Institute of Formal Methods in Computer Science

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor Thesis

Fully-polynomial-time Approximation Schemes for the Euclidean Shortest Path Problem

Axel Schneewind

Course of Study:	Informatik B. Sc.
Examiner:	Prof. Dr. Stefan Funke
Supervisor:	Prof. Dr. Stefan Funke
Commenced:	November 15, 2023
Completed:	May 16, 2024

Abstract

The shortest path problem is a well-studied problem in computer-science. For transport networks, there exist natural graph representations and highly efficient algorithms that can compute shortest paths on millions of nodes within milliseconds. In contrast, computing shortest paths in space (e.g. in $\mathbb{R}^2$) poses some challenges.

Shortest path computations in space have applications in robotics, naval routing or video games. As shortest paths in a continuum are hard to compute (with the Euclidean shortest-path problem in 3D even proven to be NP-hard), approximations can be necessary to obtain acceptable runtimes.

In this thesis, an approximation scheme is studied that guarantees solutions with cost at most $(1 + \varepsilon)$ times the optimum. It uses a triangulation of the domain and, given ε, construct a discretization. By performing a Dijkstra search, one can then approximate shortest paths with the given quality guarantee. This scheme is implemented and its practicality evaluated on larger instances.

Contents

1 Introduction

Computation of shortest paths is a well-studied field in computer science. Network-constrained shortest paths in particular have gotten a lot of attention from researchers, that resulted in highly optimized algorithms that are applicable to navigation in road networks or public transport. Geometric shortest path computations, where the domain is not restricted to a discrete graph have gotten less attention, despite having applications in e.g. robotics, geo-information systems, naval routing or video games. The Euclidean shortest path problem, which is studied in this thesis, is about finding an optimal path in multidimensional space (e.g. $\mathbb{R}^2$), restricted by polygons that define obstacles.

1.1 Problem statement

This work is mainly based on an algorithm presented for the shortest path on polyhedral surface problem (SPPS) , which is applied to the Euclidean shortest path problem (ESPP). In the following, these problems will be defined as well as the concept of polynomial-time approximation schemes.

1.1.1 Shortest path on polyhedral surface problem

Given the surface $\mathcal{P}$ of a polyhedron (consisting of non-convex faces), nonzero weights w_i for its faces $f_i \in F_\mathcal{P}$ and two vertices $s, t \in V_\mathcal{P}$, find the path Π^* from s to t on $\mathcal{P}$ with minimal cost. The cost is defined as the weighted sum of Euclidean distances. [AMS00]

1.1.2 Euclidean shortest path problem (2-dimensional)

Given a subset $S \subseteq \mathbb{R}^2$ (i.e. the domain) and two vertices $s, t \in V_\mathcal{P}$, find the path Π^* from s to t that remains on S and has minimal euclidean length.

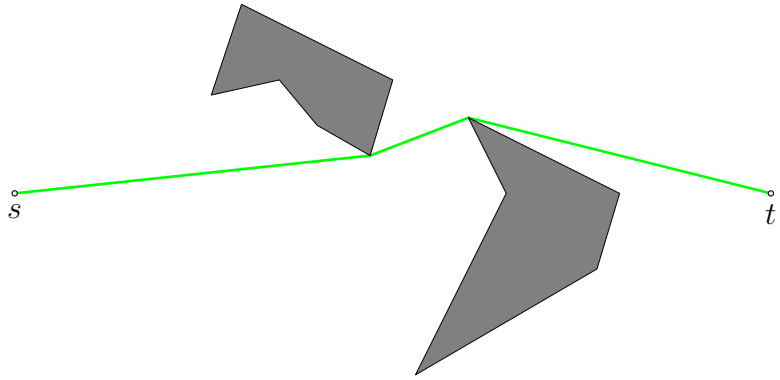


Figure 1.1: example of a shortest path avoiding obstacles

S can be defined as a polyhedral surface $\mathcal{P}$. In this thesis, s, t are restricted to lie on edges of $\mathcal{P}$. Vertices of $\mathcal{P}$ that are at the boundary between S and $\mathbb{R}^2 \setminus S$ will be referred to as *obstacle* or *boundary* vertices.

1.1.3 (Fully) Polynomial-time approximation scheme

For problems with high time complexity, it is sometimes desired to obtain lower runtimes in exchange for lower solution quality. A polynomial-time approximation scheme [ACG+12] is a method that finds a solution where the cost is bounded by a constant factor $(1 + \varepsilon)$ times the optimum with polynomial time complexity in the problem size. Fully-polynomial-time approximation schemes are also required to have time complexity polynomial in $1/\varepsilon$.

1.2 Related work

Various contributions have already been made to the field, notably exact approaches using visibility graphs or wavefront propagations. Approximation schemes pose an alternative where 'good enough' solutions are found with potentially less computational effort.

A visibility graph can be constructed in $\mathcal{O}(n \log n + E)$ time, where E is the number of edges of the visibility graph [GM91]. Such a graph can then be used to compute shortest paths in $\mathcal{O}(n \log n + E)$. However, the upper bound of $E \in \Theta(n^2)$ causes algorithms searching such a graph to have similar worst-case runtimes. For this upper bound, consider the inside of a convex polyhedron with n vertices: The visibility graph for such an instance is the complete graph K_n, which has $\frac{n(n-1)}{2} \in \Theta(n^2)$ edges.

A different class of algorithms relies on the continuous Dijkstra method, which works similarly to wavefront propagation over time [HS99]: At time t, a wavefront contains all vertices with distance to s at most t. The shape of the wavefront is defined by a set of wavelets, i.e. a circular arc with radius $t - t'$ around a vertex with distance t' in the wavefront. These arcs change based on wavefront-wavefront- and wavefront-obstacle-collisions.

This method can be used for constructing a *shortest path map*, i.e. a subdivision of free space such that all points in a cell have the same sequence of vertices on their path to a starting point s. Hence, this type of algorithm solves a more general problem than visibility-based algorithms, which only provide information on a discrete set of vertices.

A theoretically optimal algorithm with $\mathcal{O}(n \log n)$ runtime based on the continuous dijkstra has been found [HS99]. However, no robust implementation of this approach is currently known.

A third class of relevant algorithms is the class of approximation algorithms, such as the one this thesis is based on [AMS00]. This approximation scheme is described for the general shortest path on polyhedral surface problem and approximates solutions of quality $(1+\varepsilon)$ times the optimum in $\mathcal{O}\left(\frac{n}{\varepsilon}\log\left(\frac{1}{\varepsilon}\right)\left(\frac{1}{\sqrt{\varepsilon}}+\log(n)\right)\right)$ time. On an unweighted surface, the algorithm runs in $\mathcal{O}\left(\frac{n}{\varepsilon}\log\left(\frac{1}{\varepsilon}\right)\log(n)\right)$. The scheme derives a discretization graph from a polyhedral surface (such as a planar subdivision), which guarantees $(1+\varepsilon)$-approximation of shortest paths. This graph can then be used to perform normal dijkstra searches, which potentially allows for similar optimizations as a network-constrained search.

The approximation scheme can also be applied to three-dimensional domains, in which the shortest path problem is NP-hard [MS04].

This thesis aims to provide an implementation of this approximation scheme, adapted to the two-dimensional Euclidean shortest path problem, and evaluate its practicality.

2 Theory

In this chapter, the theory behind the problem and the approximation scheme of [AMS00], is explained.

2.1 Euclidean shortest path problem

As this work is based on an approximation scheme for the weighted shortest path on polyhedral surface problem, it is important to examine the relation between these problems.

First, an instance of the Euclidean shortest path problem can be stated as an instance of the SPPS problem, given a planar subdivision $\mathcal{P}$ of a subset of $\mathbb{R}^2$ (represented by an undirected planar graph). However, there are certain properties of Euclidean shortest paths in $\mathbb{R}^2$, which allow for simpler/more efficient algorithms than in the case of the SPPS problem:

2.1.1 Bending

In a weighted domain, shortest paths can be subject to bending at any point between differently weighted regions, comparable to the phenomenon in optics. One can easily find an instance where the shortest path between two points bends.

In the example in Figure 2.1, the path vpv' with cost $|vp|w_1 + |pv'|w_2 \leq 101\frac{\sqrt{2}}{2}$ is cheaper than the non-bending path vv' with cost $w_1|vv'| = 100$ (assuming the sides of the square have length 1).

In contrast, for Euclidean shortest paths, bending can only occur at (convex) boundary vertices. Intuitively, this can be seen as follows: Consider the path vpv' between v and v' where $v, p, v' \in \mathbb{R}^2$ are three points in 2D-space. Assume that p is not a boundary vertex, then p can be shifted towards the line vv' without vpv' intersecting an obstacle. The new path vpv' is shorter and thus cheaper than the old one. Hence, a path where bending occurs at a non-boundary point, cannot be optimal.

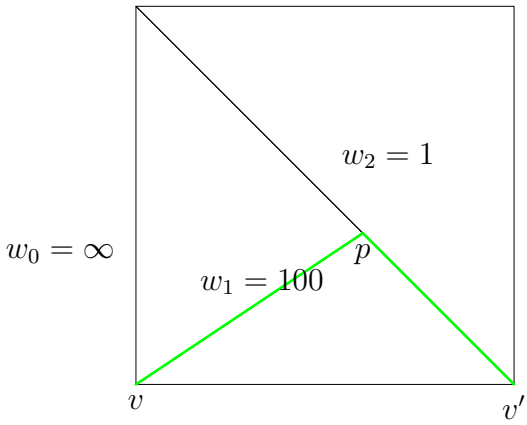


Figure 2.1: a bending path vpv' between v and v'

In consequence, a shortest path from s to t can be defined as $\Pi^*(s,t) = sv_1 \dots v_k t$, where all v_i are obstacle vertices. When considering paths on the faces of a planar subdivision $\mathcal{P}$, such a path can contain additional non-bending points p_j on $E_\mathcal{P}$ between the bending points v_i (when the path crosses different faces).

2.1.2 Edge-using segments in shortest paths

As shown in Figure 2.1, an optimal path on a weighted planar subdivision can partially use an edge between two regions.

This is not the case for unweighted Euclidean shortest paths: An optimal path $\Pi^*(v,v')$, where v, v' do not lie on e, either uses the entire edge, intersects it in a single point, or does not intersect it at all. If a shortest path used an edge e partially, this would directly imply bending at a non-boundary point.

2.1.3 Consequences for shortest-path computations

The above properties allow for shortest-path computations based on a visibility graph, i.e. a graph $G_V = (V_V, E_V)$ where V_V is the set of boundary vertices and $E_V = \{(u,v) |$ segment uv does not intersect obstacles$\}$. Optimal paths between boundary vertices then consist entirely of edges in E_V.

Alternatively, an algorithm finding shortest paths in weighted polyhedral surfaces can be optimized by not checking paths with bending in free space. This is the approach followed in this work.

2.2 General approach of the approximation scheme

The authors of [AMS00] provide a way to discretize a weighted polyhedral surface $\mathcal{P}$, which can also be applied to planar subdivisions (i.e. planar graphs $(V_\mathcal{P}, E_\mathcal{P})$) on subsets of $\mathbb{R}^2$. The presented approach places Steiner points on the edges in $E_\mathcal{P}$, resulting in a Graph $G_\varepsilon = (V_\varepsilon, E_\varepsilon)$. These Steiner points are connected as follows: $\{p_i, p_{i+1}\} \in E_\varepsilon$ for neighboring Steiner points on the same edge, $\{v, p_j\} \in E_\varepsilon$ for each vertex in $V_\mathcal{P}$ neighboring a Steiner point, $\{v, p_j\} \in E_\varepsilon$ for each vertex in $V_\mathcal{P}$ and a Steiner point p_j that can be reached by crossing a face, and $\{p_i, q_j\} \in E_\varepsilon$ for any pair of Steiner points that can be connected by crossing a face.

[AMS00] do not require the face-crossing segments between vertices and Steiner points. these are included in this thesis to allow for approximations closer to optimal paths (where bending occurs exactly in boundary vertices, which is not the case for the weighted problem). These segments do not change the scaling of the graph sizes and quality guarantees.

These points have to be placed in a way that ensures a given quality constraint on shortest paths approximated using G_ε. Intuitively, when considering two points a and b on the edges that border the same face, it must be possible to approximate the segment ab by snapping to the closest Steiner points p_1, p_2 such that $|p_1, p_2| \leq (1 + \varepsilon)|ab|$. As $|ab|$ gets smaller when a and b lie close to the same vertex, Steiner points have to be placed closer around vertices. To require only finitely many points, some region around a vertex has to be ignored when inserting points.

