Progressive minimum-link simplification of polygonal curves

Reddingius, W.W.T.

Award date:
2017
Master’s Thesis

Progressive Minimum-Link Simplification of Polygonal Curves

by

Wim Reddingius

Supervisor
dr. K.A. Buchin

Eindhoven, July 2017
Abstract

Simplifying polygonal curves at different levels of detail is an important problem with many applications. To facilitate exploration of these simplifications, we wish to seamlessly switch between different levels of detail. These simplifications must thus be consistent with one another. We call such a set of inter-consistent simplifications a *progressive simplification*. Existing geometric optimization algorithms used for curve simplification minimize the complexity of the simplified curve for each level of detail independently. These algorithms therefore do not produce progressive simplifications. All currently known progressive simplification algorithms are based on heuristics and therefore do not strictly minimize the simplification complexity.

In this thesis, we wish to bridge the gap between these two types of algorithms. More specifically, we wish to produce progressive simplifications for which the sum or integral of the simplification complexity over all levels of detail is minimized. We present an algorithm that solves this problem in $O(n^3m)$ time, where $n$ is the length of the input curve and $m$ the number of different levels of detail. This algorithm is compatible with any distance measure such as Hausdorff or Fréchet, and can be used to compute an optimal simplification for continuous scaling in $O(n^5)$ time. We further explore two greedy algorithms with a running time of $O(n^2m)$. These algorithms greedily construct each simplification, and may therefore yield a set of simplifications for which the cumulative complexity is non-minimal.

All algorithms proposed in this thesis are based on an existing computational framework that employs finding shortest paths in so-called shortcut graphs. These are graphs in which each edge represents a single line segment that is a valid simplification for a contiguous subsequence of the input curve. Each such line segment is called a *shortcut*.

To speed up the simplification algorithms, we present multiple supporting techniques pertaining to these shortcut graphs. First of all, we propose a space-efficient representation of the shortcut graph that can be used to find shortest paths in $O(n \log n)$ time in practice. This is an improvement over using breadth-first search in $O(n^2)$ time. Furthermore, we present an algorithm for efficiently constructing shortcut graphs under the Hausdorff distance for many different levels of detail. This algorithm computes the error of all shortcuts in $O(n^2 \log n)$ time, improving over $O(n^3)$ time using existing techniques.

Experimental evaluation of the developed techniques on animal movement data reveals that the new representation of the shortcut graph drastically lowers the memory usage while significantly improving the running time of using shortcut graphs for curve simplification when compared to existing techniques. The optimal progressive simplification algorithm produces simplifications that are comparable in size to minimal non-progressive simplifications, but was found to be limited to small input curves. We showed that one of the greedy algorithms is a good alternative which produces near-minimal progressive simplifications.
## Contents

### Abstract

#### 1 Introduction

#### 2 Preliminaries

- 2.1 Minimum-link Simplification .................................................. 3
- 2.2 Progressive Simplification .................................................... 6
- 2.3 Computational Framework ..................................................... 7
- 2.4 Contributions ................................................................. 8

#### 3 Exploiting Real-World Structures in the Shortcut Graph

- 3.1 Shortcut Intervals ............................................................. 9
- 3.2 Finding Shortest Paths ......................................................... 10
- 3.3 Experimental Evaluation ..................................................... 12
  - 3.3.1 Performance by Curve Length ...................................... 13
  - 3.3.2 Performance by Error Bound ....................................... 14

#### 4 Finding Shortcuts for Multiple Scales

- 4.1 Independent Construction .................................................... 17
- 4.2 Construction for Arbitrary Scale .......................................... 19
  - 4.2.1 Operations .............................................................. 21
  - 4.2.2 Running Time .......................................................... 23
  - 4.2.3 Correctness .............................................................. 23
- 4.3 Experimental Evaluation ..................................................... 25
  - 4.3.1 Construction of Shortcut Interval Sets .......................... 25
  - 4.3.2 Construction for Multiple Scales .................................. 27

#### 5 Progressive Simplification of Polygonal Curves

- 5.1 An Optimal Algorithm ........................................................ 29
  - 5.1.1 Optimization .............................................................. 31
  - 5.1.2 Correctness ............................................................... 33
- 5.2 Greedy Construction Techniques ......................................... 35
  - 5.2.1 Top-down Construction ............................................... 35
  - 5.2.2 Bottom-up Construction ............................................. 36
  - 5.2.3 Integrating Single-Scale Simplification ......................... 37
  - 5.2.4 Worst-case Performance ............................................. 38
- 5.3 Experimental Evaluation ..................................................... 39
  - 5.3.1 Performance by Curve Length ...................................... 39
  - 5.3.2 Performance by Number of Scales .................................. 43
6 Conclusions and Future Work 45
Bibliography 47
Chapter 1

Introduction

Given a polygonal curve as input, the curve simplification problem asks for a polygonal curve that approximates the input well and that uses as few points as possible. The approximation error between the input curve and the simplification is typically defined by the maximum distance – typically Hausdorff [12] or Fréchet [3] – between an edge of the simplification and the corresponding subcurve of the input. An example of such a simplification is given in Figure 1.1a.

Because of the importance of data reduction, curve simplification has a wide range of applications. Cartography is such an example, where the visual representation of line features such as rivers, roads, countries or boundaries of regions needs to be reduced. Nowadays maps are interactive, so we need curve simplification that works with different levels of detail. Zoomable maps require progressive simplification, that is, a simplification that progressively increases the complexity of the representation when zooming in. To illustrate how progressive simplifications are different from non-progressive simplifications, consider Figure 1.1b and Figure 1.1c. Each progressive simplification produces a set of curves that is inter-consistent, meaning if we zoom in on the curve, we add additional detail while retaining the existing points.

Existing progressive algorithms (e.g. [5]) work by simplifying a curve, then simplifying the previous simplification, and so on. More concretely, a common approach is to discard the point whose removal introduces the smallest error (according to some criterion). Next, we proceed with a simplification where we remove the point with smallest error from the simplified curve of the previous round. For instance, the algorithm by Visvalingam and Whyatt [24] always removes the point which together with its neighboring points forms the smallest area triangle.

Such approaches stand in stark contrast to (non-progressive) curve simplification algorithms that aim to minimize the complexity of the simplification while guaranteeing a (global) bound on the error between the simplification and the input curve. The most prominent algorithm with a global bound on the Hausdorff distance is the Douglas-Peucker simplification algorithm [10]. While heuristically aiming at a simplification with few points, this algorithm does not actually
Chapter 1. Introduction

minimize the number of points. An algorithm that minimizes the number of points and guarantees a certain bound on the simplification error is called a min-# simplification. Alternatively, one can fix the number of points in a simplification and minimize its error value $\varepsilon$. This problem is referred to as the min-$\varepsilon$ simplification problem.

The error bound in existing progressive algorithms is relative to the previous simplification and may accumulate during the simplification process. In this thesis we present the first progressive min-# simplification algorithm which does not compromise the error bound. We extend an existing non-progressive min-# algorithm and are faced with optimizing computational aspects from this algorithm which form a bottleneck either in terms of memory usage or running time. Therefore, we present several supporting techniques for computing min-# simplifications. Although this thesis centers around progressive simplification, these techniques can be applied in a non-progressive setting as well. We further outline these contributions in technical terms at the end of Chapter 2.

Related Work

Curve simplification is a well-studied problem in the past 30 years due to its importance to applications in various domains. Imai and Iri [16] introduced a general approach for computing min-# simplifications by finding shortest paths in so-called shortcut graphs. Similar algorithms were presented in a series of papers [17, 23], resulting in algorithms under the Hausdorff distance with a running time of $O(n^2)$ and $O(n^2 \log n)$ for min-# simplification and min-$\varepsilon$ simplification respectively [6]. Shortcut graphs are flexible and can be used in conjunction with any error measure. One such error measure was proposed by Imai and Iri [15] for covering rectangles. They solved the min-$\varepsilon$ problem for this error measure using an $O(mn(\log n)^2)$-time algorithm minimizing the widths of $m$ covering rectangles.

For the $L_1$-metric, Agarwal and Varadarajan [2] presented an $O(n^{4/3+\varepsilon})$-time algorithm using a clique-cover to represent the shortcut graph. Agarwal et al. [1] later devised a greedy approximation algorithm for the min-# problem under the Fréchet distance, running in $O(n \log n)$ time. This algorithm produces simplifications for error bound $\varepsilon$ that are at most as large as optimal min-# simplifications for error bound $\varepsilon/2$.

Instead of approximation, applications often use heuristics, such as the simplification algorithm by Douglas and Peucker [10]. This algorithm uses the Hausdorff distance, and aims to heuristically minimize the number of points. The worst case running time of this heuristic is $O(n^2)$. Hershberger and Snoeyink [13] showed that this heuristic can be implemented to run in $O(n \log n)$ time, which they later improved to $O(n \log^* n)$ time for non self-intersecting polygonal paths in the line model [14].

As previously mentioned, progressive simplifications are commonly used in cartography [19]. A popular such algorithm was proposed by Visvalingam and Whyatt [21], which iteratively removes the point which is part of the triangle with the smallest area. Inspired by this, Daneshpajouh et al. [8] defined an error measure for non-progressive simplification by measuring the sum of the area in between a simplification and the input curve. Under the area difference measure, they devised a quadratic-time min-# approximation algorithm.
Chapter 2
Preliminaries

Before delving into the engineering of the algorithms, we present technical details that lie at the core of the problems that we are trying to solve, and the framework we use to solve these problems. The details discussed here will remain central throughout this thesis, and should therefore provide a frame of reference for the chapters to follow.

We open this chapter with a discussion of minimum-link polygonal curve simplifications on a single scale (Section 2.1), followed by how this problem can be extended to produce so-called progressive simplifications (Section 2.2). In Section 2.3 we present an existing computational framework used for single-scale simplification which we extend to compute these progressive simplifications. Finally, in Section 2.4 we outline how aspects of this framework relate to the main contributions of this thesis.

2.1 Minimum-link Simplification

In this thesis we are interested in the so called min-# problem, which is an optimization problem that takes as input a polygonal curve $C = \langle p_1, \ldots, p_n \rangle$ represented by a sequence of points in $\mathbb{R}^d$ where $d > 1$, and an error bound $\varepsilon \geq 0$. We wish to simplify this polygonal curve to a polygonal curve $S$ on which the following conditions hold:

- The size of $S$ should be minimized, meaning $|S|$ is minimal.
- $S$ sufficiently approximates $C$, meaning $D(C, S) \leq \varepsilon$, where $D$ is an error function.

Different error functions $D$ exist that define how the similarity, or “distance”, between polygonal curves is determined or measured. Among these error functions are the Hausdorff distance [12], and the Fréchet distance [4]. In this thesis, we will mostly consider the Hausdorff distance. However, most of these error measures are computed with the same time complexity, meaning that in many cases any error measure can be used. To define the Hausdorff distance, we use $p \dashv P$ to denote any point $p$ that lies on polygonal curve $P$.

Hausdorff Distance: For any two polygonal curves $P$ and $Q$, the Hausdorff distance $\mathcal{H}(P, Q)$ is defined as:

$$\mathcal{H}(P, Q) = \max [\mathcal{H}(P \rightarrow Q), \mathcal{H}(Q \rightarrow P)]$$

$$\mathcal{H}(P \rightarrow Q) = \max_{p \dashv P, q \dashv Q} \text{dist}(p, q)$$

3
We use \( \text{dist}(x, y) \) to denote the Euclidean distance between points \( x \) and \( y \).

Simply stated, the Hausdorff distance is the greatest of all the distances from a point on one of the curves to the closest point on the other curve. These distances are visualized in Figure 2.1a and Figure 2.1b. By limiting this distance to \( \varepsilon \), all points of the simplification \( S \) must remain within the tolerance region of \( C \). This is visualized for a single line segment in Figure 2.1c.

Figure 2.1: The Hausdorff distance.

Note how the problem as defined so far has a solution set which is continuous, since points of \( S \) may be chosen anywhere in \( \mathbb{R}^d \). The min-\# problem is therefore typically simplified such that \( S \) is a subsequence of \( C \) including \( p_1 \) and \( p_n \). This has the advantage that there is a clear association between segments of \( S \) and subcurves of \( C \). Therefore, if \( S \) contains the line segment \((p_i, p_j)\), we want the subcurve of \( C \) from \( p_i \) to \( p_j \) to be close to this segment. This gives us the following set of constraints:

- \( S \) is a subsequence of \( C \), denoted \( S \subseteq C \).
- The first and last point of \( C \) must be part of any simplification, thus \( p_1, p_n \in S \).
- Any line segment \((p_i, p_j)\) ∈ \( S \) sufficiently approximates the subsequence \( \langle p_i, \ldots, p_j \rangle \subseteq C \). We therefore have \( \max_{(p_i, p_j) \in S} D(\langle p_i, \ldots, p_j \rangle, \langle p_i, p_j \rangle) \leq \varepsilon \).

By redefining the error this way, we are more stringent on the shape of simplification \( S \) with respect to the input curve \( C \). For instance, by defining the Hausdorff distance globally, i.e. \( \mathcal{H}(C, S) \leq \varepsilon \), the closest point on \( C \) to any point on \( S \) can be picked anywhere on \( C \), and vice versa. Therefore, some edges of \( S \) may poorly approximate the corresponding subsequence on \( C \), without violating the maximum Hausdorff distance. This is illustrated in Figure 2.2.

This definition of the error on subsequences of \( C \) also facilitates area-based error measures, such as the measures suggested by Daneshpajouh et al. [8]. These measures define the error of line segment \((p_i, p_j)\) ∈ \( S \) in terms of the area of the simple polygons formed by the intersections of \( \langle p_i, p_j \rangle \) and \( \langle p_i, \ldots, p_j \rangle \).

Figure 2.2: An example where globally defining the Hausdorff distance gives a simplification where the shape of \( C \) is lost. Note that \( \mathcal{H}(C, S) \leq \varepsilon \), but \( \max_{(p_i, p_j) \in S} \mathcal{H}(\langle p_i, \ldots, p_j \rangle, \langle p_i, p_j \rangle) > \varepsilon \).

Another advantage of imposing a maximum error on each subsequence, is that most error measures are considerably easier to compute when one of the polygonal curves is just a single line.
is also called the perpendicular distance from a point to a line. Now prove that this indeed computes \( H(p_i, p_j) \) as the length of the line segment perpendicular to \( (p_i, p_j) \) which joins \( p_k \) to \( (p_i, p_j) \). This distance is also called the perpendicular distance from a point to a line.

**Lemma 2.1.** For any polygonal curve \( C = \langle p_1, \ldots, p_n \rangle \), the Hausdorff distance \( H((p_i, \ldots, p_j), (p_i, p_j)) \) of any line segment \( (p_i, p_j) \) where \( 1 \leq i < j \leq n \) is equal to \( \max_{p_k \in (p_i, \ldots, p_j)} \text{dist}(p_k, (p_i, p_j)) \).

**Proof.** We know that the shortest distance from a line segment \( (p_i, p_j) \) to a point \( p_k \in C \) is the same as the shortest distance from \( (p_i, p_j) \) to a point \( p_k \in C \). This means we have:

\[
H((p_i, \ldots, p_j) \to (p_i, p_j)) = \max_{x \in (p_i, \ldots, p_j)} \min_{y \in (p_i, p_j)} \text{dist}(x, y) = \max_{p_k \in (p_i, \ldots, p_j)} \min_{y \in (p_i, p_j)} \text{dist}(p_k, y) = \max_{p_k \in (p_i, \ldots, p_j)} \text{dist}(p_k, (p_i, p_j))
\]

Now let us assume that \( H((p_i, p_j) \to (p_i, \ldots, p_j)) > H((p_i, \ldots, p_j) \to (p_i, p_j)) \). This means that there is a point \( u \in (p_i, p_j) \) for which there is a closest point \( v \) on some line segment \( (p_k, p_{k+1}) \) in the subsequence \( (p_i, \ldots, p_j) \), such that the distance between \( u \) and \( v \) is larger than \( H((p_i, \ldots, p_j) \to (p_i, p_j)) \). This situation is sketched in Figure 2.3a. Note that \( (p_i, \ldots, p_j) \) cannot pass through the illustrated circle with radius \( \text{dist}(u, v) \), since otherwise \( v \) is not the closest point from \( u \) on \( (p_1, \ldots, p_j) \). Furthermore, \( (p_i, \ldots, p_j) \) cannot touch or cross the boundaries sketched at the top and the bottom, because otherwise \( H((p_i, \ldots, p_j) \to (p_i, p_j)) \geq \text{dist}(u, v) > H((p_i, \ldots, p_j) \to (p_i, p_j)) \). Therefore, there is no conceivable configuration of \( (p_i, \ldots, p_j) \). By contradiction, we may thus conclude \( H((p_i, p_j) \to (p_j, \ldots, p_i)) \leq H((p_i, \ldots, p_j) \to (p_i, p_j)) \).

