Gradual Line Simplification

Nick Krumbholz
Universität Konstanz, Fachbereich Informatik
Universitätsstraße 10, 78464 Konstanz, Germany

January 10, 2023

Abstract

Simplifying polygonal lines is an important task faced in many different applications. There already exist many progressive line simplification algorithms that simplify a polygonal line into different level of details. In this work we will deal with another approach of simplifying, so called gradual line simplification. It asks for simplifications such that we repeatedly remove one vertex of the line in each simplification step, until we are left with one line segment only. For that we propose an algorithm to compute the exact solution in $\mathcal{O}(n^2(f_X(n) + n))$ where n is the number of vertices in the line and f is the runtime function of distance measure X. To speed up we also propose an 4-approximation algorithm that runs in $\mathcal{O}(n(f_X(n) + n))$ and 3 heuristics that run in $\mathcal{O}(n)$. The Hausdorff distance and the Fréchet distance are used as similarity measures between lines in this work.
When using these algorithms on real world gps-traces we find that the exact algorithm is only applicable for small instances and the greedy algorithm is suitable for most bigger instances. When optimizing runtime two of the three heuristics also work pretty well and can be considered for big instances.

1 Introduction

The simplification of polygonal lines asks for a curve that approximates the original curve by using shortcuts with some minimal error criterion. This is used in applications like cartography, network visualization or stock time series where lines are needed to be displayed with some lower level of detail.
When finding a series of simplifications for multiple level of details there are 2 different approaches. The first approach is to simplify the line into different level of details by applying a simplification algorithm on the original line for different errors (denoted as "non-progressive"). In contrast the second approach tries to simplify a line progressively such that every line with less level of detail only contains vertices that are included in lines with larger level of detail (denoted as "progressive"). When applying progressive line simplification every level of detail is predefined by some maximum error and its simplification target is to minimize the number of line segments. For a polygonal line with n line segments some can compute m progressive simplifications in $\mathcal{O}(n^3m)$ with the method shown by Buchin et al. [4]. The progressive approach is anticipated to be better for applications that visualize multiple scales in a series like zooming out on a road map since there no vertices appear from "thin air".
When applying progressive simplification we can not be certain how many line segments are used on each simplification, since it is the optimization goal. This brings us to the approach of simplifying by removing exactly one vertex each simplification step (denoted as "gradually"). When using gradual simplification we are no longer interested in minimizing the number of line segments because we have a guaranteed number of line segments in each simplification. Instead we are interested in minimizing the summed error of each simplification.

There exist algorithms that already use the gradual approach. One such algorithm is the algorithm proposed by Visvalingam and Whyatt [2] that simplifies by repeatedly removing the vertex that has the least area with its neighbouring vertices. But it does not compute the optimal simplification for the summed error for typical distance measures used for measuring similarity of polygonal lines.
This work will be dedicated algorithms computing gradual simplifications while minimizing the summed error of any distance measure. After introducing some definitions and basic algorithms in section 2 we show an algorithm that computes such a series in $\mathcal{O}(n^3)$ for the Hausdorff distance, an approximation algorithm that runs in $\mathcal{O}(n^2)$ for the Hausdorff distance and finally some heuristics that run in linear time in section 3. In section 4 we compare these algorithm in their run time and their solution quality. Finally we discuss about these results and make conclusions in section 5 and also talk about open questions in section 6.

2 Preliminaries

A polygonal line $\mathcal{L} = \langle v_1, \ldots, v_n \rangle$ with $v_i \in \mathbb{R}^2$ is a line defined by a sequence of vertices and the induced straight lines between consecutive points. The subline from i to j is denoted as $\mathcal{L}[i,j] = \langle v_i, \ldots, v_j \rangle$.

2.1 Similarity measures

There are multiple ways to measure the similarity between two polygonal lines. The Hausdorff distance and the Fréchet distance would be 2 applicable distance measures.

2.1.1 Hausdorff distance

The Hausdorff distance between two lines $\mathcal{L}_1, \mathcal{L}_2$ is defined as the biggest shortest distance between those lines, formally defined as follows:

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

where d denotes the euclidean distance. It can be computed rather easily in $\mathcal{O}(nm)$ where $n = |\mathcal{L}_1|$ and $m = |\mathcal{L}_2|$ since there are $\mathcal{O}(nm)$ euclidean distances of which we would have to take the maximum.

2.1.2 Fréchet distance

The Fréchet distance is a little more complex. Imagine a dog is walking on a line and its owner is walking on the other line. The dog is on a leash and both of them are only able to move line forwards. Then the Fréchet distance would be the smallest possible leash length such that they both traverse the lines in parallel from start to end. Formally defined as follows:

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

where α, β are continuous non-decreasing functions with $\alpha(0) = \beta(0) = 0$, $\alpha(1) = |\mathcal{L}_1|$, $\beta(1) = |\mathcal{L}_2|$ and $\mathcal{L}(i+\lambda) = (1-\lambda)\mathcal{L}(i) + \lambda\mathcal{L}(i+1)$ with $i \in \mathbb{N}$ and $\lambda \in [0,1]$. With the method described by Alt and Godau [3] we can compute the Fréchet distance in $\mathcal{O}((n^2m + nm^2)\ log\ nm)$ by getting a list of possible errors, sorting them and then searching the smallest viable error with binary search.
They also describe a way to determine whether some ε holds $\varepsilon \geq d_F(\mathcal{L}_1, \mathcal{L}_2)$ in $\mathcal{O}(nm)$. Using that we can use a simple interval search algorithm to approximate the real value $d_F(\mathcal{L}_1, \mathcal{L}_2)$:

Algorithm 1 $d_{Fapprox}(\mathcal{L}_1, \mathcal{L}_2)$

```
lower ← lowerBound(L1, L2)
upper ← upperBound(L1, L2)
for i = 1 to k do
    mid ← (lower + upper)/2
    if mid ≥ dF(L1, L2) then
        upper ← mid
    else
        lower ← mid
    end if
end for
return upper
```

There are multiple lower and upper bounds that can be computed in $\mathcal{O}(n)$. One lower bound could be the Hausdorff distance or even simpler would be to pick 0. As upper bound one could take the maximum euclidean distance between the first point of $\mathcal{L}_1$ and all points of $\mathcal{L}_2$. Its obvious to see that this algorithm runs in $\mathcal{O}(knm)$. Since the Fréchet distance also considers the order of the points it is often said to be better when measuring similarity between two lines.

