MASTER

Non-crossing paths with fixed endpoints

Verbeek, K.A.B.

Award date:
2008

technische universiteit eindhoven
Department of Mathematics and Computer Science

Master's Thesis

Non-crossing paths with fixed endpoints

by
Kevin Verbeek

Supervisor
dr. B. Speckmann

Eindhoven, October 3, 2008

Abstract

Given a collection of n rectilinear paths with fixed endpoints and a total of k links and m rectangular obstacles, the problem is to find n non-crossing paths that are homotopic to, and use the same endpoints as, the input paths and have the minimum total number of links. We present a 2-approximation for this problem that runs in $O((m+n)k)$ time and uses $O((m+n)k)$ storage. We also consider a variant of the problem where endpoints are restricted to a region instead of being fixed. We prove that if endpoints are restricted to regions, it is NP-Complete to decide if every path can be drawn with only one link. We do however drop the restriction on the homotopy classes for this variant of the problem.

Contents

Chapter 1

Introduction

For a long time maps have been used to convey information to the general public. For example, maps containing road networks like in Figure 1.1 can be used to find the route between two locations on the map. But as these maps became more and more complex and tried to convey more information, the exact geographical shapes of road networks distracted the user from important information. That is why the concept of schematic maps was created. Although maps are generally used to convey some sort of geographical information, not all of the exact details are necessary to provide the required information. So for road networks the exact shape of the roads is not that important, but the connections are. In a schematic map it is therefore allowed to simplify the shape of the roads while keeping the connections correct. Harry Beck was one of the first to use schematic maps or diagrams when he designed the first schematic representation of London's tube map (subway) as shown in Figure 1.2. This schematic map did not use the exact geographical locations of the stations and the railway as this information was not important for the travelers. Instead it showed a simplified representation of the connections between the stations. Due to the simplification, this schematic map was much easier to understand and therefore became very popular.

Nowadays we use more and more schematic maps to convey different kinds of information. Schematic maps were usually created by cartographers, which was a very laborious task. In the current digital era, computers can be used to simplify or completely take over the task of

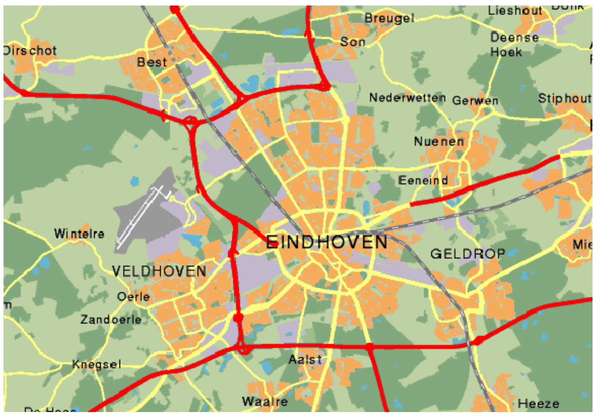

Figure 1.1: Road network near Eindhoven

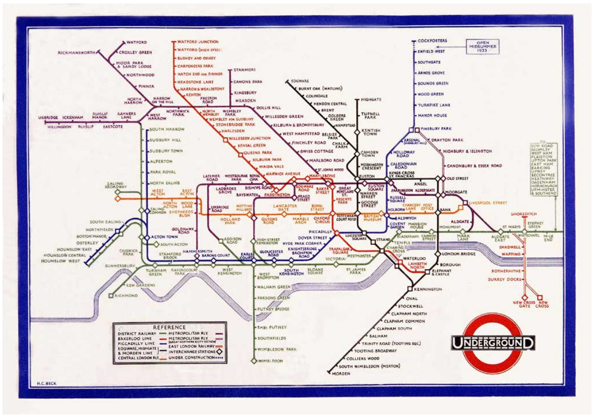

Figure 1.2: Schematic map of London's subway by Harry Beck

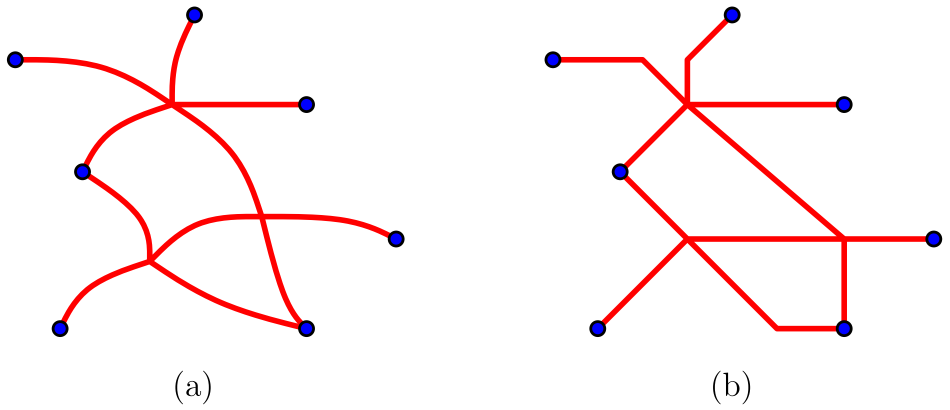

Figure 1.3: A road network (a) and its schematic representation (b).

creating schematic maps. This is part of the field known as automated cartography. The field of automated cartography is about all tasks regarding designing and drawing different kinds of maps (tasks usually performed by cartographers) using the computer. It involves tasks like cartographic generalization, label placement and as mentioned map schematization.

In this thesis we consider a problem which can be encountered when creating such schematic maps. For schematic maps it is often useful that the relevant locations like stations or cities are at least close to their real geographical location. Assume the relevant location are cities. Then the problem that we consider is that of how to create a simplified representation of the connections between cities with the cities at a fixed position (like their exact geographical location). For example, we would like to simplify a road network as shown in Figure 1.3(a). A possible solution for this example is shown in Figure 1.3(b). A road network can have a complex structure though, so as an initial attempt we might want to simplify the problem to be on connections with a simpler structure. Therefore we will assume that every city is connected to exactly one other city and that these connections are not crossing each other (see Figure 1.4(a)). A valid solution to this problem would be a simplified representation of the connections without any crossings, because allowing crossings would make the map harder to understand. Also the simplified connections should pass the other cities on the same side

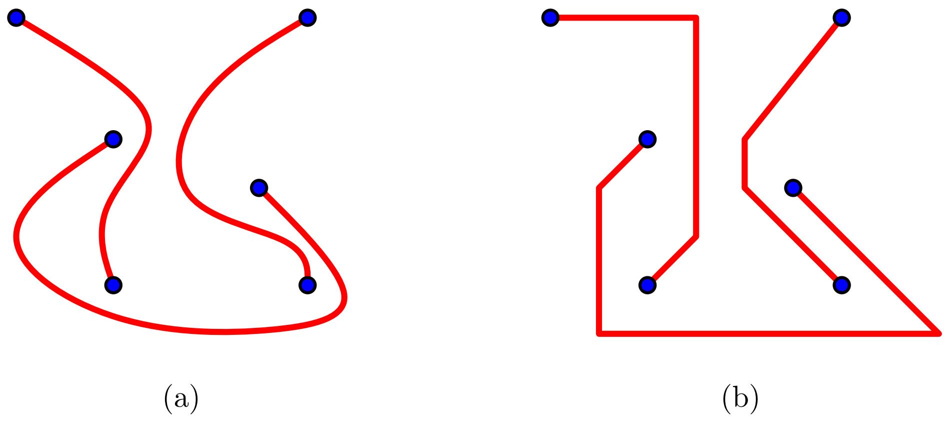

Figure 1.4: Cities with non-crossing connections (a) and its schematic representation (b).

as the original connections to make connections easier to recognize. A valid solution is shown in Figure 1.4(b).

The problem described above also turns out to be an important problem in the field of chip design or VLSI design. When designing chips, the modules on these chips need to be connected to other modules using wires. Obviously these wires should not cross each other. So the problem considered in this thesis also has an application in the field of VLSI design.

Before we discuss related work, we give formal definitions of the problems considered in this thesis in the next chapter.

1.1 Problem definitions

In this chapter we give formal problem definitions for the problems considered in this thesis. Before this can be done, some definitions must be introduced. Although many of these definitions are common, we include these definitions anyway for the sake of completeness.

Paths. In this chapter we will look at the formal definition of a path and at definitions relating to paths that are relevant in this thesis. We restrict the discussion to paths in two dimensions. A path in two dimensions can be formally defined as a function $\pi : [0, 1] \to \mathbb{R}^2$. In this definition the endpoints of the path are $\pi(0)$ and $\pi(1)$. We can also separate the function for the different coordinates resulting in $\pi_x : [0, 1] \to \mathbb{R}$ and $\pi_y : [0, 1] \to \mathbb{R}$. So we get that $\pi(t) = (\pi_x(t), \pi_y(t))$ for $0 \leqslant t \leqslant 1$. Now we can define monotone paths (see Figure 1.5(b)).

Definition 1 *A path π is called* x-monotone *(*y-monotone*) if the function π_x (π_y) is monotone.*

An alternative definition for monotone paths is as follows. A path is called x-monotone (y-monotone) if every line orthogonal to the x-direction (y-direction) can intersect the path at most once. This is also exactly the property of monotone paths that we will use.

In a geometric setting, the above formal definition of a path is often too general and hence we usually consider only polygonal paths (see Figure 1.5(c)). Polygonal paths consist of a connected series of straight line segments. These paths can be defined by the endpoints of the straight line segments. We call the straight line segments of a path the *links* of a path. The endpoints of the straight line segments of a path are called the *bends* of a path, except for the

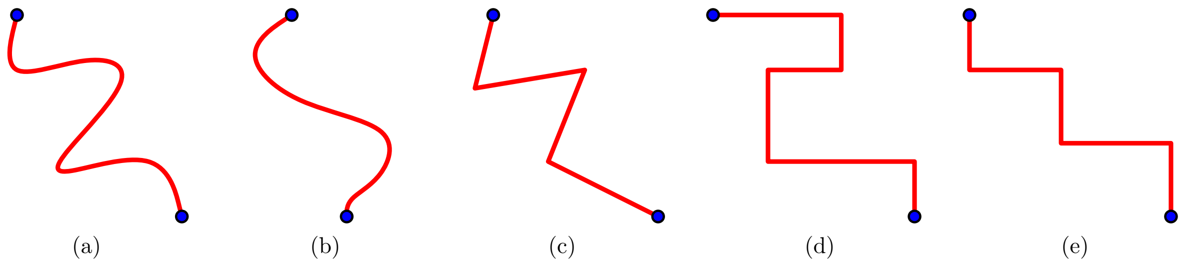

Figure 1.5: A general path (a), a y-monotone path (b), a polygonal path (c), a rectilinear path (d) and a positive staircase path (e).

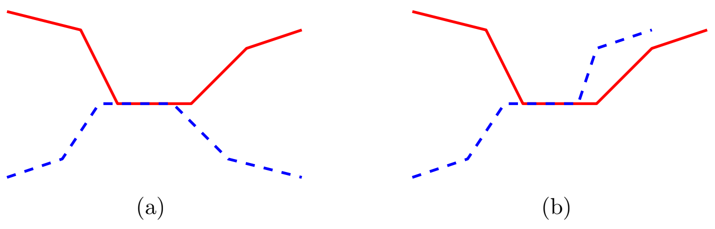

Figure 1.6: Two non-crossing paths (a) and two crossing paths (b).

endpoints of the path itself. Now the complexity of a path can be defined as the number of links of a path. If a path has the minimum number of links (under some constraints) then we call this path a *minimum-link path*. The length of a path can be computed by summing up the lengths of the links. The path with minimum length (under some constraints) is simply called the *shortest path*.

Although it is relatively easy to work with polygonal paths, for some applications it is better to have paths where the links can have only a limited number of directions. This is for example the case in VLSI design and cartography (schematic maps). Paths where each link can have only one of c directions are called *c-oriented paths*. In particular paths with only horizontal or vertical links are called *rectilinear paths* (see Figure 1.5(d)). In the main part of this thesis we will consider only rectilinear paths. Rectilinear paths that are both x-monotone and y-monotone are called *staircase paths*. In general there are four types of staircase paths. Staircase paths can be of type $(+x, +y), (+x, -y), (-x, +y)$ or $(-x, -y)$, where $+x$ is with increasing x-coordinate and $-x$ is with decreasing x-coordinate (similar for $+y$ and $-y$). However if the direction of the path is irrelevant, we get only two types of staircase paths, because then $(+x, +y)$ and $(-x, -y)$ are the same and $(+x, -y)$ and $(-x, +y)$ are the same. We call the staircase paths with increasing x-coordinate *positive staircase paths* (see Figure 1.5(e)) and the staircase paths with decreasing x-coordinate *negative staircase paths*.

Finally we define when two paths are considered non-crossing. A common definition is that of disjoint paths. Two paths π_1 and π_2 are disjoint if $\pi_1(t_1) \neq \pi_2(t_2)$ for all $0 \leqslant t_1, t_2 \leqslant 1$. However this definition is too strong in our case. We actually require the following.

Definition 2 *Two paths π_1 and π_2 are* non-crossing *if for every $\epsilon > 0$ we can move the bends of π_1 and π_2 by at most ϵ such that the resulting paths are disjoint.*

Intuitively this means that the paths are allowed to overlap, but only in such a way that the parts of the links that overlap can be pushed away from each other to make the paths disjoint (see Figure 1.6). Note that this definition of non-crossing can also be applied to a single path in that the path does not cross itself. Also, if we have n paths that are pairwise non-crossing and every path is non-crossing itself, then we say that these n paths are non-crossing.

Homotopy classes. A homotopic relation is defined with regard to a set $\mathcal{B}$. We call this set the *blocker set*. The set $\mathcal{B}$ always consists of a collection of obstacles. In this thesis $\mathcal{B}$ also contains the endpoints of the paths. We therefore use the term *blockers* to denote both obstacles and endpoints. No path is allowed to contain a point in $\mathcal{B}$, except for its own endpoints. The homotopic relation among paths w.r.t. $\mathcal{B}$ is commonly defined as follows.

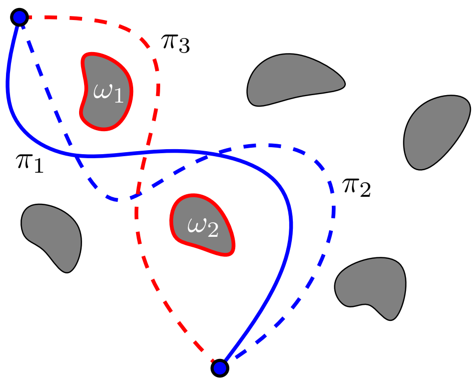

Figure 1.7: The path π_1 is homotopic to the dashed path π_2, but not to the dashed path π_3.

Definition 3 *Two paths $\pi_1, \pi_2 : [0,1] \to \mathbb{R}^2$ with the same endpoints are called* homotopic *(notation $\pi_1 \sim_h \pi_2$) w.r.t. $\mathcal{B}$ if there exists a continuous function $\Gamma : [0,1] \times [0,1] \to \mathbb{R}^2$ with the following properties:*

- $\Gamma(0,t) = \pi_1(t)$ *and* $\Gamma(1,t) = \pi_2(t)$ *for* $0 \leqslant t \leqslant 1$.
- $\Gamma(s,0) = \pi_1(0) = \pi_2(0)$ *and* $\Gamma(s,1) = \pi_1(1) = \pi_2(1)$ *for* $0 \leqslant s \leqslant 1$.
- $\Gamma(s,t) \notin \mathcal{B}$ *for* $0 \leqslant s \leqslant 1$ *and* $0 < t < 1$.

Intuitively, the homotopic relation can be thought of as follows. Two paths π_1 and π_2 are homotopic if we can deform π_1 into π_2 without ever pushing π_1 through a point in $\mathcal{B}$. For example, in Figure 1.7 π_1 and π_2 are homotopic, but π_1 and π_3 are not, because then we would need to push π_1 through ω_1 and ω_2. Note that the homotopic relation is an equivalence relation. This means that there are equivalence classes for the homotopic relation. If a path π is in the equivalence class $\mathcal{C}$, we call $\mathcal{C}$ the *homotopy class* of path π.

