Compressing spatio-temporal trajectories[⋆]

Joachim Gudmundsson[a,∗] Jyrki Katajainen[b,1]
Damian Merrick[a,c] Cahya Ong[d] Thomas Wolle[a]

[a]*NICTA Sydney, Locked Bag 9013, Alexandria NSW 1435, Australia*[2]

[b]*Department of Computing, University of Copenhagen, Universitetsparken 1, 2100 Copenhagen East, Denmark*

[c]*School of Information Technologies, University of Sydney, Sydney NSW 2006, Australia*

[d]*School of Computer Science and Engineering, University of New South Wales, Sydney NSW 2052, Australia*

⋆ The results presented in this paper were announced at the 18th International Symposium on Algorithms and Computation held in Sendai in December 2007.

∗ Corresponding author.
Email addresses: `joachim.gudmundsson@nicta.com.au` (Joachim Gudmundsson), `jyrki@diku.dk` (Jyrki Katajainen), `damian.merrick@xpsoftware.com` (Damian Merrick), `thomas.wolle@nicta.com.au` (Thomas Wolle).

1 Part of this research was conducted when the author visited NICTA Sydney. 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").

2 NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

Abstract

A trajectory is a sequence of locations, each associated with a timestamp, describing the movement of a point. Trajectory data is becoming increasingly available and the size of recorded trajectories is getting larger. In this paper we study the problem of compressing planar trajectories such that the most common spatio-temporal queries can still be answered approximately after the compression has taken place. In the process, we develop an implementation of the Douglas-Peucker path-simplification algorithm which works efficiently even in the case where the polygonal path given as input is allowed to self-intersect. For a polygonal path of size n, the processing time is $O(n \log^k n)$ for $k = 2$ or $k = 3$ depending on the type of simplification.

Key words: trajectory compression, polygonal-chain approximation, path simplification, spatio-temporal queries, half-plane furthest-point queries, convex hulls

1 Introduction

Technological advances in location-aware devices, surveillance systems and electronic transaction networks are producing more and more opportunities to trace moving entities. Consequently, an eclectic set of disciplines including geography [10], database research [13], animal-behaviour research [16], surveillance and security analysis [21] and transport analysis [18] shows an increasing interest in movement patterns of various entities moving in various spaces over various timescales (see also the survey by Gudmundsson et al. [12]).

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. [6] proposed a way of modelling trajectories in 3-dimensional space so that 3-dimensional path-simplification techniques could be applied to trajectories. Their idea works well in practice, and in their experiments compression power was in most cases well over 90%. However, their approach has two main drawbacks that we improve upon 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-neighbour* and *spatial-join*. However, they were only able to prove that their approach is “sound” (to be defined) for three of the five types of queries. In this paper, we show that by making a small modification to their model all the five types of queries can be answered approximately after compression.
(2) They used the Douglas-Peucker path-simplification algorithm which has a quadratic running time in 3-dimensional space. In our specific case, we show that an approximate version of the Douglas-Peucker simplification algorithm can be computed in subquadratic time.

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 both spatial and temporal information.

In the *path-simplification problem*, a polygonal path $\langle v_1, v_2, \ldots, v_n \rangle$ and tolerance $\varepsilon > 0$ are given, and the task is to find a subsequence of $\langle v_1, v_2, \ldots, v_n \rangle$ such that each edge (v_i, v_j), $i < j$, in the simplification approximates subpath $\langle v_i, v_{i+1}, \ldots, v_j \rangle$ with tolerance ε. A *minimum-cardinality path simplification* is a path simplification that contains as few vertices as possible. Imai and Iri [17] solved the problem of computing a minimum-cardinality path simplification via a reduction to a graph problem: construct a directed acyclic graph, the edges of which model all possible short cuts with tolerance ε between the vertices $\{v_1, v_2, \ldots, v_n\}$, and use breadth-first search to compute a shortest path from v_1 to v_n in that graph (shortest with respect to the number of edges). Their algorithm runs in $O(n^2 \log n)$ time. Chan and Chin [7], and Melkman and O'Rourke [19] improved the running time to quadratic. Most of the known algorithms use $\Omega(n^2)$ time and space. A notable exception is the algorithm by Agarwal and Varadarajan [1] that requires $O(n^{\frac{4}{3}+\delta})$ 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 an almost linear-time algorithm for computing a minimum-cardinality path simplification remains unsolved, several heuristics have been proposed. The most widely used heuristics is the Douglas-Peucker method [8] (together with 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 $1 \leq i \leq n$, 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 that lies furthest from line ℓ and recursively approximate the two pieces. A straightforward implementation requires $\Theta(n)$ time to find the point furthest from line ℓ. Since the recursion depth is linear in the worst case, the worst-case running time is $\Theta(n^2)$.

A crucial aspect of path-simplification algorithms is the tolerance criterion used: When is a line segment accepted as an appropriate approximation for a subpath? Originally in the Douglas-Peucker algorithm, the Euclidean distance between the points on the given subpath and a line is measured (*line model*), where the line is defined by the endpoints of the subpath. This can lead to counter-intuitive simplifications as illustrated in Fig. 1(a). The input path is the solid path, and the distance between p_3 and the line containing p_1 and p_5 is much smaller than the distance between p_3 and the line segment connecting p_1 and p_5. In the line model, the final simplification would be the line segment (p_1, p_5), which might not be what we wish. That is why we also consider the tolerance criterion where the Euclidean distance between the points on the given subpath and a line segment is measured (*line-segment model*). Of course, many other tolerance criteria may be relevant in practical applications (see, e.g. [6,17]).

Even though the Douglas-Peucker algorithm does not necessarily output a minimum-cardinality simplification and its worst-case running time is $\Theta(n^2)$, it is often used due to its simplicity and efficiency in practice. In the case where the given polygonal path is simple (that is, non-self-intersecting) or monotone, faster methods have been developed. Hershberger and Snoeyink [14] showed that for simple paths in the line model, the running time can be improved by using 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, giving an algorithm with total running time of $O(n \log n)$. This was later improved to $O(n \log^* n)$ by the same authors [15]. 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 of Cao et al. [6] such that the five types of queries proposed by them can be approximated in a sound way (Section 2). As a result, 3-dimensional path-simplification algorithms can be used to compress trajectories.

(2) We propose an algorithm that produces a simplification of a (possibly self-intersecting) path in three dimensions, or a trajectory in the plane (Section 3). The algorithm is very similar to the Douglas-Peucker algorithm and can be viewed as an "approximate" version of it. The key idea is that the distance between a point and a line segment (or a line) is approximated. That is, if every vertex v_i, for $1 \leq i \leq n$, has distance at most ε to the line segment (v_1, v_n), then (v_1, v_n) is accepted as an approximation. If the distance between (v_1, v_n) and a vertex furthest from (v_1, v_n) is greater than ε but at most $(1+\delta)\varepsilon$, then the line segment (v_1, v_n) may, or may not, be accepted as an approximation. Otherwise, if there is a vertex further than $(1+\delta)\varepsilon$ from (v_1, v_n), then split the path at a vertex that lies further than ε from (v_1, v_n) and recursively approximate the two pieces. We call the output produced by this approach a $(1+\delta)$-approximation of the Douglas-Peucker simplification. The running time of our algorithm is $O(\frac{1}{\delta^3} n \log^2 n)$ in the line model and $O(\frac{1}{\delta^3} n \log^3 n)$ in the line-segment model.

(3) 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 (Section 4).

2 Modelling trajectories

In this section, we strengthen the model proposed in [6] for compressing trajectories. To begin with, we define spatio-temporal queries and soundness of distance functions. Then we describe our approach and prove its soundness.

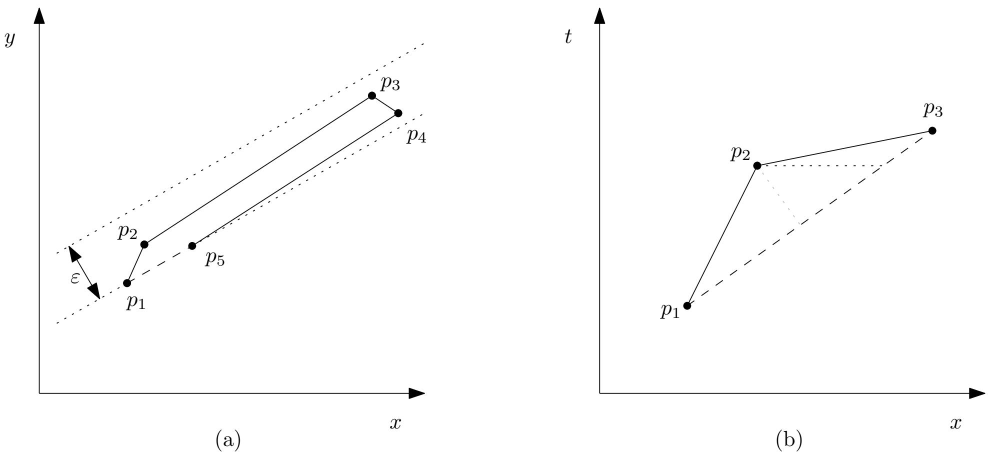


Fig. 1. The solid path shows the original trajectory and the dashed line shows the simplification. An example that is bad for (a) the line model since the line segment (p_1, p_5) would be a valid simplification of the path $\langle p_1, \ldots, p_5 \rangle$, and (b) the distance function E_u.