2.2 Simplification approaches

Let d_X denote some function to measure the distance between 2 lines. A shortcut s_{ij} with $i < j$ and $i, j \in \mathbb{N}$ symbolizes a direct connection between v_i and v_j. That means a simplification $\mathcal{S}$ of $\mathcal{L} = \langle v_1, \ldots, v_n \rangle$ that uses the shortcut s_{ij} holds that $\mathcal{S} = \langle v_1, \ldots, v_i, v_j, \ldots, v_n \rangle$.
The shortcut error $d_{X;\mathcal{L}}(s_{ij})$ is given by the local distance of the shortcut with $d_{X;\mathcal{L}}(s_{ij}) = d_X(\langle v_i, v_j \rangle, \mathcal{L}[i, j])$.

2.2.1 Minimizing number of segments

In the first approach we try to find a simplification that uses as few vertices as possible while being restricted to some upper bound error. Formally described below:

Given a polygonal line $\mathcal{L}$ and some error $\varepsilon > 0$. Then polyline simplification with minimum number of line segments asks for a polygonal line $\mathcal{S} \sqsubseteq \mathcal{L}$ that holds $d_X(\mathcal{L}, \mathcal{S}) \leq \varepsilon$ while minimizing $|\mathcal{S}|$.

The algorithm presented by Chan and Chin [1] computes a polyline simplification with minimum number of line segments in $\mathcal{O}(n^2)$.
When solving for multiple, on each other depending simplifications Chan and Chins algorithm [1] is not applicable anymore. This is stated as a different problem that needs a different solution:

Progressive Line Simplification Given a polygonal line $\mathcal{L}$ (also denoted as $\mathcal{S}_0$) and a sequence of errors $\mathcal{E} = \langle \varepsilon_1, \ldots, \varepsilon_m \rangle$ with $0 < \varepsilon_1 < \cdots < \varepsilon_m$. Then Progressive Line Simplification asks for a sequence of m simplifications of $\mathcal{L}$ such that $\mathcal{S}_m \sqsubseteq \cdots \sqsubseteq \mathcal{S}_0$ and $d_X(\mathcal{L}, \mathcal{S}_i) \leq \varepsilon_i$ for all $1 \leq i \leq m$ while minimizing the summed number of vertices $\sum_{i=1}^{m} |\mathcal{S}_i|$.

The progressive line simplification problem can be solved with dynamic programming in $\mathcal{O}(n^3 m)$ with the approach of introducing costs shown by Buchin et al. [4].

2.2.2 Minimizing error

Instead of defining some error to find a simplification with according maximum error another approach would be to define some number of points to find a simplification that contains exactly that number of points while minimizing the error. The advantage of using this approach is that we can assure that our simplification holds a target number of segments. Formally the problem is defined as follows.

Given a polygonal Line $\mathcal{L}$ and some target number of vertices $n < |\mathcal{L}|$. Then polyline simplification with minimum error asks for a polygonal line $\mathcal{S} \sqsubset \mathcal{L}$ that holds $|\mathcal{S}| = n$ while minimizing $d_X(\mathcal{L}, \mathcal{S})$.

Finding a simplification with minimum error can be computed in $\mathcal{O}(n^2 \, log \, n)$ for the Hausdorff distance with the method shown by Chan and Chin [1].
But when approaching multiple scales that depend on each other the algorithm of Chan and Chin [1] is again not applicable anymore. We instead define a new problem where we try to remove exactly one vertex on each simplification until one line segment is left:

Definition 1: Gradual Line Simplification
Given some polygonal line $\mathcal{L}$, gradual line simplification asks for $n-2$ simplifications such that $\mathcal{S}_{n-2} \sqsubset \cdots \sqsubset \mathcal{S}_0$ with $|\mathcal{S}_{i-1}| = |\mathcal{S}_i| + 1$ for $1 \leq i \leq n-2$ while minimizing the summed error of all shortcuts $\sum_{s_{ij} \in \gamma} d_{X;\mathcal{L}}(s_{ij})$ where $\gamma = \{s_{ij} | \mathcal{S}_k \textit{ is a simplification of } \mathcal{S}_{k-1} \textit{ that uses } s_{ij}\}$ is a collection of all shortcuts used between each simplification.

A simple way of representing $S_{n-2}, \cdots, S_1$ is to give some ordering of $v_1, \cdots v_n$ that represents the ordered removal of each simplification step.

3 Algorithm & Implementation

In this section we will talk about algorithms computing a gradual simplification.

3.1 Exact Algorithm

When gradually simplifying a line $\mathcal{L}$ we have to find a sequence of shortcuts that we can use while minimizing the summed error.
When using a shortcut s_{ij} between 2 non-consecutive vertices v_i and v_j we know that we had to remove some vertex v_k with $i < k < j$ between. And this would imply that we would also use the shortcuts s_{ik} and s_{kj}. That means the summed error ε_{sum} when using the shortcut s_{ij} and distance X, denoted as $\varepsilon_{sum}(s_{ij}$,X), where v_i and v_j are non-consecutive is given by

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

Since we want to minimize the summed error we have to pick the k that minimizes the error.
If points v_i and v_j are consecutive ($i + 1 = j$) we would have no error since 2 consecutive vertices already have a straight line connection and their shortcut would be the same line.