Using the definition of the Hausdorff distance, we conclude \( H((p_i, \ldots, p_j), (p_i, p_j)) \) is equal to \( \max_{p_k \in (p_i, \ldots, p_j)} \text{dist}(p_k, (p_i, p_j)) \).

Although the min-# problem is well defined for any number of dimensions, we will solely focus on two-dimensional polygonal curves. We can now formally define the problem as follows:

**Single Scale Min-#**

**Input:** A polygonal curve \( C = \langle p_1, \ldots, p_n \rangle \) in \( \mathbb{R}^2 \), and an error bound \( \varepsilon \geq 0 \).

**Output:** A polygonal curve \( S \) in \( \mathbb{R}^2 \) where \( S \subseteq C \), such that \( |S| \) is minimal, and:

- \( p_1 \in S \) and \( p_n \in S \).
- \( \max_{(p_i, p_j) \in S} D((p_i, \ldots, p_j), (p_i, p_j)) \leq \varepsilon \).
2.2 Progressive Simplification

By extending the min-# simplification problem to multiple scales, we are concerned with finding \( m \) simplifications \( \langle S_1, \ldots, S_m \rangle \) for \( m \) error bounds \( E = \langle \varepsilon_1, \ldots, \varepsilon_m \rangle \) where \( \varepsilon_m > \ldots > \varepsilon_1 \geq 0 \).

We can construct each simplification \( S_k \) by using any algorithm to solve Single Scale Min-# for \( C \) and \( \varepsilon_k \). However, by constructing each simplification independently, we run the risk that some simplifications have a different shape than others. To facilitate interactive exploration of these simplifications where one can seamlessly switch between different simplifications, we therefore desire some form of inter-consistency. More specifically, we require a mapping from each subsequence of points in \( S_i \) to a subsequence of points in \( S_j \) where \( i \neq j \). The points shared between any two simplifications thereby provide a frame of reference by which one can infer the relationship between these simplifications.

We perform this mapping by enforcing monotonic containment on the points of each simplification, from \( S_m \) to \( S_1 \). We thus require \( S_i \subseteq S_{i-1} \) for all \( 1 \leq i \leq m \), and \( S_1 \subseteq C \). This way, detail is progressively added to simplifications at lower scales. Figure 2.4 illustrates this monotonic containment relation. We call each set of simplifications of \( C \) for \( E \) that adheres to this monotonic containment relation a progressive simplification. The min-# progressive simplification problem can be defined as follows:

**Progressive Min-#**

**Input:** A polygonal curve \( C = \langle p_1, \ldots, p_n \rangle \) in \( \mathbb{R}^2 \) and a sequence of error bounds \( E = \langle \varepsilon_1, \ldots, \varepsilon_m \rangle \) where \( \varepsilon_m > \ldots > \varepsilon_1 \geq 0 \).

**Output:** A sequence of polygonal curves \( \langle S_1, \ldots, S_m \rangle \) in \( \mathbb{R}^2 \) where \( S_m \subseteq \ldots \subseteq S_1 \subseteq C \), such that \( \sum_{i=1}^{m} |S_i| \) is minimal, and for all \( 1 \leq i \leq m \) we have:

- \( p_1 \in S_i \), and \( p_n \in S_i \).
- \( \max_{(p_x, p_y) \in S_i} \mathcal{D}(\langle p_x, \ldots, p_y \rangle, \langle p_x, p_y \rangle) \leq \varepsilon_i \).

If we wish to prioritize the minimization of simplifications at specific scales, we can generalize Progressive Min-# such that we have a weight for every simplification which represents the importance of the minimization of that simplification. We thus have a sequence of weights \( W = \langle w_1, \ldots, w_m \rangle \) in \( \mathbb{R} \), where \( w_i > 0 \) for all \( 1 \leq i \leq m \). We now wish to minimize \( \sum_{i=1}^{m} w_i |S_i| \). We call this generalized problem Weighted Progressive Min-#.

So far we have considered discrete progressive simplifications, meaning we simplify for a given set of error bounds. We may instead want to simplify continuously, capturing every possible level of detail. We thus wish to find a simplification \( S_\varepsilon \) for all \( 0 \leq \varepsilon \leq \varepsilon_z \), where \( \varepsilon_z = \mathcal{D}(\langle p_1, \ldots, p_n \rangle, \langle p_1, p_n \rangle) \), which is the error of simplification \( \langle p_1, p_n \rangle \). Note that monotonic containment is still imposed, implying \( S_{\varepsilon'} \subseteq S_\varepsilon \) for all \( \varepsilon' > \varepsilon \). The goal is to minimize the cumulative size of this continuous set.
of simplifications. We thus want to minimize \( \int_{0}^{\varepsilon} |S_{\varepsilon}| \, d\varepsilon \). We call this continuous generalization **Continuous Progressive Min-\#**. Now let us prove that **Continuous Progressive Min-\#** can be reduced to **Weighted Progressive Min-\#**.

**Lemma 2.2.** Given a polygonal curve \( C = \langle p_1, \ldots, p_n \rangle \), we can solve **Continuous Progressive Min-\#** by solving **Weighted Progressive Min-\#** for \( \binom{n}{2} \) error bounds.

**Proof.** Assume we have the error \( D(\langle p_i, \ldots, p_j \rangle, \langle p_i, p_j \rangle) \) of every line segment \( (p_i, p_j) \) where \( j > i \). Let \( \langle \varepsilon_1, \ldots, \varepsilon_j \rangle \) be the sorted list of these errors in ascending order without duplicates. Note that it is possible that \( \varepsilon_i < \varepsilon(\frac{j}{2}) \).

Now consider the claim \( S_{\varepsilon} = S_{\varepsilon_i} \) for all \( \varepsilon \in [\varepsilon_i, \varepsilon_{i+1}) \). We know this holds, since assuming \( S_{\varepsilon} < S_{\varepsilon_i} \) would imply all \( S_{\varepsilon'} \) where \( \varepsilon' \in [\varepsilon_i, \varepsilon] \) are not minimal. Therefore, we have

\[
\int_{0}^{\varepsilon} |S_{\varepsilon}| \, d\varepsilon = \sum_{k=1}^{z-1} (\varepsilon_{k+1} - \varepsilon_k) |S_{\varepsilon_k}|
\]

Thus, we can solve **Continuous Progressive Min-\#** by solving **Weighted Progressive Min-\#** for error bounds \( \varepsilon = \langle \varepsilon_1, \ldots, \varepsilon_z \rangle \) and weights \( W = \langle w_1, \ldots, w_z \rangle \), where \( w_i = \varepsilon_{i+1} - \varepsilon_i \) for all \( 1 \leq i < z \).

## 2.3 Computational Framework

In order to solve the problems stated in the previous sections, we use a computational framework proposed by Imai and Iri [18]. This framework uses the notion of a **shortcut**, which is defined as follows:

**Shortcut:** For any polygonal curve \( C = \langle p_1, \ldots, p_n \rangle \) and an error bound \( \varepsilon \geq 0 \), any line segment \( (p_i, p_j) \) where \( j > i \) is a shortcut of \( C \) for \( \varepsilon \) if and only if \( D(\langle p_1, \ldots, p_j \rangle, \langle p_i, p_j \rangle) \leq \varepsilon \).

Intuitively, a shortcut \( (p_i, p_j) \) is a line segment that sufficiently approximates all points in \( C \) between \( p_i \) and \( p_j \), given an error bound \( \varepsilon \). Hence, a shortcut of \( C \) for \( \varepsilon \) is a line segment that can be included in any simplification \( S \) with an error bound of at most \( \varepsilon \). Note how for any \( 1 \leq i < n \), \( (p_i, p_{i+1}) \) is always a shortcut, regardless of \( C \) and \( \varepsilon \), because we always have \( D((p_i, p_{i+1}), (p_i, p_j)) = 0 \).

The computational framework defines a **shortcut graph** as a graph that contains all shortcuts for a given error bound as its edges. We formally define this graph as follows:

**Shortcut Graph:** Given a polygonal curve \( C = \langle p_1, \ldots, p_n \rangle \) and an error bound \( \varepsilon \geq 0 \), a shortcut graph is a graph \( G(C, \varepsilon) = (V, E) \), where \( V = C \) and \( E = \{ (p_i, p_j) \mid (p_i, p_j) \text{ is a shortcut of } C \text{ for } \varepsilon \} \).

We can solve **Single Scale Min-\#** by finding a shortest path from \( p_1 \) to \( p_n \) in \( G(C, \varepsilon) \) using a breadth-first search [18] in \( O(n^2) \) time. An example of each step of the computational framework is shown in Figure 2.4.

To prove correctness of this framework, we first show that any simplification it creates is valid, meaning that each of its edges adheres to the error bound. We furthermore prove that it is minimal by showing that there cannot be another valid simplification that is smaller.

**Lemma 2.3.** For any polygonal curve \( C = \langle p_1, \ldots, p_n \rangle \) and an error bound \( \varepsilon \geq 0 \), if \( \pi \) is a path in \( G(C, \varepsilon) \), then \( \max_{(p_i, p_j) \in \pi} D(\langle p_i, \ldots, p_j \rangle, \langle p_i, p_j \rangle) \leq \varepsilon \).

**Proof.** For any \( (p_i, p_j) \in \pi \), we know \( D((p_i, \ldots, p_j), (p_i, p_j)) \leq \varepsilon \), since \( (p_i, p_j) \in G(C, \varepsilon) \). Since \( (p_i, p_j) \) is an arbitrary edge in \( \pi \), we conclude \( \max_{(p_i, p_j) \in \pi} D((p_i, \ldots, p_j), (p_i, p_j)) \leq \varepsilon \). \( \square \)
Chapter 2. Preliminaries

Lemma 2.4. For any polygonal curve $C = \langle p_1, \ldots, p_n \rangle$ and an error bound $\epsilon \geq 0$, any shortest path $\pi$ from $p_1$ to $p_n$ in $G(C, \epsilon)$ is at most as long as any smallest polygonal curve $S$ for which $\max(p_i, p_j) \in S$ $D(\langle p_1, \ldots, p_j \rangle, \langle p_i, p_j \rangle) \leq \epsilon$, $S \subseteq C$ and $p_1, p_n \in S$.

Proof. Consider any edge $(p_i, p_j) \in S$. For this edge we have $D(\langle p_1, \ldots, p_j \rangle, \langle p_i, p_j \rangle) \leq \epsilon$. Therefore, $(p_i, p_j)$ must be a shortcut of $C$ for $\epsilon$ which means that $(p_i, p_j)$ is an edge in $G(C, \epsilon)$. Because $(p_i, p_j)$ is an arbitrary link in $S$, the polygonal curve $S$ must be a path in $G(C, \epsilon)$. Because $\pi$ is a shortest path from $p_1$ to $p_n$ in $G(C, \epsilon)$, we conclude $|\pi| \leq |S|$. □

2.4 Contributions

Throughout this thesis we discuss different adaptations of the computational framework described in Section 2.3. In Chapter 3, we investigate an alternative representation of the shortcut graph called a shortcut interval set. In practice, this representation has sub-quadratic storage requirements which allows us to find shortest paths in $O(n \log n)$ time. This is an improvement over using breadth-first search in an explicit representation of the shortcut graph using $O(n^2)$ time and space. Experimental evaluation reveals that a significant improvement is realized by using shortcut interval sets, both in terms of memory usage and running time.

In Chapter 4, we discuss how shortcut graphs can efficiently be constructed for many different error bounds using the Hausdorff distance. We present an algorithm that computes for each line segment the maximum error at which it is a valid shortcut. These errors can be used to efficiently construct the shortcut graph associated with any error bound. This algorithms runs in $O(n^2 \log n)$ time, which is an improvement over $O(n^3)$ time using existing techniques by Imai and Iri [10]. The results indicate that this algorithm yields benefits over existing techniques, but only when constructing a large number of shortcut graphs. This algorithm is therefore only suitable when simplifying for many different levels of detail.

Finally, In Chapter 5 we elaborate on how we can use shortcut graphs to compute progressive simplifications. We present the first progressive min-\# simplifications algorithm running in $O(n^3 m)$ time. We furthermore propose two $O(n^2 m)$-time greedy algorithms which do not strictly minimize the cumulative simplification size. Analysis of the performance of these algorithms in practice reveals that the optimal progressive simplification algorithm produces simplifications that are comparable in size to minimal non-progressive simplifications. However, we found that this algorithm is unscalable to large input curves due to its cubic time complexity. One of the greedy algorithms was shown to be a good alternative which produces near-minimal progressive simplifications.
Chapter 3

Exploiting Real-World Structures in the Shortcut Graph

As outlined in Section 2.3, the data structure central to the computational framework is the shortcut graph. This graph encodes all contiguous subsequences of $\mathcal{C}$ which can be sufficiently approximated with a single line segment called a shortcut. A natural way of implementing a graph is by storing all edges incident to any node explicitly. We refer to shortcut graphs that are implemented this way as explicit shortcut graphs. In this chapter, we present how common patterns in real-world data can be exploited to obtain a compressed representation of the shortcut graph, called a shortcut interval set. Aside from lowering memory usage, this representation aims to optimize running time of finding shortest paths in practice.

We open this chapter with an outline of the aforementioned patterns and how they can be used to compress the shortcut graph to a sub-quadratic representation (Section 3.1). Next, we address how this new representation can be integrated with the computational framework to find shortest paths (Section 3.2). Finally, we experimentally evaluate how effective the compression of this new representation is in practice (Section 3.3).

3.1 Shortcut Intervals

As outlined in Chapter 1, polygonal curve simplification is mostly applied to spatial data, such as line features on a map, or movement trajectories of migratory animals. Most types of spatial data are structured, and follow the following common intuitions:

- Most consecutive points are near one another.
- Most consecutive line segments have similar heading.

Whenever these patterns are present, we expect most consecutive points $p_i$ and $p_j$ to share points with which they form shortcuts. This means that when $(p_x, p_i)$ is a shortcut for some point $p_x$, we expect $(p_x, p_j)$ to be a shortcut as well. Similarly, if $(p_i, p_y)$ is a shortcut for some point $p_y$, we expect $(p_j, p_y)$ to be a shortcut. This yields contiguous subsequences of $\mathcal{C}$ with which $p_i$ forms a sequence of shortcuts. This is illustrated in Figure 3.1. We call each such subsequence a shortcut interval, which is defined as follows:

**Shortcut Interval:** For any polygonal curve $\mathcal{C} = (p_1, \ldots, p_n)$, an error bound $\varepsilon \geq 0$, and $1 \leq i < x \leq y \leq n$, the interval $[x, y]$ is a shortcut interval for $p_i$ and $\varepsilon$ if and only if for all $x \leq z \leq y$, $(p_i, p_z)$ is a shortcut of $\mathcal{C}$ for $\varepsilon$, and $[x, y]$ is maximal.
Chapter 3. Exploiting Real-World Structures in the Shortcut Graph

Figure 3.1: Shortcut intervals associated with some point $p_i$.

We combine these shortcuts intervals into sets called **shortcut interval sets**, which are defined as follows:

**Shortcut Interval Set:** For any polygonal curve $C = \langle p_1, \ldots, p_n \rangle$ and an error bound $\varepsilon \geq 0$, a shortcut interval set of $C$ for $\varepsilon$ is a set $I(C, \varepsilon) = \langle I_1(\varepsilon), \ldots, I_n(\varepsilon) \rangle$ where for all $1 \leq i \leq n$, $I_i(\varepsilon)$ is the collection of all shortcut intervals $[x, y]$ for $p_i$ and $\varepsilon$.

Note that a shortcut interval set is a minimally sized set of shortcut intervals that covers all shortcuts. We can visualize a shortcut interval set in a matrix where the rows and columns correspond to the points in $C$, and the shading of each cell at position $i, j$ corresponds to whether $(p_i, p_j)$ is a shortcut. In Figure 3.2 we can see such matrices for four different error bounds. Note how regardless of the error bound, every row and column only has a few black regions. This means we typically have $|I_i| = O(1)$ shortcut intervals for any $p_i \in C$. This implies that $I(C, \varepsilon)$ has a storage complexity of $O(n)$ in practice, regardless of $\varepsilon$. This is an order of magnitude smaller than the explicit representation of the shortcut graph $G(C, \varepsilon)$, which has a storage complexity of $O(n^2)$.

