Homotopic Polygonal Line Simplification
Lasse Deleuran

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark
Homotopic Polygonal Line Simplification

A Dissertation
Presented to the Faculty of Science
of Aarhus University
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Lasse Deleuran
June 7, 2012
Abstract (English)

This thesis presents three contributions to the area of polygonal line simplification, or simply line simplification. A polygonal path, or simply a path is a list of points with line segments between the points. A path can be simplified by morphing it in order to minimize some objective function, such as the number of points or the length of path under some distance metric. Line simplification is the problem of simplifying one or more paths while preserving some restrictions on the morphing. We are primarily interested in problems where the morphing of paths does not intersect a set of obstacles. This is the intuition of the restriction of preserving homotopy, and simplification problems with this restriction are called homotopic. We present some recent advances within the area of homotopic line simplification while focusing on three areas of this kind of problems. We present a new contribution for each area:

- **Restricted simplification problems** have the restriction on a simplification that it must be a subset of the input path. Our contribution for these kind of problems are some algorithms that minimize the number of points of a path in the plane while staying homotopic to a set of obstacle points.

- **Unrestricted problems** are without the restriction of restricted problems; Points in a simplification can be placed anywhere. Our contribution is an algorithm that can be used to improve the best known running time of minimizing the Euclidean length of paths while staying homotopic to point obstacles being the endpoints of the paths. For an input consisting of $n$ paths with the total size of $m$, our algorithm improves the running time from $O(n \log^{1+\epsilon} n + m \log n)$ to $O(n \log^{1+\epsilon} n + m)$, where $\epsilon > 0$.

- **Heuristic algorithms** are simplification algorithm where the reduction based on the complexity measure is not necessarily optimal. Our contribution is the first practical algorithm to simplify paths using the famous simplification algorithm by Douglas Peucker [35] on large datasets while taking measures to maintain homotopy and avoid intersections. We present experimental results to support the assumption of practicality.
Denne afhandling præsenterer tre artikler indenfor området liniesimplifikation. En polygon-sti, eller blot sti er en liste af punkter med liniesegmenter mellem punkterne. En sti kan simplificeres ved at morfe den for at minimere en given objektfunktion, såsom antallet af punkter eller længden af stien under en given metrik. Liniesimplifikation er problemet at simplificere en eller flere stier mens en mængde begrænsninger på morfningsen overholdes. Vi er primært interesserede i problemer hvor stier ikke støder ind i forhindringer under morfningen. Dette er intuitionen bag begrænsningen kaldet homotopi, og simplifikationsproblemer med denne begrænsning kaldes homotopiske. Vi præsenterer nogle af de seneste fremskridt indenfor området vedrørende homotopisk liniesimplifikation og fokuserer på tre specifikke områder hvor vi præsenterer nye resultater inden for hvert område. De tre områder er:

- **Punktbegrænsede simplifikationsproblemer** har begrænsningen at punkterne i en simpliceret sti skal være blandt punkterne i en oprindelige sti. Vores bidrag til denne type problemer er nogle algoritmer som minimerer antallet af punkter mens hvert enkelt liniesegment i en simpliceret sti er homotopisk til den del af stien fra input som den spænder.

- **Ikke-punktbegrænsede simplifikationsproblemer** er, som navnet antyder, simplifikationsproblemer uden punktbegrænsningen. Vores bidrag er en algoritme som kan bruges til at forbedre udførelstiden af den hurtigst kendte algoritme til at minimere den euklidske længde af en mængde stier mens de er homotopiske til en mængde punkter bestående af stierens endepunkter. Givet et input bestående af $n$ stier med den samlede størrelse værende $m$, forbedrer vores algoritme den asymptotiske udførelstid fra $O(n \log^{1+\epsilon} n + m \log n)$ til $O(n \log^{1+\epsilon} n + m)$, hvor $\epsilon > 0$.

- **Heuristiske algoritmer** er simplifikationsalgoritmer som ikke nødvendigvis beregner den optimale sti i forhold til objektfunktionen. Vores bidrag er den første praktiske algoritme til at simplificere stier ved hjælp af den berømte Douglas Peucker simplifikationsalgoritme [35] på store datasæt mens der tages højde for at overholde homotopi, samt ikke at introducere skæringer mellem segmenterne af de simplificerede stier. Vi præsenterer desuden eksperimenter som understøtter de praktiske antagelser om vores algoritme.
Acknowledgements

I would like to thank my adviser Lars Arge for encouraging me to take a Ph.D. at the new research center MADALGO, for teaching me how to work in this first class research environment, and assisting me in my own research; Even through sickness has he woken up horribly early one morning in order to go through his remarks on how to make an almost hopeless amount of changes to my papers. Similarly, I would like to thank Mohammad Ali Abam for acting as my co-adviser when he was visiting MADALGO and Pankaj K. Agarwal for co-advising me during my stay at Duke University. I thank my other co-authors of the three papers in this thesis for working together with me with the objective of producing research that explores and improves upon current boundaries of Computer Science. Here I must also thank Henrik Blunch with whom I am working on a paper which is still a work in progress.

The people of MADALGO and Department of Computer Science at Aarhus University, especially Else Magård, Ellen Lindstrom and Gerth Brodal deserve special thanks for making it a pleasure to work at the center and answering all of my questions, many of which are barely research related. Similarly I thank the administrative staff and colleagues at the Department of Computer Science at Duke University, especially Jewel Wheeler, Thomas Mølhave and his wife Shannon, for making my visit to Duke University both enjoyable and memorable.

Special thanks goes to my officemate Konstantinos Tsakalidis who has tried to make sense of all of my ideas even when they were far from his primary field of research. Dorthe Haagen Nielsen deserves special thanks for helping me plan my studies when time seemed hopelessly short, and again when not meeting my deadlines and the lack of time was even more critical.

My family should be thanked for supporting me through my education. My mother Laila Kosecki and grandmother Elena Nielsen should especially be thanked for supporting me financially the last half year, where I have been working on my thesis after official fundings ended.

Finally, I would like to thank all my friends who have either helped me in finding corrections for this thesis or visited me at my office during the last couple of years.

Lasse Deleuran,
Aarhus, June 7, 2012.
## Contents

Abstract (English) v  
Abstract (Danish) vii  
Acknowledgements ix  

I Overview 1  

1 Line Simplification Problems in GIS and VLSI 5  
   1.1 Line Simplification Problems in GIS .................. 5  
   1.2 Line Simplification Problems in VLSI .................. 7  

2 Homotopic Line Simplification Problems 9  
   2.1 Unrestricted Line Simplification Problems ............. 11  
   2.2 Restricted Line Simplification Problems .............. 14  
   2.3 Heuristics ............................................. 17  

3 Simplification Problems in Various Areas 21  
   3.1 Types of Objects to be Simplified ...................... 21  
   3.2 Objective function .................................... 22  
   3.3 Constraints ............................................ 23  
   3.4 Obstacles ............................................ 23  

4 Conclusion 25  

II Papers 27  

5 Homotopic X-Shortest Paths 29  
   5.1 Introduction ........................................... 31  
   5.2 Homotopic X-Shortest Paths Algorithm ................. 33  
      5.2.1 Limbs .............................................. 33  
      5.2.2 Algorithm .......................................... 35  
   5.3 Lower Bound for Computing Homotopic X-Shortest Paths 42  
   5.4 Conclusion ............................................ 42
Part I

Overview
Introduction

Simplification of geometric objects is a well studied problem within the field of computational geometry. We are primarily interested in simplifying (polygonal) paths, which are lists of points with line segments between them; and polygons, which are paths where the first and last point overlap. A Line simplification problem (also known as a polygonal line simplification problem) is a problem where given one or more paths and a set of obstacles (being points, polygons or other geometric objects), to compute a simplified path or simplification for each input path which minimizes a given objective function and satisfies a set of constraints based on the input path and obstacles. To be able to morph two paths into each other without intersecting any obstacles during the morphing is the intuition behind homotopy and line simplification problems with the restriction of simplified paths being homotopic to the input paths are called homotopic line simplification problems.

Homotopic line simplification problems are of interest for areas such as cartography, geographic information systems (GIS), very large scale integration (VLSI), robot motion planning, and air traffic management (ATM). In Chapter 1 we present a brief overview of line simplification problems in the different areas with focus on GIS and VLSI as the problems for these areas give a nice general overview of the line simplification problems that arise in practice.

When line simplification problems have the restriction that all the points of a simplification must be present in the points of the path of the input, then the problem is restricted. Unrestricted line simplification problems are line simplification problems without this restriction. In Chapter 2 we present a formal definition of homotopic line simplification problems and focus on three kinds of homotopic line simplification problems: Unrestricted homotopic line simplification, restricted homotopic line simplification and heuristic algorithms, which are line simplification algorithms that do not minimize the objective function, but are expected to be efficient in practice.

Simplification problems is the superset of line simplification problems where the objects to be simplified are not necessarily paths. In Chapter 3 we present a short survey with an overview of a variety of simplification problems that arise in the areas which are only briefly mentioned in Chapter 1.

In Chapter 4 we present a summary of our contributions.

In Part II we present our papers. The Euclidean length of a path is the sum of the Euclidean lengths of its line segments\(^1\). Minimizing the Euclidean length

\(^1\)Euclidean length of a line with endpoints \((x_1, y_1)\) and \((x_2, y_2)\) is \(\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}\).
of a path while maintaining homotopy to a set of point obstacles corresponds intuitively to stretching a string on a board with pushpins acting as obstacles. The unrestricted line simplification problem of minimizing the Euclidean length of a set of paths where the obstacles are the endpoints of the paths is called Homotopic Shortest Paths. In Chapter 5 we present a new result for this problem which improves the running time of the best known solution.

The link length (or link distance) of a path is its number of segments and minimum link (path) problems are problems that seek paths with minimal link lengths. In Chapter 6 we present a set of algorithms computing minimum link paths while maintaining the restriction that every segment of a simplified path is homotopic to the part of the input path that it corresponds to. We also present a polynomial time algorithm that computes the homotopic minimum link path which proves that the problem is solvable in polynomial time.

When the size of data becomes extremely large, the time it takes to move data between main memory and disk becomes a significant factor for the running time of algorithms. I/O-efficient algorithms are algorithms designed to take this into account. In Chapter 7 we present the first practical I/O-efficient algorithm to simplify paths using the heuristic called Douglas Peucker’s line simplification algorithm[35]. Our algorithm takes measures to maintain homotopy and avoid intersections in the simplifications. We support our claim of practicality by presenting experimental results.
Chapter 1

Line Simplification Problems in GIS and VLSI

Simplification problems are of interest in areas such as cartographic generalization, where geometric objects, such as borders, are simplified while making sure that points of interest, such as cities, are not on the wrong side of a border after simplifying; GIS, which are systems for storing, manipulating, and displaying geographic data such as the data of interest in cartography; VLSI, where wires are routed on integrated circuits; Robot motion planning, where robots have to move efficiently without interfering with obstacles; and ATM, where flight routes are planned while taking weather systems into account. In this chapter we present some line simplification problems that arise in the areas of GIS, and VLSI as these problems represent a broad variety of the line simplification problems that are of general interest for us, and especially homotopic line simplification problems, which is the specific kind of simplification problems that are the subject of the next chapter and the papers of this thesis.

1.1 Line Simplification Problems in GIS

GIS is used in cartographic generalization, where it is of interest to visualize geographic objects such as borders of countries or other geographic regions. A natural simplification problem to consider is to simplify borders while having cities being obstacles. The restrictions on simplifying involve maintaining a certain degree of similarity between the original borders and the simplified borders, that is, the simplification is within a certain distance in a given distance metric, as well as simplifying in a way where cities can not end up on the wrong side of borders. Other geometric objects which are of interest to visualize and simplify are contour lines (or simply contours). Visualization of contours provide an easy way to understand the topography of a terrain. While we consider contours in this example as indicating heights, contours can generally indicate any type of data where it is of interest to describe where the data has certain values. We say that two contours are adjacent if it is possible to construct a path which intersects the two contours but no other contour. If two contours are adjacent and one of them is fully enclosed in the region bounded by the other, then we
Chapter 1. Line Simplification Problems in GIS and VLSI

Figure 1.1: Contour lines for a part of Funen. The contours are constructed for every 50cm and from points collected using LIDAR in a 2m by 2m resolution.

call the enclosed contour the child and the other being its parent. The parent/child relationship between contours is called the topology of the contours. It is important that the topology does not change when simplifying contours, as it ensures the property that the maximal difference in heights indicated by any pair of adjacent contours doesn’t change. The problem of simplifying contours is more complicated than simplifying a single border among (static points indicating) cities because the other contours, and thus the obstacles, are simplified as well. See Figure 1.2 for an example of contours being simplified and where self-intersections, intersections among adjacent contours and changing topology are shown. Contours have historically been drawn manually, resulting in nicely flowing curves. Recent advances in mapping technologies, such as laser based LIDAR technology have lead to the advent of extremely high resolution contours. COWI A/S has, as an example, constructed a dataset of Denmark with approximately 12,4 billion \((x,y,z)\) points. Since tools for processing and visualizing terrain data are often not designed to handle data that is larger than main memory (RAM), using these tools for handling large data results in extremely long running times because data is being moved between main memory and disk. Another problem is that contours generated from high-resolution LIDAR data are very detailed, resulting in a large amount of excessively jagged and spurious contours—see Figure 1.1. This in turn hinders their primary applications, since it becomes difficult to interpret the maps and gain understanding of the topography of the terrain.

These are the main reasons why it is of interest to simplify contours. Simpli-
1.2. Line Simplification Problems in VLSI

Figure 1.2: (a) The polygons $P_1$, $P_2$ and $P_3$ to be simplified. (b) A simplification $Q_1$, $Q_2$ and $Q_3$ of the three polygons. The topology between $P_1$ and $P_2$ is not maintained because $P_2$ is not inside $P_1$, but $Q_2$ is inside $Q_1$. The simplification $Q_3$ contains a self-intersection and it does also intersect $Q_1$.

Simplifying boundaries leads to similar benefits in the rendering speed of visualization software. It is furthermore of interest to simplify at several levels of details in order to support fast renderings at various zoom levels [80].

One oft-recurring problem when simplifying contours is that the simplifications should not have self-intersections or intersections with other contours—see Figure 1.2. One can circumvent this problem by simplifying the terrain instead and generate the contours on the simplified terrain. These contours are simplified compared to contours created on the unsimplified terrain. Simplifying the terrain also brings the advantage of spacial awareness when simplifying: A typical (planar) simplification algorithm that only works in the plane does not take the heights of the underlying terrain into account when simplifying.

1.2 Line Simplification Problems in VLSI

In the area of VLSI, where the paths of wires on integrated circuits are computed, the objective function of interest is typically the Euclidean length, link distance (also known as the number of bends in the literature), or some combination thereof [32,85]. Simplification problems arise when efficient paths of wires have to be computed. The challenges in this area include avoiding obstacles and finding efficient routing for several wires at the same time. The simplification problems are typically partitioned into two: Placement and routing. Placement is the problem of computing the homotopy of the wires with respect to a set of obstacles. Routing is the homotopic line simplification problem of simplifying the wires with the given homotopy.

Generally, only rectilinear routings are considered, that is, every part of a wire is axis parallel. Sometimes the vertices of wires are even considered to be on a grid. Considering only rectilinear paths makes the routing problem easier and the Euclidean length is then equal to the $L_1$ distance\footnote{The $L_p$ distance between two points $(x_1, y_1)$ and $(x_2, y_2)$ is $\left( |x_1 - x_2|^p + |y_1 - y_2|^p \right)^{1/p}$ with the special $L_\infty$ distance being $\max(|x_1 - x_2|, |y_1 - y_2|)$.}—refer to Figure 1.3 for some examples of shortest paths in these metrics. Similarly, the obstacles are also often considered to be rectilinear, and some times even rectangles [32,49,61–
Additional restrictions for problems in this area include routing chunks of wires or require certain distances between them. This second restriction can be indirectly obtained by considering thick paths [74] which are polygonal paths with a given non-zero width. By simplifying thick paths so they do not intersect, they correspond to thin paths with a certain minimal distance between them.

Figure 1.3: (a) A rectilinear wire/path and a single rectilinear obstacle (filled). (b) A homotopic path minimizing the $L_1$ metric is shown. (c) A rectilinear homotopic minimum link path (solid) and the homotopic Euclidean shortest path (dashed). Notice that preservation of homotopy forces all paths to go around the obstacles to its right side.
Chapter 2

Homotopic Line Simplification Problems

In this chapter we discuss homotopic line simplification problems in greater detail. These are the types of simplification problems which are considered in the papers of this thesis. After presenting precise definitions of paths, polygons, line simplification problems and homotopic equivalence, we survey our own results by dedicating a section to each of the three types of homotopic line simplification problems which are of interest for them: In Section 2.1 we present unrestricted homotopic line simplification problems while focusing on algorithms for computing Euclidean shortest paths and minimum link paths. The problem of homotopic shortest paths is also discussed as this is unrestricted line simplification problem for which we improve the best known algorithm. In Section 2.2 we present restricted homotopic line simplification problems and focus on minimum link path problems where various error metrics are used to define restrictions on the simplifications. Results in this area often consider Imai and Iri’s framework for restricted line simplification [59] in order to implement these restrictions. Our own results, where we compute homotopic line segments, do also fit into this framework, allowing minimum link paths to be computed with the restriction of every segment in the simplification is homotopic. Finally, in Section 2.3 we consider heuristic algorithms and focus on Douglas Peucker’s line simplification algorithm. There are various results which improve this heuristic algorithm in different ways, and our results use one of these to prevent intersections.

Paths and polygons. Let \( p_1, \ldots, p_n \), be a sequence of \( n > 1 \) points in \( \mathbb{R}^2 \). The path \( P \) defined by these points is the set of line segments defined by pairs of consecutive points in the sequence. The points \( p_1, \ldots, p_n \) are called the vertices of \( P \). A simple path is a path where only consecutive segments intersect, and only at the endpoints. Given integers \( 1 \leq i < j \leq n \), the sub-path \( P_{ij} \) is the path defined by the vertices \( p_i, p_{i+1}, \ldots, p_j \). We abuse notation slightly by using \( P \) to denote both the sequence of vertices, and the path itself. We define the size of \( P \) as its number of segments, i.e. \( |P| = n - 1 \). A polygon (simple polygon) is a path (simple path) \( P \) where \( p_1 = p_n \). A simple polygon \( P \) partitions \( \mathbb{R}^2 \setminus P \)
into two open sets — a bounded one called inside of $P$, and an unbounded one called outside of $P$.

**Line simplification problem.** Let $\mathcal{P} = \{P_1, \ldots, P_n\}$ be a set of $n$ paths in the plane, $S \in \mathbb{R}^2$ be a set of geometric objects (such as simple polygons or points), and $f : \mathbb{R}^2 \to \mathbb{R}$ be a function. A line simplification problem is a problem that given the paths to simplify $\mathcal{P}$ and the set of obstacles $S$: Compute, for each path $P$, a simplification $Q$ which is a path minimizing the objective function $f(Q)$ while satisfying a set of restrictions which are typically based on $P$, $Q$ and $S$.

Recall that a line simplification problems is called restricted when it has the restriction that the vertices of a simplification $Q$ of a path $P$ must be a subset of the vertices of $P$, and appear in the same order as in $P$. Unrestricted line simplification problems simply refers to line simplification problems without this restriction.

**Homotopic Equivalence.** Homotopic line simplification problems are line simplification problems which have the restriction that the paths to be simplified must maintain homotopy to a set of simple polygons, also known as a polygonal domain.

Intuitively, two paths $P$ and $Q$ are homotopic with regards to a polygonal domain $\mathcal{P}$ if $P$ can be continuously transformed into $Q$ without intersecting any of the polygons of $\mathcal{P}$. We refer to Figure 2.1 for an illustration of this. More precisely, $P$ and $Q$ are homotopic with regards to $\mathcal{P}$ if they have the same endpoints and there exists a continuous mapping $f : [0, 1] \times [0, 1] \to \mathbb{R}^2 \setminus \mathcal{P}$ morphing $P$ to $Q$. In other words if $P(\cdot)$ and $Q(\cdot)$ represents the continuous curves (parametrized in $[0, 1]$) defined by the paths $P$ and $Q$ then the following should hold for $f$:

\[
\begin{align*}
    f(0, t) &= P(t) \quad \text{and} \quad f(1, t) = Q(t), \quad \text{for } t \in [0, 1], \\
    f(s, 0) &= P(0) = Q(0) \quad \text{and} \quad f(s, 1) = P(1) = Q(1), \quad \text{for } s \in [0, 1].
\end{align*}
\]
While our definition of homotopy suffices for our studies, there are other definitions which offer an even finer control on the paths. Cabello et al. [22, 41] study the endpoints of the paths more closely and group homotopies based on three variants of restrictions on how the paths are allowed to go around the endpoints. Their variants are called tacks, pins and pushpins respectively. Our definition of homotopy corresponds to their tacks variant.

2.1 Unrestricted Line Simplification Problems

In this section we present some results in the area of unrestricted homotopic line simplification problems for the Euclidean and minimum link metrics.

**Euclidean shortest path.** Homotopic path simplification problems are very closely related to shortest path problems: Given a polygonal domain $P$ and two points $s$ and $t$, the shortest path problem is given an objective function $f$ to compute a path with endpoints $s$ and $t$ minimizing $f$ which does not intersect the area inside of any simple polygon of $P$. In its simplest form, the polygonal domain consists of a single simple polygon and the shortest path is then considered the area inside of $P$.

The earliest results for homotopic line simplification for the Euclidean metric are based on algorithms for computing shortest paths in simple polygons. Back in the early '80s, Chazelle [26] and Lee and Preparata [61] presented algorithms for computing the shortest path in the Euclidean metric (ESP) between two points $s$ and $t$ in a simple polygon $P$ of size $n$ (ESP($s,t$)). The algorithm initially computes a triangulation $D$ of $P$. Let $D$ be the dual graph to $D$. Notice that because $P$ is a simple polygon, $D$ is a tree. Let $D_s$ and $D_t$ be the triangles in $D$ containing $s$ and $t$, respectively. $D_s$ and $D_t$ correspond to nodes in $D$ and it is easy to prove that the triangles on the shortest path between $s$ and $t$ in $D$ correspond to a path $D_{st}$ in $D$ from $D_s$ to $D_t$ where every node on the path is visited at most once. It then follows that one should only consider the triangles corresponding to the nodes on the path $D_{st}$ when constructing the shortest path. The path $D_{st}$ can be found using a simple depth-first search, so the problem is reduced to a simple funnel-traversal [26, 58] through a list of connected triangles, which is solved by the algorithm by Lee and Preparata [61]. Given $D$, all these steps take linear time, thus resulting in a linear time algorithm to compute ESP($s,t$). With the linear time triangulation algorithm by Chazelle et al. [28] (or the simpler randomized algorithm by Amato et al. [11]), the whole algorithm for computing ESP($s,t$) given only $P$ and the endpoints is computed in linear time as well.

Hershberger and Snoeyink [55] show how to extend this algorithm to compute homotopic line simplification in a polygonal domain. The main problem of extending the algorithm for computing ESP($s,t$) in a simple polygon to a polygonal domain is that the dual of the triangulation of the polygonal domain isn’t a tree; it contains cycles. They get around this by formally “lifting” the path $P$ into the universal cover of the polygonal domain. See Erickson’s lecture notes [40] for a short and precise description of this algorithm, where also the
intuition behind lifting is presented.

The running time of the algorithm by Hershberger and Snoeyink [55] depends on the time it takes to construct a triangulation $D$ of the polygonal domain and the number of triangles that $P$ intersects. The triangulation of a polygonal domain $P$ of size $n$ can be constructed in time $O(n \log n)$ using practically efficient algorithms such as [60, 78] and [45], but every segment of $P$ might intersect every triangle, so if $P$ has size $m$, the running time of the algorithm becomes $O(nm + n \log n)$. This bound can be reduced by using the special triangulation by Chazelle et al. [29], which is computed in time $O(n\sqrt{h} + h^{3/2} \log h + n \log n)$, where $h$ is the number of simple polygons in the polygonal domain, and has the property that any line segment intersects at most $O(h^{1/2} \log n)$ triangles. The running time of the simplification algorithm is then $O(n\sqrt{h} + h^{3/2} \log h + n \log n + m\sqrt{h} \log n)$. Alternatively, by using the theoretically fast triangulation algorithm by Bar-Yehuda and Chazelle [16], the running time of the simplification algorithm becomes $O(h \log^{1+\epsilon} h + mn)$, where $\epsilon > 0$.