Taking this together the optimal ε_{sum} with minimum error when using shortcut s_{ij} and distance X is then given by:

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

When using gradual simplification the last shortcut used is guaranteed to be $s_{1;n}$ since the last simplification $\mathcal{S}_{n-2}$ only consists of the first and last vertex. That means the optimal summed error of a line $\mathcal{L}$ is given by $\varepsilon_{opt}(s_{1;n})$. There are multiple ways to solve that equation. We propose the usage of an $n \times n$ matrix S where $S(i,j) = \varepsilon_{opt}(s_{ij})$ and another $n \times n$ matrix K where $K(i,j)$ denotes the k used for computing $\varepsilon_{opt}(s_{ij})$ (S and K are symmetric matrices). Algorithm 2 computes these matrices by computing those with a smaller hop distance $|j-i|$ first. When the K matrix is filled we are able to find the ordered removal by backtracking from $K(1,n)$ just as described in Algorithm 3.

Algorithm 2 *optimal*$(\mathcal{L}, X)$

```
n ← |L|
S ← n × n matrix with initial values 0
K ← n × n matrix
for hop = 2 to n − 1 do
    for i = 1 to n − hop do
        j ← i + hop
        S(i, j) ← ∞
        sDist ← d_{X;L}(s_ij)
        for k = i + 1 to j − 1 do
            kDist ← sDist + S(i, k) + S(k, j)
            if kDist < S(i, j) then
                S(i, j) ← kDist
                K(i, j) ← k
            end if
        end for
    end for
end for
ε_opt ← S(1, n)
removal ← backtracking(K)
```

Obviously Algorithm 2 uses $\mathcal{O}(n^2)$ space. Since for every index of S we have to compute the shortcut error and find the minimal k it runs in $\mathcal{O}(n^2(n+f_X(n)))$ where f_X is the runtime function of the distance measure X.

Algorithm 3 *backtracking*(K)

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

3.2 Approximation Algorithm

In many applications we need low computation time and are not interested in finding the optimal solution. Thus we would want an algorithm that has some lower asymptotic runtime bounds.
Another approach of finding a removal sequence could be by using a greedy algorithm that repeatedly removes the vertex that produces the smallest error. This can be done efficiently by keeping all removal errors with their according vertex in a min-heap (first and last node having an infinite error). Then after extracting the minimum both neighbouring vertices must be updated since their error changed.

For the initialization of the heap $\mathcal{O}(n)$ removal errors must be computed, which takes $\mathcal{O}(n \cdot f_X(n))$ time where f_X is the runtime function of the distance measure X. Building the heap can then be done in linear time afterwards. The algorithm extracts $\mathcal{O}(n)$ times where each iteration updating neighbouring errors can be done in $\mathcal{O}(f_X(n))$ and the heap can be re-heapified in $\mathcal{O}(log\ n)$ since only 3 values have to be sink accordingly in the heap. This results in a total runtime of $\mathcal{O}(n(f_X(n) + log\ n))$ for the Algorithm 4.

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

$\varepsilon_{greedy} \leftarrow 0$
$n \leftarrow |\mathcal{L}|$
$removal \leftarrow \mathbb{N}^{n-2}$
$heap \leftarrow$ MinHeap containing removal errors
for $i = 1$ **to** $n - 2$ **do**
  $minError \leftarrow heap.extractMin()$
  $minVertex \leftarrow$ vertex belonging to $minError$
  update neighbour errors
  re-heapify updated errors
  $\varepsilon_{greedy} \leftarrow \varepsilon_{greedy} + minError$
  $removal[i] \leftarrow minVertex$
end for

It can be shown that the solution given by the greedy algorithm has an upper bound of 4 to the optimal solution, this means it is a 4-approximation algorithm.

3.3 Heuristics

We also introduce 3 heuristic algorithms that compute a solution in linear time.

3.3.1 In-Order Simplification

In this simplification method we repeatedly remove the first removable vertex. This would give us the ordered removal sequence $\gamma = \langle v_2, \ldots, v_{n-1} \rangle$. This sequence can be obviously build in linear time.

3.3.2 Random Simplification

Here we repeatedly remove a random removable vertex. This would give us a removal sequence that would be equal to a random permutation of $\langle v_2, \ldots, v_{n-1} \rangle$. This can be done in linear time with the Fisher-Yates shuffle that ensures equal probability for each permutation as seen in Algorithm 5.

Algorithm 5 *fisher-yates-shuffle*$(array)$

$n \leftarrow |array|$
for $i = n$ **to** 2 **do**
  $j \leftarrow$ random integer with $1 \leq j \leq i$
  swap $array[i]$ and $array[j]$
end for

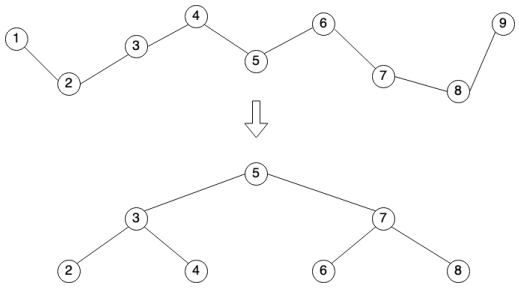

Figure 1: Tree projection of a line

3.3.3 Equal Simplification

The last heuristic tries to equally distribute removals. Then we would want to remove the middle vertex at last. When repeating this to the left and the right of the middle vertex this can be represented as a tree just as shown in figure 1. Since we would want to remove a node after the removal of its children in the tree we could describe the procedure with the following algorithm:

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

Building this tree is not necessary and instead can be adapted to the following iterative variant:

Algorithm 6 *equal*$(\mathcal{L})$

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

Since there are $\mathcal{O}(n)$ iterations, the total runtime of Algorithm 6 is given by $\mathcal{O}(n)$.

3.4 Adapting similarity measures

The runtime of the exact algorithm and the approximation algorithm both depend on the runtime of the distance measure used. That means the quicker we are able to compute a distance, the quicker the algorithms will work. Since in gradual simplification we only remove one vertex per simplification step we only have to compute the shortcut error of the shortcut used.

