Map Folding

by

Tom Morgan

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

Author .....
Department of Electrical Engineering and Computer Science
May 21, 2012

Certified by .....
Erik D. Demaine
Professor
Thesis Supervisor

Accepted by .....
Dennis M. Freeman
Chairman, Department Committee on Graduate Theses

Acknowledgments

(More TBD)

The idea of the ray diagram was introduced by David Charlton, and further elaborated by Yoyo Zhou, in the context of an open problem session held as part of an MIT class 6.885 on Geometric Folding Algorithms in Fall 2007. I thank the participants of that open problem session for helpful early discussions about $2 \times n$ map folding.

I would also like to thank Eric Liu for his close help in the early stages of this project and Professor Erik Demaine for his valuable guidance throughout.

Contents

Chapter 1

Introduction

Determining whether a crease pattern is flat-foldable is one of the most fundamental problems in the field of geometric folding algorithms. A *crease pattern* is a set of line segment creases on a piece of paper. For our purposes, every crease pattern will have a mountain-valley assignment, which specifies which direction each crease should be folded in. A flat folding must fold 180° along each crease in the direction specified by the mountain-valley assignment, and must not have folds anywhere else. It is known that in general, deciding whether a crease pattern has a flat folding is NP-hard [2].

In this thesis, we will examine a natural special class of crease patterns, those which form a $m \times n$ rectangular lattice. We refer to this problem as the *map* folding problem because informally, it describes how to refold a road map while respecting the intended crease directions. Arkin et al. [1] considered the related problem of folding an $m \times n$ map via a sequence of *simple folds* (each folding one line in its entirely), which turns out to reduce to a series of 1D folding problems, solvable in polynomial time. However, when the restriction to simple folds is lifted, the problem becomes much more complex.

Jack Edmonds[1] posed the problem of deciding whether a given map has a folded state as interesting from a computational complexity perspective, conjecturing that it is NP-complete in general. This problem has remained open, even for $2 \times n$ maps, for the past 15 years.

[1]Personal communication, August 1997.

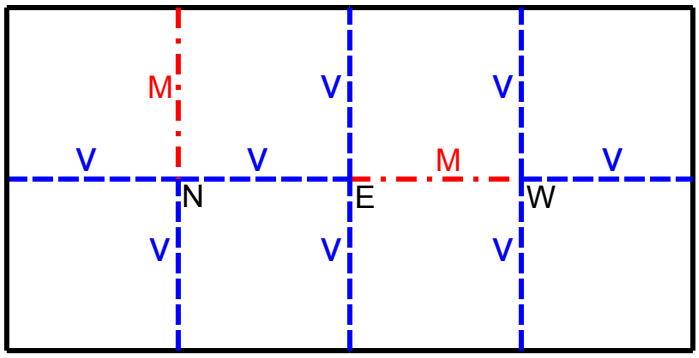

Figure 1-1: NEW map.

1.1 Map Folding Basics

1.1.1 $m \times n$ Map Folding Basics

An unfolded $m \times n$ map has two rows and n columns of cells. The creases either lie along one of the $m-1$ evenly spaced horizontal lines or along one of $n-1$ evenly spaced vertical lines. There are $n(m-1)$ horizontal creases, and $(n-1)m$ vertical creases. By possible inversion of all crease directions (which preserves foldability), assume that the upper leftmost horizontal crease is a valley.

NSEW vertex labels. By Maekawa's Theorem [5], every vertex must have one mountain and three valleys or vice versa. We can therefore label each vertex by the unique crease direction that is different from the other three, one of *north*, *south*, *east*, or *west*. This labeling was introduced by Justin [6]. Fig. 1-1 shows the 2×4 map corresponding to the sequence NEW. There are $2^{2(n+m-3)+(n-2)(m-2)}$ possible vertex assignments for an $m \times n$ map. This follows from assigning the vertices sequentially row by row. After the first row and column have been assigned, each subsequent vertex has only two possibilities.

Folded state. A *flat folded state* of a map is a total order of the cells that induces no crossings and respects the mountain/valley labels.

For consistency, view the top-left cell as having a fixed position; this also fixes the orientation of all cells (imagine attaching coordinate axes to each cell). The cells can

be indexed by a a pair (i, j) with $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, n\}$. Then the cells can take one of four possible parity values given by $(i \bmod 2, j \bmod 2)$. In a valid flat-folded state, all cells of the same parity will have the same orientation in the folded state. Using the fixed orientation of the top-left cell and the parity relations, mountains and valleys also translate into above and below relations for cells of paper in the folded state.

1.1.2 $2 \times n$ Map Folding Basics

We introduce additional definitions, specific to the case when $m = 2$.

Centerline. The *centerline* is the single horizontal line running through the paper, and its n corresponding horizontal creases.

Segments. A *segment* refers to a pair of cells (in the same column of cells) touching a common centerline crease. A segment may also refer directly to that common centerline edge.

Tunnels. A *tunnel* is a (contiguous) sequence of all-mountain or all-valley horizontal creases along the centerline. For example, the NEW map (Fig. 1-1) has three distinct tunnels. For a given horizontal centerline crease (tunnel), we will refer to the two cells bordering that crease as the "walls" of that tunnel.

Intuitively, a tunnel can turn or be manipulated in all the ways that a $1 \times m$ strip could, by suitable settings of the vertical creases. Adjacent tunnels either nest inside one another or they have disjoint interiors. In particular, they cannot intersect. Note that a tunnel on the interior of another tunnel must at least follow the containing tunnel's folds, but the interior tunnel may have extra folds.

East and west vertices create new tunnels by changing the centerline fold from mountain to valley or vice versa. North and south vertices change the direction (or orientation) of the current tunnel. Note that, in different tunnels, only north vertices can fold inside of other north vertices, and similarly for south vertices.

1.1.3 Valid $m \times n$ Folded States

The $m \times n$ map problem, is to determine, given an $m \times n$ map, whether it has a valid folded state which, as described earlier in the section, we represent by a total order of the cells. In order for a folded state to be valid, it must fulfill two conditions. First, the map's attachments between cells must not cross each other. Second, the crease directions (mountain or valley) must be obeyed. The first condition can be checked by looping over every pair of creases on each of the four sides, and checking if they intersect. To check the second condition, we first fix the position of the upper leftmost cell. This cell and all cells with the same parity must lie above any cell with which they share a valley crease, and below any cell with which they share a mountain crease. For cells with the opposite parity, these conditions are reversed. This implicitly gives us an $O\left((mn)^2\right)$ algorithm for checking the validity of a folded state, thus showing the problem to be in NP.

We do not concern ourselves with what sequence of folds is necessary to reach a valid folded state. This is because these folds may necessarily involve complex origami folds. Furthermore, it is known that folded state can be reached by a continuous folding motion (using additional intermediate creases) [4]. Were we restricted to simple folds, in which at each step we fold completely along a single line through the paper, many of valid folded states would be unreachable.

1.2 Contributions

This thesis provides a number of theoretical contributions to the $m \times n$ map folding problem. The most significant of which is the polynomial-time algorithm for the $2 \times n$ special case of the problem, which runs in $O(n^9)$ time. This algorithm is achieved through a series of reductions. First, in Section 2.1, we show equivalence between $2 \times n$ and finding a two-dimensional diagram called the "top-edge view". Next, in Section 2.2, we prove that this top-edge view is equivalent to the "ray diagram", which is another two-dimesnional structure which further exploits the fact that there is only a single centerline in the diagram to expose useful local properties. The last

reduction, in Section 4.2, reduces the problem of finding a ray diagram via a kind of dual transform to the problem of finding a valid "tree structure". In Section 4.1, we show how to solve the resulting "hidden tree problem" in polynomial time using dynamic programming techniques. This results results in an overall running time of $O(n^9)$ for map folding.

By further making use of the ray diagram, we present several faster than $O(n^9)$ time algorithms for special cases of $2 \times n$ map folding in Chapter 3. In particular, we present an $O(n^5)$ time algorithm for the case when the ray diagram forms "interlocking teeth", and an $O(n^4)$ algorithm for the case when the ray diagram forms a "spiral".

In Chapter 5 we present algorithms for solving certain optimization variants of the $2 \times n$ map folding. In particular, rather than simply finding if any folded state exists for the given $2 \times n$ map, they find the folded state that is, by some metric, simplest.

Finally, in Chapter 6 we present results for $m \times n$ map folding for any m. First, we develop a set of necessary conditions for an $m \times n$ map to be flat foldable that are checkable in polynomial time. Then, we develop a fixed parameter tractable algorithm for deciding the flat foldability of an $m \times n$ map, which runs in $O((mn)^{2.5}\ 2^{\log e(P)})$ time, where $\log e(P)$ is the entropy in the partial order induced by the mountain valley pattern of the map.

Chapter 2

Ray Diagram for $2 \times n$ Map Folding

In Section 2.1, we will prove that $2 \times n$ map folding is equivalent to finding a two-dimensional diagram called the "top-edge view", which represents each cell of the map by a horizontal segment and represents their stacking order by vertical position. Next, in Section 2.2, we prove that this top-edge view is equivalent to another two-dimensional diagram called the "ray diagram", which represents the "tunnel" between vertically adjacent cells by a single horizontal segment and uses vertical position to represent the "nesting" of these tunnels. The ray diagram structure will be the basis for the algorithms we develop for $2 \times n$ map folding in Chapter 3 and Chapter 4.

2.1 Top-Edge View

One way to represent the $2 \times n$ map folding problem is the *top-edge view*. A flat folded state of the map can be visualized as a stack of cells. If the paper lies in the xy plane with upper left corner at the origin, the top-edge view looks down on the xz plane (imagine that the paper has nonzero thickness).

The middle of Fig. 2-1 is an example of a valid top-edge view for the sequence NEW. The top-edge view initially consists of two parallel lines, representing the walls of the current tunnel. That is, the straight lines of the top-edge view directly correspond to an edge of one of the $2 \times n$ cells of the map. The semicircular curves represent creases; these are drawn with nonzero thickness for clarity. The centerline is

not drawn explicitly, but conceptually it lies between the two walls. The blue bars in figure Fig. 2-1 connect the two walls of each separate tunnel component[1]; their value in checking the non-intersection condition is explained in the next paragraph. Fig. 2-2 shows how the paths of these two lines are affected by the four atomic components of a top-edge view: W, E, S, and N folds. These patterns are easily verified with folds of a 2×2 piece of paper. In Fig. 2-2, the walls are drawn left-to-right; rotate the figures 180° if the path is right-to-left (e.g., after the N fold in the middle part of Fig. 2-1).

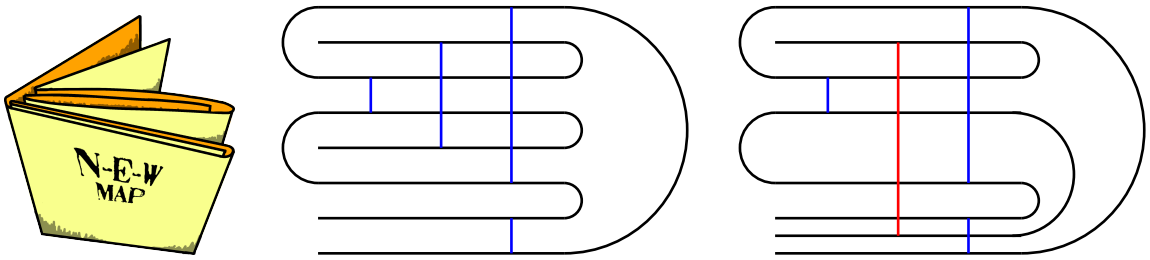

Figure 2-1: A valid folded state (left) and corresponding top-edge view (middle) for the sequence NEW, and an invalid top-edge view (right) for the same sequence.

