Algorithms for Gradual Polyline Simplification

Nick Krumbholz
University of Konstanz, Germany

Stefan Funke ✉
University of Stuttgart, Germany

Peter Schäfer
University of Konstanz, Germany

Sabine Storandt ✉
University of Konstanz, Germany

Abstract

Displaying line data is important in many visualization applications, and especially in the context of interactive geographical and cartographic visualization. When rendering linear features as roads, rivers or movement data on zoomable maps, the challenge is to display the data in an appropriate level of detail. A too detailed representation results in slow rendering and cluttered maps, while a too coarse representation might miss important data aspects. In this paper, we propose the gradual line simplification (GLS) problem, which aims to compute a fine-grained succession of consistent simplifications of a given input polyline with certain quality guarantees. The core concept of gradual simplification is to iteratively remove points from the polyline to obtain increasingly coarser representations. We devise two objective functions to guide this simplification process and present dynamic programs that compute the optimal solutions in $\mathcal{O}(n^3)$ for an input line with n points. For practical application to large inputs, we also devise significantly faster greedy algorithms that provide constant factor guarantees for both problem variants at once. In an extensive experimental study on real-world data, we demonstrate that our algorithms are capable of producing simplification sequences of high quality within milliseconds on polylines consisting of over half a million points.

2012 ACM Subject Classification Theory of computation → Computational geometry; Theory of computation → Design and analysis of algorithms

Keywords and phrases Polyline simplification, Progressive simplification, Fréchet distance

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.19

Funding This work was partailly funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project ID 251654672 – TRR 161.

1 Introduction

Polyline simplification is the process of reducing the complexity of linear structures while ensuring that the output still resembles the input. There is a variety of applications for polyline simplification, including data compression [17, 14], noise reduction in movement trajectories [15, 16], and visualization of linear data on maps [19]. In particular in interactive cartographic visualizations, there are often large data volumes that need to be rendered efficiently. For example, the front-end of a trajectory search engine must be able to display huge amounts sets of trajectory data over a map layer, and map rendering tools that allow customization (e.g. rayshader [18]) must accommodate selected features quickly, see Figure 1. Simplifying the data to the desired level of detail prior to rendering is a suitable step to reduce the data complexity, thereby reducing the rendering time and avoiding visual clutter.

Formally, a polyline $P = p_1, p_2, \dots, p_n$ is defined as a sequence of points and the induced straight line segments between consecutive points. In the classical polyline simplification problem, the input is a polyline P, a distance measure d_X, and a threshold $\varepsilon > 0$.

© Nick Krumbholz, Stefan Funke, Peter Schäfer, and Sabine Storandt;
licensed under Creative Commons License CC-BY 4.0

Konstanzer Online-Publikations-System (KOPS)
URL: http://nbn-resolving.de/urn:nbn:de:bsz:352-2-kbd4mdiqgz1m8

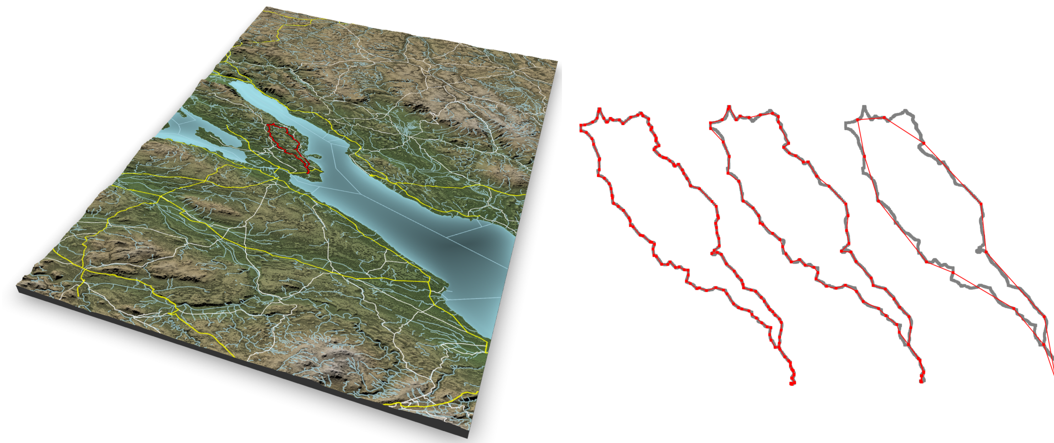

Figure 1 Left: 3D map (created with rayshader [18]), overlaid with linear features extracted from OSM, namely waterways (blue), primary and secondary roads (yellow and white), and a GPS track (red). The waterways and roads consist of roughly 240,000 points in total, the GPS track of 1481 points. Right: Simplifications of the GPS track with 10%, 5% and 1% of the original points in red (original track for visual comparison in gray).

A line segment S_{ij} – also called shortcut – between points p_i and p_j with $j > i$ is called *valid* if $d_X(P[i,j], S_{ij}) \leq \varepsilon$, where $P[i,j]$ refers to the subpolyline of P from p_i to p_j and $d_X(P[i,j], S_{ij})$ denotes the shortcut error. The goal of polyline simplification is to compute a minimum-sized path from p_1 to p_n that only uses valid shortcuts. The endpoints of these shortcuts define the simplified polyline $P' \subseteq P$. Typical similarity measures for polylines are the Hausdorff distance d_H and the Fréchet distance d_F. The polyline simplification problem can be solved optimally in $\mathcal{O}(n^2)$ for d_H [8] and in $\mathcal{O}(n^2 \log n)$ for d_F [22].

However, a single simplification of the given input is often not sufficient for the application scenario. If the linear features are to be displayed on a zoomable digital map, a simplified representation is needed for each zoom level. There are three main guidelines the resulting sequence of simplifications should adhere to:

- *Complexity.* The level of detail should decrease monotonically when zooming out.
- *Conformity.* The shape should be sufficiently preserved across all zoom levels.
- *Consistency.* Once simplified away, polyline points should not reappear later.

To generate such simplification sequences, the concept of *progressive line simplification* problem is used in [5]. Here the input is a polyline P and a sequence of distance thresholds $\varepsilon_1 < \varepsilon_2 < \cdots < \varepsilon_k$. The task is to compute a valid polyline simplification for each ε_i, where the simplification for ε_i has to be a subset of the points chosen for the simplification for ε_{i-1} for all $i > 1$. This subset constraint ensures visual consistency between the simplifications. The optimization goal is to find a sequence of simplifications with the smallest number of shortcuts accumulated over all simplification levels. It was shown in [5] that the problem can be solved to optimality in $\mathcal{O}(n^3 k)$ for both d_H and d_F. However, determining an appropriate sequence of distance thresholds for progressive line simplification is a non-trivial task that demands the user to have knowledge about the polyline structure. For example, it can happen that several consecutive ε values in the sequence induce the same or a very similar simplification, which only adds computational complexity but no further visual benefits. Thus, a continuous version was discussed in [5], where the distance threshold sequence does

not need to be specified, since all possible ε values at which the simplification might change are taken into account. However, the algorithm for computing the respective simplification sequence has a rather impractical running time of $\mathcal{O}(n^5)$.

In this paper, we introduce the *gradual line simplification (GLS)* optimization problem. GLS is based on the concept of iterative point removal to generate a fine-grained and consistent simplification sequence. The objective function ensures that the shape of polyline is well preserved throughout the simplification process. We will discuss two problem variants and show that both can be solved very efficiently in theory and practice. In contrast to progressive line simplification, GLS does not require the user to specify distance thresholds, but simplification sequences with any desired level of detail and with any number of zoom levels can be extracted based on a single GLS solution.