The authors of [AMS00] formalize this by the following definitions:

Definition 2.2.1 (Discrete Path)
a discrete path $\tilde{\Pi}(v_1, v_2)$ *between nodes* $v_1, v_2 \in V_\mathcal{P}$ *is a path consisting only of nodes* $\tilde{v} \in V_\varepsilon$. *The shortest discrete path is the shortest of all paths* $\tilde{\Pi}(v_1, v_2)$ *and denoted by* $\tilde{\Pi}^*(v_1, v_2)$.

Definition 2.2.2 (ε-short path)
a discrete path $\tilde{\Pi}(v_1, v_2)$ *is called* ε*-short if cost*$(\tilde{\Pi}(v_1, v_2)) \leq (1 + \varepsilon)|\Pi^*(v_1, v_2)|$ *for the corresponding shortest path* $\Pi^*(v_1, v_2)$.

Formally, the construction of G_ε has to ensure that for any shortest path Π^* starting and ending on edges of $\mathcal{P}$, there exists a discrete path $\tilde{\Pi}^*$ in G_ε with cost at most $(1 + \varepsilon) \cdot |\Pi^*|$.

The following section (section 2.3) describes the construction of G_ε. Then, the quality guarantee of paths in G_ε is explained in section 2.4. For the stated time complexity, a method of pruning is necessary, which is described in section 2.7.

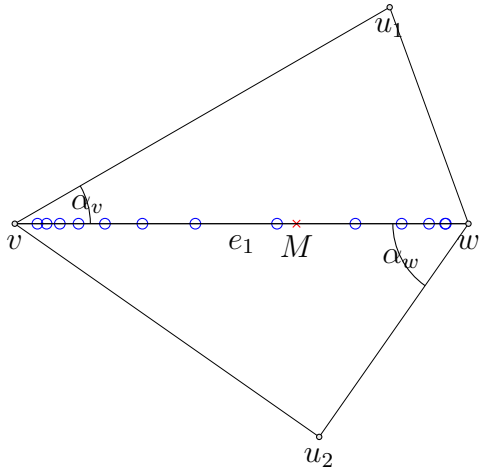


Figure 2.2: e with Steiner points determined by the minimal angles α_v, α_w

2.3 Discretization based on a planar subdivision

[AMS00] provides a way to construct G_ε which will be explained in this section. As they consider the shortest-path problem with weighted faces (which are not required to be triangles), their definitions are more general than required here. Thus, some definitions will be omitted or simplified.

Definition 2.3.1 (distances to edges)
For any point p on an edge $e \in E_\mathcal{P}$, let

$$d(p) = \min_{e' \in E_\mathcal{P} \setminus E_\mathcal{P}(p)} |pe'|$$

where $|pe'|$ is the distance from point p to the edge e' and $E_\mathcal{P}(p)$ is the set of edges incident to p. For an edge $e \in E_\mathcal{P}$, let $d(e) = \sup_{p \in e} d(p)$. [AMS00]

For an edge $e = (v, w)$ of a triangulation, this distance function can be defined over the interval $[0, 1]$ such that for a point $p = v + x \cdot (w - v)$, the equality $d(x) = \frac{d(p)}{|vw|}$ holds: Let α_v, α_w be the minimal angles between e and other edges at v and w, then:

$$d : [0, 1] \to \mathbb{R} : x \mapsto \min\{\sin(\alpha_v) \cdot x, \sin(\alpha_w) \cdot (1 - x)\} \tag{2.1}$$

An angle larger than $\pi/2$ has to be treated as $\pi/2$ because $d(p)$ is defined by the distance to the respective endpoint in that case.

Definition 2.3.2 (vertex-vicinity)
[AMS00] define a radius for a vertex $v \in V_\mathcal{P}$ (in the unweighted case) as follows:

$$r(v) = \frac{d(v)}{5}$$

where $d(v)$ is the minimal distance between v and an edge reachable by crossing a face incident to v.

The vertex-vicinity $S(v)$ of v is the polygon of the points at distance $\varepsilon r(v)$ on the incident edges in $E_{\mathcal{P}}$.

The relative value $\frac{r(v)}{|vw|}$ will be referred to as r_{vw} in the following.

Definition 2.3.3 (Middle point M of an edge e)
[AMS00] define M as a point with $d(M) = d(e)$.

While M is not necessarily defined uniquely for general polyhedral surfaces, it is for triangulations. The position of that point M with maximal distance to other edges can be expressed in terms of α_v, α_w: The relative position $m_{vw} = \frac{|vM|}{|vw|}$ between v and w is

$$m_{vw} = \frac{1}{1 + \frac{\sin(\alpha_v)}{\sin(\alpha_w)}} \tag{2.2}$$

This can be found by calculating the intersection m_{vw} of $\sin(\alpha_v) \cdot x$ and $\sin(\alpha_w) \cdot (1 - x)$, which has to be the maximum of $d(x)$:

$$\begin{aligned}
\sin(\alpha_w) \cdot (1 - m_{vw}) &= \sin(\alpha_v) \cdot m_{vw} \\
\frac{1 - m_{vw}}{m_{vw}} &= \frac{\sin(\alpha_v)}{\sin(\alpha_w)} \\
\frac{1}{m_{vw}} - 1 &= \frac{\sin(\alpha_v)}{\sin(\alpha_w)} \\
\frac{1}{m_{vw}} &= \frac{\sin(\alpha_v)}{\sin(\alpha_w)} + 1 \\
m_{vw} &= \frac{1}{1 + \frac{\sin(\alpha_v)}{\sin(\alpha_w)}}
\end{aligned}$$

2.3.1 Placement of Steiner points

On each edge $e = (v, w) \in E_{\mathcal{P}}$, with mid point M, the points $p_1, \ldots, p_k$ are inserted into the graph G_ε. To fullfill the $(1+\epsilon)$-criterion, they need to be placed such that the following holds [AMS00]:

$$\begin{aligned} |vp_1| &= \varepsilon \cdot r(v), & \\ |p_{i-1}p_i| &= \varepsilon \cdot d(p_{i-1}) & \forall 1 < i < j, \\ p_j &= M, & \\ |p_i p_{i+1}| &= \varepsilon \cdot d(p_{i+1}) & \forall j < i < k, \\ |p_k w| &= \varepsilon \cdot r(w) & \end{aligned}$$

With that, the relative position $x_i = \frac{|vp_i|}{|vw|}$ of a Steiner point p_i on e can be defined non-recursively as

$$x_i = \begin{cases} (1 + \varepsilon \sin(\alpha_v))^{i-1} \cdot \varepsilon r_{vw}, \text{ if } 1 \leq i < j \\ m_{vw}, \text{ if } i = j \\ 1 - (1 + \varepsilon \sin(\alpha_w))^{k-i} \cdot \varepsilon r_{wv}, \text{ if } j < i \leq k \end{cases} \tag{2.3}$$

[AMS00] also use such definitions to estimate $|vp_i|$ in their proofs. Using this explicit formula, the coordinates of a Steiner point can be computed in constant time. This definition is equivalent to the one above, which can be shown inductively as follows:

Proof 2.3.1

$$\begin{aligned} x_1 &= (1 + \varepsilon \sin(\alpha_v))^0 \cdot \varepsilon r_{vw} = \frac{\varepsilon r(v)}{|vw|} = \frac{|vp_1|}{|vw|} \\ x_i &= (1 + \varepsilon \sin(\alpha_v))^{i-1} \cdot \varepsilon r_{vw} \\ &= (1 + \varepsilon \sin(\alpha_v)) \cdot (1 + \varepsilon \sin(\alpha_v))^{i-2} \cdot \varepsilon r_{vw} \\ &= (1 + \varepsilon \sin(\alpha_v)) x_{i-1} \\ &= x_{i-1} + \varepsilon d(x_{i-1}) = \frac{|vs_{i-1}| + \varepsilon d(p_{i-1})}{|vw|} = \frac{|vs_{i-1}| + |s_{i-1}s_i|}{|vw|} = \frac{|vs_i|}{|vw|} \end{aligned}$$

The case $j < i$ works analogously.

2.3.2 Mapping points to Steiner intervals

With the computation of Steiner point coordinates defined, the inverse is described here: Given a point $q = v + x \cdot (w - v)$ (i.e. lying on edge (v, w)), the index i has to be found

such that q lies between the Steiner points p_i and p_{i+1} on e (i.e. on the Steiner interval (p_i, p_{i+1})). This can be done as follows:

Assume that q lies between v and M, then

$$i := \max\left\{\left\lfloor \log_{1+\varepsilon\sin(\alpha_v)}\left(\frac{x}{\varepsilon r_{vw}}\right)\right\rfloor + 1, 0\right\}$$

Similarly, for q between M and w, then

$$i := \min\left\{\left\lfloor k - \log_{1+\varepsilon\sin(\alpha_w)}\left(\frac{1-x}{\varepsilon r_{wv}}\right)\right\rfloor, k\right\}$$

This index can be computed in constant time. $i = 0$ corresponds to points q that lie between v and p_1, and $i = k$ to points between p_k and w.

The first equation can be obtained by solving $x = (1 + \varepsilon\sin(\alpha_v))^{i-1} \cdot \varepsilon r_{vw}$ for i and flooring. The second one works analogously.

2.4 Approximation quality

In this section, the quality of paths on G_ε is examined. By construction, the following hold [AMS00]:

Lemma 2.4.1
Let s be a face crossing segment that goes through the interval (p_i, p_{i+1}), then

$$|p_i p_{i+1}| \leq \varepsilon |s|$$

Lemma 2.4.2
Let $s = (a, b)$ be a face crossing segment where b lies on the interval (p_i, p_{i+1}), then

$$\angle p_i a p_{i+1} \leq \frac{\pi}{2}\varepsilon$$

2.4.1 Quality of face-crossing segments

First, the local quality of discrete segments needs to be shown, i.e. that for a single face-crossing segment, there exists a ε-short segment in G_ε. Three cases have to be considered:

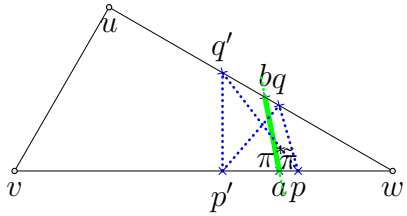


Figure 2.3: path between points on triangle edges

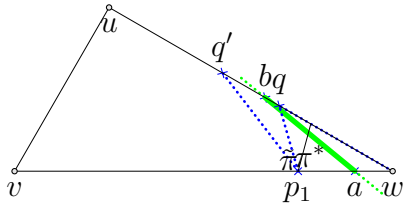


Figure 2.4: path between points on triangle edges

1. a face crossing segment ab, where a lies between the neighboring Steiner points p, p' and b between q, q'. Figure 2.3 shows the segment ab and the neighboring discrete counterparts $pq, pq', p'q, p'q'$. [AMS00] show that

$$\max\{\min\{|pq|, |pq'|\}, \min\{|p'q|, |p'q'|\}\} \leq (1+\varepsilon)|ab|$$

   which means that for both p and p', an ε-short segment can be found.

2. a face crossing segment ab, where a lies between p_1 and w in the vertex vicinity of w and b between the Steiner points q, q' (see Figure 2.4). For this case, [AMS00] show that

$$\max\{|p_1q|, |p_1q'|\} \leq (1+\varepsilon)|ab| + \varepsilon r(w)$$

3. a face crossing segment ab, where a lies on the Steiner interval (p_1, w) and b on (q_1, v) (see Figure 2.5). For this case, [AMS00] show that

$$|p_1q_1| \leq |ab| + \varepsilon(r(v) + r(w))$$

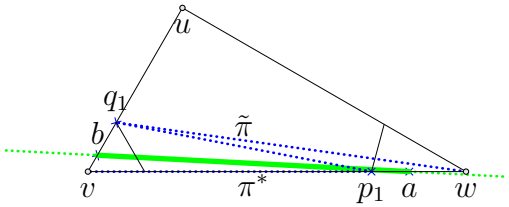


Figure 2.5: path between points on triangle edges

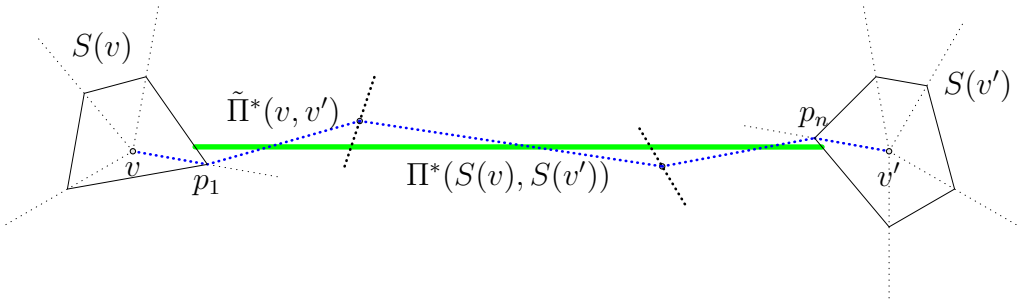


Figure 2.6: discrete and continuous paths between two vertex vicinities

Case 1 can also be extended to sequences of face-crossing segments: For each sequence $s_1, \ldots, s_l$ of face crossing segments that do not intersect vertex-vicinities, there exists a sequence $\tilde{s}_1, \ldots \tilde{s}_l$ of discrete segments such that $\sum |\tilde{s}_i| \leq (1 + \varepsilon) \sum |s_i|$.