The construction of a top-edge view for a given crease pattern is nontrivial. For example, the right side of Fig. 2-1 displays an invalid top-edge view for NEW; the red bar highlights the walls of the fold that induce self-intersection. When proceeding left-to-right through a given vertex sequence (e.g., NEW), there are multiple possible choices for each turn. Two possibilities for NEW are shown in Fig. 2-1. However, not all of the possible outcomes are valid. For example, the top-edge view in the right

[1]Hence there are four blue lines since there are four horizontal centerline creases in NEW.

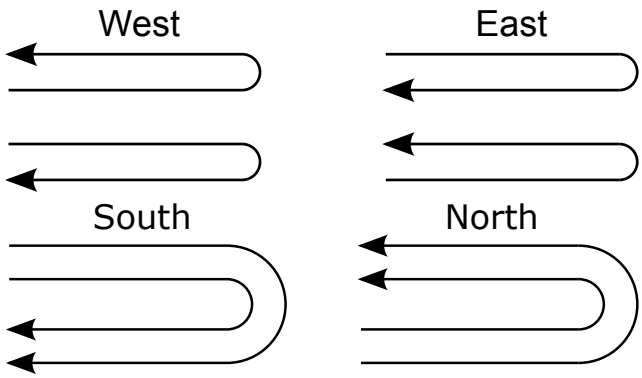

Figure 2-2: The top-edge view representation of west, east, south, and north vertices.

side of Fig. 2-1 uses only the four atomic components shown in Fig. 2-2, but it is invalid because one tunnel intersects another.

The nonintersection condition can be specified as follows. First, number the walls of the top-edge view from bottom to top, starting at 1. Then the crease (or tunnel component) connecting walls i and k crosses the crease connecting walls j and l if $i < j < k < l$. This condition cannot be applied to any wall-pair i, k; it only applies if i, k are walls of the same tunnel (as shown by the vertical bars in Fig. 2-1). In the middle and right parts of Fig. 2-1, the noncrossing condition is satisfied by all wall pairs connected by blue bars. However, in the right side of Fig. 2-1, the problematic walls are those connected by the red bar; they would have to intersect the centerline creases represented by the two right-most blue bars to "physically" produce this top-edge view.

A valid top-edge view and a flat folded map both provide an ordering of the map cells in the flat folded state. Because the two representations contain the same information, they are equivalent. Thus we can reduce the problem of deciding flat foldability of a $2 \times n$ map to the problem of finding a valid top-edge view.

The top-edge view does not easily generalize to $m \times n$ map folding for $m \neq 2$. In the $2 \times n$ case, the top-edge view visually represents all information about the folded state except for the centerline crease. But the walls of all tunnels remain visible, allowing the location of the centerline (and hence the complete folded state) to be deduced. For $m > 2$, in addition to the centerline(s) being invisible, some walls may be obscured as well. In this case, it is not clear how to deduce the complete folded state.

Unfortunately, the top-edge views are generally complex and difficult to study. Tracking two separate lines for each tunnel edge quickly gets unwieldy as the number of creases grows. In particular, constructing a valid top-edge view is difficult because there is little obvious locality in the structure. The "ray diagram" introduced in the next section alleviates this issue.

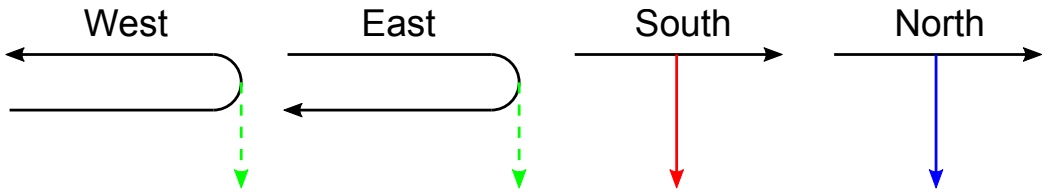

Figure 2-3: The ray diagram representation of west, east, south, and north vertices.

2.2 Ray Diagram

The *ray diagram* is a convenient two-dimensional representation of a folded state of a $2 \times n$ map. Note that as with the top-edge view, the ray diagram does not apply to $m \times n$ map folding for $m \neq 2$. The ray diagram's usefulness for solving map folding can be seen through its equivalence to the top-edge view. Since finding a valid top-edge view is equivalent to deciding flat foldability, the $2 \times n$ map folding problem reduces to deciding whether a valid ray diagram exists.

The ray diagram represents the centerline with a solid, black line. By convention, the centerline is drawn from left to right.[2] Mechanically, the four atomic components of the top-edge view (shown in Fig. 2-1) have equivalent ray diagram representations (shown in Fig. 2-3). Because east and west vertices correspond to changes in tunnel direction from valley to mountain (or vice versa), these two vertex types are represented by 180° turns in the centerline. West corresponds to an upward turn; east corresponds to a downward turn, as shown in Fig. 2-3. The green dotted ray extends downward to infinity; it marks the location of a east or west vertex. North and south vertices do not change tunnel directions. They are denoted by a red ray (South) or a blue ray (North) extending downward from the centerline. Points where the red and blue rays cross the centerline (including the rays' starting point) are called north and south markers, respectively. The point of origin for east and west rays is an east or west marker; further intersections of these green rays with the centerline are irrelevant. In Fig. 2-4, we give a ray diagram representation of the sequence NEW equivalent to the top-edge view for NEW in Fig. 2-1.

As previously defined, a segment on a ray diagram is represented by the region

[2]Given the previous notion of tunnels, we can arbitrarily label a rightward centerline as a valley tunnel and a leftward centerline as a mountain tunnel.

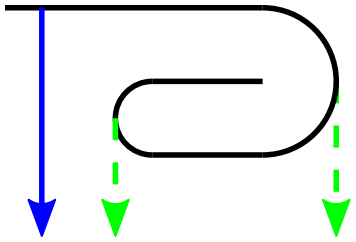

Figure 2-4: A valid ray diagram representing the sequence NEW, equivalent to the top-edge view in Fig. 2-1 (middle).

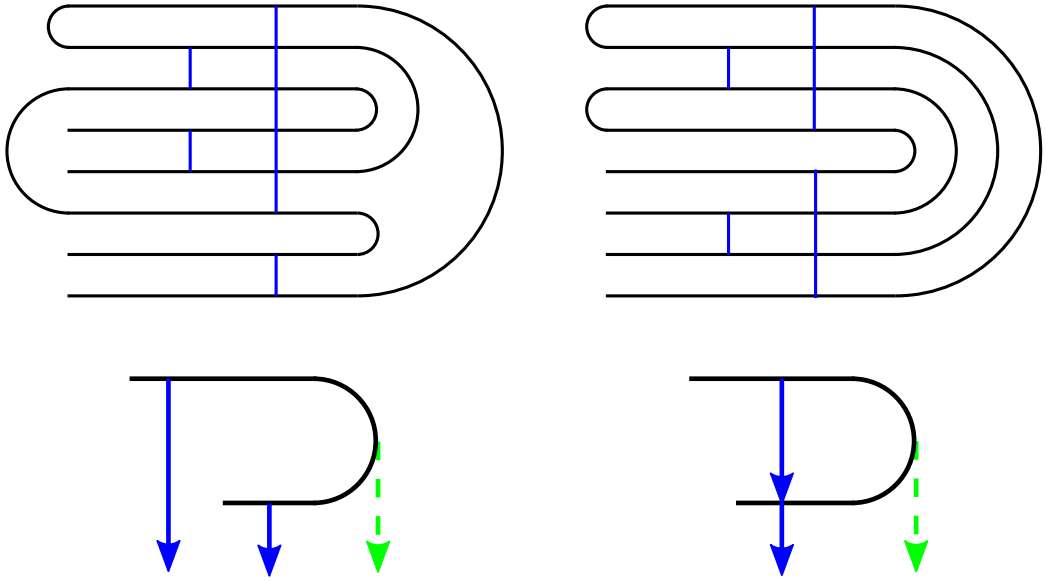

Figure 2-5: The two valid top edge views for the sequence NEN and their corresponding ray diagrams.

on the centerline between two adjacent markers. Here, the two end points of the centerline are counted as markers. For example, the colored parts of the centerlines in Fig. 2-6 and Fig. 2-7 are segments. Unlike segments (walls) in the top-edge view, segments of the ray diagram's centerline do not directly represent lengths of paper. Variations in distances are for convenience only. In a ray diagram, only the relative position of markers is important, not the actual distances between them.

We say that a segment s is *constrained* if its downward projection intersects other segments at every point of s. That is, at each point along s, cast a ray downward to infinity. If all of these rays intersects other segments, then s is constrained. For example, the left and middle of Fig. 2-6 depict constrained segments, while the rightmost drawing depicts an unconstrained segment. Fig. 2-7 also contains additional examples

of constrained and unconstrained segments. Recall that each segment represents a part of a tunnel. Intuitively, the set of all segments intersected by rays cast from s correspond to other tunnels "tucked" inside of the tunnel represented by s.

Lastly, note that a ray diagram for a particular crease pattern is not unique; neither is the top-edge view. There are often multiple choices for how to perform a particular folding, and neither representation makes assumptions about such choices. For example, Fig. 2-5 shows two valid top edge views and ray diagrams for the sequence NEN. In the left figure, all segments are unconstrained; as a result, the first N has no bearing on the rest of the diagram. In the right figure, one segment is constrained. As we will discuss, having the two N rays intersect in the ray diagram indicates that the folds are "tucked in" to each other, an effect which is clearly visible in the top-edge view.

2.2.1 Constraints on Valid Ray Diagrams

If the ray diagram satisfies the following three constraints, then the $2 \times n$ map it represents is flat foldable. Again, the converse is also true. The constraints are as follows:

1. The centerline cannot self-intersect.
2. North and south rays can only be (vertically) intersected by a ray of a matching type; additionally, any north or south ray reaching the centerline *must* intersect a matching ray.
3. There must be equality between the number of north and south markers visible to a constrained segment;[3] this constraint does not count the endpoints bounding a constrained segment.

Fig. 2-6 shows the three possibilities with segments: a valid constrained segment, an invalid constrained segment, and an unconstrained segment. Fig. 2-7 is an example of a more complex and valid ray diagram. Generally, a ray diagram that violates the constraints corresponds to a folding that self-intersects or attempts a nonphysical

[3] A marker is visible to a segment if the upward projection of that marker first intersects the segment in question.

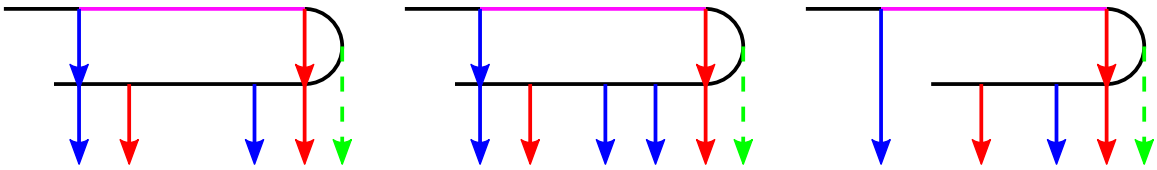

Figure 2-6: Constrained and unconstrained segments. The highlighted (pink) segment is: (left) constrained, satisfying constraint 3; (middle) constrained, violating constraint 3; (right) unconstrained.