1.1 Related Work

The concept of generalization of geometrical objects for visualization on zoomable maps is well-established in cartography [23, 21, 7, 4]. There are many different incarnations as, e.g., object shrinking or fading, feature elimination, smoothing, or merging of different objects. For linear structures displayed on maps such as country borders, rivers, roads, or GPS-based movement trajectories, polyline simplification is typically applied to obtain coarser representations [25, 11].

Optimal algorithms for polyline simplification rely on first constructing a shortcut graph and computing a shortest link path therein [8, 22]. However, heuristics typically avoid the expensive shortcut graph construction phase and instead select points to be preserved in the simplification in a greedy fashion. Iteratively discarding points as long as the shortcut distance threshold is not violated is a wide-spread algorithm design paradigm here. For example, the heuristic by Visvalingam and Whyatt [24] works by always removing the point from the polyline which together with its neighbors forms a triangle of smallest area. The algorithm runs in $\mathcal{O}(n \log n)$ time. The famous Douglas-Peucker algorithm [9] proceeds the other way around by selecting promising points one by one until the shortcut distance threshold is obeyed. The Douglas-Peucker algorithm can be implemented to run in $\mathcal{O}(n \log n)$ under d_H and in $\mathcal{O}(n^2)$ under d_F [13]. Further examples of iterative simplifications are the algorithms by Wang-Müller [26] or Zou-Jones [10]. All algorithms following the design paradigm of iterative point removal generate a consistent simplification sequence as a by-product. But the existing methods do not provide quality guarantees for the resulting sequence.

To cater for multiple zoom levels, hierarchical simplification was studied e.g. in [19]. In their setup, they have multiple non-intersecting polylines and a sequence of distance thresholds $\varepsilon_1 < \cdots < \varepsilon_k$ and they seek to find for each ε_i simplifications for all polylines that obey the respective distance threshold and additionally remain intersection-free. However, they do not take consistency into account.

Polyline simplification algorithms can also be useful for simplifying polygons or more complex structures as polygonal subdivisions and graphs. For polygons, a single point is chosen as start and end point of the polyline (one could also iterate over all points in that role and record the best result). For subdivisions and graphs, the simplification problem can be reduced to a set of independent polyline simplifications by retaining all nodes with a degree of three or more [11]. The process of iterative point removal and shortcut insertion can however also be applied to graphs with arbitrary node degrees, as explored in [3] for consistent and continuous road network simplification. But the focus there is also on fast heuristics to deal with large inputs.

1.2 Contribution

We introduce gradual line simplification (GLS) as a formal optimization problem and provide several theoretical and practical results. We consider two problem variants, one in which we aim to minimize the maximum error of any introduced shortcut (max-error) and one in which we minimize the sum over all shortcut errors (sum-error). We show that optimal max-error or sum-error solutions can be computed in time $\mathcal{O}(n^3)$ and space $\mathcal{O}(n^2)$. This result applies for shortcut errors being measured by Fréchet or Hausdorff distance. Under the Fréchet distance, we prove that a simple greedy algorithm simultaneously approximates the max-error by a factor of 2 and the sum-error by a factor of 4. This greedy algorithm can be implemented to run in $\mathcal{O}(n^2)$ time. But to achieve this running time heavy machinery is used. Therefore, we also present a more practical greedy variant that only relies on simple methods and allows to trade running time against quality. Furthermore, we propose output-sensitive algorithms to extract any simplification of desired size from the GLS sequence. The underlying data structure has linear size and can be computed in linear time. This allows a user to easily select simplification subsequences of desired complexity. Finally, we provide an extensive experimental study in which we evaluate our methods on real GIS data. As the main outcome, we observe that the proposed approximation algorithms are orders of magnitude faster than the exact algorithms while producing close-to-optimal results.

2 Preliminaries and Definitions

In this section, we provide the formal GLS problem definition, and describe notation and basic algorithmic building blocks that will be used throughout the paper.

2.1 Shortcutting and Gradual Simplification

The core concept of GLS is to obtain a sequence of $n-1$ consistent simplifications of size $n, n-1, \dots, 2$ by iterative point removal from the input polyline. Formally, this removal is referred to as point shortcutting as defined below.

▶ **Definition 1** (Point Shortcutting). *Given a polyline $P = p_1 \dots, p_n$ and a point p_i with $i \in \{2, \dots, n-1\}$, the shortcutting operation removes p_i from P and creates $S_{i-1,i+1}$.*

Based on this notion, GLS is defined as follows.

▶ **Definition 2** (Gradual Line Simplification (GLS)). *Given a polyline $P = p_1, \dots, p_n$ and a point shortcutting order $\pi : \{p_2, \dots, p_{n-1}\} \to \{1, \dots, n-2\}$, we denote the sequence of polyline simplifications $P_0, \dots, P_{n-2}$ as gradual simplification of P where $P_0 = P$ and P_i is derived from P_{i-1} by shortcutting the point p_j with $\pi(p_j) = i$. The total set of shortcuts that results from the respective point shortcutting operations is denoted by $\mathcal{S}_\pi(P)$.*

Figure 2 illustrates GLS on a small example instance.

According to the definition, the final simplification always consists solely of the shortcut from the start to the end point of the input polyline, that is $P_{n-2} = p_1, p_n = S_{1n}$. We observe that any shortcutting order results in a consistent polyline simplification sequence as we clearly ensure $P_i \subset P_j$ for any $i > j$. Furthermore, the simplification sequence is the most fine-grained one possible among all consistent simplification sequences, i.e., it contains the maximum number of distinct simplifications.

To guide the gradual simplification process, we devise two different optimization problems. Both aim at simplification sequences that preserve the shape of the input polyline.

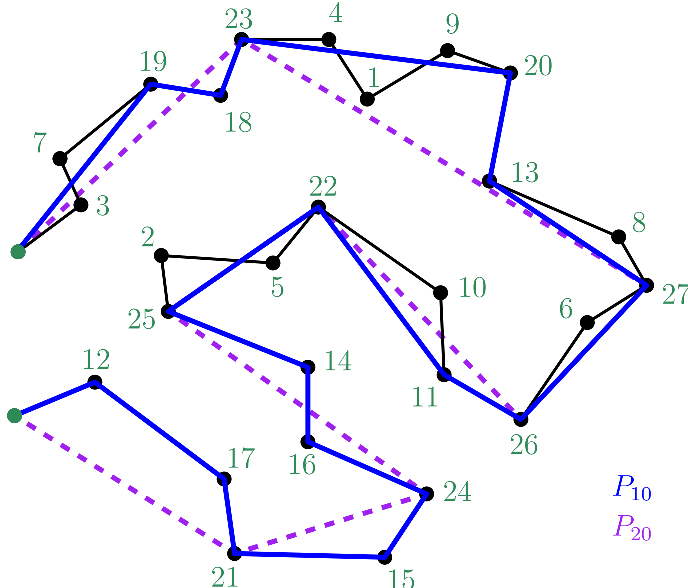

Figure 2 Example polyline (black) with $n = 29$ nodes. A shortcutting order of the inner points is indicated by the green labels. Additionally, the resulting simplifications P_{10} (solid blue, containing only points with a label higher than 10) and P_{20} (dashed purple, containing only points with a label higher than 20) are displayed.

▶ **Definition 3** (Max-error GLS). *Given a polyline $P = P_1, \ldots, p_n$ and a distance measure d_X, find a shortcutting order π that minimizes the maximum shortcut error under d_X, that is,* $\min \max_{\mathcal{S} \in S_\pi(P)} \varepsilon(S)$.

