Global Polyline Simplification Under the Fréchet Distance: Theory and Practice

Christian Abdullahad
University of Konstanz, Germany

Sabine Storandt
University of Konstanz, Germany

Abstract

Given an input polyline with n vertices, the global polyline simplification problem seeks a simplified polyline with the minimum number of vertices whose distance to the original polyline does not exceed a given bound. For the vertex-restricted variant, where the simplified polyline is required to be a subsequence of the input vertices, an algorithm with a running time of $\mathcal{O}(n^3)$ was presented in previous work, using the Fréchet distance as the polyline similarity measure. A closely related variant is the local polyline simplification problem, in which the distance bound is required to hold for every individual shortcut segment replacing a sub-polyline. This condition implies that any locally valid simplification is also globally valid, whereas the converse does not hold. As a consequence, globally optimal simplifications may use substantially fewer vertices than locally optimal ones. Indeed, in previous work, instances were constructed in which the optimal global simplification is smaller by a constant factor. On the algorithmic side, optimal local simplifications can be computed significantly faster, namely in $\mathcal{O}(n^2 \log n)$ under the Fréchet distance, and efficient heuristics are also available. This raises the question of which problem variant is more suitable for practical application.

In this paper, we first show that there exist instances for which the optimal solution sizes of global and local polyline simplification differ by a factor in $\Theta(n)$, substantially strengthening the previously known constant-factor separation. We then present the first practical implementations of existing algorithms for global polyline simplification and experimentally evaluate their performance. To this end, we introduce several engineering techniques that considerably accelerate these algorithms. Moreover, we develop an implicit Fréchet framework that allows many Fréchet-related problems to be addressed in a weaker computational model. Within this framework, explicit geometric computations can be reduced to simple comparisons, resulting in significantly more robust implementations. Somewhat surprisingly, our experimental results reveal that, despite the large worst-case gap established by our theoretical result, the difference in solution size between optimal global and local simplifications is negligible in practice. Motivated by this observation, we propose a heuristic for global polyline simplification that is guaranteed to produce solutions of size equal to or smaller than the optimal local simplification. On a benchmark consisting of one million polylines, the heuristic yields suboptimal results on only eight while being significantly faster than the optimal algorithms.

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

Keywords and phrases Polyline Simplification, Shortcut Graph, Fréchet Distance

Digital Object Identifier 10.4230/LIPIcs.SEA.2026.1

1 Introduction

Polyline simplification is the process of algorithmically reducing the number of vertices of a polyline while preserving its general shape. Applications include the simplification of map contours in geographic information systems [7], noise reduction in movement trajectories [13, 14], and compressing linear data [15, 12].

There are different flavours of polyline simplification, which differ by the constraints the simplified polyline must obey and how the shape similarity to the input polyline is measured. In this paper, we focus on vertex-restricted simplifications, where the simplified polyline must

be a subsequence of the original polyline vertices. The simplification problem then asks for the simplified polyline of smallest size for which the distance to the original polyline does not exceed a given error bound ε. Different distance measures are used in the literature, with the most common ones being the Hausdorff and the Fréchet distance. These distance measures can be applied both locally and globally. In the local version, the error bound must be obeyed for each shortcut between consecutive points in the simplified polyline and the original sub-polyline it replaces. In the global version, the error bound simply needs to be obeyed for the original and the simplified polyline as a whole. The local version has been studied for a long time [11]. Optimal local simplifications in $\mathbb{R}^2$ can be computed in time $\mathcal{O}(n^2)$ under the Hausdorff distance [5] and in $\mathcal{O}(n^2 \log n)$ under the Fréchet distance [17], and practical implementations are available as well. In contrast, the global variant has been studied only recently [19, 18]. Somewhat surprisingly, global simplification under the Hausdorff distance turned out to be NP-hard. For the Fréchet distance, $\mathcal{O}(n^3)$-time algorithms are known [4, 18], but they have not been experimentally evaluated so far.

By definition, any local simplification respecting ε is also a feasible global simplification for the same error bound, but the converse is not true. Therefore, global simplifications might be smaller than local ones, but the price to pay is a substantially higher theoretical running time. The goal of this paper is to compare the practical applicability of local and global simplification. In particular, we are interested in the following two core questions:

- Is it possible to implement known exact algorithms for global polyline simplification such that they scale to large inputs despite their (super)cubic theoretical running times?
- How much do global and local simplification sizes differ in practice?

To answer these questions, we provide novel theoretical insights, present several engineering techniques for global simplification, and conduct a thorough experimental evaluation on synthetic and real data sets.

1.1 Related Work

Local simplification has been extensively studied. Under the Fréchet distance, the Imai-Iri algorithm computes an optimal solution in $\mathcal{O}(n^3)$ by constructing a so-called shortcut graph in which the shortest path coincides with the smallest feasible simplification. The efficiency of the graph construction step was improved in [17], resulting in a running time of $\mathcal{O}(n^2 \log n)$ and significantly faster computation in practice. Furthermore, heuristics are commonly applied, such as the Douglas-Peucker algorithm with a running time of $\mathcal{O}(n^2)$ (and no quality guarantee) [7] and the approximation algorithm by Agarwal et al. [1], which runs in $\mathcal{O}(n \log n)$ and guarantees that the computed simplification size does not exceed the optimal one for a parameter $\varepsilon/2$.

The concept of global simplification was touched upon in [1] and [3], but only in the context of weak simplification or when using the discrete Fréchet distance as the error measure. The systematic study of vertex-restricted global polyline simplification was initiated by van Kreveld et al. [19]. They show that neither the Douglas-Peucker nor the Imai-Iri algorithm produce optimal solutions for this problem variant, and present an exact algorithm with a running time in $\mathcal{O}(k^* n^5)$, where k^* denotes the output size. This running time was later reduced to $\mathcal{O}(n^3)$ by Bringmann and Chaudhury [4] as well as van de Kerkhof, Kostitsyna, Löffler, Mirzanezhad, and Wenk [18]. In [4], a conditional lower bound is presented, making the existence of subcubic algorithms for global simplification in high dimensions unlikely.

The one dimensional case is the only case where subcubic algorithms are known for the Fréchet setting, in fact local and global Fréchet simplifications have the same size and can be computed in linear time. Although this result was established by Driemel et al. [9], its

significance appears to have been overlooked in subsequent literature[1], likely because the original authors framed it within a non-optimality claim. In the setting where the start and end vertex are required to be in the simplification, their finding is, in fact, optimal.

Their algorithm can be stated simply: Include the first polyline vertex in the simplification and traverse all vertices until the first $P(i)$ outside of $[P(0) - \varepsilon, P(0) + \varepsilon]$ is found and go towards the direction where $P(i)$ leaves this range; for simplicity, assume that we go right. Maintain the right-most vertex $P(e)$ found. If $P(e) - P(i) > 2\varepsilon$ for the current vertex i, append $P(e)$ to the simplification and reverse the direction. A similar algorithm also works for the local Hausdorff setting with the small modification that we change direction when a vertex left of $P(i') - \varepsilon$ is found (or right of $P(i') + \varepsilon$ when going left), where $P(i')$ is the previously found vertex in the simplification. The global Hausdorff setting is trivial since we only need to include the start and end vertices as well as potentially the left-most and right-most vertices. If the left-most or right-most vertex is within ε of the start or end, or between the two, we do not need to include them; otherwise, we do.

1.2 Contribution

First, we investigate the maximum possible difference in size between optimal global and local simplification. Previous work established that there can be a constant-factor gap between them [18]. We significantly strengthen this separation by constructing a class of polylines for which optimal global and local simplification differ by a factor in $\Theta(n)$. Thus, our bound is asymptotically tight, rules out local simplification algorithms as approximation algorithms for global simplification, and illustrates the significant potential of simplification size reduction when using global simplification.