2.1 Preliminaries

According to Cao et al. [6], most spatio-temporal queries are composed of the following five types of queries: *where-at*, *when-at*, *intersect*, *nearest-neighbour* and *spatial-join*. Let $T = \langle p_1, p_2, \ldots, p_n \rangle$ be a trajectory, and assume that for each $i \in \{1, 2, \ldots, n\}$ the location of p_i is (x_i, y_i) at time t_i. For a trajectory T, the semantics of the two most basic queries is defined as follows:

- *where-at*(T, t) returns the location of the entity, moving along T, at time t. If $t < t_1$ or $t > t_n$, the answer is undefined.
- *when-at*$(T, (x, y))$ returns the time at which the entity, moving along T, is expected to be at location (x, y). If the location is not on the trajectory, or the entity visits the location more than once or is stationary at that location, the answer is undefined. [3]

See [6] for details on the other types of queries. For *query* $\in$ {*where-at*, *when-at*} and trajectory T, let *query*$(T, .)$ denote the answer to *query* with input T (and another unspecified input).

Let us now embed the trajectory T into 3-dimensional space such that the coordinates of p_i are (x_i, y_i, t_i). The notion of soundness of 3-dimensional distance functions was discussed in [6]. To make the dependence on both the underlying distance function *dist* and tolerance ε explicit, we let a $(dist, \varepsilon)$-*simplification* denote a compressed trajectory that is obtained by computing a 3-dimensional path simplification using *dist*.

[3] This definition is taken from [5]; the definition in [6] is similar but considers the stationary case as a special case.

Definition 1 *The 3-dimensional distance function dist is* sound *for query* $\in \{where\text{-}at, when\text{-}at\}$*, if for any tolerance* $\varepsilon > 0$ *there exists an error bound* $\delta(\varepsilon) > 0$ *such that for every trajectory* T *and its* $(dist, \varepsilon)$*-simplification* S *we have* $|query(T, .) - query(S, .)| \leq \delta(\varepsilon)$*. Let* t *be a time and* (x, y) *a location in the plane. For the where-at query,* $|query(T, t) - query(S, t)|$ *refers to the Euclidean distance between the locations given as answers, and for the when-at query,* $|query(T, (x, y)) - query(S, (x, y))|$ *refers to the difference between the returned times.*

Cao et al. [6] defined the following distance functions between any point (not necessarily on a trajectory) $p_m = (x_m, y_m, t_m)$ and a line segment $\overline{p_ip_j}$: E_2 (2-dimensional Euclidean distance which ignores temporal information), E_3 (3-dimensional Euclidean distance), E_u ($E_u(p_m, \overline{p_ip_j}) = \sqrt{(x_m - x_c)^2 + (y_m - y_c)^2}$ where p_c is the point on $\overline{p_ip_j}$ with $t_m = t_c$), and E_t ($E_t(p_m, \overline{p_ip_j}) = |t_m - t_c|$ where p_c is the point on the 2-dimensional projection of $\overline{p_ip_j}$ onto the xy-plane that is closest to the 2-dimensional projection of p_m onto the xy-plane). They also showed 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 proposed 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 distance functions, but also their weaknesses. This combined distance function resulted in the worst compression power among the researched distance functions.

We argue that using E_u gives rise to another problem. Consider the trajectory shown in Fig. 1(b), where an entity moves along the x-axis (i.e. $y = 0$) and slightly changes 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. To see this, note that in Fig. 1(b) the shortest distance (grey line) from vertex p_2 of the trajectory (solid line) to a point on the simplification (dashed line) is much smaller than the distance (dotted line) between p_2 and a point on the simplification that has the same time as p_2, which is the distance defined by E_u.

2.2 Our model

As in [6], 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 function 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 μ that transforms time units into space units. Given a point p in 3-dimensional space, the 3-dimensional ball $B(p, \varepsilon)$ 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 q is within distance ε of p, then this is the same as asking whether q is inside $B(p, \varepsilon)$.

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

Definition 2 *The distance $dist_\mu$ between a point p_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 one time unit is equivalent to μ space units, i.e.*

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

The three distance functions E_2, E_3 and E_u defined in [6] 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 $\mu = 0$ 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 as shown in Fig. 1(b), but it is not sound for the *where-at* query. The other extreme, $\mu \to \infty$, denoted as $dist_\infty$, means that the ball $B(p, \varepsilon)$ is flattened into a 2-dimensional disk, which is parallel to the xy-plane. That is, the distance between a point p and a line segment $\overline{p_i p_j}$ is the Euclidean distance between p and q, where q 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 like the one in Fig. 1(b).

Apart from being more general, our approach to be able to choose μ 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 $dist_\mu$ to be sound for any μ 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 3 *Let $\ell = \overline{p_i p_j}$ be a line segment that is part of a $(dist_\mu, \varepsilon)$-simplification of the trajectory $T = \langle p_1, \ldots, p_i, \ldots, p_j, \ldots, p_n \rangle$, and let t be any moment in time with $t_i \leq t \leq t_j$. Then we have*

$$|where\text{-}at(T,t) - where\text{-}at(\ell,t)| \leq \delta_s := \frac{\varepsilon \cdot \sqrt{s_\ell^2 + \mu^2}}{\mu}.$$

PROOF. Let $p_m = (x_m, y_m, t_m)$ be the point (not necessarily a vertex) on the trajectory T and $p_k = (x_k, y_k, t_k)$ be the point on the line segment ℓ with $t = t_m = t_k$, see Fig. 2. Let p_c be the point on the line containing ℓ that is closest to p_m. We know that for any p_h on ℓ, $p_h \neq p_c$, the angle $\angle p_h p_c p_m$ is a right angle. The spatial error is the distance between p_k and p_m:

$$|where\text{-}at(T,t) - where\text{-}at(\ell,t)| = |p_k p_m| = \frac{|p_c p_m|}{\sin(\angle p_c p_k p_m)} \leq \frac{\varepsilon}{\sin(\angle p_c p_k p_m)}.$$

Let γ denote the angle between ℓ and the xy-plane, and note that $\overline{p_k p_m}$ is parallel to the xy-plane. Now consider the angle $\angle p_c p_k p_m$. This angle is at most $\frac{\pi}{2}$, since $\triangle p_c p_k p_m$ is a right triangle with the angle at p_c being equal to $\frac{\pi}{2}$. Furthermore, this angle is at least γ, because it is minimised when $\overline{p_k p_m}$ is in the plane that both contains ℓ and is perpendicular to the xy-plane. Hence we have that $\gamma \leq \angle p_c p_k p_m \leq \frac{\pi}{2}$. Therefore, $\sin \gamma \leq \sin(\angle p_c p_k p_m)$. Now, note that $\tan \gamma = \frac{\mu \cdot (t_j - t_i)}{\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}}$. Since $s_\ell = \frac{\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}}{t_j - t_i}$, we obtain $\tan \gamma = \frac{\mu}{s_\ell}$. Altogether, we obtain

$$\frac{\varepsilon}{\sin(\angle p_c p_k p_m)} \leq \frac{\varepsilon}{\sin \gamma} = \frac{\varepsilon}{\sin(\arctan \frac{\mu}{s_\ell})} = \frac{\varepsilon \cdot \sqrt{s_\ell^2 + \mu^2}}{\mu},$$

where the last equality follows from using the identity $\sin(\arctan(x)) = \frac{x}{\sqrt{1+x^2}}$, which holds for any real number $x > 0$. The theorem also holds for the degenerate case, where the angle between ℓ and $\overline{p_k p_m}$ is a right angle, because in this case $p_c = p_k$ and $|where\text{-}at(T,t) - where\text{-}at(\ell,t)| \leq \varepsilon$. $\Box$

The previous theorem tells us that the bigger $\frac{\mu}{s_\ell}$ becomes, the smaller gets δ_s. Hence, the distance function $dist_\mu$ is sound according to Definition 1 for the *where-at* query for any $\mu > 0$ as long as $0 < s_\ell < \infty$. However, in practice only a restricted range of values for μ might be sensible. For instance, setting $\mu = s_{\max}$, where $s_{\max}$ is the maximum speed along the trajectory, results in an upper bound on the error bound of $\delta_s \leq \sqrt{2} \cdot \varepsilon$ for the entire trajectory. Also values smaller than $s_{\max}$ might make sense for μ 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 $dist_\mu$, if $\mu \neq 0$ and $s_\ell > 0$.