Other algorithms for computing Euclidean shortest paths in a polygonal domain typically compute the visibility graph [50, 72, 73, 76] and search it using the continuous Dijkstra algorithm [57, 68, 83].

**Minimum link path.** The minimum link path MLP($s, t$) is closely related to the shortest Euclidean path ESP($s, t$). Unlike ESP($s, t$), MLP($s, t$) is not unique. Suri [79] was the first to present an algorithm to compute MLP($s, t$) in a simple polygon, and Hershberger and Snoeyink [55] presented an algorithm based on the algorithm for computing ESP($s, t$), which, again, can be extended into the problem of computing homotopic paths in polygonal domains.

The running time of the minimum link path algorithm by Hershberger and Snoeyink [55] is similar to the algorithm for Euclidean shortest path, but depends on both the time it takes to construct a triangulation $D$ of the polygonal domain, the number of triangles that $P$ intersects, and the number of triangles that the simplification $Q$ intersects. Observe that even if the path $P$ only intersects a small amount of triangles, the simplification $Q$ might consist of up to $|P|$ segments that all intersect all triangles of $D$. Another problem with the algorithm to compute a minimum link path is that, unlike the algorithm for computing the Euclidean shortest path, the simplification might contain self-intersection even when the input path is simple.

**Euclidean shortest path for multiple paths.** That the running times of the ESP algorithms depend on some internal details being the type of triangulation used in the algorithms is an undesirable property. More recent results have focused on the closely related problem of homotopic shortest paths which is the homotopic line simplification problem of given $n$ paths $P = \{P_1, P_2, \ldots, P_n\}$ with combined size $m$, to simplify them while maintaining homotopy in the polygonal domain consisting of the up to $2n$ point obstacles being the endpoints of the paths. The objective function is the Euclidean length of the paths.

Efrat et al. [39] and Bespamyatnikh [20] both present algorithms to solve
the problem of computing homotopic shortest paths. Their algorithms have two phases. In the first phase they solve the problem of homotopic $x$-shortest paths, which is the problem of computing homotopic shortest paths, but where the objective function is the length of the projections of the paths on the $x$-axis rather than the Euclidean length of the paths—see Figure 2.2. Both algorithms utilize a decomposition algorithm by Bar-Yehuda and Chazelle [16] to triangulate the polygonal domain and the idea of segment dragging by Chazelle [27].

![Figure 2.2](image)

(a) The paths of $P_1$, $P_2$, and $P_3$. (b) An example of homotopic $x$-shortest paths $Q_1$, $Q_2$, and $Q_3$. (c) Another version of homotopic $x$-shortest paths $Q_1'$, $Q_2'$, and $Q_3'$. (d) The unique homotopic shortest paths $Q_1''$, $Q_2''$, and $Q_3''$.

The second phases of their algorithms compute homotopic shortest paths from the homotopic $x$-shortest paths. Efrat et al. [39] present a simple randomized algorithm which uses $O(n \log n)$ time to handle the second phase with a total running time of the algorithm of $O(n \epsilon^{-1} \log n + m \log n)$, where $\epsilon > 0$. Bespamyatnikh [20] presents a more complicated divide-and-conquer algorithm which is deterministic and uses specialized data structures to obtain the same running time.

Our results in the paper of Chapter 5 improve the algorithms for computing homotopic $x$-shortest paths from running in time $O(n \log^{\epsilon+1} n + m \log n)$ to $O(n \log^{\epsilon+1} n + m)$ by avoiding having to use segment dragging. Instead, we explore some properties of the decomposition in order to build a pointer-based structure which keeps track on the position of obstacles in relation to the paths. This structure is updated as the paths are simplified.

The running time of our algorithm is close to the algebraic decision tree model lower bound of the problem, which we prove to be $\Omega(n \log n + m)$. By using our algorithm for the first phase of the algorithm by Bespamyatnikh [20] for computing homotopic shortest paths, the running times improves similarly from $O(n \log^{\epsilon+1} n + m \log n)$ to $O(n \log^{\epsilon+1} n + m)$. Similarly, our algorithm can be used for the first phase of the randomized homotopic shortest path algorithm by Efrat et al. [39] in order to achieve the same improvement in the expected
Undesirable properties of unrestricted simplification problems. The use of shortest path algorithms, such as algorithms for Euclidean shortest paths and minimum link paths \([26, 55, 61]\), to solve homotopic line simplification problems has the unfortunate side effects of undesired visual properties, or artifacts (see Figure 2.3): Consider the problems of computing homotopic shortest paths in a polygonal domain where shortest refers to Euclidean length and link length, respectively. For a given path \(P\), the Euclidean shortest path intersects vertices of the polygonal domain \([61]\) in all vertices but the two endpoints. A minimum link path is not unique, so we do not claim that every possible minimum link path has undesirable artifacts, but specific algorithms such as the one by Hershberger and Snoeyink \([55]\) compute a minimum link path by making the line segments reach as far as possible within the polygonal domain, which again includes intersecting the vertices of it. Having the simplifications intersect a large amount of vertices from the polygonal domain is considered (aesthetic) artifacts and are undesirable for visualization of data such as contour lines.

![Figure 2.3](a) A simple polygon (fat) to be simplified and two other simple polygons representing a polygonal domain. (b) The polygon is simplified using a homotopic minimum link path algorithm. (c) The polygon is simplified using a homotopic Euclidean shortest path algorithm.

### 2.2 Restricted Line Simplification Problems

In this section we present restricted line simplification problems and focus on problems that minimize the link length of paths. By considering restricted line simplification problems, the artifacts that we have mentioned are easily avoided: If the vertices of the paths to be simplified are disjoint from those of the polygonal domain, then the vertices of the simplifications have the same property.

Historically, rather than considering homotopic equivalence, results for restricted line simplification problems often consider an error criteria consisting of a distance function \(f\) and have the restriction that the distance between an input path and its simplification must be less than some given error \(\epsilon\) under \(f\). This type of problems is historically called the \(\text{min-k}\) problem and is considered together with the closely related problem of given a maximal number of vertices \(k\), to compute a simplification minimizing the error under the distance...
function. This related problem is known as the min-$\epsilon$ version. The two versions of problems are more precisely:

(i) min-$k$ version: given a path $P$ and a maximum error $\epsilon$ of a distance function $f(\cdot, \cdot)$, compute a simplification $Q$ of path $P$ with the minimum number of vertices such that $f(P, Q) \leq \epsilon$.

(ii) min-$\epsilon$ version: given a path $P$ and a maximum number of vertices $k$, compute a simplification $Q$ of $P$ with the smallest possible error that uses at most $k$ vertices.

Given an algorithm to solve a min-$k$ problem, the corresponding min-$\epsilon$ problem can be solved by performing a binary search on the error-parameter and apply the min-$k$ algorithm at each step of the binary search [59]. In the rest of this section we focus on min-$k$ problems. Restricted min-$k$ simplification is implied whenever a simplification problem is considered.

Imai and Iri’s framework for restricted line simplification. There are many results for restricted line simplification problems under different distance functions. Let $f(\cdot, \cdot)$ be the distance function between a point and a path measuring the shortest Euclidean distance between the point and any point on the path. The Hausdorff distance between two paths $P$ and $Q$ is the maximal value of $f(p, Q)$, where $p \in P$. ¹

Suri [79] solves both versions of the minimum link problems under the Hausdorff distance by modeling them as shortest-path problems on directed acyclic graphs. They use an approach known as Imai and Iri’s framework for restricted line simplification [59]. This framework is often used when designing restricted simplification algorithms, and works as follows: Let $P = p_1, \ldots, p_n$ be a (polygonal) path in the plane, $f$ a distance function and $\epsilon$ be a given error. An unweighted directed graph $G_\epsilon(P)$ (or simply $G_\epsilon$) is defined as follows: $G_\epsilon = (V, E_\epsilon)$ where $V = \{p_1, \ldots, p_n\}$ and

$$E_\epsilon = \{(p_i, p_j) | d_f(p_ip_j, P_{ij}) \leq \epsilon\}$$

where $d_f(p_ip_j, P_{ij})$ is the error between the segment $p_ip_j$ and sub-path $P_{ij}$ under the distance function $f$. A simplification $Q$ with error at most $\epsilon$ corresponds to a path in $G_\epsilon$ from $p_1$ to $p_n$ which can be computed by a breadth-first search (BFS) in time $O(|G_\epsilon|)$.

The running times of the algorithms by Suri [79] are proven to be quadratic and near quadratic by Chin and Chan [25], and Melkman and O’Rourke [67]. Agarwal and Varadarajan [8] improve the running time to $O(n^{4/3+\delta})$, for any fixed $\delta > 0$, for the $L_1$-metric and the uniform metric.²

The Fréchet distance of two paths is intuitively the minimum length of the leash you need when walking on one path while your dog walks on the other and both you and the dog never walk backward on the paths. Simplification

¹This is technically the Hausdorff distance under the Euclidean metric.
²In the uniform metric the distance between the two points is $|y_2 - y_1|$ if $x_1 = x_2$ and $\infty$ otherwise.
problems under the Fréchet distance were first studied by Godau [47]. Later, Alt and Godau [10] proposed an algorithm to compute the Fréchet distance between two polygonal paths in quadratic time; combined with the framework of Imai and Iri this can be used to compute an optimal solution to the min-ε or the min-k problem for the Fréchet distance in $O(n^3)$ time.

**Results that do not use Imai and Iri’s framework.** de Berg et al. [33,34] were the first to study restricted homotopic line simplification for the Hausdorff distance. Their $O(n(n + m) \log n)$ algorithm finds the optimal simplification provided that $P$ is monotone. The simplification is not guaranteed to be optimal if $P$ is not monotone. Guibas et al. [51] show that the optimal homotopic line simplification problem in the unrestricted version is NP-hard when the simplification is forced to be non-self-crossing like the original path.

Chambers et al. [24] study the restricted homotopic Fréchet distance, where the “leash” is not allowed to cross obstacles (points and polygons of a polygonal domain). They present a polynomial time algorithm.

**Strongly homotopic problems.** So far we have discussed restrictions defined using a distance function (and allowed error under this function). The framework by Imai and Iri only allows certain kinds of restrictions to be added. These restrictions have the property that given a $P$ and a simplification $Q$ with the given restriction, then the restriction holds for every segment $p_ip_j \in Q$ and sub-path $P_{ij}$. The restriction of homotopy does not have this property: For a path $P$ it is possible to construct a restricted homotopic simplification $Q$ where there are segments of $Q$ which are not homotopic to the corresponding sub-paths in $P$—see Figure 2.4.

![Figure 2.4](image)

Figure 2.4: While the simplified path $Q$ and original path $P$ are homotopic, some segments like $p_1p_i$ and $p_jp_k$ are not homotopic to $P_{1i}$ and $P_{jk}$ respectively. Therefore, $P$ and $Q$ are not strongly homotopic.

A path $P$ is strongly homotopic [30] to a simplification $Q$ if every segment $p_ip_j$ in $Q$ is homotopic to the corresponding sub-path $P_{ij}$. It follows that $P$ and $Q$ are also homotopic.

In Chapter 6 we present algorithms that given a path of size $n$ and a set of $m$ obstacle points, compute all strongly homotopic line segments of the path. Our algorithms are used in Ima and Iri’s framework in order to add the restriction
of strong homotopy, and can thus be used in combination with the restrictions from the previous results: Given a distance function and maximum distance $\varepsilon$, an algorithm from the previous results is used to compute the graph $G_\varepsilon$. Using our algorithms, all edges which are not strongly homotopic can further be removed from $G_\varepsilon$ before the minimum link path with both restrictions is computed.

First we study the case where the path is $x$-monotone. Notice that any restricted homotopic simplification of an $x$-monotone paths is also strongly homotopic. We present an $O(m \log nm + n \log n \log(nm) + k)$ time algorithm for computing all $k$ strongly homotopic segments.

Our main result is an algorithm for general paths with a running time of $O(m(m + n) \log nm)$. This result uses a modified version of the algorithm by Cabello et al. [22] for comparing homotopies of paths.

Finally, we present a polynomial time minimum link path algorithm for the (normal) homotopic restriction in order to show that this problem is not NP-hard.

### 2.3 Heuristics

Line simplification problems seek paths that are optimal in some way, that is, paths that minimize the given objective function. Sometimes it can be very difficult to construct simplification algorithms that compute optimal simplifications. In many cases shortest path problems are even NP-complete. We refer the reader to the survey by Bern and Eppstein [18] and results by Guibas et al. [51] for examples and proofs of NP-completeness of simplification problems. While NP-complete problems are discouraging for the construction of algorithms, algorithms that construct simplifications that are within a given factor of optimal are still of interest. This kind of algorithms are called approximation algorithms. See [18, 51, 69, 70] for a wide variety of approximation algorithms for shortest path and simplification problems. We distinguish approximation algorithms from heuristic algorithms, or simply heuristics, which are algorithms that do not even promise the simplification being within a given factor of optimal. Heuristics are popular for practical purposes. Algorithms that compute simplifications in optimal time often hide large constants in their running times. The worst case running time of heuristics is not interesting. Rather, the expected running time and subjective quality of simplifications are considered as important features. They do, however, still adhere strictly to any given restrictions of normal simplification algorithms, such as maintaining homotopy or output simplifications that remain within a given error of the input paths.

A heuristic which is often favored for its high subjective and objective quality on real life data [64, 82] is Douglas Peucker’s line simplification algorithm [35], which computes a restricted minimum link path simplification with the (additional) restriction that given an input path $P$ and an error $\varepsilon$, then any segment $p_ip_j$ of the simplification $Q$ has at most Euclidean distance $\varepsilon$ from the vertices of the sub-path $P_{ij}$. We study this heuristic and various improvements
Chapter 2. Homotopic Line Simplification Problems

Douglas Peucker’s line simplification algorithm. The heuristic of Douglas Peucker’s line simplification algorithm is simple: Given a path $P = p_1, p_2, \ldots, p_n$ and an error parameter $\varepsilon$ as input, it computes a restricted simplification $Q = q_1, q_2, \ldots, q_m, q_1 = p_1, q_m = p_n$ by adding vertices from $P$ to $Q$ as follows (see Figure 2.5): Initially $p_1$ and $p_n$ are added. The pivot vertex $p_v, 1 \leq v < n$, which is the vertex of $P$ with greatest Euclidean distance to the line $p_1p_n$ is computed. If this distance is greater than $\varepsilon$, then $p_v$ is added to the output and the algorithm is called recursively on the sub-paths $P_{1,v}$ and $P_{v,n}$. If, on the other hand, the distance is less than $\varepsilon$, then no vertex is added to the output and the algorithm terminates.

![Figure 2.5](image)

Figure 2.5: (a) The path $p_1, \ldots, p_n$ is to be simplified. The pivot vertex $p_i$ is farthest from the line $p_1p_n$ (dotted). (b) The recursion continues on the sub-path $p_1, \ldots, p_i$. Assume the distance between the pivot vertex $p_j$ and the line through $p_1p_i$ is less than the allowed error, then $p_j$ is not added to the output path. (c) For the sub-path $p_i, \ldots, p_n$ we assume the distance between the pivot vertex $p_k$ and the line through $p_ip_n$ is greater than the allowed error, so $p_k$ is added to the output path.

While this heuristic is easy to describe and performs well in practice where its running time is expected to be $O(n \log n)$, it does have some shortcomings: The worst case running time is $O(n^2)$ and the simplification might contain self-intersections. It gets even worse when you use the algorithm to simplify several paths as their simplifications might intersect each other.

Improving Douglas Peucker’s line simplification algorithm. Hershberger and Snoeyink [54] use convex hulls to speed up the worst case running time of Douglas Peucker’s line simplification algorithm to $O(n \log n)$. This improvement comes at the cost of increasing the complexity of the algorithm and changing the best case running time from linear to $O(n \log n)$.

The problem of self-intersections among the simplified segments is studied by Saalfeld [77]. They provide an in-depth analysis of the properties of Douglas Peucker’s line simplification algorithm and give suggestions of what to consider when attempting to avoid self-intersections in the simplification being constructed during the algorithm, and alternatively how intersections can be removed from the output of the basic algorithm after it has been computed.

Wu et al. [84] present an algorithm which is a modification to Douglas Peucker’s line simplification algorithm where there are no intersections in the
2.3. Heuristics

output. Their algorithm is practically efficient, but produces a lot of extra vertices in the output compared to the basic heuristic.

de Berg and Kreveld [33] present an algorithm to maintain homotopy when simplifying an $x$-monotone path in a polygonal domain where the obstacles are points and argue that their approach to ensure homotopy can be applied to Douglas Peucker’s line simplification algorithm. Their algorithm only works for monotone paths, and they handle non-monotone paths by cutting them into monotone sub-paths. Self-intersections can be prevented by adding the points of the input path to the obstacles. Their algorithm takes $O(n(n + m) \log n)$ time to preprocess an input consisting of a path of size $n$ and $m$ obstacles, but they expect it to be efficient in practice.

In the paper of Chapter 7 we present the first practical algorithm for constructing contours on massive data sets and simplifying them using Douglas Peucker’s line simplification algorithm. Recall that algorithms are designed for the I/O model of computation [9] in order to be able to handle input that is larger than the size of main memory. Here the transfer of data between disk and main memory (I/Os) is often a bottleneck (see e.g. [31]), and the complexity of an algorithm is measured in terms of the total number of I/Os it performs—see Figure 2.6.

![Figure 2.6: The I/O-model of Computation has a main memory of size $M$ and a disk of infinite size. The CPU can only perform computation on data in main memory and data is transferred between main memory and disk in blocks of size $B$.](image)

Recall also, that the topology of contours must be maintained when simplifying, and that there may not be intersections among the segments of the simplified contours. Topology can be maintained by enforcing homotopy between a path and its simplification with the obstacles being all other contours. We add strong homotopy rather than (normal) homotopy as this restriction can easily be added to Douglas Peucker’s line simplification algorithm of simplifying a path $P$: Rather than stopping the recursion when a pivot vertex is within distance $\varepsilon$ of a line $p_ip_j$, we continue until it additionally holds that the line segment $p_ip_j$ is homotopic to the sub-path $P_{ij}$. Enforcing strong homotopy ensures that the line segments of a simplified contour do not intersect segments of any other contour. A simplification might, however, contain self-intersections. One of the ideas by Saalfeld [77] is used to remove self-intersections: After computing a simplification $Q$, we scan $Q$ for self-intersections (using a simple sweep line algorithm). For any couple of segments that intersect we add vertices to $Q$ as if the recursion of Douglas Peucker’s line simplification algorithm
had continued on the intersecting segments. After handling all the intersections that are found during a scan, the algorithm is restarted on \( Q \) as the additional segments might cause new intersections. This procedure is relatively inefficient, but we expect that self-intersections rarely happen in practice. We study this assumption during our practical experiments. For our practical experiments we construct contours for every 50cm across Denmark and we construct additional contours with height difference of 20cm above and below each contour. These additional contours are not simplified, but simply ensure that the simplified contours are restricted to be within the given height difference. The additional contours cause the number and combined size of constructed contours to be tripled, resulting in 4,793,518,863 points on 7,260,043 contours. Our practical results show the practicality of our algorithm both in terms of running time (it takes 49 hours to simplify the contours, which is efficient when considering that it takes 6 hours just to read data of this size from disk) and the size of simplifications (the simplified contours contain only 9,2\% of the vertices of the original contours when simplified with an error parameter of 5m for Douglas Peucker’s line simplification algorithm). It is confirmed that self-intersections rarely happen in practice (only 3,1\% of the simplified contours contain self-intersections), so our choice of an inefficient way of handling self-intersections does not affect the running time significantly.
Chapter 3

Simplification Problems in Various Areas

In the previous chapters we have only discussed a small number of homotopic line simplification problems motivated by problems in the areas of cartographic generalization, GIS and VLSI. Recall that simplification problems are also of interest for the areas of robot motion planning and ATM. In this chapter we present a variety of simplification problems which are motivated by applications in these areas. We consider different variants of either the type of objects to be simplified, the objective function, constraints, or types of obstacles of interest for simplification problems.

Mitchell [69, 70] presents more comprehensive surveys which include the properties that we discuss. His surveys are of (geometric) shortest path problems, but shortest path problems are very closely related to line simplification problems, and the properties of shortest path problems that he discuss apply for simplification problems as well.

3.1 Types of Objects to be Simplified

The objects to simplify in homotopic line simplification problems are simply (polygonal) paths in the plane ($\mathbb{R}^2$). There is a wide variety of other objects to be simplified which are of interest for the areas mentioned. In this section we present a brief overview of simplification problems where the objects are in $\mathbb{R}^3$ as well as different kinds of objects to simplify in $\mathbb{R}^2$.

Objects in $\mathbb{R}^3$. Recall that in GIS it is of interest to simplify a terrain, which is represented by an object in $\mathbb{R}^3$. For our results in Chapter 7 we use the practical and I/O-efficient algorithm by Arge et al. [15] to simplify the terrain before we use it for the construction of contour lines. They simplify the terrain by “filling up” any small depression whose depth is less than a given threshold. By removing such depressions (and small hills similarly), nearly 70% of all contours are removed in the datasets we consider. A lot of clutter is thus removed when visualizing the contours.
For path simplification problems motivated by robot motion planning it is similarly of interest to consider paths in $\mathbb{R}^3$ should the robot move in a non-planar environment. The surveys by Mitchell [69, 70] present results for simplifying in higher dimensional space, and focuses on $\mathbb{R}^3$ as this is the space with most immediate applications and thus the most well studied space for simplification algorithms next to $\mathbb{R}^2$.

**Other types of objects in $\mathbb{R}^2$.** Curved paths are of interest for the areas of ATM and robot motion planning as they better resemble the paths that the physical objects (airplanes and robots) move along. Similarly, in cartography (and GIS), curved paths are preferred for visualization. One way to model curved paths is to use parametrized curves, such as Bezier curves. Non-intersecting fat paths are of interest for both ATM, where flight routes may not overlap; robot motion planning, where a robot has a footprint which isn’t a point; and VLSI, where real wires should not overlap, so computing fat path that do not intersect ensures a minimal distance between the corresponding thin paths [44].

### 3.2 Objective function

Simplification problems have an objective function for the simplified objects to minimize. In the previous chapters we have studied problems where the objective functions are the Euclidean length and link length and briefly mentioned other metrics based on pairs of points, such as the $L_p$ distances and the uniform metric, as well as metrics based on pairs of paths, such as the Hausdorff and Fréchet distances. These are all simple metrics. In the areas of VLSI, robot motion planning and ATM it is also of interest to study composite objective functions: For VLSI it is of interest to minimize both the Euclidean length and the link length of wires simultaneously, so the objective function becomes a linear combination of these two metrics. See de Berg et al. [32] and Yang et al. [85] for algorithms with this kind of combined objective function. In ATM it is of interest to minimize a cost function which includes various parameters, such as fuel cost, salaries and location specific taxation\(^1\), while in robot motion planning it is of similar interest to minimize the time it takes for a robot to reach its destination. The complexity of the time function increases as more real world aspects of robot motion planning are included, such as varying speed and acceleration. The robot might move at different speeds either because of the environment (the robot moves at different speed across different surface types or there might be speed limits in certain regions) or the state of the robot (it moves slower when carrying objects). Modelling acceleration is complicated further when considering the physical properties when turning: Not only does it take time to turn, but the speed of the robot does also affect its ability to turn. Consider for example a car based robot. Such a robot moves most efficiently in gentle curves rather than paths made out of straight line segments.

---

\(^1\)it costs extra to fly through certain flight zones.
3.3 Constraints

In the simplification problems considered in the previous chapters we have considered the restrictions of homotopy, strong homotopy and the name giving restriction of restricted simplification problems. Recall that we only consider the “tacks” variant of the homotopies studied by Cabello et al. [22, 41]. Recall also that restricted simplification problems of the min-k variant have the restriction that a simplification is within a given distance of the input path under some distance function. Just as with objective functions, constraints are added to simplification problems in order to model more aspects of real life problems.

3.4 Obstacles

The obstacles of simplification problems (also known as the input geometry) refers to the type of obstacles to avoid when simplifying. In the previous chapters we have considered obstacles as being polygonal domains, and sometimes with the restriction that every obstacles is a single point (rather than a simple polygon). In the areas of GIS and robot motion planning it is also of interest to study curved obstacles such as splinegons\(^2\) [65, 66] and parametrized curves such as non-uniform rational B-splines or Bezier curves [75].