To compare the simplification sizes and running times of global and local simplification algorithms in practice, we provide the first implementations[2] of the exact algorithms for global simplification by Bringmann and Chaudhury [4] and by van Kreveld et al. [19]. We describe several engineering techniques to significantly reduce their running times and space consumption. Furthermore, we address the fact that these algorithms rely on geometric computations prone to numerical issues. We present an implicit Fréchet framework that abstracts explicit equation solving into a small set of decision problems. We show that all algorithms discussed here, as well as many related algorithms, can be rewritten using these decision problems. This allows us to assume a weaker model of computation. In the Euclidean case, this allows us to avoid square root computations, thus operating within the regular RAM model of computation. Using this framework in our implementation yields a slower but more robust implementation. Despite the theoretical cubic running time, our experimental evaluation demonstrates that optimal solutions for polylines with hundreds of vertices can be computed in seconds. This allows us to evaluate the difference between global and local simplification sizes on practical instances for the first time. Interestingly, instances on which they do not produce the same solution size are rare and the maximum observed gap is tiny (especially compared to our theoretical worst-case gap example). Finally, using the classic Imai-Iri algorithm as a basis, we derive a global simplification algorithm that is equivalent to the one by the authors of [18] but presented in a more intuitive way. We

[1] For example, [18] rediscovered the linear time algorithm but only for the curve-restricted setting; they do not mention it for the other settings.

[2] The implementation as well as data is publicly available at `https://github.com/Ingemisco/Global-Polyline-Simplification-under-the-Frechet-Distance-Theory-and-Practice`

then show how to modify this algorithm into a simple heuristic that is guaranteed to find a solution at least as good as the local optimal one. In practice, the heuristic outperforms the exact algorithms while yielding optimal solutions on almost all tested instances.

In Appendix A.1, we describe a data structure that can be constructed in $\mathcal{O}(n^4 \log n)$ and returns the optimal global simplification for any given ε in $\mathcal{O}(\log n)$. In Appendix A.2, we provide matching lower bounds for crucial steps in the construction of the data structure.

2 Preliminaries

A d-dimensional polyline of length n is a sequence of points $\langle u_0, \ldots, u_n \rangle$ (called vertices) in d-dimensional space. We interpret a polyline as a continuous functions $P : [0, n] \to \mathbb{R}^d$ with $P(i + t) = (1 - t)u_i + tu_{i+1}$ for $t \in [0, 1]$ and $i \in \{0, 1, \ldots, n - 1\}$. We denote by $P[t' \ldots t]$ the sub-polyline $\langle P(t'), P(\lfloor t' \rfloor + 1), \ldots, P(\lceil t \rceil - 1), P(t) \rangle$. For two points u, v, the polyline of length 1 is also written as $\overline{uv} = \langle u, v \rangle$ and is called line segment or shortcut.

▶ **Definition 1** (Minkowski Distance). *Let $\ell \geq 1$. The ℓ-Minkowski distance δ_ℓ between two points is defined as $\delta_\ell(u, v) = \left(\sum_{i=1}^{d} |u_i - v_i|^\ell\right)^{1/\ell}$. The special case of δ_1 is called the Manhattan distance, δ_2 is the Euclidean distance. We also set $\delta_\infty(u, v) = \max_{i=1,\ldots,d} |u_i - v_i|$ which is the Chebyshev distance.*

▶ **Definition 2** (Fréchet Distance). *Given a distance δ between points, the Fréchet distance between polylines P and Q of lengths p and q, respectively, is*

$$\delta^F(P, Q) = \inf_{f,g} \max_{t \in [0,1]} \delta(P(f(t)), Q(g(t))),$$

where $f : [0, 1] \to [0, p]$ and $g : [0, 1] \to [0, q]$ are continuous non-decreasing functions with $f(0) = g(0) = 0$, $f(1) = p$, and $g(1) = q$.

▶ **Definition 3** (Polyline Simplification). *Fix a distance measure δ' between polylines. Given a polyline P of length n and $\varepsilon > 0$. A subsequence $Q = \langle P(i_0), P(i_1), \ldots, P(i_{k-1}), P(i_k) \rangle$ of P of length k, with $0 = i_0 < i_1 < \cdots < i_{k-1} < i_k = n$, is called*

- *a global simplification of P if $\delta'(P, Q) \leq \varepsilon$.*
- *a local simplification of P if $\delta'(P[i_j \ldots i_{j+1}], \overline{Q(j)Q(j+1)}) \leq \varepsilon$ for all $j \in \{0, \ldots, k - 1\}$.*

If k is minimal, we call the simplifications minimal global simplification and minimal local simplification, respectively.

These are also called vertex-restricted simplifications because the simplification can only select among the polyline vertices. In the remainder of the paper, unless stated otherwise, we are concerned with vertex-restricted polyline simplification in $d = 2$ dimensions under the Fréchet distance with the Euclidean norm.

3 Global vs. Local Simplification Size

Van Kreveld et al. [19] provided an example polyline with ten vertices for which, under a carefully chosen Fréchet distance bound ε, the optimal local simplification contains all input vertices, while the optimal global simplification contains only eight. This polyline gadget can be concatenated to produce a polyline of arbitrary length for which the local simplification size is 1.25 times larger than the global one. This left open the question[3] of whether the local simplification always provides a constant-factor approximation for the global one.

[3] Similar results are already known but to our knowledge unpublished.

We now answer this question negatively.

▶ **Theorem 4.** *For any $n \geq 4$, there exists a polyline with n vertices for which the optimal local simplification is larger than the optimal global simplification by a factor in $\Theta(n)$.*

Proof. We construct a polyline for which the optimal local simplification contains all original vertices, while the optimal global simplification contains only a constant number. The polyline is illustrated in Figure 1. To construct such a polyline of length n, we choose the first

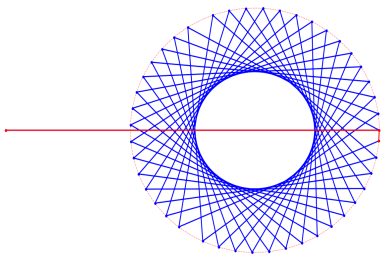


Figure 1 Polyline which maximizes the difference between local and global simplification size.

two vertices such that the shortcut between them passes through the centre and the second vertex lies on the circle of radius ε. We then choose[4] an irrational number $\alpha \in (1/4, 1/3)$ and place all remaining vertices on the circle such that each subsequent vertex forms an angle of $2\pi\alpha$ with the previous vertex and the centre. The exact value of α must be chosen so that $\delta_2(P(n), P(1)) \leq \varepsilon$. The irrationality of α guarantees that no other shortcut passes through the centre. In the figure, the red dotted circle represents the ε-neighbourhood, and the global simplification is shown in red. The optimal global simplification requires only the vertices $P(0), P(1), P(n)$, because the shortcut $\overline{P(0)P(1)}$ passes through the centre of the circle, from which all points are within distance ε. However, no local shortcut that skips at least one vertex exists because, by construction, for any such shortcut there is at least one skipped vertex whose distance to the shortcut exceeds ε. Thus, the local simplification requires all vertices. ◀

4 Algorithms for Global Simplification

In this section, we first briefly recap the exact algorithms for global simplification presented in [19] and [4], which have running times in $\mathcal{O}(k^* n^5)$ and $\mathcal{O}(n^3)$, respectively, where n is the number of vertices in the input polyline and k^* denotes the output size. For more details, we refer to the original papers. Both algorithms use dynamic programming to store and compute all information necessary to construct the simplification. We also describe several engineering techniques to boost their practical performance. Subsequently, we propose a novel heuristic for global simplification. This heuristic is based on a reinterpretation of the algorithm by the authors of [18] and leverages the Imai-Iri algorithm, which was originally introduced for local simplification.

[4] One way to obtain a valid α is as follows: with n vertices approximately $n/3$ rotations around the circle are needed for $\alpha \approx 1/3$. Shrink α minimally such that the condition on the endpoint is met. At most one full rotation from the $\approx n/3$ will be removed. Distributed over all edges $\approx 1/n$ might be subtracted from α which yields a valid α for n large enough. Similar constructions can be found manually for small $n \geq 4$.

4.1 Exact Algorithms

To describe the exact algorithms concisely, we introduce the following notation.

