Compressing spatio-temporal trajectories

Joachim Gudmundsson[1], Jyrki Katajainen[2,⋆], Damian Merrick[1,3], Cahya Ong[4], and Thomas Wolle[1]

[1] NICTA⋆⋆, Sydney, Australia
{joachim.gudmundsson, thomas.wolle, damian.merrick}@nicta.com.au
[2] Department of Computing, University of Copenhagen, Denmark
jyrki@diku.dk
[3] School of Information Technologies, University of Sydney, Australia
[4] Department of Engineering, University of New South Wales, Australia

Abstract. Trajectory data is becoming increasingly available and the size of the trajectories is getting larger. In this paper we study the problem of compressing spatio-temporal trajectories such that the most common queries can still be answered approximately after the compression step has taken place. In the process we develop an $O(n \log^k n)$-time implementation of the Douglas-Peucker algorithm in the case when the polygonal path of n vertices given as input is allowed to self-intersect.

1 Introduction

Technological advances in location-aware devices, surveillance systems, and electronic transaction networks are producing more and more opportunities to trace moving individuals. Consequently, an eclectic set of disciplines including geography [7], database research [9], animal-behaviour research [12], and transport analysis [14] shows an increasing interest in movement patterns of various entities moving in various spaces over various times scales (see also the survey by Gudmundsson et al. [8]).

Large sets of data on the movement of entities create the problem of storing, transmitting, and processing this data. Hence, simplifying this data becomes an important problem. Recently, Cao et al. [4] proposed a way of modelling trajectories in 3-dimensional space so that a 3-dimensional path simplification techniques could be applied. Their idea works well in practice and in their experiments the compression rate is in most cases well over 90%. However, their approach has two main drawbacks that we improve on in this paper.

1. They argued that most spatio-temporal queries in databases are composed of the following five types of queries: *where-at*, *when-at*, *intersect*, *nearest-*

⋆ Part of this research was conducted when the author visited Sydney Research Laboratory at NICTA. The research of this author was partially supported by the Danish Natural Science Research Council under contract 272-05-0272 (project "Generic programming—algorithms and tools").

⋆⋆ NICTA is funded through the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

neighbour and *spatial-join*. However, in their paper they were only able to prove that their approach is "sound" (to be defined) for three of the five query types. In this paper we show that by making a small modification to their model one can prove that all the queries can be approximated.

2. They used the Douglas-Peucker path simplification algorithm which in 3-dimensional space has a running time of $O(n^2)$. In our specific case we show that it can be approximated in $O(n \log^k n)$ time, where k depends on the model.

Simplifying polygonal paths is a well-researched area in cartography, geographic information systems, digital image analysis, and computational geometry. However, trajectories differ from polygonal paths, because trajectories do not only contain information about a sequence of locations, but also *when* an entity has been at these locations. Therefore, simplifying trajectories differs from simplifying polygonal paths, as we might wish to preserve some temporal information. The movement of a point object p is described by a sequence of coordinates given at n time steps $\langle (x_1, y_1, t_1), \ldots, (x_n, y_n, t_n) \rangle$. The aim is to simplify the trajectory such that both spatial and temporal information is maintained.

In this paper we propose an approach that enables us to use 3-dimensional path simplification algorithms that compute a simplified path containing a subset of the vertices of the original path. The computational problem of path simplification is to compute an *optimal* or *minimum ε-simplification*, i.e. an ε-simplification with as few vertices as possible. In applications, this can considerably reduce storage space and processing time.

Imai and Iri [13] formulated the path simplification problem graph theoretically: construct a directed acyclic graph that models all possible edges in a simplification and compute a shortest path in the graph. Their algorithm runs in $O(n^2 \log n)$ time. Chan and Chin [5], and Melkman and O'Rourke [15] improve this running time to quadratic. Most of the known algorithms use $O(n^2)$ time and space. An exception is the algorithm by Agarwal and Varadarajan [1] that achieves $O(n^{4/3+\delta})$ time and space, where $\delta > 0$ is an arbitrarily small constant. However, their algorithm only works for the L_1 metric.

Since the problem of developing a near-linear time algorithm for computing the optimal ε-simplification remains unsolved, several heuristics have been proposed. The most widely used heuristic is the Douglas-Peucker method [6] (and its variants), originally proposed for simplifying curves under the Hausdorff error measure. For a real number $\varepsilon > 0$, the polygonal path $\langle v_1, \ldots, v_n \rangle$ is approximated as follows. If every vertex v_i, for $1 < i < n$, has a distance at most ε to the line ℓ determined by v_1 and v_n, accept the line segment (v_1, v_n) as an approximation for the whole path. Otherwise, split the path at a vertex further than ε from line ℓ and recursively approximate the two pieces. A straightforward implementation requires $O(n)$ time to find the point furthest from line ℓ. Since the recursion depth can be linear, the running time is bounded by $O(n^2)$.

