Embedding paths with directional constraints on a grid

Blijjenberg, Wietske T.P.

Award date:
2023

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Algorithms, Geometry & Applications

Embedding paths with directional constraints on a grid

Master's thesis

Wietske Blijjenberg

01-05-2023

Supervision:
Wouter Meulemans

Assessment committee:
Wouter Meulemans
Fernando Paulovich
Bart Jansen

Credits: 30

This is a public Master's thesis.

This Master's thesis has been carried out in accordance with the rules of the TU/e Code of Scientific Conduct.

Abstract

Grid maps are a useful tool to visualize complex spatial data, however, drawing high quality grid maps can be challenging. In this thesis, we investigate how to draw paths with directional constraints on an integer grid as a means to construct grid maps where the relative positioning of spatial elements is mostly preserved. Although previous work has attempted to preserve relative positioning using a 4-approximation algorithm and point-set matching, it has failed to identify the complexity of preserving relative positioning in the context of point-set matching. We provide an exact algorithm for the problem of drawing a path with directional constraints on an integer grid and prove that this problem is NP-hard. This indicates that drawing graphs with directional constraints on an integer grid is NP-hard as well.

Contents

Chapter 1

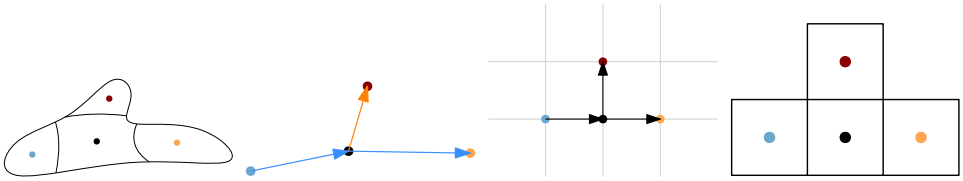
Introduction

Visualisation of data is an effective strategy for understanding said data. Data with spatial elements (countries, municipalities, cities) is often visualised on a map, but the more complex the data becomes, the more difficult it is to preserve the readability of the map while also keeping elements at the correct spatial location. Thus, instead of maps with high spatial accuracy, it might be practical to use grid maps: a schematization technique where every spatial element is schematized into a grid tile. The consistent shapes of the grid tiles have the benefit of making each spatial element equally easy to observe.

Creating a grid map, however, introduces its own challenges, like how to best preserve properties of the underlying data such as location, adjacency, and relative positioning of the spatial elements. Techniques for creating a grid map exist [8] [9] [11], but can still be improved upon, as they struggle with efficiently computing a tile arrangement that accurately represents salient features of the containing shape, and keeps the boundary smooth. Another challenge is to find a suitable mapping between geographic regions and grid cells [14].

In this thesis, we study a problem that might be useful for improving upon existing techniques: the problem of embedding a graph with directional constraints on a grid. In this context, embedding a graph on a grid means drawing the graph such that each vertex of the graph is placed at an intersection on the grid, each edge of the graph has equal length, and each edge of the graph follows an edge of the grid.

A set of spatial elements can be mapped to vertices in a graph, for example by letting each centre of a spatial element be a vertex. The relationship between spatial elements can then be modelled as edges with directional constraints. For example, two adjacent spatial elements can be modelled as adjacent vertices u and v, and we can give the edge $e = (u, v)$ between them a directional constraint (e.g. up, down, left, right) specifying their relative positioning: if, for example, the centre of the second spatial element v is located mostly to the left of the centre of the first spatial element u, edge e



(a) A set of spatial elements.

(b) A possible graph G obtained from the set of spatial elements.

(c) G drawn on a grid such that as many directional constraints as possible are satisfied.

(d) A grid map obtained from the grid embedding of G.

Figure 1.1: An example of how grid embeddings of graphs with directional constraints might be used to construct grid maps. Centres of different spatial elements are indicated with differently coloured vertices. Edges with directional constraint *right* are indicated in blue, edges with directional constraint *left* are indicated in orange.

gets directional constraint *left*.

After drawing a graph with directional constraints representing spatial elements and their relative positioning on a grid such that as many directional constraints as possible are satisfied and no two vertices are drawn at the same intersection of the grid, we have obtained a graph G' that specifies a way to place centres of spatial elements such that their relative positioning is mostly preserved, no two spatial elements are placed on the same spot, and the distance between each two adjacent spatial elements is equal. We can obtain a grid map from such a graph by replacing each vertex of the graph with a grid cell representing the spatial element belonging to the vertex. As such, we obtain a grid map where as many spatial elements as possible are in the correct relative position. An example of what this process might look like is given in Figure 1.1.

To reduce the problem complexity, we reduce our problem space to path graphs – as shown in Chapter 5, this reduced problem is already NP-hard. We can use path graphs to model curves like territorial outlines: given a curve representing a territorial outline, we can divide this curve into segments based on some requirement – for example, we can choose segments of equal lengths, or if the territory consists of multiple geographical subdivisions, we can divide the territorial outline into segments based on the different geographical subdivisions the parts of the outline belong to. Each segment starts at a vertex v_i and ends at a vertex v_{i+1}. The relative position of these vertices on the curve can be mapped to a directional constraint, which will belong to the edge $e = (v_i, v_{i+1})$. Understanding how to embed path graphs on a grid can be a first step to understanding how to embed other kinds of graphs on a grid. We limit our set of directional constraints to $\{Up, Down, Left, Right\}$, which corresponds to grid maps with square

tiles.

We formulate the problem studied in this thesis as follows: given an input path P with directional constraints, we want to embed P on an integer grid such that as many directional constraints as possible are satisfied, with the restriction that we are not allowed to visit any intersection of the grid more than once: each vertex of P should be placed at a unique intersection in the grid. Our solution consists of a set S of k edge flips, such that after applying the k edge flips to P, the resulting path P' does not contain any cycles. Our goal is to minimise k.

Contributions and organization. The rest of this thesis is organised as follows: in Chapter 2, we define some notation and terminology used throughout the thesis. In Chapter 3, we explore the problem, which includes describing and proving some properties of the problem as well as proving a lower and upper bound on the number of directional constraints to violate. In Chapter 4 we discuss an exact algorithm for the problem, including its correctness and running time. In Chapter 5 we prove that the problem is NP-hard. Finally, in Chapter 6, we discuss our findings and possible directions for future research.

1.1 Related work

1.1.1 Constructing grid maps

The practical use and manual construction of grid maps, also called tile maps [9] or equal area unit maps [14], has been discussed in many blog posts [15] [18] [19]. However, considerably less research exists about the automatic generation of grid maps.

Most automated methods of generating grid maps are only applicable in simple cases, for example when the tile arrangement is a complete grid, and the input has a close-to-uniform spatial distribution. More complicated cases make it difficult to select a suitable tile arrangement.

McNeill and Hale [9] present a first approach to fully automatically generating grid maps that is suitable for more complicated case as well as simple cases. The algorithm they propose transforms region centroids such that neighbours are equidistant, but the relative orientation between neighbours is preserved. Then the boundary polygon is transformed to match the transformations of the region centroids, after which the same number tiles as regions are fitted into the boundary polygon. Finally, transformed centroids are matched to tile centroids minimizing the sum of squared distances.

Although McNeill and Hale focus on problems where the set of regions is contiguous, the resulting grid map is not necessarily contiguous. Furthermore, salient features are not always well-preserved using this method.

The most recent approach to automatically generating grid maps is the pipeline proposed in [11], which consists of three steps: first the containing shape is decomposed based on salient features and the number of spatial elements per part is recorded; then a mosaic cartogram based on the parts is computed, using the number of spatial elements as weight; and finally, point-set matching is used to assign spatial elements to the tiles. Each of these steps is a well-studied problem with multiple algorithmic solutions, which, given that they meet the requirements of each step, can theoretically all be used. While this pipeline retains more characteristic features than the method of McNeill and Hale, it is also slower. Furthermore, the use of mosaic cartograms sometimes leads to a jagged boundary where the input shape was smooth.

Both these methods of automatically computing grid maps make use of point-set matching to assign regions to tiles. Eppstein et al. [8] studied the point-set matching problem in the context of generating grid maps when the specified point set is (close to) rectangular. They consider three optimization criteria: preserving location, adjacency, and relative orientation. Since preserving adjacency is proven to be NP-hard, the authors focus on preserving location and directional relations. They propose a 4-approximation for preserving the latter, noting that it is unknown if the problem of preserving directional relations in the context of point set matching is NP-hard and that a better approximation factor or even a polynomial time algorithm might be obtainable.

1.1.2 Path simplification

Embedding a path with directional constraints on a grid can also be seen as a form of path simplification: we simplify a path such that the orthogonal order of vertices is mostly preserved, every edge has fixed length and is drawn in one of four directions, and the resulting path never crosses itself. The simplification of paths is well-studied, with one of the most popular methods being the classic line simplification algorithm by Douglas and Peucker [7]. This algorithm, however, is more focused on reducing the number of points necessary to represent a line, instead of limiting edge directions.

The most similar path simplification problem seems to be the d-oriented orthogonal-order preserving graph drawing problem. The input to this problem is an embedded path P and a positive integer d. The goal is to find an embedding of P that preserves the orthogonal order of all vertices in the input and in which every edge has a direction that is an integer multiple of $90/d^\circ$ [6]. The main difference between the problem studied in this thesis and the problem of generating d-oriented orthogonal-order preserving drawings is that in the former, edge lengths are fixed, and we solve any embedding issues by violating directional constraints. The latter, on the other hand, allows edges to have variable lengths but keeps the orthogonal order strictly

intact. Brandes and Pampel showed that the orthogonal-order preserving rectilinear graph drawing problem and the orthogonal-order preserving uniform edge-length drawing problem are NP-hard, even for embedded paths [1].

Another problem that focuses on limiting edge direction is C-oriented schematization: a form of line schematization where the admissible edge directions are limited to a given set C of directions [12]. In a grid embedding, the set C obviously consists of the directions parallel to the edges of the cells of the grid. There are many different variants of the C-oriented schematization problem, with different sets of requirements.

A common requirement of C-oriented schematization problems, however, is a certain similarity of the output path to the input path. To measure this similarity and thus the quality of the solution, quality measures such as the Fréchet distance or the Hausdorff distance of the path simplification from the set of input points are used [10]. In this thesis, neither of these similarity metrics are used: the output path is theoretically allowed to diverge considerably from the input path. As such, a solution where the relative positioning of vertices in the input path is better preserved is favoured above a solution where the output curve is within a certain distance of the input curve. This allows us a certain freedom that does not exist in C-oriented schematization. However, preservation of relative positioning of vertices already guarantees some sort of closeness to the input path, especially if few directional constraints are violated.

A natural result of requiring a certain similarity to the input path is that the edge lengths of the output path are close to the edge lengths of the input path. However, in a C-oriented schematization, different edges are often allowed to have different lengths, while a path that is embedded on a grid has the property that all edges have equal length.

Another common goal in C-oriented schematization is minimization of line segments, also called links. This is fundamentally different to the problem studied in this thesis: we need the output path to have exactly the same edges as the input path, as each edge represents an adjacency of two elements in the corresponding curve.

A variation between different C-oriented schematization problems is the input: while some versions only consider paths, some versions consider sets of paths, like the C-oriented routing problem as explained in [16], where the goal is to find, given a set of input paths, a set of non-crossing C-oriented paths that are homotopic to the input paths using as few links as possible. The C-oriented schematization problem has also been considered for polygons, for example in the area-preserving C-oriented schematization problem, where an additional requirement is to preserve the area of the input polygon [2] [3]. In this thesis, however, we limit our attention to single paths.

Another difference between the input of C-oriented schematization problems and the problem studied in this thesis is the directional constraints:

many C-oriented schematization problems take as input an arbitrary path, and produce as output a C-schematized path, while the problem studied in this thesis takes as input a path P with directional constraints, and produces as output the path P embedded on a grid such that no vertex in the grid is visited more than once and as many directional constraints as possible are satisfied. As such, C-oriented schematization might seem less limited in directions allowed for a certain path segment. However, the similarity requirement of C-oriented schematization problems means having to pick directions for line segments that are close to the original direction of the line segment, which limits options for directions after all – though options are still less limited than having a single directional constraint for each edge.

Some versions of C-oriented schematization problems do specify concrete requirements on directions for specific path segments: Delling et al. defined a schematization cost, which is the number of edges that are not drawn with their closest C-oriented direction, and gave an algorithm to compute C-oriented drawings of monotone paths that preserve the orthogonal order of vertices, are intersection-free, and have minimum schematization cost [5].

Another problem similar to embedding paths with directional constraints on an integer grid is the metro-map layout problem. In this problem, an embedded graph is to be redrawn octilinearly, often with some additional criteria. For example, the mental map of passengers should be roughly supported, and there should be as few bends as possible. The input to the metro-map layout problem is often a set of paths instead of a single path. The metro-map layout problem does allow lines to cross, but only when the crossings exist in reality, for example when we have a station; a single input path that does not cross itself is rarely allowed to cross itself in the output drawing. Furthermore, metro-map layout problems often require that these crossing lines continue straight on both sides of the crossing so that the lines are easy to follow. The metro-map layout problem is proven to be NP-hard [13].

Similarly to the problem studied in this thesis, some approaches to the metro-map layout problem restrict edge directions to the set of the three octilinear directions closest to the input direction to optimise relative positioning [17]. However, often no concrete constraints are given that preserve the orthogonal order of vertices. Moreover, edge length is variable in this problem, which makes it fundamentally different.

Chapter 2

Preliminaries

Input. Our input consists of a *path* P: a graph of which the n vertices can be listed in the order $v_1, v_2, \ldots, v_n$ such that the $n-1$ edges are (v_i, v_{i+1}) where $i = 1, 2, \ldots, n-1$. Given an edge $e = (v_i, v_{i+1})$, we use $source(e)$ to refer to vertex v_i, and $target(e)$ to refer to vertex v_{i+1}.

Each edge of P has length 1. Each edge has a *directional constraint*, which is one of {Up, Down, Left, Right}. We use $direction(e)$ to refer to the directional constraint of an edge e. Directional constraints are often shortened to their first letter, i.e. U, D, L, R. We often denote a path as a sequence of directional constraints, e.g. $P = UULLD$. Each direction has an opposite direction: U and D are opposite directions, and L and R are opposite directions. Let $A \in \{U, D, L, R\}$ be any direction. We use $\bar{A}$ to denote the direction opposite to A, e.g. $\bar{R} = L$. We use $\#A$ to denote the number of edges with directional constraint A in a path P.

Subpaths. A path $P = (V, E)$ consists of several *subpaths*: paths whose vertex set $V' \subseteq V$ and whose edge set $E' \subseteq E$. A subpath has a startpoint and an endpoint: the *startpoint* of a subpath is the first vertex of a subpath, and the *endpoint* of a subpath is the last vertex of a subpath. Likewise, the *startedge* of a subpath is the first edge of a subpath, and the *endedge* of a subpath is the last edge of a subpath.

Subpaths can overlap: we call subpaths P_i and P_j *vertex-wise overlapping* if they share vertices, but no edges. We call subpaths P_i and P_j *edgewise overlapping* if they share vertices and edges. Since in this thesis the sharing of edges is more relevant than the sharing of vertices, we often use *overlapping* to refer to edgewise overlapping.

We call a set W of subpaths of $P = (V, E)$ a *subpath decomposition* of P if it has the following two properties:

- For all $P_a = (V_a, E_a), P_b = (V_b, E_b) \in W$ with $a \neq b$, $E_a \cap E_b = \emptyset$: all subpaths in W are edgewise non-overlapping.
- $\bigcup_{P_i \in W} E_i = E$: all edges in P are included in at least on subpath $P_i \in W$.

A *chain* is a subpath consisting of a single direction. We denote a chain of n occurrences of direction A as A^n, e.g. $RRRR = R^4$.

Cycles. We call a subpath P_i of P a *cycle* when it consists of at least one edge, and its edges have equal numbers of opposing directional constraints: $\#U = \#D$ and $\#L = \#R$, e.g. $LLRR$ or $ULRD$.

We can classify two types of cycles:

Fourdirectional cycles: Cycles that contain all four edge directions ($\#L = \#R > 0$ and $\#U = \#D > 0$).

Bidirectional cycles: Cycles that contain only two edge directions. This type has two subtypes:

- **Vertical cycles:** Cycles that consist of only the directions U and D ($\#L = \#R = 0$ and $\#U = \#D > 0$).
- **Horizontal cycles:** Cycles that consist of only the directions L and R ($\#U = \#D = 0$ and $\#L = \#R > 0$).

A cycle has an ingoing path and an outgoing path. The *ingoing path* of a cycle of length n in P is the subpath P_i that includes all edges occurring before the cycle, and the succeeding $\lfloor n/2 \rfloor$ edges in the cycle. The *outgoing path* of a cycle of length n in P is the subpath P_o that includes all edges occurring after the cycle, and the preceding $\lfloor n/2 \rfloor$ edges in the cycle.

A subpath P_i *contains* a cycle c if c is a subpath of P_i: c both starts and ends with an edge in P_i.

Edge flips. When drawing an edge e of a path P, we can either draw it in the direction specified by its directional constraint, or we can draw it in a different direction. We *flip* an edge of a path P when we draw it in a direction different from its directional constraint. An edge flip can be specified in the form (e, d), where $e \in P$ specifies an edge to flip, and $d \in \{Up, Down, Left, Right\}$ specifies the direction to flip it in. Given a set S of edge flips, we can *apply* these to an input path P to obtain a path P'. The *cost* of obtaining P' from P is $|S|$. Note that if S contains multiple edge flips for a single edge e_i, the configuration of the path P' we obtain after applying S to P depends on the order we apply the edge flips in. Since flipping a single edge e_i multiple times is less efficient than flipping it a single time, in this thesis we will mostly consider sets where each edge e_i has no more than one edge flip.

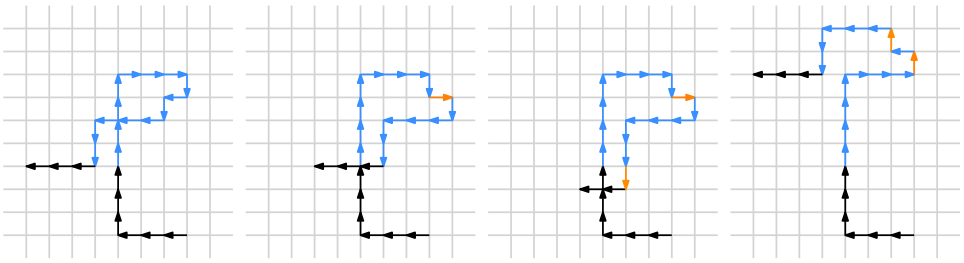
When denoting P', we denote the directional constraint of any flipped edges as the direction specified by their edge flip. For example, if we have a path $P = UDUL$, and we flip the third edge to D, we denote P' as $UDD'L$.

After applying an edge flip (e, d) to a path P, the resulting path P' has a pre-flip path and a post-flip path. The *pre-flip path* is the subpath starting

at the first vertex of P', and ending at $source(e)$. The *post-flip path* is the subpath starting at $target(e)$ and ending at the last vertex of P'. When applying multiple edge flips to a path P, the resulting path P' has a pre-flip and post-flip path for each flipped edge e.

Solving a path. We *solve* a path P when we find a set S of edge flips such that the path P' obtained after applying the edge flips in S to P does not contain any cycles. We can solve a subpath P_i of P in isolation or as part of P. Solving a subpath in isolation means removing P_i from P and finding a set S of edge flips within P_i such that after applying S to P_i, the resulting path P_i' contains no cycles. Solving a subpath as part of P means finding a set S of edge flips within the complete path P such that after applying all flips to P, no cycles start or end with an edge in P_i.

We can also solve a set A of edgewise non-overlapping subpaths of P in isolation or as part of P. Solving a set A of edgewise non-overlapping subpaths of P in isolation means finding a set S of edge flips within the subpaths in A such that after applying S to P, no cycle in P both starts and ends with an edge of a subpath $P_i \in A$. Solving a set A of edgewise non-overlapping subpaths of P as part of P means finding a set S of edge flips within P such that after applying all flips to P, no cycles start or end with an edge in belonging to any of the subpaths in A.



(a) A path P containing an unsolved subpath P_i.

(b) An isolated optimal solution of P_i that is not a non-isolated optimal solution for P_i.

(c) A non-isolated optimal solution of P_i that does not solve P.

(d) A non-isolated optimal solution of P_i that also solves P.

Figure 2.1: An example of the difference between an isolated optimal solution and a non-isolated optimal solution of a subpath P_i of P. The subpath P_i is indicated in blue, flipped edges are indicated in orange.

A subpath P_i of P has isolated optimal solutions and non-isolated optimal solutions. An *isolated optimal solution* is an optimal solution obtained when solving the subpath in isolation. As such, it consists only of flips of edges that are part of the subpath. A *non-isolated optimal solution* of a subpath is an optimal solution obtained when solving the subpath as part of

P. A visual example of the difference between an isolated optimal solution and a non-isolated optimal solution is given in Figure 2.1.

A subpath P_i of P can be solved *optimally within* P if it can be solved using one of its isolated optimal solutions.

An isolated optimal solution *blocks* all vertices it visits. If a vertex is blocked, no other subpath can visit it without causing a cycle. Two subpaths *share* a vertex if both subpaths have an isolated optimal solution that blocks that vertex.

We call an isolated optimal solution *blocked* if there is some path blocking any of the vertices required by this optimal solution. That is, an isolated optimal solution S of a subpath P_i of P is blocked if applying S causes a cycle within P. If all isolated optimal solutions of a subpath P_i of P are blocked, P_i cannot be solved optimally within P. Note, however, that any of the isolated optimal solutions of a subpath P_i of P being unblocked does not imply that P_i can be solved optimally within P.

Chapter 3

Problem Exploration

In this chapter we explore the problem, which includes proving some properties of paths and their optimal solutions in Section 3.1, identifying structures in paths in Section 3.2, and proving a lower bound and upper bound for the problem in Section 3.3.

3.1 Paths and subpaths

To reason about paths, it can be useful to divide the input path P into subpaths. This allows us to focus on smaller, less complicated structures instead of the entire, often complicated, input path.

We can make the following observation about solving cycles in subpaths:

Observation 1. *A flip of an edge outside a subpath P_i cannot solve cycles fully contained in P_i.*

After all, the directional constraints of the edges outside the subpath do not affect the ratios of directional constraints within the subpath, and thus flips of edges outside the subpath do not affect cycles contained in the subpath.

Using this observation, we can prove two lemmas about an input path P and its subpaths. We can formulate the first lemma as follows:

Lemma 1. *For any subpath P_i of P, the size of any of its isolated optimal solutions is smaller than or equal to the size of any of its non-isolated optimal solutions.*

Proof. We will prove this statement by contradiction. Consider an arbitrary path P of length n. Consider some subpath P_i of P with some non-isolated optimal solution R. Let S be an arbitrary isolated optimal solution of P_i. Assume that $|R| < |S|$. This means that R is not also an isolated optimal solution, since R and S would have the same size otherwise. Since $|R| < |S|$, the reason that R is not an isolated optimal solution must be that R contains flips of edges not in P_i.

Let $T \subseteq R$ be the set of flips of edges not part of P_i. Since R is a non-isolated optimal solution of P_i, after applying all edge flips in $R \backslash T$ at least one cycle in P remains that starts or ends with an edge in P_i. However, since T contains only flips for edges not in P_i, this remaining cycle cannot be fully contained in P_i. After all, as stated in Observation 1, a flip of an edge outside a subpath P_i cannot solve cycles fully contained in P_i. This means that all remaining cycles include at least one edge not in P_i, which in turn means that no cycles exist when isolating P_i. However, if flipping all edges in $R \setminus T$ eliminates all cycles that are fully contained in P_i, $R \backslash T$ is an isolated optimal solution for P_i. Thus, since all isolated optimal solutions must have equal size, $|R \backslash T| = |S|$. However, since $|T| \geq 1$, we have that $|R \backslash T| \leq |R| - 1 < |R|$. Thus, $|S| < |R|$. This, however, contradicts our earlier assumption that $|R| < |S|$, which means this must be false. Therefore, it must be the case that $|R| \geq |S|$. □

Observation 1 and Lemma 1 lead to the following observation:

Observation 2. *For any subpath P_i of P, the size of a non-isolated optimal solution is equal to the size of an isolated optimal solution if and only if the non-isolated optimal solution is also an isolated optimal solution.*

If the non-isolated optimal solution for a subpath P_i of P is also an isolated optimal solution for P_i, it is easy to see that the sizes of any non-isolated optimal solution and isolated optimal solution should be the same. On the other hand, if a non-isolated optimal solution R has a size equal to an isolated optimal solution, it must only flip edges part of P_i: otherwise, there is an isolated optimal solution S smaller than R. Thus, R is also an isolated optimal solution.