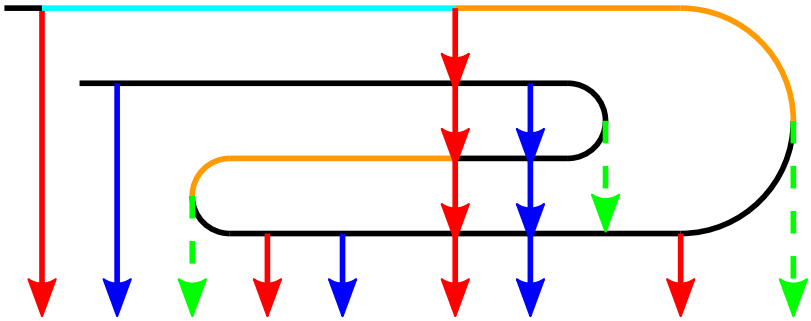

Figure 2-7: An example of a valid ray diagram. The diagram has two nontrivial constrained segments (orange), which satisfy constraint 3. It also has one nontrivial unconstrained segment (cyan). Reading from the upper left, this diagram represents a way to flat fold SSESNSNSWSNWNSN.

vertex labeling (e.g., labeling a vertex north and south simultaneously); more specifics about the necessity of these constraints are given below.

These conditions are necessary for the ray diagram to correspond to a valid flat folding. Note that a flat foldable sequence may have multiple ray diagram representations (as with top-edge views), but not all of these representations will be valid; consider SENNS, for example. The necessity of the first condition is straightforward; violating it corresponds to self-intersection of the paper or the formation of a closed loop.

The second constraint represents the previously discussed idea that only a north can fit inside another north on a different tunnel (and similarly for souths). This idea should be intuitive because the crease directions would not match if one tried to place a north inside of a south. So if an outer tunnel has a N or S, the tunnels contained on its interior must match those norths or souths to prevent self-intersection. For example, in the right side of Fig. 2-5, the second N is folded inside the first N. If the

sequence were NES instead, then an arrangement similar to the left of Fig. 2-5 would still be possible. However, the right side arrangement would be impossible since an S cannot be tucked inside an N. But having the N and S markers match is not enough on its own. As mentioned in regard to the top-edge view, N and S folds correspond to turns in the centerline. So for an S to fit in another S, their centerline "states" must match, as discussed in the next paragraph. Finally, it is not valid for an outer tunnel to have N (or S) markers while the inner tunnel(s) do not. This is easily seen with the top-edge view, where it is clear that an inner tunnel must follow all the folds of its containing tunnel, although it may have additional folds.

The third constraint ensures that the tunnel "states" match to prevent self-intersection. Consider the nonflat folding sequence ENWS; one ray-diagram representation is shown in Fig. 2-8. This sequence is not flat foldable because it violates constraint 3. The top-edge view (Fig. 2-9) makes this clear: the S fold has nowhere to go. Notice that adding an extra S after the N (ENSWS) to satisfy constraint 3 results in a flat foldable sequence.

In the intuition for a constrained segment, we noted that the segments constraining it (i.e., those segments below the constrained segment) correspond to folds occurring on the interior of the tunnel represented by the constrained segment. Constrained segments are problematic because their tunnel region must be compatible with the other tunnel(s) on their interior. Having a mismatched number of N and S markers visible to a constrained segment means that the states of the tunnels will not match. For example, with an odd number of extra N markers, the interior tunnel will be pointed in the wrong direction. This causes ENWS to be non-flat foldable as show in Fig. 2-8 and Fig. 2-9. Or with an even number, the interior tunnel will be pointed in the right direction, but it will have spiraled in on itself. The arrangement on the right side of Fig. 2-5 can also be used to see why NESN and NESSN cause problems whereas NESNN is flat foldable.[4] As long as the number of N and S markers is mismatched, self-intersection can occur, which should be clear from the top-edge view. Satisfying

[4]This is more of a thought experiment, since all of these sequences are flat foldable unless we require that the first and last N rays intersect.

constraint 3 resolves this issue with constrained segments.

Finally, note that constraint 3 is not an issue for unconstrained segments. Unconstrained segments are essentially independent of the rest of the folding in that their bounding markers can be removed without affecting the global foldability. Contiguous sequences of unconstrained segments are trivially folded flat just based on Maekawa's local foldability conditions. Unconstrained region can be folded-flat and then appended to a tunnel at the beginning or end of the vertex sequence or tucked inside of an existing tunnel.

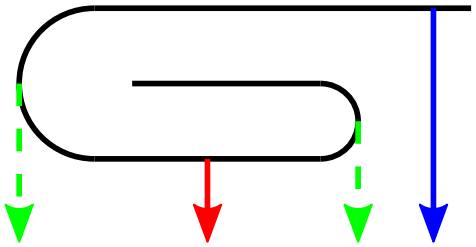

Figure 2-8: One possible ray diagram for the nonflat folding sequence ENWS. This diagram (and all other possible diagrams for ENWS) violates constraint 3. Note that this example is minimal in the sense that removing any one of the four markers results in a flat foldable sequence.

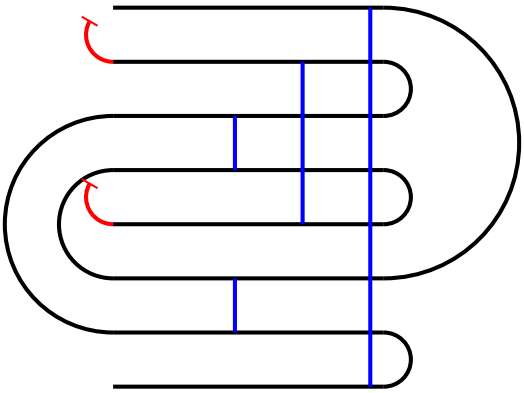

Figure 2-9: One possible top-edge view for the nonflat folding sequence ENWS. See Fig. 2-8 for the ray diagram for this sequence. The S vertex is not drawn fully; it is noted in red because there is no valid way to draw it.

2.2.2 A Simple Application

Certain $2 \times n$ crease patterns simplify into 1D folding problems. In particular, if the centerline is a single tunnel (mountain or valley), deciding flat foldability is easy [1]. This case is represented by a ray diagram with no E or W markers. Then the entire centerline is unconstrained, and any sequence of N and S markers will be flat foldable. The other 1D case occurs when only E and W markers are present. Then every vertical crease is either entirely a mountain or entirely a valley—reducing the

problem to 1D. Again, the ray diagram shows that any sequence of E and W markers will be flat foldable, because a single line with 180° turns need not self-intersect. These two 1D cases only use simple folds, covering the cases discussed in [1].

Chapter 3

Algorithms for $2 \times n$ Special Cases

In this section, we will describe efficient algorithms for various special cases of $2 \times n$ map folding. A summary of these algorithms can be found in Table 3.1.

3.1 Cone

Here we examine the case when we have a vertex sequence without any Es or without any Ws. In either case, we can construct a ray diagram in a "cone" shape, as shown in Fig. 3-1. With this shape, we draw each layer larger than the one below it and put all Ns and Ss to the side so they don't hit any anything below them. As a result, there are no intersections between segments and N or S rays, and there are no constrained segments so the diagram is always valid. Therefore, we know that any vertex sequence without any Es or without any Ws is flat foldable. Thus we have an optimal $O(1)$

Table 3.1: $2 \times n$ Map Folding Special Case Results

Special Case	Vertex Restriction	Running Time
Cone	No Es or no Ws	$O(1)$
Interlocking Teeth (only Ns or Ss)	One alternation between Es and Ws and no Ns or no Ss	$O(n^3)$
Interlocking Teeth	One alternation between Es and Ws	$O(n^5)$
Spiral	No consecutive Es or Ws	$O(n^4)$

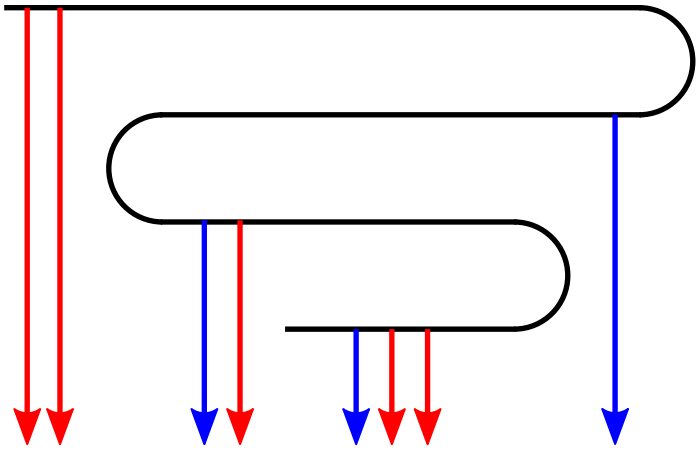

Figure 3-1: A valid ray diagram construction for a $2 \times n$ map with no Ws.

time algorithm for checking if a $2 \times n$ map without any Es or without and Ws is flat foldable - the answer is always yes.

3.2 Interlocking Teeth

Now we will examine the case in which our vertex sequence consists of a sequence of Es followed by a sequence of Ws (or Ws followed by Es) with Ns and Ss interspersed. The restriction here is that there can be only one alternation between Es and Ws. We refer to this case as "teeth" due the interlocking between the Es and the Ws, as seen in Fig. 3-2. Once we add this single alternation (as opposed to the zero alternations in the previous case), we can no longer trivially construct a ray diagram in a way that involves no interaction among the "teeth." Moreover, there are many possible ways to go about constructing the ray diagram as any of the exponentially many interlockings of the teeth could be necessary. We will overcome this problem with a simple dynamic programming algorithm.

The region of the centerline on either side of every other E or W, which protrudes into the middle of the structure will be referred to as a "tooth". We will work upwards from the bottom of the rows of teeth, placing one tooth at a time. We can think of this as incrementally layering one tooth after another onto the stack. A natural element of the state space is thus which tooth is next to be considered in each row. We observe

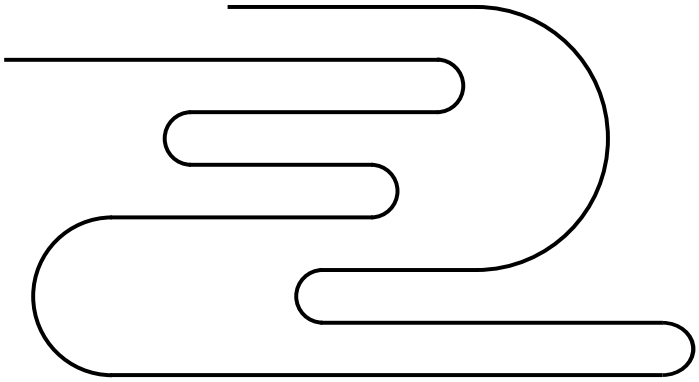

Figure 3-2: A ray diagram of "teeth"

that at a given step of this algorithm, the only information that affects our future decisions is what set of rays are 'visible' from where the next teeth will be layered. How those rays came to be visible (what the previous layering of teeth was exactly) is unimportant, since it will not effect future tooth placement. Because of this, if we add to the state a precise representation of the visible rays, then it will be complete. At a given step of the algorithm there are only three distinct places for rays to be visible from: to the left of the start of the next left tooth, to the right of the start of the next right tooth and in between the two teeth as depicted in Fig. 3-3. If we did not have any Ss (or we did not have any Ns) then we could simply encode exactly how many Ns (or Ss) there are in these three regions, giving us $O(n^3)$ possibilities and a total state space of $O(n^5)$ (including which teeth are next). However, once we include both Ns and Ss in our diagram, we can no longer easily enumerate the possible sets of rays visible at a given state as there are now 2^n possibilities for a given segment rather than just n. In order to efficiently encode the state, we will first need the following lemma.

Lemma 3.2.1. *Given a fixed starting place, when selecting which N and S rays to intersect with below a given segment, it is optimal to cover the minimum number of N and S rays below the segment.*

The proof of Lemma 3.2.1 can be found in Appendix A. Given this lemma, we observe that the rays visible to the left of the left tooth are uniquely determined by which left tooth it is (and similarly for the rays to the right of the right tooth).

