Gradual Line Simplification for Different Objective Functions

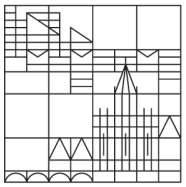
Bachelor Thesis

submitted by

Nick Krumbholz (01/1078145)

at the

Universität Konstanz



Algorithmics

Department of Computer and Information Science

Advisor: Prof. Dr. Sabine Storandt

Reviewer: Prof. Dr. Michael Grossniklaus

Konstanz, 4th July 2023

Abstract

In this work, we address the problem of line simplification, which involves generating simplified representations of a polygonal curve while preserving its shape and features (see figure 1 below). This problem is highly relevant and significant in various applications. To tackle the problem, we present a novel approach that centers around the development of progressive simplifications. Specifically, we introduce the concept of gradual line simplification, which entails iteratively removing vertices to achieve increasingly broader representations of the curve (see figure 2 below).

There is no universal objective function that can optimally address every application and target requirement. As a result, we put forth a total of six different objective functions. Additionally, we address these functions by introducing dynamic programming approaches that solve three of them in $\mathcal{O}(n^3)$ time complexity. For the remaining problems, we present modifications to the dynamic programs that reduce the theoretical runtime at the expense of quality. Furthermore, we introduce a greedy algorithm that provides a constant factor approximation for two objectives. To enhance the practicality of the greedy algorithm, we offer a practical adaptation and investigate its impact on the approximation bounds. Considering the need for efficient access to simplifications with limited memory usage, we propose an output-sensitive algorithm capable of extracting a simplification using linear space.

In order to analyze the practicability of our algorithms, we conducted an experimental study where we tested them on real-world data. There, we found that our algorithms may retrieve good solutions but scale strong in runtime. Using greedy approaches or adaptations of algorithms, we show that we are able to tackle this problem, while achieving a reasonable quality.

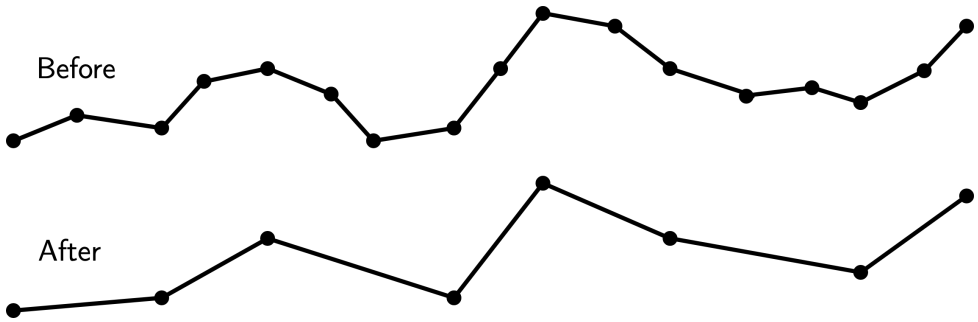


Figure 1: An example line simplification.

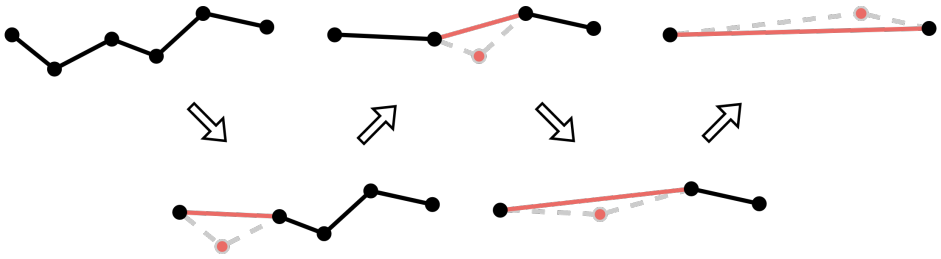


Figure 2: An example gradual line simplification.

Contents

1 Introduction

A polygonal curve is a sequence of 2-dimensional points, where two consecutive points induce a straight line between them. Many applications face the problem of visualizing these polygonal curves with far more detail than necessary for the human eye. One such application can be observed when visualizing large networks [14]. Another use can be found in movement analysis, particularly in the examination of spatio-temporal data of mobile devices [5]. Therefore, to optimize render time and storage usage, there is a need of simplifying polygonal curves while preserving its shape and features. In line simplification we try to approach this problem by removing a set of vertices from the sequence and ensure that the resulting curve remains a faithful representation of the original curve (see figure 3 below) by measuring a distance between those curves. Geometric distance metrics are utilized to measure this distance. The most commonly used distance metrics are the Hausdorff distance and the Fréchet distance.

There exist two sub-problems that can be either employed in line simplification. The first problem aims to find the minimum number of vertices while ensuring that the induced distance is bounded by a given error, so called *min-#* problem. This is useful when we want to guarantee a certain level of similarity while not being concerned with maintaining a specific level of complexity. On the other hand, the second problem focuses on minimizing the induced distance while restricting the number of vertices in the simplified curve, being called the *min-ε* problem. This is advantageous when we want to preserve some level of complexity and not necessarily guarantee any similarity bounds. Depending on the aim of the application, either of the problems can be considered reasonable and utilized effectively.

The challenge of multi-scale line simplification arises in applications where polygonal curves need to be simplified for various complexities, rather than just once. This problem is commonly faced in cartography, particularly when zooming in or out in road maps to maintain varying levels of detail based on the zoom factor [15]. One potential approach to solve this is to apply a single-scale simplification algorithm for each scale individually. However, this method can result in complications when visualizing multiple scales in a sequence, as the resulting simplified curves may look different from one another and lack inter-dependency despite sharing a similar origin. Progressive line simplification [3] is the problem dedicated to fix this issue. In contrast to vanilla multi-scale line simplification it requires that included vertices of each simplification must also be present in all simplifications before. This is especially useful when zooming out in a road map because we imply monotony and hence, no vertices reappear after their removal. Its target optimization function, as defined by Buchin et al. [1], shares similarities with the *min-#* objective. In contrast, the key distinction lies in minimizing the sum of the number of vertices across each simplification, as opposed to minimizing the number of vertices in a single simplification.

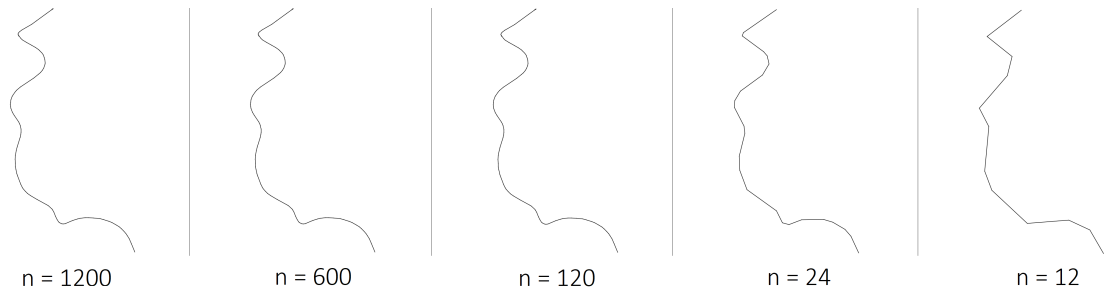


Figure 3: A polygonal curve shown at multiple scales

As progressive line simplification addresses the multi-scale version of the *min-#* problem, it is also necessary to have a *min-ε* alternative for multi-scale line simplification. Having this in mind, we propose **Gradual Line Simplification (GLS)**. The gradual line simplification approach involves repeatedly removing a single vertex until we are left with the first and last vertex. Each removal then forms a new simplification (see figure 4 below). It is obvious to see that it maintains the progressive nature of simplifications being built upon one another. As each removal leads to its own error, we could try to minimize the sum of all of them, similar to progressive line simplification, with the main difference that we consider the error instead of the number of vertices. However, there are several other viable objective functions that can be utilized. One such function is the max function, which corresponds to the maximum implied distance among all simplifications.

This study focuses on gradual line simplification for a variety of objective functions. As a result, after the formal introduction of general terms in section 3, we propose different reasonable objective functions in section 4 and section 5, each with its own unique properties. To address these functions, algorithms will be proposed in these sections, including those that provide exact solutions for certain objectives, as well as approximations and heuristics. Additionally, in section 6, we show an output-sensitive algorithm for the extraction of simplifications, as the extraction is an important part in practical applications. Finally, the performance of the simplification algorithms will be evaluated on real-world data in section 7, in order to compare their effectiveness under the proposed objective functions. We will do this for different distance metrics, such as the Hausdorff distance and the Fréchet distance. Furthermore, we discuss the findings in our experiments in section 8 and conclude our study in section 9. Finally, we propose interesting future work in section 10.

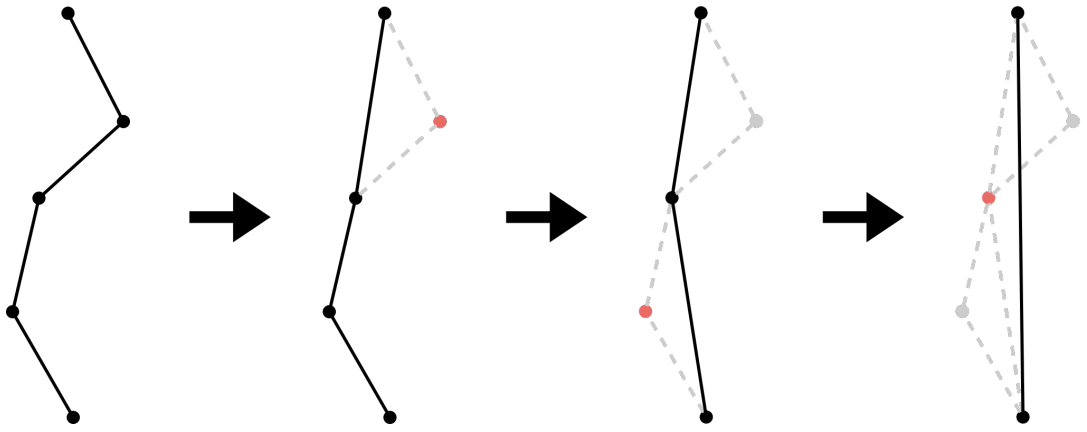


Figure 4: An example Gradual Line Simplification.

2 Related Work

Line simplification is a field of research that has been studied for a long time. One historical significant algorithm was proposed by Douglas and Peucker [8]. It is able to simplify a polygonal curve for a given error into a single scale. However, it does not aim to minimize the number of vertices. Therefore, it is not suitable to compute an exact solution for the *min-#* problem. Imai and Iri [12] proposed a method to address this issue by identifying shortest paths within a so called shortcut graph. This is a graph containing all line segments between two points that induce an error upper bounded by some value. For a polygonal curve with n points, the proposed technique of Imai and Iri has a runtime of $\mathcal{O}(n^3)$. Moreover, it was firstly improved by Melkman and O'Rourke [13] to a runtime of $\mathcal{O}(n^2 \log n)$ by using the geometrical constraints of the problem. With further improvements to the creation of shortcut graphs under the Hausdorff distance by Chan and Chin [6], this runtime was additionally reduced to $\mathcal{O}(n^2)$. Imai and Iri [12] also proposed an algorithm that solves the *min-ε* problem in $\mathcal{O}(n^3)$. Chan and Chins [6] improvements to the shortcut graphs also enhanced the runtime of this algorithm to $\mathcal{O}(n^2 \log n)$.

In multi-scale line simplification the vanilla approach of simplifying a polygonal curve for m scales individually then takes $\mathcal{O}(mn^2)$ for the *min-#* and $\mathcal{O}(mn^2 \log n)$ for the *min-ε* problem. When adding the progressive constraint, Buchin et al. [3] propose an algorithm that solves the problem in $\mathcal{O}(mn^3)$. To accomplish this, they use shortcut graphs and incorporate shortcut costs, which correspond to the number of line segments induced when utilizing a shortcut. To address the issue of a strong scaling runtime, Cao et al. [5] proposed an algorithm computing a progressive heuristic. This algorithm involves using a single scale algorithm on the polygonal curve to find the first simplification, and then repeatedly applying the same algorithm, while utilizing the previously determined simplification as input to find the next one. Since it uses the simplification as input it does not compute the exact solution, but it works fairly good in practice. Its runtime is given by $\mathcal{O}(mn^2)$.

Visvalingam and Whyatt [16] proposed an approach that is very similar to gradual line simplification. They iteratively discard the vertex that induces the smallest area with its neighbouring vertices. Inspired by them, Daneshpajouh [7] defined two area-based distance metrics, both using the induced area between two polygonal curves. The Fréchet distance is a far more well-studied distance metric, named after Maurice Fréchet. By utilizing a decision oracle and a binary search on critical values, Alt and Godau [2] proposed an algorithm computing the Fréchet distance between two polygonal curves. For two curves with lengths p and q, the algorithm runs in $\mathcal{O}((p^2q + pq^2) \log(pq))$. In line simplification, calculating the distance between two complete polygonal curves is often unnecessary, as one curve typically represents a simplified version of the other. Therefore, measuring the distance between a polygonal curve and a straight line segment between two vertices on the curve is often sufficient. Buchin et al. [4] show that using a special data structure, this can be computed in sublinear time. Furthermore, the discrete Fréchet distance is a discrete adaptation of the Fréchet distance, and was formally introduced by Eiter and Mannila [10]. Moreover, Eiter and Mannila have shown that, using a simple dynamic program, we are able to compute the discrete Fréchet distance in $\mathcal{O}(pq)$.

3 Preliminaries

Let us first formally introduce general terms, that are necessary to understand the following sections.

3.1 Line Simplification

A polygonal curve $\mathcal{L} = \langle v_1, \ldots, v_n \rangle$ with $v_i \in \mathbb{R}^2$ is a sequence of vertices, where each vertex represents a point on a 2 dimensional plane. Two consecutive vertices induce a straight line between them, which are called *line segments*. Furthermore, we let $\mathcal{L}(i)$ denote v_i and $\mathcal{L}[i,j] = \langle v_i, \ldots, v_j \rangle$ denote the sub-curve from v_i to v_j. The length of a curve $|L| = n$ is given by its number of vertices. For a geometric distance metric X, the distance $d_X(\mathcal{L}_1, \mathcal{L}_2)$ is the distance between the polygonal curves $\mathcal{L}_1$ and $\mathcal{L}_2$ under X. A simplification $\mathcal{S} \sqsubset \mathcal{L}$ with $v_1, v_n \in \mathcal{S}$ is a polygonal curve, that is created by removing a set of vertices from the original curve $\mathcal{L}$.

Moreover, a shortcut $s_{ij} = \{v_i, v_j\}$ with $i \leq j$ is a direct connection between vertices v_i and v_j. A simplification $\mathcal{S}$ *uses* a shortcut s_{ij} if $\mathcal{L}(i), \mathcal{L}(j) \in \mathcal{S}$ and $\forall i < k < j : \mathcal{L}(k) \notin \mathcal{S}$. The shortcut error $d_{\mathcal{L};X}(s_{ij})$ is given by $d_{\mathcal{L};X}(s_{ij}) = d_X(\langle \mathcal{L}(i), \mathcal{L}(j) \rangle, \mathcal{L}[i,j])$. We call two shortcuts s_{ij} and s_{xy} *independent* if their shortcutted vertices $v_{i+1}, \ldots, v_{j-1}$ and $v_{x+1}, \ldots, v_{y-1}$ do not overlap, given by $\max\{i,j\} \leq \min\{x,y\} \vee \max\{x,y\} \leq \min\{i,j\}$. It is important to note, that every simplification $\mathcal{S} \sqsubseteq \mathcal{L}$ can be created by applying a set of shortcuts onto $\mathcal{L}$.

Next, we formally introduce gradual line simplification, using previous notation. To simplify notation, we will often refer a non-simplified curve $\mathcal{L}$ as $\mathcal{S}_0$.

[Definition 3.1] Gradual Line Simplification (GLS)

Given a polygonal curve $\mathcal{L} = \mathcal{S}_0 = \langle v_1, \ldots, v_n \rangle$ with $n \geq 3$. The gradual line simplification problem then asks for simplifications $\mathcal{S}_1, \ldots, \mathcal{S}_{n-2}$, with $\mathcal{S}_0 \sqsupset \mathcal{S}_1 \sqsupset \cdots \sqsupset \mathcal{S}_{n-2}$ and $\mathcal{S}_{n-2} = \langle v_1, v_n \rangle$, while minimizing some objective function $f(X, \mathcal{S}_0, \ldots, \mathcal{S}_{n-2})$ under a geometric distance metric X.

As we remove exactly one vertex in each step, in gradual line simplification, we are interested in finding the corresponding removal sequence $r = \langle v_x, \ldots, v_y \rangle$ that minimizes the objective function, with $|r| = n - 2$, $\forall v_i \in r : 1 < i < n$ and $v_i \neq v_j$ if $i \neq j$. Then the i-th removal $r(i)$ is given by the vertex v that holds $v \in \mathcal{S}_{i-1} \wedge v \notin \mathcal{S}_i$. The shortcut used for the k-th removal is given by the one that transforms S_{k-1} into S_k, and we will denote it with $s(\mathcal{S}_k)$. In gradual line simplification, alternatively to the removal sequence, we can find a valid shortcut sequence that minimizes the objective function.

We previously described, that we introduce multiple objective functions. As these objective functions may have to be handled individually, each of them represent a problem themselves. Hence, for an objective function called Y, we will denote the according minimization problem as the 'Min$Y$ problem'.

3.2 Geometric Distance Metrics for Polygonal Curves

Geometric distance metrics are functions used to determine a distance between two geometric objects. In this work we will focus on geometric distance metrics that are typically used to determine the similarity between two polygonal curves. To measure the spatial distance between to vertices, we let $d(v_i, v_j)$ denote the euclidean distance between v_i and v_j.

3.2.1 Hausdorff distance

The Hausdorff distance between two polygonal curves is defined as the longest shortest distance between every point that exists on one curve to every point that exists on the other curve.

[Definition 3.2] Hausdorff Distance

Let $p(\mathcal{L}) = \{(1-\lambda)\cdot\mathcal{L}(i) + \lambda\cdot\mathcal{L}(i+1) \mid i \in \mathbb{N} \wedge 1 \leq i < |\mathcal{L}| \wedge \lambda \in \mathbb{R} \wedge \lambda \in [0;1]\}$ be a function, returning the set of all points on a polygonal curve (including points on line segments).
Given 2 polygonal curves $\mathcal{L}_1, \mathcal{L}_2$, the Hausdorff distance between them is then defined by:

$$d_H(\mathcal{L}_1, \mathcal{L}_2) = \max\left\{\max_{x\in p(\mathcal{L}_1)}\min_{y\in p(\mathcal{L}_2)} d(x,y), \max_{y\in p(\mathcal{L}_2)}\min_{x\in p(\mathcal{L}_1)} d(y,x)\right\}$$

3.2.2 Fréchet distance

The (continuous) Fréchet distance is a little more complex to understand. Imagine a dog is walking on one polygonal curve while its owner is walking on the other one. Initially, both stand on the first vertex of their respective vertex sequence. Furthermore, the dog is on a leash, and they can only move towards the next vertex in their sequence, excluding any previously visited vertices. In this scenario, the Fréchet distance would be the smallest possible leash length, such that they both traverse the curves completely in parallel from start to end.

[Definition 3.3] (Continuous) Fréchet distance

Let α, β be continuous non-decreasing functions with $\alpha(0) = \beta(0) = 0$, $\alpha(1) = |\mathcal{L}_1|$, $\beta(1) = |\mathcal{L}_2|$ and $\mathcal{L}(i+\lambda) = (1-\lambda)\cdot\mathcal{L}(i) + \lambda\cdot\mathcal{L}(i+1)$ with $i \in \mathbb{N}$ and $\lambda \in [0,1]$.
Then the (continuous) Fréchet distance between two polygonal curves $\mathcal{L}_1, \mathcal{L}_2$ is given by:

$$d_F(\mathcal{L}_1, \mathcal{L}_2) = \min_{\alpha,\beta}\max_{t\in[0,1]} d\left(\mathcal{L}_1(\alpha(t)), \mathcal{L}_2(\beta(t))\right)$$

Due to the consideration of point order, the Fréchet distance is commonly regarded as the superior choice when measuring the similarity between two curves, compared to the Hausdorff distance.

3.2.3 Discrete Fréchet distance

The discrete Fréchet distance is a discrete adaptation of the Fréchet distance. It can be better described with the following illustrative scenario. In this scenario, we replace the dog and its owner with two frogs. They use the same setup used in the Fréchet distance, with the difference, that frogs can only jump from one vertex to another. Only one frog can jump at a time, while the other one stays grounded until it lands again. Consequently, the discrete Fréchet distance is determined by the minimum leash length required for the frogs to traverse the curves entirely.

[Definition 3.4] Discrete Fréchet distance

Given two polygonal curves $\mathcal{L}_1, \mathcal{L}_2$ with lengths n and m. Let $\alpha : \mathbb{N} \mapsto \mathbb{N}, \beta : \mathbb{N} \mapsto \mathbb{N}$ be two continuous non-decreasing functions with $\alpha(0) = \beta(0) = 0$, $\alpha(n+m) = n$, $\beta(n+m) = m$ and $\alpha(i) + \beta(j) = i + j$ for $i, j > 0$.
Then the Discrete Fréchet distance between these curves is given by:

$$d_F(\mathcal{L}_1, \mathcal{L}_2) = \min_{\alpha,\beta} \max_{0 \leq t \leq n+m} d\left(\mathcal{L}_1(\alpha(t)), \mathcal{L}_2(\beta(t))\right)$$

4 Simple Objective Functions

An objective function is used to evaluate the performance of a solution by assigning a real number to an outcome. A lower value indicates that the result aligns better with the objective, while a higher value indicates the opposite. Consequently, this value is commonly referred to as the "error" of an outcome. The objective functions we will employ are defined by the signature $f(X, \mathcal{S}_0, \ldots, \mathcal{S}_{n-2})$, where X represents a geometric distance metric and $\mathcal{S}_0, \ldots, \mathcal{S}_{n-2}$ is the series of simplifications. As mentioned earlier, there are several reasonable objective functions available. In this section, we will specifically introduce the MAX and SUM function.

It is important to remember that $s(\mathcal{S}_i)$ represents the shortcut that converts $\mathcal{S}_{i-1}$ into $\mathcal{S}_i$.

[Definition 4.1] Objective Function - MAX

Given a sequence of simplifications $\mathcal{S}_1, \ldots, \mathcal{S}_{n-2}$ of a polygonal curve $\mathcal{S}_0$ created by a GLS and a geometric distance metric X, the MAX function returns the maximum shortcut error:

$$f_{MAX}(X, \mathcal{S}_0, \ldots, \mathcal{S}_{n-2}) = \max\{d_{\mathcal{S}_0;X}(s(\mathcal{S}_i)) \mid 1 \leq i \leq n-2\}$$

[Definition 4.2] Objective Function - SUM

Given a sequence of simplifications $\mathcal{S}_0, \ldots, \mathcal{S}_{n-2}$ created by a GLS, the SUM function returns the sum of all shortcut errors under a geometric distance metric X:

$$f_{SUM}(X, \mathcal{S}_0, \ldots, \mathcal{S}_{n-2}) = \sum_{i=1}^{n-2} d_{\mathcal{S}_0;X}(s(\mathcal{S}_i))$$

4.1 Algorithm

To find an optimal solution, we need to find a sequence of shortcuts that we can use, to produce a GLS, while minimizing the objective function MAX or SUM.

[Observation 4.3]

In GLS, utilizing a shortcut s_{ij} between two non-adjacent vertices v_i and v_j necessitates the removal of a vertex v_k between with $i < k < j$. The removal of v_k also mandates the utilization of shortcuts (or original line segments) s_{ik} and s_{kj}.

This is quite handy, as the maximum shortcut error induced by the utilization of s_{ij} is then given by the maximum of the shortcut error of s_{ij} and the maximum induced shortcut errors of the removal v_k. Therefore, we can define this in a simple recursive function.

[Definition 4.4] Recurvsive Function MAX

Given a polygonal curve $\mathcal{L}$ and a geometric distance metric X. The maximum shortcut error ε_{max} induced by the removal of v_k, while using a shortcut s_{ij} with $i < k < j$, is given by

$$\varepsilon_{max}(s_{ij}) = \max\{d_{\mathcal{L};X}(s_{ij}), \varepsilon_{max}(s_{ik}), \varepsilon_{max}(s_{kj})\}$$

[Lemma 4.5]

Given the sub-curve $\mathcal{L}[i,j]$. Assuming that all we know the minimum MAX error $\varepsilon_{MinMAX}(s_{a,b})$ of all sub-sub-curves $L[a,b]$ with $i \leq a \leq b \leq j$ and $|i-j| > |a-b|$. Then the minimum MAX error $\varepsilon_{MinMAX}(s_{ij})$ of the curve $\mathcal{L}[i,j]$ can be achieved by utilizing the removal of v_k with $k = \underset{i<x<j}{\operatorname{argmin}} \max\{\varepsilon_{MinMAX}(s_{ix}), \varepsilon_{MinMAX}(s_{xj})\}$.