Curved obstacles are of interest in robot motion planning when robots, whose footprints are more than single points, are considered. The simplest kind of robot to consider is one whose footprint consists of a point because its movement can be modelled by a polygonal path. While this isn’t a realistic model for a physical robot, it can still be useful: Moving a robot with a circular footprint among polygonal obstacles corresponds to moving a point robot among obstacles whose boundaries are the Minkowski sum of the original obstacles and the circle of the robot. These boundaries are splinegons. See the work by Melissaratos et al. [65, 66] for shortest path results with this kind of boundaries. Similarly, if a convex robot is not allowed to turn (it can still move freely using multi-directional wheels or similar), then it corresponds to moving a point robot among obstacles whose boundaries again have been augmented to the Minkowski sum of the original boundaries and the robot. Finally, by only allowing a constant set of orientation for a convex robot, its movement can be modelled using a constant set of augmented boundaries.

Problems with rectilinear obstacles (and paths) are of significant interest in VLSI, where rectilinear wires are routed among obstacles [32,63,85]. Leiserson et al. [62,63] also consider the even more restricted problem where the vertices of paths are on a grid. Problems often benefit from paths and obstacles being rectilinear, as it reduces the complexity of the problems and thus the running time of algorithms (Refer to [49] and [61] for some examples).

All the obstacles we have mentioned are static (non-moving). Non-static obstacles are of interest for simplification problems in the area of robot motion planning, where obstacles such as heavy machinery or sentries must be avoided.

\(^2\)A splinegon is a geometric object whose boundary consists of line segments and parts of circle boundaries.
Chapter 4

Conclusion

In this thesis we present new results for homotopic line simplification problems in the three areas of unrestricted homotopic line simplification, restricted homotopic line simplification and heuristic algorithms.

Recall that the problem of computing homotopic shortest paths of given $n$ paths $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ with combined size $m$, is to compute their homotopic Euclidean shortest paths in the polygonal domain consisting of the up to $2n$ point obstacles being the endpoints of the paths. The problem of homotopic $x$-shortest paths is the same problem, but where the objective function is the length of the projection of the paths on the $x$-axis (rather than the Euclidean length). For unrestricted homotopic line simplification problems our contribution in Chapter 5 is an algorithm for computing homotopic $x$-shortest paths which improves the running time of the best known algorithms by Efrat et al. [39] and Bespamyatnikh [20] from $O(n^{\epsilon+1} \log n + m \log n)$, where $\epsilon > 0$, to $O(n \log^{\epsilon+1} n + m)$. This is close to the algebraic decision tree model lower bound of the problem, which we prove to be $\Omega(n \log n + m)$. The $\epsilon$ factor in the running time comes from the construction of a decomposition of the polygonal domain consisting of the input paths. If it is possible to construct a decomposition in optimal time $O(n \log n + m)$, then the running time of our algorithm would improve to $O(n \log n + m)$ as well. By using our algorithm as a component in the algorithms by Bespamyatnikh [20] and Efrat et al. [39] for computing homotopic shortest paths, the running times improves similarly from $O(n \log^{\epsilon+1} n + m \log n)$ to $O(n \log^{\epsilon+1} n + m)$, where the algorithm by Efrat et al. [39] is randomized, so this is its expected running time, as opposed to the worst case complexity of the more complicated algorithm by Bespamyatnikh [20].

For restricted homotopic line simplification problems our contributions in Chapter 6 are two algorithms for computing homotopic line segments, which we recall is the problem of given a path $P = p_1, \ldots, p_n$ and $m$ point obstacles $S$, compute all segments $p_ip_j$ which are homotopic to $P_{ij}$ in the presence of $S$. We present an $O(m \log nm + n \log n \log(nm) + k)$ time algorithm, where $k$ is the number of homotopic segments, for the case where $P$ is $x$-monotone and an $O(m(m + n) \log nm)$ time algorithm for general paths. Our algorithms are used in Imai and Iri’s framework in order to add the restriction of strong
homotopy: By constructing the graph $G = (V, E)$, where $V = \{p_1, \ldots, p_n\}$ and $p_ip_j \in E$ for any strongly homotopic line segment $p_ip_j$, the line segments of a restricted strongly homotopic minimum link path can be computed simply by a BFS search from $p_1$ to $p_n$ in $G$. Similarly, given a distance function (such as the Hausdorff or Fréchet distances) and maximum distance $\varepsilon$, an algorithm from the previous results can be used to compute the graph $G_\varepsilon$ which contains an edge for every line segment which is within the allowed distance under the distance function. Using our algorithms, all edges which are not strongly homotopic can further be removed from $G_\varepsilon$. A minimum link path which is both within the allowed error under the distance function and strongly homotopic can now be computed by a BFS search through $G_\varepsilon$. We also present a polynomial time restricted homotopic minimum link path algorithm, showing that this problem is not NP-complete.

For heuristic algorithms our contribution in Chapter 7 is the first I/O-efficient algorithm for constructing contour lines of massive datasets and simplify them using a modified version of the heuristic of Douglas Peucker’s line simplification algorithm. Our modifications to the heuristic include maintaining strong homotopy (in order to preserve the topology of the contour lines and prevent intersections between adjacent contours) and an additional algorithm to remove self-intersections in simplified contours. Both of these modifications are implemented using an idea recommended by Saalfeld [77]. Our algorithm makes two practically realistic assumptions about the contours, which allow practical implementations: The first assumption is that the contour segments intersected by any horizontal line, fit in main memory. The second assumption is that the combined number of vertices of any contour, as well as all contours adjacent to it, fit in main memory. This second assumption allows us to use an internal memory (RAM model) algorithm to simplify the contours, so the practically I/O-efficient parts of our main algorithm only have to be designed for constructing contours, constructing their topology and laying out the contours in a way that allows each contour to be accessed efficiently together with the contours that are adjacent to it. We present practical results which confirm the practicality of the algorithm by simplifying contour lines for a very large dataset using a couple of days of computation time. The experiments also confirm our assumptions that the restrictions we impose on simplifications, and our algorithms to enforce these restrictions, do not significantly impact the running time, nor do they result in an unreasonable number of additional vertices in the simplifications.
Part II

Papers
Chapter 5

Homotopic X-Shortest Paths

The paper *Improving Homotopic Shortest Paths Using Homotopic X-Shortest Paths* presented in this chapter has yet to be submitted.

Improving Homotopic Shortest Paths Using Homotopic $X$-Shortest Paths

Mohammad Ali Abam\textsuperscript{*} \hspace{0.5cm} Lasse Deleuran\textsuperscript{†}

Abstract

We present an algorithm for the problem of computing homotopic $x$-shortest paths of $n$ planar simple paths with combined size $m$, which is the problem of minimizing the length of the projection of the paths on the $x$-axis while maintaining homotopy to the set of endpoints of the paths. Our algorithm runs in time $O(n \log^{1+\epsilon} n + m)$, where $\epsilon > 0$, while using $O(n + m)$ space. This reduces the best known running time from $O(n \log^{1+\epsilon} n + m \log n)$ and is close to the algebraic decision tree model lower bound of the problem, which we prove to be $\Omega(n \log n + m)$. The algorithm can be used to improve the running time of the best known algorithm for computing homotopic shortest paths from $O(n \log^{1+\epsilon} n + m \log n)$ to $O(n \log^{1+\epsilon} n + m)$.

5.1 Introduction

Computing geometric shortest paths is a well-studied topic within computational geometry. See e.g. [69, 70]. In this paper we consider the problem of finding geometric shortest paths of given homotopy types. We describe an improved algorithm for a special case of the problem (computing homotopic $x$-shortest paths) and show how it can be used to improve the best known algorithm for the problem of computing homotopic shortest paths.

Problem definition. Let $p_1, \ldots, p_n$ be a sequence of $n \geq 1$ points in $\mathbb{R}^2$. The polygonal path (or simply path) $\alpha$ defined by these points is the set of line segments defined by pairs of consecutive points in the sequence. The points $p_1$ and $p_n$ are called the endpoints of $\alpha$. A simple polygonal path (simple path) is a path where only consecutive segments intersect, and only at the endpoints. We abuse notation slightly by using $\alpha$ to denote both the sequence of points and the path itself. We define the size of $\alpha$ as its number of segments, i.e. $|\alpha| = n - 1$.

Two paths $\alpha$ and $\beta$ are homotopically equivalent with respect to a set of barrier points $B$ in $\mathbb{R}^2$ iff they have the same endpoints and there exists a

\textsuperscript{*}Computer Engineering Department, Sharif University of Technology, Iran.

\textsuperscript{†}Center for Massive Data Algorithmics at Computer Science Department of Aarhus University, Denmark.
continuous mapping \( f : [0, 1] \times [0, 1] \to \mathbb{R}^2 \setminus B \) morphing \( \alpha \) to \( \beta \), that is, such that

\[
\begin{align*}
f(0, t) &= \alpha(t) & \text{and} & & f(1, t) &= \beta(t), & \text{for } t \in [0, 1] \\
f(\lambda, 0) &= \alpha(0) = \beta(0) & \text{and} & & f(\lambda, 1) &= \alpha(1) = \beta(1), & \text{for } \lambda \in [0, 1]
\end{align*}
\]

where \( \alpha(\cdot) \) and \( \beta(\cdot) \) denote the continuous curves (parametrized in \([0, 1]\)) defined by the paths \( \alpha \) and \( \beta \).

We consider the problem of finding the homotopic shortest paths of a set of simple polygonal paths: Let \( \mathcal{P} = \{\alpha_1, \alpha_2, \ldots, \alpha_n\} \) be a set of \( n \) disjoint simple polygonal paths, and let \( B \) be the set of at most \( 2n \) endpoints of the paths in \( \mathcal{P} \). The problem consists of computing a set of paths \( \mathcal{P}' = \{\alpha'_1, \alpha'_2, \ldots, \alpha'_n\} \) such that every \( \alpha'_i \) is the (Euclidean) shortest path that is homotopically equivalent to \( \alpha_i \) with respect to \( B \). Note that while the input paths are simple the output paths do not need to be. We will assume, without loss of generality, that none of the points defining \( \mathcal{P} \) share the same \( x \)-coordinate.

Homotopic shortest path algorithms typically work by initially computing homotopic \( x \)-shortest paths, which is the same as computing homotopic shortest paths except that the length of the projection of the paths on the \( x \)-axis is minimized rather than the Euclidean length. Note that unlike the homotopic shortest paths, homotopic \( x \)-shortest paths are not unique. Refer to Figure 5.1.

![Figure 5.1](image_url)

Figure 5.1: (i): The paths of \( \mathcal{P} = (\alpha_1, \alpha_2, \alpha_3) \). (ii): An example of homotopic \( x \)-shortest paths of \( \mathcal{P} \). (iii): Another version of homotopic \( x \)-shortest paths of \( \mathcal{P} \). (iv): The unique homotopic shortest paths of \( \mathcal{P} \).

**Previous Results.** Efrat et al. [39] showed how to compute homotopic shortest paths of a set \( \mathcal{P} \) of \( n \) disjoint simple paths of combined size \( m \). They present both an \( \mathcal{O}(n^{3/2} + m \log n) \)-time deterministic algorithm, and a randomized algorithm running in expected time \( \mathcal{O}(n \log^{\epsilon+1} n + m \log n) \) for \( \epsilon > 0 \). Both algorithms consist of two phases, where the first phase computes the homotopic \( x \)-shortest paths of \( \mathcal{P} \), while the second phase computes the homotopic shortest
paths of the \(x\)-shortest paths. The first phase of both algorithms are identical and deterministic. First a trapezoidal decomposition of \(P\) is computed in \(O(n \log^{\epsilon+1} n + m)\) time using an algorithm due to Bar-Yehuda and Chazelle [16]. The homotopic \(x\)-shortest paths are then computed using a modified version of a \textit{path shortcutting} algorithm due to Cabello \textit{et al.} [22]. The shortcutting algorithm uses the idea of \textit{segment dragging} [27] and takes \(O(m \log n)\) time, resulting in a total first phase running time of \(O(n \log^{\epsilon+1} n + m \log n)\). The second phase differs in the two algorithms, with the randomized algorithm using \(O(n \log n + m)\) expected time, and the deterministic algorithm using \(O(n^{3/2} + m)\) time.

Bespamyatnikh [20] gave a deterministic algorithm for computing homotopic shortest paths using \(O(n \log^{\epsilon+1} n + m \log n)\) time, that is, they remove the randomization from the algorithm by Efrat \textit{et al.} [39]. Their algorithm has the same two phases as the algorithms by Efrat \textit{et al.} [39], with the first phase relying on segment dragging using \(O(n \log^{\epsilon+1} n + m \log n)\) time, and the second phase taking only \(O(n \log n + m)\) time.

**Our Results.** In Section 5.2 we present an improved \(O(n \log^{\epsilon+1} n + m)\) time algorithm for computing homotopic \(x\)-shortest paths, and in Section 5.3 we prove an \(\Omega(n \log n + m)\) algebraic decision tree model lower bound for the problem. Our algorithm distinguishes itself from previous algorithms by not relying on segment dragging. Instead, it computes a trapezoidal decomposition of the input paths in \(O(n \log^{\epsilon+1} n + m)\) time and exploits properties of the decomposition in order to compute the homotopic \(x\)-shortest paths in additional \(O(n + m)\) time. By applying our algorithm to the first phase of the algorithm by Bespamyatnikh [20], we improve the running time of their algorithm to \(O(n \log^{\epsilon+1} n + m)\).

## 5.2 Homotopic \(X\)-Shortest Paths Algorithm

Our algorithm for computing homotopic \(x\)-shortest paths is based on sub-paths that we call **limbs**. We define limbs and various relations between limbs in Section 5.2.1, where we also present an algorithm for constructing the limbs and their relations. In Section 5.2.2 we then show how to compute homotopic \(x\)-shortest paths by changing these limbs while maintaining the relations between them.

### 5.2.1 Limbs

A simple polygonal path is called **\(x\)-monotone** iff any vertical line intersects the path in at most one point. It is easy to realize that we obtain \(x\)-monotone paths \(\{\alpha_1^m, \alpha_2^m, \ldots, \alpha_j^m\} \), \(j < n\), if we divide a simple path \(\alpha = p_1, p_2, \ldots, p_n\) at any point \(p_i\) where \(p_{i-1}\) and \(p_{i+1}\) both have either smaller or larger \(x\)-coordinates than \(p_i\); \(p_i\) is called a **locally rightmost** (locally leftmost) point if \(p_{i-1}\) and \(p_{i+1}\) have smaller (larger) \(x\)-coordinates than \(p_i\), and we say that \(\alpha_i^m\) is a **maximally \(x\)-monotone path** of \(\alpha\).
Chapter 5. Homotopic X-Shortest Paths

We define a limb $\Lambda$ to be two maximal $x$-monotone paths $\alpha_{i}^{m}$ and $\alpha_{i+1}^{m}$ sharing a point $p_{i}$; the point $p_{i}$ is called the peak of $\Lambda$ and denoted $\text{Peak}(\Lambda)$, and the position of $\Lambda$ is the position (coordinates) of $\text{Peak}(\Lambda)$. We say that a limb $\Lambda$ is right pointing if $\text{Peak}(\Lambda)$ is a locally rightmost point and left pointing is $\text{Peak}(\Lambda)$ is a locally leftmost point. The intersections between $\Lambda$ and a vertical line with $x$-coordinate slightly smaller than the $x$-coordinate of $\text{Peak}(\Lambda)$ if $\Lambda$ is right pointing, and with $x$-coordinate slightly larger than the $x$-coordinate of $\text{Peak}(\Lambda)$ if $\Lambda$ is left pointing, naturally defines an above-below relation on the paths $\alpha_{i}^{m}$ and $\alpha_{i+1}^{m}$; we denote the top path $U(\Lambda)$, the bottom path $L(\Lambda)$ and the endpoints of the two paths (other than $\text{Peak}(\Lambda)$) $E_{U}(\Lambda)$ and $E_{L}(\Lambda)$, respectively. For convenience we also consider $\alpha_{1}^{m}$ and $\alpha_{n}^{m}$ to be limbs as well with $p_{1}$ and $p_{n}$ being their peaks. Whenever endpoints are mentioned in the rest of this paper, we mean endpoints of the paths $P$ (as opposed to endpoints of limbs).

To compute the homotopic $x$-shortest paths of $P$ efficiently in Section 5.2.2, we will define a few relations between the right pointing limbs. Consider adding a horizontal line below and a horizontal line above all paths in $P$, and for each limb $\Lambda$ connect points $E_{U}(\Lambda)$ and $E_{L}(\Lambda)$ with the first line segment hit by a vertical ray emanating down from the point (the line segment immediately below each point). For a right pointing limb $\Lambda$ a down-right-path is now defined as a path that starts at the line segment immediately below $\text{Peak}(\Lambda)$ and follows line segments and segment-point connections in the positive $x$-direction. Similarly, we define a up-right-path as a path that starts at the line segment immediately above $\text{Peak}(\Lambda)$ and follow line segments and connections between segments and points defined by rays connecting upwards. We say that a right-path reaches any path where it can reach a line segment in the path. We now define the following for each right pointing limb $\Lambda$ (refer to Figure 5.2(1)).

- $B(\Lambda)$: The limb with largest $x$-coordinate peak among all right pointing limbs with down-right-paths reaching $L(\Lambda)$.
- $N_{U}(\Lambda)$: The limb with largest $x$-coordinate peak among all right pointing paths with up-right-paths reaching $U(\Lambda)$. 

Figure 5.2: (i): The right pointing limb $\Lambda$ (highlighted) and its relations. (ii): The trapezoidal decomposition of the paths.
5.2. Homotopic X-Shortest Paths Algorithm

limbs with down-right-paths reaching $U(\Lambda)$.

- $N_L(\Lambda)$: The limb with largest x-coordinate peak among all right pointing limbs with up-right-paths reaching $L(\Lambda)$.

Computing limbs and relations. Given the simple path $\alpha = p_1, p_2, \ldots, p_n$ it is easy to compute all limbs, along with $E_U(\Lambda), E_L(\Lambda), U(\Lambda), L(\Lambda)$ and $\text{Peak}(\Lambda)$ for each limb $\Lambda$, in $O(n)$ time in a few scans of $\alpha$. Thus given $n$ disjoint simple paths $\mathcal{P} = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ of combined size $m$, we can compute all limbs in $O(m)$ time.

To compute $B(\Lambda), N_U(\Lambda)$ and $N_L(\Lambda)$ for each right pointing limb $\Lambda$, we first compute the trapezoidal decomposition [16] of the paths in $\mathcal{P}$, that is, we compute the line segments hit by vertical rays emanating up and down from each of the $m$ points in the simple paths in $\mathcal{P}$. Refer to Figure 5.2(ii).

For each right pointing limb $\Lambda$ we find any limb $\lambda$ for which $\Lambda = B(\Lambda)$ by traversing all down-right-paths from $\Lambda$, branching the traversal whenever there is a choice to follow a segment-point connection up, and stopping a traversal whenever a vertical line, emanating down from the peak of a right pointing limb, is encountered. All $N_U$ relations are computed the same way and for $N_L$ relations we similarly follow all up-right-paths and stop once rays emanating up from peaks of right pointing limbs are encountered.

It is easy to prove that the algorithm correctly computes the $B$ relations for all right pointing limbs since it traverses all down-right-paths. Stopping a branch in the traversals once a peak of right pointing limb is encountered limits the down-right-paths considered. This limit is sound because given a limb $\Lambda$, any down-right-path $\Lambda$ extending past the peak of a right pointing limb $\lambda$ (that is, where a ray emanating vertically down from $\text{Peak}(\lambda)$ intersects the down-right-path before anything else) will never be considered for a relation, as the $x$-coordinate of $\text{Peak}(\lambda)$ is greater than the $x$-coordinate of $\text{Peak}(\Lambda)$. The correctness for computing the $N_U$ and $N_L$ relations follows by symmetry.

Bar-Yehuda and Chazelle [16] showed how a trapezoidal decomposition of $n$ simple paths of combined size $m$ can be computed in $O(n \log^{1+\epsilon} n + m)$ time for any $\epsilon > 0$ using $O(n + m)$ space. After computing the trapezoidal decomposition, our algorithm traverses each line segment of it (where the segment-point connections are among the vertical lines of the decomposition) at most once in order to compute all $B$ relations using $O(m + n)$ time. The $N_U$ and $N_L$ relations are computed similarly. Thus overall we compute all limbs and the relationships between them in $O(n \log^{1+\epsilon} n + m)$ time and $O(n + m)$ space:

**Theorem 5.1.** Computing all limbs and the relationships between the right pointing limbs of $n$ disjoint simple paths with combined size $m$ takes $O(n \log^{1+\epsilon} n + m)$ time for any $\epsilon > 0$, and $O(n + m)$ space.

### 5.2.2 Algorithm

In this section we present our algorithm for changing limbs in order to compute homotopic $x$-shortest paths. Given a limb $\Lambda$ we define $E_R(\Lambda)$ to be the point
Chapter 5. Homotopic X-Shortest Paths

$E_U(\Lambda)$ if its $x$-coordinate is closer to the $x$-coordinate of $\text{Peak}(\Lambda)$ than the $x$-coordinate of $E_L(\Lambda)$, and $E_R(\Lambda) = E_L(\Lambda)$ otherwise. A limb $\Lambda$ is contracted to a point $p = (x, y)$, where $x$ is between the $x$-coordinates of $\text{Peak}(\Lambda)$ and $E_R(\Lambda)$, by replacing all points of the limb with $x$-coordinates between $x$ and the $x$-coordinate of $\text{Peak}(\Lambda)$ (including the peak) with the two points in the intersections between $\Lambda$ and a vertical line at $x$. Let $p$ be the point among all points in the area bounded by $U(\Lambda)$ above, $L(\Lambda)$ below and a vertical line at the $x$-coordinate of $E_R(\Lambda)$ whose $x$-coordinate is the closest to the $x$-coordinate of $\text{Peak}(\Lambda)$. If $p$ exists, then $\Lambda$ can be contracted to $p$ and we call $p$ the blocking endpoint of $\Lambda$. Otherwise, $\Lambda$ can be contracted to $E_R(\Lambda)$. We can now describe a simple algorithm to contract all limbs. The algorithm simply picks any limb $\Lambda$ and contracts it to the blocking endpoint if it exists, else it is contracted to $E_R(\Lambda)$. This repeats until no limb can be contracted. This algorithm computes homotopic $x$-shortest paths:

**Lemma 5.1** (Efrat et al. [39](Lemma 1)). Given simple paths $\mathcal{P}$. If no limb of the paths can be contracted, then the paths are homotopic $x$-shortest.

Unfortunately this algorithm might generate crossing segments among the paths of the output, and it is not efficient since one has to search for the blocking points of the up to $m$ limbs. We present an efficient algorithm for computing homotopic $x$-monotone paths which does not cause crossing among line segments. Our algorithm is divided into three phases: In the first phase we contract all right pointing limbs. In the second phase we contract all left pointing limbs. After contracting all left pointing limbs, there might be degeneracies. In the third phase we handle the degeneracies. All steps consist of contracting limbs, so the correctness of the algorithm follows from not being able to contract any limb after the third phase.

**The first phase: Contracting all right pointing limbs.** Consider the simple algorithm from the previous section. If we only contract right pointing limbs, then there will be no crossings among the line segments [39]. Furthermore no new limbs are created when only contracting right pointing limbs.

In the first phase we completely ignore that there exists left pointing limbs. Any limb mentioned in this phase is understood as being right pointing.

In order to maintain homotopy when contracting, a limb $\Lambda$ can be contracted to $B(\Lambda)$, or $E_R(\Lambda)$ if $B(\Lambda)$ either does not exist or the $x$-coordinate of $B(\Lambda)$ is less than the $x$-coordinate of $E_R(\Lambda)$. To support this kind of contractions, all $B$ relations have to be re-computed before each contraction/iteration of the algorithm. By contracting $\Lambda$ only if the limb $B(\Lambda)$ either has already been contracted, is a limb where the peak is an endpoint, or does not exist, we ensure that any limb is only contracted once. This is a consequence of Theorem 5.2:

**Theorem 5.2.** By only contracting a limb $\Lambda$ once $B(\Lambda)$ either has been contracted, $B(\Lambda)$ is a limb where the peak is an endpoint, or $B(\Lambda)$ does not exist, contracting $\Lambda$ will either result in $\Lambda$ being removed or contracted to an endpoint.

**Proof.** This is easily proved by induction in the $x$-coordinates of the peaks of the limbs sorted by increasing order. The theorem trivially holds for the limb
with the peak whose $x$-coordinate is minimal. Consider a limb $\Lambda$ and assume
that the theorem holds for any limb $\lambda$ where the $x$-coordinate of $\text{Peak}(\lambda)$ is
less than the $x$-coordinate of $\text{Peak}(\Lambda)$. Since the $x$-coordinate of $\text{Peak}(\text{B}(\Lambda))$
is less than the $x$-coordinate of $\text{Peak}(\Lambda)$ by construction, the lemma holds for $\text{Peak}(\Lambda)$, so contracting $\Lambda$ to either $\text{N}_R(\Lambda)$ (which removes $\Lambda$) or $\text{B}(\Lambda)$ makes the lemma hold for $\Lambda$.

