iterative depth-first search starting at each vertex with in-degree 0. Any cycle found in a graph implies that an infinite assembly is formed and that S is not a smallest SAS.

Putting it together, polynomial time is spent on each of a polynomial number of bins to compute the products of the bin. The bins are processed in topological order, starting at the roots. After computing the products of all bins, the single leaf bin's product set is compared to $\{A\}$. If they are equal, accept, otherwise reject. □

Next, we show that the smallest SAS problem is NP-hard, matching the complexity of the smallest CFG problem. We reduce from the k-coloring

problem, a classic NP-hard problem:

Problem 3.2.8 (k-coloring)**.** *Given a graph $G = (V, E)$ and a positive integer k, can the vertices of G be colored such for any two vertices $u, v \in V$ with the same color, $(u, v) \notin E$?*

Karp [Kar72] showed that this problem is NP-hard if k is allowed to grow unbounded. Later, Garey and Johnson [GJ79] showed that the problem remains NP-hard if k is fixed and $k \geq 3$.

Lemma 3.2.9. *The* 3*-coloring problem is* NP*-hard. [GJ79]*

We use an approach reminiscent the NP-hardness proofs in Chapter 2, reducing from a graph problem (in this case k-coloring) by encoding each vertex and edge constraints as a short substring. The label string consists of distinct symbols, with the exception of a pair of substrings $l_i l_i'$ and $r_i r_i'$ for each vertex v_i, which appear in multiple constraint substrings. The vertex colors are encoded as a glue assignment of the east glues of the assemblies with label strings $l_i l_i'$ and the west glues of the assemblies with label strings $r_i r_i'$. One assembly for each $l_i l_i'$ and $r_i r_i'$ substring is possible if and only if a valid 3-coloring is possible, where each vertex has the same color across all incident edges and all pairs of adjacent vertices have different colors.

Lemma 3.2.10. *The smallest SAS problem is* NP*-hard.*

Proof. Let $G = (V, E)$ be an input graph to the 3-coloring problem. We set $k = 3$ (the number of glues permitted in the SAS) and convert G to an input label string $s = s_1 s_2$:

$$s_1 = \prod_{v_i \in V} \#^2 l_i l_i' r_i r_i' \#^2 \# \qquad s_2 = \prod_{(v_i, v_j) \in E} \#^2 l_i l_i' \# r_j r_j' \#^2 \#$$

By Lemma 3.2.6, any smallest SAS for s is a SSAS since $k = 3$. All symbols in s are unique, save for the repeating $l_i l_i'$ and $r_i r_i'$ substrings. So any subassembly appears only once in the final product assembly except for subassemblies with label substrings l_i, l_i', r_i, r_i', $l_i l_i'$, or $r_i r_i'$ for some i.

Suppose that a bin v has two child bins v_1 and v_2. Let d be the first common descendant bin of v_1 and v_2 in the mix graph. If the product of v has a label string not repeated in s, then any two parents of d must have the same product: the product of d. So the SSAS must not be smallest. So any bin with two or more child bins must have a product with a repeating label substring.

Consider creating a solution SSAS using only a single subassembly for each label string $l_i l_i'$ and $r_i r_i'$. Because this assembly is used to produce an assembly with label string containing s_1, the east glue of $[l_i l_i']$ and the west glue of $[r_i r_i']$ must be the same. Similarly, since these assemblies are used to produce an assembly with label string containing s_2, the east glue of $[l_i l_i']$ must be different than the west glue of $[r_j r_j']$ for every adjacent vertex pair $(v_i, v_j) \in E$.

The pair of # symbols bookending each gadget allow neighboring gadgets to be attached regardless of the glues used for each $[l_i l_i']$ and $[r_i r_i']$ assembly. We use the following convention: given an assignment (from $\{1, 2, 3\}$) of g_1 for $g_2[l_i l_i']g_1$ or $g_1[r_i r_i']g_2$, let $g_2 = \min(\{1, 2, 3\} - \{g_1\})$. Given such glue assignments for the pair of $[l_i l_i']$ and $[r_i r_i']$ assemblies used in a gadget with label string s_g, Table 3.1 gives sets of #-labeled tiles and smallest mix graphs for assembling $1[s_g]2$. The final assembly with label string s can be assembled by growing a large assembly eastward by alternating successive gadget assemblies and the assembly $2[\#]1$. So if G has a valid 3-coloring, constructing a solution SSAS with a single subassembly for each such label string ($2|V|$ assemblies total) is possible.

$[l_i l_i']$	$[r_i r_i']$	Mix graph and [#] glue assignments
Gadgets in s_1: $\#^2 l_i l_i' r_i r_i' \#^2$		
$2[l_i l_i']1$	$1[r_j r_j']2$	((1[#]3 ((3[#]2 $2[l_i l_i']1$) $1[r_i r_i']2$)) 2[#]3) 3[#]2
$1[l_i l_i']2$	$2[r_j r_j']1$	(1[#]3 (3[#]1 $1[l_i l_i']2$)) (2[$r_i r_i'$]1 1[#]3)) 3[#]2
$1[l_i l_i']3$	$3[r_j r_j']1$	(1[#]2 (2[#]1 $1[l_i l_i']3$)) (3[$r_i r_i'$]1 (1[#]3 3[#]2))
Gadgets in s_2: $\#^2 l_i l_i' \# r_i r_i' \#^2$		
$2[l_i l_i']1$	$2[r_j r_j']1$	1[#]3 ((((3[#]2 $2[l_i l_i']1$) 1[#]2) $2[r_i r_i']1$) (1[#]3 3[#]2))
$2[l_i l_i']1$	$3[r_j r_j']1$	((1[#]3 3[#]2) ($2[l_i l_i']1$ 1[#]3)) (3[$r_j r_j'$]1 (1[#]3 3[#]2))
$1[l_i l_i']2$	$1[r_j r_j']2$	1[#]3 ((((3[#]1 $1[l_i l_i']2$) 2[#]1) $1[r_j r_j']2$) 2[#]1) 1[#]2
$1[l_i l_i']2$	$3[r_j r_j']1$	((1[#]3 (3[#]1 $1[l_i l_i']2$)) ((2[#]3 3[$r_j r_j'$]1) 1[#]3)) 3[#]2
$1[l_i l_i']3$	$1[r_j r_j']2$	(1[#]2 (2[#]1 $1[l_i l_i']3$)) (((3[#]1 $1[r_j r_j']2$) 2[#]1) 1[#]2)
$1[l_i l_i']3$	$2[r_j r_j']1$	(1[#]2 (2[#]1 $1[l_i l_i']3$)) ((3[#]2 $2[r_j r_j']1$) (1[#]3 3[#]2))

Table 3.1: Glue assignments for #-labeled assemblies and mix graphs to assemble gadgets in s. Assemblies are mixed in the order specified by the nested parentheses.

If G has no valid 3-coloring, then any solution SSAS must have at least one pair of bins with products whose label strings are the same $l_i l_i'$ or $r_i r_i'$ string. Creating such a pair requires four edges (two incoming edges for each bin producing one of the assemblies), while creating one such bin only requires two edges. Moreover, using fewer edges than the previously described construction elsewhere in the mix graph is impossible, since all other symbols are unique and each mixing has only two parents by Lemma 3.2.6.

So the smallest SSAS producing an assembly with label string s generated from a 3-colorable graph with $|V|$ vertices and $|E|$ edges is strictly smaller than the smallest SSAS for a label string s generated from a 3-colorable graph of the same size that is not 3-colorable. So the smallest SAS problem is NP-hard.

□

Theorem 3.2.11. *The smallest SAS problem is* NP*-complete.*

We note the interesting distinction between this hardness proof and those in Chapter 2. Though both use short repeating substrings in the input string to ensure local consistency and constraint satisfaction, the gadgets in Chap-

ter 2 use partitioning of symbols to encode the problem, whereas this reduction uses glue assignment. Though a partitioning-based reduction is likely possible, this reduction demonstrates that the smallest SAS problem is difficult in a way unique to self-assembly: deciding which glues to use. The design of the reduction also has implications for a related problem on singleton staged assembly systems:

Problem 3.2.12 (Smallest SSAS)**.** *Given an input string s, find the smallest SSAS with at most k glues that produces an assembly with label string s.*

Lemma 3.2.6 proves that the smallest SAS and smallest SSAS problems are equivalent for inputs with $k = 3$. Since the reduction used in the proof of Theorem 3.2.10 uses a system with $k = 3$, this implies that the smallest SSAS problem is also NP-complete.

Corollary 3.2.13. *The smallest SSAS problem is* NP*-complete.*

3.3 Relation between the Approximability of CFGs and SSASs

In this section we show that converting between an CFG G in 2NF with language $\{s\}$ and a SSAS instance $\mathcal{S}$ producing an assembly with label string s is possible with only a constant-factor scaling. As a result, any $O(f(n))$-approximation to either the smallest grammar problem or the smallest SSAS problem implies an $O(f(n))$-approximation for both.

3.3.1 Converting CFGs to SSASs

Let G be an CFG in 2NF with language $\{s\}$. We begin by converting each production rule of G' to a SSAS bin. However, a problem occurs if the same

$[s_A]$	$[s_B]$	$[s_C]$
$1[s_A]2$	$1[s_B]3$	$3[s_B]2$
$1[s_A]3$	$1[s_B]2$	$2[s_B]3$
$2[s_A]1$	$2[s_B]3$	$3[s_C]1$
$2[s_A]3$	$2[s_B]1$	$1[s_C]3$
$3[s_A]1$	$3[s_B]2$	$2[s_C]1$
$3[s_A]2$	$3[s_B]1$	$1[s_C]2$

Table 3.2: The set of mixings to produce all necessary glue pair variations for assembly corresponding to non-terminal A in the production $A \to BC$ where A, B, C derive the strings s_A, s_B, and s_C, respectively.

non-terminal appears as a right-hand side symbol in several production rules. Recall that a production in the grammar specifies the left-to-right order in which the right-hand side symbols appear, while the west-to-east order in which assemblies attach is determined by their glues. To produce exactly the product assembly desired in a mixing requires combining reagents with the correct glues.

We resolve this issue using 3 glues and constructing a copy of every assembly for each of the 6 possible east-west glue pairs. Since the grammar is 2NF, at most two assemblies are mixed in each bin and so three glue pairs is enough to uniquely specify the bin's product. Given a production $A \to BC$, we create 6 bins whose products that share a common label string (the string derived from A) and the 6 possible east-west glue pair combinations. The reagents for these bins are the products of a similar set of 6 bins for assemblies with the label strings derived from B and C (see Table 3.2).

Theorem 3.3.1. *For any CFG G in 2NF with language $\{s\}$, a SSAS $\mathcal{S}$ can be constructed from G such that $|\mathcal{S}| \leq 6|G|$.*

Proof. Construct a SSAS $\mathcal{S}$ in the following way. First, create 6 bins for each terminal symbol a in G, one for each east-west glue pair (e.g. $1[a]2$, $1[a]3$, $2[a]1$, etc.). For each production rule $A \to BC$ in G (with B and C possibly

terminal), use the 12 bins constructed for B and C as the parent bins for the 6 bins for A as in Table 3.2. The resulting mix subgraph has 6 bins for A, each containing a single product with a distinct east-west glue pair and the same label string derived by A. So $\mathcal{S}$ contains 6 bins for the start symbol of G, and each bin has a single product with label string s. Since each production rule had size 2 and was converted into 6 mixings with two parent bins each, $|\mathcal{S}| \leq 6|G|$. □

3.3.2 Converting SSASs to CFGs

Let $\mathcal{S}$ be a SSAS constructing an assembly with label string s. Convert the mix graph $M = (V, E)$ of $\mathcal{S}$ to a grammar by creating a terminal symbol for each root vertex in V, a non-terminal symbol for each non-root vertex in V, and a production rule for each non-terminal symbol. For each production rule, the r.h.s. symbols are those corresponding to the children of the vertex in M, and the order of the symbols is determined by the order that the reagent assemblies appear in the product assembly.

Theorem 3.3.2. *For any SSAS $\mathcal{S}$ producing an assembly with label string s, a CFG G with language $\{s\}$ can be constructed from S such that $|G| = |\mathcal{S}|$.*

Proof. The terminal symbols of G are equal to the label strings of their corresponding tiles. Each mixing in $\mathcal{S}$ produces a single assembly with a label string equal to the string derived by the corresponding non-terminal symbol in G, because the production orders the right-hand side symbols in the same order that they combine in $\mathcal{S}$. So the start symbol of G derives a string equal to the label string of the assembly produces in the leaf of the mix graph of $\mathcal{S}$. So G derives s. Each edge of the mix graph of $\mathcal{S}$ causes a right-hand side symbol to appear in a production of G. So $|G| = |\mathcal{S}|$. □

Note that for any string, the smallest 2NF grammar is at most twice the size of the smallest unrestricted grammar for any string. This fact, along with Theorems 3.3.1 and 3.3.2, immediately implies that an approximation algorithm for either problem transfer to the other at a constant-factor loss.

Corollary 3.3.3. *An $O(f(n))$-approximation algorithm for the smallest grammar problem exists if and only if an $O(f(n))$-approximation algorithm for the smallest SSAS problem exists.*

Several $O(\log n)$-approximation algorithms to the smallest grammar problem exist [CLL+05, Ryt02, Sak05, Jeż13], implying that the smallest SSAS problem is also $O(\log n)$-approximable. A barrier to achieving any $o(\log n/\log\log n)$-approximation to the smallest grammar problem also exists. As Lehman pointed out [Leh02], the best-known approximation to the following problem is $O(\log n/\log\log n)$:

Problem 3.3.4 (Generalized shortest addition chain)**.** *Given a set $S = \{n_1, n_2, \ldots, n_m\}$ of positive integers with $N = \max(S)$, find the shortest sequence $1 = a_0 < a_1 < \cdots < a_r = N$ such that $S \subseteq \{a_i \mid 0 \leq i \leq r\}$ and each $a_i = a_j + a_k$ with $j, k < i$.*

The unary version of this problem (where each n_i is specified in unary) is a special case of the smallest grammar problem, as seen by converting an input integer set $S = \{n_1, n_2, \ldots, n_m\}$ for the generalized unary shortest addition chain problem to the input string $s = \#a^{n_1}\#a^{n_2}\#\ldots\#a^{n_m}\#$ for the smallest grammar problem. In 1976, Yao [Yao76] gave an $O(\log N/\log\log N)$-approximation that runs in $O(|S|\log N)$ time, i.e. time polylogarithmic for unary input and polynomial for binary input. Shortly after, in 1981, Downey, Leong, and Sethi [DLS81] showed that the binary version of the problem is NP-hard[2]. Curiously, these results remain the best known for the generalized

[2]Also called *weakly* NP-*hard* in contrast to *strongly* NP-*hard* where the problem is NP-hard for both unary and binary input.

shortest addition chain problem. Improving the gap for this problem is a barrier to understanding the approximability of the smallest grammar problem.

3.4 CFG over SAS Separation

Now we show that SASs, unlike SSASs, are *not* equivalent to CFGs. The proof is constructive: we give a set of strings and describe a set of SAS instances that produce assemblies with these label strings. We then show that any CFG whose language is one of these strings is asymptotically larger than some SAS instance producing the corresponding label string.

It might appear obvious that allowing bin parallelism should allow a reduction in the amount of work needed to construct an assembly. However, using parallelism has two costs that make saving work difficult. First, for any smallest SAS, each bin has at most k^2 products (Lemma 3.2.4). As a result, additional parallelism requires more glues, which in turn requires more root bins, and thus more work. Second, since the goal of an assembly system is to construct a single goal assembly, bins with parallelism must eventually be "collapsed" into a single bin with a single object (otherwise the parallelism was extraneous). Collapsing bins with parallelism involves adding tiles to join the various assemblies together, and since the glues on each assembly are unique, creating and mixing the joining tiles requires additional work proportional to the amount of parallelism in the bin.

To derive an asymptotic bound between SASs and CFGs, we use a special set of strings that can be built by small SASs but require large CFGs. Each string consists of a sequence of *interleavings* of pairs of smaller strings. We note that the construction is limited to odd values of k, so k is assumed to be odd when used as a parameter in generating instances of this construction.

Intuitively, the strings exploit the incompressibility of *interleaved* patterns. Given two strings $A_4 = a_0a_1a_2a_3$ and $B_4 = b_0b_1b_2b_3$, an interleaving of these strings is $a_0b_0a_1b_1a_2b_2a_3b_3$ created by placing the symbols in the second string in order between consecutive symbols of the first string. Moreover, a *shift permutation* or *rotation* of B_4 is any string of the form $B_iB_{i+1}\ldots B_3B_1B_2\ldots B_{i-1}$ for some $0 \leq i \leq 3$.

Rytter [Ryt02] proves that the number of factors produced by a variant of the LZ77 [ZL77] when run on a string is a lower bound for the size of any grammar deriving the string. Because the LZ algorithm scans the string left-to-right and produces a new factor for each substring not previously encountered, every interleaving of A_4 with a shift permutation of B_4 produces an entirely new set of factors. So concatenating all such interleavings produces a number of factors proportional to the length of such a string. The strings S_k described below are, at a high-level, exactly this construction.

3.4.1 A Set of Strings S_k

Let $\textsc{Binary}(i, \ell)$ be the binary representation of i of length ℓ. The following is a function used to double every symbol in a string:

$$\textsc{Double}(b_1b_2b_3\ldots b_n) = b_1b_1b_2b_2b_3b_3\ldots b_nb_n$$

Let $s_1 \circ s_2$ denote the concatenation of string s_1 followed by s_2. We wish to encode a number of distinct "symbols" in binary. To construct a suitably hard-to-compress string, we want to ensure that the beginning and end of each encoded symbol are clearly delineated. To that end, we define the following strings for all values of k and all values of $i < 2k$:

$$A_{k,i} = (01) \circ \textsc{Double}(\textsc{Binary}(i, 1 + \lceil \log k \rceil)) \circ (01)$$

We wish to use these symbols to construct a string with complex structure (so that it is efficiently constructible using a SAS) but minimal repetition (so that it is not efficiently constructible using a CFG). To minimize repetition, we choose a string with the property that no sequential pair of symbols is repeated. We define the following functions, which are permutations for $0 \leq x < k$:

$$\pi_{k,0}(x) = 2x \bmod k \qquad \pi_{k,1}(x) = 2x + 1 \bmod k$$

We use these two simple functions to construct a more complex permutation. Let the bits of $\textsc{Binary}(i, \ell)$ be $b_1, \ldots, b_\ell$. Then:

$$\Pi_{k,\ell,i}(j) = \pi_{k,b_\ell}(\pi_{k,b_{\ell-1}}(\ldots \pi_{k,b_2}(\pi_{k,b_1}(j)) \ldots))$$

Because k is odd, this function is a permutation for $0 \leq j < k$. In addition, as long as $0 \leq i < 2^\ell$, this function has the property that $\Pi_{k,\ell,i}(j) = \left(2^\ell \cdot j + i\right) \bmod k$. That is, for fixed values of k, ℓ, and j, each value of i such that $0 \leq i < k$ will generate a different value of $\Pi_{k,\ell,i}(j)$. This permutation can be used to ensure that no sequential pair of symbols is repeated. To do so, we construct pairs of symbols as follows:

$$C_{k,i,j} = A_{k,j} \circ (01)^{\lceil \log k \rceil} \circ A_{k,k+\Pi_{k,\lceil \log k \rceil,i}(j)} \circ (01)^{\lceil \log k \rceil}$$

We concatenate these pairs to construct $P_{k,i} = C_{k,i,0} \circ C_{k,i,1} \circ \ldots \circ C_{k,i,k-1}$. Note that the length of each $C_{k,i,j}$ is $12 + 8\lceil \log k \rceil$, and therefore the length of each $P_{k,i}$ is $(12 + 8\lceil \log k \rceil) \cdot k$. We concatenate each $P_{k,i}$ to get the string we wish to compress:

$$S_k = 01 \circ P_{k,0} \circ 01 \circ P_{k,1} \circ 01 \circ \ldots \circ 01 \circ P_{k,k-1} \circ 01$$

In the next two subsections we give bounds on compressing S_k using both a CFG and a SAS.

3.4.2 A SAS Upper Bound for S_k

Now we describe a SAS using bin parallelism that produces an assembly with S_k as its label string. The system is broken down into several subsystems described in this section. A diagram of the SAS for S_3 is seen in Figure 3.3.

Constructing $A_{k,i}$ for all $0 \leq i < 2k$

Say that we are given $2k$ glue pairs x_i, y_i, and that we want to assemble $x_i[A_{k,i}]y_i$ for each $0 \leq i < 2k$. Additionally, let g_0, g_1, and g_2 be three additional distinct glues. Let $\ell = 1 + \lceil \log k \rceil$.

For each binary string s of length $\leq \ell$, we construct two bins: I_s and F_s. Let $s = t \circ b$, where $b \in \{0, 1\}$. The bin I_s will produce an assembly with west glue g_0, east glue g_1, and label string $\textsc{Double}(t) \circ b$. The bin F_s will produce an assembly with west glue g_0, east glue g_2, and label string $\textsc{Double}(s)$. The product of I_s will be constructed by mixing the tile $g_2[b]g_1$ with the product of bin F_t. The product of F_s will be constructed by mixing the tile $g_1[b]g_2$ with the product of bin I_s.

To finish this construction, we add the constant-sized reagent assemblies $x_i[01]g_0$ and $g_2[01]y_i$ to the bin $F_{\textsc{Binary}(i,\ell)}$. This ensures that for $0 \leq i < 2k$, the bin $F_{\textsc{Binary}(i,\ell)}$ has a product with the label string $A_{k,i}$. The total number of mixings required for this construction is $\Theta(k)$.

Fixed and Rotating Bins

The fixed bin has the following set of products:

$$1[A_{k,0}]2, 3[A_{k,1}]4, \ldots, (2k-1)[A_{k,k-1}](2k)$$

The rotating bin has the following set of products:

$$2[A_{k,k+0}]3, 4[A_{k,k+1}]5, \ldots, (2k)[A_{k,k+(k-1)}](2k+1)$$

Permutation and Renormalization Bins

Permuting the assemblies in the rotating bin is simulated by attaching *permutation tiles* to the east and west ends of those assemblies. The permutations $\pi_{k,0}$ and $\pi_{k,1}$ are implemented as two sets of $2k$ tiles, each set in a separate *permutation bin*. A third set of $2k$ tiles are put in a *renormalization bins* used to solve a technical issue with the permutation bins.

The permutation bins for $\pi_{k,0}$ have a set of single tiles that replace the primal glues of assembly i ($2i+2$ and $2i+3$) with the dual glues of assembly $\pi_{k,0}(i)$ ($2\pi_{k,0}(i)+2+(2k+1)$ and $2\pi_{k,0}(i)+3+(2k+1)$) for all assemblies $0 \leq i \leq k-1$. The permutation bin for $\pi_{k,1}$ is constructed analogously. Each tile attaches to either the east or west end of the assembly and correspondingly has the primal and dual glues on its east and west sides. The tiles attaching to the east end of the assembly have the label 0; the tiles attaching to the west end of the assembly have the label 1.

The renormalization bin has a pair of single-tile products for changing the dual glues of assembly i ($(2i+2)+(2k+1)$ and $(2i+3)+(2k+1)$) to its primal glues ($(2i+2)$ and $(2i+3)$). The tiles attaching to the east end of each assembly have the label 1; the tiles attaching to the west end of each assembly

have the label 0.

Creating Interleaved Assemblies

The permutation and renormalization bins are applied in a branching manner to produce all permutation sequences of length ℓ. First $\pi_{k,0}$ and $\pi_{k,1}$ are mixed separately with the rotating bin, then $\pi_{k,0}$ and $\pi_{k,1}$ are each mixed separately with the products of both of these bins, etc. After each mixing with a permutation bin, the renormalization bin is mixed with the products. The result is the tree-like subgraph of the mix graph in the center of Figure 3.3 branching upwards. After all permutation sequences are created, the fixed bin is mixed with each, creating bins with single products that have label strings $P_{k,i}$ for all $0 \leq i < k$.

Combining Interleaved Assemblies

The final step is to combine each assembly with label string $P_{k,i}$ into a single assembly. Each assembly is in a separate bin after its production, and has glue 1 on its west side and glue $2k+1$ on its east side. To the assembly with label $P_{k,i}$, the tiles $(2k+2+i)[1]1$ and $(2k+1)[0](2k+3+i)$ are added. Then these assemblies are combined to produce a single long assembly with glue $(2k+2)$ on the west side, and glue $(3k+2)$ on the east.

Theorem 3.4.1. *The SAS described in Section 3.4.2 has size $O(k)$.*

Proof. Break the SAS into the following sections:

1. Creating the fixed and rotating bins.
2. Creating the permutation and renormalization bins.
3. Creating the interleaved assemblies.

4. Combining interleave assemblies.

Item 1 requires $O(k)$ work to create all $A_{k,i}$ and mixing them together. Item 2 requires $O(k)$ work to create three bins, each with a pair of product tiles for each c-buffered element of the rotating bin. Item 3 requires $O(k)$ work: this portion of the mix graph resembles an upside-down tree and contains no more than two leaves per permutation assembly. Item 4 requires $O(1)$ work per assembly (and thus $O(k)$ work total) to add two location-specifying tiles and combine it with the other assemblies into a single bin. In total, k interleave assemblies (one per shift) are created, so $O(k)$ edges are in this portion of the mix graph. Combining interleave assemblies is done by adding at most two tiles to each interleave assembly followed by combining them into a single bin. A constant number of edges exist for each assembly, so $O(k)$ edges exist in this portion of the mix graph. □

3.4.3 A CFG Lower Bound for S_k

We use a lower bound for the smallest CFG previously used by Rytter [Ryt02] to develop an approximation algorithm for the smallest CFG problem. In that work, he defines the *LZ-factorization*[3] of a string s (denoted $LZ(s)$). Then $LZ(s)$ is the decomposition of s into substrings $s_1 s_2 \dots s_m$ by a single left-to-right pass, where $|s_1| = 1$ and for each i in $2 \leq i \leq m$, s_i is the longest prefix of the remaining portion of s that appears in $s_1 s_2 \dots s_{i-1}$.

Lemma 3.4.2. *For an CFG G deriving a string s, $|LZ(s)| \leq |G|$. [Ryt02]*

Note that each factor of the LZ-factorization forms a "non-terminal" deriving the substring with a single "production rule" encoding the location and

[3]The specific factorization used is that of the LZ77 algorithm [ZL77] but *without* self-referencing, as introduced by Farach and Thorup [FT98].

length of the substring. Rytter proves the result by showing that any grammar must use as many non-terminals as there are factors in the LZ-factorization. Now we prove properties about the label strings of the assemblies constructed in Section 3.4.2.

Lemma 3.4.3. *All factors in the LZ-factorization of S_k have size less than $16\lceil \log k \rceil + 26$.*

Proof. Assume by contradiction that the LZ-factorization of S_k contains some factor y of size $\geq 16\lceil \log k \rceil + 26$. Then the factor is long enough that there must be some i, j such that $C_{k,i,j}$ is a substring of y. Let x be the part of the string preceding y. Then by the definition of LZ factorization, y is a substring of x, and therefore $C_{k,i,j}$ is a substring of x.

The string $C_{k,i,j}$ contains the substring $A_{k,j}$. To ensure the correct parity on sequences of symbols, the portion of x where $A_{k,j}$ is found must have been completely generated by some other A_{k,j^*}. Then it must be that $A_{k,j} = A_{k,j^*}$ and $j = j^*$. So the portion of x where $C_{k,i,j}$ is found must have been completely generated by some other $C_{k,i^*,j}$, where $i \neq i^*$. Then $A_{k,k+\Pi_{k,\lceil \log k \rceil,i}(j)} = A_{k,k+\Pi_{k,\lceil \log k \rceil,i^*}(j)}$ and it follows that $k+\Pi_{k,\lceil \log k \rceil,i}(j) = k+\Pi_{k,\lceil \log k \rceil,i^*}(j)$. Therefore, $i = i^*$, which is a contradiction. □

Theorem 3.4.4. *The smallest CFG with language $\{S_k\}$ has size $\Omega(k^2)$.*

Proof. By Lemma 3.4.3, the maximum length of an LZ factor is $16\lceil \log k \rceil + 29$. The sum of the lengths of the LZ factors is equal to $|S_k| = \Theta(k^2 \log k)$. Hence, the number of LZ factors is $\Omega(k^2)$. By Theorem 3.4.2, the size of the smallest grammar must therefore be $\Omega(k^2)$. □

3.4.4 CFG over SAS Separation for S_k

We define *separation* as the maximum ratio of the smallest CFG over the smallest SAS across all label strings. Here we give bounds on separation, using the strings S_k for a lower bound ($\Omega(k)$) and a conversion algorithm from any SAS to a CFG for an upper bound ($O(k^2)$). Recall that k is the number of glues used in the SAS producing an assembly with label string S_k in Section 3.4.3 and $n = |S_k|$.