As we can define a path or a pair of paths to be non-crossing, we can do the same for homotopy classes. A homotopy class $\mathcal{C}$ is called non-crossing if there is a path $\pi \in \mathcal{C}$ which is non-crossing. Similarly, two homotopy classes $\mathcal{C}_1$ and $\mathcal{C}_2$ are called pairwise non-crossing if there are two paths $\pi_1 \in \mathcal{C}_1$ and $\pi_2 \in \mathcal{C}_2$ such that π_1 and π_2 are pairwise non-crossing (see Figure 1.8). For paths it is easy to see that if n paths are pairwise non-crossing and

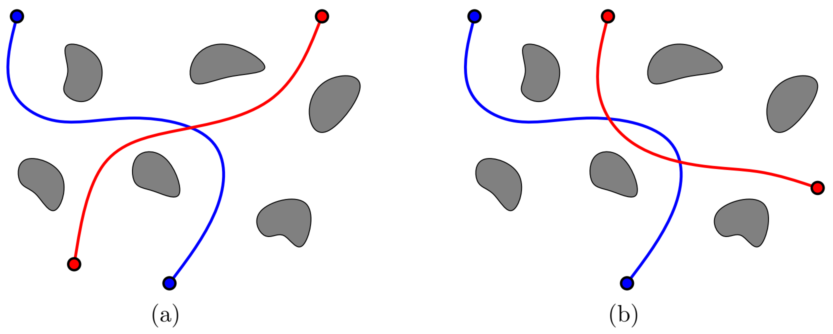

Figure 1.8: Two paths of crossing homotopy classes (a) and of non-crossing homotopy classes (b).

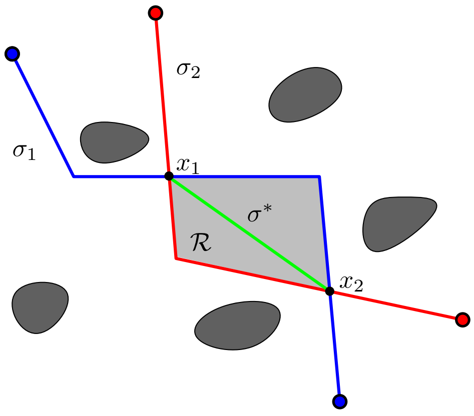

Figure 1.9: Shortest paths cannot cross.

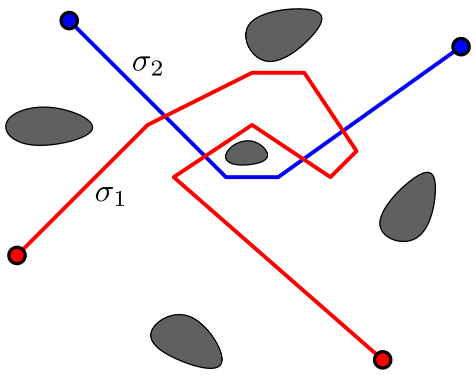

Figure 1.10: Path σ_1 is obviously not the shortest.

non-crossing themselves, then the collection of n paths is non-crossing. For homotopy classes this is not so trivial. Therefore we need the following lemma. Although this is a known fact, we include a sketch of the proof here for the sake of completeness.

Lemma 1 *Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two non-crossing homotopy classes which are pairwise non-crossing. Then the shortest paths of these homotopy classes $\sigma_1 \in \mathcal{C}_1$ and $\sigma_2 \in \mathcal{C}_2$ are non-crossing.*

Proof sketch. We proof this lemma by contradiction. Assume that σ_1 and σ_2 are crossing. Choose one of the endpoints of σ_1 as starting endpoint and consider the first crossing x_1 from this endpoint of σ_1 with σ_2 (see Figure 1.9). Because the homotopy classes $\mathcal{C}_1$ and $\mathcal{C}_2$ are non-crossing and because the endpoints of σ_1 and σ_2 are part of the blocker set $\mathcal{B}$, there must also be a second crossing x_2. The parts of the paths σ_1 and σ_2 between the crossings x_1 and x_2 form a small region $\mathcal{R}$ (a simple polygon) with a non-zero area (else the paths would be non-crossing). The region $\mathcal{R}$ cannot contain an element of the blocker set $\mathcal{B}$, because then the homotopy classes $\mathcal{C}_1$ and $\mathcal{C}_2$ would be crossing. There are exceptions (see Figure 1.10), but the paths of such an exception can always be shortened and hence cannot be shortest paths. Now consider the shortest path σ^* between x_1 and x_2 in $\mathcal{R}$. Because the region $\mathcal{R}$ has a non-zero area, either σ_1 or σ_2 could follow σ^* to become shorter. Because $\mathcal{R}$ does not contain an element of the blocker set $\mathcal{B}$, this also does not change the homotopy class. Contradiction. $\square$

From Lemma 1 it easily follows that a collection of n non-crossing homotopy classes is non-crossing if the homotopy classes are pairwise non-crossing.

Problems considered and results. We are now ready to introduce the problems considered in this thesis. In the first problem we are given a collection of n pairs of endpoints (a_i, b_i) $(1 \leqslant i \leqslant n)$ in the plane. We denote the set of endpoints by $\mathcal{E}$. Between each pair of endpoints a_i and b_i there is a rectilinear path π_i. We denote the set of paths by $\Pi = \bigcup_{i=1}^{n} \pi_i$. In addition to these paths and endpoints, there is a collection of m rectangular obstacles ω_i. We denote the set of rectangular obstacles by $\Omega = \bigcup_{i=1}^{m} \omega_i$. The paths π_i do not cross any obstacle in

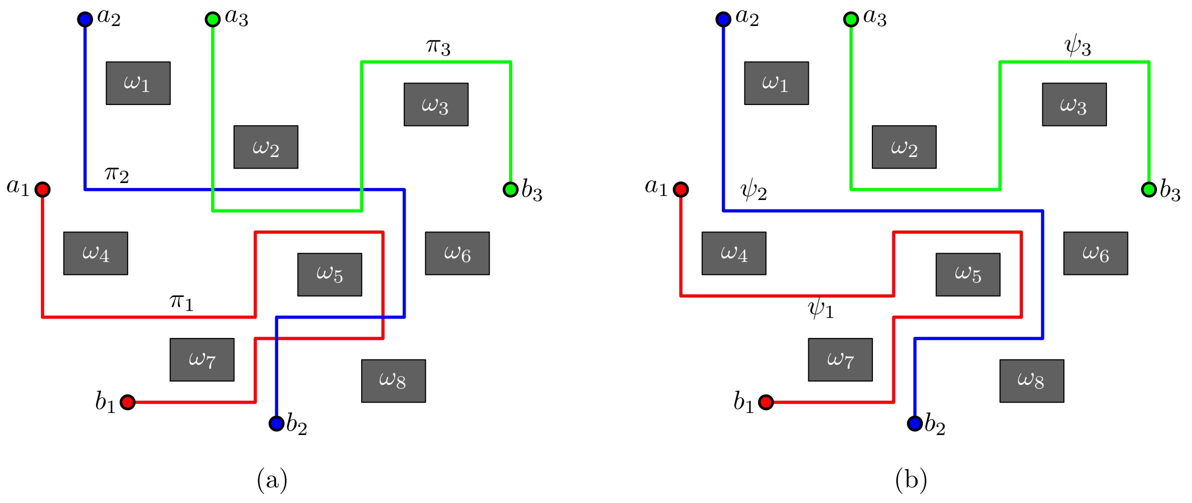

Figure 1.11: A problem instance of the routing problem (a) and its solution (b).

Ω, but they can overlap (part of) the boundary of one or more obstacles. Finally we define the blocker set $\mathcal{B}$ as $\mathcal{B} = \mathcal{E} \cup \Omega$. We furthermore require that the homotopy classes of the π_i w.r.t. $\mathcal{B}$ are non-crossing. We are looking for a collection of "untangled" rectilinear paths $\Psi = \bigcup_{i=1}^{n} \psi_i$ with the following properties:

- All paths ψ_i are homotopic to π_i, i.e. $\psi_i \sim_h \pi_i$ w.r.t. $\mathcal{B}$ for $1 \leqslant i \leqslant n$.
- The collection of paths Ψ is non-crossing.
- The total number of links of all paths in Ψ is minimal.

We formally call the problem described above Minimum-link Rectilinear Homotopic Routing. For brevity we usually refer to it as the routing problem. An example of a problem instance and its solution is shown in Figure 1.11. We have already mentioned in the introduction that we restrict the problem to paths as an initial attempt to tackle the more general problem on networks. For schematic map construction it is essential to fix the homotopy classes, because the winding of paths (or roads) around obstacles (cities and landmarks) is used to identify the paths. For the readability of a schematic map it is also important to allow no crossings, minimize the number of links, and to restrict the orientations of the links. Although rectilinear paths might be too restricting, this restriction is a good first step towards the more general and preferable c-oriented paths.

Note that the requirement of the homotopy classes of π_i to be non-crossing is necessary to find any solution. We show that it is also sufficient. Although the shortest paths of given homotopy classes are certainly non-crossing, these paths are not rectilinear. We will however show in Chapter 2 that a rectilinear solution does exists. This solution is a 2-approximation as it uses no more than twice the total number of links of an optimal solution. We first solve the problem for staircase paths and then we reduce the problem for general rectilinear paths to the problem for staircase paths. Our algorithm runs in $O((m+n)k)$ time and uses $O((m+n)k)$ storage, where k is the total number of links of the input paths and m and n are as described above. We also show that a 2-approximation is the best possible when the size of

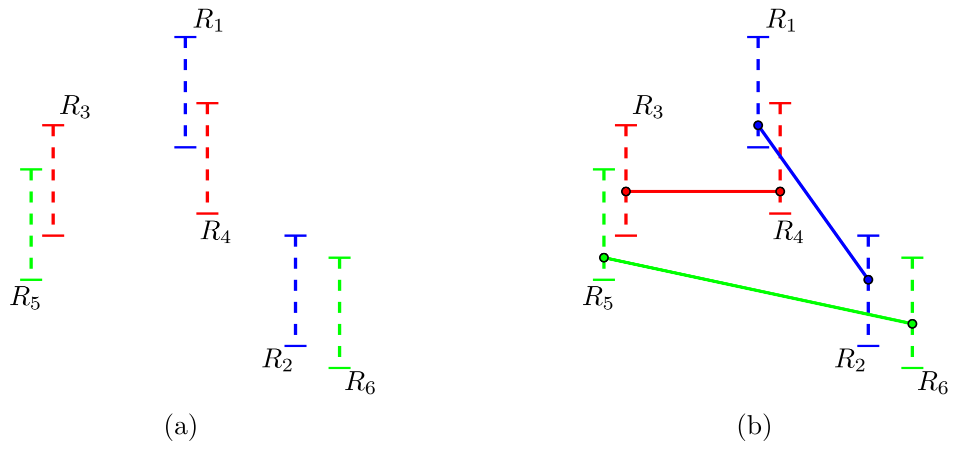

Figure 1.12: A problem instance of Vertical Matching (a) and a valid solution (b).

the optimal solution is lower bounded by the sum of optimal individual paths. Unfortunately the complexity status of this problem remains open.

In Chapter 3 we study a different problem. In the routing problem, the endpoints of the paths were fixed. Here we explore what would happen if the endpoints are not completely fixed, but are in fact allowed to be in a given region of the plane. If every endpoint can be anywhere in the plane, then the problem reduces to finding a planar embedding of paths, which is trivial. However if the regions are more limited, the problem becomes harder. To demonstrate this, we consider a restricted problem using regions and show that this problem is NP-Complete. This problem is as follows.

We are given a collection of $2n$ vertical segments R_i $(1 \leqslant i \leqslant 2n)$ which each consist of a triplet $R_i = (x_i, m_i, M_i)$. These vertical segments are the regions and region R_i is the vertical segment at x-coordinate x_i that is between m_i and M_i, which are the minimum and maximum y-coordinates, respectively. We want to connect region R_{2j-1} with region R_{2j} with a straight line segment for $1 \leqslant j \leqslant n$. The decision problem becomes the following. Are there y-coordinates y_i for each region R_i satisfying $m_i \leqslant y_i \leqslant M_i$ such that the line segments created by using the coordinates (x_i, y_i) are non-crossing?

We call this problem Vertical Matching. An example of a problem instance and its solution is shown in Figure 1.12. Note that this problem would be trivial with fixed endpoints. It is also important to mention that for Vertical Matching, the homotopy classes of the line segments are not given and are therefore not fixed, unlike in the routing problem. Finally we could also generalize this problem to different classes of regions instead of only vertical segments. Although we can use different names for the problems using different classes of regions, we just use one name for all these problems which is Region Matching. We will actually show that for many classes of regions, Region Matching is also NP-Complete.

1.2 Related work

The routing problem is related to the wire routing problem in VLSI design. The wire routing problem is that of connecting modules using wires such that these wires do not cross. Usually the endpoints of these wires are called terminals. Note that in VLSI design the homotopy classes do not need to be fixed. Unfortunately, as was mentioned in [6] in 1984, many variants of this problem are NP-hard (this was later also proven for many variants in [1] by Bastert and Fekete). That is why in [6] Cole and Siegel simplified the problem such that the homotopy classes are given. They then showed how to check if such a wiring can be satisfied without crossings. Four years later in 1988 Gao *et al.* [9] provided an algorithm that can find such a wiring in $O(n^3 \log n)$ time using $O(n^3)$ space where n is the complexity of the output paths plus the number of features (terminals and obstacles). They did however also require a minimum distance between two wires. A more complete coverage of this and similar problems was finally given by Miller Malley in [17] (1990). Unfortunately none of these algorithms minimize the number of links and hence these algorithms cannot be used to solve our problem.

Although simplification by using fixed homotopy classes made the problem more approachable, there were also some other variants of the original wiring problem that were solvable in polynomial time. In 1992 Takahashi *et al.* [22] presented an algorithm that finds k non-crossing paths connecting k terminal pairs in an undirected plane graph G with non-negative edge lengths. Their algorithm minimized the total length of these paths in $O(n \log n)$ time, where n is the number of vertices in G. This algorithm was then used in 1993 [23] by the same authors to connect k terminal pairs with k rectilinear paths among rectilinear obstacles. This algorithm also had a running time of $O(n \log n)$ time, but in this case n was the number of terminal pairs k plus the number of rectilinear obstacles. The only restriction for these algorithms was that the terminal pairs had to be on two specified face boundaries. Because the terminal pairs are restricted to two face boundaries and because only the length is minimized, this algorithm cannot be used for our problem.

More research was however done for the problem variant using fixed homotopy classes. In 1994 Hershberger and Snoeyink [12] presented algorithms to find shortest paths of a given homotopy class among a collection of points. They also presented algorithms to find paths of minimum complexity (minimum-link paths) of a given homotopy class for general paths and c-oriented paths (see Chapter 1.1). Their algorithm used a triangulation of the point set and could compute the shortest path of a given homotopy class (represented by a path α) in $O(C_\alpha + \Delta_\alpha)$ time, where C_α is the complexity of path α and Δ_α is the number of times α crosses a triangulation edge. Finding the minimum-link path of a given homotopy class required $O(C_\alpha + \Delta_\alpha + \Delta_{\alpha_{\min}})$ time where $\alpha_{\min}$ is the minimum-link path. Although the algorithm for shortest paths could directly be used to find k shortest non-crossing paths, it turned out that this could be done even more efficiently. This was done independently by Bespamyatnikh [2] and by Efrat *et al.* [8]. The result was an algorithm for finding k non-crossing shortest paths of given homotopy classes in $O(n\sqrt{n} + k_{in} \log n + k_{out})$ time, where n is the number of endpoints plus the number of obstacles and k_{in} and k_{out} are the complexities of the input paths (representing the homotopy classes) and output paths respectively.

In these problems the total length of the paths was often minimized. Relatively little research had been done to find non-crossing paths of minimum complexity or the minimum number of links. In 1996 it was already shown by Bastert and Fekete [1] that, among other wiring problems, finding non-crossing paths with the minimum number of links without given homotopy classes is NP-Hard. In 1997 however the problem was solved by Yang *et al.*[24] for a pair of rectilinear paths inside a rectilinear polygon. They provided an $O(n)$ time algorithm, with n the complexity of the rectilinear polygon, to find a pair of non-crossing rectilinear paths inside the rectilinear polygon with the minimum number of links and minimum length. Unfortunately they were unable to generalize this solution to more than two paths. In [1] Bastert and Fekete also showed that for n terminal pairs, finding non-crossing paths might require one path to have at least $O(\log n)$ links. This bound was later improved in 1998 by Pach and Wenger [20]. They showed that for n terminal paths it could happen that at least $O(n)$ paths require $O(n)$ links to make the paths non-crossing, which makes the total complexity of the non-crossing paths $O(n^2)$. They also presented an algorithm to find a planar embedding of a graph in $O(n^2)$ time with fixed vertices and polygonal curves as edges using $O(n)$ links per edge. Although this result is asymptotically optimal, they did not really minimize the number of links. This was however attempted in 2007 by Gupta and Wenger [11]. They presented a constant factor approximation algorithm for finding non-crossing paths with the minimum number of links where all terminals are vertices on a simple polygon. Note that when all terminals are vertices of a simple polygon, the homotopy classes are actually fixed. Their algorithm runs in $O(n \log m + M \log m)$ time, where n is the complexity of the simple polygon, m is the number of terminal pairs and M is the total number of links of the optimal solution. Unfortunately the number of links of their solution can only be bounded by $120M + 127m$.