Proof.

$$\begin{aligned}
\varepsilon_{max}(s_{ij}) \text{ is minimal} &\Leftrightarrow \max\{d_{\mathcal{L};X}(s_{ij}), \varepsilon_{max}(s_{ik}), \varepsilon_{max}(s_{kj})\} \text{ is minimal} \\
&\Leftrightarrow \max\{d_{\mathcal{L};X}(s_{ij}), \varepsilon_{MinMAX}(s_{ik}), \varepsilon_{MinMAX}(s_{kj})\} \text{ is minimal} \\
&\Leftarrow \max\{\varepsilon_{MinMAX}(s_{ik}), \varepsilon_{MinMAX}(s_{kj})\} \text{ is minimal} \\
&\Leftrightarrow k = \underset{i<x<j}{\operatorname{argmin}} \max\{\varepsilon_{MinMAX}(s_{ix}), \varepsilon_{MinMAX}(s_{xj})\}
\end{aligned}$$

□

Resulting, if we pick the v_k that minimizes the equation in each recursion, we are able to find the minimum MAX error when utilizing the shortcut s_{ij}.

[Corollary 4.6] Minimal MAX

The minimum MAX error $\varepsilon_{max}(s_{ij})$, is given by $\varepsilon_{MinMAX}(s_{ij})$ with

$$\varepsilon_{MinMAX}(s_{ij}) = \begin{cases} 0 & \text{if } |i-j| \leq 1 \\ \min\limits_{i<k<j} \max\{d_{\mathcal{L};X}(s_{ij}), \varepsilon_{MinMAX}(s_{ik}), \varepsilon_{MinMAX}(s_{kj})\} & \text{otherwise} \end{cases}$$

Proof. First case is trivial. Second case can be derived from definition 4.4 and lemma 4.5. □

For the SUM problem, we can utilize observation 4.3 to create a similar recursive function similar to the one defined in definition 4.4.

[Definition 4.7] Recurvsive Function SUM

Given a polygonal curve $\mathcal{L}$ and a geometric distance metric X. The sum of all shortcut errors ε_{sum} induced by the removal of v_k, while using a shortcut s_{ij} with $i < k < j$, is given by

$$\varepsilon_{sum}(s_{ij}) = d_{\mathcal{L};X}(s_{ij}) + \varepsilon_{sum}(s_{ik}) + \varepsilon_{sum}(s_{kj})$$

Furthermore, we can translate lemma 4.5 for the SUM problem.

[Lemma 4.8]

Given the sub-curve $\mathcal{L}[i,j]$. Assuming that all we know the minimum SUM error $\varepsilon_{MinSUM}(s_{a,b})$ of all sub-sub-curves $L[a,b]$ with $i \leq a \leq b \leq j$ and $|i-j| > |a-b|$. Then the minimum SUM error $\varepsilon_{MinSUM}(s_{ij})$ of the curve $\mathcal{L}[i,j]$ can be achieved by utilizing the removal of v_k with $k = \underset{i<x<j}{\operatorname{argmin}}\, \varepsilon_{MinSUM}(s_{ix}) + \varepsilon_{MinSUM}(s_{xj})$.

Proof.

$$\begin{aligned}
\varepsilon_{sum}(s_{ij}) \text{ is minimal} &\Leftrightarrow d_{\mathcal{L};X}(s_{ij}) + \varepsilon_{max}(s_{ik}) + \varepsilon_{max}(s_{kj}) \text{ is minimal} \\
&\Leftrightarrow d_{\mathcal{L};X}(s_{ij}) + \varepsilon_{MinSUM}(s_{ik}) + \varepsilon_{MinSUM}(s_{kj}) \text{ is minimal} \\
&\Leftrightarrow \varepsilon_{MinSUM}(s_{ik}) + \varepsilon_{MinSUM}(s_{kj}) \text{ is minimal} \\
&\Leftrightarrow k = \underset{i<x<j}{\operatorname{argmin}}\, \varepsilon_{MinSUM}(s_{ix}) + \varepsilon_{MinSUM}(s_{xj})
\end{aligned}$$

□

[Corollary 4.9] Minimal SUM

The minimum SUM error $\varepsilon_{sum}(s_{ij})$, is given by $\varepsilon_{sum}(s_{ij})$ with

$$\varepsilon_{MinSUM}(s_{ij}) = \begin{cases} 0 & \text{if } |i-j| \leq 1 \\ \min\limits_{i<k<j} d_{\mathcal{L};X}(s_{ij}) + \varepsilon_{MinSUM}(s_{ik}) + \varepsilon_{MinSUM}(s_{kj}) & \text{otherwise} \end{cases}$$

Proof. First case is trivial. The second case can be derived from definition 4.7 and lemma 4.8. □

In gradual line simplification, the final shortcut is always ensured to be $s_{1;n}$. This is because the last simplification, $\mathcal{S}n-2$, only includes the first and last vertex of the polygonal curve $\mathcal{L}$. Consequently, the minimum MAX error can be determined by $\varepsilon_{MinMAX}(s_{1;n})$, while the minimum SUM error can be determined by $\varepsilon_{MinSUM}(s_{1;n})$. There exist multiple ways to solve these equations. We propose allocating

an $n \times n$ matrix S, where $S(i,j)$ represents $\varepsilon_{MinMAX}(s_{1;n})$ under the MAX objective and $\varepsilon_{MinSUM}(s_{1;n})$ under the SUM objective. Furthermore, we require another $n \times n$ matrix K, where $K(i,j)$ indicates the value of k used to compute $S(i,j)$. This information will be useful for extracting the removal sequence at a later stage. It is obvious that S and K are symmetric matrices, and therefore, it is sufficient to compute values on, or below its diagonal. Using this matrix, we can employ an easy dynamic program to compute all values. Since each computation necessities the knowledge of values with smaller hop distance, we prioritize those with a smaller hop distance $|j-i|$ first, while we keep track of the choice of k in the matrix K. The algorithm for the MinMAX problem can be seen in algorithm 1.

Algorithm 1: MinMAX($\mathcal{L}, X$)

Data: polygonal curve $\mathcal{L}$, geometric distance metric X
Result: minimal error matrix S, k matrix K

```
1  n ← |L|
2  S ← n × n matrix with initial values 0
3  K ← n × n matrix
4  for hop = 2 to n − 1 do
5  |  for i = 1 to n − hop do
6  |  |  j ← i + hop
7  |  |  S(i, j) ← ∞
8  |  |  sDist ← d_{L;X}(s_ij)
9  |  |  for k = i + 1 to j − 1 do
10 |  |  |  kDist ← max{sDist, S(i, k), S(k, j)}
11 |  |  |  if kDist < S(i, j) then
12 |  |  |  |  S(i, j) ← kDist
13 |  |  |  |  K(i, j) ← k
14 |  |  |  end
15 |  |  end
16 |  end
17 end
```

We are able to trivially adapt the algorithm for the MinSUM problem, by changing the evaluation of the objective function to the SUM function. Then, we would have to change line 10 of the algorithm to '$kDist \leftarrow sDist + S(i,k) + S(k,j)$'.

[Theorem 4.10]

Let f_X be the runtime function of the geometric distance metric X. Then the MinMAX problem and MinSUM problem can be solved in $\mathcal{O}(n^2(f_X(n)+n))$ using $\mathcal{O}(n^2)$ space.

Proof. It is obvious to see, that the algorithm 1 needs $\mathcal{O}(n^2)$ space. For each cell $S(i,j)$ we compute $d_{\mathcal{L};X}(s_{ij})$ in $\mathcal{O}(f_X(n))$, as well as picking the minimal k, where there exists $\mathcal{O}(n)$ many. Therefore, the algorithm has a runtime of $\mathcal{O}(n^2(f_X(n)+n))$, since there exist n^2 cells that we have to compute.
The adaptation for the the runtime and space consumption then stays the same, as line 10 has the same space consumption and runtime as '$kDist \leftarrow sDist + S(i,k) + S(k,j)$'. □

Moreover, we can extract the removal sequence of the minimal problem by utilizing the following algorithm.

Algorithm 2: *backtracking*(K)

Data: K matrix
Result: removal sequence r

```
r ← ℕ^n
q ← new empty Queue
q.enqueue((1, n))
for i = n to 1 do
    (a, b) ← q.dequeue()
    k ← K(a, b)
    r[i] ← k
    if k − a > 1 then
        q.enqueue((a, k))
    end
    if b − k > 1 then
        q.enqueue((k, b))
    end
end
return removal
```

It is worth mentioning that the backtracking algorithm is not strictly limited to the utilization of a queue. The specific order of shortcuts is not really crucial. As long as we take them in a valid order that satisfy the constraints of gradual line simplification, we achieve the same MAX and SUM error. Alternatively, for example, we could employ a stack without violating any GLS constraints, while achieving the same objective error.

[Theorem 4.11]

The backtracking algorithm runs in $\mathcal{O}(n)$.

Proof. The algorithm has n iterations, whereas each iteration needs constant time. Therefore, the algorithm needs linear time in total. □

4.2 Geometric Distance Measures for Shortcuts

According to theorem 4.10, the previous MinMAX/MinSUM algorithm scales with the runtime of the selected geometric distance metric. Therefore, the quicker we are able to compute a distance, the faster the algorithm runs. In the definition of proposed objectives (definition 4.1 and definition 4.2) we can see that we are interested in the error of shortcuts exclusively. This way we do not need to compute the error between two whole polygonal curves, and instead, exclusively compute the local induced error of the shortcut. As a result, we now discuss how we are able to calculate the introduced distances for shortcuts.

4.2.1 Hausdorff distance for shortcuts

The calculation of the Hausdorff distance of a shortcut with length n can trivially be accomplished in $\mathcal{O}(n)$. To do this, we have to compute the shortest euclidean distance between each shortcutted vertex to the shortcut segment. Finally, the maximum of these distances corresponds to the Hausdorff distance of the shortcut. Consequently, according to theorem 4.10, we can solve the MinMAX problem and the MinSUM problem in $\mathcal{O}(n^3)$ for the Hausdorff distance.

4.2.2 Fréchet distance for shortcuts

The (continuous) Fréchet distance is a little more complex. Alt and Godau [2] showed that using a free-space diagram, the computation of the Fréchet distance between two polygonal curves with lengths p and q can be achieved in $\mathcal{O}((p^2q + pq^2)\ \log(pq))$. Therefore, for a polygonal curve $\mathcal{L}$ with length n, the computation of the Fréchet distance of a shortcut s_{ij} can be accomplished in $\mathcal{O}(n^2\ \log\ n)$. This is because the shortcut error is given by $d_{\mathcal{L};F}(s_{ij}) = d_F(\langle\mathcal{L}(i), \mathcal{L}(j)\rangle, \mathcal{L}[i, j])$, and hence, one sub-curve has always a constant length of 2 while the other one has a length of $\mathcal{O}(|i - j| + 1) \subseteq \mathcal{O}(n)$.

But the Fréchet distance can be further adapted for shortcuts. During the computation of the Fréchet distance using the algorithm proposed by Alt and Godau [2], we determine critical values ε that open a path between neighboring cells, and thus, potentially provide a valid path in the 2-dimensional cell-based free-space-diagram they described. As the shortcut itself consist of a single line segment exclusively, the free space diagram for a shortcut s_{ij} has the cell dimension of $1 \times |i - j|$. Then, it can be reduced to a single dimension of cells. Consequently, instead of performing a binary search on the critical values, we are able to find the maximum of all critical values, as the path between each neighboring cell has to be open. Then, for a polygonal curve with length n, we can then compute a shortcut s_{ij} in $\mathcal{O}(n^2)$, since there are $\Theta(n^2)$ critical values, as described by Alt and Godau [2]. As a result, the $MinMAX$ problem and the $MinSUM$ problem can be solved in $\mathcal{O}(n^4)$, using the adapted Fréchet distance computation for shortcuts.

There exists even better upper bounds for the calculation of shortcut errors under the Fréchet distance. Buchin et al. [4] propose a data structure that is able to extract the Fréchet distance of a shortcut in $\mathcal{O}((n/k)\log^2\ n + \log^4\ n)$, where k is a choosable factor. Using $k = \sqrt{n}$ we can extract the Fréchet distance of a shortcut in sub-linear time. Then, the resulting pre-computation time is in $o(n^3)$. Therefore, we can solve the MinMAX and MinSUM problem in $\mathcal{O}(n^3)$ in total, for the Fréchet distance. But, due to its complexity, it is not really applicable in practice and thus, in the following, we will stick with the adapted $\mathcal{O}(n^2)$ version of Alt and Godau [2].

4.2.3 Interval search for the Fréchet distance

Furthermore, Alt and Godau [2] presented a method to determine whether some error ε and two polygonal curves $\mathcal{L}_1, \mathcal{L}_2$ with lengths p and q hold $\varepsilon \geq d_F(\mathcal{L}_1, \mathcal{L}_2)$ in $\mathcal{O}(pq)$. Therefore, for a shortcut s_{ij}, this means we can compute whether an error ε upper bounds $d_{\mathcal{L};F}(s_{ij})$ in $\mathcal{O}(|i-j|) \subseteq \mathcal{O}(n)$. We can use this decision oracle in an interval search algorithm to approximate the real value of $d_{\mathcal{L};F}(s_{ij})$. This concept will be referred to as the 'Interval Search Fréchet distance' (short 'IS Fréchet'). Its computation can be seen more precisely in algorithm 3.

Algorithm 3: *interval-search-fréchet*$(\mathcal{L}, s_{ij}, k)$

Data: polygonal curve $\mathcal{L}$, shortcut s_{ij}, number of iterations k
Result: approximated $d_{\mathcal{L};F}(s_{ij})$

```
lower ← lowerBound(L, s_ij)
upper ← upperBound(L, s_ij)
for i = 1 to k do
    mid ← (lower + upper)/2
    if mid ≥ d_{L;F}(s_ij) then
        upper ← mid
    else
        lower ← mid
    end
end
return upper
```

For a polygonal curve $\mathcal{L}$ with length n, there are multiple lower and upper bounds that can be computed in $\mathcal{O}(n)$. One lower bound could be the Hausdorff distance, and even simpler is to choose 0. As upper bound one could take the maximum euclidean distance between some vertex on the shortcut and all shortcutted vertices.

[Theorem 4.12]

Assuming the lower and upper bound are computed in $\mathcal{O}(n)$, then algorithm 3 runs in $\mathcal{O}(kn)$.

Proof. According to the assumption the lower and upper bound are computed in $\mathcal{O}(n)$. The algorithm then has k iterations, where in each iteration we make use of the decision oracle in $\mathcal{O}(n)$ time. The resulting runtime is then given by $\mathcal{O}(n + kn) \subseteq \mathcal{O}(kn)$. □

Hence, we can solve the MinMAX/MinSUM algorithm under the IS Fréchet in $\mathcal{O}(kn^3)$.

4.2.4 Discrete Fréchet distance

As discussed, Eiter and Mannila [10] proposed a dynamic program to compute the discrete Fréchet distance for two polygonal lines with lengths p and q in $\mathcal{O}(pq)$. Then, for a polygonal curve with length n, we can compute the discrete Fréchet distance of any shortcut in $\mathcal{O}(n)$. Therefore, the MinMAX problem and the MinSUM problem can be solved in $\mathcal{O}(n^3)$ in total under the discrete Fréchet distance.

4.3 Approximation Algorithms

Depending on the choice of the geometric distance metric, the runtime of the MinMAX / MinSUM algorithm is cubic or even quartic. When dealing with large inputs, these runtime bounds may be too high for practical usage. Hence, to deal with these applications, we propose approximation algorithms that have a better theoretical runtime. We analyze the approximation bounds under the Fréchet distance.

4.3.1 MinMAX 2-approximation

First we show a lower bound for the MinMAX problem.

[Lemma 4.13]

For the MinMAX problem, every gradual line simplification of a polygonal curve $\mathcal{L}$ is lower bounded by $d_{\mathcal{L};X}(s_{1;n})$ with $n = |\mathcal{L}|$.

Proof. Using definition 4.4, we can compute the MAX error with $\varepsilon_{max}(s_{1;n})$.

$$\varepsilon_{max}(s_{1;n}) = \max\{d_{\mathcal{L};X}(s_{ij}), \varepsilon_{max}(s_{ik}), \varepsilon_{max}(s_{kj})\} \geq d_{\mathcal{L};X}(s_{1;n})$$

□

Furthermore, we would like to reference a well-known lemma by Agarwal [1], which provides a bound on the Fréchet distance of bridged shortcuts. To enhance clarity, we have restated the lemma using our own wording in the following section.

[Lemma 4.14]

Let s_{ij} be a shortcut of a polygonal curve $\mathcal{L}$. Then for every shortcut s_{ab} bridged by s_{ij} with $i \leq a < b \leq j$ it holds:

$$d_{\mathcal{L};F}(s_{ab}) \leq 2 \cdot d_{\mathcal{L};F}(s_{ij})$$

Proof. See Agarwal [1] □

Combining these lemmas we get the following corollary:

[Corollary 4.15]

Any removal sequence is a 2-approximation for the MinMAX problem under the Fréchet distance.

Proof. L et ε_{any} be the MAX error of some removal sequence.

$$\begin{aligned}
\varepsilon_{MinMAX}(s_{ij}) &= \min_{i<k<j} \max\{d_{\mathcal{L};F}(s_{ij}), \varepsilon_{MinMAX}(s_{ik}), \varepsilon_{MinMAX}(s_{kj})\} \\
&\leq \min_{i<k<j} \max\{d_{\mathcal{L};F}(s_{ij}), 2 \cdot d_{\mathcal{L};F}(s_{ij}), 2 \cdot d_{\mathcal{L};F}(s_{ij})\} && \text{lemma 4.13} \\
&= 2 \cdot d_{\mathcal{L};F}(s_{ij}) \\
&\leq 2 \cdot \varepsilon_{any} && \text{lemma 4.12}
\end{aligned}$$

□

4.3.2 MinSUM 4-approximation

Now we will find an approximation algorithm for the MinSUM problem. First we introduce an algorithm that repeatedly takes the shortcut with the smallest shortcut error, while preserving the constraints of GLS, denoted as the 'Greedy algorithm'. It can be efficiently implemented by keeping all current removal errors in a heap, while repeatedly extracting the minimum and updating its neighbours to keep track of current errors:

Algorithm 4: $greedy(\mathcal{L}, X)$

Data: polygonal curve $\mathcal{L}$, geometric distance metric X
Result: greedy removal sequence r

```
1  n ← |L|
2  r ← ℕ^{n−2}
3  heap ← MinHeap containing initial removal errors under X
4  for i = 1 to n − 2 do
5  |  v ← heap.extractMin()
6  |  update removal errors of neighbours of v
7  |  re-heapify updated errors
8  |  r[i] ← v
9  end
10 return r
```

[Theorem 4.16]

Let f_X be the runtime function of geometric distance metric X. Then for a polygonal curve with length n, the Greedy algorithm (algorithm 4) runs in $\mathcal{O}(n(f_X(n) + \log\ n)$

Proof. Initially, there exist $\Theta(n)$ shortcuts. In the beginning, all of them shortcut exactly one vertex, and therefore, this allows the computation of each error in constant time across all presented distance metrics. Afterwards, the heap can be built in linear time as well. In each iteration, we extract the minimum in constant time. Updating the neighboring errors of a removed vertex can be accomplished in $\mathcal{O}(f_X(n))$ time complexity. Due to an update of 2 values, we have to reheapify the heap, which can be done in $\mathcal{O}(\log\ n)$. As we have $n-2$ iterations, the total runtime of the algorithm 4 is given by $\mathcal{O}(n(f_X(n) + \log\ n))$.
Furthermore, it is trivial to see that the algorithm uses linear space, since the heap can be implemented using linear space. □

[Theorem 4.17]

The solution computed by the Greedy algorithm is a 4-approximation for the MinSUM problem under the Fréchet distance.

Proof. The input polygonal curve with length n is denoted as $\mathcal{L}$. We let $s_1, \ldots, s_{n-2}$ be the resulting shortcut sequence and $r_1, \ldots, r_{n-2}$ be the according removal sequence produced by the Greedy algorithm.

Furthermore, we let $s_1^*, \ldots, s_{n-2}^*$ be the optimal shortcut sequence under the SUM function, hence the one, that is produced by the MinSUM algorithm, and $r_1^*, \ldots, r_{n-2}^*$ be the optimal removal sequence. We use L_i to denote the sub-curve shortcutted by s_i. Accordingly, we use L_j^* to denote the sub-curve shortcutted by s_j^*.
Next, we assign shortcuts $s_1, \ldots, s_{n-2}$ to the shortcuts $s_1^*, \ldots, s_{n-2}^*$. To be particular, we assign a shortcut s_i to a shortcut s_j^* if we meet the following two conditions:

- Prior to the Greedy algorithm shortcutting r_i, there exist at least three vertices in L_j, including r_i, that have not been shortcutted yet.
- The index j is the smallest index that meets the previous condition.

s_{n-2}^* is always set to be $s_{1;n}$ and the first condition always holds for $s_{1;n}$, because its endpoints are never shortcutted. Hence, the assignment is well-defined.
First, we show how we can bound the error of s_i by its assigned shortcut s_j^*. According to the first condition, prior to the usage of s_i, at least three vertices are not yet shortcutted in L_j^*. We let s_m be the shortcut produced by the removal of any middle non-shortcutted vertex in L_j^*. From lemma 4.14 we know that the shortcut s_m is upper bounded by $2 \cdot d_{\mathcal{L};F}(s_j^*)$. Due to the picking of the Greedy algorithm, the shortcut error of s_i must be smaller or equal to $d_{\mathcal{L};F}(s_m)$, as s_i is currently the shortcut choice with the smallest induced error. Combining these inequalities, we now know that it holds $d_{\mathcal{L};F}(s_i) \leq d_{\mathcal{L};F}(s_m) \leq 2 \cdot d_{\mathcal{L};F}(s_j^*)$. Now, we let c_j be the number of shortcuts that has been assigned to s_j^*. Then we can use this to upper bound the total SUM error ε_{SUM} produced by the Greedy algorithm by $\varepsilon_{SUM} = \sum_{i=1}^{n-2} d_{\mathcal{L};F}(s_i) \leq \sum_{j=1}^{n-2} c_j \cdot 2 \cdot d_{\mathcal{L};F}(s_j^*)$.
Next, we show that we can upper bound all c_j by 2. To do this, for contradiction, we now assume that there exists some c_j with $c_j \geq 3$. Then there exists at least three shortcuts that have been assigned to s_j^* and we let v_1, v_2, v_3 be those vertices that are contradicted in that order. Now, we not only know that $v_1, v_2, v_3 \in L_j^*$, but we also know that there exist 2 additional non-contradicted vertices v_4, v_5 in L_j^* at the time v_3 is removed, as stated in the first condition. Furthermore, we know that at the time the optimal solution utilizes the shortcut s_j^*, only vertices v_p, r_j^*, v_q are not contradicted in L_j^*, where v_p is the left endpoint of L_j^* and v_q is the right one. Therefore, the shortcuts / line segments $s_l^* = \{v_p, r_j^*\}$ and $s_r^* = \{r_j^*, v_q\}$ are also part of the optimal solution. As v_1, v_2, v_3, v_4, v_5 are in L_j^*, at least three of them have to be in L_l^* or L_r^*. Without loss of generality we assume that L_l^* contains at least three of these vertices, which also implies that s_l^* is a shortcut and not a line segment. At the moment we remove the v_i with the smallest index in L_l^*, we know that at that time there exists 2 further non-contradicted vertices. Hence, we assign the corresponding shortcut to s_l^*, as the first condition holds and it has a smaller index than s_j^*. But this contradicts our claim that the shortcut has been assigned s_j^*. Thus, we can conclude that this cannot hold, and instead, it must hold $c_j \leq 2$. Resulting, we get this inequality:

$$\varepsilon_{SUM} \leq \sum_{j=1}^{n-2} c_j \cdot 2 \cdot d_{\mathcal{L};F}(s_j^*) \leq \sum_{j=1}^{n-2} 2 \cdot 2 \cdot d_{\mathcal{L};F}(s_j^*) = 4 \cdot \sum_{j=1}^{n-2} d_{\mathcal{L};F}(s_j^*) = 4 \cdot OPT$$

□

4.3.3 MinSUM Practical Approximation Adaption

When applying the Greedy algorithm for the Fréchet distance, the resulting runtime is in $\mathcal{O}(n^3)$. This may not be applicable in practice and thus, we need a different solution for the Fréchet distance.
It is obvious that we do not need to determine the exact errors of each shortcut. Instead, when picking

a vertex, we only need to identify the one with the smallest distance. Hence, it would suffice to find a threshold in which a shortcut error lies, because if its upper bound is smaller than the lower bounds of other ones we know that it must be minimal. Recall, that the decision oracle presented by Alt and Godau [2] is able to test, whether an error ε holds $\varepsilon \geq d_{\mathcal{L};F}(s_{ij})$. Instead of calculating the exact shortcut error, we can make use of the decision oracle, to determine a shortcut threshold for each shortcut individually, similar to the proposed interval search algorithm (algorithm 3).