Theorem 3.4.5. *The separation of CFGs over SASs is $\Omega(k)$.*

Proof. By Theorem 3.4.4, any CFG deriving S_k has size $\Omega(k^2)$. By Theorem 3.4.1, a SAS of size $O(k)$ exists that produces an assembly with label string S_k. So the ratio of the size of any grammar deriving S_k to the size of some SAS instance is $\Omega(k)$. □

Corollary 3.4.6. *The separation of CFGs over SAS is $\Omega(\sqrt{n/\log n})$.*

Proof. The length of S_k is $\Theta(k^2 \log k)$. So $k = \Theta(\sqrt{n/\log n})$. By Theorem 3.4.5, the separation is $\Omega(k)$. So the separation is also $\Omega(\sqrt{n/\log n})$. □

Given that the number of glues is limited in practice, it is natural to consider whether $\Omega(k)$ separation is possible for k glues where $k \ll n$. We show this is possible for $k = \Theta(\log n)$.

Theorem 3.4.7. *The separation of CFGs over SASs with $O(\log n)$ glues is $\Omega(k)$.*

Proof. Define the recursive string $T_{k,t} = 01 \circ T_{k,t-1} \circ 01 \circ T_{k,t-1} \circ 01$, where $T_{k,1} = S_k$, and note that the length of $T_{k,t}$ is $\Theta(2^t|S_k|) = \Theta(2^t k^2 \log k)$. Since $T_{k,k}$ has S_k as a substring, any CFG deriving $T_{k,k}$ has size $\Omega(k^2)$ by Theorem 3.4.4. To construct a SAS to generate this string, we first use the SAS described in

Section 3.4.2 to generate an assembly $g_1[S_k]g_2$. We can then add a constant number of tiles to get two assemblies $g_3[1S_k0]g_5$ and $g_5[1S_k0]g_4$, which when combined produce the assembly $g_3[1S_k01S_k0]dg_4$. We then add two more tiles to construct the assembly $g_1[01S_k01S_k01]a_2$. This process can then be repeated k times. In total $O(k)$ additional work is performed, so the new SAS has size $O(k)$. The length n of the string is $\Theta(2^k k^2 \log k)$, so $k = \Theta(\log n)$. □

We now give an upper bound on the separation by giving an algorithm that converts any SAS using k glues to a CFG with a factor $O(k^2)$ blowup.

Lemma 3.4.8. *Given a SAS $\mathcal{S}$ using k glues and producing an assembly with label string s, a CFG of size $O(k^2|A|)$ with language $\{s\}$ can be constructed.*

Proof. For each bin in $\mathcal{S}$ and product assembly of the bin, construct one bin in the SSAS $\mathcal{S}'$. By Lemma 3.2.4, the number of bins in $\mathcal{S}'$ will be at most k^2 times the number of bins in $\mathcal{S}$.

Now consider what happens when ℓ bins in $\mathcal{S}$ are combined to create a single bin v with several product assemblies. How many edges must we add to $\mathcal{S}'$ to ensure that each product assembly of v is correctly constructed in $\mathcal{S}'$? To determine this, define G to be a directed graph with a node corresponding to each glue and, for each reagent assembly $g_1[s]g_2$, a directed edge from g_1 to g_2. Then each product assembly of v corresponds to a source-sink pair in G, and each possible way to construct that assembly corresponds to a path in G from the source of the assembly to the sink of the assembly.

Say that there exist three glues g_1, g_2, g_3 such that (g_1, g_2) and (g_2, g_3) are edges in G but (g_1, g_3) is not an edge in G. Then we can mix the assembly corresponding to the edge (g_1, g_2) with the assembly corresponding to the edge (g_2, g_3) to get an assembly with glue g_1 to the west and glue g_3 to the east. This is equivalent to adding the edge (g_1, g_3) to G. Each such bin requires

us to add a constant number of nodes and edges to the mix graph of $\mathcal{S}'$, and increases the number of edges in G by one. The graph G can never have more than k^2 edges, so repeated mixings of this type add a total of $O(k^2)$ work to $\mathcal{S}'$. Hence, any bin in $\mathcal{S}$ can be replaced by $O(k^2)$ bins with binary mixes in $\mathcal{S}'$. As a result, $|\mathcal{S}'| = O(k^2|\mathcal{S}|)$, and can therefore be converted to an CFG with size $O(k^2|\mathcal{S}|)$ by Theorem 3.3.2. □

Theorem 3.4.9. *The separation of CFGs over SASs is $O(k^2)$.*

Proof. Let $\mathcal{S}$ be a SAS using k glues that produces an assembly with label string s. By Lemma 3.4.8, there is an CFG of size $O(k^2|\mathcal{S}|)$ that derives s. So separation is at most $O(k^2)$. □

Theorem 3.4.10. *The separation of CFGs over SAS is $O((n/\log n)^{2/3})$.*

Proof. Let $\mathcal{S}$ be a SAS with k glues producing an assembly with label string s, $|s| = n$. Either $k = O((n/\log n)^{1/3})$ or $k = \omega((n/\log n)^{1/3})$. If $k = O((n/\log n)^{1/3})$ then by Lemma 3.4.8 there is an CFG of size $O(k^2|\mathcal{S}|) = O((n/\log n)^{2/3} \cdot |\mathcal{S}|)$ with language $\{s\}$. So the separation is at most $O((n/\log n)^{2/3})$.

Now suppose $k = \omega((n/\log n)^{1/3})$. Then $|\mathcal{S}| = \omega((n/\log n)^{1/3})$. Lemma 2 of Section 2.2 in [Leh02] shows that there is a CFG of size $O(n/\log n)$ with language $\{s\}$. Hence, the separation is $o((n/\log n)^{2/3})$. So in both cases the separation is $O((n/\log n)^{2/3})$. □

We conjecture that the set of strings S_k yields the worst possible separation and leave this as an open problem:

Conjecture 3.4.11. *The separation of CFGs over SASs is $\Theta(k)$ and $\Theta(\sqrt{n/\log n})$.*

3.5 Unlabeled Shapes

Though this chapter has focused on labeled 1D assemblies, optimal construction of unlabeled 1D shapes (with label strings of the form a^n) using SASs still remains non-trivial. First, note that any string of the form a^n can be encoded by a CFG in 2NF of size $O(\log n)$ and this is tight:

Lemma 3.5.1. *The smallest CFG with language* $\{a^n\}$ *has size* $\Theta(\log n)$.

Proof. For the upper bound, we can an algorithm equivalent to the one described by Brauer [Bra39] for finding short *addition chains* (first defined in German by Scholz [Sch37] and English by Brauer). First, write n as a binary number and create a set of $\lfloor \log n \rfloor$ production rules of the form $A_i = A_{i-1}A_{i-1}$ for $i \in 1, 2, \ldots \lfloor \log n \rfloor$ and additional rule $A_0 \to a$. By definition, each A_i derives the string a^{2^i}. Next, create a rule $S \to A_{b_1}A_{b_2} \ldots A_{b_j}$ where $b_1, b_2, \ldots, b_j$ are the 1-valued indices of the binary representation of n. This rule has size $O(\log n)$, and the start symbol S derives the string $a^{2^{b_1}}a^{2^{b_2}} \ldots a^{2^{b_j}} = a^n$. So the grammar consisting of these rules and start symbol S has language $\{a^n\}$.

Next, consider any smallest grammar G in 2NF with language $\{s\}$. For each rule with two right-hand side symbols, the left-hand side symbol derives a string of length at most twice the length of either right-hand side symbol. So the addition of the rule increases $|G|$ by at most 2 and the length of the longest string derived by G by a factor of 2. So the longest string derived by G is $2^{|G|/2}$. Then the smallest grammar (in 2NF or general form) with language $\{a^n\}$ has size $\Omega(\log |a^n|) = \Omega(\log n)$. □

Combining Lemma 3.5.1 and Theorems 3.3.1 and 3.3.2 then implies an unlabeled (single-labeled) $n \times 1$ assembly can be constructed using a SSAS of size $O(\log n)$ and this is tight. Robert Schweller [Sch13] asked whether SASs can do substantially better, possibly achieving $O(\log \log n)$ assembly of

a $n \times 1$ assembly as done for two-dimensional binary counters of size $\Theta(n)$ as in [DDF+08a]. Here we show that achieving $o((\log n)^{1/3})$ is not possible by invoking a prior lemma.

Lemma 3.5.2. *Any SAS producing an assembly with label string a^n has size $\Omega((\log n)^{1/3})$.*

Proof. Let $\mathcal{S}$ be a SAS with k glues producing an assembly with label string a^n. By Lemma 3.4.8, $|\mathcal{S}| = \Omega(\log n/k^2)$. Moreover, any SAS with k glues has at least k distinct tile types and so has size $\Omega(k)$. The maximum of these two bounds is minimized when $\log n = k^3$, i.e. $k = (\log n)^{1/3}$. So $|\mathcal{S}| = \Omega(k) = \Omega((\log n)^{1/3})$. □

This result led Schweller to emit "Wow, your paper [referring to [DEIW11]] is pretty useful." Now we show that the paper was not all that useful by giving a stronger bound in Theorem 3.5.4 using a new argument. The existence of a stronger lower bound is not entirely surprising, as Lemma 3.4.8 is thought to be weak (Conjecture 3.4.11) and the argument uses a lower bound for intricate label strings S_k.

Lemma 3.5.3. *Any smallest SAS $\mathcal{S}$ with k glues producing an assembly with label string a^n has size $\Omega(\log_k n)$.*

Proof. Let the *stage* of a bin v be the length of the longest path in the mix graph from a root bin to v. We bound from above the size of any product of a bin in stage i, giving a lower bound on the number of stages and thus $|\mathcal{S}|$. Since $\mathcal{S}$ has k glues, each bin has at most k^2 reagent assemblies (by Lemma 3.2.4) and each product contains at most one copy of each reagent assembly (by Lemma 3.2.2). So the size of any product of a stage-i bin is at most k^2 times the size of any product of a stage-$(i-1)$ bin and any product

of a bin in stage i has size at most $(k^2)^i = k^{2i}$. So $\mathcal{S}$ must have h stages (and size at least h), where $k^{2h} \geq n$. So $|\mathcal{S}| = \Omega(\log_k n)$. □

Replacing $\Omega(\log n/k^2)$ with $\Omega(\log_k n)$ in the proof of Lemma 3.5.2 gives a stronger lower bound on $|\mathcal{S}|$ for a^n:

Theorem 3.5.4. *Any SAS producing an assembly with label string a^n has size $\Omega(\log n/\log\log n)$.*

Although this lower bound is nearly approaching the trivial upper bound, we believe that there is still room for improvement, possibly by using a more intricate analysis of the number of distinct glue pairs and size of the largest product in sequential assemblies.

Conjecture 3.5.5. *The smallest SAS producing an assembly with label string a^n has size $\Theta(\log n)$.*

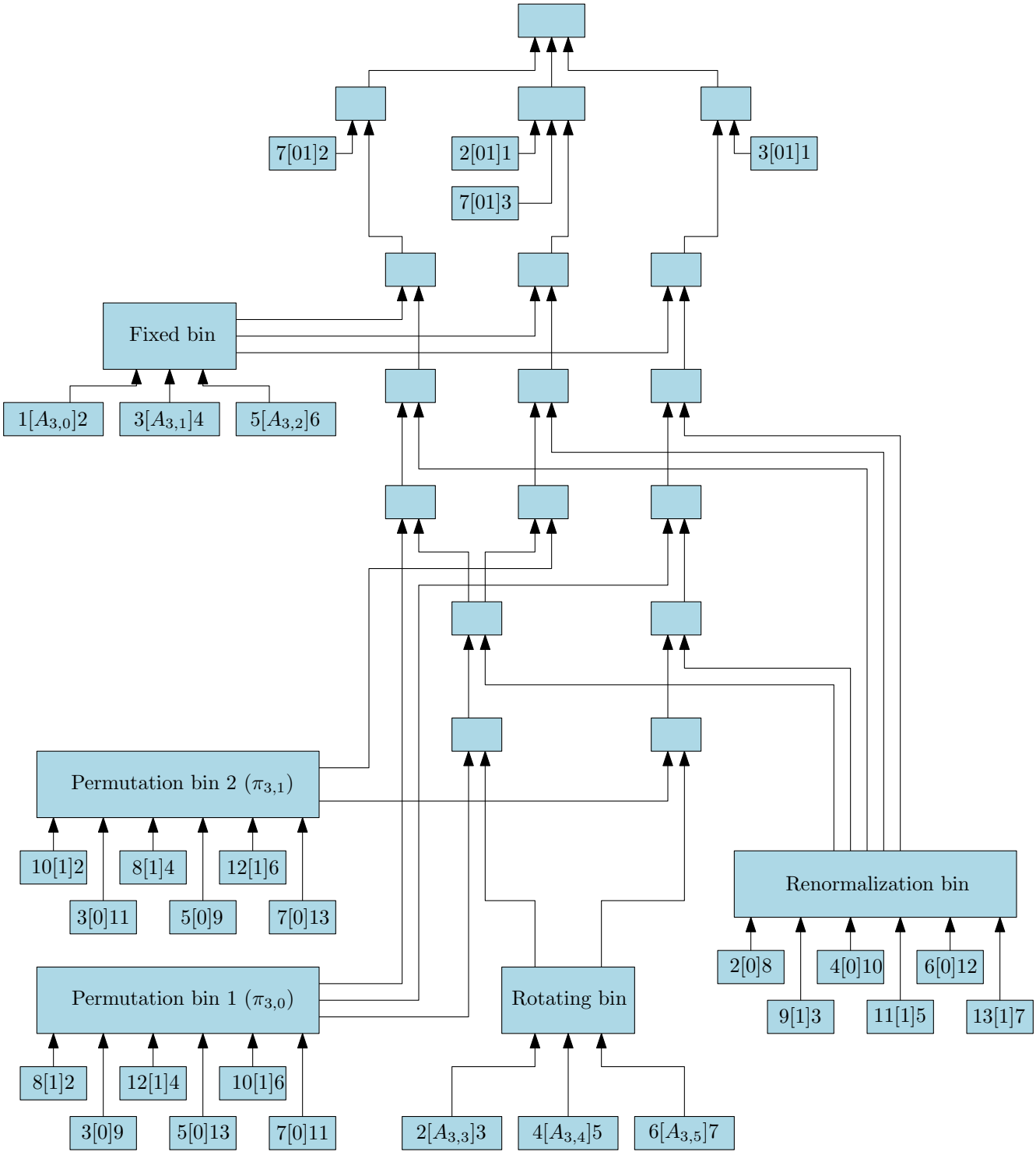


Figure 3.3: The mix graph for a SAS producing an assembly with label string S_3.

4

Polyomino Context-Free Grammars

The results of Chapter 3 give evidence of the fruitfulness of comparing CFGs and SASs. However, the results only apply to a special case of staged self-assembly in which produced assemblies are shapeless one-dimensional assemblies. The design of tile and assembly shape is used heavily in assembly techniques [CFS11, DDF$^+$08a, DDF$^+$12, FPSS12, KSX12, Rot01] and applications call for assembling a wide range of structures, such a circuits or cages. So while an understanding of one-dimensional staged assembly is useful, limiting assembly to one dimension leaves a large number of staged self-assembly techniques and problems untouched.

In Chapter 5 we extend the approach of Chapter 3 to two dimensions, comparing context-free grammars to two-dimensional staged self-assembly. These results have similar flavor to those in Chapter 3, but involve more complex techniques and constructions. Before this work can begin, we address a essen-

Portions of the work in this chapter have been published as [Win12].

tial question: what is a two-dimensional context-free grammar?

In this chapter, we explore this question in three steps that culminate with a new definition of two-dimensional context-free grammars we call *polyomino context-free grammars*. We begin in Section 4.1 by generalizing strings to *polyominoes*: connected arrangements of labels on the square grid. Next, we review existing definitions of two-dimensional grammars and their shortcomings in Section 4.2. As suggested by the existence of numerous proposed definitions, each existing definition fails to preserve some significant aspect of context-free grammars. In particular, all existing definitions fail to either map non-terminals to subpolyominoes, have languages with arbitrary polyominoes, or allow parsing and derivation in polynomial-time.

Finally, we introduce polyomino context-free grammars (PCFGs) in Section 4.3. Our focus on the smallest grammar problem and singleton languages proves to be key in developing this new definition, as PCFGs trade the inclusion of non-singleton languages for a natural and well-defined notion of two-dimensional context-free grammars with none of the failures previously described. A PCFG with language $\{P\}$ is defined as a recursive decomposition of P into subpolyominoes, an approach only possible if all languages are singleton.

4.1 Polyominoes

In this section we define labeled polyominoes and prove a set of results concerning only polyominoes. These results are used in to prove results related to polyomino context-free grammars, defined in Section 4.3.

4.1.1 Definitions

A *labeled polyomino* or *polyomino* $P = (S, L)$ is defined by a connected set of points S on the square lattice (called *cells*) and a label function $L : S \to \Sigma(P)$ mapping each cell of P to a *label* contained in an alphabet $\Sigma(P)$. The *size* of P is $|S|$, the number of cells in P, is denoted $|P|$. The label of the cell at lattice point (x, y) is denoted $L((x, y))$ and we define $P(x, y) = L((x, y))$ for notational convenience. For convenience in describing constructions, we refer to the *label* or *color* of a cell interchangeably.

A *translation* of P is a polyomino $P' = (S', L')$ such that for some (δ_x, δ_y), $S' = \{(x + \delta_x, y + \delta_y) \mid (x, y) \in S\}$ and $L'((x + \delta_x, y + \delta_y)) = L((x, y))$ for all $(x, y) \in S$. Two polyominoes $P = (S, L)$ and $P' = (S', L')$ are *overlapping* if there exists some (x, y) such that $(x, y) \in S$ and $(x, y) \in S'$. Similarly, two polyominoes are *compatible* if for each (x, y), either $(x, y) \notin S$, $(x, y) \notin S'$, or $P(x, y) = P'(x, y)$.

The polyomino $P' = (S', L')$ is a *superpolyomino* of $P = (S, L)$ if there exists a translation of P that is compatible with P' and whose cells are a subset of S'. Equivalently, P is a *subpolyomino* of P'.

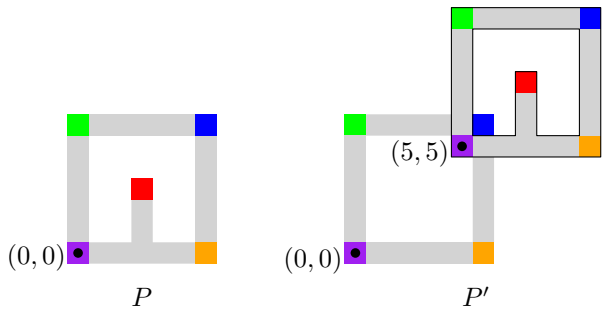


Figure 4.1: A polyomino P and superpolyomino P'. The polyomino P' is a superpolyomino of P, since the translation of P by $(5, 5)$ (shown in dark outline in P') is compatible with P' and the cells of this translation are a subset of the cells of P'.

4.1.2 Smallest common superpolyomino problem

The smallest common superpolyomino problem restricted to $n \times 1$ polyominoes is the well-known *shortest common superstring problem*:

Problem 4.1.1 (Shortest common superstring)**.** *Given a set of strings* $S = \{s_1, \ldots, s_m\}$ *with* $\sum_1^m |s_i| = n$*, find the shortest string* s *such that each* s_i *in* S *is a substring of* s*.*

The smallest common superstring problem is known to be NP-hard [GMA80], 2.5-approximable [Swe94], and 1.00082-inapproximable [Vas05]. Blum et al. [BJL$^+$94] proved that the trivial greedy algorithm which repeatedly merges the two strings with the largest overlap achieves a 4-approximation. Here we show that the smallest common superpolyomino problem is $O(n^{1/3-\varepsilon})$-inapproximable:

Problem 4.1.2 (Smallest common superpolyomino)**.** *Given a set of polyominoes* $S = \{P_1, \ldots, P_m\}$ *with* $\sum_1^m |P_i| = n$*, find the smallest polyomino* P *such that each* P_i *in* S *is a subpolyomino of* P*.*

Our reduction is from the *chromatic number problem*, the optimization version of the k-coloring problem, where the goal is to find the smallest k for which a k-coloring exists:

Problem 4.1.3 (Chromatic number)**.** *Given a graph* $G = (V, E)$*, find the smallest* k *such that* G *admits a* k*-coloring.*

The chromatic number problem is easily shown to be NP-hard by a reduction from k-coloring, and was shown to be difficult to approximate within almost any non-trivial factor by Zuckerman [Zuc07]:

Lemma 4.1.4. *The chromatic number problem is* $O(|V|^{1-\varepsilon})$*-inapproximable for all* $\varepsilon > 0$*. [Zuc07]*

The reduction Given a graph $G = (V, E)$, convert each vertex $v \in V$ into a polyomino P_v that encodes v and the neighbors of v in G (see Figure 4.2). Each P_v is a rectangular $2|V| \times |V|$ polyomino with up to $|V| - 1$ single squares removed. The four corners of all P_v have a common set of four colors: green, blue, purple, and orange. Let the lower-left corner of P_v be $(0, 0)$. Cells at locations $\{(2i+1, 1) \mid 0 \leq i < |V|\}$ are colored black if $v_i = v$, red if $(v, v_i) \in E$, or are empty locations if v_i is not v or a neighbor of v. All remaining cells are colored gray.

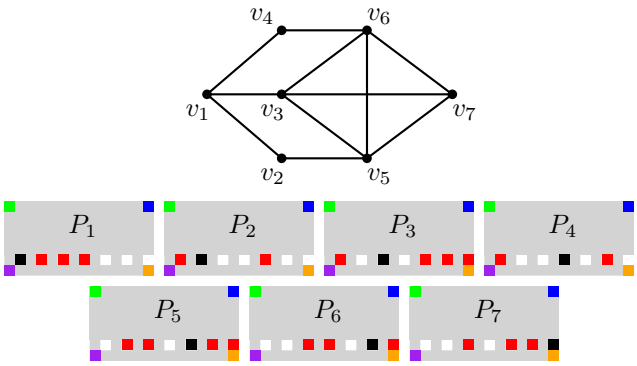


Figure 4.2: An example of the set of polyominoes generated from an input graph by the reduction in Section 4.1.2.

Consider how two polyominoes P_u and P_v can be compatible and overlapping. Because the four corners have unique colors, P_u and P_v are only compatible when these four locations in P_v are translated to the same locations in P_u. In this translation, the cells at location $(2i + 1, 1)$ in P_u and P_v are compatible exactly when $(u, v) \notin E$, i.e. when u and v are not neighbors. All other cells are gray and are compatible.

The superpolyomino formed by a pair of compatible and overlapping polyominoes P_u and P_v has the common set of four colored corner cells, two black cells corresponding to u and v, and a number of red cells corresponding to the combined neighborhoods of u and v. More generally, any set of two or more polyominoes overlap to form a superpolyomino if and only their corresponding

vertices form an independent set I. We call the superpolyomino resulting from such a set of overlapping polyominoes a *deck* (see Figure 4.3).

Each deck is a rectangular $2|V| \times |V|$ polyomino of mostly gray cells, with the common set of four colored corner cells, a black cell for each vertex in I, a collection of red cells for the neighbors of vertices in I, and up to $|V| - 2$ single cells removed for the vertices neither in I nor the neighborhood of I. Since polyominoes can only overlap to form decks, any superpolyomino of the polyominoes $\{P_v \mid v \in V\}$ consists of a disjoint arrangement of decks.

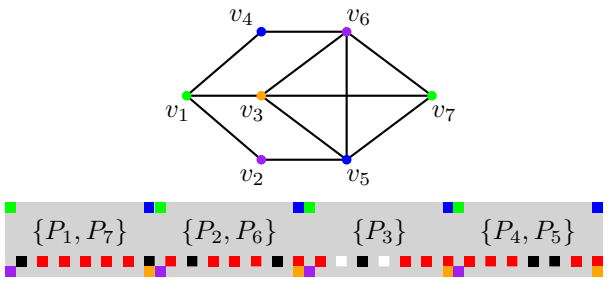


Figure 4.3: An example of a 4-deck superpolyomino and corresponding 4-colored graph. Each deck is labeled with the input polyominoes the deck contains, e.g. the leftmost deck is the superpolyomino of overlapping P_1 and P_7 input polyominoes.

Recall that each P_v is a $2|V| \times |V|$ rectangle with $|V|$ cells colored black, red, or are not present. The size of P_v is then between $2|V|^2 - |V| + 1$ and $2|V|^2$ depending upon the number of neighbors of v, and each deck of polyominoes also has size in this range.

Lemma 4.1.5. *For a graph $G = (V, E)$, there exists a superpolyomino of size at most $2k|V|^2$ for polyominoes $\{P_v \mid v \in V\}$ if and only if the vertices of V can be k-colored.*

Proof. First, consider extreme sizes of superpolyominoes consisting of k and $k + 1$ decks. For any V and k with $1 \leq k \leq |V|$:

$$\begin{aligned}(2|V|^2 - |V|)(k+1) &= 2|V|^2k + 2|V|^2 - |V|k - |V| \\ &= 2|V|^2k + |V|(2|V| - (k+1)) \\ &= 2|V|^2k + |V|(|V| - 1) \\ &> 2|V|^2k\end{aligned}$$

That is, any superpolyomino of $k+1$ or more decks is larger than any superpolyomino of k decks. Now we prove both implications of the lemma. First, assume that a superpolyomino of size at most $2|V|^2k$ exists. Then the superpolyomino must consist of at most k decks. Each deck is the superpolyomino of a set of polyominoes forming an independent set, so G can be k-colored. Next, assume G can be k-colored. Then the polyominoes $\{P_v \mid v \in V\}$ can be translated to form k decks, one for each color, each with size at most $2|V|^2$. Placing these decks adjacent to each other yields a superpolyomino of size at most $2|V|^2k$. □

Note that only $|V|$ cells of each P_v are distinct and depend on v, while the other $2|V|^2 - |V|$ cells are identical for all P_v. The extra cells are needed for the inequality used in Lemma 4.1.5; their purpose is to "drown out" variation in the size of each deck due to missing cells.

Theorem 4.1.6. *The smallest common superpolyomino problem is $O(n^{1/3-\varepsilon})$-inapproximable for any $\varepsilon > 0$.*

Proof. Consider the smallest common superpolyomino problem for the polyominoes generated from a graph $G = (V, E)$ with chromatic number k. There are $|V|$ of these polyominoes, each of size $\Theta(|V|^2)$, so the polyominoes have

total size $n = \Theta(|V|^3)$.

By Lemma 4.1.5, a superpolyomino of size at most $2|V|^2k'$ exists if and only if there exists a k'-coloring of G. So by Lemma 4.1.4, finding a superpolyomino of size at most $2|V|^2k'$ with $\frac{2|V|^2k'}{2|V|^2k} = \frac{k'}{k} = O(|V|^{1-\varepsilon})$ is NP-hard. So the smallest common superpolyomino problem is $O(|V|^{1-\varepsilon}) = O(n^{1/3-\varepsilon})$-inapproximable.

□

Now we show that the same problem restricted to polyomino sets of a single color remains NP-hard. Our reduction is from the *set cover problem* using families of polyominoes as seen in Figures 4.4 and 4.5.

Problem 4.1.7 (Set cover)**.** *Given a* universe $U = \{1, 2, \ldots, u\}$ *and set of sets* $S = \{S_1, \ldots, S_m\}$ *with* $\bigcup_{1 \le i \le m} S_i = U$*, find the smallest subset* $S' \subseteq S$ *such that* $\bigcup_{S_i \in S'} S_i = U$*.*

Lemma 4.1.8. *The set cover problem is* NP*-hard. [Kar72].*

Given an instance of the set cover problem, we create a set of u *universe polyominoes* and one *set polyomino*. The universe polyominoes $\{P_1, P_2, \ldots, P_u\}$ each consist of a $(u+1) \times (u+1)$ *base*, a $1 \times 2u$ *flagpole*, and a $u \times 1$ *flag*. The lower-left corners of the three components for polyomino P_i are placed at $(0, 0)$, $(0, u)$, and $(1, u + 2i)$, respectively.