In this paper we show how the algorithm can be implemented more efficiently if we allow the distance computation to be approximate. That is, assume that we are given $\varepsilon > 0$ and that a segment ℓ is about to be tested. Let p be the point furthest from ℓ, and let d be the smallest distance between p and ℓ. We say that

we have an α-approximation, for $\alpha > 1$, if and only if ℓ is accepted if $d \leq \varepsilon$ and discarded if $d > \alpha \cdot \varepsilon$. Note that this implies that ℓ can be either accepted or discarded if $\varepsilon < d \leq \alpha \cdot \varepsilon$.

A crucial aspect of simplification algorithms is how the distance between a point and a line segment is measured. Originally in the Douglas-Peucker algorithm, the Euclidean distance between a point and a line is used (*line model*), where the line is defined by the corresponding line segment. This can lead to counter-intuitive simplifications. That is why we also use the Euclidean distance from a point to a line segment (*line-segment model*). Even though the Douglas-Peucker algorithm does not output the minimum number of vertices and its worst-case running time is $O(n^2)$, it is often used due to its simplicity and efficiency in practice. However, in the case where the path is assumed to be non-self-intersecting, or even monotone, faster methods have been developed. Hershberger and Snoeyink [10] showed that in the line model the running time can be improved in the case where the path does not self-intersect by making use of the fact that the furthest point has to be a vertex of the convex hull of the point set. Allowing $O(n \log n)$ preprocessing they showed how the furthest point can be found in $O(\log n)$ time. This was later improved further to $O(n \log^* n)$ in [11] by the same authors. We will use a similar approach with two crucial differences: the input path may self-intersect, and we consider both the line and the line-segment models. The contribution of this paper is threefold:

1. We consider the problem of simplifying trajectories and modify the model by Cao et al. [4], such that the five types of queries proposed in [4] can be approximated in a sound way (Section 2). As a result it follows that the 3-dimensional path simplifications can be used to compress trajectories.
2. We propose an algorithm that produces an approximate Douglas-Peucker simplification of a trajectory, i.e. a z-monotone path in 3-dimensional space (Section 4). That is, given two real values $\varepsilon > 0$ and $\delta > 0$, the output is a $(1 + \delta)\varepsilon$-simplification. The running time of our algorithm is $O(\frac{1}{\delta^2} n \log^2 n)$ in the line model and $O(\frac{1}{\delta^2} n \log^3 n)$ in the line-segment model. Previously no sub-quadratic time (approximation) algorithm was known.
3. In the process we present an $O(n \log^2 n)$-time (line model) and an $O(n \log^3 n)$-time (line-segment model) implementation of the Douglas-Peucker algorithm in the plane in the case where the polygonal path can self-intersect (Sec. 3).

Due to space constraints, all proofs and figures have been omitted and can be found in the full version.

2 Modelling trajectories

In this section, we introduce our model for trajectories, which generalises the results in [4]. We give some preliminary definitions e.g. of spatio-temporal queries and soundness of distance functions. Then we describe our model and prove its effectiveness regarding soundness.

2.1 Preliminaries

According to Cao et al. [4] most spatio-temporal queries are composed of the following five types of queries: *where-at*, *when-at*, *intersect*, *nearest-neighbour* and *spatial-join*. We state the semantics of the two most basic queries *where-at* and *when-at* on a trajectory $T = \langle (x_1, y_1, t_1), \ldots, (x_n, y_n, t_n) \rangle$ as follows.

- *where-at*(T, t) returns the location of the entity corresponding to T at time t according to T. If $t < t_1$ or $t > t_n$, then the answer is undefined.
- *when-at*(T, x, y) returns the time t at which a moving object on trajectory T is expected to be at location (x, y). If the location is not on the trajectory, or the moving object visits the location more than once, or is stationary at the location, then the answer is undefined.[5]

Also the notion of soundness of distance functions is discussed in [4]. For a trajectory T, let $q(T)$ denote the answer of some spatio-temporal query q with input T. To make the dependence on both ε and the underlying distance function *dist* explicit, we let a $(dist, \varepsilon)$*-simplification* denote a simplification that is computed using *dist*.