According to lemma 4.14 we can use $2 \cdot d_{\mathcal{L};X}(s_{1;n}) = 2 \cdot \varepsilon$ as an upper bound. As lower bound we can choose 0, because there exists no negative shortcut. Trivially, we then can apply an interval search to tight the threshold, until we clearly estimate the vertex with the smallest induced shortcut error.

But, if two or more shortcut errors are highly similar, we could encounter an unusual runtime. This is because the decision oracle would then need to run exceptionally frequently to minimize the threshold and find the smaller shortcut distance. Thus, we adapt the search interval process, and for each shortcut we will repeatedly tighten its threshold by dividing the upper bound by some $b > 1$ until the decision oracle returns "false" or the bound is lower than $\frac{\varepsilon}{n^c}$ for some factor $c \geq 1$. When repeating this procedure for every shortcut, we then can efficiently extract one with the lowest bound by using a Min-Heap, where its inserted key is given by the tightened upper bound. It is worth to mention, that two or more shortcuts are able to have the same minimal upper bound, and we then extract some arbitrary of them. We will call this procedure the 'Practical Greedy' (short 'PGreedy') and it can be seen in algorithm 5, while the tighten procedure can be seen in algorithm 6.

Algorithm 5: *practical-greedy*$(\mathcal{L}, b, c)$

Data: polygonal curve $\mathcal{L}$, real number $b > 0$, real number $c > 0$
Result: removal sequence r

$n \leftarrow |\mathcal{L}|$
$r \leftarrow \mathbb{N}^{n-2}$
$heap \leftarrow$ MinHeap containing tightened upper bounds of the removal errors
for $i = 1$ ***to*** $n - 2$ **do**
  $v \leftarrow heap.extractMin()$
  update tightened upper bounds of removal errors of neighbours of v
  re-heapify updated values
  $r[i] \leftarrow v$
end

Algorithm 6: *tightened-upper*$(\mathcal{L}, s_{ij}, b, c)$

Data: polygonal curve $\mathcal{L}$, shortcut s_{ij}, real number $b > 0$, real number $c > 0$
Result: tightened upper bounds *bound*

$n \leftarrow |\mathcal{L}|$
$\varepsilon \leftarrow d_{\mathcal{L};F}(s_{1;n})$
$bound \leftarrow 2\varepsilon$
while $bound \geq \frac{\varepsilon}{n^c} \wedge \frac{bound}{b} \geq d_{\mathcal{L};F}(s_{ij})$ **do**
  $bound \leftarrow \frac{bound}{b}$
end
return *bound*

[Lemma 4.18]

The tighten process has at most $\log_b(2) + c \cdot \log_b(n)$ rounds.

Proof. It is obvious to see, that the tightened process has the most rounds, if the decision oracle always returns true. In these cases the tighten process stops if the bounds fails to meet $bound \geq \frac{\varepsilon}{n^c}$. The bound in the $k-th$ iteration is given by $\frac{\varepsilon}{b^k}$ because in each iteration we divide it by b.

$$\begin{aligned}
\text{tighten process stops} &\Leftarrow bound < \frac{\varepsilon}{n^c} \\
&\Leftrightarrow \frac{2 \cdot \varepsilon}{b^k} < \frac{\varepsilon}{n^c} \\
&\Leftrightarrow 2\varepsilon < \frac{\varepsilon}{n^c} \cdot b^k \\
&\Leftrightarrow 2n^c < b^k \\
&\Leftrightarrow \log_b(2n^c) < k \\
&\Leftrightarrow \log_b(2) + c \cdot \log_b(n) < k
\end{aligned}$$

□

[Theorem 4.19]

The practical Greedy algorithm runs in $\mathcal{O}(c \cdot n^2 \log_b\ n)$ using linear space for any $b > 1$, $c \geq 1$.

Proof. We can precompute $d_{\mathcal{L};X}(s_{1;n})$ in $\mathcal{O}(n^2)$ to compute the distance once, instead of computing it in each tightening process individually.
Moreover, according to lemma 4.18 we have at most $\log_b(2) + c \cdot \log_b(n)$ iterations of tightening the threshold. Each time we use the decision oracle to determine whether $\frac{bound}{b} \geq d_{\mathcal{L};F}(s_{ij})$ holds (see algorithm 6), we need $\mathcal{O}(n)$ time. The resulting total runtime of the tightening process is given by $\mathcal{O}((\log_b(2) + c \cdot \log_b(n)) \cdot n) = \mathcal{O}(c \cdot n \cdot \log_b\ n)$.
All of the initial shortcuts have a length of 2, as they shortcut exactly one vertex. Thus their tightened upper bounds can be computed in $\mathcal{O}(cn \cdot \log_b(1))$. Afterwards, we have exactly $n-2$ iterations. In each iteration the minimum is extracted in constant time, two tightened upper bounds are recalculated in and we re-heapify the heap in $\mathcal{O}(\log_2\ n)$. The resulting runtime is then given by
$\mathcal{O}(n^2 + cn \cdot \log_b(1) + (n-2) \cdot (c \cdot n \cdot \log_b\ n + \log_2\ n)) \subseteq \mathcal{O}(c \cdot n^2 \log\ n)$. □

[Theorem 4.20]

The solution computed by the practical Greedy algorithm provides an approximation of the MinSUM problem with factor $(4b + n^{1-c})$.

Proof. As there exist at most n shortcuts with an error smaller or equal to $\frac{\varepsilon}{n^c}$, their error can be upper bounded by $n \cdot \frac{\varepsilon}{n^c} = \varepsilon \cdot n^{1-c}$. It is obvious that it holds $\varepsilon \leq OPT$, since OPT includes the shortcut $s_{1;n}$

with error $d_{\mathcal{L};X}(s_{1;n}) = \varepsilon$. Hence, the error of all removals with shortcut error less or equal than $\frac{\varepsilon}{n^c}$ can be upper bounded by $\varepsilon \cdot n^{1-c} \leq OPT \cdot n^{1-c}$.
Next, we consider all shortcuts with an error larger than $\frac{\varepsilon}{n^c}$. To do this, we create the same assignment, as the one described in theorem 4.17, using the same notation. We recall, that at the time some shortcut s_i is utilized, there exists another candidate with shortcut costs, that are upper bounded by $2d_{\mathcal{L};F}(s_j^*)$, where s_j^* is the assigned shortcut of s_i. The binary search of the error lies within one factor of b, and thus, we can upper bound s_i by $b \cdot 2d_{\mathcal{L};F}(s_j^*)$. Again, the number of assigned shortcuts is still upper bounded by 2, since the presented contradiction still holds.
As a result, we are able to construct the following inequality for the practical error ε_{SUM}:

$$\varepsilon_{SUM} = \sum_{i=1}^{n-2} d_{\mathcal{L};X}(s_i) \leq \varepsilon \cdot n^{1-c} + \sum_{i=1}^{n-2} d_{\mathcal{L};X}(s_i) \leq OPT \cdot n^{1-c} + \sum_{j=1}^{n-2} c_j 2b \cdot d_{\mathcal{L};X}(s_j)$$

$$\leq OPT \cdot n^{1-c} + 4b \sum_{j=1}^{n-2} \cdot d_{\mathcal{L};X}(s_j) = OPT \cdot n^{1-c} + 4b \cdot OPT = (4b + n^{1-c}) \cdot OPT$$

□

4.4 Heuristics

Some applications are not in need of any exact solution or even an approximation bound. Instead, they need algorithms that assure a fast runtime. For example, those applications that are in constant need of re-rendering new lines in real-time. Therefore, our goal in this section is to propose heuristics that retrieve a reasonable solution in linear time.

One trivial heuristic can be achieved by using the order of vertices as contraction order. We will call this heuristic the 'In Order' heuristic (short 'IOH') . Another heuristic can be achieved by finding a random order of the contraction. We call this heuristic the 'Random Order' heuristic (short 'ROH'). Using the Fisher-Yates shuffle introduced by Durstenfeld [9], this can be achieved in linear time. The complete algorithm can be seen in algorithm 7.

Algorithm 7: *fisher-yates-shuffle*(*array*)

Data: Array *array*
Result: random permutation of *array*

```
random ← |array|
for i = n to 2 do
    j ← random integer with 1 ≤ j ≤ i
    swap array[i] and array[j]
end
return random
```

Initially, we can choose any valid ordering as the input array, while achieving the same statistical result.

Regarding the last heuristic, our objective is to discover an algorithm that evenly distributes all removals. Ideally, the last removal should be performed on the middle vertex. By repeating this process on both sides of the central vertex, we can represent it as a binary tree. Figure 5 provides an example of such a tree.

As we want to remove a node after the removal of its children in the tree we could describe the procedure with the following algorithm:

1. Construct a tree just as described
2. Remove all leafs from the tree as well as all corresponding vertices in the line in an arbitrary order
3. Repeat step 2 until tree is empty

Due to its nature, we will call this algorithm the 'Tree Order' heuristic (short 'TOH').
But, building this tree is unnecessarily complicated in practice. Instead, we can use the previous introduced recursive principle to adapt it to the following iterative variant:

Algorithm 8: $TreeOrder(\mathcal{L})$

Data: polygonal curve $\mathcal{L}$
Result: equal removal sequence r

$n \leftarrow |\mathcal{L}|$
$r \leftarrow \mathbb{N}^{n-2}$
$queue \leftarrow$ new empty Queue
$queue.enqueue((2, n-1))$
for $i = n-2$ ***to*** 1 **do**
  $(l, r) \leftarrow queue.dequeue()$
  $mid \leftarrow \lfloor (l+r)/2 \rfloor$
  $r[i] \leftarrow mid$
  if $l \leq mid - 1$ **then**
    $queue.enqueue((l, mid-1))$
  end
  if $mid + 1 \leq r$ **then**
    $queue.enqueue((mid+1, r))$
  end
end
return r

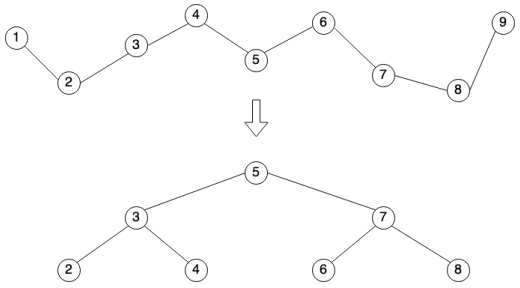


Figure 5: Tree projection of a line

[Theorem 4.21]

All 3 heuristics (IOH,ROH,TOH) run in linear time.

Proof. The order of IOH can be built trivially in linear time. Furthermore, using the removal order of the IOH as input array for the Fisher-Yates shuffle, we are able to produce a ROH in linear time as well. Finally, it is obvious that the Tree Order algorithm has $\mathcal{O}(n)$ many iterations. As we are able to perform all other operations in constant time, the total runtime is linear. □

5 Extended Objective Functions

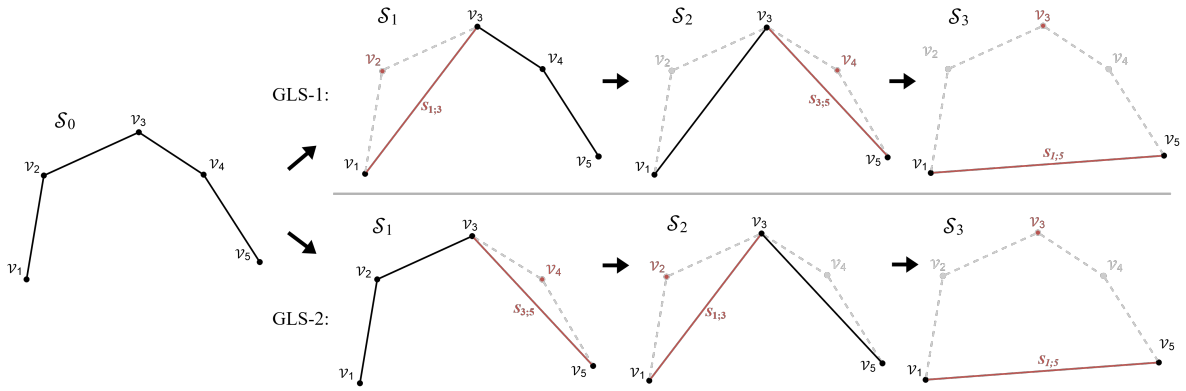


Figure 6: Two different possible gradual line simplifications

In figure 6 we can see two different gradual line simplifications for the same line. As defined in definition 4.1, the MAX function for GLS-1 evaluates to $\max\{d_{\mathcal{S}_0;X}(s_{1;3}), d_{\mathcal{S}_0;X}(s_{3;5}), d_{\mathcal{S}_0;X}(s_{1;5})\}$, while the MAX function for GLS-2 evaluates to $\max\{d_{\mathcal{S}_0;X}(s_{3;5}), d_{\mathcal{S}_0;X}(s_{1;3}), d_{\mathcal{S}_0;X}(s_{1;5})\}$. It is obvious to see that both statements are the same, because both use similar shortcuts, and therefore, their objective function errors are equal. We would encounter the same phenomenon under the SUM function. The independence of the shortcuts $s_{1;3}$ and $s_{3;5}$ causes this, as we can swap the order of taking them, while achieving the same performance. However, this can get problematic, since in most applications we should try to keep the error low as long as possible, to achieve a more accurate approximation of the curve as long as possible. To address this problem, we have to adapt the existing objective functions.

Currently, the defined objective functions MAX and SUM only consider the induced shortcut for each removal individually. Instead, we could also additionally consider shortcuts induced by previous removals. Then we can distinguish between two classes:

- **total:** For each removal, objective functions in this class consider the shortcut induced by the current removal, as well as all shortcuts induced by previous removals.
- **active:** For each simplification, objective functions in this class consider all 'active' shortcuts, that means all shortcuts that the simplification currently uses.

Furthermore, we need to combine the considered shortcuts and their induced errors to a total removal error for each simplification individually. Trivially, we can do this by using the same methods, used by the MAX and SUM functions. Then, we can build the sum of these errors to determine a total aggregated error of all simplifications. Combining total and active classes with the MAX and SUM function, we are able to introduce 4 new objective functions.

We recall that $s(\mathcal{S}_i)$ represents the shortcut that transforms $\mathcal{S}_{i-1}$ to $\mathcal{S}_i$. To simplify notation, in the following, we let $a(\mathcal{S}_i)$ denote the 'active' shortcuts of $\mathcal{S}_i$, that means those that $\mathcal{S}_i$ uses and would transform $\mathcal{S}_0$ into $\mathcal{S}_i$. Moreover, we let $t(\mathcal{S}_i)$ denote the set of 'total' shortcuts that have been used upon the i-th removal, given by $t(\mathcal{S}_i) = \bigcup_{k=1}^{i} \{s(\mathcal{S}_k)\}$.

[Definition 5.1] Objective Function - SumMaxActive (SMA)

Given a sequence of simplifications $\mathcal{S}_1, \ldots \mathcal{S}_{n-2}$ of a polygonal curve $\mathcal{S}_0$ created by a GLS, for each simplification the SumMaxActive (SMA) objective function takes the maximum induced shortcut error of the used shortcuts, and then builds the sum of these values:

$$f_{SMA}(X, \mathcal{S}_0, \ldots, \mathcal{S}_{n-2}) = \sum_{i=1}^{n-2} \max\{d_{\mathcal{S}_0;X}(s) | s \in a(\mathcal{S}_i)\}$$

[Definition 5.2] Objective Function - SumMaxTotal (SMT)

Given a sequence of simplifications $\mathcal{S}_1, \ldots \mathcal{S}_{n-2}$ of a polygonal curve $\mathcal{S}_0$ created by a GLS, for each simplification the SumMaxTotal (SMT) objective function takes the maximum induced shortcut error of the current and previous shortcuts, and then builds the sum of these values:

$$f_{SMT}(X, \mathcal{S}_0, \ldots, \mathcal{S}_{n-2}) = \sum_{i=1}^{n-2} \max\{d_{\mathcal{S}_0;X}(s) | s \in t(\mathcal{S}_i)\}$$

[Definition 5.3] Objective Function - SumSumActive (SSA)

Given a sequence of simplifications $\mathcal{S}_1, \ldots \mathcal{S}_{n-2}$ of a polygonal curve $\mathcal{S}_0$ created by a GLS, for each simplification the SumSumActive (SSA) objective function takes the sum of all active shortcut errors, and then builds the sum of these values:

$$f_{SSA}(X, \mathcal{S}_0, \ldots, \mathcal{S}_{n-2}) = \sum_{i=1}^{n-2} \sum_{s \in a(\mathcal{S}_i)} d_{\mathcal{S}_0;X}(s)$$

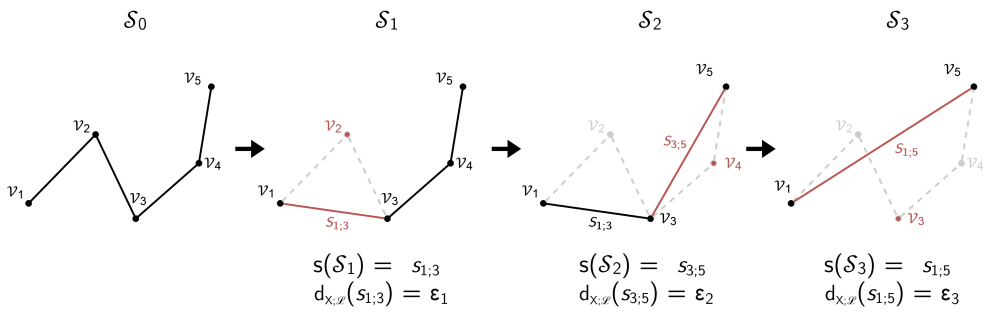
[Definition 5.4] Objective Function - SumSumTotal (SST)

Given a sequence of simplifications $\mathcal{S}_1, \ldots \mathcal{S}_{n-2}$ of a polygonal curve $\mathcal{S}_0$ created by a GLS, for each simplification the SumSumTotal (SST) objective function takes the sum of all shortcut errors of the current and previous shortcuts, and then builds the sum of these values:

$$f_{SST}(X, \mathcal{S}_0, \ldots, \mathcal{S}_{n-2}) = \sum_{i=1}^{n-2} \sum_{s \in t(\mathcal{S}_i)} d_{\mathcal{S}_0;X}(s)$$

[Example 5.5]

To get a greater understanding of all introduced objective functions, we will make an example evaluation for all functions. The example GLS that we evaluate, is given by the following:



We let ε_i denote $d_{\mathcal{L};X}(s(\mathcal{S}_i))$, as indicated in the graphic.

- MAX (definition 4.1): $f_{MAX}(X, \mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3) = \max\{\varepsilon_1, \varepsilon_2, \varepsilon_3\}$
- SUM (definition 4.2): $f_{SUM}(X, \mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3) = \varepsilon_1 + \varepsilon_2 + \varepsilon_3$
- SMA (definition 5.1): $f_{SMA}(X, \mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3) = \max\{\varepsilon_1\} + \max\{\varepsilon_1, \varepsilon_2\} + \max\{\varepsilon_3\}$
- SMT (definition 5.2): $f_{SMT}(X, \mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3) = \max\{\varepsilon_1\} + \max\{\varepsilon_1, \varepsilon_2\} + \max\{\varepsilon_1, \varepsilon_2, \varepsilon_3\}$
- SSA (definition 5.3): $f_{SSA}(X, \mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3) = (\varepsilon_1) + (\varepsilon_1 + \varepsilon_2) + (\varepsilon_3)$
- SST (definition 5.4): $f_{SST}(X, \mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3) = (\varepsilon_1) + (\varepsilon_1 + \varepsilon_2) + (\varepsilon_1 + \varepsilon_2 + \varepsilon_3)$

5.1 Divide and Conquer Approach

For the "simple" objective functions, we used the principles of divide and conquer to solve the MinMAX and MinSUM problem, using a recursive function. To be exact, we found the optimal solutions for sub-problems, then merged those solutions that together invoke the smallest error, to determine the solution of a bigger instance. We can try to apply this for the new objective functions as well, by finding a sequence of used shortcuts in sub-problems, and then assume that this sequence will also be a part of the solution for a bigger problem. Therefore, this means that if we use a shortcut s_1 first and then we use another shortcut s_2 in a sub-problem, we assume that this order is also respected in a bigger instance, and thus, the shortcut s_2 would still appear after s_1. But as discussed, we now deal with objective functions that consider the order of the removals, by contemplating previous shortcuts for each contraction as well. Hence, the solution of a sub-problem may not be part of a solution of a bigger problem. Furthermore, if we try to apply the divide and conquer principle for the "extended" objective functions, the merging process of these sub-problems is not trivial anymore, compared to the previous merging process, given by algorithm 2. Now, the order actually affects the summed error. Therefore, in the following, we firstly discuss how to properly merge two sub-problems, to minimize the resulting error.

To do this, we formally introduce notation denoting the sequence of active shortcuts and sequence of total shortcuts that a sub-problem uses. Recall the functions a and t return the active shortcuts (a) or the total shortcuts (t) of a simplification, as introduced in the beginning of section 5.

[Definition 5.6] Active sequence

Given a sub-problem $\mathcal{S}[i,j]_0$ with length $n = |i-j|+1$ and the induced simplifications $\mathcal{S}[i,j]_1, \ldots, \mathcal{S}[i,j]_{n-2}$ of some shortcut sequence. Then the active sequence $aseq(i,j)$ is a sequence consisting of active shortcut sets of the simplifications. Consequently, the k-th $aseq(i,j)$ is given by $aseq(i,j)[k] = a(\mathcal{S}[i,j]_k)$.

[Definition 5.7] Total sequence

Given a sub-problem $\mathcal{S}[i,j]_0$ with length $n = |i-j|+1$ and the induced simplifications $\mathcal{S}[i,j]_1, \ldots, \mathcal{S}[i,j]_{n-2}$ of some shortcut sequence. Then the total sequence $tseq(ij)$ is a sequence consisting of total shortcut sets of the simplifications. Hence, the k-th $tseq(i,j)$ is given by $tseq(i,j)[k] = t(\mathcal{S}[i,j]_k)$.

Moreover, using these sequences, we now define the minimal merging of two shortcut set sequences as a new problem itself. We distinguish between the merge process under the MAX function (for SMA and SMT) and the merge process under the SUM function (for SSA and SST).

[Definition 5.8] SumMaxMerge (SMM)

Given two sequences of shortcut sets $sseq1$ and $sseq2$. Let $n = |sseq1|$ be the length of the first shortcut set sequence and $m = |sseq2|$ be the length of the second one. Then the SumMaxMerge problem asks for two non-decreasing 'traversal' functions $tr1 : \mathbb{N} \mapsto \mathbb{N}$ and $tr2 : \mathbb{N} \mapsto \mathbb{N}$ with $tr1(0) = tr2(0) = 0$, $tr1(n+m) = n$, $tr2(n+m) = m$ and $tr1(i)+tr2(i) = 1+tr1(i-1)+tr2(i-1)$, while minimizing $\sum_{k=1}^{n+m} \max\{d_{\mathcal{L};X}(s_{ij}) \mid s_{ij} \in (sseq1[tr1(k)] \cup sseq2[tr2(k)])\}$.

[Definition 5.9] SumSumMerge (SSM)

Given two sequences of shortcut sets $sseq1$ and $sseq2$. Let $n = |sseq1|$ be the length of the first shortcut set sequence and $m = |sseq2|$ be the length of the second one. Then the SumSumMerge problem asks for two non-decreasing 'traversal' functions $tr1 : \mathbb{N} \mapsto \mathbb{N}$ and $tr2 : \mathbb{N} \mapsto \mathbb{N}$ with $tr1(0) = tr2(0) = 0$, $tr1(n+m) = n$, $tr2(n+m) = m$ and $tr1(i)+tr2(i) = 1+tr1(i-1)+tr2(i-1)$, while minimizing $\sum_{k=1}^{n+m} \left(\sum_{s_{ij} \in sseq1[tr1(k)]} d_{\mathcal{L};X}(s_{ij}) \right) + \left(\sum_{s_{ij} \in sseq2[tr2(k)]} d_{\mathcal{L};X}(s_{ij}) \right)$.