It is expensive to re-compute the relations between limbs after each con-
traction. We present a modified algorithm where the relations are updated
efficiently. For this algorithm we make use of the $\text{N}_U$ and $\text{N}_L$ relations and
define a limb $\Lambda$ to be ready once it for all relations $\text{B}(\Lambda)$, $\text{N}_U(\Lambda)$ and $\text{N}_L(\Lambda)$
holds that the relation is either a contracted limb, a limb whose peak is an end-
point or the relation does not exist. The algorithm only contracts limbs that
are ready, with their order being irrelevant. Contracting one limb can cause
other limbs to become ready. A limb is contracted and the relations between
limbs are updated following the description of Algorithm 1.

\begin{algorithm}
\caption{Contract($\Lambda$): Contraction of a right pointing limb $\Lambda$}
\begin{algorithmic}[1]
\item[1:] Let $v$ be the rightmost of $\text{B}(\Lambda)$ and $\text{E}_R(\Lambda)$. $\Lambda$ is contracted to $v$.
\item[2:] The $\text{N}$ and $\text{B}$ relations of other limbs pointing to $\Lambda$ are updated to reflect
the change as follows: Let $u$ be the rightmost of $\text{N}_U(\Lambda)$, $\text{N}_L(\Lambda)$ and $v$. For
any limb $\lambda$ where $\Lambda = \text{N}_U(\lambda)$, $\text{N}_U(\lambda)$ becomes $u$. For any limb $\lambda$ where $\Lambda = \text{N}_L(\lambda)$, $\text{N}_L(\lambda)$ becomes $u$. For any limb $\lambda$ where $\Lambda = \text{B}(\lambda)$, $\text{B}(\lambda)$ becomes $u$.
\item[3:] If $v$ is $\text{E}_U(\Lambda)$, then $\Lambda$ is removed. Let $u$ be the rightmost of $\text{N}_U(\Lambda)$ and
$\text{N}_L(\Lambda)$. For any limb $\lambda$ where $\Lambda = \text{N}_U(\lambda)$, $\lambda$ becomes $u$. For any limb $\lambda$ where $\Lambda = \text{B}(\lambda)$, $\lambda$ becomes $u$.
\item[4:] If the two merging limbs have an upper $x$-monotone path of one of them
merge with the lower path of the other, then, once $\lambda$ becomes ready, $\text{B}(\lambda)$ is
changed to the rightmost of $\text{B}(\lambda)$ and all $\text{B}$ relations from limbs that have
merged into $\lambda$. We call these $\text{B}$ relations the candidates for $\text{B}(\lambda)$. (Note
that the candidates are dead, so $\lambda$ only has to maintain the rightmost of them.)
\item[5:] Similarly, if $v$ is $\text{E}_L(\Lambda)$, then all limbs with relations to $\Lambda$ have their relations
changed to $v$ and $\Lambda$ is again merged into another limb $\lambda$.
\item[6:] Finally, if the two merging limbs meet at their lower $x$-monotone paths, then $\text{B}(\Lambda)$
becomes a candidate to $\text{N}_L(\lambda)$.
\end{algorithmic}
\end{algorithm}
Correctness of the algorithm of the first phase. We argue that the algorithm for the first phase contracts all right pointing limbs to their blocking endpoints.

Notice first, that all limbs are contracted eventually since the $x$-coordinates of the peaks of the limbs of $B$, $N_U$ and $N_L$ relations are less than the $x$-coordinate of the peak of limbs themselves, so the relations form a (non-cyclic) partial ordering on the limbs. The correctness will thus follow from proving that any limb either gets removed or contracted to its blocking endpoint (or another limb contracted to this endpoint).

Let $\Lambda$ be a limb. We define the area $A(\Lambda)$ to be the enclosed area bounded by $U(\Lambda)$ above, $L(\Lambda)$ below and a vertical line intersecting the rightmost of $E_U(\Lambda)$ and $E_L(\Lambda)$ to the left–refer to Figure 5.3.

The correctness of the algorithm of phase 1 follows from Lemma 5.2:

**Lemma 5.2.** Let $\Lambda$ be any limb which is not removed during the algorithm of phase 1. Consider the point in time during the algorithm right before the $\Lambda$ is contracted. Let $p$ be the rightmost endpoint inside of $A(\Lambda)$ and $\lambda$ be the last of the limbs which have contracted to $p$ (or the limb with $p$ as peak if no limb contracts to $p$). When handling $\Lambda$ we have $B(\Lambda) = \lambda$.

**Proof.** We refer to the point in time right before $\Lambda$ is contracted as $s$ and the time at the beginning of the algorithm (before any contraction takes place) as $s_0$.

Let the limb $\Lambda$ be the result from merging the limbs $\Lambda_1, \Lambda_2, \ldots, \Lambda_d$, $d \geq 0$ into $\Lambda$ up until time $s$ and recall that $B(\Lambda)$ is chosen as the rightmost of the dead peaks $B(\Lambda_1), \ldots, B(\Lambda_d)$ and $B(\Lambda)$ when handling $\Lambda$.

The first step of this proof is to prove that at time $s_0$ there exists a chain $C = \lambda_0, \lambda_1, \ldots, \lambda_c$ of peaks, where $c \geq 0$, $\Lambda_0 = \lambda$, $\lambda_c = B(\Lambda_i)$ and for any $0 < j \leq c$: $\lambda_{j-1} \in \{N_U(\lambda_j), N_L(\lambda_j), B(\lambda_j)\}$. If any of these conditions do not hold during the algorithm, then we say $C$ is broken.

Consider the enclosed area $A$ bounded by the limbs $\Lambda_0, \ldots, \Lambda_d$ and a vertical line intersecting $p$ at time $s_0$. Since the algorithm simply contracts right pointing limbs, the area $A$ always contains $\lambda$ and all the non-dead peaks of $C$. Proving that $C$ exists is simply a consequence of the construction of the relations:

Recall that we traverse right from every peak and set up the relations to other peaks. Since $A$ encloses the area in which all the peaks of the chains
5.2. Homotopic X-Shortest Paths Algorithm

are connected, and since any boundary but the left boundary of A consists of limbs, any limb inside of it will be related to at least one other limb during the construction, so there is at least one way to construct the chain of peaks $C$ from $\lambda$ to $B(\Lambda_i)$.

The second step of this proof is to prove that $C$ will not be broken when contracting limbs. This follows easily from the following observations:

1. Since a limb is first contracted after it has been made ready, the limbs of $C$ contract in order $\lambda_0, \lambda_1, \lambda_2, \ldots$.

2. Since $p$ is the rightmost endpoint inside of A, any limb $\lambda'$ with Peak($\lambda'$) inside of A either contracts to $p$, some other limb which has been contracted to $p$, some point to the left of $p$, or is removed altogether.

For $c = 0$ it is easy to see that the lemma holds, so assume $c > 0$. The first observation gives us that the first limb of the chain that contracts is $\lambda_0 = \lambda$. $\lambda$ (by definition) contracts to $p$ or some other limb which has contracted to $p$, and since $\lambda$ is a candidate for either the $B(\lambda_1)$, $N_U(\lambda_1)$ or $N_L(\lambda_1)$, and no other candidate has its peak to the right of $p$ (observation (2)), the chain remains unbroken when contracting $\lambda$.

For any contracting limb $\lambda_i \in C$, $i > 0$, the observations give that $\lambda_i$ is removed due to the contraction, and the handling of relations in step 5 to 7 of algorithm 1 ensures that the chain remains after $\lambda_i$ is removed, that is, that $\lambda$ (which is either $B(\lambda_i)$, $N_U(\lambda_i)$ or $N_L(\lambda_i)$ right before contracting $\lambda_i$) becomes a candidate for either $N_U(\lambda_{i+1})$, $N_U(\lambda_{i+1})$ or $B(\lambda_{i+1})$ once $\lambda_{i+1}$ is contracted.

Observation (2) also gives that no limb $\lambda' \notin C$ can become a chosen candidate for $B(\lambda_i)$, $N_U(\lambda_i)$ or $N_L(\lambda_i)$ of any limb $\lambda_i \in C$ once $\lambda_i$ is contracted.

Analysis of the first phase. The algorithm for the first phase contracts all limbs. Every limb is contracted once. A contraction causes some points to be removed and two points to be added to the input paths. Since a limb is first contracted when it is ready, that is, all its own relations are dead, the $N_U(\Lambda)$, $N_L(\Lambda)$ and $B(\Lambda)$ relations of any limb $\Lambda$ can at most change once (if they change, they always change to dead peaks or are removed altogether). Contracting all limbs is thus done in linear time of the size and number of limbs:

Theorem 5.3. Given the limbs of $n$ paths $P$ with the combined size $m$, and all the relations of the right pointing limbs, contracting all right pointing limbs is done in $O(m)$ time.

The second phase: Contracting all left pointing limbs. For the second phase of the algorithm we take the resulting paths from the first phase, and for any point $p = (x, y)$ in these paths we flip the sign of the $x$-coordinate, that is, replace it with $-x$. After flipping the signs, the paths have been transformed as if mirrored in the $y$-axis and we repeat the whole algorithm from the beginning on these paths, that is, construct a trapezoidal decomposition, construct the limbs, construct their relations, and contract them. After contracting the limbs we transform the paths back by flipping the signs of the $x$-coordinates.
Chapter 5. Homotopic X-Shortest Paths

Figure 5.4: (i) The original paths. (ii) The limbs $\lambda$ and $\Lambda$ are contracted during the first phase. (iii) After the second phase the limb $\Lambda$ is degenerate and can be contracted further in the third phase.

again. The left pointing limbs of the resulting paths, denoted $P'$, can not be contracted any further, but there might now be right pointing limbs which can be contracted—refer to Figure 5.4.

Let $\Lambda$ be a right pointing limb of $P'$. If at the peak of $\Lambda$ there are two partially overlapping vertical line segments sharing a point, then $\Lambda$ is a degenerate limb and the point that the vertical line segments share is the peak $\text{Peak}(\Lambda)$. Notice that since we have contracted all right pointing limbs in the first phase and all the left pointing limbs in the second phase, any right pointing that can be contracted further after the second phase must be a degenerate right pointing limb. We handle the degenerate limbs in the third phase.

Analysis for the second phase. A contraction removes at least one point from a limb and adds at most two new points, so the paths in the output of the first phase contain at most $n$ paths with at most $2m$ points. The running time of the algorithm of the second phase is thus $O(n \log^{1+\epsilon} n + m)$ for constructing the decomposition, limbs and relation (theorem 5.1), $O(n + m)$ for contracting the limbs (theorem 5.3) and $O(m)$ for flipping the signs of the $x$-coordinates twice:

Theorem 5.4. The combined time for contracting the $n$ simple paths $P$ with the combined size of $m$ up until the end of the second phase where the homotopic paths $P'$ have been computed is $O(n \log^{1+\epsilon} n + m)$.

The third phase: Handling degeneracies. All degenerate limbs can be found by scanning through the segments from the output paths $P'$ of the second phase. We handle the degenerate limbs of $P'$ in order to obtain homotopic $x$-shortest paths $P''$.

Let $\Lambda$ be a degenerate limb, $E_R(\Lambda)$ be the point of $E_U(\Lambda)$, $E_L(\Lambda)$ with $x$-coordinate closest to the $x$-coordinate of $\text{Peak}(\Lambda)$, and $p$ be the endpoint in $A(\Lambda)$ with $x$-coordinate closest to the $x$-coordinate of $\text{Peak}(\Lambda)$. $p$ is called the blocking endpoint of $\Lambda$. $\Lambda$ is contracted to $p$ if it exists. If not, $\Lambda$ is contracted to $E_R(\Lambda)$, which might result in a new degenerate left pointing limb. Contracting a degenerate left pointing limb can similarly result in a new degenerate right
pointing limb. Refer to Figure 5.4 for an example where a degenerate (right pointing) limb causes a degenerate left pointing limb.

Our algorithm to handle all the degenerate limbs is now simple to describe. Given the paths $\mathcal{P}'$ we construct a trapezoidal decomposition of $\mathcal{P}'$ which enables us to compute the blocking endpoint of any degenerate limb by simply traversing the decomposition interior to the limb. After computing the decomposition we traverse all paths in order to find the degenerate limbs. Any degenerate limb is contracted and if the contraction results in another degenerate limb, then this limb is contracted immediately thereafter.

**Correctness of the algorithm of the third phase.** We have argued that only degenerate limbs can be contracted, and contracting a degenerate limb might result in another degenerate limb to be contracted in the other direction. Our algorithm simply searches for and contracts all degenerate limbs, and since paths can only be $x$-shortened by contracting limbs, it is easy to realise that we end up with $x$-shortest paths after the third phase, given that our algorithm for contracting degenerate limbs is correct. To prove that our algorithm for contracting a degenerate limb is correct we need to prove that we preserve homotopy and the blocking endpoints are found correctly. For this we make the following observation: Both the first and last line segment of any maximal $x$-monotone path in $\mathcal{P}'$ touch endpoints.

From the observation it follows that any degenerate limb has its peak at an endpoint, and when contracting, it contracts to another endpoint. Homotopy is preserved because if there is some limb $\lambda$ inside a contracting degenerate limb $\Lambda$, then $\lambda$ will have (blocking) endpoints at its own (limb) endpoints, and $\Lambda$ contracts to the endpoint with $x$-coordinate closest to the $x$-coordinate of Peak($\Lambda$).

If a degenerate limb $\Lambda$ has a blocking endpoint then the observation gives that there are no limbs between the blocking endpoint and Peak($\Lambda$), and that this area remains unchanged no matter what happens before $\Lambda$ is contracted. The blocking endpoint is thus correctly found by simply traversing the trapezoidal decomposition of the paths $\mathcal{P}'$.

**Analysis of the algorithm of the third phase.** By the same argument as for the second phase the decomposition of the third phase is constructed in $O(n \log^{1+\epsilon} n + m)$ time. Contracting the limbs this time is easy and can be done in a few linear scans of the paths of $\mathcal{P}'$ and any trapezoid of the trapezoidal decomposition is only traversed once when searching for the blocking endpoints. We thus arrive at the final running time for our algorithm to compute homotopic $x$-shortest paths:

**Theorem 5.5.** The combined time for computing homotopic $x$-shortest paths of $n$ simple paths $\mathcal{P}$ with the combined size of $m$ is $O(n \log^{1+\epsilon} n + m)$. 
5.3 Lower Bound for Computing Homotopic X-Shortest Paths

In this section we present our proof for a lower bound of the problem of computing homotopic $x$-shortest paths. This result comes from a simple reduction to sorting.

**Lemma 5.3.** The lower bound for computing homotopic $x$-shortest paths for $n$ simple paths with the combined size of $m$ is $\Omega(n \log n)$.

**Proof.** We prove the lower bound by reducing the problem to sorting in the algebraic decision tree model. In the algebraic decision tree model the lower bound for sorting $n$ numbers is $\Omega(n \log n)$.

Let $x_1, \ldots, x_n$ be $n$ unsorted numbers. We construct $2n$ simple paths and show how the corresponding homotopic shortest paths can be used to compute the sorted order of the numbers. For any number $x_i$ we construct the path with the points $(0, x_i)$, $(x_i, x_i)$, $(x_i, -x_i)$, $(0, -x_i)$ (a box without the left side) and the path consisting of the single point $(x_i, x_i)$, where we make sure that the comparator for the points makes this point greater than the overlapping point for the path. This is a simple extension on the lexicographic comparator.

Computing the homotopic $x$-shortest paths on these paths will cause any box shaped path for a number $x_i$ to contract to the path consisting of a point corresponding to the predecessor $x_j$. The relation between any number and its predecessor is found, so the sorted sequence of numbers can be computed trivially in linear time.

The lower bounds since computing the homotopic $x$-shortest paths gives us the sorted sequence. \hfill $\Box$

The input and output to the algorithm has size $m$, so the larger lower bound follows from Lemma 5.3:

**Corollary 5.1.** The lower bound for computing homotopic $x$-shortest paths for $n$ simple paths with the combined size of $m$ is $\Omega(n \log n + m)$ in the algebraic decision tree model.

5.4 Conclusion

We have developed a new algorithm for computing homotopic $x$-shortest paths and used it to improve the best known algorithms for homotopic shortest paths. The bottleneck in our algorithm is the computation of the decomposition, which takes $O(n \log^{\epsilon+1} n + m)$ time, for $\epsilon > 0$, and $O(n + m)$ space.

We have proven the lower bound of $\Omega(n \log n + m)$, which is equal to the lower bound for computing the decomposition. If it is possible to compute the decomposition in the optimal time of $O(n \log n + m)$ [16], then the running time of our algorithm improves to the same.
Chapter 6

Computing Homotopic Line Simplification in a Plane

The paper Computing Homotopic Line Simplification in a Plane presented in this Chapter has been published as an extended abstract at European Workshop on Computational Geometry (EuroCG) 2011 [30] and submitted as a journal version to Computational Geometry: Theory and Applications (CGTA) 2012 [1].


The journal version extends the extended abstract with the first polynomial-time restricted algorithm which computes the optimal homotopic simplification of a polygonal path. We present the journal version in this chapter with minor typographical changes.
Computing Homotopic Line Simplification in a Plane

Shervin Daneshpajouh*  Mohammad Ali Abam*
Lasse Deleuran†  Shayan Ehsani*  Mohammad Ghodsi*

Abstract

We study a variant of the line-simplification problem where we are given a polygonal path \( P = p_1, p_2, \ldots, p_n \) and a set \( S \) of \( m \) point obstacles in a plane, and the goal is to find the optimal homotopic simplification, that is, a minimum subsequence \( Q = q_1, q_2, \ldots, q_k \) \((q_1 = p_1 \text{ and } q_k = p_n)\) of \( P \) defining a polygonal path which approximates \( P \) within the given error \( \varepsilon \) and is homotopic to \( P \). We assume all shortcuts \( p_i p_j \) whose errors under a distance function \( F \) are at most \( \varepsilon \) can be computed in \( T_F(n) \) time where \( T_F(n) \) is polynomial for all widely-used distance functions. We present the first polynomial-time algorithm which computes the optimal homotopic simplification in \( O(n^6m^2) + T_F(n) \) time. Moreover, we define the new concept of strongly homotopic simplification where every link \( q_lq_{l+1} \) of a simplification \( Q \) corresponding to the shortcut \( p_i p_j \) of a given path \( P \) is homotopic to the sub-path \( p_i, \ldots, p_j \) of \( P \). We present a general method running in time \( O(n(m + n)\log(nm)) \) for identifying every shortcut \( p_i p_j \) that is homotopic to the sub-path \( p_i, \ldots, p_j \) of \( P \), called a homotopic shortcut. In the case of \( x \)-monotone paths we propose an efficient and simple method to compute all homotopic shortcuts in \( O(m\log(nm) + n\log n\log(nm) + k) \) time where \( k \) is the number of homotopic shortcuts. Under any desired measure \( F \), both methods can be simply combined with Imai and Iri’s framework to obtain the optimal strongly-homotopic simplification in \( O(n(m + n)\log(nm)) + T_F(n) \) and \( O(m\log(nm) + n\log n\log(nm) + k) + T_F(n) \) time for general paths and \( x \)-monotone paths respectively.

6.1 Introduction

Motivation  Suppose we want to visualize a large geographical map as a collection of non-intersecting chains representing some features like rivers or country borders, and points representing places like cities, at different levels of details. This leads us to simplify the map. A simplified map must resemble the original map, that is, it must satisfy conditions (i) the distance between each point on an original chain and the simplified chain is within a given error tolerance, and (ii) each original chain and the simplified chain must be in

---

*Computer Engineering Department, Sharif University of Technology, Iran.
†Center for Massive Data Algorithmics at Computer Science Department of Aarhus University, Denmark.
the same homotopy class; roughly speaking it means that if a point (city for instance) is below a chain (river for example), the point must remain below the simplified chain. We, however, pay our attention to a simpler variant of the above problem where we are given a polygonal path \( P = p_1, p_2, \ldots, p_n \) and a set \( S \) of \( m \) point obstacles in a plane, and we wish to simplify the path \( P \) such that the simplified path is close enough to stay within the given error tolerance of the original path, while still homotopic to the original path.

**Background** The problem mentioned above is a variant of the line-simplification problem. In the line-simplification problem—also known as path, curve and chain simplification in the literature—one is given a polygonal path \( P = p_1, p_2, \ldots, p_n \) in a plane, and the goal is to find a path \( Q = q_1, q_2, \ldots, q_k \) (\( q_1 = p_1 \) and \( q_k = p_n \)) with fewer vertices approximating the path \( P \) within the given error tolerance \( \epsilon \). This problem arises in many applications like GIS [33–35], image processing and computer graphics [21, 43] where the goal is to perform data reduction on a polygonal shape in a plane in order, for instance, to reduce the complexity of costly processing operations. In these applications, preserving the homotopy of the shape plays an important role. Indeed, it ensures that after reduction, the *above*ness relation\(^1\) of points and chains remains unchanged.

Many variants of the line-simplification problem have been considered in computational geometry which can be classified into two classes, namely *unrestricted version* and *restricted version*. In the former, the vertices of a simplification \( Q \) are allowed to be arbitrary points, not just the vertices of \( P \)—see [48, 51, 58] for some results. In the restricted version, the vertices of \( Q \) are a subsequence of \( P \) and each \( q_l q_{l+1} \) is called a *link*. In this paper, we are interested in the restricted version, so we dedicate our notation and definitions to this class.

Each link \( q_l q_{l+1} \) of the simplified path corresponds to a shortcut \( p_i p_j \) (with \( j > i \)) of the original path \( P \), and the error of the link is defined as the distance between \( p_i p_j \) and the sub-path \( p_i, \ldots, p_j \) denoted by \( P(i, j) \). To measure the distance between \( p_i p_j \) and \( P(i, j) \) the Hausdorff distance and the Fréchet distance are often used. The *error* of the simplification \( Q \) denoted by error\((Q, P)\) is now defined as the maximum error of any of its links. For each error function, there are two constrained optimization problems that are of interest: (i) min-\( k \) problem: given a path \( P \) and a maximum error \( \epsilon \), compute a simplification \( Q \) of path \( P \) with the minimum number of vertices such that error\((Q, P)\) \( \leq \epsilon \). (ii) min-\( \epsilon \) problem: given a path \( P \) and a maximum number of vertices \( k \), compute a simplification \( Q \) of \( P \) with the smallest possible error that uses at most \( k \) vertices. The min-\( \epsilon \) is usually computed by performing a binary search over the pre-computed errors and applying a min-\( k \) algorithm at each step of the binary search. Hence, from now on we focus on the min-\( k \) problem in the restricted model and for ease of writing, we simply write simplification instead of min-\( k \) line simplification in the restricted model if no confusion arises.

\(^1\)Object \( p \) (point, segment, curve, ...) is above object \( q \) if there exists a vertical line intersecting both \( p \) and \( q \) with the intersection of \( p \) being above \( q \).
Fig. 6.1: (i) Two simplifications $Q$ and $Q'$ of path $P$. Only $Q$ is homotopic to $P$, (ii) While the simplified path $Q$ and original path $P$ are homotopic, some shortcuts like $p_ip_i$ and $p_jp_k$ are not homotopic to $P(1,i)$ and $P(j,k)$ respectively. Therefore, $P$ and $Q$ are not strongly homotopic.

Let $S$ be a set of $m$ point obstacles in the plane that do not intersect $P$. A simplification $Q$ is homotopic to path $P$ if it is continuously deformable to $P$ without passing over any points of $S$ while keeping its end-vertices fixed. Precisely, two paths $\alpha, \beta : [0, 1] \rightarrow \mathbb{R}^2$ sharing starting and ending endpoints are homotopically equivalent with respect to obstacle set $S$ if there exists a continuous function $\Gamma : [0, 1] \times [0, 1] \rightarrow \mathbb{R}^2$ with the following properties:

(i) $\Gamma(0, t) = \alpha(t)$ and $\Gamma(1, t) = \beta(t)$ for $0 \leq t \leq 1$,

(ii) $\Gamma(s, 0) = \alpha(0) = \beta(0)$ and $\Gamma(s, 1) = \alpha(1) = \beta(1)$,

(iii) $\Gamma(s, t) \notin S$ for $0 \leq s \leq 1$ and $0 < t < 1$.