Definition 1. *Let T be a trajectory and T' its $(dist, \varepsilon)$-simplification. The distance function dist is sound for query q, if for each ε there exists a bound δ, such that $|q(T, .) - q(T', .)| \leq \delta$. For the where-at query $|q(T, t) - q(T', t)|$ is the Euclidean distance between the two points given as answers, and for the when-at query $|q(T, x, y) - q(T', x, y)|$ is the difference between the two returned times.*

Cao et al. [4] define distance functions between a point p_m and a line segment $\overline{p_i p_j}$ in 3-dimensional space: E_2 (2-dimensional Euclidean distance), E_3 (3-dimensional Euclidean distance), E_u ($E_u(p_m, \overline{p_i p_j}) = \sqrt{(x_m - x_c)^2 + (y_m - y_c)^2}$ where p_c is the point on $\overline{p_i p_j}$ with $t_m = t_c$) and E_t ($E_t(p_m, \overline{p_i p_j}) = |t_m - t_c|$ where p_c is the point on the 2-dimensional projection of $\overline{p_i p_j}$ onto the xy-plane that is closest to the 2-dimensional projection of p_m onto the xy-plane). They also show that only the distance function E_u is sound for the *where-at* query, and only the distance function E_t is sound for the *when-at* query. Hence, they propose to use a combined distance function based on E_u and E_t which is sound for both queries. This approach combines the strength of both single distances, but also their weaknesses. This combined distance function results in the worst compression ratio among the researched distance functions.

We argue that using E_u gives rise to another problem. Consider a trajectory where an entity moves with high speed along the x-axis (i.e. $y = 0$) and changes slightly its speed. (The effect can be amplified by repeating this pattern.) From a practical point of view we might wish to simplify this trajectory to a line segment, as we are not interested in preserving the marginal speed changes of an entity (e.g. a car) on a long line segment (e.g. a motorway). However, with the E_u distance we are unable to do so.

[5] This definition is taken from [3]. The definition in [4] is similar but considers the stationary case as a special case.

2.2 Our model

As in [4], we think of a trajectory as a polygonal path in 3-dimensional space. The x- and y-dimensions correspond to the two spatial dimensions in which the entities move. The third dimension is the time t, which enables us to preserve temporal information. If we want to apply a path-simplification algorithm on such a 3-dimensional path, we need a distance measure between points (or lines or line segments) in 3-dimensional space. The two spatial dimensions have the same physical units, but the time dimension has a different unit. We choose to use the Euclidean distance in 3-dimensional space and therefore propose to use a conversion parameter α that transforms time units into space units. Given a point p in 3-dimensional space, the 3-dimensional ball B_p with centre at p and radius ε contains exactly those points within distance at most ε from p. Hence, if we would like to know whether point p' is within distance ε of p, then this is the same as asking whether p' is inside B_p.

In our distance function $dist_\alpha$, the impact of α can be seen in two different ways: either as 'stretching' the t-axis or as 'flattening' the ball B_p. In the former, we can say that the bigger α, the longer the time axis (i.e. the more spatial length units that correspond to one time unit), and always consider a perfect ball B as basis for the distance between two points. In the latter, we keep the coordinate system fixed, but the bigger α the flatter the ball B in the t-dimension. Formally, the distance function is defined as follows.

Definition 2. *The distance $dist_\alpha$ between a point $p_m = (x_m, y_m, t_m)$ and a line segment $\overline{p_i p_j}$ is the shortest Euclidean distance in 3-dimensional space from p_m to a point p_c on $\overline{p_i p_j}$ where 1 time unit is equivalent to α space units, i.e.:*

$$dist_\alpha(p_m, \overline{p_i p_j}) = \sqrt{(x_m - x_c)^2 + (y_m - y_c)^2 + \alpha \cdot (t_m - t_c)^2}$$

The three distance functions E_2, E_3, and E_u defined in [4] are special cases of our distance function, namely $dist_0 \equiv E_2$ (where '$\equiv$' denotes equivalence), $dist_1 \equiv E_3$, and $dist_\infty \equiv E_u$. Choosing $\alpha = 0$ renders the time information irrelevant, and hence it is equivalent to projecting the line segment onto the xy-plane and using the Euclidean distance on it. This distance function has the advantage that it does simplify trajectories but it is not sound for the *where-at* query. The other extreme, $\alpha \to \infty$, denoted as $dist_\infty$, means the ball B_p is flattened into a 2-dimensional disk, which is parallel to the xy-plane. This means that the distance between a point p and a line segment $\overline{p_i p_j}$ is the Euclidean distance between p and p', where p' is the point on $\overline{p_i p_j}$ that has the same time value. This distance function has the advantage that it is sound for the *where-at* query, but it does not simplify trajectories.