Theorem 4 *Let $\ell = \overline{p_i p_j}$ be a line segment that is part of a $(dist_\mu, \varepsilon)$-simplification of 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*

$$|when\text{-}at(\langle p_i, \ldots, p_j \rangle, (x, y)) - when\text{-}at(\ell, (x, y))| \leq \delta_t := \frac{\varepsilon \cdot \sqrt{s_\ell^2 + \mu^2}}{s_\ell}.$$

Hence, the smaller $\frac{s_\ell}{\mu}$ is, the bigger δ_t is. While 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. Nevertheless, being able to choose μ allows a user to fine-tune the trade-off between spatial and temporal soundness of $dist_\mu$, as reflected by Theorems 3 and 4.

Cao et al. showed in [6] 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 3 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 $\mu_1 \leq \mu_2$ we have that $dist_{\mu_1}(p_m, \overline{p_i p_j}) \leq dist_{\mu_2}(p_m, \overline{p_i p_j})$. From the results proved in [6] it follows that $dist_{\mu_2}$ is sound for the *spatial-join* query that uses the Hausdorff distance function based on $dist_{\mu_1}$ as the distance function between trajectories.

We believe that the definition of the *when-at* query as given in [6] is too strict. When considering the soundness of a distance function, we compare the original trajectory T and its simplification S. Although all points of S

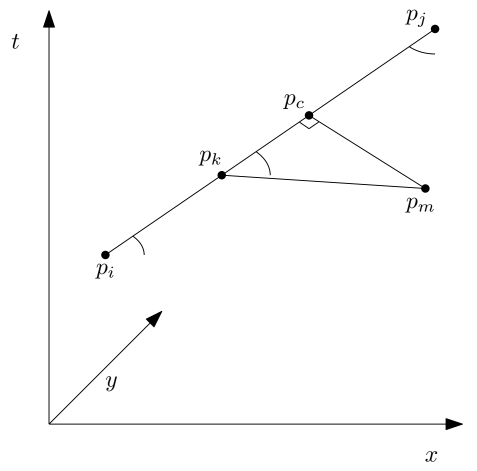


Fig. 2. Illustrating the proof of Theorem 3.

have a distance of at most ε to T, we could expect that *when-at*$(T,(x,y))$ or *when-at*$(S,(x,y))$ is undefined for almost all points (x,y), which makes reasoning about soundness difficult. Hence, we propose different semantics for the *when-at* query (similar considerations can also be made for other queries). As simplified trajectories are approximations anyway, we allow a query region instead of a query point:

- *apx-when-at*$(T,(x,y),\lambda)$ returns a time t at which an entity, moving along trajectory T, is 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 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 an error bound that has both a spatial and temporal component. To simplify the statement of the theorem we define *set-apx-when-at*$(T,(x,y),\lambda)$ 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 5 *Let S be a $(dist_\mu,\varepsilon)$-simplification of trajectory $T=\langle p_1,\dots,p_n\rangle$. Given a query point $q=(x,y)$ in the xy-plane and $\lambda\geq 0$, let t_1 be the time reported by apx-when-at$(S,(x,y),\lambda+\varepsilon)$. There exists a time point t_2 in set-apx-when-at$(T,(x,y),\lambda+2\varepsilon)$ such that $|t_1-t_2|\leq\frac{\varepsilon}{\mu}$ and $|T(t_1)-S(t_2)|\leq\varepsilon$.*

PROOF. From the definition of *apx-when-at* we have that $|S(t_1)-q|\leq\lambda+\varepsilon$. Since S is a $(dist_\mu,\varepsilon)$-simplification of the trajectory T, there exists a point $T(t_2)$ on T such that $|S(t_1)-T(t_2)|\leq\varepsilon$. Hence, $S(t_1)$ must be within the ball $B(T(t_2),\varepsilon)$ and due to the scaling of the t-axis this implies that $|t_1-t_2|\leq\frac{\varepsilon}{\mu}$. It remains to prove that $t_2\in$ *set-apx-when-at*$(T,(x,y),\lambda+2\varepsilon)$. This is equivalent to proving that the distance between $T(t_2)$ and q is at most $\lambda+2\varepsilon$, which is true since $|T(t_2)-q|\leq|T(t_2)-S(t_1)|+|S(t_1)-q|\leq\lambda+2\varepsilon$. $\square$

Note that, if we set μ 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 trajectory. This is the first time any such bound has been shown using a single distance function, even though it is approximate both in time and space.

3 A fast path simplification algorithm in 3-dimensional space

In this section, we show how an approximate version of the Douglas-Peucker simplification can be obtained in $\mathbb{R}^3$ by projecting the path onto a constant number of planes in $\mathbb{R}^3$ and then running a Douglas-Peucker algorithm on each projection in parallel. In Section 4, we will then provide a fast implementation of the Douglas-Peucker algorithm for any path (that may self-intersect) in $\mathbb{R}^2$. The algorithm can be used for 3-dimensional paths 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. We assume that $\delta \leq 1$, if this is not the case, we set $\delta = 1$ in our computations, which guarantees a solution which is at least as good as one for the given δ. 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 smaller δ may result in a larger number of vertices in the simplified path. As for the original Douglas-Peucker algorithm, this approach is a heuristics, 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 $O(\frac{1}{\delta^3})$ rotations of the xy-plane, systematically spaced in angle around the y- and z-axes, yielding a 2-dimensional projection of the original path that may contain self-intersections. An implementation of the Douglas-Peucker algorithm for a path in $\mathbb{R}^2$ is then executed on each of the planes, up to the stage where the simplified path is to be split at a vertex. At this stage, 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 a simplification efficiently in 3-dimensional space.

Before we describe a set Ψ of projection planes, we make the following observations. When orthogonally projecting a set of points in $\mathbb{R}^3$ onto a plane ψ, only the orientation of the plane is ultimately important for the projected image. This image remains the same if the projection plane is translated in any direction. Therefore, defining a projection plane by its normal vector is sufficient. A normal vector can be defined as a single point on the unit sphere. Hence, our set Ψ of projection planes can be defined as a set of points on the unit sphere. To do this, we first define a grid on the unit sphere similar to the longitude/latitude system that is used for the earth (see Fig. 3(a)). And we place points that represent normal vectors on the lines of the grid.

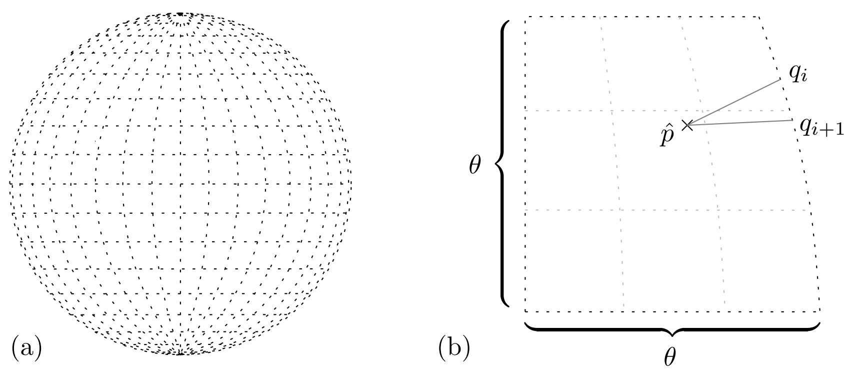


Fig. 3. (a) the grid on the unit sphere; (b) nine neighbouring grid cells enlarged

The fineness of the grid is governed by δ, such that the angular difference between three consecutive grid cells is θ (see Fig. 3(b)), where

$$\theta = \frac{\pi}{\lceil \frac{2\pi}{\arccos(\frac{1}{1+\delta})} \rceil}.$$

Note that for any real $\delta > 0$, it holds that $0 < \theta < \frac{\pi}{4}$.

The fineness of the points on the lines of the grid depends on θ (and hence, also on δ) in the following way: We put $\frac{8}{\theta}$ points per grid cell edge, equally spaced. This defines a set Π of points. We define another set Π' of points as a copy of Π and rotate it by $\frac{\pi}{2}$, such that the poles of Π' lie on the equator of Π. Recall that each of the points in $\Pi \cup \Pi'$ defines a vector which is the normal vector of the corresponding plane. Our set of projection planes Ψ consists of all those planes. From the definition of Ψ, we can conclude the following lemma.