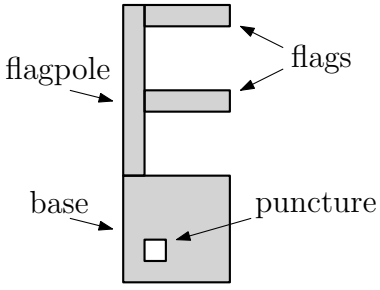


Figure 4.4: The components of universe polyominoes and set gadgets used in the reduction from set cover to the smallest common superpolyomino problem.

The set polyomino $\overline{P}$ consists of a set of m *punctured bases*: a base with cell $(1, 1)$ missing. These bases are arranged horizontally and attached by single cells. Each punctured base has a flagpole and some number of flags (called a *set gadget*) encoding the elements of a set S_i in S. For the set $S_i = \{e_1, e_2, \ldots, e_l\}$, flags are placed at locations $\{((u + 2)(i - 1) + 1, u + 2e_j) \mid 1 \leq j \leq l\}$.

Universe polyominoes Set polyomino

Figure 4.5: An example of the set of polyominoes generated from the input set $\{\{1, 2\}, \{1, 4\}, \{2, 3, 4\}, \{2, 4\}\}$ by the reduction from set cover to the smallest common superpolyomino problem.

Consider forming a superpolyomino of the set polyomino $\overline{P}$ and some universe polyomino P_i corresponding to an element $i \in U$. Define a translation of P_i that places the lower-left corner of its base at the lower-left corner of a punctured base of $\overline{P}$ as an *alignment* of the bases. If the base of some P_i is not aligned with a punctured base of $\overline{P}$, then the resulting superpolyomino is at least as large as *any* superpolymino of $\overline{P}$ and all P_i in which each P_i is aligned with some base:

Lemma 4.1.9. *Any superpolyomino of the set polyomino $\overline{P}$ and universe polyomino P_i in which the base of P_i is not aligned with the punctured base of a set gadget corresponding to a set containing e_i has size at least $\overline{P} + u$.*

Proof. If the base of P_i is aligned with the punctured base of a set gadget that does not correspond to a set containing i, then the cells of the flag of P_i do not overlap with the cells of $\overline{P}$. The flag has u cells, and so the superpolyomino

has size at least $|\overline{P}| + u$.

If the base of P_i is *not* aligned with any punctured base in $\overline{P}$, then it must be horizontally or vertically misaligned. If the base is horizontally misaligned, then some column that does not contain any punctured base of $\overline{P}$ contains cells from the base of P_i. Such a column then must contain at least $u+1$ cells of P_i and at most 1 cell of $\overline{P}$, so the superpolyomino has size at least $|\overline{P}| + u$.

If the base is not horizontally misaligned but is vertically misaligned, then some row containing a row of the base of P_i has at most 1 cell of $\overline{P}$. This row is either a row entirely below or above the set gadget aligned with P_i or a row that contains a cell of the flagpole but no flag. In either case, the superpolyomino has at least $u + 1$ cells of P_i and at most 1 cell of $\overline{P}$, so the superpolyomino has size at least $|\overline{P}| + u$. □

Intuitively, Lemma 4.1.9 implies that "playing by the rules" and aligning each universe polyomino P_i with a set gadget in $\overline{P}$ is always better than "cheating" by not aligning some universe polyomino with any punctured base. So finding a minimum superpolyomino is equivalent to deciding which set gadget to align each universe polyomino with, and the goal is to find set of alignments that minimize the number of "patched" punctured bases (see Figure 4.6).

Figure 4.6: The smallest common superpolyomino of the polyominoes in Figure 4.5, corresponding to the set cover $\{S_1, S_3\}$.

Theorem 4.1.10. *The smallest common superpolyomino problem for one-color polyomino sets is* NP*-hard.*

Proof. By Lemma 4.1.9, the smallest common superpolyomino of the entire set of universe polyominoes and the set polyomino has each universe polyomino P_i aligned with some set gadget of $\overline{P}$ corresponding to a set containing i. Each set gadget used by some universe polyomino increases the size of the superpolyomino by 1. So a superpolyomino of size $|\overline{P}| + k$ exists if any only if there exists a set cover of size k. Then by Lemma 4.1.8, finding the smallest superpolyomino is NP-hard. □

We conjecture that this problem is approximable within some small constant factor, possibly even using a greedy algorithm.

Conjecture 4.1.11. *The smallest common superpolyomino problem for one-color polyomino sets has a c-approximation for some constant $c > 1$.*

4.1.3 Largest common subpolyomino problem

Generalizing the shortest common superstring problem to the smallest common superpolyomino problem (on polyomino sets with multiple colors) yielded a problem that was still in NP, but much more inapproximable. On the other hand, the *longest common substring problem* is solvable in polynomial time:

Problem 4.1.12 (Longest common substring)**.** *Given a set of strings $S = \{s_1, \ldots, s_m\}$ with $\sum_{i=1}^{m} |s_i| = n$, find the longest string s such that s is a substring of each $s_i \in S$.*

Hui [Hui92] showed that the longest common substring problem is solvable in $O(n)$ time by reducing the problem to an ancestor problem in the generalized suffix tree [Wei73, GK97] constructed from the input strings. Somewhat surprisingly, generalizing the longest common substring problem yields a highly inapproximable problem:

Problem 4.1.13 (Largest common subpolyomino)**.** *Given a set of polyominoes $S = \{P_1, \ldots, P_m\}$ with $\sum_{i=1}^{m} |P_i|$, find the largest polyomino P such that P is a subpolyomino of every polyomino in S.*

Like the smallest common superpolyomino problem, we achieve $O(n^{1/3-\varepsilon})$-inapproximability for the largest common subpolyomino problem (Theorem 4.1.17). A similar reduction combined with a classic number-theoretic result gives $O(n^{1/4-\varepsilon})$-inapproximability for the problem restricted to one-color polyomino sets (Theorem 4.1.20). The reductions are done from the independent set problem, a problem closely related to the chromatic number problem:

Problem 4.1.14 (Independent set)**.** *Given a graph $G = (V, E)$, find the largest set of vertices $S \subseteq V$ such that for any pair of vertices $v_i, v_j \in S$, $(v_i, v_j) \notin E$.*

In the same work proving the inapproximability of the chromatic number problem (Lemma 4.1.4), Zuckerman [Zuc07] gave a similar result for the independent set problem:

Lemma 4.1.15. *The independent set problem is $O(|V|^{1-\varepsilon})$-inapproximable for all $\varepsilon > 0$. [Zuc07]*

In Section 4.1.2, the chromatic number problem was reduced to the smallest common superpolyomino problem by creating a set of polyominoes that encode the independence constraints of the vertices. The goal was then to minimize the union of a compatible arrangement of these polyominoes. A similar idea is used for the reduction here: a set of polyominoes is used to encode the independence constraints of the vertices, and the goal is to maximize the intersection of a compatible arrangement of the polyominoes.

The reduction Given a graph $G = (V, E)$, construct a polyomino P_v for each $v \in V$, and an additional polyomino P_{base}. The polyomino P_{base} is a $(2|V| - 1) \times (|V| + 1)$ rectangle with gray alternating columns missing, save for their bottommost cells. The resulting shape is a *comb*, and the present columns are *comb teeth*. The bottommost row alternates between the gray color of the comb teeth, and $|V|$ other distinct colors (see Figure 4.7).

Each polyomino P_v has two *choice gadgets* connected horizontally with a single row of width $2|V|-1$. The two choice gadgets are each identical to P_{base} with the exclusion of one or more comb teeth found. For the polyomino P_{v_i} with $1 \leq i \leq |V|$, the left choice gadget has column $2i - 1$ removed and the right choice gadget has the columns $\{2j - 1 \mid (v_i, v_j) \in E\}$ removed, i.e. the set of columns corresponding to the neighbors of v_i. An example of a graph and corresponding polyominoes is seen in Figure 4.7.

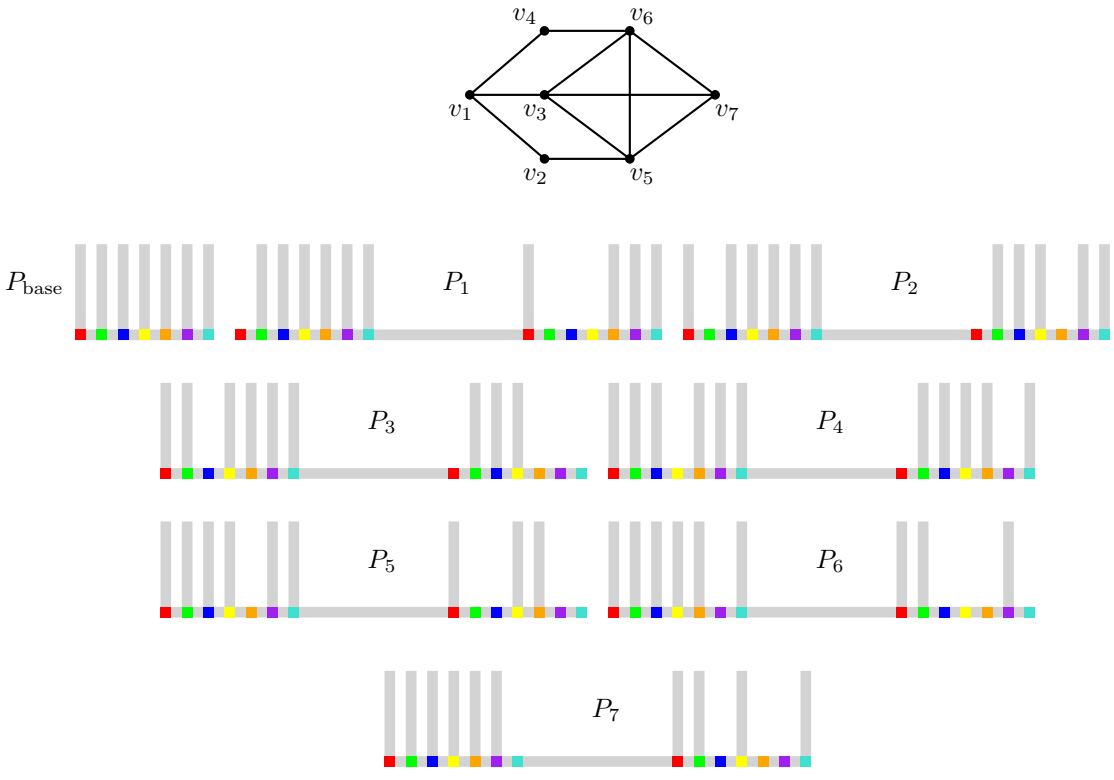


Figure 4.7: An example of the set of polyominoes generated from an input graph by the reduction in Section 4.1.3.

Starting with P_{base} as the largest possible common subpolyomino, the subpolyomino must fit into one of two choice gadgets in each polyomino P_v (seen in Figure 4.8). So either the comb tooth corresponding to v or the comb teeth corresponding to the neighbors of v are missing.

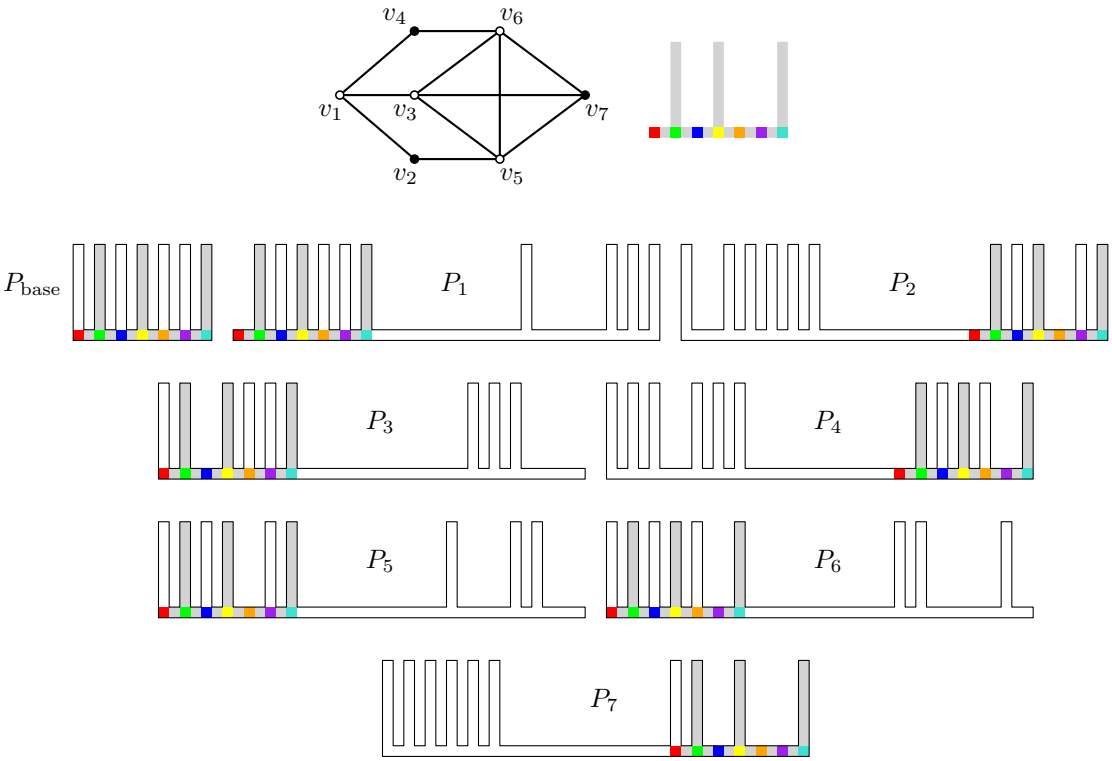


Figure 4.8: An example of a corresponding largest subpolyomino and maximum independent set (top) and locations of the subpolyomino in each polyomino produced by the reduction.

Lemma 4.1.16. *For a graph $G = (V, E)$, there exists a common subpolyomino of size at least $2|V| - 1 + k|V|$ if and only if there exists an independent set of vertices in V of size at least k.*

Proof. Let $P_{\max}$ be the largest common subpolyomino for the set of polyominoes generated from a graph $G = (V, E)$. Then $P_{\max}$ contains at least one non-gray cell, as every connected set of gray cells in P_{base} has size at most $|V| + 1$ and the bottommost row of P_{base} is a common subpolyomino and has size $2|V| - 1$. Furthermore, since $P_{\max}$ contains a non-gray cell, it contains the

entire bottommost row of P_{base}, as otherwise it can be extended to include this row. Finally, since every column of containing more than one cell is a comb tooth, every column of $P_{\max}$ should contain either one cell or $|V|+1$ cells. So $P_{\max}$ has shape and pattern identical to P_{base} except for some number of some missing teeth.

Consider the forward direction of the implication, with a common subpolyomino P_{soln} such that $|P_{\text{soln}}| \geq 2|V| - 1 + k|V|$. By previous arguments, P_{soln} must have at least k teeth. Consider a pair of teeth t_i, t_j in P_{soln} corresponding to vertices $v_i, v_j \in V$. Since P_{v_i} contains P_{soln} and t_i is not found in the left choice gadget of P_i, the right choice gadget of P_i must contain P_{soln}. So the right choice gadget has teeth t_i and t_j and thus $(v_i, v_j) \notin E$. So the set of vertices corresponding to the set of teeth in P_{soln} is an independent set, and G has an independent set of size at least k.

Proving the backwards direction is nearly the same. Given an independent set $I \subseteq V$ with $|I| = k$, construct a comb polyomino P_{soln} with teeth corresponding to the set of vertices in the independent set. This polyomino has size $2|V| - 1 + k|V|$, as the bottommost row has size $2|V| - 1$ and each tooth has size $|V|$. Consider placing P_{soln} inside one of two choice gadgets for a polyomino P_v. If $v \notin I$, then P_{soln} does not have the tooth corresponding to v and is a subpolyomino of the left choice gadget of P_v. If $v \in I$, then since I is an independent set, none of the teeth corresponding to the neighbors of v are found in P_{soln}, and P_{soln} is a subpolyomino of the right choice gadget of P_v. So the constructed P_{soln} is a subpolyomino of every P_v (and P_{base} by construction) and has size at least $2|V| - 1 + k|V|$. □

Theorem 4.1.17. *The largest common subpolyomino problem is $O(n^{1/3-\varepsilon})$-inapproximable.*

Proof. The proof proceeds identically to that of Theorem 4.1.6. Consider the

largest common subpolyomino for the polyominoes generated from a graph $G = (V, E)$ with independence number k. There are $|V| + 1$ of these polyominoes, each of size $\Theta(|V|^2)$, so the polyominoes have total size $n = \Theta(|V|^3)$.

By Lemma 4.1.16, a subpolyomino of size at least $2|V| - 1 + k'|V|$ exists if and only if there exists an independent set of size at least k'. So by Lemma 4.1.15, finding a subpolyomino of size at least $2|V| - 1 + k'|V|$ with $\frac{2|V|-1+k|V|}{2|V|-1+k'|V|} \geq \frac{1}{2}\frac{k|V|}{k'|V|} = O(|V|^{1-\varepsilon})$ is NP-hard. So the largest common subpolyomino problem is $O(|V|^{1-\varepsilon}) = O(n^{1/3-\varepsilon})$-inapproximable. □

Unlike the smallest common superpolyomino problem, we are also able to achieve polynomial inapproximability for one-color polyomino sets. The reduction is modified to replace the distinct colors of the cells at the bases of teeth with distinct horizontal spacings between teeth. The spacings used are generated by a set of integers whose pairwise distances are each distinct. Sidon [Sid32] showed that for some $c > 0$, there exists a subset of the integers $\{1, 2, \ldots, cn^4\}$ of size n such that all pairwise sums of the integers are distinct. Erdös and Turan [ET41] showed that the same result applies to the set of integers $\{1, 2, \ldots, c'n^2\}$ for some $c' > 0$, by giving an example of such a set:

Lemma 4.1.18. *For any prime p and four integers $a_1, a_2, a_3, a_4 \in \{2pi + (i^2 \bmod p) \mid 1 \leq i < p\}$, if $a_1 + a_2 = a_3 + a_4$ then $\{a_1, a_2\} = \{a_3, a_4\}$ [ET41].*

Since n integers induce $\binom{n}{2}$ pairwise distances, a lower bound of $\Omega(n^2)$ applies, so this result is asymptotically tight. We transform this result into a slightly different form regarding the distances (differences) between integers:

Lemma 4.1.19. *For any prime p and four integers $a_1, a_2, a_3, a_4 \in \{2pi + (i^2 \bmod p) \mid 1 \leq i < p\}$, if $a_1 - a_2 = a_3 - a_4$ with $\{a_1, a_2\} \neq \{a_3, a_4\}$, $a_1 \neq a_2$, $a_3 \neq a_4$, then $a_1 = a_3$ and $a_2 = a_4$.*

Proof. Suppose such a set of integers is given. Since $a_1 - a_2 = a_3 - a_4$, then $a_1 + a_4 = a_3 + a_2$ and by Lemma 4.1.18, $\{a_1, a_4\} = \{a_3, a_2\}$. Also, $a_1 \neq a_4$ and $a_2 \neq a_3$, since otherwise $|\{a_1, a_4\}| = |\{a_2, a_3\}| = 1$ and $a_1 = a_2$. Then since $a_1 \neq a_2$ and $a_3 \neq a_4$, $a_1 = a_3$ and $a_4 = a_2$. □

From this result the necessary modification follows: rather than place teeth in every other column of P_{base} and each P_v, select a prime $p \geq |V| + 1$ and place teeth in columns $\{2(2pi + (i^2 \bmod p)) - 1 \mid 1 \leq i < p\}$. Then any pair of teeth found in a common subpolyomino corresponds to a distinct pair of teeth in the choice gadgets, and any set of teeth corresponds to a well-defined set of vertices of the input graph.

Theorem 4.1.20. *The largest common subpolyomino problem for polyomino sets with a one-symbol alphabet is $O(n^{1/4-\varepsilon})$-inapproximable.*

Proof. We modify the previous construction as described: for both P_{base} and each P_v, replace the multi-colored choice gadgets with one-color choice gadgets. The bottommost row of the choice gadget has width $f(p) = 2(2p(p-1) + ((p-1)^2 \bmod p)) - 1 = 4p^2 - 4p + 1$, as does the single row connecting both choice gadgets. For each tooth t_i in column $2i - 1$ of a gadget in the original construction, place a tooth with height $f(p)$ in column $2(2pi + (i^2 \bmod p)) - 1$.

Consider the largest common subpolyomino $P_{\max}$ of this modified one-color set of polyominoes for an input graph $G = (V, E)$. Assume G has an independent set of size at least two. Then $|P_{\max}| \geq 3f(p)$ by arguments in the proof of Theorem 4.1.17.

Because $|P_{\max}| \geq 3f(p)$ and is contained in P_{base}, $P_{\max}$ has at least two columns containing multiple cells. Pick two such columns separated by horizontal distance d. By the construction and Lemma 4.1.19, columns with multiple cells separated by distance d appear only once in each choice gadget.

So $P_{\max}$ is contained in one of two choice gadgets in each P_v, and so contains the entire bottommost row of P_{base}. Then $P_{\max}$ has the same properties of $P_{\max}$ in the proof of Theorem 4.1.17: containment in one of two choice gadgets in each P_v, containing the entire bottommost row of P_b, and full teeth selected from the teeth of P_{base}. So the teeth in $P_{\max}$ correspond to a largest independent set $I \subseteq V$.

Now consider the inapproximability factor achieved. The well-known Chebyshev's theorem[1] [Erd32, Ram19] says that a prime p such that $|V| < p < 2|V|$ exists, and so such a p can be found by naive primality testing in $O(|V|^{3/2})$ time and $f(p) = \Theta(|V|^2)$. So each polyomino has total size $\Theta((|V|+1)f(p)) = \Theta(|V|^3)$ and the entire set of polyominoes has size $n = \Theta(|V|^4)$.

Then by Lemma 4.1.15, finding a subpolyomino of size at least $(k'+1)f(p)$ with $\frac{(k+1)f(p)}{(k'+1)f(p)} \geq \frac{1}{2}\frac{k}{k'} = O(|V|^{1-\varepsilon})$ is NP-hard. So the largest common subpolyomino problem for one-color sets of polyominoes is $O(n^{1/4-\varepsilon})$-inapproximable. □

4.1.4 Longest common rigid subsequence problem

Now we apply the ideas from Section 4.1.3 to give a new simple proof of a result on *rigid subsequences* of strings: subsequences where each symbol has a fixed offset from the first symbol of the sequence. A *rigid sequence* is a pair of sequences $((c_1, c_2, \ldots, c_m), (o_1 = 1, o_2, \ldots, o_m))$ where $(c_1, \ldots, c_m)$ is a sequence of symbols and $(o_1, \ldots, o_m)$ is a strictly increasing sequence of integer offsets specifying the relative positions of each c_i relative to c_1. Similarly, a *rigid subsequence* of a string s is a rigid sequence $((c_1, \ldots, c_m), (o_1, \ldots, o_m))$ such that for some integer δ, the $(o_i+\delta)$th symbol of s is c_i for all $1 \leq i \leq m$. For instance, the string *blabatile* has a rigid subsequence $((a, b, l, e), (1, 2, 6, 7))$ (let $\delta = 2$).

[1] Alternatively called "Tschebyschef's theorem" or "Bertrand's postulate".

Rigid subsequences can also be denoted by a sequence of symbols with whitespace or wildcards at indices not appearing in the offset sequence, e.g. *ab le* or *ab...le*. The *length* of a rigid subsequence $r = ((c_1, \ldots, c_m), (o_1, \ldots, o_m))$ is m, and is denoted $|r|$. Ma and Zhang [MZ05] and later with Bansal and Lewenstein [BLMZ10] considered the problem of finding the longest common rigid subsequence for a set of input strings:

Problem 4.1.21 (Longest common rigid subsequence)**.** *Given a set of strings* $S = \{s_1, s_2, \ldots, s_m\}$ *with* $\max(|s_i|) = n$*, find the longest rigid sequence* r *such that* r *is a rigid subsequence of each* $s_i \in S$*.*

For instance, the longest common rigid subsequence of $\{abadale, baracade, clanadare\}$ is $((a, a, a, e), (1, 3, 5, 7))$. Bansal, Lewenstein, Ma, and Zhang [BLMZ10] obtained the following inapproximability result for the problem:

Theorem 4.1.22. *The longest common rigid subsequence problem is hard to approximate within a factor of* $o(m)$ *and* $O(n^{1-\varepsilon})$ *for sets of strings with alphabet size 4 [BLMZ10].*

We give simple proofs of two results nearly matching the result of Bansal et al. Although the approach was conceived independently by the author, it effectively combines the proof of Jiang and Li [JL95] that the longest common subsequence problem is n^{δ}-inapproximable with the proofs of Theorem 4.1.17and 4.1.20.

Theorem 4.1.23. *The longest common rigid subsequence problem is* $O(m^{1-\varepsilon})$*-inapproximable and* $O(n^{1-\varepsilon})$*-inapproximable.*

Proof. Given a graph $G = (V, E)$, generate a string s_{base} and set of $|V|$ strings $\{s_i s_i' \mid 1 \leq i \leq |V|\}$, with:

$$s_{\text{base}} = a_1 a_2 \ldots a_{|V|}$$

$$s_i = a_1 a_2 \dots a_{i-1} 0 a_{i+1} \dots a_{|V|-1} a_{|V|}$$

$$s'_i = f(i,1) f(i,2) \dots f(i,|V|-1) f(i,|V|)$$

and $f(i,j)$ defined as 0 if $(v_i, v_j) \in E$ and a_i otherwise. For instance, if $|V| = 5$ and vertex v_3 has neighbors v_1, v_2, and v_5, then $s_3 = a_1 a_2 0 a_4 a_5$ and $s'_3 = 00 a_3 a_4 0$.

Consider a rigid subsequence $r = ((c_1, \dots, c_m), (o_1, \dots, o_m))$ of the generated string set. Since r is a rigid subsequence of s_{base}, $|r| \leq |V|$ and $c_i \neq 0$ for all c_i. Moreover, since each a_i appears once in ascending order in s_{base} and thus $(c_1, \dots, c_m)$, r is a rigid subsequence of either s_i or s'_i for each $1 \leq i \leq |V|$.

Now consider the set of vertices $I = \{v_i \mid a_i \in (c_1, \dots, c_m)\}$, which we claim is an independent set for G. For each vertex $v_i \in I$, r must be a rigid subsequence of s'_i and not s_i, since $a_i \notin s_i$. So for any vertex v_j such that $(v_i, v_j) \in E$, $a_i \notin s'_i$, and so $v_j \notin I$. That is, I is an independent set. So a common rigid subsequence r of $\{s_b\} \cup \{s_i s'_i \mid 1 \leq i \leq |V|\}$ with $|r| \geq k$ exists if and only if G has an independent set of size k. Also, $|V| + 1 = m$ and $2|V| = n$. Then by Lemma 4.1.15, approximating the largest common rigid subsequence is $O(|V|^{1-\varepsilon}) = O(m^{1-\varepsilon}) = O(n^{1-\varepsilon})$-inapproximable. □

Modifying this string construction in the same way as Theorem 4.1.20 by replacing unique symbols with unique spacing yields a set of $|V| + 1$ strings, each of size $O(|V|^2)$. These strings have alphabet size 2 and length $n = \Theta(|V|^2)$, so the inapproximability ratio achieved is weaker in n, but applies to subsequences on a smaller alphabet (size 2) than before (size 4 in Thm. 4.1.22):

Corollary 4.1.24. *The longest common rigid subsequence problem is hard to approximate within a factor of $O(m^{1-\varepsilon})$ and $O(n^{1/2-\varepsilon})$ for strings with alphabet size 2.*

One might argue that the complexity of the proof has simply been pushed to the difficulty of showing that the maximum independent set problem is $O(n^\delta)$-inapproximable. Note that the polynomial inapproximability of the independent set problem had been known for at least seven years (1998) [ALM+98] before the initial results of Ma and Zhang on the largest common rigid subsequence problem [MZ05] (2005). Moreover, our reduction can be interpreted as simply: "The longest common rigid subsequence problem is as hard to approximate as the independent set problem" whereas the argument of [BLMZ10] is more intricate: "The largest common rigid subsequence problem is as hard to approximate as a gap-amplified and derandomized version of the maximum dicut problem (k-fold maximum dicut)."

4.2 Existing 2D CFG definitions

With some helpful results established, we now shift back to the goal of developing a definition of two-dimensional context-free grammars. Consider the geometry involved in applying a production rule $E \to ff$ to a partially derived string $abEcd$ to produce $abffcd$. If the string is placed on a two-dimensional square lattice such that a occupies the lattice point $(0, 0)$ and d occupies $(4, 0)$, then applying the rule requires either moving ab one unit to the left or cd one unit to the right to create two horizontally-adjacent cells for ff to replace the single cell of E. In two dimensions, this shifting may cause other portions of the polyomino to *shear* by intersecting or disconnecting.