▶ **Definition 4** (Sum-error GLS). *Given a polyline $P = P_1, \ldots, p_n$ and a distance measure d_X, find a shortcutting order π that minimizes the sum of shortcut errors under d_X, that is,* $\min \sum_{\mathcal{S} \in S_\pi(P)} \varepsilon(S)$.

2.2 Shortcut Error Computation

The GLS problem and the algorithms we will design crucially rely on access to shortcut errors $\varepsilon(S_{ij}) := d_X(P[i,j], S_{ij})$ under a given distance measure. Throughout this paper, we focus on the Hausdorff distance d_H and the Fréchet distance d_F. Classical polyline simplification algorithms usually only require an oracle that decides whether a shortcut S_{ij} for $1 \leq i < j \leq n$ has distance at most ε to its respective subpolyline $P[i,j]$. Such an oracle can be implemented for both d_H and d_F in time $\mathcal{O}(n)$. However, for gradual line simplification, the actual shortcut errors are relevant. Under d_H, such errors can be computed in $O(n)$. Under d_F, using the algorithms by Alt and Godau [2], the error can be computed in $\mathcal{O}(n^2)$ in a quite simple fashion, or in $\mathcal{O}(n \log n)$ if parametric search is applied. We will refer to the more expensive variant also as vanilla error computation method. Recently, Buchin et al. [6] introduced a Fréchet distance data structure (FDS) that allows to preprocess a given polyline in time and space $\mathcal{O}(n \cdot k^{3+\delta} + n^2)$ for $\delta > 0$ and then answers distance queries from any line segment to a selected subpolyline in time $\mathcal{O}(\frac{n}{k} \log^2 n + \log^4 n)$ for some of $k \in [n]$. We will exploit this data structure with a careful choice of k to compute relevant shortcut errors faster in our algorithms.

3 Exact Algorithms

We first show that the max-error GLS and the sum-error GLS problem can be solved to optimality in polytime under both d_H and d_F. In particular, we design efficient dynamic programs (DP) that require $\mathcal{O}(n^3)$ time and $\mathcal{O}(n^2)$ space.

The main observation is that we can decompose our problem into subproblems which encode whether or not a certain shortcut is contained in the solution. Let $\varepsilon(S_{ij})$ denote the error of shortcut S_{ij}. Further, let $\varepsilon_{\max}(S_{ij})$ denote the max-error of an optimal solution of the gradual simplification problem restricted to $P[i,j]$, which implies that S_{ij} is part of the solution. Similarly, $\varepsilon_{sum}(S_{ij})$ denotes the optimal sum-error of gradual polyline simplification for $P[i,j]$. Let p_k be the last point shortcut before p_i and p_j, i.e. the point whose shortcutting resulted in the insertion of S_{ij}. Then the shortcuts (or original segments) S_{ik} and S_{kj} are part of the solution as well, and we have $\varepsilon_{max}(S_{ij}) = \max\{\varepsilon_{\max}(S_{ik}), \varepsilon_{\max}(S_{kj}), \varepsilon(S_{ij})\}$ and $\varepsilon_{sum}(S_{ij}) = \varepsilon_{sum}(S_{ik}) + \varepsilon_{sum}(S_{kj}) + \varepsilon(S_{ij})$. Based on these formulas, we can construct the solution set recursively starting with S_{1n}, which always has to be contained in any solution. However, we don't know the value of k if we go top-down. But this can easily be overcome by iterating over all possible k with $i < k < j$ and picking the minimum resulting max- or sum-error. We can store already computed solutions for S_{ij} in a look-up table to avoid redundant computations. This results in the following dynamic program: We allocate an $n \times n$ table and initially set all entries to 0. In cell c_{ij} with $i < j$, we store $\varepsilon_{\max}(S_{ij})$ or $\varepsilon_{sum}(S_{ij})$, respectively. The table cells are filled by using the above formulas. As we always need access to all shortcuts of smaller hop length to get the correct results, we consider the cells sorted increasingly by $j - i$.

▶ **Theorem 5.** *The DP approach solves max-error and sum-error GLS in time $\mathcal{O}(n^3)$ under d_H and d_F, respectively, using quadratic space.*

Proof. The created table has a space consumption of $\Theta(n^2)$. The running time is determined by the time needed to fill the $\Theta(n^2)$ cells. To get the correct cell value c_{ij}, we first need to compute $\varepsilon(S_{ij})$ and then iterate over all values k between i and j and check cells c_{ik} and c_{kj}. The latter part can be accomplished in constant time per considered cell and hence takes $\mathcal{O}(n)$ in total. The computation of $\varepsilon(S_{ij})$ depends on the distance measure. For d_H, it takes time $\mathcal{O}(n)$. For the Frèchet distance, as discussed above, this would take $\mathcal{O}(n \log n)$ when using the parametric search technique. However, based on the FDS by Buchin et al. with a choice of $k \in \Theta(n^{1/3})$, preprocessing the polyline and determining all potential shortcut errors can be accomplished in $o(n^3)$ using $\mathcal{O}(n^2)$ space. Thus, we spend on average $O(n)$ time per cell for both d_H and d_F, which amounts to an overall running time of $\mathcal{O}(n^3)$. ◀

Figure 3 shows an example where the optimal outcomes for sum-error and max-error coincide. If in the optimal sum-error solution the max-error is assumed by the final shortcut S_{1n}, the solution is automatically also optimal for max-error. However, as visible in the example, the max-error is not necessarily assumed by S_{1n} and thus the two problem variants have different solutions in general.

4 Approximation Algorithms under the Fréchet Distance

The cubic running time and the quadratic space consumption of the exact algorithm might be prohibitive in practice, especially when dealing with large inputs. Therefore, we next investigate approximation algorithms with better performance. We will present constant-factor approximations for both max-error and sum-error GLS under d_F. Our algorithms rely on a well-known lemma by Agarwal et al. [1], rephrased below in our terminology and illustrated in Figure 4.

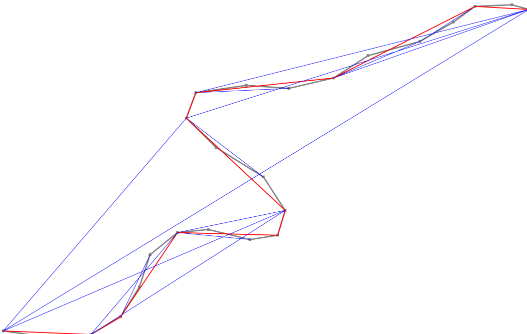

Figure 3 Example polyline (grey) and optimal gradual simplification (blue and red shortcuts) for sum-error and max-error. The red polyline corresponds to the respective simplification of the input polyline after half the points have been shortcut.

▶ **Lemma 6.** *Given a polyline P, consider a shortcut S_{ij} with error ε. Then for any shortcut S_{ab} with $i \leq a < b \leq j$, its error under d_F is bounded by $d_F(P[a,b], S_{ab}) \leq 2\varepsilon$.*

We remark that for any distance measure where the factor between the error of S_{ij} and S_{ab} is bounded by a constant, the algorithms we describe below will yield constant-factor approximations for max-error and sum-error GLS. As this does not apply to d_H, though, our approximation results do not transfer to the Hausdorff distance. However, the Fréchet distance is usually deemed the more appropriate polyline distance measure anyhow, as it takes the course of the polyline explicitly into account while the Hausdorff distance treats the polyline as unordered set of contained points.