Similar compression techniques were used by Alewijnse et al. [3] to speed up trajectory segmentation.

![Figure 3.2: Matrix visualization of the shortcut interval sets of a polygonal curve composed of 757 points for four different error bounds. Each black cell represents a shortcut.](image)

3.2 Finding Shortest Paths

Let us now discuss how we can find shortest paths from a source node $p_s$ to a target node $p_t$ in a shortcut interval set. On explicit shortcut graphs, this is most efficiently computed using breadth-first search in $O(n^2)$ time [13]. In this section we elaborate on how we can exploit shortcut intervals to improve this running time in practice.

We construct a balanced binary search tree $T$ containing every point $p_i \in \langle p_s, \ldots, p_t \rangle$ ordered by the index $i$ of any point $p_i$. Consider we have a point $p_b \in T$ where all points in $\langle p_a, \ldots, p_c \rangle$ are rooted at $p_b$. We wish to annotate $p_b$ with the following two paths:
### 3.2. Finding Shortest Paths

#### Figure 3.3: Finding the shortest path from $p_i$ to $p_t$ by means of range queries. For every grey node we consider the subtree annotation, and for every white node we consider the node annotation.

- A shortest path from $p_b$ to $p_t$.
- A shortest path from any point in $\langle p_a, \ldots, p_c \rangle$ to $p_t$.

We therefore have an annotation of every node, and of every subtree. We represent the subtree annotation using the path’s first point (some point in $\langle p_a, \ldots, p_c \rangle$) and the path’s length. The node annotation is represented using the path’s second point (the point after $p_b$) and the path’s length. This way, the annotations encode the next step in a shortest path to $p_t$. Any shortest path can therefore be reconstructed by following the annotations from node to node until we reach $p_t$.

The complete annotation of $T$ is achieved by inserting every point $p_i$ in $C$ from $p_t$ down to $p_s$ in sequence, and obtaining the node annotation of $p_i$ using a customized range query on the binary search tree for every shortcut interval in $I_i(\varepsilon)$. Every range query for shortcut interval $[x, y] \in I_i(\varepsilon)$ finds shortest path candidates from $p_j$ to $p_t$ where $(p_i, p_j)$ is a shortcut of $C$ for $\varepsilon$ and $x \leq j \leq y$. Specifically, we start at the root node of $T$, and for every node $p_b$ where all nodes in $\langle p_a, \ldots, p_c \rangle$ are rooted at $p_b$, we do the following:

- If $x \leq a \leq c \leq y$, consider the subtree annotation of $p_b$.
- Otherwise:
  - If $x \leq b \leq y$, consider the node annotation of $p_b$.
  - If $x < b$ and $y \geq a$, traverse to the left child of $p_b$, containing points $p_i$ where $i < b$.
  - If $y > b$ and $x \leq c$, traverse to the right child of $p_b$, containing points $p_i$ where $i > b$.

In the first case we have $x \leq a \leq c \leq y$, and therefore all points in $\langle p_a, \ldots, p_c \rangle$ lie in the shortcut interval $[x, y]$, meaning that the subtree annotation of $p_b$ is a path that can be prepended with $p_i$ to form a path from $p_i$ to $p_t$. By performing these range queries, we therefore efficiently collect the shortest possible paths from $p_i$ to $p_t$ via all shortcuts in $I_i(\varepsilon)$. An example is given in Figure 3.3.

After all candidate paths are found, we choose the path $\pi$ with the shortest length. We prepend $\pi$ with $p_i$, and use this path as the node annotation of $p_i$. The subtree annotation of $p_i$ is efficiently maintained using the node annotation of $p_i$ and the subtree annotation of both children of $p_i$.

After all points are inserted, we reconstruct the shortest path from $p_s$ to $p_t$ using the node annotation of $p_s$. 
Chapter 3. Exploiting Real-World Structures in the Shortcut Graph

Running time Each range query on some shortcut interval \([x, y]\) follows a path down the search tree \(T\) to locate a subtree which has \(p_x\) as a minimum. Similarly, we locate a subtree which has \(p_y\) as a maximum. We therefore only traverse the children of at most two nodes at any level of the binary search tree. This means that on each level of the tree, we spend \(O(1)\) time. Since the binary search tree is balanced, it has a height of \(O(\log n)\). Each range query therefore takes \(O(\log n)\) time.

Furthermore, note that annotating the tree does not influence the asymptotic running time of inserting a node. This is because any subtree annotation of some node only has a possible impact on the subtree annotation of all its ancestors. The subtree annotation of a node is maintained in \(O(1)\) by checking the subtree annotation of all its children. Because the tree is balanced, any node has at most \(O(\log n)\) ancestors, implying that updating these annotations takes at most \(O(\log n)\) time.

We conclude that each range query and insertion takes \(O(\log n)\) time in the worst case. We know that for all \(s \leq i \leq t\), we typically have \(|I_i(\varepsilon)| = O(1)\), and we therefore perform \(O(1)\) range queries for every point in practice. Since we may need to insert \(O(n)\) points, finding a shortest path in a shortcut interval set therefore takes \(O(n \log n)\) time in practice. In the worst case, we have \(O(n^2)\) shortcut intervals in total, in which case the running time is \(O(n^2 \log n)\).

Optimization Despite the expectation that \(|I_i(\varepsilon)| = O(1)\) for any point \(p_i\), it is possible that some shortcut intervals in \(I_i(\varepsilon)\) are infinitesimal. In the worst case, a shortcut interval is so small that performing the range query takes more time than simply checking the node annotation of all points in the interval. Therefore, if a shortcut interval \([x, y] \in I_i(\varepsilon)\) is smaller than \(O(\log n)\), we can save time by doing a brute force determination of the corresponding shortest path in \(O(y - x)\) time.

Note how this influences the worst case number of range queries from \(O(n)\) to \(O(\frac{n}{\log n})\). Because each range query takes \(O(\log n)\) time, we conclude that this optimization brings the worst case running time down to \(O(n^2)\).

3.3 Experimental Evaluation

The performance of using shortcut interval sets to find shortest paths depends highly on the signature of \(\mathcal{C}\), and the signature of the shortcut intervals contained in \(I(\mathcal{C}, \varepsilon)\). In this section we therefore investigate both the impact of using shortcut interval sets for a variable length of input curve \(\mathcal{C}\), and for a variable error bound \(\varepsilon\).

One motivation for progressively simplifying polygonal curves is to visualize and interactively explore movement trajectories on multiple scales. All algorithms and techniques presented in this thesis are therefore experimentally evaluated using a movement trajectory. More specifically, we use a movement trajectory of a griffon vulture migrating across south Europe by Schmidt-Rothmund [21]. This trajectory is shown on a geographic map in Figure 3.4. This is a trajectory composed of 340,000 points with street-level detail, starting in south Germany and ending in southern Spain.

Shortcut graphs in the worst case have \(\binom{n}{2}\) edges. This means that explicit shortcut graphs in the worst case require quadratic memory. In order to have experiments that are scalable to dense shortcut graphs, we therefore reduce the number of the points in the input curve. We do this by
only including the desired number of points at the beginning of the curve.

Furthermore, all experiments presented in this thesis are performed on a 64-bit Intel Core i7-2630QM machine running Windows 10 with 8 gigabytes of DDR3 SDRAM. The source code is written in C# 6.0 and compiled for 64-bit CPU’s.

3.3. Performance by Curve Length

In this section we evaluate how the length of the input curve impacts the comparative asymptotic performance of using shortcut interval sets instead of explicit shortcut graphs in practice. For this, we choose the error bound for every length of the input curve such that the resulting shortcut graph includes the shortcuts with the 50% smallest errors. The resulting shortcut graph will therefore always have a density of 50%. We choose this density so that there are many shortcuts, yet shortest paths are non-trivial.

We experimentally compare the memory footprint by comparing the number of shortcuts and the number of shortcut intervals. Furthermore, we compare the asymptotic running time in practice of using breadth-first search (BFS) on explicit shortcut graphs, and range queries on shortcut interval sets. The goal of each running-time experiment is to find a shortest path from $p_1$ to $p_n$.

The range queries on shortcut intervals are performed using left-leaning red-black trees [22]. The red-black tree [7] is a popular type of self-balancing binary search tree. Although left-leaning red-black trees perform worse compared to regular red-black trees, they are significantly easier to implement and achieve the same asymptotic time-complexity for each operation.

The results of these experiment are presented in Table 3.1.

<table>
<thead>
<tr>
<th>Length of the input curve</th>
<th>500</th>
<th>1000</th>
<th>2000</th>
<th>3000</th>
<th>4250</th>
<th>5500</th>
<th>7000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-First Search (sec.)</td>
<td>0.003</td>
<td>0.011</td>
<td>0.049</td>
<td>0.104</td>
<td>0.209</td>
<td>0.379</td>
<td>0.588</td>
</tr>
<tr>
<td>Range Queries (sec.)</td>
<td>0.005</td>
<td>0.011</td>
<td>0.027</td>
<td>0.037</td>
<td>0.059</td>
<td>0.078</td>
<td>0.097</td>
</tr>
<tr>
<td>Shortcuts ($\times 10^5$)</td>
<td>6.287</td>
<td>25.073</td>
<td>100.151</td>
<td>225.225</td>
<td>451.878</td>
<td>765.613</td>
<td>1225.52</td>
</tr>
<tr>
<td>Shortcut Intervals ($\times 10^5$)</td>
<td>0.104</td>
<td>0.163</td>
<td>0.314</td>
<td>0.303</td>
<td>0.699</td>
<td>0.694</td>
<td>0.789</td>
</tr>
</tbody>
</table>

Table 3.1: Running time in seconds of finding a shortest path from $p_1$ to $p_n$, and number of shortcuts and shortcut intervals for various lengths of the input curve, on a shortcut graph with a density of 50%.

A first glance at these results reveals that the number of shortcuts grows quadratically in the length of the input curve, whereas the number of shortcut intervals grows linearly, and non-monotonically. We confirm this insight by plotting the number of shortcuts/shortcut intervals against the length of the input curve, as shown in Figure 3.5.

Recall that by using range queries to find shortest paths in shortcut interval sets, we spend time relative to the number of shortcut intervals. Breadth-first search on the other hand spends time for every shortcut it traverses. To investigate how this different allocation of time determines the overall running time of each approach, consider Figure 3.6. We observe that the running time of breadth-first search is quadratic in the length of the input curve, whereas the range queries exhibit near-linear performance.

In conclusion, the results indicate the linear memory footprint of shortcut intervals sets holds true in practice, which in turn yields a significant improvement to the time required to find shortest paths. We foresee this significant compression of the shortcut graph to be an important stepping stone towards constructing min-# simplifications in near-linear time on large data.
Chapter 3. Exploiting Real-World Structures in the Shortcut Graph

3.3.2 Performance by Error Bound

The level of compression that can be obtained by using shortcut interval sets depends highly on the density of the shortcut graph. This phenomenon is observable in Figure 3.2. As the error bound grows, the density of the shortcut graph grows, which typically means that the shortcut intervals become coarser. An extreme example of this is a complete shortcut graph, which can be represented with \( n-1 \) shortcut intervals \([i+1, n]\) for every point \( p_i \in C \) where \( 1 \leq i < n \).

We perform the same experiments as in Section 3.3.1, except that we vary the error bound instead of the length of the input curve. We fix the length of the input curve to 10,000 points. The results obtained are given in Table 3.2. Note that the last error bound \( \varepsilon = 0.095 \) yields a complete shortcut graph.

Inspection of these results reveals that the number of shortcut intervals sporadically increases and decreases. As described earlier, this is related to the growth in coarseness among the shortcut intervals as the error bound grows. The relation between the shortcut graph complexity and the error bound is shown in Figure 3.7. We observe monotonic growth in the number of shortcuts, whereas the number of shortcut intervals peeks around \( \varepsilon = 0.05 \), which corresponds to a shortcut graph density of 80%.

A plot of the running time of both path finding algorithms is given in Figure 3.8. We observe how for small error bounds \( \varepsilon < 0.00015 \), breadth-first search on explicit shortcut graphs is faster compared to using range queries on shortcut interval sets. For these error bounds, there are few shortcuts, and path finding via range queries therefore wastes time inserting \( n \) points in the annotated balanced binary search tree, without making use of the tree.

To analyze how the shortest path finding performance relates to the number of shortcuts and
### 3.3. Experimental Evaluation

<table>
<thead>
<tr>
<th>Error bound</th>
<th>0.0001</th>
<th>0.001</th>
<th>0.01</th>
<th>0.03</th>
<th>0.06</th>
<th>0.08</th>
<th>0.095</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-First Search (sec.)</td>
<td>0.116</td>
<td>0.465</td>
<td>0.77</td>
<td>1.234</td>
<td>2.367</td>
<td>2.619</td>
<td>2.623</td>
</tr>
<tr>
<td>Range Queries (sec.)</td>
<td>0.161</td>
<td>0.183</td>
<td>0.187</td>
<td>0.177</td>
<td>0.17</td>
<td>0.152</td>
<td>0.138</td>
</tr>
<tr>
<td>Shortcuts ($\times 10^6$)</td>
<td>2.313</td>
<td>9.682</td>
<td>15.684</td>
<td>24.044</td>
<td>44.991</td>
<td>49.984</td>
<td>49.995</td>
</tr>
<tr>
<td>Shortcut Intervals ($\times 10^6$)</td>
<td>0.0197</td>
<td>0.01385</td>
<td>0.01520</td>
<td>0.0151</td>
<td>0.0164</td>
<td>0.0100</td>
<td>0.0100</td>
</tr>
</tbody>
</table>

**Table 3.2:** Running time in seconds of finding a shortest path from $p_1$ to $p_n$, and number of shortcuts and shortcut intervals for 10,000 points and various error bounds.

**Figure 3.7:** Number of shortcuts and shortcut intervals for 10,000 points and various error bounds.

**Figure 3.8:** Running time of finding a shortest path from $p_1$ to $p_n$ for 10,000 points and various error bounds.

shortcuts, consider Figure 3.9. Here we see the time spent per shortcut and the time spent per shortcut interval, as the error bound grows. In Figure 3.9a, we see that breadth-first search always spends roughly the same amount of time per shortcut. We observe that for range queries on shortcut intervals, the time spent per shortcut decreases as the error bound – and thus the density of the shortcut graph – grows.

In Figure 3.9b, we see that finding shortest paths using range queries spends constant time on every shortcut interval, regardless of the error bound. Breadth-first search on the other hand
Chapter 3. Exploiting Real-World Structures in the Shortcut Graph

Figure 3.9: Time spent per shortcut and per shortcut interval in microseconds for finding a shortest path from $p_1$ to $p_n$ for 10,000 points and various error bounds.

grows to be around 19 times slower per shortcut interval on a complete shortcut graph.

We conclude that shortcut interval sets yield significant benefits both in space- and time complexity for any error bound compared to explicit shortcut graphs. The compression is most effective for larger error bounds that yield dense shortcut graphs. However, for smaller error bounds, the running time of finding shortest paths is highly variable due to the unstable number of shortcut intervals. For highly sparse shortcut graphs, breadth-first search is faster than using range queries on shortcut interval sets, due to the small size of the shortcut intervals.
Chapter 4

Finding Shortcuts for Multiple Scales

Now that we have investigated how to represent the shortcut graph, let us focus on its construction. In this chapter, we will solely consider construction of shortcut graphs under the Hausdorff distance. As mentioned in Chapter 2, the first step of the computational framework by Imai and Iri [16] is to construct a shortcut graph for a polygonal curve $C$ using a single error bound $\varepsilon$. This problem can be described as follows:

**Shortcuts Single Scale**
- **Input:** A polygonal curve $C = \langle p_1, \ldots, p_n \rangle$ in $\mathbb{R}^2$ and an error bound $\varepsilon \geq 0$.
- **Output:** A shortcut graph $G(C, \varepsilon)$.