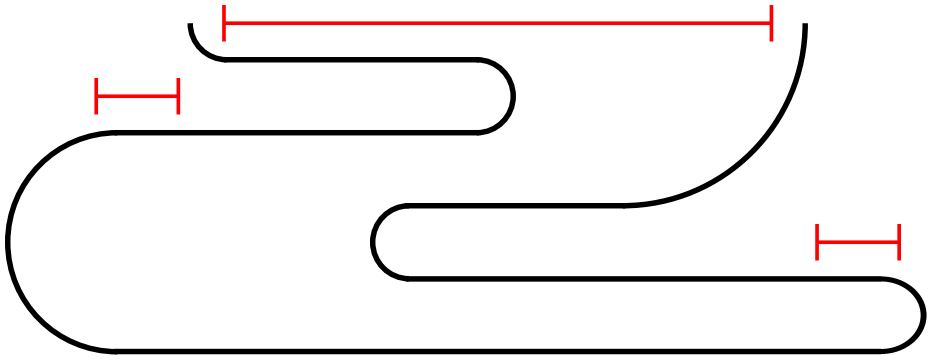

Figure 3-3: The three distinct regions of visible rays in the dynamic program state for teeth

This is true by induction on the left teeth (working upwards) since the selection and placement of right teeth does not change the rays visible to the left of the left tooth. Additionally, what is visible to the left of the left tooth after placing a left tooth is determined only by what is visible there before placing the tooth as, by Lemma 3.2.1, there is one optimal way to place the tooth. Thus our dynamic program does not need any additional information beyond which tooth was last placed on each side in order to encode the contents of the leftmost and rightmost visible regions. This observation immediately reduces state space of the case with no Ns or no Ss to $O(n^3)$.

Now all that remains is to succinctly represent the area in between the two teeth. This can be done by tracking the last (highest) interlocking between a left and right tooth. We define an *interlocking* here as the overlap between a left and right tooth such that no other tooth fits in between them. This can be done simply in $O(n^3)$ space, as there are $O(n^2)$ possible selections of one tooth from each side, and $O(n)$ possible ways for them to overlap. We can further reduce the space to specify the interlocking to $O(n^2)$ by observing that the index of the first ray not covered up by the top tooth simultaneously tells us how the teeth overlap, and what the index of the bottom tooth. Given this overlap, no rays from teeth below it are visible. Additionally, as there are no overlapping teeth above this pair (by definition), any teeth that have been placed above the interlocking pair can be built up in any order, thus giving us the complete picture of visible rays. Thus our total state space is $O(n^4)$.

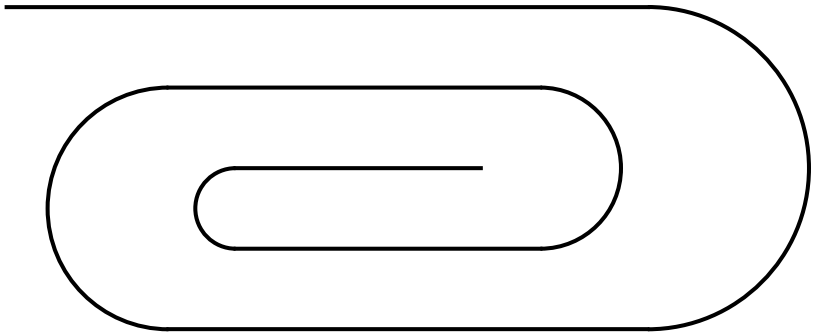

Figure 3-4: A spiral in the ray diagram

As a result of Lemma 3.2.1, once we have chosen which tooth to place, there is only one way to place it and we can find this placement in $O(n)$ time. Thus, the total running time of our algorithm is $O(n^5)$. We can do even better for the case with no Ss (or no Ns). This is because if we precompute the number of Ns (or Ss) in each tooth, then we can compute the resulting number of exposed Ns (or Ss) in $O(1)$ time so the whole dynamic program takes $O(n^3)$ time.

3.3 Spiral

With the interlocking teeth problem we restricted our sequences of Es and Ws to one alternation in order to prohibit spiraling. In this section we will show that spiraling, at least in its simplest form, is not difficult to deal with. We will be working with the subproblem of $2 \times n$ map folding in which we have an alternating string of Es and Ws with Ns and Ss interspersed. This sequence does not guarantee a single spiral; instead, it could generate a double spiral. But for now we will assume that it forms a single spiral. Once we have an algorithm for checking the sequence assuming it forms a single spiral, we will extend the algorithm to allow for double spirals.

The basic idea behind the algorithm is: first check the feasibility of the innermost level of the spiral and then work outwards. In Fig. 3-5, we depict the inner-most section of a spiral. One should note that if the last direction was an E (possibly followed by Ns and Ss) instead of the W depicted here, then there is nothing to check as there would just be an A and a C and no B. In this case, all of the Ns and Ss in

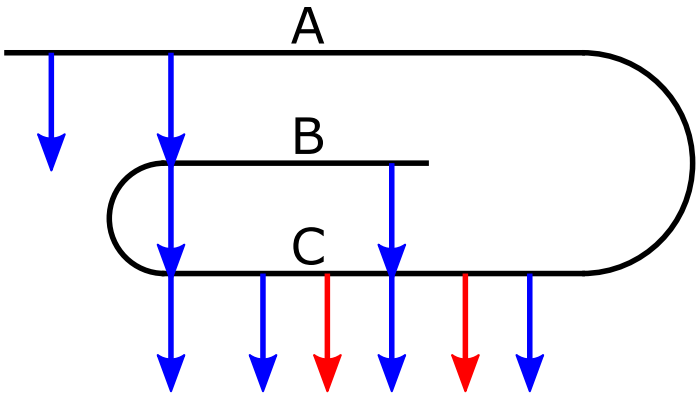

Figure 3-5: The inner most layer of the spiral

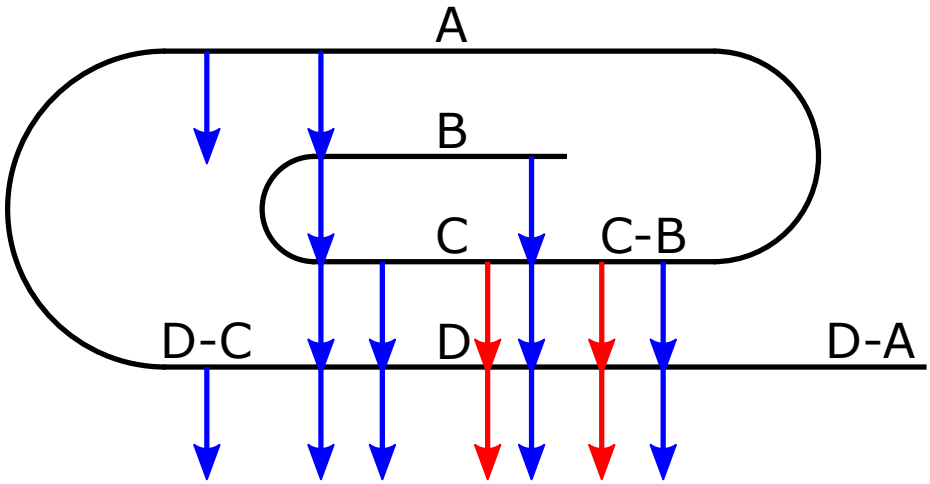

Figure 3-6: One step out from the inner most layer of the spiral

A could be placed to the left of C and there would not be interaction between A and C. However, in the case depicted in Fig. 3-5, a flat folding is not always possible and we must do some work to check if it is possible to build the ray diagram.

In this case, the Ns and Ss in A are not of interest, since they can all dangle off the left side, making A independent of both B or C. However, B and C must interact so we must check if it is possible to build the ray diagram according to the necessary constraints. In order for a valid ray diagram to exist, B must be a constrained substring of C. To place B's rays with respect to C we will simply use our procedure from Lemma 3.2.1. We know that among all possible arrangements this one is optimal and takes $O(n)$ to compute. If this procedure reports that no such arrangement exists then we know that our map is not flat foldable.

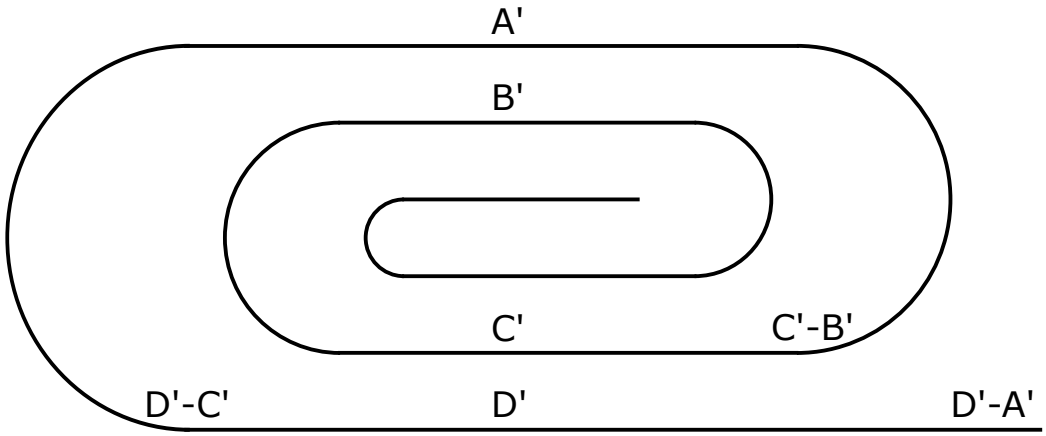

Figure 3-7: Two steps out from the inner most layer of the spiral

Now, we step out from this center a little bit more, as depicted in Fig. 3-6. Not only must B be a constrained substring of C, but A must be a constrained substring of the concatenation of D-C, B, and C-B (hereafter we will call this E) where D-C is the part of D sticking out to the left of C, and C-B is the amount of C sticking out to the right of B. Also, C must be a constrained substring of D. Our algorithm to check feasibility is as follows. We already know exactly what B and C-B are from our previous step thus all that remains in order to place A is D-C. There are only $O(n)$ possible D-C, so we will check them all. For each one, we check if A is a constrained substring of E in $O(n)$ using essentially the same procedure as in Lemma 3.2.1. If it is, we then use the procedure from Lemma 3.2.1 to arrange C. If such an arrangement exists we will have found D-A. By Lemma 3.2.1 it is optimal to maximize the size of D-A, thus if there are multiple possible arrangements we select the one that maximizes D-A. The running time of these steps is $O(n^2)$ as we perform $O(n)$ computations for each of the $O(n)$ possible D-C.

Next, we step out one more level to the diagram in Fig. 3-7. B' and C' in Fig. 3-7 correspond directly to A and D from Fig. 3-6, respectively. We observe that by renaming labels, we have the same problem as in the previous step, and can use the exact same algorithm. Thus, we can check feasibility and compute D'-A' in $O(n^2)$. We can then keep repeating this process as we iterate outwards, thus performing $O(n^2)$ work $O(n)$ times for a total running time of $O(n^3)$.

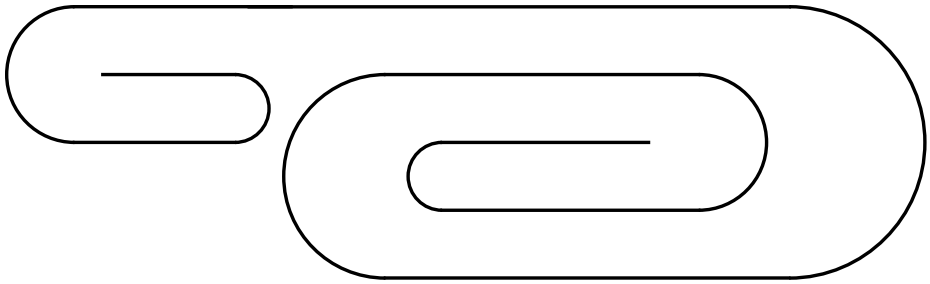

Figure 3-8: A double spiral

As mentioned earlier, an alternating sequence of Es and Ws will not necessarily form a simple spiral; it could form a double spiral as depicted in Fig. 3-8. Luckily, this is easy to account for. There are only $O(n)$ possible distinct double spirals that a given sequence could form, since there are $O(n)$ points at which it could transition from one spiral to another. We can simply iterate through these $O(n)$ possibilities, and check the individual spirals in each case in $O(n^3)$ time. The two spirals do not interact at all if we position them as in Fig. 3-8 and the the top segment can put all of its Ns and Ss between the two spirals thus avoiding interacting with either. Thus, we can check the flat foldability of any vertex sequence consisting of alternating Es and Ws (with any Ns and Ss in between) in $O(n^4)$ time.