Fig.6.1(i) illustrates two simplifications $Q$ and $Q'$. The simplification $Q$ is homotopic to $P$ but $Q'$ is not; so only $Q$ is of our interest. In some applications, we do not only need a simplification to be homotopic to the original path but we also need every shortcut used in the simplification to be homotopic to its corresponding sub-path. This leads us to define the concept of strongly homotopic simplification as follows. A path $P$ and its simplification $Q$ are strongly homotopic if for any link $q_{q+1}$ of $Q$ corresponding to the shortcut $p_ip_j$, the sub-path $P(i, j)$ and $p_ip_j$ are homotopic. For ease of presentation, such a shortcut is called a homotopic shortcut. If a path $P$ and its simplification $Q$ are strongly homotopic, they are clearly homotopic as well but if $P$ and $Q$ are homotopic, they are not necessarily strongly homotopic as an example is shown in Fig. 6.1(ii). However, if $P$ is $x$-monotone, we can simply conclude $P$ and $Q$ are strongly homotopic if they are homotopic\(^2\).

**Related work** There are many results on line simplification under different error criteria, though most of them do not generate homotopic simplifications. The oldest and most popular algorithm in the restricted setting is the Douglas-Peucker algorithm [35, 56]. However, this algorithm is only a heuristic and it is

\(^2\)The simple proof is as follows. For the sake of contradiction, assume $p_ip_j$ is not homotopic to subpath $P(i, j)$. Then there is an obstacle point $O$ above $p_ip_j$ and below $P(i, j)$ (or vice versa). Since $Q$ is $x$-monotone and $p_ip_j$ is a link of $Q$, then $O$ must be above $Q$. Therefore $O$ is above $Q$ and below $P$ which contradicts with the assumption $Q$ and $P$ are homotopic.
not guaranteed to be optimal (in terms of the number of vertices used). Imai
and Iri [59] solved both the min-ε and the min-k versions by modeling each
version as a shortest-path problem on directed acyclic graphs. The running
time of their method was proved to be quadratic or near quadratic by Chin
and Chan [25], and Melkman and O’Rourke [67] for the Hausdorff distance under
the Euclidean metric. We refer readers to [2, 4] and reference therein for more
results.

Recently geometers have been interested in quickly determining whether two
paths are homotopic [22], or computing the shortest path being homotopic to a
path [20] or deforming a path to another path with a minimum cost [19,38]. In
the context of the line simplification, De Berg et al. [33, 34] were first to study
homotopic line simplification in the restricted model for the Hausdorff distance
under the Euclidean metric. Their algorithm running in \(O(n(n+m) \log n)\) time
finds the optimal simplification provided that \(P\) is \(x\)-monotone; otherwise the
simplification is not guaranteed to be optimal. Guibas et al. [51] showed that
the optimal homotopic line-simplification problem in the unrestricted model is
NP-hard when the simplification is forced to be non-self-intersecting like the
original one. A general version, subdivision simplification, was later proved to
be MIN PB-complete and some heuristic algorithms were given by Estkowski
and Mitchell [42].

**Our results**  The first step of computing the optimal homotopic simplification
is to compute all shortcuts whose errors are within the given error \(\varepsilon\) which can be
performed by existing algorithms as mentioned above. To compute the optimal
strongly-homotopic simplification, we must compute the homotopic shortcuts
out of the computed shortcuts in the first step. We can simply compute them
in \(O((n^3 + n^2 m) \log(nm))\) as we will explain in Section 6.3 but the challenging
question is how to compute them efficiently. In this paper, we propose
two efficient algorithms: For \(x\)-monotone paths \(P\) of length \(n\), our algorithm
computes all homotopic shortcuts in \(O(m \log(nm) + n \log n \log(nm) + k)\) time
where \(k\) is the number of homotopic shortcuts. In the case of general paths \(P\)
of length \(n\), we propose an \(O(n(m + n) \log(nm))\)-time algorithm to compute
all homotopic shortcuts. Both algorithms can be combined with Imai and Iri’s
general framework to compute the optimal strongly-homotopic simplification.
Computing homotopic shortcuts is not sufficient to compute the optimal homotopic
simplification for general paths as Fig. 6.1(ii) illustrates an example
where simplification \(Q\) is homotopic to \(P\) but some of its links are not homotopic
shortcuts. One easy way is to look at every simplification \(Q\) of \(P\) in the
increasing order of their lengths and test whether \(Q\) is homotopic to \(P\). The
running time of this algorithm is not polynomial as we know there may exist
an exponential number of simplifications. In this paper, we present the first
polynomial-time algorithm that computes the optimal homotopic simplification
for general paths. This algorithm can be applied to non-simple paths as well.

The paper is organized as follows: In Section 6.2 we recall some existing
approaches and definitions. Section 6.3 presents our algorithm to compute the
optimal strongly-homotopic simplification. In Section 6.4 we present the first
6.2 Preliminaries

Imai and Iri’s framework Suppose \( P = p_1, \ldots, p_n \) is a polygonal path in a plane. Most simplification algorithms given in the restricted model are based on Imai and Iri’s general framework in which for a given error \( \varepsilon \), an unweighted directed graph \( G_\varepsilon(P) \) (or simply \( G_\varepsilon \)) is defined as follows: \( G_\varepsilon = (V, E_\varepsilon) \) where 

\[ V = \{p_1, \ldots, p_n\} \text{ and } E_\varepsilon = \{(p_i, p_j) | d_F(p_ip_j, P(i, j)) \leq \varepsilon\} \]

where \( d_F(p_ip_j, P(i, j)) \) is the \( F \) distance of shortcut \( p_ip_j \) and \( P(i, j) \) for some error function \( F \). Each simplification \( Q \) with the error of at most \( \varepsilon \) corresponds to a path in \( G_\varepsilon \) from \( p_1 \) to \( p_n \) and therefore, the optimal simplification is the shortest path (i.e. the path with the minimum number of edges) in \( G_\varepsilon \) from \( p_1 \) to \( p_n \) which can be computed by a breadth first search in time \( O(|G_\varepsilon|) \).

Canonical sequences Suppose \( \alpha \) is a path in a plane in which there are some point obstacles not intersecting \( \alpha \). One way of representing \( \alpha \) is to write the sequence of obstacles that it passes above (overbar) and below (underbar). Precisely, imagine that a vertical line is drawn from each point obstacle. We walk along \( \alpha \) from the source \( s \) to the destination \( t \). Whenever we reach a vertical line drawn from an obstacle, say \( A \), if we are above \( A \), we put symbol \( \overline{A} \) in the sequence and otherwise we put symbol \( \underline{A} \) in the sequence. For example, the sequence of \( \alpha \) illustrated in Fig. 6.2(a) is \( \overline{ABCDDCCCBA} \). If two adjacent symbols in the sequence of \( \alpha \) are equal (i.e. both adjacent symbols are \( \overline{A} \) or \( \underline{A} \) for some obstacle \( A \)), both symbols can be deleted by deforming (or shortening) \( \alpha \) without changing the homotopy class. Fig. 6.2(b) illustrates a deformation that deletes \( \overline{DDD} \) in the sequence of \( \alpha \). This deletion can be continued until a canonical sequence (denoted by \( \text{CS}(\alpha) \)) is obtained. Notice that in \( \text{CS}(\alpha) \), two adjacent symbols are always different—Fig. 6.2(c) illustrates \( \alpha \) after several deformations whose sequence is exactly the CS of the original \( \alpha \). It is known [22] that two paths \( \alpha \) and \( \beta \) sharing their endpoints are homotopic if and only if \( \text{CS}(\alpha) = \text{CS}(\beta) \). From now on whenever we say we shorten a path, we mean we deform the path in order to finally compute its canonical sequence.

6.3 Optimal strongly-homotopic simplification

Suppose \( P = p_1, \ldots, p_n \) is a non-self-intersecting polygonal path in a plane that includes \( m \) points as obstacles. Our general algorithm is as follows. We first compute \( G_\varepsilon \) in the absence of the obstacles under the given error function \( F \), and then test whether each edge \( p_ip_j \) in \( G_\varepsilon \) is homotopic to \( P(i, j) \) in the presence of the obstacles. We remove non-homotopic shortcuts from \( G_\varepsilon \) and finally compute the shortest path in \( G_\varepsilon \) from \( p_1 \) to \( p_n \) by a breadth first search to obtain the optimal strongly-homotopic simplification.
Chapter 6. Computing Homotopic Line Simplification in a Plane

The main step of our algorithm is how to compute homotopic shortcuts; the rest exists in the literature as already mentioned. This step can be done in $O((n^3 + n^2m) \log(nm))$ time by individually applying the homotopy-testing algorithm given in [22] to each edge in $G_\varepsilon$ and its corresponding sub-path. This, of course, is far from being efficient. Note that for the most widely used distance functions, either $T_F(n)$ is near quadratic or an approximation graph of $G_\varepsilon$ (i.e. the error of each edge in the approximation graph is at most $c\varepsilon$ for some constant $c$) can be computed in near quadratic time. Therefore, the bottleneck is computing homotopic shortcuts and thus the challenging question is whether it is possible to compute all homotopic shortcuts faster. We affirmatively answer this question using some novel ideas and nice observations. Precisely, we present two algorithms in the remainder of this section to compute homotopic shortcuts for $x$-monotone paths and general paths in the total time of $O(m \log(nm) + n \log(nm) + k)$ and $O(n(m+n) \log(nm))$ respectively where $k$ is the number of homotopic shortcuts, $n$ is the length of the path and $m$ is the number of the obstacles—see Lemmas 6.2 and 6.5. Therefore, we get our main results.

**Theorem 6.1.** Suppose $P$ is a non-self-intersecting polygonal path with size $n$ in a plane containing $m$ point obstacles. Suppose that $T_F(n)$ is the time needed to compute $G_\varepsilon$ under the error function $F$ in the absence of the obstacles. Then (i) if $P$ is $x$-monotone, the optimal strongly-homotopic simplification can be computed in $T_F(n) + O(m \log(nm) + n \log(nm) + k)$ time where $k$ is the number of homotopic shortcuts. (ii) if $P$ is a general path, the optimal strongly-homotopic simplification can be computed in $T_F(n) + O(n(m+n) \log(nm))$ time.

### 6.3.1 Computing homotopic shortcuts for $x$-monotone paths

Let $P = p_1, \ldots, p_n$ be an $x$-monotone polygonal path and let $S = \{s_1, \ldots, s_m\}$ be a set of point obstacles in a plane under the assumption of general position. The main idea behind our algorithm is to enclose $P$ inside a simple polygon $\Psi(P,S)$ such that $p_i p_j$ is a homotopic shortcut if and only if $p_i$ can see $p_j$ inside the simple polygon $\Psi(P,S)$.

We define the simple polygon $\Psi(P,S)$ as follows. We first determine the aboveness relation of the obstacles and path $P$ in $O(m \log n)$ time by locating each obstacle with respect to the path $P$ in $O(\log n)$ time. We then compute...
6.3. Optimal strongly-homotopic simplification

in $O(n+m)$ time an axis-parallel rectangle containing the path vertices and obstacles in its interior. Let $\lambda$ be a sufficiently small number less than the $x$-coordinate difference of any two points of the path vertices and obstacles—$\lambda$ can simply be computed in $O((n+m) \log(nm))$ time. For each obstacle $s_i$ above $P$, we define two points $s_i^+$ and $s_i^-$ on the upper boundary of the rectangle with $x$-coordinates $x(s_i) + \lambda$ and $x(s_i) - \lambda$ respectively where $x(s_i)$ is the $x$-coordinate of $s_i$. We connect $s_i$ to $s_i^+$ and $s_i^-$ and remove the part of the rectangle joining $s_i^+$ to $s_i^-$. Similarly, we define $s_i^+$ and $s_i^-$ and the associated edges for obstacles $s_i$ below $P$. All together, this gives us the simple polygon $\Psi(P,S)$.

**Lemma 6.1.** A shortcut $p_ip_j$ is a homotopic shortcut if and only if $p_i$ can see $p_j$ inside $\Psi(P,S)$.

**Proof.** Suppose for the sake of contradiction that $p_i$ is vertically above an edge $s_js_j^-$ (or $s_js_j^+$) on the upper boundary of $\Psi(P,S)$ for some $j$. Then $x(s_j) - x(s_j^-)$ must be bigger than $x(s_i) - x(p_i)$ contradicting the choice of $\lambda$. A similar argument can be applied to edges on the lower boundary. Therefore $p_i$ must be inside $\Psi(P,S)$. If $p_i$ can see $p_j$ inside $\Psi(P,S)$, then $p_ip_j$ and $P(i,j)$ make a cycle inside the simple polygon $\Psi(P,S)$ which can be simply deformed to a single point implying that $p_ip_j$ and $P(i,j)$ are homotopic. Now assume $p_ip_j$ is a homotopic shortcut. Suppose for the sake of contradiction that $p_i$ and $p_j$ are not visible to each other inside $\Psi(P,S)$. This means that $p_ip_j$ must intersect both $s_k^-s_k^-$ and $s_k^-s_k^+$ for some $k$—note that both $p_i$ and $p_j$ are inside $\Psi(P,S)$. This simply implies $s_k$ is between the shortcut $p_ip_j$ and $P(i,j)$ contradicting $p_ip_j$ being a homotopic shortcut.

The above lemma reduces the problem to computing the visibility graph of $n$ points inside a simple polygon with size $3m + 4$. Thanks to Ben-Moshe et al. [17] who presented an $O(m+n \log n \log (nm) + k)$-time algorithm to compute all visible pairs in $O(n + m + k)$ space, where $k$ is the number of visible pairs. Putting it all together we get the following result.

**Lemma 6.2.** Let $P = p_1, \ldots , p_n$ be an $x$-monotone polygonal path and let $S$ be a set of $m$ point obstacles in a plane. All homotopic shortcuts $p_ip_j$ can be computed in $O(m \log (nm) + n \log n \log (nm) + k)$ time and $O(n + m + k)$ space where $k$ is the number of homotopic shortcuts.
6.3.2 Computing homotopic shortcuts for general paths

Suppose \( P = p_1, \ldots, p_n \) is a non-self-intersecting polygonal path in a plane in the presence of point obstacles \( S = \{s_1, \ldots, s_m\} \). We first briefly describe the algorithm given in [22] which tests whether two non-self-intersecting paths \( \alpha \) and \( \beta \) are homotopic or not. Then we explain how to exploit this algorithm to get an efficient algorithm to find all homotopic shortcuts of path \( P \).

The algorithm by Cabello et al. [22] The algorithm tests whether two non-self-intersecting paths \( \alpha \) and \( \beta \) with common fixed endpoints and total size \( n \) in the presence of \( m \) point obstacles are homotopic or not. Their algorithm first computes \( \text{CS}(\alpha) \) and \( \text{CS}(\beta) \). To efficiently compute \( \text{CS}(\alpha) \) (\( \text{CS}(\beta) \) as well), the algorithm works as follows. Path \( \alpha \) is first decomposed into maximal \( x \)-monotone sub-paths. The obstacles and \( x \)-monotone sub-paths induce an aboveness relation that is acyclic. The aboveness relation defines a partial order which can be extended to a total order. The path \( \alpha \) can now be represented in the rectified form, so-called rectified path, where each maximal \( x \)-monotone sub-path is treated as a horizontal segment whose \( y \)-coordinate is its rank in the total order. The rectified path is then shortened to obtain the canonical rectified path (denoted by \( \text{CRP}(\alpha) \)) representing \( \text{CS}(\alpha) \)—see Fig. 6.4 illustrating the initial steps of the algorithm.

To test whether paths \( \alpha \) and \( \beta \) are homotopic, it first tests whether turn obstacles of \( \alpha \) and \( \beta \) define the same set where a turn obstacle \( O \) of \( \alpha \) is an obstacle at which \( \text{CRP}(\alpha) \) makes an \( U \)-turn or precisely either \( \overline{OQ} \) or \( \overline{OQ} \) exists in \( \text{CS}(\alpha) \). If not, clearly \( \alpha \) and \( \beta \) are not homotopic. Otherwise, both \( \text{CRP}(\alpha) \) and \( \text{CRP}(\beta) \) are broken at the turn obstacles and consequently decomposed into \( x \)-monotone sub-paths. Then each sub-path of \( \text{CRP}(\alpha) \) and its corresponding one in \( \text{CRP}(\beta) \) (i.e. the one ending at the same turn obstacles) are tested to see whether they are homotopic. These testings are together performed by simultaneously sweeping \( x \)-monotone sub-paths of \( \text{CRP}(\alpha) \) and \( \text{CRP}(\beta) \) from left to right.

Our algorithm In the rest of this section, we describe how we can exploit the algorithm by Cabello et al. to compute all homotopic shortcuts \( p_ip_j \) in \( O(n(m+n) \log(nm)) \) time. We fix \( i = 1 \) and show homotopic shortcuts \( p_ip_j \) can be altogether computed in \( O((m+n) \log(nm)) \) time which trivially can be extended to any \( i \) obtaining our main result as \( i \) changes from 1 to \( n \).
6.3. Optimal strongly-homotopic simplification

Our global strategy is as follows. We inductively compute \( CRP(P(1,i)) \) (for short, \( CRP(i) \)). If \( CRP(i) \) has a turn obstacle, it cannot be homotopic to the shortcut \( p_1p_i \) as the CRP of \( p_1p_i \) does not have any turn obstacle. Therefore, we maintain \( CRP(i) \) provided that it is \( x \)-monotone. But unfortunately a separate maintenance of all \( x \)-monotone \( CRP(i) \) may require \( O(n^2) \) space and time which results in an inefficient \( O(n^3) \)-time algorithm to compute all homotopic shortcuts \( p_1p_j \). Hence, we encode all \( x \)-monotone \( CRP(i) \) into a tree \( \sqcap \) of size \( O(n) \) where each \( x \)-monotone \( CRP(i) \) corresponds to a path from the root to a node or leaf in \( \sqcap \). Finally, all shortcuts \( p_1p_i \) whose \( CRP(i) \) are \( x \)-monotone, are rectified all together and we follow the main step which is testing whether the paths \( CRP(i) \) encoded in \( \sqcap \) and the corresponding rectified shortcuts are homotopic. See Fig. 6.5 depicting all these steps. Note that \( CRP(5) \) and \( CRP(11) \) are not maintained in \( \sqcap \) since they are not \( x \)-monotone and thus the shortcuts \( p_1p_5 \) and \( p_1p_{11} \) are not rectified while the others get rectified and shown in Fig. 6.5(iv). Next we go into details of each step.

**Rectifying \( P \)** Since \( P \) is non-self-intersecting, the edges of \( P \) and the obstacles of \( S \) induce an aboveness relation which is acyclic and computable in time \( O((n + m) \log(n + m)) \) \([71]\). The aboveness relation indeed defines a partial order. This partial order can be easily extended to a total order. Let \( \text{rank}_p(O) \) be the rank of an object \( O \) (obstacle or edge) in the total order. By letting the \( y \)-coordinate of any point of object \( O \) be \( \text{rank}_p(O) \), the path \( P \) gets rectified, i.e., each edge is represented as a horizontal segment. We join two horizontal segments corresponding to two consecutive edges in \( P \) by a vertical segment in order to maintain the original connectivity—see Fig. 6.5(ii).

**Orthogonal-range queries** Our algorithm relies on a three-sided orthogonal range-query data structure defined over \( m \) point obstacles in order to compute
CRP(i). This data structure is used to find the closest obstacle to any given vertical segment. We use Chazelle’s data structure [27] that preprocesses the obstacles in time $O(m \log m)$ while using $O(m)$ space in order to answer three-sided orthogonal-range queries in $O(\log m)$ time.

**Canonical rectified paths** After rectifying $P$ and constructing the orthogonal-range-query data structure we are ready to compute all CRP(i) inductively, i.e., CRP(i + 1) is obtained from CRP(i). Since separately maintaining CRP(i) may require $O(n^2)$ space, we implicitly store them in a tree $\sqcup$ such that CRP(i) can be extracted by traversing $\sqcup$ from the root to either a leaf or an internal node of $\sqcup$, provided that it is $x$-monotone. Recall that if CRP(i) is not $x$-monotone, it can not be homotopic with its corresponding shortcut $p_{1}p_{i}$.

We maintain two stacks $S_{c}$ and $S_{u}$. Upon processing $p_{i+1}$, stack $S_{c}$ maintains CRP(i) as a sequence of horizontal and vertical segments and stack $S_{u}$ maintains $U$-turns as a sequence of vertical segments at which CRP(i) makes $U$-turns. At the beginning, $S_{u}$ is set to be empty and $S_{c}$ is set to be the first edge of the rectified $P$. We also initialize the tree $\sqcup$: we add new nodes $N(p_{1})$ and $N(p_{2})$ as well as the directed edge $(N(p_{1}),N(p_{2}))$ to the empty tree $\sqcup$ and set $N(p_{1})$ to be the root of $\sqcup$ where $N(p)$ denotes the node with label $p$ placed at the position of point $p$. Now CRP(i + 1) is computed as follows. Consider the horizontal segment of the rectified $P$ ending at $p_{i+1}$. Let the other endpoint of this segment be $q$. We pop $S_{c}$ which is a segment ending at $p_{i}$ and starting at another point, say $r$. If $rp_{i}$ is vertical, we should pop one more segment from $S_{c}$. Without loss of generality, assume $rp_{i}$ is horizontal and $p_{i}$ is right of $r$ — the cases where $p_{i}$ is to left of $r$ can be handled similarly. Note that $p_{i}$ and $q$ are connected in the rectified $P$ by a vertical segment. We distinguish four cases based on the positions of segments $rp_{i}$ and $qp_{i+1}$ and the obstacles as depicted in Fig. 6.6.

(i) The segments $rp_{i}, p_{i}q$ and $qp_{i+1}$ are simply pushed to $S_{c}$. Moreover, if $S_{u}$ is empty, we update $\sqcup$: we add new nodes $N(p_{i+1})$ and $N(q)$, as well as new edges $(N(p_{i}),N(q))$ and $(N(q),N(p_{i+1}))$ to $\sqcup$.

(ii) Let $r'q'$ be the vertical segment touching $s_{k}$ from right where $s_{k}$ is the right most obstacle on the left side of $p_{i}q$. We push $rr', r'q'$ and $q'p_{i+1}$ into $S_{c}$ — we indeed erase $r'p_{i}, p_{i}q$ and $qq'$. We also push $r'q'$ into $S_{u}$ as a new $U$-turn. Since $S_{u}$ is not empty, we do not update $\sqcup$.

(iii) Let $rr'$ be the vertical segment hanging from $r$ which can be obtained by popping $S_{c}$ one more time. We glue $r'r$ and $rq'$ and push $r'q'$ into $S_{c}$ as well as $q'p_{i+1}$. Moreover, if the top of $S_{u}$ is $r'r$, we pop $r'r$ from $S_{u}$. If $S_{u}$ is empty, we update tree $\sqcup$: we insert new nodes $N(r')$ (if it does not exist), $N(q')$ and $N(p_{i+1})$ as well as new directed edges $(N(r'),N(q'))$ and $(N(q'),N(p_{i+1}))$ to tree $\sqcup$. If $N(r')$ already does not exist, we have to split the edge $(N(z),N(w))$ of $\sqcup$ whose embedding in the plane (i.e., segment $zw$) contains point $r'$, into new directed edges $(N(z),N(r'))$ and $(N(r'),N(w))$. If $r'q'$ can be still moved to left which can be tested in constant time by considering the last three segments of $S_{c}$, we pop $q'p_{i+1}$.
6.3. Optimal strongly-homotopic simplification

\begin{figure}[h]
\centering
\begin{tabular}{cccc}
\textbf{(i)} & \textbf{(ii)} & \textbf{(iii)} & \textbf{(iv)} \\
\begin{tikzpicture}
    \node (r) at (0,0) {$r$};
    \node (p_i) at (1,1) {$p_i$};
    \node (p_{i+1}) at (2,1) {$p_{i+1}$};
    \node (q) at (2,0) {$q$};
    \draw (r) -- (p_i) -- (p_{i+1}) -- (q);
\end{tikzpicture} & \begin{tikzpicture}
    \node (r) at (0,0) {$r$};
    \node (p_i) at (1,1) {$p_i$};
    \node (q) at (2,0) {$q$};
    \node (s) at (1,0) {$s$};
    \draw (r) -- (p_i) -- (s);\node (q) at (2,0) {$q$};
    \draw (s) -- (q);
\end{tikzpicture} & \begin{tikzpicture}
    \node (r) at (0,0) {$r$};
    \node (p_i) at (1,1) {$p_i$};
    \node (q) at (2,0) {$q$};
    \node (p_{i+1}) at (3,1) {$p_{i+1}$};
    \node (w) at (1,2) {$w$};
    \node (r') at (2,0) {$r'$};
    \node (s) at (2,1) {$s$};
    \draw (r) -- (p_i) -- (s) -- (q);\node (r') at (2,0) {$r'$};
    \draw (w) -- (s) -- (r') -- (p_{i+1});
\end{tikzpicture} & \begin{tikzpicture}
    \node (r) at (0,0) {$r$};
    \node (p_i) at (1,1) {$p_i$};
    \node (q) at (2,0) {$q$};
    \node (p_{i+1}) at (3,1) {$p_{i+1}$};
    \node (w) at (1,2) {$w$};
    \node (r') at (2,0) {$r'$};
    \node (s) at (2,1) {$s$};
    \draw (r) -- (p_i) -- (s) -- (q);\node (r') at (2,0) {$r'$};
    \draw (w) -- (s) -- (r') -- (p_{i+1});
\end{tikzpicture}
\end{tabular}
\caption{Different cases depending on the positions of $rp_i$ and $qp_{i+1}$ and the obstacles when $p_i$ is right of $r$.}
\end{figure}

and consider it as a new horizontal segment to be inserted and repeat step (iii).