Apart from being more general, our approach to be able to choose α has the advantage of allowing any distance function between $dist_0 \equiv E_2$ and $dist_\infty \equiv E_u$. Intuitively, we can fine-tune the trade-off between 'soundness' and 'sensible simplification', and we can prove $dist_\alpha$ to be sound for all α under certain conditions. To make this more precise, we incorporate the speed of entities in our considerations, where the speed s_ℓ along the line segment ℓ is defined as the distance in the xy-plane divided by the time difference corresponding to ℓ.

Theorem 1. *Let $\ell = \overline{p_i p_j}$ be a line segment with speed s_ℓ that is part of a $(dist_\alpha, \varepsilon)$-simplification of the trajectory $T = \langle p_1, \ldots, p_i, \ldots, p_j, \ldots, p_n \rangle$, and let t be any moment of time with $i \leq t \leq j$. Then we have:*

$$|\textit{where-at}(T,t) - \textit{where-at}(\ell,t)| \;\leq\; \delta_s \;:=\; \frac{\varepsilon}{\sin(\arctan \frac{\alpha}{s_\ell})}$$

The previous theorem tells us that the bigger $\frac{\alpha}{s_\ell}$ becomes the smaller gets δ_s. Hence, the distance function $dist_\alpha$ is sound according to Definition 1 for the *where-at* query for any $\alpha > 0$ as long as $0 < s_\ell < \infty$. However, in practice only a restricted range of values for α might be sensible. For instance setting $\alpha = s_{\max}$, where $s_{\max}$ is the maximum speed along the trajectory, results in $\delta_s \leq \sqrt{2} \cdot \varepsilon$ for the entire trajectory. Also values smaller than $s_{\max}$ might make sense for α in practice. In this case, the slower the speed on a line segment of the simplification is, the smaller δ_s is. In the same way as for the *where-at* query we also obtain that the *when-at* query is sound for $dist_\alpha$, if $\alpha \neq 0$ and $s_\ell > 0$.

Theorem 2. *Let $\ell = \overline{p_i p_j}$ be a line segment with speed s_ℓ that is part of a $(dist_\alpha, \varepsilon)$-simplification of the trajectory $T = \langle p_1, \ldots, p_i, \ldots, p_j, \ldots, p_n \rangle$, and let (x, y) be any point that lies exactly once on both the projections of ℓ and $\langle p_i, \ldots, p_j \rangle$ onto the xy-plane. Then we have:*

$$|\textit{when-at}(\langle p_i, \ldots, p_j \rangle, x, y) - \textit{when-at}(\ell, x, y)| \;\leq\; \delta_t \;:=\; \frac{\varepsilon}{\sin(\arctan \frac{s_\ell}{\alpha})}$$

Hence, the smaller $\frac{s_\ell}{\alpha}$ is, the bigger δ_t is. In practice it is sensible to assume that the speed of entities is bounded from above, it is unreasonable to assume that all entities have a minimum speed; this would forbid an entity to be stationary. Being able to choose α allows a user to fine-tune the trade-off between spatial and temporal soundness of $dist_\alpha$, as reflected by Theorems 1 and 2.

Cao et al. show in [4] that, if a distance function is sound for the *where-at* query, then it is also sound for the *nearest-neighbour* and *intersect* queries, and hence, Theorem 1 carries over to those queries, too. The *spatial-join* is special in the sense that the query itself uses a distance function between trajectories. For $\alpha_1 \leq \alpha_2$ we have that $dist_{\alpha_1}(p_m, \overline{p_i p_j}) \leq dist_{\alpha_2}(p_m, \overline{p_i p_j})$. From results in [4] it then follows that $dist_{\alpha_2}$ is sound for the *spatial-join* that uses the Hausdorff distance function based on $dist_{\alpha_1}$ as distance function between trajectories.

We believe that the definition of the *when-at* query as given in [4] is too strict. When considering the soundness of a distance function, we compare the original trajectory T and its simplification T'. Although all points of T' have a distance to T of at most ε, we could expect that $\textit{when-at}(T, x, y)$ or $\textit{when-at}(T', x, y)$ is undefined for almost all points (x, y), which renders any reasoning about soundness to be difficult. Hence, we propose different semantics for the *when-at* query. As simplified trajectories are approximations anyway, we allow a query region instead of a query point.

- $\textit{apx-when-at}(T, x, y, \lambda)$ returns a time t at which a moving object on trajectory T is expected to be within distance λ from location (x, y). If there is no such location on the trajectory, then the answer is undefined.