The Hausdorff distance of a shortcut s_{ij} can easily be computed in $\mathcal{O}(i-j)$, since we have to compute the shortest euclidean distance to the segment $\langle v_i, v_j \rangle$ for every shortcutted vertex between i and j and then take the maximum one.
The Fréchet distance can easily be adapted for shortcuts. When computing the Fréchet distance with the algorithm proposed by Alt and Godau [3], we compute critical values ε that potentially could give us a valid path in the 2-dimensional cell-based free-space-diagram they described. Each of the ε are critical because they open a new path between cells. In the case of solving this for a shortcut error of shortcut s_{ij} the free space diagram would have the cell dimension $1 \times |i-j|$ since the shortcut itself only consist of a single line segment. Thus it can be reduced to a single dimension of cells and all of our paths need to be open. That means instead of performing a binary search on the critical values, we have to calculate the maximum of all critical values. Since there are $\mathcal{O}((j-i)^2)$ critical values for a shortcut s_{ij}, the shortcut Fréchet distance $d_{F;\mathcal{L}}(s_{ij})$ can be calculated in $\mathcal{O}((j-i)^2)$.

Nowadays there are even better upper bounds for calculating the shortcut error with the Fréchet distance. Buchin et al. [5] show that it is even possible to calculate the Fréchet shortcut distance in sublinear time. But in the following we will stick with the adapted version of Alt and Godau [3], just as described. Thus the runtime of our algorithms are as follows:

	Hausdorff	Fréchet Approx	Fréchet
Exact Algorithm	$\mathcal{O}(n^3)$	$\mathcal{O}(kn^3)$	$\mathcal{O}(n^4)$
Approximation Algorithm	$\mathcal{O}(n^2)$	$\mathcal{O}(kn^2)$	$\mathcal{O}(n^3)$
Heuristics	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Table 1: Overview of proposed algorithms and their runtime with the corresponding error measure used.

3.5 Implementational Details

We implemented all algorithms in Java and no external libraries were used.
We represented the polygonal line as an array of vertices where each vertex consists of an x and y double value. This is especially useful for calculating a shortcut error since it gives us constant access time for any given vertex. For the approximation algorithm we additionally added pointer to predecessors and successors for each vertex, as well as its heap index, such that we can access its neighbors in constant time in the heap.
Also we halved the space consumption of all symmetric matrices by projecting one halve of it on a one dimensional array.
Finally since many algorithms require tuples we also implemented a Tuple class that consists of a left and a right element.

4 Experimental Evaluation

In this section, we will show our experiments and their results.

4.1 Data and Hardware

We conducted all experiments on an AMD Ryzen Threadripper 3970X 32-Core processor (clocked max 3700 MHz) with 128 GB main memory. All test instances were randomly selected GPS-traces picked from `https://www.openstreetmap.org/traces`.

4.2 Results

First of all we were interested to find an optimal k for algorithm 1 approximating the Fréchet distance, such that we can expect a low error (prefered under 1%)

while using a low number of iterations. We evaluated the solution of 50 randomly selected gps-traces with lengths between 6 and 3000 computed by the exact algorithm and compared the approximated Fréchet distance depending on the k to the exact Fréchet distance as seen in figure 2.

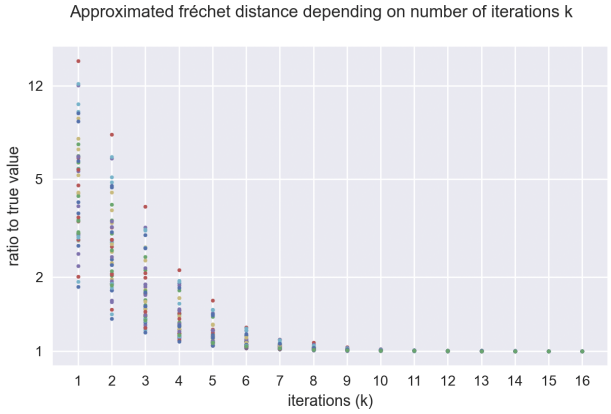

Figure 2: This figure shows the ratio to the true value (computed by dividing the approximated distance by the exact distance) of the Fréchet approximation (y-axis) in correlation with the number of iterations used (x-axis). The y-axis is logarithmic scaled.

iterations k	average overestimation
1	416.8251 %
2	186.2463 %
3	82.1739%
4	36.1940%
5	17.6909%
6	8.8057 %
7	4.4621 %
8	2.3498 %
9	1.2143 %
10	0.6239 %
11	0.3068 %
12	0.1574 %
13	0.0732 %
14	0.0316 %
15	0.0102 %
16	0.0054 %

Table 2: This table shows the accuracy depending on the number of iterations

All of the approximated Fréchet distances are greater than 1 for all k. The average overestimation per k can be seen in table 2.
In the rest of the paper we will settle with $k = 10$ since it is the lowest number of iterations where the overestimation is lower than 1%.

Next we were interested in the quality solution of the algorithms. Thus we compared them using the Hausdorff distance in figure 3 and Fréchet distance in figure 4 for 100 lines with lengths upon 10000. There you can generally see that in most cases it holds In-Order>Random>Equal>Greedy>Exact for the respective solution size of the algorithms.
This can be approved when looking at table 3, that illustrates how much worse each algorithm performed in comparison to the exact algorithm. There you can see that the In-Order simplification performs the worst with solutions up to approximately 696 times worse than the exact solution for the Hausdorff distance. The Random heuristic performs worse than the Equal heuristic in average, but is still a lot better than the In-Order heuristic. That means that the Equal heuristic performs the best out of the heuristics. But the greedy algorithm performs a lot better than all heuristics, being only 4.02% worse for the Fréchet distance and 4.58% worse for the Hausdorff distance compared to the exact solution in average. The worst solution of the greedy algorithm was only 11.77% worse than the exact solution for the Hausdorff distance and even only 9.42% worse for the Fréchet distance. Thus it held its approximation bounds.

Quality comparison summarization

Simplification Algorithm	Hausdorff distance mean	Hausdorff distance max	Fréchet distance mean	Fréchet distance max
In-Order	20242%	69592%	10082%	26438%
Random	96.53%	153.3%	101%	166.5%
Equal	43.35%	92.51%	41.16%	79.08%
Greedy	4.58%	11.77%	4.02%	9.42%

Table 3: This table summarizes the findings of figure 3 and figure 4 by comparing the solution size of each algorithm and the exact solution for both distance measures. Each cell denotes how much worse each algorithm performed compared to the exact solution. Only lines where the exact algorithm was able to compute a solution in a time frame of 15 minutes were considered in this table.