(iv) We push $rr'$ and $r'p_{i+1}$ into $S_c$. Moreover, if $S_u$ is empty, we update $\sqcup$: we add new nodes $N(r')$ and $N(p_{i+1})$ and remove the edge $(N(p_i), N(r))$ and add new directed edges $(N(r), N(r'))$, $(N(r'), N(p_i))$ and $(N(r'), N(p_{i+1}))$.

Since in each of the $n$ iterations, a constant number of edges is added to $\sqcup$, the size of $\sqcup$ is $O(n)$. Note that in some iteration, step (iii) may be processed recursively several times but it is only for the last recurrence that we might have to update $\sqcup$ and moreover the number of such recurrences in total is $O(n)$ as each recurrence can be charged to a horizontal segment of the path and each horizontal segment can be charged at most twice. Therefore, we have the following lemma.

Lemma 6.3. All $x$-monotone CRP($i$) can be encoded in $O(n \log m)$ time into a tree $\sqcup$ of size $O(n)$ where each $x$-monotone CRP($i$) corresponds to a path from the root to a node or leaf in the tree $\sqcup$.

Rectifying shortcuts Let $I$ be the set of indices $i$ such that CRP($i$) is $x$- monotone. In fact, $I$ presents all shortcuts remaining candidates to be homotopic shortcuts after computing all CRP($i$). Consider all shortcuts $p_1p_i$ where $i \in I$ and $m$ obstacles in $S$. They induce an aboveness relation defining a partial order which can be simply extended to a total order. Let rank$_s(O)$ be the rank of an object $O$ (obstacle or shortcut) in this total order. We set the $y$-coordinate of any point of object $O$ to be rank$_s(O)$. This rectifies all shortcuts. Note that there is not any relation between rank$_p(s_i)$ and rank$_s(s_i)$ for obstacle $s_i$ as they are coming from two different total orders—for instance, the $y$-coordinates of the second obstacle from the right in Fig. 6.5(iii) and Fig. 6.5(iv) are different.

Testing homotopy for tree $\sqcup$ and the rectified shortcuts To summarize, we have two planes: (i) the plane $\mathcal{H}_1$ includes the obstacles and tree $\sqcup$ (for instance, Fig. 6.5(iii)), and (ii) the plane $\mathcal{H}_2$ includes the obstacles and the rectified shortcuts corresponding to $x$-monotone CRP($i$) encoded in $\sqcup$ (for instance, Fig. 6.5(iv)). The tree $\sqcup$ has $O(n)$ edges and all its edges embedded in $\mathcal{H}_1$ are either horizontal or vertical—note that $\sqcup$ is not necessary non-self-intersecting and therefore a partial order of paths encoded in $\sqcup$ may not exist. The $y$-coordinates of all horizontal segments and the obstacles in $\mathcal{H}_1$ come from
rank\(_p\) while in \(\mathcal{H}_2\) come from rank\(_s\)—recall that rank\(_p\)(\(s_i\)) is not necessary equal to rank\(_s\)(\(s_i\)) for obstacle \(s_i\). To this end, we recall that \(\mathcal{I}\) is the set of indices \(i\) such that CRP\((i)\) is \(x\)-monotone.

For a horizontal edge \(e\) of \(\sqcup\), let above\((e)\) (below\((e)\)) be the set of obstacles \(s_j\) satisfying (i) rank\(_p\)(\(s_j\)) > rank\(_p\)(\(e\)) (rank\(_p\)(\(s_j\)) < rank\(_p\)(\(e\))) and (ii) the \(x\)-coordinate of \(s_j\) lies between the \(x\)-coordinates of the endpoints of \(e\). As every obstacle above (or below) CRP\((i)\) is above (or below) exactly one edge of all edges appearing in CRP\((i)\), the following lemma is straightforward.

**Lemma 6.4.** For any \(i \in \mathcal{I}\), the rectified shortcut \(p_1p_i\) and CRP\((i)\) are homotopic if and only if for any horizontal edge \(e\) of \(\sqcup\) appearing on CRP\((i)\) and any obstacle \(s_j \in \text{above}(e)\) (\(s_j \in \text{below}(e)\)), the condition (A) rank\(_s\)(\(p_1p_i\)) < rank\(_s\)(\(s_j\)) (rank\(_s\)(\(p_1p_i\)) > rank\(_s\)(\(s_j\))) is satisfied.

This lemma simply implies that any edge \(e\) and any object \(s_j \in \text{above}(e)\) (\(s_j \in \text{below}(e)\)) violating the condition (A) removes \(p_1p_i\) of being a homotopic shortcut and if there is not such a pair \(e\) and \(s_j\), the shortcut \(p_1p_i\) is definitely a homotopic shortcut. However, testing condition (A) for each edge \(e\) and any \(s_j \in \text{above}(e)\) is costly in total. The following useful observation shows among all obstacles either above or below \(e\), at most two obstacles together with \(e\) are necessary to be tested whether they satisfy condition (A).

**Observation 6.2.** It suffices to verify the condition (A) for each horizontal edge \(e\) of \(\sqcup\) and the obstacle \(s_j\) which has the minimum (maximum) rank\(_s\) in \(\text{above}(e)\) (\(\text{below}(e)\)); let \(\min(\text{above}(e)) = \text{rank}_s(s_j)\) (\(\max(\text{below}(e)) = \text{rank}_s(s_j)\)).

Now, the main question is how fast we can perform the verification of condition (A) for any horizontal edge \(e\) of \(\sqcup\) to remove non-homotopic shortcuts as we know above\((e)\) or below\((e)\) may contain many obstacles and \(\mathcal{I}(e)\) may contain several indices where \(\mathcal{I}(e)\) is the set of all indices \(i \in \mathcal{I}\) that \(e\) appears in CRP\((i)\). From now on, we pay our attention to above\((e)\) as below\((e)\) can be handled similarly.

One easy way to compute \(\min(\text{above}(e))\) for any horizontal edge \(e\) of \(\sqcup\) is to build an orthogonal-range query data structure \(\sqcup_{\text{obs}}\) over obstacles in \(\mathcal{H}_1\) which is a two-level tree. For a node \(\nu\) in the second level of \(\sqcup_{\text{obs}}\) we maintain extra information namely the minimum \(\text{rank}_s\) of obstacles lying at the subtree rooted at \(\nu\). Hence \(\min(\text{above}(e))\) can be computed in \(O(\log^2 m)\) time. However, this can be performed faster as we know all edges in advance. We can sweep the horizontal edges and the obstacles from top to bottom and maintain a binary tree over the swept obstacles based on their \(x\)-coordinates. When the sweep line reaches an edge \(e\), above\((e)\) can be seen as the union of \(O(\log m)\) subtrees in the binary tree and consequently \(\min(\text{above}(e))\) can be computed in \(O(\log m)\) time.

In order to collectively verify the condition (A) in which an edge \(e\) is involved, we define a new ordering of elements in \(\mathcal{I}\) such that elements in \(\mathcal{I}(e)\) become consecutive. This ordering is obtained by an in-order traversal of \(\sqcup\)—note that for any \(i \in \mathcal{I}\), there is a node in \(\sqcup\) labeled with \(p_i\). Let \(\sigma(i)\) be the rank of \(i \in \mathcal{I}\) in this ordering. With each \(i \in \mathcal{I}\), we associate the
6.4. Optimal homotopic simplification

point \((\sigma(i), \text{rank}_x(p_ip_j))\) in a new plane \(H\). Each edge \(e\) and its associated \(\min(\text{above}(e))\) define a three-sided orthogonal range in \(H\). It is easy to see every \(i \in I\) whose corresponding point lies in this range, violates the condition \((A)\) and consequently must be removed from \(I\). As these three-sided ranges are available in advance, we can sweep points and ranges in \(H\) from top to bottom and maintain a binary tree over the points based on their \(x\)-coordinates (i.e. their \(\sigma\)-coordinate) like the one described in previous paragraph. Upon processing a three-sided range, we simply remove \(O(\log n)\) subtrees from the binary tree. Therefore the sweeping takes \(O(n \log n)\) time in total. After handling \(\text{below}(e)\) for horizontal edges \(e\) of \(\sqcup\) in a similar manner, any remaining shortcut is a homotopic shortcut. Putting it all this together we get the following result.

**Lemma 6.5.** Let \(P = p_1, \ldots, p_n\) be a non-self-intersecting polygonal path and let \(S\) be a set of \(m\) point obstacles in a plane. All homotopic shortcuts \(p_ip_j\) can be computed in \(O(n(m + n)\log(nm))\) time and \(O(n + m + k)\) space where \(k\) is the number of homotopic shortcuts.

The running time can be improved to \(O(m \log m + nm + n^2 \log(nm))\), if we are allowed to use extra \(O(m \log m)\) space. To get this improvement, at the beginning we compute the aboveness relation of the obstacles and edges of \(P\) as well as the aboveness relation of the obstacles and all shortcuts, and exploit these relations in each iteration where we fix \(i\) to be \(1, \ldots, n\). Moreover, we maintain a persistent binary tree \([36]\) over the obstacles to compute all \(\min(\text{above}(e))\) in total time \(O(n \log m)\) instead of \(O(m \log m + n \log m)\) in each iteration. This persistent binary tree occupies \(O(m \log m)\) space.

6.4 Optimal homotopic simplification

Suppose \(P = p_1, \ldots, p_n\) is a polygonal path in a plane in which there are \(m\) points as obstacles—path \(P\) may have self-intersections. In this section, we present our algorithm that finds the optimal simplification for general paths. Our method exploits the observation that the path \(P\) and every simplification of \(P\) can be seen as strings, namely their canonical sequence. In this setting, the goal will be finding a simplification \(Q\) with the minimum number of links satisfying \(\text{CS}(Q) = \text{CS}(P)\). We use the dynamic-programming paradigm to find such a simplification \(Q\).

We first computes \(G_\varepsilon\) under a given distance function \(F\), and \(\text{CS}(P)\) and \(\text{CS}(p_ip_j)\) for any edge \(p_ip_j\) in \(G_\varepsilon\)—we denote the \(k\)th symbol of \(\text{CS}(\alpha)\) with \(\text{CS}(\alpha)[k]\). When all \(\text{CS}(p_ip_j)\) are available, the CS of a simplification \(Q\) can be simply computed by concatenating the CS of its links and then removing adjacent repeated symbols. The proposed dynamic programming is based on the observation that if \(\text{CS}(Q) = \text{CS}(P)\), any prefix of \(\text{CS}(P)\) must be produced by concatenating the CS of the first \(i\) links of \(Q\) and the first \(j\) symbols of the CS of \((i + 1)\)th link of \(Q\) for some \(i\) and \(j\) and then removing the repeated symbols. Indeed, the last symbol of the prefix of \(\text{CS}(P)\) must match a symbol in \(\text{CS}(Q)\) as we know \(\text{CS}(Q) = \text{CS}(P)\). Therefore, this symbol must exist in
the CS of a link \((i+1)\)th link for some \(i\) of \(Q\) as \(CS(Q)\) are initially obtained by concatenating the CS of its links. If the position of this symbol in the CS of \((i+1)\)th link of \(Q\) is \(j\), therefore the prefix of \(CS(P)\) is the CS of the concatenation of the CS of the first \(i\) links of \(Q\) and the first \(j\) symbols of the CS of \((i+1)\)th link of \(Q\).

We define a sub-simplification \(Q\) of \(P\) to be a subsequence \(q_1, q_2, \ldots, q_r\) of sequence \(p_1, p_2, \ldots, p_n\) where \(q_1\) and \(q_r\) are not necessary equal to \(p_1\) and \(p_n\) respectively. A simplification of \(P\) indeed is a sub-simplification where \(q_1 = p_1\) and \(q_r = p_n\). Now let \(Q = q_1, q_2, \ldots, q_r\) be a sub-simplification starting at point \(p_1\), i.e. \(q_1 = p_1\). We use notation \(seq(Q, b)\) for any \(b \geq 0\) to denote the sequence \(CS(q_1q_2)CS(q_2q_3)\ldots CS(q_{r-2}q_{r-1})\) concatenated with the first \(b\) symbols of \(CS(q_{r-1}q_r)\). We define \(OptHS[i, j, b, l]\) to be the number of links of the optimal sub-simplification \(Q = q_1, \ldots, q_r\) whose last link is the shortcut \(p_ip_j\) (i.e. \(q_r = p_j\) and \(q_{r-1} = p_i\)) and that starts at \(p_1\) (i.e. \(q_1 = p_1\)) and \(CS(seq(Q, b))\) matches the first \(l\) symbols of \(CS(P)\). It is easy to see \(\min_{i=1}^n OptHS[i, n, |CS(p_ip_n)|, |CS(P)|]\) is equal to the number of links of the optimal homotopic simplification where \(|.|\) denotes the size of the sequence.

To fill in \(OptHS\) matrix, we often need to know whether the CS of a sub-simplification is empty or not. For a sub-simplification with the first link \(p_i, p_j\) and the last link \(p_i, p_{j+1}\) and middle links \(\ell_1, \ldots, \ell_r\) we denote the concatenation of the last \(a\) symbols of \(CS(p_ip_{j+1})\) and \(CS(\ell_1)CS(\ell_2)\ldots CS(\ell_r)\) and the first \(c\) symbols of \(CS(p_ip_{j+1})\) by \(seq(i_1, j_1, i_2, j_2, a, c)\). We define \(OptNHS[i_1, j_1, i_2, j_2, a, c]\) to be the number of links of the optimal sub-simplification whose first and last link are \(p_i, p_{j+1}\) and \(p_i, p_{j+1}\) resp. and that \(CS(seq(i_1, j_1, i_2, j_2, a, c))\) is empty. Next we describe how to fill in the matrices \(OptHS\) and \(OptNHS\).

The main idea of computing \(OptHS[i, j, b, l]\) and \(OptNHS[i_1, j_1, i_2, j_2, a, c]\) is as follows. Let \(Q\) be the sub-simplification specifying \(OptHS[i, j, b, l]\). There must be a link \(p_ip_j\) in \(Q\) such that \(CS(p_ip_j)[b'] = CS(P)[l]\) for some \(b'\) and

![Figure 6.7: Let Φ = CS(P) and Q be the sub-simplification specifying OptHS[i, j, b, l]. There must be a link p_i p_j in Q such that CS(p_i p_j)[b'] = CS(P)[l] for some b' and CS(seq(i', j', i, j, |CS(p_i p_j) − b', b)) is empty.](image)
6.4. Optimal homotopic simplification

Figure 6.8: If $Q$ is the sub-simplification specifying $\text{OptNHS}[i_1, j_1, i_2, j_2, a, c]$, there exists a link $p_{i}p_{j'}$ in sub-simplification $Q$ such that $\text{CS}(p_{i}p_{j'})[a'] = \text{CS}(p_{i_1}p_{j_1})[i - a + 1]$ where $a$ is the size of last symbol of $\text{CS}(p_{i_1}p_{j_1})$ and $a'$ is the index of $\mathbf{A}$ in the link $\text{CS}(p_{i}p_{j'})$. Thus, the CS of both gray areas should be empty.

$\text{CS}(\text{seq}(i', j', i, j, |\text{CS}(p_{i}p_{j'})| - b', b))$ is empty—see Fig. 6.7. Thus $\text{OptHS}[i, j, b, l] = \min_{i < j' < i}(\text{OptHS}[i', j', b' - 1, l - 1] + \text{OptNHS}[i', j, i, j, |\text{CS}(p_{i}p_{j'})| - b', b] - 1)$ that min is over all links $p_{i}p_{j'}$ where $\text{CS}(p_{i}p_{j'})[b'] = \text{CS}(P)[l]$. In a similar way, if $Q$ is the sub-simplification specifying $\text{OptNHS}[i_1, j_1, i_2, j_2, a, c]$, there exists a link $p_{i'}p_{j'}$ in sub-simplification $Q$ such that $\text{CS}(p_{i'}p_{j'})[a'] = \text{CS}(p_{i_1}p_{j_1})[|\text{CS}(p_{i_1}p_{j_1})| - a + 1]$, so that $\text{CS}(p_{i'}p_{j'})[a']$ and $\text{CS}(p_{i_1}p_{j_1})[|\text{CS}(p_{i_1}p_{j_1})| - a + 1]$ can eliminate each other if $\text{OptNHS}[i_1, j_1, i', j', i - 1, a' - 1]$ be empty—see Fig. 6.8. Thus, $\text{OptNHS}[i_1, j_1, i_2, j_2, a, c] = \min_{i_1 < i' < i_2}(\text{OptNHS}[i_1, j_1, i', j', a, a' - 1] + \text{OptNHS}[i', j', i_2, j_2, |\text{CS}(p_{i'}p_{j'})| - a', c] - 1)$. We refer readers to Appendix A for the precise computing of $\text{OptHS}[i, j, b, l]$ and $\text{OptNHS}[i_1, j_1, i_2, j_2, a, c]$.

**Reporting**  
$\text{OptHS}$ stores the size of the optimal homotopic sub-simplifications and in order to report the optimal homotopic sub-simplifications, we, of course, must spend more space and time but as we just need the optimal homotopic simplification specifying $\min_{i=1}^{n} \text{OptNHS}[i, n, |\text{CS}(p_{i}p_{n})|, |\text{CS}(P)|]$, we can find it by passing over the matrices once more without asymptotically increasing the time and space complexities.

**Time and space complexity**  
As in $\text{OptNHS}[i_1, j_1, i_2, j_2, a, c]$ it is know that $1 \leq i_1, i_2, j_1, j_2 \leq n$ and $0 \leq a, c \leq m$, and in $\text{OptHS}[i, j, b, l]$ we have $1 \leq i, j \leq n$, and $0 \leq b \leq m$, and $0 \leq l \leq mn$, these two matrices occupy $O(n^4m^2)$ space—note that the size of CS of an edge is at most $m$ and $|\text{CS}(P)|$ can be at most $mn$ as the number of the edges of $P$ is at most $n - 1$. In order to fill one entry of matrices we take min over at most $O(n^2)$ links $p_{i}p_{j'}$ which implies we have to spend $O(n^6m^2)$ time—note that finding the symbol of $\text{CS}(p_{i}p_{j'})$ that matches the $l$th symbol of $P$, takes $O(m)$ time but we can perform it in $O(1)$ time by constructing an auxiliary 2-dimensional array of size $O(n^2m)$ to store the position of each symbol in CS of each link. After computing matrices,
\[
\min_{i=1}^{n} \text{OptHS}[i, n, |\text{CS}(p_i, p_n)|, |\text{CS}(P)|]
\]
can be computed in \(O(n)\) time which asymptotically does not increase the time complexity of our algorithm.

**Theorem 6.3.** Suppose that \(P\) is a polygonal path with size \(n\) in a plane containing \(m\) point obstacles, and that \(T_F(n)\) is the time needed to compute \(G_\varepsilon\) under the distance function \(F\) in the absence of the obstacles. The optimal homotopic simplification can be computed in \(O(n^6m^2) + T_F(n)\) time and \(O(n^4m^2)\) space.

### 6.5 Conclusion

We have proposed the first polynomial-time algorithm to compute the optimal simplification \(Q\) of a polygonal path \(P\) being homotopic to \(P\) with respect to some point obstacles in a plane. We have also presented algorithms to compute the optimal strongly-homotopic simplification of a \(x\)-monotone and a general non-self-intersecting polygonal path \(P\) in \(T_F(n) + O(m \log(nm) + n \log n \log(nm) + k)\) and \(T_F(n) + O(n(m + n) \log(nm))\) time resp. where \(n\) is the size of \(P\) and \(m\) is the number of point obstacles and \(T_F(n)\) is the time needed to compute \(G_\varepsilon\) under the error function \(F\).

Unfortunately our algorithms except the one proposed for \(x\)-monotone paths, do not guarantee to produce a non-self-intersecting simplification which can be an important desire in many areas. We leave computing non-self-intersecting homotopic simplification as a topic for future research.
Chapter 7

Simplifying Massive Contour Maps

The paper *Simplifying Massive Contour Maps* presented in this Chapter has been submitted to Proc. European Symposium on Algorithms (ESA) 2012 [13] as a conference version. During development the draft of this paper was titled “Practical Contour Line Simplification on Massive Data”.


In this chapter we present a slightly modified version where the details presented in the appendices of the conference version have been moved into the appropriate sections.
7.1. Introduction

Simplifying Massive Contour Maps

Lars Arge∗ Lasse Deleuran∗ Thomas Mølhave† Morten Revsbæk∗ Jakob Truelsen‡

Abstract

We present a simple, efficient and practical algorithm for constructing and subsequently simplifying contour maps from massive high-resolution DEMs, under some practically realistic assumptions on the DEM and contours. The algorithm guarantees that the contours in the simplified contour map are homotopic to the unsimplified contours (that is, nested in the same way), non-intersecting, and within a distance of $\varepsilon_{xy}$ and $\varepsilon_z$ of the unsimplified contours in the $xy$ plane and $z$ direction, respectively. We also present the result of experiments with the algorithm using a terrain data set of Denmark with over 12 billion points.

7.1 Introduction

Motivated by a wide range of applications, there is extensive work in many research communities on modeling, analyzing, and visualizing terrain data. A (3D) digital elevation model (DEM) of a terrain is often represented as a planar triangulation $M$ with heights associated with the vertices (also known as a triangulated irregular network or simply a TIN). The $l$-level set of $M$ is the (2D) segments obtained by intersecting $M$ with a horizontal plane at height $l$. A contour is a connected component of a level set, and a contour map $\mathcal{M}$ is the level sets at a set of heights; refer to Figure 7.2. Contour maps are widely used to visualize a terrain primarily because they provide an easy way to understand the topography of a terrain from a simple two-dimensional representation.

Early contour maps were created manually, severely limiting the size and resolution of the created maps. However, with the recent advances in mapping technologies, such as laser based LIDAR technology, billions of $(x, y, z)$ points on a terrain, at sub-meter resolution with very high accuracy ($\sim$10-20 cm), can be acquired in a short period of time and with a relatively low cost. The massive size of the data (DEM) and the contour maps created from them creates problems, since tools for processing and visualising terrain data are often not designed to handle data that is larger than main memory. Another problem

∗MADALGO, Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation, Aarhus University.
†Department of Computer Science, Duke University.
‡SCALGO, Scalable Algorithmics ApS, Denmark.
Chapter 7. Simplifying Massive Contour Maps

Figure 7.1: (a) A (simplified) contour map. (b) The original map obtained from detailed LIDAR based DEM. See also Figure 7.2.

is that contours generated from high-resolution LIDAR data are very detailed, resulting in a large amount of excessively jagged and spurious contours; refer to Figure 7.1 and Figure 7.2. This in turn hinders their primary applications, since it becomes difficult to interpret the maps and gain understanding of the topography of the terrain. Therefore we are interested in simplifying contour maps.

Previous work: The inefficiency of most tools when it comes to processing massive terrain data stems from the fact that the data is too large to fit in main memory and must reside on slow disks. Thus the transfer of data between disk and main memory is often a bottleneck (see e.g. [31]). To alleviate this bottleneck one needs algorithms designed in the I/O-model of computation [9]. In this model, the machine consists of a main memory of size $M$ and an infinite-size disk. A block of $B$ consecutive elements can be transferred between main memory and disk in one I/O operation (or simply I/O). Computation can only take place on elements in main memory, and the complexity of an algorithm is measured in terms of the number of I/Os it performs. Over the last two decades, I/O-efficient algorithms and data structures have been developed for several fundamental problems. See recent surveys [12, 81] for a comprehensive review of I/O-efficient algorithms. Here we mention that scanning and sorting $N$ elements takes $O(\text{Scan}(N)) = O(N/B)$ and $O(\text{Sort}(N)) = O(N/B \log_{M/B}(N/B))$ I/Os, respectively. The problem of computing contours and contour maps have previously been studied in the I/O-model [5, 6]. Most relevant for this paper, Agarwal et al. [5] presents an optimal I/O-efficient algorithm that computes a contour map (along with the nesting of the individual contours) in $O(\text{Sort}(N) + \text{Scan}(|M|))$ I/Os, where $|M|$ is the number of segments in the contour map and $N$ is the number of triangles in the DEM. However, this algorithm is not practical.