It seems impossible to prove the soundness of this query in the same sense as above. However, we can prove that *apx-when-at* will report a point at time t for which it holds that the entity must have been close to (x, y) at some point in time that is close to t. That is, we can prove a 'soundness' bound that has both a spatial and temporal error. To simplify the statement of the theorem we define *set-apx-when-at*(T, x, y, λ) as reporting the set of time points when the trajectory T is within distance λ from (x, y). For a trajectory T we use $T(t)$ to denote the position in the xy-plane of the entity along T at time t.

Theorem 3. *Let P be a ($dist_\alpha, \varepsilon$)-simplification of trajectory $T = \langle p_1, \ldots, p_n \rangle$. Given a query point $q = (x, y)$ in the xy-plane, let t_1 be the time reported by apx-when-at$(P, x, y, \lambda + \varepsilon)$. There exists a time point t_2 in set-apx-when-at$(T, x, y, \lambda + 2\varepsilon)$ such that $|t_1 - t_2| \leq \varepsilon/\alpha$ and $|T(t_1) - P(t_2)| \leq \varepsilon$.*

Note that if we set α to be greater than the largest speed of the entity then both *where-at* and *apx-when-at* can be sound for small errors at the same time for any input path. This is the first time any such bound has been shown using a single distance function, even though it is approximate in both time and space.

3 A fast implementation of the Douglas-Peucker algorithm for self-intersecting polygonal paths

In this section we present a fast implementation of the Douglas-Peucker algorithm in the case when the polygonal path may self-intersect. We consider two variants of the algorithm, one which works in the line-segment model and another which works in the line model. Hershberger and Snoeyink [11] gave an $O(n \log^* n)$-time algorithm working in the line model when the path does not self-intersect. However, their approach heavily rely on the fact that the path does not self-intersect since additional structure can be used in this case to develop efficient algorithms. Furthermore, their algorithm is developed to work in the line model, not in the line-segment model. In the self-intersecting case, only the trivial $O(n^2)$ time bound is known, to the best of the authors' knowledge.

As a first step, we will prove that, just as in the line model, the furthest point has to be a vertex on the convex hull of the point set (this is the only structural result we were able to reuse from [10, 11]). For simplicity we will throughout this section assume that no three points lie on a line.

Lemma 1. *Given a set of $n \geq 3$ points S and a line segment ℓ, the maximum distance between S and ℓ is defined between a vertex p on the convex hull of S.*

An important subproblem that we need to consider is the following.

Problem 1. (Line-segment furthest-point queries (LSFP-queries)) Preprocess an ordered set of n points $p_1, \ldots, p_n$ in convex position in the plane into a data structure supporting the following query: given a line segment (p_i, p_j), $1 \leq i < j \leq n$, report the point p_k that is furthest from (p_i, p_j) such that $i < k < j$.

Below we will prove that the LSFP-query problem can be transformed into the following problem with only a small loss in time and space complexity.

Problem 2. (Half-plane furthest-point queries (HPFP-queries)) Preprocess a set of n points $p_1, \ldots, p_n$ in convex position in the plane into a data structure supporting the following query: given a point q and a directed line ℓ, report the point p_i that is furthest from q subject to being to the left of ℓ.

Lemma 2. *A set S of n points in convex position in the plane can be preprocessed in $2F(n)+O(n \log n)$ time using $O(n)+S(n)$ space such that LSFP-queries can be answered in $2Q(n) + O(\log n)$ time, where $F(n)$ is the preprocessing time needed to store S in a data structure of size $S(n)$ that answers HPFP-queries in $Q(n)$ time.*

The $O(n \log n)$ time bound in the above lemma comes from the fact that we need to compute the convex hull of S. However, if the points in S are sorted with respect to their x-coordinates in increasing order, then this step can be done in $O(n)$ time. Unfortunately this improvement will not affect the overall time complexity of the Douglas-Peucker algorithm.

3.1 Half-plane furthest-point queries

The HPFP-query problem was first studied by Aronov et al. [2] and they showed the following two results:

Fact 1 *(Corollary 5 in [2]) There is a data structure that requires $O(n^{1+\beta})$ space and preprocessing time, and supports HPFP-queries in $O(2^{1/\beta} \log n)$ time on n points in convex position, for any real number $\beta > 0$.*

Fact 2 *(Corollary 11 in [2]) There is a data structure that requires $O(n \log^3 n)$ space and polynomial preprocessing time, and supports HPFP-queries in $O(\log n)$ time on n points in convex position.*

We present a data structure that has slightly higher query time, but because of smaller preprocessing time and smaller space consumption our approach leads to a more efficient implementation of the Douglas-Peucker algorithm.