However, to facilitate progressive simplification, we wish to construct a sequence of shortcut graphs for a set of error bounds $E = \langle \varepsilon_1, \ldots, \varepsilon_m \rangle$. We call the corresponding problem **Shortcuts Multi-Scale**. A naive solution for this problem would be to use any algorithm for solving **Shortcuts Single Scale** $m$ times to find each shortcut graph independently. However, in the presence of many scales, such an independent construction is likely to cause overhead. We therefore instead determine for each shortcut the maximum error at which it is valid. This problem is defined as follows:

**Shortcuts All Scales**
- **Input:** A polygonal curve $C = \langle p_1, \ldots, p_n \rangle$ in $\mathbb{R}^2$.
- **Output:** A set $E_{\text{max}} = \{ D(\langle p_i, \ldots, p_j \rangle, \langle p_i, p_j \rangle) | 1 \leq i < j \leq n \}$.

We call $D((p_i, \ldots, p_j), (p_i, p_j))$ the shortcut error of $(p_i, p_j)$. The solution to this problem can be used to construct a shortcut graph for every error bound $\varepsilon_k \in E$. We do this by filtering the error values in $E_{\text{max}}$, which takes $O(1)$ time for each potential shortcut. We can thus solve **Shortcuts Multi-Scale** in $O(n^2m)$ time from $E_{\text{max}}$.

We first outline a widely used algorithm for solving **Shortcuts Single Scale** (Section 4.1). Next, we propose a new algorithm for solving **Shortcuts All Scales** (Section 4.2). Finally, we experimentally evaluate how these algorithms compare in practice (Section 4.3).

### 4.1 Independent Construction

A well known algorithm for the construction of a single shortcut graph using a fixed error bound under the Hausdorff distance was proposed by Chin and Chan [6]. This algorithm maintains a geometric data structure for every point $p_i \in C$ that is capable of sequentially determining for any point $p_j$ whether $(p_i, p_j)$ is a shortcut in $O(1)$. One can visualize this data structure as a pie wedge $w$ that has $p_i$ at its center. For every point $p_j$, $(p_i, p_j)$ is a shortcut if and only if $p_j$ is
inside this wedge. After determining this, we intersect the shape of \( w \) with the shape of pie wedge \( w_j \). The pie wedge \( w_j \) has \( p_i \) at its center, and is the smallest wedge that fully includes a circle with radius \( \varepsilon \) which center is located at \( p_j \). If the resulting intersection of \( w \) with \( w_j \) is an empty shape, we know that there is no shortcut \((p_i, p_k)\), where \( k > j \). An example of this algorithm is shown in Figure 4.1. Here, \((p_i, p_{i+1})\) and \((p_i, p_{i+3})\) are shortcuts, and \((p_i, p_{i+2})\) is not, as it lies outside of \( w_{i+1} \).

However, there is a degenerate case, which is shown in Figure 4.2. Here, we can see that \( p_{i+4} \) is inside the wedge \( w \). However, it is clear to see that \( \text{dist}(p_{i+3}, (p_i, p_{i+4})) \gg \varepsilon \), and thus using Lemma 2.1 we know that \((p_i, p_{i+4})\) is not actually a shortcut.

Chin and Chan [6] suggested these cases should be handled by first traversing \( C \) from start to end, producing for every point \( p_i \) all shortcuts \((p_i, p_j)\) as described above. Next, we do the same but for a reverse traversal of \( C \), producing for every point \( p_j \) all shortcuts \((p_i, p_j)\). Finally, these two sets of shortcuts are intersected, which yields the set of edges for shortcut graph \( G(C, \varepsilon) \). For the traversal of \( C \) in the forward direction, we provide pseudo code in Algorithm 4.1.

### Integration of shortcut intervals
Recall that in the worst case, there are \( \binom{n}{2} \) shortcuts. Therefore, intersecting two sets of shortcuts as described earlier takes \( O(n^2) \) time. We can optimize this by using shortcut interval sets, since intersecting any two intervals takes \( O(1) \) time, and we have \( O(n) \) shortcut intervals in practice. Thus, we can intersect two shortcut interval sets \( I'(C, \varepsilon) = (I'_1(\varepsilon), \ldots, I'_n(\varepsilon)) \) and \( I''(C, \varepsilon) = (I''_1(\varepsilon), \ldots, I''_n(\varepsilon)) \) in \( O(n) \) time by progressively scanning \( I'_i(\varepsilon) \) and \( I''_i(\varepsilon) \) for intersections in \( O(1) \) time for every \( p_i \in C \). An example of such an intersection of \( I_i(\varepsilon) \) and \( I'_i(\varepsilon) \) is shown in Figure 4.3. Note that in the worst case we have \( O(n^2) \) shortcut intervals, in which case the intersection takes \( O(n^2) \) time.
Algorithm 4.1 ChinChan($C, \varepsilon$)

1: for $i \leftarrow 1$ to $n - 1$ do
2: \hspace{1em} $w \leftarrow$ Full plane
3: \hspace{1em} for $j \leftarrow i + 1$ to $n$ do
4: \hspace{2em} if $p_j$ inside $w$ then
5: \hspace{3em} Add $(p_i, p_j)$ to $G(C, \varepsilon)$
6: \hspace{2em} end if
7: \hspace{1em} Intersect $w$ with $w_j$
8: \hspace{1em} if $w$ is empty then
9: \hspace{2em} break
10: \hspace{1em} end if
11: \hspace{1em} end for
12: \hspace{1em} end for
13: \hspace{1em} return $G(C, \varepsilon)$

Running time This algorithm spends $O(1)$ time for every shortcut. Therefore, it solves Shortcuts Single Scale in $O(n^2)$ time, since there is a worst case number of $\binom{n}{2}$ shortcuts.

This algorithm has excellent performance on most real-world data sets when $\varepsilon$ is small, because of its ability to prune significant portions of line segments whenever $w$ becomes empty. However, it cannot efficiently be extended to facilitate construction of shortcut graphs for multiple scales, since wedge $w$ can only be used to determine the validity of shortcuts for a single error bound.

4.2 Construction for Arbitrary Scale

Imai and Iri [16] used a brute-force approach of building the shortcut graph under the Hausdorff distance, which can be extended to solve Shortcuts All Scales. The algorithm computes the Hausdorff distance for every line segment $(p_i, p_j)$ where $j > i + 1$ by explicitly calculating the maximum distance from any point $p_k$ to $(p_i, p_j)$ for all $i < k < j$. Recall that we can calculate the Hausdorff distance this way, as proven by Lemma 2.1.

This algorithm spends $O(j - i)$ time for any line segment $(p_i, p_j)$ to determine its furthest point distance, where $1 \leq i < j \leq n$. Because there are $\binom{n}{2}$ such line segments, this algorithm solves Shortcuts All Scales with a running time of $O(n^3)$. In this section we discuss how we can improve the running time of this algorithm by determining the shortcut error of every line segment $(p_i, p_j)$ in $O(\log(j - i))$ time. We do this by maintaining a convex hull $CH \subseteq C$ for any $p_i \in C$ in which we incrementally insert all points $p_j \in C$ where $j > i$.

After inserting some point $p_j$, we find extreme points $X_{i,j}^U$ and $X_{i,j}^D$ on the convex hull using the upward normal $\vec{n}_{i,j}$ and downward normal $-\vec{n}_{i,j}$ of line segment $(p_i, p_j)$. An extreme point of a convex hull in the direction of some vector is defined as follows:
For any convex hull \( CH = \langle q_1, \ldots, q_t \rangle \), an extreme point of \( CH \) in the direction of a vector \( \vec{n} \) is any point \( x \in CH \) for which there is no point \( y \in CH \) such that \( \vec{n} \cdot (y - x) > 0 \).

An example of these extreme points is shown in Figure 4.4a. Here we see a line through \( X_t^{ij} \) parallel to \( \langle p_i, p_j \rangle \). The point \( X_t^{ij} \) is an extreme point for \( \vec{n}_{i,j} \), since all points in \( CH \) (and therefore all points in \( C \)) are below the line through \( X_t^{ij} \) with normal \( \vec{n}_{i,j} \).

Although finding both extreme points may seem sufficient to find the furthest point from a given line segment \( \langle p_i, p_j \rangle \), it is not when there are points in the subsequence \( \langle p_1, \ldots, p_j \rangle \) that lie to the left of \( p_i \) or to the right of \( p_j \) within the reference frame of \( \langle p_i, p_j \rangle \). This is illustrated in Figure 4.4b, where we separate the area around the line segment \( \langle p_i, p_j \rangle \) into regions: \( T \) (top), \( B \) (bottom), \( L \) (left) and \( R \) (right). As shown, the extreme points of \( \langle p_i, p_j \rangle \) are \( X_t^{ij} \) and \( X_b^{ij} \), but \( X_t^{ij} \) and \( X_b^{ij} \) are furthest away from \( \langle p_i, p_j \rangle \). Using extreme point queries, we therefore only find the furthest points in regions \( T \) and \( B \). In order to expand the search area to \( R \) and \( L \), we need a technique for finding the furthest point \( X_t^{ij} \) from \( p_i \) in \( L \), and the furthest point \( X_b^{ij} \) from \( p_j \) in \( R \). This gives us Algorithm 4.2.

**Algorithm 4.2 ShortcutsConvexHull(\( C \))**

```
1: for \( i \leftarrow 1 \) to \( n - 1 \) do
2:   \( CH \leftarrow \) Empty convex hull
3:   Insert \( p_i \) into \( CH \)
4: for \( j \leftarrow i + 1 \) to \( n \) do
5:   Insert \( p_j \) into \( CH \)
6:   \( X_t^{ij} \leftarrow \) Extreme point of \( CH \) in the direction \( \vec{n}_{i,j} \)
7:   \( X_b^{ij} \leftarrow \) Extreme point of \( CH \) in the direction \( -\vec{n}_{i,j} \)
8:   \( X_t^{ij} \leftarrow \) Furthest point from \( p_i \) in \( CH \) in region \( L \) of \( \langle p_i, p_j \rangle \)
9:   \( X_b^{ij} \leftarrow \) Furthest point from \( p_j \) in \( CH \) in region \( R \) of \( \langle p_i, p_j \rangle \)
10: \( E_{\text{max}}[i][j] \leftarrow \) max[dist(\( X_t^{ij} \), \( p_i \)), dist(\( X_b^{ij} \), \( p_i \)), dist(\( X_t^{ij} \), \( p_j \)), dist(\( X_b^{ij} \), \( p_j \))]
11: end for
12: end for
13: return \( E_{\text{max}} \)
```

In order to obtain the desired running time of \( O(\log(j - i)) \) for any line segment \( \langle p_i, p_j \rangle \), we need to be able to do all queries and insertions in \( CH \) in \( O(\log(j - i)) \) time. To achieve this, we separate the convex hull into a top hull \( CH_t \) and a bottom hull \( CH_b \) and represent each using a balanced binary search tree without duplicates ordered by the \( x \)-coordinates of its points. This allows for finding \( X_t^{ij} \) and \( X_b^{ij} \) using a binary search on \( CH_t \) and \( CH_b \) respectively. The insertion
of any point \( p_j \) into \( CH \) is performed by an insertion of \( p_j \) in both \( CH_b \) and \( CH_t \). Finally, we annotate the nodes of the search trees associated with \( CH_t \) and \( CH_b \), and perform a customized range query on these trees to obtain \( X^{ij}_t \) and \( X^{ij}_b \). Hence, \( X^{ij}_t \) is selected from two furthest point candidates \( X^{ij}_t \) and \( X^{ij}_b \) from \( CH_t \) and \( CH_b \) respectively. Similarly, \( X^{ij}_b \) is obtained from two candidates \( X^{ij}_t \) and \( X^{ij}_b \).

We now discuss the details of each operation. We limit our discussion to the top hull \( CH_t = (q_1, \ldots, q_r) \), since all operations on \( CH_b \) are analogous.

### 4.2.1 Operations

Inserting \( p_j \) Insertion into the top hull \( CH_t \) of a point \( p_j \) is performed by finding the left-most point \( q_l \in CH_t \), and the right-most point \( q_r \in CH_t \) such that the line segments \((q_l, p_j)\) and \((q_r, p_j)\) do not pass through \( CH_t \). After we find these two points, we remove all points on the hull in the subsequence \((q_{l+1}, \ldots, q_{r-1})\) and insert \( p_j \) in between \( q_l \) and \( q_r \). Note how this extends the area of the convex hull while maintaining its convexity.

We can find \( q_l \) and \( q_r \) by doing a binary search on the upward normals of the line segments of \( CH_t \). For example, the binary search for \( q_r \) works by finding the right-most point so that the angle of \( \bar{n}_{r,j} \) is in between the angles of \( \bar{n}_{r-1,r} \) and \( \bar{n}_{r,r+1} \). Since the hull is convex, as \( r \) increases, \( \bar{n}_{r-1,r} \) and \( \bar{n}_{r,r+1} \) rotate monotonically clockwise, and \( \bar{n}_{r,j} \) rotates monotonically counter-clockwise. At some point the angles of these vectors must converge, and thus a binary search will succeed if \( q_r \) exists.

An example of inserting \( p_j \) into \( CH_t \) is shown in Figure 4.5a. Note how the point between \( q_l \) and \( q_r \) is removed as a result of the insertion and how the convexity of \( CH_t \) is preserved.

If neither \( q_l \) nor \( q_r \) is found during the binary search, this implies that \( p_j \) must lie inside the convex hull, in which case we do not insert \( p_j \). If only \( q_l \) was found, we remove all point on the hull after \( q_l \) and insert \( p_j \) at the end. We do the reverse when only \( q_r \) is found.

![Figure 4.5: Binary search on the convex hull using normals.](image)

Finding \( X^{ij}_t \) and \( X^{ij}_b \) The extreme point queries in \( T \) and \( B \) are similar to insertions, and also perform a binary search on the normals of the line segments of the convex hull. Specifically, we find \( X^{ij}_t \) in \( CH_t \) using upward normal \( \bar{n}_{i,j} \) of line segment \((p_i, p_j)\), and we find \( X^{ij}_b \) in \( CH_b \) using downward normal \(-\bar{n}_{i,j}\). This is illustrated for \( X^{ij}_t \) in \( CH_t \) in Figure 4.5b.

Finding \( X^{ij}_t \) In order to efficiently determine \( X^{ij}_t \), we annotate the nodes of the binary search tree of \( CH_t \) and \( CH_b \). We annotate any node \( q_x \in CH_t \) rooted at subtree \( T_x \) with the furthest point from \( p_i \) in \( T_x \). An example of such an annotation of the binary search tree of \( CH_t \) is shown in Figure 4.6. We now discuss how we can find \( X^{ij}_t \) in \( CH_t \).
Chapter 4. Finding Shortcuts for Multiple Scales

By annotating the search tree of $CH_t$ in this manner, we are able to find $X_{ij}^{kl}$ using a customized range query. We do this by traversing the binary search tree and finding all nodes for which all descendant nodes correspond to points that lie inside $L$. We then use the annotated point of each such node as a candidate for furthest point from $p_i$.

We start at the root of the tree, and for every node $q_x$ annotated with point $q_a$ rooted at subtree $T_x = \langle q_l, \ldots, q_r \rangle$, we do the following:

- If both $q_l$ and $q_r$ lie inside $L$, consider subtree annotation $q_a$ as a furthest point candidate.
- Otherwise:
  - If $q_x$ lies inside $L$, consider $q_x$ as a furthest point candidate.
  - If either $q_l$ or $q_r$ lies inside $L$, traverse to both children of $q_x$.

Note that we only check whether left-most point $q_l$ in $T_x$, right-most point $q_r$ in $T_x$, and $q_z$ lie inside $L$. This is an $O(1)$ operation. The intuition is that in most cases, if the start and end of a subsequence of points on the hull are inside $L$, then so are all other points in that subsequence, due to the convexity of the hull. However, this is not always the case when $p_i$ lies horizontally between $q_l$ and $q_r$. In this scenario, we need to be aware of the following two degenerate cases:

- Figure 4.7a: neither $q_l$ nor $q_r$ lies inside $L$, and $p_j$ lies below $p_i$. Only $q_x$ is considered for furthest point from $p_i$. However, the furthest point from $p_i$ in $L$ may be some other point in $T_x$ and is therefore potentially missed.
- Figure 4.7b: both $q_l$ and $q_r$ lie inside $L$, and $p_j$ lies above $p_i$. We obtain the annotation of $q_x$, which is the furthest point from $p_i$ in $T_x$. However, not all points in $T_x$ lie inside $L$.