[AMS00] define a segment ab and a discrete segment pq as *neighboring*, if a lies on the same edge as p with no other Steiner point in between (the same has to hold for b). Similarly, two sequences of segments $s_1, \ldots, s_n$ and $\tilde{s}_1, \ldots, \tilde{s}_n$ are called *neighboring* if each segment s_i neighbors its corresponding discrete segment $\tilde{s}_i$.

2.4.2 Quality of approximated paths

For entire paths Π [AMS00] show an upper bound of $(1 + 3\varepsilon)$, (higher due to the $r(v)$ terms in case 2 and case 3 and due to edge-using segments). In the unweighted case, a slightly better upper bound can be shown. The proof from [AMS00] is briefly stated in the following, adapted to the fact that an optimal path does not partially use edges:

Consider the shortest path $\Pi^*(S(v), S(v'))$ that joins two arbitrary points in the vertex vicinities $S(v), S(v')$ and the shortest neighboring discrete path $\tilde{\Pi}(v, v') = vp_1 \ldots p_n v'$, where $p_1 \ldots p_n$ are Steiner points. p_1 is part of the vertex vicinity of v and p_n of v'.

For the path $p_1 \ldots p_n$, the inequality $|p_1 \ldots p_n| \leq (1 + \varepsilon)|\Pi^*(p_1, p_n)|$ directly follows from case 1: With $|\tilde{\Pi}(p_1, p_n)| = \sum_i |\tilde{s}_i|$, the upper bound holds as $|\tilde{s}_i| \leq (1 + \varepsilon)|s_i|$ for each discrete segment $\tilde{s}_i$ and its continuous counterpart s_i.

This does not hold for the weighted case: There, the optimal path $\Pi^*(p_1, p_n)$ can also use edges partially. For an edge-using discrete segment $\tilde{s}_j$, [AMS00] bound the cost by $|\tilde{s}_j| \leq |s_j| + \varepsilon|s_{j-1}|$, where s_{j-1} is the preceding face-crossing segment. As a result, when summing up all costs, this s_{j-1} can be counted twice (once for $\tilde{s}_j$ and once for $\tilde{s}_{j-1}$) and the quality guarantee for $p_1 \ldots p_n$ can only be shown to be $1 + 2\varepsilon$.

The segments p_1p_2 and $p_{n-1}p_n$ fall into case 3 (or case 2 if v, v' are part of the same triangle). Additionally, the segments vp_1 and p_nv' contribute $\varepsilon(r(v) + r(v'))$ to the distance. With that combined, one gets

$$|\tilde{\Pi}(v, v')| \leq (1 + \varepsilon)|\Pi^*(S(v), S(v'))| + 2\varepsilon(r(v) + r(v'))$$

To get the final quality guarantee, the term $\varepsilon(r(v) + r(v'))$ has to be bounded: [AMS00] state that $r(v) + r(v') \leq \frac{1}{2}|\Pi^*(S(v), S(v'))|$. Therefore

$$|\tilde{\Pi}(v, v')| \leq (1 + 2\varepsilon)|\Pi^*(S(v), S(v'))|$$

As an optimal path can be represented by a sequence of such paths $\Pi^*(s, t) = s\Pi^*(S(s), S(v_1))\Pi^*(S(v_1), S(v_2)) \dots \Pi^*(S(v_n), S(t))t$ the statement from above also applies to an entire discrete path $\tilde{\Pi} = \tilde{\Pi}(s, v_1)\tilde{\Pi}(v_1, v_2) \dots \tilde{\Pi}^*(v_n, t)$:

$$|\tilde{\Pi}^*(s, t)| \leq (1 + 2\varepsilon)|\Pi^*(s, t)|$$

2.4.3 Maximal bending angle of discrete paths at Steiner points

While in a weighted domain, large bending angles can occur at Steiner points, this does not hold in the unweighted case. For the weighted domain, [AMS00] prove bounds on such angles, depending on the involved face weights using Snell's law. For an unweighted domain, such a bound can be found as follows:

Let a, b be arbitrary points on the edges reachable from edge e, where ab intersects e between the Steiner points p and p'. By 2.4.2, the angle $\angle(ap, ap')$ between segments ap and ap' is at most $\frac{\pi}{2}\varepsilon$. Therefore, also the angles $\angle(ap, ab)$ and $\angle(ap', ab)$ are smaller. Symmetrically, this holds for $\angle(bp, ba)$ and $\angle(bp', ba)$ too.

For $p^* \in \{p, p'\}$, consider the triangle ap^*b. As seen above, the inner angles $\angle bap^*, \angle p^*ba$ are below $\frac{\pi}{2}\varepsilon$. Therefore, $\angle ap^*b \geq \pi - 2\frac{\pi}{2}\varepsilon$ and the bending angle between ap^* and p^*b can be bounded by

$$\pi - (\pi - 2\frac{\pi}{2}\varepsilon) = \pi\varepsilon$$

As a result, when given ap^*, one only has to search the outgoing segments with bending angles below $\pi\varepsilon$ to find one required for approximating ab. Further, when given both ap and ap', one can find paths to both Steiner points neighboring b.

2.5 Size of the discretization

[AMS00] provide an upper bound for the number of points inserted on an edge of a polyhedral surface: For an edge $e = (v, w)$, they define a value $C(e) < D(e) := 4(|e|/d(e)) \log(|e|/\sqrt{r(v)r(w)})$ which is constant for a fixed graph instance, and the maximal number of points inserted as $3 + C(e)\frac{1}{\varepsilon} \log \frac{2}{\varepsilon}$.

They also show that the maximal number of Steiner points inserted into the whole surface $\mathcal{P}$ is bounded by $\tilde{C}(\mathcal{P})\frac{|E_\mathcal{P}|}{\varepsilon} \log \frac{2}{\varepsilon}$, where $\tilde{C}(\mathcal{P}) \leq 3 + \frac{1}{|E_\mathcal{P}|} \sum_{e \in E_\mathcal{P}} C(e)$.

With that, the degree of a Steiner point on edge e can be bounded by

$$4 + \sum_{e' \in E_\mathcal{P} \text{visible from } e} \left(3 + C(e')\frac{1}{\varepsilon} \log \frac{2}{\varepsilon}\right) \in \mathcal{O}(\frac{1}{\varepsilon} \log \frac{1}{\varepsilon})$$

where the 4 counts the neighboring Steiner points and the vertices in $V_\mathcal{P}$, that can be reached by crossing a face.

Therefore, the size of the graph scales as follows:

$$|V_\varepsilon| \in \mathcal{O}\left(\frac{n}{\varepsilon} \cdot \log \frac{1}{\varepsilon}\right)$$
$$|E_\varepsilon| \in \mathcal{O}\left(\frac{n}{\varepsilon^2} \cdot \left(\log \frac{1}{\varepsilon}\right)^2\right)$$

As the performance of shortest-path computations depends on the size of G_ε and as the values $C(e)$ can grow arbitrarily large (depending on $\mathcal{P}$), further estimates on the size of V_ε are provided here.

2.5.1 Number of Steiner points depending on angles

Assuming that $\alpha_v = \alpha_{min}$ for the minimal angle α_v at v, the given upper bound $D(e)$ scales with α_{min} as follows:

$$D(e) \geq \frac{2}{\sin(\alpha_{min})} \log\left(\frac{5^2}{\sin(\alpha_{min})}\right)$$

and as a result

$$D(e) \in \Omega\left(\frac{1}{\sin(\alpha_{min})} \log\left(\frac{1}{\sin(\alpha_{min})}\right)\right)$$
$$\subseteq \Omega\left(\frac{1}{\alpha_{min}} \log\left(\frac{1}{\alpha_{min}}\right)\right)$$

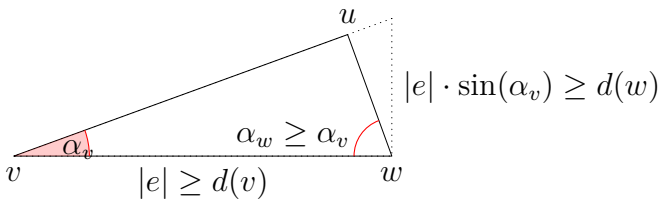


Figure 2.7: triangle with a small angle α_v

Proof 2.5.1
By definition in Equation 2.1 we have

$$\frac{d(e)}{|e|} \leq \sin(\alpha_{min})$$

(equivalently $\frac{|e|}{d(e)} \geq \frac{1}{\sin(\alpha_{min})}$*). Recall that* $r(v) = d(v)/5$*. Assuming* $\alpha_{min} = \alpha_v$ *gets arbitrarily small, we have* $r(v) \leq |e|$ *and* $d(w) \leq |e| \sin(\alpha_{min})$ *(as depicted in Figure 2.7)*

$$\begin{aligned}
\frac{\sqrt{r(v)r(w)}}{|e|} &= \frac{\sqrt{\frac{d(v)\cdot d(w)}{5^2}}}{|e|} \\
&\leq \frac{\sqrt{\frac{d(v)\cdot \sin(\alpha_{min})|e|}{5^2}}}{|e|} \\
&\leq \frac{\sqrt{\frac{|e|\cdot \sin(\alpha_{min})|e|}{5^2}}}{|e|} \\
&\leq \sqrt{\frac{\sin(\alpha_{min})}{5^2}}
\end{aligned}$$

resulting in $\log\left(\frac{|e|}{\sqrt{r(v)r(w)}}\right) \geq \frac{1}{2}\log\left(\frac{5^2}{\sin(\alpha_{min})}\right)$.

It is not known however, if the number of points actually scales with $\Theta(D(e))$.

From the calculations above, an upper bound $B(t)$ on the number of face-crossing segments on a triangle $t \in G_\varepsilon$ with minimal inner angle α can be derived. Consider the two sides e, e' of t that enclose α. The numbers k, k' of Steiner points on these edges each can be bounded by $D(e) \approx D(e')$ computed from α. As the set of Steiner points inserted on e and e' forms the full bipartite graph $K_{k,k'}$, it contributes up to $2 \cdot D(e) \cdot D(e')$ face-crossing segments to $|E_\varepsilon|$. Therefore, the upper bound from above has to be squared:

$$B(t) \in \Theta\left(\left(\frac{1}{\sin(\alpha)}\right)^2 \log\left(\frac{1}{\sin(\alpha)}\right)^2\right)$$

2.5.2 Number of Steiner points depending on vertex radii

The number of points inserted also depends on the relative node radius values r_{vw}, which can be low even for triangles with large inner angles: This can occur e.g. for an equilateral triangle where an edge borders an arbitrarily skinny triangle.

The worst-case scaling of k in the minimal r_{vw} on an edge can be estimated by:

$$\begin{aligned} k &\geq \log_{1+\varepsilon \sin(\alpha_v)}(m_{vw}/r_{vw}) \\ &\in \Theta\left(\log\left(\frac{1}{r_{vw}}\right)\right) \end{aligned}$$

As a result, the number of Steiner points scales with this r_{vw}, although not as strongly as in the minimal angles.

2.6 Properties of planar subdivisions

For the size of the discretization graph and the quality of solutions, the following properties of planar subdivisions are important.

2.6.1 Quality of paths on a delaunay-triangulation

[Xia11] showed that the stretch factor for delaunay triangulations is less than 1.998. Equivalently, when using a triangulation without any additional points inserted, shortest paths can be approximated with cost of at most 1.998 times the optimum. In other words, a $(1 + 0.998)$-approximation is guaranteed without any additional computations.

2.6.2 Refining of triangulations

A Delaunay-triangulation has the property that it maximizes the minimal inner angles of its triangles. However, this minimal angle can still be small depending on the set of points triangulated.

As seen in section 2.5, the number of points inserted scales with the minimal inner angles of $\mathcal{P}$. For this reason, it is desirable to have a planar subdivision that ensures some lower bound on inner angles.