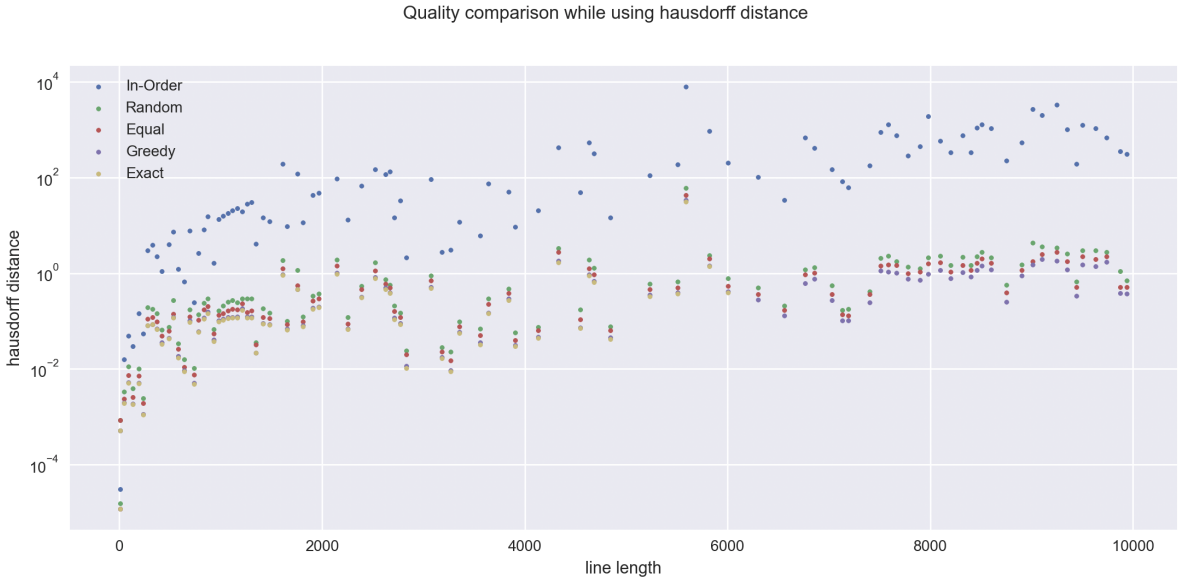

Figure 3: This figure shows the Hausdorff distance (y-axis) of the solutions computed by the algorithms for some lines with different lengths (x-axis). Note that the y-axis is logarithmic scaled and the exact solution was only computed on instances up to 6000 due to a time limit of 15 minutes.

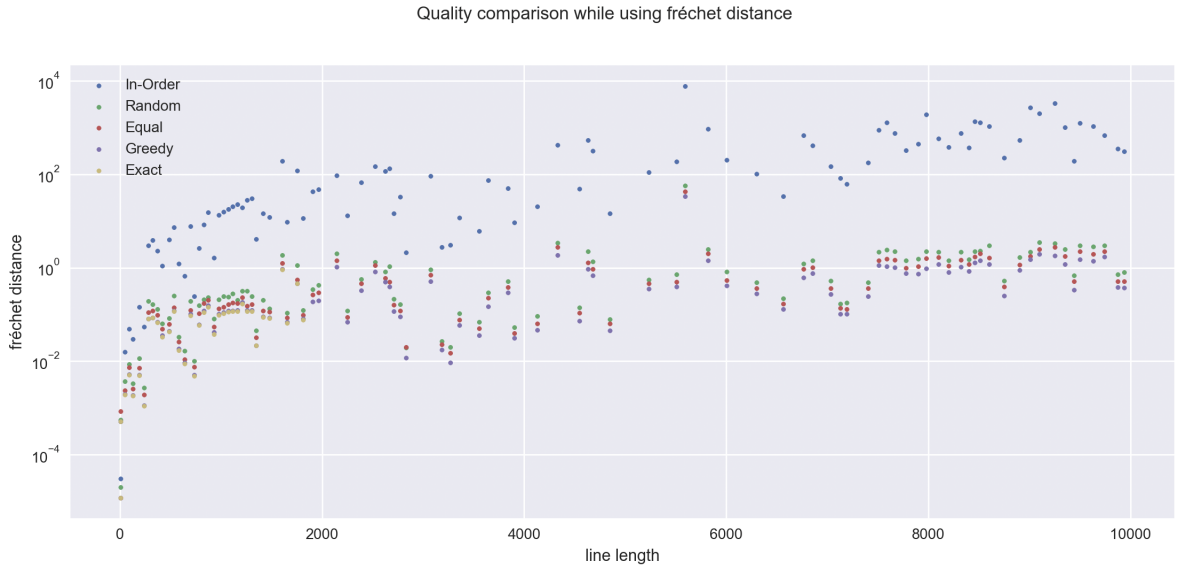

Figure 4: This figure shows the Fréchet distance (y-axis) of the solutions computed by the algorithms for some lines with different lengths (x-axis). Note that the y-axis is logarithmic scaled and the exact solution was only computed on instances up to 2800 due to a time limit of 15 minutes.