▶ **Definition 5.** *Fix $\varepsilon > 0$ and a distance function δ. Given a line segment e and a point u, we define $\hat{t}_0(e,u) = \min\{t \in [0,1] \mid \delta(e(t),u) \leq \varepsilon\}$ and $\hat{t}_1(e,u) = \max\{t \in [0,1] \mid \delta(e(t),u) \leq \varepsilon\}$. If the set $\{t \in [0,1] \mid \delta(e(t),u) \leq \varepsilon\}$ is empty, we define both functions to be ∞.*

The algorithm by van Kreveld et al. [19] uses dynamic programming with a three-dimensional table DP. The value stored in $DP[k,i,j]$ is the smallest $t \in [0,1]$ for which there exists a partial simplification Q of $P[0 \dots i]$ with exactly k shortcuts such that $\delta^F(Q, P[0 \dots j+t]) \leq \varepsilon$. If no such t exists, we set $DP[k,i,j] = \infty$. If $t = DP[k,n,n-1] \neq \infty$ for some k, a simplification can be extracted from DP because $P[0 \dots n] = P$ and $P(n-1+t)$ lies on the last line segment. The smallest such k is denoted by k^*. For $k = 0$, the entries are computed trivially. For $k > 0$, each layer is computed from the previous one. To compute $DP[k,i,j]$, we iterate over all $DP[k-1,i',j']$ with $i' < i$ and $j' \leq j$, and determine the minimum t for which $\delta^F(P[j'+t' \dots j+t], \overline{P(i')P(i)}) \leq \varepsilon$ (if such a t exists), where $t' = DP[k-1,i',j']$. To test whether this Fréchet distance is within ε and obtain the respective t, a modified[5] version of the algorithm from Alt and Godau [2] can be used which runs in $\mathcal{O}(n)$. Thus, the minimization can be performed in cubic time per cell entry, leading to a total runtime of $\mathcal{O}(k^* n^5)$. For better practical runtimes, we introduce the following trivial optimizations:

- Reachability: If no point on $\overline{P(j)P(j+1)}$ lies within ε of $P(i)$, then $DP[k,i,j] = \infty$ for all k, and no minimization is needed.
- Local Minimality: When computing $DP[k,i,j]$, the minimization can be stopped early if the computed minimum equals $\hat{t}_0(\overline{P(j)P(j+1)}, P(i))$.
- Global Minimality: If $DP[k-1,i,j] = \hat{t}_0(\overline{P(j)P(j+1)}, P(i))$, this value can be copied to $DP[k,i,j]$.

The correctness of the first optimization is straight-forward. For the local and global minimality optimizations, correctness follows from comparing with the lower bound for $DP[k,i,j]$. Finally, we note that the computations for each layer can be parallelized because they depend only on the previous layer.

To reduce the running time to $\mathcal{O}(n^3)$, Bringmann and Chaudhury split the minimization for computing $DP[k,i,j]$ into two cases: $j' = j$ and $j' < j$. The value of $DP[k,i,j]$ is then the minimum of the values computed for these two cases. The case $j' = j$ can be solved in constant time per cell entry by maintaining a minimum. The case $j' < j$ is more complicated; they introduce the cell reachability problem to compute n minimal cell entries in linear time, yielding constant amortized time per cell entry. Their modifications restructure the algorithm because of the more complex data flow. The outermost loop is over i instead of k, resulting in total runtime of $\mathcal{O}(n^3)$ rather than the expected $\mathcal{O}(k^* n^2)$[6].

The more complicated data flow prevents the application of the optimizations discussed above for the van Kreveld et al. algorithm. Furthermore, parallelization is difficult to apply in this restructured algorithm. Thus, especially for small polylines, the van Kreveld et al. algorithm may be faster in practice.

[5] We do not need to explicitly compute the Fréchet distance allowing a simpler algorithm.

[6] Revisiting their conditional lower bound proof rules out $\mathcal{O}(f(k^*)n^{3-\beta})$ algorithms for any f and $\beta > 0$, as the simplification in their reduction has constant size.

4.2 A Heuristic for Global Simplification

Next, we discuss how to adapt the classic Imai-Iri algorithm, originally proposed for local simplification, to the global case. This yields an algorithm equivalent to the one from the authors of [18] but provides a fresh perspective. We argue that this algorithm achieves cubic (and possibly subcubic) runtime for certain classes of polylines and use this observation to formulate a new heuristic that has the potential for improved running time in practice.

4.2.1 Local and Global Imai-Iri

The Imai-Iri algorithm [11] for local simplification relies on the construction of a shortcut graph whose vertices correspond to the polyline vertices, with directed edges $(P(i), P(j))$ for $i < j$ if $\overline{P(i)P(j)}$ is a local shortcut, i.e., $\delta^F(P[i \dots j], \overline{P(i)P(j)}) \leq \varepsilon$. The simplification is the shortest path from $P(0)$ to $P(n)$ in the shortcut graph.

We first derive the notion of a global shortcut graph (GSG) for global simplification. Unlike in the local setting, storing only whether a valid shortcut exists is insufficient because each shortcut may correspond to multiple sub-polylines $P[a \dots b]$ with $a \leq b$. For a shortcut e, a sub-polyline $P[a \dots b]$ is called *e-admissible* if $\delta^F(P[a \dots b], e) \leq \varepsilon$. For brevity, we denote sub-polylines $P[a \dots b]$ by the pair (a, b). We need to compute and store all e-admissible sub-polylines for each shortcut e. Admissibility depends on the points near the endpoints of e. For the rest of this section $a, b, c, d, s, t \in [0, n]$ are real-valued points of the polyline and $i, j \in \{0, \dots, n\}$ are vertices.

▶ **Definition 6.** *For a vertex $P(i)$, the non-empty set $C_P(i) = \{t \in [0, n] \mid \delta(P(i), P(t)) \leq \varepsilon\}$ is the disjoint union of at most n closed intervals (at most one per line segment). We denote the ordered list of these intervals by $I_P(i)$. Intervals may contain multiple line segments, and for each i there is a unique interval in $I_P(i)$ containing i. Moreover, if (a, b) is $\overline{P(i')P(i)}$-admissible, then $a \in C_P(i')$ and $b \in C_P(i)$.*

▶ **Lemma 7.** *Let $e = \overline{P(i')P(i)}$ be a shortcut and (s, t) be e-admissible.*
1. *If $s \leq s' \leq t$ with $s' \in C_P(i')$, then (s', t) is e-admissible.*
2. *If $s \leq t' \leq t$ with $t' \in C_P(i)$, then (s, t') is e-admissible.*
3. *If $t, t'' \in I \in I_P(i)$, and $s \leq t''$, then (s, t'') is e-admissible.*
4. *If $s, s'' \in I \in I_P(i')$, and $s'' \leq t$, then (s'', t) is e-admissible.*

Proof. For all of these properties note that each cell in the free space diagram is convex. By the preconditions there is a path through the cell which can be adapted to the desired path. Refer to Figure 2. ◀

By Lemma 7, $\overline{P(i')P(i)}$-admissibility is a property between intervals $I \in I_P(i')$ and $J \in I_P(i)$. We say that (I, J) is e-admissible if (a, b) is e-admissible for some $a \in I$ and $b \in J$. Note that we still cannot go backward so any such (a, b) is e-admissible only if $a \leq b$. Further, for a fixed I, the admissible intervals J are all consecutive in $I_P(i)$, meaning that no inadmissible interval in $I_P(i)$ is between two admissible ones, allowing to store all of them as one meta-interval.

▶ **Definition 8** (Global Shortcut Graph)**.** *The global shortcut graph of a polyline P of length n is the directed graph $G_P = (V, E)$ where $V = \{0, \dots, n\}$ and $E = \{(i, j) \in V \times V \mid i < j\}$. Additionally, for each $i \in V$, we store the ordered list $I_P(i)$. For each shortcut $\overline{P(i')P(i)}$, we store the admissible sub-polylines as an ordered list of pairs of intervals $(I_1, \mathcal{J}_1), \dots, (I_k, \mathcal{J}_k)$. Each I_i is an interval from $I_P(i')$, and the corresponding $\mathcal{J}_i$ is the smallest closed interval containing all intervals $J \in I_P(i)$ for which (I_i, J) is admissible.*

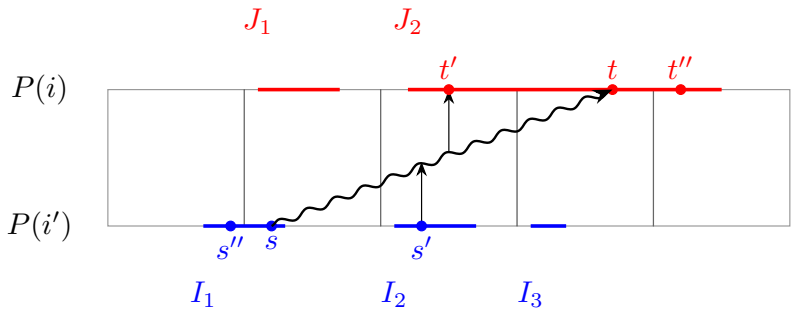


Figure 2 Idea of Lemma 7. For the first two statements, an existing admissible path can be extended in the free space by going straight upward in a cell as long as the endpoints are in the free space. For the final two statements, the path only needs to be made longer on one side while staying in the same interval.