This can be achieved by starting with a triangulation of the boundary vertices and adding additional vertices such that all triangles have sufficiently large inner angles.

2.6.3 Inner angles of planar subdivisions

Through the inner angles of the faces, the structure of the underlying planar subdivision strongly influences the number of Steiner points that have to be added. As seen in section 2.5, it is desirable to have a subdivision where all inner angles are at least $\pi/2$.

In general, for faces with f sides, the sum of inner angles is $(f-2)\pi$ and as a result, the minimal inner angle cannot be larger than $(f-2)\pi/f$. For a triangle, the sum of inner angles is π, so the smallest of those three angles cannot be larger than $\pi/3$ (i.e. $60°$). For faces with $f \geq 4$ sides, the upper bound on the minimal inner angle is $(f-2)\pi/f \geq \pi/2$.

For this reason, planar subdivisions that consist of faces with at least 4 sides can result in smaller graph sizes and might be more well-suited for this approximation scheme.

2.7 Pruned djikstra search

Unlike planar graphs (e.g. road networks), the described Graph G_ε has the property that a node's degree is not bounded by some small constant but grows depending on ε (and the values $C(e)$). [AMS00] provide a way of reducing the number of edges that are checked when performing a search, while still fulfilling 7ε-correctness.

Definition 2.7.1 (Geodesic path)
A geodesic path $\Pi = v_1 v_2 \ldots$ is a path that is locally optimal i.e. changing a single node v_i cannot improve the quality.

Intuitively, this approach reduces the number of outgoing segments for a node $p \in V_\varepsilon$, by using information about the already found discrete path $\tilde{\Pi}(a_0, p)$. In particular, it is known that geodesic path only bends at boundary vertices. For this reason, most of the outgoing segments of a Steiner point p cannot approximate a geodesic path as the resulting discrete path would bend too much.

Definition 2.7.2 (Cone)
For a segment ab, the cone $Cone(a, b, \theta)$ is defined as the set of segments $s \in E_\varepsilon$, that are incident to b and where the angle between s and ab is at most θ.

Definition 2.7.3 (ε-spanner)
A ε-spanner $Cone_\varepsilon(a, b, \theta)$ is defined by a partitioning of $Cone(a, b, \theta)$ into nonempty subcones c_i that have angles not exceeding $\frac{\pi\varepsilon}{2}$. $Cone_\varepsilon(a, b, \theta)$ consists of the shortest edge e_i of each c_i. This set contains at most $\left\lceil \frac{2\theta}{\pi\varepsilon} \right\rceil$ elements. [AMS00]

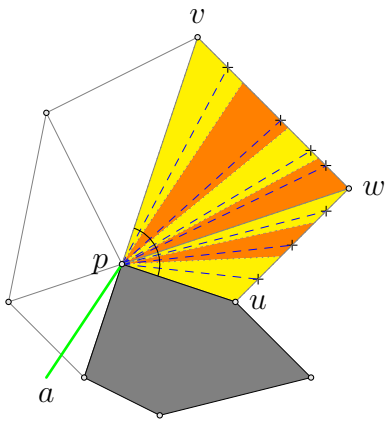


Figure 2.8: an ε-spanner at bending point p over the points on edges (v, w), (w, u), with subcones highlighted in yellow and orange

[AMS00] show an important property of ε-spanners: For any segment $s = (p, q) \in Cone(a, p, \theta)$, there exists a segment $\tilde{s} = (p, q') \in Cone_\varepsilon(a, p, \theta)$ such that

$$|pq'| \leq |pq| \text{ and } |pq| \leq |pq'| + |q'q| \leq (1 + \frac{\pi\varepsilon}{2})|pq| \tag{2.4}$$

In other words, searching ε-spanners instead of all neighbors reduces the solution quality by at most $(1 + \frac{\pi\varepsilon}{2})$. They define a class of paths $\mathcal{C}^*(a_0)$, that (in the unweighted case) contains all 7ε-short paths starting at a_0:

Definition 2.7.4 ($\mathcal{C}^*(a_0)$)
$\mathcal{C}^*(a_0)$ contains each e incident to a_0.
A path $\tilde{\Pi} = a_0 \dots a_{i-1}a_i$ is contained in $\mathcal{C}^*(a_0)$ if $a_0 \dots a_{i-1} \in \mathcal{C}^*(a_0)$ and the last segment (a_{i-1}, a_i) is part of the class $Cone^*(a_0 \dots a_{i-1})$. [AMS00]

It remains to define $Cone^*(\tilde{\Pi})$ for $\tilde{\Pi} = a_0a_1 \dots a_{i-1}a_i$. As [AMS00] define it for the weighted problem, a simplified definition is used here:

1. $a_i \in V_\mathcal{P}$ and a_i is a *boundary* vertex. Then $Cone^*(\tilde{\Pi})$ consists of the ε-spanner $Cone_\varepsilon(a_{i-1}, a_i, \pi)$ and a segment to each steiner point of $S(a_i)$.
2. $a_i \in V_\mathcal{P}$ and a_i is not a *boundary* vertex. Let a_j be the last boundary vertex (or a_0 if none) on $\tilde{\Pi}$. Then $Cone^*(\tilde{\Pi})$ contains the shortest segments from the subcones left and right of the ray a_ja_i. It also contains a segment to each steiner point of $S(a_i)$.
3. $a_i \in V_\varepsilon \setminus V_\mathcal{P}$. Let a_j be the last boundary vertex (or a_0 if none) on $\tilde{\Pi}$. Then $Cone^*(\tilde{\Pi})$ contains the shortest segments from the subcones left and right of the ray a_ja_i.

For the weighted SPPS problem, case 3 requires an ε-spanner over the maximum bending angle of a discrete geodesic path at any Steiner point. As this angle is $\frac{\pi}{\varepsilon}$ (see

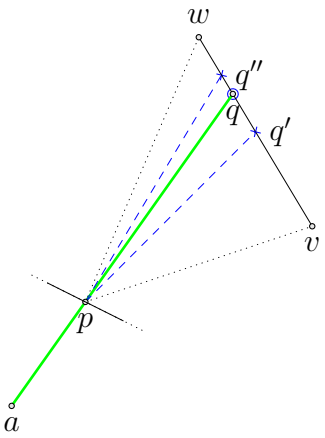


Figure 2.9: The path segment apq over a non-bending-point where q lies on edge (v, w)

subsection 2.4.3) in the unweighted case, such an ε-spanner consists of only one subcone to the left and one to the right of the ray from a_j to a_i.

Another difference is that for the weighted SPPS problem, omnidirectional ε-spanners are generated not for boundary vertices but instead for any Steiner point that neighbors a vertex of $\mathcal{P}$. Simplifying this by only generating ε-spanners for boundary vertices reduces the number of ε-spanners, which then scales with the number of boundary vertices instead of their degree. These ε-spanners still fullfill the property described above (Equation 2.4).

Further, for the weighted SPPS problem, each case requires searching another cone defined by preceding edge-using segments (if any). In other words, if a point a_l is part of the discrete path to a_i and these points are connected by edge-using segments, an ϵ-spanner with the direction of $a_l a_i$ is also generated. As for the ESPP, the direction of the approximated path is already defined by the direction of the last bending point a_j to a_i, this can be omitted here.

2.7.1 Selecting neighbors from subcones

Note that the shortest segment pq in a subcone is always the one that is 'most orthogonal' to the target edge, i.e. the angle between pq and vw is closest to $\pi/2$. This can result in a high bending angle at p (e.g. if the direction $a_j p$ and vw are almost parallel). Thus, selecting q such that the bending angle at p is minimal might lead to better results. This method is not proven to guarantee a bound on solution quality. However, this method can be implemented more efficiently and is therefore included in the implementation.

2.7.2 Time complexity of the pruned search

The size of the set $Cone^*(\tilde{\Pi})$ to check for the different cases 1, 2 and 3 can be bounded by:

1. $\left|Cone^*(\tilde{\Pi})\right| \leq \deg(a_i) + \left\lceil \frac{1}{\varepsilon} \right\rceil \in \mathcal{O}(\varepsilon^{-1})$
2. $\left|Cone^*(\tilde{\Pi})\right| \leq \deg(a_i) + 2 \in \mathcal{O}(1)$, amortized over all vertices
3. $\left|Cone^*(\tilde{\Pi})\right| \leq 4 \in \mathcal{O}(1)$

For a boundary vertex v (1), an outgoing segment is generated for each edge in $E_{\mathcal{P}}$ that is incident to v. For a boundary vertex v (2), an outgoing segment is generated for each edge in $E_{\mathcal{P}}$ that is incident to v, and two for the Steiner interval intersected by the current search direction. For a Steiner point p 3, two outgoing segments are generated for the Steiner interval intersected by the current search direction and up to two for the neighboring Steiner points on the same edge as p.

The total number of edges to check when performing a one-to-all search is

$$\begin{aligned}\sum_{a \in V_\varepsilon} |Cone^*(a)| &\leq n \cdot \left(\max_{v \in V_{\mathcal{P}}} \deg(v) + \left\lceil \frac{1}{\varepsilon} \right\rceil\right) + 4 \cdot \frac{m}{\varepsilon} \log\left(\frac{2}{\varepsilon}\right) \\ &\in \mathcal{O}\left(\frac{n}{\varepsilon} \log\left(\frac{1}{\varepsilon}\right)\right)\end{aligned}$$

as also stated by [AMS00]. In the weighted domain, the number of outgoing segments of Steiner points (case 3) is not constant but in $\mathcal{O}(\varepsilon^{-1/2})$, resulting in $\mathcal{O}(\frac{n}{\varepsilon^{3/2}} \log(\frac{2}{\varepsilon}))$ edges.

In general, the complexity of a dijkstra search can be described by $\Theta(|E| + |V| \cdot \log|V|)$ using the theoretically optimal Fibonacci heap as a priority queue.

For the weighted case, the complexity of the pruned search can be analyzed using $|V| = |V_\varepsilon| \in \mathcal{O}(\frac{n}{\varepsilon} \log \frac{1}{\varepsilon})$ as estimated in section 2.5 and $|E| \in \mathcal{O}(\frac{n}{\varepsilon^{\frac{3}{2}}} \log \frac{1}{\varepsilon})$ as given above. This results in a theoretical complexity of [AMS00]

$$\begin{aligned}&\mathcal{O}\left(\frac{n}{\varepsilon^{\frac{3}{2}}} \log\left(\frac{1}{\varepsilon}\right) + \frac{n}{\varepsilon} \log\left(\frac{1}{\varepsilon}\right) \log\left(\frac{n}{\varepsilon} \log\left(\frac{1}{\varepsilon}\right)\right)\right) \\ \subseteq &\mathcal{O}\left(\frac{n}{\varepsilon} \log\left(\frac{1}{\varepsilon}\right) \cdot \left(\frac{1}{\sqrt{\varepsilon}} + \log\left(\frac{n}{\varepsilon} \log\left(\frac{1}{\varepsilon}\right)\right)\right)\right) \\ \subseteq &\mathcal{O}\left(\frac{n}{\varepsilon} \log \frac{1}{\varepsilon} \left(\frac{1}{\sqrt{\varepsilon}} + \log n\right)\right)\end{aligned}$$

The last simplification step is due to

$$\frac{n}{\varepsilon}\log\left(\frac{1}{\varepsilon}\right)\log\left(\frac{1}{\sqrt{\varepsilon}}+\frac{n}{\varepsilon}\log\left(\frac{1}{\varepsilon}\right)\right)=\frac{n}{\varepsilon}\log\left(\frac{1}{\varepsilon}\right)\left(\log(n)+\frac{1}{\sqrt{\varepsilon}}+\log\left(\frac{1}{\varepsilon}\right)+\log\log\left(\frac{1}{\varepsilon}\right)\right)$$

where the $\log(\frac{1}{\varepsilon})$-terms are dominated by $\frac{1}{\sqrt{\varepsilon}}$.