Lemma 3. *One can preprocess a planar set S of n points in convex position in $O(n \log n)$ time using $O(n \log n)$ space such that HPFP-queries on S can be answered in $O(\log^2 n)$ time.*

3.2 Path simplification in the line-segment model

In this section we merge the results into one single data structure. In particular, we study the problem of preprocessing a polygonal path P with n vertices such that, given a line segment ℓ and a subpath P' of P, the point in P' furthest from ℓ is reported. We will prove the following lemma.

Lemma 4. *A polygonal path $P = \langle v_1, v_2, \ldots, v_n \rangle$ with n vertices in the plane can be preprocessed in time $O(n \log^2 n) + \sum_{i=0}^{\log n} 2^{i+1} F(\frac{n}{2^i}))$, using $O(n \log n) + \sum_{i=0}^{\log n} 2^i S(\frac{n}{2^i})$ space such that, given a line segment ℓ and a subpath $P' = \langle v_i, \ldots, v_j \rangle$ of P, the point in P' furthest from ℓ can be reported in time $O(\log^2 n) + 2\sum_{i=0}^{\log n} Q(\frac{n}{2^i})$, where $F(n)$ is the preprocessing time needed to construct a data structure of size $S(n)$ that can answer HPFP-queries in $Q(n)$ time.*

The standard Douglas-Peucker algorithm iterates over at most n line segments. Thus, by combining Lemmas 2, 3 and 4 we obtain the following theorem.

Theorem 4. *(line-segment model) For a polygonal path P with n vertices in the plane, the Douglas-Peucker algorithm can be implemented in time $O(n \log^3 n)$ using $O(n \log n)$ space.*

Note that in Lemma 4 presorting could be used to improve the preprocessing time by a logarithmic factor, but this does not have any effect on the asymptotic efficiency of the Douglas-Peucker algorithm.

3.3 Path simplification in the line model

Even if Theorem 4 also holds in the line model, the inclusive structure of the distance queries is not fully utilised. It turns out that path simplification is easier in the line model. Next we show how both the time and the space bounds can be improved by a logarithmic factor. The tools used in this improved construction are basically the same as those used before. The main reason for obtaining this improvement is that a vertex of a convex hull furthest from a line can be reported fast by binary search [16] by determining the two tangents parallel to the given line and returning the furthest of the vertices on these tangents.

The algorithm operates in four steps. First, the vertices on the given polygonal path P of size n are partitioned into canonical sets whose size is a power of 2. Let the collection of these sets be $\mathcal{P} = \{P_1, P_2, \ldots, P_h\}$. The size of P_1 should be the largest power of 2 no greater than n, the size of P_2 the largest power of 2 no greater than $n - |P_1|$, and so on. That is, $h \leq \lceil \log n \rceil$. Second, the canonical sets of $\mathcal{P}$ are presorted according to their x-coordinate. Let $\mathcal{S}$ be the corresponding collection of sorted sets of vertices. Also, associate each vertex in a sorted set with its index in the polygonal path. Third, the convex hulls of the canonical sets are computed. Let the resulting collection be $\mathcal{C}$. Due to presorting, the computation of each convex hull only takes linear time if we use Graham's convex-hull algorithm. Fourth, the recursive subroutine, to be described next, is called with ε, P, $\mathcal{P}$, $\mathcal{S}$, and $\mathcal{C}$.

Assume that the input of the recursive subroutine is real number ε and polygonal path $\langle v_i, v_{i+1}, \ldots, v_k \rangle$ together with the corresponding collections of canonical sets, sorted sets, and convex hulls. The functioning of the recursive subroutine is as follows:

1. Compute the furthest point between the polygonal path and the line ℓ determined by v_i and v_k. This is done by computing the furthest point between ℓ and the convex hulls, one by one, and by determining the overall furthest point. Let v_j be this vertex.
2. If the distance between line ℓ and vertex v_j is less than or equal to ε, return the line segment (v_i, v_k) as a simplification for P and stop this branch of recursion.
3. Split the path into two subpaths $\langle v_i, v_{i+1}, \ldots, v_j \rangle$ and $\langle v_j, v_{j+1}, \ldots, v_k \rangle$. Correspondingly, split the canonical set containing v_j into smaller canonical sets whose size is a power of 2. This is done by repeatedly halving the canonical

set containing v_j until v_j forms a singleton set. For each canonical set created during this process, compute the sorted set of vertices by scanning the sorted set corresponding to the parent canonical set. Finally, compute the convex hulls of the new canonical sets created. After halving a canonical set, it and the corresponding sorted set and convex hull are disposed.

4. Call the recursive routine for both subpaths together with the corresponding collections of canonical sets, sorted sets, and convex hulls.