4.1 2-Approximation for Max-Error GLS

We first consider the max-error GLS problem. In the following lemma, we show the somewhat surprising result that *any* shortcutting order produces a maximum shortcut error within a factor of 2 of the optimal one under the Freéchet distance.

▶ **Lemma 7.** *Any order π is a 2-approximation for max-error GLS under d_F.*

Proof. According to the GLS definition, we know that for any π we have $S_{1n} \in \mathcal{S}_\pi(P)$. Thus, $\varepsilon(S_{1n})$ is a lower bound for the optimum value OPT. Lemma 6 implies that for any possible $S_{ab} \in \mathcal{S}_\pi(P)$ it holds $\varepsilon(S_{ab}) \leq 2\varepsilon(S_{1n})$ and hence $\varepsilon(S_{ab}) < 2OPT$. As this applies for any shortcut error it clearly also applies to the maximum shortcut error in any set $\mathcal{S}_\pi(P)$. ◀

The lemma illustrates that solely focusing on the max-error does not provide enough guidance to obtain practically useful simplification sequences. Thus, we will from now on focus on the sum-error variant.

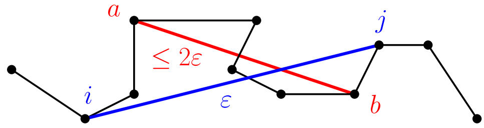

Figure 4 Illustration of Lemma 6.

4.2 4-Approximation for Sum-Error GLS

In the following we present and analyze a simple greedy algorithm for sum-error GLS: Given the current simplification P_i, we always choose the next point $p \in P_i$ to remove by selecting the one whose shortcutting results in the currently smallest shortcut error among all candidates. We will now prove that this strategy yields a constant-factor approximation algorithm and subsequently investigate its running time and space consumption.

▶ **Theorem 8.** *GREEDY is a 4-approximation algorithm for sum-error GLS under d_F.*

Proof. Let P be a polyline of size n. Let $S_1, \ldots, S_{n-2}$ be the shortcuts created by the greedy algorithm in their insertion order and let $\pi_1, \ldots, \pi_{n-2}$ denote the points in P according to their contraction order. Further, let $S_1^*, \ldots, S_{n-2}^*$ be the shortcuts inserted based on an optimal point ordering. We use P_i or P_j^* to refer to the subpolylines shortcut by S_i or S_j^*, respectively. As before, we use $\varepsilon(S)$ to denote the shortcut error of a shortcut S.

We construct an assignment of shortcuts inserted by the greedy algorithm to optimal shortcuts. In particular, we assign shortcut S_i to S_j^* if the following two conditions are met:

- At the moment before π_i is shortcut by the greedy algorithm, there are at least three points in P_j^* that are not yet shortcut, including π_i.
- Index j is the smallest index in the optimal ordering for which the above property holds.

Note that the assignment is well-defined, as we have $S_{n-2}^* = s_{1n}$ and the first condition is always true for S_{n-2}^* as we never shortcut its endpoints.

We now show that if S_i is assigned to S_j^* it holds $\varepsilon(S_i) \leq 2\varepsilon(S_j^*)$. This applies because if at least three points of P_j^* are not shortcut before the shortcutting of point π_i, we know that shortcutting the middle of these three points would result in a shortcut for a subpolyline of P_j^*. According to Lemma 6, the error of any such shortcut is upper bounded by $2\varepsilon(S_j^*)$. As the greedy algorithm selects the next point to shortcut based on the minimum possible induced error at the current stage, we thus conclude that $\varepsilon(S_i) \leq 2\varepsilon(S_j^*)$ has to hold as well.

Let now c_j be the number of shortcuts S_i assigned to a particular shortcut S_j^* in the optimal solution. Then the greedy sum-error can be upper bound by $\sum_{i=1}^{n-2} \varepsilon(S_i) \leq \sum_{j=1}^{n-2} c_j \cdot 2\varepsilon(S_j^*)$.

To complete the proof, we will argue that $c_j \leq 2$ for all $j = 1, \ldots, n-2$. Assume now for contradiction that there exists a shortcut S_j^* to which we assigned at least three greedy shortcuts. Let the respective points that resulted in these shortcut insertions in that order be q_1, q_2, q_3. Based on the assignment criterion, we have $q_1, q_2, q_3 \in P_j^*$. Furthermore, there need to be three points that are not shortcut yet at the point greedy considers q_3 in order for the respective shortcut to get assigned to S_j^*. Hence we have at least also $q_4, q_5 \in P_j^*$ that are shortcut after q_3. Now we consider the point p^* that was shortcut by S_j^* in the optimal solution. At that moment, there were only the points p_l, p^*, p_r left in L_j^*, where p_l is the left endpoint of S_j^* and p_r the right one. Hence the shortcuts (or original line segments) $S_l = (p_l, p^*)$ and $S_r = (p^*, p_r)$ exist in the optimal solution. If those are shortcuts, their index needs to be smaller than j, as they were constructed before S_j^*. Now if we have q_1, q_2, q_3, q_4, q_5 in P_j^*, at least three of them have to be contained in either the subpolyline belonging to S_l or S_r. W.l.o.g. assume it is S_l. That automatically implies that S_l is indeed a shortcut and not an original segment. Now if we shortcut the q_i with smallest index i in S_l (which implies $i \leq 3$), we assign S_i to S_l, as S_l has a smaller index than S_j^*, and there are at least three not yet shortcut points in the respective subpolyline including q_i. This contradicts our claim that S_i is assigned to S_j^*. We therefore conclude that $c_j \leq 2$ holds. Accordingly, we get $\sum_{j=1}^{n-2} c_j \cdot 2\varepsilon(S_j^*) \leq \sum_{j=1}^{n-2} 2 \cdot 2\varepsilon(S_j^*) = 4 \sum_{j=1}^{n-2} \varepsilon(S_j^*) = 4 \cdot OPT$. ◀

According to Lemma 7 and Theorem 8, a greedy ordering provides simultaneously a 2-approximation for max-error and a 4-approximation for sum-error.

▶ **Lemma 9.** *GREEDY runs in* $\mathcal{O}(n^2 \log n)$ *using linear space.*

Proof. The greedy algorithm needs to compute linearly many shortcut errors. The initial errors of shortcuts S_{ii+2} for $i = 1, \ldots, n-2$ can be computed in $\mathcal{O}(1)$. Then, after selecting the next point to shortcut, only the values for its two former neighbors need to be updated which amounts to $\mathcal{O}(n)$ non-trivial error computations. If we use parametric search for shortcut error computation, the overall running time is $\mathcal{O}(n^2 \log n)$. As only linearly many shortcut errors need to be stored at any time, the space consumption is linear. ◀

For a faster implementation of the greedy approach, we can again exploit the FDS by Buchin et al., however, at the cost of an increased space consumption.

▶ **Lemma 10.** *GREEDY can be implemented to run in* $\mathcal{O}(n^2)$ *time using quadratic space.*

Proof. Recall that FDS requires a preprocessing time of $\mathcal{O}(n \cdot k^{3+\delta} + n^2)$ and has a query time of $\mathcal{O}(\frac{n}{k} \log^2 n + \log^4 n)$ for some $k \in [1, n]$. As we need to issue $\mathcal{O}(n)$ shortcut error queries, the total time of `GREEDY` based on FDS can be expressed as $\mathcal{O}(n \cdot k^{3+\delta} + n^2 + \frac{n^2}{k} \log^2 n + n \log^4 n)$. If we choose k to be slightly smaller than $n^{1/3}$, e.g. $k = n^{(3-\delta)/9}$, then all summands are in $\mathcal{O}(n^2)$. Selecting $n-2$ times the one with minimum error among the current $\mathcal{O}(n)$ shortcut candidates can clearly also be accomplished in $\mathcal{O}(n^2)$. ◀

This implementation of the `GREEDY` algorithm thus matches the space consumption of the `DP` for exact sum-error GLS computation but reduces the running time by a factor of n.