We denote this ordered list of pairs of intervals for the shortcut $\overline{P(i')P(i)}$ by $M_P(i', i)$ and call its elements mappings.

For an example, refer to Figure 3. This shows all the data for the shortcut $e = \overline{P(0)P(1)}$. Paths that witness admissibility in the free space diagram are shown in black.

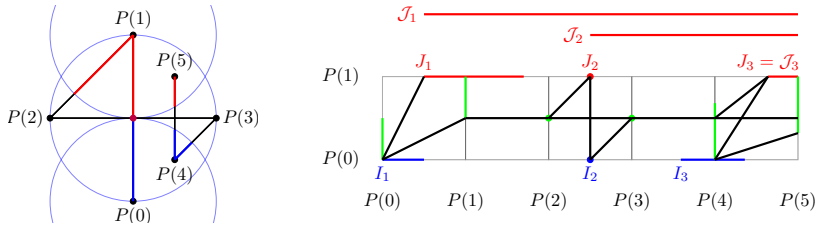


Figure 3 Example for the shortcut $\overline{P(0)P(1)}$. Here, $I_P(0) = (I_1, I_2, I_3)$ and $I_P(1) = (J_1, J_2, J_3)$. $M_P(0, 1)$ contains the three mappings $(I_1, \mathcal{J}_1), (I_2, \mathcal{J}_2), (I_3, \mathcal{J}_3)$.

With this data structure, testing whether a sub-polyline (a, b) is $\overline{P(i')P(i)}$-admissible is simple: we check that $a \leq b$, $a \in C_P(i')$, $b \in C_P(i)$, and $(a, b) \in I_k \times \mathcal{J}_k$ for some[7] mapping $(I_k, \mathcal{J}_k) \in M_P(i', i)$. The interval data I_P per vertex can be computed in linear time per vertex, yielding total quadratic runtime. Computing the mappings M_P can be done in worst-case linear time per edge using a procedure similar to the cell reachability algorithm from [4] (the free space diagram in Figure 3 has almost identical structure to cell reachability).

For each shortcut $\overline{P(i')P(i)}$ maintain an initially empty queue[8] of active intervals $I \in I_P(i')$ while traversing the polyline and intervals in $I_P(i')$. If the queue is empty, skip in the polyline to the next interval in $I_P(i')$. Otherwise, process the vertices like in cell reachability. If a new interval in $I_P(i')$ starts, push it to the front of the queue with a value of 0. Between two consecutive cells there are passage (in Figure 3 shown in green) that restrict how paths

[7] The pair is unique since a is contained in at most one such interval I_k.
[8] Bringmann and Chaudhury maintain an ordered list; a queue suffices.

are allowed to pass to the next cell. When transitioning to the next cell, remove all intervals whose assigned value is above the passage from the back of the queue and merge all intervals whose value is below the passage at the front of the queue and assign as new value the start of the passage.

Thus, the graph can be computed in cubic time, dominated by the computation of the mappings. In practice, computing the mappings can be accelerated as follows:

1. If the queue is empty, skip to the cell corresponding to the start of the next interval in $I_P(i')$.
2. When inserting an interval into an empty queue, skip all cells until the end of the interval (accounting for intervals in $I_P(i)$ that intersect it).
3. If no further interval exists in $I_P(i)$, stop.

With these optimizations, the construction time can be quadratic for favourable polylines and values of ε. Having computed all data, we traverse the graph to construct the simplification. This traversal is more complicated than in the original Imai-Iri algorithm. For each vertex i, we compute all sub-polylines $[0, a]$ for which there exists a partial simplification Q using the vertices $P(0), P(1), \ldots, P(i)$ (starting at $P(0)$ and ending at $P(i)$) such that $\delta^F(Q, P[0 \ldots a]) \leq \varepsilon$. These sub-polylines are represented as intervals. For $P(0)$, we store the interval $[0, a]$ in $I_P(0)$ and associate it with an empty simplification. For a vertex $P(i)$, we iterate over all previous vertices $P(i')$ and attempt to append the shortcut $\overline{P(i')P(i)}$. This involves traversing the reachable data for i' and the mappings $M_P(i', i)$ in a merge-like procedure. Finally, we consider intervals of the form $[b, n] \in I_P(n)$ and select the smallest simplification found.

The runtime depends on the amount of data that must be considered per vertex. For each vertex i and interval $I \in I_P(i)$, at most n different simplifications (one per possible length) may be needed. Specifically, if any point in I is reachable with a partial simplification, then all points to the right in I are reachable with the same simplification. Since we need only minimal partial simplifications, this data can be sorted in decreasing order of simplification size. Propagation per vertex can be done in quadratic time per shortcut, leading to cubic time per vertex and a total quartic time. Intervals can be split in advance to allocate space, or lazily as needed. Lazy splitting occurs only for mappings of the form $([a, b], [c, d])$ with $c < a$, because only the subinterval $[a, d]$ is reachable (we cannot traverse backward). If such *negative* mappings (for an example, the mapping (I_3, J_2) in Figure 2 is a negative mapping) are rare, the algorithm runs in cubic time, or even in quadratic time if the number of intervals is small. This is an advantage over the Bringmann and Chaudhury algorithm, whose running time is $\Theta(n^3)$. Without negative mappings, only one entry per interval is needed, resulting in quadratic space usage[9].

4.2.2 Heuristic

The algorithm described above produces the optimal simplification in $\mathcal{O}(n^4)$ time. However, for GSGs without negative mappings, the running time is cubic or better. Not all polylines produce such GSGs. Therefore, we propose the following transformations which turn the exact algorithm into a heuristic:

- For each $i \in V$, split the unique interval $[a, b] \in I_P(i)$ containing i into $[a, i]$ and $[i, b]$.
- For each negative mapping $([a, b], [c, d]) \in M_P(i', i)$, shrink the interval $[c, d]$ to $[a, d]$ in $I_P(i)$.

[9] This requires computing edge data in the GSG only when needed, an optimization also applicable to the local Imai-Iri algorithm.

The second step modifies negative mappings but may also remove some mappings because the shrunken interval may no longer correspond to a valid mapping. To ensure that a simplification is always found, we use the first step. This guarantees that all local shortcuts remain in the graph because interval $[a, i]$ always contains i (we only shrink from the left). Thus, the computed simplification is at least as good as the optimal local simplification and is suboptimal only if the interval structure is very intricate. For example, for the polyline constructed in Figure 1, the heuristic yields the optimal global simplification, demonstrating that it can produce significantly smaller simplifications than any local simplification algorithm.

5 Implicit Fréchet Framework

Typically, the Fréchet distance is studied in the real RAM model, where equations for $\hat{t}_0$ and $\hat{t}_1$ (see Definition 5) can be solved with arbitrary precision in constant time. This simplifies runtime analysis, but algorithms require sufficiently accurate approximations. For the Manhattan and Chebyshev distance, these equations use only addition, subtraction, and multiplication, so the regular RAM suffices. The Euclidean case, however, requires square roots for quadratic equations. Higher Minkowski distances lack closed-form solutions due to the Abel-Ruffini theorem. In the implicit Fréchet framework, we replace these explicit solution computations with new primitives. The key insight is that only comparisons are needed. Instead of explicit solutions, we provide solvers for the comparisons.

These techniques to avoid square roots in the Euclidean case are already known in the field of computational geometry. Our primitives reduce to inequalities of the form $a+\sqrt{b} < c \pm \sqrt{d}$ which can be decided without the use of square roots (e.g., refer to [6]). Here, we present a structured approach to apply such techniques to Fréchet-based problems, especially polyline simplification. We chose to include this discussion since we studied and implemented all relevant primitives for polyline simplification and no previous global simplification algorithms were implemented.