Let us now analyse the performance of this algorithm for a polygonal path of n vertices. The amount of work done in the three first steps of the main routine is dominated by that required by sorting, i.e. the running time is $O(n \log n)$. In the recursive subroutine in connection with each halving, sorted sets are scanned and convex hulls may be computed, both requiring time linear on the size of the subpaths considered. Since each vertex is involved in $O(\log n)$ halvings, the overall running time of all splits is $O(n \log n)$. At each recursive step, in the furthest-point calculation the number of convex hulls to be considered is bounded by $O(\log n)$ and each distance computation between a line and a convex hull takes $O(\log n)$ time. Naturally, the number of recursive calls is linear in the worst case. Therefore, the total running time of the algorithm is $O(n \log^2 n)$. At any given point in time, each vertex can be in at most one canonical set. Hence, the space bound is $O(n)$. The above discussion can be summarised as follows:

Theorem 5. *(line model) For a polygonal path P with n vertices in the plane, the Douglas-Peucker algorithm can be implemented in time $O(n \log^2 n)$ using $O(n)$ space.*

4 A fast implementation of the Douglas-Peucker algorithm in 3-dimensional space

In this section, we present a fast, approximate version of the Douglas-Peucker algorithm in $\mathbb{R}^3$. The algorithm can be used for 3-dimensional paths that are monotone along the z-axis or for trajectories with two spatial dimensions and one temporal dimension. In addition to taking as input a distance error threshold ε, it takes a real number $\delta > 0$, and produces a simplified path that is within a distance of $(1 + \delta)\varepsilon$ from every vertex of the original path. It is possible to set $\varepsilon = \frac{\varepsilon^*}{1+\delta}$ to obtain a distance error bound of exactly some desired value ε^*. In this case, δ does not affect the distance threshold, but a larger δ may result in a larger number of vertices in the simplified path. As for the original Douglas-Peucker algorithm, this approach is a heuristic, and we present no bound on the number of vertices.

The general idea of the algorithm is as follows. First, we project the vertices of the original path onto $O(1/\delta^2)$ rotations of the xy-plane, equally spaced in angle around the y- and z-axes, yielding a 2-dimensional projection of the original path that may contain self-intersections. One of the 2-dimensional algorithms of Section 3 is then executed on each of the planes, up to the point where the

simplified path is to be split at a vertex. At this point, a split vertex has been chosen for each projection plane, based on the distance in the projection between that vertex and the proposed simplified line segment. From these potential split vertices, take the one with the maximum distance to the line segment over all of the projection planes. Split at this vertex in all planes, and continue executing. We will show that the original distance between the vertex and the line segment in $\mathbb{R}^3$ is at most $(1+\delta)$ times the maximum projected distance over all of the planes. This property allows us to construct an approximate simplification efficiently in 3-dimensional space.

We start by defining a set Ψ of projection planes. Given two angles $0 \leq \alpha \leq \pi$ and $0 \leq \beta \leq 2\pi$, let $\psi_{\alpha,\beta}$ be the plane obtained by rotating the xy-plane around the y-axis by α radians and around the z-axis by β radians, i.e. the plane with normal vector $\langle \sin\alpha\cos\beta, \sin\alpha\sin\beta, \cos\alpha\rangle$. Suppose we wish to perform $k = \lceil 2\pi/\arccos(1/(1+\delta))\rceil$ discrete rotations around the y-axis, and $2k$ around the z-axis. The angle between successive rotations around either of the axes will be $\theta = \pi/k$. Note that for any real $\delta > 0$, it holds that $0 < \theta < \pi/4$. Now we can define a set of projection planes $\Psi = \{\psi_{i\theta,j\theta} \mid i,j \in \mathbb{Z}, 0 \leq i < (k/2), 0 \leq j < k\}$.

Lemma 5. *Given a plane with normal vector $\hat{\boldsymbol{n}}$, there exists a plane $\psi^* \in \Psi$ with normal vector $\hat{\boldsymbol{n}}^*$ such that the angle between $\hat{\boldsymbol{n}}$ and $\hat{\boldsymbol{n}}^*$ is no more than θ.*

Given a point $p \in \mathbb{R}^3$ and a plane $\psi \in \Psi$, let $proj(p,\psi)$ be the orthogonal projection of p onto the plane ψ, defined as the point of intersection between ψ and the line orthogonal to ψ passing through p. To prove an approximation bound, we first need a bound on the distance between two projected points from their original distance in $\mathbb{R}^3$.