Shearing is not an issue for strings because strings always admit the expansive motion necessary to apply a production rule without causing disconnections or overlaps. For polyominoes, such a motion in general is not possible. Understanding shearing is helpful because existing definitions of two-

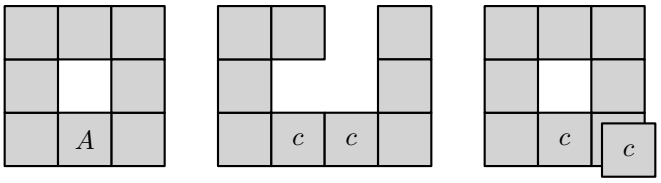


Figure 4.9: Two shearing possibilities (middle and right) resulting from applying the production rule $A \to cc$.

dimensional grammars can be classified according to how they handle shearing, with three primary approaches used. Each approach results in a definition lacking at least one of the following properties: tractable (polynomial-time) parsing and derivation, languages of arbitrary polyominoes, or non-terminals that corresponding to subpolyominoes.

Picture languages *Picture languages* (using the terminology of [GR97]) are grammars restricted to rectangular shapes, summarized in the survey of Cherubini and Pradella [CP09]. Kolam and matrix grammars [SSK72], and tile grammars [RP05, CRP08] are of this variety. The grammar-based compression work by Hayashida, Ruan, and Akutsu [HRA10] also uses grammars of this variety. These definitions restrict production rules to the addition of complete rows or columns, which avoids shearing by ensuring the necessary expansive motion is always possible. As a side effect, picture languages are limited to rectangular polyominoes.

Array grammars The *array grammars* of Milgram and Rosenfeld [MR71] and *puzzle grammars* of Laroche et al. [LNS92] have both been proposed as a generalized framework for context-free grammars in two dimensions that accommodate general polyominoes. However, both array and puzzle grammars are permitted to have derivations that result in a self-intersecting polyomino, which is then simply discarded from the grammar's language. As Morita et al. [MYS83] note: "...[array grammars] come to have the ability to sense the

local shape of a host array as a kind of 'context', in spite of their apparent context-freeness.". This results in undecidable emptiness ("Is this grammar's language non-empty?") and PSPACE-hard membership ("Does this grammar's language contain polyomino p of size n?") problems for both definitions [MYS83, LNS92]. On the other hand, both problems are P-complete for one-dimensional CFGs [GHR95].

To address this issue, isotonic (also called isometric) array grammars were created as a restricted form of array grammars, where production rules are required to have equally sized left- and right-hand sides, where a special background symbol (usually #) fills all locations not in the polyomino. Though the membership problem for this restricted form is "only" NP-complete, rules with multiple symbols on the left-hand side still gives a form of context-sensitivity.

The intractability of simple problems such as emptiness and membership not only makes array grammars unsuitable for many applications, but also causes the smallest grammar problem to likely be $\mathrm{NP}^{\mathrm{NP}}$-complete or harder.

Chain codes The *picture descriptions* of Freeman [Fre61], later extended to *chain codes* by Feder [Fed68] and Maurer, Rozenberg, and Welzl [MRW82, SW85], reduce the general problem to the one-dimensional setting by converting the polyomino into a sequence of steps taken along a path through the polyomino. Chain codes only derive unlabeled polyominoes, but can be generalized by associating a label with each step in the path, i.e. "go up, write a", "go left, write b".

Note that any path through a tree-shaped polyomino must revisit some cells multiple times, and portions of the path must overlap. So any chain code deriving such a polyomino must have non-terminals corresponding to subpaths that overlap, unlike one-dimensional CFGs, where each non-terminal

corresponds to a distinct substring of the derived string. Then not only do chain codes fail to generalize one-dimensional CFGs, but the correspondence developed in Chapter 3 between non-terminals/substrings of a CFG and mixings/subassemblies of a SSAS cannot be generalized to two dimensions.

Combinatorics grammars Finally, we note that context-free grammars on polyominoes have been studied in the setting of combinatorics (see [DF92] and [DR04] for examples) as a basis for producing generating functions of polyomino families. However, these grammars are for unlabeled families of polyominoes and are designed to produce simple generating functions, rather than form a framework for a universal shape language.

4.3 Polyomino Context-Free Grammars

As examining previous work has shown, a definition of two-dimensional grammars that simultaneously inherits all of the useful properties of one-dimensional grammars remains an open problem. In the next section we present a new definition of two-dimensional grammars called *polyomino context-free grammars* or *polyomino grammars*. Polyomino grammars generalize one-dimensional grammars, achieving polynomial-time parsing and derivation, languages of arbitrary polyominoes, and non-terminals that correspond to subpolyominoes of derived polyominoes. As a tradeoff, polyomino grammars are limited to singleton languages.

In Chapter 3 we used grammar-based arguments to give algorithmic and complexity-theoretic results for the smallest SAS problem for one-dimensional assemblies. In Chapter 5 we do the same for two-dimensional assemblies. In both chapters, the only grammars considered are smallest grammars deriving a given string or polyomino, and so are deterministic and singleton. Then

because our application is limited to singleton languages, polyomino grammars are a more attractive generalization of one-dimensional grammars than existing definitions.

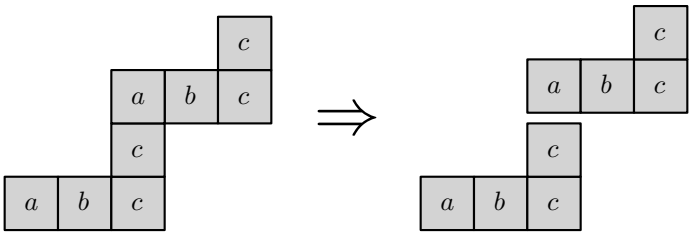
4.3.1 Deterministic CFGs are decompositions

Let G be a deterministic one-dimensional grammar with $L(G) = \{s\}$. Then every non-terminal symbol N of G derives a unique substring of s which we denote $\sigma(N)$. Beginning with the start symbol S, the rule $S \to AB$ replaces S with two non-terminals A and B where $\sigma(S) = \sigma(A)\sigma(B)$. That is, the rule $S \to AB$ *decomposes* $\sigma(S)$ into $\sigma(A)$ and $\sigma(B)$. Similarly, the rule $A \to CD$ decomposes $\sigma(A)$ into $\sigma(C)$ and $\sigma(D)$, implying $\sigma(S) = \sigma(C)\sigma(D)\sigma(B)$. In the case of a rule with more than two right-hand side symbols, e.g. $A \to B_1 \dots B_j$, $\sigma(A)$ is decomposed into k substrings.

In the other direction, any recursive decomposition of a string s can be interpreted as a deterministic grammar with singleton language $\{s\}$. Each substring s_1 appearing in the decomposition is equivalent to either a non-terminal ($|s_1| > 1$) or terminal ($|s_1| = 1$) symbol, and each decomposition of a substring into two or more substrings $s_2 s_3 \dots s_k$ is equivalent to a production rule replacing the symbol for s_1 with the symbols for s_2 through s_k.

4.3.2 Generalizing decompositions to polyominoes

Consider generalizing the decomposition interpretation of a deterministic one-dimensional grammar to a language $\{P\}$, with P a polyomino. Each non-terminal symbol N in G corresponds to a single subpolyomino of P (denoted $\sigma(N)$), beginning with the start symbol S. The production rule $S \to AB$ defines a decomposition of $\sigma(S) = P$ into two subpolyominoes $\sigma(A)$ and $\sigma(B)$ (see Figure 4.10).



$N_1 \to (N_2, (0,0))(N_2, (2,2))$

Figure 4.10: Each production rule in a PCFG deriving a single shape can be interpreted as a partition of the left-hand side non-terminal shape into a pair of connected shapes corresponding to the pair of right-hand side symbols.

In one dimension, simply specifying the left-to-right order of the right-hand side symbols is sufficient to define the decomposition. However, this crucially used a one-to-one mapping between left-to-right orderings and decompositions. Consider the case of four production rules $A \to ab$, $A \to ba$, $A \to \substack{a \\ b}$, $A \to \substack{b \\ a}$. These four rules cannot be encoded by distinct left-to-right orderings of the right-hand side symbols.

More generally, there may be an exponential number of ways to decompose a polyomino of n cells into two subpolyominoes. For instance, consider the possible ways to decompose a polyomino with dimensions $n/3 \times 3$ into two subpolyominoes, each consisting of a polyomino containing a $n/3 \times 1$ row of cells and some of the remaining $n/3$ cells. The remaining $n/3$ cells can be distributed in $2^{n/3} \approx 1.26^n$ possible ways.

We resolve the ambiguity by augmenting each right-hand side symbol with an (x, y) coordinate specifying the lower-leftmost cell of the symbol's subpolyomino relative to the lower-leftmost cell of the left-hand side non-terminal symbol. For example, the previous four production rules are written: $A \to (a, (0,0))(b, (1,0))$, $A \to (b, (0,0))(a, (1,0))$, $A \to (a, (0,1))(b, (0,0))$, $A \to (b, (0,1))(a, (0,0))$.

The addition of these coordinates enables a two-dimensional context-free

grammar definition with the property that for any polyomino p, any smallest grammar G with language $\{P\}$ is equivalent to a recursive decomposition of P. A rule $A \to (B_1, (x_1, y_1)) \ldots (B_j, (x_j, y_j))$ is equivalent to the partition of $\phi(A)$ into k subpolyominoes $B_1 \ldots B_j$ placed at offsets $(x_1, y_1) \ldots (x_j, y_j)$, respectively.

One problem remains: what if the polyominoes corresponding to a set of right-hand side symbols fails to form a valid polyomino? For instance, multiple right-hand side symbols may have the same coordinate or may fail to form a connected set of cells. We resolve this problem in a blunt way: any set of production rules with this property are simply not permitted, a property checkable in polynomial time by expanding the start symbol into the (unique) set of terminal symbols and checking their coordinates. This contrasts with the approach of array grammars that permit the formation of invalid polyominoes, but discard them from the language. The two approaches can be interpreted as "compile-time" (polyomino grammars) versus "run-time" (array grammars) validity testing, with run-time checking incurring a larger cost (PSPACE-hard derivation versus polynomial-time) in the worst case.

4.3.3 Definitions

Define a *polyomino context-free grammar (PCFG)* to be a quadruple $G = (\Sigma, \Gamma, S, \Delta)$. The set Σ is a set of *terminal symbols* and the set Γ is a set of *non-terminal symbols*. The symbol $S \in \Gamma$ is a special *start symbol*. Finally, the set Δ consists of *production rules*, each of the form $A \to (B_1, (x_1, y_1)) \ldots (B_j, (x_j, y_j))$ where $A \in \Gamma$ and unique to the rule, $B_i \in N \cup T$, and each (x_i, y_i) is a pair of integers.

A polyomino P can be derived by starting with S, the start symbol of G, and repeatedly replacing a non-terminal symbol with a set of non-terminal and

terminal symbols. The set of valid replacements is Δ, the production rules of G, where a non-terminal symbol A with lower-leftmost cell at (x, y) can be replaced with a set of symbols B_1 at $(x+x_1, y+y_1)$, B_2 at $(x+x_2, y+y_2)$, ..., B_j at $(x + x_j, y + y_j)$ if there exists a rule $A \to (B_1, (x_1, y_1)) \dots (B_j, (x_j, y_j))$. Additionally, the coordinates of every set of terminal symbols derivable starting with S must be connected and pairwise disjoint. The polyomino that can be derived using a grammar G is called the *language of* G, denoted $L(G)$.

4.4 The Smallest PCFG Problem

Problem 4.4.1 (The smallest PCFG)**.** *Given a polyomino P, find the smallest PCFG G such that $L(G) = \{P\}$.*

Unfortunately, most of the results achieved indicate that the smallest grammar problem for PCFGs give evidence for the intractability of approximating the problem within any reasonable factor. We analyze some known approaches approximation algorithms for the one-dimensional smallest grammar problem and show that they fail to generalize. On the positive side, we give an approximation algorithm that beats the naive algorithm by a logarithmic factor.

4.4.1 Generalizing smallest CFG approximations

The thesis of Eric Lehman [Leh02] and brief survey by Artur Jeż [Jeż13] provide an overview of the approaches used in known approximation algorithms for the smallest grammar problem on strings. Here we consider some results that suggest generalizing known approximation algorithms and analysis techniques for the smallest CFG problem are unlikely.

4.4.2 The $O(\log^3 n)$-approximation of Lehman

The $O(\log^3 n)$-approximation algorithm for the smallest CFG problem by Lehman [Leh02] centers around computing (approximately) the *shortest common superstring* of a set of strings, i.e. approximations for the *shortest common superstring problem*. Blum et al. [BJL+94] showed that the shortest common superstring problem is NP-hard and that a simple greedy algorithm that repeatedly merges the pair of strings with the greatest overlap is a 3-approximation. A number of inapproximability results have been shown, and currently a bound of 333/332 by Karpinski and Schmied [KS12] is the best known.

Given the approximation factor and simplicity of this $O(\log^3 n)$-approximation, it is interesting to ask whether the algorithm could be generalized to two dimensions. The generalized problem of the smallest common superstring problem is the smallest common superpolyomino problem, studied in Section 4.1.2. In Section 4.1.2, the problem was shown to be inapproximable within a $O(n^{1/3-\varepsilon})$ factor for any $\varepsilon > 0$ unless P = NP. Thus a generalization of the $O(\log^3 n)$-approximation of Lehman is unlikely to be both polynomial-time and a $O(n^{1/3-\varepsilon})$-approximation for the smallest PCFG problem.

4.4.3 The mk lemma

In nearly all of the best-known approximation ratio upper bounds for smallest grammar approximations (all those achieved by Lehman [Leh02]), the following result plays a key role:

Lemma 4.4.2 (mk lemma)**.** *Let G be the smallest grammar with language $\{s\}$. Then the number of distinct substrings of s with length k is at most $|G|k$ [Leh02].*

The power of the mk lemma lies in bounding the complexity of the string as a function of the size of the smallest grammar. When combined with results bounding the size of the grammar produced by an approximation algorithm to the complexity of the string, relative ratios between approximate grammars and smallest grammars are achieved.

For instance, the BISECTION algorithm recursively splits a string into two equal-size substrings. At the ith level of recursion, strings of length $n/2^{i-1}$ are split into strings of length $n/2^i$. The mk lemma says that if an input string s has a smallest grammar G, then the number of distinct strings in the dth level of recursion is not only at most 2^i, but also at most $|G|n/2^i$. Then a simple counting argument using an upper bound of 2^i for small i and $|G|n/2^i$ for large i gives a good approximation ratio of $O(\sqrt{n/\log n})$ for BISECTION.

Given the importance of this result in establishing the best-known approximation ratios for nearly all smallest CFG approximation algorithms, establishing a generalized (but weakened) version would be a good start in understanding the approximation ratios for any future smallest PCFG approximation algorithms.

Consider a $n \times n$ polyomino P with every cell a unique color. Every subpolyomino of P distinct, and for any polyomino with $\lfloor\sqrt{n}\rfloor$ cells, there exists a subpolyomino of P with the same shape. So P has $\Omega(\lambda^n)$ subpolyominoes of size $\lfloor\sqrt{n}\rfloor$, where $\lambda > 3.98$ by Barequet et al. [BMRR06]. As a result, the best possible mk-lemma-like result would be a $O(m3.98^k)$-lemma.

4.4.4 A $O(n/(\log^2 n/\log\log n))$-approximation

Here we give a simple algorithm that improves on the trivial $O(n/\log n)$-approximation, if only slightly. The idea is to exploit the relatively few distinct polyominoes below some size threshold to avoid using $\Theta(n)$ distinct non-

terminals to derive these small subpolyominoes.

Lemma 4.4.3. *For any $\varepsilon > 0$, the number of subpolyominoes of a polyomino P of size at most $0.2 \cdot \log|P|/\log|\Sigma(P)|$ is $O(|P|^{0.99})$.*

Proof. Klarner and Rivest [KR73] showed that the number of distinct monochrome polyominoes with n cells is $O(4.65^n)$. Let $f(P) = 4.65|\Sigma(P)|$. Then any polyomino P with alphabet $\Sigma(P)$ has at most $(f(P))^k$ subpolyominoes of size k. So the number of subpolyominoes of P with size at most k is $O(f(P)^k)$, since $\Sigma_{i=1}^{k} f(P)^i = O(f(P)^k)$, and the number of subpolyominoes of size at most $\log_{f(P)}(|P|^{0.99})$ is $O(|P|^{0.99})$. Rounding down the size of the polyominoes:

$$\begin{aligned}
\log_{f(P)}(|P|^{0.99}) &= \log(|P|^{0.99})/\log(f(P)) \\
&= 0.99 \cdot \log|P|/(\log 4.65 + \log|\Sigma(P)|) \\
&\geq 0.99 \cdot \log|P|/(2.22 + \log|\Sigma(P)|) \\
&\geq 0.99 \cdot \log|P|/(4 \cdot \log|\Sigma(P)|) \\
&\geq 0.2 \cdot \log|P|/(\log|\Sigma(P)|)
\end{aligned}$$

□

Theorem 4.4.4. *The smallest PCFG problem admits a $O(n/(\log^2 n/\log\log n)$-approximation, where n is the size of the input polyomino.*

Proof. We use Lemma 4.4.3 to develop the first part of an approximation algorithm: construct all polyominoes of size at most $0.2 \cdot \log P/\log|\Sigma(P)|$ using a grammar of size $O(|P|^{0.99})$. Next, create a naive grammar of size $O(|P|/k)$ assembling P, starting with subpolyominoes of size between $k/4$ and k. These two grammars combined yield a grammar G for P with size

$O(|P|/(\log|P|/\log|\Sigma(P)|)) + O(|P|^{0.99}) = O(|P|/(\log|P|/\log|\Sigma(P)|))$. Any smallest grammar for P has size at least $\log|P| + |\Sigma(P)|$, as it has $|\Sigma(P)|$ terminal symbols, and each rule creates a polyomino of size at most double the size of any right-hand side symbol.

Let $|P| = n$. Then $|G| = O(n/(\log n/\log|\Sigma(P)|))$ and any grammar for P has size $\Omega(\log n + |\Sigma(P)|)$. Consider two cases for $|\Sigma(P)|$: either $|\Sigma(P)| \leq \log^2 n$ or $\log^2 n < |\Sigma(P)|$. In the first case, the algorithm produces a grammar of size $O(n/(\log n/\log\log n))$ and any grammar has size $\Omega(\log n)$, so the approximation ratio is $O(n/(\log^2 n/\log\log n))$. In the second case, the algorithm produces a grammar of size $O(n)$ and any grammar has size $\Omega(\log^2 n)$, so the approximation ratio is $O(n/\log^2 n)$. □

Conjecture 4.4.5. *The smallest PCFG problem is $O(n^c)$-inapproximable for some $c > 0$.*

5

Two-Dimensional Staged Self-Assembly

Here we apply a similar analysis as Chapter 3 but for general two-dimensional assemblies, using our new definition of two-dimensional context-free grammars, called PCFGs, from Chapter 4. The work begins with the definitions of *simulation* in Section 5.1, used later to develop macrotile systems that trade a scale factor for other desirable properties.

In Section 5.2 we revisit the complexity of the smallest SAS problem. Recall that in Section 3.2 the smallest SAS problem was shown to be NP-complete, where membership in NP follows from inductively "filling in" the products of each mixing of a non-deterministically selected SAS. In two dimensions such an approach is not possible and we are forced to give a weaker non-tight result that the smallest SAS problem can be solved using only polynomial space, i.e. the problem lies in PSPACE.

The remainder of the chapter (Sections 5.3 through 5.6) explores the sep-

Portions of the work in this chapter have been published as [Win13].

aration between PCFGs and SASs. As detailed in Section 5.3, the separation bounds are more complex than in one dimension, with smaller PCFGs possible for some polyominoes, and smaller SASs for others. However, the takeaway of the results remains the same as in one dimension: staged assembly systems can be significantly smaller than grammars for some inputs, and grammars can never much smaller than staged assembly systems.

5.1 Definitions

Self-assembly systems in two dimensions use the definitions of Section 3.1, and differ from one-dimensional systems only in that they lack the restriction that north and south glues are null. Extending the concept of an assembly's *label string*, the *label polyomino* of a two-dimensional assembly is the polyomino with labeled cells corresponding to the locations and labels of the tiles in the assembly.

The results of Section 5.6.4 use the notion of a self-assembly system $\mathcal{S}'$ *simulating* a system $\mathcal{S}$ by carrying out the same sequence of mixings and producing a set of scaled assemblies. Formally, we say a system $\mathcal{S}' = (T', G', \tau, M', B')$ *simulates* a system $\mathcal{S} = (T, G, \tau, M, B)$ at *scale* b if there exist two functions f, g with the following properties:

(1) The function $f : (\Sigma(T') \cup \varnothing)^{b^2} \to \Sigma(T) \cup \varnothing$ maps the labels of $b \times b$ regions of tiles (called *blocks*) to a label of a tile in T. The empty label $\varnothing$ denotes no tile.

(2) The function $g : S' \to V$ maps a subset S' of the vertices of the mix graph M' to vertices of the mix graph M such that g is an isomorphism between the subgraph induced by S' in M' and the graph M.

(3) Let $P(v)$ be the set of products of the bin corresponding to vertex v in a mix graph. Then for each vertex $v \in M$ with $v' = g^{-1}(v)$, $P(v) = \{f(p) \mid p \in P(v')\}$.

Intuitively, f defines a correspondence between the b-scaled macrotiles in $\mathcal{S}'$ simulating tiles in $\mathcal{S}$, and g defines a correspondence between bins in the systems. Property (3) requires that f and g do, in fact, define correspondence between what the systems produce.

The self-assembly systems constructed in Sections 5.5 and 5.6 produce only *mismatch-free assemblies*: assemblies in which every pair of incident sides of two tiles in the assembly have the same glue. A system is defined to be *mismatch-free* if every product of the system is mismatch-free.

In Section 5.5 we construct systems that produce assemblies whose label polyominoes are scaled versions of other polyominoes, with some amount of "fuzz" in each scaled cell. A polyomino $P' = (S', L')$ is said to be a (c, d)-*fuzzy replica* of a polyomino $P = (S, L)$ if there exists a vector $\langle x_t, y_t \rangle$ with the following properties:

1. For each block of cells $\mathcal{S}'_{(i,j)} = \{(x, y) \mid x_t+di \leq x < x_t+d(i+1), y_t+dj \leq y < y_t + d(j + 1)\}$ (called a *supercell*), $\mathcal{S}'_{(i,j)} \cap S' \neq \varnothing$ if and only if $(i, j) \subseteq S$.

2. For each supercell $\mathcal{S}'_{(i,j)}$ containing a cell of P', the subset of *label cells* $\{(x, y) \mid x_t+di+(d-c)/2 \leq x < x_t+d(i+1)+(d-c)/2, y_t+dj+(d-c)/2 \leq y < y_t + d(j + 1) + (d - c)/2\}$ consists of c^2 cells of P', with all cells having identical label, called the *label of the supercell* and denoted $\mathcal{L}_{(i,j)}$.

3. For each supercell $\mathcal{S}'_{(i,j)}$, any cell that is not a label cell of $\mathcal{S}'_{(i,j)}$ has a common *fuzz label* in L'.

4. For each supercell $\mathcal{S}'_{(i,j)}$, the label of the supercell $\mathcal{L}'_{(i,j)} = P(i,j)$.

Properties (1) and (2) define how sets of cells in P' replicate individual cells in P, and the labels of these sets of cells and individual cells. Property (3) restricts the region of each supercell not in the label region to contain only cells with a common fuzz label. Property (4) requires that each supercell's label matches the label of the corresponding cell in P.

5.2 The Smallest SAS Problem

In Section 3.2 we defined the smallest SAS problem restricted one-dimensional staged self-assembly. Here we redefine the problem for two dimensions:

Problem 5.2.1 (Smallest SAS)**.** *Given an input polyomino P, find the smallest SAS using at most k glues that produces an assembly with label polyomino P.*

The problem in one dimension was shown to be NP-complete, giving a tight bound on the exact-solution complexity of the problem. Since this is the smallest SAS problem restricted to one-dimensional label polyominoes (strings), the two-dimensional generalization is also NP-hard.

Ideally, we would show that the generalization of the problem in two dimensions still remains in NP. Recall that the proof of Lemma 3.2.10 (the NP-hardness of the 1D smallest SAS problem) "filled in" the products of all bins explicitly, crucially using polynomial bounds on the number and size of product assemblies of each mixing. This idea fails in two dimensions, where the number of products in a mixing can be exponential in $|\mathcal{S}|$, a fact used in nearly every construction in this chapter. In place of storing products explicitly, we use machines that decide whether a given assembly is a product of a mix-

ing. Specifically, we give machines $\mathcal{P}$ and $\mathcal{C}$ for the *product* and *containment problems*:

Problem 5.2.2 (Product)**.** *Let $\mathcal{S}$ be a SAS with mixing v, and A', A be assemblies. 1. Are all products of v subassemblies of A? 2. Is A' a product of v?* [1]

Problem 5.2.3 (Containment)**.** *Given a SAS $\mathcal{S}$, mixing v, and assembly A, are all products of v subassemblies of A?*

Define the *depth* d of a mixing v in a mix graph G to be the longest path from a root of G to v, i.e. the longest path of nodes ending at v. We start by describing the machines, followed by a inductive proof of their correctness and containment in PSPACE, the set of problems solvable using only polynomial space.

Lemma 5.2.4. *The machines $\mathcal{P}$ and $\mathcal{C}$ decide the product problem and containment problem, respectively.*

Proof. The machines are correct for depth-0 bins trivially. Assume by the inductive hypothesis that they are correct for the problems on the parents of v, which have a smaller depth than v. Since $\mathcal{P}$ on bin v runs $\mathcal{C}$ on v (but not vice versa), we first prove $\mathcal{C}$ correct.

Observe that $\mathcal{C}$ accepts if and only if the following two conditions are met: 1. All products of parent bins of v are subassemblies of A (lines 2-4), and 2. For all assemblies A' with $|A'| \leq 2|A|$ with A' not a subassembly of A, A' is not an assemblable assembly of v (line 5). By condition (1), all reagent subassemblies of v have size at most $|A|$. Then if v has some product not a subassembly of A of size larger than $2|A|$, it has an assemblable assembly A' of size between $|A|$

[1] An affirmative answer to the product problem means that the answers to both subquestions are affirmative.

Algorithm 1 Machine $\mathcal{P}$ for the product problem

procedure $\mathcal{P}(\mathcal{S}, v, A', A)$
    Run $\mathcal{C}(\mathcal{S}, v, A)$, reject if it rejects.
    Non-deterministically select a set $I = \{B_1, \ldots, B_j\}$ of assemblies assembling A'.
    Non-deterministically select a mapping $f : I \to V$ to the parents of v.
    for all $B_i \in I$ **do**
        Run $\mathcal{P}(\mathcal{S}, f(B_i), B_i, A)$, reject if it rejects.
    end for
    Run $\mathcal{T}_1(\mathcal{S}, v, A', A)$, reject if it accepts.
    Accept.
end procedure

procedure $\mathcal{T}_1(\mathcal{S}, v, A', A)$ ▷ Determine if A' can assemble with any reagent assembly.
    Non-deterministically select an assembly A'' with $|A''| \leq |A|$.
    Non-deterministically select a parent bin p of v.
    Run $\mathcal{P}(S, p, A'', A)$, reject if it rejects.
    Reject if A'' does not assemble with A'.
    Accept.
end procedure

Algorithm 2 Machine $\mathcal{C}$ for the containment problem

procedure $\mathcal{C}(\mathcal{S}, v, A)$
    for all Parents p of v **do**
        Run $\mathcal{C}(\mathcal{S}, p, A)$, reject if it rejects.
    end for
    Run $\mathcal{T}_2(\mathcal{S}, v, A)$, reject if it accepts.
    Accept.
end procedure

procedure $\mathcal{T}_2(\mathcal{S}, v, A)$▷ Determine if there is an assemblable assembly of v not a subassembly of of A.
    Non-deterministically select an assembly A' with $|A'| \leq 2|A|$ and $A' \not\subseteq A$.
    Non-deterministically select a set $I = \{B_1, \ldots, B_j\}$ of assemblies assembling A'.
    Non-deterministically select a mapping $f : I \to V$ to the parents of v.
    for all $B_i \in I$ **do**
        Run $\mathcal{P}(\mathcal{S}, f(B_i), B_i, A)$, reject if it rejects.
    end for
    Accept.
end procedure

and $2|A|$. But by condition (2), no such assembly exists. So $\mathcal{C}$ accepts if and only if all products of v are subassemblies of A, and so decides the product problem.