4.3 A More Practical Greedy Approach

Both greedy variants described in Lemma 9 and Lemma 10 rely on sophisticated search techniques or data structures and are cumbersome to implement. If we use the vanilla approach by Alt and Godau [2] with a running time of $\mathcal{O}(n^2)$, the implementation is much easier. However, the resulting running time is in $\mathcal{O}(n^3)$ and thus too slow for large input polylines. Therefore, we now discuss another variant of the greedy algorithm tailored to practical implementation. That is, we only want to use easy-to-implement subroutines but get competitive running times nevertheless.

To achieve this goal, we first observe that the greedy algorithm does not necessarily require the knowledge of the precise shortcut errors of all candidates, but only needs to identify the currently smallest one. Thus, it would suffice to identify a threshold ε such that one candidate has a smaller error and the others a larger one. This concept would allow us to only use the decision oracle whether a shortcut exhibits an error of at most ε as subroutine (just like in classical polyline simplification). Binary search over the range of possible ε values until we have the desired division of candidates sounds appealing. But this approach suffers from the issue that if two shortcuts exhibit very similar errors, then the number of search steps might be huge; potentially even incurring a larger running time than the naive implementation using vanilla shortcut error computation. But if we abort the binary search after a fixed number of search steps, then there might be still several candidates in the remaining interval and the ratio between their errors could be unbounded (in particular if the lower bound of the current search interval is still zero). If that error ratio is unbounded, the greedy algorithm loses its approximation guarantee. Thus, we need a sensible stopping criterion for the search that allows us to bound the number of steps as well as the approximation factor. The next theorem shows that this is indeed possible.

▶ **Theorem 11.** *GREEDY can be implemented to run in $\mathcal{O}(c \cdot n^2 \log_b n)$ time using linear space, with an approximation factor of $(4b + \frac{1}{n^c})$ for any $b, c > 1$.*

Proof. Let $\varepsilon = \varepsilon(S_{1n})$ be the shortcut error of the final shortcut, which is necessarily a part of any GLS. Accordingly, we have $OPT \geq \varepsilon$, and we know by Lemma 6 that all shortcut errors are bounded by 2ε.

In every round, we then proceed as follows with the two newly created shortcut candidates: We first invoke the decision oracle for 2ε. As long as the oracle returns true, i.e., that the shortcut errors is below or equal to the threshold, we divide the current bound by b and repeat. We abort the process if the oracle returns false or if the bound drops below $\frac{\varepsilon}{n^c}$ for some $c \geq 1$. This happens after at most $c \cdot \log_b(n) + \log_b(2) \in \mathcal{O}(c \log_b n)$ rounds. Then, we choose one of the shortcut candidates with smallest known error upper bound for the next insertion. The overall running time is in $\mathcal{O}(c \cdot n^2 \log_b n)$ as there are less than n rounds, and in each round we conduct at most two searches with $\mathcal{O}(c \log_b n)$ steps each, where a single search step consists of invoking the decision oracle which takes linear time. The value of ε to initialize the searches can be computed using the vanilla approach in $\mathcal{O}(n^2)$.

To prove the approximation guarantee, we first consider the shortcuts that were inserted because their error is at most $\frac{\varepsilon}{n^c}$. As there are less than n shortcuts in total, the summed error is upper bounded by $\frac{n\varepsilon}{n^c} \leq OPT \cdot n^{1-c}$. For the shortcuts that came with a larger error, we identified an interval with its lower bound being within a factor of b of the upper bound. Now if we perform the same assignment of shortcuts in the greedy solution to shortcuts in the optimal solution as described in Theorem 8, we still assign at most two greedy shortcuts to each optimal shortcut S_j^*. For the greedy shortcut S_i this implies that at the time of its insertion there was also a candidate shortcut with error at most $2\varepsilon(S_j^*)$. As the binary search approximates the error within a factor of b, we thus can guarantee that $\varepsilon(S_i) \leq 2b \cdot \varepsilon(S_j^*)$. Plugging this into the sum formula from the proof of Theorem 8, we end up with an accumulated error of at most $4b \cdot OPT$. Combined with the $OPT \cdot n^{1-c}$ error bound from the first part of the analysis, we hence get an overall approximation guarantee of $(4b + \varepsilon) \cdot OPT$ where $\varepsilon = n^{1-c}$ can be made as small as desired. ◀

The theorem allows a simple implementation of the greedy algorithm that trades running time against approximation quality. As b is the crucial parameter for this trade-off, we will from now on refer to this greedy variant as b-GREEDY. For practical application, it is also worthwhile to use a priority queue (PQ) data structure (e.g. a min-heap) to take care of the selection of the next best shortcut to insert, reducing the running time with the naive sweep from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$. Here, we use the upper bounds of the error intervals as key. While the overall running time is still dominated by the upper bound searches in the worst case, the reduced selection time still has an impact in practice.

5 Indexing and Output-Sensitive Extraction

The result of GLS of an input polyline P is a s sequence consisting of $n-1$ simplifications. For some applications the sequence will be used as a whole (e.g. morphing between representations [20], or continuous zooming [23]), for other applications we might only want to use a certain subsequence at a time (e.g. when dealing with a fixed number of zoom levels) or even only select a single simplification of desired size.

Thus, we would like to have fast access to individual simplifications. Given a polyline P and a shortcutting order π, we want to index this data such that for given k, the simplification consisting of k line segments (that is the one after contracting $n-k-1$ points) can be

extracted in $\mathcal{O}(k)$. A simple way to achieve this is to store all $n-1$ simplifications explicitly. But this requires quadratic space. We note that it suffices for given k to retrieve the set of points p with $\pi(p) \geq n-k$ in their order of occurrence in P. The corresponding shortcuts are simply the line segments between consecutive points in the ordered set. This gives rise to the following simple data structure: We store the points of P in the given order in an array and augment them with their π-values. We also store the position of the point contracted last, that is, the point with $\pi(p) = n-2$. Furthermore, for each point p_i, we store the index of the point p_l in $P[1, i-1]$ with largest π-value smaller than that of p_i such that for all points $p' \in [p_l + 1, p_i - 1]$, we have $\pi(p') < \pi(p_l)$. Similarly, we store the point p_r in $P[i+1, n]$ with largest π-value smaller than that of p_i such that for all points $p' \in [p_i + 1, p_r - 1]$, we have $\pi(p') < \pi(p_r)$. If p_l or p_r do not exist, we store a default index instead. Clearly, the data structure has linear space consumption. It can also be computed in linear time based on an algorithm for 1D range maxima queries [12].

In a query, where $k \in [n]$ is provided by the user, we use the constructed data structure as follows. If $k = 1$, we simply return the first and the last point in the array. For $k > 1$, we initialize a double linked list P' with p_1, p_n and then insert the point p with $\pi(p) = n-2$ into the middle of this list. As its position information is precomputed, it only takes constant time to access the point and to look up its stored values for p_l and p_r. If $\pi(p_l) \leq n-k$, we insert p_l as left neighbor of p in P' and proceed recursively. Similarly, if $\pi(p_r) \leq n-k$, we insert it as right neighbor of p in P' and then proceed recursively as well. Clearly, inserting a new element in P' as neighbor of a currently considered element takes constant time. In total, we insert $k-1$ additional elements after initialization. For each element or point p, we spend constant time on checking the π-values of the point itself as well as p_l and p_r, respectively. Thus, overall the extraction time is $\mathcal{O}(k)$, making the described algorithm output-sensitive.