Lemma 6. *Given two points $p, q \in \mathbb{R}^3$, it holds that*

$$|\overline{pq}|\cos\theta \leq \max_{\psi\in\Psi} |\overline{proj(p,\psi)proj(q,\psi)}| \leq |\overline{pq}|$$

In the Douglas-Peucker algorithm, we are not only interested in the distance between two points, but also in the distance between a point and a line. We therefore need to look at the projection of the triangle given by the point and two points on the line.

Lemma 7. *Given three points $p, q, r \in \mathbb{R}^3$ such that $\angle pqr > 2\theta$, it holds that*

$$dist(q,\overline{pr}) \geq \max_{\psi\in\Psi} dist(proj(q,\psi), \overline{proj(p,\psi)proj(r,\psi)}) \geq \frac{dist(q,\overline{pr})}{\sqrt{2-\cos^2\theta}}$$

We are now ready for the final result of this section.

Theorem 6. *Given a real number $\delta > 0$, a $(1+\delta)$-approximate Douglas-Peucker simplification can be computed in the line-segment model in $O(\frac{1}{\delta^2} n \log^3 n)$ time using $O(\frac{1}{\delta^2} n \log n)$ space, and in the line model in $O(\frac{1}{\delta^2} n \log^2 n)$ time using $O(\frac{1}{\delta^2} n)$ space.*

References

1. P. K. Agarwal and K. R. Varadarajan. Efficient algorithms for approximating polygonal chains. *Discrete & Computational Geometry*, 23(2):273–291, 2000.
2. B. Aronov, P. Bose, E. D. Demaine, J. Gudmundsson, J. Iacono, S. Langerman, and M. Smid. Data structures for halfplane proximity queries and incremental Voronoi diagrams. In *Proceedings of the 7th Latin American Symposium on Theoretical Informatics*, volume 3887 of *Lecture Notes in Computer Science*, pages 80–92. Springer-Verlag, 2006.
3. H. Cao, O. Wolfson, and G. Trajcevski. Spatio-temporal data reduction with deterministic error bounds. In *Proceedings of the 2003 Joint Workshop on Foundations of Mobile Computing*, pages 33–42. ACM Press, 2003.
4. H. Cao, O. Wolfson, and G. Trajcevski. Spatio-temporal data reduction with deterministic error bounds. *The VLDB Journal*, 15(3):211–228, 2006.
5. W. S. Chan and F. Chin. Approximation of polygonal curves with minimum number of line segments. In *Proceedings of the 3rd Internattional Symposium on Algorithms and Computation*, volume 650 of *Lecture Notes in Computer Science*, pages 378–387. Springer-Verlag, 1992.
6. D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer*, 10(2):112–122, 1973.
7. A. U. Frank. Socio-Economic Units: Their Life and Motion. In A. U. Frank, J. Raper, and J. P. Cheylan, editors, *Life and motion of socio-economic units*, volume 8 of *GISDATA*, pages 21–34. Taylor & Francis, London, 2001.
8. J. Gudmundsson, P. Laube, and T. Wolle. *Encyclopedia of GIS*, chapter Movement Patterns in Spatio-Temporal Data. Springer, 2008. To appear.
9. R. Güting, M. H. Boehlen, M. Erwig, C. S. Jensen, N. Lorentzos, E. Nardelli, M. Schneider, and M. Vazirgiannis. A Foundation for representing and querying moving objects. *ACM Transactions on Database Systems*, 25(1):1–42, 2000.
10. J. Hershberger and J. Snoeyink. Speeding up the Douglas-Peucker line-simplification algorithm. In *Proceedings of the 5th International Symposium on Spatial Data Handling*, pages 134–143. IGU Commission on GIS, 1992.
11. J. Hershberger and J. Snoeyink. Cartographic line simplification and polygon CSG formulæ in $O(n \log^* n)$ time. *Computational Geometry—Theory and Applications*, 11(3–4):175–185, 1998.
12. I. A. R. Hulbert. GPS and its use in animal telemetry: The next five years. In A. M. Sibbald and I. J. Gordon, editors, *Proceedings of the Conference on Tracking Animals with GPS*, pages 51–60, Aberdeen, UK, 2001. Macaulay Insitute.
13. H. Imai and M. Iri. Computational-geometric methods for polygonal approximations of a curve. *Computer Vision, Graphics, and Image Processing*, 36(1):31–41, 1986.
14. M. P. Kwan. Interactive geovisualization of activity-travel patterns using three dimensional geographical information systems: A methodological exploration with a large data set. *Transportation Research Part C*, 8(1–6):185–203, 2000.
15. A. Melkman and J. O'Rourke. On polygonal chain approximation. In *Computational Morphology*, pages 87–95. North-Holland, 1988.
16. M. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23(2):166–204, 1981.