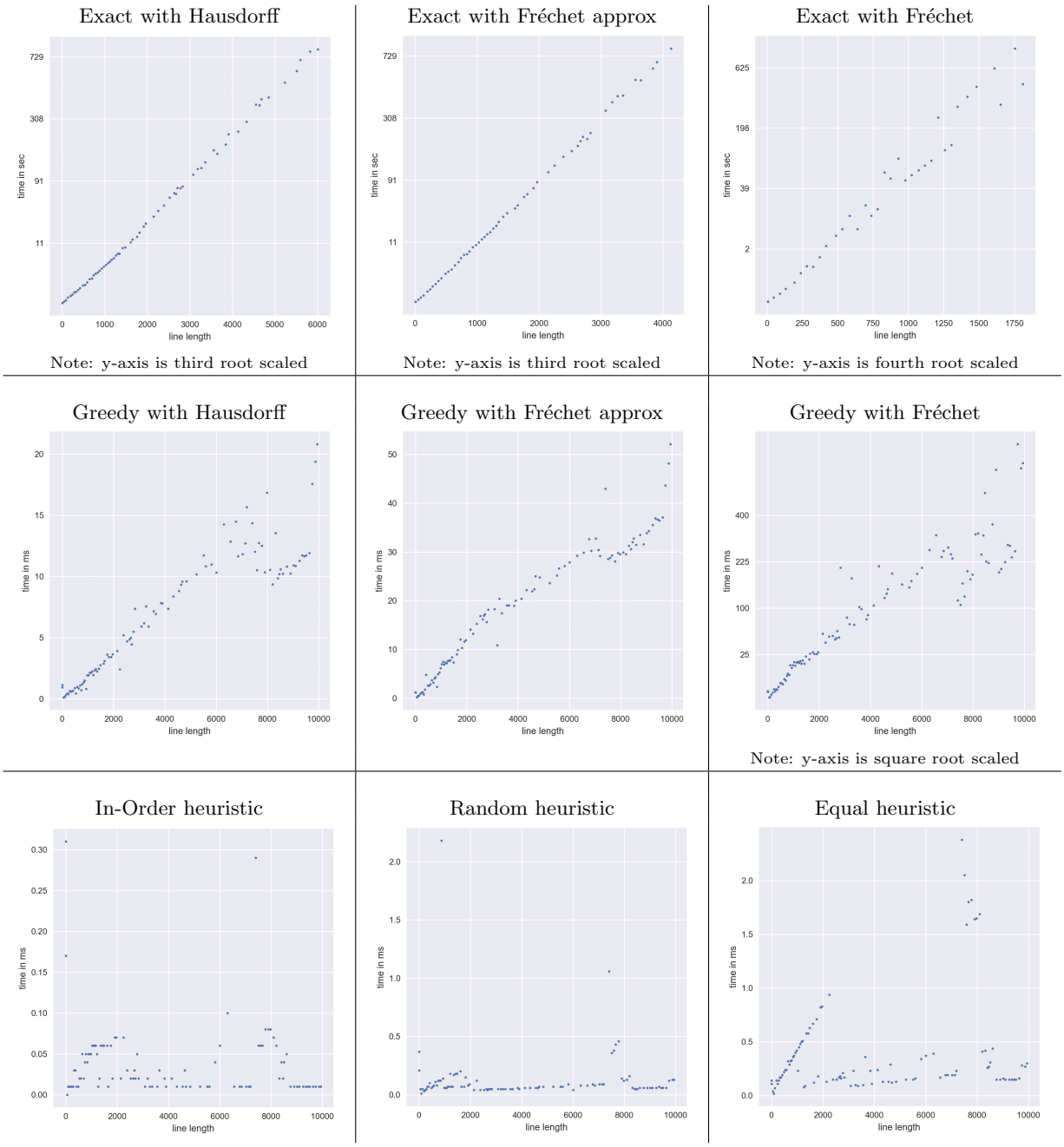

Table 4: This table shows the runtime of all algorithms used in correlation with the length of a line up to instances of length 10000. Some notes under figures indicate if a non-linear scaling was used for an axis.

Now we looked at the runtime of the algorithms. In the table 4 you can see all algorithm runtimes with different distance measures used on lines up to a length of 10000.
There you can see a definite correlation between the exact algorithm and the length of the line as expected. Since it uses a third root scale on the time axis when using the Hausdorff distance and the approximated Fréchet distance, both of their runtime really seems to be in $\mathcal{O}(n^3)$. This also seems to apply when using the Fréchet distance, there we used a fourth root scale on the y-axis and thus seems to hold its runtime that is in $\mathcal{O}(n^4)$.
On the other hand the greedy algorithm seems to underlie its runtime bounds. When using the Hausdorff distance or the approximated Fréchet distance it seems to scale even linear. Also when using the exact Fréchet distance it seems to scale in quadratic time instead of the theoretical cubic runtime since the y-axis is square root scaled. This may be caused by the input size being too small.
The scaling of the In-Order heuristic was not expected. It even seems to be constant time, probably due to the input size being too small. The Random and Equal heuristic on the other hand seem to scale very linear with time as expected. There are multiple suitable regression lines for the Equal heuristic, probably due to different parallel usage of the system.

Also we compared the runtime of the algorithms in figure 5 with the Hausdorff distance, in figure 6 with the approximated Fréchet distance and in figure 7 with the Fréchet distance which you can see below. There you can see a clear trend of In-Order<Random<Equal<Greedy<Exact in terms of runtime for most problem instances for every distance measure.
Also for each computation we have set a time limit of 15 minutes. Thus you can already see in these figures the maximum problem instances for the exact algorithm that can be computed in this time limit. For the Hausdorff instance it is able to compute solutions for instances up to 6000, for the approximated Fréchet distance up to instances of size 4200 and finally for the Fréchet distance it is able to compute for instances up to 1800 in the time frame of 15 minutes.
For the greedy algorithm we did a separate test to find its maximum instances, which can be seen in figure 8. There you can see that the greedy algorithm was able to compute solutions for instances with length up to 45 million for the Hausdorff distance, up to 31 million for the approximated Fréchet and up to 310 thousand for the Fréchet distance in the time frame of 15 minutes. The greedy algorithm is much more suitable for bigger problem instances. Since the heuristics all work in linear time they should be able to run in arbitrary instance sizes and would probably be rather limited by memory capacity.