Although we have simplified the problem to be on paths instead of networks, not everyone has done so. There is some related work on drawing schematized networks and metro maps. In 2005 Cabello *et al.* [4] worked on drawing schematized networks with fixed endpoints. Their algorithm computes a schematized map topologically equivalent to an input map in $O(n \log n)$ time, where n is the total number of links in the input map. Every path of the schematized map has two or three links with restricted orientations. They could also add additional constraints like a minimum vertical separation between paths. Unfortunately if no map existed with these restrictions, the algorithm would simply report that. This and other related problems are studied in Cabello's PhD thesis [3]. For drawing metro maps, the endpoints are usually not entirely fixed. In 2005 Nöllenburg and Wolff [19] used a mixed-integer program to draw metro maps using only horizontal, vertical and diagonal links. They distinguished between hard and soft constraints and used the mixed-integer program to enforce the hard constraints and optimize the soft constraints. Although the results were good and they used heuristics to improve the running time, mixed-integer programming is NP-Hard and hence their algorithm is infeasible for larger instances. In 2006 Merrick and Gudmundsson [18] used a different approach using path simplification. They provided an algorithm to simplify a path such that the resulting links conform with a restricted set of directions $\mathcal{C}$. This algorithm runs in $O(|\mathcal{C}|^3 n^2)$ time, where n is the number of vertices of the original path. The authors then described how to use this for a network. Unfortunately their algorithm does not keep the input topology intact.

These problems are usually considered with paths that are infinitely thin, but in practice these paths always have a certain thickness. The thickness of the paths can also be used to convey extra information in schematic maps. So it might be useful to consider the thickness of a path as part of the problem. This was done in 2002 by Duncan *et al.* [7]. They presented an algorithm to maximize the distance between paths with given homotopy classes, which can be seen as finding paths with maximum thickness. Their algorithm runs in $O(kn + n^3)$ time where n is the number of paths and k is the maximum of the input and output complexities.

They also showed how to extend this algorithm to work for general planar graphs. In 2007 Polishchuk and Mitchell [21] did the same for paths inside a simple polygon. However their algorithm computes a representation of the thick paths in $O(n + K)$ time, where n is the number of vertices of the polygon and K is the number of paths. Using this representation they can output a thick path in time proportional to the complexity of the path. Unfortunately only the lengths of the paths are optimized in [7] and [21]. No work has been done in minimizing the number of links of non-crossing thick paths.

In most problems discussed above, all the endpoints of the paths are fixed. In some applications however (like for metro maps) it is enough to restrict the endpoints to some region. Relatively little research has been done in this direction. In 2005 Abellanas *et al.* [16] described a heuristic method to draw a graph for which the vertices are restricted to regions. They use a force-directed algorithm to optimize some aesthetic criteria, like the the number of edge crossings and the location of a vertex in its region w.r.t. the center of the region. Unfortunately this is only a heuristic method. In 2007 Löffler [14] proved the NP-Completeness of a problem involving regions. Although his problem involved imprecise points, this is essentially the same as restricting points to regions. Given an ordered set of regions representing a polygon, the problem was to decide whether it is possible to place the points inside their regions in such a way that the resulting polygon is simple. He also showed it is NP-hard to minimize the length of a simple tour visiting the regions in order. A more complete version of this paper is given in [15]. Note that the problem proven to be NP-Complete by Löffler is very similar to Vertical Matching. In fact, both problems are about embedding a graph in the plane without crossings with the vertices restricted to a region, but for Vertical Matching the graph is a matching and in [14] the graph is a cycle.

Chapter 2

Minimum-link rectilinear homotopic routing

In this chapter we study the problem Minimum-link Rectilinear Homotopic Routing.

We are given a collection of n pairs of endpoints (a_i, b_i) $(1 \leqslant i \leqslant n)$ in the plane with $\mathcal{E} = \bigcup_{i=1}^{n} \{a_i\} \cup \{b_i\}$, a collection of n rectilinear paths $\Pi = \bigcup_{i=1}^{n} \pi_i$ connecting a_i to b_i, and a collection of m rectangular obstacles $\Omega = \bigcup_{i=1}^{m} \omega_i$. The blocker set $\mathcal{B}$ is defined by $\mathcal{B} = \mathcal{E} \cup \Omega$. Assuming that the homotopy classes of Π w.r.t. $\mathcal{B}$ are non-crossing, we would like to untangle the paths in Π, that is, to find a collection of n paths $\Psi = \bigcup_{i=1}^{n} \psi_i$ such that $\psi_i \sim_h \pi_i$ w.r.t. $\mathcal{B}$ for $1 \leqslant i \leqslant n$, Ψ is non-crossing, and the total number of links in Ψ is minimal.

We present an algorithm that calculates a 2-approximation for this problem.

This chapter is organized as follows. First we look at locally optimal paths and their role as lower bound for the optimal solution in Chapter 2.1. Then we restrict the problem to staircase paths and describe a 2-approximation for that in Chapter 2.3. After that we extend the result to general rectilinear paths in Chapter 2.4. Then we discuss how to do this efficiently in Chapter 2.5. Finally we show in Chapter 2.6 that the factor 2 of the approximation is optimal when using locally optimal paths as lower bound for the optimal solution.

2.1 Locally optimal paths

We approach the problem in the following way. We choose a collection of paths that is almost a solution and then we change this collection of paths until it is a valid solution. The initial paths that we use are the rectilinear minimum-link paths of the given homotopy classes. We call these paths the *locally optimal paths*. Locally optimal paths (i) are not unique (see Figure 2.1(a)) and (ii) can have crossings (see Figure 2.1(b)). While one locally optimal path might be a better starting point for our algorithm than another, any locally optimal path is sufficient for an approximation ratio of 2. Also there are configuration where any collection of locally optimal paths has crossings (see, again, Figure 2.1(b)). Since the locally optimal paths are the optimal solution if crossings are allowed, the total number of links of the locally optimal paths are a lower bound for the optimal solution of the routing problem.

Finding these locally optimal paths is not trivial. Fortunately this problem has been solved before. An algorithm for finding a rectilinear minimum-link path of a given homotopy class is given in [12]. We discuss this more in depth in Chapter 2.5.

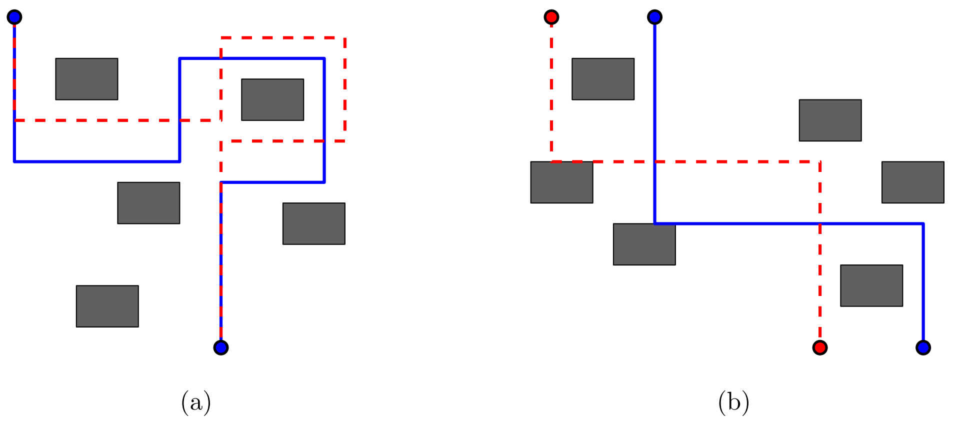

Figure 2.1: Locally optimal paths are not unique (a) and can have crossings (b).

We now have a collection of paths for which the number of links is a lower bound for the solution of the routing problem, but there can be crossings between these paths. These crossings need to be removed in order to get a valid solution. We need to do this in such a way that the number of links does not increase too much. In fact, to find a 2-approximation, we need to remove the crossings by no more than doubling the number of links. How to do this will be discussed in Chapter 2.3 and 2.4. But before we do that, we need to introduce some additional concepts.

2.2 Intersection regions for y-monotone paths

To remove the crossings of the locally optimal paths, we need to move the links of the paths. Before this can be done, we need to make sure that these links are not moved through endpoints or obstacles as this would change the homotopy class. Of course we can easily check this, but it would be better if we would just know where we can safely move the links without changing the homotopy class. For that we use *intersection regions*.

To give a definition of intersection regions for y-monotone paths, we first restrict the routing problem to have only y-monotone paths. So the input paths π_i and the output paths ψ_i are y-monotone (and rectilinear). With this restriction we can prove the following lemma.

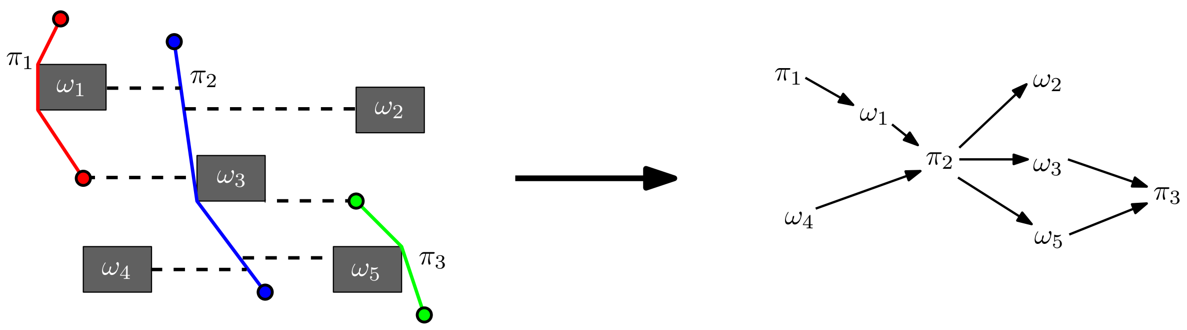

Figure 2.2: The partial order on a collection of shortest paths and obstacles.

Lemma 2 *Given a problem instance of the routing problem with only y-monotone paths and non-crossing homotopy classes, the homotopy classes of all paths can be characterized by a total order $\mathcal{O}$ on the paths and the obstacles.*

Proof. Because the homotopy classes are non-crossing, we can consider the shortest paths which are non-crossing. These shortest paths must be y-monotone as well. Consider the relation "to the left of" for these paths and obstacles. This relation is clearly defined between two paths/obstacles as long as they share a y-coordinate (see Figure 2.2). Because the paths are y-monotone and non-crossing, this relation defines a partial order on the paths and obstacles, which can easily be extended to a total order $\mathcal{O}$. For y-monotone paths the homotopy class dictates only on which side (left or right) a path passes an obstacle or endpoint. This information is now completely given by the total order $\mathcal{O}$. $\square$

Using Lemma 2, the homotopy classes of the y-monotone paths can be characterized by a total order $\mathcal{O}$ on the paths and obstacles. Note that the paths include the endpoints, so the positions of the endpoints in the order is also defined. We use $\mathcal{O}(\pi_i)$ and $\mathcal{O}(\omega_i)$ for the position in the order of paths and obstacles, respectively. Now we can give the following definition for intersection regions.

Definition 4 *Assume that π_i and π_j are two y-monotone paths with $\mathcal{O}(\pi_i) < \mathcal{O}(\pi_j)$. An* intersection region *of π_i and π_j is a region enclosed by π_i and π_j at y-coordinates where the paths are out of order, i.e. where π_j is to the left of π_i.*

Consider Figure 2.3. Two paths can be out of order multiple times, but we do not consider these regions as one intersection region, but rather as multiple intersection regions. So intersection regions are y-monotone rectilinear polygons. Also, intersection regions are always defined between only two paths. In this way we have more control over the shape of intersection regions.

Intersection regions have one convenient property: They cannot contain blockers. This is shown by the following lemma.

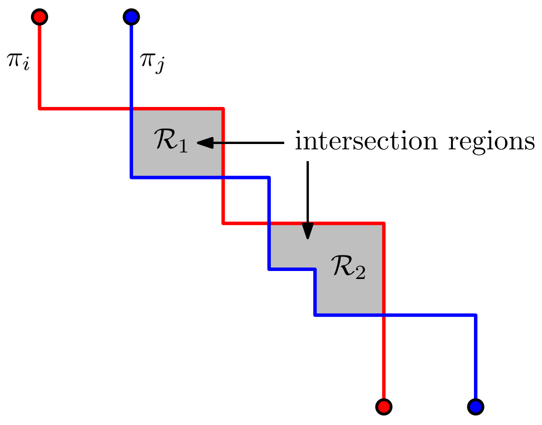

Figure 2.3: Two paths π_i and π_j and their intersection regions $\mathcal{R}_1$ and $\mathcal{R}_2$.

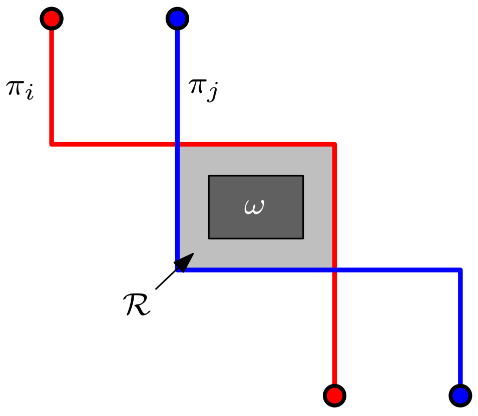

Figure 2.4: Intersection regions cannot contain any blocker.

Lemma 3 *Let π_i and π_j be two y-monotone paths with $\mathcal{O}(\pi_i) < \mathcal{O}(\pi_j)$ and assume they have an intersection region $\mathcal{R}$. Then there can be no blocker in $\mathcal{R}$.*

Proof. We proof this lemma by contradiction. Because the paths are y-monotone, we know that there is a total order $\mathcal{O}$ on the paths and blockers. Assume there is a blocker ω in $\mathcal{R}$ (see Figure 2.4). We know that $\mathcal{O}(\pi_i) < \mathcal{O}(\pi_j)$. Because π_i passes ω on the right, we also know that $\mathcal{O}(\omega) < \mathcal{O}(\pi_i)$. And because π_j passes ω on the left, we also know that $\mathcal{O}(\pi_j) < \mathcal{O}(\omega)$. These together imply that $\mathcal{O}(\pi_j) < \mathcal{O}(\pi_i)$. Contradiction. □

Lemma 3 proves that every intersection region is free of blockers. Hence we can move links of paths through intersection regions without changing the homotopy class of a path.

2.3 A 2-approximation for positive staircase paths

In this chapter we describe how the crossings of locally optimal paths can be removed while no more than doubling the number of links. We first consider positive staircase paths and then extend our method to general rectilinear paths. Note that the case where all paths are negative staircase paths is symmetrical. As the staircase paths are y-monotone, we can use Lemma 2 which implies an order on the paths. We assume that the paths are numbered accordingly, so $\mathcal{O}(\pi_i) < \mathcal{O}(\pi_j)$ iff $i < j$.

Rectangular intersection regions. To simplify the following discussion we first ensure that all intersection regions are rectangular without adding additional links to the locally optimal paths. We can simply do this by pushing the horizontal links of all paths as much down as possible and the vertical links as much to the right as possible. We call these paths *downmost rightmost*.