The second lemma can be formulated as follows:

Lemma 2. *Let A be a set of edgewise non-overlapping subpaths of P. Then the size of the isolated optimal solution of A is greater than or equal to the sum of the sizes of the isolated optimal solutions of the subpaths of A.*

Proof. Consider an arbitrary set A of j edgewise non-overlapping subpaths of P. Let $s_1, s_2, \ldots, s_j$ be the sizes of their corresponding isolated optimal solutions. Let $r_1, r_2, \ldots, r_j$ be the sizes of their corresponding non-isolated optimal solutions. By Lemma 1, we know that for each subpath P_i with $1 \leq i \leq j$, $s_i \leq r_i$. Thus, the size of the optimal isolated solution of A can only be smaller than $\sum_{i=1}^{j} s_i$ if the non-isolated optimal solutions overlap.

However, in every subpath $P_i \in A$, we have to flip at least s_i edges. After all, if there is some isolated optimal solution Q for A that flips $x < s_i$ edges in some subpath $P_i \in A$, we would have an isolated optimal solution for P_i with less than s_i edge flips: according to Observation 1, a flip of an edge outside a subpath P_i cannot solve cycles fully contained in P_i, so all cycles fully contained in P_i must be solved by the x edge flips that are part

of P_i, which must then form an isolated optimal solution. This, however, contradicts that s_i is the size of the isolated optimal solution of P_i.

Since the j subpaths are edgewise non-overlapping, the s_i edge flips needed for each subpath P_i cannot overlap. Therefore, the minimum number of edge flips required to solve A in isolation is the sum of the sizes of the optimal solutions of all its subpaths.

□

The most extreme case of Lemma 2 is when the set A of non-overlapping subpaths forms a subpath decomposition of P. We can make two observations about the isolated optimal solution and the non-isolated optimal solution of a subpath decomposition W of P, and their relation to the optimal solution of P.

Observation 3. *For any subpath decomposition W of P, the size of its isolated optimal solution is equal to the size of its non-isolated optimal solution.*

Proof. Let S be an isolated optimal solution of W. Since S is an isolated optimal solution of W, after applying all edge flips in S, no cycle starts in a subpath $P_i \in W$ and ends in a subpath $P_j \in W$ (P_i can be P_j). Since all edges of P are part of at least one subpath $P_i \in W$ by definition, this implies that P contains no cycles, which means that no cycle starts or ends with an edge belonging to any of the subpaths in W, which means that S is also a non-isolated solution.

Let R be a non-isolated optimal solution of W. Since R is a non-isolated optimal solution of W, after applying all edge flips in R, no cycle starts or ends with an edge part of a subpath $P_i \in W$. This implies that no cycle both starts and ends with an edge part of a subpath $P_i \in W$ either. Since all edges of P are part of at least one subpath in W, R contains only flips of edges part of a subpath of W, and thus R is also an isolated solution.

If each isolated optimal solution is also a non-isolated solution, and each non-isolated optimal solution is also an isolated solution, the isolated optimal solution and the non-isolated optimal solution must have the same size. □

Observation 4. *Let W be an arbitrary subpath decomposition of P. Then any isolated optimal solution for W is an optimal solution for P.*

Proof. Consider a path P with an arbitrary subpath decomposition W. Let S be an isolated optimal solution for W. Since W covers all edges in P, S is also a non-isolated solution of W. By Observation 3, S has the same size as a non-isolated optimal solution for W; thus S is also a non-isolated optimal solution for W. After applying S to P, no cycle starts or ends with an edge of a subpath $P_i \in W$. Since W covers all edges of P, this means that P contains no cycles after applying S. Thus, S is a solution for P.

Consider an optimal solution R for P. By definition, after applying all edge flips in R, P contains no cycles. This means that no cycle in P starts

or ends with an edge of a subpath $P_i \in W$, and thus W is solved. Therefore, R must also be a solution for W.

If any optimal solution for W is also a solution for P, and any optimal solution for P is also a solution for W, then any isolated optimal solution for W is an optimal solution for P. □

We can now formulate the following corollary of Lemma 2:

Corollary 2.1. *Let W be an arbitrary subpath decomposition of P. Then the size of the optimal solution of P is greater than or equal to the sum of the sizes of the isolated optimal solutions of the subpaths in W.*

Proof. By definition, a subpath decomposition W of P consist of edgewise non-overlapping subpaths of P. By Lemma 2, this means that the size of the isolated optimal solution of W is greater than or equal to the sum of the sizes of the isolated optimal solutions of the subpaths of W. As stated in Observation 4, any isolated optimal solution for W is an optimal solution for P. Thus, the size of the optimal solution of P is greater than or equal to the sum of the sizes of the isolated optimal solutions of the subpaths of W. □

Based on Lemma 2, we can make the following observation:

Observation 5. *Let A be a set of edgewise non-overlapping subpaths of P. Then the size of the isolated optimal solution of A is equal to the sum of the sizes of the isolated optimal solutions of the subpaths of A if and only if in the isolated optimal solution of A, each subpath is solved according to one of its isolated optimal solutions.*

Consider an arbitrary set A of j edgewise non-overlapping subpaths of P. Let $s_1, s_2, \ldots, s_j$ be the sizes of their corresponding isolated optimal solutions. Assume the size of the isolated optimal solution of A is equal to $\sum_{i=1}^{j} s_i$. From Observation 1 we know that an edge flip outside a subpath P_i cannot solve a cycle fully contained in P_i. Thus, to solve all cycles fully contained in a subpath $P_i \in A$, we need to flip at least s_i edges of P_i. Each isolated optimal solution of a subpath P_i specifies a combination of exactly s_i flips of edges of P_i that solves all cycles fully contained in P_i. Each non-isolated optimal solution of a subpath P_i needs to solve all cycles fully contained in P_i as well, which it can either do by flipping the same s_i edges as specified in an isolated optimal solution, or by flipping $x > s_i$ different edges. From Observation 2 and Lemma 1 follows that no non-isolated optimal solution R of size s_i or smaller exists, unless it is also an isolated optimal solution. Thus, the only way of flipping at most s_i edges in each path P_i is by solving each subpath P_i according to one of its isolated optimal solutions. On the other hand, if we obtained an isolated optimal solution for A by solving each of its subpaths according to one of its isolated

optimal solutions, it is easy to see that the size of the isolated optimal solution for A is equal to the sum of the sizes of the optimal solutions of its subpaths. After all, since the subpaths in A are non-overlapping, the isolated optimal solutions of the subpaths of A cannot overlap.

3.2 Constructions in paths

Within a path P, we can distinguish two main constructions: cycles and spirals. In this section, we discuss both in detail.

3.2.1 Cycles

Because our goal is to obtain a path that does not contain any cycles, we need to eliminate cycles. We call the process of flipping edges to eliminate cycles from P solving cycles. We call a cycle c *completely solved* with x edge flips if, after x edge flips, three conditions are met:

- The total number of cycles in P has decreased,
- All remaining cycles in P already existed before the x edge flips were applied, and
- c no longer exists.

Note that the last property holds after any single flip of an edge part of c, as any single flip of an edge part of c will impact $\#U, \#D, \#L$, or $\#R$ in c such that $\#U \neq \#D$ or $\#L \neq \#R$ and thus c is not a cycle anymore. Double edge flips, however, can keep c intact.

We distinguish two types of cycles: regular cycles, and loops.

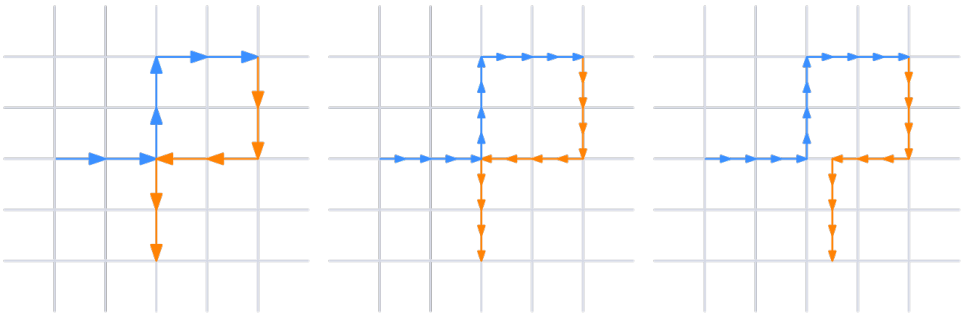
Regular cycles. *Regular cycles* are cycles of which the ingoing and outgoing paths do not cross at the start- and endpoint of the cycle: if we split all edges e_i of path P into two edges e_{i_1} and e_{i_2}, each with edge length 0.5 and direction identical to the direction of e_i, we can completely solve the cycle by removing at most one horizontal and one vertical edge half from the ingoing or outgoing path. We use this technique to simulate moving the ingoing path a distance less than one edge from the outgoing path, or vice versa. We need a distance less than one edge of movement to prevent forced collisions with structures surrounding the cycle, which are at least one edge away.

We cannot move the ingoing or outgoing path without flipping edges while keeping all properties intact; that is, simply moving vertices and adjusting edge lengths such that vertices are not necessarily placed on intersections on the grid and edges of the path do not necessarily follow edges of the grid, does not affect cycles, as cycles are defined as subpaths of at least one

edge with equal numbers of opposing directional constraints. Furthermore, when simply moving vertices and adjusting edge lengths, constructions that would intuitively be a cycle would not be classified as cycle, since they might start or end in the middle of an edge instead of at a vertex. This makes them almost undefinable. Therefore, we opted for the solution of splitting edges and removing edge halves to simulate moving the ingoing and outgoing path away from each other.

Another reason to split edges before testing if edge removal solves the cycle is to retain all features of the cycle: by allowing only half an edge to be removed, the remaining edge half can continue to represent certain features of the cycle, preventing us from solving a cycle by removing characteristic features of the cycle.

An example of completely solving a cycle by removing at most one horizontal and one vertical edge half from the ingoing or outgoing path is given in Figure 3.1.



(a) A path P containing a simple regular cycle.

(b) The path P with its edges split.

(c) P after removing a single horizontal edge.

Figure 3.1: An example of completely solving a cycle by removing at most one horizontal and one vertical edge half from the ingoing or outgoing path.

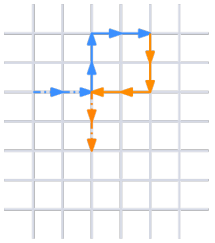
Let the location of the start- and end vertices of the original cycle when placing P on the grid be (a, b). We call a cycle c *nearly completely solved* if, after our x edge removals, the total number of cycles in P has decreased, c no longer exists, and each remaining cycle s in P meets one of the following requirements:

- s already existed before the x edge removals.
- s starts and ends at a vertex that is placed at least a distance 1 from (a, b), and s can be mapped to a distinct previously existing cycle t other than c such that the distance between the location of the start- and end vertex of s on the grid and the location of the start- and end vertex of t on the grid is at most 0.5.

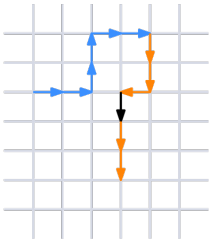
We call cycles that are nearly completely solved after the at most 2 edge removals, but not completely solved *near-regular cycles*. Near-regular cycles

are often fully contained in a loop or a spiral, which we will discuss in the next sections.

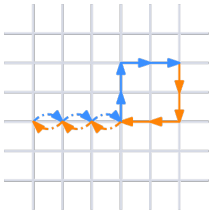
The fact that removing at most one horizontal and one vertical edge half from the ingoing or outgoing path is enough to solve a regular cycle makes a regular cycle the simplest type of cycle to solve, given that no other construction is present in the path: regular cycles can be completely solved with a single edge flip. Figure 3.2 shows a few examples of subpaths containing regular cycles, as well as ways to completely solve these cycles.



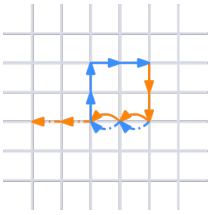
(a) The simplest regular cycle, with 6 optimal solutions.



(b) A regular cycle that fully contains a loop.



(c) A regular cycle where the ingoing and outgoing paths follow each other, causing multiple overlapping cycles.



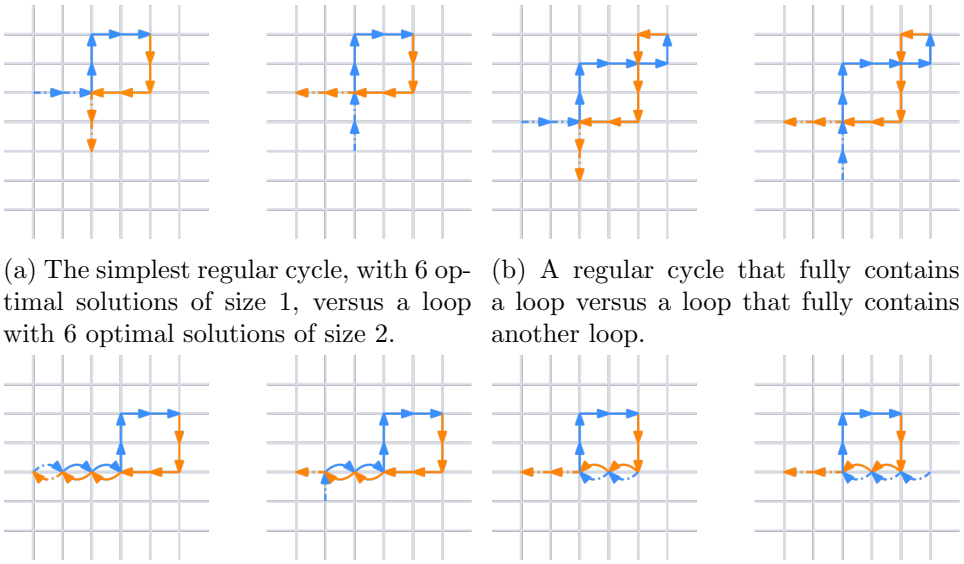
(d) A regular cycle which can only be solved optimally by flipping an edge part of the ingoing path.

Figure 3.2: Four regular cycles and their solutions. Ingoing paths are indicated in blue, outgoing paths in orange. In the unsolved cycles, the part of the ingoing and outgoing paths that is not part of the cycle is indicated with dashed arrows. Flipped edges are indicated in black.

Loops. *Loops* on the other hand are cycles of which the ingoing and outgoing paths do cross at the start- and endpoint of the cycle: if we split all edges e_i of path P into two edges e_{i_1} and e_{i_2}, each with edge length 0.5 and direction identical to the direction of e_i, it is impossible to completely or nearly completely solve the cycle by removing at most one horizontal and one vertical edge half from the ingoing or outgoing path.

As such, they present more of a challenge, and often need multiple edge flips to be completely solved. An example of the difference between regular cycles and loops is given in Figure 3.3.

As mentioned before, the lower bound of the optimal solution of regular cycles is 1. This lower bound is achievable because the ingoing and outgoing



(a) The simplest regular cycle, with 6 optimal solutions of size 1, versus a loop with 6 optimal solutions of size 2.

(b) A regular cycle that fully contains a loop versus a loop that fully contains another loop.

(c) Four overlapping regular cycles versus a loop that contains two near-regular cycles.

(d) Three overlapping regular cycles versus two overlapping loops and one near-regular cycle.

Figure 3.3: Several regular cycles (left) versus similar looking loops (right). The ingoing paths are indicated in blue, the outgoing paths in orange. In the loops, these paths cross each other, while in the regular cycles, they do not.

paths of a regular cycle do not cross: we can shift the paths away from each other without causing any issues. Loops, however, do not have this property: if we try to shift the ingoing and outgoing paths away from each other, we end up creating a smaller or a bigger loop, until we have shifted the paths far enough from each other to create a regular cycle, which can be solved with a single edge flip. To completely solve the loop, we need the entire outgoing path to be on a single side of the ingoing path, and vice versa.

When we are flipping edges, we are shifting the post-flip path away from the pre-flip path. We can shift the post-flip path at most a distance 2 away from the pre-flip path per edge flip. Thus, if we want to shift the post-flip path a distance x with respect to the pre-flip path, we need at least $\lceil x/2 \rceil$ edge flips.

We define a *suitable space* for a path P to be a set S of nodes and edges in the grid that are arranged in the same shape as P, and of which all nodes and edges are either completely free or visited by P. To solve a loop with the minimum number of edge flips, we need to find the cheapest way to shift the outgoing path to one side of the ingoing path. This translates to the shortest distance between the outgoing path and a suitable space around the

ingoing path. This distance often depends on the shape of the cycle, as well as the shapes of the ingoing and outgoing path and the distances between them.

Let the distance between the outgoing path and the closest suitable space on one side of the ingoing path be a. Then we can theoretically solve the loop in $\lceil a/2 \rceil$ flips if suitable edges are available. Thus, the lower bound for loops is $\lceil a/2 \rceil$ edge flips.

Let the *thickness* of a cycle be its width at its widest point, and the *tallness* of a cycle its height at its highest point. Let c be a loop with thickness x and tallness y. Then c often has a suitable space for its outgoing path at distances $(x + 1)$ and $(y + 1)$ from the outgoing path.

Two examples of optimal solutions of loops are given in Figure 3.4.

Determining if a cycle is a loop. The difference between regular cycles and loops is similar to the difference between weakly simple polygons and non-weakly simple polygons. A *weakly simple polygon* is a polygon that can be made simple by an arbitrarily small perturbation of the entire polygon. This corresponds to a cycle being regular if we can completely solve it by removing at most one horizontal and one vertical half edge: each vertex after a removed horizontal edge is displaced by 0.5 on the x-axis, and each vertex after a removed horizontal edge is displaced by 0.5 on the y-axis. We can generalize the definition of a regular cycle to hold if we split all edges e_i into n edges, with $n \geq 2$, making the displacement arbitrarily small.

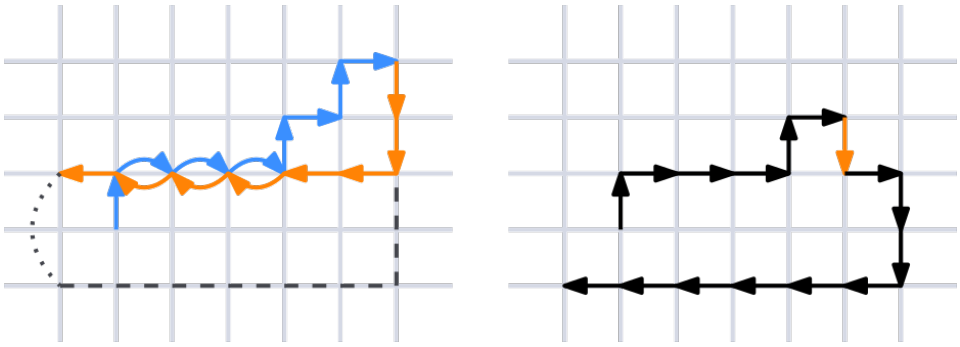
Thus, once we have detected a cycle, we can determine if it is a (near-) regular cycle or a loop by using an (adapted) algorithm for weakly simple polygon detection, like the one proposed by Chang et al. [4]. We can do this by mapping cycles to polygons such that if the polygon is weakly simple, it is a regular cycle, and if the polygon is not weakly simple, it is a loop.

The issue remains how to construct polygons from cycles such that loops are not weakly simple polygons, while regular cycles are. Although each cycle is a polygon, using the cycle on its own is not sufficient – cycles being a loop or not does not depend on the shape of the cycle itself, but on what happens at the first/last node of the cycle: does the ingoing path cross the outgoing path, or not?

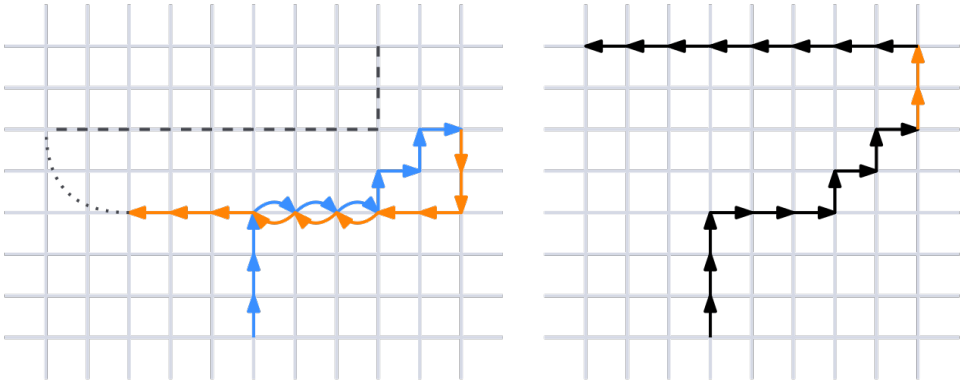
Thus, we need to include some edges of the ingoing path and outgoing path that occur outside the cycle in our polygon, such that we can determine if these paths cross. There are two questions at this point:

- How many edges do we need?
- How do we convert this subpath to a closed polygon?

For the first question, we know we need at least one edge on each side. We also know that we should be careful not to include multiple cycles and their ingoing/ outgoing paths. After all, we only need one loop in our polygon



(a) The closest suitable space of the outgoing path of this loop is located at distance 2 from the outgoing path, making the lower bound of this loop 1. This lower bound can be achieved if we can flip an edge from up to down, as this is the only edge flip that generates the required shift. Since we have a suitable edge for this flip, we have an optimal solution of size 1.



(b) The closest suitable space of the outgoing path of this loop is located at distance $\sqrt{8}$ from the outgoing path, making the lower bound of this loop 2. To shift the outgoing path to the closest free space, we need to make a shift of $(-2, 2)$, which can only be achieved by flipping a R to L and a D to U. Although we do not have suitable edges for both flips, we can still achieve an optimal solution of size 2 by shifting the outgoing path to a suitable space slightly further away.

Figure 3.4: Two examples of optimal solutions of loops. The loops are shown on the left side, with their ingoing path indicated in blue and their outgoing path indicated in orange. A suitable space for each outgoing path is indicated with dashed grey edges. On the right side, optimal solutions for the loop are shown, with flipped edges indicated in orange.

for the polygon to be classified as not weakly simple. However, if we have multiple cycles in our polygon, we cannot be sure which of these is the loop. On the other hand, if our polygon is weakly simple, we can be sure that all cycles in the polygon are regular cycles.

Since cycles can overlap, ensuring that only a single cycle is contained in our polygon is sometimes impossible; for example, when a smaller cycle is fully contained by a bigger cycle. In these cases, the smaller cycle needs to be evaluated first. If it is a regular cycle, we can proceed with the bigger cycle as usual. If it is a loop on the other hand, we need to transform it into a regular cycle. We do this as follows. Assume we have a path $P = DDLURRDDLLLUUURR$. P is a cycle and contains the loop $DLUR$ starting at the second edge. To convert this loop to a regular cycle, we simply need to follow it backwards: we reverse the edges to obtain $RULD$, then invert each edge to get its opposite edge: $LDRU$. This makes sure the cycle remains at exactly the same place on the grid, thus it does not introduce any new cycles or loops. At the same time, it is not a loop anymore.

Thus, we have a way to ensure that the cycle c_i we want to evaluate does not fully contain a loop. Now we need to make sure that we do not include any cycles that partly overlap with c_i. We can do this by including, apart from the edges in c_i, the edge immediately preceding c_i and the edge immediately succeeding c_i. By including only a single edge on each side of the cycle, we prevent the polygon from containing multiple partly overlapping cycles. After all, we need at least one edge of the parts of the outgoing and ingoing paths outside a cycle c to determine if c is a loop. Thus, if we include only a single edge on each side of c_i, it is impossible to also have included at least one edge of both the part of the ingoing path outside the cycle and the part of the outgoing path outside the cycle of a partly overlapping cycle c_j, and c_j will not be evaluated. Therefore, we take the edge immediately preceding c_i and the edge immediately succeeding c_i.

The drawback of doing this is that a certain class of loops will not be detected as loops, as they are only loops when considering a bigger portion of the ingoing and outgoing paths. All loops in this class are overlapped by other cycles. We have two tactics of handling this class of loops:

- Evaluate the entire subpath with overlapping cycles. If the subpath is a weakly simple polygon, all overlapping cycles are regular. If the subpath is not a weakly simple polygon, at least one of the cycles is not regular, but we cannot know which.

- If the ingoing and outgoing paths are following each other in the same direction, we can collapse them. If the collapsed path is weakly simple, the cycle might be regular. If the collapsed path is not weakly simple, we have two overlapping loops.

For the second question, we can simply add two edges at the endpoint

of the path: one that has a directional constraint opposite to the directional constraint of the last edge of the path, and one that has a directional constraint opposite to the directional constraint of the first edge of the path. This is shown in Figure 3.5.

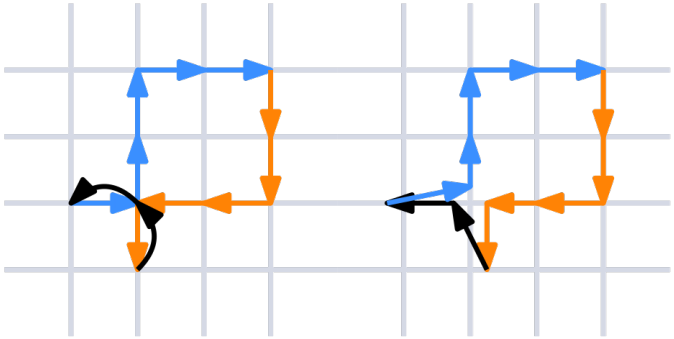


Figure 3.5: A regular cycle converted to a polygon by adding two edges (left) and the polygon made simple by a small perturbation of the polygon (right). The ingoing path is indicated in blue, the outgoing path in orange. Added edges are indicated in black.