While the problem that contour maps generated from high-resolution LIDAR data contain excessively jagged contours, can be alleviated by contour map simplification (while also alleviating some of the scalability problems encountered when processing contour maps), the main issues is of course to guarantee the accuracy of the simplified contour map. There are two fundamental ap-
Figure 7.2: Top: Original contour map $\mathcal{M}$. Bottom: Simplified contour map. A snapshot of contours generated from the Danish island of Als. The contour granularity is $\Delta = 0.5\text{m}$, the $xy$-constraints used was $\varepsilon_{xy} = 4\text{m}$ and the vertical constraint used was $\varepsilon_z = 0.2\text{m}$.

There has been a lot of work on simplifying DEMs; refer e.g. to [23, 46] and the references therein. However, most often the simple approaches do not provide a guarantee on the simplification accuracy, while the more advanced approaches are not I/O-efficient and therefore do not scale to large data sets. Developing I/O-efficient DEM simplification algorithms with simplification guarantees have shown to be a considerable challenge, although an $O(\text{Sort}(N))$ I/O (topological persistence based) algorithm for removing “insignificant” features
from a DEM (resulting in small contours) have recently been developed [7].

Simplifying the contour map $\mathcal{M}$ directly is very similar to simplifying a set of polygons (or polygonal lines) in the plane. Polygonal line simplification is a well studied problem; refer e.g. to [53] for a comprehensive survey. However, there are a number of important differences between contour maps and polygonal line simplification. Most noticeably, simplifying a contour line in the plane using a polygonal line simplification algorithms will, even if it guarantees simplification accuracy in the plane, not provide a $z$-accuracy guarantee. Furthermore, simplifying the contours individually may lead to intersections between the simplified contours. Finally, when simplifying contour maps it’s very important to preserve the relationships between the contours (the homotopic relationship), that is, maintain the nesting of the contours in the map. Note that intersections are automatically avoided and homotopy preserved when simplifying the DEM before computing the contour map.

One polygonal line simplification algorithm that is often favored for its simplicity and high subjective and objective quality on real life data is Douglas and Peucker’s line simplification algorithm [35]. The algorithm simplifies a contour by removing segment endpoints from the contour while ensuring that the distance between the original and the simplified contour is within a distance parameter $\varepsilon_{xy}$ (but it does not guarantee that it removes the optimal number of endpoints under this constraint). Modifications of this algorithm have been developed, that removes self-intersections in the output [77], as well as ensures homotopy relative to a set of obstacles (polygons) [22, 33]. However, these modified algorithms are complicated and/or not I/O-efficient (and do also not consider $z$-accuracy). Note that a common way of handling I/O-efficiency in practice is to divide the contours (or terrain) into memory size chunks and then process these chunks individually. This approach, however, often creates undesirable artifacts at the chunk boundaries.

**Our results:** In this paper we present a simple, efficient and practical algorithm for constructing and subsequently simplifying contour maps from massive high-resolution DEMs, under some practically realistic assumptions on the DEM and contours. The algorithm guarantees that the contours in the simplified contour map are homotopic to the unsimplified contours, non-intersecting, and within a distance of $\varepsilon_{xy}$ of the unsimplified contours in the $xy$ plane. Furthermore, it guarantees that for any point $p$ on a contour in the $l$-level-set of the simplified contour map, the difference between $l$ and the elevation of $p$ in $\mathcal{M}$ (the $z$-error) is less than $\varepsilon_z$. We also present experimental results that show a significant improvement in the quality of the simplified contours along with a major (about 90%) reduction in size.

Overall, our algorithm has three main components. Given the levels $\ell_1, \ldots, \ell_d$, the first component, described in Section 7.3, computes the segments in the contour map $\mathcal{M}$ containing a level-set for each input level. The component also computes level-sets for each levels $\ell_i \pm \varepsilon_z$, $1 \leq i \leq d$. The contours generated from these extra levels will be used to ensure that the $z$-error is bounded by $\varepsilon_z$. We call these contours constraint contours and mark the contours in $\mathcal{M}$ that
7.2 Preliminaries

are not constraint contours. The component also orders the segments around each contour and computes how the contours are nested. It uses $O(\text{Sort}(|M|))$ I/Os under the practically realistic assumptions that each contour, as well as the contour segments intersected by any horizontal line, fit in memory. This is asymptotically slightly worse than the theoretically optimal but complicated algorithm by Agarwal et al. [5]. The second component, described in Section 7.4, computes, for each of the marked contours $P$, the set $\mathcal{P}$ of contours that need to be considered when simplifying $P$. Intuitively, $\mathcal{P}$ contains all (marked as well as constraint) contours that can be reached from $P$ without crossing any other contour of $M$. Although each contour can be involved in many sets $\mathcal{P}$, the entire algorithm uses only $O(\text{Sort}(|M|))$ I/Os. The third component, described in Section 7.5, simplifies each marked contour $P$ within $\mathcal{P}$. This is done using a modified version of Douglas and Peucker’s simplification algorithm [35]. As with Douglas Peucker’s algorithm it guarantees that the simplified contour $P'$ is within distance $\varepsilon_{xy}$ of $\mathcal{P}$, but it also guarantees that $P'$ is homotopic to the original contour $P$ (with respect to $\mathcal{P}$) and that $P'$ does not have self intersections. The existence of the constraint contours in $\mathcal{P}$ together with the homotopy guarantee ensures the $z$-error is within $\varepsilon_z$. Under the practically realistic assumptions that each contour $P$ along with $\mathcal{P}$ fits in internal memory, the algorithm does not use any extra I/Os.

The details on the implementation of our algorithm is given in Section 7.6 along with experimental results on a terrain data set of Denmark with over 12 billion points. We e.g. construct a contour map for the entire country with a granularity of 0.5m (i.e. with a level-set for every 0.5m) and simplify it with $\varepsilon_{xy} = 5m$ and $\varepsilon_z = 0.2m$ in 49 hours. While the original contour have about 4.8 billion segments in about 7 million contours (occupying about 53 gigabytes just for the $xyz$ coordinates) the simplified contour map only has about 9.2% of the original points (and taking up only 5 gigabytes). Besides the obvious decrease in storage requirements, the generated contour map is much more visually pleasing than the original map.

7.2 Preliminaries

Let $M = (V, E, F)$ be a triangulation of $\mathbb{R}^2$, with vertex, edge, and face (triangle) sets $V, E,$ and $F$, respectively. We assume that $V$ contains a vertex $v_{\infty}$, set at infinity, and that each edge $\{u, v_{\infty}\}$ is a ray emanating from $u$. The triangles in $M$ incident to $v_{\infty}$ are unbounded. Let $h : \mathbb{R}^2 \rightarrow \mathbb{R}$ be a continuous height function with the property that the restriction of $h$ to each triangle of $M$ is a linear map. Given $M$ and $h$, the graph of $h$ is a terrain. It is represented as an $xy$-monotone triangulated surface whose triangulation is induced by $M$. That is, vertices, edges, and faces of $M$ are in one-to-one correspondence with those of $M$ and with a slight abuse of terminology we refer to $V, E,$ and $F$, as vertices, edges, and triangles of both the terrain $M$ and $t$. For convenience we assume that $h(u) \neq h(v)$ for all vertices $u \neq v$, and that $h(v_{\infty}) = -\infty$. Within each bounded triangle $f \in F$, $h$ is uniquely determined as the linear interpolation of the height of the vertices of $f$. This is not the case for an unbounded face $f$.
Chapter 7. Simplifying Massive Contour Maps

since interpolation using $h(v_{\infty}) = -\infty$ is undefined; in which case to determine $h$ on $f$ an extra parameter, such as the height of a point in $f$, is needed.

A minimum/maximum of $M$ is a vertex whose neighbor vertices all have higher/lower elevation. A saddle of $M$ is a vertex $v$ with four vertices $w_1, w_2, w_3, w_4$ among its neighbors that satisfy $\max(h(w_1), h(v_3)) < h(v) < \min(h(w_2), h(v_4))$ and appear in the order $w_1, w_2, w_3, w_4$ clockwise around $v$. A critical vertex of $M$ is either a minimum, maximum or saddle vertex.

Paths and polygons: Let $p_1, \ldots, p_n$, be a sequence of $n > 1$ points in $\mathbb{R}^2$. The path $Q$ defined by these points is the set of line segments defined by pairs of consecutive points in the sequence. The points $p_1, \ldots, p_n$ are called the vertices of $Q$. A simple path is a path where only consecutive segments intersect, and only at the endpoints. Given integers $1 \leq i < j \leq n$, the sub-path $Q_{ij}$ is the path defined by the vertices $p_i, p_{i+1}, \ldots, p_j$. We abuse notation slightly by using $Q$ to denote both the sequence of vertices, and the path itself. We define the size of $Q$ as its number of segments, i.e. $|Q| = n - 1$. A path $Q'$ is a simplification of a path $Q$ if $Q' \subseteq Q$ and the vertices of $Q'$ appear in the same order as in $Q$.

A polygon (simple polygon) is a path (simple path) $P$ where $p_1 = p_n$. A simple polygon $P$ partitions $\mathbb{R}^2 \setminus P$ into two open sets — a bounded one called inside of $P$ and denoted by $P^i$, and an unbounded one called outside of $P$ and denoted by $P^o$. We define a family of polygons to be a set of non-intersecting and vertex-disjoint simple polygons. Consider two simple polygons $P_1$ and $P_2$ in a family of polygons $A$. $P_1$ and $P_2$ are called neighbors if no other polygon $P \in A$ separates them, i.e., one of $P_1$ and $P_2$ is contained in $P^i$ and the other in $P^o$. If $P_1$ is neighbor to $P_2$ and $P_1 \subset P_2$, then $P_1$ is called a child of $P_2$, and $P_2$ is called the parent of $P_1$, we will refer to the parent of $P$ as $\hat{P}$. The topology of a family of polygons $M$ describes how the polygons are nested i.e. the parent/child relationship between polygons.

Given a polygon $P$ in $A$, the polygonal domain of $P$, denoted $\mathcal{P}$, consists of the neighbors $P_1 \ldots P_k$ of $P$ in $A$, see Figure 7.3. We define the size of $\mathcal{P}$ to be $|\mathcal{P}| = |P| + \sum_i |P_i|$.

Intuitively, two paths $Q$ and $Q'$ are homotopic with regards to a polygonal domain $\mathcal{P}$ if one can be continuously transformed into the other without intersecting any of the polygons of $\mathcal{P}$. We refer to Figure 7.3 for an illustration.

![Figure 7.3: (a) Polygonal domain $\mathcal{P}$ (solid lines) of $P$ (dashed lines). (b) The valid simplification $P'$ of $P$ within $\mathcal{P}$ and a polygon $Q$ not homotopic to $P$.](image)
exists a continuous mapping $f$ of this. More precisely, $Q$ and $Q'$ are homotopic with regards to $P$ if there exists a continuous mapping $f : [0,1] \times [0,1] \to \mathbb{R}^2 \setminus P$ morphing $Q$ to $Q'$, see Figure 7.3. In other words if $Q(\cdot)$ and $Q'(\cdot)$ represents the continuous curves (parametrized in $[0,1]$) defined by the paths $Q$ and $Q'$ then the following should hold for $f$:

$$f(0,t) = Q(t) \text{ and } f(1,t) = Q'(t), \text{ for } t \in [0,1],$$

$$f(\alpha,0) = Q(0) = Q'(0) \text{ and } f(\alpha,1) = Q(1) = Q'(1), \text{ for } \alpha \in [0,1].$$

Path $Q'$ is strongly homotopic to $Q$ if $Q'$ is a simplification of $Q$ and if every segment $q'_i q'_{i+1}$ in $Q'$ is homotopic to the corresponding sub-path $Q_{k,i}$ where $q'_i = q_k$ and $q'_j = q_l$. It follows that $Q$ and $Q'$ are also homotopic, but the reverse implication does not necessarily hold.

Given two indices $1 \leq i,j \leq n$ we define the distance $d(p,i,j)$ between any point $p \in \mathbb{R}^2$ and line segment $p_i p_j$ as the distance from $p$ perpendicular to the line defined by $p_i p_j$. We define the error $\varepsilon(i,j)$ to be the maximum distance between the vertices of $P_j$ and the line segment $p_i p_j$, i.e. $\varepsilon(i,j) = \max \{d(i,j, p_i), d(i,j, p_{i+1}), \ldots, d(i,j, p_k)\}$. Let $P'$ be a simplification of $P$.

A simplification threshold $\varepsilon$, we say that $P'$ is a valid simplification of $P$ if it is a simple polygon homotopic to $P$ in $P$ and $\varepsilon(i,j) < \varepsilon$ for any segment $p_i p_j$ of $P'$.

**Contours and contour maps:** For a given terrain $M$ and a level $\ell \in \mathbb{R}$, the $\ell$-level set of $M$, denoted by $M_\ell$, is defined as $h^{-1}(\ell) = \{x \in \mathbb{R}^2 \mid h(x) = \ell\}$. A contour of a terrain $M$ is a connected component of a level set of $M$.

We refer to a level $\ell$ as critical if $M_\ell$ contains a critical vertex and regular if $M_\ell$ contains no critical vertex. Since $h$ is linear across the triangles of $M$, a contour at a regular level $\ell$ is a simple polygon with vertices corresponding to intersections between $z_\ell$ and the edges of $M$, $z_\ell$ is the plane $z = \ell$. At critical levels a contour may be a single point (an extremum), or the contour might correspond to a polygon with self-intersections at saddle vertices of $M$.

Given a list of levels $\ell_1 < \ldots < \ell_d \in \mathbb{R}$, the contour map $\mathcal{M}$ of $M$ is defined as the union of the level-sets $M_{\ell_1}, \ldots, M_{\ell_d}$. For simplicity, we assume that no vertex of $M$ has height $\ell_1, \ldots, \ell_d$ and we assume that $M$ is given such that $M_{\ell_1}$ consists of only a single boundary contour $U$ (e.g. $\ell_1 < \min_{v \in V - \{v_\infty\}}(h(v))$).

This implies that all levels are regular and that the collection of contours in the contour map form a family of polygons and that each polygon in the family, except $U$, has a parent. It allows us to represent the topology of $\mathcal{M}$ as a tree $T = (V,E)$ where the vertices $V$ is the family of polygons and where $E$ contains an edge from each polygon $P \neq U$ to its parent polygon $\hat{P}$. The root of $T$ is $U$. We will refer to $T$ as the topology tree of $\mathcal{M}$.

### 7.3 Building the Contour Map

In this section we describe our practical and I/O-efficient algorithm for constructing the contour map $\mathcal{M}$ of the terrain $M$ and the topology tree $T$ of $\mathcal{M}$, given a list of regular levels $\ell_1 < \ldots < \ell_d \in \mathbb{R}$. We will represent $\mathcal{M}$ as
Chapter 7. Simplifying Massive Contour Maps

Figure 7.4: The possible events that occur when the sweep-line $y_\mu$ encounters vertex $v$ on polygon $P$. The vertex $u = \text{pred}(v)$ precedes $v$ in the clockwise ordering of the vertices of $P$, and $w = \text{succ}(v)$ succeeds it. The solid-colored region denotes $P^i$ i.e. the interior of $P$. The edges intersecting $y_\mu$ are stored in $S$ and are tagged with their parent polygon.

a sequence of line segments such that the clockwise ordered segments in each polygon $P$ of $\mathcal{M}$ appear consecutively in the sequence, and $\mathcal{T}$ by a sequence of edges $(P_2, P_1)$ indicating that $P_2$ is the parent of $P_1$; all segments in $\mathcal{M}$ of polygon $P$ will be augmented with (a label for) $P$ and the BFS number of $P$ in $\mathcal{T}$. We will use that the segments in any polygon $P$ in $\mathcal{M}$, as well as the segments in $\mathcal{M}$ intersecting any horizontal line, fit in memory.

Computing contour map $\mathcal{M}$: To construct the line segments in $\mathcal{M}$, we first scan over all the triangles of $\mathcal{M}$. For each triangle $f$ we consider each level $\ell_i$ within the elevation range of the three vertices of $f$ and construct a line segment corresponding to the intersection of $z_{\ell_i}$ and $f$. To augment each segment with a polygon label, we then view the edges as defining a planar graph such that each polygon is a maximally connected component in this graph. We find these connected components practically I/O-efficiently using an algorithm by Arge et al. [14], and then we use the connected component labels assigned to the segments by this algorithm as the polygon label. Next we sort the segments by their label. Then, since the segments of any one polygon fit in memory, we can in a simple scan load the segments of each polygon $P$ into memory in turn and sort them in clock-wise order around the boundary of $P$.

Computing the topology tree $\mathcal{T}$ of $\mathcal{M}$: We use a plane-sweep algorithm to construct the edges of $\mathcal{T}$ from the sorted line segments in $\mathcal{M}$. During the algorithm we will also compute the BFS number of each polygon $P$ in $\mathcal{T}$. After the algorithm it is easy to augment every segment in $\mathcal{M}$ with the BFS number of the polygon $P$ it belongs to in a simple sorting and scanning step.

For a given $\mu \in \mathbb{R}$, let $y_\mu$ be the horizontal line through $\mu$. Starting at $\mu = y_\infty$, our algorithm sweeps the line $y_\mu$ though $\mathcal{M}$ in the negative $y$-direction by scanning a list $L$ of the vertices in $\mathcal{M}$ (except for those in $\mathcal{U}$) sorted by increasing $y$-coordinate. Assume that a vertex $v$ in $L$ is augmented with the (label) of the polygon $P$ to which it belongs, and its immediate predecessor
pred(v) and successor succ(v) in the cyclic clockwise order of P. We can easily construct L from the segments in M in a simple scan and sort step. During the sweep we maintain the set of line segments that intersect yμ in a search tree S, ordered by the x-coordinate of the intersection between the segment and yμ. Each line segment uv in S is augmented with pred(u), succ(u), pred(v) and succ(v), as well as with the parent polygon of the polygon P in M containing uv. Additionally, we maintain a dictionary D over the set of (labels of) polygons intersection yμ. The dictionary D contains the set of tuples (P, s, P̂, o), where P ∈ M is any polygon for which there is currently a segment in S and P̂ is the parent of P in M. The integer s denotes how many segments of S belong to P, and o the BFS number of P in T. Note that since P is a simple polygon, s is always an even positive integer. Initially D contains just one entry corresponding to the boundary polygon U, without a parent and with BFS number 0.

Now for simplicity, assume that no two vertices of L have the same y-coordinate. Whenever the sweep line yμ encounters a new vertex v we update S and D. With v in L we obtain the (label of the) polygon P containing v along with u = pred(v) and w = succ(v). Let low(v) = \arg \min_{p \in \{u,w\}} (y(p)) and high(v) = \arg \max_{p \in \{u,w\}} (y(p)) be the lower and highest point of u and v, here y(p) denotes the y-coordinate of point p. Based on the y-coordinate of u and w there are now three different case. (i) u and w lie on different sides of yμ, i.e. y(high(v)) > μ > y(low(v)), corresponding to case (R) and (L) in Figure 7.4. In this case we can simply delete the old segment high(v)v from S and insert the new segment low(v)v. The parent of the new segment is the same as that of the old segment. (ii) both u and w are above yμ, i.e. y(low(v)) > μ, corresponding to case (E1) and (E2) in Figure 7.4. In this case we simply delete the two segments uv and wv from S. Unless v is on the boundary U, we locate and delete the entry for P from D. Let (P, s, P̂, o) be this just-deleted entry. If s is larger than 2, there are still segments from P intersecting yμ and we insert (P, s − 2, P̂, o) into D. However, if s = 2 (this can only happen in case (E1)), v must be the bottom-most vertex of P and we do not insert anything into D. At this point we output (P, P̂) as an edge of the topology tree T, along with the BFS number of P. (iii) both u and w are below yμ, i.e. y(high(v)) < μ, corresponding to case (B1) and (B2) in Figure 7.4. In this case we need to insert two new segments vu and vw into S. If v is on U we are done, since U has no parent and D was initialized with an entry for U. Otherwise we update D, but to do that we need to find the parent polygon of P. This is found by a look-up in D. If P is not in D, v must be the top-most vertex of P. Since P is a simple polygon this implies that we are in situation (B1), in other words P′ is beneath yμ. In this case we find the predecessor segment ab of vu in S, this predecessor exists because v is not in U. We observe that the polygon P′ containing a is either the parent of P, or a sibling of P. By looking at the relative order of a and b and their predecessor and successors stored in S we can easily determine which is which by a similar case analysis as the one depicted in Figure 7.4. If it is a sibling we look up P′ in D and get a new polygon P′′ and breadth-first rank o′′ corresponding to the parent P′ and we insert (P, 2, P′′, o′′ + 1) into D,
otherwise we simply insert \((P, 2, P', o' + 1)\) where \(o'\) is the breadth-first rank of the polygon associated with \(P'\) returned by \(D\) initially.

**Analysis:** The algorithm for computing the contour map \(M\) uses \(O(\text{Sort}(|M|))\) I/Os: First it scans the input triangles to produce the segments of \(M\) and invokes the practically efficient connected component algorithm of Arge et al. [14] that uses \(O(\text{Sort}(|M|))\) I/Os under the assumption that the segments in \(M\) intersecting any horizontal line fit in memory. Then it sorts the labeled segments using another \(O(\text{Sort}(|M|))\) I/Os. Finally, it scans the segments to sort each polygon in internal memory, utilizing that the segments in any polygon \(P\) in \(M\) fits in memory. Also the algorithm for computing the topology tree \(T\) uses \(O(\text{Sort}(|M|))\): After scanning the segments of \(M\) to produce \(L\), it performs one sort and one scan of \(L\), utilizing the assumption that \(S\) fits in memory. Augmenting each segment in \(M\) with the BFS number of the polygon \(P\) that it belongs to, can also be performed in \(O(\text{Sort}(|M|))\) in a simple way.

### 7.4 Simplifying Families of Polygons

This section describes our practical and I/O-efficient algorithm for simplifying a set of marked polygons in a family of polygons given an algorithm for simplifying a single polygon \(P\) within its polygonal domain \(\mathcal{P}\). Thus in essence the problem consist of computing \(\mathcal{P}\) for each marked polygon \(P\). We assume the family of polygons is given by a contour map \(M\) represented by a sequence of line segments such that the clockwise ordered segments in each polygon \(P\) of \(M\) appear consecutively in the sequence, and a topology tree \(T\) given as a sequence of edges \((P, \hat{P})\) indicating that \(\hat{P}\) is the parent of \(P\); all segments in \(M\) of polygon \(P\) are augmented with (a label for) \(P\) and the BFS number of \(P\) in \(T\). To compute \(\mathcal{P}\) for every \(P\) we need to retrieve the neighbors of \(P\) in \(M\). These are exactly the parent, siblings and children of \(P\) in \(T\). Once \(\mathcal{P}\) and the simplification \(P'\) of \(P\) has been computed we need to update \(M\) with \(P'\). We describe an I/O-efficient simplification algorithm that allows for retrieving polygonal domains and updating polygons without spending a constant number of I/Os for each polygon. The algorithm simplifies the polygons across different BFS levels of \(T\) in order of increasing level, starting from the root. Within a given level the polygons are simplified in the same order as their parents were simplified. Polygons with the same parent can be simplified in arbitrary (label) order. The following paragraphs describe the steps of our algorithm. In the first step, we reorder the polygons in \(M\) such that they appear in the order they will be simplified. In the second step, we describe how to simplify \(M\).

**Reordering:** We compute the simplification rank of every polygon \(P\) i.e. the rank of \(P\) in the simplification order described above. The simplification rank for the root of \(T\) is 0. To compute ranks for the remaining polygons of \(T\), we sort the edges \((P, \hat{P})\) of \(T\) in order of increasing BFS level of \(P\). By scanning through the sorted list of polygons we assign simplification ranks to vertices one layer at a time. When processing a given layer we have already determined
Simplifying Families of Polygons

Figure 7.5: Left: The contour with the dark gray area inside is to be simplified. The light gray parent, gray siblings and children make up the boundaries. Right: The topology tree \( T \).

the ranks of the previous layer and can therefore order the vertices according to the ranks of their parents.