For any fixed $\varepsilon > 0$, we define three decision problems:

▶ **Definition 9.** *Let e be a line segment and u, v be points.*

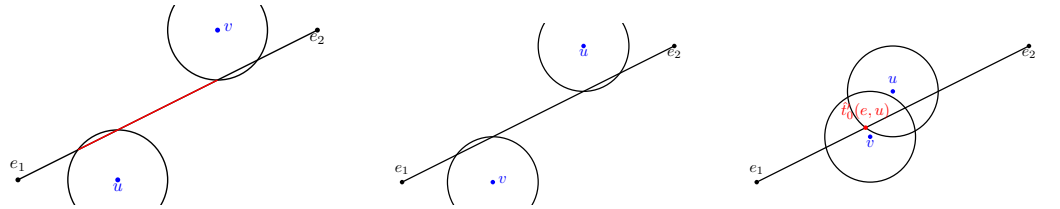
- *u reaches e (denoted $u \leftrightarrow e$) if $\hat{t}_0(e, u) \neq \infty$; equivalently, a point on e is within ε of u.*
- *If $u \leftrightarrow e$, then u proceeds to v on e (denoted $u \xrightarrow{e} v$) if $v \leftrightarrow e$ and $\hat{t}_0(e, u) \leq \hat{t}_0(e, v)$.*
- *If $u \leftrightarrow e$, then u waits for v on e (denoted $u \xleftarrow{e} v$) if $v \leftrightarrow e$ and $\hat{t}_0(e, v) \leq \hat{t}_0(e, u) \leq \hat{t}_1(e, v)$.*

These relations $\xrightarrow{e}$ and $\xleftarrow{e}$ can also be viewed in the dog-walking intuition that is often used when it comes to the Fréchet distance. The dog acts as the restriction while the path that you travel on is the line segment e. If the dog runs away, it forces you to move ($\xrightarrow{e}$) to a new position. The dog may also walk somewhere in your surrounding, in which case you can just wait ($\xleftarrow{e}$).

Note that $\leftrightarrow$ can be implemented using $\xrightarrow{e}$ because $u \leftrightarrow e$ if and only if $e(0) \xrightarrow{e} u$. We include it for intuition, for ease of solution, and because it is sometimes necessary. Examples are shown in Figure 4.

Many Fréchet algorithms only track earliest reachable points, which can only proceed or wait but never go backward. This behaviour is modeled by $\xrightarrow{e}$ and $\xleftarrow{e}$. Instead of an explicit earliest point, we store the vertex whose solution causes it, called a *restriction*.

Table 1 summarizes how to translate various explicit computations with implicit ones. This includes all operations necessary for the described exact algorithms for global simplification. We use $\leftarrow$ for variable assignment and $\xleftarrow{e}$ for the wait relation.



(a) $u \xrightarrow{e} v$. The reachable point moves from $\hat{t}_0(e,u)$ to $\hat{t}_0(e,v)$, and v becomes the new restriction.

(b) Neither $u \xrightarrow{e} v$ nor $u \xleftarrow{e} v$.

(c) $u \xleftarrow{e} v$. The reachable points cannot move backward, but v is reachable, we can wait at $\hat{t}_0(e,u)$ u remains the restriction.

Figure 4 Illustration of the relations and how to interpret them.

Table 1 Explicit-to-Implicit Translation Table.

Explicit operation	Implicit operation	Comment
$t \leftarrow \hat{t}_0(e,u)$	$r \leftarrow u$	r replaces solution t
$t_0 \leq t_1$	$r_0 \xrightarrow{e} r_1$	t_0, t_1 are solutions on e, and r_0, r_1 are restrictions
$t \leftarrow \min(t_0, t_1)$	if $r_0 \xrightarrow{e} r_1$ then $r \leftarrow r_0$ else $r \leftarrow r_1$	t, t_0, t_1 are solutions on e and r, r_0, r_1 are restrictions
$t \leftarrow \max(t_0, t_1)$	if $r_0 \xrightarrow{e} r_1$ then $r \leftarrow r_1$ else $r \leftarrow r_0$	t, t_0, t_1 are solutions on e and r, r_0, r_1 are restrictions
$\hat{t}_0(e,u) \leq t \leq \hat{t}_1(e,u)$	$r \xleftarrow{e} u$	r is the restriction for t
$\hat{t}_0(e,u)$ exists	$u \leftrightarrow e$	

To solve these relations in Euclidean space, we write the explicit solutions for $\hat{t}$ and compare them directly. This results in inequalities of the form $a - b \leq c \pm \sqrt{d}$ since $\hat{t}$ are solutions to quadratic equations. How to decide such inequalities is already well-established.

5.1 Applying the decision problems

We demonstrate the conversion using the Alt and Godau algorithm [2], which decides whether $\delta^F(P,Q) \leq \varepsilon$. For simplicity, we assume $Q = \overline{uv}$ is a line segment (as in the minimization step of van Kreveld et al.'s algorithm). The general case and other algorithms convert similarly. Let n be the length of P. In the explicit variant, the value $\hat{t}_0(e,u)$ is computed and stored dynamically. In our implicit variant, we instead store restrictions (vertex indices). First, test whether $\delta(u, P(0)) \leq \varepsilon$ and $\delta(v, P(n)) \leq \varepsilon$. For general Minkowski distances, this test may involve higher-order roots, but it can be performed without them by raising both sides to the appropriate power. The algorithm tracks the earliest reachable point on Q while traversing P. We replace these points with restrictions. The initial restriction is $P(0)$; note that $P(0) \leftrightarrow Q$ because $\delta(P(0), Q(0)) \leq \varepsilon$. Let r be the current restriction. When processing vertex $P(i)$, test whether $r \xrightarrow{Q} P(i)$. If so, update $r \leftarrow P(i)$. If $r \xleftarrow{Q} P(i)$ keep r. Otherwise, report $\delta^F(P,Q) > \varepsilon$.

Listing 1 Explicit AG.

```
1  AG(P[j'... j+1], P(i')P(i), t', ε)
2  if j' = j
3    t_0 = t̂_0(P(j)P(j+1), P(i))
4    t_1 = t̂_1(P(j)P(j+1), P(i))
5    if t_0 = ∞ or t_1 < t'
6      return ∞
7    return max(t', t_0)
8
9  fr = 0
10
11 for index = j'+1...j
12   t_0 = t̂_0(P(i')P(i), P(index))
13   t_1 = t̂_1(P(i')P(i), P(index))
14   if t_0 = ∞ or t_1 < fr
15     return ∞
16   fr = max(fr, t_0)
17
18 return t_0(P(j)P(j+1), P(i))
```

Implicit AG.

```
AGI(P[j'... j+1], P(i')P(i), r', ε)
if j' = j
  e = P(j)P(j+1)
  if P(r') -e-> P(i)
    return i
  if P(r') <-e- P(i)
    return r'
  return ∞
r = i'
e = P(i')P(i)
for index = j'+1...j
  if P(r) -e-> P(index)
    r = index
  else if not P(r) <-e- P(index)
    return ∞
if P(i) ↔ P(j)P(j+1)
  return i
return ∞
```

In Listing 1, we show both the explicit and implicit version of the algorithm from Alt and Godau as used in the algorithm from van Kreveld et al. This version takes as additional input a parameter $t' \in [0,1]$ and outputs a $t \in [0,1]$ or ∞ as mentioned in Section 4.1.

5.2 Semiexplicit Algorithms

The semiexplicit approach provides another method for implementing Euclidean Fréchet algorithms without square roots. It combines characteristics of both explicit and implicit algorithms. In the Euclidean case, the decision problems require only comparisons of the form $a - \sqrt{b} \leq c \pm \sqrt{d}$, which can be decided without evaluating square roots. The idea of semiexplicit Euclidean variants is to define data types that store such (a, b) and (c, d) and define appropriate comparison operators between them. In programming languages that support parametric polymorphism, the code for semiexplicit and explicit variants can be identical. For example, in C++, we can use templates and operator overloading to implement algorithms. By instantiating the templates with different data types, we obtain both explicit and semiexplicit variants. This makes it straightforward to implement both variants while retaining the benefit of avoiding square root computations.

6 Experimental Results

The goal of this section is to compare the running times and output sizes of local and global simplification algorithms. We implemented the Imai-Iri algorithm to obtain optimal local simplifications, the two discussed exact algorithms for global simplification, and our novel heuristic. We refer to the algorithm by van Kreveld et al. as KLW and the algorithm by Bringmann and Chaudhury as the BC algorithm. All algorithms were implemented to work with the Manhattan, Chebyshev, and Euclidean norms. For the Euclidean norm, we implemented explicit, semiexplicit, and implicit variants. All algorithms were implemented in C++ and compiled with GCC using optimization flags -O3 and -flto. The variants of the KLW algorithm were parallelized using OpenMP with dynamic scheduling and a chunk size 32. All experiments were run on a machine with an AMD Ryzen 5 7520U CPU (4.38 GHz, 8 cores) running Arch Linux.