We handle both cases by considering the node annotation of $q_x$ and traversing to both children.
Finding $X_{ij}^r$ Note that we traverse $C$ from start to end, and can therefore not directly apply the same approach for finding $X_{ij}^l$ to find $X_{ij}^r$. We instead apply the same technique used by ChinChan (Algorithm 4.1), which is to run the algorithm a second time using a reversed traversal of $C$. This means we defer the calculation of the shortcut error until after all furthest point candidates are found.

During the forward traversal of $C$, we obtain $X_{ij}^t$, $X_{ij}^b$ and $X_{ij}^l$ at every iteration, whereas during the reverse traversal we only obtain $X_{ij}^r$ at every iteration. After all candidates are found for every line segment, we compute the shortcut error as stated on line 10 of ShortcutsConvexHull.

4.2.2 Running Time

Note how the annotation of the convex hull is much like the annotation of the binary search tree used for finding shortest paths in shortcut interval sets (see Section 3.2). Therefore, the same arguments about running time apply. Because both $CH_t$ and $CH_b$ have at most $O(j - i)$ points when inserting $p_j$, all binary search tree operations run in $O(\log(j - i))$ time.

Due to the convexity of the hull, all candidates for furthest point from $p_i$ in $L$ or from $p_j$ in $R$ lie on at most one subsequence of points on the convex hull, as shown in Figure 4.8a. The degenerate case shown in Figure 1.7b is the only exception, where there are candidates in two subsequences. However, these subsequences are on both extremes of the tree, as illustrated in Figure 4.8b. For this case we therefore also traverse the children of at most two nodes at each level of the tree. Therefore, the range queries have a time complexity of $O(\log(j - i))$.

We conclude ShortcutsConvexHull solves Shortcuts All Scales with a running time of $O(n^2 \log n)$.

**Figure 4.8:** Subtrees of $CH_t$ containing closest point candidates for $p_i$ in $L$.

4.2.3 Correctness

Consider we have a line segment $(p_i, p_j) \in C$ where $j > i$, and that $CH_t$ and $CH_b$ are the top and bottom parts of the convex hull obtained by incrementally inserting $p_i$ up to $p_j$. We first prove that the furthest point from $(p_i, p_j)$ in $(p_i, \ldots, p_j)$ can be found in $CH_t$ or $CH_b$.

**Lemma 4.1.** If $p_k$ is the furthest point from $(p_i, p_j)$ in the subsequence $(p_i, \ldots, p_j)$, then $p_k \in CH_t \cup CH_b$.

**Proof.** Assume $p_k \notin CH_t \cup CH_b$. Because $CH_t$ and $CH_b$ are convex and together form a single convex shape, we know that $p_k$ must lie strictly below all line segments of $CH_t$ and strictly above all line segments of $CH_b$. 


Consider the case where the line segment from the closest point from \( p_k \) on \((p_i, p_j)\) to \( p_k \) goes upwards. We know that by extending this line, it intersects \( CH_t \) in some line segment \((q_x, q_{x+1})\), as sketched in Figure 4.9. Note that either \( q_x \) or \( q_{x+1} \) must be further away from \((p_i, p_j)\) than \( p_k \), regardless of the geometric configuration. Thus, \( p_k \) is not actually the furthest point from \((p_i, p_j)\) in \((p_i, \ldots, p_j)\). By contradiction, we obtain that in this case, \( p_k \in CH_t \cup CH_b \).

An analogous argument can be given for the case where the perpendicular line from \((p_i, p_j)\) to \( p_k \) goes downwards through \( CH_b \). We conclude by contradiction that \( p_k \in CH_t \cup CH_b \) always holds.

Let us prove that among all points on \( CH_t \) and \( CH_b \), the set of candidates for the furthest point from \((p_i, p_j)\) found by SHORTCUTSCONVEXHULL contains the actual furthest point in the hull from \((p_i, p_j)\). We do this by proving that the algorithm identifies the furthest point in each region of \((p_i, p_j)\) (see Figure 4.4b). Let us start with region \( T \).

**Lemma 4.2.** \( X^{ij}_t \) is the furthest point from \((p_i, p_j)\) in \( CH_t \) in region \( T \) of \((p_i, p_j)\).

**Proof.** Let us define \( X^{ij}_t = q_x \). Assume \( q_x \) is not the furthest point in \( CH_t \) from \((p_i, p_j)\) in region \( T \) of \((p_i, p_j)\).

Therefore, there is a point \( q_y \) on the hull that lies beyond the line through \( q_x \) with normal \( \vec{n}_{i,j} \).

Due to the convexity of the hull, the line must pass through the hull. Since the normals of the line segments on the hull rotate monotonically, the angle of \( \vec{n}_{i,j} \) does not lie in between the angles of \( \vec{n}_{x-1,x} \) and \( \vec{n}_{x,x+1} \). This implies \( q_x \) is not an extreme point, as is illustrated in Figure 4.10a.

By contradiction, we conclude that \( q_x = X^{ij}_t \) must be the furthest point in \( CH_t \) from \((p_i, p_j)\) in region \( T \) of \((p_i, p_j)\).

An analogous proof can be given to show that \( X^{ij}_b \) is the furthest point from \((p_i, p_j)\) in \( CH_b \) in region \( B \). We continue with region \( L \).

**Lemma 4.3.** \( X^{ij}_i \) is the furthest point from \( p_i \) in \( CH_t \) in region \( L \) of \((p_i, p_j)\).

**Proof.** Assume the furthest point from \( p_i \) in \( CH_t \) in region \( L \) of \((p_i, p_j)\) is not \( X^{ij}_i \), but rather some other point \( q_a \in CH_t \).

We know that \( q_a \) is not visited, because otherwise we would have \( q_a = X^{ij}_i \) by checking its annotation, yielding a contradiction. Therefore, at some ancestor \( q_x \) of \( q_a \), we did not continue on a path down to \( q_a \). We further know that the annotation of \( q_x \) is not considered, because otherwise \( q_a = X^{ij}_i \), yielding the same contradiction.

![Figure 4.9: Furthest point \( p_k \not\in CH_t \).](image)

![Figure 4.10: Proving completeness of the furthest point candidates found by Algorithm 4.2](image)
Thus, both the left-most point \( q_l \) and right-most point \( q_r \) of the subtree \( T_x \) rooted at \( p_x \) must lie outside of \( L \), and \( p_i \) does not lie in between \( q_l \) and \( q_r \). This situation is sketched in Figure 4.10b. Because \( CH_t \) is convex, all points in \( T_x \) must lie outside of \( L \). This implies \( q_a \) lies outside of \( L \), since \( q_a \) is a descendant of \( q_x \), and must therefore be in \( T_x \).

By contradiction, we conclude \( X_{ij}^{tl} \) must be the furthest point in \( CH_t \) from \( p_i \) in region \( L \) of \( (p_i, p_j) \). □

Analogous proofs can be given to prove analogous claims for \( X_{ij}^{bl}, X_{ij}^{tr} \) and \( X_{ij}^{br} \).

**Theorem 4.4.** Given a polygonal curve \( C = \langle p_1, \ldots, p_n \rangle \), we can compute \( H(\langle p_i, \ldots, p_j \rangle, \langle p_i, p_j \rangle) \) for all \( 1 \leq i < j \leq n \) in \( O(n^2 \log n) \).

**Proof.** From Lemma 4.2 and Lemma 4.3 we deduce that the furthest point from \( (p_i, p_j) \) in \( CH_t \cup CH_b \) must be found by \( \text{ShortcutConvexHull} \). Using Lemma 4.1, it follows that this furthest point from \( (p_i, p_j) \) in \( CH_t \cup CH_b \) is the furthest point from \( (p_i, p_j) \) in \( \langle p_i, \ldots, p_j \rangle \). We therefore obtain \( E_{max}[i][j] = \max_{p_k \in \langle p_i, \ldots, p_j \rangle} \text{dist}(p_k, (p_i, p_j)) \). Correctness follows by Lemma 2.1. □

### 4.3 Experimental Evaluation

In Section 4.1 we discussed how shortcut interval sets can be exploited to speed up construction of the shortcut graph using CHINCHAN. In this section, we aim to investigate how this technique performs in practice, with regards to the length of the input curve and the error bound. Furthermore, we explore how well \( \text{ShortcutConvexHull} \) performs at constructing shortcut graphs for multiple scales.

For the experimental setup, refer to Section 3.3.

#### 4.3.1 Construction of Shortcut Interval Sets

In this section, we evaluate how the representation of the shortcut graph influences its construction using CHINCHAN. We start with determining how the length of the input curves influences the running time. We fix the shortcut graph density to 25% by choosing the error bound for every input such that the resulting shortcut graph includes the shortcuts with the 25% smallest errors. The results are listed in Table 4.1.

<table>
<thead>
<tr>
<th>Length of the input curve</th>
<th>500</th>
<th>1000</th>
<th>1500</th>
<th>2500</th>
<th>3500</th>
<th>4250</th>
<th>5000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit Shortcut Graph (sec.)</td>
<td>0.329</td>
<td>1.267</td>
<td>3.052</td>
<td>8.278</td>
<td>16.125</td>
<td>23.511</td>
<td>32.705</td>
</tr>
<tr>
<td>Shortcut Interval Set (sec.)</td>
<td>0.152</td>
<td>0.541</td>
<td>1.247</td>
<td>3.270</td>
<td>5.954</td>
<td>8.703</td>
<td>13.030</td>
</tr>
<tr>
<td>Shortcuts (( \times 10^4 ))</td>
<td>3.169</td>
<td>12.587</td>
<td>28.256</td>
<td>78.344</td>
<td>154.43</td>
<td>226.15</td>
<td>312.94</td>
</tr>
<tr>
<td>Shortcut Intervals (( \times 10^4 ))</td>
<td>0.084</td>
<td>0.190</td>
<td>0.242</td>
<td>0.373</td>
<td>0.659</td>
<td>0.868</td>
<td>0.816</td>
</tr>
</tbody>
</table>

**Table 4.1:** Running time in seconds of CHINCHAN and the associated number of shortcuts and shortcut intervals for constructing a shortcut graph with 25% density for various lengths of the input curve.

Plotting the running time results against the length of the input curve gives us Figure 4.11. We observe that construction of shortcut interval sets consistently exhibits about 40% of the growth in running time given by the construction of explicit shortcut graphs. Recall that CHINCHAN
constructs explicit shortcut graphs by intersecting two sets of shortcuts. To facilitate this intersection, each of these sets must maintain an index of the shortcuts it contains. No such index needs to be maintained by shortcut interval sets. Thus, insertion of new shortcuts into shortcut interval sets is faster than inserting them into sets that explicitly store shortcuts. The running time improvements are therefore not solely caused by the fast intersection of intervals discussed in Section 4.1.

**Figure 4.11:** Running time in seconds of ChinChan with explicit shortcut graphs versus shortcut interval sets for various lengths of the input curve, and 25% density of the shortcut graph.

Next, we investigate how the density of the shortcut graph influences the construction time. For this, we vary the error bound while fixing the length of the input curve to 3000. The results obtained are given in Table 4.2. Note that the last error bound \( \varepsilon = 0.085 \) yields a complete shortcut graph.

<table>
<thead>
<tr>
<th>Error bound</th>
<th>0.001</th>
<th>0.005</th>
<th>0.01</th>
<th>0.03</th>
<th>0.05</th>
<th>0.07</th>
<th>0.085</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit Shortcut Graph (sec.)</td>
<td>2.375</td>
<td>11.299</td>
<td>16.881</td>
<td>27.305</td>
<td>37.768</td>
<td>44.705</td>
<td>47.326</td>
</tr>
<tr>
<td>Shortcut Interval Set (sec.)</td>
<td>0.945</td>
<td>4.083</td>
<td>5.887</td>
<td>8.929</td>
<td>13.876</td>
<td>15.154</td>
<td>14.101</td>
</tr>
<tr>
<td>Shortcuts ( \times 10^4 )</td>
<td>20.698</td>
<td>99.908</td>
<td>161.82</td>
<td>264.17</td>
<td>346.68</td>
<td>430.044</td>
<td>449.85</td>
</tr>
<tr>
<td>Shortcut Intervals ( \times 10^4 )</td>
<td>0.499</td>
<td>0.458</td>
<td>0.545</td>
<td>0.458</td>
<td>0.503</td>
<td>0.369</td>
<td>0.300</td>
</tr>
</tbody>
</table>

**Table 4.2:** Running time in seconds of ChinChan for both shortcut graph representations and the associated number of shortcuts and shortcut intervals for various error bounds and 3000 points.

A plot of these results is given in Figure 4.12. As in Figure 4.11, the construction of shortcut interval sets shows a growth in running time with respect to the error bound that is on average 40% that of the growth exhibited by explicit shortcut graphs.

This relationship on the time complexity is further illustrated by Figure 4.13a where the time spent per shortcut is plotted against the error bound. We observe that the time spent per shortcut for both representations remains stable as the error bound grows.

Figure 4.13a shows a similar plot of the time spent per shortcut interval. We can see how the time spent per shortcut interval non-monotonically increases as the error bound grows. This is due to the non-monotonic growth of the number of shortcut intervals. As can be seen in Table 4.2, the number of shortcut intervals sporadically increases and decreases as the density of the shortcut graph grows.
4.3. Experimental Evaluation

Figure 4.12: Running time in seconds of ChinChan with explicit shortcut graphs versus shortcut interval sets for various error bounds on 3000 points.

Figure 4.13: Time spent per shortcut and per shortcut interval in microseconds by ChinChan with explicit shortcut graphs versus shortcut interval sets for various error bounds and 3000 points.

We conclude that ChinChan is most efficient using shortcut interval sets, yielding significant running time improvements for large input curves. Recall that in Section 3.3 we concluded that shortcut interval sets in practice also yield significant benefits in terms of memory usage, and running time for finding shortest paths. Shortcut interval sets thus yield benefits in each computational aspect of the min-# simplification framework [16].

4.3.2 Construction for Multiple Scales

We described in Section 4.1 how ChinChan has a worst case running time of $O(n^2 m)$. Furthermore, in Section 4.2.2 we showed that integrated construction of $m$ shortcut graphs using ShortcutsConvexHull takes $O(n^2 \log n + n^2 m)$ time. In this section we investigate how these terms relate to running-time performance in practice, and at what number of scales the initial time investment of $O(n^2 \log n)$ spent by ShortcutsConvexHull to compute all shortcut errors amortizes to be more efficient than ChinChan.

Like the binary search tree used for finding shortest paths in shortcut interval sets (see Section 3.2), we use left-leaning red-black trees [22] to represent the convex hulls $CH_t$ and $CH_b$ of ShortcutsConvexHull. Furthermore, we use the most efficient implementation of ChinChan using shortcut interval sets.

We sample the error bounds from the distribution of shortcut errors of $C$. We first obtain all shortcut errors using any algorithm to solve Shortcuts All Scales. Next, we sort the errors and
linearly sample \( m \) error bounds. This yields a shortcut graph \( G(C, \varepsilon_m) \) with a density of 100%. By linearly sampling, the density of the shortcut graphs grows linearly as the scale increases. Recall that CHINCHAN performs well for small error bounds, and poorly for large error bounds. By sampling up to an error bound that yields a complete shortcut graph, we thus exhaustively capture the strengths and weaknesses of CHINCHAN.

<table>
<thead>
<tr>
<th>Number of scales</th>
<th>1</th>
<th>5</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>60</th>
<th>80</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convex hulls</td>
<td>285.41</td>
<td>300.27</td>
<td>317.93</td>
<td>335.46</td>
<td>368.20</td>
<td>397.67</td>
<td>455.58</td>
<td>528.11</td>
<td>594.19</td>
</tr>
<tr>
<td>Chin-Chan</td>
<td>10.45</td>
<td>45.59</td>
<td>77.59</td>
<td>146.84</td>
<td>216.09</td>
<td>288.04</td>
<td>426.75</td>
<td>579.63</td>
<td>724.74</td>
</tr>
</tbody>
</table>

Table 4.3: Running time in seconds for constructing multiple shortcut interval sets using independent construction by CHINCHAN versus integrated construction by SHORTCUTSCONVEXHULL on 3500 points, with error bounds linearly sampled from the smallest to the largest shortcut error.

The running-time measurement are listed in Table 4.3. The corresponding plot is given in Figure 4.14. We observe that the running time of both algorithms is linear in the number of scales. For the construction of a single shortcut graph, SHORTCUTSCONVEXHULL is 27 times slower than CHINCHAN. However, CHINCHAN spends on average 7.2 seconds for every additional error bound, whereas SHORTCUTSCONVEXHULL spends on average only 3.1 seconds. Therefore, for every additional scale, SHORTCUTSCONVEXHULL is over twice as efficient as CHINCHAN. Beyond 65 error bounds, SHORTCUTSCONVEXHULL is the preferred construction algorithm in terms of running time.