3.2.2 Spirals

A second construction is the *spiral*. A spiral is any construction that occurs around one or more cycles and that prevents the cycle from being completely solved with the number of edges specified by its lower bound. Spirals can come in many shapes and sizes, which can make them tricky to detect. We call the process of solving a cycle inside a spiral 'solving a spiral'. An example of a spiral is given in Figure 3.6.

We can distinguish two classes of spirals: spirals that, in theory, have room to accommodate the cycle(s) inside the spiral, and spirals that do not.

Spirals that can accommodate. We call the number of vertices that can be available within the spiral after one edge flip the *maximum capacity* of a spiral. Spirals that have room to accommodate the cycle(s) inside the spiral have the property that their maximum capacity is greater than or equal to the length of the path inside the spiral. We can calculate the maximum capacity of a spiral by taking $max(width+1*height-1, width-1*height+1, width*height)$: in theory, the number of vertices inside an area of width x and height y is $(x-1)*(y-1)$, and a single edge flip can either increase the width by 2, increase the height by 2, or increase both by 1.

Spirals that have room to accommodate their cycles will mostly occur at the beginning and end of the input path, since we will have no path before or after the spiral there. These spirals can also occur when a spiral

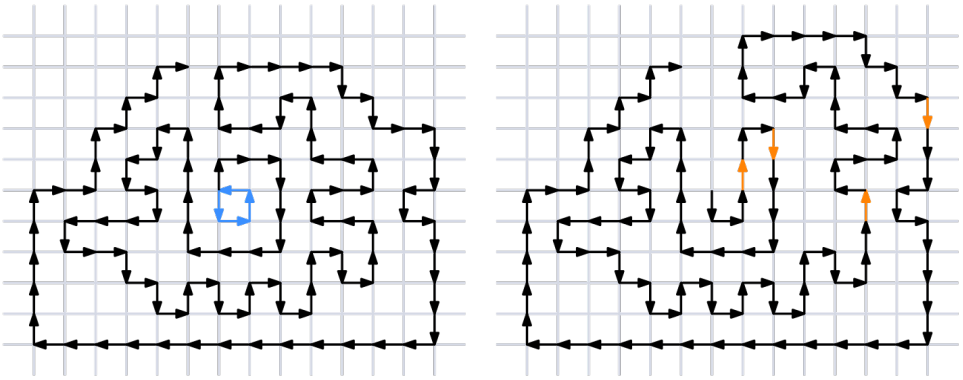


Figure 3.6: A path P containing a regular cycle within a spiral (left) and the optimal solution of P (right). The regular cycle is indicated in blue, flipped edges are indicated in orange.

is cleanly left, which means that the layers of the spiral consist partly of the path before the cycle inside the spiral, and partly by the path after the cycle inside the spiral, such that the layers of the cycle do not need to be crossed to leave the spiral. The spiral given in Figure 3.6 is an example of a spiral that has room to accommodate its cycle. The maximum capacity of this spiral is 120, while the path inside the spiral has length 64. Thus, the spiral can easily accommodate its cycle.

Because this class of spiral has room to accommodate the cycle(s) inside the spiral, they can theoretically be solved by making room in each layer to accommodate the cycle. This takes on average two edges per layer that compensate for each other: one edge makes room, and one edge shifts the rest of the spiral back to its place. However, since different layers may already have room, or may not have candidate edges, this number cannot be used as a hard upper- or lower bound but should merely be used as an estimation of the number of edge flips necessary to solve the spiral.

Spirals that need to be escaped. The second class consists of spirals that do not have room to accommodate the cycle(s) inside them, and thus they need 'escaping', as the number of vertices that need to be visited inside the spiral is significantly larger than the number of vertices available in the spiral. This spiral often occurs in the middle of a path, and each layer is often crossed. As such, this spiral consists of a number of overlapping loops. For this reason, a solving tactic for this spiral that is often optimal, is to simply solve all loops. An example of such a spiral is given in Figure 3.7. Note that, while this spiral has room to accommodate, it still needs to be escaped in order to be solved optimally. This is because of the loops that are also contained within the spiral.

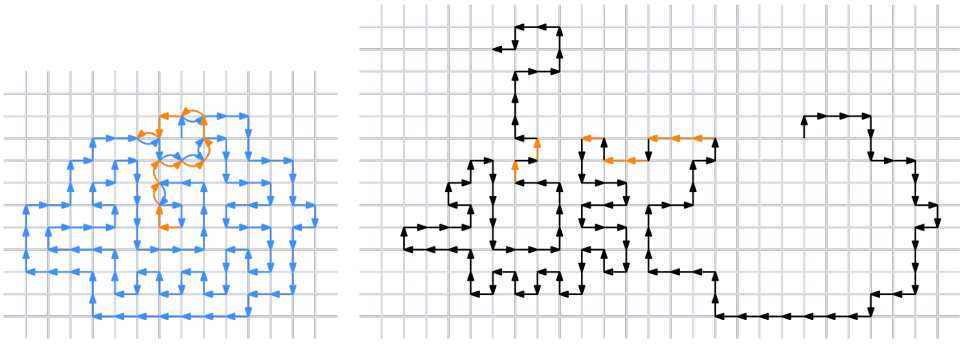


Figure 3.7: Left: a path P containing a near-regular cycle (and multiple other cycles) within a spiral. Edges of the ingoing path are indicated in blue; edges of the outgoing path are indicated in orange.
Right: the optimal solution of P. Flipped edges are indicated in orange.

A lower bound for solving this type of spiral is

$$n * \lceil min(heightcycle, widthcycle)/2 \rceil + n$$

where n is the number of layers around the cycle. This follows from the fact that we need to shift each layer out of its surrounding layer, which takes at least

$$\lceil (min(heightlayer, widthlayer) + 1)/2 \rceil$$

edge flips. Each layer i has height at least equal to the height of the cycle $+ i$, and width equal to the width of the cycle $+ i$. Thus, each layer i needs at least

$$\lceil (min(heightcycle, widthcycle) + 1 + i)/2 \rceil$$

edge flips, which is greater than or equal to

$$\lceil min(heightcycle, widthcycle)/2 \rceil + 1$$

edge flips, since having a spiral indicates having at least one layer. Thus, if we have n layers, we need at least

$$n*(\lceil min(heightcycle, widthcycle)/2 \rceil + 1) = n*\lceil min(heightcycle, widthcycle)/2 \rceil + n$$

edge flips.

3.3 Upper bound and lower bound

Given a path P, we can determine an upper bound and a lower bound on the number of directional constraints that must be violated. We will discuss the lower bound in Section 3.3.1 and the upper bound in Section 3.3.2.

3.3.1 Lower bound

An input path P is solved once we have found a set S of edge flips such that after applying these edge flips to P, the resulting path P contains no cycles. This indicates that the number of edge flips necessary to solve an input path P is tied to the number of cycles in P. Cycles are subpaths of P. Thus, according to Lemma 1, the isolated optimal solution of a cycle is a lower bound for the non-isolated optimal solution of a cycle. In isolation, any cycle needs at least one edge flip to be solved: if we flip no edges, the cycle remains a cycle. Regular cycles need exactly one edge flip in isolation. Thus, for each cycle in the input path P, we need to flip at least one edge belonging to that cycle. However, since cycles may share edges, it is possible to eliminate multiple cycles by flipping a single edge. Since this is only possible when cycles share an edge, it follows that two cycles that are non-overlapping – they do not share any edge – cannot be eliminated by flipping a single edge. Hence, if we find the largest set of non-overlapping cycles in P – the largest set C of cycles in which no cycle in C overlaps with any other cycle in C – we have found a lower bound of the number of edges to flip: this lower bound is $|C|$. We detail an algorithm to find the largest set of non-overlapping cycles in an input path P in Section 4.1.

We can use Lemma 2 to get a more specific lower bound: after finding a largest set of non-overlapping cycles C in P, we find a subpath decomposition W of P such that each subpath $P_i \in W$ contains exactly one cycle $c \in C$, and each cycle $c \in C$ is contained by exactly one subpath $P_i \in W$. Such a subpath decomposition exists because the cycles in C are non-overlapping, thus to each contain exactly one cycle $c \in C$, the subpaths in W do not need to overlap. Each cycle in c can be contained by exactly one subpath P_i, with the smallest subpath to contain a cycle being the cycle itself. We can include all edges of P in the subpaths by simply growing each subpath until it meets another subpath.

By Lemma 2, we know that a lower bound for the optimal solution of P is equal to the sum of the sizes of the isolated optimal solutions of the subpaths in W. To find an accurate lower bound for the isolated optimal solution of each of the subpaths in W, we can check each subpath for loops. If a subpath P_i does not contain a loop, its lower bound $LB(P_i)$ is 1. After all, P_i contains exactly one cycle c in C, so all cycles in P_i overlap. Since all cycles are regular, all cycles can be fixed with a single edge flip. If the subpath P_i does contain a loop, we can calculate the lower bound for this loop when

solving P_i in isolation with the formula from Section 3.2.1. The lower bound of P_i, $LB(P_i)$, is then equal to the lower bound of the loop: to solve P_i, we need to solve the loop. The lower bound of P is $\sum_{P_i \in W} LB(P_i)$. Note that this value is not necessarily equal for each choice of subpath decomposition, as some subpath decompositions might allow loops to remain undetected.

3.3.2 Upper bound

To find an upper bound for the number of edges we need to flip, we need to find a set of edge flips that is guaranteed to fix any possible existing cycle and does not introduce any new cycles. For a subpath to be a cycle, it needs to contain at least two opposite directions: both L and R are included in the subpath, or both U and D are included in the subpath.

Thus, a simple way to eliminate all cycles from a path is to eliminate two orthogonal directions entirely, which can be done in four ways:

- Flip all occurrences of D to U, and all occurrences of L to R.
- Flip all occurrences of D to U, and all occurrences of R to L.
- Flip all occurrences of U to D, and all occurrences of L to R.
- Flip all occurrences of D to U, and all occurrences of R to L.

The adjusted path now consists of the remaining two orthogonal directions, and therefore it contains no cycles. Thus, a first upper bound is $min(\#U + \#R, \#U + \#L, \#D + \#R, \#D + \#L)$.

However, it is easy to see that this is not the smartest way to eliminate all cycles. After all, all fourdirectional cycles are already eliminated after flipping one entire direction to its opposite direction. This may even eliminate some bidirectional cycles, namely the cycles that consisted of the flipped direction and its opposite direction. That means that the only cycles that still exist, are cycles consisting of the directions orthogonal to the flipped direction – if we removed the direction R by flipping all occurrences of R to L, the only cycles that still exist are bidirectional cycles consisting of U and D. Note that flipping an entire direction A to its opposite direction $\bar{A}$ does not introduce any new cycles. After all, it does not affect the ratio of the directions orthogonal to A in any subpaths of P, as it does not remove any occurrences of directions orthogonal to A, nor does it introduce these.

Hence, we are left with bidirectional cycles of a single subtype: only horizontal cycles or only vertical cycles. Any set of overlapping bidirectional cycles of a single type can be fairly easily solved by separating the two directions by flipping a single shared edge to an orthogonal direction. To ensure that we do not introduce any new cycles when eliminating the bidirectional cycles, we choose the orthogonal direction that we did not eliminate. For example, if we have a set of overlapping bidirectional cycles $UUUDDD$, and

we just eliminated the direction R from the path, we can solve this set by flipping the rightmost U or the leftmost D to L: $UULDDD$ or $UUULDD$.

Thus, after eliminating an entire direction, we only need k more edge flips to solve the entire path, where k is the size of the largest set of non-overlapping cycles in the path P' obtained after flipping an entire direction A in P.

Using this knowledge, we can formulate an upper bound of $E+k$, where E is the number of occurrences of the direction we want to eliminate in the original path P, and k is the size of the largest set of non-overlapping cycles after flipping this direction.

The value of $E+k$ largely depends on the direction we decide to eliminate, since this decision affects both the value of E and the value of k. After all, if the direction A we eliminate is a horizontal direction (R or L), the largest set of non-overlapping cycles in the path P' obtained after flipping all occurrences of A in P consists of only vertical cycles. If A is a vertical direction (U or D), this largest set of non-overlapping cycles consists of only horizontal cycles.

Assume we have an input path P where the largest set of non-overlapping horizontal cycles has size h and the largest set of non-overlapping vertical cycles has size v. If we eliminate a horizontal direction, $k = v$. If we eliminate a vertical direction, $k = h$. Since horizontal and vertical cycles do not overlap by definition, we have to flip at least $h + v$ edges that are part of cycles in these sets, regardless of which edges we flip that are not part of cycles in these sets.

Let A be the horizontal direction that occurs least, and B the vertical direction that occurs least. Then our upper bound becomes $min(\#A + v, \#B + h)$ edge flips. To try to express this upper bound in terms of path length n, let us first make a few observations about $\#A$ and $\#B$.

Observation 6. *Let A be the least occurring horizontal direction and B the least occurring vertical direction. Then $\#A + \#B \leq n/2$.*

If A is the least occurring horizontal direction, then, $\#\bar{A} \geq \#A$. Similarly, $\#\bar{B} \geq \#B$. Thus, $\#\bar{A}+\#\bar{B} \geq \#A+\#B$. Since $\#\bar{A}+\#\bar{B}+\#A+\#B = n$, $\#A + \#B$ can be at most $n/2$.

Observation 7. *Let A be the least occurring horizontal direction and B the least occurring vertical direction. If $\#A = n/x$, then $\#B \leq 0.5(n - 2n/x)$. Vice versa, if $\#B = n/x$, then $\#A \leq 0.5(n - 2n/x)$.*

This follows directly from Observation 6. Since $\#A + \#B \leq n/2$, if $\#A = n/x$, then $\#B \leq n/2 - n/x$, which can be rewritten as $0.5(n - 2n/x)$.

Now let us try to express the upper bound of $min(\#A+v, \#B+h)$ edge flips in terms of n. We know that $\#A = h + a$ for some $a \geq 0$, since each cycle in our largest set of non-overlapping horizontal cycles contains exactly

one edge going in direction A. Similarly, $\#B = v + b$ for some $b \geq 0$. Thus our upper bound can be rewritten as $min(a + h + v, b + h + v)$ edges. By Observation 6, $\#A + \#B = a + h + b + v \leq n/2$, which means our upper bound is less than or equal to $n/2$.

Since our upper bound can be written as $min(a+h+v, b+h+v)$ edges, we know that the exact size of the upper bound depends on which direction has the smallest number of edges that are not part of our largest sets of horizontal or vertical non-overlapping cycles. If a or b are zero, we can solve the path optimally: after flipping h or v edges respectively, we have eliminated all occurrences of A or B, after which we can solve all remaining bidirectional cycles with v or h edge flips.

Assume $a + h + v \leq b + h + v$. Then $a \leq b$. Assume $\#A > \#B$. Since $\#A = a + h$, $\#B = b + v$, and $a \leq b$, we know that $h > v$. We also know that $\#B < n/4$, as by Observation 6 $\#A + \#B \leq n/2$, thus $\#A$ can only be bigger than $\#B$ if $\#B < n/4$. Thus, $b + v < n/4$. This means that $b+v+h < n/4+h$. Since $a \leq b$, $a+h+v \leq b+h+v < n/4+h$. Thus, our upper bound is smaller than $n/4 + h$ in this case. Now assume $\#B \geq \#A$. By Observation 6, this means that $\#A \leq n/4$. Thus, $a + h + v \leq n/4 + v$, meaning our upper bound is smaller than or equal to $n/4 + v$ in this case.

Assume $a + h + v \geq b + h + v$. Then $a \geq b$. Assume $\#B > \#A$. Since $\#A = a + h$, $\#B = b + v$, and $a \geq b$, we know that $v > h$. We also know that $\#A < n/4$, as by Observation 6 $\#A + \#B \leq n/2$, thus $\#B$ can only be bigger than $\#A$ if $\#A < n/4$. Thus, $a + h < n/4$. This means that $a+h+v < n/4+v$. Since $a \geq b$, $b+h+v \leq a+h+v < n/4+v$. Thus, our upper bound is smaller than $n/4 + v$ in this case. Now assume $\#A \geq \#B$. By Observation 6, this means that $\#B \leq n/4$. Thus, $b + h + v \leq n/4 + h$, meaning our upper bound is smaller than or equal to $n/4 + h$ in this case.

Since our upper bound is always smaller than the larger of $n/4 + h$ and $n/4 + v$, our upper bound in terms of n is $min(max(n/4 + h, n/4 + v), n/2)$ edges.

Considering that the we get the same result when eliminating the horizontal (vertical) direction that occurs least in P as when eliminating the horizontal (vertical) direction that occurs most in P, a smart choice would be to eliminate either the horizontal direction that occurs the least in the input path P or the vertical direction that occurs the least in the input path P. The best choice between the least-occurring horizontal and the least-occurring vertical direction depends on which direction has the most edges that are not part of any cycles in the largest set of horizontal non-overlapping cycles or the largest set of vertical non-overlapping cycles. After all, this would require us to flip the smallest number of edges.

If one direction does not occur in the path at all, we only have to flip h or v edges, as one direction is already eliminated entirely.

Buffering a direction. We can reduce the bound even further: to ensure that the path P contains no fourdirectional cycles, eliminating an entire direction is unnecessary. The only requirement is that we cannot *access* any of the edges going in the direction A we want to eliminate: any occurrence of A should be surrounded by edges that eliminate the possibility of A being part of a cycle. We can do this by creating a 'buffer' around edges going in direction A, such that even if these edges are included in a subpath P_i, P_i cannot be a cycle.

We can build a buffered path as follows. Assume we have a chain s of $|s|$ edges going in direction A. Let us put chains of $|s|+1$ edges going in direction $\bar{A}$ on either side of s to create a path T:

$$\bar{A}^{|s|+1}A^{|s|}\bar{A}^{|s|+1}$$

Now A is buffered: assume we have subpaths consisting of all directions except A on both ends of T. Whenever we try to find a subpath with all four directions, we need to include at least $|s|+1$ edges in direction $\bar{A}$. However, the maximum number of edges going in direction A we can include is $|s|$, thus we will always have more edges going in direction $\bar{A}$ than in direction A in our subpath. Therefore, we cannot have any fourdirectional cycles.

We do however have a few bidirectional cycles, for example the one that starts at the first edge of s and ends at the $|s|^{th}$ edge of the buffer. To ensure that such cycles do not exist either, we place an edge of one of the directions orthogonal to A between s and the buffers on both sides. It is crucial to ensure that both these edges go in the same direction; otherwise, we would have a cycle. Now we have, from left to right, a chain r of $|s|+1$ edges going in direction $\bar{A}$, a single edge going in direction B, a chain s of $|s|$ edges going in direction A, a single edge going in direction B, and a chain t of $|s|+1$ edges going in direction $\bar{A}$:

$$\bar{A}^{|s|+1}BA^{|s|}B\bar{A}^{|s|+1}$$

For any cycle to occur, this subpath has to be preceded by an edge e_i going in direction A – directly or with a number of edges going in directions B and $\bar{B}$ inbetween, with $\#B = \#\bar{B} - 1$. However, if we buffer each occurrence of A, we have included $x > 0$ edges going in direction $\bar{A}$ in our subpath after reaching e_i, and thus we cannot have a cycle.

Of course, we cannot simply add edges to our path P, and thus we have to flip edges to create our buffer. As we are trying to make any occurrence of A unreachable, and the direction A occurs the least, it makes most sense to flip edges of our chains of edges going in direction A. However, when we flip edges going in direction A, our chain gets shorter, and the number of edge flips required to buffer this shorter chain is lower than the number of edge flips required to buffer the initial chain. We can calculate the number

of edge flips necessary as follows. Assume we have a chain $s = A^a$. We need to flip x edges to transform this chain into a buffered chain. After applying these x edge flips, we have $a-x$ edges going in direction A remaining, which means that we need to flip at least $a - x + 1$ edges to direction $\bar{A}$ on both sides. We also need 2 edge flips to insert our edges going in direction B. This means we need $2(a - x + 1) + 2 = 2a - 2x + 4$ edge flips. Rewriting this, we get $x = (2a + 4)/3$. Since this is a lower bound on the number of edge flips we need and flipping edges partially is impossible, we round it up to $\lceil (2a + 4)/3 \rceil$. Figure 3.8 shows an example of how to flip edges such that the chain R^7 is buffered.

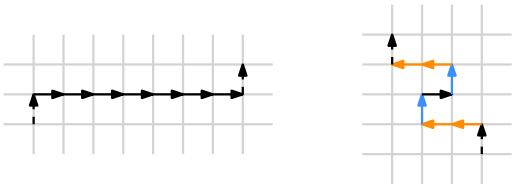
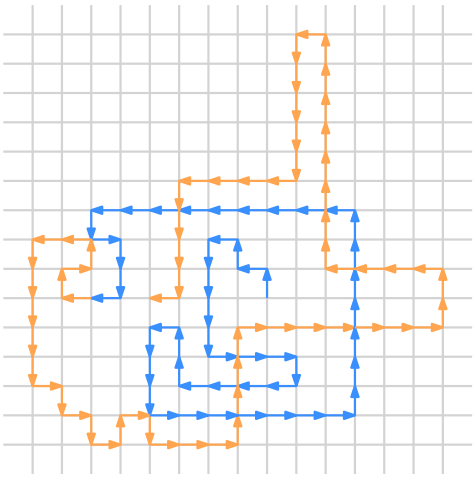


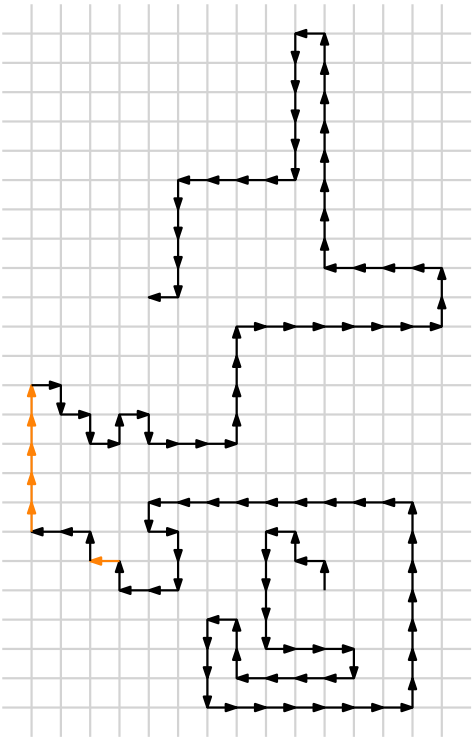
Figure 3.8: An example of how to flip edges such that a chain R^7 is buffered. Edges flipped from R to L are indicated in orange, edges flipped from R to U are indicated in blue.

Following this technique, for every chain s_i of edges going in direction A in path P, we have to flip $\lceil (2|s_i| + 4)/3 \rceil$ edges, and can leave $\lfloor (|s_i| - 4)/3 \rfloor$ edges unflipped. Note that this means that we still have to flip all edges for any chain s_i of length less than 7. Figure 3.9 shows a visual example of what buffering looks like in a complete path. Figure 3.9a shows a path P containing various cycles but no bidirectional cycles. We select the least occurring direction as the direction to buffer. This is direction R, with 26 occurrences. U and D both have 28 occurrences, and L has 30 occurrences. Using the buffering technique, we have to flip 24 edges before we are sure that the resulting path P' contains no cycles. Note that when using this buffering technique, buffering the direction that occurs least does not necessarily lead to the smallest number of edge flips.

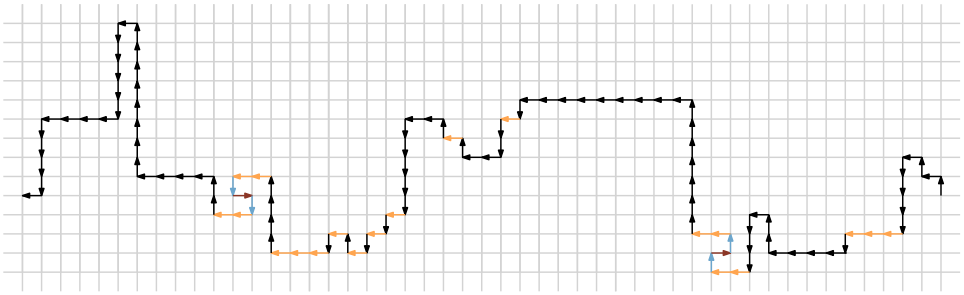
To ensure that as many edges as possible can remain unflipped, we can use a trick to find subpaths with as many occurrences of our direction A chosen to eliminate as possible: instead of considering only chains, thus stopping the subpath whenever we encounter an edge going in a direction other than A, we can also allow the subpath to contain edges going in direction B or $\bar{B}$ (but not both). After all, it does not matter how many edges with direction B our subpath contains, as long as from any edge with direction A, we need to pass $\#A+1$ edges with direction $\bar{A}$ to reach an edge with direction $\bar{B}$. Let $\#A_i$ be the number of edges with direction A in a suitable subpath P_i. Then the number of edge flips required to transform P_i into a buffered subpath is $\lceil (2\#A_i + 4)/3 \rceil$.