Lemma 4 *Positive staircase paths of given non-crossing homotopy classes that are downmost rightmost have only rectangular intersection regions.*

Proof. We proof this lemma by contradiction. Assume we have an intersection region between two paths π_i and π_j with $i < j$ (see Figure 2.5). Because both paths are positive staircase paths, the boundary of this intersection region consists of two parts: the lower-left

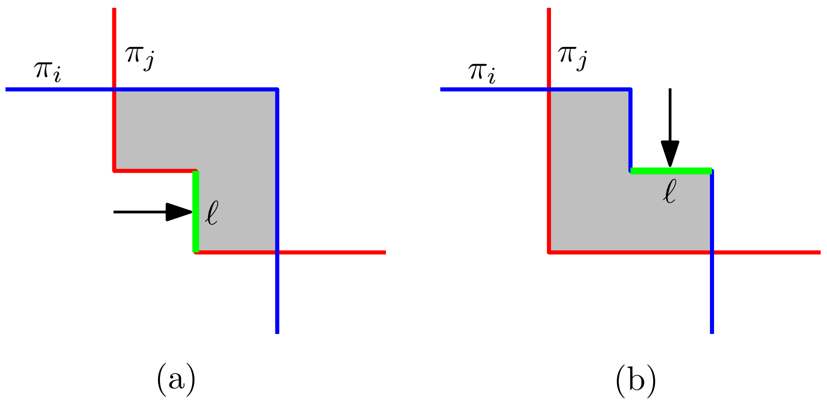

Figure 2.5: Vertical (a) or horizontal (b) link ℓ is not rightmost or downmost.

part formed by π_j and the upper-right part formed by π_i. By definition this intersection region is simply connected, so these two parts cannot cross. Assume that the lower-left part does not consist of only two links. In that case, as shown in Figure 2.5(a), there must be a vertical link ℓ that is completely part of the boundary of the intersection region. Because of Lemma 3, the intersection region is free of blockers. But that means that the link ℓ can be moved to the right, which contradicts the fact that π_j is rightmost. Assume that the upper-right part does not consist of only two links as is shown in Figure 2.5(b). Then there must be a horizontal link that is completely part of the boundary of the intersection region. Following the above argumentation, this contradicts the fact that π_i is downmost. So both parts must consist of only two links and therefore the intersection region is rectangular. □

We either directly find locally optimal paths that are downmost rightmost or we use any collection of locally optimal paths and push the links to be downmost rightmost. Note that this works only for positive staircase paths. An example is shown in Figure 2.6. We have to make negative staircase paths downmost leftmost to get rectangular intersection regions.

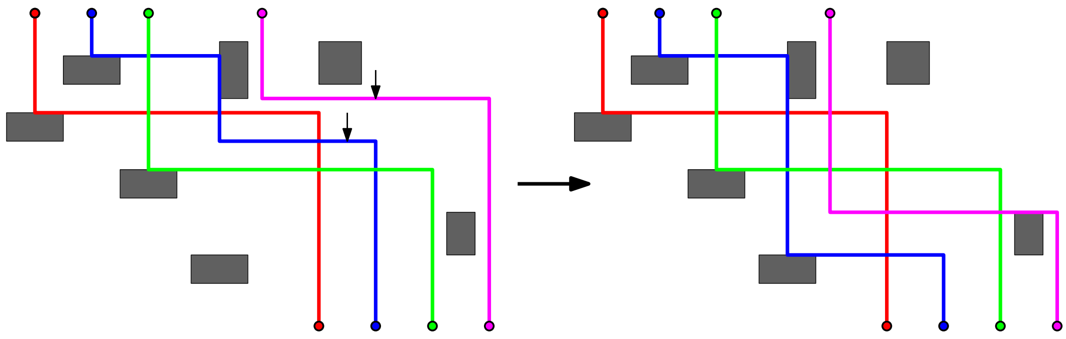

Figure 2.6: Making intersection regions rectangular.

Positive staircase paths have only two types of bends. A bend where the direction changes from down to right is called a *lower-left bend*. A bend where the direction changes from right to down is called an *upper-right bend* (see Figure 2.7).

Lemma 5 *Let π_i and π_j be two positive staircase paths with rectangular intersection regions $(i < j)$. An upper-right bend of π_j can never be to the left of π_i.*

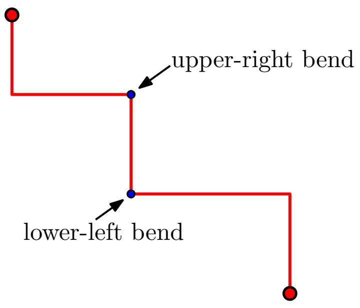

Figure 2.7: Bends of positive staircase paths.

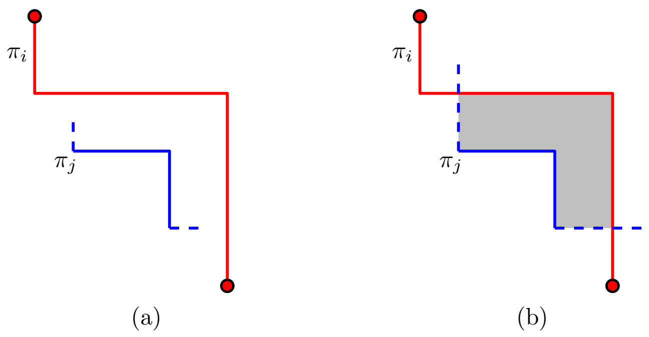

Figure 2.8: Upper-right bend of π_j to the left of π_i (a) and the intersection region of π_i and π_j (b).

Proof. We proof this lemma by contradiction. Assume that an upper-right bend of π_j is to the left of π_i (see Figure 2.8(a)). Because $i < j$, this means that at this upper-right bend the paths π_i and π_j are out of order and must therefore have an intersection region at this bend. But the path π_j forms the lower-left part of the boundary of this intersection region and the upper-right bend has only links extending to the left and down. So this intersection region is not rectangular (Figure 2.8(b)). Contradiction. □

Untangling paths. We now have an ordered collection of locally optimal paths that are positive staircase paths and that have only rectangular intersection regions. We want to remove the crossings between these paths – untangle the paths – or equivalently remove the intersection regions.

Untangling the paths works as follows. We use an incremental algorithm, adding the paths from left to right. When adding a new path we remove the crossings with the previously added paths. We do this in such a way that we do not need more than twice the number of links we started with. We call this algorithm UNTANGLE.

To keep it simple, we first look at the first two paths π_1 and π_2 (the first path can be added without problems). These two paths can of course have crossings and therefore intersection regions (see Figure 2.9). Since we made all paths downmost rightmost, these intersection regions are nicely rectangular. We essentially have two choices now. Either we reroute π_1 along π_2 or we reroute π_2 along π_1. As we are adding π_2 at this time, we decide to reroute π_2 along π_1, removing the crossings between π_1 and π_2. It is easy to see that in doing so we no more than doubled the number of links of π_2.

Unfortunately we cannot continue doing this for all paths. Rerouting π_2 might have changed the shape of the intersection regions with any path π_k ($k > 2$). If the intersection regions are no longer rectangular, then rerouting costs more links. We hence need to keep track of where the paths have been rerouted. For π_2 the difference between the rerouted path and the original locally optimal path is exactly the (rectangular) intersection region. To keep track of where the paths are rerouted, we introduce the *reroute box* (see Figure 2.9).

Definition 5 *A* reroute box *of a path denotes the difference between the original (locally optimal) path and the rerouted path. The lower-left part of the reroute box follows the original path, whereas the upper-right part of the reroute box follows the rerouted path.*

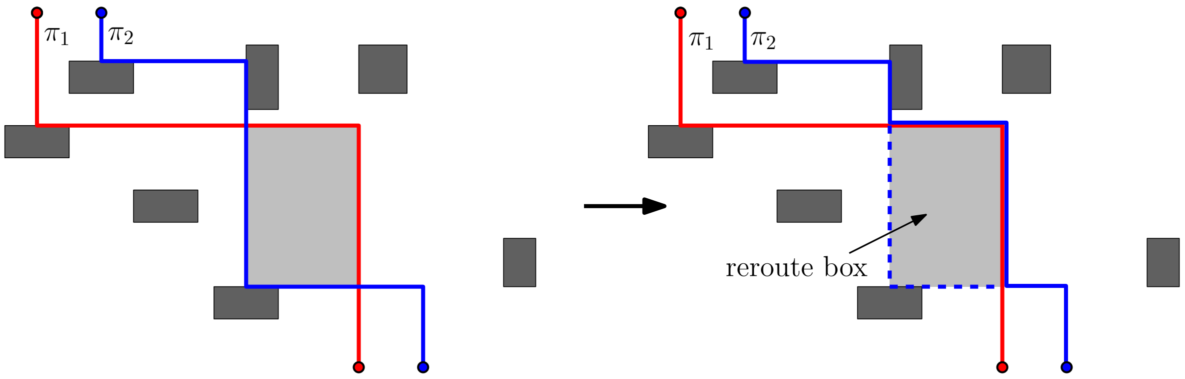

Figure 2.9: Untangling the first two paths.

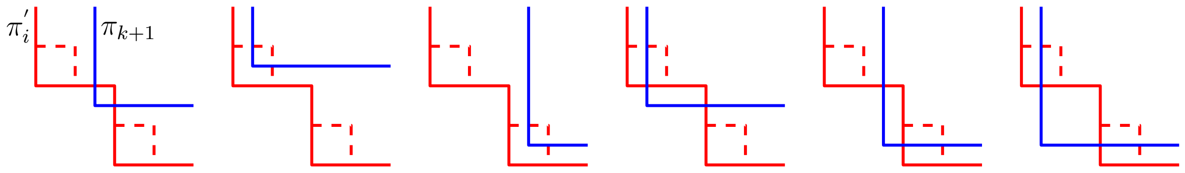

Figure 2.10: All possible cases for crossings between π_{k+1} and a path π_i.

The algorithm UNTANGLE changes the locally optimal paths π_i to make them non-crossing. The changed paths are denoted by π_i'. Note that a path is only changed after it is added. Before we discuss adding a path π_{k+1} with $k > 1$, we first give the invariants that must be maintained by UNTANGLE. These invariants are the following after adding a path π_k.

Invariant 1 (NonCross) *The paths π_1' to π_k' are non-crossing.*

Invariant 2 (Reroute) *The paths π_1 to π_k have at most one reroute box per lower-left bend.*

Invariant 3 (UpperRightBend) *An upper-right bend of a path π_j can never be to the left of a path π_i' with $1 \leqslant i < j \leqslant n$.*

It is clear that Invariant *NonCross* and Invariant *Reroute* are initially true. Lemma 5 proves that also Invariant *UpperRightBend* is initially true.

Now we discuss the algorithm when adding a path π_{k+1}. We go through all the links of π_{k+1} from top to bottom. This is done until a crossing is encountered. Note that the first crossing must always be with a vertical link of π_{k+1}, because the horizontal links move to the right and therefore away from the other paths. This link can have crossings with multiple other paths. Before we can find a way to remove these crossings, we first need to consider which shapes of intersection regions we can get. We know that π_{k+1} is still unchanged and the other paths π_i' for $1 \leqslant i \leqslant k$ are just the original paths π_i plus the reroute boxes. We can still use the fact that the intersection regions between the original paths π_i are rectangular. All possible cases that we can get are shown in Figure 2.10. We can get only cases with 1, 2 or 3 upper-right bends in the upper-right part of the boundary of the intersection region. Also for all cases, the lower-left part of the boundary of the intersection region consists of only two links. Note however that we have ignored one case that would be theoretically possible as that case would also result in a rectangular intersection region for the original paths. That is the case where the following upper-right bend of path π_{k+1} is in a reroute box. This is however not possible due to Invariant *UpperRightBend*.

To reroute we cannot simply follow the rightmost path that is crossed, because this can cost more than two links. To change this, we need to make the intersection region of the rightmost crossed path with π_{k+1} rectangular. We use an incremental algorithm to achieve this. We call this subroutine GROWRECTANGLE. Assume the crossing paths are paths π_a' to π_b'. These paths are not crossing each other due to Invariant *NonCross*. Now assume that the intersection region $\mathcal{R}$ of π_i' with π_{k+1} is rectangular and we want to make the intersection region of π_{i+1}' with π_{k+1} rectangular. We can achieve this by moving the links of π_{i+1}'. We do not want to move the links through $\mathcal{R}$, because that would introduce crossings. The cases of Figure 2.10 for π_{i+1}' can be handled as follows (Figure 2.11).

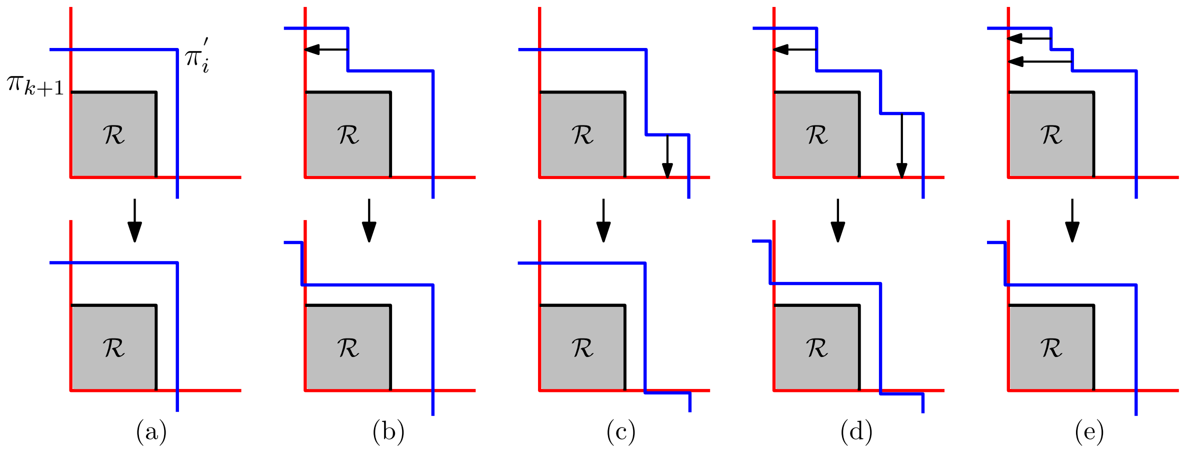

Figure 2.11: Handling all different cases.

One upper-right bend: It is already rectangular, so we leave it unchanged (Figure 2.11(a)).

Two upper-right bends: Due to Invariant *NonCross* of the untangling algorithm, the path π_{i+1}' cannot cross $\mathcal{R}$. To make the intersection region rectangular, we need to move the first vertical link to the left or the last horizontal link down (see Figure 2.11(b)-(c)). Because $\mathcal{R}$ is a rectangle, we can always do one of these moves without moving a link through $\mathcal{R}$.

Three upper-right bends: Again π_{i+1}' cannot cross $\mathcal{R}$. There are two cases. The first case is that $\mathcal{R}$ is in the middle corner (Figure 2.11(d)). Then we can safely move the first vertical link to the left and the last horizontal link down to form a rectangular intersection region. The second case is that $\mathcal{R}$ is in one of the other corners (Figure 2.11(e)). In that case we can simplify the middle corner and handle it as a case with two upper-right bends.

Initially we can use $\mathcal{R} = \emptyset$ for π_a'. Following the algorithm GrowRectangle results in a rectangular intersection region between π_b' and π_{k+1}. We can now create a reroute box like we did with π_2. An example of this is shown in Figure 2.12. This concludes the algorithm.

There is one more thing that must be mentioned. When performing the subroutine GrowRectangle we allowed the links to be moved. The key is however that all moves are to the left or down. So moving the links of a path π_i' does (i) not change the cases (Figure 2.10) and (ii) does not introduce crossings with a path π_j' $(j > i)$. Because we make sure that $\mathcal{R}$ is not crossed in the subroutine GrowRectangle and because adding the reroute box removes the crossings with π_{k+1}, Invariant *NonCross* is maintained. Invariant *Reroute* is also maintained, because we only add reroute boxes when a path is added and only at lower-left bends of the locally optimal paths. Finally Invariant *UpperRightBend* is also maintained. Paths are only changed by moving links or by adding reroute boxes. A reroute box of π_{k+1} cannot contain an upper-right bend of π_j $(j > k+1)$, because then this bend must be to the left of the rightmost crossed path π_b', which cannot happen according to Invariant *UpperRightBend*. Moving links left or down also cannot violate this invariant.

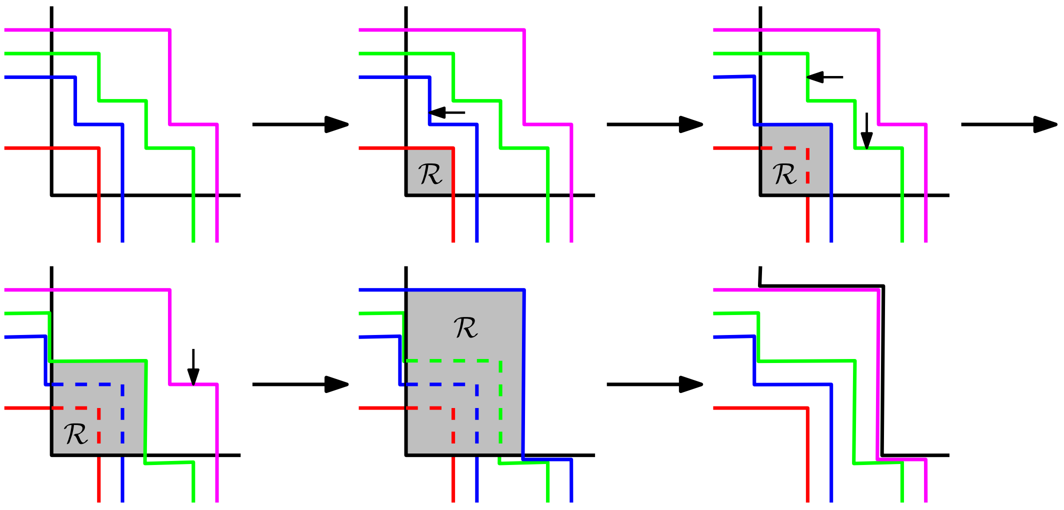

Figure 2.12: Growing a rectangle using the GrowRectangle subroutine.