[AMS00] states (without proof) that for the unweighted case, this simplifies to $\mathcal{O}\left(\frac{n}{\varepsilon}\log\frac{1}{\varepsilon}\log n\right)$. It is not clear why that is the case, as the calculation above yields $\mathcal{O}\left(\frac{n}{\varepsilon}\log\frac{1}{\varepsilon}\left(1+\log\left(\frac{n}{\varepsilon}\log(\frac{1}{\varepsilon})\right)\right)\right)$, where (by $\log\log\frac{1}{\varepsilon}$ being dominated by $\log\frac{n}{\varepsilon}$) one would expect a resulting complexity of

$$\mathcal{O}\left(\frac{n}{\varepsilon}\log\left(\frac{1}{\varepsilon}\right)\log\left(\frac{n}{\varepsilon}\right)\right)$$

In both weighted and unweighted domains, the complexity without pruning would be defined by the full set of edges E_ε (see section 2.5), as also stated by [AMS00]:

$$\begin{aligned}
&\mathcal{O}\left(\frac{n}{\varepsilon}\log\frac{1}{\varepsilon}\log\left(\frac{n}{\varepsilon}\log\frac{1}{\varepsilon}\right)+\frac{n}{\varepsilon^2}\log\left(\frac{1}{\varepsilon}\right)^2\right)\\
\subseteq&\mathcal{O}\left(\frac{n}{\varepsilon}\log\frac{1}{\varepsilon}\left(\log\left(\frac{n}{\varepsilon}\log\frac{1}{\varepsilon}\right)+\frac{1}{\varepsilon}\log\left(\frac{1}{\varepsilon}\right)\right)\right)\\
\subseteq&\mathcal{O}\left(\frac{n}{\varepsilon}\log\frac{1}{\varepsilon}\left(\log(n)+\log\left(\frac{1}{\varepsilon}\right)+\log\log\left(\frac{1}{\varepsilon}\right)+\frac{1}{\varepsilon}\log\left(\frac{1}{\varepsilon}\right)\right)\right)\\
\subseteq&\mathcal{O}\left(\frac{n}{\varepsilon}\log\frac{1}{\varepsilon}\left(\log(n)+\frac{1}{\varepsilon}\log\left(\frac{1}{\varepsilon}\right)\right)\right)
\end{aligned}$$

2.7.3 Quality of the pruned search

The pruned search does not find the shortest of all possible paths, but only the shortest in the subset $\mathcal{C}^*(a_0)$. Consequently, the quality guarantee for the pruned search gets worse. [AMS00] provide a quality guarantee of $1+15\varepsilon$ on paths found by the pruned search in the weighted case.

From Equation 2.4, it follows that the shortest path in $\mathcal{C}^*(a_0)$ to any node a is at most $1+\pi\varepsilon/2$ times as long as the shortest of all paths from a_0 to a.

With the improved quality of approximated paths in the unweighted case, the quality guarantee with pruning also improves. The proof that [AMS00] provides is stated below with the updated quality guarantee.

Let a_0 be the source node and $\tilde{\Pi}^*(a_0, a)$ be the shortest discrete path to a found by the pruned search (assuming $\varepsilon \leq 1$). Let $\mathcal{C}(a_0)$ be the set of all paths starting at a_0:

$$\begin{aligned}
|\tilde{\Pi}^*(a_0, a)| &= \min\{|\Pi| : \Pi \in \mathcal{C}^*(a_o)\} \\
&\leq (1 + \varepsilon\pi/2) \min\{|\Pi| : \Pi \in \mathcal{C}(a_o)\} \\
&\leq (1 + \varepsilon\pi/2) \cdot (1 + 2\varepsilon)|\Pi^*(a_0, a)| \\
&\leq (1 + \varepsilon\pi/2 + 2\varepsilon + \pi\varepsilon^2)|\Pi^*(a_0, a)| \\
&\leq (1 + 7\varepsilon)|\Pi^*(a_0, a)|
\end{aligned}$$

The quality guarantee thus improves to $1 + 7\varepsilon$. For a given ε, one can also calculate the factor with ε^2 included, to get a slightly better guarantee.

3 Implementation

The described algorithms and data structures are implemented in C++20 and provided at [Sch24]. The computations are implemented using only constructs of the standard library.

3.1 Implicit graph representation

The described discretization requires a representation by a suitable data structure. In the following, a data structure with memory complexity of $\mathcal{O}(|V_\mathcal{P}| + |E_\mathcal{P}|)$ is presented, which is independent of the actual size of the graph G_ε. Note that an explicitly stored graph would require $\Theta(|V_\varepsilon| + |E_\varepsilon|)$ space.

The data structure consists of the triangulation $\mathcal{P}$, and information on Steiner point placement for each edge. Information such as coordinates, sets of outgoing segments and their lengths is computed on demand. A node $p \in V_\varepsilon$ can be identified by (e, i) where e is the edge p lies on and i the index of p on that edge.

For shortest-path computations, efficient handling of the following queries is necessary:

- the coordinates for $v \in V_\varepsilon$
- the length for $e \in E_\varepsilon$
- the set of edges $e \in E_\varepsilon$ where e is incident to a given $v \in V_\varepsilon$. This requires:
  - for a boundary vertex, the set of edges that can be reached by crossing a face
  - for an edge $e \in E_\mathcal{P}$, the set of edges that can be reached by crossing a face

The presented implementation provides $\mathcal{O}(1)$ access to the first two queries and computes the sets of outgoing segments in $\mathcal{O}(l)$ (where l is the number of outgoing segments).

3.1.1 Datastructure for the triangulation

$\mathcal{P}$ is represented by two directed edge lists E_1, E_2, such that for an undirected edge $\{v_1, v_2\}$ of $\mathcal{P}$, its directed counterparts (v_1, v_2) and (v_2, v_1) each reside in one of the two edge lists. These edge lists are implemented by an offset array (mapping nodes to the index of their first outward edge) and the actual list of edges with an edge (v_1, v_2) being represented by the entry (v_2). An undirected edge $e = \{v_1, v_2\}$ is identified by its index in E_1.

The information on how to place Steiner points is stored in a separate list. The entry for an edge $e \in \mathcal{P}$ is stored at the same index as its entry in E_1, removing the need for an additional mapping.

Mapping edges to reachable edges

As shortest path computations require access to the edges that can be reached by crossing a face, a tuple (e_j, e_k, e_l, e_m) is stored for each edge e, where the edges e, e_j, e_k and e, e_l, e_m form the two triangles adjacent to e. All neighbors of a Steiner point p on e lie on those four edges.

Alternatively, one could store the edges for each face and the two indices of the adjacent faces for each edge. For planar subdivisions with faces consisting of higher numbers of vertices/edges, this approach could reduce the space demand. For a triangulation, however, it requires storing 4 indices per edge (the two face indices and the indices of e stored for each adjacent face), which does not reduce memory demand. Due to its higher computational overhead, this approach is not followed.

Mapping vertices to reachable edges

Similar information is stored for each vertex in $V_{\mathcal{P}}$: A list of edges that can be reached from a vertex by crossing a face is stored using an offset array.

3.1.2 Steiner point placement

To compute the position of Steiner points $p \in V_\varepsilon \setminus V_{\mathcal{P}}$ as described in Equation 2.3, it suffices to store a few values per edge: For each $e = (v, w)$ the tuple

$$(r_{vw}, r_{wv}, m_{wv}, (1 + \varepsilon \sin(\alpha_v)), (1 + \varepsilon \sin(\alpha_w)), k, m)$$

is stored, where k is the number of Steiner points points and m the number of points between v and M (including v).

The position of any node $p_j \in V_\varepsilon$ (where $p_0 = v, p_{k+1} = w$) then can be computed by

$$p_j = \begin{cases} v & j = 0 \\ v + r_{vw}(1 + \varepsilon \sin(\alpha_v))^j \cdot (w - v) & 1 \leq j < m \\ v + m_{vw} \cdot (w - v) & j = m \\ w + r_{wv}(1 + \varepsilon \sin(\alpha_w))^{j-k-2} \cdot (v - w) & m < j \leq k \\ w & j = k + 1 \end{cases}$$

The total size of the graph can be computed without an explicit representation using

$$|V_\varepsilon| = |V_\mathcal{P}| + \sum_{e \in E_\mathcal{P}} k_e$$

and for the (directed) edges

$$|E_\varepsilon| = \sum_{e \in E_\mathcal{P}} 2 \cdot (k_e + 1) + \sum_{\substack{e, e' \in E_\mathcal{P} \\ e' \text{ visible for } e}} k_e \cdot k'_e + \sum_{\substack{v \in V_\varepsilon \\ e \in E_\varepsilon \text{ visible for } v}} 2k_e$$

counting the edge-using (first sum) and face-crossing segments (second sum for edges connecting Steiner points and third for connecting vertices to Steiner points).

To improve the performance of computing powers, the values $\ln(1 + \varepsilon \sin(\alpha))$ are stored instead of $(1 + \varepsilon \sin(\alpha))$. This allows for a potentially faster power computation using `std::exp` instead of `std::pow`. Analogously, computation of $\log_{1+\varepsilon \sin(\alpha)}$ can be performed using `std::log` and dividing by the stored $\ln$ value.

The $(1 + \varepsilon \sin(\alpha))$-values are represented by the `double` datatype. Note that this representation is wasteful as the value is always in the interval $[1, 2]$, and in consequence, the sign and exponent are the same for almost all possible values. Storing $\varepsilon \sin(\alpha)$ could provide more precision, or allow using smaller datatypes at similar precision. However, to avoid the overhead associated with transforming between the different representations, this approach is not pursued here.

Per default, the number of Steiner points per edge is limited to 2^{31}, due to using `int` values for k and m. If this limit is exceeded for any edge, the implementation reports an error. Where not stated otherwise, this limit is sufficient for graph instances used for the experiments.

Finally, the precomputation of the described values can be performed in $\mathcal{O}(|V_\mathcal{P}|)$. First, the r_{vw}-values can be computed by iterating over all vertices and checking the distances

to the reachable edges which can be looked up as described above. As each edge in $E_{\mathcal{P}}$ is only seen by two vertices, the number of computed distances is in $\mathcal{O}(|E_{\mathcal{P}}|)$. As the ≤ 4 reachable edges and their endpoints can be looked up, computing the angles α_v, α_w for a single edge takes constant time. From these, the values m_{vw} and $(1 + \varepsilon \sin(\alpha))$ can be computed. Computing k, m for a single edge is also possible in $\mathcal{O}(1)$, analogously to subsection 2.3.2 as the required values are already known. In consequence, values for Steiner point placement can be computed in $\mathcal{O}(|E_{\mathcal{P}}|)$ ($= \mathcal{O}(|V_{\mathcal{P}}|)$ due to planarity of $\mathcal{P}$).

Table-based storage of information on Steiner points

An alternative for implementing Steiner point placement (which is not implemented) works as follows: Consider the Steiner points between v, M. Their relative positions x_i only depend on $r_{vw}, (1 + \varepsilon \sin(\alpha_v))$ (see Equation 2.3). The values $(1 + \varepsilon \sin(\alpha_v))^i$ can be stored for classes of values for $\sin(\alpha_v)$. Each edge e is then mapped to the classes of $\sin(\alpha_v), \sin(\alpha_w)$. When computing the coordinates of such a point p_i, one can simply look up the value $(1 + \varepsilon \sin(\alpha_v))^{i-1}$ and multiply it with r_{vw}. The resulting coordinates can be computed via linear interpolation between v and w.

This approach does not provide the exact placement of Steiner points as described in subsection 2.3.1. By computing the relative values using $\sin(\alpha_l)$ for the class of values in $[\sin(\alpha_l), \sin(\alpha_r)]$, the Steiner points are placed at least as close to each other as the placement according to Equation 2.3, making it a valid placement for satisfying $(1 + \varepsilon)$-correctness.

Compared to the semi-explicit representation from above, this approach involves more computations when computing coordinates, but potentially requires less memory. Compared to the fully implicit representation, it involves less computations (in particular, does not require computing powers) and requires more memory.

3.1.3 Semi-explicit representation

As the graph size is mainly dominated by the set of edges, storing these is not feasible for a small ε. Storing the coordinates of Steiner points is less space-demanding. For this reason, the implementation provides the option to keep Steiner point coordinates in an offset-array-based data structure which is filled when reading the graph. The offset-array maps each edge to the index of the first Steiner point p_1 on that edge, such that a point's coordinates can be accessed by a key of the form (e, i). This list of coordinates holds two `double`-values per node of V_ε and an index for each edge in $E_{\mathcal{P}}$, therefore requiring $|V_\varepsilon| \cdot 16B + |E_{\mathcal{P}}| \cdot 4B$ of memory ($\mathcal{O}(|V_\varepsilon| + |E_{\mathcal{P}}|)$).

This removes the overhead of computing the coordinates while performing the search. In consequence, computing the length of a segment only requires a single Euclidean distance computation.