(a) A path P containing various cycles. The path is coloured in blue and orange for easier parsing.



(b) An optimal solution of P. Flipped edges are indicated in orange.



(c) P with all occurrences of R buffered. Flipped edges are indicated in blue and orange, with edges flipped to U or D indicated in blue, and edges flipped to L indicated in orange. Edges in direction R that remain unflipped are indicated in dark red.

Figure 3.9: An example of buffering all occurrences of R in a path P.

If P_i starts at the beginning of a path or ends at the end of a path, the number of edge flips required is smaller: any subpath that starts at the beginning of a path or ends at the end of a path only needs a buffer on one side, as the other side does not neighbour any other subpath. This decreases the number of edge flips needed for that subpath to $\lceil \#A_i/2 \rceil + 1$.

We can also consider subpaths that already have direction $\bar{A}$, as long

as these subpaths do not contain cycles in the input path – so we never have an edge $\bar{A}$ neighbouring an edge A, or vice versa. Thus, our subpath can be a subpath that contains at most three directions and no cycles. Since for any set of overlapping bidirectional cycles we have to flip one edge regardless of any other edge flips we apply, we can choose to first solve all sets of overlapping bidirectional cycles consisting of directions A and $\bar{A}$ by flipping one occurrence of A to $\bar{A}$ or B for each set, depending on the second neighbour of A - if it is B, we flip A to $\bar{A}$, and if it is another A, we flip A to B. Depending on the location of the edges going in direction $\bar{A}$, we can include these edges in our buffer, decreasing the number of edge flips necessary. Consider for example the following subpath:

$$\bar{B}\bar{A}\bar{A}BABABAABA\bar{B}\bar{B}$$

Assume A is the direction we want to buffer. The longest subpath containing the most occurrences of A is

$$\bar{A}\bar{A}BABABAABA\bar{A}$$

This subpath contains less than 7 occurrences of A, so according to our earlier formula, we would have to flip all edges with direction A. However, we already have partial buffers on both sides: on the left side we have two occurrences of $\bar{A}$, buffering one occurrence of A, and on the right side we have one occurrence of $\bar{A}$. We have 4 occurrences of A in total; we need to flip x more edges on the left side and y more edges on the right side to buffer the occurrences of A completely, leaving us with $4-x-y$ occurrences of A. The buffer on each side needs to consist of at least $4-x-y+1$ edges with direction $\bar{A}$. The left side already has 2 edges, so $x = 4 - x - y + 1 - 2 = 1.5 - 0.5y$. The right side already has one edge, so $y = 4 - x - y + 1 - 1 = 2 - 0.5x$. Filling in the value of x in this formula, we get $y = 5/3$. Filling in this value in the formula for x, we get $x = 2/3$. Since flipping edges partially is impossible, this means we have to flip at least one more edge on the left side and 2 more edges on the right side. This is possible without causing any cycles due to the placements of the edges going in direction B, allowing us to leave 1 edge unflipped. We get the following buffered subpath:

$$\bar{A}\bar{A}B\bar{A}'BABA\bar{A}'\bar{A}'B\bar{A}$$

We can apply this tactic in general: if we are buffering occurrences of A, and we have a subpath S_i starting or ending in $\bar{A}$, we can use these edges in our buffer. Let the sum of the sizes of the partial buffers on the left and right of the subpath be t. Then we need at most $max(\lceil(2(\#A_i + t) + 4)/3\rceil - t, 2)$ edge flips to completely buffer A in S_i.

Edges in direction $\bar{A}$ that occur somewhere else in the subpath can also reduce the number of edge flip required to completely buffer A in S_i, as long

as they are in the buffer zone: the further they are to the middle of the subpath, the smaller the chance that they can be included in the buffer. Let $\#\bar{A}_i$ be the number of edges in direction $\bar{A}$ in subpath S_i. Then the buffer zones are either the $\lceil(\lceil(2(\#A_i + \#\bar{A}_i) + 4)/3\rceil)/2\rceil$ edges on the left and the $\lfloor(\lceil(2(\#A_i + \#\bar{A}_i) + 4)/3\rceil)/2\rfloor$ edges on the right, or the $\lfloor(\lceil(2(\#A_i + \#\bar{A}_i) + 4)/3\rceil)/2\rfloor$ edges on the left and the $\lceil(\lceil(2(\#A_i + \#\bar{A}_i) + 4)/3\rceil)/2\rceil$ edges on the right. If all occurrences of $\bar{A}$ are in a buffer zone, the number of edge flips required to buffer A in S_i is at most $max(\lceil(2(\#A_i + \#\bar{A}_i) + 4)/3\rceil - \#\bar{A}_i, 2)$.

An example of how to buffer occurrences of R in the path P shown in Figure 3.9a with longest subpaths instead of longest chains is shown in Figure 3.10.

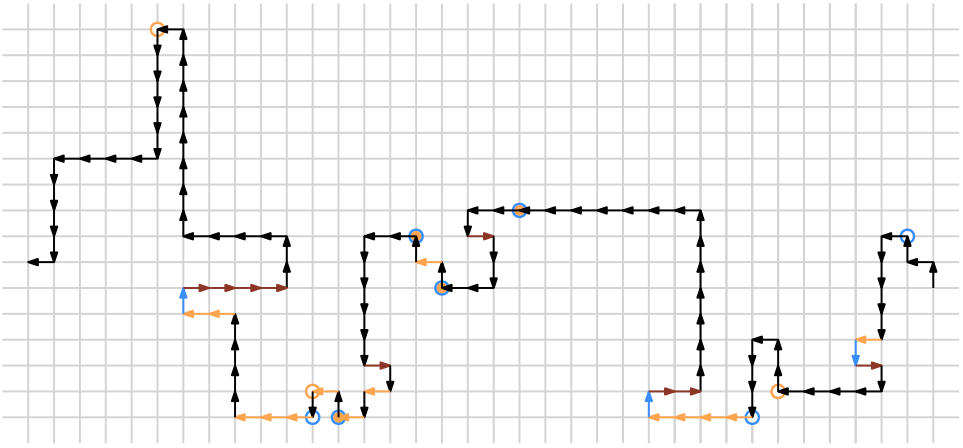


Figure 3.10: An example of how to buffer occurrences of R in the path P shown in Figure 3.9a with longest subpaths instead of longest chains. Blue circles show the start vertex of a considered subpath, orange circles show the end vertex. Edges that are flipped to L are indicated in orange, edges that are flipped to U or D are indicated in blue. Edges in direction R that remain unflipped are indicated in dark red.

As is visible in Figure 3.10, instead of having to flip 24 edges before we are sure that the resulting path P' contains no cycles, we now only have to flip 17 edges. This difference is due to partial buffers already existing at some places and being able to include multiple chains in a single subpath at other places.

Theoretically, we can also include edges going in direction $\bar{B}$ in our subpath, as long as from any chain A^s, before reaching a chain $\bar{B}^t$, we pass at least $t + 1$ edges going in direction B. This ensures no cycles exist in the subpath, nor can they be caused by flipping edges going in direction A to $\bar{A}$ or B. An example of how to buffer occurrences of R in the path P shown in Figure 3.9a with longest subpaths including $\bar{B}$ instead of longest subpaths excluding $\bar{B}$ is shown in Figure 3.11.

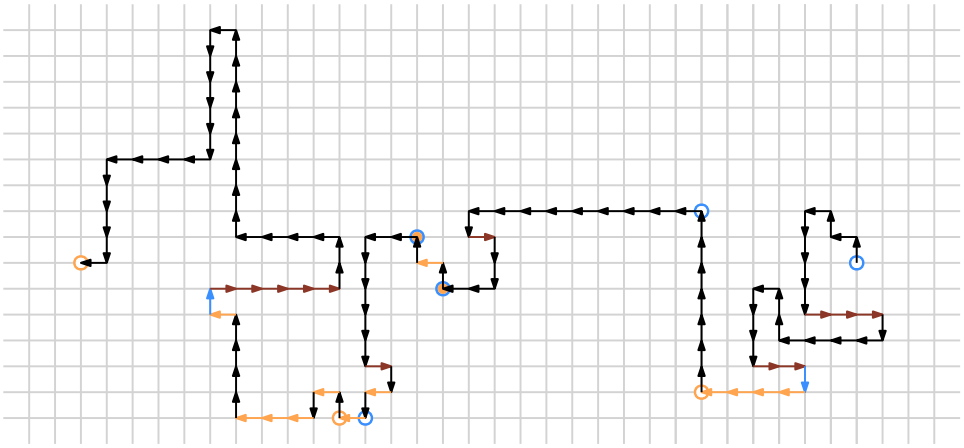


Figure 3.11: An example of how to buffer occurrences of R in the path P shown in Figure 3.9a with longest subpaths including $\bar{B}$ instead of longest subpaths excluding $\bar{B}$. Blue circles show the start vertex of a considered subpath, orange circles show the end vertex. Edges that are flipped to L are indicated in orange, edges that are flipped to U or D are indicated in blue. Edges in direction R that remain unflipped are indicated in dark red.

As is visible in Figure 3.11, we now only flip 14 edges. All extra edges that can remain unflipped are located in the first and last subpaths as those paths are able to benefit most from the few extra edges included in the subpaths. Note that the last and second to last subpaths overlap with exactly one edge. Subpaths are allowed to overlap with exactly one edge, as long as in the final subpath we have at least $s + t + 1$ occurrences between the s occurrences of A in the first subpath and the t occurrences of A in the second subpath.

Chapter 4

Exact algorithm

In this chapter, we will describe an exact algorithm for solving the problem and discuss the running time of this algorithm. We will start by discussing two subroutines necessary to compose the algorithm: finding a largest set of non-overlapping cycles, and finding a solution of a pre-defined size.

4.1 Finding a largest set of non-overlapping cycles

A key requirement of our exact algorithm is knowledge of the lower bound of the optimal solution to any input path P. As we have seen in Section 3.3.1, a lower bound for any input path P is the size of the largest set of non-overlapping cycles in P. To find such a set, we use the algorithm FindMaxCycleSet(P). FindMaxCycleSet(P) takes as input a path P, and outputs the start and end indices of all cycles in the largest set of non-overlapping cycles in P.

This method adds the cycle that ends first to the independent set, then adds the cycle that starts after this cycle and ends first, etc. This will result in a largest independent set of non-overlapping cycles. A more precise description is given in Algorithm 1.

Proof of correctness. Let C be the set of cycles returned by the algorithm and let S be an arbitrary largest set of non-overlapping cycles of P. Assume $S \neq C$. This means there is a set $S' \subseteq S$ of cycles that are in S but not in C, and a set $C' \subseteq C$ of cycles that are in C but not in S. Let $s \in S'$ be the cycle that, of all cycles in S', occurs first when traversing P. This cycle starts with edge $e_i \in P$ and ends with edge $e_j \in P$, with $j > i$. Since the algorithm did not add s it to C, we know that s was never fully contained in our search space: at all times, e_i occurred before the endpoint of some cycle c already added to C, or we were handling an edge e_k with $k < j$. At the moment of handling e_j, the latter was false, so the former must have been true: e_i occurred before the endpoint of some cycle c already added to C. This means s overlaps with c, and $c \in C'$. However, s is not fully contained

Algorithm 1: FindMaxCycleSet(P)

Input: A path P.

Output: Start- and end indices of each cycle in the largest set of non-overlapping cycles of P.

```
Coordinates ← Empty associative array;
Add ((0, 0), 0) to Coordinates;
CycleSet ← ∅;
EndPreviousCycle ← 0;
CurrentCoordinate ← (0, 0);
for Edge e_i in P do
    if Direction(e_i) = DOWN then
        NextCoordinate ← CurrentCoordinate + (0, −1);
    else if Direction(e_i) = UP then
        NextCoordinate ← CurrentCoordinate + (0, 1);
    else if Direction(e_i) = RIGHT then
        NextCoordinate ← CurrentCoordinate + (1, 0);
    else if Direction(e_i) = LEFT then
        NextCoordinate ← CurrentCoordinate + (−1, 0);
    if No index is associated to key NextCoordinate then
        Add (NextCoordinate, i + 1) to Coordinates;
    else if An index j is associated to key NextCoordinate then
        Add j, i + 1 to CycleSet;
        EndPreviousCycle ← i + 1;
        Clear Coordinates;
        Add (NextCoordinate, i + 1) to Coordinates;
    CurrentCoordinate ← NextCoordinate;
return CycleSet
```

by c: after all, e_j was handled after c was already found and added to C, which means s must end after c. Thus, either c is fully contained by s, or c starts before s and ends within s. Since s is the first occurring cycle in S', all cycles in S that occur before s are also part of C. This means that c does not overlap with any cycle in S other than s, so if we swap s with c, S is still a set of non-overlapping cycles in P with a size equal to before the swap. We can continue replacing each first occurring cycle in S' with cycles in C' until S' is empty, and $S = C$. Since each replacement kept the size and validity of S intact, and S was originally a largest set of overlapping cycles, C is a largest set of non-overlapping cycles.

Running time analysis. *Coordinates* is an associative array which stores indices of vertices, with as key their coordinate. If this associative array is

implemented using a hash table with a suitable hash function, the average time complexity for the operations we need (initialize, insert, clear, get) is O(1). However, if the hash function used results in a lot of collisions, the time complexity of these operations becomes O(n). Assuming the associative array is implemented such that a time complexity of O(1) can be achieved, the algorithm has the following running time: Line 1-5 all take O(1) time. The loop on lines 6-22 is executed n times where n is the length of the input path P. Lines 7-22 all take O(1) time. Thus, the running time of FindMaxCycleSet(P) is $O(n)$.

Alternatively, instead of an associate array, we can implement *Coordinates* as a height-balanced binary search tree. In a height-balanced binary search tree, the necessary operations (initialize, insert, clear, search) take $O(\log n)$ time, resulting in a running time of $O(n \log n)$ for FindMaxCycleSet(P).

4.2 Finding a solution of a specific size

Since the size of the largest set S of non-overlapping cycles of a path P is a lower bound for the optimal solution of P, we know that the size of S never exceeds the size of the optimal solution of a path P. We can utilize this knowledge to determine if an edge flip is *bad* – it will not lead us to an optimal solution - or *good* – it might lead us to an optimal solution. Assume we know that k is the size of the optimal solution for a given path P. Now if we flip any edge of P in an arbitrary direction to obtain a path P', and calculate the size s of the largest set of non-overlapping cycles in P', we can determine if the flip was a good or bad flip as follows:

- If $s > k - 1$, the flip was bad.
- If $s \leq k - 1$, the flip was good.

After all, if the largest set of independent cycles in P' has size s, we know that we need to flip at least s edges to solve P'. However, if $s > k-1$, then $s + 1 > k$, which means that, even if we are indeed able to solve P' with s edge flips, if we keep this edge flip the total cost to solve P will be more than the size of the optimal solution. Thus, the flip was bad. If $s \leq k-1$, keeping this flip might result in an optimal solution, since it might be possible to solve P' with s flips. It costs 1 flip to go from P to P', and $s + 1 \leq k$, so it is not yet impossible to reach an optimal solution. Thus, the flip was good. Of course, we do not know k when given an arbitrary input path P. We can, however, use this knowledge to, given an input path P, find a solution with a size smaller than or equal to a specific size (if such a solution exists). We simply treat the specified size as the size of the optimal solution. An algorithm that finds a solution of a specified size is given in Algorithm 2. Note that instead of finding a smallest set S of edge flips such that after

applying S to P, the resulting path P' contains no cycles, the algorithm finds P'.

Algorithm 2: FindSolutionOfSize(Size s, Path p, CycleSet C, path *Solution*)

Input: A size s, a path p, a cycle set C, and a path *Solution*.
Result: *Solution* contains a path p' of identical length as p, which contains no cycles, and which differs with p in at most s edges.
Output: True if a solution of size s for path p has been found, False otherwise.

```
1  Cycle ← first cycle of C;
2  for edge in Cycle do
3      if the edge is in its original direction then
4          for every direction that is not the original direction of the
             edge do
5              p' ← flip edge in direction;
6              CycleSet ← FindMaxCycleSet(p');
7              if size CycleSet = 0 then
8                  Solution ← p';
9                  return True;
10             else if size Cycleset ≤ s − 1 then
11                 if FindSolutionofSize(s - 1, p', CycleSet, Solution) =
                     True then
12                     return True;
13 return False;
```

Given a size s, a path P and its largest set of non-overlapping cycles C, and a path *Solution*, this algorithm returns true if path P has a solution of size at most s, and false otherwise. Furthermore, if P has a solution of size at most s, *Solution* is modified such that it contains a path P' that contains no cycles such that the cost of obtaining P' from P is at most s.

It does this by taking the first cycle c of C and handling each edge e of c in order. If the edge is unflipped, for each direction d that is not its original direction we apply the edge flip (e, d) to get a path P'. After finding the largest set of non-overlapping cycles of P', we check if P' still contains cycles. If it does not, we found a solution and return true. If it does, we check if (e, d) was good or bad. If (e, d) was good, we continue exploring P' by trying to find a solution of size $s - 1$ for P'. If such a solution exists, we can return true. Otherwise, we continue to the next edge flip.

Running time analysis. Line 1 takes $O(1)$ time. The for loop on lines 2 - 12 is executed at most n times, with the worst case happening when the cycle is n edges long, and the last edge of the cycle is the only edge of the cycle whose flip leads to an optimal solution.

The check on line 3 takes $O(1)$ time. If the check passes (which happens at most n times, when no edges of the path have been flipped yet), the for loop on lines 4-12 is executed.

The for loop on lines 4 - 11 is executed at most 3 times, one for each direction that is not the original direction of the edge. Thus, everything inside this for loop is executed at most $3n$ times.

Flipping an edge takes $O(n)$ time. FindMaxCycleSet() takes $O(n)$ time. Lines 7 - 10 also take $O(1)$ time each. In line 11, the algorithm calls itself with $s = s - 1$.

If FindSolutionOfSize() is called with $s = 1$, execution takes $O(n^2)$ time. Thus, the following recurrence relation can be derived:

$$T(s) = \begin{cases} 3nT(s-1) + O(n^2), & \text{if } s > 1. \\ O(n^2), & \text{if s} = 1. \end{cases} \tag{4.1}$$

This recurrence relation indicates that the running time of FindsolutionOfSize() is $O(3^s * n^{s+1} - n)$, where n is the number of edges in the path, and s is the size of the solution. We will prove this by induction on s: we will prove that there exists a constant $c > 0$ such that for all $s \geq 1$, $T(s) \leq c * (3^s * n^{s+1} - n)$.

First, we use the definition of big O notation to eliminate the occurrences of big O notation in our recurrence relation:

$$T(s) \leq \begin{cases} 3nT(s-1) + b * n^2, & \text{if } s > 1. \\ d * n^2, & \text{if s} = 1. \end{cases} \tag{4.2}$$

for some $b > 0$ and some $d > 0$.

Base Case ($s = 1$): According to the recurrence relation, we have:

$$T(1) \leq d * n^2.$$

For all $c \geq d/2$, we have:

$$d * n^2 \leq c * 3^1 * n^2 - cn$$

Thus, if we pick $c > d/2$, $T(1) \leq c * (3^1 * n^2 - n)$.

Induction Hypothesis: There exists a constant $c > 0$ such that for all s' with $1 \leq s' \leq s$, $T(s) \leq c * (3^s * n^{s+1} - n)$.

Inductive Step: Assume the induction hypothesis holds for s. We will prove that it also holds for $s + 1$, that is, $T(s + 1) \leq c * (3^{s+1} * n^{s+2} - n)$.

If we fill in $s + 1$ in the upper equation of recurrence relation 4.2, we get the following equation:

$$T(s + 1) \leq 3nT(s) + b * n^2$$

By applying the induction hypothesis, we get the following:

$$T(s+1) \leq 3nc * (3^s * n^{s+1} - n) + b * n^2$$

We can rewrite this as:

$$T(s+1) \leq c(3^{s+1} * n^{s+2}) - (3c - b)n^2$$

Thus, for $T(s+1)$ to be smaller than or equal to $c * (3^{s+1} * n^{s+2} - n)$, we need $(3c - b) \geq c$. This holds for all $c \geq b/2$, so for all $c \geq b/2$ we have:

$$T(s+1) \leq c * (3^{s+1} * n^{s+2} - n)$$

If we pick $c = max(b, d)$, we have a constant $c > 0$ which is both bigger than $d/2$ and $b/2$, which means we have found a $c > 0$ such that for all $s \geq 1$, $T(s) \leq c * (3^s * n^{s+1} - n)$. Thus, we can conclude that the running time of FindSolutionOfSize(s, P, C, Solution) is $O(3^s * n^{s+1} - n)$.

4.2.1 FindSolutionofSize() versus bruteforce methods

Instead of using FindSolutionofSize(), we can use bruteforce to find a solution of size s. After all, there are only $3^s * \frac{n!}{s!(n-s)!}$ possible paths P' that can be obtained when applying s edge flips to a path P of length n. Assuming that obtaining a path P' and checking if it contains cycles takes $O(n)$ time, a bruteforce method that checks all possible paths P' would take $3^s * \frac{n!}{s!(n-s)!} * O(n)$ time. This does not exceed $O(3^s * n^{s+1} - n)$. However, in practice, FindSolutionofSize() might find a solution significantly faster than its asymptotically worst running time. This is because of a couple of reasons, which we detail in this section.

Cycle length. First of all, we have the length of the cycles. In our running time analysis, we considered the worst case of each cycle being n edges long. However, if a cycle is n edges long and in the cycle set, it is the only cycle in the entire path P. Furthermore, if a cycle is n edges long and in the cycle set, it is not in a spiral, nor is it a loop. Thus, it can always be fixed with only one edge, and it is very unlikely that the only edge with which such a cycle can be fixed, is the last edge. Thus, even though the for-loop on lines 2-12 is executed at most n times, it is highly likely that it is executed significantly less times. Let the length of any of the cycles we encounter be at most n/r, with $1 \leq r \leq n/2$ since the minimum length of a cycle is 2 and the maximum length is n. Then our recurrence relation becomes

$$T(s) = \begin{cases} 3\frac{n}{r}T(s-1) + O(\frac{n^2}{r}), & \text{if } s > 1. \\ O(\frac{n^2}{r}), & \text{if s} = 1. \end{cases} \tag{4.3}$$

The corresponding running time is

$$O((3/r)^s * n^{s+1} - n).$$

We can prove this by induction on s: we will prove that there exists a constant $c > 0$ such that for all $s \geq 1$, $T(s) \leq c * ((3/r)^s * n^{s+1} - n)$. First, we use the definition of big O notation to eliminate the occurrences of big O notation in our recurrence relation:

$$T(s) \leq \begin{cases} 3\frac{n}{r}T(s-1) + b * \frac{n^2}{r}, & \text{if } s > 1. \\ d * \frac{n^2}{r}, & \text{if s} = 1. \end{cases} \tag{4.4}$$

for some $b > 0$ and some $d > 0$.

Base Case $(s = 1)$**:** According to the recurrence relation, we have:

$$T(1) \leq d * \frac{n^2}{r}$$

For all $c > 2d/5$, we have:

$$d * \frac{n^2}{r} \leq c((3/r)^1 * n^2 - n)$$

Thus, if we pick $c > 2d/5$, $T(1) \leq c * ((3/r)^1 * n^2 - n)$.

Induction Hypothesis: There exists a constant c such that for all s' with $1 \leq s' \leq s$, $T(s) \leq c * ((3/r)^s * n^{s+1} - n)$.

Inductive Step: Assume the induction hypothesis holds for s. We will prove that it also holds for $s+1$, that is, $T(s+1) \leq c * ((3/r)^{s+1} * n^{s+2} - n)$. If we fill in $s+1$ in the upper equation of recurrence relation 4.4, we get the following equation:

$$T(s+1) \leq 3\frac{n}{r}T(s) + b * (\frac{n^2}{r})$$

By applying the induction hypothesis, we get the following:

$$T(s+1) \leq 3\frac{n}{r}c * ((3/r)^s * n^{s+1} - n) + b * (\frac{n^2}{r})$$

We can rewrite this as:

$$T(s+1) \leq c * (3/r)^{s+1} * n^{s+2} - (\frac{3c}{r} - \frac{b}{r})n^2$$

Now for all $c > 2b/5$ we have:

$$T(s+1) \leq c * (\frac{3}{r}^{s+1} * n^{s+2} - n)$$

If we pick $c = max(b, d)$, we have a constant $c > 0$ which is both bigger than $2b/5$ and $2d/5$, which means we have found a $c > 0$ such that for all $s \geq 1$, $T(s) \leq c * ((3/r)^s * n^{s+1} - n)$. Thus, we can conclude that when the length of any cycle we encounter is at most n/r, the running time of FindSolutionofSize(s, P, C, Solution) is $O(\frac{3}{r}^s * n^{s+1} - n)$.