Using the previous definitions, we now can define functions, similar to the Minimal MAX function (4.6) and the Minimal SUM function (4.9). They describe the error of the new objective functions for some sub-problem $\mathcal{L}[i,j]$ by considering the costs of s_{ij}, as well as taking the minimal sub-problem merge into account. Therefore, we let $SMM(sseq1, sseq2)$ be the optimal SumMaxMerge error of the shortcut set sequences $sseq1$ and $sseq2$, as well as $SSM(sseq1, sseq2)$ be the optimal SumSumMerge error of them. Furthermore, we let $total(i,j)$ denote the summed error over all shortcuts used in $tseq(i,j)$, given by $total(i,j) = \sum_{s_{ij} \in T} d_{\mathcal{L},X}(s_{ij})$ where $T = \{s_{ij} \in A \mid A \in tseq(i,j)\}$.

[Definition 5.10] Divide and Conquer - SMA

Given the knowledge some solution of all sub-curves of $\mathcal{L}[i,j]$. Then the optimal solution for $\mathcal{L}[i,j]$ under SMA using these sub-solutions is given by:

$$\varepsilon_{SMA}(s_{ij}) = \begin{cases} 0 & \text{if } |i-j| \leq 1 \\ \min\limits_{i<k<j} d_{\mathcal{L};X}(s_{ij}) + SMM(aseq(i,k), aseq(k,j)) & \text{otherwise} \end{cases}$$

[Definition 5.11] Divide and Conquer - SMT

Given the knowledge some solution of all sub-curves of $\mathcal{L}[i,j]$. Then the optimal solution for $\mathcal{L}[i,j]$ under SMT using these sub-solutions is given by:

$$\varepsilon_{SMT}(s_{ij}) = \begin{cases} 0 & \text{if } |i-j| \leq 1 \\ \min\limits_{i<k<j} d_{\mathcal{L};X}(s_{ij}) + total(i,k) + total(k,j) + SMM(tseq(i,k), tseq(k,j)) & \text{otherwise} \end{cases}$$

[Definition 5.12] Divide and Conquer - SSA

Given the knowledge some solution of all sub-curves of $\mathcal{L}[i,j]$. Then the optimal solution for $\mathcal{L}[i,j]$ under SSA using these sub-solutions is given by:

$$\varepsilon_{SSA}(s_{ij}) = \begin{cases} 0 & \text{if } |i-j| \leq 1 \\ \min\limits_{i<k<j} d_{\mathcal{L};X}(s_{ij}) + SSM(aseq(i,k), aseq(k,j)) & \text{otherwise} \end{cases}$$

[Definition 5.13] Divide and Conquer - SST

Given the knowledge some solution of all sub-curves of $\mathcal{L}[i,j]$. Then the optimal solution for $\mathcal{L}[i,j]$ under SST using these sub-solutions is given by:

$$\varepsilon_{SST}(s_{ij}) = \begin{cases} 0 & \text{if } |i-j| \leq 1 \\ \min\limits_{i<k<j} d_{\mathcal{L};X}(s_{ij}) + total(i,k) + total(k,j) + SSM(tseq(i,k), tseq(k,j)) & \text{otherwise} \end{cases}$$

The resulting $aseq$ is the shortcut set sequence produced by the appendage of the shortcut set sequence of the merging algorithm and $\langle\{s_{ij}\}\rangle$. Consequently, the resulting $tseq$ is the shortcut set sequence created by the appendage of the shortcut set sequence of the merging algorithm and the set of all shortcuts (including s_{ij}). Then we should store the resulting $aseq$ or $tseq$ for each sub-problem, as it may be necessary for other merges.
Similar to the MinMAX and MinSUM algorithm (algorithm 1), we can solve the introduced equations by using a dynamic program. This is generalized for all objective functions of this section in algorithm 9.

Algorithm 9: *DivideAndConquer*$(\mathcal{L}, X, \varepsilon)$

Data: polygonal curve $\mathcal{L}$, geometric distance metric X,
objective function ε ($\varepsilon_{SMA}, \varepsilon_{SMT}, \varepsilon_{SSA}$ or ε_{SST})
Result: removal sequence r

```
n ← |L|
S ← n × n matrix with initial values 0
SSEQ ← n × n matrix with initial values ⟨⟩
K ← n × n matrix
for hop = 2 to n − 1 do
    for i = 1 to n − hop do
        j ← i + hop
        S(i, j) ← ε(s_ij)
        SSEQ(i, j) ← resulting aseq or tseq
        K(i, j) ← k that minimized ε(s_ij)
    end
end
r ← ℕ^n
seq ← shortcut sequence of SSEQ(1, n)
for k = 1 to n do
    s_ij ← seq[k]
    r[k] ← K(i, j)
end
return r
```

[Theorem 5.14]

The *DivideAndConquer* algorithm uses $\mathcal{O}(n^4)$ space and $\mathcal{O}(n^2(f_X(n) + n \cdot m(n, n)))$ time, where n is the length of the curve, f_X is the runtime function of geometric distance metric X and m is the runtime function of the minimal merge of two sub-problems.

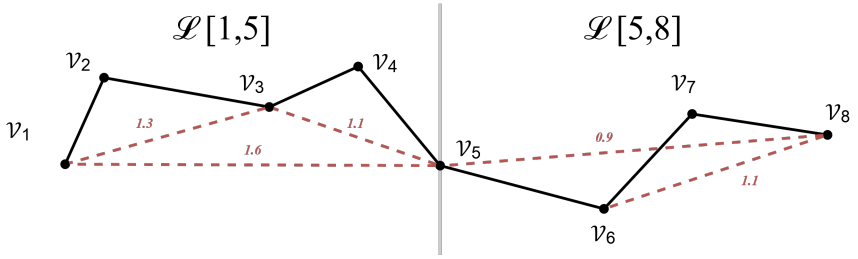
Proof. Just like the MinMAX algorithm, there exist $\mathcal{O}(n^2)$ cells. As the $SSEQ$ matrix stores the shortcut set sequence, we need a total of $\mathcal{O}(n^4)$ space, as each shortcut set sequence needs up to $\mathcal{O}(n^2)$ space.
For each cell, we have to find out the distance of the shortcut s_{ij}, which we can do in $f_X(|i-j|) \subseteq f_X(n)$ for all shortcuts. Furthermore, we have to compute the merge error for $|i-j|-1$ many possible combinations. Each merge error can be computed in $m(|k-i|, |j-k|)$ for all $i < k < j$, and it holds $m(|k-i|, |j-k|) \subseteq m(n, n)$. The final shortcut sequence can be extracted using the shortcut set sequence $SSEQ(1, n)$, by additionally remembering the used shortcut between each set during the merge. Using the K matrix, we can compute the removal sequence in linear time. Then the runtime is given by $\mathcal{O}(n^2 \cdot (f_X(n) + n \cdot m(n, n))$. □

5.2 Merge Algorithms

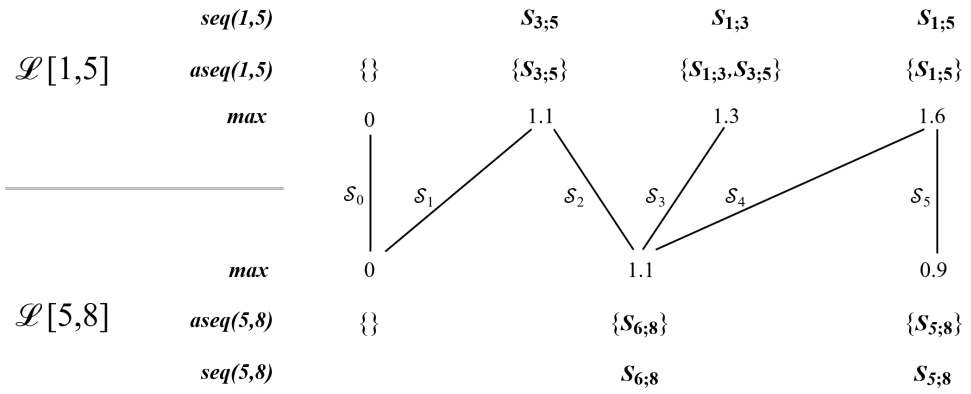
Next, it is important that we are able to solve the SumMaxMerge problem and the SumSumMerge problem.
To get a better idea of the problem overall, we will make an example for the SumMaxActive objective function. Thus, we use the active sequence (aseq) and the SumMaxMerge (as seen in definition 5.10).
In the following, we let $max(set) = max\{d_{\mathcal{L};X}(s_{ij}) \mid s_{ij} \in set\}$ be a function that returns the maximum shortcut error that exists in the given set. Equivalent, we let $sum(set) = \sum_{s_{ij} \in set} d_{\mathcal{L};X}(s_{ij})$ be a function that returns the sum of all shortcut errors in set.

[Example 5.15] Merge under SumMaxActive

Given two sub-problems $\mathcal{L}[1,5]$ and $\mathcal{L}[5,8]$, as seen in the figure below, and its shortcut sequences $seq(1,5) = \langle s_{3;5}, s_{1;3}, s_{1;5} \rangle$ and $seq(5,8) = \langle s_{6;8}, s_{5;8} \rangle$. The shortcut errors are denoted for each shortcut individually.



Its active sequences are then given by $aseq(1,5) = \langle \{s_{3;5}\}, \{s_{1;3}, s_{3;5}\}, \{s_{1;5}\} \rangle$ and $aseq(5,8) = \langle \{s_{6,8}\}, \{s_{5,8}\} \rangle$. One possible traversal of these sequences can be visualized by the following figure:



As you can see, each step of the traversal then forms a new simplification. The resulting shortcut sequence is given by $seq(1,8) = \langle s_{3;5}, s_{6;8}, s_{1;3}, s_{1;5}, s_{5;8} \rangle$. And the merge error under max is given by $\varepsilon_{SMA} = \max\{1.1, 0\} + \max\{1.1, 1.1\} + \max\{1.3, 1.1\} + \max\{1.6, 1.1\} + \max\{1.6, 0.9\} = 6.7$.

5.2.1 Exact Algorithm

Now, our task is to find the traversal functions $tr1$ and $tr2$ that minimize the SumMaxMerge / SumSumMerge, while holding its constraints as defined. We will start by finding an algorithm that solves the SumSumMerge. To achieve this, we apply the same concepts as we have applied to compute a solution for the MinMAX and MinSUM problem.

First of all, we observe that we can redefine the merge error using a recursive function.

[Definition 5.16] SSM error recursion

The SumSumMerge error for simplifications $\mathcal{S}_0, \ldots, \mathcal{S}_i$ can be computed with

$$SMM(i) = \begin{cases} 0 & \text{if } i = 0 \\ sum(sseq1[tr1(i)] \cup sseq2[tr2(i)]) + SMM(i-1) & \text{otherwise} \end{cases}$$

Next, we notice that due to the traversal properties, we know that if we choose $tr1(i) = a$ and $tr2(i) = b$, then it holds $tr1(i-1) = a-1 \wedge tr1(i-1) = b$ or $tr1(i-1) = a \wedge tr1(i-1) = b-1$. Therefore, we can choose the one that induces a smaller error, to determine the smallest total induced error.

[Lemma 5.17] Minimal SSM

Given two sub-problems $\mathcal{L}_1$ and $\mathcal{L}_2$ and their shortcut set sequences $sseq1$ and $sseq2$. If we chose $tr1(i) = a$ and $tr2(i) = b$ with $a, b \geq 0$, then the minimal error of merged simplifications $\mathcal{S}_0, \ldots, \mathcal{S}_{a+b}$ under the SumSumMerge is given by

$$SSM(a,b) = \begin{cases} 0 & \text{if } a = b = 0 \\ sum(sseq1[a]) + SSM(a-1, 0) & \text{if } b = 0 \wedge a \neq 0 \\ sum(sseq2[b]) + SSM(0, b-1) & \text{if } a = 0 \wedge b \neq 0 \\ sum(sseq1[a] \cup sseq2[b]) + \min\{SSM(a-1,b), SSM(a,b-1)\} & \text{otherwise} \end{cases}$$

Proof. The first 3 cases are trivial. We proof the 4th case using contradiction. Thus, we assume that there exists $sseq1$ and $sseq2$, such that $sum(sseq1[a] \cup sseq2[b]) + min\{SSM(a-1,b), SSM(a,b-1)\}$ is not minimal. We know the correctness of the recursion from the definition.
Since $sum(sseq1[a] \cup sseq2[b])$ is a constant error of the simplification $\mathcal{S}_{a+b}$ and independent of the choice $SSM(a-1,b)$ and $SSM(a,b-1)$, the choice between $SSM(a-1,b)$ and $SSM(a,b-1)$ must then not be minimal. This means the one not chosen by the min is minimal, which is retrieved using max. Then it holds $\max\{SSM(a-1,b), SMM(a,b-1)\} < \min\{SSM(a-1,b), SSM(a,b-1)\}$. This cannot hold, due to the property of the min and max function. □

Due to the characteristics of the traversal, the last merge holds $tr1(|sseq1| + |sseq2|) = |sseq1|$ and $tr2(|sseq1| + |sseq2|) = |sseq2|$. Hence, the total merge error is given by $SSM(|sseq1|, |sseq2|)$. This can be solved easily using a dynamic program again. Similar to the other dynamic programs that we

proposed, we recommend using a matrix that stores the error for each combination, then computing this matrix from bottom up.
This can be seen in algorithm 10.

Algorithm 10: SumSumMerge($sseq1, sseq2$)

Data: shortcut set sequence $sseq1$, shortcut set sequence $sseq2$
Result: error matrix ε

$n \leftarrow |sseq1|$
$m \leftarrow |sseq2|$
$\varepsilon \leftarrow (n+1) \times (m+1)$ matrix with initial values 0
for $i = 1$ ***to*** n **do**
  $\varepsilon(i, 0) \leftarrow \varepsilon(i-1, 0) + sum(sseq1[i])$
end
for $j = 1$ ***to*** m **do**
  $\varepsilon(0, j) \leftarrow \varepsilon(0, j-1) + sum(sseq2[j])$
end
for $i = 1$ ***to*** n **do**
  for $j = 1$ ***to*** m **do**
    $\varepsilon(i, j) \leftarrow sum(sseq1[i]) + sum(sseq2[j]) + \min\{\varepsilon(i-1, j), \varepsilon(i, j-1)\}$
  end
end
return ε

The total error $SMM(|sseq1|, |sseq2|)$ is then given by $\varepsilon(|sseq1|+1, |sseq2|+1)$. Comparable to the MinMAX / MinSUM algorithm, using the error matrix, we can backtrack the shortcut sequence. We can do this by starting at the cell $\varepsilon(n+1, m+1)$, then repeatedly finding the origin cell used for the calculation, by finding the previous cell with the smaller error. Furthermore, it is important that we retrieve the resulting shortcut set sequence, as it may be necessary for other merges. We can see the backtracking algorithm adressing this problem in algorithm 11.

Algorithm 11: backtrack(ε)

Data: error matrix ε
Result: resulting $sseq$

$(n+1, m+1) \leftarrow$ dimension of ε
$sseq \leftarrow$ Array of $n+m$ elements
for $i = n+m$ *to* 1 **do**
  $sseq[i] \leftarrow sseq1[n] \cup sseq2[m]$
  $max(sseq[i]) \leftarrow max\{max(sseq1[n]), max(sseq2[m])\}$
  $sum(sseq[i]) \leftarrow sum(sseq1[n]) + sum(sseq2[m])$
  $c_1 \leftarrow$**if** $n > 0$ **then** $\varepsilon(n, m+1)$ **else** ∞
  $c_2 \leftarrow$**if** $m > 0$ **then** $\varepsilon(n+1, m)$ **else** ∞
  if $c_1 \leq c_2$ **then**
    $n \leftarrow n-1$
  else
    $m \leftarrow m-1$
  end
end
return $sseq$

[Theorem 5.18]

For two shortcut set sequences $sseq1$ and $sseq2$, with $|sseq1| = n$ and $|sseq2| = m$, the minimal SumSumMerge can be solved in $\mathcal{O}(nm)$.

Proof. Algorithm 10 computes exactly $(n+1)(m+1)$ many cells. Assuming that the sum of a set can be retrieved in constant time (by saving it to each shortcut set individually), each computation is in constant time, and the resulting time is in $\mathcal{O}((n+1)(m+1)) = \mathcal{O}(nm)$.
Furthermore, algorithm 11 has $n+m$ iterations. In iteration we can perform the union of both sets in constant time. Moreover, as the max and sum is stored for each set individually, we can retrieve the max and sum of the resulting sequence for each set in constant time. Hence, the resulting total runtime is given by $\mathcal{O}((nm) + (n+m)) = \mathcal{O}(nm)$. □

Next we solve the SumMaxMerge. It is quite obvious that we can translate the recursive function of the SumSumMerge (definition 5.16) for the SumMaxMerge.

[Definition 5.19] SMM error recursion

The SumMaxMerge error for simplifications $\mathcal{S}_0, \ldots, \mathcal{S}_i$ can be computed with

$$SMM(i) = \begin{cases} 0 & \text{if } i = 0 \\ max(sseq1[tr1(i)] \cup sseq2[tr2(i)]) + SMM(i-1) & \text{otherwise} \end{cases}$$

Similar to Lemma 5.17, we find the minimal traversal by applying the same decision rule.

[Lemma 5.20] Minimal SMM

Given two sub-problems $\mathcal{L}_1$ and $\mathcal{L}_2$ and their shortcut set sequences $sseq1$ and $sseq2$. If we chose $tr1(i) = a$ and $tr2(i) = b$ with $a, b \geq 0$, then the minimal error of merged simplifications $\mathcal{S}_0, \ldots, \mathcal{S}_{a+b}$ under SumMaxMerge is given by

$$SMM(a,b) = \begin{cases} 0 & \text{if } a = b = 0 \\ max(sseq1[a]) + SMM(a-1, 0) & \text{if } b = 0 \wedge a \neq 0 \\ max(sseq2[b]) + SMM(0, b-1) & \text{if } a = 0 \wedge b \neq 0 \\ max(sseq1[a] \cup sseq2[b]) + min\{SMM(a-1,b), SMM(a,b-1)\} & \text{otherwise} \end{cases}$$

Proof. Similar to proof of Lemma 5.17. □

Then we can trivially adapt algorithm 10 for the SumMaxMerge by changing each occurrence of the *sum* function to the *max* function (lines 5,8 and 12), while also minimizing the error. Resulting, we achieve

the same runtime for the SumMaxMerge as for the SumSumMerge. But in the following we will explore how we can retrieve the exact SumMaxMerge, using a more greedy approach.

5.2.2 Greedy Top Down

The minimal recursive function, as defined for the SMM and SSM, prerequisites the knowledge of previous cells. Instead, we propose greedy picking, by moving into that direction that currently achieves a greater error. To be exact, if we are in the cell $\varepsilon(i,j)$, we choose $\varepsilon(i-1,j)$ as the previous cell if $max(sseq1[i]) \geq max(sseq2[j])$, else we choose $\varepsilon(i,j-1)$. We call this algorithm the 'Greedy Top Down' (short GTD) and it can be seen for the SumMaxMerge in the following.

Algorithm 12: SumMaxMerge-GTD($sseq1, sseq2$)

Data: shortcut set sequence $sseq1$, shortcut set sequence $sseq2$
Result: GTD error ε, resulting sseq

```
1  i ← |sseq1|
2  j ← |sseq2|
3  ε ← 0
4  sseq ← array with n + m space
5  while i ≠ 0 ∨ j ≠ 0 do
6  |   sseq[i + j] ← sseq1[i] ∪ sseq2[j]
7  |   ε ← ε + max(sseq1[i] ∪ sseq2[j])
8  |   if max(sseq1[i]) ≤ max(sseq2[j]) ∧ i > 0 then
9  |   |   i ← i − 1
10 |   else
11 |   |   j ← j − 1
12 |   end
13 end
```

This algorithm can be trivially adapted for the SumSumMerge, by replacing each occurrence of the *max* function with the *sum* function.

[Theorem 5.21]

The GTD under SumMaxMerge and SumSumMerge can be computed in $\mathcal{O}(n+m)$.

Proof. Initially it holds $i = n$ and $j = m$ and in each iteration we decrement exactly one of them. Since i and j do not get negative, we have $n+m$ many iterations. In each iteration we apply the max/sum, as well as the union in constant time. Hence, the resulting runtime is given by $\mathcal{O}(n+m)$. □

[Lemma 5.22]

The GTD error under SumMaxMerge evaluates to $\left(\sum_{i=1}^{n} max(sseq1[i])\right) + \left(\sum_{j=1}^{m} max(sseq2[j])\right)$.

Proof. We let $\varepsilon_1 + \varepsilon_2 + \cdots + \varepsilon_{n+m}$ be the sum built by the objective function of the SumMaxMerge. Then ε_k is given by $\max\{d_{\mathcal{L};X}(s_{ij}) \mid s_{ij} \in (sseq1[tr1(k)] \cup sseq2[tr2(k)])\}$. It is obvious that this is equal to $\max\{max(sseq1[tr1(k)]), max(sseq2[tr2(k)])\}$.
Case 1 $\max\{max(sseq1[tr1(k)]), max(sseq2[tr2(k)])\} = max(sseq1[tr1(k)])$: Then the algorithm moves backwards in the $tr1$ function, given by $tr1(k-1) = tr1(k) - 1$. Resulting, no other ε_b with $b < k$ can evaluate to the term $max(sseq1[a])$ with $a = tr1(k)$.
Case 2 $\max\{max(sseq1[tr1(k)]), max(sseq2[tr2(k)])\} = max(sseq2[tr2(k)])$: The same logic holds here. That means that also no other ε_b with $b < k$ can evaluate to the term $max(sseq2[a])$ with $a = tr2(k)$.
Therefore, generally speaking, each evaluated term $max(sseq1[a])$ for all $1 \leq a \leq n$ and $max(sseq2[b])$ for all $1 \leq b \leq m$ can appear only once at most. Since the sum consists of exactly $n + m$ elements, each term then appears exactly once. Resulting, $\varepsilon_1 + \cdots + \varepsilon_2$ is the sum over all $max(sseq1[a])$ and over all $max(sseq1[b])$ with $1 \leq a \leq n$ and $1 \leq b \leq m$. As the sum is commutative, an ordered version of this sum, as the one given in the lemma, retrieves the same value. □

[Lemma 5.23]

The error $SMM(i,j)$ of the SumMaxMerge (adapted version of algorithm 9) is lower bounded by $\left(\sum_{k=1}^{i} max(sseq1[k])\right) + \left(\sum_{k=1}^{j} max(sseq2[k])\right)$ for any traversal.

Proof. We will show this using induction over the number of elements $i + j$ that we have traversed to get to the cell. Then our induction hypothesis is given by

$$IH: \; SMM(i,j) \geq \left(\sum_{k=1}^{n} max(sseq1[k])\right) + \left(\sum_{k=1}^{m} max(sseq2[k])\right)$$

.
$\mathbf{i + j = 0}$: We know that $SMM(0,0)$ is given by 0, since we have not taken any shortcuts and thus do not induce any error. This is equal to the sum of no elements.
$\mathbf{i + j - 1 \to i + j}$: Then $SMM(i,j)$ has two possible predecessors. Without loss of generality we assume that $SMM(i-1,j)$ is the predecessor (the following can also be shown if $SMM(i,j-1)$ is the predecessor). Using the recursive definition, we can form the following inequality:

$$\begin{aligned}
SMM(i,j) &= \max\{max(sseq1[i]), max(sseq2[j])\} + SMM(i-1,j) \\
&\overset{(IH)}{\geq} \max\{max(sseq1[i]), max(sseq2[j])\} + \left(\sum_{k=1}^{i-1} max(sseq1[k])\right) + \left(\sum_{k=1}^{j} max(sseq2[k])\right) \\
&\geq max(sseq1[i]) + \left(\sum_{k=1}^{i-1} max(sseq1[k])\right) + \left(\sum_{k=1}^{j} max(sseq2[k])\right) \\
&= \left(max(sseq1[i]) + \sum_{k=1}^{i-1} max(sseq1[k])\right) + \left(\sum_{k=1}^{j} max(sseq2[k])\right) \\
&= \left(\sum_{k=1}^{i} max(sseq1[k])\right) + \left(\sum_{k=1}^{j} max(sseq2[k])\right)
\end{aligned}$$

□

[Corollary 5.24]

The SumMaxMerge-GTD retrieves a minimal merge.