6.1 Benchmark Data Sets

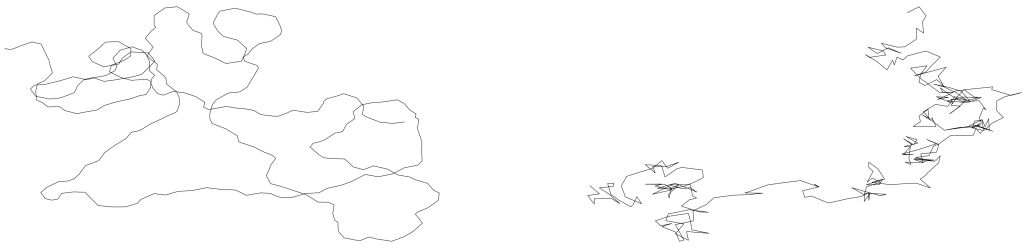
We use synthetic and real-world data in our experiments. Synthetic polylines were generated using a custom method that takes as input the vertex count, minimum and maximum segment lengths m and M, and maximum deviation angle α between consecutive line segments (e.g., 180° means no restriction, 0° yields a straight line). Smaller values of α produce smoother polylines. Deviation angles are drawn uniformly from $[-\alpha, \alpha]$. Segment lengths are generated via inverse transform sampling using the formula length $= \sqrt[d]{m^d + (M^d - m^d)U}$, where $U \sim$ Uniform[0,1] and d is the dimension. This ensures that the vertices are distributed uniformly in the annulus section defined by d, m, M, and α. We set $m = 2$ and $M = 10$, respectively. We created two benchmarks sets; example instances from each are shown in Figure 5:

- well-behaved (w) polylines with a maximum angle of 60°, and
- non-well-behaved (n) polylines with a maximum angle of 180°.

Unless stated otherwise, we use $\varepsilon = 2$ in all experiments. We use this because for our choices of parameters for data generation it often yielded interesting simplifications. Smaller ε tend to use almost all vertices and large ε often use a constant number of vertices. The choices for m, M, d, α for the data generation were arbitrary.

Our real-world data consists of 100 openly available GPS traces[10] with vertex counts between 100 and 800. We used the longitude and latitude coordinates, translated each trace to start at the origin, and scaled them so that the median segment length is 10. Translation does not affect the simplifications, and scaling only changes the effective value of ε.

6.2 Runtimes



(a) Well-behaved polyline with 300 vertices. **(b)** Non-well-behaved polyline with 300 vertices.

Figure 5 Comparison of well-behaved and non-well-behaved polylines.

First, we compare the effect of our three proposed optimizations on the runtime of the KLW algorithm (Figure 6). We observe that they significantly reduce the running time for all norms and algorithm configurations. For example, in the implicit Euclidean setting, the runtime decreases from about 6.0 seconds (non-well-behaved) and 5.4 seconds (well-behaved) to less than 300 milliseconds for both.

Next, we compare the KLW algorithm (with all optimizations) and the BC algorithm (Figure 7). The KLW algorithm is competitive for polylines of length up to 200, after which the BC algorithm dominates. The BC algorithm's runtime is smooth due to its rigid structure which yields a cubic running time on all instances.

[10] Data from https://www.openstreetmap.org/traces/ uploaded December 29–30 and January 9, converted from GPX.

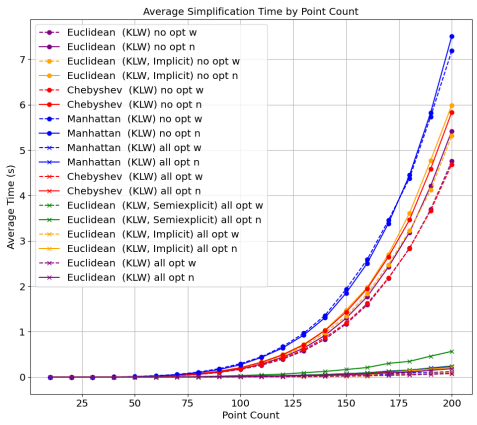


Figure 6 KLW algorithm with and without optimizations.

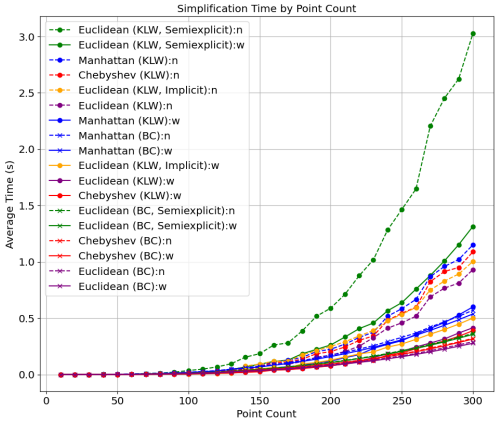


Figure 7 KLW and BC algorithm.

Nevertheless, our implementation allows us to compute exact global simplifications for polylines with up to 800 vertices within a few seconds (see Figure 8). For larger polylines, space consumption becomes the bottleneck on our hardware. On the feasible instances, we also compare the scalability of the BC algorithm and our heuristic in the same figure. The heuristic is approximately 200 times faster, and this speedup grows significantly for larger polylines. Moreover, since the heuristic requires only quadratic space, it can be applied to substantially longer polylines. The heuristic achieved optimal results on all instances, which makes the comparison to the optimal algorithm more fair.

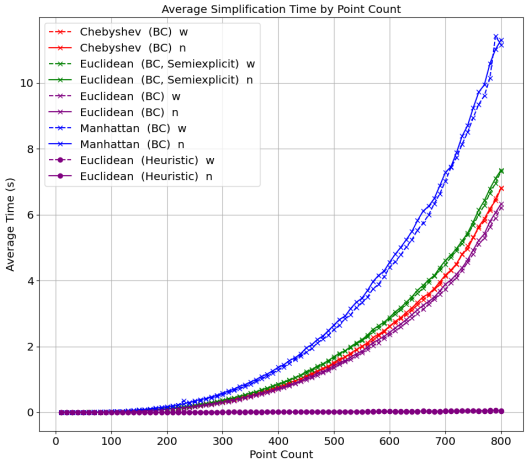


Figure 8 BC algorithm and heuristic for polylines of length ≤ 800 on well-behaved and non-well-behaved data. This uses only two polylines per point count.

6.3 Simplification Sizes

One of the main practical questions is how much global and local simplifications differ on different types of polylines. To answer this, we compare the sizes of optimal global and local simplification of well-behaved and non-well-behaved polylines (Figure 9). The plot contains

four data sets, but they are visually difficult to distinguish because the output sizes are, somewhat surprisingly, almost identical for local and global simplification. Indeed, for well-behaved polylines, the sizes were always identical, while for non-well-behaved polylines we observed only marginal differences. The plot also shows that well-behavedness dramatically affects simplification sizes because much longer shortcuts are feasible on well-behaved data. To compute the local simplifications we used a naïve implementation of the Imai-Iri algorithm. This already outperforms the BC algorithm as can be seen in Figure 10.

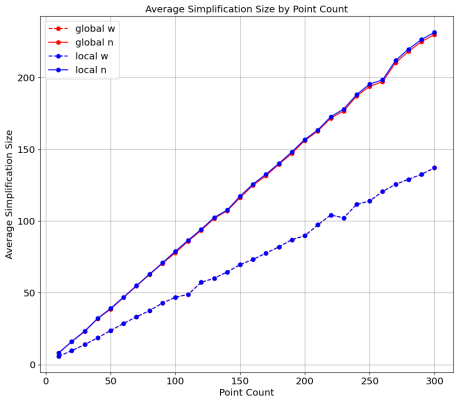


Figure 9 Sizes of local and global simplifications on well-behaved and non-well-behaved data. This uses only Euclidean distance.

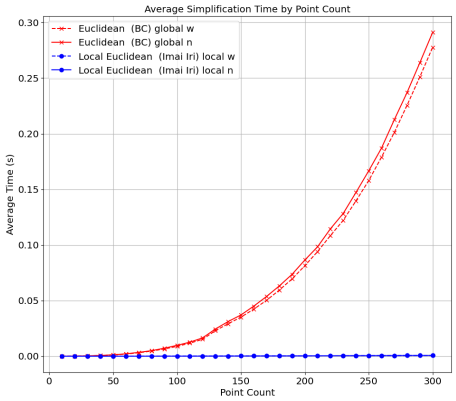


Figure 10 Runtimes of Figure 9.

Given the small difference between global and local simplification sizes, our heuristic (which by construction produces an output size between those two) is nearly always optimal. Testing one million random polylines of length 100, we found only eight instances where the heuristic was non-optimal, with the worst case requiring two extra vertices. By comparison, approximately 10% of these polylines had differing local and global sizes.