Theorem 6 *Algorithm* Untangle *correctly computes a 2-approximation for the routing problem for positive staircase paths.*

Proof. To prove that the solution is valid, we need to prove two things: The resulting paths must be non-crossing and the resulting paths must be homotopic to the original paths. We have argued above that the invariants *NonCross*, *Reroute* and *UpperRightBend* are maintained by Untangle. Invariant *NonCross* ensures that the resulting paths are non-crossing. Paths are only changed by moving links or by adding reroute boxes. Reroute boxes are also intersection regions which are free of blockers due to Lemma 3. In the subroutine GrowRectangle the links are only moved through intersection regions (see Figure 2.11). So adding reroute boxes or moving links does not change the homotopy class of a path. Finally we need to prove that this algorithm is a 2-approximation. The original paths were locally optimal. We have only added links at the reroute boxes. In fact, for each reroute box we have added only two links. It follows from Invariant *Reroute* that there is at most one reroute box per lower-left bend. Note that the endpoints can never have a reroute box. The number of lower-left bends of a positive staircase path is at most $x/2$ if x is the number of links of a positive staircase path. If the total number of links of the original (locally optimal) paths is L, then the total number of links of the resulting paths is at most $L^{'} \leqslant L + 2L/2 = 2L$. Because L is a lower bound for the optimal solution, this proves that Untangle computes a 2-approximation for the routing problem for positive staircase paths. □

An example of this algorithm (after adding the first two paths in Figure 2.9) is shown in Figure 2.13. This algorithm can easily be adapted to work for negative staircase paths, but it cannot easily be adapted to work for general rectilinear paths. This will be discussed in the next chapter.

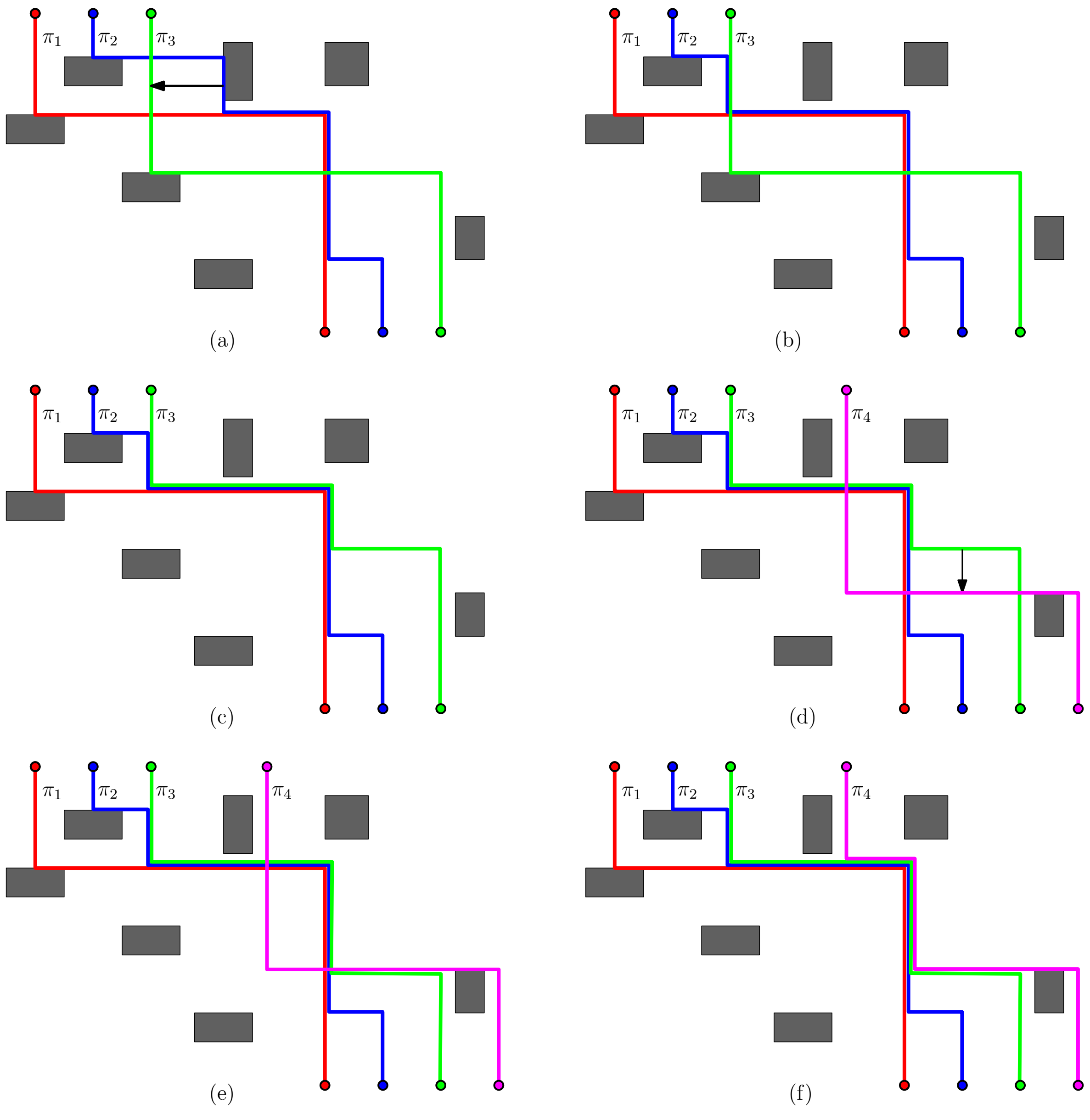

Figure 2.13: Example of untangling positive staircase paths.

2.4 Extension to general rectilinear paths

In this chapter we extend the result of Chapter 2.3 to general rectilinear paths. Note however that simply extending UNTANGLE to work for general rectilinear paths is problematic, because the monotonicity of the paths has been used throughout the entire algorithm. Our approach is therefore not to directly solve the problem for general rectilinear paths, but instead to reduce the problem for general rectilinear paths to the problem for staircase paths. After that we can use the algorithm of Chapter 2.3 to solve the routing problem. We call this entire algorithm ROUTING.

The way to reduce the routing problem to the problem for staircase paths is to manipulate the locally optimal paths. However, doing this requires us to first calculate the locally optimal paths. We can compute the locally optimal paths using the algorithm of Hershberger and Snoeyink in [12], but we can also manipulate the input paths directly.

The first step is to reduce the routing problem to the problem on y-monotone paths. The way this can be done is by making the "U-turns tight".

Definition 6 *A* horizontal U-turn *(see Figure 2.14) consists of a horizontal link with on both ends a vertical link extending in the same direction (both up or both down). The side from the horizontal link in the direction of the vertical links is called the* inside *of the U-turn. If instead the middle link is a vertical link and the other links are horizontal, then this is a* vertical U-turn.

Using this definition of U-turns, we can define *tight U-turns* (see Figure 2.14).

Definition 7 *A U-turn is called* tight *if there is a blocker on the inside of the U-turn touching the middle link.*

The first thing we must do for reducing the problem to y-monotone paths is to make all horizontal U-turns tight. This can be done separately for each path. We just consider a horizontal U-turn and move the horizontal link towards the inside of the U-turn. Then two things can happen (see Figure 2.15). Either the horizontal link can hit a blocker in which case the U-turn is tight or the horizontal link reaches the end of a neighboring vertical link in which case the path can be simplified to form a new U-turn. This simplification cannot happen forever, because this reduces the number of links in the path, which is finite. So after following this procedure all horizontal U-turns are tight. How this can be done efficiently is discussed in Chapter 2.5.

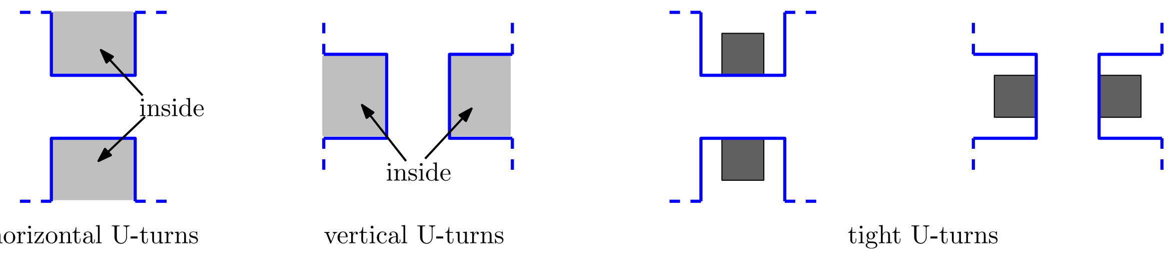

Figure 2.14: Different types of U-turns and tight U-turns.

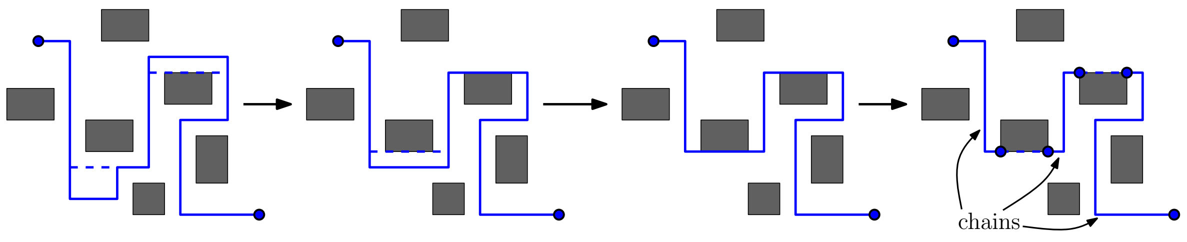

Figure 2.15: Making the horizontal U-turns tight and splitting up the path into y-monotone chains.

We now have a path with only tight horizontal U-turns. Note that while making the horizontal U-turns tight, no links were added. So if we would have started this with a locally optimal path, the path would still be locally optimal. Because the horizontal U-turns are tight, all U-turns have a blocker on the inside. The middle link at a tight U-turn is split up at the edges of the blocker (Figure 2.15). We ignore the middle part of a split following the border of a rectangular obstacle. Note that all the parts of the path are now y-monotone. We call these parts of the paths *chains*. The split points form the endpoints of the chains. These endpoints have an important property.

Lemma 7 *The endpoints of the y-monotone chains of a path π are always part of the shortest path σ with $\pi \sim_h \sigma$.*

Proof. We proof this lemma by contradiction. Let e be the first endpoint of a y-monotone chain of π which is not on σ (Figure 2.16). Assume without loss of generality that this is a lower endpoint of a chain. This means that σ must pass e along the bottom, because else $\pi \nsim_h \sigma$. Consider the horizontal line ℓ through e. Because the previous endpoint before e is on σ and not below ℓ, σ must cross ℓ once. But σ must also cross ℓ a second time to go around the blocker inside the next tight U-turn (or to go to the endpoint of π). But then following ℓ is shorter. Contradiction. □

By splitting the paths up into y-monotone chains we have reduced the routing problem to the problem on y-monotone chains. Note that we can also order the y-monotone chains now. How to do this efficiently will be discussed in Chapter 2.5. Some of these y-monotone chains

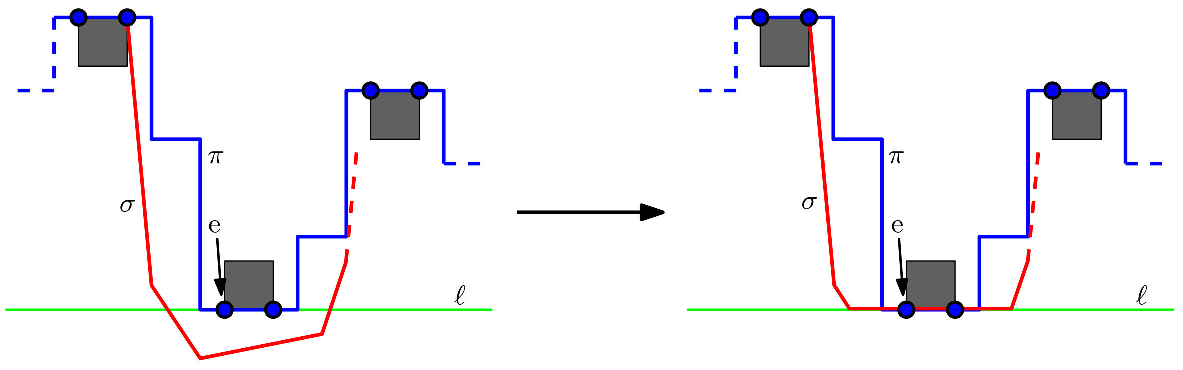

Figure 2.16: The endpoints of the y-monotone chains are always on the shortest path.

are homotopic. Chains that are homotopic can be bundled and routed together. Therefore we replace groups of homotopic chains by a single chain.

The next step is to reduce the routing problem on y-monotone chains to the problem on staircase chains (chains that are staircase paths). Luckily we can perform the same trick for the vertical U-turns. We first make the vertical U-turns tight and then we split the y-monotone chains up at the blockers inside the vertical U-turns. This results in a collection of staircase chains. Lemma 7 can easily be extended to hold for the endpoints of the staircase chains.

Corollary 8 *The endpoints of the staircase chains of a path π are always part of the shortest path σ with $\pi \sim_h \sigma$.*

We do however get a mix of positive and negative staircase chains. Note that the algorithm UNTANGLE of Chapter 2.3 can handle only staircase paths of one type and not a mix. Luckily the following lemma holds.

Lemma 9 *For a collection of rectilinear paths with non-crossing homotopy classes and tight U-turns, a positive staircase chain cannot cross a negative staircase chain.*

Proof. We proof this lemma by contradiction. Assume a positive staircase chain c_1 of a path π_1 is crossing a negative staircase chain c_2 of a path π_2 (see Figure 2.17(a)). Note that due to the monotonicity there can only be one crossing. Because the homotopy classes are non-crossing, the shortest paths are non-crossing according to Lemma 1. Corollary 8 states that the endpoints of the staircase chains are part of the shortest paths. This means that the shortest paths between the endpoints of the staircase chains must be non-crossing. Assume without loss of generality that the shortest path σ_1 ($\sigma_1 \sim_h \pi_1$) passes σ_2 ($\sigma_2 \sim_h \pi_2$) on the side of the top endpoint e_3 (see Figure 2.17(a)). Now there are three cases: There is a horizontal U-turn at e_3, there is a vertical U-turn at e_3, or e_3 is an endpoint of π_2. If e_3 is an endpoint of π_2, then $\sigma_1 \nsim_h \pi_1$, because e_3 is a blocker. If there is a horizontal U-turn at e_3, then there must be a blocker ω_1 below σ_2 (see Figure 2.17(b)). Because $\sigma_1 \sim_h \pi_1$, σ_1 must pass ω_1 on the left side. But we assumed that σ_1 passes e_3 on the right side. This means that σ_1 must cross the horizontal part of σ_2 above ω_1. If there is a vertical U-turn at e_3, then there must be a blocker ω_2 to the left of σ_2 (see Figure 2.17(c)). Because $\sigma_1 \sim_h \pi_1$, σ_1 must

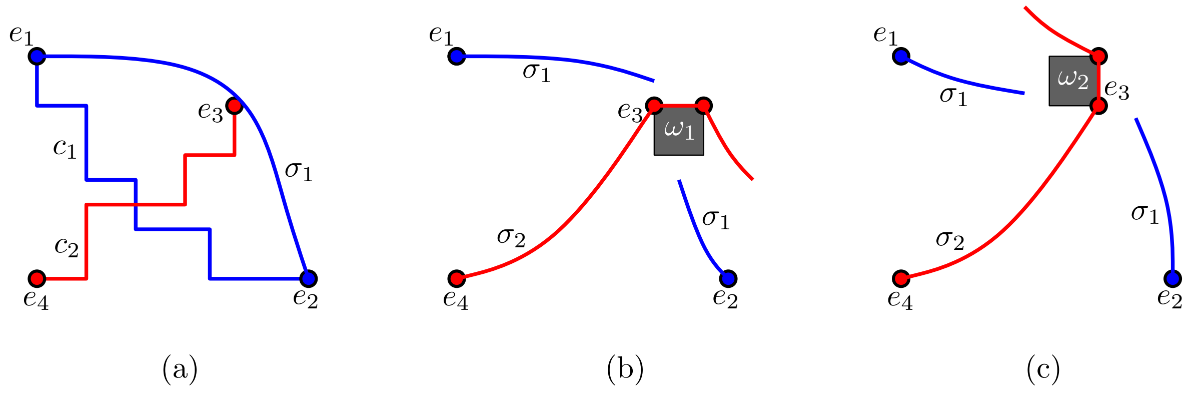

Figure 2.17: Positive and negative staircase chains with tight U-turns cannot cross.

pass ω_2 along the bottom. But we assumed that σ_1 passes e_3 along the top. This means that σ_1 must cross the vertical part of σ_2 to the right of ω_2. So the paths σ_1 and σ_2 are crossing. Contradiction. □

Using Lemma 9 we can simply split the staircase chains up into two sets: the positive staircase chains and the negative staircase chains. Then we can just use the algorithm Untangle of Chapter 2.3 to solve the problem for these two collections of staircase chains independently. After that we can reconnect the chains to form the intended rectilinear paths, which are then non-crossing. Note however that the (crossing) staircase chains obtained by this method are in general not locally optimal unless we started with locally optimal paths. This is not a problem though, because it is relatively easy to find locally optimal paths that are homotopic to the staircase chains of the paths. This will be further discussed in Chapter 2.5.