We conclude that SHORTCUTSCONVEXHULL is only preferable to CHINCHAN when constructing a large number of shortcut graphs. Its applications are therefore limited to simplification for many levels of detail, such as interactive visualizations with many zoom levels.

Figure 4.14: Running time in seconds of constructing multiple shortcut interval sets using CHINCHAN versus SHORTCUTSCONVEXHULL on 3500 points for various numbers of scales.
Chapter 5

Progressive Simplification of Polygonal Curves

In this chapter we discuss two approaches for solving Progressive Min-#. We propose a progressive simplification algorithm that minimizes the number of points across all scales (Section 5.1). Next, we present two greedy algorithms with better running times that do not provide these guarantees on the simplification size, and show how existing non-progressive algorithms can be extended to greedily compute progressive simplifications (Section 5.2). Finally, we experimentally evaluate how both of these approaches compare, and how they compare to a minimal non-progressive simplification algorithm (Section 5.3).

5.1 An Optimal Algorithm

Recall that a progressive simplification is defined by a set of simplifications for various error bounds on which a monotonic containment relation exists. This means that for any two simplifications $S_\ell$ and $S_k$ where $1 \leq \ell \leq k$, $S_\ell$ must include all points of $S_k$. Including a shortcut $(p_i, p_j)$ in any simplification $S_k$ thus has an impact on the structure of all simplifications at a lower scale. Therefore, a global minimum has to be computed on the cumulative size of the simplifications.

We compute this global minimum by associating a cost with including any shortcut in any simplification. Specifically, we have an integer cost $c_{ij}^k \in \mathbb{N}$ associated with including any shortcut $(p_i, p_j)$ from the shortcut graph $G(C, \varepsilon_k)$ in the simplification $S_k$. This integer cost represents the minimal combined number of shortcuts $(p_x, p_y) \in \langle p_i, \ldots, p_j \rangle$ that will have to be added to any $S_\ell$ where $1 \leq \ell \leq k$, if we include $(p_i, p_j)$ in $S_k$. This gives us the following cost function:

$$
c_{ij}^k = \begin{cases} 
1 & \text{if } (p_i, p_j) \in G(C, \varepsilon_k) \wedge k = 1 \\
1 + \min_{\pi \in \prod_{j=1}^{k-1} (p_x, p_y) \in \pi} c_{xy}^{k-1} & \text{if } (p_i, p_j) \in G(C, \varepsilon_k) \wedge k > 1 \\
\infty & \text{otherwise}
\end{cases}
$$

We use $\prod_{j=1}^{k-1}$ to denote the set of all paths in $G(C, \varepsilon_k)$ from $p_i$ to $p_j$.

Using the costs for scale $m$, we can find $S_m$ such that the sum of all costs associated with its shortcuts is minimized. Then for any shortcut $(p_i, p_j) \in S_m$, we can build a simplification for $(p_i, \ldots, p_j)$ in $S_{m-1}$ using the costs for scale $m-1$. By chaining all these simplifications together, we obtain $S_{m-1}$. We continue doing this until we have found $S_1$.

Determining the cost values as described earlier is possible by adding the costs as weights to
Chapter 5. Progressive Simplification of Polygonal Curves

Figure 5.1: Shortcut graphs weighted by cost, and a minimal progressive simplification.

the shortcut graph at every scale. Specifically, any edge \((p_i, p_j)\) in shortcut graph \(G(C, \varepsilon_k)\) has weight \(c_{ij}^k\). The weight of a shortcut \(c_{ij}^k\) can then be retrieved by finding a path \(\pi\) in \(G(C, \varepsilon_{k-1})\) from \(p_i\) to \(p_j\) such that \(\sum_{(p_x, p_y) \in \pi} c_{xy}^{k-1}\) is minimized. Similar minimum-weight paths are used to construct the simplifications. In Figure 5.1, a complete example is illustrated which shows how the weights are determined, and how the simplifications are constructed.

From this figure, we observe that for any shortcut \((p_i, p_j)\) at scale \(k\), if \((p_i, p_j) \in G(C, \varepsilon_{k-1})\), we have \(c_{ij}^k = c_{ij}^{k-1} + 1\). Therefore, if a shortcut at scale \(k\) is present at scale \(k-1\), we can reuse its weight by incrementing it by one. We can prove this as follows:

**Lemma 5.1.** For any \(1 < k \leq m\) and \(n \geq j > i \geq 1\), if \((p_i, p_j) \in G(C, \varepsilon_{k-1})\), then \(c_{ij}^k = c_{ij}^{k-1} + 1\).

**Proof.** Assume \((p_i, p_j) \in G(C, \varepsilon_{k-1})\). Note that \(G(C, \varepsilon_k) \subseteq G(C, \varepsilon_{k-1})\) for all \(1 \leq k < \ell \leq m\), because \(\varepsilon_{\ell} \geq \varepsilon_k\). Thus, for all \(1 \leq k < \ell \leq m\), any shortcut in \(G(C, \varepsilon_k)\) must also be present in \(G(C, \varepsilon_{\ell})\). Because \((p_i, p_j) \in G(C, \varepsilon_{k-1})\), this implies that \((p_i, p_j) \in G(C, \varepsilon_k)\).

**Case \(k = 2\):** We prove \(c_{ij}^2 = c_{ij}^1 + 1\) as follows:

\[
  c_{ij}^2 = 1 + \min_{\pi \in \prod_{ij}} \sum_{(p_x, p_y) \in \pi} c_{xy}^1
\]

\[
  = 1 + \min_{\pi \in \prod_{ij}} \sum_{(p_x, p_y) \in \pi} 1
\]

\[
  = 1 + \min_{\pi \in \prod_{ij}} |\pi|
\]

\[
  = 1 + 1
\]

\[
  = c_{ij}^1 + 1
\]

**Case \(k > 2\):** We prove \(c_{ij}^k = c_{ij}^{k-1} + 1\) by showing that both the upper and lower bound of \(c_{ij}^k\) is equal to \(c_{ij}^{k-1} + 1\). Let us start with the upper bound.
5.1. An Optimal Algorithm

Now let us prove the lower bound.

\[ c_{ij}^k = 1 + \min_{\pi \in \prod^{k-1}_{ij}} \sum_{(p_x, p_y) \in \pi} c_{xy}^{k-1} + 1 \]

\[ \leq \sum_{(p_x, p_y) \in \{(p_i, p_j) \}} c_{xy}^{k-1} + 1 \quad (p_i, p_j) \in G(\mathcal{C}, \varepsilon_{k-1}) \]

\[ = c_{ij}^{k-1} + 1 \]

The last inequality holds, since a path from \( p_i \) to \( p_j \) constructed by concatenating several shortest paths is always at least as long as a single shortest path from \( p_i \) to \( p_j \).

We may now conclude that for all \( 1 < k \leq m \) and \( n \geq j > i \geq 1 \), we have \( c_{ij}^k = c_{ij}^{k-1} + 1 \) if \((p_i, p_j) \in G(\mathcal{C}, \varepsilon_{k-1})\). \( \square \)

We can use this result to efficiently assign a weight to any shortcut that was encountered at an earlier scale, without recomputing a shortest path. The pseudo-code of the resulting algorithm is given in Algorithm 5.1.

### Weighted Simplifications

Recall that for Weighted Progressive MIN-#, we are given positive weights \( W = \langle w_1, \ldots, w_m \rangle \) in \( \mathbb{R} \) and need to minimize \( \sum_{k=1}^{m} w_k |S_k| \). We can trivially extend the cost function \( c_{ij}^k \) to facilitate solving this problem. Specifically, we assign a cost of \( w_k \) (instead of 1) to every line segment in any simplification \( S_k \), as follows:

\[
  c_{ij}^k = \begin{cases} 
    w_k & \text{if } (p_i, p_j) \in G(\mathcal{C}, \varepsilon_k) \land k = 1 \\
    w_k + \min_{\pi \in \prod^{k-1}_{ij}} \sum_{(p_x, p_y) \in \pi} c_{xy}^{k-1} & \text{if } (p_i, p_j) \in G(\mathcal{C}, \varepsilon_k) \land k > 1 \\
    \infty & \text{otherwise}
  \end{cases}
\]

#### 5.1.1 Optimization

In this section we explore (potential) techniques for optimizing the running time of ImaiIriProgressive.

**Determining shortcut cost** A naive technique for finding a shortest path on line 9 is to run any single-source shortest path algorithm for every shortcut \((p_i, p_j)\). Because a shortcut graph contains at most \( \binom{n}{2} \) shortcuts, any such algorithm would run in at least \( O(n^2) \) time. However, this approach potentially discards shortest path information that can be reused.
Algorithm 5.1 ImaiIriProgressive($C, \mathcal{E}$)

1: $G(C, \epsilon_1), \ldots, G(C, \epsilon_m) \leftarrow \text{Shortcuts Multi-Scale}(C, \mathcal{E})$
2: for $k \leftarrow 1$ to $m$ do
3:  for each $(p_i, p_j)$ in $G(C, \epsilon_k)$ do
4:  if $k = 1$ then
5:      $c_{ij}^1 \leftarrow 1$
6:  else if $(p_i, p_j) \in G(C, \epsilon_{k-1})$ then
7:      $c_{ij}^k \leftarrow 1 + c_{ij}^{k-1}$
8:  else
9:      $c_{ij}^k \leftarrow 1 + \text{weight of the shortest path from } p_i \text{ to } p_j \text{ in } G(C, \epsilon_{k-1})$
10: end if
11: end for
12: end for
13: for $k \leftarrow m$ to 1 do
14:  if $k = m$ then
15:      $S_k \leftarrow \text{shortest path from } p_1 \text{ to } p_n \text{ in } G(C, \epsilon_k)$
16:  else
17:      for each $(p_i, p_j)$ in $S_{k+1}$ do
18:         $S_k \leftarrow S_k \cup \text{shortest path from } p_i \text{ to } p_j \text{ in } G(C, \epsilon_k)$
19:     end for
20: end if
21: end for
22: return $\langle S_1, \ldots, S_m \rangle$

Dijkstra’s algorithm [9] takes any source node $u$ in a graph $G = (V, E)$ and builds a data structure in $O(|V| \log |V| + |E|)$ time which holds the shortest path distance from $u$ to any target node $v$. This data structure can be used to construct any of these shortest paths in $O(|V|)$ time. If we run Dijkstra’s algorithm every time we reach line 9, we obtain an amortized running time of $O(n^3)$. This is the case since we need to find a weight for a worst case total of $O(n^2)$ shortcuts.

Now consider we instead build this data structure for every scale $k$ only once for every $p_i$ for which there exists some shortcut $(p_i, p_j) \in G(C, \epsilon_k)$ where $(p_i, p_j) \notin G(C, \epsilon_{k-1})$. This yields $O(n)$ data structures from which the shortest path for any shortcut at scale $k$ can be obtained in $O(n)$ time. Constructing each data structure takes $O(n^2)$ time, and thus we have a total worse-case running time of $O(n^3m)$.

Integration of shortcut intervals

Note that the algorithm to find shortest paths in shortcut interval sets presented in Section 3.2 is not limited to unweighted shortcuts, and can trivially be extended to find paths with lowest weight. We assign a weight $w$ to every shortcut interval $[x, y] \in I_k(\epsilon)$ for some point $p_i$, such that every shortcut $(p_i, p_j)$ has weight $w$ where $x \leq j \leq y$.

By weighting the intervals, we potentially fragment an unweighted interval into many weighted intervals. Therefore, to effectively employ weighted shortcut interval sets, we need most consecutive shortcuts $(p_i, p_j)$ and $(p_i, p_{j+1})$ to share the same weight.

In Figure 5.2 we see a heat map of the costs of shortcuts in $G(C, \epsilon_m)$ given by ImaiIriProgressive in practice. The rows and columns correspond to the points in $C$. Each cell $i, j$ is colored according to the cost $c_{ij}^m$ of

Figure 5.2: Heat map of the shortcut costs in $G(C, \epsilon_m)$. From low to high cost: black, red, yellow, white.
5.1. An Optimal Algorithm

shortcut \((p_i, p_j)\). Although we observe many regions with a similar color, analysis shows that almost all consecutive shortcuts have a slightly different cost. Therefore, by using weighted shortcut interval sets in IMAIIRIPROGRESSIVE using the costs of the shortcuts, we have \(O(n^2)\) weighted intervals in practice. This means no running time advantages can be obtained by using shortcut interval sets for this algorithm.

5.1.2 Correctness

As in the correctness proof of SINGLE SCALE MIN-\# given in Section 2.3, we prove the correctness of IMAIIRIPROGRESSIVE by arguing that its output is both valid and minimal. Specifically, the combined number of points should be minimized while having a sequence of simplifications among which there exists a monotonic containment relation. Furthermore, each simplification \(S_k\) should not violate the maximum error \(\varepsilon_k\).

Consider \(\langle S_1, \ldots, S_m \rangle\) is the progressive simplification produced by IMAIIRIPROGRESSIVE for some polygonal curve \(C\) and set of error bounds \(E\). Enforcement of the monotonic containment relation follows directly from the structure of the algorithm. Lines 17-19 impose that for any \(1 \leq k < m\) and shortcut \((p_i, p_j)\) \(\in S_{k+1}\), there exists a subsequence \((p_i, \ldots, p_j)\) \(\subseteq S_k\). We therefore must have \(S_{k+1} \subseteq S_k\).

Furthermore, by applying Lemma 2.3, we deduce that each simplification \(S_k\) does not violate the error bound \(\varepsilon_k\).

It remains to be shown that the cumulative size of all simplifications is minimal. To this end, we define the following for any \(n\) of links for subsequence \(\langle n \rangle\):

\[
S_{ij}^k = \{ (p_x, p_y) \in S_k \mid x \leq i < j \leq y \}
\]

Intuitively, this definition includes all line segments of \(S_k\) that span the subcurve \((p_i, \ldots, p_j)\). Let us now prove that the cost of a shortcut \((p_i, p_j)\) in \(G(C, \varepsilon_k)\) is the same as the combined number of links for subsequence \((p_i, \ldots, p_j)\) in all simplifications from scale 1 to \(k\).

Lemma 5.2. For any \(1 \leq k \leq m\) and \(n \geq j > i \geq 1\), if \((p_i, p_j) \in S_k\), then \(c_{ij}^k = \sum_{\ell=1}^k |S_{ij}^\ell|\).

Proof. We show \(c_{ij}^k = \sum_{\ell=1}^k |S_{ij}^\ell|\) by induction on \(k\) using the following induction hypothesis:

\[
\text{For any } n \geq j > i \geq 1, \text{ if } (p_x, p_y) \in S_k, \text{ then } c_{ij}^k = \sum_{\ell=1}^k |S_{ij}^\ell| \quad (\text{IH})
\]

Base \(k = 1\): Take any shortcut \((p_i, p_j) \in S_1\). This implies \((p_i, p_j) \in G(C, \varepsilon_1)\). Furthermore, we know \(S_{ij}^1 = \{(p_i, p_j)\}\), so \(|S_{ij}^1| = 1\). We derive the following:

\[
c_{ij}^1 = 1 = \sum_{\ell=1}^1 1 = \sum_{\ell=1}^1 |S_{ij}^\ell|
\]

Step \(k > 1\): Take any line segment \((p_i, p_j) \in S_{k+1}\). This implies \((p_i, p_j) \in G(C, \varepsilon_{k+1})\), and \(S_{ij}^{k+1} = \{(p_i, p_j)\}\), so \(|S_{ij}^{k+1}| = 1\).