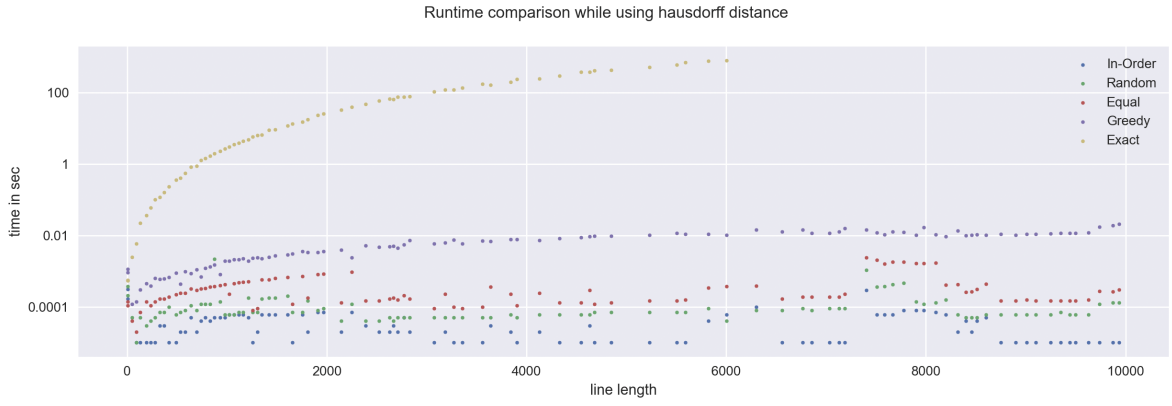

Figure 5: This figure shows the runtime of all algorithms while using the Hausdorff distance depending on the length of the line. Notice that the y-axis is log-scaled.

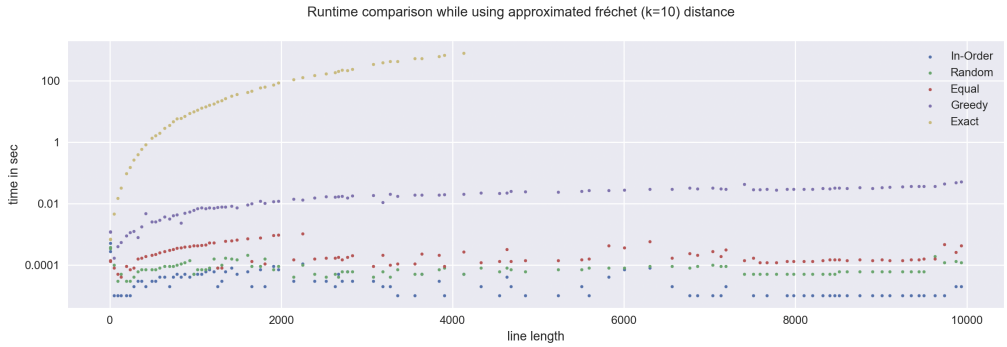

Figure 6: This figure shows the runtime of all algorithms while using the approximated Fréchet distance depending on the length of the line. Notice that the y-axis is log-scaled.

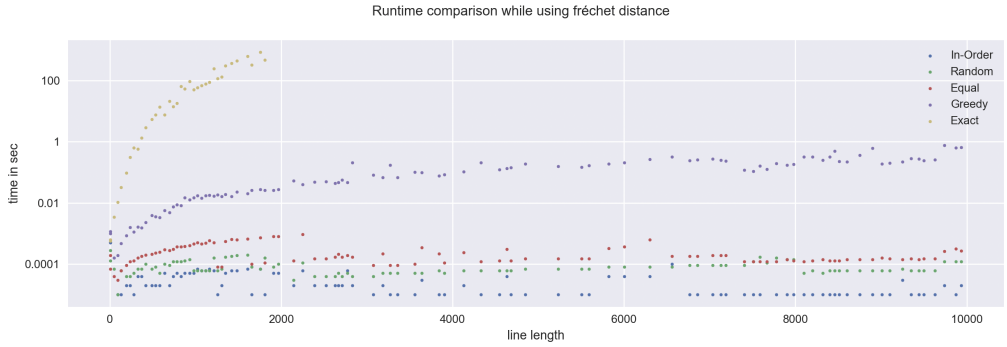

Figure 7: This figure shows the runtime of all algorithms while using the Fréchet distance depending on the length of the line. Notice that the y-axis is log-scaled.

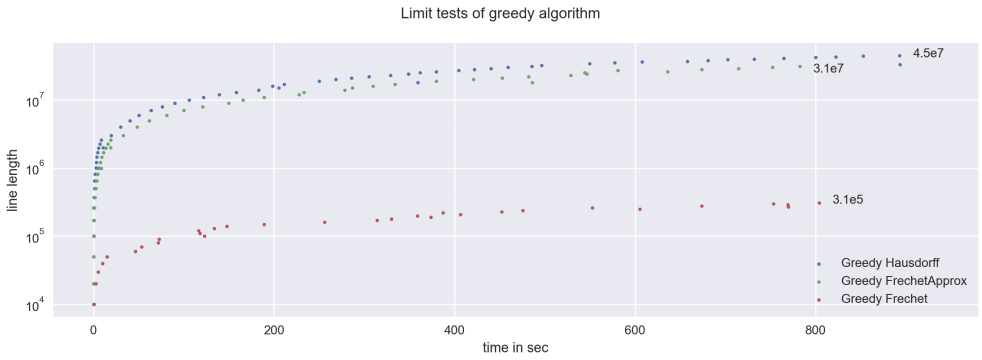

Figure 8: This figure shows the correlation of the greedy algorithm using all distance measures. It was applied on increasing problem instances until the limit of 15 minutes was reached. Notice that the y-axis is log-scaled.

Finally we wanted to compare the runtime of the distance measures by applying the same algorithm for different distance measures and compare their runtime. This can be seen in figure 9 for the greedy algorithm and in figure 10 for the exact algorithm.
When using the greedy algorithm you can definitely see how close the runtime of the approximated Fréchet distance and Hausdorff distance are due to their linear runtime. In average the greedy algorithm using the approximated Fréchet distance took 2.95 times longer than the greedy algorithm using the Hausdorff distance. The quadratic runtime of the Fréchet distance is very visible due to it increasing much faster.