We further remark that if the user wants to extract a simplification subsequence from the data structure with sizes $k_1 < k_2 < \cdots < k_s$, a total running time of $\mathcal{O}(k_s \log k_s)$ can be achieved no matter the value of s. We simply need to ensure in the query for the simplification of size k_s that whenever we consider multiple candidate points with $\pi(p) \leq n - k_s$, we add the one with highest π value first. In that way, we faithfully reconstruct the whole suffix of the simplification sequence down to the one of size k_s and thus can easily return any requested subsequence. Keeping track of the maximum takes $\mathcal{O}(k_s \log k_s)$ in total when using a max-heap data structure. Of course, if only a subsequence is needed, one could attempt to directly solve the optimization problem where given a size sequence $k_1 < k_2 < \cdots < k_s$, the goal is to find a consistent simplification sequence adhering to the given sizes with smallest maximum or summed shortcut error. The advantage of instrumenting GLS is that it serves as a lightweight data structure from which any desired sequence can be extracted quickly, e.g. to cater for different devices with varying rendering capabilities or to dynamically adapt the zoom level. Consistency is always guaranteed by construction.

6 Experimental Study

For experimental evaluation, we implemented the proposed algorithms for GLS in C++. Experiments were conducted on a single core of an AMD Ryzen processor at 4.2 GHz.

All algorithms were implemented using the Fréchet distance to measure shortcut errors. The best theoretical running time bounds for `DP` and `GREEDY` rely on FDS. However, this data structure is very sophisticated and was not implemented so far. Also, the parametric search variant by Alt and Godau to compute shortcut errors in $\mathcal{O}(n \log n)$ is quite involved. The authors themselves recommend to rather use the vanilla variant with a running time

Table 1 Running times of the implemented algorithms for GLS when shortcut errors under d_F are computed in $\mathcal{O}(n^2)$. Approximation factors for max- and sum-error GLS are provided below.

	max-DP	sum-DP	GREEDY	b-GREEDY
time	$\mathcal{O}(n^4)$	$\mathcal{O}(n^4)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2 \log_b n)$
apx max	1	2	2	2
apx sum	-	1	4	$4b + \varepsilon$

of $\mathcal{O}(n^2)$ for practical purposes. The vanilla variant is easy to implement and numerically robust. We hence only use this method for shortcut error computation in our experiments. Table 1 provides an overview of the resulting running times of our proposed algorithms. Furthermore, we implemented four simple baselines:

- `ORDER`. Use the given input point order as order for gradual simplification.
- `RAND`. Use a random permutation of the points.
- `HOPS`. Choose the next shortcut to insert to be the one where the corresponding subpolyline of P contains the smallest number of points, i.e. has smallest hop-distance. The idea is that shortcuts that span fewer points are more likely to induce a small Fréchet error than shortcuts that span long parts of the polyline.
- `AREA`. Choose the next shortcut to insert to be the one with the smallest triangle area, i.e. the triangle formed by the new shortcut and the two shortcuts it is replacing [24].

The first three baseline algorithms can all be implemented to run in linear time. `AREA` takes $\mathcal{O}(n \log n)$ time. By virtue of Lemma 7, all of them provide a 2-approximation for max-error GLS. But none of them comes with an approximation guarantee for sum-error GLS.

6.1 Comparative Performance of GLS Algorithms

To test our algorithms on a diverse set of polylines, we used openly available trajectories from OSM. The database holds almost a million trajectories of varying size. For the experiments, we picked 100 random trajectories of each size (if available) and used different upper bounds on the trajectory size. We always specify along with each experiment which sets were used.

We first conducted comparative experiments on the `DP`, the `GREEDY` algorithm, 2-`GREEDY`, and the four baselines with respect to running time and solution quality. Table 2 provides an overview of the results on instances with up to 1000 points (containing roughly 100000 trajectories). As expected, the DP approach is very slow, taking several minutes already

Table 2 Running times and shortcut errors compared to optimal results (computed with `DP`). Input size $\leq 1,000$ vertices. Running times measured in microseconds, except for `DP` and `GREEDY`.

	running time (μs)		approximation ratio					
			sum error			*max error*		
	avg.	max.	min.	avg.	max.	min.	avg.	max.
`DP`	584 s	2,423 s						
`GREEDY`	3 s	58 s	1.0	1.3	1.4	1.0	1.1	2.0
`2-GREEDY`	897	1,934	1.0	1.5	3.4	1.0	1.1	2.0
`AREA`	290	464	1.0	1.7	7.1	1.0	1.1	2.0
`HOPS`	272	434	1.2	1.5	2.9	1.0	1.0	1.8
`RAND`	233	382	1.5	2.2	4.5	1.0	1.1	2.0
`ORDER`	109	227	2.5	69.2	197.6	1.0	1.1	2.0

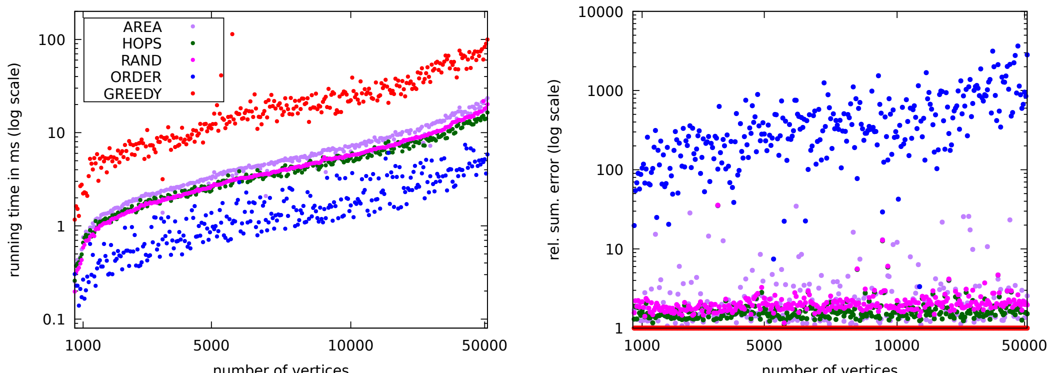

Figure 5 Comparison between 2-GREEDY and the baselines with respect to running time (left) and solution quality (right).

on small input polylines. GREEDY is two orders of magnitude faster than DP, but the exact computation of the Fréchet distance values for the candidates is still quite expensive. 2-GREEDY is vastly faster. On the sample instances, all results were computed in less than 2 milliseconds. This yields a speed-up of more than a factor of 1000 over the GREEDY algorithm with naive Fréchet distance computation. For max-error GLS our experiments show that the theoretical approximation ratio of 2 is tight and that there is little difference in performance among the algorithms. For sum-error GLS, we observe that all approaches except ORDER produce reasonable results. GREEDY performs best and stays way beyond the theoretical approximation factor of 4. 2-GREEDY, with an approximation guarantee of ≈ 8, also performs much better in practice but slightly worse than GREEDY. It thus offers a good trade-off between running time and quality.