To ensure that the similarity between global and local simplification sizes is not an artifact of our parameter choices, we performed an additional experiment with 500000 polylines of 150 vertices each. These were generated with random maximum segment lengths and deviation angles, as well as random ε. In this experiment, the average difference in size was 0.3 vertices, with a maximum difference of 8 vertices.

6.4 Results on Real Data

Finally, we compare the runtime and the result size of the BC algorithm, the local algorithm and the heuristic on the real-world data using $\varepsilon = 2, 8, 32, 128$. Here, the heuristic was always optimal. The local simplification was suboptimal on only 5 out of the 400 test cases. We also compute for each ε the total runtime of the BC algorithm and of the heuristic and show the ratio in Table 2. For small ε, the heuristic is almost 90 times faster, while for larger ε it is still 12 times faster than the BC algorithm. We also compare the heuristic against our implementation of the local Imai-Iri algorithm. Surprisingly, it is only 2-4 times slower, which illustrates that the overhead introduced by the interval handling in the heuristic is rather small. In conclusion, simply using local simplification is the most efficient way to obtain a simplification of good quality in practice (especially as it can be computed even faster than

Table 2 Runtime comparison of BC, heuristic, and Imai-Iri algorithm on real-world data.

ε	BC / heuristic	heuristic / local
2	88.24	3.86
8	59.26	2.96
32	28.04	2.65
128	12.40	3.13

with Imai-Iri [17]). However, our heuristic also scales well and occasionally produces better solutions. On our example instance that illustrates the $\Theta(n)$ gap between global and local simplification, the heuristic indeed produces the optimal results.

7 Conclusion and Future Work

Local Fréchet simplifications are known to be globally optimal in 1D. We show that in practice this still holds almost true in 2D on our tested benchmarks. We could only identify meaningful differences to optimal global simplifications in crafted examples. This questions the practical utility of global Fréchet simplification.

Our implicit algorithms are slower than explicit variants but are theoretically more appealing. A practical advantage of them is that, anecdotally, they cause fewer numerical problems that lead to crashes than explicit variants, especially for small ε and collinear vertices in the polyline. Our semiexplicit variant provides a reasonable trade-off.

There are several open questions that result from this work:

- Are there practically relevant polyline classes where local and global simplifications differ vastly? Conversely, under which conditions are they similar? For example, on polylines with certain attributes as c-packedness [8] or low density [10], it might be possible to prove that local and global simplification sizes are always close.
- Is our heuristic an approximation? If not, can other splitting/narrowing strategies yield an approximation algorithm outperforming optimal ones?
- Can local techniques be used to improve the runtime of the exact algorithms or the heuristic? Optimal local simplifications can be computed in near quadratic time [17]. We also derived our algorithms from the Imai-Iri framework. Given this new perspective, local acceleration techniques might become applicable for the global setting.

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):203–219, 2005. doi:10.1007/S00453-005-1165-Y.

2 Helmut Alt and Michael Godau. Computing the Fréchet distance between two polygonal curves. *International Journal of Computational Geometry & Applications*, 05(01n02):75–91, 1995. doi:10.1142/S0218195995000064.

3 Sergey Bereg, Minghui Jiang, Wencheng Wang, Boting Yang, and Binhai Zhu. Simplifying 3d polygonal chains under the discrete Fréchet distance. In *Latin American Symposium on Theoretical Informatics*, pages 630–641. Springer, 2008. doi:10.1007/978-3-540-78773-0_54.

4 Karl Bringmann and Bhaskar Ray Chaudhury. Polyline simplification has cubic complexity. *Journal of Computational Geometry*, 11(2):94–130, 2021. doi:10.20382/JOCG.V11I2A5.

5 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.

6 Jacobus Conradi, Ivor van der Hoog, and Eva Rotenberg. On computing the (exact) Fréchet distance with a frog. *arXiv preprint arXiv:2512.07728*, 2025. doi:10.48550/arXiv.2512.07728.

7 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.

8 Anne Driemel, Sariel Har-Peled, and Carola Wenk. Approximating the Fréchet distance for realistic curves in near linear time. In *Proceedings of the twenty-sixth annual symposium on Computational geometry*, pages 365–374, 2010. doi:10.1145/1810959.1811019.

9 Anne Driemel, Amer Krivošija, and Christian Sohler. Clustering time series under the Fréchet distance. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 766–785. SIAM, 2016. doi:10.1137/1.9781611974331.CH55.

10 Joachim Gudmundsson, Zijin Huang, and Sampson Wong. Approximating the λ-low-density value. In *International Computing and Combinatorics Conference*, pages 71–82. Springer, 2023. doi:10.1007/978-3-031-49190-0_5.

11 Hiroshi Imai and Masao Iri. Computational-geometric methods for polygonal approximations of a curve. *Computer Vision, Graphics, and Image Processing*, 36(1):31–41, 1986. doi:10.1016/S0734-189X(86)80027-5.

12 Georgios Kellaris, Nikos Pelekis, and Yannis Theodoridis. Map-matched trajectory compression. *Journal of Systems and Software*, 86(6):1566–1579, 2013. doi:10.1016/J.JSS.2013.01.071.

13 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. doi:10.1145/2666310.2666409.

14 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. doi:10.1109/ICCSE.2016.7581629.

15 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. doi:10.1145/3474373.

16 Anna Lubiw and András Rácz. A lower bound for the integer element distinctness problem. *Information and Computation*, 94(1):83–92, 1991. doi:10.1016/0890-5401(91)90034-Y.

17 Peter Schäfer, Sabine Storandt, and Johannes Zink. Optimal polyline simplification under the local Fréchet distance in (near-) quadratic time, 2023.

18 Mees van de Kerkhof, Irina Kostitsyna, Maarten Löffler, Majid Mirzanezhad, and Carola Wenk. Global curve simplification. In *27th Annual European Symposium on Algorithms (ESA 2019)*, volume 144, 2019. doi:10.4230/LIPIcs.ESA.2019.67.

19 Marc van Kreveld, Maarten Löffler, and Lionov Wiratma. On optimal polyline simplification using the Hausdorff and Fréchet distance. *Journal of Computational Geometry*, 11(1):1–25, 2020. doi:10.20382/JOCG.V11I1A1.

A Appendix

A.1 Simplification Data Structure

For a given polyline P of length n we can construct a data structure that allows to obtain optimal ε simplification in $\mathcal{O}(\log n)$ by precomputing all possible simplifications and performing a binary search[11]. For this, we note that there is a unique sequence of interval $[\varepsilon_n, \varepsilon_{n-1}), [\varepsilon_{n-1}, \varepsilon_{n-2}), \ldots, [\varepsilon_1, \varepsilon_0)$ such that $\varepsilon_n = 0$, $\varepsilon_0 = \infty$ and for any $\varepsilon \in [\varepsilon_i, \varepsilon_{i-1})$ the simplification using ε needs exactly i shortcuts.

[11] To actually achieve logarithmic runtime we can only return a reference to a precomputed simplification as a simplification can have linear size.

We compute the interval boundaries and for each non-empty interval $[\varepsilon_i, \varepsilon_{i-1})$ we compute a simplification with $\varepsilon = \varepsilon_i$ which is a valid simplification for all $\varepsilon \in [\varepsilon_i, \varepsilon_{i-1})$.

To construct the intervals, we construct a set of all possible ε candidates that can be a boundary for an interval and extract the correct ones in a second step. For the candidates, we consider the implicit relations since they are sufficient to implement a simplification algorithm. A simplification size can only change if the result of at least one decision changes. The wait and proceed relation depend on one line segment and two vertices. There are $\mathcal{O}(n^4)$ possibilities to choose such an edge and two vertices (in total four vertices); thus we only need to determine how many candidates ε there are for each combination of four vertices.

Fix a line segment e and two vertices u, v. The candidates for ε are values at which the result of the wait or proceed relation changes. This happens when the relative position of the solutions $\hat{t}_{0/1}(e, u)$ and $\hat{t}_{0/1}(e, v)$ changes. This happens for $t \in [0, 1]$ such that $\varepsilon = \delta(e(t), u) = \delta(e(t), v)$. Intuitively, consider the Voronoi diagram for the vertices u and v. The result of the relations only changes on the boundary between the two cells. Thus, we need the intersections of this boundary with the line segment. In the Euclidean case, this intersection is a unique point, empty or an interval (if the line segment lies exactly on the boundary). Only in the case where it is a unique point does the result of the decision change and thus there are $\mathcal{O}(n^4)$ many candidates in the Euclidean case.