Chapter 4

Algorithm for General $2 \times n$ Map Folding

In this section, we will present an algorithm for determining the flat foldability of any $2 \times n$ map. We will present this algorithm in two steps. In Section 4.2, we reduce the problem of finding a ray diagram via a kind of dual transform to the problem of finding a valid "tree structure". In Section 4.1, we show how to solve the resulting "hidden tree problem" in polynomial time using a roughly quadratic-time dynamic program, which because of the sequence of reductions and oracle calls, results in an overall running time of $O(n^9)$ for map folding.

4.1 Hidden Tree Problem

Before we proceed to our final reduction, we need to introduce the target problem, called the *hidden tree problem*. In this problem, our goal is to construct a "valid" tree on a given polynomial set V of candidate vertices, given two oracles to navigate and test hypothetical partially constructed trees. Each vertex $v \in V$ has a known degree $d_v \geq 1$ in any valid tree containing v. Not all of the candidate vertices in V need to be used in the solution; in fact, selecting some vertices may preclude others.

Validity of a candidate tree T is defined by the oracle Valid, by calling Valid$(T, \emptyset)$. More generally, Valid(T, P) determines whether a partially constructed tree is so far

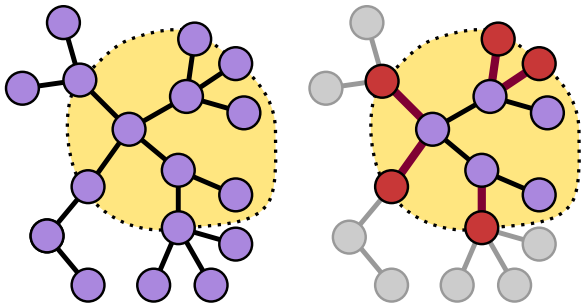

Figure 4-1: A tree (left) and a partial tree thereof (right). Dark/red vertices are partial.

valid, and thus could conceivably be completed into a valid tree. More precisely, a *partial tree* (T, P) is a tree $T = (V_T, E_T)$ on a subset V_T of V, with the vertices in $P \subseteq V_T$ marked *partial*, meaning that there may be additional edges in the complete tree missing from the partial tree T; see Figure 4-1. As in the figure, we will guarantee that every partial vertex in P is a leaf within the partial tree T, and thus the oracle only needs to be defined for such partial trees.

To help navigate the space of (partial) trees, we are also given a *separation oracle*. Given a vertex $v \in V$, Separate(v) divides the candidate vertex set V into disjoint subsets $V_1, V_2, \ldots, V_{d_v}$ such that every valid tree T that includes vertex v has d_v connected components in $T - v$, and the components can be numbered so that the ith component uses vertices only from V_i.

These oracles make the hidden tree problem decomposable in a sense, enabling us to solve the problem in polynomial time. Before we give this algorithm, however, we must formally define the axioms we require of the validity and separation oracles.

4.1.1 Validity Oracle

The task of the *validity oracle* Valid(T, P) is to decide whether a given partial tree (T, P) is valid under the constraints of the problem. For every nonleaf (and hence nonpartial) vertex v of T, let $v_1, v_2, \ldots, v_{d_v}$ denote the d_v neighbors of v in T. Then the removal of v from T results in d_v connected components; label them $C_1, C_2, \ldots, C_{d_v}$, where $v_i \in C_i$. Let C_i^+ denote the tree resulting from C_i by adding vertex v and edge (v, v_i). Then the validity oracle Valid must have the property that the partial tree

(T, P) is valid precisely when every component C_i^+ is valid, where v is treated as a new partial vertex in each component. More precisely, we must have

Validity Property: For every vertex v of T, $\text{Valid}(T, P) = \bigwedge_{i=1}^{d_v} \text{Valid}\left(C_i^+, (P \cap C_i) \cup \{v\}\right)$.

4.1.2 Separation Oracle

The *separation oracle* $\text{Separate}(v)$ returns d_v disjoint subsets $\text{Separate}(v, 1)$, $\text{Separate}(v, 2)$, $\ldots$, $\text{Separate}(v, d_v)$ of the remaining candidate vertex set $V - \{v\}$. The separation oracle must fulfill the following three properties:

Separation Property: For any valid complete (or partial) tree T, the removal of vertex v from T results in d_v connected components that can be numbered $C_1, C_2, \ldots, C_{d_v}$ such that C_i only uses vertices from $\text{Separate}(v, i)$.

Symmetry Property: For any candidate vertices $u, v \in V$, $v \in \bigcup \text{Separate}(u)$ if and only if $u \in \bigcup \text{Separate}(v)$. Intuitively, either of these memberships represents that candidate vertices u and v are consistent together, and this consistency should be the same viewed from either side.

Acyclic Property: If $u \in \text{Separate}(v, i)$, then $\text{Separate}(u, j) \subset \text{Separate}(v, i)$ for all j except the one where $v \in \text{Separate}(u, j)$. This property guarantees that recursive application of the separation oracle results in a nested tree structure.

4.1.3 Algorithm

Theorem 4.1.1. *The hidden tree problem can be solved in* $O(|V|^2\Delta(t_{\text{Valid}} + t_{\text{Separate}} + \Delta))$ *time, where* Δ *is the maximum degree,* t_{Valid} *is the time to compute* $\text{Valid}(T, P)$*, and* t_{Separate} *is the time to determine which subset returned by* $\text{Separate}(v)$ *contains a given vertex* u*.*

We prove this theorem by giving a dynamic-programming algorithm for the hidden tree problem. Define the subproblems of the dynamic program as follows: $f(v, i)$ is a 0/1 value indicating whether there is a tree T on $\text{Separate}(v, i) \cup \{v\}$, in which v is a leaf, such that $\text{Valid}(T, \{v\})$. There are $|V|\Delta$ such subproblems.

We claim that we can compute $f(v, i)$ recursively as follows:

$$f(v,i) = \bigvee_{u \in \text{Separate}(v,i)} \Bigg(\text{Valid}(\{v,u\}, \{v\}) \vee \Bigg((|\text{Separate}(u)| \geq 2 \vee \text{Valid}(\{v,u\}, \{v,u\})) \wedge \bigwedge_{j : v \notin \text{Separate}(u,j)} f(u,j) \Bigg) \Bigg). \tag{4.1}$$

Intuitively, this recurrence searches over all possible vertices u that could be adjacent to v in the subtree. For a given u, it checks two cases. In the first case, the subtree consists only of a single edge and thus u is a leaf ($\text{Valid}(\{v,u\}, \{v\})$). In the second case, u is not a leaf; thus, to compute the validity of the whole subtree, we partition around u in the manner of the Validity Property. We prove correctness formally below.

Next we claim that the hidden tree problem reduces to solving the subproblems. Our goal is to determine whether there is a complete tree T such that $\text{Valid}(T, \emptyset)$. By the Validity Property, this condition is true if and only if there is a candidate vertex $v \in V$, and trees $T_1, \ldots, T_k$ sharing vertex v but no other vertices, such that $\text{Valid}(T_i, (P \cap T_i) \cup \{v\})$ for $i \in \{1, 2, \ldots, d_v\}$. By the Separation Property, T_i must be within $\text{Separate}(v, i) \cup \{v\}$, so this condition is true if and only if there is a candidate vertex $v \in V$ such that $f(v, i) = 1$ for all $i \in \{1, 2, \ldots, d_v\}$.

Thus, given solutions to all of the subproblems, we can solve the hidden tree problem by computing $\bigvee_{v \in V} \bigwedge_{i=1}^{d_v} f(v, i)$. Using the standard back-pointer technique of dynamic programming, we can also find the desired tree if it exists, with just a constant-factor overhead.

By memoizing the recurrence Equation 4.1, we spend $O(|V|(t_{\text{Valid}} + t_{\text{Separate}} + \Delta))$ time once for each of the $|V|\Delta$ subproblems. Therefore the total running time is $O(|V|^2 \Delta(t_{\text{Valid}} + t_{\text{Separate}} + \Delta))$.

4.1.4 Correctness

We prove by induction on $|\text{Separate}(v, i)|$ that the recurrence Equation 4.1 produces the intended $f(v, i)$ values.

The base case is $|\text{Separate}(v, i)| = 1$. In this case, if there is a valid tree T on (a subset of) $\text{Separate}(v, i)$ such that $\text{Valid}(T \cup \{v\}, \{v\})$, then it must consist of only a single vertex u where $\text{Separate}(v, i) = \{u\}$. The recurrence in this case simplifies to just $f(v, i) = \text{Valid}(\{v, u\}, \{v\})$ as desired.

Suppose that there is a tree T on a subset of $\text{Separate}(v, i)$ such that $\text{Valid}(T \cup \{v\}, \{v\})$. Let u be the vertex adjacent to v in T. The case in which u is leaf vertex follows immediately from the recurrence, as T then consists of the single edge (v, u), so by taking the disjunction over all possible u, $f(v, i) = 1$. If u is a nonleaf vertex, then by the Validity Property,

$$\text{Valid}(\{v, u\}, \{v, u\}) \wedge \bigwedge_{j : v \notin \text{Separate}(u,j)} \text{Valid}((\text{Separate}(u, j) \cap T) \cup \{u\}, \{u\}) = 1.$$

The recurrence Equation 4.1 checks every possible vertex u that could be adjacent to v in T. By the Acyclic Property, $|\text{Separate}(u, j)| < |\text{Separate}(v, i)|$ for all j with $v \notin \text{Separate}(u, j)$. Thus, by our inductive hypothesis, $f(u, j) = 1$ if and only if there exists a $T' \subseteq \text{Separate}(u, j)$ such that $\text{Valid}(T' \cup \{u\}, \{u\})$. Because $\text{Separate}(u, j)$ and $\text{Separate}(u, j')$ are disjoint for $j \neq j'$, $\bigwedge_{j : v \notin \text{Separate}(u,j)} f(u, j) = 1$ if and only if $\bigwedge_{j : v \notin \text{Separate}(u,j)} \text{Valid}((\text{Separate}(u, j) \cap T) \cup \{u\}, \{u\}) = 1$ for some T. Thus $f(v, i) = 1$.

Similarly, if there is no tree $T \subseteq \text{Separate}(v, i)$ such that $\text{Valid}(T \cup \{v\}, \{v\})$, then $f(v, i) = 0$ because Equation 4.1 implicitly constructs a T and checks that it is valid.

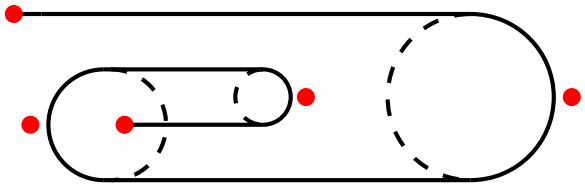

Figure 4-2: An example of where the tree vertices lie in the plane relative to the ray diagram.

4.2 Reducing Ray Diagrams to the Hidden Tree Problem

4.2.1 Tree Structure

Any valid ray diagram has an underlying tree structure. We will define this tree to lie in the same plane as the ray diagram. Intuitively, the vertices correspond to the Es and Ws and their position in the ray diagram. The edges correspond to sequences of Ns and Ss. To differentiate the vertices in the tree from the vertices and edges of the original map, we will refer to them as *tree vertices* and *tree edges*. We will begin by describing how tree vertices are placed before defining their role in the tree.