Next, observe that $\mathcal{P}$ accepts if and only if the following two conditions are met: 1. All products of v and its parents are are subassemblies of A (line 2), and 2. A' is an assemblable assembly of v (lines 3-7) that does not assemble with any reagent of v (line 8). Condition (2) is equivalent to A' being a product of v, so conditions (1) and (2) are exactly the product problem, and $\mathcal{P}$ decides the problem. □

Using these machines on a non-deterministically chosen candidate smallest SAS yields a machine $\mathcal{M}$ for the smallest SAS problem at $\tau = 1$:

Algorithm 3 Machine $\mathcal{M}$ for the smallest SAS problem at $\tau = 1$

1: **procedure** $\mathcal{M}(P, k, n)$ ▷ n specifies $|\mathcal{S}| \leq n$.
2: Non-deterministically select an assembly A with label polyomino P.
3: Non-deterministically select a SAS $\mathcal{S}$ with $|\mathcal{S}| \leq \min(|P|, n)$ and single leaf bin l.
4: Run $\mathcal{C}(S, l, A)$, reject if it rejects.
5: Run $\mathcal{T}_3(\mathcal{S}, l, A)$, reject if it accepts.
6: Accept.
7: **end procedure**
1: **procedure** $\mathcal{T}_3(\mathcal{S}, l, A)$ ▷ Determines if l has a second product contained in A.
2: Non-deterministically select an assembly $A' \subsetneq A$.
3: Run $\mathcal{P}(S, l, A', A)$, reject if it rejects.
4: Accept.
5: **end procedure**

Theorem 5.2.5. *The smallest SAS problem at $\tau = 1$ is in* PSPACE.

Proof. First, we show $\mathcal{M}$ solves the smallest SAS problem at $\tau = 1$. The machine $\mathcal{M}$ accepts if and only if there exists a SAS $\mathcal{S}$ with single leaf bin l and assembly A with label polyomino P such that: 1. Every product of l is a subassembly of A (line 4), and 2. A is the only subassembly of A that

is a product (line 5). In other words, A is the only product of l. So $\mathcal{S}$ is a sufficiently small SAS producing an assembly with label polyomino P, and $\mathcal{M}$ decides the smallest SAS problem at $\tau = 1$.

Next, consider the complexity of $\mathcal{M}$. The machines $\mathcal{P}$ and $\mathcal{C}$ are NP^{NP} machines that recursively call themselves a number of times proportional to the depth d of the input node v. So an instance with an input polyomino P, whose smallest SAS $\mathcal{S}$ has size $|\mathcal{S}| \leq |P| = n$, can be solved in polynomial space. □

Theorem 5.2.6. *The smallest SAS problem is in* PSPACE.

Proof. Notice that even at $\tau > 1$, Lemma 3.2.5 holds, i.e. all products are subassemblies of the final product. The machines $\mathcal{P}$ and $\mathcal{C}$ use the $\tau = 1$ assumption in two ways. First, $\mathcal{P}$ and $\mathcal{C}$ assume that an assembly A' is an assemblable assembly of a mixing if it can be decomposed into reagents of the mixing (lines 3-7 of $\mathcal{S}$ and lines 3-7 of $\mathcal{T}_2$). Second, $\mathcal{P}$ assumes that an assemblable assemble A'' is a product of a mixing if it does not assemble with any reagent of the mixing (lines 2-5 of $\mathcal{T}_1$).

The first assumption can be eliminated by replacing lines 3-4 of $\mathcal{S}$ and lines 3-4 of $\mathcal{T}_2$ with a recursive decomposition of A' into pairs of τ-stable assemblies, effectively choosing an assembly process for A', and a check that the resulting subassemblies of A' are reagents of v. The second assumption can be eliminated by replacing lines 3-4 of $\mathcal{T}_1$ with a non-deterministic selection a recursive decomposition of A'' into pairs of τ-stable assemblies, and a check that the lowest level assemblies in the decomposition are products of parent mixings. Note that selecting only A'' with $|A''| \leq |A|$ is still sufficient, as $\mathcal{C}$ has already been used to verify that all products (and thus assemblable assemblies) of v are subassemblies of A.

The modifications yield machines $\mathcal{P}$ and $\mathcal{C}$ that decide the product and con-

tainment problems for general τ, and a machine $\mathcal{M}$ still using only polynomial space. □

One barrier to improving the upper bound is coping with the possibly exponential number of products in each mixing. Proving that they do not exist in smallest SASs is one approach, as is using some compressed representation of these sets that enables compact (polynomial) representation and operations for computing the product set of a bin given the product sets of parent bins. In any case, we conjecture that upper bound proved here is not tight and that the problem is NP-complete.

Conjecture 5.2.7. *The smallest SAS problem is* NP*-complete.*

However, even showing the problem lies in the polynomial hierarchy appears difficult. With the exception of the NP^{NP}-complete result of Bryans et al. [BCD+11], no problems in irreversible tile self-assembly have been shown to be higher in the polynomial hierarchy than the first level without being undecidable, so the smallest SAS problem may turn out to be unique in this regard.

5.3 The Landscape of Minimum PCFGs, SSASs, and SASs

In Chapter 3, the relationships between CFGs and the two varieties of self-assembly systems in one dimension were shown to be relatively simple: CFGs and SSASs are equivalent (up to a constant factor) and SASs can be better by up to a factor of $\Omega(\sqrt{n/\log n})$. The amount of "betterness" was formally called *separation* (see Section 3.4.4), and was defined as the maximum ratio across all strings of the smallest CFG over the smallest SAS for the string.

In this chapter we discover that the situation in two dimensions is significantly more complex. As we show in Section 5.4, PCFGs can be better than both SSASs and SASs by a factor of $\Omega(\log n / \log\log n)$. Because of this, we generalize *separation* to refer to the maximum ratio of a minimum-size instances of a specified pair of encodings, e.g. "the separation of CFGs *over* SASs is $\Omega(\sqrt{n/\log n})$".

In these terms we are now ready to describe the relationships of these families of minimum-size objects. In Section 5.4 we prove the aforementioned result that the separation of SASs and SSASs over PCFGs is $\Omega(\log n / \log\log n)$ (even for single-label polyominoes), and in Section 5.5 give a constructive proof that the separation is $O(\log n)$, i.e. that the lower bound is nearly tight for SASs.

In the other direction, we show in Section 5.6.1 that SASs can be much better than PCFGs, achieving a separation of $\Omega(n/\log\log n)$ even for single-label polyominoes. However, the construction exploits the limited form "context-sensitivity" found in self-assembly systems, and is an unusual serpentine shape with many thin, nearly-touching pieces. In Section 5.6.3 the same problem restricted to square assemblies is shown (using a far more complex construction) to have similar $\Omega(n/\log n)$ separation.

5.4 SAS over PCFG Separation Lower Bound

This result uses a set of shapes we call *n-stagglers*; an example is seen in Figure 5.1. The shapes consist of $\log n$ bars of dimensions $n/\log n \times 1$ stacked vertically atop each other, with each bar horizontally offset from the bar above it by some amount between $-(n/\log n - 1)$ and $n/\log n - 1$. We use the shorthand that $\log n = \lfloor \log n \rfloor$ for conciseness. Every sequence of

$\log n - 1$ integers, each between $-(n/\log n - 1)$ and $n/\log n - 1]$, encodes a unique staggler and by the pigeonhole principle, some n-staggler requires $\log((2n/\log n - 1)^{\log n - 1} = \Omega(\log^2 n)$ bits to specify.

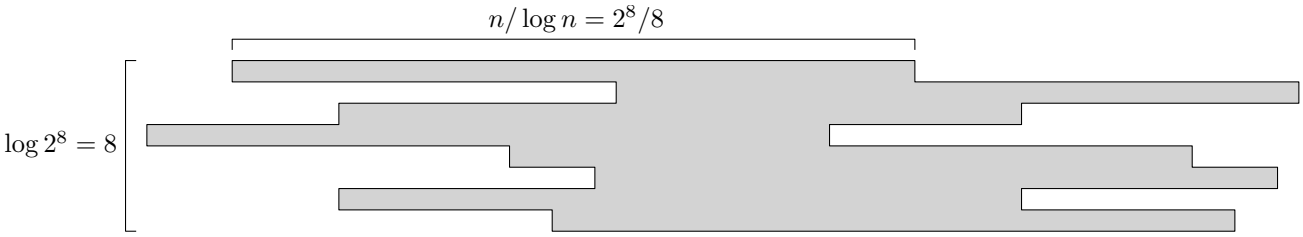


Figure 5.1: The 2^8-staggler specified by the sequence of offsets (from top to bottom) $-18, 13, 9, -17, -4, 12, -10$.

Lemma 5.4.1. *Any n-staggler can be derived by a PCFG of size $O(\log n)$.*

Proof. A set of $O(\log n)$ production rules deriving a bar (of size $\Theta(n/\log n) \times 1$) can be constructed by repeatedly doubling the length of the bar, using an additional $\log n$ rules to form the bar's exact length. Creating the stack of bars at their relative offsets can be described using a single rule with $\log n$ r.h.s. symbols. So an n-staggler can be derived by a PCFG of total size $O(\log n)$. □

Lemma 5.4.2. *For every n, there exists an n-staggler P such that any SAS or SSAS producing an assembly with label polyomino P has size $\Omega(\log^2 n/\log\log n)$.*

Proof. The proof is information-theoretic. Recall that more than half of all n-stagglers require $\Omega(\log^2 n)$ bits to specify. Now consider the number of bits contained in a SAS $\mathcal{S}$. Recall that $|\mathcal{S}|$ is the number of edges in the mix graph of $\mathcal{S}$. Any SAS can be encoded naively using $O(|\mathcal{S}|\log|\mathcal{S}|)$ bits to specify the mix graph, $O(|T|\log|T|)$ bits to specify the tile set, and $O(|\mathcal{S}|\log|T|)$ bits to specify the tile type at each leaf node of the mix graph. Because the number of tile types cannot exceed the size of the mix graph, $|T| \leq |\mathcal{S}|$. So the total number of bits needed to specify $\mathcal{S}$ (and thus the number of bits of information

contained in $\mathcal{S}$) is $O(|\mathcal{S}| \log |\mathcal{S}| + |T| \log |T| + |\mathcal{S}| \log |\mathcal{S}|) = O(|\mathcal{S}| \log |\mathcal{S}|)$. So some n-staggler requires a SAS $\mathcal{S}$ such that $O(|\mathcal{S}| \log |\mathcal{S}|) = \Omega(\log^2 n)$ and thus $|\mathcal{S}| = \Omega(\log^2 n / \log \log n)$. □

Theorem 5.4.3. *The separation of SASs and SSASs over PCFGs is* $\Omega(\log n / \log \log n)$.

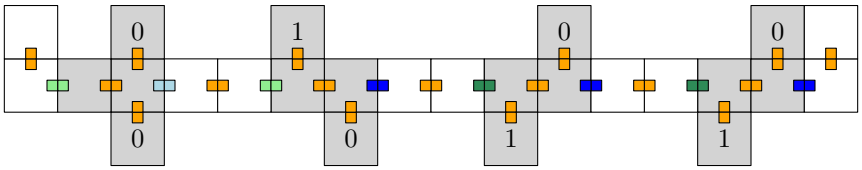
Proof. By the previous two lemmas, more than half of all n-stagglers require SASs and SSASs of size $\Omega(\log^2 n / \log \log n)$ and all n-stagglers have PCFGs of size $O(\log n)$. So the separation is $\Omega(\log n / \log \log n)$. □

Note that scaling the n-staggler by a c-factor produces a shape which is derivable by a CFG of size $O(\log n + \log c)$. That is, the result still holds for n-stagglers scaled by any amount polynomial in n. For instance, the $O(n)$-factor of the construction of Theorem 5.5.5.

At first it may not be clear how PCFGs achieve smaller encodings. After all, each rule in a PCFG G or mixing in SAS $\mathcal{S}$ specifies either a set of right-hand side symbols or parent bins and so has up to $O(\log |G|)$ or $O(\log |\mathcal{S}|)$ bits of information. The key is the coordinate describing the location of each right-hand side symbol. These offsets have up to $O(\log n)$ bits of information and in the case that G is small, say $O(\log n)$, each rule has a number of bits *linear* in the size of the PCFG!

5.5 SAS over PCFG Separation Upper Bound

Next we show that the separation lower bound of the last section is nearly large as possible by giving an algorithm for converting any PCFG G into a SSAS $\mathcal{S}$ with system size $O(|G| \log n)$ such that $\mathcal{S}$ produces an assembly that is a fuzzy replica of the polyomino derived by G. Before describing the full construction, we present approaches for efficiently constructing general binary counters and for simulating glues using geometry.



Increment 0011_b by 1, yielding 0100_b.

Figure 5.2: A binary counter row constructed using single-bit constant-sized assemblies. Dark blue and green glues indicate 1-valued carry bits, light blue and green glues indicate 0-valued carry bits.

The *binary counter row assemblies* used here are a generalization of those by Demaine et al. [DDF+08a] consisting of constant-sized bit assemblies, and an example is seen in Figure 5.2. Our construction achieves $O(\log n)$ construction of arbitrary ranges of rows and increment values, in contrast to the construction of [DDF+08a] that only produces row sets of the form $0, 1, \ldots, 2^{2^m} - 1$ that increment by 1. To do so, we show how to construct two special cases from which the generalization follows easily.

Lemma 5.5.1. *Let i, j, n be integers such that $0 \leq i \leq j < n$. There exists a SSAS of size $O(\log n)$ with a set of bins that, when mixed, assemble a set of $j - i + 1$ binary counter rows with values $i, i + 1, \ldots, j$ incremented by 1.*

Proof. Representing integers as binary strings, consider the prefix tree induced by the binary string representations of the integers i through j, which we denote $T_{(i,j)}$. The prefix tree $T_{(0,2^m-1)}$ is a complete tree of height m, and the prefix tree $T_{(i,j)}$ with $0 \leq i \leq j \leq 2^m - 1$ is a subtree of $T_{(0,2^m-1)}$ with $j - i + 1$ leaf nodes See Figure 5.3 for an example with $m = 4$.

Now let $n = 2^m - 1$. If $T_{(0,n)}$ is drawn with leaves in left-to-right order by increasing integer values, then the leaves of the subtree $T_{(i,j)}$ appear contiguously. So the subtree $T_{(i,j)}$ has at most $2 \log n$ internal nodes with one child forming the leftmost and rightmost paths in $T_{(i,j)}$. Furthermore, if one removes these two paths from $T_{(i,j)}$, the remainder of $T_{(i,j)}$ is a forest of complete trees

with at most two trees of each height and $2 \log n$ trees total.

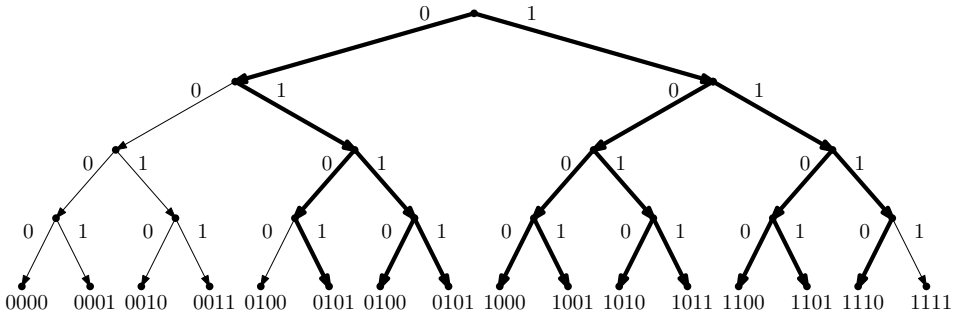


Figure 5.3: The prefix tree $T_{(0,15)}$ for integers 0 to $2^4 - 1$ represented in binary. The bold subtree is the prefix subtree $T_{(5,14)}$ for integers 5 to 14.

Note that a complete subtree of the prefix tree corresponds to a set of all possible 2^h suffixes of length h, where h is the height of the subtree. The leaves of such a subtree then correspond to the set of strings of length l with a specific prefix of length $l - h$ and any suffix of length h. For the assemblies we use the same geometry-based encoding of each bit as [DDF+08a], and a distinct set of glues used for each bit of the assembly encoding both the bit index and carry bit value from the previous bit.

Left and right bins. We build a mix graph (seen in Figure 5.4) consisting of two disjoint paths of bins (called *left bins* and *right bins*) that are used to iteratively assemble partial counter rows i and j by the addition of distinct constant-sized assemblies for each bit. The partial rows are used to produce the assemblies in the subtree and missing bit bins (described next). In the suffix trees, the bit strings of these assemblies are progressively longer subpaths of the leftmost and rightmost paths in the subtree of binary strings of the integers i to j.

Subtree bins. Assemblies in subtree bins correspond to assemblies encoding prefixes of binary counter row values. However, unlike left and right bins that

encode prefixes of only a single value, subtree bins encode prefixes of many binary counter values between i the j – namely a set of values forming a maximal complete subtree of the subtree of binary strings of integers from i to j, hence the name *subtree bins*. For example, if $i = 12$ and $j = 16$, then the set of binary strings for values 12 (01100_b) to 15 (01111_b) have a common prefix 011_b. In this case a subtree bin containing an assembly encoding the three bits 011 would be created. Since there are at most $2 \log n$ such complete subtrees, the number of subtree bins is at most this many. Creating each bin only requires a single mixing step of combining an assembly from a left or right bin with a single bit assembly, for example adding a 1-bit assembly to the left bin assembly encoding the prefix 01_b.

Missing bit bins. To add the bits not encoded by the assemblies in the subtree bins, we create sets of four constant-sized assemblies in individual *missing bit bins*. Since the assemblies in subtree bins encode bit string prefixes of sets of values forming complete subtrees, completing these prefixes with *any* suffix forms a bit string whose value is between i and j. This allows complete non-determinism in the bit assemblies that attach to complete the counter row, provided they properly handle carry bits. For every bit index missing in *some* subtree bin assembly, the four assemblies encoding the four possibilities for the input and carry values are assembled and placed into separate bins. When all bins are mixed, subtree assemblies mix non-deterministically with all possible assemblies from missing bit bins, producing all counter rows whose binary strings are found in the subtree. In total, up to $4 \log n$ missing bit bins are created, and each contains a constant-sized assembly and so requires constant work to produce.

The total number of total bins is clearly $O(\log n)$. Consider mixing the

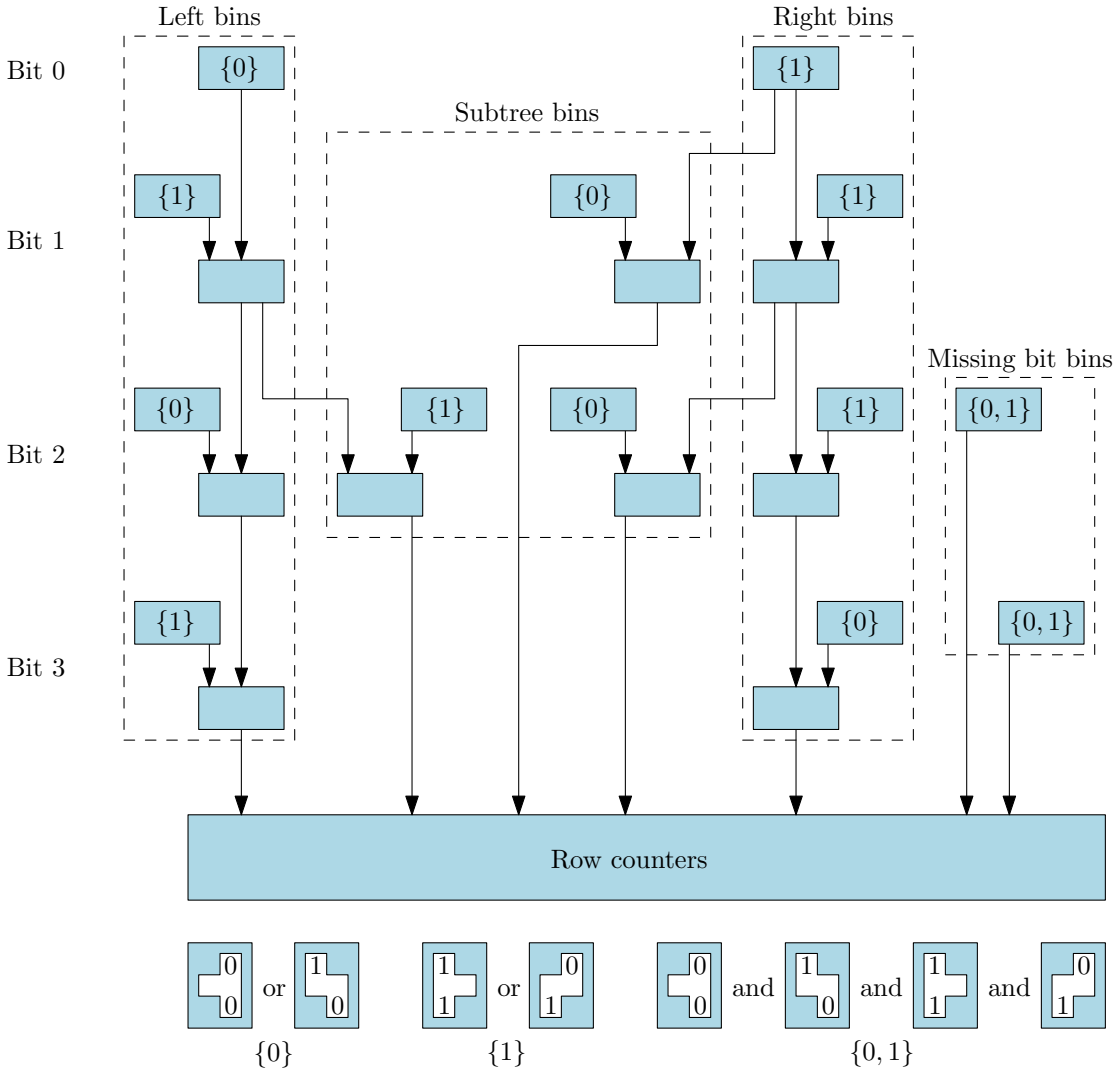


Figure 5.4: The mix graph constructed for the prefix subtree $T_{(5,14)}$ seen in Figure 5.3.

left and right bins containing completed counter rows for i and j, all subtree bins, and all missing bit bins. Any assembly produced by the system must be a complete binary counter row, as all assemblies are either already complete rows (left and right bins) or are partial assemblies (subtree bins and missing bit bins) that can be extended towards the end of the bit string by missing bit bin assemblies, or towards the start of the bit string by missing bit and then subtree bin assemblies. □

The second counter generalization is incrementing by non-unitary values:

Input bits			Output bits	
bth bit of k	bth bit	$(b-1)$st carry bit	bth bit	bth carry bit
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 5.1: All bit combinations for a binary adder incrementing n by k.

Lemma 5.5.2. *Let k, n be integers such that $0 \leq k \leq n$ and $n = 2^m$. There exists a SSAS of size $O(\log n)$ with a set of bins that, when mixed, assemble a set of 2^m binary counter rows with values $0, 1, \ldots, 2^m - 1$ incremented by k.*

Proof. For each row, the incremented value of the bth bit of the row depends on three values: the previous value of the bth bit, the carry bit from the $(b-1)$st addition, and the bth bit of k. The resulting output is a pair of bits: the resulting value of the bth bit and the bth carry bit (seen in Table 5.1).

Create a set of four $O(1)$-tile subassemblies for each bit of the counter, selecting from the first or second half of the combinations in Table 5.1, resulting in $4 \log n$ assemblies total. Each subassembly handles a distinct combination of the bth bit value of the previous row, $(b-1)$st carry bit, and bth bit value of k by encoding each possibility as a distinct glue. When mixed in a single bin, these subassemblies combine in all possible combinations and producing all counter rows from 0 to $2^m - 1$. □

Lemma 5.5.3. *Let i, j, k, n be integers such that $0 \leq i \leq j < n$ and $0 \leq k \leq n$. There exists a SSAS of size $O(\log n)$ with a set of bins that, when mixed, assemble a set of $j - i + 1$ binary counter rows with values $i, i+1, \ldots, j$ incremented by k.*

Proof. Combine the constructions used in the proofs of Lemmas 5.5.1 and 5.5.2 by using mixing sequences as in the proof of Lemma 5.5.1 and sets of four subassemblies encoding input, carry, and increment bit values as in the proof of Lemma 5.5.2. □

Theorem 8 of Demaine et al. [DDF+08a] describes how to reduce the number of glues used in a system by replacing each tile with a large *macrotile* assembly, and encoding the tile's glues via unique geometry on the macrotile's sides. We prove a similar result for labeled tiles, used for proving Theorems 5.5.5, 5.5.6, and 5.6.11.

Lemma 5.5.4. *Any mismatch-free $\tau = 1$ SAS (or SSAS) $\mathcal{S} = (T, G, \tau, M)$ can be simulated by a SAS (or SSAS) $\mathcal{S}'$ at $\tau = 1$ with $O(1)$ glues, system size $O(\Sigma(T)|T| + |\mathcal{S}|)$, and $O(\log |G|)$ scale.*

Proof. The proof is constructive. Produce a set of north *macroglue assemblies* for the glue set: $O(\log |G|) \times O(1)$ assemblies, each encoding the integer label of a glue i via a sequence of bumps and dents along the north side of the assembly representing the binary sequence of bits for i, as seen in Figure 5.5. All north macroglue assemblies share a pair of common glues: an *inner glue* on the west end of the south side of the assembly (green in Figure 5.5) and an *outer glue* on the west end of the north side of the assembly (blue in Figure 5.5). The null glue also has the sequence of bumps and dents (encoding 0), but lacking the outer glue. Repeating this process three more times yields sets of east, west, and south macroglue assemblies.

For each label $l \in \Sigma(T)$, repeat the process of producing the macroglue assemblies once using a tile set exclusively labeled l. Also produce a square $\Theta(\log |G|) \times \Theta(\log |G|)$ *core assembly*, with a single copy of the inner glue on the counterclockwise end of each face. Use the macroglue and core assemblies

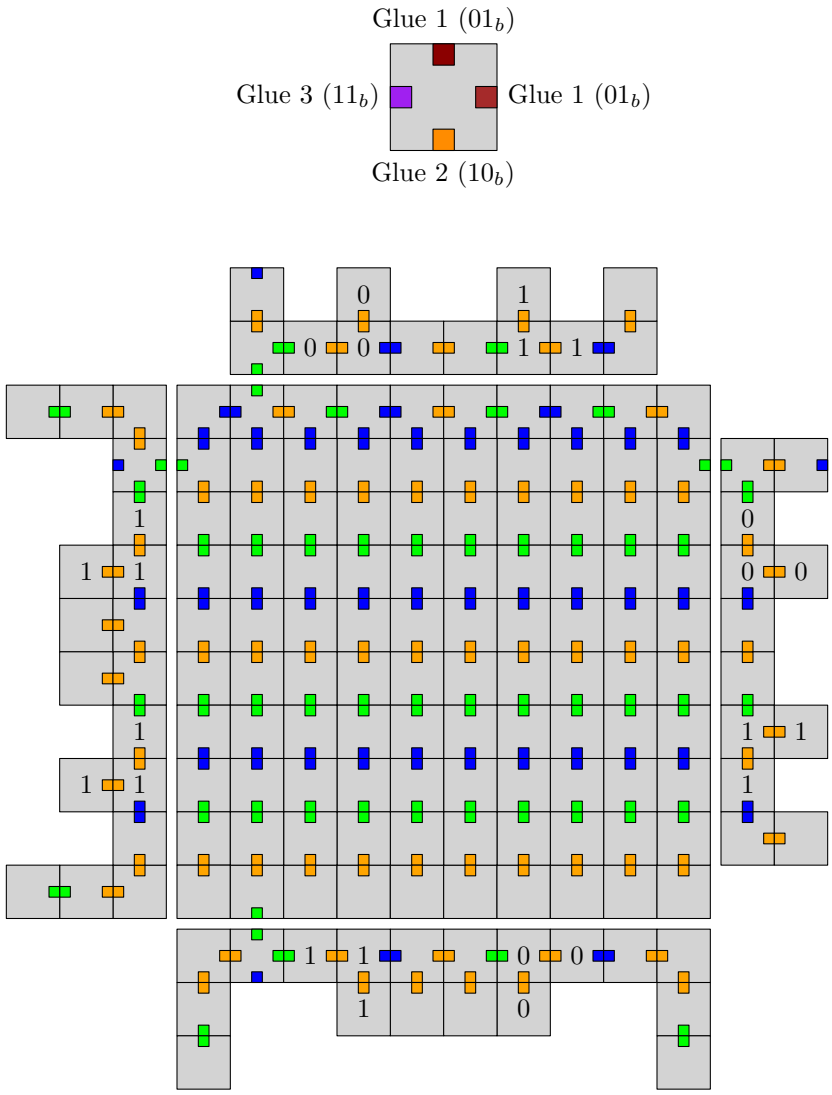


Figure 5.5: Converting a tile in a system with 7 glues to a macrotile with $O(\log |G|)$ scale and 3 glues. The gray label of the tile is used as a label for all tiles in the core and macroglue assemblies, with the 1 and 0 markings for illustration of the glue bit encoding.

to produce a set of *macrotiles*, one for each $t \in T$, consisting of a core assembly whose tiles have the label of t, and four glue assemblies encode the four glues of t and whose tiles have the label of t. Extend the mix graph M' of $\mathcal{S}'$ by carrying out the mixings of M but starting with the equivalent macrotiles. Define the simulation function f to map each macrotile to the label found on the macrotile, and the function g to take the portion of M' and g to be the

portion of the mix graph carrying out the mixings of $\mathcal{S}$.