3.2 Dijkstra search

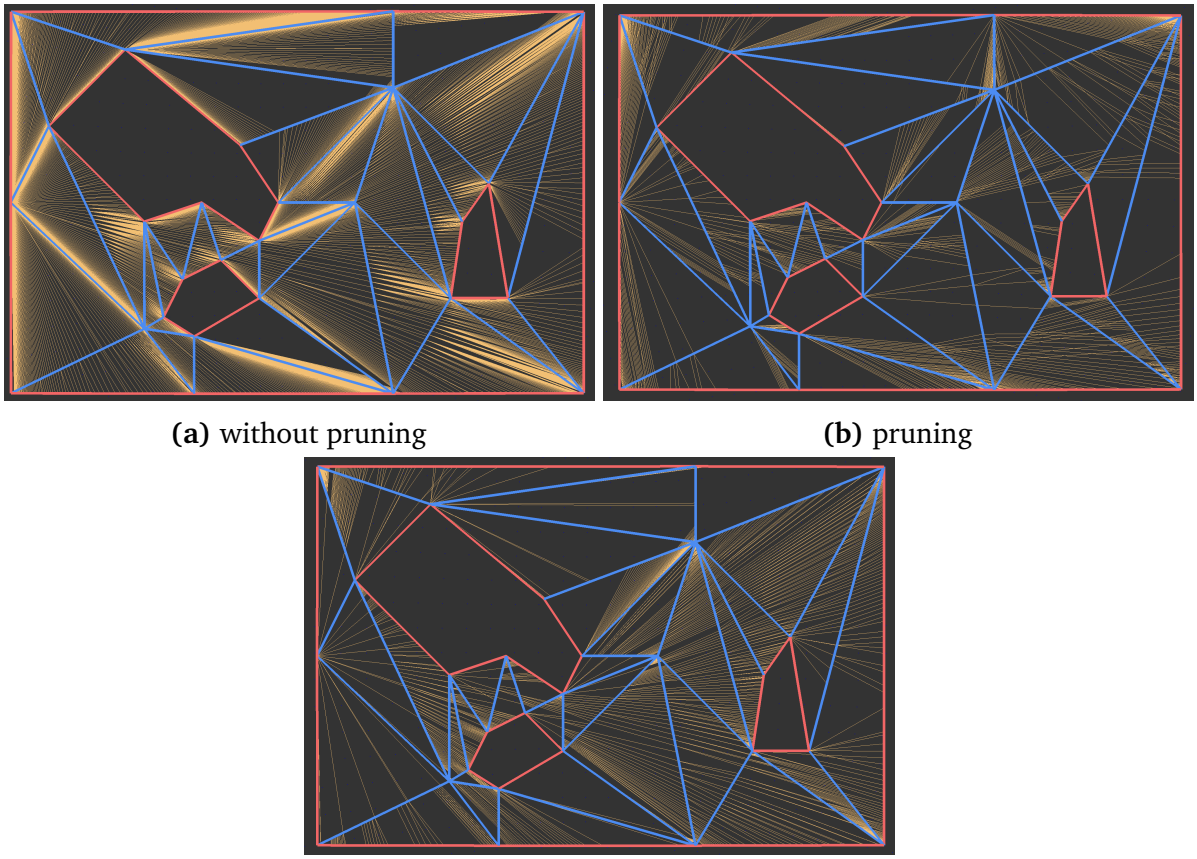
Below, the pseudocode for the implemented dijkstra search routine is listed.

```
distance := infinity for each vertex
parent := undefined for each vertex
last_bending_point := src for each vertex

distance[s] := 0
parent[s] := s
last_bending_point[s] = s

while Q is not empty:
    u := extract_min(Q)

    N := outgoing segments of u given last_bending_point[u]
    for (u,n) in N:
        cost = distance[u] + distance(coordinates(u), coordinates(n))
        if cost <= distance[n]:
            distance[n] := cost
            parent[n] := u
            if u is boundary vertex:
                last_bending_point[n] = u
            push(Q, (n,cost))
```

For `Q`, a `std::priority_queue` is used, which is based on a binary heap (for GCC). As it does not provide a decrease-key operation, nodes with updated cost are simply reinserted. As improving distances are updated when inserting entries into the queue, it is not necessary to update these values when pulling.

The neighbors of a node are selected according to section 2.7, with further implementation details stated below. The set of neighbors depends on the direction that a node is reached from. As the parent of a node can be arbitrarily close and bending can occur due to discretizing, deriving the direction from the last boundary vertex on the path to v is more stable.

Finally, for storing information about the visited nodes (cost, parent node, last bending point), a data structure is presented below (section 3.3), which grows dynamically with the size of the search tree and provides efficient access.



(a) without pruning

(b) pruning

(c) pruning (minimal angles)

Figure 3.1: shortest-path-trees (yellow) for the pruning variants at $\varepsilon = 1/8$

3.2.1 Generating outgoing segments

As the discretization graph is represented implicitly, the set of outward edges of a node $p \in V_\varepsilon$ cannot be simply looked up when searching for neighbors of p. For this reason, distances to the neighbors have to be computed when required. For each neighbor that should be added, the coordinates are computed as described in Equation 2.3 and then used to determine the Euclidean distance from the current node to the neighbor. This distance is then used as the cost for that segment.

Concerning pruning, the implementation provides three variants:

- no pruning: searches all edges in G_ε
- pruning: searches all edges in G_ε according to section 2.7
- pruning with minimal bending: searches all edges in G_ε according to section 2.7, but selects neighbors from subcones with least bending instead the shortest ones

```
angle = epsilon * pi/2
output = {}
# face-crossing segments
for each edge e2 reachable from v:
    x = intersection of ray(face_crossing_predecessor[v], v) with e2
    if x lies on e2:
        ql, qr = steiner interval of x
        output += cone_left(v, e2, ql, angle)
        output += cone_right(v, e2, qr, angle)

# edge-using segments
for each edge e2 incident to v:
    q = steiner point on e2 neighboring v
    output += (v, q)
```

Figure 3.2: selecting neighbors for a vertex v in $V_{\mathcal{P}}$

```
output = {}
# face-crossing segments
for each edge e2 reachable from v:
    output += epsilon_spanner(v, e2)

# edge-using segments
for each edge e2 incident to v:
    q = steiner point on e2 neighboring v
    output += (v, q)
```

Figure 3.3: selecting neighbors for a boundary vertex v in $V_{\mathcal{P}}$

The generation of the list of neighbors for a node p is described in this section. Figure 3.2 describes the selection of outgoing segments for vertices in $V_{\mathcal{P}}$, Figure 3.3 for boundary vertices, and Figure 3.4 for Steiner points.

3.2.2 Generation of ε-spanners

`cone_left, cone_right, epsilon_spanner` denote the subroutines for selecting neighbors from subcones and generating an ε-spanner over an edge. These differ for the two variants of pruning.

`cone_left(p, e, q, angle)` selects one segment pq' where q' lies on e (left or equal to q) and has an angle lower than the given one. When selecting neighbors by minimal bending angle, `cone_left` simply returns the point q. When selecting neighbors by minimal distance, a linear search is performed over the points q' left of q and the closest one

```
angle = epsilon * pi/2
output = {}
# face-crossing segments
e := edge that p lies on
for each edge e2 reachable from e:
    x = intersection of ray(face_crossing_predecessor[p], p) with e2
    if x lies on e2:
        ql, qr = steiner interval of x
        output += cone_left(p, e2, ql, angle)
        output += cone_right(p, e2, qr, angle)

# edge-using segments
ql,qr = steiner points neighboring p on e
output += (p, ql)
output += (p, qr)
```

Figure 3.4: selecting neighbors for a Steiner point p

returned. This could potentially be implemented in $\mathcal{O}(1)$ without linear search, which is omitted here. `cone_right` works symmetrically.

`epsilon_spanner(p, q, e)` selects the set of segments that make up the ε-spanner from p over the points on edge e. This is done by starting at q and iterating over the points to the left and to the right of q. When iterating over the points, a new subcone is recognized by the angle between the current segment and the first segment of the previous subcone exceeding the angle of $\varepsilon\pi/2$. When selecting by minimal bending angle, the segment of the first point q' of each subcone is selected. When selecting by distance, the closest point q' from each subcone is used. Both variants have linear complexity in the number of points on the target edge e. Theoretically, this could be improved by using exponential search for finding the start of the next subcone. As the runtime is mainly dominated by neighbor generation for Steiner points, which does not involve this routine, this is omitted.

3.2.3 Search of neighbors on the current search direction

The construction of $Cone^*(\tilde{\Pi})$, (with $\tilde{\Pi} = s \dots a \dots p$) requires finding the Steiner interval, which is intersected by the ray starting at p in the direction given by a parent in $\tilde{\Pi}$ (see Figure 2.9). It is important to find such an interval with minimal computational demand.

Let q be the intersection of the ray from a over p and the edge $e = (v, w)$. Let the points on e be positioned in order $v, q_1, \ldots, q_k, w$. The corresponding steiner points q_i, q_{i+1} have to be found such that q lies between these points.

One can either find q (defined by $x \in [0, 1] : q = v + x(w - v)$) and map it to the index of the corresponding steiner interval (subsubsection 3.2.3, subsubsection 3.2.3), or search the set of steiner points on e directly(subsubsection 3.2.3).

The variant mainly used in the benchmarks is the first one.

Computing line intersections from parameter representations

The required value x can be computed by finding the intersection of the lines from p to q and v to w. These are defined by their parameter representations $p + x' \cdot d_p$ and $v + x \cdot d_v$, with $d_p = p - a, d_v = w - v$. From the solution (x, x'), x is the desired value.

The problem can be formulated as a linear equation system:

$$\begin{aligned}
p + x' \cdot d_p &= v + x \cdot d_v \\
x' \cdot d_p - x \cdot d_v &= v - p \\
\begin{pmatrix} -d_{v1} & d_{p1} \\ -d_{v2} & d_{p2} \end{pmatrix} \cdot \begin{pmatrix} x \\ x' \end{pmatrix} &= v - p \\
\begin{pmatrix} x \\ x' \end{pmatrix} &= \begin{pmatrix} -d_{v1} & d_{p1} \\ -d_{v2} & d_{p2} \end{pmatrix}^{-1} \cdot (v - p) \\
\begin{pmatrix} x \\ x' \end{pmatrix} &= \frac{1}{-d_{v1} \cdot d_{p2} + d_{p1} \cdot d_{v2}} \begin{pmatrix} d_{p2} & -d_{p1} \\ d_{v2} & -d_{v1} \end{pmatrix} \cdot (v - p)
\end{aligned}$$

Then we have

$$x := \frac{1}{-d_{v1} \cdot d_{p2} + d_{p1} \cdot d_{v2}} \cdot ((v_1 - p_1)d_{p2} - (v_2 - p_2)d_{p1})$$

This computation requires $\mathcal{O}(1)$ time and only requires floating point addition, multiplication and division.

Using angles

Intuitively, this method considers two triangles: The one consisting of p, v, w and the one consisting of p, v, q. Here, the coordinates of q are not known, and the direction of

the ray ap is used instead. Using the law of sines, the ratio of $\frac{|vq|}{|vw|}$ is computed, which is the desired value x.

Let $\gamma := \angle vpq$, which is computed from the angle between the directions of ap and pv. Let $\alpha := \angle qvp$, which can be computed from the positions p, v, w. Let $\theta := \angle pqv = \pi - \gamma - \alpha$.

In general, the law of sines states that for a triangle with sides a, b, c and the respective opposite angles α, β, γ the equation $\frac{|a|}{\sin(\alpha)} = \frac{|b|}{\sin(\beta)} = \frac{|c|}{\sin(\gamma)}$ holds. Applied to the triangle pvq, it follows that

$$\frac{|vq|}{\sin(\gamma)} = \frac{|pv|}{\sin(\theta)}$$

which can be rewritten to

$$x := \frac{|vq|}{|wv|} = \frac{|pv|}{|wv|} \cdot \frac{\sin(\gamma)}{\sin(\theta)}$$

This computation requires $\mathcal{O}(1)$ time. However, the required computation of angles can make it rather costly.

Binary search with orientation test

A third method is searching for the steiner point $q_i \in V_\varepsilon$ that has the smallest angle $\angle pq_i, ap$ between the lines pq_i and ap. Equivalently, one can search for q_i with the smallest value of $|\sin(\angle pq_i, ap)|$. This can be done by performing a binary search using the sign of $\sin(\angle pq_i, ap)$ as an orientation test for whether the solution lies left or right of the tested element.

Instead of computing the sine of the angle between pq_i and ap, one can select a direction d orthogonal (either to the left or right) to ap and compute the cosine. This direction d is chosen such that a higher value $\cos(\angle pq_i, d)$ corresponds to higher indices i. By that construction, the sign of $\cos(\angle pq_i, d)$ serves as an orientation test for whether the solution has lower or higher index than i.

Selecting d can be done by rotating ap to the right and multiplying with the sign of $(w - v) \cdot d$, as that ensures that $d \perp ap$ and $(w - v) \cdot d > 0$.