Consider any \(1 \leq \ell \leq k\) and a path \(\pi \in \prod_{ij}^{k}\) such that \(\sum_{(p_x, p_y) \in \pi} |S_{ij}^\ell|\) is minimal. We derive that \(\pi = S_{ij}^\ell\), such that \(S_{ij}^\ell\) is minimal for all \((p_x, p_y) \in \pi\). Note that \(\pi = S_{ij}^\ell \subseteq G(C, \varepsilon_{\ell}) \subseteq G(C, \varepsilon_k)\), since \(\varepsilon_k \geq \varepsilon_{\ell}\). From this we may conclude that \(\pi\) is both in \(\prod_{ij}^{k}\) and \(\prod_{ij}^{\ell}\). From this it follows that:

\[
\min_{\pi \in \prod_{ij}^{k}} \sum_{(p_x, p_y) \in \pi} |S_{ij}^\ell| = \min_{\pi \in \prod_{ij}^{\ell}} \sum_{(p_x, p_y) \in \pi} |S_{ij}^\ell| \quad (5.1)
\]
Furthermore, from $\pi = S^i_j$ we can derive that $S^x_j \cap S^y_i = \emptyset$ for any $(p_x, p_y)$ and $(p_y, p_z)$ in $\pi$. Thus, combining $S^x_j$ for all $(p_x, p_y) \in \pi$ yields a non-overlapping sequence of shortcuts from $p_i$ to $p_j$. This gives us:

$$
\min_{\pi \in \Pi^i_j} \sum_{(p_x, p_y) \in \pi} |S^x_j| = |S^i_j|
$$

We may now derive the following:

$$
c^{k+1}_{ij} = 1 + \min_{\pi \in \Pi^i_j} \sum_{(p_x, p_y) \in \pi} c^k_{xy}
$$

$$
= 1 + \min_{\pi \in \Pi^i_j} \sum_{(p_x, p_y) \in \pi} \sum_{\ell=1}^{k} |S^x_j| \tag{IH}
$$

$$
= 1 + \sum_{\ell=1}^{k} \min_{\pi \in \Pi^i_j} \sum_{(p_x, p_y) \in \pi} |S^x_j| \tag{Equation 5.1}
$$

$$
= 1 + \sum_{\ell=1}^{k} |S^i_j| \tag{Equation 5.2}
$$

$$
= \sum_{\ell=1}^{k+1} |S^i_j| \tag{|S^i_{k+1}| = 1}
$$

Note that we can apply the induction hypothesis, because $(p_x, p_y) \in S_k$ as stated on line 18 of IMAIHIPROGRESSIVE. We conclude $c_{ij} = \sum_{\ell=1}^{k} |S^i_j|$ if $(p_i, p_j) \in S_k$ for all $1 \leq k \leq m$. 

**Theorem 5.3.** Given a polygonal curve $C$ and set of error bounds $E$, a minimal progressive simplification can be computed in $O(n^3 m)$ time under distance measures for which the validity of a shortcut can be determined in $O(n)$ time, such as Fréchet, Hausdorff and area-based measures.

**Proof.** It remains to be proven that the cumulative size of the simplifications computed by IMAIHIPROGRESSIVE is minimal. Let $\langle S^1_1, \ldots, S^m_m \rangle$ be a minimal progressive simplification, and let $\langle S_1, \ldots, S_m \rangle$ be the simplification computed by IMAIHIPROGRESSIVE. We derive the following:

$$
\min_{\pi \in \Pi^i_j} \sum_{(p_x, p_y) \in \pi} c^m_{xy} = \min_{\pi \in \Pi^i_j} \sum_{(p_x, p_y) \in \pi} \sum_{\ell=1}^{m} |S^x_j| \tag{Lemma 5.2}
$$

$$
= \sum_{\ell=1}^{m} \min_{\pi \in \Pi^i_j} \sum_{(p_x, p_y) \in \pi} |S^x_j| \tag{Equation 5.1}
$$

$$
= \sum_{\ell=1}^{m} |S^i_j| \tag{Equation 5.2}
$$

$$
= \sum_{\ell=1}^{m} |S_{\ell}|
$$

Note that we can apply Lemma 5.2 since $(p_x, p_y) \in S_m$, as stated on line 15 of IMAIHIPROGRESSIVE. We observe that the cumulative size of the solution produced by the algorithm constitutes a minimization of the cost function. Therefore, the output of IMAIHIPROGRESSIVE is a combination of paths in the shortcut graphs $G(C, \varepsilon_1), \ldots, G(C, \varepsilon_m)$ among which there exists a monotonic containment relation, such that the combined length is minimal.
Furthermore, by following the same reasoning as in the proof of Lemma 2.4, we know that every $S'_1 \in \langle S'_1, \ldots, S'_m \rangle$ has to be a path in $G(C, \varepsilon_\ell)$.

From these two claims we conclude $\sum_{\ell=1}^m |S_\ell| \leq \sum_{\ell=1}^m |S'_\ell|$. $\Box$

Note that these proofs also apply if we extend the cost function to solve the weighted progressive simplification problem Weighted Progressive Min-#, as described in Section 5.1. Therefore, by Theorem 5.3 and Lemma 2.2 we can deduce the following:

Corollary 5.4. Given a polygonal curve $C$, a continuous progressive simplification can be computed in $O(n^3)$ time under distance measures for which the validity of a shortcut can be determined in $O(n)$ time, such as Fréchet, Hausdorff and area-based measures.

5.2 Greedy Construction Techniques

Because the global minimization of the cumulative simplification size of Progressive Min-# is complex, we investigate two techniques for solving this problem where no requirements are placed on the size of the simplifications. Instead of searching for a global minimum of the combined simplification size, we greedily build the simplifications scale by scale, finding local minima while maintaining the monotonic containment. The simplest way is by building the simplifications from the bottom up, or from the top down. In this section we explore these techniques and discuss how they can be integrated with existing single-scale simplification algorithms.

5.2.1 Top-down Construction

By constructing simplifications from the top down, we employ the divide-and-conquer design paradigm. At every scale $k$, we divide $C$ into subcurves by simplification with error bound $\varepsilon_k$.

Next, on scale $k-1$, we simplify each of these subcurves using $\varepsilon_{k-1}$, thereby generating more fine-grained subcurves of $C$. Note that we have already seen this technique in ImaiIriProgressive. The difference is that we greedily choose a simplification $S_m$ by finding a shortest path in the unweighted shortcut graph $G(C, \varepsilon_m)$. The simplification $S_{m-1}$ is then found by concatenating all greedily chosen shortest paths from $p_i$ to $p_j$ for every $(p_i, p_j) \in S_m$. We continue this process until we have found $S_1$. This gives us Algorithm 5.2.

**Algorithm 5.2 ImaiIriTopDown($C, \mathcal{E}$)**

1: $G(C, \varepsilon_1), \ldots, G(C, \varepsilon_m) \leftarrow$ SHORTCUTS MULTI-Scale($C, \mathcal{E}$)

2: for $k \leftarrow m$ to 1 do
3: if $k = m$ then
4: $S_k \leftarrow$ shortest path from $p_1$ to $p_n$ in $G(C, \varepsilon_k)$
5: else
6: for each $(p_i, p_j)$ in $S_{k+1}$ do
7: $S_k \leftarrow S_k \cup$ shortest path from $p_i$ to $p_j$ in $G(C, \varepsilon_k)$
8: end for
9: end if
10: end for
11: return $\langle S_1, \ldots, S_m \rangle$

**Running time** The running time is bound by the time spent solving SHORTCUTS MULTI-Scale($C, \mathcal{E}$), giving us $O(n^2 m)$ time (see Section 4.1) or $O(n \log n + n^2 m)$ time (see Section 4.2).
Optimization  Due to the greedy selection of any simplification and the monotonic containment relation among the simplifications, potentially many shortcuts are never considered. For any scale $1 \leq k < m$ and shortcut $(p_x, p_y) \in S_{k+1}$, we find a simplification of $(p_x, \ldots, p_y)$ in $G(C, \epsilon_k)$. This means that any shortcut $(p_i, p_j) \in G(C, \epsilon_k)$ can never be included in $S_k$ if $j > x > i$ or $j > y > i$.

We can therefore optimize the algorithm by not considering these shortcuts when constructing the shortcut graphs. Instead of solving Shortcuts Multi-Scale, we solve Shortcuts Single Scale at every scale $k$ using an algorithm that prunes the search space whenever possible.

We extend the shortcut graph construction algorithm by Chin and Chan [6] (Section 4.1). Recall that we construct $G(C, \epsilon_k)$ by intersecting two sets of shortcuts obtained by traversing $C$ in both directions. When traversing $C$ from start to end, we skip from $p_i$ to $p_{i+1}$ whenever we reach a shortcut $(p_i, p_j)$ such that $i < y < j$ for a line segment $(p_x, p_y) \in S_{k+1}$. Conversely, when traversing from the end to the start of $C$, we skip from $p_j$ to $p_{j-1}$ whenever we consider a shortcut $(p_i, p_j)$ such that $i < x < j$ for a line segment $(p_x, p_y) \in S_{k+1}$. These two types of pruning are illustrated in Figure 5.3.

![Figure 5.3: Pruning the search space for shortcut graph $G(C, \epsilon_k)$.](image)

5.2.2 Bottom-up Construction

By taking greedy choices from the bottom up, we start at the lowest scale with the smallest error bound, thereby starting with the least aggressive greedy choice. Construction at any scale $k$ works by finding a shortest paths from $p_1$ to $p_n$ in $G(C, \epsilon_k)$, and removing all nodes and edges in $G(C, \epsilon_{k+1})$ associated with any point $p_i \notin S_k$, thereby directly imposing $S_{k+1} \subseteq S_k$. This is illustrated in Figure 5.4. The algorithm’s pseudo code is presented in Algorithm 5.3.

**Algorithm 5.3 ImaiIriBottomUp($C, \mathcal{E}$)**

1: $G(C, \epsilon_1), \ldots, G(C, \epsilon_m) \leftarrow$ Shortcuts Multi-Scale($C, \mathcal{E}$)
2: for $k \leftarrow 1$ to $m$ do
3:   $S_k \leftarrow$ shortest path from $p_1$ to $p_n$ in $G(C, \epsilon_k)$
4:   for each $p_k$ in $C$ do
5:     if $p_i \notin S_k$ then
6:       Remove $p_i$ from $G(C, \epsilon_{k+1})$
7:     end if
8:   end for
9: end for
10: return $(S_1, \ldots, S_m)$

Running time  As for ImaiIriTOPDOWN, the running time is bound by the time spent constructing the shortcut graphs, yielding $O(n^2m)$ (Section 4.1) or $(n^2 \log n + n^2m)$ (Section 4.2).
5.2. Greedy Construction Techniques

As for the top-down approach, we can improve the performance of this algorithm by integrating the construction of the shortcut graphs with the greedy choices that are made at every scale. Instead of pruning the shortcut graph on line 6, we can limit the search space while constructing the shortcut graph using CHNCHN (see Section 4.1).

Consider we are constructing $G(C, \varepsilon_k)$ for some $1 \leq k \leq m$. During the forward traversal of $C$, we skip from $p_i$ to $p_{i+1}$ whenever $p_i \not\in S_{k-1}$. Furthermore, for every shortcut $(p_i, p_j)$ where $p_j \not\in S_{k-1}$, we do not check whether $p_j$ is inside the wedge $w$, since $(p_i, p_j)$ cannot be included in $S_k$. Note that we do intersect wedge $w$ with wedge $w_j$, since there may be valid shortcuts $(p_i, p_k)$ where $k > j$. Similar pruning is performed during the reverse traversal of $C$.

Naive construction Cao et al. [5] devised an efficient heuristic for simplifying polygonal curves progressively. This heuristic works by constructing the shortcut graph at any scale $k > 1$ on $S_{k-1}$, instead of on $C$. This way, we can obtain $S_k$ by finding a shortest path from $p_1$ to $p_n$ in $G(S_{k-1}, \varepsilon_k)$, as opposed to the pruned shortcut graph $G(C, \varepsilon_k)$. Although this would improve the running time since $|S_{k-1}| \leq |C|$, guarantees on the maximum error are lost. This is illustrated for the Hausdorff distance in Figure 5.5. Note how for higher scales, the simplifications become progressively worse in terms of the Hausdorff distance to the input curve. The bound on the error of simplification $S_k$ is $\max_{(p_i, p_j) \in S_k} \mathcal{H}((p_i, \ldots, p_j), (p_i, p_j)) \leq \sum_{\ell=1}^{k} \varepsilon_{\ell}$ for all $1 \leq k \leq m$.

5.2.3 Integrating Single-Scale Simplification

The structure of IMAI-IRI TOPDOWN (Algorithm 5.2) can trivially be exploited to facilitate other simplification techniques. We do this by replacing the shortest path queries with any algorithm

![Figure 5.4: Bottom-up greedy construction of progressive simplifications using Imai-Iri [16].](image)

![Figure 5.5: Naive bottom-up construction of simplifications using shortcut graphs. The Hausdorff distance to the input curve becomes progressively larger.](image)
that solves Single Scale Min-# either optimally or greedily.

Such an algorithm was developed by Ramer [20], and Douglas and Peucker [10], which greedily simplifies polygonal curves under the Hausdorff distance. For any polygonal curve $C = \langle p_1, \ldots, p_n \rangle$, this algorithm recursively splits the polygonal curve in two until the remaining subcurves can safely be replaced with a single line segment. The splitting point $p_s$ is found for any subcurve $C' = \langle p_i, \ldots, p_j \rangle \subseteq C$ by finding the furthest point in $C'$ from the line segment $(p_i, p_j)$. Pseudo code for this simplification heuristic is given in Algorithm 5.4.

**Algorithm 5.4 DouglasPeucker($C, \varepsilon$)**

1: $d_{max} \leftarrow 0$
2: for $i \leftarrow 2$ to $n - 1$ do
3: $d_i \leftarrow \text{dist}(p_i, (p_1, p_n))$
4: if $d_i > d_{max}$ then
5: $d_{max} \leftarrow d_i$
6: $p_s \leftarrow p_i$
7: end if
8: end for
9: if $d_{max} > \varepsilon$ then
10: $S_1 \leftarrow \text{DouglasPeucker}(p_1, \ldots, p_s, \varepsilon)$
11: $S_2 \leftarrow \text{DouglasPeucker}(p_s, \ldots, p_n, \varepsilon)$
12: $S \leftarrow S_1 \cup S_2$ Include $p_s$ only once
13: else
14: $S \leftarrow \{p_1, p_n\}$
15: end if
16: return $S$

The simplification heuristic by Cao et al. [5] discussed in Section 5.2.2 works for any curve simplification algorithm, and can therefore be applied to DouglasPeucker. We thus have $S_k = \text{DouglasPeucker}(S_{k-1}, \varepsilon_k)$ for any $1 < k \leq m$. Despite this naive construction heuristic, we retain our bound on the Hausdorff distance across all scales, since the algorithm splits each subcurve on the furthest point from the corresponding shortcut. This is unlike the naive bottom-up construction using shortcut graphs as discussed in Section 5.2.2. Furthermore, as a result of this splitting strategy, constructing from the bottom up always yields the same simplification as constructing from the top down.

**Running time** If we expect that every split of every subcurve $C' \subseteq C$ is roughly in the middle, the expected running time complexity of this algorithm can be described by the linear recurrence $T(n) = 2T(\frac{n}{2}) + O(n)$. Solving this recurrence using the Master theorem yields $T(n) \in \Theta(n \log n)$ meaning the expected running time of DouglasPeucker to greedily solve Single Scale Min-# is $\Theta(n \log n)$. In the worst case, every split yields a subcurve with size $\Theta(n)$, meaning that the worst-case running time is $\Theta(n^2)$.

We conclude that both the top-down and the bottom-up extension of DouglasPeucker to greedily solve Progressive Min-# yields an expected running time of $\Theta(nm \log n)$, and a worst case running time of $\Theta(n^2m)$.

### 5.2.4 Worst-case Performance

One disadvantage of using greedy algorithms for progressive curve simplifications is that a single bad choice on a certain scale can lead to a poor simplification on another scale. To highlight this, consider Figure 5.6. Here we see a recurring zigzag pattern of points between $p_i$ and $p_j$ with a length of $O(n)$. Note that the subcurve $\langle p_i, \ldots, p_j \rangle$ can only be simplified by $(p_i, p_j)$ for any error.
bound $\varepsilon \geq \varepsilon_x$. Therefore, any greedy choice that excludes $(p_i, p_j)$ yields a simplification that is $O(n)$ larger than the minimum.

Consider the case where $\varepsilon_1 = \varepsilon_x$ and $\varepsilon_2 = \varepsilon_y$. In this case, IMAIIRIBOTTOMUP includes shortcut $(p_i, p_j) \in S_1$. However, IMAIIRITOPDOWN yields a simplification $S_2 = \langle p_1, p_{i+1}, p_n \rangle$ using $\varepsilon_y$ at the second scale. This results in an independent simplification of $(p_1, \ldots, p_{i+1})$ and $(p_{i+1}, \ldots, p_n)$ at the first scale using $\varepsilon_x$. Therefore, shortcut $(p_i, p_j)$ cannot be included in $S_1$, meaning IMAIRIITOPDOWN is forced to include $(p_{i+1}, \ldots, p_j)$ in $S_1$.