Lemma 6 *Given three points p, q and r in $\mathbb{R}^3$, let ψ be the plane with normal vector $\vec{n}$ containing these points. There exists a plane $\psi^* \in \Psi$ with normal vector $\vec{n}^*$, such that the angle between $\vec{n}$ and $\vec{n}^*$ is no more than θ, and the angle between $\overline{pr}$ and ℓ, where ℓ is the line of intersection of ψ and ψ^*, is no more than θ.*

PROOF. Let $\hat{p}$ be the point on the unit sphere that corresponds to ψ and $\vec{n}$, respectively. Consider either Π or Π', depending on in which coordinate system $\hat{p}$ has smaller latitude. W.l.o.g., let Π be this point set, and note that point $\hat{p}$ on the unit sphere has a latitude of at most $\frac{\pi}{4}$. That is, the vector corresponding to $\hat{p}$ forms an angle of at most $\frac{\pi}{4}$ with the plane defined by the equator of Π.

We need points on the grid lines near $\hat{p}$, such that the vectors corresponding to these points and $\hat{p}$ form an angle of at most θ. The point $\hat{p}$ lies inside a grid cell (if $\hat{p}$ lies on the grid, then choose an arbitrary neighbouring cell) surrounded by eight other grid cells (see Fig. 3(b)). For an easier argumentation and analysis, we disregard grid lines indicated in grey and only focus on grid lines in black in Fig. 3(b). The first key observation is that any point on the black grid lines as described above (see Fig. 3(b)) corresponds to a vector that forms an angle of at most θ with the vector corresponding to $\hat{p}$. This follows from the definition of the grid itself.

Let $q_1, q_2, \ldots$ be all those points on the black grid lines sorted around $\hat{p}$. We also need an upper bound for the maximum angle

$$\max_i \angle q_i \hat{p} q_{i+1} \leq \theta. \tag{1}$$

We put the points on the black grid lines such that they are dense enough to obtain (1). (Additional points on grey grid lines near $\hat{p}$ can only decrease the maximum angle.) The worst case (i.e. the largest angle $\angle q_i \hat{p} q_{i+1}$) occurs when $\hat{p}$ is near or on the crossing of two grid lines and has high latitude, because in such a case $\hat{p}$ is closest to some points, say, q_i and q_{i+1} on the grid lines.

The shortest distance on the sphere along grid lines between any two grid intersection points in Fig. 3(b) is bounded from below by $\frac{\theta}{6}$, which follows from $\hat{p}$ having latitude at most $\frac{\pi}{4}$ and $\theta < \frac{\pi}{4}$. Hence, the shortest direct distance between any two grid intersection points is bounded from below by $\sin(\frac{\theta}{6}) \geq \frac{\theta}{12}$, since $\theta < \frac{\pi}{4}$. Now, assume we consider two consecutive points q_i and q_{i+1} on the black grid lines that have both a distance of $\frac{\theta}{12}$ to point $\hat{p}$. Putting $\frac{8}{\theta}$ equally spaced points per grid cell edge results in a distance between them of at most $\frac{\theta^2}{24}$, since a grid cell has a length of at most $\frac{\theta}{3}$. Hence, $\angle q_i \hat{p} q_{i+1} \leq \theta$, because $\sin(\frac{\angle q_i \hat{p} q_{i+1}}{2}) = \frac{1}{2} \cdot \frac{\theta^2}{24} / \frac{\theta}{12} = \frac{\theta}{4} \leq \sin(\frac{\theta}{2})$.

This implies the existence of a point in Π on the black grid lines that corresponds to a plane as required in the lemma. $\square$

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 approximate bound, we first need a bound on the distance between two projected points from their original distance in $\mathbb{R}^3$.

Lemma 7 *Given two points $p, q \in \mathbb{R}^3$, it holds that*

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

PROOF. It follows from Lemma 6 that for any plane containing $\overline{pq}$ with normal $\vec{n}$, there exists a plane $\psi^* \in \Psi$ with normal $\vec{n}^*$ such that the angle ω between $\vec{n}$ and $\vec{n}^*$ is between 0 and θ. The length of $\overline{pq}$ projected onto ψ^* is $|\overline{proj(p,\psi^*) proj(q,\psi^*)}| = |\overline{pq}| \sin(\frac{\pi}{2} - \omega) = |\overline{pq}| \cos\omega$. The maximum value of $|\overline{proj(p,\psi)proj(q,\psi)}|$ over all $\psi \in \Psi$ occurs when ω is smallest. Hence, $|\overline{pq}|\cos\theta \leq \max_{\psi\in\Psi} |\overline{proj(p,\psi)proj(q,\psi)}| \leq |\overline{pq}|$. $\square$

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 8 *Let q be a point and l be a line in $\mathbb{R}^3$. Let p and r be two points on line l such that both $\angle qrp$ and $\angle qpr$ are less than $\frac{\pi}{2} - \theta$, i.e. $dist(q,l) = dist(q,\overline{pr})$. It holds that*

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

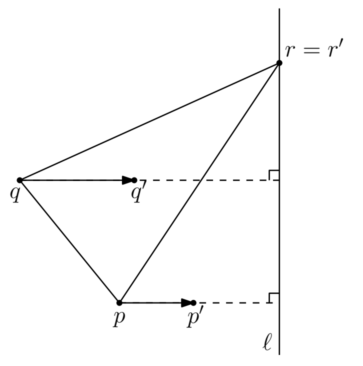
PROOF. The first inequality follows from the fact that the distance between any two points may only be reduced when orthogonally projected on any plane (see Lemma 7).

For the second inequality, let ψ be the plane containing p, q and r. By Lemma 6, there is a plane $\psi^* \in \Psi$ within an angle of θ from ψ. Consider the line ℓ of intersection between ψ and ψ^*. Note that we can translate ψ^* and hence ℓ arbitrarily and obtain the same orthogonal projection; only the orientation of ψ^* and ℓ are ultimately important. Without loss of generality, assume that ψ is the xy-plane and that ℓ is the vertical line passing through the point r, as in Fig. 4(a).

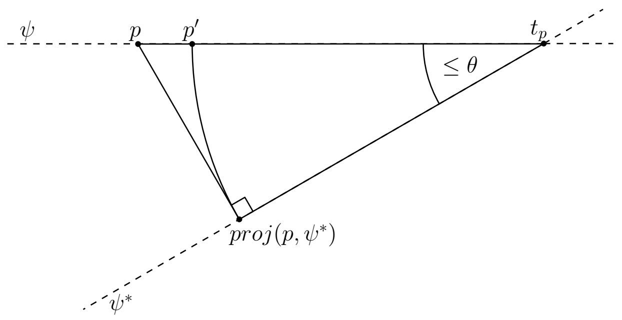
To simplify the analysis, we rotate the projected points $proj(p,\psi^*)$, $proj(q,\psi^*)$ and $proj(r,\psi^*)$ back into the xy-plane and call the rotated points p', q' and r', respectively. Let t_p be the point on ℓ that is closest to p (Fig. 4(b)). Then $|\overline{t_p p'}| = |\overline{t_p proj(p,\psi^*)}| \geq |\overline{t_p p}|\cos\theta$. Similarly, $|\overline{t_q q'}| \geq |\overline{t_q q}|\cos\theta$ and $|\overline{t_r r'}| \geq |\overline{t_r r}|\cos\theta$.

Let ρ be the angle in the triangle $\triangle pqr$ at r, let ϕ be the angle between ℓ and $\overline{p'r}$ (see Fig. 4(c)). Define the point u as the intersection of $\overline{qt_q}$ with $\overline{p'r}$, v as the intersection of $\overline{p'r}$ with its perpendicular through q', and w as the intersection of $\overline{pr}$ with its perpendicular through q.

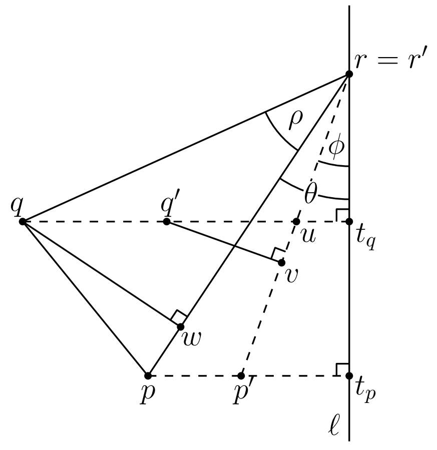
Let $h_{old} = |\overline{qw}| = dist(q,\overline{pr})$ and $h_{new} = |\overline{q'v}| = dist(q',\overline{p'r'})$. Our aim is to find the ratio $\frac{h_{old}}{h_{new}}$. Note that we can always choose $\psi^* \in \Psi$ such that the angle between ℓ and $\overline{pr}$ is at most θ, as a direct consequence of Lemma 6. Furthermore, by our choice of p and r, it follows that the projection of q is in between the projections of r and p, when projected onto ℓ. The best case occurs when the angle between ℓ and $\overline{pr}$ is zero, and we obtain $\frac{h_{old}}{h_{new}} = 1$. We



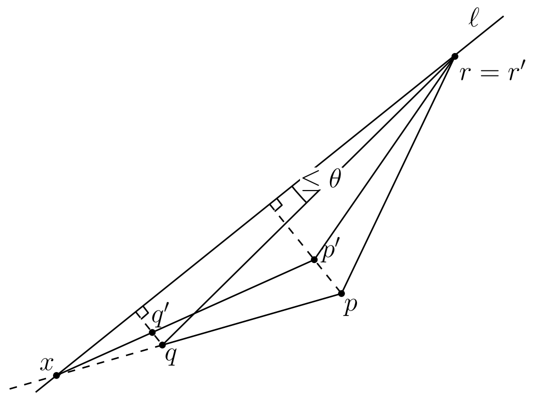
(a) p, q and r move orthogonally towards ℓ when projected.