Unfortunately the algorithm Routing as described above does not compute a 2-approximation. To make the algorithm Routing compute a 2-approximation, we need to consider the number of reroute boxes of all paths. In Theorem 6 we used the fact that a positive staircase path with x links can have at most $x/2$ reroute boxes, one for each lower-left bend. We can do better when we consider all cases (see Figure 2.18). The number of lower-left bends depends on the type of the first and last bend. For each of these bends that is not a lower-left bend, we can reduce the number of lower-left bends (and hence reroute boxes) by 1/2. We can do the same analysis for negative staircase chains. If these chains are also added from left to right using the algorithm of Chapter 2.3, then the reroute boxes are at the upper-left bends. Now consider all possible U-turns (see Figure 2.19). Note that for the horizontal U-turns there is always one neighboring chain starting/ending with a type of bend without a reroute box. Unfortunately this is not true for a left U-turn. However for right U-turns both neighboring chains start/end with a type of bend without a reroute box. So the right U-turns could compensate for the left U-turns, but then there should be more right U-turns than left U-turns. Unfortunately this is not always the case. We can however add the paths from right to left instead of from left to right. In that case the reroute boxes would be at the upper-right and lower-right bends. This swaps the roles of the right and left U-turns. So by making the right choice on the order of how the paths are added, the vertical U-turns can compensate each other. This means that on average for every U-turn the number of reroute boxes can be reduced by 1/2 (from the normal bound for reroute boxes).

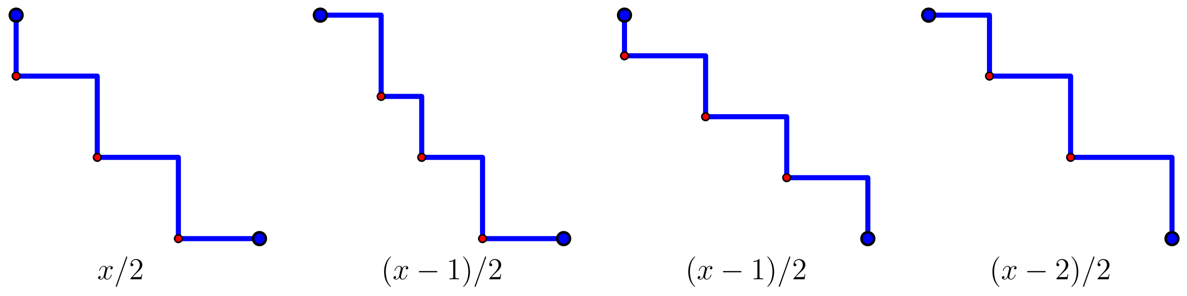

Figure 2.18: Positive staircase paths with different first/last bends. The number of lower-left bends with respect to the number of links x is shown.

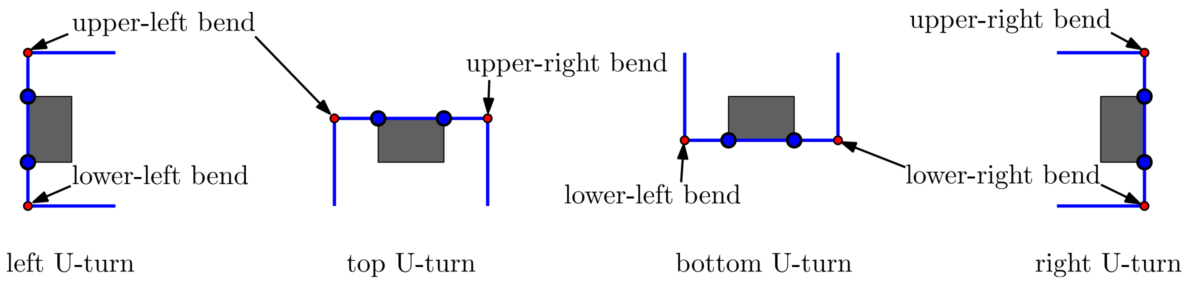

Figure 2.19: Different types of U-turns and their neighboring types of bends.

Theorem 10 *The algorithm* Routing *correctly computes a 2-approximation for the routing problem.*

Proof. We first need to prove that the resulting paths are non-crossing and homotopic to the original paths. First of all the resulting staircase chains of the paths are non-crossing and have the same homotopy class due to theorem 6. Making the U-turns tight also did not change the homotopy classes of the paths. So the resulting paths are homotopic to the original paths. Although the staircase chains are non-crossing, there could in theory still be a crossing at a U-turn. This is not the case, because there is a clear order on the y-monotone chains (homotopic chains are bundled) and the homotopy classes are non-crossing. So the resulting paths are non-crossing. Finally we must prove that this algorithm computes a 2-approximation. We know that two links must be added for each reroute box. Let L be the total number of links for all paths and let U be the total number of U-turns. Note that the total number of links for all staircase chains is $L + U$ due to the splitting (the middle link is not used). Like in Theorem 6, we can bound the number of reroute boxes by $(L + U)/2$, but as mentioned above this bound can be reduced by $U/2$ so we get $L/2$. Untangling the staircase paths using the algorithm Untangle results in at most $L + U + 2L/2 = 2L + U$ links. Afterwards we can reconnect the links at the U-turns which reduces the number of links by U. So the resulting number of links is $L^{'} \leqslant 2L + U - U = 2L$. Because L can be the number of links of locally optimal paths, this means that the algorithm Routing computes a 2-approximation if the initial paths are locally optimal. Note however that the tight U-turns are always locally optimal so that we need to make only the staircase chains locally optimal. □

2.5 Efficient algorithm

In this chapter we give an indication of how to make an efficient implementation of the algorithm Routing presented in chapters 2.3 and 2.4. The focus of our research was however not on finding an efficient implementation for this algorithm, so in this chapter we just give some ideas on how the algorithm can be implemented efficiently.

The algorithm Routing mainly consists of the following parts.

1. Making the U-turns tight and splitting up the paths into staircase chains.
2. Ordering and bundling the y-monotone chains.
3. Finding locally optimal paths for staircase chains.
4. Untangling the staircase chains (the algorithm Untangle).

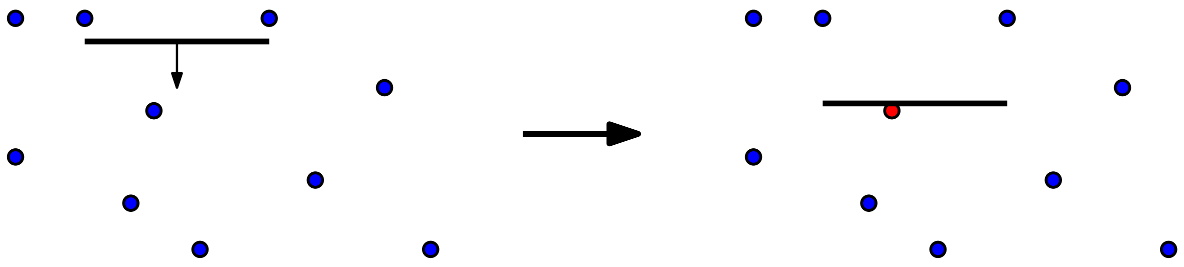
Figure 2.20: Segment dragging.

We discuss efficient implementations for these parts separately. In this chapter we assume that the input consists of n paths with a total of k links (for all paths) and m rectangular obstacles. So the number of blockers is $2n + m$. We also assume that the paths are stored as a linked list of bends.

Tight U-turns. In Chapter 2.4 we discussed the algorithm for making the U-turns tight. This involves moving the middle link of a U-turn to the inside until it hits a blocker or it simplifies the path. Now we want to do this efficiently. This can be solved using segment dragging. The segment dragging problem is defined as follows. Given a collection of points in the plane, pick an arbitrary horizontal line segment and move it vertically down until it hits one of the points if any. So an algorithm for segment dragging reports the first point hit by any horizontal line segment (see Figure 2.20). An efficient data structure for segment dragging is given by Chazelle in [5]. The algorithm uses $O(n)$ storage and $O(n \log n)$ preprocessing time and can answer queries in $O(\log n)$ time, where n is the number of points. Although the problem is defined for moving horizontal line segments down, this could easily be adapted to work for horizontal line segments moving up or for vertical line segments moving left and right (we just use multiple data structures).

The point set for segment dragging must be formed by our blockers. For endpoints this is simple, but for the rectangular obstacles we have to use the four corner points. Note that it is impossible to "miss" the rectangular obstacles by dragging a segment in between these points, because in that case the path would be crossing the rectangular obstacle. We look for all U-turns by searching linearly through the paths and make the U-turns tight using segment dragging. This can require simplifying the path, but this can be done locally. Also, this can add new U-turns only locally. Because the point set has size $O(m + n)$ (the blockers) and we need to do at most $O(k)$ queries, making the U-turns tight can be done in $O(k \log(n + m))$ time plus $O((m + n) \log(n + m))$ time for preprocessing. Note however that this can also be done in $O((m + n)k)$ time if we do not use a specialized data structure (by simply checking all points). Finally, splitting up the paths into staircase chains afterwards is trivial (at least if we store the tight blockers with the U-turns).

Ordering and bundling y-monotone chains. The first step is to order the y-monotone chains. The idea is to order the chains along with the obstacles. This makes bundling homotopic chains easy, because we just have to check chains that are next to each other in the order. So the only problem that remains is to order the chains.

To order the chains, we need the following observation. If two y-monotone chains are not homotopic, then there is at least one blocker that is in between the two chains or the endpoints of the chains are different. We know that there are only $O(m+n)$ blockers, but also the number of different endpoints of the chains is $O(m+n)$, because these endpoints can be only at the endpoint of a path or at the corner of a rectangular obstacle. The number of y-monotone chains can be bounded by $O(k)$. We can build a graph representing the partial order of the paths and obstacles. The nodes of this graph are the paths and the blockers. We add a directed edge from a blocker node to a path node if the blocker is left from the path and a directed edge from a path node to a blocker node if the blocker is right from the path. Note that we can form this graph by simply checking every link with every blocker in $O((m+n)k)$ time. This also means that the graph has only $O((m+n)k)$ edges. With this graph we can use topological sorting to find an order on the paths and obstacles in $O((m+n)k)$ time. Unfortunately this algorithm also uses $O((m+n)k)$ storage. Although this algorithm can probably be more efficient (especially its storage use), we settle for this algorithm and we leave any improvements as an open problem.

We should note that bundling these y-monotone chains is actually very convenient. That is because it is proven by Efrat et al. in [8] that there are only $O(m+n)$ y-monotone chains with different homotopy classes. So bundling has reduced the number of y-monotone chains from $O(k)$ to $O(m+n)$.

Locally optimal staircase paths. Now we describe how to efficiently compute the locally optimal staircase paths. Note that in Chapter 2.3 it was mentioned that for positive staircase paths, the locally optimal paths must be downmost rightmost. It turns out that making a path downmost rightmost makes the path locally optimal. To be precise, a downmost rightmost positive staircase path is locally optimal if the first link is correct (horizontal or vertical). Note that for chains starting at U-turns, the first link is already dictated by the U-turn. So this choice is only relevant for endpoints of a path in which case we can just try both possibilities.

Lemma 11 *A positive staircase path that is downmost rightmost is locally optimal depending on the first link.*

Proof. Without loss of generality we assume that the first link is optimal and is vertical. Consider the next horizontal link. For a downmost rightmost path π this next horizontal link is at the height of the highest blocker the path needs to pass along the top (or the endpoint)

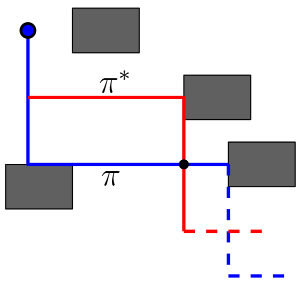

Figure 2.21: The downmost rightmost positive staircase path is locally optimal.

that is to the right of the first vertical link (else the path would not be a staircase path). Assume this choice does not lead to a locally optimal path. Then a locally optimal path π^* must have the horizontal link higher, because it cannot be lower (see Figure 2.21). Now consider the next vertical link of π^* (there must be one). Note that this link is crossed by the horizontal link of π, because else the next vertical link is not rightmost (there must be a next link or π^* is not locally optimal). At that crossing, path π can follow π^* and then π would be locally optimal. This is a contradiction. We can continue this argumentation for all links of π. This means that π must be locally optimal. □

To make the positive staircase paths downmost rightmost, we can use segment dragging again. This works the same as for making U-turns tight. Either the link becomes downmost or rightmost or we can simplify the path. Using segment dragging we get the downmost rightmost locally optimal paths. So the time required for this is the same as the time required to make the U-turns tight.

Untangling chains. The only problem for performing the algorithm UNTANGLE efficiently is to determine (efficiently) which links are involved in a crossing. The algorithm is as follows. The chains are added incrementally from left to right as is described in Chapter 2.3. Let the next chain to be added be chain i. The links of chain i are added incrementally from top to bottom. Note that we do not have to sort the links, because they are stored in a linked list for each chain and the chains are y-monotone. We know that a crossing can occur only with a vertical link of chain i. The only thing we need to check for a vertical link is which of the links of the previously added chains cross this link. So for each vertical link ℓ of chain i, we follow the links of the paths j $(1 \leqslant j < i)$ until they pass the next horizontal link of chain i. Now we can easily deduce which links cross ℓ and then it is straightforward to remove the crossings using the subroutine GROWRECTANGLE as is described in Chapter 2.3.

Note that as we add links incrementally from top to bottom, we can handle the different staircase chains of the same y-monotone chain consecutively. So basically we can perform the algorithm on y-monotone chains directly. We need only to keep track of which type of staircase path we are currently handling. This means we need to move from top to bottom only as often as there are y-monotone chains. As mentioned before, there are only $O(m+n)$ different y-monotone chains.

Lemma 12 *The algorithm* UNTANGLE *runs in* $O((m+n)k)$ *time.*

Proof. As mentioned above, we need to add $O(m+n)$ y-monotone chains incrementally. For each y-monotone chain we need to move from top to bottom to remove the crossings. Assume we have to add chain i and let k_i be the number of vertical links of chain i. For each vertical link ℓ_j, we need to follow the links of the previously added paths until they pass the horizontal link after ℓ_j. This costs $O(i + x_j)$ time, where x_j is the total number of links followed. Then we must remove the crossings using the subroutine GROWRECTANGLE. As is shown in Chapter 2.3, only a constant number of links is involved in this calculation for each chain. So GROWRECTANGLE runs in $O(i)$ time. Because $\sum_{j=1}^{k_i} O(x_j) = O(k)$, adding chain i costs $\sum_{j=1}^{k_i} O(i + x_j) = O(ik_i + k)$ time. Summing this up for all chains results in $\sum_{i=1}^{O(m+n)} O(ik_i+k) \leqslant \sum_{i=1}^{O(m+n)} O((m+n)k_i+k) = O((m+n)k)$. So the algorithm UNTANGLE runs in $O((m+n)k)$ time. □

Total algorithm.