When using the exact algorithm you also can see the same asymptotic growth between the approximated Fréchet distance and the Hausdorff distance. In this case the approximated Fréchet took 3.24 times longer than the Hausdorff distance in average. So in general we could say that using the approximated Fréchet distance with $k = 10$ takes around 3 times longer than the Hausdorff distance in our implementation. The exact Fréchet distance definitely takes longer and has a greater asymptotic growth than the other distance measures.

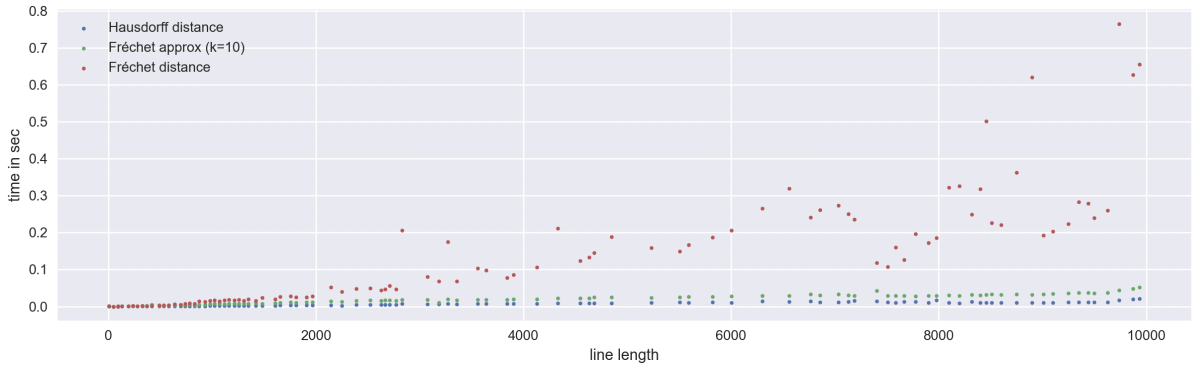

Figure 9: In this figure you can see the runtime of the greedy algorithm while using different distance measures.

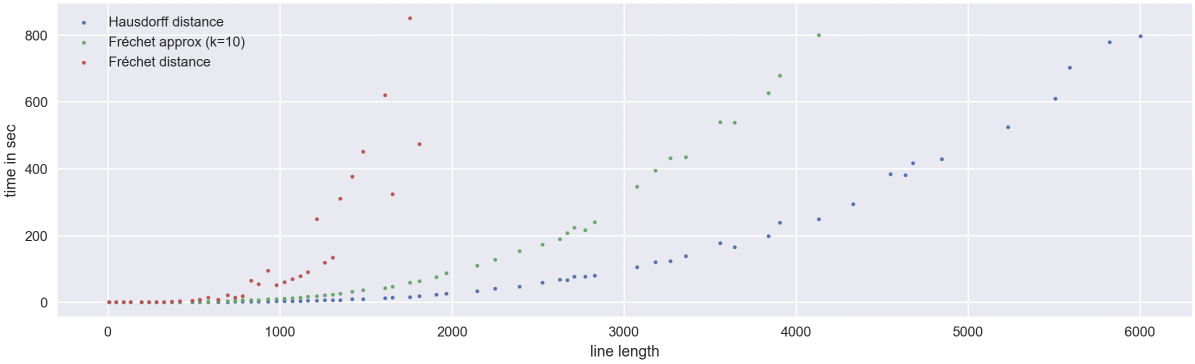

Figure 10: In this figure you can see the runtime of the exact algorithm while using different distance measures.

5 Discussion and Conclusion

In this work we introduced another approach of simplifying polygonal lines and proposed one exact algorithm solving the optimization target, one greedy 4-approximation algorithm and three heuristics. We also used the Hausdorff distance, Fréchet distance as well as an approximation of the Fréchet distance to measure the quality of our simplifications and used them to analyze and compare the runtime and quality of the algorithms.
We showed that the exact algorithm is able to compute solutions on smaller instances, but is very slow towards long polygonal lines. Its computation time also varies very hard on the distance measure and its implementation.
For most larger instances the greedy algorithm works very well and computes solutions rather fast while its solution quality is mostly almost as good as the one provided by the exact algorithm. Its runtime also varies hard depending on the distance measure.
But when it comes to minimizing runtime of big instances one of the heuristics should be considered. The in-order heuristic was the fastest but had some real bad solutions. The random heuristic was slower but computed solutions that were much better. Also its worst case was way better than the worst case of the in-order heuristic. The equal heuristic had the best solutions, but was slower than all other heuristics. When in use of one heuristic we would not recommend using the in-order heuristic, since its solution quality was very bad compared to the other heuristics.

6 Open Questions

Nowadays there are quicker algorithms to compute the Fréchet distance of a shortcut than the implementation we used. One open question is how good the exact and greedy algorithm would perform when using the sublinear algorithm proposed by Buchin et al. [5] to compute the Fréchet distance.
Also a comparison between the proposed gradual line simplification algorithms and existing progressive line simplification algorithms or existing non-progressive line simplification algorithms in terms of runtime would be very interesting.
We only considered the error of the shortcut in each simplification step once. How would the algorithms change if we would also consider earlier removed vertices as well in each simplification step (sum-sum problem)?

References

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

[2] M. Visvalingam **and** J. D. Whyatt. "Line generalisation by repeated elimination of points". **in***The Cartographic Journal*: 30.1 (**june** 1993), **pages** 46–51. URL: https://doi.org/10.1179/caj.1993.30.1.46.

[3] Helmut Alt **and** Michael Godau. "Computing the Frechet Distance between two Polygonal Curves". **in***International Journal of Computational Geometry Applications*: 05.01n02 (**march** 1995), **pages** 75–91. URL: https://doi.org/10.1142/s0218195995000064.

[4] Kevin Buchin, Maximilian Konzack **and** Wim Reddingius. *Progressive Simplification of Polygonal Curves*. 2018. URL: https://arxiv.org/abs/1806.02647.

[5] Maike Buchin **and others**. *Efficient Fréchet distance queries for segments*. 2022. URL: https://arxiv.org/abs/2203.01794.