Computing the cosine of the angle between two direction vectors u, v in $\mathbb{R}^2$ can be done efficiently using the scalar product:

$$\cos(\angle(u, v)) = \frac{u \cdot v}{|u||v|}$$

As only the sign of the $\cos$-value is required, dividing by the product of norms can be omitted.

This method results in $\mathcal{O}(\log k)$ runtime. An iteration using the given orientation test is cheap as it only requires a single dot product computation. For low numbers of steiner points, this method might be faster than the $\mathcal{O}(1)$ alternatives.

3.3 Datastructure for the shortest path tree

The search in G_ε requires a data structure that holds a distance value $d(v)$ and the parent $parent(v)$ for each node v that is visited.

For one-to-all searches, an array of size $|V_\varepsilon|$ can be used in combination with an offset array I that contains the indices of the first point for each edge.

For one-to-one searches, where the search does not necessarily visit all nodes, the list can grow dynamically. When inserting information for a node p which lies on an edge e with k Steiner points, k entries are appended to the list and the index of e in I updated to point to the first one.

This data structure combines the space-efficiency of a map-based data structure with the performance of array-based ones, as no hashing is required. Additionally, this data structure should provide good data locality, as the neighbors of the most recently visited nodes are likely to appear at the end of the list.

3.3.1 Partial storage of the tree

With a shortest-path-tree being a subgraph of G_ε with up to $|V_\varepsilon|$ nodes, its memory demand can be too high for larger graph instances at lower ε-values. To solve this, a data structure can be used which discards tree information as soon as it is not needed for further computations.

Let $w = \max_{e \in E_\mathcal{P}}\{|e|\}$ be the length of the longest edge in the triangulation. Let d be the distance from the source node to the closest node in the queue. Then nodes with distance lower than $d - w$ cannot be reached by any node that is still in the queue. Thus, distance values for closer nodes can be discarded. To keep information on the shortest path tree, the distance and parent id are written to a file when a node is removed from the queue. When only requiring distances for a subset of nodes, it is also possible to only output these.

This functionality is implemented by a hashmap that maps an edge to a list of distances for the Steiner points placed on it. Additionally, a priority-queue is used which contains information on when the information for an edge can be discarded. Whenever an edge

is seen for the first time by assigning a distance to a point on it, an entry for its Steiner points is created in the map. The edge is then inserted into the priority queue, with the distance of the point as its priority. Whenever a node is removed from the search queue, the data structure is informed of this new distance d and edges with priorities lower than $d - 2w$ removed from the hashmap.

This data structure is also implemented and functional, but not benchmarked.

As the use of a hashmap creates some overhead due to hashing, the data structure could be optimized by using an array-based approach as described above. However, this requires the removal of elements to be implemented, which is omitted.

3.4 Geometric functions

The presented algorithms require the implementation of angle und norm computations, which are briefly explained here.

3.4.1 Norm

For distances between points p, q, or equivalently the norm of the vector $q - p$ is implemented using the euclidean norm:

$$\text{norm\_squared}((x, y)) = x^2 + y^2$$

$$\text{norm}((x, y)) = \sqrt{x^2 + y^2}$$

When calculating products or ratios of distances, the number of calls to `std::sqrt` is reduced by using the squared norm for intermediate results and computing the root in the last step.

3.4.2 Angle

An angle $\angle uv$ between two vectors $u = (x_1, y_1), v = (x_2, y_2)$ is computed via

$$\theta = \min\left\{\left|\arctan\left(\frac{y_1}{x_1}\right) - \arctan\left(\frac{y_2}{x_2}\right)\right|, 2\pi - \left|\arctan\left(\frac{y_1}{x_1}\right) - \arctan\left(\frac{y_2}{x_2}\right)\right|\right\}$$

using `std::atan2` from the C++ standard library.

Faster computation of Cosines between vectors

The cosine of the angle between to vectors can be computed using the dot product, without the need for calls to `std::atan2`:

$$\cos(\angle uv) = \frac{u \cdot v}{|u| \cdot |v|}$$

4 Evaluation

To evaluate the practicality of the presented implementation, some experiments are conducted on real-world instances.

4.1 Methodology

The implementation is compiled using GCC at version 11.4.0 with optimization flags `-O3 -ffast-math`. The benchmarks are run on an Intel Core i5-9500 CPU clocked at 3.00GHz with 64GiB memory.

In the following, ε refers to the value used for discretization, the solution quality is therefore bounded by the factor $1 + 7\varepsilon$.

4.1.1 Problem instances

The used instances are based on parts of the global coastlines dataset from OpenStreetMap. From the points and coastline information, a constrained Delaunay triangulation is generated. The coordinates are projected via the wgs84-projection and the resulting values interpreted as x, y-values in Euclidean space.

For refinement, the triangle-package [She96] is used to generate triangulations with inner angles of minimum $25^\circ = \frac{5\pi}{36}$, where applicable. For some graph instances (namely *pata* and *medi*), this implementation does not terminate when generating refined meshes. An alternative method of refinement is also used, which does not always ensure any bounds on inner angles but works more reliably on the available graph instances. In particular, this implementation can create skinny triangles near boundary edges, resulting in larger discretization graphs.

The following instances are used:

1. *aegaeis*: the aegaeis sea, with latitudes in (34.0343, 40.9997) and longitudes in (22.0001, 28.0000):

a) *aegaeis-triangle*: a refined version, consisting of 522985 vertices and 816249 triangles, refined using the triangle-package.

b) *aegaeis-ref*: a refined version, consisting of 556965 vertices and 873611 triangles.

c) *aegaeis-unref*: an unrefined version, consisting of 208885 vertices and 218875 triangles.

d) *aegaeis-vis*: a visibility graph, consisting of 556965 vertices and 313816822 edges.

This graph consists of two unconnected components, causing some queries to have no solution, these are manually removed. Further, although the coastline information of *aegaeis-unref* and *aegaeis-vis* is equal, the boundaries differ in the ocean, where the graphs were cut out. This also causes different shortest paths for some queries, which are manually removed as well. Both methods of refining generate a similar number of vertices and faces, the inner angles of these faces differ as shown below.

2. *milos*: an island in the Aegaeis sea, with latitudes in (36.508614, 36.8876) and longitudes in (24.09291, 24.71498):

   a) *milos-ref*: a refined version, extracted from the *aegaeis-triangle* graph, consisting of 47610 vertices and 74795 triangles.

   b) *milos-unref*: an unrefined version, extracted from the *aegaeis-unref* graph, consisting of 19447 vertices and 21025 triangles.

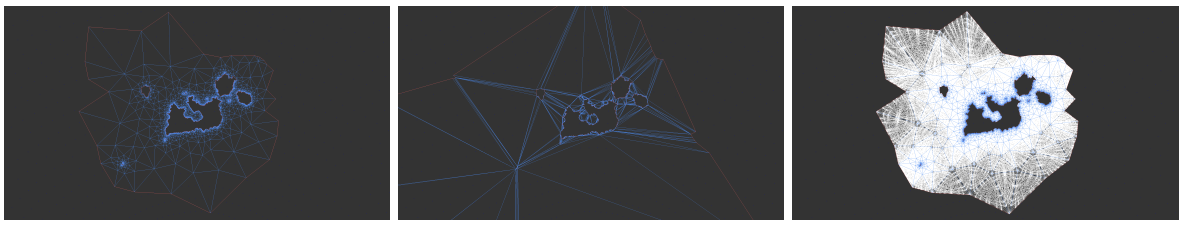
   c) *milos-explicit-ε*: explicit representations of the discretization graph generated from *milos-ref* for $\varepsilon \in \{1, 1/2, 1/4\}$.

   As these instances are generated by selecting a subset of vertices and faces from the *aegaeis*-instances, the area covered by the triangulation differs between the *ref* and *unref* graphs (see Figure 4.1). For this reason, only the *ref* instance is suited for comparison to *explicit*.

3. *medi*: the Mediterranean sea, with latitudes in (29.99434, 40.76925), and longitudes in (-0.72809, 36.21847)

   a) *medi-ref*: a refined version, consisting of 826150 vertices and 1291659 triangles.

   b) *medi-vis*: a visibility graph, consisting of 826150 vertices and 721394892 edges.



(a) *milos-ref* **(b)** *milos-unref* **(c)** *milos-ref-explicit-1*

Figure 4.1: the different *milos* instances

This instance is larger than *aegaeis*, with ~ 1.48 times the number of nodes and more than ~ 2.30 times the edges. This number of edges is higher because this instance has large open areas, where high numbers of vertices are pairwise visible.

4. *patagonia*: the western coast of the Patagonia region in Chile and Argentinia, with latitudes in (-55.86781, -39.08036) and longitudes in (-80.43261, -69.90562). In contrast to the *aegaeis*-instance, this graph consists of more small obstacles and smaller visible areas.

   a) *pata-ref*: a refined version, consisting of 2303943 vertices and 3543559 triangles.

   b) *pata-vis*: a visibility graph, consisting of 2303943 vertices and 315653758 edges.

   For the *pata*-instance, the size of G_ε grows too large for shortest path computations to be possible with this implementation. For $\varepsilon = 1/2$, the $|V_\varepsilon|$ is already at $> 1,27 \cdot 10^{15}$, which makes storing a shortest-path-tree unrealistic. For this reason, it is **not included** in the results.

4.1.2 Experiments

The first aspect to evaluate is the size of discretization graphs for a fixed instance, depending on ε. For this, the required information on steiner point placement is computed to extract the number of nodes and edges. The results of this experiment are listed at section 4.2.

Another aspect concerning graph sizes is the scaling of a discretization graph with the minimal inner angle of the graph instance. For this experiment, a class of graph instances consisting of a triangle with a small inner angle is used (see subsection 4.2.1).

As the performance depends on some implementation details of the pruned search, these are compared as well (see subsection 4.3.2). Similarly, the performance differences

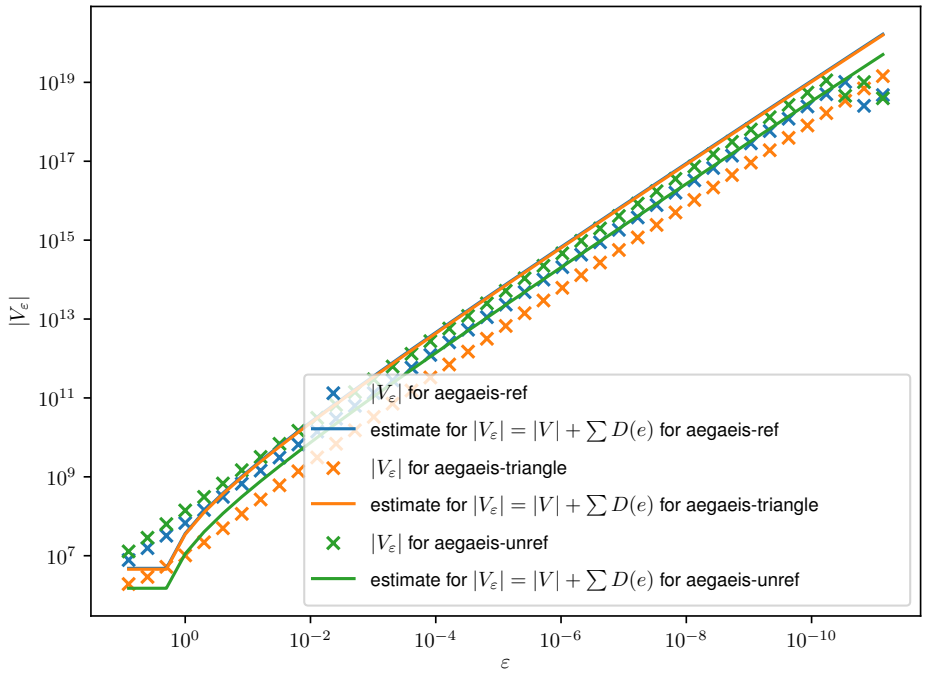


Figure 4.2: $|V_\varepsilon|$ for refined and unrefined aegaeis graphs

between implicit, explicit and semi-explicit (nodes only) representations are measured (see subsection 4.3.1).

To evaluate the performance and solution qualities on larger instances, one-to-all and one-to-one computations are performed on the different graph instances (i.e. triangulations with different ε-values and the respective visibility graph). For this, the *aegaeis* and *medi* instances are used. The approximated distances are then compared to the exact distances computed in the visibility graph.

4.2 Graph sizes and space demand

The scaling of graphs in ε is examined for the *aegaeis* graph. The maximum number of Steiner points per edge is set to $2^{63} \approx 9.223 \cdot 10^{18}$ and indices stored in 64-bit integers. This maximum is never exceeded here.