**Simplifying:** Assume that the polygons in \( \mathcal{M} \) appear in order of increasing simplification rank. Consider the sibling polygons \( P_1, P_2 \ldots P_k \) in \( \mathcal{M} \) all sharing the same parent \( P \) in \( T \). The polygonal domains of these sibling polygons all share the polygons \( P, P_1, P_2 \ldots P_k \). We will refer to these shared polygons as the open polygonal domain of \( P \) and denote them \( \mathcal{P}_{\text{open}}(P) \). It is easily seen that \( \mathcal{P} \) for \( P_i \) where \( i = 1 \ldots k \) is equal to \( \mathcal{P}_{\text{open}}(P) \) together with the children of \( P_i \).

When traversing the polygons of \( \mathcal{M} \) in the order specified by their simplification ranks, we will refer to \( P \) as an unfinished polygon if we have visited \( P \) but not yet visited all the children of \( P \). During the traversal we will maintain a queue \( Q \) containing an open polygonal domain for every unfinished polygon. The algorithm handles each polygon \( P \) as follows; if \( P \) is the root of \( T \) then \( \mathcal{P} \) simply corresponds to the children of \( P \) which are at the front of \( \mathcal{M} \). Given \( \mathcal{P} \) we invoke the polygon simplification algorithm to get \( P' \). Finally, we put \( \mathcal{P}_{\text{open}}(P') \) at the back of \( Q \). If \( P \) is not the root of \( T \), it will be contained in the open polygonal domain \( \mathcal{P}_{\text{open}}(\hat{P}) \). Since \( \hat{P} \) is the unfinished polygon with lowest simplification rank, \( \mathcal{P}_{\text{open}}(\hat{P}) \) will be the front element of \( Q \). If \( P \) is the first among its siblings to be visited, we retrieve \( \mathcal{P}_{\text{open}}(P) \) from \( Q \), otherwise it has already been retrieved and is available in memory. To get \( \mathcal{P} \), we then retrieve the children of \( P \) from \( \mathcal{M} \) and combine them with \( \mathcal{P}_{\text{open}}(P) \) (if \( P \) is a leaf then \( \mathcal{P} = \mathcal{P}_{\text{open}}(\hat{P}) \)). Finally, we invoke the polygon simplification algorithm on \( P \) and \( \mathcal{P} \) to get \( P' \) and put the open polygonal domain of \( P' \) at the back of \( Q \).

**Analysis:** The computation of simplification ranks requires sorting and scanning of the vertices in \( T \) and can therefore be done using \( O(\text{Sort}(|T|)) \) I/Os. Once the simplification rank is known for every polygon, it is easy to reorder the polygons in \( \mathcal{M} \) using sorting and scanning of \( \mathcal{M} \) i.e. using \( O(\text{Sort}(|\mathcal{M}|)) \) I/Os. The simplification algorithm performs a scan of \( \mathcal{M} \) which requires \( O(\text{Scan}(|\mathcal{M}|)) \).
I/Os. During the scan the open polygonal domain of every polygon internal to $T$ is inserted and removed from $Q$ once which again requires $O(\text{Scan}(|M|))$. So in total the reordering and simplification steps can be done using $O(\text{Sort}(|M|))$ I/Os.

### 7.5 Internal Simplification Algorithm

In this section we present our simplification algorithm which is given a single polygon $P$ to simplify along with its polygonal domain, $P$. The output $P'$ is a valid simplification of $P$ and is thus homotopic to $P$ in $P$.

**Simplifying $P$:** We can describe how to compute a valid simplification $P'$ of $P$ using a modified version of Douglas-Peucker’s simplification algorithm [35]. The modifications ensure that the output simplification $P'$ is homotopic to $P$. Although the algorithm is initially invoked on polygon $P$, we describe it in terms of a path $Q$.

The algorithm is given indices $i$ and $j$, $1 \leq i < j \leq n$ of vertices in some path $Q$. We maintain a list of vertices which at the end will be the output simplification $Q^*$, initially $Q^*$ consists of $p_1$ and $p_n$. We use a recursive procedure that works on a sub-path $Q_{ij}$. The procedure has two steps and is invoked initially using $i = 0$ and $j = n$ as the parameters. In the first step (1) we check if $\varepsilon(i,j) < \varepsilon$, if that is the case we check whether segment $p_ip_j$ is homotopic to $Q_{ij}$. If both checks succeed we do not need to further simplify $P_{ij}$ and we are done. Otherwise we proceed to step (2) and find the vertex internal to $Q_{ij}$ that maximizes the error $\varepsilon(i,j)$, i.e. $k = \arg \max_{i<k<j}(d(i,j,p_k))$. We add vertex the $p_k$ to $Q^*$ and recurse on $(i,k)$ and $(k,j)$. Note that at the first step when $Q_{1n} = Q$, $\varepsilon(1,n)$ is not necessarily well-defined since the segment $p_1p_n$ is degenerate if $Q$ is a closed path (i.e. a polygon) — in this case we skip the test in step (1) and go directly to step (2). The algorithm terminates since any segment is homotopic to itself with error 0 and $0 < j-i$ strictly decreases in the recursion. The output path $Q^*$ is homotopic to $Q$ since by construction every segment $p_ip_j$ of $Q^*$ is homotopic to $Q_{ij}$. This implies that $Q^*$ is strongly homotopic to $Q'$ which again implies that $Q^*$ and $Q$ are homotopic.

Note when the input to the algorithm above is the polygon $P$ the output is also a polygon $P^*$. Even though $P^*$ is homotopic to $P$, it is not necessarily a valid simplification since $P^*$ may not be a simple polygon, see Figure 7.6(b). Thus after the conclusion of the algorithm above we turn $P^*$ into a simple polygon $P'$ which is the final valid simplification of $P$ output from the simplification algorithm. In the remainder of this section we will describe how this is done, and also describe how we perform the segment-sub-path homotopy check in step (1) of the algorithm above.

**Removing Self intersections:** Here we describe how we turn $P^*$ into a valid simplification $P'$. There are various ways of dealing with this problem [77] but we use a simple algorithm that is easy to implement and works well in practice. First we use a standard line-segment intersection finding algorithm
7.5. Internal Simplification Algorithm

to report the list of segments \( s_1, \ldots, s_m \) of \( P^* \) involved in intersections, see Figure 7.6(b). We note that our implementation finds both normal intersections between segments as well as intersections where lines cross using overlapping points.

If \( k = 0 \), \( P^* \) is already a valid simplification, and we are done. Otherwise we invoke our simplification algorithm described in the beginning of this section on each segment \( s_t = p_ip_j \) and force it to go through at least one step of the recursion, i.e. we find the vertex \( p' \neq p_i, p_j \) among the vertices of \( P_{ij} \) maximizing \( \varepsilon(i,j) \) and insert this point into \( P^* \) and recurse as needed on the two new segments \( p_ip' \) and \( p'p_j \) that may no longer be homotopic to their corresponding sub-paths. Once this has been done for all segments \( s_1, \ldots, s_k \), we re-run the line-segment intersection algorithm and repeat the process if new intersections are found. This process stops when no two segments of \( P^* \) intersect.

This algorithm has a poor worst case performance of \( O(|P^*|^2 \log(|P^*|)) \) because every segment might cause an intersection and thus an iteration of the sweep line algorithm. However we expect it to work well in practice since contours are relatively well behaved.

7.5.1 Checking segment-sub-path homotopy:

We use the ideas of Cabello et al. [22] but arrive at a simpler algorithm by taking advantage of the fact that since \( P \) is given as a set of ordered simple polygons, we can efficiently and easily compute its trapezoidal decomposition \( D \) using a simple sweep line algorithm on the segments of \( P \). By having access to \( D \), navigating the space around \( P \) becomes easier and this lets us tweak the ideas of Cabello et al. and arrive at a simple and very practical algorithm.

**The trapezoidal decomposition of \( P \):** For any point \( p \in P \), we imagine a ray \( \sigma^+ \) in the positive y-direction starting at \( p \). This ray may intersect multiple segments from \( P \), and we define \( p^+ \) to be the segment whose intersection point with \( \sigma^+ \) is closest to \( p \) along \( \sigma^+ \). Similarly we imagine a ray \( \sigma^- \) in the negative y-direction and define \( p^- \) to be the closest intersection point, if such a point exists. The *trapezoidal decomposition* \( D \) of \( P \) consists of the segments \( pp^+ \) and \( pp^- \) for all points \( p \in P \) where these are defined, as well as the segments from \( P \). These segments can, however, be split up: If a segment \( s = uv \) from \( P \) is hit by a ray \( \sigma^+ \) from point \( p \) it is split into two sub-segments \( up^+ \) and \( p^+v \), and similar for \( p^- \) points. If \( s \) is hit by multiple such rays it is split into multiple segments. We note that \( D \) is a planar graph with \( O(|P|) \) edges and faces and that every face of \( D \) is a trapezoid defined by two upper segments from \( D \) and two vertical segments. Note that in the case of degeneracies a face may become triangular.

**Computing \( D \):** There already exist several algorithms for constructing trapezoidal decompositions of a set of line segments in the plane (e.g. [45, 60, 78]), however since we only need to add vertical segments to an already existing polygonal domain, constructing \( D \) is not hard and can be done by a sweep-line algorithm scanning the segments of \( P \) in the positive x-direction. Let \( \mu \) be the
current $x$ position and $x_\mu$ the vertical line $x = \mu$. We maintain a binary search tree $S$ containing the segments currently intersecting $x_\mu$, this can be done using the same idea as in Section 7.3. When $x_\mu$ encounters some segment endpoint $v$ we delete/insert $vu$ and $vw$, where $u = \text{succ}(u)$ and $w = \text{pred}(u)$ into $S$ as appropriate (similar to Figure 7.4 rotated 90 degrees). Furthermore, we can find $p^\uparrow$ by finding the predecessor of $p$ in $S$, and $p^\downarrow$ by finding the successor of $p$. The complexity of this algorithm is $O(|D| \log |D|)$ and it uses no I/O’s since it assumes that $D$ is given in memory. We store $D$ in such a way that we can easily traverse the edges and faces of this planar graph in time linear in the length of the traversed path. This can easily be achieved using a number of standard data-structures.

**The homotopy-check:** We define the *trapezoidal sequence* of a path, $t(Q)$, to be the sequence of trapezoids traversed by $Q$, sorted in the order of traversal. If two paths have the same trapezoidal sequence they are homotopic, this can be proven by the argument similar to the ones in [22,55]. It can be shown that if $t(Q)$ contains the subsequence $tt't$ for trapezoids $t, t' \in D$ then this subsequence can be replaced by $t$ without affecting $Q$’s homotopic relationship to any other path, we call this a *contraction* of $t(Q)$. By repeatedly performing contractions on the sequence $t(Q)$ until no more contractions are possible we get a new sequence $t_c(Q)$. We call this sequence the *canonical trapezoidal sequence* of $Q$. $Q$ and $Q'$ are homotopic if and only if $t_c(Q) = t_c(Q')$ [22,55]. This suggests an easy algorithm for checking if two paths are homotopic: Simply compute and compare their canonical trapezoidal sequences. Note however that the size of $t(Q')$ and $t_c(Q')$ is not necessarily linear in the size of the decomposition $D$. In our case, we are interested in checking an instance of the strong homotopy condition, i.e. if a candidate segment $s = p_ip_j$ is homotopic to $Q_{i,j}$. Since $s$ is a line segment $t_c(s) = t(s)$ and we check directly if $s$ traverses the set of trapezoids in $t_c(Q_{i,j})$ without first computing $t(Q_{i,j})$. This is easily done by traversing $D$ for $s$ and $P_{i,j}$ simultaneously, which can be done in $O(|t(Q_{i,j})|)$ time.
7.6 Experiments

In this section we describe the experiments performed to verify that our algorithm for computing and simplifying a contour map $M$ performs well in practice.

**Implementation:** We implemented our algorithm using the TPIE environment for efficient implementation of I/O-efficient algorithms, while taking care to handle all degeneracies (contours with height equal to vertices of $M$, contour points with the same $x$– or $y$-coordinate, and the existence of a single boundary contour). The implementation takes an input TIN $M$ along with parameters $\varepsilon_{xy}$ and $\varepsilon_z$, and $\Delta$, and produces a simplified contour map $M$ with equi-spaced contours at distance $\Delta$.

We implemented one major internal improvement compared to the described algorithm, which results in a speed-up of an order of magnitude: As described in Section 7.5 we simplify a polygon $P$ by constructing a trapezoidal decomposition of its entire polygonal domain $\mathcal{P}$. In practice, some polygons are very large and have many relatively small children. In this case, even though a child polygon is small, its polygonal domain (and therefore also its trapezoidal decompositions) will include the large parent contour together with its siblings. However, it is easy to realize that for a polygon $P$ it is only the subset of $\mathcal{P}$ within the bounding box of $P$ that can constrain its simplification. Line segments outside the bounding box can be ignored when constructing the trapezoidal decomposition. We incorporate this observation into our implementation by building an internal memory R-tree [52] for each polygon $P$ in the open polygonal domain $\mathcal{P}_{\text{open}}(\hat{P})$. These R-trees are constructed when loading large open polygonal domain into memory. To retrieve the bounding box of a given polygon $P$ in $\mathcal{P}_{\text{open}}(\hat{P})$, we query the R-trees of its siblings and its parent, and retrieve the children of $P$ as previously.

**Data and Setup:** All our experiments were performed on a machine with an 8-core Intel Xenon CPU running at 3.2GHz and 12GB of RAM out of which 10GB were available for our experiments. For our experiments we used a terrain model for the entire country of Denmark constructed from detailed LIDAR measurements (the data was generously provided to us by COWI A/S). The model is a $2m$ grid model giving the terrain height for every $2m \times 2m$ in the entire country, which amounts to roughly 12.4 billion grid cells. From this grid model we built a TIN by triangulating the grid cell center points. Before triangulating and performing our experiments, we used the concept of topological persistence [37] to compute the depth of depressions in the grid.

**Topological persistence:** Topological persistence [37] on a triangulation matches each minimum vertex $v$ to a higher saddle vertex $w$ and assigns a persistence value $\pi(v)$ to $v$; where $\pi(v)$ is defined as the height difference between $w$ and $v$. For a given pair of minimum and saddle vertices, we can define the corresponding depression of the terrain as the maximal connected component
containing $v$ and vertices with heights between $h(v)$ and $h(w)$. Intuitively, the persistence value of a depression can be seen as the depth of the depression and a depression will correspond to a contour in all $\ell$-level sets with $h(v) \leq \ell \leq h(w)$. The definitions above also apply to peaks in the terrain by simply inverting the terrain heights.

![Contour Maps](image)

Figure 7.7: A small sample of the non-simplified contour map for an area around the Danish island Læsø. To the left, no depressions or peaks have been removed. To the right, depressions and peaks with persistence value less than $0.5m$ have been removed.

When generating a contour map of level-sets for every $\Delta$ meter it is purely coincidental whether a depression/peak with persistence value less than $\Delta$ will result in a contour or not. Contours corresponding to depressions/peaks that have persistence value less than $\Delta$, often appear as small spurious contours spread around the contour map. Refer to Figure 7.7. For all our experiments we have used an I/O-efficient algorithm given by [7] to compute the persistence value of all depression/peaks and subsequently raised/lowered depressions/peaks with persistence value less than $\Delta$.

**Experimental Results:** In all our experiments we generate contour maps with $\Delta = 0.5m$ since this seems to be widely used in detailed topographic maps. Furthermore, since the LIDAR measurements on which the terrain model of Denmark is based have a height accuracy of roughly $0.2m$, we used $\varepsilon_z = 0.2m$ in the experiments. In order to determine a good value of $\varepsilon_{xy}$ we first performed experiments on a subset of the Denmark dataset, consisting of 844,554,140 grid cells and covering the island of Funen. Below we first describe the results of these experiments and then we describe the result of the experiments on the entire Denmark dataset. When discussing our results, we will divide points in the simplified contour map (output points) into $\varepsilon_z$ points and $\varepsilon_{xy}$ points. These are the points that were not removed due to respectively the constraint contours and the constraints of our polygon simplification algorithm e.g. $\varepsilon_{xy}$.

**Funen dataset:** The non-simplified contour map generated from the triangulation of Funen consists of 636,973 contours with 365,641,479 points (not counting constraint contours). The results of our test runs are given in Table 7.1. The number of output points is given as a percentage of the number of points in the non-simplified contour map (not counting constraint contours). From the table it can be seen that the number of output points drops significantly as $\varepsilon_{xy}$
is raised from $0.2m$ up to $5m$. However, for values larger than $5m$ the effect on output size of increasing $\varepsilon_{xy}$ diminishes. This is most likely linked with high percentage of $\varepsilon_z$ points in the output e.g. for $\varepsilon_{xy} = 10m$ we have that $71.7\%$ of the output points are $\varepsilon_z$ points (and increasing $\varepsilon_{xy}$ will not have an effect on these).

<table>
<thead>
<tr>
<th>$\varepsilon_{xy}$ in m.</th>
<th>0.2</th>
<th>0.5</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output points (%)</td>
<td>40.4</td>
<td>23.7</td>
<td>15.2</td>
<td>10.2</td>
<td>8.8</td>
<td>7.9</td>
</tr>
<tr>
<td>$\varepsilon_z$ points</td>
<td>0.8</td>
<td>5.0</td>
<td>13.9</td>
<td>33.5</td>
<td>46.0</td>
<td>59.3</td>
</tr>
<tr>
<td>$\varepsilon_{xy}$ points</td>
<td>99.2</td>
<td>95.0</td>
<td>86.1</td>
<td>66.5</td>
<td>54.0</td>
<td>40.7</td>
</tr>
</tbody>
</table>

Table 7.1: Results for Funen with different $\varepsilon_{xy}$ thresholds ($\varepsilon_z = 0.2m$ and with $\Delta = 0,5m$).

**Denmark dataset:** When simplifying the contour map of the entire Denmark dataset we chose $\varepsilon_{xy} = 5m$, since our test runs on Funen had shown that increasing $\varepsilon_{xy}$ further would not lead to a significant reduction in output points. Table 7.2 gives the results of simplifying the contour map of Denmark. The non-simplified contour map consists of 4,793,518,863 points on 7,260,043 contours. Adding constraint contours increases the contour map size with a factor 3.0 (the constraint factor) both in terms of points and contours. In total it took 49 hours to generate and simplify the contour map and the resulting simplified contour map contained 9.2\% of the points in the non-simplified contour map (not counting constraint contours). Since 65.7\% of the output points were $\varepsilon_z$ points, it is unlikely that increasing $\varepsilon_{xy}$ would reduce the size of the simplified contour map significantly. This corresponds to our observations on the Funen dataset. Table 7.2 also contains statistics on the number of self-intersections that is removed after the simplification (as discussed in Section 7.5); both the actual number of intersections and the percentage of the contours with self-intersections are given. As it can be seen these numbers are relatively small and their removal does not contribute significantly to the running time.
Chapter 7. Simplifying Massive Contour Maps

Table 7.2: Results for Funen and Denmark with $\Delta = 0.5m$, $\varepsilon_z = 0.2m$ and $\varepsilon_{xy} = 5m$. 

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Funen</th>
<th>Denmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input points</td>
<td>365,641,479</td>
<td>4,793,518,863</td>
</tr>
<tr>
<td>Contours</td>
<td>636,973</td>
<td>7,260,043</td>
</tr>
<tr>
<td>Constraint factor</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Running time (hours)</td>
<td>1.5</td>
<td>49</td>
</tr>
<tr>
<td>Output points (% of input points)</td>
<td>7.9</td>
<td>9.2</td>
</tr>
<tr>
<td>$\varepsilon_z$ points (% of output points)</td>
<td>59.3</td>
<td>65.7</td>
</tr>
<tr>
<td>$\varepsilon_{xy}$ points (% of output points)</td>
<td>40.7</td>
<td>34.3</td>
</tr>
<tr>
<td>Total number of intersections</td>
<td>38,992</td>
<td>699,027</td>
</tr>
<tr>
<td>Contours with intersections (% of input contours)</td>
<td>2.4</td>
<td>3.1</td>
</tr>
</tbody>
</table>
Bibliography


Appendix A

Computing OptHS and OptNHS

In this section we describe the precise computing of both matrices OptHS and OptNHS. Note that the matrices OptHS and OptNHS maintain the number of links of the optimal sub-simplifications.

**OptHS** In order to compute OptHS\([i, j, b, l]\), where \(1 \leq i < j \leq n\), \(0 \leq b \leq |\text{CS}(p_ip_j)|\), and \(0 \leq l \leq |\text{CS}(P)|\), we distinguish the three cases:

- **\(b = 0\):**
  - \(\text{OptHS}[i, j, 0, l] = \min_{i'} \text{OptHS}[i', i, |\text{CS}(p_{i'}p_i)|, l] + 1\) where \((i' < i)\)

- **\(l = 0\):**
  - \(\text{OptHS}[i, j, b, 0] = \min_{i'} \text{OptNHS}[1, i', i, j, |\text{CS}(p_1p_{i'})|, b]\) where \((i' \leq i)\)

- **\(b, l > 0\):**
  - \(\text{OptHS}[i, j, b, l] = \min_{i', j'} \text{OptHS}[i', j', b' - 1, l - 1] +\)
    - \(\text{OptNHS}[i', j', i, j, |\text{CS}(p_{i'}p_{j'})| - b' + 1, b] - 1\)
  - where \(((i' = i, j' = j, b' = b - 1)\text{ or } (i' < j' \leq i))\) and \(\text{CS}(P)[l] = \text{CS}(p_{i'}p_{j'})[b']\)

**OptNHS** To fill OptNHS\([i_1, j_1, i_2, j_2, a, c]\), where \(1 \leq i_1 < j_1 \leq i_2 < j_2 \leq n\), \(0 \leq a \leq |\text{CS}(p_{i_1}p_{j_1})|\), and \(0 \leq c \leq |\text{CS}(p_{i_2}p_{j_2})|\), we distinguish the following three cases:

- **\(c > 0\):**
  - \(\text{OptNHS}[i_1, j_1, i_2, j_2, a, c] = \min_{i', j', a'} \text{OptNHS}[i_1, j_1, i', j', a, a' - 1] +\)
    - \(\text{OptNHS}[i', j', i_2, j_2, |\text{CS}(p_{i'}p_{j'})| - a', c - 1] - 1\)
  - where \((j_1 \leq i' < j' \leq i_2)\text{ and } (0 < a' \leq |\text{CS}(p_{i'}p_{j'})|)\)
    - and \(\text{CS}(p_{i'}p_{j'})[a'] = \text{CS}(p_{i_2}p_{j_2})[c]\)
• $a > 0$ and $c = 0$:

$$
\text{OptNHS}[i_1, j_1, i_2, j_2, a, 0] = \min_{i', j', c'} \text{OptNHS}[i_1, j_1, i', j', a - 1, c' - 1] + \\
\text{OptNHS}[i', j', i_2, j_2, |\text{CS}(p_i p_j)| - c', 0] - 1
$$

where $(j_1 \leq i' < j' \leq i_2)$ and $(0 < l' \leq |\text{CS}(p_i p_j)|)$

and $\text{CS}(p_{i'} p_{j'})[a'] = \text{CS}(p_{i_1} p_{j_1})[|\text{CS}(p_{i_1} p_{j_1})| - a + 1]$

• $a, c = 0$ and $j_1 \neq i_2$:

$$
\text{OptNHS}[i_1, j_1, i_2, j_2, 0, 0] = 3 \text{ if } \text{CS}(p_{j_1} p_{i_2}) = \text{NULL} \text{ and otherwise } + \infty
$$

• $a, c = 0$ and $j_1 = i_2$:

$$
\text{OptNHS}[i_1, j_1, i_2, j_2, 0, 0] = 2 \text{ if } \text{CS}(p_{j_1} p_{i_2}) = \text{NULL} \text{ and otherwise } + \infty
$$
Appendix B

Samples From Our Datasets

In Figure B.1 and B.2 we illustrate a subset of a contour map simplified with different $\varepsilon_{xy}$. The contours were made using $\Delta = 0, 5m$ and $\varepsilon_z = 0, 2m$.

Figure B.1: The top left figure shows the original curves while the remaining figures show the contour map subset for different values of $\varepsilon_{xy}$. 

0.2m.  
0.5m.  
1m.  
2m.  
3m.  

Figure B.1: The top left figure shows the original curves while the remaining figures show the contour map subset for different values of $\varepsilon_{xy}$. 
Figure B.2: Shows the contour map subset for different values of $\varepsilon_{xy}$. 