Theorem 13 *The algorithm* Routing *runs in $O((m+n)k)$ time and uses $O((m+n)k)$ storage.*

Proof. According to Lemma 12 untangling the paths takes $O((m+n)k)$ time. The same holds for ordering and bundling the y-monotone chains. Although making the U-turns tight and finding the locally optimal staircase paths can be done more efficiently, we can do this in $O((m+n)k)$ time without using a specialized data structure. Finally note that we need more than linear storage only for ordering the y-monotone chains, which requires $O((m+n)k)$ storage. So the presented algorithm runs in $O((m+n)k)$ time and uses $O((m+n)k)$ storage. □

2.6 Lower bound

In Chapter 2.3 we presented the algorithm Untangle that computes a 2-approximation for the routing problem on positive staircase paths. In this chapter we show that we cannot improve the approximation factor 2 if we use the locally optimal paths as lower bound. In other words, we show that untangling positive staircase paths might double the number of links.

Consider the example shown in Figure 2.22 with n paths. Note that every path is locally optimal and uses exactly 3 links. The drawn paths are also the only locally optimal paths.

Lemma 14 *The example in Figure 2.22 requires at least $5n-2$ links to remove all crossings.*

Proof. First note that if we want to add links to a path, then we have to add at least 2 links. So if we add links to all paths, then that would result in a total of at least $5n$ links, which is more than $5n-2$. So we leave at least one path as it is. Every other path must cross this path. To remove these crossings we need to add links to all other paths. This results in a total number of links of at least $3+5(n-1)=5n-2$ links. □

Unfortunately Lemma 14 does not show that we sometimes need to double the number of links. But we can repeat the construction of Figure 2.22 x times like is shown in Figure 2.23.

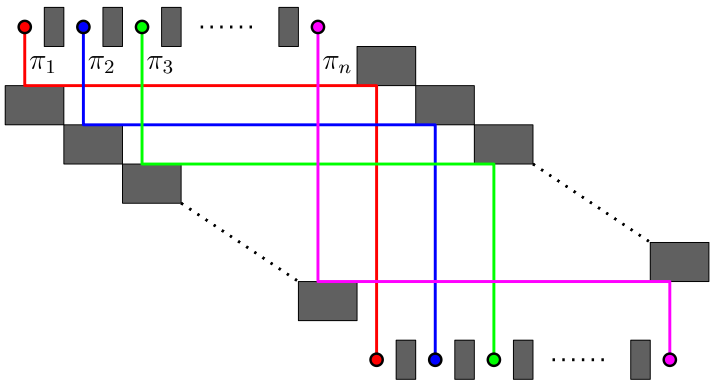

Figure 2.22: An example with fixed locally optimal paths.

The endpoints can be replaced by two obstacles next to each other. Using two obstacles next to each other and forcing the path to go in between two obstacles, we can ensure that the path passes through a specific point.

Theorem 15 *Untangling paths can require doubling the total number of links.*

Proof. Consider the construction of Figure 2.23. Note that the total number of links is $(2x+1)n$. Using Lemma 14 it is easy to see that removing the crossings requires at least $2x+1+(4x+1)(n-1) = (4x+1)n - 2x$ links. Dividing the total number of links without crossings by the total number of links of the locally optimal paths, we get the following.

$$\begin{aligned}\frac{(4x+1)n-2x}{(2x+1)n} &= \frac{4x+1}{2x+1} - \frac{2x}{(2x+1)n} \\ &= 2 - \left(\frac{1}{2x+1} + \frac{2x}{(2x+1)n}\right) \\ &= 2 - \left(\frac{1}{n} + \frac{n-1}{n} \cdot \frac{1}{2x+1}\right) \\ &\geqslant 2 - \left(\frac{1}{n} + \frac{1}{2x+1}\right)\end{aligned}$$

We can let n and x grow to infinity, which results in $\lim_{n,x\to\infty} 2 - (\frac{1}{n} + \frac{1}{2x+1}) = 2$. □

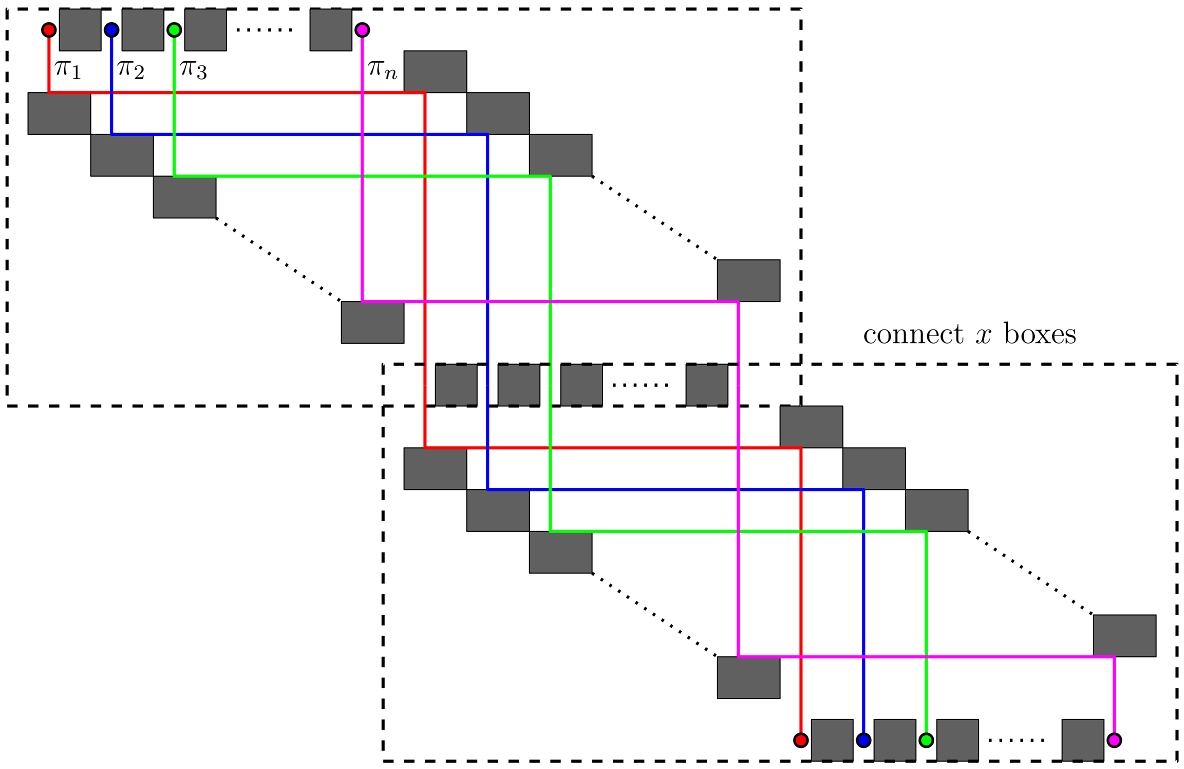

Figure 2.23: An example which requires to double the number of links. Instead of 2 boxes, x boxes should be connected.

Chapter 3

Embedding matchings with regions

For some problems it is not necessary to restrict the endpoints to a fixed location in the plane. Sometimes it is good enough if the endpoints are in some region. For example in a schematic map about connections between countries, it is sufficient that the endpoints are within the country they represent. That is why in this chapter we consider problems where the endpoints are not restricted to a fixed location, but instead to a region. We demonstrate that this extra freedom provided by the regions makes the problems more complicated. We do this by showing that VERTICAL MATCHING is NP-Complete. For convenience we repeat the problem definition here.

We are given a collection of $2n$ vertical segments R_i ($1 \leqslant i \leqslant 2n$) which each consist of a triplet $R_i = (x_i, m_i, M_i)$ and represent the vertical segments at x-coordinate x_i that is between m_i and M_i, which are the minimum and maximum y-coordinates respectively. We want to connect region R_{2j-1} with region R_{2j} with a straight line segment for $1 \leqslant j \leqslant n$. The decision problem becomes the following. Are there y-coordinates y_i for each region R_i satisfying $m_i \leqslant y_i \leqslant M_i$ such that the line segments created by using the coordinates (x_i, y_i) are non-crossing?

Note that this problem is trivial when the endpoints are restricted to a fixed location. After proving the NP-Completeness of VERTICAL MATCHING, the result is extended to classes of regions other than vertical segments as well. We call this class of problems REGION MATCHING.

3.1 Vertical Matching is NP-Complete

In this chapter we prove that VERTICAL MATCHING is NP-Complete. We do this using a reduction from the problem PLANAR 3-SAT to VERTICAL MATCHING. The problem PLANAR 3-SAT is a variant of the problem 3-SAT. These problems are defined as follows.

Definition 8 *An instance of the problem* 3-SAT *consists of n variables x_i and m clauses c_j. A variable x_i or its negation $\overline{x_i}$ is called a literal. Each clause consists of 3 literals. The problem is to decide if we can assign truth values to the variables x_i such that for each clause there is at least one true literal.*

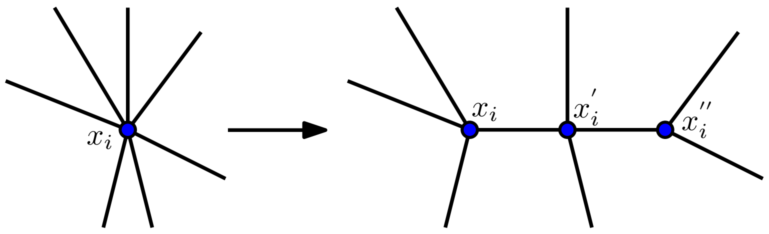

Figure 3.1: Splitting variable nodes.

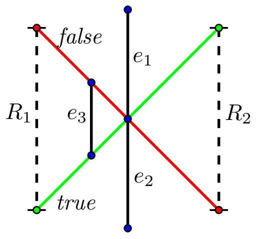

Figure 3.2: An edge gadget.

Definition 9 *The* Planar 3-SAT *problem is like the ordinary* 3-SAT *problem, except that the variables* $\mathcal{X} = \{x_1, \ldots, x_n\}$ *and clauses* $\mathcal{C} = \{c_1, \ldots, c_m\}$ *have the added restriction that the graph* $G = (V, E)$ *with* $V = \mathcal{X} \cup \mathcal{C}$ *and* $E = \{(x, c) \mid x \in \mathcal{X} \wedge c \in \mathcal{C} \wedge \text{“}x \text{ occurs in } c\text{”}\}$ *is planar.*

We should note that Planar 3-SAT is often defined differently, but to prove the NP-Completeness of Vertical Matching, this definition is sufficient. Using this definition we also get the following result from [13].

Lemma 16 Planar 3-SAT *is NP-Complete.*

The reduction of Planar 3-SAT to Vertical Matching is roughly as follows. We take the planar embedding of an instance of Planar 3-SAT and transform this to an orthogonal drawing. Then we use this orthogonal drawing to construct an instance of Vertical Matching, which can be drawn without crossings iff the original Planar 3-SAT problem instance is satisfiable.

The planar embedding can be transformed into an orthogonal drawing using the algorithm given in [10]. This requires the graph to have a maximum degree of 4, because else an orthogonal drawing is not possible. All clause nodes have degree 3 (or less), but this is not the case for the variable nodes. We can however split these nodes x_i into a collection of nodes $x_i, x_i^{'}, x_i^{''}, \ldots$ as shown in Figure 3.1, until all nodes have a degree of at most 4. After that we can construct an orthogonal drawing.

Now we need to construct a problem instance of Vertical Matching using this orthogonal drawing. This consists of variable gadgets and clause gadgets. To construct the variable gadget of variable x_i, we use the variable nodes $x_i, x_i^{'}, x_i^{''}, \ldots$ and the neighboring edges of the orthogonal drawing. To construct the clause gadget of clause c_j, we use the clause node of c_j and the neighboring edges.

Variable gadgets. To construct the variable gadgets, we need to build part of an orthogonal drawing as an instance of Vertical Matching such that it has exactly two valid configurations. These two configurations represent the two possible truth values *true* and *false*. We first do this for a single edge. Consider the construction shown in Figure 3.2. It consists of 3 edges e_1, e_2, and e_3 that are fixed (using regions with $m_i = M_i$) and two regions R_1 and R_2 that need to be connected. Note that the fixed edges e_1 and e_2 touch in the middle. Because edges are allowed to touch, the edge connecting R_1 and R_2 can go (only) through the point where e_1 and e_2 touch. This in combination with the fixed edge e_3 makes that

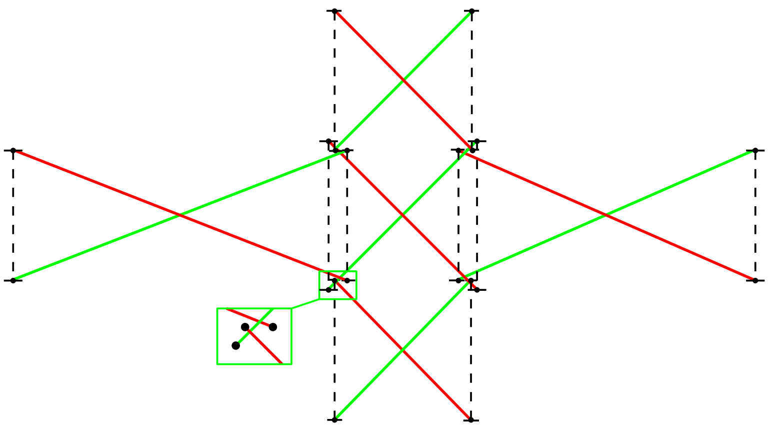

Figure 3.3: A connection gadget.

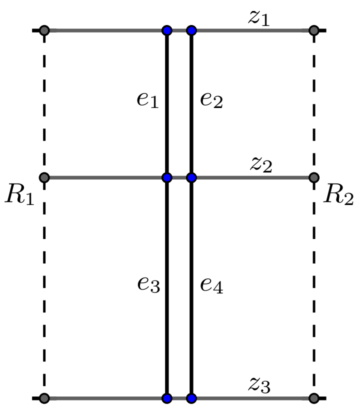

Figure 3.4: Construction for clauses.

there are only two possibilities to connect R_1 and R_2. The only two possibilities are an edge from the bottom of R_1 to the top of R_2 (which is associated with *true*) and an edge from the top of R_1 to the bottom of R_2 (which is associated with *false*). Using this edge gadget we can build a variable gadget. We need only to connect the edge gadgets at variable nodes or bends. This can be done using the construction shown in Figure 3.3. This construction has one edge gadget in the middle and four edge gadgets extending in four different directions (up, down, left and right). It is easy to see that this construction is crossing unless all edge gadgets use *true* edges or all edge gadgets use *false* edges. Finally note that the edge gadgets can be arbitrarily scaled. So the construction shown in Figure 3.3 can be used to construct the variable gadgets such that it has only two valid configurations.

Clause gadgets. For the clause gadgets we use the construction shown in Figure 3.4. It consists of 2 regions R_1 and R_2 that need to be connected and 4 fixed edges e_1, e_2, e_3 and e_4 in between. The only possible ways to connect R_1 and R_2 are a horizontal line above e_1

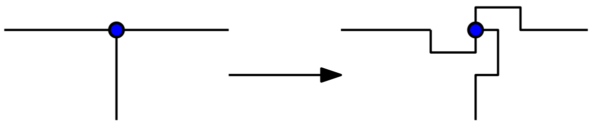

Figure 3.5: Correcting clause node.

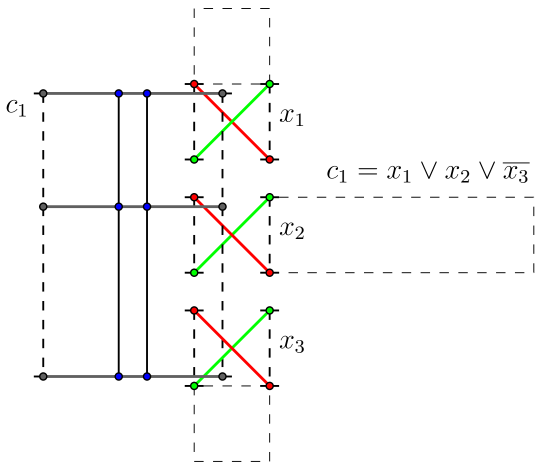

Figure 3.6: Combining clause and variable gadgets.

and e_2 (z_1), a horizontal line below e_3 and e_4 (z_3), or a horizontal line below e_1 and e_2 and above e_3 and e_4 (z_2). So this construction has exactly 3 valid configurations. To complete the clause gadget for a clause c_j, we need to connect the construction shown in Figure 3.4 to the variable gadgets of the variables involved in c_j. We do this as shown in Figure 3.6. If x_i occurs in the clause, then one of the horizontal edges must have a crossing with the *false* edge of x_i. If $\overline{x_i}$ occurs in the clause, then one of the horizontal edges must have a crossing with the *true* edge of x_i. Note that for the construction shown in Figure 3.4, the y-coordinates for all possible horizontal edges z_1, z_2 and z_3 can be chosen to align with the proper *true* or *false* edges. The clause gadget shown in Figure 3.6 requires that a clause node has two vertical incident edges and one horizontal incident edge. If this is not the case, then this can easily be changed as shown in Figure 3.5.