Recursive calls. Second, we have the number of recursive calls that are actually made. The algorithm only makes a recursive call when, after x edge flips, the size of the cycle set is less than or equal to $s - x$.

Consider the case where s is equal to the size of the largest set C of non-overlapping cycles in the input path P. This means that edge flips only lead to a recursive call if said edge flip decreases the size of the cycle set. An edge flip of a cycle c_i can only decrease the size of the cycle set if it does not introduce any new cycles that end before c_{i+1} starts. Each edge e_i in the cycle has at least one flip which causes a cycle that ends before c_{i+1} starts namely flipping it in the opposite direction of e_{i-1}. Each edge e_i whose neighbouring edges e_{i-1} and e_{i+1} have different directions has two flips which cause cycles that end before c_{i+1} starts, unless e_i is the final edge of c_i. Some edges might have no directions that trigger a recursive call. Thus, even if only the last edge of the cycle has a flip that leads to a solution, we will have made at most $2 * cyclelength$ recursive calls instead of $3 * cyclelength$, which reduces the running time to $O((2/r)^{s+1} * n^{s+2} - n)$.

Of course, this only holds when s is equal to the size of the largest set C of non-overlapping cycles. The bigger the difference between the two, the more paths we are allowed to explore, and the longer the running time.

Determining if a solution of size s exists. Third, while a bruteforce method would need to check all $3^s * \frac{n!}{s!(n-s)!}$ possible paths before it can be sure that there is no solution of size s, FindSolutionofSize() returns false as soon as it finds that all paths reachable by flipping an edge of the first cycle need more than $s - 1$ flips to be solved. In the best case, it does not need to make any recursive calls to find this out, as we can already know this after calculating the cycle set of a path. This property is especially convenient for our intentional use of the algorithm, which is running it on the same path P with increasing s until we find a value for s for which a solution exists.

4.3 SmarterBruteForce

We can use FindSolutionofSize(s, p, C, Solution) to find the optimal solution for a given path P by executing this algorithm multiple times, each time increasing s until the algorithm has found a solution. This is shown in Algorithm 3. Given an input path P, this algorithm first calculates a lower bound for P which it uses as initial size in FindSolutionOfSize(s, p, C, Solution). The algorithm calls FindSolutionOfSize(s, p, C, Solution) until a solution is found, at which point it returns the solution.

SmarterBruteForce takes as input a path P and outputs a path P' of the same length which contains no cycles, and which is as close as possible to path P (least number of edges are violated).

Algorithm 3: SmarterBruteForce(path p)

Input: A path P.

Output: A path P' of identical length as P, which contains no cycles, and which is as close as possible to P (least number of edge directions are violated).

```
1  MaxCycleSet ← FindMaxCycleSet(p);
2  LowerBound ← size MaxCycleSet;
3  if LowerBound = 0 then
4      return p;
5  FoundSolution ← False;
6  Solution← ∅;
7  for i ← LowerBound to length(p) do
8      FoundSolution ← FindSolutionOfSize(i, p, MaxCycleSet,
         Solution);
9      if FoundSolution = True then
10         return Solution;
11 return Solution;
```

Running time analysis. The running time of SmarterBruteForce() is largely dependent on the running time of FindSolutionOfSize(). SmarterBruteForce() executes FindSolutionOfSize() $k-i+1$ times, where k is the size of the optimal solution, and i is the lower bound we found.

In SmarterBruteForce, FindSolutionOfSize(i, p, C, Solution) with running time $T(i) = O(3^i * n^{i+1} - n)$ is executed $k-$ *LowerBound* (LB) times, where k is the number of edges flipped in the optimal solution. Each time, i is increased by 1. Thus, we have

$$\sum_{i=LB}^{k} T(i) \leq \sum_{i=LB}^{k} c(3^i * n^{i+1} - n)$$

In the worst case, the computed lower bound is 1, so we get

$$\sum_{i=1}^{k} c(3^i * n^{i+1} - n) \approx \frac{cn(3n(3^k n^k - 1) - 3kn + k)}{3n - 1} = O(3^{k+1} n^{k+2})$$

Thus, the running time of SmarterBruteForce is $\mathrm{O}(3^{k+1} n^{k+2})$.

Chapter 5

NP-Hardness

Planar 3-SAT is NP-hard. We show that this problem is also NP-hard by showing how to make a reduction from planar 3-SAT.

For any planar 3SAT formula F, we show how to construct a path P such that the size of the optimal solution of P is less than or equal to some number k if and only if F is satisfiable. The exact value of k depends on

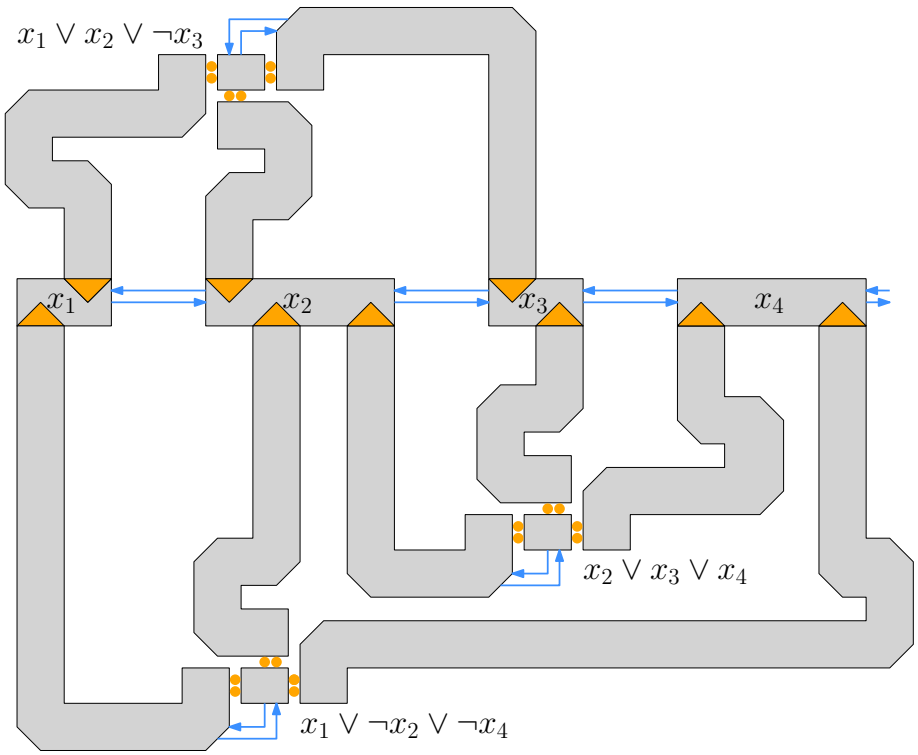


Figure 5.1: Overview of a possible final construction. Grey blocks represent gadgets. Orange triangles indicate shared subpaths, orange circles indicate shared vertices. The blue arrows connect the gadgets to obtain a connected path P.

the final construction of P.

We define variable gadgets, propagation gadgets, and clause gadgets, which represent the variables, clauses, and edges of F. We then use these to construct a path P that can be solved with k edge flips if and only if F is satisfiable.

Each gadget consists of a number of connected subpaths, and is, in turn, a (collection of) subpath(s) of P. Subpaths interact with each other through shared vertices. Gadgets interact with each other through shared vertices, or shared subpaths.

A schematic overview of a resulting construction is given in Figure 5.1. Grey blocks represent gadgets. Orange triangles indicate shared subpaths, orange circles indicate shared vertices. The blue arrows connect the gadgets to obtain a connected path P.

5.1 Variable gadget

A variable gadget is a set V of subpaths of P that have exactly two isolated optimal solutions, arranged such that the entire set V has exactly two isolated optimal solutions. The size of the isolated optimal solution of V is equal to the sum of the sizes of the isolated optimal solutions of all subpaths in V. The variable gadget has a number of subpaths on its top and bottom sides that can be shared with the propagation gadget as a form of connection point. A schematized version of a variable gadget is shown in Figure 5.2.

5.1.1 The binary subpath

The main building block of a variable gadget, as well as a propagation gadget, is the path shown in Figure 5.3. For the purpose of easy referencing, we call this subpath the *binary subpath*. We can construct our variable gadget by rotating, mirroring, and reversing duplicates of the binary subpath. The binary subpath can be rotated, mirrored, and reversed without affecting the size of the optimal solutions or the number of optimal solutions. This holds for any subpath: if both the shape of the subpath and the number of edges in a subpath remains the same, the number of solutions of the subpath is unaffected. Thus, rotating, mirroring, and reversing any subpath does not affect the size of the optimal solution or the number of optimal solutions, since as long as any of these operations is performed on the entire subpath the shape of the subpath does not change, nor does the number of edges in the subpath.

The binary subpath has two isolated optimal solutions of size two, which are shown in Figure 5.4. Each optimal solution of the binary subpath results in a distinct path, as indicated in Figure 5.4. We label these paths true (blue, right) and false (orange, left).

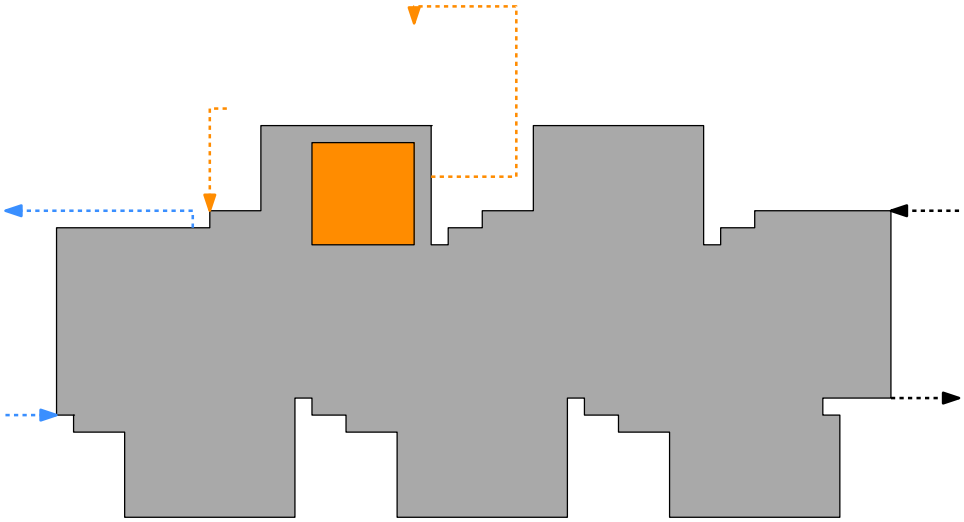


Figure 5.2: A schematized version of a variable gadget. The black arrows show the entry and exit path of the gadget. The blue arrows show where a variable gadget might be connected to another variable gadget. The orange arrows show an example of where a variable gadget might be connected to a propagation gadget. The orange rectangle shows an example of the location of a subpath that can be shared with a propagation gadget.

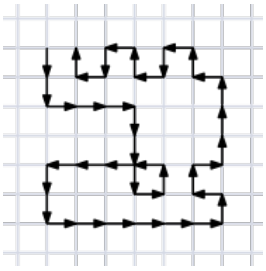


Figure 5.3: Subpath that is used in the construction of propagation and variable gadgets, drawn on an integer grid.

We can reason that these are indeed the only isolated optimal solutions as follows. There is no solution of size 0, because the path contains a cycle, and solving a cycle requires at least 1 edge flip. Figure 5.5 shows all possible paths B' that can be obtained by flipping a single edge of the cycle. All of these paths still contain at least one cycle, indicating that an optimal solution consists of at least 2 edge flips.

Of all paths that can be caused by flipping a single edge of the original cycle, only two (5.5d and 5.5h) can in turn be solved by flipping a single edge of the new cycles. Each of these paths P' has only one isolated optimal solution, thus there are only 2 isolated optimal solutions of size 2 for

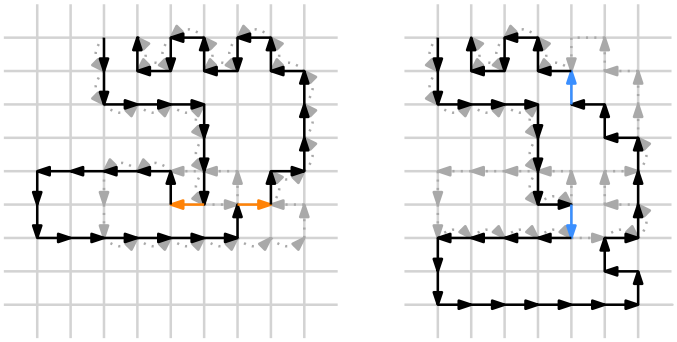


Figure 5.4: The two optimal solutions of the binary subpath used in construction of propagation and variable gadgets, drawn on an integer grid.

the original path P. One can verify with a bruteforce algorithm that the solutions shown in Figure 5.4 are indeed the only isolated optimal solutions of size two.

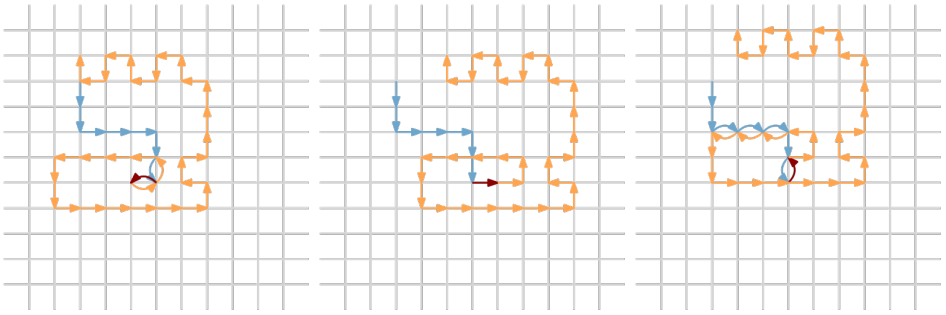
The reason that P has only 2 isolated optimal solutions is the 'tail' of the path; the part where P narrows down close to its start- and endpoint. After flipping any edge of the cycle, the pre-flip path entangles with the post-flip path in such a way that no matter which edge you flip, untangling it is impossible – except when you can perform a compensating edge flip, which essentially undoes the shift performed by the first edge flip. Because of the edges present in the path and the edges present in the cycle, there are only two edge flips in the cycle for which compensation works. These two edge flips lead to the two isolated optimal solutions of size 2.

Based on the knowledge that the subpath has two isolated solutions of size 2, we can make the following observation:

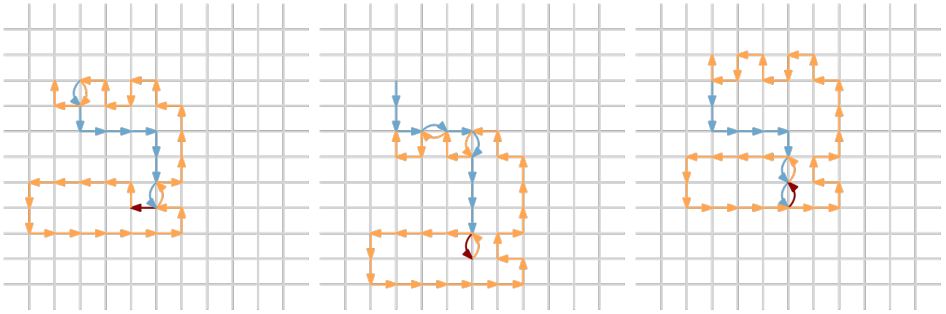
Observation 8. *The binary subpath as shown in Figure 5.3 has at most two non-isolated optimal solutions of size 2, and no non-isolated optimal solution of size smaller than 2.*

By Observation 2, we know that the size of any of the non-isolated optimal solutions of the binary subpath is bigger than or equal to 2. More specifically, we know that the size of a non-isolated optimal solution is equal to 2 if it is also an isolated optimal solution, and larger than 2 if it is not also a non-isolated optimal solution. Thus, no matter what path precedes or succeeds the binary subpath, we have at most two potential solutions of size 2, and no non-isolated optimal solutions of size smaller than 2.

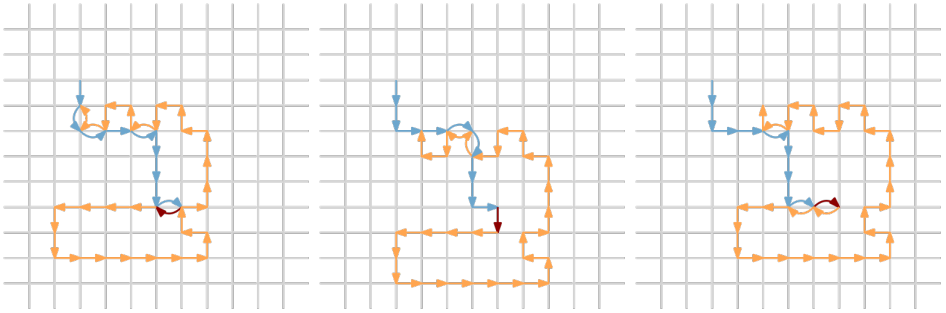
An important feature of the isolated optimal solutions of the binary subpath is that they consist of compensating edge flips. This means that the shift that is induced by the first edge flip is compensated by the second edge flip, which results in the final edges of the path returning to their original position. As a result, the optimal solutions do not cause shifts of



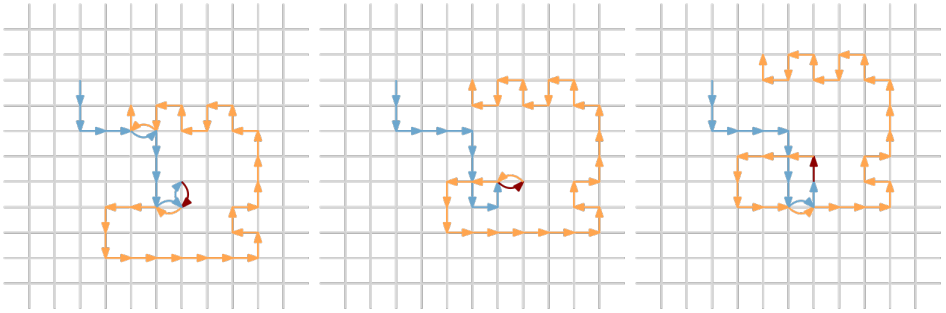
(a) A $D \leftarrow L$ flip. (b) A $D \leftarrow R$ flip. (c) A $D \leftarrow U$ flip.



(d) A $R \leftarrow L$ flip. (e) A $R \leftarrow D$ flip. (f) A $R \leftarrow U$ flip.



(g) A $U \leftarrow L$ flip. (h) A $U \leftarrow D$ flip. (i) A $U \leftarrow R$ flip.



(j) A $L \leftarrow D$ flip. (k) A $L \leftarrow R$ flip. (l) A $L \leftarrow U$ flip.

Figure 5.5: All possible paths B' resulting from a flip of a single edge that is part of the cycle contained in the subpath shown in Figure 5.3, and the cycles they cause. The pre-flip path is indicated in orange, the post-flip path is indicated in blue. The flipped edge itself is indicated in dark red. Paths 5.5d and 5.5h lead to an optimal solution.

the subsequent paths. This means that the only cycles that can be caused by applying these isolated optimal solutions to the path are caused by a collision between the shifted section of the subpath (the section between the two flipped edges) and any path that succeeds or precedes the subpath. Thus, the binary subpath can be solved optimally within a path Q if at least one of its isolated optimal solutions is not blocked by Q.

5.1.2 Binary triplets

We construct our variable gadget by arranging sets of three instances of the binary subpath in *binary triplets*, as shown in Figure 5.6.

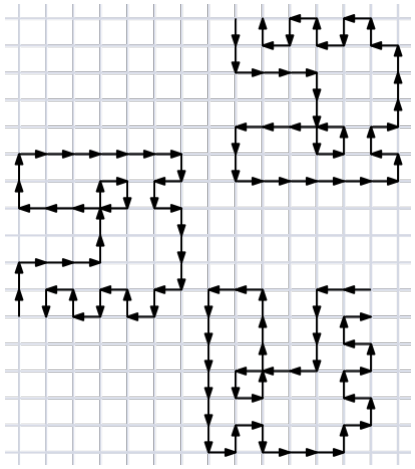


Figure 5.6: Triplet consisting of three binary subpaths whose isolated optimal solutions share vertices.

A binary triplet has two isolated optimal solutions of size 6: either all three of its binary subpaths are solved according to the blue solution, or all three of its subpaths are solved according to the orange solution. In Figure 5.7 the different optimal solutions of the binary subpaths are indicated in blue and orange.

We can confirm that the triplet indeed has two isolated optimal solutions of size 6 by reasoning as follows. As stated in Lemma 2, a lower bound to the optimal solution of a path T is equal to the sum of the isolated optimal solutions of all paths P_i of a subpath decomposition W.

Assume the binary subpaths B_a, B_b, and B_c in a triplet are connected to form a complete path T by a set S of edgewise non-overlapping subpaths, each with an isolated optimal solution of size 0. Assume the paths in S do not block any of the isolated optimal solutions of the binary subpaths. Then a subpath decomposition of T is $W = B_a \cup B_b \cup B_c \cup S$. Thus, a lower bound for the optimal solution of T is the sum of the isolated optimal solutions of B_a, B_b, B_c and all subpaths in S. Since all subpaths in S have an isolated

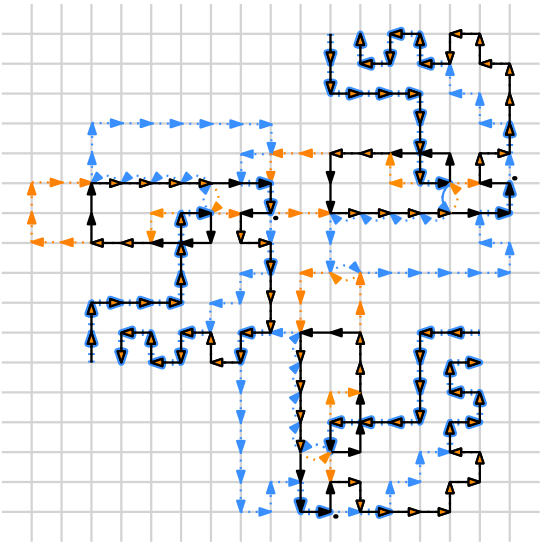


Figure 5.7: The different optimal solutions of the subpaths indicated in blue and orange.

optimal solution of size 0, the lower bound simply becomes the sum of the isolated optimal solutions of B_a, B_b, and B_c. Each of these three subpaths has an isolated optimal solution of size 2, so a lower bound of the isolated optimal solution of T is 6. According to Observation 5, this lower bound can be achieved if we can solve all three binary subpaths according to one of their isolated optimal solutions.

As is visible in Figure 5.7, the orange solution of each binary subpath of a triplet shares vertices with the blue solution of its counterclockwise neighbour. This automatically means that the blue solution of each subpath shares vertices with the orange solution of its clockwise neighbour. As such, choosing the blue solution for a subpath B_a blocks the orange solution for its clockwise neighbour B_b, which results in the only remaining isolated optimal solution for B_b being the blue solution. Choosing the blue solution for B_b blocks the orange solution for its clockwise neighbour B_c, which results in the only remaining isolated optimal solution for B_c being the blue solution. Thus, if we want to solve all three subpaths with one of their isolated optimal solutions, we have to pick the same solution for all subpaths. This results in two different solutions for the entire triplet: either all three subpaths are solved with the blue solution, or all three subpaths are solved with the orange solution.

We can also pick different isolated optimal solutions for neighbouring subpaths. However, this causes a cycle within the triplet, which needs to be solved with an additional edge flip, which leads to a solution that is larger

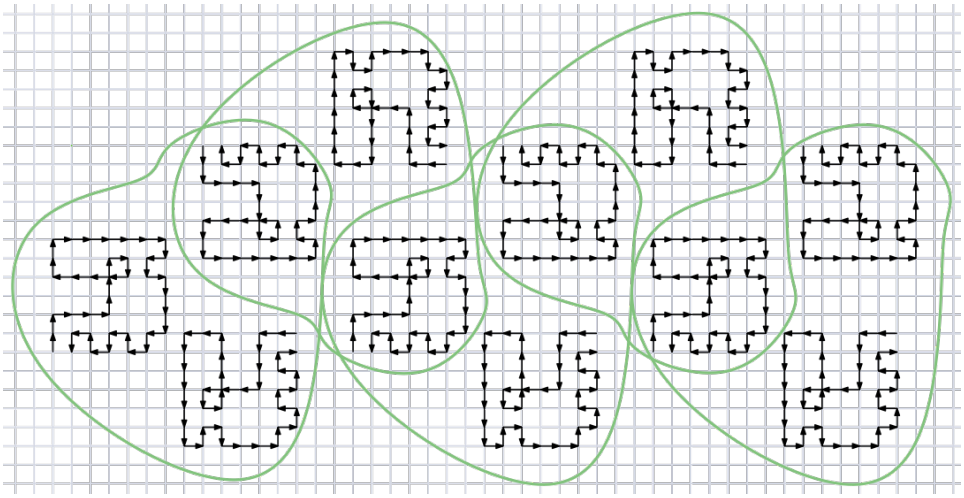


Figure 5.8: A variable gadget, with triplets circled.

than 6 and hence not optimal. Picking any non-isolated optimal solution for a subpath of the triplet that is not also an isolated optimal solution leads to a solution that is larger than 6 as well, since according to Observation 2, a solution that is only a non-isolated optimal solution is always larger than a solution that is also an isolated optimal solution.