(b) $|\overline{t_p p'}| = |\overline{t_p proj(p, \psi^*)}| \geq |\overline{t_p p}| \cos\theta$.



(c) $|\overline{qw}| \geq |\overline{q'v}|\sqrt{2 - \cos^2\theta}$



(d) $\angle q'p'r' \geq \angle qpr$

Fig. 4. Illustrating the proofs of Lemma 8 and Theorem 9.

now analyse the case when this angle is θ. First, we calculate ϕ from $\triangle p' t_p r$:

$$\begin{aligned}
|\overline{p' t_p}| &= |\overline{p t_p}| \cos\theta \\
|\overline{p t_p}| &= |\overline{r t_p}| \tan\theta \\
|\overline{p' t_p}| &= |\overline{r t_p}| \tan\theta \cos\theta = |\overline{r t_p}| \sin\theta \\
\tan\phi &= \frac{|\overline{p' t_p}|}{|\overline{r t_p}|} = \frac{|\overline{r t_p}| \sin\theta}{|\overline{r t_p}|} = \sin\theta \\
\phi &= \arctan(\sin\theta)\,. \qquad (2)
\end{aligned}$$

We can obtain h_{old} from $\triangle qwr$:

$$h_{old} = |\overline{qw}| = |\overline{qr}| \sin\rho\,. \qquad (3)$$

Now we consider h_{new}. Observe that $\angle q'uv = \angle rut_q$, and thus $\angle vq'u = \phi$.

$$\begin{aligned} h_{new} &= |\overline{q'v}| = |\overline{q'u}| \cos\phi \\ |\overline{q'u}| &= |\overline{q't_q}| - |\overline{ut_q}| \\ |\overline{q't_q}| &= |\overline{qt_q}| \cos\theta \\ |\overline{qt_q}| &= |\overline{qr}| \sin(\rho+\theta)\,. \end{aligned}$$

So we have

$$|\overline{q't_q}| = |\overline{qr}| \sin(\rho+\theta)\cos\theta\,. \tag{4}$$

Now calculate $|\overline{ut_q}|$:

$$\begin{aligned} |\overline{ut_q}| &= |\overline{rt_q}| \tan\phi \\ |\overline{rt_q}| &= |\overline{qr}| \cos(\rho+\theta) \\ |\overline{ut_q}| &= |\overline{qr}| \cos(\rho+\theta)\tan\phi\,. \end{aligned} \tag{5}$$

Combining (4) and (5) gives us $|\overline{q'u}|$:

$$\begin{aligned} |\overline{q'u}| &= |\overline{q't_q}| - |\overline{ut_q}| \\ &= |\overline{qr}| \sin(\rho+\theta)\cos\theta - |\overline{qr}| \cos(\rho+\theta)\tan\phi \\ &= |\overline{qr}| \left(\sin(\rho+\theta)\cos\theta - \cos(\rho+\theta)\tan\phi\right)\,. \end{aligned}$$

We can now calculate h_{new}:

$$\begin{aligned} h_{new} &= |\overline{q'v}| \\ &= |\overline{q'u}| \cos\phi \\ &= |\overline{qr}| \cos\phi \left(\sin(\rho+\theta)\cos\theta - \cos(\rho+\theta)\tan\phi\right)\,. \end{aligned}$$

Substituting (2) into this, we obtain

$$\begin{aligned} h_{new} &= |\overline{qr}| \cos\left(\arctan(\sin\theta)\right)\left(\sin(\rho+\theta)\cos\theta - \cos(\rho+\theta)\sin\theta\right) \\ &= |\overline{qr}| \cos\left(\arcsin\left(\frac{\sin\theta}{\sqrt{\sin^2\theta+1}}\right)\right)\sin((\rho+\theta)-\theta) \\ &= |\overline{qr}| \cos\left(\arccos\left(\sqrt{1-\frac{\sin^2\theta}{\sin^2\theta+1}}\right)\right)\sin\rho \\ &= |\overline{qr}|\sqrt{\frac{\sin^2\theta+1-\sin^2\theta}{\sin^2\theta+1}}\sin\rho\,. \end{aligned}$$

Thus,

$$h_{new} = \frac{|\overline{qr}|\sin\rho}{\sqrt{2-\cos^2\theta}}\,. \tag{6}$$

We can now combine (3) and (6) to calculate the desired ratio

$$\frac{h_{old}}{h_{new}} = \frac{|\overline{qr}| \sin\rho \sqrt{2-\cos^2\theta}}{|\overline{qr}| \sin\rho}\,.$$

Thus,

$$\frac{h_{old}}{h_{new}} = \sqrt{2-\cos^2\theta}\,. \tag{7}$$

When the angle between ψ and ψ^* is θ, $dist(proj(q,\psi^*), \overline{proj(p,\psi^*)proj(r,\psi^*)}) \geq \frac{dist(q,\overline{pr})}{\sqrt{2-\cos^2\theta}}$. This derivation works out also for all angles between 0 and θ. And since $\cos(\theta)$ is monotone for $0 < \theta < \frac{\pi}{4}$, it follows that the same inequality holds for all angles between 0 and θ. Therefore

$$\begin{aligned} dist(q,\overline{pr}) &\geq \max_{\psi\in\Psi} dist(proj(q,\psi), \overline{proj(p,\psi)proj(r,\psi)}) \\ &\geq \frac{dist(q,\overline{pr})}{\sqrt{2-\cos^2\theta}}. \qquad \square \end{aligned}$$

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

Theorem 9 *Given two real numbers $\delta, \varepsilon > 0$ and a path in $\mathbb{R}^3$, a $(1+\delta)$-approximate version of the Douglas-Peucker algorithm can be implemented to run in $O(\frac{1}{\delta^3}\cdot T(n,\varepsilon))$ time using $O(\frac{1}{\delta^3}\cdot S(n,\varepsilon))$ space, where $T(n,\varepsilon)$ and $S(n,\varepsilon)$ are the time and space required by an implementation of the Douglas-Peucker algorithm in $\mathbb{R}^2$.*

PROOF. First, consider the line model. Construct a set of planes Ψ as described earlier. Project the input points onto each of these planes. Now execute the Douglas-Peucker algorithm in $\mathbb{R}^2$ for each plane, and whenever a split is to be performed, choose the split vertex with the maximum projected distance over all of the projection planes. Luckily, our implementations of the Douglas-Peucker algorithm to be described in Section 4 operate iteratively by computing the split vertex and solving the subproblems recursively. Moreover, the split vertex can be selected arbitrarily, even determined by an external source as here, without affecting the running time of the algorithm.

Let q be the chosen split vertex, and p and r be the endpoints of the current simplified line segment. By Lemma 8, we know that as long as both $\angle qrp$ and $\angle qpr$ are less than $\frac{\pi}{2}-\theta$, there is some plane in which the actual distance from q to the line through $\overline{pr}$ is at most $\sqrt{2-\cos^2\theta}$ times the projected distance. The same lemma also gives us that in all planes the projected distance is at most equal to the actual distance. Since we are interested in the distance to a line, not a line segment, we can always choose p and r on the same line that are sufficiently far apart to guarantee that the conditions are satisfied, and therefore Lemma 8 gives an approximate bound for every point-to-line distance.