Definition 4.2.1 (Tree Vertex Location)**.** *A* tree vertex *corresponds to an E, W, or one of the two endpoints of the centerline and the point on the centerline directly above and below this point. In the plane of the ray diagram, the tree vertex lies just outside the half circle describing the turn in the centerline due to the E or W, as depicted in Fig. 4-2. In the case of a tree vertex corresponding to an endpoint, the vertex will lie exactly on the end of the centerline.*

Each tree vertex represents a gluing of two points on the centerline through an E or W.

Definition 4.2.2 (Gluing Line)**.** *A tree vertex's gluing line is a line formed by projecting a line up and down from the tree vertex until it hits the centerline. No part of the centerline can intersect a gluing line.*

Gluing lines are drawn as dotted lines in Fig. 4-3; arrowheads on gluing lines

indicate gluing to infinity. Each tree vertex must maintain some information about its location and the rays around it in order to construct the hidden tree problem.

Definition 4.2.3 (Tree Vertex)**.** *The tree vertex represents a gluing between the E, W or endpoint and two points on the centerline (Definition 4.2.2). Because this vertex may lie within a constrained segment (the two rays it lies between in the top segment), it additionally must track the difference between the Ns and Ss visible to the top segment on the left of itself (and the number visible to the right must sum with this to zero).*

As a special case for an E, W, or an endpoint that does not have a segment above or below it, these fields may be empty. This represents the E, W, or endpoint being glued to infinity.

A tree vertex v tracks at most the following information:

1. *v.index, the index of the E, W or endpoint — $O(n)$;*
2. *v.top, the index of the ray to the left of its gluing line's intersection with the top segment — $O(n)$;*
3. *v.bottom, the index of the ray to the left of its gluing line's intersection with the bottom segment — $O(n)$;*
4. *v.difference, the difference between the number of Ns and Ss visible to the top segment on the left of its gluing line— $O(n)$;*

for a total of $O(n^4)$ possible tree vertices.

Two tree vertices have a tree edge between them if they are *visible* to each other. We say that one tree vertex is visible to another if there is a path between them which does not intersect the centerline or the gluing line of another tree vertex at any point. By this definition, each endpoint tree vertex has degree at most three. As a special case, the leftmost tree vertex (one of the two which are glued above and below to infinity) does not have an edge to vertices below it. This prevents the cycle that would otherwise occur around the exterior of the ray diagram.

Observe that the gluing lines corresponding to two adjacent tree vertices bound two contiguous segments of centerline; e.g., the two orange gluing lines bound the

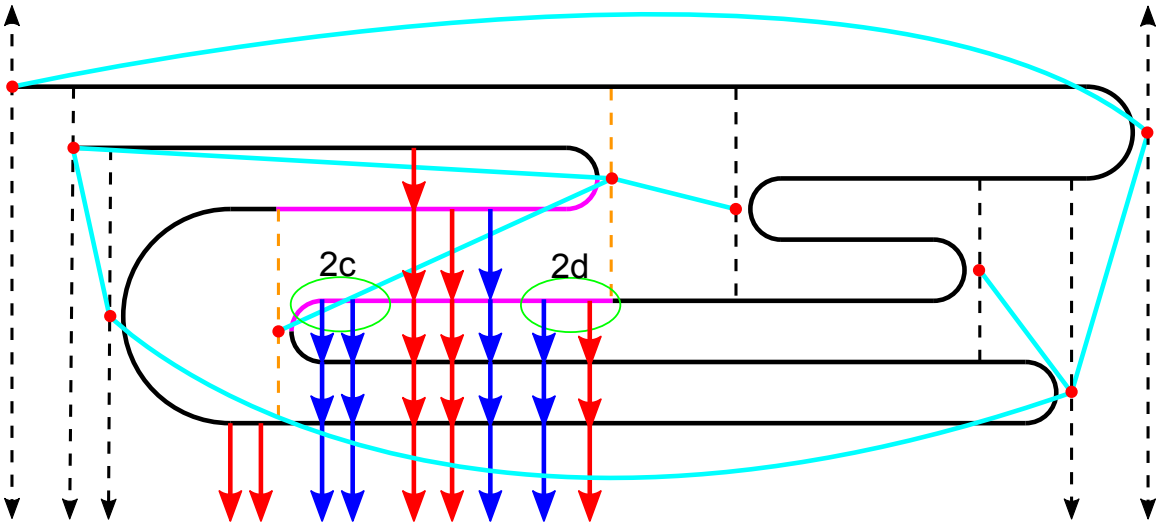

Figure 4-3: A valid ray diagram with its tree representation (red nodes, cyan edges). Gluing lines are dotted. The two orange gluing lines demarcate the purple segments of the centerline. In these regions, the labels 2c and 2d correspond to rays checked in 2c and 2d of the validity oracle (see subsection 4.2.4).

two purple centerline segments in Fig. 4-3. One of these segments is the *top* and the other is the *bottom*. Observe that once the tree vertices are chosen and their gluings established, no part of the centerline can cross in between these two segments. We will consider the N and S rays along these segments to "belong" to the tree edge connecting the tree vertices. The validity oracle will ensure that the N and S rays along the top segment are arranged relative to the N and S rays in the bottom segment so as to satisfy ray diagram constraints 2 and 3.

As a special case, the gluing lines might not intersect the centerline (they go up or down to infinity). In this case, the centerline cannot cross between the segment (if there is one) and the corresponding infinity. The task of the validity oracle is much simpler here as regardless of which direction the gluing line goes to infinity in, constraints 2 and 3 are easily satisfied.

Fig. 4-4 depicts a valid ray diagram and Fig. 4-5 depicts the underlying tree structure for that diagram.

This tree exactly defines the layout of the ray diagram, as the location of each E, W and centerline endpoint is determined by a tree vertex and the location of each N

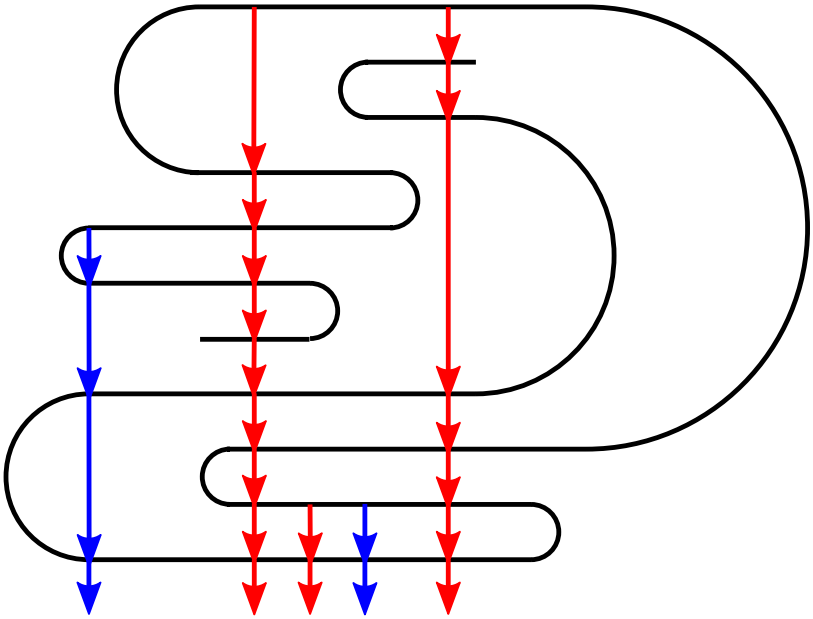

Figure 4-4: A valid ray diagram.

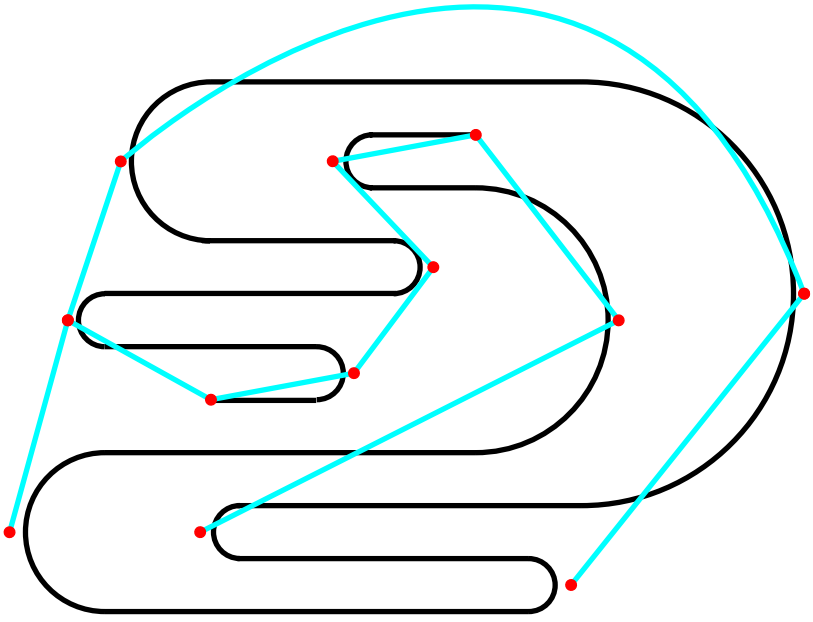

Figure 4-5: The underlying tree representation corresponding to Fig. 4-4.

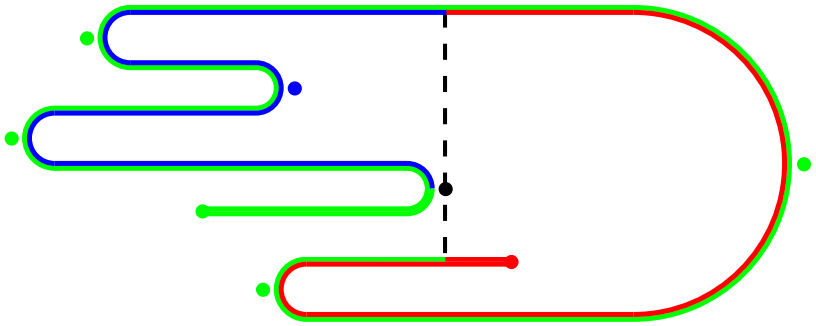

Figure 4-6: A representation of the loops that a (black) tree vertex partitions the diagram into. Here the different loops are colored red, green and blue. In places the centerline is split in half between two loops. The colored vertices describe which loop each E, W and endpoint falls into, thus how the set of possible tree vertices is restricted.

and S is determined by a tree edge or adjacent tree vertex. We can thus determine the validity of the ray diagram (whether it obeys both constraints) directly from these trees.

4.2.2 Loop Partition

A tree vertex v gives a gluing which separates the diagram into a set of closed loops. A loop is defined by a contiguous region of the centerline, together with v's gluing line. These loops may split the centerline into two halves, as seen in Fig. 4-6. For example, in some regions of the centerline, the green and blue sections represent the two halves, whereas the regions of the centerline with only one color are not split. When the centerline is split, ownership of an E or W ray (for separation oracle purposes) belongs to the loop containing the outer half of the centerline along the E or W semicircle.[1] Responsibility for an N or S ray may be split across two loops. Imagine that each ray starts in the top half of the centerline. Consider some N ray i_N. The loop containing the top half of the centerline must know about ray i_N so that it can decide whether any ray coming from above intersects ray i_N. The loop containing the bottom half of the centerline must also know about ray i_N so that it can decide whether i_N intersects any ray below it. In particular, the loop containing the bottom

[1]Here, "outer" has the same meaning as it did in the specification of tree vertex locations (Definition 4.2.1).

half of the centerline at i_N communicates with rays that i_N could intersect with via their upper centerline halves.