Proof. The lower bound (Lemma 5.23) of the total SMM error $SMM(n, m)$ is achieved by the GTD (Lemma 5.22). □

Using this knowledge, we now show that the Divide and Conquer approach is optimal for the SumMaxActive problem.

[Lemma 5.25]

Given the optimal active sequence $aseq(i, j)$ induced by a sub-problem $\mathcal{L}[i, j]$. Then the SumMaxMerge error $\varepsilon_{SMA}(s_{ij})$ is given by $\sum_{a \in aseq(i,j)} max(a)$.

Proof. We let $rseq$ be the resulting sequence of an optimal merge. As discussed, the active sequence $aseq(i, j)$ of a problem is given by $rseq \cdot \langle\{s_{ij}\}\rangle$.

$$
\begin{aligned}
\varepsilon_{SMA}(s_{ij}) &= \min_{i<k<j} d_{\mathcal{L};X}(s_{ij}) + SMM(aseq(i,k), aseq(k,j)) && 5.10 \\
&= d_{\mathcal{L};X}(s_{ij}) + \min_{i<k<j} SMM(aseq(i,k), aseq(k,j)) \\
&= d_{\mathcal{L};X}(s_{ij}) + \sum_{k=1}^{n+m} \max\{d_{\mathcal{L};X}(s_{ij}) \mid s_{ij} \in (sseq1[tr1(k)] \cup sseq2[tr2(k)])\} && 5.8 \\
&= max(\langle\{s_{ij}\}\rangle) + \sum_{k=1}^{n+m} max(rseq[k]) \\
&= \sum_{a \in (rseq \cdot \langle\{s_{ij}\}\rangle)} max(a) \\
&= \sum_{a \in aseq(i,j)} max(a)
\end{aligned}
$$

□

[Lemma 5.26]

The minimal recursive function for the SumMaxActive problem using the Divide and Conquer approach for a shortcut s_{ij} is given by

$$
\varepsilon_{SMA}(s_{ij}) = \begin{cases} 0 & \text{if } |i-j| \leq 1 \\ \min_{i<k<j} d_{\mathcal{L};X}(s_{ij}) + \varepsilon_{SMA}(s_{ik}) + \varepsilon_{SMA}(s_{kj}) & \text{otherwise} \end{cases}
$$

Proof. The first case is trivial. For the second case we have the following:

$$\begin{aligned}
\varepsilon_{SMA}(s_{ij}) &= \min_{i<k<j} d_{\mathcal{L};X}(s_{ij}) + SMM(aseq(i,k), aseq(k,j)) && 5.10\\
&= \min_{i<k<j} d_{\mathcal{L};X}(s_{ij}) + \left(\sum_{a\in aseq(i,k)} max(a)\right) + \left(\sum_{a\in aseq(i,k)} max(a)\right) && 5.24 \text{ and } 5.22\\
&= \min_{i<k<j} d_{\mathcal{L};X}(s_{ij}) + \varepsilon_{SMA}(s_{ik}) + \varepsilon_{SMA}(s_{kj}) && 5.25
\end{aligned}$$

□

[Corollary 5.27]

The minimal SumMaxActive error of the Divide and Conquer approach is equal to the MinSUM error.

Proof. The recursive function of the minimal SMA error (5.26) is equal to the recursive function of MinSUM (4.9). The resulting errors must then also be equal. □

[Lemma 5.28]

The SMA error is lower bounded by the SUM error.

Proof. If we take a shortcut in the i-th simplification, the shortcut is then active and must be in the active set of that shortcut. Therefore, it holds $s(\mathcal{S}_i) \in a(\mathcal{S}_i)$. Then we can make the following inequality.

$$\begin{aligned}
f_{SMA}(X, \mathcal{S}_0, \ldots, \mathcal{S}_{n-2}) &= \sum_{i=1}^{n-2} \max\{d_{\mathcal{S}_0;X}(s) | s \in a(\mathcal{S}_i)\} && 5.1\\
&\geq \sum_{i=1}^{n-2} d_{\mathcal{S}_0;X}(s(\mathcal{S}_i))\\
&= f_{SUM}(X, \mathcal{S}_0, \ldots, \mathcal{S}_{n-2}) && 4.1
\end{aligned}$$

□

[Theorem 5.29]

The Divide and Conquer approach is optimal for the SumMaxActive problem.

Proof. From Lemma 5.28 we know that the SMA error is lower bounded by the MinSUM error. According to Corollary 5.27 the error of the recursive function of the Divide and Conquer Approach returns the same error as the minimal SUM error. As it achieves its lower bounds, it is then minimal. □

From now on, we will refer the Divide and Conquer approach for the SumMaxActive as the 'MinSMA algorithm', as it retrieves the optimal solution under SMA.

5.2.3 Greedy Bottom Up

In addition to the 'Greedy Top Down' Merge, for comparison reasons, we also present a Bottom Up alternative. Therefore, this greedy variant starts with the cell $\varepsilon(0,0)$, as we have $tr1(0) = tr2(0) = 0$, and then repeatedly travels from a cell $\varepsilon(a,b)$ to the cell $\varepsilon(a+1,b)$ or $\varepsilon(a,b+1)$, depending which of those cells invokes a smaller error. Analogous to the GTD, this algorithm is called the 'Greedy Bottom Up' (short GBU) and it can be seen for the SumSumMerge in the following algorithm:

Algorithm 13: SumSumMerge-GBU($sseq1, sseq2$)

Data: shortcut set sequence $sseq1$, shortcut set sequence $sseq2$
Result: GBU error ε, resulting sseq

```
1  n ← |sseq1|
2  m ← |sseq2|
3  i, j ← 0
4  ε ← 0
5  sseq ← array with n + m space
6  while i ≠ n ∨ j ≠ m do
7  |  c_i ← if i ≠ n then sum(sseq1[i + 1] ∪ sseq2[j]) else ∞
8  |  c_j ← if j ≠ m then sum(sseq1[i] ∪ sseq2[j + 1]) else ∞
9  |  if c_i ≤ c_j then
10 |  |  i ← i + 1
11 |  else
12 |  |  j ← j + 1
13 |  end
14 |  sseq[i + j] ← sseq1[i] ∪ sseq2[j]
15 |  ε ← ε + sum(sseq1[i] ∪ sseq2[j])
16 end
17 return ε, sseq
```

The algorithm can be implemented for the SumMaxMerge, by replacing each occurrence of the *sum* function with the *max* function (lines 7,8 and 15).

[Theorem 5.30]

The GBU under SumMaxMerge and SumSumMerge can be computed in $\mathcal{O}(n+m)$.

Proof. It is obvious to see, that each iteration we increment either i or j. Since i cannot exceed n and j cannot not exceed m, because costs c_i and c_j get infinite, we know we have exactly $n+m$ iterations. In each iteration we can compute the costs in constant time. Furthermore, the union can be implemented to work in constant time as well. Resulting, the total runtime is given by $\mathcal{O}(n+m)$. □

5.3 Greedy Candidate-Estimation algorithm

The Divide and Conquer algorithm (algorithm 9) performs a merge for every possible candidate of k. Therefore, it performs $\mathcal{O}(n)$ many merges for $\mathcal{O}(n^2)$ cells, which accumulates to $\mathcal{O}(n^3)$ total merges. In total, using the GTD as merge algorithm we achieve a runtime of $\mathcal{O}(n^4)$ for the SumMax problems. However, for the SumSum problems, we necessitate $\mathcal{O}(n^5)$ time, as the dynamic program needs to be used in every merge. It is quite obvious that in many cases we do not need to perform an exact merge algorithm for every candidate k. This is because we could ignore pairings of sub-problems that already exhibit a great error that is much worse than the one of other pairings. Hence, the Divide an Conquer algorithm possibly performs more merges as needed.

Another alternative can be achieved, by estimating how well two sub-problems perform, and then merge only those, that realize the best estimation. Then we are able to save valuable time, if the estimation runs faster than the merge algorithm. We call this algorithm the 'Greedy Candidate Estimation' (short GCE) algorithm and it can be seen in the following:

Algorithm 14: *GreedyCandidateEstimation*$(\mathcal{L}, X, estimation, merge)$

Data: polygonal curve $\mathcal{L}$, geometric distance metric X, *estimation* function, *merge* algorithm
Result: removal sequence r

```
1  n ← |L|
2  S ← n × n matrix with initial values 0
3  SSEQ ← n × n matrix with initial values ⟨⟩
4  K ← n × n matrix
5  for hop = 2 to n − 1 do
6  |  for i = 1 to n − hop do
7  |  |  j ← i + hop
8  |  |  candidateK ← 0
9  |  |  candidateEst ← ∞
10 |  |  for k = i + 1 to j − 1 do
11 |  |  |  est ← estimation(SSEQ(i, k), SSEQ(k, j))
12 |  |  |  if est < candidateEst then
13 |  |  |  |  candidateEst ← est
14 |  |  |  |  candidateK ← k
15 |  |  |  end
16 |  |  end
17 |  |  S(i, j) ← error of merge(SSEQ(i, candidateK), SSEQ(candidateK, j))
18 |  |  SSEQ(i, j) ← resulting aseq or tseq of merge
19 |  |  K(i, j) ← candidateK
20 |  end
21 end
22 r ← ℕ^n
23 seq ← shortcut sequence of SSEQ(1, n)
24 for k = 1 to n do
25 |  s_ij ← seq[k]
26 |  r[k] ← K(i, j)
27 end
28 return r
```

[Theorem 5.31]

The Greedy Candidate Estimation algorithm runs in $\mathcal{O}(n^2(f_X(n)+n\cdot g_e(n,n)+g_m(n,n)))$, where f_X is the runtime function of geometric distance measure X, g_e is the runtime function of the *estimation* function and g_m is the runtime function of the *merge* algorithm.

Proof. For each cell we compute the according shortcut error in $f_X(n)$ time. Furthermore, we have $\mathcal{O}(n)$ possible candidates to choose from. For each candidate we retrieve its estimation in $g_e(n,n)$ time. Moreover, we then merge the subproblems, that induce the lowest estimation error, in $\mathcal{O}(g_m(n,n))$ time. Therefore, we need $\mathcal{O}(f_X(n)+n\cdot g_e(n,n)+m_e(n,n))$ time to compute the value of each cell.
The backtracking of the removal sequence is similar to the backtracking of the Divide and Conquer algorithm and hence, needs linear time. We have $\mathcal{O}(n^2)$ cells and therefore, the total runtime is given by $\mathcal{O}(n^2(f_X(n)+n\cdot g_e(n,n)+m_e(n,n)))$. □

Next, we explore reasonable choices for the estimation and merge function.

For the SumSumActive and SumSumTotal problem one obvious choice is to pick the GTD or GBU algorithm as an estimation, as it is able to determine a merge error in linear time. Then we can choose the exact SSM as merge algorithm. We call these algorithms {SSA/SST}-{GTD/GBU}-Exact, depending on the algorithm choice (e.g. SSA-GBU-Exact).

[Theorem 5.32]

The {SSA/SST}-GTD-Exact and the {SSA/SST}-GBU-Exact have a runtime that is in $\mathcal{O}(n^4)$.

Proof. The estimation function runtime ($g_e(n,n)$) of the GTD/GBU is in $\mathcal{O}(n)$. Furthermore, we are able to perform a merge in $\mathcal{O}(n^2)$ (runtime of $g_m(n,n)$). Therefore, according to theorem 5.31, the GCE with the GTD/GBU and the exact SSM (GCE-GTD-Exact/GCE-GBU-Exact) achieves a total runtime of $\mathcal{O}(n^2(f_X(n)+n\cdot n+n^2))=\mathcal{O}(n^4)$ for all introduced geometric distance metrics. □

Now, we show a very handy choice for the SMA and for the SMT.

[Theorem 5.33]

For the SumMaxActive and SumMaxTotal problem, we can retrieve the same solution as the one produced by the divide and conquer approach in $\mathcal{O}(n^2(f_X(n)+n))$ using the GCE algorithm.

Proof. For the SumMaxActive and SumMaxTotal problem we recall that the GTD merge error is given by the sum of all maximum shortcut distances of the shortcut set sequence (lemma 5.22). Then we can use this property as the estimation function to determine the pair that produces the minimal merge, by finding the pair that induces the lowest sum. Furthermore, for the merge algorithm we choose the GTD, as it is able to determine an optimal merge in linear time for both SumMax problems. The estimation runtime can be implemented in constant time, by storing the sum of each shortcut set sequence individually. Then, according to theorem 5.31, the resulting runtime is given by $\mathcal{O}((n^2(f_X(n)+n))$. □

This means for proposed geometric distance metrics, we are able to solve the Divide and Conquer solutions of both SumMax problems in $\mathcal{O}(n^3)$, using the GCE approach. With current algorithms and estimations, we are not able to make make a reasonable estimation for the SumSumMerge, while achieving a cubic runtime in total. This is because we are not able to estimate the SumSumMerge error in constant time. Therefore, in the following, we explore how we could estimate for the SumSumMerge in a reasonable manner in $\mathcal{O}(1)$.

To do this we analyze how the SSM error emerges. Currently, the error is computed by the dynamic program, introduced in algorithm 10. The error can be viewed as the minimal sum of the path from the cell (0,0) to the cell (n,m). This can be seen in the following figure.

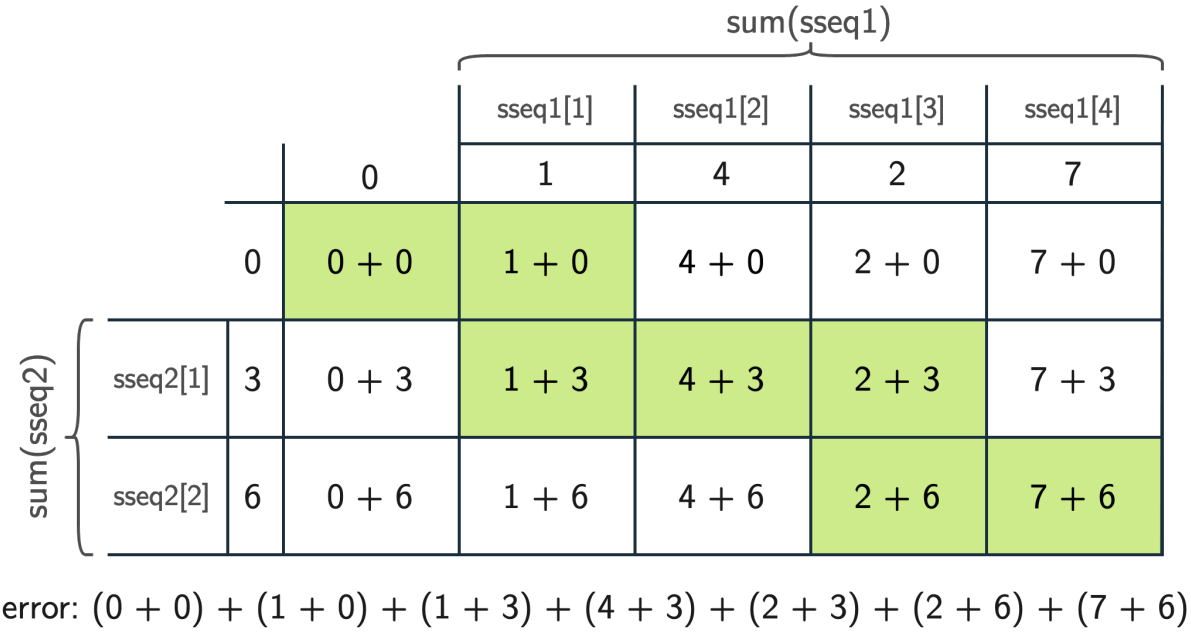


Figure 7: The figure shows the table produced by the dynamic program of a SumSumMerge, with a chosen path from cell (0,0) to (4,2), indicated by the green highlighted boxes. In each cell we can see the induced error of the cell itself, while the total merge error of the path can be seen below the table.

It is obvious that the creation of the merge error can also be seen as the sum the product of each element and its number of occurrences. Formally, we can then define the merge error as follows.

[Definition 5.34]

We let $occ1(k)$ and $occ2(k)$ be the number of occurrences of $sum(sseq1[k])$ and $sum(sseq2[k])$. Then the error of the SumSumMerge using the exact dynamic program is given by:

$$\left(\sum_{k=0}^{n} sum(sseq1[k]) \cdot occ1(k)\right) + \left(\sum_{k=0}^{m} sum(sseq2[k]) \cdot occ2(k)\right)$$

We now try to find appropriate values for $occ1$ and $occ2$. To do this we use a statistical approach, by using the mean number of occurrences of an error. First, we observe that the path consists of $n+m+1$ cells. Furthermore, there are exactly $n+1$ many columns, and therefore, the error of each column appears $\frac{n+m+1}{n+1}$ times in the sum on average. Respectively, it is obvious that then the error of any row appears $\frac{n+m+1}{m+1}$ times on average. Hence, we can set $occ1$ and $occ2$ to these values to find an estimation that should be statistically similar to the real error.

[Definition 5.35] SSM mean estimation (MEST)

The mean estimation (short MEST) of the SumSumMerge error for two shortcut set sequences $sseq1$ and $sseq2$ with lengths n and m is given by:

$$\frac{n+m+1}{n+1} \cdot \left(\sum_{k=1}^{n} sum(sseq1[k]) \right) + \frac{n+m+1}{m+1} \cdot \left(\sum_{k=1}^{n} sum(sseq1[k]) \right)$$

[Theorem 5.36]

When using the MEST as estimation function in the GCE algorithm, we can compute it in $\mathcal{O}(1)$.

Proof. In the merge process we compute $\sum_{k=1}^{n} sum(sseq[k])$ of the resulting shortcut set sequence. If we denote this value for each shortcut set individually, we can compute the MEST in constant time. □

Now, we can now introduce new GCE variants for the SumSum problems. They use the MEST as estimation function and the GTD/GBU as merge algorithm. We call those variants GCE-MEST-GTD and GCE-MEST-GBU.

[Theorem 5.37]

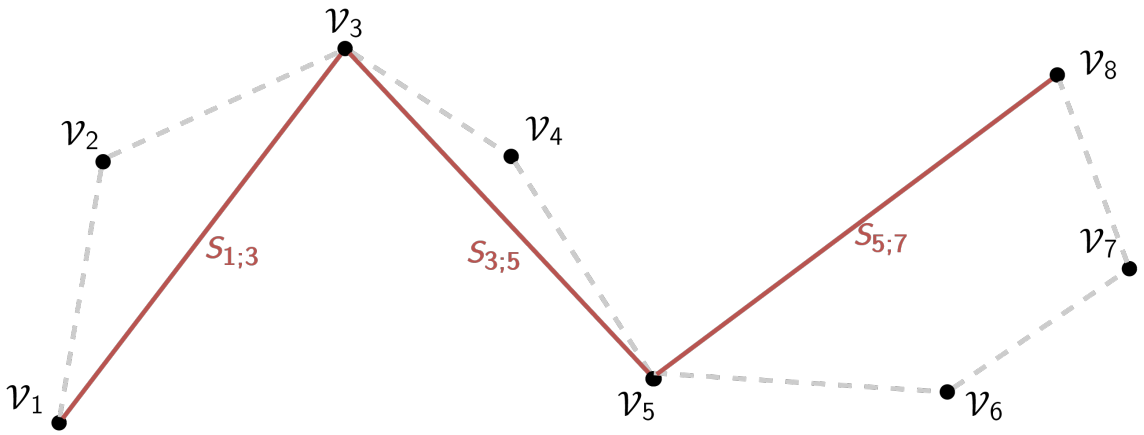
The GCE-MEST-GTD and the GCE-MEST-GBU have a runtime that is in $\mathcal{O}(n^2 \cdot (f_X(n) + n))$, where f_X is the runtime function of geometric distance measure X.

Proof. From the theorem 5.36 we know that we can compute the MEST in constant time. Furthermore, the merge algorithm runtime of the GTD/GBU is in $\mathcal{O}(n)$. Therefore, according to theorem 5.31, the GCE with the MEST and the GTD/GBU (GCE-MEST-GTD/GCE-MEST-GBU) achieves a total runtime of $\mathcal{O}(n^2(f_X(n) + n \cdot 1 + n)) = \mathcal{O}(n^2 \cdot (f_X(n) + n))$ for all introduced geometric distance metrics. □

5.4 Greedy Difference Algorithm

Each time we use a shortcut two line segments are forfeited, and instead, a new line segment is produced. The proposed Greedy algorithm (algorithm 4) repeatedly takes the shortcut that induces the lowest error. But it only accounts the error introduced by the newly created segment. Consequently, it may choose a shortcut that eliminates two line segments which, in themselves, have already resulted in a relatively low error. However, when using extended objective functions, the error of multiple shortcuts is considered for each contraction. Then the previous Greedy algorithm may not be applicable anymore. The reason behind this is that we would prefer to pick a shortcut that eliminates two line segments that currently induce a high error, even tho the error of the chosen shortcut may not be the lowest.

To realize this, we can adapt the current Greedy algorithm, by changing the picking error, that we use in the min-heap. Currently, the picking error is given by the shortcut error. But additionally, in the extended case, we have to account the error of the forfeited line segments as well. Therefore, instead, we could simply subtract both forfeited line segment errors from the shortcut error, since a high error of them should affect the picking error positively, and thus minimize it. This way we are able to balance the produced shortcut error and the induced removal of shortcut errors. As we now pick the one with the minimal difference, we call the adaptation the 'Greedy Difference' (short 'GD') algorithm. An example for the evaluation of the difference error can be seen in the following figure.



$$\varepsilon_{\text{diff}}(v_3) = d_{\mathscr{L};X}(s_{1;5}) - d_{\mathscr{L};X}(s_{1;3}) - d_{\mathscr{L};X}(s_{3;5})$$

$$\varepsilon_{\text{diff}}(v_5) = d_{\mathscr{L};X}(s_{3;8}) - d_{\mathscr{L};X}(s_{3;5}) - d_{\mathscr{L};X}(s_{5;8})$$

Figure 8: The figure shows the a polygonal line with 3 removed vertices. Furthermore, it shows the shortcuts (red) and the previous discussed picking errors of 2 potential candidates.

[Theorem 5.38]

The Greedy Difference algorithm has a runtime of $\mathcal{O}(n(f_X(n) + \log\ n)$.

Proof. As mentioned, the algorithm is an adaptation of algorithm 4. Unlike the Greedy algorithm, it inserts the introduced difference error in the heap, rather than the plain shortcut error. With the knowledge of all three shortcut errors, the difference error can be computed in constant time. As both forfeited shortcut errors are computed for a previous removal, we can store them in a reasonable data structure (i.e. matrix) to access them in constant time. The remaining shortcut error is computed anyways in the Greedy algorithm. Since we only add a constant factor for each insertion, the runtime bounds stays the same as the plain Greedy algorithm, which is given by $\mathcal{O}(n(f_X(n) + \log\ n)$ according to theorem 4.16. □

6 Output-Sensitive Simplification Extraction

Every application utilizing the simplification of curves is in need of extracting the simplification of some length effectively. This can be trivially achieved by storing the set of vertices for each simplification individually. But, this requires space that is in $\mathcal{O}(n^2)$, which may be too much space consumption when dealing with big instances. Therefore, our goal is to find an algorithm that is able to extract a simplification with length m in $\mathcal{O}(m)$, while using linear space.

To achieve this, first, we propose a new data structure. Given a removal sequence $r = \langle v_i, \ldots, v_j \rangle$ with $|r| = n - 2$, the data structure mainly consists of three arrays A, L and R, each of length n. At every position k within the arrays, we store information related to the k-th vertex of the curve (v_k).
In $A[i]$ we store the index of the appearance of v_i in the removal sequence r. Additionally, we keep track of the vertex that is removed lastly, which corresponds to the one with the highest value in A (we call this vertex index *last*). This enables us to access it quickly in constant time. Furthermore, in $L[i]$, we store the index $j < i$ of a vertex v_j that is the most recently removed vertex before the removal of v_i (left removal before). Therefore, it holds that $L[i] = j \Rightarrow \forall\ 1 < k < i : A[j] \geq A[k] \vee A[k] > A[i]$. Similarly, the array R operates in the same manner, except we search for the index $j > i$ of a vertex v_j with the most recent appearance before the removal of v_i in the removal sequence (right removal before). Hence, it holds $R[i] = j \Rightarrow \forall\ i < k < n : A[j] \geq A[k] \vee A[k] > A[i]$. It is important to note that the distinction between $L[i]$ and $R[j]$ lies in the fact that $L[i] < i$ and $R[i] > i$. This means that $L[i]$ stores the index of the before removed vertex to the left, while $R[i]$ stores the index of the before removed vertex to the right.
Any cell $L[i]$ or $R[j]$ that cannot be assigned according to the previous rules is given a value of -1.