Recall our definition of θ:

$$\theta = \frac{\pi}{\lceil \frac{2\pi}{\arccos(\frac{1}{1+\delta})} \rceil} \text{ for } \delta > 0\,.$$

By this definition, $0 < \theta < \frac{\pi}{4}$. Then we have

$$\begin{aligned}
\theta &\leq \frac{\pi}{\left(\frac{2\pi}{\arccos\left(\frac{1}{1+\delta}\right)}\right)} \\
2\theta &\leq \arccos\left(\frac{1}{1+\delta}\right) \\
\cos(2\theta) &\geq \frac{1}{1+\delta} \\
1+\delta &\geq \frac{1}{\cos(2\theta)} \\
&\geq \frac{1}{\cos\theta}\,.
\end{aligned}$$

Thus,

$$1+\delta \geq \frac{1}{\cos\theta} \tag{8}$$

and

$$1+\delta \geq \sqrt{2-\cos^2\theta}\,, \tag{9}$$

which follows from (8). Combining (9) with (7) from Lemma 8 gives

$$\frac{h_{old}}{h_{new}} \leq 1+\delta\,.$$

Looking back at the grid on the unit sphere that we used to define Ψ, we see that we have $\frac{6\cdot\pi}{\theta}$ grid lines for longitude and $\frac{6\cdot\pi}{\theta}$ grid lines for latitude. Each grid cell has $\frac{32}{\theta}$ points. Therefore, Ψ contains $O(\frac{1}{\theta^3})$ projection planes. Note that $\lceil \frac{2\pi}{\arccos\frac{1}{1+\delta}} \rceil \leq \frac{1}{\delta} + 6 = O(\frac{1}{\delta})$, and thus $|\Psi| = O(\lceil \frac{1}{\arccos\frac{1}{1+\delta}} \rceil^3) = O(\frac{1}{\delta^3})$. Hence the total time required is $O(\frac{1}{\delta^3} \cdot T(n,\varepsilon))$ and the amount of space used is $O(\frac{1}{\delta^3} \cdot S(n,\varepsilon))$.

It remains to consider the line-segment model. Whenever the closest point to q on the line through $\overline{pr}$ lies between the segment endpoints p and r, the distance from q to the line segment defined by p and r is equal to the distance from q to the line defined by p and r. Hence, in this case the situation is identical to the line model and the theorem holds. If this is not the case, then the actual distance in $\mathbb{R}^3$ is $\min(|\overline{pq}|, |\overline{qr}|)$. Assume without loss of generality that the distance is $|\overline{pq}|$. Now consider the projection of p, q and r onto a given plane $\psi^* \in \Psi$, and define p', q' and r' as in Lemma 8.

Recall that we can always choose $\psi^* \in \Psi$ such that the angle between ℓ and $\overline{qr}$ is at most θ, as in Fig. 4(d). Let x be the point of intersection between ℓ and the line through $\overline{pq}$. Since both x and r are on ℓ, they do not move when projected (i.e. $x' = x, r' = r$). The point q is on the line segment $\overline{px}$, and therefore q' must be on the line segment $\overline{p'x}$. The point p' is closer to $\overline{rx}$ than p, and hence $\angle q'p'r' = \angle xp'r \geq \angle qpr \geq \frac{\pi}{2}$. And because $\angle q'p'r' \geq \frac{\pi}{2}$, the distance between q and $\overline{pr}$ in the plane ψ^* will be $|\overline{p'q'}|$. By Lemma 7 we know that $|\overline{p'q'}| \geq |\overline{pq}| \cos\theta$, and thus by (8) $\frac{|\overline{pq}|}{|\overline{p'q'}|} \leq 1 + \delta$.

Thus there is always a plane $\psi^* \in \Psi$ such that the line-segment distance between q and $\overline{pr}$ in $\mathbb{R}^3$ is at most $(1+\delta)$ times the line-segment distance of $proj(q, \psi^*)$ from $\overline{proj(p, \psi^*)proj(r, \psi^*)}$.

Apart from the difference in the 2-dimensional algorithm, the 3-dimensional algorithm for the line-segment model is identical to that for the line model, and hence it runs in $O(\frac{1}{\delta^3} \cdot T(n, \varepsilon))$ time and uses $O(\frac{1}{\delta^3} \cdot S(n, \varepsilon))$ space. $\square$

In the next section we shall show how to compute a Douglas-Peucker simplification efficiently in 2-dimensional space. Combining the discussion above with the results to be proved (see Theorems 19 and 20), we have

Theorem 10 *Given two real numbers $\varepsilon, \delta > 0$ and a path in $\mathbb{R}^3$, a $(1+\delta)$-approximation of the Douglas-Peucker simplification can be computed in the line model in $O(\frac{1}{\delta^3} n \log^2 n)$ time using $O(\frac{1}{\delta^3} n)$ space, and in the line-segment model in $O(\frac{1}{\delta^3} n \log^3 n)$ time using $O(\frac{1}{\delta^3} n \log n)$ space.*

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

To complete the algorithm described in the previous section, we here give a fast implementation of the Douglas-Peucker algorithm in two dimensions for a path that 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. As mentioned in the introduction, Hershberger and Snoeyink [15] gave an $O(n \log^* n)$-time algorithm working in the line model when the path does not self-intersect. Their approach heavily relies on the fact that the path does not self-intersect since a lot of 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.

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 [14,15]). For simplicity, we will throughout this section assume that no three points lie on a line.

Lemma 11 *Given a set S of $n \geq 3$ points and a line segment ℓ, the maximum distance between a point in S and ℓ is realised by a vertex p on the convex hull of S.*

PROOF. Consider a point $p \in S$ that realises the maximum distance between S and ℓ. We need to prove that p lies on the convex hull of S. For simplicity we assume that ℓ is horizontal. We will have two cases:

(a) If p has a perpendicular projection onto ℓ, then consider the line ℓ' through p and parallel to ℓ, see Fig. 5(a). If p lies on an edge e of the convex hull, then it is easily seen that one can move p to one of the endpoints of the edge such that the distance does not decrease. If p does not lie on the convex hull, then there must be a point p' of S that lies above ℓ'; otherwise p would be on the convex hull. However, since p' lies above ℓ', it follows that $dist(p, \ell) < dist(p', \ell)$, which is a contradiction and thus p must lie on the convex hull.

(b) If p does not have a perpendicular projection onto ℓ, then let q be the endpoint of ℓ closest to p, as illustrated in Fig. 5(b). Consider the line ℓ' through p and orthogonal to the line through q and p. Assume that p does not lie on the convex hull. Then there must be a point p' of S that lies on the opposite side of ℓ' compared to ℓ; otherwise p would be on the convex hull. However, this implies that $dist(p, \ell) < dist(p', \ell)$, which is a contradiction and thus p must lie on the convex hull. $\square$

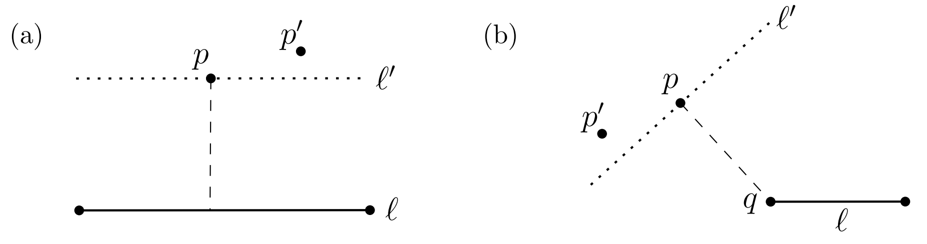


Fig. 5. Illustrating the proof of Lemma 11.

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

Problem 12 *(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 13 *(Half-plane furthest-point queries (HPFP-queries)) Preprocess 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 14 *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.*

PROOF. As a first step, compute the convex hull of S in $O(n \log n)$ time by sorting the points in angular order [11]. Assume that $\ell = \overline{pq}$ is horizontal and that p lies to the left of q, see Fig. 6(a). Partition the points of the convex hull into four sets; the set S_T of points on the convex hull above ℓ and with a perpendicular projection onto ℓ, the set S_B of points on the convex hull below ℓ and with a perpendicular projection onto ℓ, the set S_L of points on the convex hull whose nearest point on ℓ is the left endpoint of ℓ, and finally, the set S_R of points on the convex hull whose nearest point on ℓ is the right endpoint of ℓ. Partitioning the points of the convex hull of S into sets S_T, S_B, S_L and S_R with respect to ℓ can be carried out by computing the intersection points, if any, between the convex hull and a directed line.

The point in S_T with the largest distance to ℓ is the point with the largest y-coordinate. One can find this point by performing a binary search along the pieces containing S_T [20]. Symmetrically, the same can be done with the set S_B. It remains to find the points in S_L and S_R that are furthest from p and q, respectively.