Thus, a binary triplet has two isolated optimal solutions: either all three of its binary subpaths are solved according to the blue solution, or all three of its binary subpaths are solved according to the orange solution.

Note that although in the figures the subpaths in the triplet are disconnected, in the final construction they are connected. We will discuss how to connect all subpaths later.

5.1.3 Final variable gadget

Now we can construct a variable gadget by carefully arranging the binary subpaths such that if one binary subpath is true, all binary subpaths should be true, and if one binary subpath is false, all binary subpaths should be false, given that we want to solve the variable gadget optimally. This construction is shown in Figure 5.8. The different triplets are circled in this figure, showing that each triplet shares one subpath with each of its neighbours. This ensures that each triplet in a gadget is solved in the same way as its neighbouring triplets, which in turn ensures that the entire variable gadget has two optimal solutions: either all subpaths are solved with the blue 'true' solution, or all subpaths are solved with the orange 'false' solution.

By chaining multiple triplets together, the variable gadget can be made as wide as necessary.

Since our goal is to obtain a complete path P, we should connect the

subpaths used in the variable gadget. We do this by adding a path between each binary subpath and its counterclockwise neighbour, apart from between the first and last binary subpath of the gadget, as shown in Figure 5.9. These paths are placed such that they do not share vertices with the isolated optimal solutions of any of the binary subpaths, thus they do not impact the number of optimal solutions of the variable gadget, nor their size.

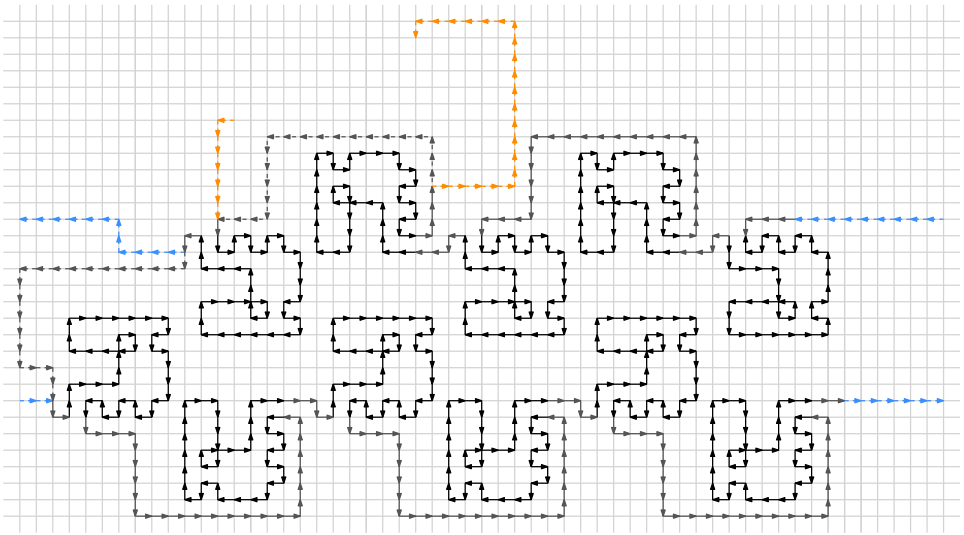


Figure 5.9: An example of how to connect the subpaths in a variable gadget. The grey arrows show the added subpaths. The blue arrows show where we can diverge from the path to connect the variable gadget to another variable gadget or to add more triplets. The orange arrows show where we can diverge from the path to connect the variable gadget to a propagation gadget.

If we do not need to connect any other gadgets to a variable gadget V, V is a complete path. We can make the following two observations based on the construction of the variable gadget:

Observation 9. *Let a connected variable gadget V consist of a set A of x copies of the subpath shown in Figure 5.3 and a set B of edgewise non-overlapping subpaths connecting the subpaths in A. Let all subpaths in B have an isolated optimal solution of size 0 which does not block any isolated optimal solution of any of the subpaths in A. Then the size of the isolated optimal solution of V is equal to $2 * x$.*

Each copy of the subpath shown in Figure 5.3 has an isolated optimal solution of size 2. A and B together form a path V with subpath decomposition $W = A + B$. As stated in Lemma 2, a lower bound for V is the sum of the isolated optimal solutions of the subpaths in W. Since in this case it is possible to solve all subpaths according to their isolated optimal

solution, the size of the isolated optimal solution of V is equal to the sum of the isolated optimal solutions of the subpaths in A, which is $2 * x$.

Observation 10. *Let a connected variable gadget V consist of a set A of x copies of the subpath shown in Figure 5.3 and a set B of edgewise non-overlapping subpaths connecting the subpaths in A. Let all subpaths in B have an isolated optimal solution of size 0 which does not block any isolated optimal solution of any of the subpaths in A. Let the size of the optimal solution of V be $2x$. Then V has two isolated optimal solutions.*

Since the size of the isolated optimal solution of V is $2x$, all x copies of the subpath shown in Figure 5.3 must have been solved according to one of their isolated optimal solutions. There are two ways of solving all x copies according to one of their optimal solutions: all subpaths are solved according to the orange solution, or all subpaths are solved according to the blue solution. Thus, V has two isolated optimal solutions: true and false.

5.2 Propagation gadget

The propagation gadget is a subpath or set of subpaths P that has two isolated optimal solutions: true and false. A propagation gadget communicates with its corresponding clause gadget by blocking vertices with its isolated optimal solutions. A propagation gadget communicates with its corresponding variable gadget via a shared subpath. A schematized example of a propagation gadget is given in Figure 5.10.

The propagation gadget consists of the same binary subpaths as the variable gadget, arranged in the same binary triplets. The propagation gadget can be connected to the variable gadget as shown in Figure 5.11. The triplets in the variable gadget (which is identical to the variable gadget as shown in Figure 5.8) are circled in green, while the triplets in the propagation gadget are circled in purple. The variable gadget and the propagation gadget have one shared binary subpath, which ensures that all binary triplets in the propagation gadget are solved in the same way as the binary triplets in the variable gadget.

5.2.1 Negated variables

The construction shown in Figure 5.11 works if the propagation gadget should take the same value as the variable gadget, which is the case if the literal used in the corresponding clause is the non-negated variable.

If the literal used in a clause is the negated variable, we connect the propagation gadget in a similar but slightly different manner. Instead of constructing a normal binary triplet using a 'free' subpath F belonging to the variable gadget, we use F to construct a negating triplet. That is,

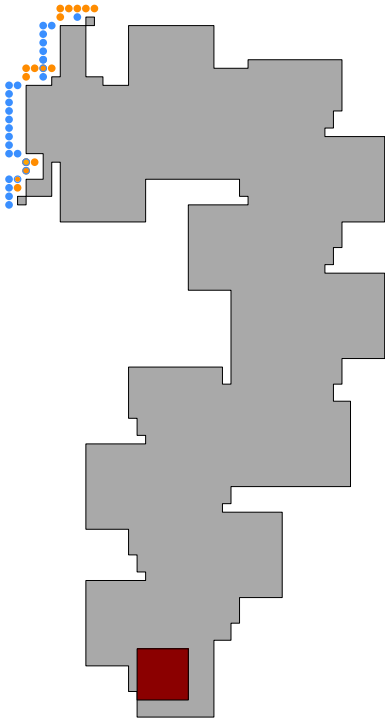


Figure 5.10: A schematized example of a propagation gadget. The red rectangle indicates a shared subpath, which is used to communicate with a variable gadget. Vertices blocked by the blue or orange solution of the propagation subpath are indicated by blue and orange dots. These vertices are used to communicate with the clause gadget.

we construct a triplet where two binary subpaths must be solved with the isolated optimal solution opposite to the isolated optimal solution of the third binary subpath. For example, if the third binary subpath is solved with the blue solution, the two other subpaths have to be solved with the orange solution, and vice versa. Such a negating triplet is shown in Figure 5.12. The orange and blue arrows indicate the subpaths that would be followed if the orange or blue solution is chosen.

Both solutions of this negating triplet are shown in Figure 5.13. Note that in this triplet, the upper right binary subpath is elongated. The purpose of elongating this binary subpath is to give its neighbouring binary subpath enough room to get connected to the other subpaths without causing any new cycles or blocking an isolated optimal solution of any of the binary subpaths. Elongating the binary subpath in this way can be done without altering the number of isolated optimal solutions, their size, or their general shape, since any flips of the inserted edges do not lead to an additional optimal solution, nor does the inserted path interfere with already existing optimal solutions.

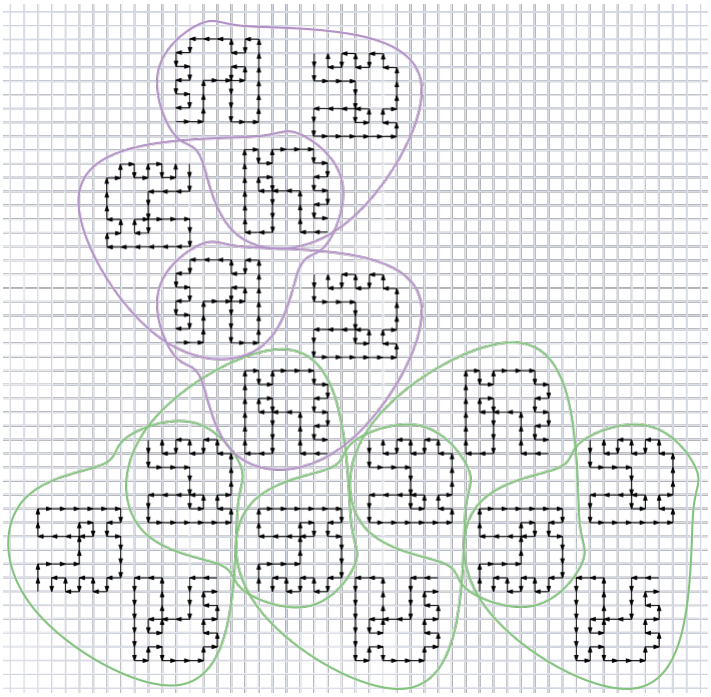


Figure 5.11: A (part of) a propagation gadget connected to a variable gadget. Triplets in the propagation gadget are circled in purple, triplets of the variable gadget are circled in green. This propagation gadget takes the same value (true or false) as the variable gadget.

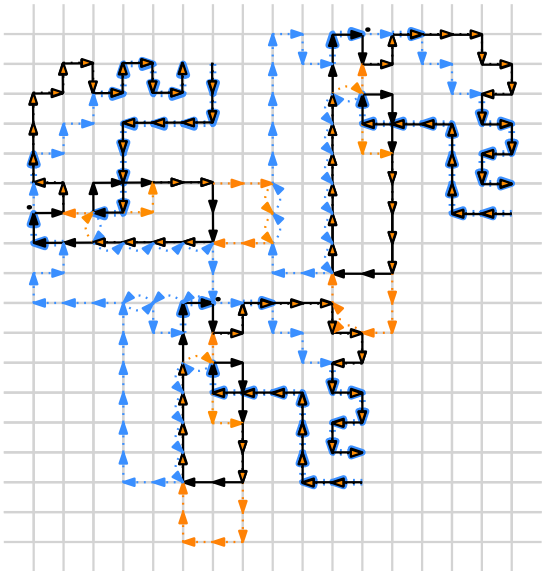


Figure 5.12: A triplet that can be used if we need the negated version of the variable.

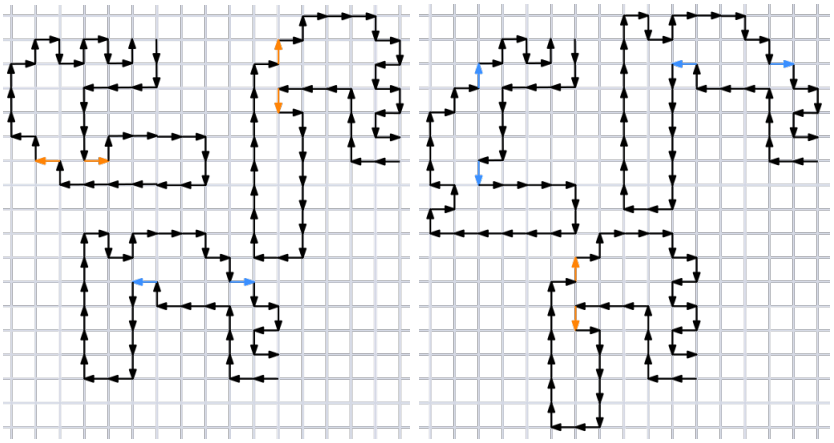


Figure 5.13: The two solutions of a negating triplet.

The upper two binary subpaths are forced to an identical solution, while the lower binary subpath should take the opposite solution. This is achieved by letting the blue solution of the lower binary subpath share vertices with the blue solution of one of the upper binary subpaths and letting the orange solution of the lower binary subpath share vertices with the orange solution of the other upper binary subpath.

The two upper subpaths B_a and B_b communicate with each other as normally; the orange solution of the left subpath B_a shares vertices with the blue solution of the right subpath B_b. This way, choosing the blue solution for the lower subpath B_c blocks the blue solution of the B_a, which will need to be solved with the orange solution if we want to solve it optimally within P. This blocks the blue solution of B_b, leaving only the orange solution for B_b if we want to solve it optimally within P.

On the other hand, if we B_c solve with the orange solution, we block the orange solution of B_b, forcing B_b to be solved with the blue solution; which forces B_a to also be solved with the blue solution.

Thus, if we want to solve all three subpaths with one of their isolated optimal solutions, we have to pick an identical solution for B_a and B_b, but the opposite solution for B_c. Since, according to Observation 5, we can only achieve the lower bound for the entire negation triplet by solving all three of its binary subpaths according to one of their isolated optimal solutions, and there is no other way to be able to solve all three binary subpaths according to one of their optimal solutions, the negation triplet does not have any more isolated optimal solutions.

When connecting the propagation gadget to the variable gadget, we let the lower binary subpath of the negating triplet be part of the variable gadget and the upper subpaths be part of the propagation gadget. This way, if the variable gadget is true, the subpaths in the propagation gadget are

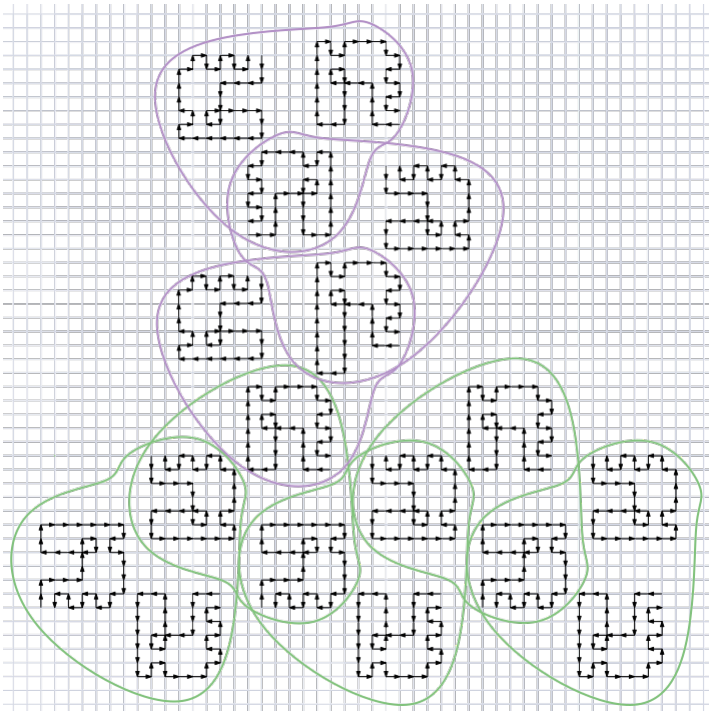


Figure 5.14: A (part of) a propagation gadget connected to a variable gadget. Triplets in the propagation gadget are circled in purple, triplets of the variable gadget are circled in green. This propagation gadget takes the opposite value (true or false) to the variable gadget.

all false, and if the variable gadget is false, the subpaths in the propagation gadget are all true. Figure 5.14 shows what this looks like.

5.2.2 Connecting subpath

The propagation gadget should be able to interact with the clause gadget; however, it should also be connected. With the binary subpaths we have discussed so far, it is impossible to achieve both: if we want the propagation gadget to interact with the clause gadget, the clause gadget should share vertices with the isolated optimal solutions of the binary subpaths. However, if we let the clause gadget share vertices with the isolated optimal solutions of the binary subpaths, we cannot connect the right half of the propagation gadget to the left half of the propagation gadget, which will result in a disconnected final path P.

Therefore, the propagation gadget needs a second building block: a subpath that connects the right half of the propagation gadget to the left half of the propagation gadget and is crucial for ensuring that the final path is connected. To make the clause gadget work properly, this subpath also

needs exactly two optimal solutions, which match the solutions of the propagation gadget: if the propagation gadget is solved as 'true', the subpath is also solved as 'true'; if the propagation gadget is solved as 'false', the subpath is also solved as 'false'. Furthermore, these solutions should be able to impact the clause gadget: we want the 'false' solution to share vertices with an optimal solution of the clause gadget.

Such a subpath, called a *connecting subpath*, is shown in Figure 5.15.

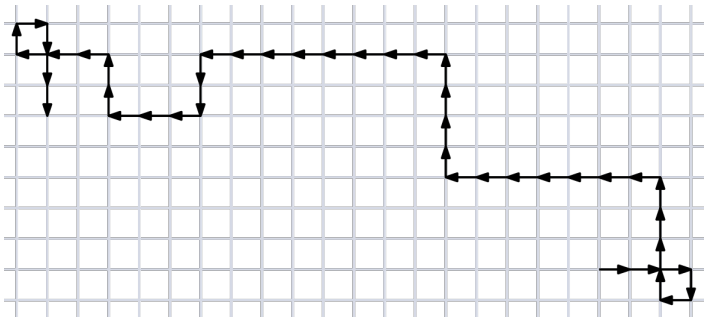


Figure 5.15: Subpath used to connect the two sides of a propagation gadget.

Figure 5.16 shows how the connect the connecting subpath to the binary triplets. It also shows how we can connect all binary subpaths in the propagation gadget, which is similar to how we connect all binary subpaths in a variable gadget.

This connecting subpath consists of two loops, connected by a path. Each of these loops has two ways of being solved with a single edge, resulting in four solutions of the subpath, of which two are distinct. These solutions are shown in Figure 5.17. The original (unsolved) path is drawn in grey as reference.

As can be seen, the solutions in Figure 5.17a and 5.17b shift (most of) the path upward, which we call the blue or true solutions, while the solutions in Figure 5.17c and 5.17d shift (most of) the path to the left, which we call the orange or false solutions. Of each of these pairs, one solution keeps the position of the last edge of the subpath intact (Figure 5.17a and 5.17c), while the other solution shifts the position of the last edge (Figure 5.17b and 5.17d).

While shifting the position of the last edge is unproblematic when solving the connecting subpath in isolation, it can become problematic when the connecting subpath is part of a larger path. Shifting the position of the last edge causes the entire subsequent path to shift with respect to the preceding path, which may cause new cycles but does not solve any cycles, unless there already was a cycle that started somewhere before the edge flip(s) that caused the shift, and ended somewhere after the edge flip(s) that caused the shift.

Therefore, the solutions that cause the path to shift might not be optimal

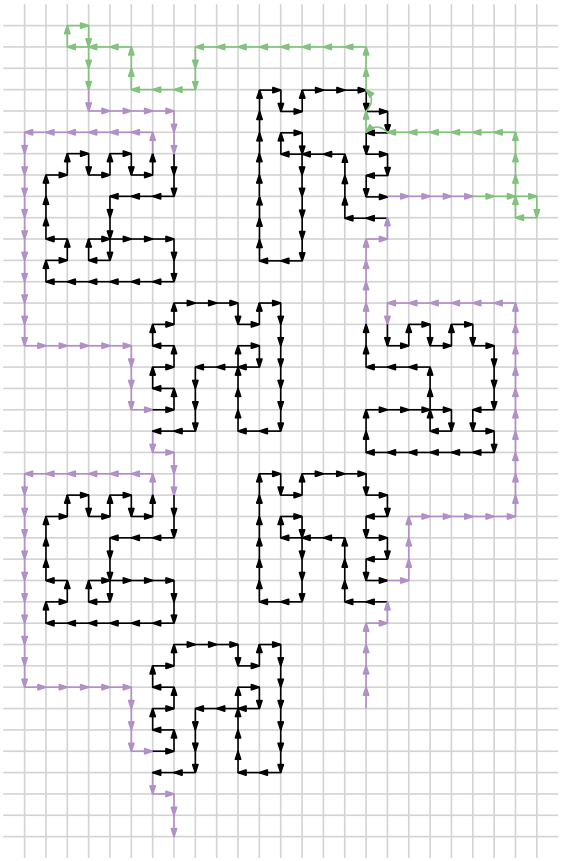
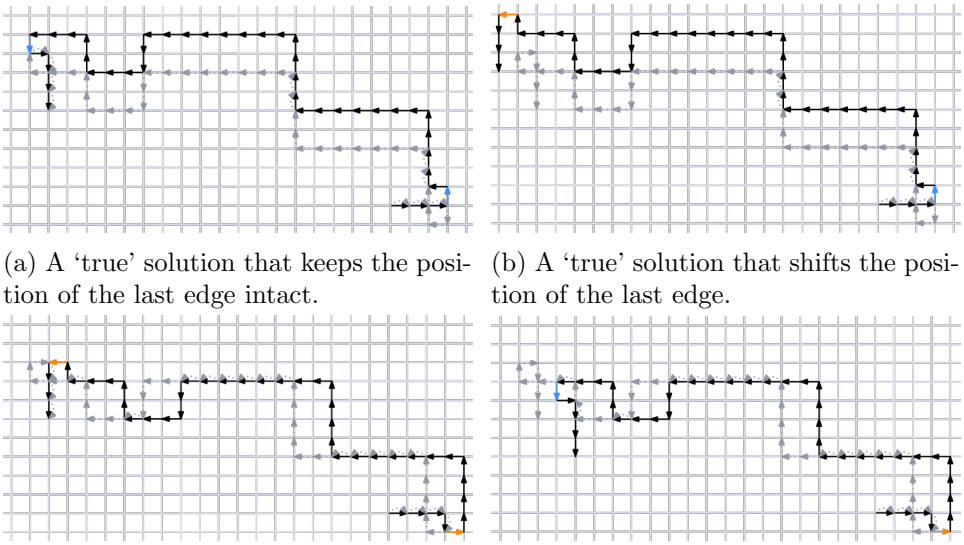


Figure 5.16: Part of a propagation gadget, showing how to connect a connecting subpath to the rest of the gadget. The connecting subpath is shown in green, edges connecting subpaths within the propagation gadget are shown in purple.

if we connect a path to the connecting subpath. Since shifting the path undermines the guarantee that every subpath in the propagation gadget is solved identically, we in fact want to ensure that the two solutions that shift the path are not optimal within P.

Thus, we need the shifts to cause cycles within the propagation gadget that cannot be solved by solving the binary subpaths according to one of their optimal solutions. This happens when we place the pre-flip path and the post-flip path such that the shift makes them collide. We have one solution that shifts the left side of the propagation gadget (when orientating the construction as shown in Figure 5.16 2 to the left and 2 up, and one solution that shifts the left side of the propagation gadget 2 to the right and 2 down. Therefore, to make sure that the left side of the propagation gadget collides with the right side, we need the left side to be at most 2 to the right and 2 down relative to the right side at some point, and 2 to the left and 2



(a) A 'true' solution that keeps the position of the last edge intact.

(b) A 'true' solution that shifts the position of the last edge.

(c) A 'false' solution that keeps the position of the last edge intact.

(d) A 'false' solution that shifts the position of the last edge.

Figure 5.17: The four isolated optimal solutions of the connecting subpath. Flipped edges are indicated in blue and orange. The original (unsolved) path is indicated in grey.

up relative to the right side at some other point.

To achieve this, we let the propagation gadget make a rough c-shape, as shown in Figure 5.18.

This c-shape curls the shifted subpath around the unshifted subpath, such that each of the two shifts causes a cycle between the unshifted subpath and the shifted subpath. These cycles occur between nodes that are essential to both optimal solutions of a binary subpath: neither the blue nor the orange solution frees up the nodes, and thus solving the cycle requires at least one extra edge flip, resulting in a suboptimal solution. Figure 5.19 and Figure 5.20 show how the subpaths in the c-shape would be shifted. Newly colliding edges are circled in green.