4.2.3 Separation Oracle

Algorithm 1 Separate(v, i)

1. If $i = 1$: let *loop* be the loop of centerline to the side of v
2. If $i = 2$: let *loop* be the loop of centerline above v
3. If $i = 3$:
   (a) If $v.top = \infty$ and $v.bottom = -\infty$ and v is the start of the centerline or has even parity among E and W rays: let $loop = \emptyset$
   (b) Else: let *loop* be the loop of centerline below v
4. Let $vertices = \emptyset$
5. For each E, W or endpoint in *loop*

   For each bottom-side ray in *loop*

   For each top-side ray in *loop*
   For i in $[-n, -n+1, \cdots, n-1, n]$
   Let $v.index$ be the index of the E, W or endpoint
   Let $v.top$ be the index of the bottom-side ray
   Let $v.bottom$ be the index of the top-side ray
   Let $v.difference = i$
   Append v to *vertices*
6. Return *vertices*

The separation oracle Separate(v) is defined as follows. The tree vertex v gives a gluing which separates the diagram into three loops, as described in the previous section. Two loops will lie on the interior and one on the exterior. Each of these loops corresponds to one of the tree vertex sets returned by the oracle. A tree vertex u is unreachable within one of these loops (and thus does not fall in its tree vertex set) if the E, W or endpoint of u is not contained in the loop. Additionally, if the top or bottom segment that u's E, W or endpoint glues to lies outside the loop, then u is not reachable. From the rays (or parts of the rays) the fall in a loop, we determine which tree vertices may appear in the corresponding vertex set, as described in Algorithm 1.

Lemma 4.2.4. *The oracle given in Algorithm 1 satisfies the Separation Property.*

Proof. By the definition of gluing lines and visibility, a tree vertex u falling in one of the loops defined by a tree vertex v may not have an edge to a tree vertex w falling

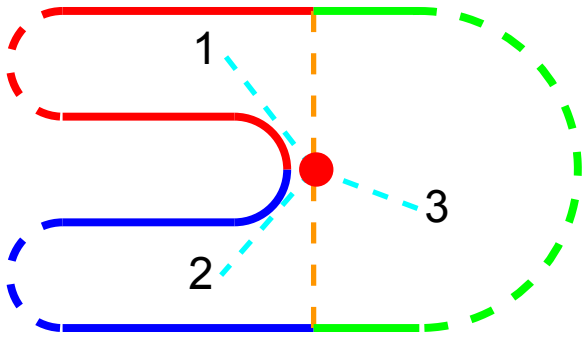

Figure 4-7: Special cases for the validity oracle (Property 3) at an E or W tree vertex. As a special case, the pictured vertex has degree less than three; selecting a dotted tree edge to keep "solidifies" that dotted edge and deletes its corresponding dotted semicircle. For example, keeping tree edge 1 deletes the red semicircle.

in one of the other loops defined by v, because the centerline of the loop and gluing line of v partition the plane. Therefore, removing v must necessarily disconnect all tree vertices falling in different loops. □

Lemma 4.2.5. *The oracle given in Algorithm 1 satisfies the the Acyclic Propertyand the Symmetry Property.*

Proof. Given one of the loops defined by a tree vertex v and a tree vertex u contained in that loop, all of the loops defined by u will be subsets of the loop defined by v except for the one containing v. Since the tree vertices are defined as a combination of rays in a given loop, the Acyclic Property and Symmetry Property follow immediately. □

4.2.4 Validity Oracle

A partial tree vertex set P in this context defines a subset of the ray diagram. Specifically, it defines a closed loop on the centerline after the gluings defined by the partial tree vertices are made. Informally, this loop could be around the exterior of the diagram or on the interior. The validity oracle walks along this loop on the centerline and returns false if any ray within this region is unaccounted for by the tree vertices in the tree T. To do this it must check the following properties:

1. Each E, W and endpoint within the loop is used by exactly one tree vertex.

2. For each tree edge, there is an arrangement of its corresponding N and S rays such that:
   (a) Each N or S ray coming from the top segment meets a corresponding ray on the bottom segment. (Constraint 2)
   (b) For a given pair of adjacent rays in the top segment, in the set of rays that lie on the bottom segment in between the top rays, the number of Ns must equal the number of Ss. (Constraint 3)
   (c) The difference between the number of Ns and Ss on the bottom segment to the left of the leftmost top ray must agree with the number specified by the left tree vertex. (Constraint 3, see Fig. 4-3)
   (d) The difference between the number of Ns and Ss on the bottom segment to the right of the rightmost top ray must agree with the number specified by the right tree vertex. (Constraint 3, see Fig. 4-3)
3. For each tree vertex that is not partial and has degree less than three[2], we need account for the N and S rays lying in region(s) lacking a tree edge. This check performs the previous step on an isolated part of the tree. The components of this special case are shown in Fig. 4-7. For example, suppose we had a degree two tree vertex with tree edges 1 and 3 in place. Then only the blue semicircle exists, and the parts of Property 2 above would need to be checked for N and S rays in the blue loop.
4. Every N and S is accounted for by one of the previous two checks.

Checking N, S, E, W and endpoint usage can easily be done in $O(n)$. For a given tree edge which contains k rays, its properties can checked with a simple greedy algorithm in $O(k)$ time, as described in Appendix A. We can similarly check the assignment of $O(k)$ Ns and Ss in a direction missing an edge from a vertex in $O(k)$ time.

Lemma 4.2.6. *The validity oracle defined in this section satisfies the Validity Property.*

Proof. We now argue that this oracle satisfies the Validity Property for any tree vertex

[2]In the case of the special globally leftmost tree vertex, this condition is replaced by "with degree less than two".

$v \in T$. Intuitively, the second and third properties (the nontrivial ones) are local to a given tree vertex or tree edge so whichever partition they fall into will account for them. Specifically, if a tree is invalid because of not having a valid arrangement of Ns and Ss, then the partition that the Ns and Ss fall in will be invalid. Similarly, if T is invalid because of not accounting for some rays, then no matter what tree vertex v it is partitioned around, the partition C_i that is lacking those rays will be invalid. □

4.2.5 Correctness

We will now show that the reduction is a correct one. That is, a valid hidden tree exists if and only if a valid ray diagram exists. First we argue that, for any tree T such that Valid$(T, \emptyset)$, T corresponds to a valid ray diagram. By the definition of the validity oracle, if Valid$(T, \emptyset)$ then T accounts for all of the rays in the ray diagram. Thus we must show that the arrangement of these rays fulfills the three constraints of a valid ray diagram. Non-intersection of the centerline is guaranteed by the planar and connected nature of the tree. Constraint 2, the fact that each N or S ray that intersects the centerline must do so at the origin of a corresponding N or S ray is guaranteed by the validity oracle checks, which explicitly specifies that each top N or S ray must meet a corresponding ray on the bottom segment. Constraint 3 is guaranteed by a combination of the validity oracle check and the definition of the tree vertex. If a constrained segment is not separated by a tree vertex (no tree vertex's gluing line falls inside the constrained segment) then it will be explicitly checked by the validity oracle. If a constrained segment is separated by a tree vertex, then by definition of the tree vertex (and guaranteed by the validity oracle) the difference between the number of Ns and Ss visible to the constrained segment on the left will sum to zero with the difference between the number of Ns and Ss visible to the constrained segment on the right.

Finally, we argue that, for any valid ray diagram, there is a corresponding tree T such that Valid$(T, \emptyset)$. This follows directly from the definitions. The tree vertices in T correspond to the parts of the ray diagram described in Definition 4.2.3. As in Fig. 4-5, we place a a tree vertex at each E, W or endpoint with its gluing lines extending up

and down to where they intersect the centerline. The edges formed by the visibility rules described at the beginning of the section. The properties checked by the validity oracle are all necessary conditions for a ray diagram to satisfy constraints 2 and 3, so $\text{Valid}(T, \emptyset) = 1$.

4.2.6 Running Time

The number $|V|$ of candidate vertices is $O(n^4)$. The maximum degree Δ of a vertex is 3. The time to evaluate the separation oracle and validity oracle is $O(n)$. Thus, by Theorem 4.1.1 the total running time is $O(n^9)$.

Chapter 5

$2 \times n$ Map Folding Extensions

The algorithm presented in the previous chapter for determining the flat foldability of a $2 \times n$ map is a powerful one, and in this section we will extend it to optimization problems. Rather than simply determining whether a flat folding exists, we will find the best possible flat folding according to some objective function. In particular, we will find the flat foldings which are, by some metric, simplest. In order to do this, we must show that our objective function is decomposable in the same way that the Validity Property is. Using the same notation as the Validity Property, we much have an objective function

$$f(T, P) = g\left(\left(C_1^+, (P \cap C_1) \cup \{v\}\right), \cdots, \left(C_{d_v}^+, (P \cap C_{d_v}) \cup \{v\}\right)\right)$$

for some function g. Our algorithm will then be identical to that of Theorem 4.1.1, except that we our dynamic program will select the v that minimizes f at each steps, among all v that are valid (according to the original dynamic program).

The most well studied measure of simplicity in folding is the maximum *crease stretch*, also known as the maximum *creep*. This is the maximum number of folds of paper within a single crease in a folded state. Unfortunately, minimizing the maximum creep has been shown to be NP-complete, even for $1 \times n$ maps [7]. As a result, the objectives we will be examining will only heuristically minimize creep.

5.1 Sum of Constrained Segments

Our first objective function that we will minimize is the total number of N and S rays that are visible to constrained segments. As described in subsection 2.2.1, N and S rays that are visible to constrained segments correspond to segments of tunnel that wind in and then back out of the region that they are nested in. Large constrained segments therefore result in large creep for the region of tunnel to which they correspond.

The tree structure that we will use to minimize this functions is exactly the one described in Section 4.2. We can evaluate the objective for a given tree and set of partial vertices by looping over all of the edges in the tree and summing the number of rays visible to the constrained segments corresponding to the regions of centerline between the two vertices. Conveniently, this is just the number rays in the bottom segment minus the number of rays in top segment. Note that we do not have to add in $v.\mathit{difference}$, because those rays will already be counted by the tree edges. Thus we can easily minimize the number of N and S rays that are visible to constrained segments in the same time that we can find any valid folded state, $O(n^9)$.

5.2 Maximum Constrained Segment

Our next objective function is the maximum number of N and S rays that are visible to a constrained segment. We first have to slightly modify the tree structure of in Section 4.2. This is because the value of the objective for an edge does not account for the tree vertices contribution to it. The rays corresponding to the constrained segment that spans a tree vertex's gluing line are not accounted for by one any one tree edge. Thus the vertex itself has to account for them, however $v.\mathit{difference}$ does not contain enough information, as it only tracks the difference between Ns and Ss while we need to count the total number of N and S rays. Thus in addition to $v.\mathit{difference}$, a vertex v will contain $v.\mathit{leftNs}$ and $v.\mathit{rightNs}$ which are the number of Ns and Ss respectively visible to the top segment on the left of v's gluing line. Therfore, there are now a

total of $O(n^6)$ possible tree vertices.

Evaluating the objective on a given tree and set of partial vertices is also more expensive than the original algorithm, because when arranging the N and S rays corresponding to an edge in the tree, we cannot simply use the $O(n)$ greedy algorithm described in Appendix A. This is because the arrangement produced by the greedy algorithm will not necessarily minimize the maximum number of bottom N and S rays between an adjacent pair of top rays. Instead, we will use a simple $O(n^3)$ dynamic program to arrange the N and S rays corresponding to a tree edge so as to minimize the objective for that edge. The state of the dynamic program is the index of leftmost unmatched top ray and the rightmost matched bottom rays ($O(n^2)$ state space). $O(n)$ time is needed to compute the value of each state, as there are $O(n)$ choices of bottom rays to match the current top ray to.