Lemma 17 Vertical Matching *is NP-Hard.*

Proof. Consider an instance of Planar 3-SAT. Take the corresponding planar graph and split the variable nodes as shown in Figure 3.1. Next construct an orthogonal drawing from this graph using the algorithm given in [10]. Then for every boolean variable x_i ($1 \leqslant i \leqslant n$), construct the corresponding variable gadget as described above. For every clause c_j ($1 \leqslant j \leqslant m$), construct the corresponding clause gadget and connect it to the variable gadgets as described above. If the original Planar 3-SAT instance is satisfiable, then we can use this satisfiable assignment to set the *true* or *false* edges. Because it is a satisfiable assignment, every clause contains at least one true literal. The horizontal edge of the clause gadget (z_1, z_2 or z_3) can be chosen to be at the true literal. Because the literal is true, this does not result in a crossing. If the original Planar 3-SAT instance is not satisfiable, then for every valuation of the boolean variables there is at least one clause where all literals are false. This means that every horizontal edge for that clause gadget (z_1, z_2 and z_3) results in a crossing, which means that the Vertical Matching instance cannot be drawn without crossings. Because of Lemma 16 and because the reduction can be done in polynomial time, Vertical Matching is NP-Hard. □

To show that Vertical Matching is also NP-Complete, we need to show that Vertical Matching is in NP. Note that it is very easy to check whether there is a crossing or not, given the y-coordinates y_i ($1 \leqslant i \leqslant 2n$). But we also need to show that these y-coordinates y_i can be represented by a polynomial number of bits. In order to show this, we look at the problem differently.

Vertical Matching requires some constraints on the y-coordinates y_i. First consider the constraint that the line segments must be non-crossing. Assume we have two line segments, one between a_1 and b_1 and one between a_2 and b_2. Let the infinite line through a_1 and b_1 be ℓ_1 and the infinite line through a_2 and b_2 be ℓ_2. Note that if both a_2 and b_2 are on the same side of ℓ_1, then the line segments are non-crossing (Figure 3.7(a)). This also holds if both a_1 and b_1 are on the same side of ℓ_2 (Figure 3.7(b)). It is important to see that if both of these properties do not hold, then the two line segments are crossing (Figure 3.7(c)). So the constraint of being non-crossing can be satisfied if either a_2 and b_2 are on the same side of ℓ_1 or a_1 and b_1 are on the same side of ℓ_2. Note that the constraint of a point being on one side of a line is a linear constraint. So checking if two line segments are non-crossing is a boolean formula on linear constraints. The constraint that $m_i \leqslant y_i \leqslant M_i$ is also clearly linear.

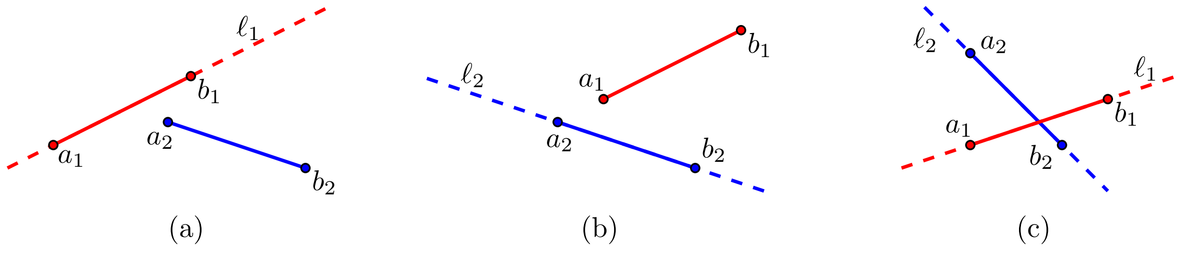

Figure 3.7: The line segments intersect iff ℓ_1 separates a_2 and b_2 and ℓ_2 separates a_1 and b_1.

Theorem 18 Vertical Matching *is NP-Complete.*

Proof. As described above the problem consists of finding a vector $(y_1, y_2, \ldots, y_{2n})$ satisfying a boolean formula on linear constraints. Assume the problem has a solution $(y_1, y_2, \ldots, y_{2n})$. This solution must be in a $2n$-dimensional cell bounded by the hyperplanes representing the linear constraints (or it is on an intersection of hyperplanes). These cells are bounded due to the constraints $m_i \leqslant y_i \leqslant M_i$. We can just choose a solution that is on one of the corner points of the cell. Note that this solution is on an intersection of hyperplanes. Because the intersection of a collection of $2n$ hyperplanes can be represented by a polynomial number of bits, the solution of a problem instance of Vertical Matching can be as well. Because given the vector $(y_1, y_2, \ldots, y_{2n})$ the linear constraints can easily be checked, the problem is in NP. Due to Lemma 17 the problem is also NP-Hard. So Vertical Matching is NP-Complete. □

3.2 Region Matching is NP-Hard

We now extend the result of Chapter 3.1 to show that Region Matching is NP-Hard. We will not prove that Region Matching is also NP-Complete, because this is not as relevant. We show that Region Matching is NP-hard in many cases by adapting the proof in Chapter 3.1 to work for other classes of regions. Before we can do this, we need the following lemma.

Lemma 19 *For any construction of Lemma 17 with a crossing, there is an $\epsilon > 0$ such that allowing every endpoint to move a distance of ϵ cannot remove a crossing.*

Proof. Let δ be the shortest distance between an edge e and an endpoint of another edge $e^{'}$ which crosses e (see Figure 3.8). In order to remove the crossing between e and $e^{'}$, the endpoint of e needs to move to the other side of the line through $e^{'}$. This can only happen

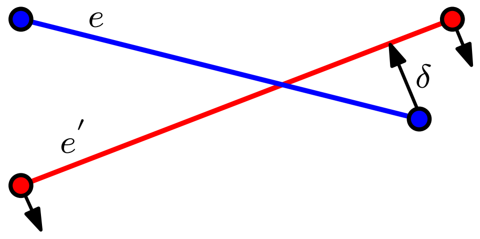

Figure 3.8: Minimum distance of node to edge.

if the endpoints are allowed to move a distance of at least $\delta/2$ (see Figure 3.8). Let $\delta_{\min}$ be the minimum of all δ. Then no crossing can be removed if $\epsilon < \delta_{\min}/2$. Now it is enough to show that $\delta_{\min} > 0$, but this is obviously true, because edges that touch are allowed and are not considered crossing. □

Using Lemma 19 it is easy to adapt the proof of Chapter 3.1 to work for other classes of regions.

Theorem 20 REGION MATCHING *for classes of regions that can be arbitrarily sized and are connected is NP-hard.*

Proof. Note that if the class of regions is connected and can have arbitrary width and height, then vertical segments belong to this class and the problem is clearly NP-Complete by Lemma 17. Now assume that the width to height ratio is constant (for example with squares or discs). Consider the construction of Lemma 17. Because the regions can have arbitrary size, the fixed edges of Figure 3.2 and Figure 3.4 can still be constructed. Replace the vertical line segments by very thin versions of the given class of regions. This does not remove any crossings due to Lemma 19. Now we can simply scale the entire construction horizontally until the regions have the correct width to height ratio. Scaling does not remove or introduce any crossings. Using this construction and following the proof of Lemma 17, we can conclude that REGION MATCHING for the mentioned classes of regions is also NP-Hard. Finally note that the regions need to be connected, because the thin versions of the regions must resemble vertical line segments. □

The proof in Chapter 3.1 can be adapted to more classes of regions than is proven in Theorem 20 (for example regions that are not connected). We do however think that the result of Theorem 20 is sufficient to demonstrate that REGION MATCHING for many classes of regions is NP-Hard. It is however required that these classes of regions contain point sized regions for the proof to work.

Chapter 4

Conclusion

In this thesis we have considered the problem of drawing non-crossing paths with fixed endpoints in the plane. This problem can also be seen as that of embedding graphs in the plane without crossings in the case that the positions of the nodes are fixed. To keep it simple we considered only matchings though, but this can be seen as a first step towards embedding general graphs with fixed nodes.

The problem of finding non-crossing paths with fixed endpoints is a very important problem in the field of VLSI design. A lot of research has been done on many different variants of this problem. More recently these kinds of problems have also been considered in the field of cartography. For example when drawing schematic maps, the nodes are often fixed or almost fixed and the edges should not be crossing. So the problem considered in this thesis is relevant for both VLSI design and cartography.

The research done in the field of VLSI design has indicated that many variants of finding non-crossing paths with fixed endpoints are intractable and also hard to approximate. That is why the problem is often simplified to use given homotopy classes. This simplification also makes sense in the field of cartography, because the way a connection winds around interesting features of a map is important to understand the map and should therefore not be changed. That is why in Chapter 2 we considered the problem of finding non-crossing rectilinear paths of given homotopy class. Note that the choice of using rectilinear paths also makes sense for both application fields, but it also simplifies the problem. We have also chosen to minimize the number of links, because this variant has not been considered very often and the number of links is directly connected to the complexity of the paths.

In Chapter 2 we have presented an approximation algorithm for the problem of finding non-crossing rectilinear paths of given homotopy class minimizing the total number of links. We first gave a 2-approximation for positive (or negative) staircase paths and then we reduced the problem for general rectilinear paths to the problem for staircase paths. This also resulted in a 2-approximation. After that we presented some ideas on how to calculate this efficiently and presented an algorithm that runs in $O((m+n)k)$ time and uses $O((m+n)k)$ storage, where n is the number of paths, m is the number of obstacles and k is the total number of links. Finally we have shown that the factor 2 of the approximation algorithm is the best we can achieve using the locally optimal paths as a lower bound.

In Chapter 3 we have considered the problem of finding non-crossing paths with the endpoints restricted to a region instead of being fixed. We have shown that the additional freedom provided by the regions usually makes the problems only harder. We demonstrated this by proving that a problem that is trivial for fixed endpoints is actually NP-Complete for endpoints restricted to regions.

4.1 Open problems

We have presented an approximation algorithm for the routing problem. Unfortunately we do not know if this problem is NP-Hard or not. If the problem turns out to be NP-Hard, then we can also consider if the approximation factor of 2 is the best we can do in polynomial time (note that we have shown the factor 2 only to be optimal w.r.t. the locally optimal paths). Another open problem is that of finding a more efficient algorithm than the one presented in Chapter 2.5. We think that ordering the y-monotone chains can be done more efficiently and can especially use less storage.

In the routing problem the paths are allowed to overlap. In many applications it is better to have some distance in between the paths. This is equivalent to finding thick disjoint paths. We would like to extend our algorithm for the routing problem to also work for thick paths or paths with a given distance in between.

It has been mentioned that the problem considered in this thesis is relevant for cartography, for example when drawing schematic maps. Schematic maps do not contain only paths, but usually form a network or graph. So we would like to extend the algorithm for the routing problem given in Chapter 2 to work for general graphs instead of just for a collection of paths. Furthermore it might sometimes be too much a restriction to require the paths to be rectilinear. Sometimes we would like to use the more general c-oriented paths. So we would like to adapt the algorithm for the routing problem to work for c-oriented paths instead of rectilinear paths.

Although we have demonstrated in Chapter 3 that restricting endpoints to regions instead of using fixed endpoints makes problems harder, we only demonstrated this using a problem for which the homotopy classes are not fixed. We could also consider the routing problem with regions. Note that this does not matter for untangling the paths, because untangling no more than doubles the number of links regardless of where the endpoints are. Finding the locally optimal paths with the endpoints restricted to regions is more problematic though. We predict this problem is NP-Hard, but we leave it as an open problem.

Bibliography

[1] O. Bastert and S. P. Fekete. Geometrische Verdrahtungsprobleme. Technical Report 247, Mathematisches Institut, Universität zu Köln, 1996.

[2] S. Bespamyatnikh. Computing homotopic shortest paths in the plane. *Journal of Algorithms*, 49(2):284–303, 2003.

[3] S. Cabello. *Geometric problems in cartographic networks.* PhD thesis, Universiteit Utrecht, 2004.

[4] S. Cabello, M. de Berg, and M. van Kreveld. Schematization of networks. *Computational Geometry*, 30(3):223–238, 2005.

[5] B. Chazelle. An algorithm for segment-dragging and its implementation. *Algorithmica*, 11(1):205–221, 1988.

[6] R. Cole and A. Siegel. River routing every which way, but loose. In *Proc. 25th Annual Symposium on Foundations of Computer Science*, pages 65–73, 1984.

[7] C. A. Duncan, A. Efrat, S. G. Kobourov, and C. Wenk. Drawing with fat edges. In *Proc. 9th International Symposium on Graph Drawing (GD '01)*, LNCS 2265, pages 162–177. Springer, 2002.

[8] A. Efrat, S. G. Kobourov, and A. Lubiw. Computing homotopic shortest paths efficiently. *Computational Geometry: Theory and Applications*, 35(3):162–172, 2006.

[9] S. Gao, M. Jerrum, M. Kaufman, K. Mehlhorn, and W. Rülling. On continuous homotopic one layer routing. In *Proc. 4th Symposium on Computational Geometry*, pages 392–402, 1988.

[10] A. Garg and R. Tamassia. A new minimum cost flow algorithm with applications to graph drawing. In *Proc. 4th International Symposium on Graph Drawing (GD '96)*, LNCS 1190, pages 193–200. Springer, 1996.

[11] H. Gupta and R. Wenger. Constructing pairwise disjoint paths with few links. *ACM Transactions on Algorithms*, 3(3):26, 2007.

[12] J. Hershberger and J. Snoeyink. Computing minimum length paths of a given homotopy class. *Computational Geometry: Theory and Applications*, 4(2):63–97, 1994.

[13] D. Lichtenstein. Planar formulae and their uses. *SIAM Journal on Computing*, 11(2):329–343, 1982.

[14] M. Löffler. Existence of simple tours of imprecise points. In *Proc. 23rd European Workshop on Computational Geometry*, pages 22–25, 2007.

[15] M. Löffler. Existence of simple tours of imprecise points. Technical Report UU-CS-2007-003, Department of Information and Computing Sciences, Utrecht University, 2007.

[16] G. H. M. Abellanas, A. Aiello and R. I. Silveira. Network drawing with geographical constraints on vertices. In *Actas XI Encuentros de Geometra Computacional*, pages 111–118, 2005.

[17] F. M. Malley. *Single-layer wire routing and compaction.* MIT Press, Cambridge, MA, USA, 1990.

[18] D. Merrick and J. Gudmundsson. Path simplification for metro map layout. In *Proc. 14th International Symposium on Graph Drawing (GD'06)*, LNCS 4372, pages 258–269. Springer, 2006.

[19] M. Nöllenburg and A. Wolff. A mixed-integer program for drawing high-quality metro maps. In *Proc. 13th International Symposium on Graph Drawing (GD'05)*, LNCS 3843, pages 321–333. Springer, 2005.

[20] J. Pach and R. Wenger. Embedding planar graphs at fixed vertex locations. In *Proc. 6th International Symposium on Graph Drawing (GD'98)*, LNCS 1547, pages 263–274, 1998.

[21] V. Polishchuk and J. S. Mitchell. Thick non-crossing paths and minimum-cost flows in polygonal domains. In *Proc. 23rd Symposium on Computational Geometry*, pages 56–65, 2007.

[22] J. Takahashi, H. Suzuki, and T. Nishizeki. Algorithms for finding non-crossing paths with minimum total length in plane graphs. In *Proc. 3rd International Symposium on Algorithms and Computation*, LNCS 650, pages 400–409. Springer, 1992.

[23] J. Takahashi, H. Suzuki, and T. Nishizeki. Finding shortest non-crossing rectilinear paths in plane regions. In *Proc. 4th International Symposium on Algorithms and Computation*, LNCS 762, pages 98–107. Springer, 1993.

[24] C. D. Yang, D. T. Lee, and C. K. Wong. The smallest pair of noncrossing paths in a rectilinear polygon. *IEEE Transactions on Computers*, 46(8):930–941, 1997.