Since we have multiple of these connecting subpaths, which each can introduce a shift, we need to consider the possibility of one shift compensating for another shift. This, however, is impossible because of the placements of the connecting subpaths. If we were to place another connecting subpath at some point before the occurrence of the newly caused cycles, this second connecting subpath might indeed compensate for the shift of the first connecting subpath. However, any second connecting subpath will always occur after the newly caused cycles, since we only place one connecting subpath per propagation gadget, and the cycles are caused within the same propagation gadget.

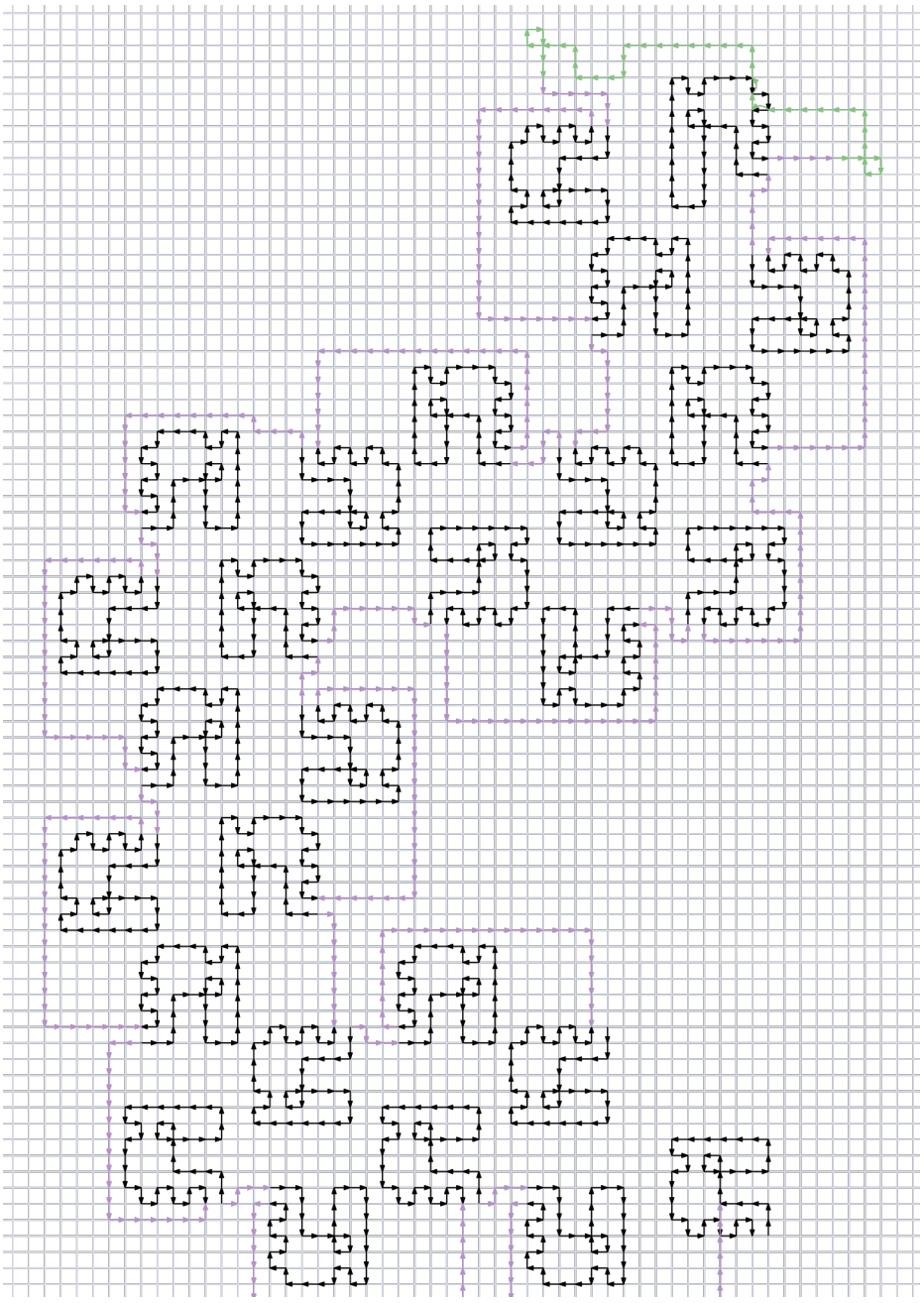


Figure 5.18: A c-shape.

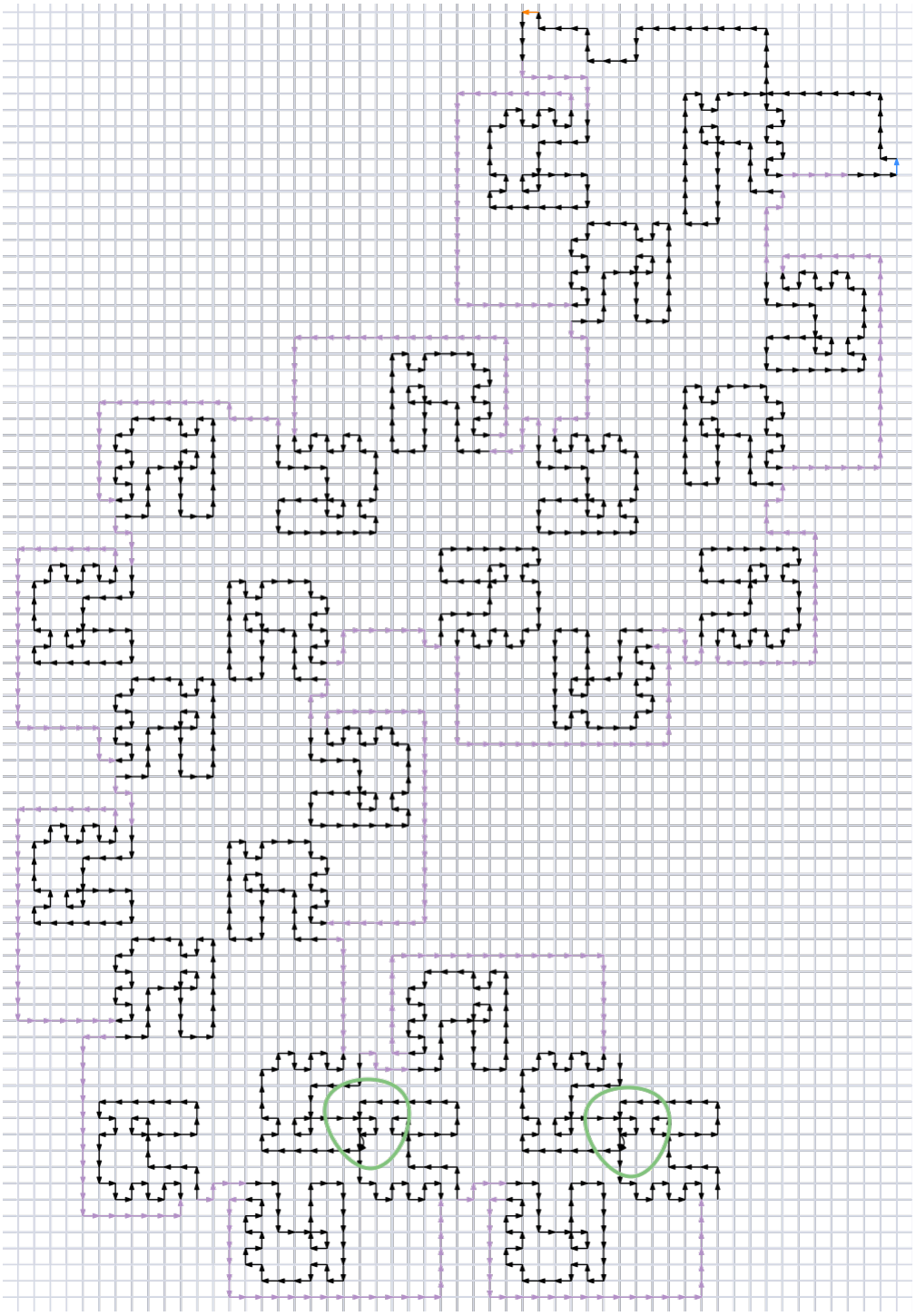


Figure 5.19: Cycles caused by allowing the connecting subpath to shift the subsequent subpath.

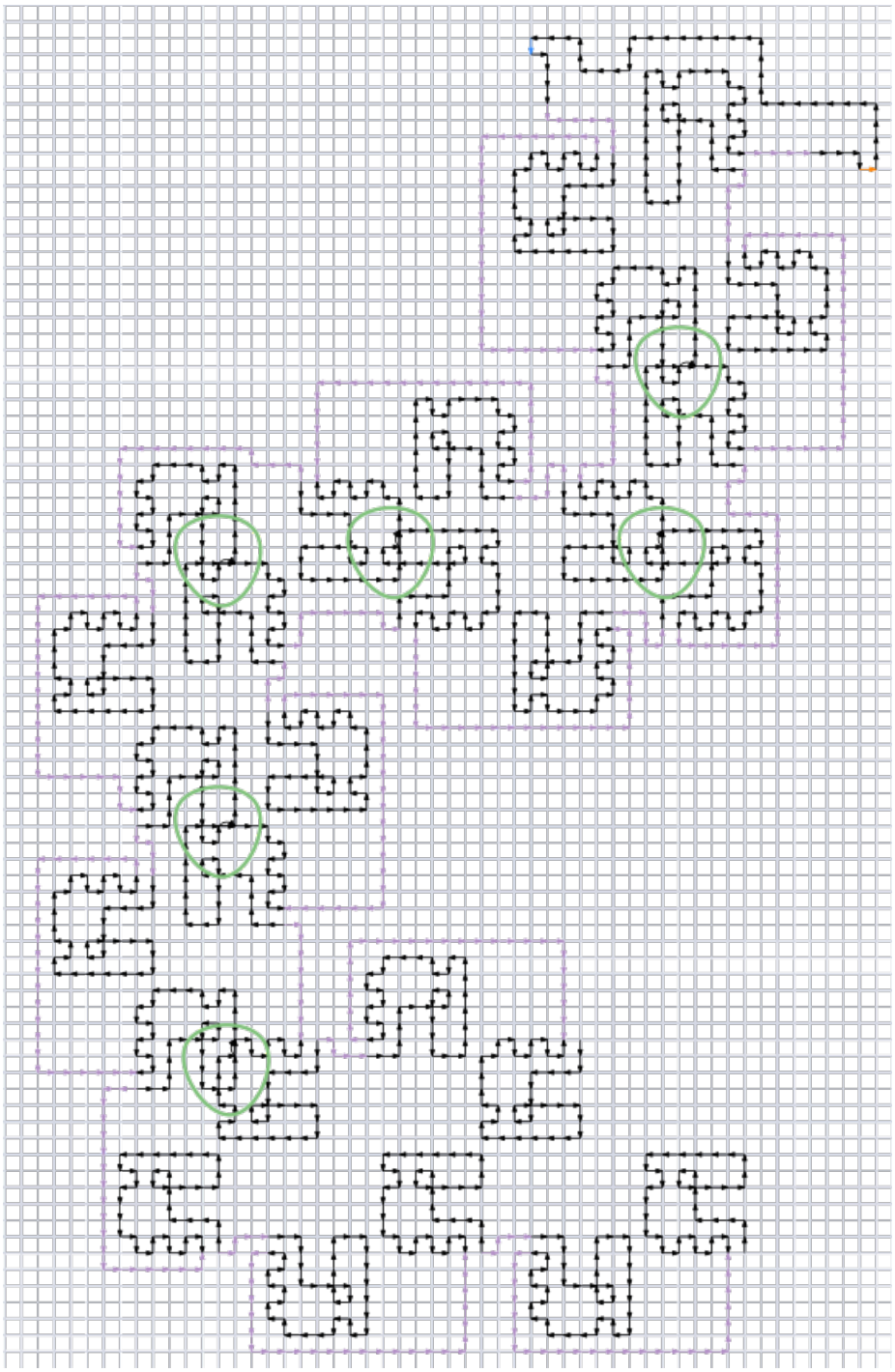


Figure 5.20: Cycles caused by allowing the connecting subpath to shift the subsequent subpath.

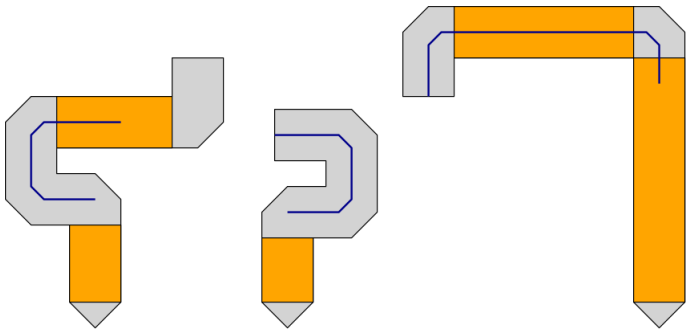


Figure 5.21: The three versions of the propagation gadget. The adjustable areas are indicated in orange. The c-shape is indicated with a blue line.

Thus, we can conclude that, while these connecting subpaths might have four optimal solutions in isolation, this number is reduced to two within the complete path P: true and false. The placement of the connecting subpath with respect to the rest of the propagation gadget ensures that the connecting subpath has to be solved with the true solution if the rest of the gadget is solved with the true solution, and that it has to be solved with the false solution if the rest of the subpath is gadget with the false solution. Thus, the entire subpath that is the propagation gadget has exactly two optimal solutions: true and false.

Because of the specification of the clause gadget, we need three different versions of the propagation gadget, which all bend in a slightly different way. A schematization of these three versions is given in Figure 5.21. The width of the c-shape is fixed, but other dimensions can be adjusted as needed.

Based on the specification of the propagation gadget, we can make the following two observations:

Observation 11. *Let Q be a propagation gadget consisting of a set X of x binary subpaths, 1 connecting subpath Q_c, and a set S of edgewise non-overlapping subpaths connecting the binary subpaths and the connecting subpath, each with an isolated optimal solution of size 0 which does not block any isolated optimal solution of any of the subpaths in Q. Then the isolated optimal solution of Q has size $2x + 2$.*

Q has a subpath decomposition $W = X \cup Q_c \cup S$. According to Lemma 2, a lower bound to the isolated optimal solution of Q is the sum of the isolated optimal solutions of all subpaths in W. Because all subpaths in S have an isolated optimal solution of size 0, this is reduced to the sum of the isolated optimal solutions of all paths in X and the isolated optimal solution of Q_c. All subpaths in X have an isolated optimal solution of size 2, and Q_c has an isolated optimal solution of size 2; thus, a lower bound to the isolated optimal solution of Q is $2x + 2$. This lower bound can be achieved if all

subpaths in W can be solved according to one of their optimal solutions, which is possible in this case.

Observation 12. *Let Q be a propagation gadget consisting of a set X of x binary subpaths, 1 connecting subpath Q_c, and a set S of edgewise non-overlapping subpaths connecting the binary subpaths and the connecting subpath, each with an isolated optimal solution of size 0 which does not block any isolated optimal solution of any of the subpaths in Q. Then Q has two isolated optimal solutions.*

Since the size of the isolated optimal solution of Q is $2x + 2$, all x copies of the binary subpath as well as the connecting subpath must have been solved according to one of their optimal solutions. There are two ways of solving all these subpaths according to their isolated optimal solutions: all subpaths are 'true' (except a possible negating subpath) or all subpaths are 'false' (except a possible negating subpath). Thus, Q has two isolated optimal solutions: true and false.

5.3 Clause gadget

The clause gadget is a single subpath B that has three isolated optimal solutions. The clause gadget interacts with propagation gadgets through shared vertices, such that the size of a non-isolated optimal solution of B is equal to the size of an isolated optimal solution of B if and only if at least one of the propagation gadgets is 'true'. The clause gadget consists of the single subpath shown in Figure 5.22.

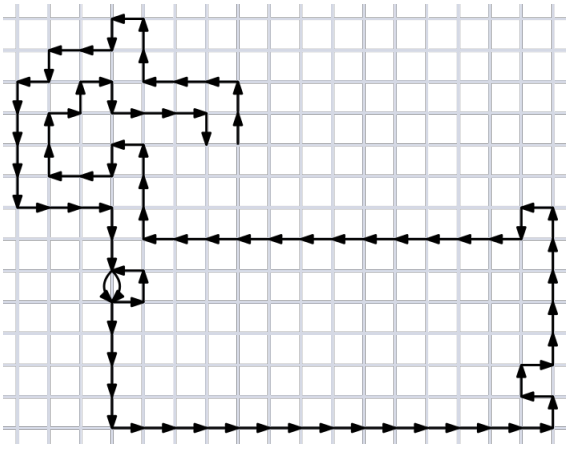


Figure 5.22: The clause gadget, drawn on an integer grid.

Isolated optimal solutions of the clause gadget. The clause gadget has three isolated optimal solutions of size 2, as shown in Figure 5.23. These

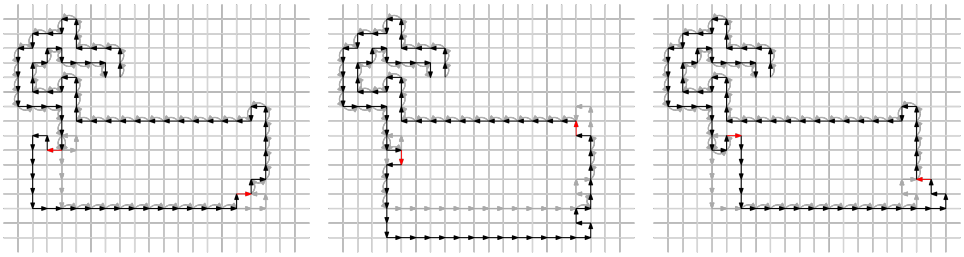


Figure 5.23: The three isolated optimal solutions of the clause gadget. Flipped edges are indicated in red, the original path B is indicated in grey.

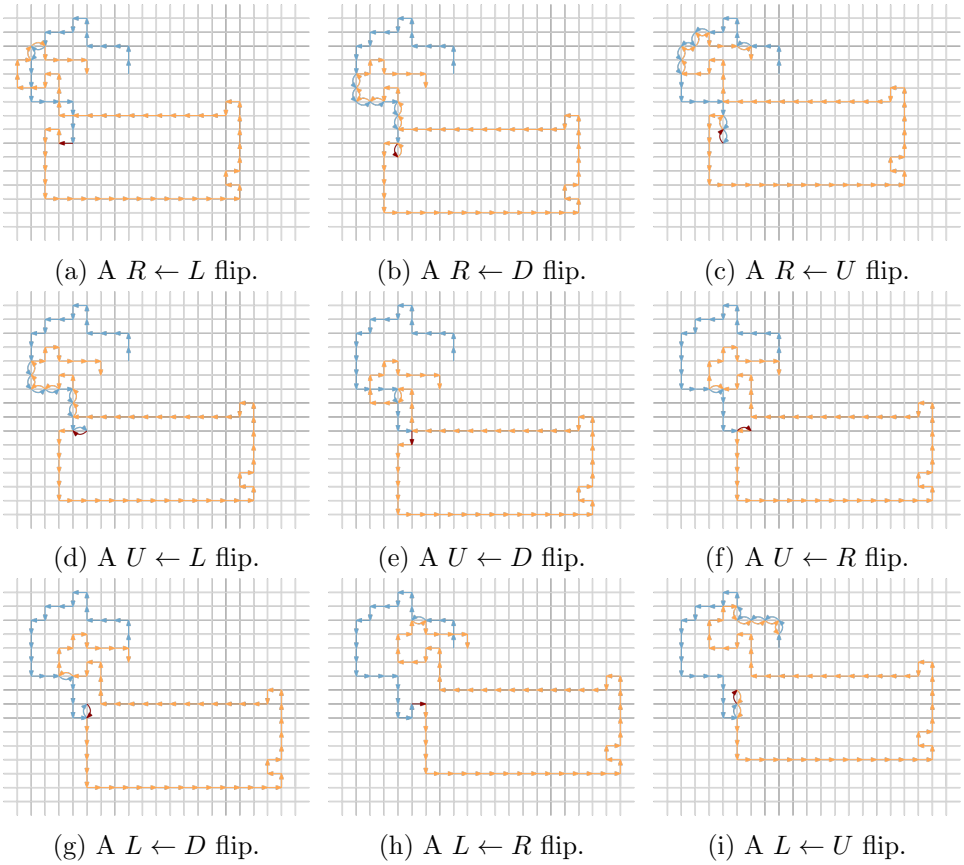
solutions all consist of compensating edge flips, and as such the ingoing path and the outgoing path have the same relative position. This is necessary because of the shape of the 'tail' of the path, which is such that after a single edge flip, the pre-flip and post-flip path are entangled in such a way that they can only be untangled by shifting the post-flip path back to its original position.

We can reason that these solutions are indeed the only three isolated optimal solutions as follows. The clause gadget contains two overlapping cycles that share three edges. Thus, if we want to solve both cycles with a single edge flip, we have three candidate edges to choose from. Figure 5.24 shows all paths B' that can be obtained by flipping any of these three edges in any direction.

As is visible in Figure 5.24, if we flip any of these three edges in any direction we cause a new cycle within the subpath, indicating that we need at least two edge flips to solve this subpath. Of all paths B' that can be obtained by flipping any of the three edges in any direction, we can solve exactly three with a single edge flip. Each of these three paths B' has only one solution of size one, resulting in three optimal solutions of size two for the input path B.

Now we have to verify that these three solutions are the only solutions of size 2. Since we already tried all possible solutions where we flip any of the shared edges of the cycles in any direction, any remaining isolated optimal solution does not flip any of these shared edges. Thus, if there is an isolated optimal solution left, it has to flip both of the edges that are in exactly one cycle in the input path. Figure 5.25 shows all paths B' that can be obtained by flipping these two edges in any combination of directions.

As is visible in Figure 5.25, no matter in which combination of directions we flip these edges, the path B' remains unsolved. This indicates that there are no optimal solutions left. Thus, the clause gadget has three isolated optimal solutions.



(a) A $R \leftarrow L$ flip. (b) A $R \leftarrow D$ flip. (c) A $R \leftarrow U$ flip.

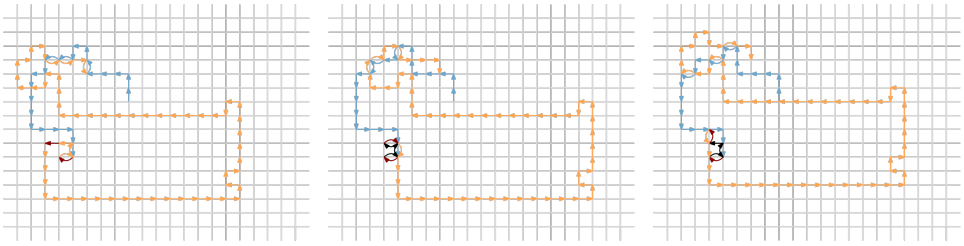
(d) A $U \leftarrow L$ flip. (e) A $U \leftarrow D$ flip. (f) A $U \leftarrow R$ flip.

(g) A $L \leftarrow D$ flip. (h) A $L \leftarrow R$ flip. (i) A $L \leftarrow U$ flip.

Figure 5.24: All paths B' that can be obtained by flipping one of the three shared edges in any direction. For clarity, the flipped edge is indicated in dark red; the pre-flip path is indicated in blue; and the post-flip path is indicated in orange. Paths 5.24a, 5.24e, and 5.24h lead to an optimal solution.

Connection to propagation gadgets. We connect the propagation gadgets to the clause gadget in such a way that each propagation gadget blocks one of the optimal solutions of the clause gadget if and only if the propagation gadget is false. Thus, if all three propagation gadgets are false (and thus the corresponding literals of the clause), the clause gadget subpath cannot be solved according to one of its isolated optimal solutions. This is illustrated in Figure 5.26. The three optimal solutions are indicated in different shades of red; the gadget either expands to the right, to the left, or down. Note that we can easily change the direction of propagation gadgets (in a similar way as how we made the c-shape) such that all three propagation gadgets go down to a variable gadget.

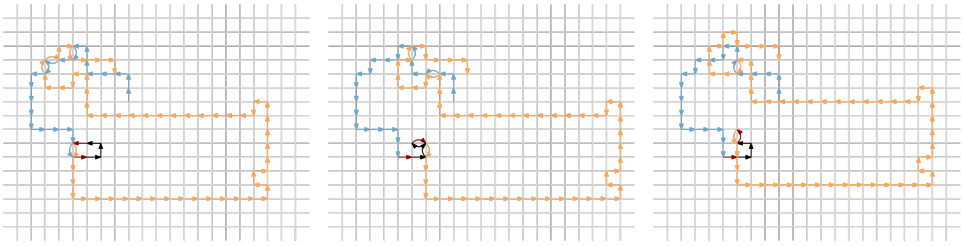
Each orange solution of a connecting subpath shares vertices with a potential optimal solution of the clause gadget. No blue solution interferes



(a) A $D \leftarrow L$ flip and a $D \leftarrow L$ flip.