In Figure 5, we further compare the running time and the solution quality of 2-GREEDY and the baselines on larger instances. As the DP was too slow to compute exact solutions, we provide a relative quality comparison to the output of 2-GREEDY. As expected, the baseline methods are considerably faster as they do not rely on shortcut error computation. But they also produce results of worse quality. While AREA often comes close to the quality of 2-GREEDY, there are some notable outliers in *sum error*. HOPS is more consistent but also worse than 2-GREEDY on almost all instances. We conclude that 2-GREEDY is the more reliable approach with reasonable running time costs.

6.2 Sensitivity Analysis for b-GREEDY

Next, we strive to investigate the performance of our proposed b-GREEDY algorithm more closely. Above, we used $b = 2$ in the comparative evaluation. Now, we study the influence of the parameter b.

Figure 6 depicts the running times and the solution quality for various values of b ranging from 1.1 to 4. As expected from the theoretical analysis, the running time decreases with growing b but the solution quality deteriorates slightly. Thus, if in a specific application one aspect is more important than the other, b can be chosen appropriately.

Setting $b = 3$, we obtain running times of around 200 milliseconds on average and a maximum of around 6 seconds for trajectories with over half a million points.

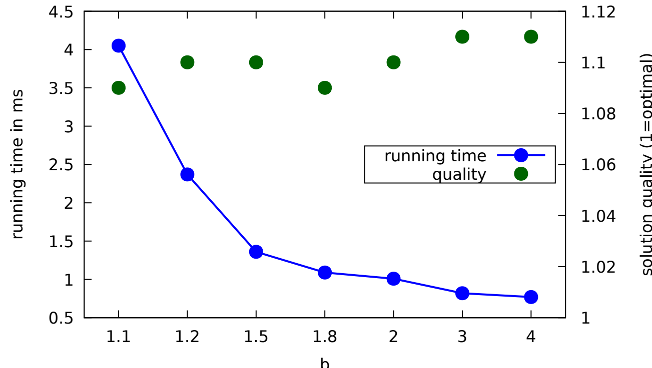

Figure 6 b-GREEDY performance for various values of b. Input sizes ≤ 1000 vertices. Quality relative to optimal $= 1$.

6.3 Further Engineering of b-GREEDY

The practical running time of the b-GREEDY approach can be further reduced in practice by decreasing the number of calls to the shortcut error oracle. We observe that in particular for shortcut candidates with small shortcut errors, our proposed procedure of starting with an oracle call with error $2\varepsilon(S_{1n})$ and proceeding to divide the current error by b until the oracle returns *false* for the first time is quite time-consuming. If we start the procedure with a tighter upper bound on the shortcut error, fewer oracle calls are induced. To get such an upper bound for shortcut S_{ij}, we use $d_F(S_{ij}) \leq max\{d(p_i, p_k), d(p_k, p_j) \mid i < k \leq j\}$. So we simply compute the maximum Euclidean distance of any point in $P[i, j]$ to the shortcut end points p_i and p_j in time $\mathcal{O}(j - i)$. As the time to compute this upper bound is equivalent to that of an oracle call, the overall running time decreases even if only a few oracle calls are saved. A second observation that helps to avoid unnecessary oracle calls is that error intervals do not need to be refined all the way down to the backstop ε/n^c immediately. Instead, if we consider the set of current shortcut candidates, refinements are only necessary until the best candidate is separated from the others. Thus, upon insertion of a new element in the PQ we issue oracle calls with decreasing error bounds until *false* is returned, or until we reach the same upper bound as we have for the current top element in the PQ. In the latter case, we refine both of them further until the oracle returns *false* for one (or both) of them. Thus, especially when the best candidate has a significantly smaller error than the others, oracle calls are saved. We investigate the impact of backstop value and of the proposed

Table 3 2-GREEDY running times as a function of ε_0. Input size $\leq 600,000$. Number of oracle calls per shortcut, and in relation to the worst-case.

	2-GREEDY time (ms)		'oracle'	
ε_0 (m)	avg.	max.	calls	% worst
100	43.5	1961	1.6	75%
10	53.1	2261	5.1	64%
1	65.1	2973	10.4	63%
0.1	80.8	3601	17.8	70%
1e-3	96.7	4171	27.0	60%
1e-5	102.0	4447	30.0	48%

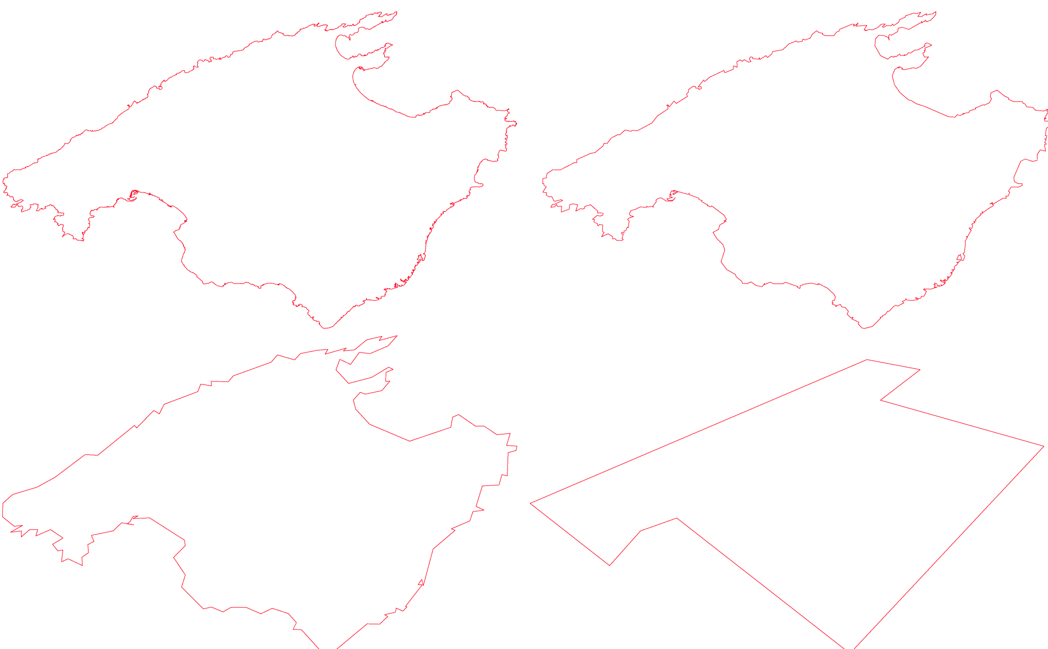

Figure 7 Results of gradual simplification for Mallorca with (from top to bottom) 35000, 548, 137, and 11 points.

engineering on the number of oracle calls for the 2-GREEDY algorithm. Table 3 summarizes our findings. The backstop value ε_0 provides a lower bound for the shortcut error interval refinement process and can be modulated via the choice of c in Theorem 11. As we deal in our experiments with real-world trajectories, we provide ε_0 in meters. As to be expected, the smaller the backstop the larger the running times as more oracle calls need to be issued to differentiate between shortcuts with small errors. As shown in the last column of the table, using improved upper bounds on the shortcut errors and only refining intervals on demand reduces the number of oracle calls by 25-52%. This helps to achieve fast processing times even for huge trajectories with over half a million points.

6.4 Showcase Appplication

As discussed in Section 5, one strength of GLS is that after the solution sequence is computed, an simplification consisting of k segments can be extracted in time $O(k)$ for any given k. This is an important feature for dynamic map rendering, where polylines or polygonal shapes are displayed on demand based on user interaction and selection.