The work done to produce the glue assemblies is $O(\Sigma(T)|G|)$, to produce the core assemblies is $O(\Sigma(T)\log\log|G|)$, and to produce the macrotiles is $O(|T|)$. Carrying out the mixings of $\mathcal{S}$ requires $O(|\mathcal{S}|)$ work. Since each macrotile is used in at least one mixing simulating a mixing in $\mathcal{S}$, $|T| \leq |\mathcal{S}|$. Additionally, $|G| \leq 4|T|$. So the total system size is $O(\Sigma(T)|G| + \Sigma(T)\log\log|G| + |T| + |\mathcal{S}|) = O(\Sigma(T)|T| + |\mathcal{S}|)$. □

Armed with these tools, we are ready to convert PCFGs into SSASs. Recall that in Section 5.4 we showed that in the worst case, converting a PCFG into a SSAS (or SAS) *must* incur an $\Omega(\log n/\log\log n)$-factor increase in system size. Here we achieve a $O(\log n)$-factor increase.

Theorem 5.5.5. *For any polyomino P with $|P| = n$ derived by a PCFG G, there exists a SSAS $\mathcal{S}$ with $|\mathcal{S}| = O(|G|\log n)$ producing an assembly with label polyomino P', where P' is a $(O(\log n), O(n))$-fuzzy replica of P.*

Proof. We combine the macrotile construction of Lemma 5.5.4, the generalized counters of Lemma 5.5.3, and a macrotile assembly invariant that together enable efficient simulation of each production rule in a PCFG by a set of $O(\log n)$ mixing steps.

Macrotiles. The macrotiles used are extended versions of the macrotiles in Lemma 5.5.4 with two modifications: a secondary, *reservoir macroglue* assembly on each side of the tile in addition to a primary *bonding macroglue*, and a thin *cage* of dimensions $\Theta(n) \times \Theta(\log n)$ surrounding each reservoir macroglue (see Figure 5.6).

Mixing a macrotile with a set of bins containing counter row assemblies constructed by Lemma 5.5.3 causes completed (and incomplete) counter rows

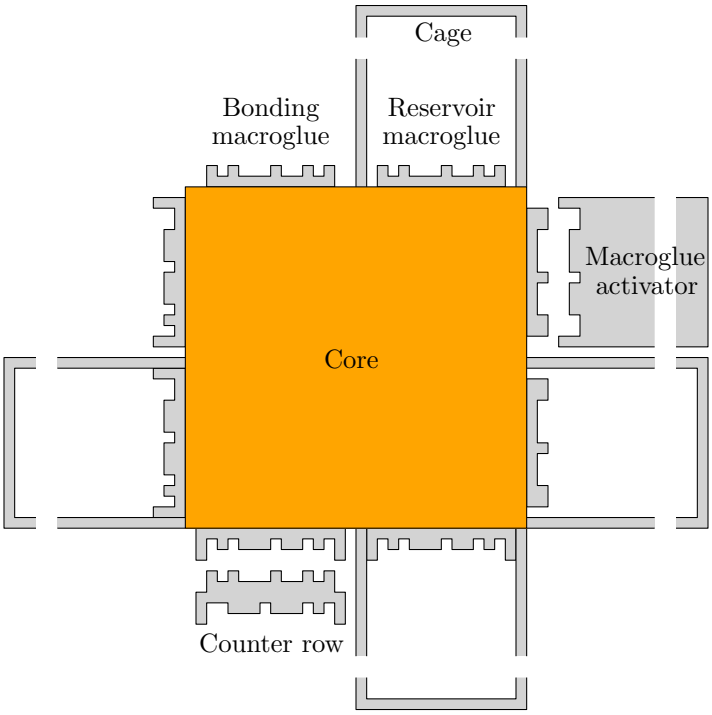


Figure 5.6: A macrotile used in converting a PCFG to a SAS, and examples of value maintenance and offset preparation.

to attach to the macrotile's macroglues. Because each macroglue's geometry matches the geometry of exactly one counter row, a partially completed counter row that attaches can only be completed with bit assemblies that match the macroglue's value. As a result, mixing the bin sets of Lemma 5.5.3 with an assembly consisting of macrotiles produces the same set of products as mixing a completed set of binary counter rows with the assembly.

An attached counter row effectively causes the macroglue's value to change, as it presents geometry encoding a new value and covers the macroglue's previous value. The cage is constructed to have height sufficient to accommodate up to n counter rows attached to the reservoir macroglue, but no more.

Because of the cage, no two macrotiles can attach by their bonding macroglues unless the macroglue has more than n counter rows attached. Alternatively, one can produce a thickened counter row with thickness sufficient to extend beyond the cage. We call such an assembly a *macroglue activator*, as it "ac-

tivates" a bonding macroglue to being able to attach to another promoted macroglue on another macrotile. Notice that a macroglue activator will never attach to a bonding macroglue's reservoir twin, as the cage is too small to contain the activator.

An invariant. Counter rows and activators allow precise control of two properties of a macrotile: the values of the macroglues on each side, and whether these glues are activated. In a large assembly containing many macroglues, the ability to change and activate glues allows precise encoding of how an assembly can attach to others. In the remainder of the construction we maintain the invariant that every macrotile has the same glue identity on all four sides, and any macrotile assembly consists of macrotiles with form a contiguous sequence of distinct glue values. For instance, the values $4, 5, 6, 7$ are permitted, but not $4, 5, 5, 6$ (not distinct) or $4, 5, 6, 8$ (not contiguous). These contiguous sequences are denoted $l..u$, e.g. $4..7$.

Production rule simulation. Consider a PCFG with non-terminal N and production rule $N \to (R_1, (x_1, y_1))(R_2, (x_2, y_2))$ and a SSAS with two bins containing assemblies A_1, A_2 with the label polyominoes of A_1 and A_2 being fuzzy replicas of the polyominoes derived by R_1 and R_2. Also assume A_1 and A_2 are assembled from the macrotiles just described, including obeying the invariant that the glue values are identical on all sides of each macrotile and are contiguous and distinct across the assembly. Let the glue values for A_1 and A_2 be $l_1..u_1$ and $l_2..u_2$, respectively.

First, we change the glue values of A_1 and A_2 depending on the values $l_1..u_1$ and $l_2..u_2$. Without loss of generality, assume $u_1 \leq u_2$. Then if $u_1 < l_2$, we increment all values of A_1 by $l_2 - u_1 - 1$. Otherwise $u_1 \geq l_2$ and we increment all values of A_2 by $u_1 - l_2 + 1$.

By Lemma 5.5.3, a set of row counters incrementing all glue values on a macrotile assembly can be produced using $O(\log n)$ work. In both cases the incrementation results in values that, if found on a common assembly, obey the invariants. Because each macrotile has a reservoir macroglue on each side, any bonding macroglue with an activator already attached has a reservoir macroglue that accepts the matching row counter, so each mixing has a single product and specifically no row counter products.

Next, select two cells, c_{R_1} and c_{R_2}, in the polyominoes derived by R_1 and R_2 adjacent in polyomino derived by N. Define the glue values of the two macrotiles in A_1 and A_2 forming the supercells mapped to c_{R_1} and c_{R_2} to be v_1 and v_2 and define the pair of coincident sides of c_{R_1} and c_{R_2} to be s_1 and s_2. We mix A_1 and A_2 with activators for sides s_1 and s_2 with values v_1 and v_2, respectively. Activators, because they are nearly rectangular save $O(\log n)$ cells of bit geometry, can also be produced using $O(\log n)$ work.

System scale. The PCFG P contains at most n production rules. Also, each step shifts glue identities by at most n (the number of distinct glues on the macrotile), so the largest glue identity on the final macrotile assembly is n^2. So we produce macrotiles with core assemblies of size $O(\log n) \times O(\log n)$ and cages of size $O(n)$. Assembling the core assemblies, cages, and initial macroglue assemblies of the macrotiles takes $O(|P| \log n + \log n + \log n) = O(|P| \log n)$ work, dominated by the core assembly production. Simulating each production rule of the grammar takes $O(\log n)$ work spread across a constant number of $O(\log n)$-sized sequences of mixings to produce sets of row counters and macroglue activators. □

Applying Lemma 5.5.4 to the construction (creating macrotiles of macrotiles) gives a constant-glue version of the previous theorem:

Theorem 5.5.6. *For any polyomino P with $|P| = n$ derived by a PCFG G, there exists a SSAS $\mathcal{S}'$ using $O(1)$ glues with $|\mathcal{S}'| = O(|G| \log n)$ producing an assembly with label polyomino P', where P' is a $(O(\log n \log\log n), O(n \log\log n))$-fuzzy replica of P.*

Proof. The construction of Theorem 5.5.5 uses $O(\log n)$ glues, namely for the counter row sub-construction of Lemma 5.5.3. With the exception of the core assemblies, all tiles of S have a common fuzz (gray) label, so creating macrotile versions of these tiles and carrying out all mixings involving these macrotiles and *completed* core assemblies is possible with $O(1 \cdot |T| + |\mathcal{S}|) = O(|\mathcal{S}|)$ mixings and scale $O(\log\log n)$. Scaled core assemblies of size $\Theta(n \log\log n) \times \Theta(n \log\log n)$ can be constructed using constant glues and $O(\log(n \log\log n)) = O(\log n)$ mixings, the same number of mixings as the unscaled $\Theta(n) \times \Theta(n)$ core assemblies of Theorem 5.5.5. So in total, this modified construction has system size $O(|S|) = O(|G| \log n)$ and scale $O(\log\log n)$. Thus it produces an assembly with label polyomino that is a $(O(\log n \log\log n), O(n \log\log n))$-fuzzy replica of P. □

The results in this section and Section 5.4 achieve a "one-sided" correspondence between the smallest PCFG and SSAS encoding a polyomino, i.e. the smallest PCFG is approximately an *upper bound* for the smallest SSAS (or SAS). Since the separation upper bound proof (Theorem 5.5.5) is constructive, the bound also yields an algorithm for converting a a PCFG into a SSAS.

5.6 PCFG over SAS and SSAS Separation Lower Bound

Here we develop a sequence of PCFGs over SAS and SSAS separation results, all within a polylogarithmic factor of optimal. As utilized in Theorem 5.6.11, the results in this section also hold for polynomially scaled versions of the polyominoes, showing that the "fuzziness" permitted in Section 5.5 was not unfair.

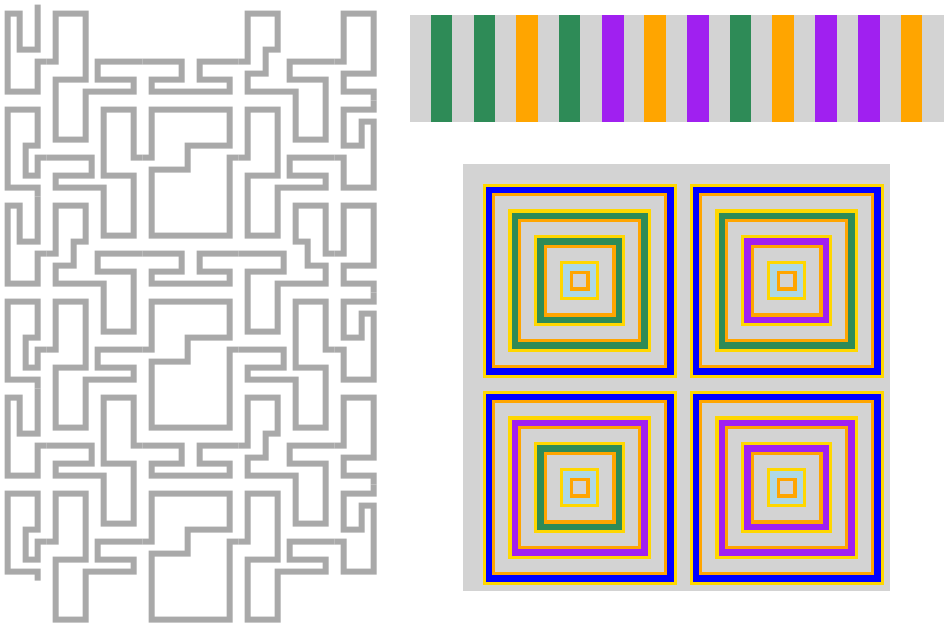


Figure 5.7: Two-bit examples of the weak (left), end-to-end (upper right), and block (lower right) binary counters used to achieve separation of PCFGs over SASs and SSASs in Section 5.6.

5.6.1 General shapes

In this section we describe an efficient system for assembling a set of shapes we call *weak counters*. An example of a rows in the original counter and macrotile weak counter are shown in Figure 5.8. These shapes are macrotile versions of the doubly-exponential counters found in [DDF$^+$08a] with three modifications:

1. Each row is a single path of tiles, and any path through an entire row uniquely identifies the row.

2. Adjacent rows do not have adjacent pairs of tiles, i.e. they do not touch.

3. Consecutive rows attach at alternating (east, west, east, etc.) ends.

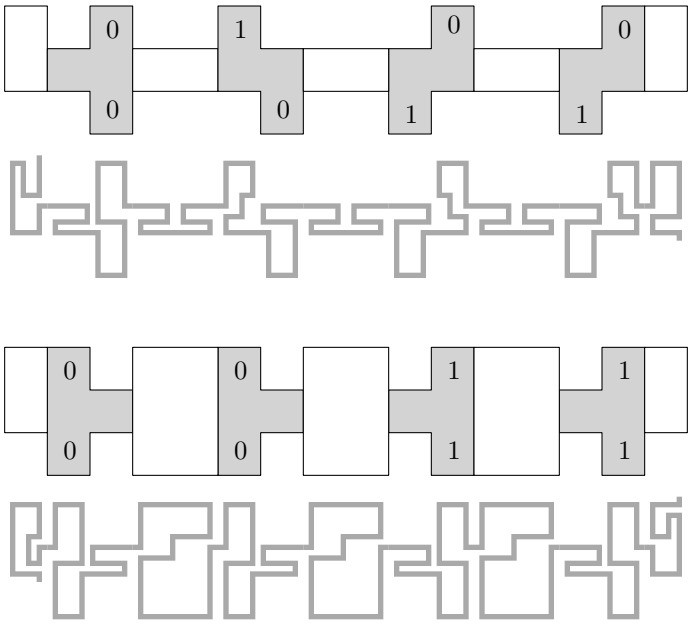


Figure 5.8: Zoomed views of increment (top) and copy (bottom) counter rows described in [DDF+08a] and the equivalent rows of a weak counter.

Figure 5.9 shows three consecutive counter rows attached in the final assembly. Each row of the doubly-exponential counter consists of small, constant-sized assemblies corresponding to 0 or 1 values, along with a 0 or 1 carry bit. We implement each assembly as a unique path of tiles and assemble the counter as in [DDF+08a], but using these path-based assemblies in place of the original assemblies. We also modify the glue attachments to alternate on east and west ends of each row. Because the rows alternate between incrementing a bit string, and simply encoding it, alternating the attachment end is trivial. Finally, note that adjacent rows only touch at their attachment, but

the geometry encoded into the row's path prevents non-consecutive rows from attaching.

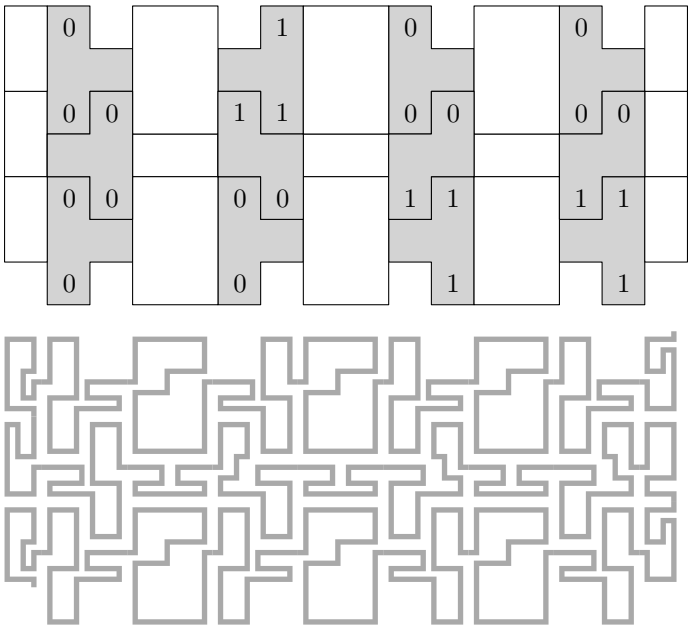


Figure 5.9: A zoomed view of adjacent attached rows of the counter described in [DDF+08a] (top) and the equivalent rows in the weak counter (bottom).

Lemma 5.6.1. *There exists a $\tau = 1$ SAS of size $O(b)$ that produces a 2^b-bit weak counter.*

Proof. The counter is an $O(1)$-scaled version of the counter of Demaine et al [DDF+08a]. They show that such an assembly is producible by a system of size $O(b)$. □

Lemma 5.6.2. *For any PCFG G deriving a 2^b-bit weak counter, $|G| = \Omega(2^{2^b})$.*

Proof. Define a *minimal row spanner* of row $\mathcal{R}_i$ to be a non-terminal symbol N of G with production rule $N \to (B, (x_1, y_1))(C, (x_2, y_2))$ such that the polyomino derived by N contains a path between a pair of easternmost and westernmost tiles of the row and the polyominoes derived by B and C do not. We claim that each row (trivially) has at least one minimal row spanner and each non-terminal of G is a minimal row spanner of at most one unique row.

First, suppose by contradiction that a non-terminal N is a minimal row spanner for two distinct rows. Because N is connected and two non-adjacent rows are only connected to each other via an intermediate row, N must be a minimal row spanner for two adjacent rows $\mathcal{R}_i$ and $\mathcal{R}_{i+1}$. Then the polyominoes of B and C each contain tiles in both $\mathcal{R}_i$ and $\mathcal{R}_{i+1}$, as otherwise either C or B is a minimal row spanner for $\mathcal{R}_i$ or $\mathcal{R}_{i+1}$.

Without loss of generality, assume B contains a tile at the end of $\mathcal{R}_i$ not adjacent to $\mathcal{R}_{i+1}$. But B also contains a tile in $\mathcal{R}_{i+1}$ and (by definition) is connected. So B contains a path between the east and west ends of row $\mathcal{R}_i$, and thus N is a not a minimal row spanner for r_i. So N is a minimal row spanner for at most one row.

Next, note that the necessarily-serpentine path between a pair of easternmost and westernmost tiles of a row in a minimal row spanner uniquely encodes the row it spans. So the row spanned by a minimal row spanner is unique.

Because each non-terminal of G is a minimal row spanner for at most one unique row, G must have at least 2^{2^b} non-terminal symbols and total size $\Omega(2^{2^b})$. □

Theorem 5.6.3. *The separation of PCFGs over $\tau = 1$ SASs for single-label polyominoes is $\Omega(n/(\log\log n)^2)$.*

Proof. By the previous two lemmas, there exists a SAS of size $O(b)$ producing a b-bit weak counter, and any PCFG deriving this shape has size $\Omega(2^{2^b})$. The assembly itself has size $n = \Theta(2^{2^b}b)$, as it consists of 2^{2^b} rows, each with b subassemblies of constant size. So the separation is $\Omega((n/b)/b) = \Omega(n/(\log\log n)^2)$. □

In [DDF+08a], the $O(\log \log n)$-sized SAS constructing a $\log n$-bit binary counter repeatedly doubles the length of each row (i.e. number of bits in the counter) using $O(1)$ mixings per doubling. Achieving such a technique in a SSAS seems impossible, but a simpler construction producing a b-bit counter with $O(b)$ work can be done by using a unique set of $O(1)$ glues for each bit of the counter. In this case, mixing these reusable elements along with a previously-constructed pair of first and last counter rows creates a single mixing assembling the entire counter at once. Modifying the proof of Theorem 5.6.3 to use this construction gives a similar separation for SSASs:

Corollary 5.6.4. *The separation of PCFGs over $\tau = 1$ SSASs for single-label polyominoes is $\Omega(n/\log^2 n)$.*

5.6.2 Rectangles

For the weak counter construction, the lower bound in Lemma 5.6.2 depended on the poor connectivity of the weak counter polyomino. This dependency suggests that such strong separation ratios may only be achievable for special classes of "weakly connected" or "serpentine" shapes. Restricting the set of shapes to rectangles or squares while keeping an alphabet size of 1 gives separation of at most $O(\log n)$, as any rectangle of area n can be derived by a PCFG of size $O(\log n)$. But what about rectangles with a constant-sized alphabet? In this section we achieve surprisingly strong separation of PCFGs over SASs and SSASs for rectangular constant-label polyominoes, nearly matching the separation achieved for single-label general polyominoes.

The construction. The polyominoes constructed resemble binary counters whose rows have been arranged in sequence horizontally, and we call them *b-bit end-to-end counters*. Each row of the counter is assembled from tall, thin

macrotiles (called *bars*), each containing a *color strip* of orange, purple, or green. The color strip is coated on its east and west faces with gray geometry tiles that encode the bar's location within the counter.

Figure 5.10: The rectangular polyomino used to show separation of PCFGs over SASs when constrained to constant-label rectangular polyominoes. The green and purple color strips denote 0 and 1 bits in the counter.

Each row of the counter has a sequence of green and purple *display bars* encoding a binary representation of the row's value and flanked by orange *reset bars* (see Figure 5.11). An example for $b = 2$ bits can be seen in Figure 5.10.

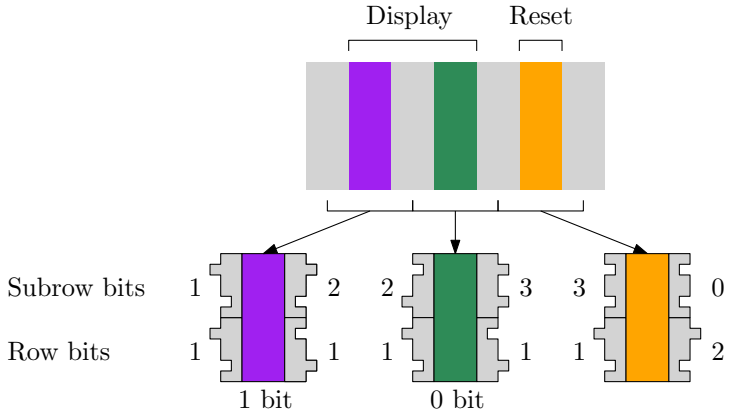


Figure 5.11: The implementation of the vertical bars in row 2 (01_b) of an end-to-end counter.

Each bar has dimensions $O(1) \times 3(\log_2 b + b + 2)$, sufficient for encoding two pieces of information specifying the location of the bar within the assembly. The *row bits* specify which row the bar lies in (e.g. the 7th row). The *subrow bits* specify where within the row the bar lies (e.g. the 4th bit). The subrow value starts at 0 on the east side of a reset bar, and increments through the display bars until reaching $b + 1$ on the west end of the next reset bar. Bars of all three types with row bits ranging from 0 to $2^b - 1$ are produced.

Efficient assembly. The counter is constructed using a SAS of size $O(b)$ in two phases. First, sequences of $O(b)$ mixings are used to construct five families of bars: reset bars, 0-bit display bars resulting from a carry, 0-bit display bars without a carry, 1-bit display bars resulting from a carry, 1-bit display bars without a carry. The mixings produce five bins, each containing *all* of the bars of a family. These five bins are then combined into a final bin where the bars attach to form the $\Theta(2^b) \times \Theta(b)$ rectangular assembly. The five families are seen in Figure 5.12.

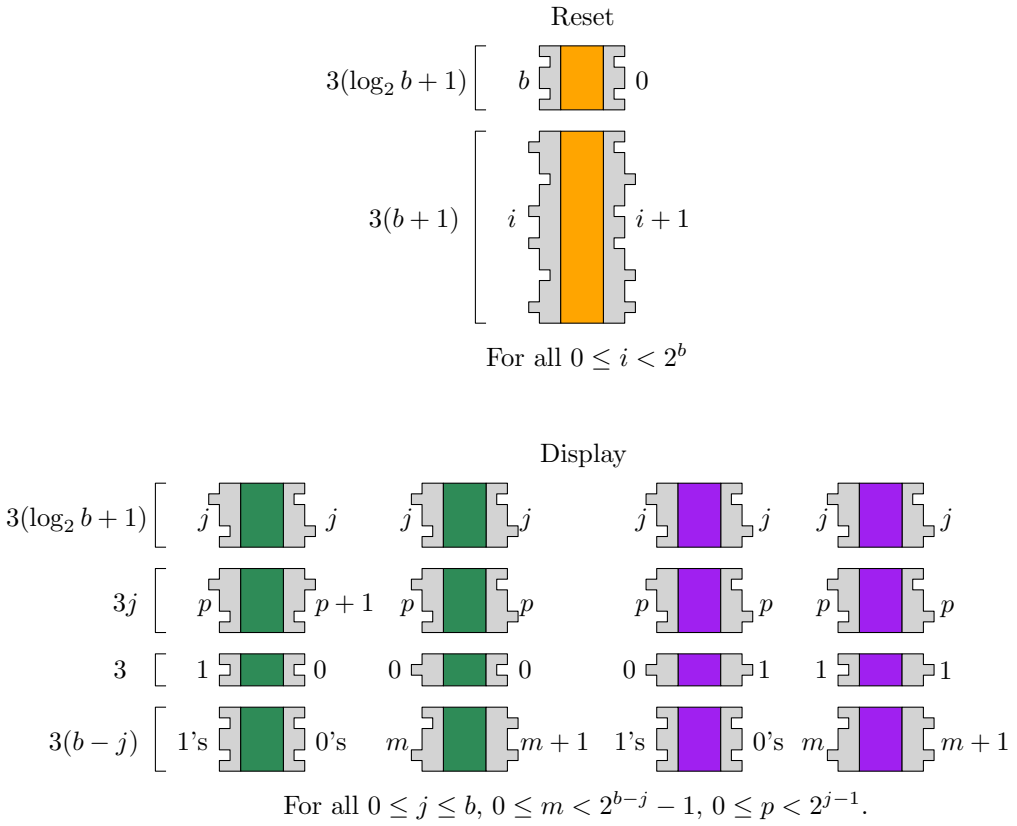


Figure 5.12: The decomposition of bars used assemble a b-bit end-to-end counter.

Efficient $O(b)$ assembly is achieved by careful use of the known approach of non-deterministic assembly of single-bit assemblies as done in [DDF+08a]. Assemblies encoding possible input bit and carry bit value combinations for each row bit and subrow bit are constructed and mixed together, and the

resulting products are every valid set of input and output bit strings, i.e. every row of a binary counter assembly.

As a warmup, consider the assembly of all reset bars. For these bars, the west subrow bits encode b and the east subrow bits encode 0. The row bits encode a value i on the west side, and $i+1$ on the east side, for all i between 0 and $2^b - 1$. Constructing all such bars using $O(b)$ work is straightforward. For each of the $\log_2 b + 1$ subrow bits, create an assembly where the west and east bits are 1 and 0 respectively, *except* for the most significant bit (bit $\log_2 b + 1$), where the west and east bits are both 0.