Consider the set S_L. The points in S_L lie to the left of the vertical line through p directed from bottom to top, while the remaining points of S lie to the right of that line. Thus the point in S_L furthest from ℓ can be obtained by performing an HPFP-query with the point p and the directed line determined by ℓ and p on S_L. Symmetrically, the same can be done with S_R. This concludes the proof of the lemma. $\square$

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 (e.g. using Graham's algorithm [11]). 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 (e.g. using Andrew's

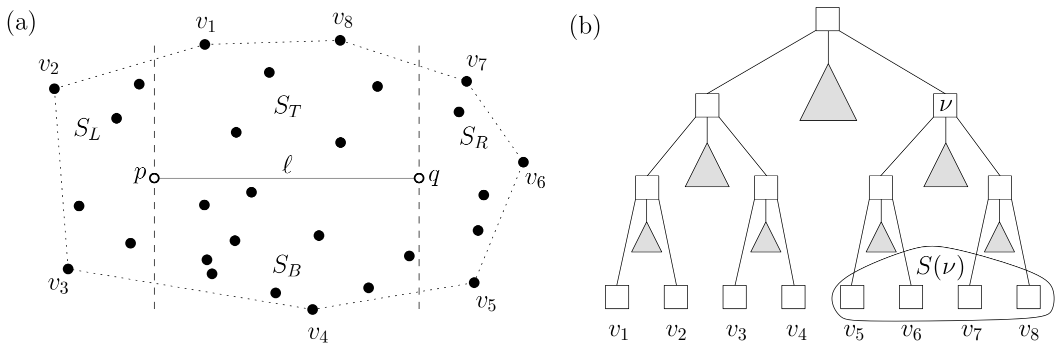


Fig. 6. Illustrations for the proofs of Lemmas 14 and 17.

algorithm [3]). Unfortunately this improvement will not affect the overall time complexity of the Douglas-Peucker algorithm.

4.1 Half-plane furthest-point queries

The HPFP-query problem was first studied by Aronov et al. [4] and they showed the following two results:

Fact 15 *([4, Corollary 5]) There is a data structure that requires $O(n^{1+\beta})$ space and preprocessing time, and supports HPFP-queries in $O(2^{\frac{1}{\beta}} \log n)$ time on n points in convex position, for any real number $\beta > 0$.*

Fact 16 *([4, Corollary 11]) There is a data structure that requires $O(n \log^3 n)$ space and polynomial preprocessing time, and that supports HPFP-queries in $O(\log n)$ time on n points in convex position.*

As an alternative, 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 17 *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.*

PROOF. To simplify the description, we consider S to be a simple path with n vertices ordered along the convex hull. We build a balanced binary tree $\mathcal{T}$ above the vertices listed in counter-clockwise order along the convex hull, as illustrated in Fig. 6(b). The subset of points stored in the leaves of a subtree rooted at a node ν is called the *canonical subset* of ν and is denoted by $S(\nu)$. For any internal or leaf node ν we store a furthest-point Voronoi diagram (FPVD) of the canonical subset, together with a point-location structure (PLS) that allows for fast point-location queries in the FPVD. We claim (*) that both the FPVD and the PLS can be built in $O(|S(\nu)|)$ time using $O(|S(\nu)|)$ space such that, given a query point p, the point in $S(\nu)$ furthest from p can be found in $O(\log|S(\nu)|)$ time.

To prove the claim (*), consider a set S of n points in convex position in the plane. The FPVD of S, denoted $\mathcal{F}(S)$, can be computed in $O(n)$ time using $O(n)$ space when the input is a convex polygon [2]. Note that $\mathcal{F}(S)$ forms a tree that partitions the plane into convex unbounded faces. Next, construct a bounding rectangle B that contains S and all the vertices in $\mathcal{F}(S)$. Let $\mathcal{D}(S)$ denote the subdivision obtained from the union of B and $\mathcal{F}(S)$, where all the edges of $\mathcal{F}(S)$ outside B have been removed, see Fig. 7. Each face in $\mathcal{D}(S)$ is a convex face. Edelsbrunner et al. [9] showed that for any subdivision where the faces are y-monotone a PLS can be built in $O(n)$ time using $O(n)$ space such that queries can be answered in $O(\log n)$ time. A convex face is also y-monotone, thus the claim holds. Since the FPVD and the PLS can be constructed in linear time for each node in $\mathcal{T}$, the total time to build $\mathcal{T}$ is $O(n\log n)$, and it requires $O(n\log n)$ space.

Given a query point p and a directed line ℓ, an HPFP-query is performed as follows. If ℓ does not intersect the convex hull of S, then we simply report the point furthest from p in S. Otherwise, let p_l and p_r be the two intersection points between ℓ and the convex hull of S and assume that p_l lies above p_r along ℓ. For simplicity, we assume that the first point of S in the order along the convex hull does not lie to the left of ℓ; if it does, we simply have to perform two queries instead of one and choose the point among the two reported points that is furthest from ℓ. The two points p_l and p_r can be found in $O(\log n)$ time by binary search along the convex hull of S [20]. Search with p_l and p_r in $\mathcal{T}$ until we get a node ν_{split}, where the search path splits. Without loss of generality, assume that p_l (respectively p_r) lies in the left (right) subtree of ν_{split}. Since the points stored in the leaves of $\mathcal{T}$ are ordered along the convex hull, the search paths from the root to p_l and p_r are well defined.

From the left child of ν_{split}, we continue the search with p_l, and at every node ν where the search path of p_l goes left, we perform a furthest-point query with p in $\mathcal{F}(S_R(\nu))$, where $S_R(\nu)$ is the point set stored in the right child of ν. As described above, this can be performed in $O(\log|S_R(\nu)|)$ time. Similarly, we continue the search with p_r at the right child of ν_{split}, and at every node ν where the search path of p_r goes to the right we perform a furthest-point query with p in $\mathcal{F}(S_L(\nu))$, where $S_L(\nu)$ is the point set stored in the left child of ν. In effect, we performed $O(\log n)$ queries on a collection of $O(\log n)$ subtrees that together contain exactly the points along the convex hull of S between p_l and p_r. The point furthest from ℓ reported from the $O(\log n)$ queries is reported to have the largest distance to ℓ among all the points on the convex hull between (p_l, p_r) (i.e. to the left of ℓ).

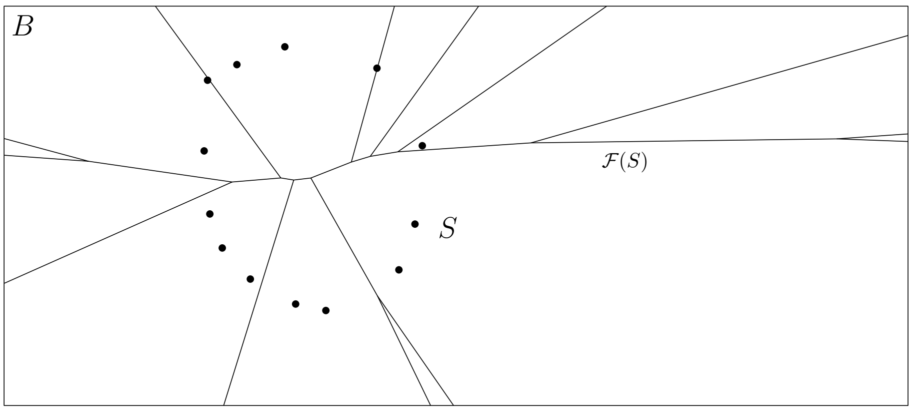


Fig. 7. Illustrating the subdivision $\mathcal{D}(S)$ obtained by clipping $\mathcal{F}(S)$ to a bounding rectangle B.

Since we perform $O(\log n)$ queries, and each query requires $O(\log n)$ time, the total query time is $O(\log^2 n)$ time. As a result Lemma 17 follows. □

4.2 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 18 *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.

PROOF. Consider a tree structure containing the points of P. Let $P(\nu)$ denote the canonical subset of points held in the leaves of a subtree rooted at a node ν. We build a two-level data structure $\mathcal{C}$, where the main tree of $\mathcal{C}$ is a balanced binary search tree $\mathcal{T}$ built on the order of the points in P. For any internal or leaf node ν in $\mathcal{T}$, the canonical subset $P(\nu)$ is stored in an LSFP-structure including the convex hull $\mathcal{H}(\nu)$ of $P(\nu)$ (as well as other data structures needed to support HPFP-queries).

A distance query is performed as follows. Search with v_i and v_j in $\mathcal{C}$ until we get a node ν_{split}, where the search path splits. From the left child of ν_{split} we continue the search with v_i, and at every node ν where the search path of v_i goes left, we perform an LSFP-query with $\mathcal{H}(\nu_R)$ and the segment (v_i, v_j), where ν_R is the right child of ν. Similarly, we continue the search with v_j at the right child of ν_{split}, and at every node ν where the search path of v_j goes to the right we perform an LSFP-query with $\mathcal{H}(\nu_L)$ and the segment (v_i, v_j), where ν_L is the left child of ν. In effect, we performed $O(\log n)$ queries on a collection of $O(\log n)$ subtrees that together contain exactly the points along P between v_i and v_j. The largest value reported from the $O(\log n)$ LSFP-queries is reported as the largest distance between (v_i, v_j) and a point along the path from v_i to v_j.