In Figure 7, top, the island of Mallorca is shown as represented in OpenStreetMap with roughly 35000 points. When rendering Mallorca as part of a larger map, it is clearly unnecessary to draw it with all these details. So a rendering engine might ask for a simplification using only few hundred points. This can be easily realized by drawing only the last k points in the GLS ordering where k can be chosen completely freely (e.g. depending on the rendering capability of the device). Figure 7 also shows simplifications of Mallorca using different numbers of points. In a zoomed out view, there is hardly any difference between the representation using 35000 points versus the representation using 137 points, which are less than 0.5% of the input. Again, note that such a representation can be obtained using only 137 steps based on our index structure after GLS computation.

Figure 8 Comparison of GLS based simplification (top) with naive subsampling (bottom) on the port of Mallorca instance using 4382 points each. In the bottom image, shape distortions and visual artifacts are clearly visible.

A naive method that allows for such fast simplification would be to perform simple subsampling, where only every n/k-th point is displayed. While not requiring any preprocessing, the results are considerably worse, as can be seen in a closeup of the port of Mallorca in Figure 8. Here we show a zoomed in view of the port of Mallorca in a simplification consisting of 4,382 points. On top, using GLS, we see that the piers are well preserved, whereas using subsampling as depicted in the bottom picture leads to a heavily distorted shape. Furthermore, using subsampling on different zoom levels violates our requirement of producing consistent simplification sequences. GLS, on the other hand, offers full flexibility in the selection of desired local simplifications and simplification sequences, which are guaranteed to be consistent with each other.

7 Conclusions and Future Work

We proposed the gradual line simplification problem and provided practically useful algorithmic solutions. In particular the b-`GREEDY` algorithm turns out to be an easy to implement and fast approximation algorithm for GLS, which allows to trade running time against quality. Choosing suitable parameters, even huge input lines and line sets can be processed in real time. By providing $n-1$ consistent simplifications of a given input polyline with a single computation at once, GLS caters to applications as continuous zooming or customized level of detail selection.

In future work different objective functions for GLS could be investigated, as e.g. the sum of squared shortcut errors, or the sum over the shortcut errors of each individual simplification. The latter takes the order in which shortcuts are inserted into account, yielding a more intricate optimization problem.

References

1. Pankaj K Agarwal, Sariel Har-Peled, Nabil H Mustafa, and Yusu Wang. Near-linear time approximation algorithms for curve simplification. *Algorithmica*, 42(3), 2005.
2. Helmut Alt and Michael Godau. Computing the fréchet distance between two polygonal curves. *International Journal of Computational Geometry & Applications*, 1995.
3. Lukas Baur, Stefan Funke, Tobias Rupp, and Sabine Storandt. Gradual road network simplification with shape and topology preservation. In *Proceedings of the 30th International Conference on Advances in Geographic Information Systems*, pages 1–4, 2022.
4. Pia Bereuter and Robert Weibel. Algorithms for on-the-fly generalization of point data using quadtrees, 2012.
5. Kevin Buchin, Maximilian Konzack, and Wim Reddingius. Progressive simplification of polygonal curves. *Computational Geometry*, 2020.
6. Maike Buchin, Ivor van der Hoog, Tim Ophelders, Lena Schlipf, Rodrigo I. Silveira, and Frank Staals. Efficient Fréchet Distance Queries for Segments. In *30th Annual European Symposium on Algorithms (ESA 2022)*, 2022. doi:10.4230/LIPIcs.ESA.2022.29.
7. Hu Cao, Ouri Wolfson, and Goce Trajcevski. Spatio-temporal data reduction with deterministic error bounds. In *Proceedings of the 2003 joint workshop on Foundations of mobile computing*, pages 33–42, 2003.
8. Wing Shiu Chan and Francis Chin. Approximation of polygonal curves with minimum number of line segments or minimum error. *International Journal of Computational Geometry & Applications*, 1996.
9. David H Douglas and Thomas K Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: the international journal for geographic information and geovisualization*, 10(2):112–122, 1973.
10. Peter F Fisher, Sheng Zhou, and Christopher B Jones. Shape-aware line generalisation with weighted effective area. In *Developments in Spatial Data Handling: 11 th International Symposium on Spatial Data Handling*, pages 369–380. Springer, 2005.
11. Stefan Funke, Thomas Mendel, Alexander Miller, Sabine Storandt, and Maria Wiebe. Map simplification with topology constraints: Exactly and in practice. In *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 185–196. SIAM, 2017.
12. Harold N Gabow, Jon Louis Bentley, and Robert E Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 135–143, 1984.
13. John Hershberger and Jack Snoeyink. An o (n log n) implementation of the douglas-peucker algorithm for line simplification. In *Proceedings of the tenth annual symposium on Computational geometry*, pages 383–384, 1994.
14. Georgios Kellaris, Nikos Pelekis, and Yannis Theodoridis. Map-matched trajectory compression. *Journal of Systems and Software*, 86(6):1566–1579, 2013.
15. Hengfeng Li, Lars Kulik, and Kotagiri Ramamohanarao. Spatio-temporal trajectory simplification for inferring travel paths. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 63–72, 2014.
16. Kunhui Lin, Zhentuan Xu, Ming Qiu, Xiaoli Wang, and Tianxiong Han. Noise filtering, trajectory compression and trajectory segmentation on gps data. In *2016 11th International Conference on Computer Science & Education (ICCSE)*, pages 490–495. IEEE, 2016.
17. Xuelian Lin, Shuai Ma, Jiahao Jiang, Yanchen Hou, and Tianyu Wo. Error bounded line simplification algorithms for trajectory compression: An experimental evaluation. *ACM Transactions on Database Systems (TODS)*, 46(3):1–44, 2021.
18. Tyler Morgan-Wall. *rayshader: Create Maps and Visualize Data in 2D and 3D*, 2023. https://www.rayshader.com, https://github.com/tylermorganwall/rayshader.

19 Nabil Mustafa, Shankar Krishnan, Gokul Varadhan, and Suresh Venkatasubramanian. Dynamic simplification and visualization of large maps. *International Journal of Geographical Information Science*, 20(3):273–302, 2006.

20 Martin Nöllenburg, Damian Merrick, Alexander Wolff, and Marc Benkert. Morphing polylines: A step towards continuous generalization. *Computers, Environment and Urban Systems*, 32(4):248–260, 2008.

21 Guo Qingsheng, Christoph Brandenberger, and Lorenz Hurni. A progressive line simplification algorithm. *Geo-spatial Information Science*, 5(3):41–45, 2002.

22 Sabine Storandt and Johannes Zink. Optimal polyline simplification under the local fréchet distance in near-quadratic time. In *35th Canadian Conference on Computational Geometry*, 2023.

23 Marc Van Kreveld. Smooth generalization for continuous zooming. In *Proc. 20th Intl. Geographic Conference*, pages 2180–2185, 2001.

24 Mahes Visvalingam and J. Duncan Whyatt. Line generalisation by repeated elimination of points. *Cartographic Journal*, 30:46–51, 1993. doi:10.1179/000870493786962263.

25 Katerina Vrotsou, Halldor Janetzko, Carlo Navarra, Georg Fuchs, David Spretke, Florian Mansmann, Natalia Andrienko, and Gennady Andrienko. Simplifly: A methodology for simplification and thematic enhancement of trajectories. *IEEE Transactions on Visualization and Computer Graphics*, 21(1):107–121, 2014.

26 Zeshen Wang and Jean-Claude Müller. Line generalization based on analysis of shape characteristics. *Cartography and Geographic Information Systems*, 25(1):3–15, 1998.