For the row bits we use the same technique as in [DDF+08a] and extended in Lemma 5.5.2: create a constant-sized set of assemblies for each bit that encode input and output value and carry bits. For bits 1 through $b - 1$ (zero-indexed) create four assemblies corresponding to the four combinations of value and carry bits, for bit 0 create two assemblies corresponding to value bits (the carry bit is always 1), for bit b create three assemblies corresponding to all combinations except both value and carry bits valued 1, and for bit $b + 1$ create a single assembly with both bits valued 0. Give each bit assembly a unique south and north glue encoding its location within the bar and carry bit value, and give all bit assemblies a common orange color strip. Mixing these assemblies produces all reset bars, with subrow west and east values of b and 0, and row values i and $i + 1$ for all i from 0 to $2^b - 1$.

In contrast to producing reset bars, producing display bars is more difficult. The challenge is achieving the correct color strip relative to the subrow and row values. Recall that the row value i locates the bar's row and the subrow value j locates the bar within this row. So the correct color strip for a bar is green if the jth bit of i is 0, and purple if the jth bit of i is 1.

We produce four families of display bars, two for each value of the jth bit

of i. Each sub-family is produced by mixing a subrow assembly encoding j on both east and west ends with three component assemblies of the row value: the *least significant bits (LSB) assembly* encoding bits 1 through $j-1$ of i, the *most significant bits (MSB) assembly* encoding bits $j+1$ through b of i, and the constant-sized jth bit assembly. This decomposition is seen in the bottom half of Figure 5.12.

The four families correspond to the four input and carry bit values of the jth bit. These values determine what collections of subassemblies should appear in the other two components of the row value. For instance, if the input and carry bit values are both 1, then the LSB assembly must have all 1's on its west side (to set the jth carry bit to 1) and all 0's on its east side. Similarly, the MSB assembly must have some value p encoded on its west side and the value $p+1$ encoded on its east side, since the jth bit and and jth carry bit were both 1, so the $(j+1)$st carry bit is also 1.

Notice that each of the four families has b sub-families, one for each value of j. Producing all sub-families of each family is possible in $O(b)$ work by first recursively producing a set of b bins containing successively larger sets of MSB and LSB assemblies for the family. Then each sub-family can be produced using $O(1)$ amortized work, mixing one of b sets of LSB assembly sub-families, one of b sets of MSB assemblies, and the jth bit assembly together. For instance, one can produce the set of b sets of MSB assemblies encoding pairs of values p and $p+1$ on bits $b-1$ through b, $b-2$ through b, etc. by producing the set on bits k through b, then adding four assemblies to this bin (those encoding possible pairs of inputs to the $(k-1)$st bit) to produce a similar set on bits $k-1$ through b.

Lemma 5.6.5. *There exists a $\tau = 1$ SAS of size $O(b)$ that produces a b-bit end-to-end counter.*

Proof. This follows from the description of the system. The five families of bars can each be produced with $O(b)$ work and the bars can be combined together in a single mixing to produce the counter. So the system has total size $O(b)$. □

Lemma 5.6.6. *For any PCFG G deriving a b-bit end-to-end counter, $|G| = \Omega(2^b)$.*

Proof. Let G be a PCFG deriving a b-bit end-to-end counter. Define a *minimal row spanner* to be a non-terminal symbol N with production rule $N \to (B, (x_1, y_1))(C, (x_2, y_2))$ such that the polyomino derived by N (denoted p_N) horizontally spans the color strips of all bars in row $\mathcal{R}_i$ including the reset bar at the end of the row, while the polyominoes derived by B and C (denoted p_B and p_C) do not. Consider the bounding box D of these color strips (see Figure 5.13).

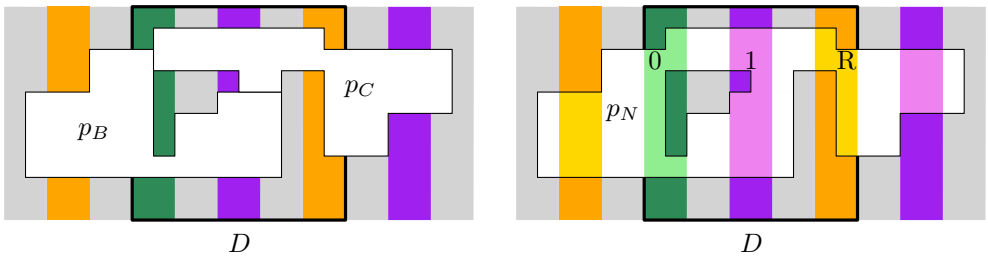


Figure 5.13: A schematic of the proof that a non-terminal is a minimal row spanner for at most one unique row. (Left) Since p_B and p_C can only touch in D, their union non-terminal N must be a minimal row spanner for the row in D. (Right) The row's color strip sequence uniquely determines the row spanned by N (01_b).

Without loss of generality, p_B intersects the west boundary of D but does not reach the east boundary, while p_C intersects the east boundary but does not reach the west boundary, so any location at which p_B and p_C touch must lie in D. Then any row spanned by p_N and not spanned by p_B or p_C must lie in D, since spanning it requires cells from both p_B and p_C. So p_N is a minimal row spanner for at most one row: row $\mathcal{R}_i$.

Because the sequence of green and purple display bars found in D is distinct and separated by display bars in other rows by orange reset bars, each minimal row spanner spans a unique row $\mathcal{R}_i$. Then since each non-terminal is a spanner for at most one unique row, G must have 2^b non-terminal symbols and $|G| = \Omega(2^b)$. □

Theorem 5.6.7. *The separation of PCFGs over $\tau = 1$ SASs for constant-label rectangles is $\Omega(n/\log^3 n)$.*

Proof. By construction, a b-bit end-to-end counter has dimensions $\Theta(2^b b) \times \Theta(b)$. So $n = \Theta(2^b b^2)$ and $b = \Theta(\log n)$. Then by the previous two lemmas, the separation is $\Omega((n/b^2)/b) = \Omega(n/\log^3 n)$. □

We also note that a simple replacement of orange, green, and purple color strips with distinct horizontal sequences of black/white color substrips yields the same result but using fewer distinct labels.

5.6.3 Squares

The rectangular polyomino of the last section has exponential aspect ratio, suggesting that this shape requires a large PCFG because it approximates a patterned one-dimensional assemblies reminiscent of those in [DEIW12]. Creating a polyomino with better aspect ratio but significant separation is possible by extending the polyomino's labels vertically. For a square this approach gives a separation of PCFGs over SASs of $\Omega(\sqrt{n}/\log n)$, non-trivial but far worse than the rectangle.

The construction. In this section we describe a polyomino that is square but contains an exponential number of distinct subpolyominoes such that each subpolyomino has a distinct "minimal spanner", using the language of the

proof of Lemma 5.6.6. These subpolyominoes use circular versions of the vertical bars of the construction in Section 5.6.2 arranged concentrically rather than adjacently. We call the polyomino a *b-bit block counter*, and an example for $b = 2$ is seen in Figure 5.14.

Each *block* of the counter is a $\Theta(b^2) \times \Theta(b^2)$ square subpolyomino encoding a sequence of b bits via a sequence of concentric rectangular *rings* of increasing size. Each ring has a *color loop* encoding the value of a bit, or the start or end of the bit sequence (the interior or exterior of the block, respectively). The color loop actually has three subloops, with the center loop's color (green, purple, light blue, or dark blue in Fig. 5.14) indicating the bit value or sequence information, and two surrounding loops (light or dark orange in Fig. 5.14) indicating the interior and exterior sides of the loop.

Efficient assembly of blocks. Though each counter block is square, they are constructed similarly to the end-to-end counter rows of Section 5.6.2 by assembling the vertical *bars* of each ring together into horizontal stacks of assemblies. Horizontal *slabs* are added to "fill in" the remaining portions of each block.

The bars are identical to those found in Section 5.6.2 with three modifications (seen in Figure 5.15). First, each bar has additional height according to the value of the subrow bits (8 tiles for every increment of the bits). Second, each bar has four additional layers of tiles on the side (east or west) facing the interior of the block, with *color bits* at the north and south ends of the side encoding three values: 11_b (if the center color subloop is purple, a 1-bit), 00_b (if the center color subloop is green, a 0-bit), or 01_b (if the center color subloop is dark blue, the end of the bit sequence). The additional layers are used to fill in gaps between adjacent rings left by protruding geometry, and the bit values

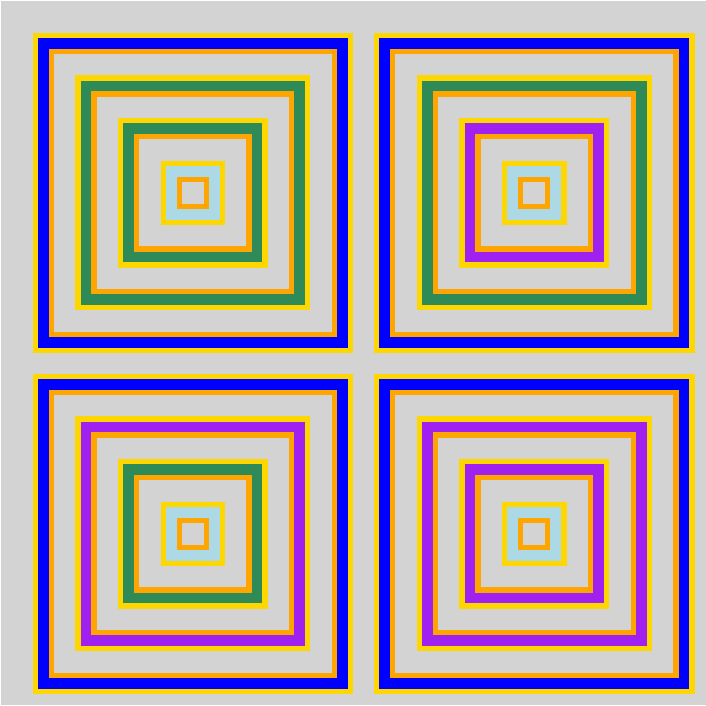


Figure 5.14: The square polyomino used to show separation of PCFGs over SASs when constrained to constant-label square polyominoes. The green and purple color subloops denote 0 and 1 bits in the counter, while the light and dark blue color subloops denote the start and end of the bit string. The light and dark orange color subloops indicate the interior and exterior of the other subloops.

are used to control the attachment of the horizontal slabs of each ring.

Third, the reset bars used in Section 5.6.2 are replaced with two kinds of bars: *start bars* and *end bars*, seen in Figure 5.17. End bars form the outermost rings of each block, and the start bars form the square cores of each block. Both start and end bars "reset" the subrow counters, and the east end bars increment the row value.

Recall that the vertical bars of the end-to-end counter in Section 5.6.2 were constructed using $O(b)$ total work by amortizing the constructing sub-families of MSB and LSB assemblies for each subrow value j. We use the same trick here for these assemblies as well as the new assemblies on the north and south ends of each bar containing the color bits. In total there are twelve families

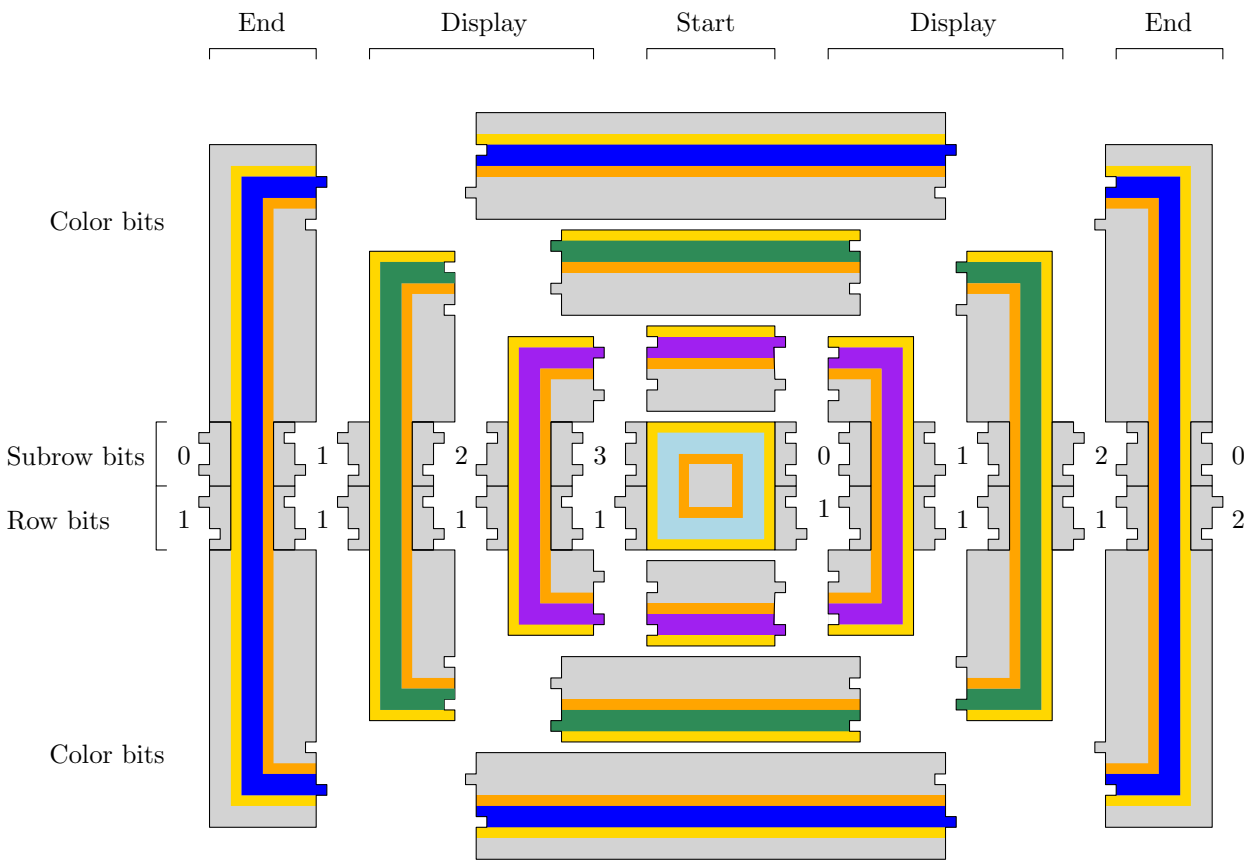
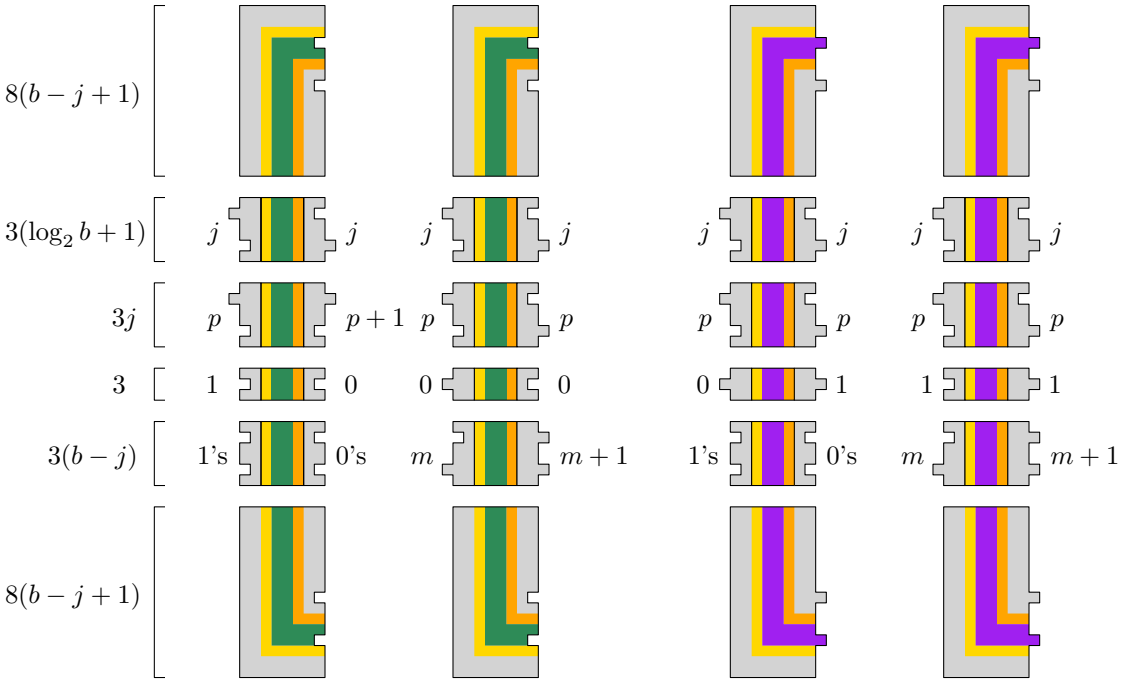


Figure 5.15: The implementation of rings in each block of the block counter.

of vertical bar assemblies (four families of west display bars, four families of east display bars, and two families each of start and end bars), and each is assembled using $O(b)$ work.

Finally, the horizontal slabs of each ring are constructed as six families, each using $O(b)$ work, as seen in Figure 5.18.

Efficient assembly of the counter. Once the families of vertical bars and horizontal slabs are assembled into blocks, we are ready to arrange them into a completed counter. Each row of the counter has $\sqrt{2^b} = 2^{b/2}$ blocks. So assuming b is even, the $b/2$ least significant bits of the westernmost block of each row are 0's, and of the easternmost block are 1's. Before mixing the vertical bar families together, we "cap" the east end bar of each block at



For all $0 \leq j \leq b$, $0 \leq m < 2^{b-j} - 1$, $0 \leq p < 2^{j-1}$.

Figure 5.16: The decomposition of vertical display bars used to assemble blocks in the b-bit block counter. Only the west bars are shown, with east bars identical but color bits and color loops reflected.

the east end of a row by constructing a set of thin assemblies (right part of Figure 5.19) and mixing them with the family of east end bars.

After this modification to the east end bar family, mixing all vertical bar families results in $2^{b/2}$ assemblies, each forming most of a row of the block counter. Mixing these assemblies with the families of horizontal slabs results in a completed set of block counter rows, each containing $2^{b/2}$ square assemblies with dimensions $\Theta(b^2) \times \Theta(b^2)$, forming $2^{b/2}$ rectangles with dimensions $\Theta(2^{b/2}b^2) \times \Theta(b^2)$.

To arrange the rows vertically into a complete block counter, a vertically-oriented version of the end-to-end counter of Section 5.6.2 with geometry instead of color strips (left part of Fig. 5.19) is assembled and used as a "backbone" for the rows to attach into a combined assembly. This modified end-

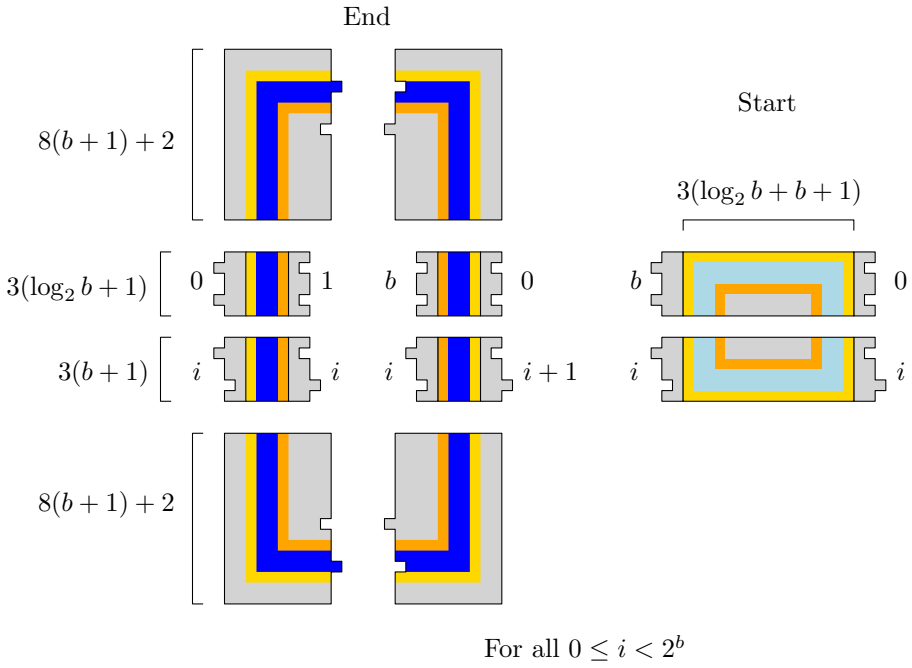


Figure 5.17: The decomposition of vertical start and end bars used to assemble blocks in the b-bit block counter.

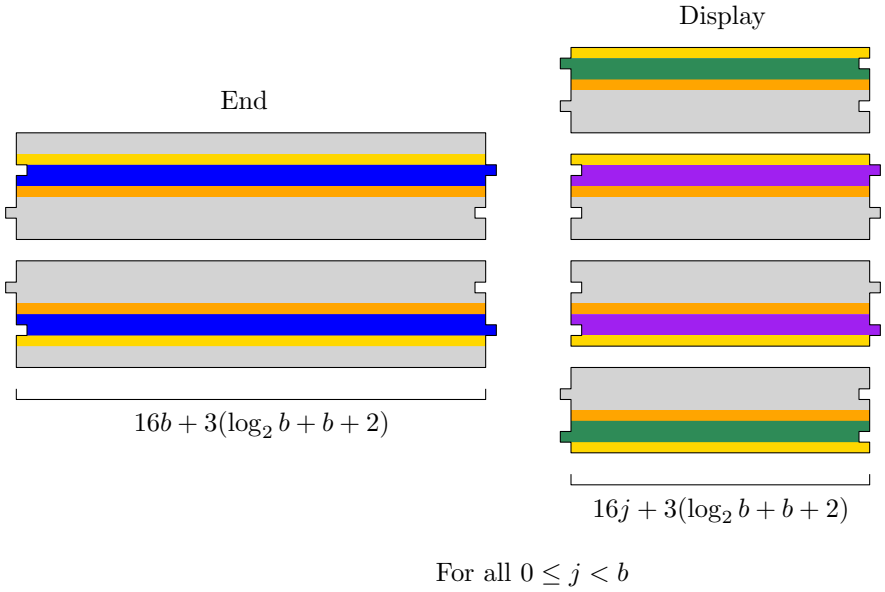


Figure 5.18: The decomposition of horizontal slabs of each ring the b-bit block counter.

to-end counter (see Figure 5.20) has subrow values from 0 to $b/2$, for the $b/2$ most significant bits of the row value of each block, and row values from 0 to $2^{b/2}$. Modified versions of reset bars with height (width in the horizontal

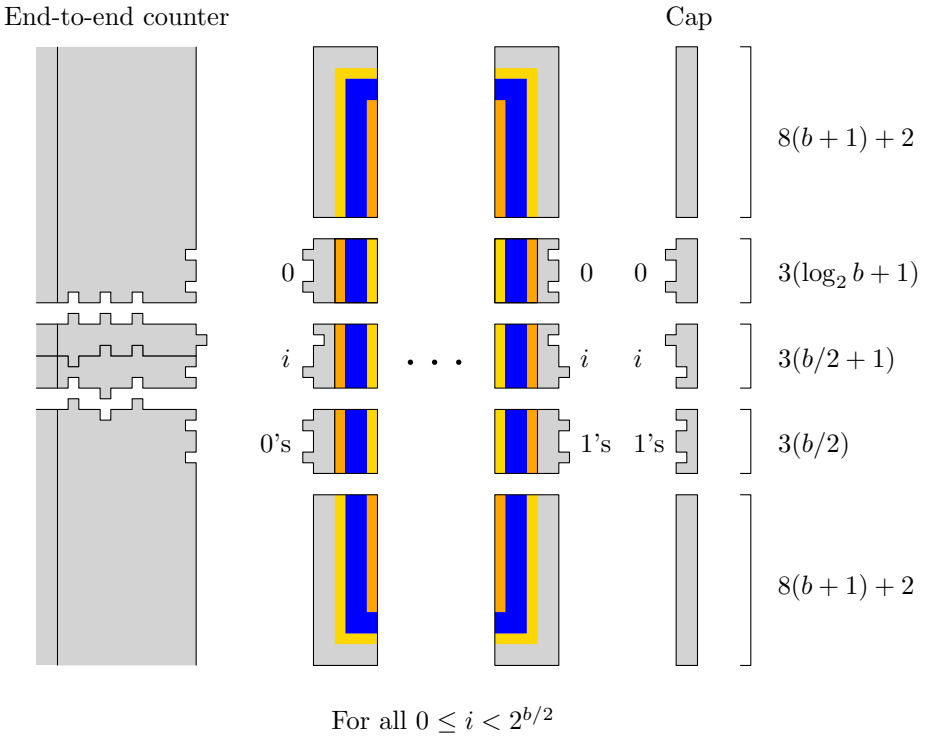


Figure 5.19: (Left) The interaction of a vertical end-to-end counter with the westernmost block in each row. (Right) The cap assemblies built to attach to the easternmost block in each row.

end-to-end counter) $\Theta(b^2)$ are used to bridge across the geometry-less portions of the west sides of the blocks, as well as the always-zero $b/2$ least significant bits of the block's row value and subrow $\log_2 b$ bits.

This modified end-to-end counter can be assembled using $O(b)$ work as done for the original end-to-end counter, since the longer reset bars only add $O(\log(b^2)) = O(\log b)$ work to the assembly process. After the vertical end-to-end counter has been combined with the blocks to form a complete block counter, a horizontal end-to-end counter is attached to the top of the assembly to produce a square assembly.

Lemma 5.6.8. *For even b, there exists a $\tau = 1$ SAS of size $O(b)$ that produces a b-bit block counter.*

Proof. The construction described builds families of vertical bars and horizon-

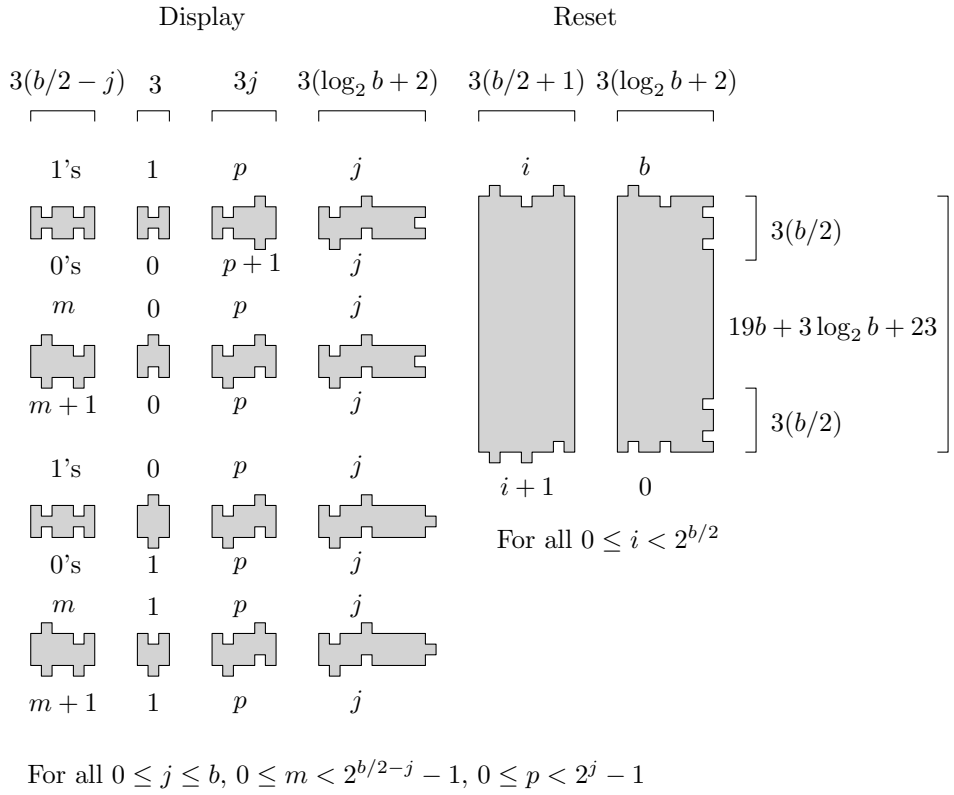


Figure 5.20: The decomposition of the bars of a vertically-oriented end-to-end counter used to combine rows of blocks in a block counter.

tal slabs that are used to assemble each the rings forming all blocks in the counter. There are a constant number of families, and each family can be assembled using $O(b)$ work. The vertical and horizontal end-to-end counters can also be assembled using $O(b)$ work each by Lemma 5.6.5. Then the b-bit block counter can be assembled by a SAS of size $O(b)$. □

We now consider a lower bound for any PCFG G deriving the counter, using a similar approach as Lemma 5.6.6.