Let $F'(n)$ be the preprocessing time needed to construct a data structure of size $S'(n)$ that can answer LSFP-queries in $Q'(n)$ time. By Lemma 14, for the whole data structure the preprocessing time is

$$O(n \log n) + \sum_{i=0}^{\log n} 2^i \cdot F'(\frac{n}{2^i}) = O(n \log^2 n) + \sum_{i=0}^{\log n} 2^{i+1} F(\frac{n}{2^i}),$$

the query time is

$$O(\log n) + \sum_{i=1}^{\log n} Q'(\frac{n}{2^i}) = O(\log^2 n) + 2 \sum_{i=0}^{\log n} Q(\frac{n}{2^i})$$

and the amount of space used is

$$O(n) + \sum_{i=0}^{\log n} 2^i \cdot S'(\frac{n}{2^i}) = O(n \log n) + \sum_{i=0}^{\log n} 2^i S(\frac{n}{2^i}).$$

This concludes the proof of the lemma. $\square$

Theorem 19 *(Line-segment model) For a polygonal path P with n vertices in the plane, our adaptation of the Douglas-Peucker algorithm runs in time $O(n \log^3 n)$ and uses $O(n \log n)$ space.*

PROOF. The standard Douglas-Peucker algorithm iterates over at most n line segments. Thus, by combining Lemmas 14, 17 and 18 we obtain the time bound but the space used will be $O(n \log^2 n)$.

The reason for the large space bound is that all the associated structures are constructed during the preprocessing step and, hence, uses $\sum_{i=0}^{\log n} 2^i S(\frac{n}{2^i}) = O(n \log^2 n)$ space. However, if we only maintain the associated structures that are needed, then we can improve the space complexity by a logarithmic factor.

Assume that the first-level structure of $\mathcal{C}$ has been built together with the associated data structures that are needed next. We say that the nodes containing associated structures are *active*, and all others are *passive*. Active nodes are the only nodes that store associated data structures.

Given a query with parameters $(P = \langle v_i, \ldots, v_k \rangle, \varepsilon)$ we do the following recursively:

(1) Compute the furthest point between the polygonal path and the line ℓ determined by v_i and v_k. Let v_j be this point. Note that during this query only associated data structures of active nodes were queried (by assumption).
(2) If the distance between the 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 the recursion.
(3) Otherwise, split the polygonal path into two subpaths $\langle v_i, v_{i+1}, \ldots, v_j \rangle$ and $\langle v_j, v_{j+1}, \ldots, v_k \rangle$, as shown in Fig. 8(a). The active nodes queried are displayed hollowed. Consider the path Γ from ν_{split} to v_j in $\mathcal{C}$. We perform two operations (see Fig. 8(b)): (1) The status of every active node in Γ is changed from active to passive and their associated data structures are deleted. (2) Every node, not in Γ, with its parent in Γ changes status from passive to active and the associated data structure for it is constructed.
(4) Call the recursive routine for both subpaths. Note that only the associated data structures belonging to the active nodes will be queried at this level.

Consider the data structure $\mathcal{C}$ at a fixed time in the execution of the algorithm. Note that an active node cannot have an ancestor that is active. This implies that at any given point in time the total number of points held in the active nodes is at most n. From this it follows that the space required by the associated data structures gets never higher than $O(n \log n)$. $\square$

Note that in Lemma 18 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.

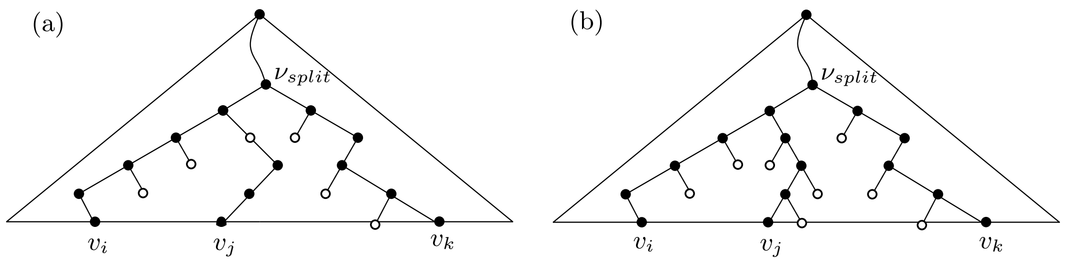


Fig. 8. Illustrating how the active nodes (hollowed) are maintained during the execution.

4.3 Path simplification in the line model

Even if Theorem 19 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 [20] 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 $\mathcal{P} = \{P_1, P_2, \dots, P_h\}$. 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, $h \le \lceil \log n \rceil$. 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. 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 $\langle v_i, v_{i+1}, \dots, v_k \rangle$ 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) Otherwise, split the polygonal path $\langle v_i, v_{i+1}, \ldots, v_k \rangle$ into two subpaths $\langle v_i, \ldots, v_j \rangle$ and $\langle v_j, \ldots, v_k \rangle$. 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, its corresponding sorted set and convex hull and itself are discarded.

(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 in 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 20 *(Line model) For a polygonal path P with n vertices in the plane, our adaptation of the Douglas-Peucker algorithm runs in $O(n \log^2 n)$ time and uses $O(n)$ space.*

5 Concluding remarks

Our main results are (i) that the five types of queries proposed in [6] can be approximated in a sound way, and (ii) in the plane, both in the line-segment model and in the line model, a Douglas-Peucker path simplification can be computed in subquadratic time in the case the input path is allowed to self-intersect. Also, we showed that trajectory compression can be carried out efficiently by transforming the input trajectory to 3-dimensional space, then computing a 3-dimensional path simplification of the transformed trajectory, and finally by projecting the simplified path back to a 2-dimensional trajectory.

In our practical experiments the Douglas-Peucker heuristics runs well even in 3-dimensional space. None of the inputs used in our experiments triggered the heuristics to use quadratic time. Therefore, we did not find it necessary to implement any of the theoretical algorithms developed in this paper. Actually, instead of implementing our 3-dimensional algorithm, one could consider implementing even simpler heuristics. For instance, if one finds that some point on the polygonal path under consideration is farther away from the line or the line segment determined by the endpoints of the path than the given tolerance, one could just split the path down the middle and continue recursively with these about equal-sized subpaths. For an input of size n, such an algorithm would never take more than $O(n \log n)$ time, and compression power can still be reasonable (but the output may not be visually as pleasing as that computed by the Douglas-Peucker heuristics).

As to our theoretical results, one can ask whether the running time of any of our algorithms could be improved. Since our trajectory compression uses the 2-dimensional Douglas-Peucker heuristics, it would be natural to focus on the 2-dimensional case. We conjecture that in the line model a Douglas-Peucker path simplification can be computed in $O(n \log n)$ time and $O(n)$ space for a polygonal path of size n. We leave it for the reader to prove or disprove this conjecture. Yet another open problem is to extend our approach for 3-dimensional trajectories (e.g. if the moving entities are birds).

Acknowledgements

We would like to thank Adel Ahmed for useful discussions.

References

[1] P. K. Agarwal and K. R. Varadarajan. Efficient algorithms for approximating polygonal chains. *Discrete & Computational Geometry*, 23(2):273–291, 2000.

[2] A. Aggarwal, L. J. Guibas, J. B. Saxe, and P. W. Shor. A linear-time algorithm for computing the Voronoi diagram of a convex polygon. *Discrete and Computational Geometry*, 4(1):591–604, 1989.

[3] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.

[4] 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.

[5] 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.

[6] H. Cao, O. Wolfson, and G. Trajcevski. Spatio-temporal data reduction with deterministic error bounds. *The VLDB Journal*, 15(3):211–228, 2006.

[7] 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.

[8] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer*, 10(2):112–122, 1973.

[9] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15(2):317–340, 1986.

[10] A. U. Frank. Socio-economic units: Their life and motion. In *Life and Motion of Socio-Economic Units*, volume 8 of *GISDATA*, pages 21–34. Taylor & Francis, 2001.

[11] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133, 1972.

[12] J. Gudmundsson, P. Laube, and T. Wolle. Movement patterns in spatio-temporal data. In *Encyclopedia of GIS*, pages 726–732. Springer-Verlag, 2008.

[13] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A foundation for representing and querying moving objects. *ACM Transactions on Database Systems*, 25(1):1–42, 2000.

[14] 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.

[15] 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.

[16] I. A. R. Hulbert. GPS and its use in animal telemetry: The next five years. In *Proceedings of the Conference on Tracking Animals with GPS*, pages 51–60. Macaulay Institute, 2001.

[17] 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.

[18] 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.

[19] A. Melkman and J. O'Rourke. On polygonal chain approximation. In *Computational Morphology*, pages 87–95. North-Holland, 1988.

[20] M. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23(2):166–204, 1981.

[21] F. Porikli. Trajectory distance metric using Hidden Markov Model based representation. In *Proceedings of the 6th International Workshop on Performance Evaluation of Tracking and Surveillance*. IEEE, 2004.