The maximum number of N and S rays visible to a constrained segment for a given tree and partial vertex set is equal to the maximum over each tree edge and the maximum value of $2(v.leftNs + v.rightNs)$ for each tree vertex v. The total running time is now $O(n^{15})$ since the $|V|$ is $O(n^6)$ and the time to evaluate the validity oracle is $O(n^3)$.

5.3 Sum of North and South Nestings

The objective of interest here is the sum of the heights of each "stack" of Ns or Ss. A stack here refers to the sequence of intersections of N and S rays. As described in subsection 2.2.1, N and S rays intersecting other N and S rays corresponds to nesting the region of tunnel corresponding to on into the other, thus long sequences of intersecting Ns or Ss corresponds to long nestings of tunnel. Once again, in order to minimize creep, one heuristic approach would be to minimize this nesting.

Conveniently this objective is equal to the total number of N and S rays, minus the number that do not intersect the centerline. This is easily maintained with the original tree structure specified in Section 4.2. The objective function evaluated on a given tree and set of partial vertices is simply equal to the total number of Ns

and Ss contained in the corresponding region, minus the number of Ns and Ss whose bounding tree vertices are glued below to infinity. This can be computed with no more time than the original dynamic program, yielding an $O(n^9)$ algorithm to minimize our objective.

Observe that the problem of minimizing the maximum height of a stack is not easily solved using these techniques. This is because in order to compute it, the tree vertices would have to somehow determine what their "height" is within the tree diagram, however this is a highly nonlocal quantity which does not obey the decomposability property that we require.

Chapter 6

$m \times n$ Map Folding

In this section we present some results for general $m \times n$ map folding. First, in Section 6.1, we will give polynomial time testable conditions that are necessary for an $m \times n$ map to fulfill in order to be flat foldable. In Section 6.2 we will give a fixed parameter tractable algorithm for deciding the flat foldability of an $m \times n$ map, where the parameter is the entropy of the partial order defined by the map's creases.

6.1 Necessary Conditions

As discussed in subsection 1.1.3, any valid folded state must be consistent with the partial order on the cells defined by the mountain valley assignment of the map. If this partial order is unsatisfiable, due to having cycles (and thus not be a consistent partial order) then no valid folded state exists for the map. Thus there being no cycles in the partial order is a necessary condition for an $m \times n$ map to flat foldable.

Because of the non-crossing conditions for the creases, we can draw further implications about partial order. In particular, take two pairs of adjacent cells, (i, j) and (k, l) such that the partial order specifies that $i < j$ and $k < l$ and the crease between each pair lies on the same side (top, bottom, left or right) of the final folded state. Because we know that we cannot have $i < k < j < l$ or $k < i < l < j$, we can draw the following implications:

- If $i < k$ and $j < l$, then $j < k$.

- If $i < k < j$ then $l < j$.
- If $i < l < j$ then $i < k$.

We can repeatedly apply these rules to the partial order until no more exist or a cycle forms. If a cycle forms, then the map is not flat foldable. This is clearly testable in polynomial time, as we can naively try all $O(mn)$ pairs of adjacent cells to see if one meets this criteria, and we need can only do this at most $O((mn)^2)$ times, as the partial order cannot have more than that many edges.

6.2 Fixed Parameter Tractable Algorithm

Here we present an algorithm for determining if an $m \times n$ map is flat foldable in $O((mn)^{2.5}\ 2^{\log e(P)})$ time, where $\log e(P)$ is the entropy in the partial order P. We achieve this by using the algorithm of Cardinal et al. for sorting a partial order in $O(n^{2.5})$ time using $O(\log e(P))$ comparisons [3]. We run this algorithm, except at each comparison we branch and try both possibilities, hence the exponentiation. This will result in every possible total order consistent with the partial order, which we then check each of in $O(n^2)$ time using the verification algorithm described in subsection 1.1.3. The total running time is then $O(((mn)^{2.5} + (mn)^2)2^{\log e(P)}) = O((mn)^{2.5}\ 2^{\log e(P)})$, where the $(mn)^2$ comes from the verification algorithm. Note that we can further improve the performance of this algorithm by using the implications described in Section 6.1, as they will strictly decrease the entropy of the partial order.

Bibliography

[1] Esther M. Arkin, Michael A. Bender, Erik D. Demaine, Martin L. Demaine, Joseph S. B. Mitchell, Saurabh Sethia, and Steven S. Skiena. When can you fold a map? *Computational Geometry: Theory and Applications*, 29(1):23–46, September 2004.

[2] Marshall Bern and Barry Hayes. The complexity of flat origami. In *Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, SODA '96, pages 175–183, Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics.

[3] Jean Cardinal, Samuel Fiorini, Gwenaël Joret, Raphaël M. Jungers, and J. Ian Munro. Sorting under partial information (without the ellipsoid algorithm). In *Proceedings of the 42nd ACM symposium on Theory of computing*, pages 359–368, 2010.

[4] Erik D. Demaine, Satyan L. Devadoss, Joseph S. B. Mitchell, and Joseph O'Rourke. Continuous foldability of polygonal paper. In *Proceedings of the 16th Canadian Conference on Computational Geometry (CCCG 2004)*, pages 64–67, Montréal, Québec, Canada, August 9–11 2004.

[5] Erik D. Demaine and Joseph O'Rourke. *Geometric Folding Algorithms: Linkages, Origami, Polyhedra*. Cambridge University Press, New York, NY, 1st edition, July 2007.

[6] Jacques Justin. Towards a mathematical theory of origami. In Koryo Miura, editor, *Proceedings of the 2nd International Meeting of Origami Science and Scientific Origami*, pages 15–29, Otsu, Japan, November–December 1994.

[7] Takuya Umesato, Toshiki Saitoh, Ryuhei Uehara, and Hiro Ito. Complexity of the stamp folding problem. In *Proceedings of the 5th international conference on Combinatorial optimization and applications*, COCOA'11, pages 311–321, Berlin, Heidelberg, 2011. Springer-Verlag.

Appendix A

Ray Diagram Greedy Algorithm

In this section we give an $O(n)$ algorithm for determining Property 2 of the ray diagram validity oracle, and prove its correctness. Lemma 3.2.1 will follow from this as corollary, in which $v_{left} = 0$ and v_{right} is ignored.

Algorithm 2 NsAndSsArrangable($A, B, v_{left}, v_{right}$)

1. Let $b = 1$ and $diff = 0$
2. While $diff \neq -v_{left}.difference$ and $b \leq |B|$
   (a) If $B[b] = N$: Add 1 to $diff$
   (b) Else: Subtract 1 from $diff$
   (c) Add 1 to b
3. For a in $\{1, 2, \cdots, |A|\}$
   (a) Let $diff = 0$
   (b) While ($A[a] \neq B[b]$ or $diff \neq 0$) and $b \leq |B|$
      i. If $B[b] = N$: Add 1 to $diff$
      ii. Else: Subtract 1 from $diff$
      iii. Add 1 to b
   (c) If $b > |B|$: Return False
4. Let $diff = 0$
5. While $b \leq |B|$
   (a) If $B[b] = N$: Add 1 to $diff$
   (b) Else: Subtract 1 from $diff$
   (c) Add 1 to b
6. If $diff = v_{right}.difference$: Return True
7. Else: Return False

Algorithm 2 is the greedy algorithm for checking Property 2 of subsection 4.2.4 in $O(k)$ time, where k is the number of N and S rays in the regions of centerline defined

by the tree edge. A is the top segment of the centerline, B is the bottom segment of the centerline and v_{left} and v_{right} are the tree vertices that whose gluing lines bound them. A and B are implemented here as arrays of Ns and Ss. The basic idea is for each ray of A to find the leftmost ray of B to intersect with such that the rays match and constraint 3 is satisfied by obeying $v_{left}.difference$ and $v_{right}.difference$ and making sure an equal number of N and S rays of B appear between adjacent rays of A.

Lemma A.0.1. *Algorithm 2 correctly determines Property 2 of the validity oracle.*

Proof. We are considering the scenario in which we have some set B of N and S rays, and we are placing another centerline segment A over it starting from the left (all of this logic works going right to left as well). Because of the arrangement of the tree vertices, A is prohibited from extending to past B; thus all rays of A must intersect with some ray in B. We can then define the arrangement of the rays of A as an injective mapping $f : 1 \cdots |A| \to 1 \cdots |B|$ from ray indices in A to ray indices of B. As a matter of notation, let $B[i : j]$ be the set of rays in B from index i to index j inclusive, which is empty for $i > j$. Also, let $d(R)$ be the difference between the number of Ns and Ss in the set of rays R. In order to fulfill Property 2, f must fulfill the following properties:

$$A[i] = B[f(i)], \forall i \in 1..|A| - 1, \tag{A.1}$$

$$d(B[1 : f(1) - 1]) = -v_{left}.difference, \tag{A.2}$$

$$d(B[f(i) + 1 : f(i + 1) - 1]) = 0, \forall i \in 1..|A| - 1, \tag{A.3}$$

$$d(B[f(|A|) + 1 : |B|]) = v_{right}.difference. \tag{A.4}$$

The intuition behind Algorithm 2 is that, when arranging the rays from left to right, by choosing the left most feasible ray of B to assign a ray of A to, we maximize the number of options for future rays of A. Given a valid arrangement f, we will

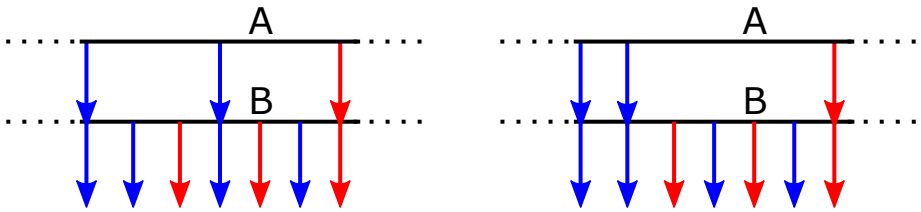

Figure A-1: Two different ways to arrange a set of rays A that is consistent with a set of rays B

construct an another arrangement f' as follows. f' will be the identical to f except for some index $k \in 1 \cdots |A|$, where $f'(k) < f(k)$. Fig. A-1 gives an example of possible assignments of f and f'. We will show that the greedy property holds, and therefore Algorithm 2 is correct, by showing that any f' that fulfills Equation A.2, Equation A.3 and Equation A.1 for indices $1 \cdots k-1$ is a valid assignment.

f' immediately satisfies Equation A.1, because f satisfies it and f' satisfies it for index k. Similarly, f' clearly satisfies Equation A.3 for all indices greater than k. All that remains is to show that Equation A.3 is satisfied for index k, that is $d(B[f'(k)+1 : f'(k+1)-1]) = 0$. Because $f'(k-1) = f(k-1)$, $f'(k+1) = f(k+1)$ and $A[k] = B[f(k)] = B[f'(k)]$,

$$\begin{aligned} d(B[f(k-1)+1 : f(k+1)-1]) &= d(B[f(k-1)+1 : f(k)-1]) + d(B[f(k)]) + \\ &\quad d(B[f(k)+1 : f(k+1)-1]) \\ &= d(B[f(k)]) \\ &= d(B[f'(k-1)+1 : f'(k+1)-1]) \\ &= d(B[f'(k-1)+1 : f'(k)-1]) + d(B[f'(k)]) + \\ &\quad d(B[f'(k)+1 : f'(k+1)-1]) \\ &= d(B[f(k)]) + d(B[f'(k)+1 : f'(k+1)-1]). \end{aligned}$$

Thus $d(B[f'(k)+1 : f'(k+1)-1]) = 0$ as desired. If $k < |A|$, then f' immediately satisfies Equation A.4. If $k = |A|$, then f' satisfies Equation A.4 by an argument nearly identical to that of Equation A.3.

□