[Theorem 6.1]

For a removal sequence r with length $n-2$, we can construct the proposed data structure in $\mathcal{O}(n)$, while using linear space.

Proof. As all three arrays use linear space and we additionally only store one variable of constant space, it is obvious that the data structure needs linear space. The A matrix can be initialized trivially, by assigning $A[1] = -1$ and $A[n] = -1$, then iterating through the removal sequence. For each v_i we can assign $A[i]$ using an iteration counter. The arrays L and R on the other hand are a little more complex. But, using an algorithm for 1D range maxima queries [11], which requires linear preprocessing time and achieves constant time queries, we are able to initialize them in linear time as well. □

Using this data structure, we can perform a simple algorithm to extract the $k-th$ simplification. Its basic idea goes as follows. First, we create a double linked list that only consists of v_1 and v_n. Then, we start by retrieving the index i with the highest value in A. Now, we look whether it holds $A[i] > k$, and if it does we add v_i between v_1 and v_n into the list. Thereafter, we do the same with v_l that has the index $l = L[i]$. If it holds $A[l] > k$ we add v_l at the left of v_i in the list. We also do the same with the vertex v_r with the index $r = R[i]$. Again, if it holds $A[r] > k$, then we add v_r at the right of v_i in the list. We repeat this procedure for every inserted vertex. The algorithm is called the 'extract' algorithm and can be seen in the following.

Algorithm 15: *extract*$(\mathcal{L}, A, L, R, last, k)$

Data: polygonal curve $\mathcal{L} = \langle v_1, \dots, v_n \rangle$, appearance array A, left removal before array L, right removal before L, last removal $last$, desired simplification index k
Result: simplification $\mathcal{S}_k$

```
1  S_k ← double linked list with v_1, v_n initially
2  if A[last] ≤ k then
3  |  return S_k
4  end
5  S_k ← v_1, v_last, v_n
6  queue ← new Queue with initially last enqueued
7  while queue not empty do
8  |  i ← queue.dequeue()
9  |  l ← L[i]
10 |  if A[l] > k then
11 |  |  add v_l to the left of v_i in S_k
12 |  |  queue.enqueue(l)
13 |  end
14 |  r ← R[i]
15 |  if A[r] > k then
16 |  |  add v_r to the right of v_r in S_k
17 |  |  queue.enqueue(r)
18 |  end
19 end
20 return S_k
```

[Theorem 6.2]

The extract algorithm needs $O(m)$ time with $m = |\mathcal{S}_k|$.

Proof. It is obvious to see, that the algorithm runtime depends on the number of iterations of the while loop. Each iteration of the while loop runs in constant time, and it runs as long, as we have elements in the queue. Furthermore, we dequeue exactly one element in each iteration. Therefore, we need to identify the number of elements, that we insert into the queue in total. Each inserted vertex i holds $A[i] > k$. As each $A[i] > 0$ is distinct from another, and the max value in the array is $n-2$, there are $n-2-k$ many inserted values in the queue. Since $|\mathcal{S}_k| = n-k$ holds, the runtime is given by $\mathcal{O}(n-2-k) = \mathcal{O}(n-k) = \mathcal{O}(|\mathcal{S}_k|) = \mathcal{O}(m)$. □

7 Experimental Study

To analyze practicability of proposed algorithms, in the following, we will evaluate them and compare their solutions under the presented objectives, using an experimental study.

7.1 Algorithm Overview

As we introduced many algorithms and variations, first, we recall all of them by giving an overview about them and their theoretical runtimes.

7.1.1 Algorithms for simple objectives (simple algorithms)

Here, we cover all algorithms of section 4, denoted as the 'simple algorithms'.

- MinMAX: Algorithm solving the MAX problem (algorithm 1)
- MinSUM: Algorithm solving the SUM problem (adapted version of algorithm 1)
- Greedy: Greedy algorithm with 4-approximation for SUM (algorithm 4)
- Practical Greedy 'PGreedy': Practical adaptation for the Fréchet distance using the distance oracle (algorithm 5)
- In Order Heuristic 'IOH': The In Order Heuristic introduced in section 4.4
- Random Order Heuristic 'ROH': The Random Order Heuristic introduced in section 4.4, using the In Order solution as input and the Fisher-Yates shuffle algorithm to get a random permutation
- Tree Order Heuristic 'TOH': The Tree Order Heuristic introduced in section 4.4 (algorithm 8)

	Hausdorff	Fréchet	Discrete Fréchet	Approximated Fréchet
MinMAX	$\mathcal{O}(n^3)$	$\mathcal{O}(n^4)$	$\mathcal{O}(n^3)$	$\mathcal{O}(kn^3)$
MinSUM	$\mathcal{O}(n^3)$	$\mathcal{O}(n^4)$	$\mathcal{O}(n^3)$	$\mathcal{O}(kn^3)$
Greedy	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	$\mathcal{O}(kn^2)$
Practical Greedy	$\mathcal{O}(c \cdot n^2 \log_b\ n)$			
In Order	$\mathcal{O}(n)$			
Random Order	$\mathcal{O}(n)$			
Tree Order	$\mathcal{O}(n)$			

Figure 9: This table shows the theoretical runtime of all simple algorithms under different distance metrics. As the practical Greedy algorithm uses the Fréchet distance oracle and the heuristics are non-dependent of any distance metric, they share one cell for all distance metrics.

7.1.2 Algorithms for extended objectives (extended algorithms)

Next, we cover all algorithms and variations, introduced in section 5, denoted as 'extended algorithms'.

- MinSMA: The algorithm solving the SumMaxActive problem, using the divide and conquer approach, solved by the GCE algorithm (see algorithm 14 and theorem 5.33)
- SMT-DNC: The divide and conquer approach for the SumMaxTotal problem, solved by the GCE algorithm (see algorithm 14 and theorem 5.33)
- {SSA/SST}-DNC: The divide and conquer approach (algorithm 9) for the SumSumActive / SumSumTotal problem, using the exact merge (algorithm 10)
- {SSA/SST}-{GTD/GBU}: The divide and conquer approach (algorithm 9) for the SumSumActive (SSA) / SumSumTotal (SST) problem, using the greedy top down merge (algorithm 12) or the greedy bottom up merge (algorithm 13)
- {SSA/SST}-{GTD/GBU}-Exact: The greedy candidate estimation algorithm (algorithm 14) for the SumSumActive (SSA) / SumSumTotal (SST) problem, using the greedy top down merge error (algorithm 12) or greedy bottom up merge error (algorithm 13) as estimation function, and the exact SMM (algorithm 10) as merge algorithm
- {SSA/SST}-MEST-{GTD/GBU}: The greedy candidate estimation algorithm (algorithm 14) for the SumSumActive (SSA) / SumSumTotal (SST) problem, using the mean estimation (definition 5.35) as estimation function, and the greedy top down (algorithm 12) or greedy bottom up (algorithm 13) as merge algorithm
- Greedy Difference 'GD': The greedy adaptation covered in section 5.4.

	Hausdorff	Fréchet	Discrete Fréchet	Approximated Fréchet
MinSMA	$\mathcal{O}(n^3)$	$\mathcal{O}(n^4)$	$\mathcal{O}(n^3)$	$\mathcal{O}(kn^3)$
SMT-DNC	$\mathcal{O}(n^3)$	$\mathcal{O}(n^4)$	$\mathcal{O}(n^3)$	$\mathcal{O}(kn^3)$
{SSA/SST}-DNC	$\mathcal{O}(n^5)$	$\mathcal{O}(n^5)$	$\mathcal{O}(n^5)$	$\mathcal{O}(kn^3 + n^5)$
{SSA/SST}-{GTD/GBU}	$\mathcal{O}(n^4)$	$\mathcal{O}(n^4)$	$\mathcal{O}(n^4)$	$\mathcal{O}(kn^3 + n^4)$
{SSA/SST}-{GTD/GBU}-Exact	$\mathcal{O}(n^4)$	$\mathcal{O}(n^4)$	$\mathcal{O}(n^4)$	$\mathcal{O}(kn^3 + n^4)$
{SSA/SST}-MEST-{GTD/GBU}	$\mathcal{O}(n^3)$	$\mathcal{O}(n^4)$	$\mathcal{O}(n^3)$	$\mathcal{O}(kn^3)$
Greedy Difference	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	$\mathcal{O}(kn^2)$

Figure 10: This table shows the theoretical runtime for all extended algorithms under different distance metrics.

7.2 Data and Hardware

We conducted all experiments on an AMD Ryzen Threadripper 3970X 32-Core processor (clocked max 3700 MHz) with 64 GB main memory. All test instances were randomly selected real world GPS-traces picked from https://www.openstreetmap.org/traces. Furthermore, sometimes it is necessary to generate polygonal curves with custom lengths. For instance, this arises when we are in need of exceptionally large instances. Therefore, we propose a simple algorithm to create artificial polygonal curves for individual lengths. It uses a random changing magnitude to create a sequence of random points and a scaling factor to scale the geometric size of the curve. It can be seen in the following algorithm.

Algorithm 16: $RandomCurve(n, scala)$

Data: length of the output curve n, scaling factor $scala$
Result: output curve $\mathcal{L}$

$(magX, magY) \leftarrow (0, 0)$
$(curX, curY) \leftarrow (0, 0)$
$\mathcal{L} \leftarrow$ empty sequence $\langle\rangle$
for $i = 1$ ***to*** n **do**
  $(r1, r2) \leftarrow$ tuple containing 2 random numbers in $\mathbb{R}$ between 0 and 1
  $(magX, magY) \leftarrow (magX + r1, magY + r2)$
  $v \leftarrow (scala \cdot magX, scala \cdot magY)$
  $\mathcal{L} \leftarrow \langle v \rangle$
end
return $\mathcal{L}$

As $scala$ is a parameter that scales the curve exclusively, in the following, we will stick with $scala = 10$.

7.3 Implementation Details

We implemented all algorithms in Java and used no external libraries. Therefore, we integrated all distance metrics ourselves as well. The polygonal curve is represented as an array of vertices, while each vertex consists of an x and y decimal value of type double. This is especially useful for the calculation of shortcut errors, as the utilization of an array allows us to access any given vertex in constant time. Additionally, each vertex includes pointer to predecessors and successors, as well as their heap index when used by the Greedy algorithms, so we can access any neighbour and its heap position in constant time. Also, we halved the space consumption of all symmetric matrices by projecting one halve of it onto a one dimensional array instead.
All proposed dynamic programs have a very similar core, and hence, we implemented an abstract DynamicProgramm class. To avoid redundancy, each simplification algorithm using a dynamic program derives from this class.
Furthermore, the min-heap of the Greedy algorithm is also custom implemented, as we made slight adaptions for each of the Greedy algorithms.

The source code can be accessed at https://github.com/nickrmb/Gradual-Line-Simplification.

7.4 Experiments and Results

In this section, we will present our experiments, show the results, and shortly discuss findings.

7.4.1 Interval search Fréchet distance parameter

First, we analyze the interval search approach for the Fréchet distance to find a reasonable k for algorithm 3. Therefore, we choose 10 random traces and pick 10 random shortcuts in each trace. For each selected shortcut, we evaluate the exact Fréchet distance using the $\mathcal{O}(n^2)$ version, as well as the approximation approach for all $k \in [1, \ldots, 16]$ with $k \in \mathbb{N}$. The results of our experiments can be found in figure 11 and figure 12.

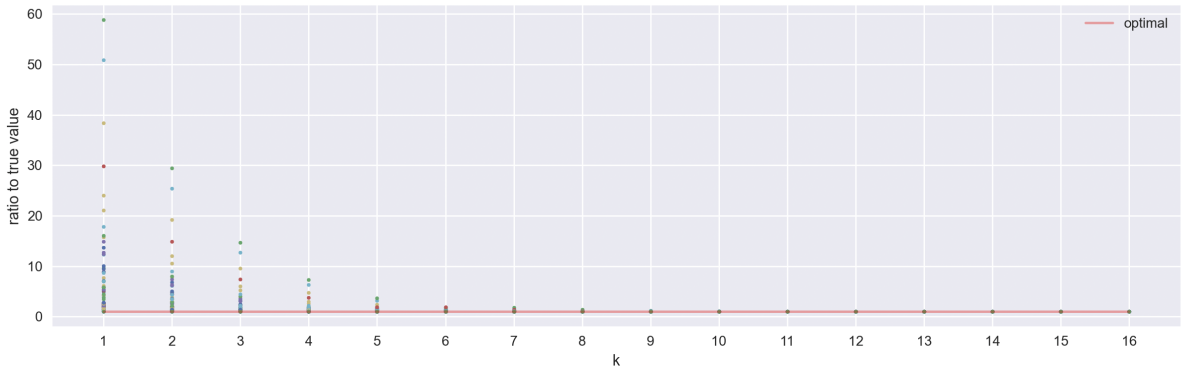


Figure 11: This graph shows the approximated Fréchet distance value of 100 random shortcuts for all k between 1 and 16. Furthermore, the light red line on the bottom marks the optimal value (1 to 1 ratio to true value).

k	1	2	3	4	5	6	7	8
avg.	6.283	3.408	2.059	1.464	1.217	1.107	1.070	1.039
k	9	10	11	12	13	14	15	16
avg	1.024	1.017	1.013	1.012	1.011	1.011	1.010	1.010

Figure 12: This table shows the average ratio to the true value of figure 11 for each k.

As we can see, a choice of $k \geq 10$ seems to work really well in practice, leading to an average deviation of less than 2%. Hence, in the following, if any algorithm uses the interval search approach for the Fréchet distance, we will choose $k = 10$.

7.4.2 Practical Greedy parameters

Next, we investigate the practical Greedy algorithm, and its choice of parameters. We choose a random trace, and apply the practical Greedy algorithm for two different series of parameter choices. In the first test series we settle with $c = 2$ and altered the choice of b for values from 1.05 up to 8. Similar, in

the second test series we decided on $b = 2$ and varied the choice of c for values between 1 and 8. Then for each test, we measured runtime and the induced error under the SUM objective function, using the Fréchet distance. Our test results can be seen in figure 13.

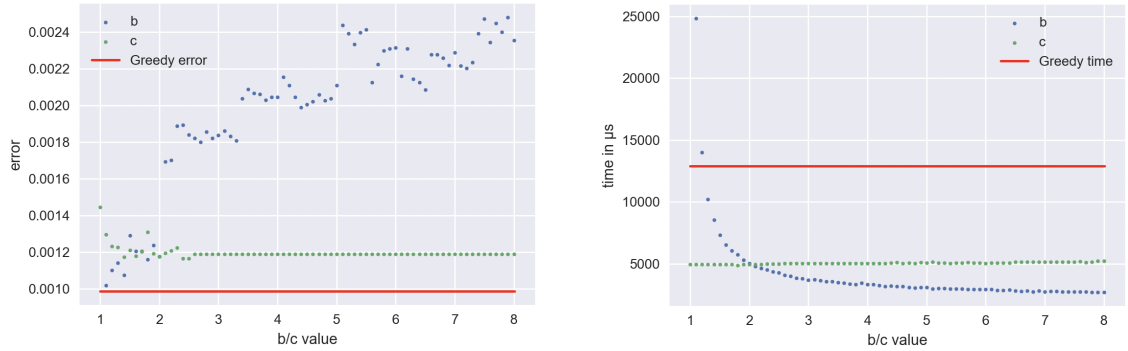


Figure 13: These graphs show a comparison between the runtime of the Greedy algorithm and the runtime of its practical adaptation for different parameter choices (left), as well as a comparison of their solution quality (right), on a single trace. The blue series represents the runtime/error for a fixed c value and a varying b value, while the green series has a fixed b value and a altering c value. The changing values are indicated by the x-axis, while the y-axis accords to the SUM error / runtime in microseconds (μs). Furthermore, the red line indicates the runtime/error of the Greedy algorithm under the Fréchet distance.

It is interesting to see that altering the b value has way more impact than changing the c value. This is probably because most shortcuts induce an error greater than $\frac{d_{\mathcal{L};X}(s_{1;n})}{n^2}$, and thus the choice of a greater c has small or even no impact on the tightening process.
On the other side, the parameter decision for the b value seems to scale with the error proportionally. Therefore, choosing a small b results in an error almost equivalent to the one provided by the Greedy algorithm. But at the same time, we then achieve an extraordinary runtime that exceeds the runtime of the Greedy algorithm. To be exact, it seems that a choice ob $b \geq 1.3$ seems to be sufficient, to achieve a runtime, faster than the Greedy algorithm. Hence, the choice of b must be done carefully, since it balances the quality of the solution and its runtime.

In the following experiments, we will use $b = 2$ and $c = 2$, as its runtime is 2.5 times faster on the given trace, while achieving an error that is 20% worse than the one produced by the Greedy algorithm, and therefore seems to be a fair balance between runtime and quality. We will denote this choice as the '2-PGreedy' algorithm. Furthermore, according to theorem 4.20, we know that the 2-PGreedy algorithm provides an $8 + \frac{1}{n^2} \approx 8$ approximation factor under the SUM function.

7.4.3 Runtime comparison of simple algorithms

After finding reasonable parameters for the practical Greedy algorithm, we are now able to compare practical performance of the simple algorithms in terms of runtime. To do this, we pick 50 random traces and apply the algorithms. We measure the runtime of all algorithms, using all 4 distance metrics. As the runtime of some algorithms scale sup-cubic (e.g. MinMAX under naive Fréchet), we set a time limit of 16 minutes for each computation. Furthermore, as the practical Greedy algorithm is designed to be an improvement for the naive Greedy algorithm under the Fréchet distance, we only compare it to other

algorithms utilizing the Fréchet distance. The results of this experiment can be seen in the following figure.

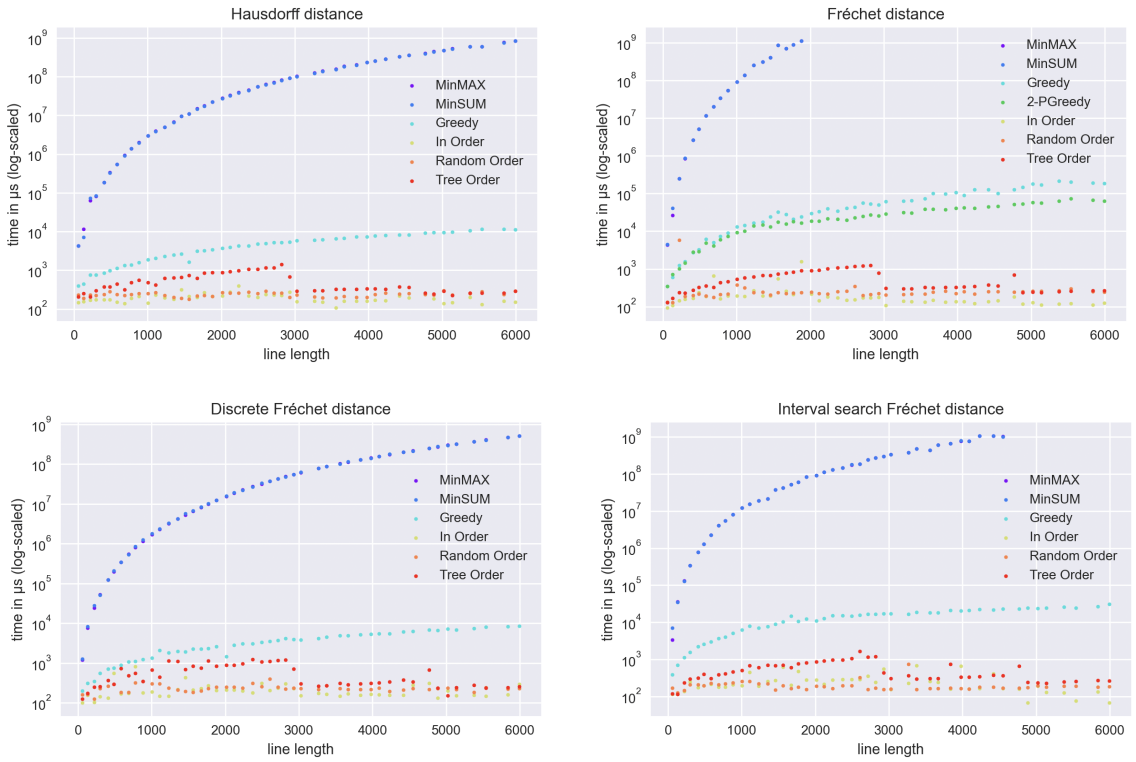


Figure 14: These graphs show a comparison of the runtime of all simple algorithms, each featuring a different geometric distance metric. The results of the MinMAX algorithm are mostly concealed by the ones of the MinSUM algorithm. On the x-axis, the line length is displayed, while the y-axis represents the runtime in microseconds (μs). It is worth noting that the y-axis is logarithmically scaled.

As expected, the MinMAX algorithm and the MinSUM algorithm yield a similar runtime, since their algorithms differ only slightly from each other. Both are also the slowest of all simple algorithms. Still they were able to solve all selected problems up to a length of 6000 under 16 minutes when utilizing the Hausdorff / discrete Fréchet distance. For the exact Fréchet distance, they were only able to handle problems with lengths close to 2000, but when utilizing the interval search approach for the Fréchet distance, we were then able to compute solutions with lengths up to roughly 4500 within the designated time frame. Furthermore, Greedy algorithm on the other side was way faster, but as presumed, not as fast as the 2-PGreedy algorithm. Finally, the runtime of the heuristics are the lowest. To be exact, the 'In Order' is the fastest, followed by the 'Random Order'. The 'Tree Order' was the slowest out of the heuristics. Therefore, we can make the following ordering in terms of runtime:
MinMAX/MinSUM > Greedy > 2-PGreedy > Tree Order > Random Order > In Order.

7.4.4 Quality Comparison of simple algorithms under MAX

Next, we are interested in the quality of the solutions under the MAX objective, produced by our algorithms. Therefore, we take the same 50 traces as before, and measure the error under the MAX objective for all algorithms, and compare their solution to the optimum, provided by the MinMAX algorithm. We compute the MinSUM and Greedy algorithm under each geometric distance metric individually. The results can be seen in the following table.

	MinSUM			Greedy			2-PGreedy		
	best	avg.	worst	best	avg.	worst	best	avg.	worst
Hausdorff	0.0%	1.21%	12.74%	0.0%	1.34%	15.93%	-	-	-
Fréchet	0.0%	0.27%	5.2%	0.0%	0.14%	2.58%	0.0%	1.21%	11.83%
IS Fréchet	0.0%	1.02%	10.95%	0.0%	1.1%	14.24%	-	-	-
Discrete Fréchet	0.0%	0.12%	4.96%	0.0%	0.31%	5.72%	-	-	-

	In Order			Random Order			Tree Order		
	best	avg.	worst	best	avg.	worst	best	avg.	worst
Hausdorff	0.0%	6.39%	46.34%	0.0%	3.14%	45.77%	0.0%	2.37%	32.86%
Fréchet	0.0%	7.42%	46.34%	0.0%	4.88%	42.05%	0.0%	2.18%	32.86%
IS Fréchet	0.0%	6.98%	39.19%	0.0%	3.01%	35.54%	0.0%	2.29%	24.74%
Discrete Fréchet	0.0%	1.93%	18.54%	0.0%	1.14%	19.46%	0.0%	0.02%	0.84%

Figure 15: These table shows a quality comparison between simple algorithms under the MAX objective for each geometric distance metric. Each value indicates the deviation ratio to the optimal value (how much worse to optimum) using the MAX error objective function under the geometric distance metric indicated on the left.

It is very interesting that all algorithms were able to compute at least one solution that resulted in a similar error to the one produced by the exact algorithm. This could be caused by some traces being very small, and therefore, some solutions could be deterministic for all algorithms. The MinSUM and Greedy algorithm both worked pretty well under all distance metrics. Also the 2-PGreedy algorithm seemed to work fine, since it computed solutions that were 1.21% worse than the optimal solution on average. Surprisingly, under the discrete Fréchet distance, the Tree Order was able to achieve the best solutions on average, being even better than the MinSUM and the Greedy algorithm. The worst solution in total, was computed by the In Order heuristic, given by 46.34% worse than the optimal solution. Therefore no algorithm exceeded the 2-approximation boundary, and as expected, corollary 4.15 seems to hold in our practical results. The Random Order heuristic worked much better on average, but was able to achieve similar worst case solutions, as the In Order heuristic. The quality retrieved by the Tree Order is definitely the best of all heuristics, but at the same time, as discussed before, it is also the slowest.