(b) A $D \leftarrow L$ flip and a $U \leftarrow R$ flip.

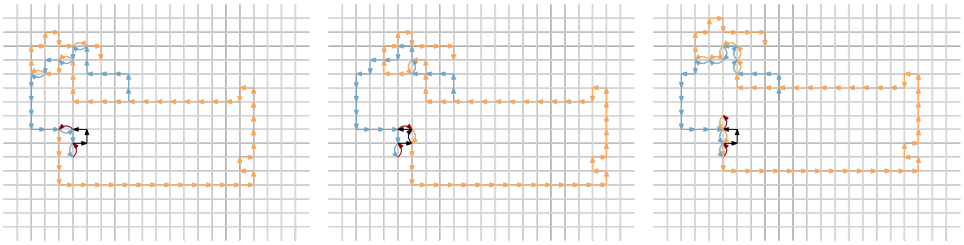
(c) A $D \leftarrow L$ flip and a $D \leftarrow U$ flip.



(d) A $D \leftarrow R$ flip and a $D \leftarrow L$ flip.

(e) A $D \leftarrow R$ flip and a $D \leftarrow R$ flip.

(f) A $D \leftarrow R$ flip and a $D \leftarrow U$ flip.



(g) A $D \leftarrow U$ flip and a $D \leftarrow L$ flip.

(h) A $D \leftarrow U$ flip and a $D \leftarrow R$ flip.

(i) A $D \leftarrow U$ flip and a $D \leftarrow U$ flip.

Figure 5.25: All paths B' that can be obtained by flipping both edges that are in exactly one cycle in any combination of directions. For clarity, the flipped edges are indicated in dark red; the path between the flipped edges is indicated in black; the pre-flip path is indicated in blue; and the post-flip path is indicated in orange.

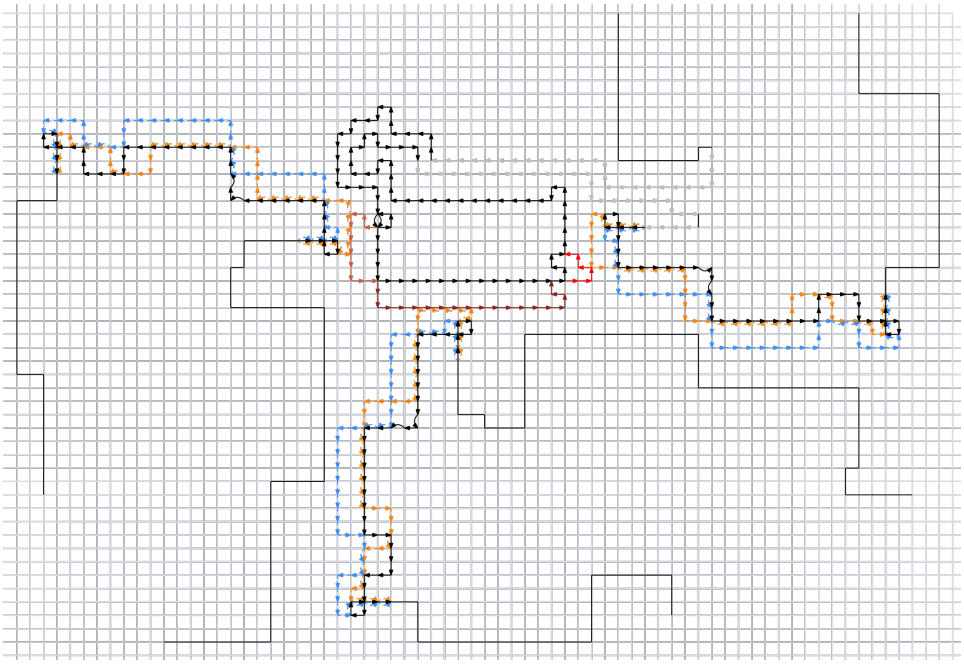


Figure 5.26: Three propagation gadgets connected to a clause gadget. The three different solutions of the clause gadget are indicated in different shades of red, the true and false solutions of the connecting subpaths of the propagation gadgets are indicated in blue and orange. The subpaths that connect the propagation gadget to the clause gadget are indicated in grey. The propagation gadgets are schematized to their border, indicated with a black line.

with the clause gadget. Different propagation gadgets are placed such that neither optimal solution of one propagation gadget shares vertices with an optimal solution of another propagation gadget. This is possible due to the width of the clause gadget, which provides room for the middle propagation gadget to bend down before interfering with the right propagation gadget.

5.4 Final construction

In this section we describe how to place all gadgets to obtain a complete path P based on the structure given by the embedded formula F. Our main concern is to ensure that none of the subpaths overlap. Furthermore, we need to ensure that we connect all subpaths, since our goal is to obtain a complete path P.

Variable gadgets. We start by constructing the variable gadgets. For this purpose, we need to know how wide these gadgets need to be, which depends on the number of propagation gadgets that need to be attached to the variable gadget, which, in turn, depends on the number of occurrences of the variable in clauses. In theory, we need 1 triplet per occurrence, since a propagation gadget can be attached to a single triplet (see Figure 5.11). However, practically, if we connect a propagation gadget to each triplet in a variable gadget, our propagation gadgets will overlap.

To be able to put proper space between propagation gadgets, we need to know how wide propagation gadgets can be. The widest point of a propagation gadget is the c-bend, which is 51 edges wide. The subpath which forms the connection point between the propagation gadget and the variable gadget is 6 edges wide. Therefore, the propagation gadget will need 45 edges next to the variable gadget. If we place two gadgets with their c-shape curling towards each other, we need 91 edges between two connection points to make sure they do not collide. Because of the width of a triplet, there are 10 edges between two consecutive possible connection points. Thus, if we connect a first propagation gadget to our first possible connection point, we need to skip 7 connection points: the first and eighth possible connection points are 106 edges apart, which is enough to make sure that the c-shapes of propagation paths do not collide, even if they curl towards each other.

As can be seen in Figure 5.11, we have connection points on top of the variable gadget and on the bottom. Using the first connection point on the top does not withhold us from also using the first connection point on the bottom. Let top be the total number of propagation gadgets we need to connect to a top connection point of the variable gadget, and bot the total number of propagation gadget we need to connect to a bottom connection point of the variable gadget. We need at least $top + (top - 1) * 6$ top triplets and $bot + (bot - 1) * 6$ bottom triplets. Since the number of top triplets can differ with the number of bottom triplets with at most one, if $top < (bot - 1)$ we set top to $bot - 1$, and if $bot < (top - 1)$ we set bot to $top - 1$.

After constructing the variable gadgets with the required number of top and bottom triplets, we can place them. We place them on a horizontal line with a distance of 91 between them. This distance is to once again make sure c-shapes of neighbouring propagation gadgets do not collide. The order of the variables is given by the embedding of F.

Clause gadgets. Now we place the clause gadgets. We start on the top side, then repeat the same process for the bottom side. We give each clause gadget a number a: 1 for the one corresponding to the lowest clause in the embedding of F, 2 for the second lowest, until we reach the n^{th} clause on the top side, which clause gadgets gets the number n.

Propagation gadgets need to be at least 113 edges long. We need 3 edges between a propagation gadget and a clause gadget; thus, we need to place a clause gadget at least 116 edges above a variable gadget. We place each clause gadget a distance of $116a$ above the connection point of the middle propagation gadget.

The horizontal placement of the clause gadget depends on the location of the connection point of the middle propagation gadget, if the literal used in the corresponding clause is the negated variable or not, and if the connection point used is located on the top side of the variable gadget or the bottom side.

If we place the clause gadget on the top side of the variable gadget, the clause gadget is located somewhat to the left of the connection point of the middle propagation gadget.

If the literal corresponding to the middle propagation gadget of the clause is the non-negated version of the variable and the connection point we use is located on the top of the variable gadget, we place the clause gadget such that there is a space of three horizontal edges between the right side of the clause gadget and the left side of the shared subpath.

If the literal corresponding to the middle propagation gadget of the clause is the negated version of the variable and the connection point we use is located on the top of the variable gadget, we place the clause gadget such that there is a space of 4 horizontal edges between the right side of the clause gadget and the right side of the subpath.

If we place the clause gadget on the bottom side of the variable gadget, the clause gadget is located somewhat to the right of the connection point of the middle propagation gadget.

If the literal corresponding to the middle propagation gadget of the clause is the negated version of the variable and the connection point we use is located on the bottom of the variable gadget, we place the clause gadget such that there is a space of three horizontal edges between the left side of the clause gadget and the right side of the shared subpath.

If the literal corresponding to the middle propagation gadget of the clause is the non-negated version of the variable and the connection point we use is located on the top of the variable gadget, we place the clause gadget such that there is a space of 4 horizontal edges between the left side of the clause gadget and the left side of the subpath.

Propagation gadgets. Now we place the propagation gadgets. We use a similar construction below the variable gadgets as above but rotated 180 degrees. We use the appropriate versions of the propagation gadgets for the left, lower, and right sides of each clause gadget. For each propagation gadget, we construct it such that it is the required length and shape, and we place the top side of these versions of the propagation gadgets as shown in Figure 5.26. The lower side is connected to the appropriate connection point.

Connecting all subpaths. Now we need to connect all subpaths to obtain a complete path P. We start at the rightmost variable gadget. Each binary subpath within this gadget gets connected to its counterclockwise neighbouring subpath as shown in Figure 5.9. Once a top propagation gadget is reached, we diverge from the grey path shown in Figure 5.9 and follow the orange path instead, until we have reached the next binary subpath, which is part of the propagation gadget.

We connect the rest of the propagation gadget as normally, drawing subpaths that do not interfere with the isolated optimal solutions of any binary subpaths between each binary subpath and its clockwise neighbour, as shown in Figure 5.16. If the propagation gadget is the rightmost version of the propagation gadget, when reaching the second to last binary subpath before the connecting subpath, instead of connecting this subpath to the last binary subpath before the connecting subpath, we connect it to the clause gadget as shown in Figure 5.26. We then connect the clause gadget to the last binary subpath before the connecting subpath, which we connect to the connecting subpath, which we connect to the binary subpath that is its counterclockwise neighbour, after which we continue our process.

If the propagation gadget is the middle or left version of the propagation gadget, we connect our second to last binary subpath before the connecting subpath to our last binary subpath before the connecting subpath, connect the last binary subpath before the connecting subpath to the connecting subpath, and connect the connecting subpath to its counterclockwise neighbouring binary subpath as shown in Figure 5.16, after which we continue as normal.

Once we traversed the entire top half of the variable gadget, we diverge from the path shown in Figure 5.9 again, this time to follow the blue path. We draw a subpath to the next variable gadget and continue our process. The bottom half is connected similarly. A schematization of the final result is shown in Figure 5.27.

5.4.1 Final proof

After combining the gadgets as described, and adding a path between variables to make sure that the entire path is connected, we have obtained a

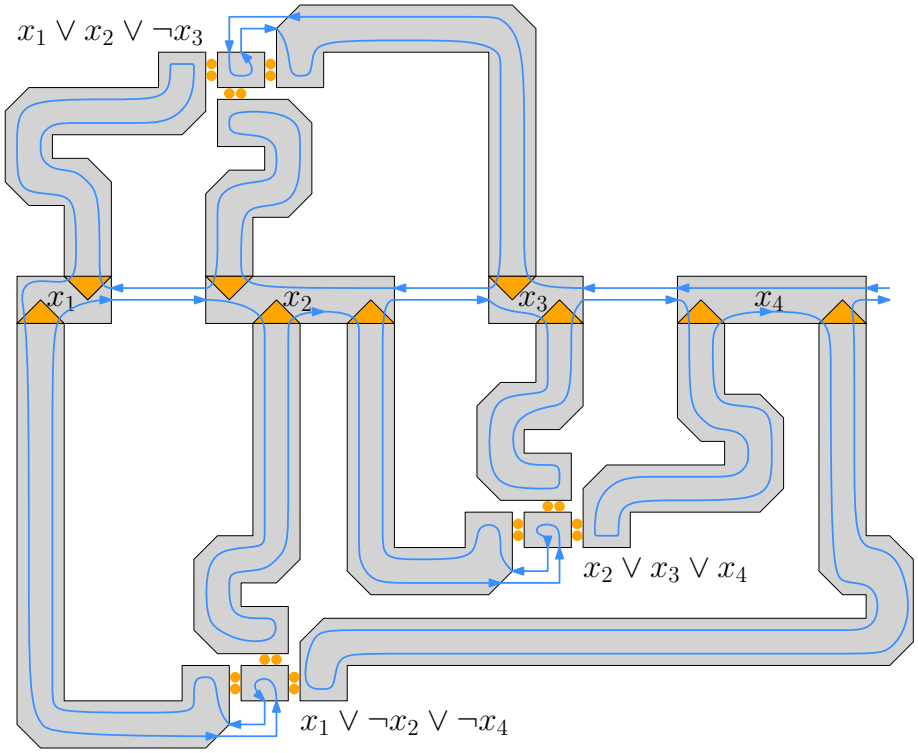


Figure 5.27: A schematization of how the construction shown in Figure 5.1 is connected. Blue lines are an approximation of the final path P.

complete path P. This path P consists of a number of subpaths with isolated optimal solutions:

- We have a set X of x binary subpaths with isolated optimal solutions true and false, of size 2.
- Let C_f be the number of clauses in F. We have a set Y of $3 * C_f$ connecting subpaths, with isolated optimal solutions true and false, of size 2.
- We have a set Z of C_f clause gadget subpaths, each with 3 optimal isolated solutions of size 2.
- We have a set S of edgewise non-overlapping subpaths that connect all other subpaths. Each of these subpaths has an isolated optimal solution of size 0 that does not interfere with an isolated optimal solution of any of the other subpaths.

These subpaths form a subpath decomposition $W = X \cup Y \cup Z \cup S$ of P. According to Corollary 2.1, a lower bound for the optimal solution of P is the sum of the isolated optimal solutions of each path $P_i \in W$. Thus, a

lower bound for the optimal solution of P is $2x+6*C_f+2*C_f = 2x+8*C_f$. According to Observation 5, this lower bound can be achieved if and only if all subpaths in W can be solved according to one of their isolated optimal solutions. We will now define $k = 2x + 8C_f$.

Now if the 3SAT formula F belonging to the path P is satisfiable, there is some assignment of values to the variables of F such that each clause has at least one true literal. This means that there is a solution for P such that each variable gadget is solved as either true or false, each propagation gadget is either true or false, and each clause gadget in our path P has at least one unblocked isolated optimal solution. As such, each clause gadget has room to be solved optimally within P. Since the true and false solutions for the variable and propagation subpaths were also isolated optimal solutions, the size of the optimal solution of path P is equal to the sum of the isolated optimal solutions of the variable, propagation, and clause subpaths, which is equal to k.

On the other hand, if P has an optimal solution of size k, we must have solved each variable, clause, and propagation subpath with one of its isolated optimal solutions: as stated in Observation 2, the size of a non-isolated optimal solution for a subpath P_i is only equal to the size of an isolated optimal solution of P_i if the non-isolated optimal solution is also an isolated optimal solution. This means that for each clause gadget, at least one of its isolated optimal solutions is unblocked. This in turn means that for each clause gadget, at least one of its literals is true, which means that there is some assignment of values to the variables of F such that it is true, which means that F is satisfiable.

Chapter 6

Conclusion

In this thesis we studied the problem of embedding paths with directional constraints on a grid. We started by identifying and proving properties of the input path: we proved that the size of the isolated optimal solution of a subpath P_i of P is always smaller than or equal to the size of its non-isolated optimal solution, and that the optimal solution of a path P depends on the sizes of the isolated optimal solutions of its non-overlapping subpaths. We also explored constructions in paths: types of cycles and their solving techniques, and spirals, which complicate solving a path. Furthermore, we proved that a lower bound for the solution of an input path P of length n is the size of the largest set of non-overlapping cycles of P, and that an upper bound for the solution of P in terms of the path length n is $min(max(n/4 + h, n/4 + v), n/2)$, where h is the size of the largest set of non-overlapping cycles after removing a vertical direction, and v is the size of the largest set of non-overlapping cycles after removing a horizontal direction. This upper bound is $min(\#A + v, \#B + h)$ when expressed in terms of the least occurring horizontal direction A and the least occurring vertical direction B.

In Chapter 4 we proposed an exact algorithm that runs in $O(3^{k+1}n^{k+2})$ time, where k is the size of the optimal solution of the input path P, and n is the length of P. Although we provided some arguments why the algorithm might run faster in practice, the running time still leaves room for improvement. One possible improvement would be to somehow use the information we gain from a failed run. For example, if we run FindSolutionOfSize(s) with $s = LowerBound$, and the algorithm only returns false after solving the x^{th} cycle, we know that we can solve the first x cycles with x edges. Thus, on the next iteration, it might make sense to wait with increasing the size of the solution we want to find until we encounter the $i+1^{th}$ cycle. This would prevent the algorithm from trying edge flips that do not decrease the cycle set for cycles we know can be solved with a single edge flip, which might drastically improve the running time. However, this requires some care, as an optimal solution for a cycle does not necessarily lead to an optimal

solution of the entire path.

Finally, we proved the NP-hardness of the problem in Chapter 5 by providing a reduction from planar 3-SAT. That the problem of drawing a path with directional constraints is NP-hard means that the problem of drawing a graph with directional constraints on an integer grid is also NP-hard, as a path is a class of graphs. This implies that a good direction for future research would be to find an approximation algorithm for the problem, as due to asymptotically large running times, exact algorithms will be unsuitable for large input.

One possible direction for finding an approximation algorithm might be to smartly use the buffering technique as explained in Section 3.3.1. After all, buffering is very similar to the optimal solving techniques for loops and some spirals. On top of that, the configuration of our subpaths already gives us considerable knowledge of the configuration of the entire path. After all, we know that our direction A occurs least, and we stop our subpaths once we have encountered as much $\bar{B}$ as B. We also know that we do not have to flip any occurrence of A in a subpath if we have encountered more edges with constraint $\bar{A}$ than A. Thus, if we have to flip edges, we know that we have less than or equal occurrences of $\bar{A}$ than A, and less than or equal occurrences of B to $\bar{B}$. This knowledge might be useful to determine what kind of constructions a path contains, which in turn can be useful to determine which strategy to apply on different parts of the path.

It would also be interesting to see how the upper bound is affected if we only buffer occurrences of A when they are part of a cycle at any point in our buffering process. Figure 6.1 shows how the path P shown in Figure 3.9 might be solved if we use this tactic.

As can be seen in Figure 6.1, only buffering occurrences of A that are part of a cycle at any point in our buffering process reduces the number of edge flips to 9, which is very close to the optimal number of edge flips as shown in Figure 3.9b.

However, even though buffering is very similar to optimal solving techniques for loops and some spirals and might lead to close-to-optimal solutions in some cases, it is far from optimal for regular cycles and spirals for which compensation is possible. Thus, before we can use it as an approximation technique, we will need to find a way to smartly deal with and distinguish between these kinds of constructions.

Another possible direction for finding an approximation algorithm is to determine if there is some class of paths for which the problem is not NP-hard, for example non-looping paths or paths with only bidirectional cycles. If the problem is not NP-hard for certain classes of paths, a fast exact algorithm can be found for these classes of paths. It might then be possible to find an approximation algorithm for the problem of embedding a path with directional constraints on a grid by flipping a minimal number of edges to convert the input path to a class of paths for which a fast exact algorithm

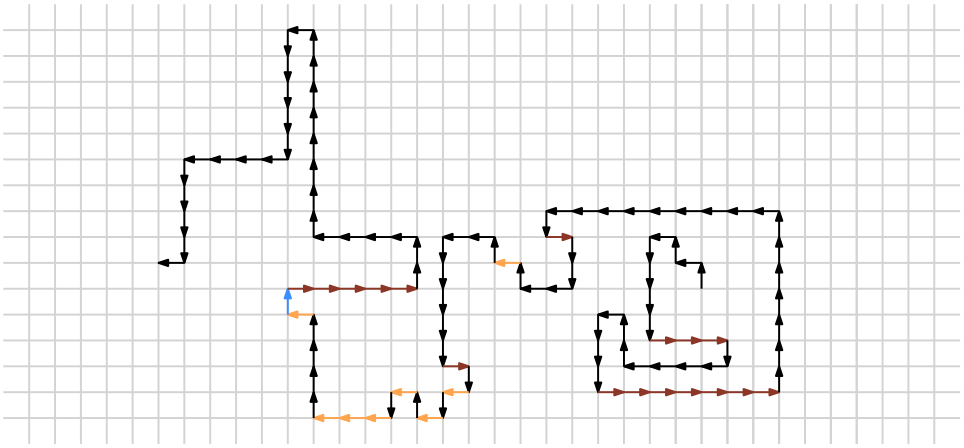


Figure 6.1: An example of how the path P shown in Figure 3.9 might be solved if we only buffer occurrences of R that are at some point part of a cycle. Edges that are flipped from R to L are indicated in orange. Edges that are flipped from R to U or D are indicated in blue. Edges in direction R that remain unflipped are indicated in dark red.

exists and solving the problem for the converted path.

Another interesting direction for future research would be to find out how the difficulties of the problem translate if we would increase or decrease the number of directional constraints. If we limit the number of directional constraints to two opposing ones, for example, the problem is not NP-hard anymore: the optimal solution will always consist of flipping the direction that occurs least. With three pairs of directional constraints that go in a triangle for a triangular grid, the problem might still be NP-hard, as loops might still exist. On the other hand, the extra directions might make it easier to find solutions.

Bibliography

[1] Ulrik Brandes and Barbara Pampel. Orthogonal-ordering constraints are tough. *Journal of Graph Algorithms and Applications*, 17(1):1–10, 2013.

[2] K. Buchin, W. Meulemans, and B. Speckmann. Area-preserving c-oriented schematization. pages 163–166, 2011. 27th European Workshop on Computational Geometry (EuroCG 2011), EuroCG ; Conference date: 28-03-2011 Through 30-03-2011.

[3] Kevin Buchin, Wouter Meulemans, André Van Renssen, and Bettina Speckmann. Area-preserving simplification and schematization of polygonal subdivisions. *ACM Trans. Spatial Algorithms Syst.*, 2(1), apr 2016.

[4] Hsien-Chih Chang, Jeff Erickson, and Chao Xu. Detecting weakly simple polygons. *CoRR*, abs/1407.3340, 2014.

[5] Daniel Delling, Andreas Gemsa, Martin Nöllenburg, and Thomas Pajor. Path schematization for route sketches. In Haim Kaplan, editor, *Algorithm Theory - SWAT 2010*, pages 285–296, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[6] Daniel Delling, Andreas Gemsa, Martin Nöllenburg, Thomas Pajor, and Ignaz Rutter. On d-regular schematization of embedded paths. *Computational Geometry*, 47(3, Part A):381–406, 2014.

[7] DAVID H DOUGLAS and THOMAS K PEUCKER. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, 1973.

[8] David Eppstein, Marc van Kreveld, Bettina Speckmann, and Frank Staals. Improved grid map layout by point set matching. In *2013 IEEE Pacific Visualization Symposium (PacificVis)*, pages 25–32, 2013.

[9] Graham McNeill and Scott A. Hale. Generating tile maps. *Computer Graphics Forum*, 36(3):435–445, 2017.

[10] Damian Merrick and Joachim Gudmundsson. Path simplification for metro map layout. In Michael Kaufmann and Dorothea Wagner, editors, *Graph Drawing*, pages 258–269, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[11] Wouter Meulemans, Max Sondag, and Bettina Speckmann. A simple pipeline for coherent grid maps. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):1236–1246, 2021.

[12] Gabriele Neyer. Line simplification with restricted orientations. In Frank Dehne, Jörg-Rüdiger Sack, Arvind Gupta, and Roberto Tamassia, editors, *Algorithms and Data Structures*, pages 13–24, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[13] Martin Nöllenburg. Automated drawing of metro maps. Technical Report 25, Universität Karlsruhe (TH), 2005.

[14] Jochen Schiewe. Distortion effects in equal area unit maps. *KN - Journal of Cartography and Geographic Information*, 71(1):71–82, jun 2021.

[15] T. Shaw. Good data visualization practice: Tile grid maps. `https://forumone.com/ideas/good-data-visualization-practice-tile-grid-maps-0`, apr 2016. Accessed April 2023.

[16] Kevin Verbeek. Homotopic $\mathcal{C}$-oriented routing. In Walter Didimo and Maurizio Patrignani, editors, *Graph Drawing*, pages 272–278, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[17] Alexander Wolff. Drawing subway maps: A survey. *Informatik - Forschung und Entwicklung*, 22(1):23–44, 2007.

[18] K. Wongsuphasawat. Whose grid map is better? quality metrics for grid map layouts. `https://kristw.medium.com/whose-grid-map-is-better-quality-metrics-for-grid-map-layouts-e3d6075d9e80`, jan 2016. Accessed April 2023.

[19] R. Zachary. Data visualization strategies using tile grid maps. `https://www.gislounge.com/data-visualization-strategies-using-tile-grid-maps/`, nov 2015. Accessed April 2023.