For the Manhattan and Chebyshev case, there may be $\mathcal{O}(d)$ many intersections and thus $\mathcal{O}((dn)^4)$ candidates. Other Minkowski distances are even more complicated. We only consider $d = 2$ and the Euclidean, Manhattan, and Chebyshev case which have $\mathcal{O}(n^4)$ candidates.

After we have found all candidates, we can sort them and perform for each simplification length a binary search to find the respective interval boundary. This takes in total $\mathcal{O}(n^4 \log n)$ to sort. To compare the simplification sizes per pair of candidates, we compute the simplifications in $\mathcal{O}(n^3)$. As each binary search needs $\mathcal{O}(\log n)$ simplification calls, the total runtime is still $\mathcal{O}(n^4 \log n)$.

The space consumption is $\mathcal{O}(n^4)$ due to the candidates. This can be reduced to a cubic space consumption using the following trick: Use a sweep line algorithm and compute the candidates in order. For each line segment, the candidate events correspond to a line intersection-like problem[12] which can be solved similarly to the Bentley-Ottmann algorithm. With this, we can compute the events in order. Compute $\mathcal{O}(n^3)$ candidates, perform binary search, discard the current batch of candidates and proceed to ensure cubic space consumption.

Interestingly, the equation for the euclidean distance simplifies to a linear equation. Thus, no candidate requires square root computations and thus, using an implicit simplification algorithm, the construction of this data structure can be done without square roots. Moreover, the same approach applies to the local setting. Here, only quadratically many candidates are necessary ($\delta^F(P[i' \dots i], \overline{P(i')P(i)})$ for each shortcut $\overline{P(i')P(i)}$).

A.2 Equation Solving Lower Bounds

For the algorithm in Appendix A.1, we need to solve equations of the form $\delta(e(t), u) = \delta(e(t), v)$ for line segment e, and points u, v. With the Euclidean distance, they are trivial to solve and have at most one solution (or one closed interval). Using the Manhattan or

[12] We actually compute intersections of more complicated curves and not just line segments but the main idea is the same.

Chebyshev distance, these equations are more complicated but it is possible to find $\mathcal{O}(d \log d)$ algorithms for both. As an interesting theoretical question, we were able to establish matching lower bounds for both of them using a reduction to integer element distinctness as introduced by Lubiw and Rácz [16].

▶ **Definition 10** (Integer Element Distinctness). *Given integers $x_1, \ldots, x_d \in \mathbb{N}_+$. Decide if there exist indices $1 \leq i < j \leq d$ such that $x_i = x_j$.*

This problem has a known lower bound of $\Omega(d \log d)$ due to Lubiw and Rácz.

For the Chebyshev case, we are given $x_1, \ldots, x_d \in \mathbb{N}_+$. Let C be a large constant that we specify later. Define the linear function $f_a(t) = C + 2a - a^2 t$, which is the tangent of $C + 1/t$ at $t = 1/a$. Note that $f_a(1/a) > f_b(1/a)$ for all $a \neq b \in \mathbb{R}_{>0}$. We construct the equation $\max_{i=1,\ldots,d} |f_{2x_i}(t)| = \max_{i=1,\ldots,d} |f_{2x_i+1}(t)|$.

This means the functions $g(t) = \max_{i=1,\ldots,d} f_{2x_i}(t)$ and $h(t) = \max_{i=1,\ldots,d} f_{2x_i+1}(t)$ consist of exactly d' distinct line segments, where d' is the number of distinct elements in $\{x_i\}$, as each distinct value contributes one segment. From $t = 0$ to $t = 1/2$, the segments are encountered in descending order of x_i. The last segment is encountered at least after $t = 1/2$, since the smallest $n = 2x_i$ is 2, so $1/n \leq 1/2$.

If all linear functions are positive on $[0, 1/2]$, then the absolute values can be removed, and $g(t)$ and $h(t)$ equal the left and right sides of this equation, respectively. For $C = \max_i 2x_i^2$, it can be verified that all terms are positive on $[0, 1/2]$.

Now, consider the equation $g(t) = h(t)$ on $[0, 1/2]$. At $t = 1/(2x_i)$, we have $g(t) > h(t)$, and at $t = 1/(2x_i + 1)$, we have $g(t) < h(t)$ for all i. Let d' be the number of distinct x_i. There are at least $2d' - 1$ solutions because the sign of $g(t) - h(t)$ alternates between these $2d'$ points. However, there cannot be more than $2d' - 1$ solutions because each of the d' segments in g can intersect the piecewise linear function h at most twice (since both are convex, and the first and last segments cannot both intersect h twice). Thus, the number of solutions in $[0, 1/2]$ is exactly $2d' - 1$. For an example refer to Figure 11.

For the Manhattan case, given an instance of the integer element distinctness problem $x_1, \ldots, x_d \in \mathbb{N}_+$, we construct an instance of the Manhattan equation problem in $2d$ dimensions as $1 + \sum_{i=1}^{d} |6x_i + 4 - 2t| = \sum_{i=1}^{d} (|3x_i + 1 - t| + |3x_i + 3 - t|)$

Observe that for $t \notin [3x_i + 1, 3x_i + 3]$, we have $|6x_i + 4 - 2t| = |3x_i + 1 - t| + |3x_i + 3 - t|$. For $t \in (3x_i + 1, 3x_i + 3)$, we have $|6x_i + 4 - 2t| < |3x_i + 1 - t| + |3x_i + 3 - t|$. Furthermore, for any $k \in \mathbb{N}_+$, the equation $1 + k|6x_i + 4 - 2t| = k(|3x_i + 1 - t| + |3x_i + 3 - t|)$ has exactly two solutions in $[3x_i + 1, 3x_i + 3]$.

Let $g(t) = 1 + \sum_{i=1}^{d} |6x_i + 4 - 2t|$ and $h(t) = \sum_{i=1}^{d} (|3x_i + 1 - t| + |3x_i + 3 - t|)$. For $t \in [3x_i, 3x_i + 1]$ for any x_i, we have $g(t) = 1 + h(t)$. For $t \in (3x_i + 1, 3x_i + 3)$, the difference $g(t) - h(t) = 1 + k|6x_i + 4 - 2t| - k(|3x_i + 1 - t| + |3x_i + 3 - t|)$, where k is the multiplicity of x_i, has exactly two solutions in the interval. Thus, the total number of solutions is $2d'$, where d' is the number of distinct values in $\{x_i\}$. Therefore, solving the Manhattan equation problem allows us to determine d', and hence the lower bound applies. For an example refer to Figure 12.

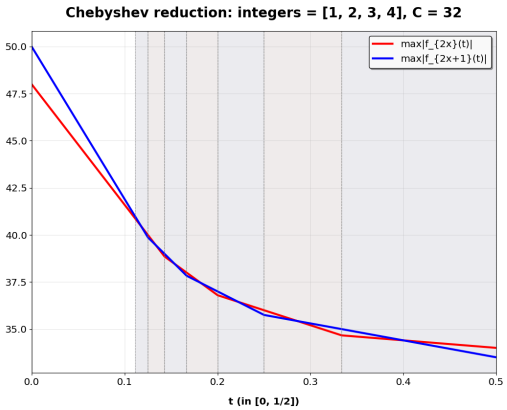


Figure 11 Chebyshev reduction with the integers 1, 2, 3, 4. The functions corresponding to the sides of the equation remain unchanged if any integer appears more than once.

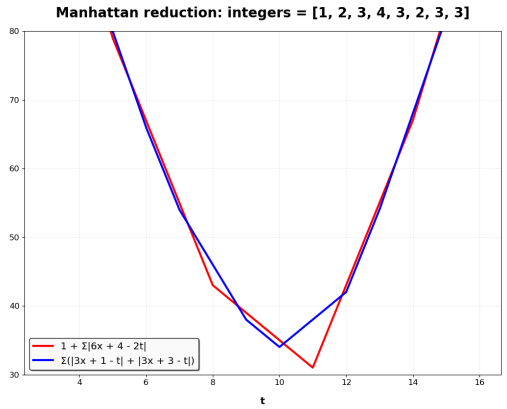


Figure 12 Manhattan reduction for the numbers 1, 2, 3, 4, 3, 2, 3, 3. Duplicates do not change the number of solutions but slightly shift their positions.