Next, consider the case where $\varepsilon_1 = 0$ and $\varepsilon_2 = \varepsilon_x$. IMAIRIITOPDOWN will efficiently simplify $(p_1, \ldots, p_n)$ at the second and first scale. However, IMAIIRIBOTTOMUP removes $p_i$ using $\varepsilon_1 = 0$ since it lies on the line segment $(p_1, p_{i+1})$. This however removes the possibility of including shortcut $(p_i, p_j)$ at the second scale. IMAIRIITOPDOWN thus must include $(p_{i+1}, \ldots, p_j)$ in $S_2$.

Note that in both cases, both a bottom-up and top-down construction of a progressive simplification using DOUGLASPEUCKER will fail to include $(p_i, p_j)$ in any simplification, due to its splitting heuristic.

Figure 5.6: Problematic case for the greedy construction algorithms.

5.3 Experimental Evaluation

We explore how the real-world performance of every proposed progressive simplification algorithm relates to its theoretical asymptotic running time. We therefore experimentally investigate the performance of each simplification algorithm along two dimensions: the length of the input curve, and the number of scales.

For the experimental setup, refer to Section 3.3.

5.3.1 Performance by Curve Length

To gain general insights into the performance of every simplification algorithm – both in terms of running time and simplification complexity – we experimentally evaluate how the performance of each algorithm relates to the length of the input curve. For these experiments, we fix the number of scales to 10. Furthermore, we linearly sample the error bounds from the smallest shortcut error up to the tenth shortcut error percentile, yielding a density of 10% for shortcut graph $G(C, \varepsilon_m)$. For every algorithm, we measure both its running time, and the cumulative simplification size $\sum_{i=1}^{m} |S_i|$.

To gauge how the running time is influenced by imposing monotonic containment on the simplifications, we use a non-progressive simplification algorithm (II Non Prog.), which independently constructs min-# simplifications using shortcut graphs (See Section 2.3).

All shortcut graphs are represented using shortcut interval sets constructed by CHINCHAN (Algorithm 4.1), on which we compute shortest paths using range queries (Section 3.2). As dis-
discussed in Section 5.1.1, we cannot use shortcut interval sets for \textsc{ImaiIriProgressive}. However, we do wish to efficiently detect all shortcuts by using shortcut intervals in conjunction with \textsc{ChinChan}. Therefore, this algorithm is implemented such that \textsc{ChinChan} provides the algorithm with shortcut interval sets, from which every shortcut is copied to an explicit shortcut graph. On each such shortcut graph, we then compute shortest paths with an implementation of Dijkstra’s algorithm \cite{9} using a priority queue implemented by a pairing heap \cite{11}. This is a popular and flexible heap type commonly used for Dijkstra’s algorithm. Testing shows that for finding shortest paths in shortcut graphs, pairing heaps show similar or better performance compared to other popular heap types such as d-ary heaps, Fibonacci heaps, or binomial heaps.

The running time and cumulative simplification size of each algorithm for various lengths of the input curve is given in Table 5.1. Plotting these running times yields Figure 5.7. \textsc{ImaiIriProgressive} (II Prog.) exhibits running times that are at least an order of magnitude worse than every other algorithm. The performance of \textsc{ImaiIriTopDown} (II TD) is worse than \textsc{ImaiIriBottomUp} (II BU) since it performs less pruning in the search space of each shortcut graph.

<table>
<thead>
<tr>
<th>Length of the input curve</th>
<th>sec.</th>
<th>#</th>
<th>sec.</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>0.651</td>
<td>719</td>
<td>7.258</td>
<td>636</td>
</tr>
<tr>
<td>1500</td>
<td>0.290</td>
<td>646</td>
<td>2.357</td>
<td>570</td>
</tr>
<tr>
<td>2500</td>
<td>0.242</td>
<td>1066</td>
<td>1.915</td>
<td>1277</td>
</tr>
<tr>
<td>3500</td>
<td>0.076</td>
<td>814</td>
<td>0.278</td>
<td>723</td>
</tr>
<tr>
<td>5000</td>
<td>0.026</td>
<td>751</td>
<td>0.097</td>
<td>675</td>
</tr>
<tr>
<td>II BU Cao</td>
<td>0.0039</td>
<td>849</td>
<td>0.0111</td>
<td>862</td>
</tr>
<tr>
<td>DP BU</td>
<td>0.0046</td>
<td>849</td>
<td>0.0091</td>
<td>862</td>
</tr>
</tbody>
</table>

Table 5.1: Running time in seconds and cumulative simplification size for various lengths of the input curve and 10 scales.

Figure 5.7: Running time for various lengths of the input curve and 10 scales.
5.3. Experimental Evaluation

To evaluate the running time of the other algorithms, consider Figure 5.8. We observe that naive bottom-up construction as presented in Section 5.2.2 (II BU Cao) is consistently twice as fast as II BU. Furthermore, both the top-down construction (DP TD) and bottom-up construction (DP BU) using DouglasPeucker run in linear time in the length of the input curve, and are an order of magnitude faster than the other greedy algorithms. DP BU exhibits running times that are around 40% faster than DP TD. For few points, DP TD is faster than DP BU due to the benefits of the divide-and-conquer approach of DP TD, which for small input curves outweighs the time saved by DP BU’s pruning of the shortcut graphs.

Figure 5.8: Running time of the greedy algorithms for various lengths of the input curve and 10 scales.

Aside from the running time, we are interested in the cumulative size of the progressive simplifications produced by the algorithms. In Figure 5.9a, we see how these cumulative simplification sizes relate proportionally. We observe that for large curves, II TD produces simplifications that are proportionally larger when compared to smaller input curves. This is explained by the greedy choice strategy of II TD, which starts with a greedy choice on \( S_m \). Any error made in this simplification propagates to all lower scales. Therefore, II TD starts with the most aggressive greedy choice. The extent of a poor greedy choice for \( S_m \) exacerbates for larger input curves.

The proportions however do stabilize for larger input curves. A visualization of the cumulative simplification size of each algorithm for the largest input curve (5000 points) is given in Figure 5.9b. Note that II Prog. yields the best progressive simplification, which is 12% larger than the best possible non-progressive simplification obtained by II Non Prog. The second best progressive simplification is produced by II BU, which is 11% larger than II Prog. This simplification size is further lowered by II BU Cao, which does so by relaxing the error bound. This algorithm produces a simplification that is 8.5% smaller than II BU, and 1.2% larger than II Prog.

Figure 5.9: Cumulative simplification size for 10 scales.
Next, let us investigate how the cumulative simplification size is distributed over all scales for each algorithm. For this, we use the same parameters, but simplify for 15 scales instead of 10. The results are listed in Table 5.2 and plotted on a logarithmic scale in Figure 5.10.

Despite starting at the highest scale, II TD produces a simplification at scale 14 that is worse compared to the other algorithms. Furthermore, note that despite its poor cumulative simplification size, II TD does produce the best simplification for the highest scale.

\[
\begin{array}{cccccccccccccccc}
\text{Simplification size} \\
\hline
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
II Prog. & 412 & 149 & 118 & 89 & 75 & 64 & 50 & 45 & 38 & 32 & 31 & 29 & 25 & 25 & 24 \\
II BU & 402 & 203 & 146 & 106 & 87 & 66 & 59 & 47 & 44 & 38 & 35 & 33 & 29 & 28 \\
DP TD/BU & 553 & 287 & 184 & 126 & 97 & 73 & 62 & 51 & 47 & 44 & 40 & 40 & 36 & 33 & 32 \\
\hline
\end{array}
\]

Table 5.2: Simplification size at every scale for 5000 points and 15 scales.

We conclude that in practice, IMAIIRIPROGRESSIVE yields simplifications that are comparable in size to minimal non-progressive simplifications. However, it is too slow to be used on larger input curve. This makes IMAIIRIBOTTOMUP a sensible alternative which is significantly faster and yields simplifications that are only slightly larger. New techniques for finding shortest paths on weighted shortcut graphs have to be developed to improve the scalability of IMAIIRIPROGRESSIVE.

Furthermore, top-down construction by IMAIIRITOPDOWN was shown to be significantly worse than IMAIIRIBOTTOMUP, both in terms of running time and simplification complexity.

Finally, bottom-up construction of progressive simplifications using the well known algorithm by Douglas and Peucker [10] yields the best running-time performance in practice for all lengths of the input curve, and is thus the most scalable progressive simplification algorithm. However, it generates simplifications that are up to 56% larger than simplifications produced by IMAIIRIPROGRESSIVE, which is unsuitable in settings where the cumulative complexity of the progressive simplification is critical.
5.3. Experimental Evaluation

5.3.2 Performance by Number of Scales

We investigate how the running time and cumulative simplification size relates to the number of scales. For these experiments, we fix the length of the input curve to 3000 points. The results are given in Table 5.3. Note that for a single scale, every algorithm produces an optimal simplification, except for the heuristic-based algorithms DP TD and DP BU.

A plot of the running times is given in Figure 5.11. Note the linear relation between the running time and the number of scales. We observe a growth in running time of II Prog. that is around four times that of II Non Prog.

<table>
<thead>
<tr>
<th>Number of scales</th>
<th>1</th>
<th>8</th>
<th>20</th>
<th>44</th>
<th>68</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sec.</td>
<td># sec.</td>
<td># sec.</td>
<td># sec.</td>
<td># sec.</td>
</tr>
<tr>
<td>II Prog.</td>
<td>3.056</td>
<td>20</td>
<td>29.143</td>
<td>435</td>
<td>68.743</td>
</tr>
<tr>
<td>II Non Prog.</td>
<td>1.992</td>
<td>20</td>
<td>6.588</td>
<td>384</td>
<td>14.772</td>
</tr>
<tr>
<td>II TD</td>
<td>1.986</td>
<td>20</td>
<td>4.889</td>
<td>1020</td>
<td>8.557</td>
</tr>
<tr>
<td>II BU</td>
<td>1.995</td>
<td>20</td>
<td>0.749</td>
<td>509</td>
<td>1.188</td>
</tr>
<tr>
<td>II BU Cao</td>
<td>1.993</td>
<td>20</td>
<td>0.351</td>
<td>467</td>
<td>0.209</td>
</tr>
<tr>
<td>DP TD</td>
<td>0.0070</td>
<td>29</td>
<td>0.018</td>
<td>649</td>
<td>0.033</td>
</tr>
<tr>
<td>DP BU</td>
<td>0.0068</td>
<td>29</td>
<td>0.016</td>
<td>649</td>
<td>0.019</td>
</tr>
</tbody>
</table>

Table 5.3: Running time in seconds and cumulative simplification size on 3000 points for various numbers of scales.

Figure 5.11: Running time on 3000 points for various numbers of scales.

The running times of the other algorithms are visualized in Figure 5.12. Unlike all other algorithms, II BU Cao is faster for many scales, which is related to the sampling of error bounds. By sampling more error bounds, the first error bound $\varepsilon_1$ becomes smaller, which means the construction of shortcut graph $G(\mathcal{C}, \varepsilon_1)$ using CHINCHAN (Algorithm 4.1) is faster, due to the additional pruning. II BU Cao prunes drastically at the first scale, and thus its running time is almost fully determined by the time spent on the first simplification.
Chapter 5. Progressive Simplification of Polygonal Curves

Recall that II BU performs less pruning, since shortcuts in $S_k$ are detected in $C$ instead of $S_{k-1}$. II BU performs best around 7 scales, where there is significant pruning at the first scale without spending too much time constructing simplifications at higher scales.

We observe that DP BU is 45% more efficient than DP TD. The growth in running time for both algorithms in the number of scales is much like the growth in the length of the input curve observed in Figure 5.8b.

Finally, consider Figure 5.13, where the cumulative simplification sizes are visualized for various numbers of scales. In Figure 5.13a, we observe the relative cumulative simplification size given by II TD increases for many scales, since poor greedy choices at higher scales cascade across more scales. Furthermore, we observe that the relative cumulative simplification size of DP TD/BU decreases as the number of scales grows. A visualization of the cumulative simplification size of each algorithm for 68 scales is shown in Figure 5.13b.

In conclusion, all algorithms exhibit performance linear in the number of scales, except II BU Cao, which performs best on many scales. The same trade-offs exist between running time and simplification size as described in Section 5.3.1. The most notable difference is that for many scales, DP TD and DP BU yield a relative simplification size that is better than for fewer scales. Thus, DP BU is a reasonable alternative to II Prog. when simplifying for many different levels of detail.
Chapter 6

Conclusions and Future Work

In this thesis we investigated both the theoretical and practical aspects of progressive simplification of polygonal curves. Progressive simplification enables exploration of zoomable visualizations of spatial data by imposing consistency between simplifications at different levels of detail. We aimed to bridge the gap between existing heuristic-based algorithms for producing progressive simplifications, and minimum-link non-progressive simplification algorithms. To solve this problem, we developed multiple extensions for a common non-progressive minimum-link simplification algorithm \[16\] using so-called shortcut graphs.

We developed an optimal algorithm that runs in $O(n^3m)$ time, and produces simplifications by finding shortest paths in weighted shortcut graphs where the weight of an edge $(p_i, p_j)$ at scale $k$ encodes the cost of the best possible sequence of simplifications up to scale $k$ on the subcurve from $p_i$ to $p_j$.

We further developed two greedy progressive simplification algorithms with a running time of $O(n^2m)$. These algorithms construct the simplifications in a certain order (bottom-up or top-down), while enforcing monotonic containment. Because each simplification is constructed using local minima, these algorithms do not strictly minimize the combined simplification complexity.

To efficiently construct shortcut graphs for multiple different error bounds, we developed an algorithm that determines the Hausdorff distance \[12\] of every shortcut to the associated contiguous subsequence on the input curve. This allows us to efficiently determine shortcut validity for any error bound. The algorithm runs in $O(n^2 \log n)$ time, which is an improvement over $O(n^3)$ time using existing techniques \[16\]. Experimental evaluation reveals that this algorithm is only viable for a large number of scales, compared to independent construction of each shortcut graph for a fixed error bound using existing algorithms \[6\]. This makes it a good choice for computing continuous progressive simplifications in $O(n^3)$ time using $\binom{n}{2}$ scales.

To quickly find shortest paths in unweighted shortcut graphs, we developed a representation of the shortcut graph called a shortcut interval set. This representation was shown to be of linear size in practice, which allows for finding shortest paths in $O(n \log n)$ time, yielding near-linear performance in practice. This is a significant improvement over the quadratic running-time performance given by breadth-first search. Furthermore, we showed how construction of shortcut interval sets can be integrated with existing techniques \[6\] to significantly speed up construction of shortcut graphs under the Hausdorff distance in practice. Shortcut interval sets can thus be integrated with all aspects of min-\# curve simplification using shortcut graphs, lowering the storage requirements from $O(n^2)$ to $O(n)$. This greatly improves the scalability of using shortcut graphs, both in progressive and non-progressive settings.

Experimental evaluation of the optimal progressive simplification algorithm reveals that it is not scalable to large input curves. We found that the best alternative is a bottom-up greedy construction of the simplifications using shortcut graphs, from the smallest error bound to the
largest error bound. This greedy algorithm has excellent running-time performance in practice, while producing near-minimal progressive simplifications.

In conclusion, we have developed several progressive simplification algorithms where for each algorithm there exists a trade-off between scalability and simplification complexity. As a by-product of our endeavors, a space-efficient representation of the shortcut graph was developed which is not only applicable to progressive simplification, but also highly suitable for computing simplifications for a single error bound.

Although shortcut interval sets yield great benefits in practice, simplification for a single error bound takes quadratic time in practice to detect all shortcuts. As a future work, it might be of interest to further optimize the construction of shortcut interval sets. This could be a stepping stone towards the first near-linear min-# simplification algorithm. This is valuable, since existing high-performance simplification algorithms either heuristically aim for the smallest number of points [10], or approximate the minimum number points by a certain factor [11], and may thus produce simplifications which are significantly worse than the optimum.

We focused on the Hausdorff distance as error measure, though any error measure can be used to determine the validity of a shortcut. To efficiently employ the developed progressive simplification algorithms in other settings, it is of interest to develop efficient techniques for constructing shortcut graphs for error measures other than the Hausdorff distance, such as Fréchet [4] or area-based measures [8].
Bibliography