Since the MinSUM algorithm produces a similar runtime as the MinMAX algorithm, there is no particular reason to choose it when minimizing the MAX objective error. Hence, we conclude, that the Greedy algorithm or its practical adaptation should be the superior choice under the MAX objective, if we want to compute a good solution in less time. Alternatively, if the runtime remains excessively high, we suggest considering the Random Order or Tree Order heuristic as viable alternatives.

7.4.5 Quality Comparison of simple algorithms under SUM

After discussing our results about the MAX objective, furthermore, we want to analyze the practical quality performance of simple algorithms under the SUM objective. As we already computed solutions of the previous traces, we use the same 50 traces of section 7.4.3 / section 7.4.4 and compare their quality under the SUM objective. We measure the quality of the 2-PGreedy algorithm only under the Fréchet distance. Let us recall that we have established a time limit of 16 minutes. The results can be seen in figure 16 and a summary can of them can be seen in figure 17.

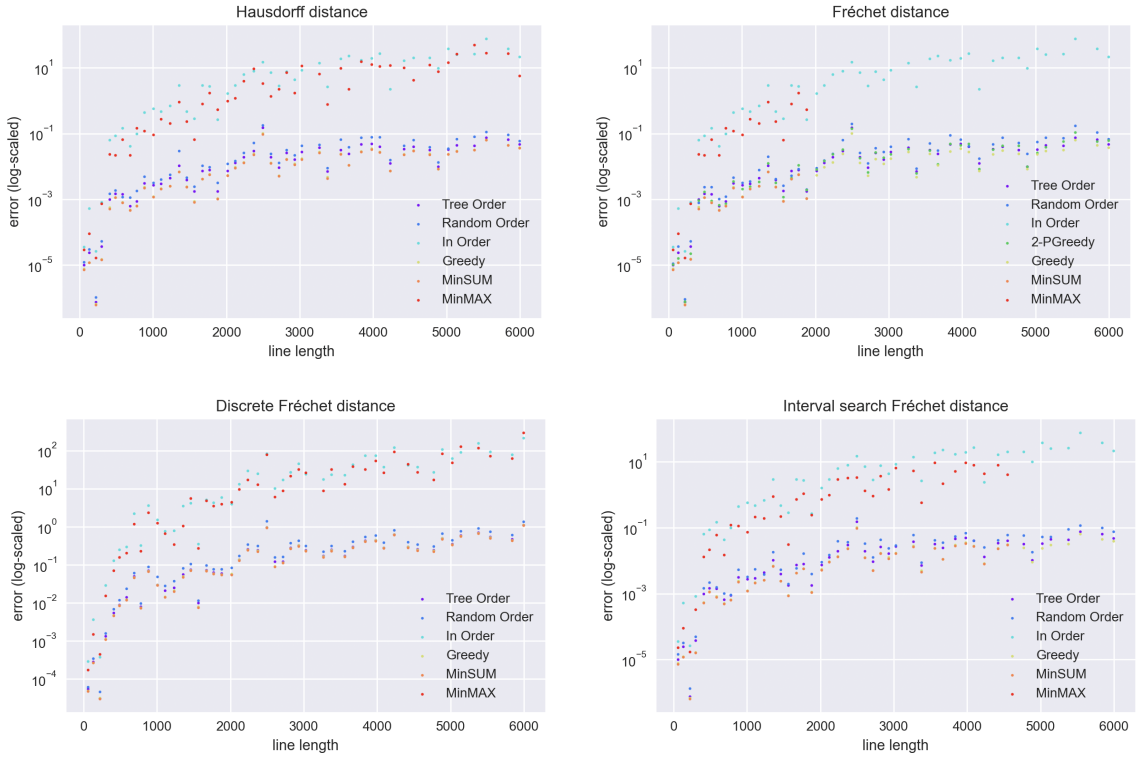


Figure 16: These graphs show a comparison of the quality under the SUM objective of all simple algorithms, while each graph features a different geometric distance metric. On the x-axis, the line length is displayed, while the y-axis represents the SUM error. It is worth noting that the y-axis is logarithmically scaled.

Again, the Greedy algorithm seems to work really well, being approximately 4% worse than the optimal value on average. Furthermore, its worst solution is only 11% worse than the optimal one, and hence, the 4-approximation bounds seem to hold in practice. With 90%, the worst solution of the practical adaptation exceeded the worst solution of the Greedy algorithm. Still, the 8-approximation bounds of the 2-PGreedy algorithm seems to hold as well in practice. Overall, the practical adaptation was slightly worse, as it computed solutions that are approximately 35% worse than the ones, produced by the Greedy algorithm.

	MinMAX			Greedy			2-PGreedy		
	best	avg.	worst	best	avg.	worst	best	avg.	worst
Hausdorff	3.05	329	1641	0.005	0.04	0.11	-	-	-
Fréchet	3.05	113	506	0.008	0.04	0.11	0.009	0.5	0.9
IS Fréchet	2.1	152	553	0.009	0.04	0.1	-	-	-
Discrete Fréchet	2.7	83	472	0.002	0.04	0.08	-	-	-

	In Order			Random Order			Tree Order		
	best	avg.	worst	best	avg.	worst	best	avg.	worst
Hausdorff	3.9	525	2173	0.5	1.28	3.31	0.2	0.57	1.5
Fréchet	3.93	230	708	0.26	1.23	2.54	0.2	0.7	1.46
IS Fréchet	3.94	419	2131	0.42	1.25	2.61	0.2	0.6	1.37
Discrete Fréchet	5.2	205	636	0.27	0.56	2.1	0.2	0.5	0.94

Figure 17: This table is a summary of the graphs of figure 16

.

On the other side, the MinMAX algorithm performs poorly under the SUM objective. Only the In Order heuristic managed to retrieve even worse solutions. With solutions above 2000 times worse than the optimal one, and an average of at least 200 times worse, it worked really bad in practice, and does not seem to be a reasonable algorithm choice under the SUM objective.
Moreover, the Tree Order heuristic worked the best out of the heuristics. Its solutions exceeded modestly the ones, produced by the Greedy algorithm. The Random Order heuristic performed worse than the Tree Order, but it can still be considered a good alternative.

Therefore, to achieve a reasonable quality under the SUM objective in less than cubic time, we suggest the usage of the the Greedy algorithm, or its practical adaptation. Furthermore, if the runtime remains too high, we recommend considering the Random or Tree Order heuristics, as the In Order heuristic is able to produce very bad solutions.

7.4.6 Impact of Geometric Distance Metrics on simple algorithms

In our next experiment, we are interested to find out the impact of our geometric distance metrics on the resulting runtime. We do this by utilizing the 50 random selected traces, and applying the Greedy algorithm and the MinSUM algorithm under each geometric distance metric individually, as both runtimes scale with the runtime of the chosen distance metric. Afterwards, for each algorithm, we are then able to compare the runtimes under the distance metrics. Since the MinMAX and MinSUM algorithms yield identical runtimes, we did not conduct any additional testing on the MinMAX algorithm.
Our results can be seen in figure 18.

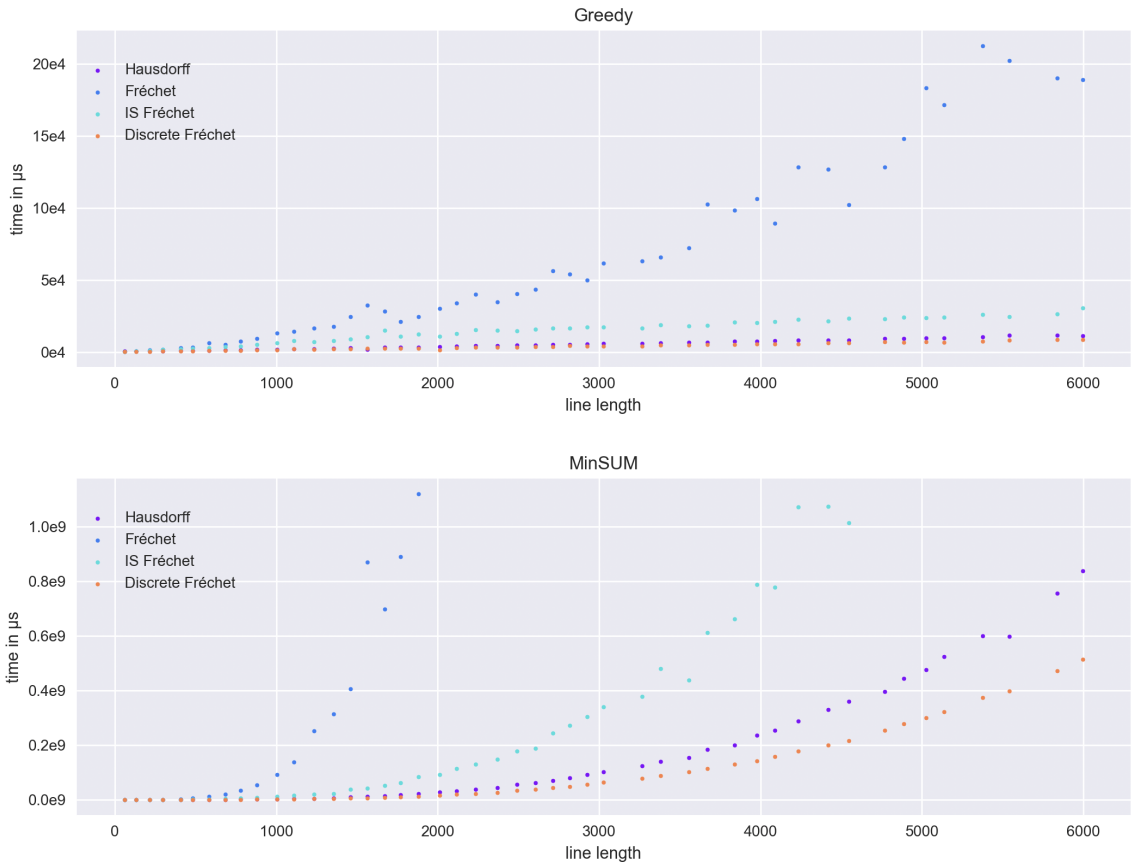


Figure 18: These graphs show a comparison between the runtime of simple algorithms under different geometric distance metrics. The algorithm is given above each graph. Furthermore, the x-axis indicates the line length, and the y-axis features the runtime in microseconds (μs).

The graphs clearly show the difference between the quadratic runtime scaling of the exact Fréchet distance and the linear runtime scaling of the other metrics. In our implementation, it seems like the discrete Fréchet distance is the fastest, followed by the Hausdorff distance. The slowest linear distance metric is the interval search Fréchet distance with parameter choice $k = 10$. Still it was able to perform vastly faster than the exact Fréchet distance.

7.4.7 Limit tests of Simple Algorithms

In this experiment we want to find out the limits of the simple algorithms, by testing their runtime with increasingly extraordinary huge instances, until they fail to meet to compute a solution within a time limit. Therefore, we use algorithm 16 to generate artificial curves of individual lengths. Furthermore, we choose a time frame of 1000 seconds (approximately 17 minutes) and start with a curve length of $n = 0$. For the Greedy algorithm under the Fréchet distance, as well as the practical algorithm, we increase the curve length by 20000 after each successful attempt. Under all other distance metrics, we increase the curve length of the Greedy algorithm, as well as the heuristics, by 1 million vertices. As it is expected that the heuristics execute very fast, instances that fail to find a solution in the time limit, would be way

too big. Hence, we stop as soon as the last Greedy algorithm fails, to verify this assumption. The exact algorithm scales vastly faster, and thus, we choose a step size of 200 for the exact algorithm (MinMAX / MinSUM) under all distance metrics. The results towards the Greedy algorithm can be seen in figure 19, while the results to the exact algorithms can be seen in figure 20.

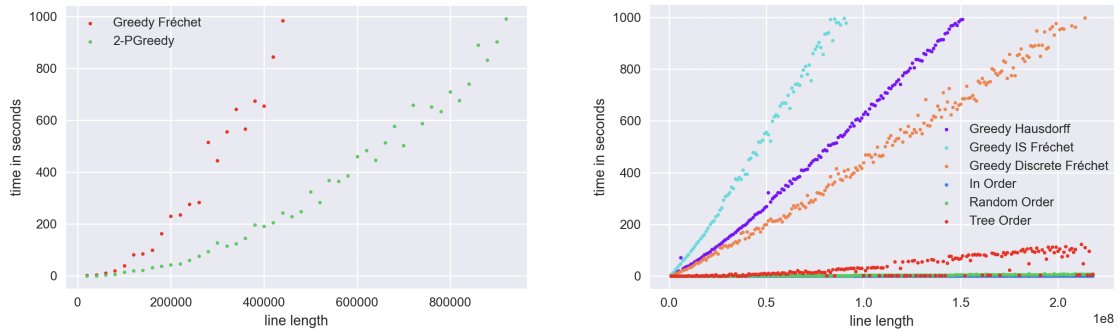


Figure 19: These graphs show the results of limit tests for the Greedy algorithm and its practical adaptation. The x-axis indicates the line length, while the y-axis represents the runtime in seconds.

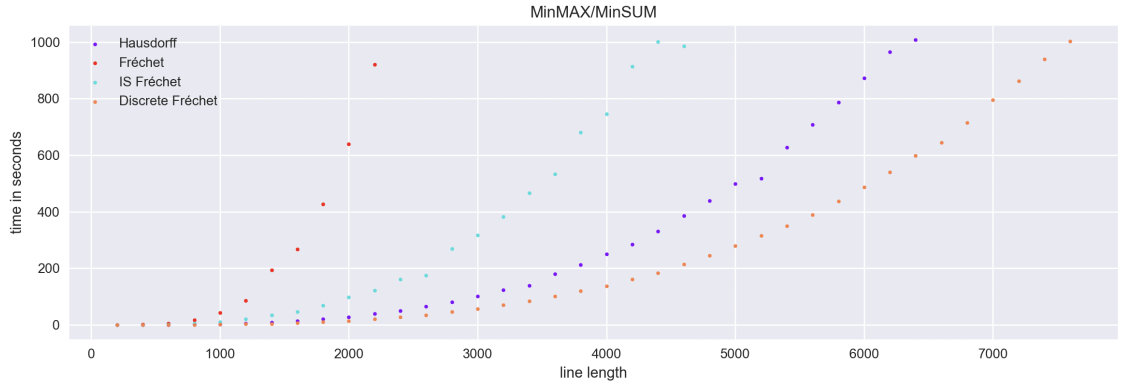


Figure 20: This graph shows the results of limit tests for the MinMAX / MinSUM algorithm. The x-axis indicates the line length, while the y-axis represents the runtime in seconds.

As we already expected, the exact algorithm under the Fréchet distance is the slowest, since its runtime is in $\mathcal{O}(n^4)$. Therefore, it was only able to solve instances up to 2200 in the designated time frame. Using the interval search method for the Fréchet distance, we are able to improve these bounds up to instances of length 4600. Under the other two distance metrics, we were able to compute instances up to 6400 for the Hausdorff distance and even 7600 using the discrete Fréchet distance.

The Greedy algorithms are way faster, and hence, able to solve much bigger instances. They also almost seem to scale linear in practice. Even tho the Fréchet distance is the slowest, we are still able to compute solutions for instances up to 420,000. And this can be further improved, up to instances of length 920,000, using the 2-PGreedy algorithm. Greedy variations using linear distance metrics are able to solve problems 100 times bigger. Moreover, under the discrete Fréchet distance, we are able to compute solutions over 200,000,000 in the 1000 seconds time frame. The heuristics are still pretty fast, and even for these instances, able to solve them in 100 seconds or less.

We conclude, that the exact algorithm seems to work on rather small instances, while the Greedy algorithm is able to solve most huge problems.

7.4.8 Variation runtime impact of extended algorithms

After testing our simple algorithms in practice, we now move on to the extended algorithms. Hence, we first of all want to compare their runtime. To do this, we select 50 random traces, and applied the extended algorithms on them, while measuring their runtime. It is important to note that we have set a time limit of 20 minutes. Also, as we compare many algorithms, to lower the complexity of our analysis, we will compare the algorithms under the discrete Fréchet distance exclusively. As we have many variations ({SSA/SST}, {GTD/GBU}), we start by analyzing the difference between them. More precisely, we want compare the impact between the choice of the SSA and the SST, as their only difference occurs in the appendage of the resulting shortcut set sequence. Furthermore, we are interested in the impact of the choice of GTD / GBU on the runtime of the algorithms. We expect a similar runtime, as all algorithm variations have the same runtime bounds. The results can be seen in figure 21.

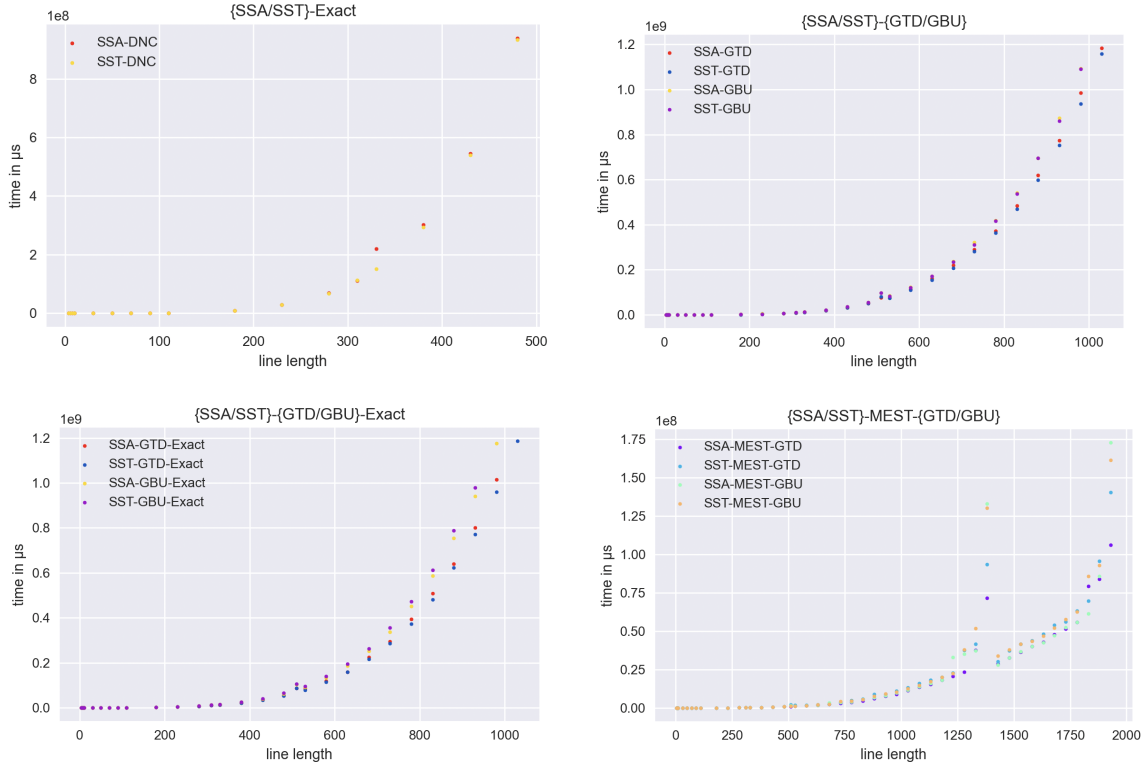


Figure 21: These graphs show a comparison between the variations of each algorithm. The algorithm is given above each graph. Furthermore, the x-axis indicates the line length, and the y-axis features the runtime in microseconds (μs).

In our results, we can see that the variations scale similar and they seem to differ by some smaller constant factor. Still, it seems like the variations using the GTD are slightly faster than algorithms using the GBU. Finally, the runtime of algorithms dedicated to the SSA and algorithms dedicated to the SST seem to be very similar. Hence, we are able to conclude that all variations seem to run in a very comparable time.

7.4.9 Runtime comparison of extended algorithms

Next, we are interested in comparing the runtime of all extended algorithms. To do this, we take the traces of section 7.4.8, and again measure the runtime of our algorithms, except this time we plot them all together. In the previous section we found that the variations for SSA and SST seem to run in a similar time. Therefore, to get a better overview, we only plot the ones for SSA, as we are able to reflect the results for the SST variations. We conduct the experiment using the Hausdorff distance as it is able to represent the other two linear time distance metrics as well. However, we extend our investigation to include the exact Fréchet distance additionally, aiming to identify potential differences between in the runtime of the algorithms, since it needs quadratic time. The results can be seen in figure 22.

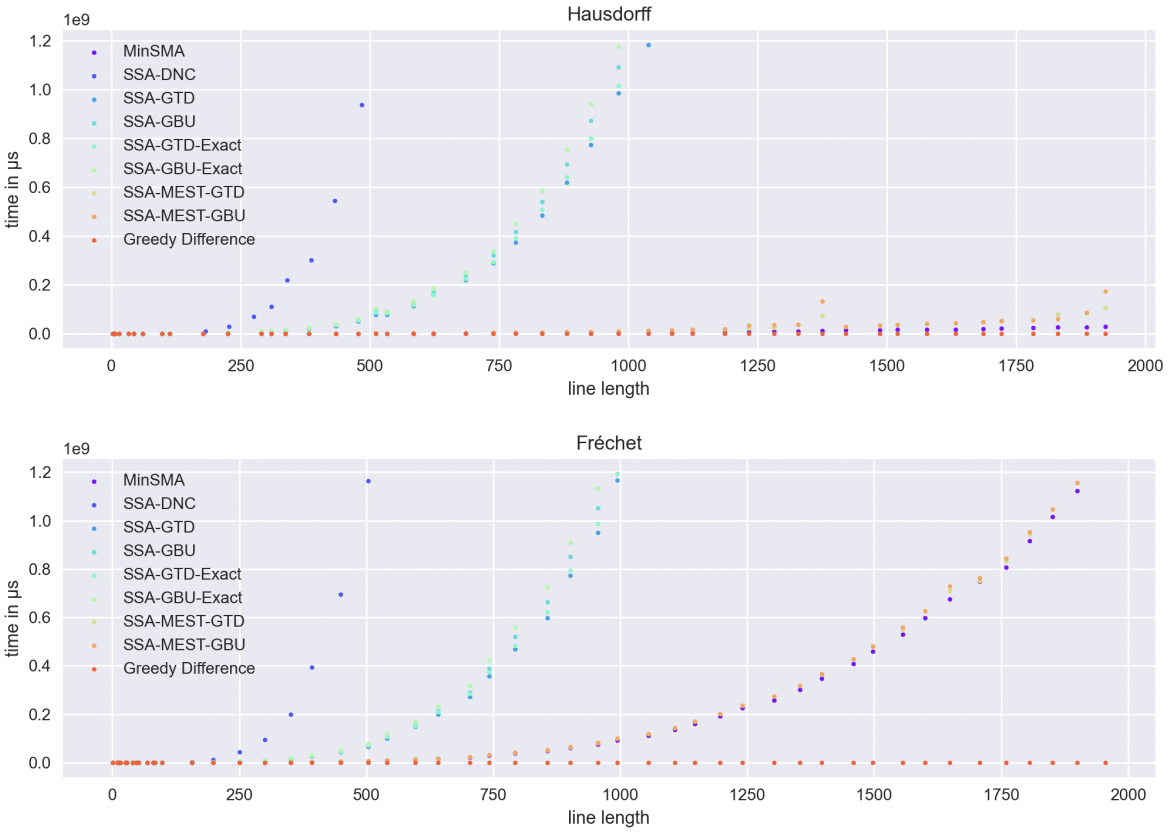


Figure 22: This graph shows a runtime comparison between our extended algorithms. The x-axis indicates the curve length, while the y-axis represents the runtime in microseconds (μs).

In case of the Hausdorff distance, we can clearly see the difference of algorithms with runtime $\mathcal{O}(n^3)$, $\mathcal{O}(n^4)$ and $\mathcal{O}(n^5)$. As expected, the SSA-Exact is the slowest, and it was only able to solve problems up to 500 in the given time frame. The algorithms SSA-{GTD/GBU} and SSA-{GTD/GBU}-Exact scale very similar since both algorithms are in $\mathcal{O}(n^4)$. It is interesting that both of them are able to solve problems up to approximately 1000 in the time limit, even tho the SSA-{GTD/GBU}-Exact performs one exact merge additionally. These results also show under the Fréchet distance. This is probably because using the Fréchet distance does not have impact on algorithms that are in $\mathcal{O}(n^4)$ or higher. Also, it is interesting that the MinSMA was able to solve instances up to 1900 under the Fréchet distance, even tho it has the same theoretical runtime as the algorithms that solved instances up to 1000. All

other algorithms run in cubic or even subcubic time, and therefore, yield a comparatively small runtime. Overall, the greedy difference algorithm achieved the smallest runtime.