Lemma 5.6.9. *For any PCFG G deriving a b-bit block counter, $|G| = \Omega(2^b)$.*

Proof. Define a *minimal block spanner* as to be a non-terminal symbol N in G with production rule $N \to (B, (x_1, y_1))(C, (x_2, y_2))$ such that the polyomino derived by N (denoted p_N) contains a path from a gray cell outside the color loop of the end ring of the counter to a gray cell inside the start color loop of

the counter, and the polyominoes derived by B and C (denoted p_B and p_C) do not.

First we show that any minimal block spanner is a spanner for at most one block. Assume by contradiction and that N is a minimal block spanner for two blocks $\mathcal{B}_i$ and $\mathcal{B}_j$ and that p_B contains a gray cell inside the start color loop of $\mathcal{B}_i$. Then B must be entirely contained in the color loop of the end ring of $\mathcal{B}_i$, as otherwise N is not a minimal block spanner for $\mathcal{B}_i$. Similarly, C must then be entirely contained in the color loop of the end ring of $\mathcal{B}_j$. Since no pair of color loops from distinct blocks have adjacent cells, p_N is not a connected polyomino and so G is not a valid PCFG.

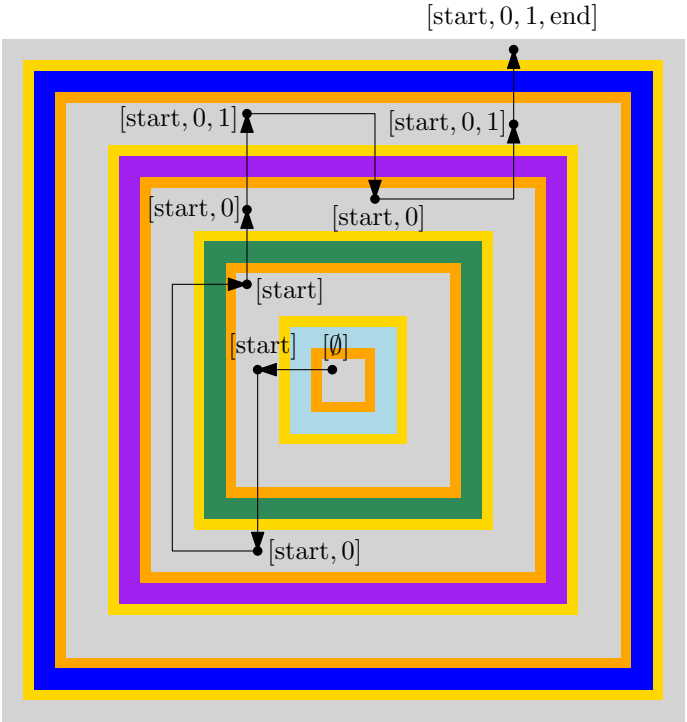


Figure 5.21: A schematic of the proof that the block spanned by a minimal row spanner is unique. Maintaining a stack while traversing a path from the interior of the start ring to the exterior of the end ring uniquely determines the block spanned by any minimal block spanner containing the path.

Next we show that the block spanned by N is unique, i.e. N cannot be reused as a minimal spanner for multiple blocks. See Figure 5.21. Let N be

a minimal spanner for a block $\mathcal{B}_i$ and p be a path of cells in p_N starting at a gray cell contained in the start ring of $\mathcal{B}_i$ and ending at a gray cell outside the end ring of $\mathcal{B}_i$. Consider a traversal of p, maintaining a stack containing the color loops crossed during the traversal. Crossing a color loop from interior to exterior (a sequence of dark orange, then green, purple, or blue, then light orange cells) adds the center subloop's color to the stack, and traversing from exterior to interior removes the topmost element of the stack.

We claim that the sequence of subloop colors found in the stack after traversing an end ring from interior to exterior encodes a unique sequence of display rings and thus a unique block. To see why, first consider that the color loop of every ring forms a simple closed curve. Then the Jordan curve theorem implies that entering or leaving each region of gray cells between adjacent color loops requires traversing the color loop. Then by induction on the steps of p, the stack contains the set of rings *not containing* the current location on p in innermost to outermost order. So the stack state after exiting the exterior of the end ring uniquely identifies the block containing p and N is a minimal spanner for this unique block.

Since there are 2^b distinct blocks in a b-bit block counter, any PCFG that derives a counter has at least 2^b non-terminal symbols and size $\Omega(2^b)$. □

Theorem 5.6.10. *The separation of PCFGs over $\tau = 1$ SASs for constant-label squares is $\Omega(n/\log^3 n)$.*

Proof. By construction, a b-bit block counter has size $\Theta(2^b b^2) = n$ and so $b = \Theta(\log n)$. By the previous two lemmas, the separation is $\Omega((n/b^2)/b) = \Omega(n/\log^3 n)$. □

Unlike the previous rectangle construction, it does not immediately follow that a similar separation holds for 2-label squares. Finding a construction

that achieves nearly-linear separation but only uses two labels remains an open problem.

5.6.4 Constant-glue constructions

Lemma 5.5.4 proved that any system $\mathcal{S}$ can be converted to a slightly larger system (both in system size and scale) that simulates $\mathcal{S}$. Applying this lemma to the constructions of Section 5.6 yields identical results for constant-glue systems:

Theorem 5.6.11. *All results in Section 5.6 hold for systems with $O(1)$ glues.*

Proof. Lemma 5.5.4 describes how to convert any SAS or SSAS $\mathcal{S} = (T, G, \tau, M)$ into a macrotile version of the system $\mathcal{S}'$ that uses a constant number of glues, has system size $O(\Sigma(T)|T| + |\mathcal{S}|)$, and scale factor $O(\log |G|)$. Additionally, the construction achieves matching labels on *all* tiles of each macrotile, including the glue assemblies. Because the labels are preserved, the polyominoes produced by each macrotile system $\mathcal{S}'$ simulating an assembly system $\mathcal{S}$ in Section 5.6 preserves the lower bounds for PCFGs (Lemmas 5.6.2, 5.6.6, and 5.6.9) of each construction. Moreover, the number of labels in the polyomino is constant and so $|\mathcal{S}'| = O(|T| + |\mathcal{S}|) = O(|\mathcal{S}|)$ and the system size of each construction remains the same. Finally, the scale of the macrotiles is $O(\log |G|) = O(\log |\mathcal{S}| = O(\log b)$, so n is increased by a $O(\log^2 b)$-factor, but since n was already exponential in b, it is still the case that $b = \Theta(\log n)$ and so the separation factors remain unchanged. □

6

Conclusion

As this work has shown, there can be significant value in connecting self-assembly models to more traditional computational models, such as context-free grammars. Of course, this has been known since the Ph.D. thesis of Winfree [Win98], in which a certain one-dimensional cellular automaton known to be Turing-universal was shown to be simulated by a simple aTAM system, demonstrating that the aTAM is computationally universal. Another example is the Kolmogorov-complexity argument used by Rothemund and Winfree [RW00], now a standard approach for tile assembly lower bounds. For the most part, the connections between tile assembly and traditional models have been Turing-universality proofs of various tile assembly models, and these proofs are now akin to NP-hardness proofs in the algorithms community. Given the existence of so many universality proofs, one might expect that the complexity of corresponding optimization and prediction problems for these models are also solved as a byproduct of characterizing tile assembly models as equivalent to Turing machines. This has turned out not to be the case.

A primary obstacle to converting universality results into complexity results on optimization problems is the lack of close correspondence between

Turing machines and tile systems. Ideally, a mapping between Turing machines and tile systems that preserves the relative sizes of corresponding machines and systems exists. Such a mapping seems unlikely, as the string world of Turing machines appears to be a long way from the shape world of tile assembly, and Turing machines lack the geometric aspect of so significant in tile assembly (e.g. the constructions of Section 5.6).

Possibly as a result of the lack of closer correspondences between Turing machine and tile assembly models, the complexity of optimization problems in tile assembly (reviewed in Section 1.4) remain largely unresolved. For instance, the problem of finding the smallest tile set uniquely assembling a patterned square remains entirely open. The *patterned self-assembly tile set synthesis (PATS)* problem is a highly constrained variant of this problem in which an initial L-shaped assembly forming two edges of the square is given for free, and all tiles must attach to the existing assembly on their south and west sides. Even with these significant restrictions, the PATS problem was not known to be NP-hard until 2012 [CP12]. One of the first optimization problems in tile assembly studied was studied by Adleman et al. [ACG$^+$02] in 2002: find the smallest $\tau = 2$ aTAM system uniquely assembling an assembly with a given shape. They were able to show that this problem is NP-hard, but left the problem of approximating the smallest tile set as an open problem and this problem remains open still.

Even considering our work in isolation, a large number of open problems remains, and these problems suggest two major lines of potential work. The first line is to improve the approximability bounds of the smallest PCFG problem. This problem may be difficult enough to require interesting intermediate results to make progress, and solutions may involve polyomino problems (such as packing), pattern recognition problems (such as identifying repeated

subpolyominoes), hierarchical compression problems (as found in the smallest addition chain problem), or planar graph problems (such as partitioning, hitting, or independence). The second line is to develop more positive results and techniques for using bin parallelism in staged assembly. Achieving tighter bounds on the complexity of the smallest SAS problem for unbounded glues, or approximating the problem with a fixed glue count are both likely to be difficult and require more understanding of how bin parallelism can be used in the average case. More generally, it would be interesting to see how staging could be applied to other models of molecular computing, such as chemical reaction networks or membrane computing, and how other variations on the concept of staging compare to staged tile assembly.

Bibliography

[ABD+10] Z. Abel, N. Benbernou, M. Damian, E. D. Demaine, M. L. Demaine, R. Flatland, S. D. Kominers, and R. Schweller. Shape replication through self-assembly and RNase enzymes. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2010.

[ACG+01] L. M. Adleman, Q. Cheng, A. Goel, M.-D. Huang, and H. Wasserman. Linear self-assemblies: Equilibria, entropy and convergence rates. In B. Aulbach, S. N. Elaydi, and G. Ladas, editors, *Proceedings of Sixth International Conference on Difference Equations and Applications*, 2001.

[ACG+02] L. Adleman, Q. Cheng, A. Goel, M.-D. Huang, D. Kempe, P. M. de Espanés, and P. W. K. Rothemund. Combinatorial optimization problems in self-assembly. In *Proceedings of Symposium on Theory of Computing (STOC)*, 2002.

[ACG+05] G. Aggarwal, Q. Cheng, M. Goldwasser, M. Kao, P. de Espanes, and R. Schweller. Complexities for generalized models of self-assembly. *SIAM Journal on Computing*, 34(6):1493–1515, 2005.

[ACGH01] L. Adleman, Q. Cheng, A. Goel, and M.-D. Huang. Running time and program size for self-assembled squares. In *Proceedings of Symposium on Theory of Computing (STOC)*, 2001.

[Adl00] L. Adleman. Toward a mathematical theory of self-assembly (extended abstract). Technical Report 00-722, University of Southern California, 2000.

[ALM+98] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45(3):501–555, 1998.

[BCD+11] N. Bryans, E. Chiniforooshan, D. Doty, L. Kari, and S. Seki. The power of nondeterminism in self-assembly. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 590–602, 2011.

[BJL+94] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis. Linear approximation of shortest superstrings. *Journal of the ACM*, 41(4):630–647, 1994.

[BK98] P. Berman and M. Karpinski. On some tighter approximability results, further improvements. Technical Report TR-98-065, ECCC, 1998.

[BLMZ10] N. Bansal, M. Lewenstein, B. Ma, and K. Zhang. On the longest common rigid subsequence problem. *Algorithmica*, 56:270–280, 2010.

[BMRR06] G. Barequet, M. Moffie, A. Ribó, and G. Rote. Counting polyominoes on twisted cylinders. *INTEGERS: The Electronic Journal of Combinatorial Number Theory*, 6(A22), 2006.

[Bra39] A. Brauer. On addition chains. *Bulletin of the American mathematical Society*, 45(10):736–739, 1939.

[BSRW09] R. D. Barish, R. Schulman, P. W. K. Rothemund, and E. Winfree. An information-bearing seed for nucleating algorithmic self-assembly. *Proceedings of the National Academic of Sciences (PNAS)*, 106(15):6054–6059, 2009.

[CD12] H.L. Chen and D. Doty. Parallelism and time in hierarchical self-assembly. In *ACM-SIAM Symposium on Discrete Algorithms*, 2012.

[CDD+13] S. Cannon, E. D. Demaine, M. L. Demaine, S. Eisenstat, M. J. Patitz, R. T. Schweller, S. M. Summers, and A. Winslow. Two hands are better than one (up to constant factors): Self-assembly in the 2HAM vs. aTAM. In *Proceedings of International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 20 of *LIPIcs*, pages 172–184. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013.

[CDS11] H.-L. Chen, D. Doty, and S. Seki. Program size and temperature in self-assembly. In T. Asano, S. Nakano, Y. Okamato, and O. Watanbe, editors, *ISAAC 2011*, volume 7074 of *LNCS*, pages 445–453. Springer Berlin Heidelberg, 2011.

[CFS11] M. Cook, Y. Fu, and R. Schweller. Temperature 1 self-assembly: determinstic assembly in 3d and probabilistic assembly in 2d. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2011.

[CGR12] H. Chandran, N. Gopalkrishnan, and J. Reif. Tile complexity of linear assemblies. *SIAM Journal on Computation*, 41(4):1051–1073, 2012.

[Cho56] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[Cho59] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 9:137–167, 1959.

[CLL+02] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Rasala, A. Sahai, and a. shelat. Approximating the smallest grammar: Kolmogorov complexity in natural models. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 792–801, New York, NY, USA, 2002. ACM.

[CLL+05] M. Charikar, E. Lehman, A. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and a. shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[CP09] A. Cherubini and M. Pradella. Picture languages: from wang tiles to 2d grammars. In S. Bozapalidis and G. Rahonis, editors, *CAI 2009*, volume 5725 of *LNCS*, pages 13–49. Springer Berlin Heidelberg, 2009.

[CP12] E. Czeizler and A. Popa. Synthesizing minimal tile sets for complex patterns in the framework of patterned DNA self-assembly. In D. Stefanovic and A. Turberfield, editors, *DNA 18*, volume 7433 of *LNCS*, pages 58–72. Springer Berlin Heidelberg, 2012.

[CRP08] A. Cherubini, S. C. Reghizzi, and M. Pradella. Regional languages and tiling: a unifying approach to picture grammars. In E. Ochmański and J. Tyszkiewicz, editors, *MFCS 2008*, volume 5162 of *LNCS*, pages 253–264. Springer Berlin Heidelberg, 2008.

[CS70] J. Cocke and J. T. Schwartz. Programming languages and their compilers. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[DDF+08a] E. D. Demaine, M. L. Demaine, S. P. Fekete, M. Ishaque, E. Rafalin, R. T. Schweller, and D. L. Souvaine. Staged self-assembly: nanomanufacture of arbitrary shapes with $O(1)$ glues. *Natural Computing*, 7(3):347–370, 2008.

[DDF+08b] E. D. Demaine, M. L. Demaine, S. P. Fekete, M. Ishaque, E. Rafalin, R. T. Schweller, and D. L. Souvaine. Staged self-assembly: nanomanufacture of arbitrary shapes with $O(1)$ glues. In M. H. Garzon and H. Yan, editors, *DNA 13*, volume 4848 of *LNCS*, pages 1–14. Springer Berlin Heidelberg, 2008.

[DDF+12] E. D. Demaine, M. L. Demaine, S. P. Fekete, M. J. Patitz, R. T. Schweller, A. Winslow, and D. Woods. One tile to rule them all: Simulating any turing machine, tile assembly system, or tiling system with a single puzzle piece. Technical report, arXiv, 2012.

[DEIW11] E. D. Demaine, S. Eisenstat, M. Ishaque, and A. Winslow. One-dimensional staged self-assembly. In L. Cardelli and W. Shih, editors, *DNA 17*, volume 6937 of *LNCS*, pages 100–114. Springer Berlin Heidelberg, 2011.

[DEIW12] E. D. Demaine, S. Eisenstat, M. Ishaque, and A. Winslow. One-dimensional staged self-assembly. *Natural Computing*, 2012.

[DF92] M. P. Delest and J. M. Fedou. Attribute grammars and useful for combinatorics. *Theoretical Computer Science*, 98:65–76, 1992.

[DLP+10] D. Doty, J. H. Lutz, M. J. Patitz, S. M. Summers, and D. Woods. Intrinsic universality in self-assembly. In *Proceedings of International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 5 of *LIPIcs*, pages 275–286. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

[DLP+12] D. Doty, J. H. Lutz, M. J. Patitz, R. T. Schweller, S. M. Summers, and D. Woods. The tile assembly model is intrinsically universal. In *Proceedings of the 53rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 302–310, 2012.

[DLS81] P. Downey, B. Leong, and R. Sethi. Computing sequences with addition chains. *SIAM Journal on Computing*, 10(3):638–646, 1981.

[Dot10] D. Doty. Randomized self-assembly for exact shapes. *SIAM Journal on Computing*, 39(8):3521–3552, 2010.

[Dot12] D. Doty. Theory of algorithmic self-assembly. *Communications of the ACM*, 55(12):78–88, 2012.

[DPR+10] D. Doty, M. J. Patitz, D. Reishus, R. T. Schweller, and S. M. Summers. Strong fault-tolerance for self-assembly with fuzzy temperature. In *Foundations of Computer Science (FOCS)*, pages 417–426, 2010.

[DPR+13] E. D. Demaine, M. J. Patitz, T. A. Rogers, R. T. Schweller, and D. Woods. The two-handed tile assembly model is not intrinsically universal. In F. V. Fomin, R. Freivalds, M. Z. Kwiatkowska, and D. Peleg, editors, *Automata, Languages and Programming (ICALP)*, volume 7965 of *LNCS*, pages 400–412. Springer Berlin Heidelberg, 2013.

[DR04] E. Duchi and S. Rinaldi. An object grammar for column-convex polyominoes. *Annals of Combinatorics*, 8(1):27–36, 2004.

[Erd32] P. Erdös. Beweis eines satzes von tschebyschef. *Acta Litterarum ac Scientiarum Szeged*, 5:194–198, 1932.

[ET41] P. Erdös and P. Turán. On a problem of Sidon in additive number theory, and on some related problems. *Journal of the London Mathematics Society*, 16:212–216, 1941.

[Fed68] J. Feder. Languages of encoded line patterns. *Information and Control*, 13(3):230–244, 1968.

[FPSS12] B. Fu, M. J. Patitz, R. T. Schweller, and B. Sheline. Self-assembly with geometric tiles. In A. Czumaj, K. Mehlhorn, A. Pitts, and R. Wattenhofer, editors, *Automata, Languages and Programming (ICALP)*, volume 7391 of *LNCS*, pages 714–725. Springer Berlin Heidelberg, 2012.

[Fre61] H. Freeman. On the encoding of arbitrary geometric configurations. *IRE Transactions on Electronic Computers*, EC-10(2):260–268, 1961.

[FT98] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998.

[GHR95] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to parallel computation: P-completeness theory*. Oxford University Press, 1995.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., 1979.

[GK97] R. Giegerich and S. Kurtz. From Ukkonen to McCreight to Weiner: a unifying view of linear-time suffix tree construction. *Algorithmica*, 19:331–353, 1997.

[GMA80] J. Gallant, D. Maier, and J. Astorer. On finding minimal length superstrings. *Journal of Computer and System Sciences*, 20(1):50–58, 1980.

[GR97] D. Giammarresi and A. Restivo. Two-dimensional languages. In *Handbook of Formal Languages*, pages 215–267. Springer, New York, 3 edition, 1997.

[HRA10] M. Hayashida, P. Ruan, and T. Akutsu. A quadrisection algorithm for grammar-based image compression. In T. h. Kim, Y. h. Lee, B.-H. Kang, and D. Ślęzak, editors, *FGIT 2010*, volume 6485 of *LNCS*, pages 234–248. Springer Berlin Heidelberg, 2010.

[Hui92] L. C. K. Hui. Color set size problem with applications to string matching. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *CPM 1992*, volume 664 of *LNCS*, pages 230–243. Springer Berlin Heidelberg, 1992.

[Jeż13] A. Jeż. Approximation of grammar-based compression via recompression. Technical report, arXiv, 2013.

[JL95] T. Jiang and M. Li. On the approximation of shortest common supersequences and longest common subsequences. *SIAM Journal on Computing*, 24(5):1122–1139, 1995.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–104. Plenum, 1972.

[KKS13] L. Kari, S. Kopecki, and S. Seki. 3-color bounded patterned self-assembly. Technical report, arXiv, 2013.

[KR73] D. A. Klarner and R. L. Rivest. A procedure for improving the upper bound for the number of n-ominoes. *Canadian Journal of Mathematics*, 25:585–602, 1973.

[KS06] M. Y. Kao and R. Schweller. Reducing tile complexity for self-assembly through temperature programming. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 571–580, 2006.

[KS08] M. Y. Kao and R. Schweller. Randomized self-assembly for approximate shapes. In L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *Automata, Languages and Programming (ICALP)*, volume 5125 of *LNCS*, pages 370–384. Springer Berlin Heidelberg, 2008.

[KS12] M. Karpinski and R. Schmied. Improved inapproximability results for the shortest superstring and related problems. Technical Report 85331-CS, University of Bonn, 2012.

[KSX12] L. Kari, S. Seki, and Z. Xu. Triangular and hexagonal tile self-assembly systems. In M. J. Dinneen, B. Khoussainov, and A. Nies, editors, *WTCS 2012*, volume 7160 of *LNCS*, pages 357–375. Springer Berlin Heidelberg, 2012.

[Las02] E. Lehman and a. shelat. Approximation algorithms for grammar-based compression. In *Proceedings of the 20th Annual Symposium on Discrete Algorithms (SODA)*, pages 205–212, 2002.

[Leh02] E. Lehman. *Approximation Algorithms for Grammar-Based Data Compression*. PhD thesis, MIT, 2002.

[LL09] M. Lange and H. Leiß. To cnf or not to cnf? an efficient yet presentable version of the cyk algorithm. *Informatica Didactica*, 8, 2009.

[LNS92] P. Laroche, M. Nivat, and A. Saoudi. Context-sensitivity of puzzle grammars. In A. Nakamura, M. Nivat, A. Saoudi, P. Wang, and I. Katsushi, editors, *Parallel Image Analysis*, volume 654 of *LNCS*, pages 195–212. Springer Berlin Heidelberg, 1992.

[Luh09] C. Luhrs. Polyomino-safe DNA self-assembly via block replacement. In A. Goel, F. C. Simmel, and P. Sosik, editors, *DNA 14*, volume 5347 of *LNCS*, pages 112–126. Springer Berlin Heidelberg, 2009.

[Luh10] C. Luhrs. Polyomino-safe DNA self-assembly via block replacement. *Natural Computing*, 9(1):97–109, 2010.

[ML08] X. Ma and F. Lombardi. Synthesis of tile sets for dna self-assembly. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(5):963–967, 2008.

[MMS06] S. Maruyama, H. Miyagawa, and H. Sakamoto. Improving time and space complexity for compressed pattern matching. In T. Asano, editor, *ISAAC 2006*, volume 4288 of *LNCS*, pages 484–493. Springer Berlin Heidelberg, 2006.

[MR71] D. L. Milgram and A. Rosenfeld. Array automata and array grammars. *IFIP Congress*, booklet TA-2:166–173, 1971.

[MRW82] H. A. Mauerer, G. Rozenberg, and E. Welzl. Using string languages to describe picture languages. *Information and Control*, 54:155–182, 1982.

[MYS83] K. Morita, Y. Yamamoto, and K. Sugata. The complexity of some decision problems about two-dimensional array grammars. *Information Sciences*, 30:241–292, 1983.

[MZ05] B. Ma and K. Zhang. On the longest common rigid subsequence problem. In A. Apostolico, M. Crochemore, and K. Park, editors, *CPM 2005*, volume 3537 of *LNCS*, pages 11–20. Springer Berlin Heidelberg, 2005.

[Oll08] N. Ollinger. Universalities in cellular automata: a (short) survey. In B. Durand, editor, *Symposium on Cellular Automata Journées Automates Cellular (JAC 2008)*, pages 102–118. MCCME Publishing House, 2008.

[Pat12] M. J. Patitz. An introduction to tile-based self-assembly. In J. Durand-Lose and N. Jonoska, editors, *UCNC 2012*, volume 7445 of *LNCS*, pages 34–62. Springer Berlin Heidelberg, 2012.

[PLS12] J. Padilla, W. Liu, and N. C. Seeman. Hierarchical self assembly of patterns from the Robinson tilings: DNA tile design in an enhanced tile assembly model. *Natural Computing*, 11(2):323–328, 2012.

[Ram19] S. Ramanujan. A proof of Bertrand's postulate. *Journal of the Indian Mathematical Society*, 11:181–182, 1919.

[Rot01] P. W. K. Rothemund. Using lateral capillary forces to compute by self-assembly. *PNAS*, 97(3):984–989, 2001.

[RP05] S. C. Reghizzi and M. Pradella. Tile rewriting grammars and picture languages. *Theoretical Computer Science*, 340(2):257–272, 2005.

[RW00] P. W. K. Rothemund and E. Winfree. The program-size complexity of self-assembled squares. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 459–468, 2000.

[Ryt02] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. In A. Apostolico and M. Takeda, editors, *Combinatorial Pattern Matching*, volume 2373 of *LNCS*, pages 20–31. Springer Berlin Heidelberg, 2002.

[Sak05] H. Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *Journal of Discrete Algorithms*, 3(2–4):416–430, 2005.

[Sch37] A. Scholz. Aufgabe 253. *Jahresbericht der deutchen Mathematiker-Vereinigung*, 47:41–42, 1937.

[Sch98] R. Schulman. *The self-replication and evolution of DNA crystals*. PhD thesis, Caltech, 1998.

[Sch13] R. Schweller. personal communication, 2013.

[Sek13] S. Seki. Combinatorial optimization in pattern assembly. In G. Mauri, A. Dennunzio, L. Manzoni, and A. E. Porreca, editors, *UCNC 2013*, volume 7956 of *LNCS*, pages 220–231. Springer Berlin Heidelberg, 2013.

[Sid32] S. Sidon. Ein satz äuber trigonomietrische polynome und siene anwendungen in der theorie der Fourier-Reihen. *Mathematische Annalen*, 106:536–539, 1932.

[SKS04] H. Sakamoto, T. Kida, and S. Shimozono. A space-saving linear-time algorithm for grammar-based compression. In A. Apostolico and M. Melucci, editors, *SPIRE 2004*, volume 3246 of *LNCS*, pages 218–229. Springer Berlin Heidelberg, 2004.

[SSK72] G. Siromoney, R. Siromoney, and K. Krithivasan. Abstract families of matrices and picture languages. *Computer Graphics and Image Processing*, 1(3):284–309, 1972.

[Sum12] S. M. Summers. Reducing tile complexity for the self-assembly of scaled shapes through temperature programming. *Algorithmica*, 63(1–2):117–136, 2012.

[SW85] I. Sudborough and E. Welzl. Complexity and decidability for chain code picture languages. *Theoretical Computer Science*, 36:173–202, 1985.

[SW05] R. Schulman and E. Winfree. Programmable control of nucleation for algorithmic self-assembly (extended abstract). In C. Ferretti, G. Mauri, and C. Zandron, editors, *DNA 10*, volume 3384 of *LNCS*, pages 319–328. Springer Berlin Heidelberg, 2005.

[Swe94] Z. Sweedyk. A $\frac{1}{2}$-approximation algorithm for shortest superstring. *SIAM Journal on Computing*, 29(3):954–986, 1994.

[Vas05] V. Vassilevska. Explicit inapproximability bounds for the shortest superstring problem. In J. Jędrzejowicz and A. Szepietowski, editors, *MFCS 2005*, volume 3618 of *LNCS*, pages 793–800. Springer Berlin Heidelberg, 2005.

[Wei73] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the IEEE 14th Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[Win98] E. Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, Caltech, 1998.

[Win12] A. Winslow. Inapproximability of the smallest superpolyomino problem. In *Proceedings of the 22nd Annual Fall Workshop on Computational Geometry (FWCG)*, 2012.

[Win13] A. Winslow. Staged self-assembly and polyomino context-free grammars. In D. Soloveichik and B. Yurke, editors, *DNA 19*, 2013.

[Woo13] D. Woods. Intrinsic universality and the computational power of self-assembly. In T. Neary and M. Cook, editors, *MCU 2013*, volume 128 of *EPTCS*, pages 16–22. Open Publishing Association, 2013.

[Yao76] A. C.-C. Yao. On the evaluation of powers. *SIAM Journal on Computing*, 5(1):100–103, 1976.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[Zuc07] D. Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. *Theory of Computing*, 3:103–128, 2007.