7.4.10 Optimal solution computation analysis of extended algorithms

In our previous work we were able to show that we can compute the exact solution for both simple objectives. On the other hand, for the extended objectives, we were only able to show that we can solve the SMA problem. Hence, it is unknown whether we are able to solve the other 3 problems: SMT,SSA and the SST. To analyze the optimality of our extended algorithms, we conduct another experiment. First, we implemented a simple brute force algorithm. It tests all possible solutions and retrieves the one that produced the lowest error under a given objective. As it has an exponential runtime and thus performs very slow, we were only able to compute solutions for curves with lengths up to 13 in a reasonable time. Therefore, we created 1000 random traces with lengths of 13 using algorithm 16. Then we performed the divide and conquer algorithms on them and compared their solutions with the one provided by brute force algorithm, by measuring the average deviation, as well as counting the number of solutions that are optimal. Our results can be seen in the following table.

	SMA		SMT		SSA		SST	
	#opt	avg. worse	#opt	avg. worse	#opt	avg. worse	#opt	avg. worse
MinSMA	1000	0.0%	736	0.42%	51	9.62%	796	0.27%
SMT-DNC	730	0.38%	999	0.0002%	44	11.2%	722	0.75%
SSA-DNC	50	9.03%	43	11.5%	989	0.008%	50	12.69%
SST-DNC	782	0.2%	728	0.4%	52	10.37%	983	0.005%

Figure 23: This table shows the performance of our extended algorithms on 1000 curves of length 13 under different objectives. The "#opt" row indicates the number of optimal solutions computed, while the "avg. worse" row represents the average deviation to the optimal value.

The MinSMA was the only algorithm that was able to solve all of the problems optimally under the SumMaxActive objective. No other algorithm was able to solve any problem optimally for all traces. Then, we can conclude that under the SMT,SSA and the SST objective, the solutions of sub-problems are not necessarily part of bigger problems. Hence, we are not able to propose an exact algorithm using the previous approaches.
Still, the divide and conquer algorithms worked very well under their respective objective. For example, the SMT-DNC only missed to compute the optimum of 1 curve under the SumMaxTotal objective. The other two algorithms performed comparable under their dedicated objectives, with 11 missed optimums for the SSA-DNC, and 17 missed optimums under the SST-DNC.
Another interesting observation is that the algorithms MinSMA, SMT-DNC and SST-DNC were able to compute over 70% optimal solutions under all objectives, except under the SSA objective, as under the SSA they achieved less than 60 optimal solutions per algorithm. On the other hand, the SSA-DNC algorithm was able to work well under the SSA objective exclusively. Under all other objectives it worked poorly, since it computed 50 or less optimal solutions.

In total, we can summarize that the MinSMA is in fact the only divide and conquer algorithm solving an extended objective, and we cannot provide a guarantee of optimal solutions when employing the proposed algorithms for the SMT, SSA, and SST objectives. Nevertheless, proposed divide and conquer algorithms demonstrate a tendency to compute an optimal solution, or at the very least, one that is in close to an optimal solution, in the vast majority of cases.

7.4.11 Quality comparisons under extended objectives

After analyzing the optimality of our algorithms, now, our goal is to evaluate the effectiveness of the adaptations using greedy merges and the greedy candidate estimation algorithm. Therefore, we pick 50 random traces and applied the extended algorithms on them. The SSA-DNC and SST-DNC algorithm failed to compute solutions for curves with lengths over 500 in a reasonable time, and thus, we settled with random curves with lengths less than 500. Furthermore, we also employ the MinSUM algorithm, as well as the Random Order, Tree Order heuristic and the (Practical) Greedy algorithm, as they may be competitive, and hence, it could be interesting to compare them. Then, we can also verify that the proposed extended algorithms are indeed better for extended objectives, compared to simple algorithms. Finally, it is important to note that we apply the algorithms under the Hausdorff distance and the Fréchet distance exclusively. This is because the other two distance metrics are adaptions of the Fréchet distance, and consequently, as we expect a similar outcome, we are able to reduce the number of experiments.

SumMaxActive

We will start by testing the greedy difference algorithm, as well as proposed simple algorithms, under the SumMaxActive objective. As reference, we will also apply the MinSMA algorithm, as it is able to minimize the problem. Our results can be seen in figure 24 and figure 25.

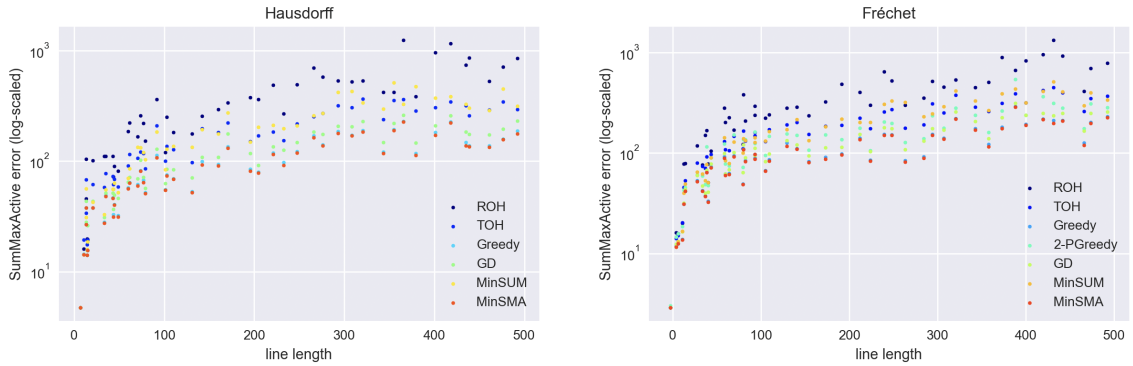


Figure 24: These graphs show the comparison between the quality achieved by various algorithms under the SumMaxActive objective. The x-axis represents the line length of the input curve, while the y-axis (logarithmic scaled) corresponds to the induced error.

	ROH	TOH	Greedy	2-PGreedy	GD	MinSUM
Hausdorff	2.316	0.785	0.027	-	0.213	0.861
Fréchet	2.119	0.746	0.023	0.420	0.169	0.757

Figure 25: This table shows the average deviation of each algorithm from the optimum provided by the MinSMA algorithm.

The findings are actually very interesting. It seems like the Greedy algorithm is the best alternative to the MinSMA algorithm, as its solutions performed less than 3% worse on average than the optimal one. This is followed by Greedy Difference algorithm with around 20%. Furthermore, the practical Greedy algorithm created solutions worse than 42% on average, and hence, may be a viable alternative.
Apparently, the MinSUM algorithm does not seem to be a well working algorithm under the SMA, as its solutions are worse than the one provided by the Tree Order heuristics. The Random Order heuristic

showed the poorest performance, with average solutions being more than twice as worse as those provided by the MinSMA algorithm.

SumMaxTotal

Next, we compared the algorithms under the SumMaxTotal objective. Hence, we applied the algorithm designed to minimize it, given by the SMT-DNC algorithm. Furthermore, we compare its solutions, with the ones provided by the simple algorithms. Results of this experiment can be seen in figure 26 and figure 27.

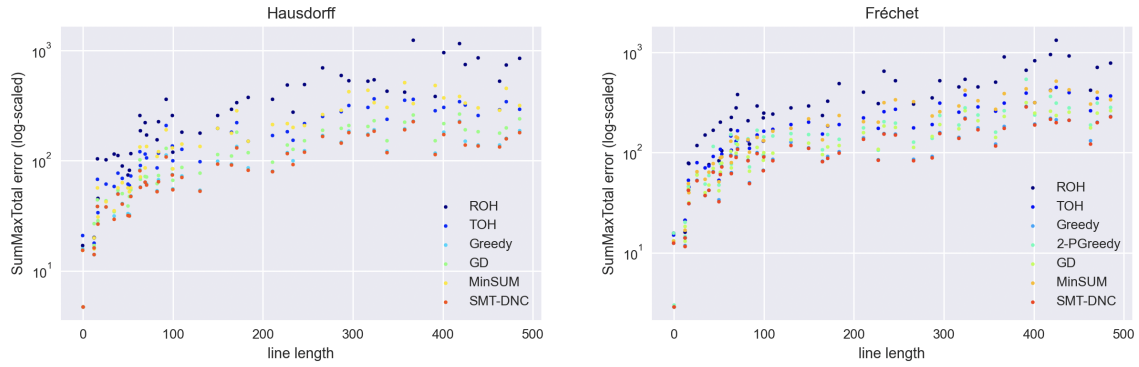


Figure 26: These graphs show the comparison between the quality achieved by various algorithms under the SumMaxTotal objective. The x-axis represents the line length of the input curve, while the y-axis (logarithmic scaled) corresponds to the induced error.

	ROH	TOH	Greedy	2-PGreedy	GD	MinSUM
Hausdorff	2.211	0.721	0.013	-	0.208	0.821
Fréchet	2.037	0.695	0.013	0.391	0.163	0.724

Figure 27: This table shows the average deviation of each algorithm from the solution provided by the SMT-DNC algorithm.

Our results clearly show that even tho the SMT-DNT may not yield an optimal solution for each curve, it remains the most effective algorithm under the SumMaxTotal objective, as it exhibits the lowest solution error across all traces.
All other algorithms, seem to resemble the results of the ones that we found for the SumMaxActive objective. To be more precise, the Greedy algorithm seems again to be a strong alternative, computing solutions that are 1.3% worse on average under both metrics. This is followed by the greedy difference algorithm with around 18% average deviation to the SMT-DNC algorithm. Furthermore, the practical adaptation of the Greedy algorithm could also be considered a good alternative, as its solutions performed approximately 40% worse on average. The MinSUM algorithm performed similar to the Tree Order heuristic, and hence, worse than the 2-PGreedy algorithm. Finally, the Random Order heuristic yielded the poorest performance, with solutions averaging a 200% deviation from those provided by the SMT-DNC algorithm.

SumSumActive

In the comparison of the algorithms under the SumSumActive objective, we are now interested in all algorithms dedicated to solve the objective, including all variations. Therefore, we do not only include

the simple algorithms and the SSA-DNC algorithm, we will additionally apply and compare the algorithms SSA-GBU, SSA-GTD, SSA-GBU-Exact, SSA-GTD-Exact, SSA-MEST-GBU and SSA-MEST-GTD. See figure 28 and figure 29 for results.

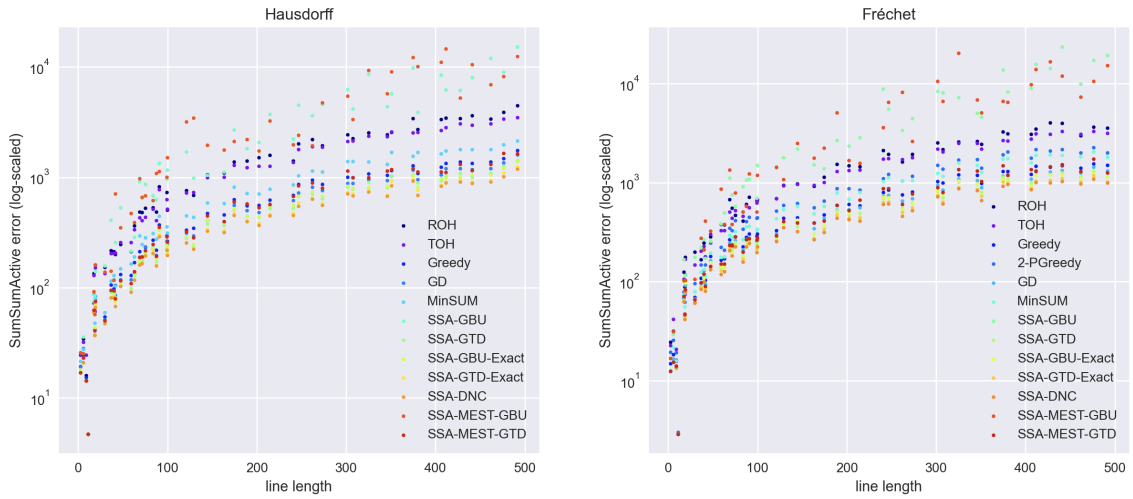


Figure 28: These graphs show the comparison between the quality achieved by various algorithms under the SumSumActive objective. The x-axis represents the line length of the input curve, while the y-axis (logarithmic scaled) corresponds to the induced error.

	ROH	TOH	Greedy	2-PGreedy	GD	MinSUM
Hausdorff	2.046	1.787	0.375	-	0.233	0.705
Fréchet	1.965	1.735	0.354	0.904	5.383	0.29

	GBU	GTD	GBU-Exact	GTD-Exact	MEST-GBU	MEST-GTD
Hausdorff	4.505	0.129	0.13	0.032	8.11	0.14
Fréchet	5.931	0.128	0.116	0.036	5.65	0.281

Figure 29: This table shows the average deviation of each algorithm from the solution provided by the SSA-DNC algorithm.

This experiment has revealed several interesting findings. Firstly, the SSA-DNC algorithm reliably computes the best solutions for each of our problem instances based on the SSA objective, while the GD algorithm emerges as the best simple algorithm. Following closely is the plain Greedy algorithm. As anticipated, the 2-PGreedy algorithm produces worse solutions, as its deviation is more significant. The MinSUM algorithm worked slightly better, but still seems to retrieve unfaithful solutions compared to other algorithms. One very interesting observation is that the TOH seems to work a lot worse now, while the ROH seems to perform similar as before. Consequently, the quality of both heuristics are now almost similar under the SSA. Regarding the variations, the GBU variation appears to be a suboptimal choice, as the GTD variation consistently generates better solutions. This particularly shows in the SSA-MEST-GBU variation, which yields the poorest solutions in total. Additionally, the best variation seems to be the SSA-GTD-Exact, as its solutions performed 3.4% worse on average, and therefore, making it the most promising alternative to the SSA-DNC algorithm. The SSA-GTD variation also performs well, with a deviation of approximately 13%. Interestingly, the SSA-MEST-GTD yielded solutions that performed only slightly worse under the Hausdorff distance.

SumSumTotal

We now test the same algorithms as before, except we choose those, that are meant to be used under the SST. Our results can be seen in figure 30 and figure 31.

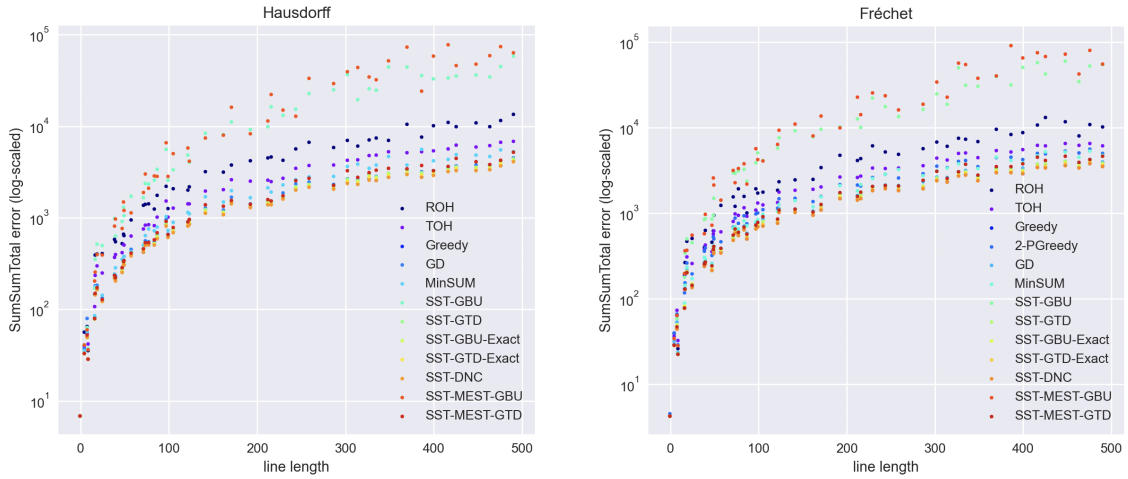


Figure 30: These graphs show the comparison between the quality achieved by various algorithms under the SumSumTotal objective. The x-axis represents the line length of the input curve, while the y-axis (logarithmic scaled) corresponds to the induced error.

	ROH	TOH	Greedy	2-PGreedy	GD	MinSUM
Hausdorff	1.693	0.725	0.017	-	0.132	0.339
Fréchet	1.630	0.742	0.018	0.45	0.115	0.308

	GBU	GTD	GBU-Exact	GTD-Exact	MEST-GBU	MEST-GTD
Hausdorff	6.224	0.07	0.044	0.02	8.11	0.14
Fréchet	6.785	0.073	0.05	0.021	8.949	0.139

Figure 31: This table shows the average deviation of each algorithm from the solution provided by the SST-DNC algorithm.

Again, the divide and conquer approach (SST-DNC) produced the best solutions for each trace. But apart from this, the experiment reveals some distinct findings to the ones of the SumSumActive experiment. A very impactful distinction is that the Greedy algorithm seems to be the best alternative to the SST-DNC under the SST objective overall. Also, the GD algorithm seems to be the worse greedy algorithm. The solutions computed by the 2-PGreedy seem to work fine, while the MinSUM algorithm only works slightly better. Furthermore, this time, the tree order heuristic computed solutions way better than the Random Order heuristic, and hence, seems to be a better heuristic. Once more, the worst performance was achieved by the SST-MEST-GBU, as it achieved an average deviation of over 800%. Despite the GTD once again proving to be the superior choice over the GBU, the performance of the SST-GTD was worse than that of the Greedy algorithm. Furthermore, it is quite surprising that even the SST-GTD-Exact performs worse than the Greedy algorithm. As the SST-MEST-GTD yielded solutions even worse, we conclude that the variations do not seem to be effective. Rather, we recommend the plain Greedy algorithm.

8 Discussion

During the initial experiments, we analyzed the practicality of the simple algorithms. We observed that while the MinMAX and MinSUM algorithms can yield optimal solutions, their runtime scales cubically, making them unsuitable for larger instances. In contrast, the greedy algorithm demonstrated impressive efficiency when handling significant larger instances, as it is able to compute solutions for instances with lengths over 200,000,000 in 1000 seconds. Also, the 2-approximation bounds under MAX and 4-approximation bounds under SUM held in practice. Furthermore, we discovered that the practical greedy algorithm allows easy trade-offs between the solution quality and the runtime, showing the advantages of the PGreedy algorithm, while providing an approximation under SUM as well. On the other hand, the In Order heuristic proved to be unreliable, producing poor solutions. Hence, the other two heuristics emerged as superior choices, as they handled very large instances with ease while achieving reasonable quality.
Regarding the extended algorithms, we found that the proposed divide and conquer algorithms were not only effective for the SMA objective, but also worked excellent for the other three objectives. They consistently computed optimal solutions, when applied to smaller traces. The variations of the SumSum algorithms performed exceptionally well under the SSA objective, granting us the flexibility to trade between runtime and quality. Particularly, the SSA-MEST-GTD achieved a good quality, while in use of cubic time. In comparison, the GBU algorithm was not a viable alternative to the GTD as its solutions performed significantly worse despite having a similar runtime. Nevertheless, for the SST problem, the greedy algorithm consistently produced superior solutions across all variations with significantly lower runtime. It was evident from our results that the greedy difference algorithm outperformed the plain greedy algorithm exclusively under the SSA objective. However, on all other objectives, it performed less effectively. Furthermore, the MinSUM algorithm proved to be unsuitable for the extended objectives, as its solutions performed worse than those generated by the plain greedy algorithm.

9 Conclusion

In this study, we presented a total of 6 reasonable objective functions for the Gradual Line Simplification problem. Additionally, we showed how to compute the optimum for 3 of them, while proposing algorithms that effectively handle the remaining 3 in practical scenarios. Furthermore, we provided approximation algorithms for 2 of the problems. One of those utilized a greedy approach and we showcased how we can adapt it for improved efficiency under the Fréchet distance and how the approximation factor changes accordingly. Moreover, we introduced two very successful heuristics, namely the Random Order heuristic and the Tree Order heuristic. Whenever algorithms demonstrated excessively large runtimes, we effectively presented viable alternatives by modifying the algorithms at the expense of quality. Finally, we presented an algorithm for output-sensitive simplification extraction, which utilizes linear space.

10 Future Work

In future research it would be highly valuable to discover algorithms that offer guaranteed optimal solutions for the remaining three objective functions. Furthermore, it would be of great interest to conduct a study to assess the visual quality of transitions using different objective functions in important applications such as zooming on cartographic maps or displaying large networks at various scales. This would provide valuable insights into how the transitions are perceived by users and their overall aesthetic appeal. Lastly, it would be interesting to compare the proposed simplification algorithms to currently utilized simplification algorithms or progressive line simplification, in order to assess their relative performance.

References

[1] P. K. Agarwal, S. Har-Peled, N. H. Mustafa, and Y. Wang. *Near-Linear Time Approximation Algorithms for Curve Simplification.* Algorithmica, 42(3-4):203–219, May 2005. URL https://doi.org/10.1007/s00453-005-1165-y. [page 14]

[2] H. ALT and M. GODAU. *COMPUTING THE FRÉCHET DISTANCE BETWEEN TWO POLYGONAL CURVES.* International Journal of Computational Geometry &amp Applications, 05(01n02):75–91, Mar. 1995. URL https://doi.org/10.1142/s0218195995000064. [page 3, 12, 17]

[3] K. Buchin, M. Konzack, and W. Reddingius. *Progressive Simplification of Polygonal Curves*, 2018. URL https://arxiv.org/abs/1806.02647. [page 1, 3]

[4] M. Buchin, I. van der Hoog, T. Ophelders, L. Schlipf, R. I. Silveira, and F. Staals. *Efficient Fréchet distance queries for segments*, 2022. URL https://arxiv.org/abs/2203.01794. [page 3, 12]

[5] H. Cao, O. Wolfson, and G. Trajcevski. *Spatio-temporal data reduction with deterministic error bounds.* The VLDB Journal, 15(3):211–228, Apr. 2006. URL https://doi.org/10.1007/s00778-005-0163-7. [page 1, 3]

[6] W. S. Chan and F. Chin. *Approximation of polygonal curves with minimum number of line segments.* In Algorithms and Computation, pages 378–387. Springer Berlin Heidelberg, 1992. URL https://doi.org/10.1007/3-540-56279-6_90. [page 3]

[7] S. Daneshpajouh, M. Ghodsi, and A. Zarei. *Computing polygonal path simplification under area measures.* Graphical Models, 74(5):283–289, Sept. 2012. URL https://doi.org/10.1016/j.gmod.2012.04.006. [page 3]

[8] D. H. DOUGLAS and T. K. PEUCKER. *ALGORITHMS FOR THE REDUCTION OF THE NUMBER OF POINTS REQUIRED TO REPRESENT A DIGITIZED LINE OR ITS CARICATURE.* Cartographica: The International Journal for Geographic Information and Geovisualization, 10(2):112–122, Dec. 1973. URL https://doi.org/10.3138/fm57-6770-u75u-7727. [page 3]

[9] R. Durstenfeld. *Algorithm 235: Random permutation.* Communications of the ACM, 7(7):420, July 1964. URL https://doi.org/10.1145/364520.364540. [page 19]

[10] T. Eiter and H. Mannila. *Computing discrete Fréchet distance.* 1994. [page 3, 13]

[11] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. *Scaling and related techniques for geometry problems.* In Proceedings of the sixteenth annual ACM symposium on Theory of computing - STOC '84. ACM Press, 1984. doi: 10.1145/800057.808675. URL https://doi.org/10.1145/800057.808675. [page 41]

[12] H. IMAI and M. IRI. *Polygonal Approximations of a Curve — Formulations and Algorithms.* In Computational Morphology - A Computational Geometric Approach to the Analysis Of Form, pages 71–86. Elsevier, 1988. URL https://doi.org/10.1016/b978-0-444-70467-2.50011-4. [page 3]

[13] A. MELKMAN and J. O'ROURKE. *On Polygonal Chain Approximation.* In Computational Morphology - A Computational Geometric Approach to the Analysis Of Form, pages 87–95. Elsevier, 1988. URL https://doi.org/10.1016/b978-0-444-70467-2.50012-6. [page 3]

[14] D. Mondal and L. Nachmanson. *A New Approach to GraphMaps, a System Browsing Large Graphs as Interactive Maps*, 2017. URL https://arxiv.org/abs/1705.05479. [page 1]

[15] G. Qingsheng, C. Brandenberger, and L. Hurni. *A progressive line simplification algorithm.* Geospatial Information Science, 5(3):41–45, Jan. 2002. URL https://doi.org/10.1007/bf02826387. [page 1]

[16] M. Visvalingam and J. D. Whyatt. *Line generalisation by repeated elimination of points.* The Cartographic Journal, 30(1):46–51, June 1993. URL https://doi.org/10.1179/caj.1993.30.1.46. [page 3]