Figure 4.2 shows the scaling for the aegaeis-graph and estimates for the number of points (as calculated in section 2.5) for angles $\alpha = \frac{5\pi}{36} \approx 25°$ and the respective numbers of edges. As the *unref* and *ref* graphs contain lower inner angles, these estimates are not upper bounds and are exceeded. The plots of a graph's size and the respective estimate

appear parallel on the logarithmic scale, suggesting that the sizes scale similarly to the estimate multiplied with some constant.

Note the lower values at $\varepsilon < 10^{-11}$. These are caused by overflows during counting, as 64-bit integers are used (with the highest representable value being $2^{64} - 1 \approx 1.84 \cdot 10^{19}$). At such graph sizes, storing 1 byte per node would exceed the maximum amount of memory addressable on 64-bit architectures.

The distributions of inner angles for the *aegaeis* instances are plotted in Figure 4.3a. These are generated by iterating over all faces and counting the angles that fall into bins of size $\frac{\pi}{200} \approx 1.8°$. As can be seen in the plot, the angles concentrate at small values for the unrefined version. These increase the number of points inserted and could potentially cause arithmetic issues. For the refined graphs, most angles fall into the interval $[\pi/4, \pi/2]$. For *aegaeis-ref*, there are still smaller angles contained, although not as many as for *aegaeis-unref*. Similar distributions can be seen with the radius values r_{vw}, which also affect the number of points inserted. For the unrefined graph, these are mostly values close to 0. In contrast, for *aegaeis-triangle*, very few values below 0.02 are present.

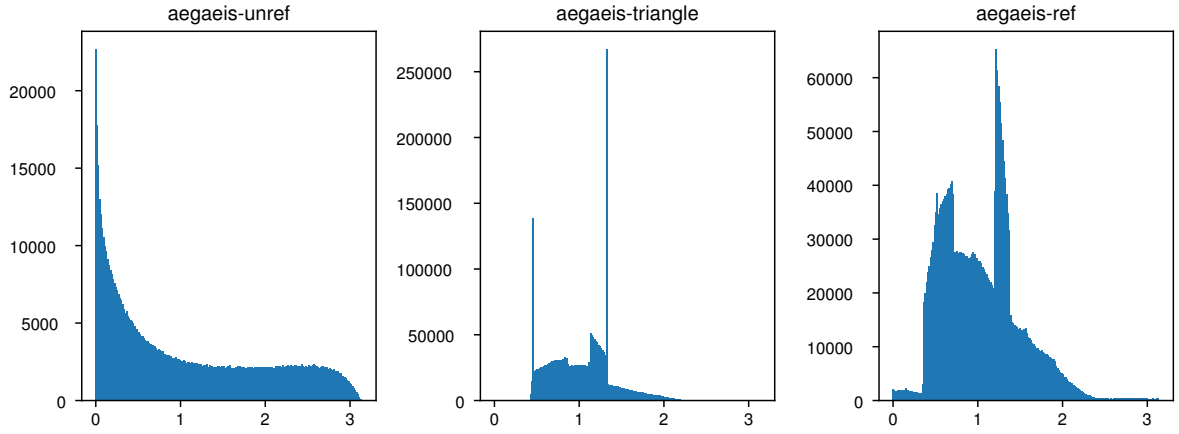
The distributions over the number of points inserted per edge can be seen in Figure 4.3c. For the refined versions, the number of points per edge show a relatively narrow distribution, while for the unrefined version, the values are distributed more broadly with higher values occurring more often. These high values of $\log(k)$ explain the higher graph size for *aegaeis-unref*.

Figure 4.4 shows the exact numbers of nodes and edges the discretizations based on *aegaeis* have. For comparison, the size of the visibility graph and its memory requirement are included as well.

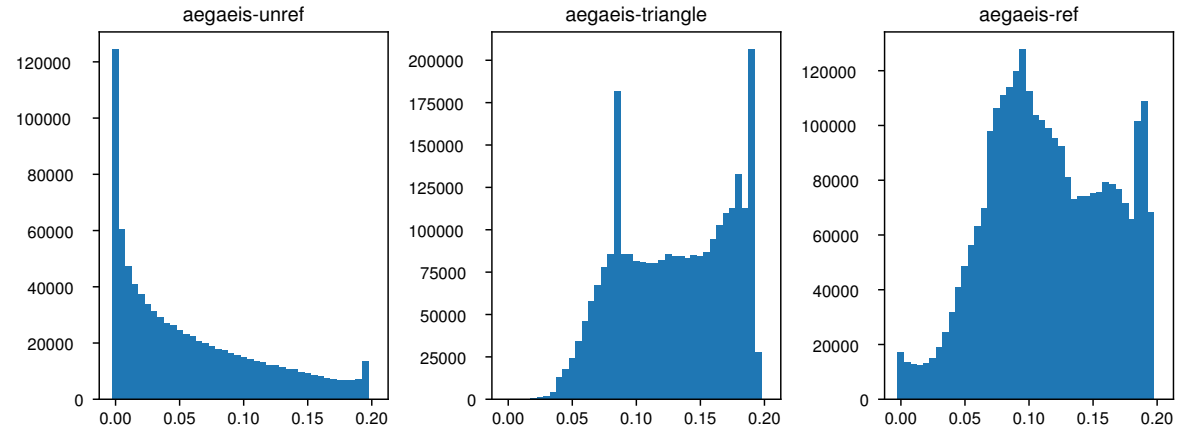
Additionally, the memory requirement for storing all nodes explicitly is included. As the implicit graph representation has $\mathcal{O}(1)$ space complexity in ε, the memory usage remains constant at `~180MiB` for the *triangle/ref*-instance and at `~60MiB` for *unref*. The semi-explicit representation stores two `double` values per node, an additional memory demand of $|V_\varepsilon| \cdot 16B$ is to be expected, which roughly matches the measured numbers. Figure 4.5 shows the graph sizes for smaller values of $\varepsilon \in \{2^{-6}, 2^{-10}, \ldots, 2^{-34}\}$. For the number of edges, overflows seem to occur on the edge counts at $\varepsilon < 2^{26}$.

Between the different *aegaeis* instances, the sizes differ greatly. For *unref*, the number of nodes is about 10 times as high as for the *triangle*-instance, despite the triangulation being far smaller. Between *ref* and *triangle* the sizes differ too, by a factor of ~ 5 although the sizes of the triangulations are very similar.

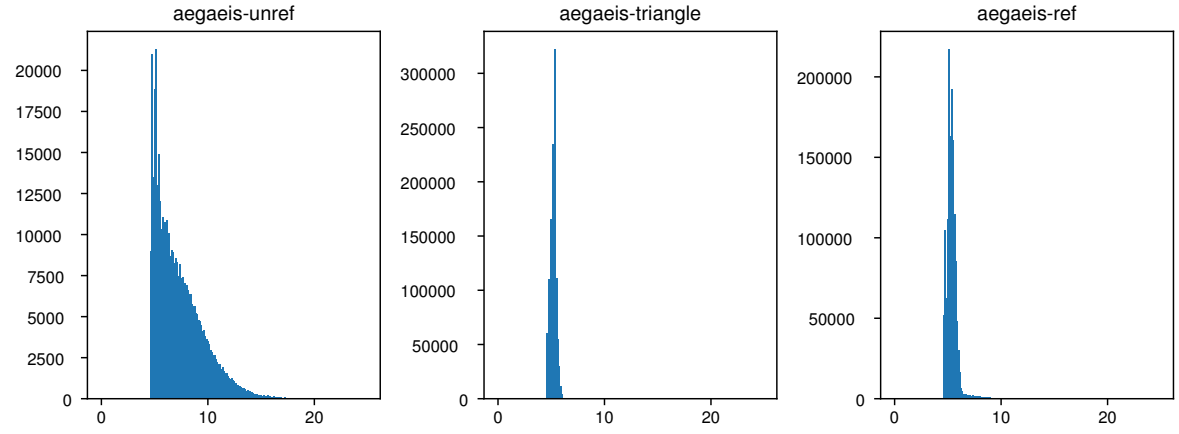
The reason for this is that there are a few edges in the *ref* and *unref* instances where a high number of Steiner points is placed. To explain the high values of $|V_\varepsilon|$ and $|E_\varepsilon|$,



(a) angles, *aegaeis*



(b) r_{vw}, *aegaeis*



(c) distribution over $\log_2(k)$ of the number k of points per edge at $\varepsilon = 1/4$

Figure 4.3: distributions over the different values affecting graph sizes

ε	$\lvert V_\varepsilon \rvert$	$\lvert E_\varepsilon \rvert$	memory semi-explicit[KiB]
4	15.61×10^6	14.34×10^{12}	491.46×10^3
2	32.13×10^6	64.32×10^{12}	757.12×10^3
1	67.56×10^6	286.64×10^{12}	1.32×10^6
1/2	144.40×10^6	1.27×10^{15}	2.51×10^6
1/4	310.45×10^6	5.60×10^{15}	5.11×10^6
1/8	668.51×10^6	24.60×10^{15}	10.71×10^6
1/16	1.44×10^9	107.54×10^{15}	22.73×10^6
1/32	3.08×10^9	468.38×10^{15}	-
1/64	6.59×10^9	2.03×10^{18}	-

(a) *aegaeis-ref*

ε	$\lvert V_\varepsilon \rvert$	$\lvert E_\varepsilon \rvert$	memory semi-explicit[KiB]
4	28.54×10^6	1.19×10^{12}	541.71×10^3
2	63.44×10^6	5.44×10^{12}	1.09×10^6
1	140.54×10^6	24.73×10^{12}	2.30×10^6
1/2	309.62×10^6	111.64×10^{12}	4.94×10^6
1/4	677.79×10^6	500.74×10^{12}	10.70×10^6
1/8	1.47×10^9	2.23×10^{15}	23.15×10^6
1/16	3.19×10^9	9.90×10^{15}	-
1/32	6.86×10^9	43.70×10^{15}	-
1/64	14.69×10^9	191.97×10^{15}	-

(b) *aegaeis-unref*

ε	$\lvert V_\varepsilon \rvert$	$\lvert E_\varepsilon \rvert$	memory semi-explicit[KiB]
4	2.90×10^6	33.63×10^6	307.96×10^3
2	5.24×10^6	92.63×10^6	331.48×10^3
1	10.18×10^6	323.60×10^6	404.15×10^3
1/2	21.94×10^6	1.43×10^9	587.03×10^3
1/4	49.78×10^6	7.17×10^9	1.02×10^6
1/8	115.18×10^6	37.94×10^9	2.05×10^6
1/16	266.34×10^6	201.75×10^9	4.41×10^6
1/32	610.83×10^6	1.06×10^{12}	9.80×10^6
1/64	1.38×10^9	5.43×10^{12}	21.88×10^6

(c) *aegaeis-triangle*

ε	$\lvert V \rvert$	$\lvert E \rvert$	memory[KiB]
0	556.97×10^3	313.82×10^6	6.00×10^6

(d) *aegaeis-vis*

Figure 4.4: exact graph sizes for *aegaeis*

ε	$\lvert V_\varepsilon \rvert$	$\lvert E_\varepsilon \rvert$
2^{-6}	1.38×10^9	5.43×10^{12}
2^{-10}	32.98×10^9	3.07×10^{15}
2^{-14}	703.33×10^9	1.40×10^{18}
2^{-18}	14.07×10^{12}	4.96×10^{18}
2^{-22}	270.06×10^{12}	4.57×10^{18}
2^{-26}	5.04×10^{15}	8.01×10^{18}
2^{-30}	92.18×10^{15}	1.61×10^{18}
2^{-34}	1.66×10^{18}	17.05×10^{18}

Figure 4.5: *aegaeis-triangle* with smaller values of ε

ε	$\lvert V_\varepsilon \rvert$	$\lvert E_\varepsilon \rvert$	memory semi-explicit[KiB]
4	19.02×10^6	13.68×10^{12}	668.24×10^3
2	38.91×10^6	61.43×10^{12}	992.15×10^3
1	81.74×10^6	274.17×10^{12}	1.65×10^6
$1/2$	175.32×10^6	1.22×10^{15}	3.11×10^6
$1/4$	378.88×10^6	5.37×10^{15}	6.30×10^6
$1/8$	820.61×10^6	23.61×10^{15}	13.21×10^6
$1/16$	1.77×10^9	103.35×10^{15}	28.12×10^6
$1/32$	3.83×10^9	450.59×10^{15}	-
$1/64$	8.22×10^9	1.96×10^{18}	-

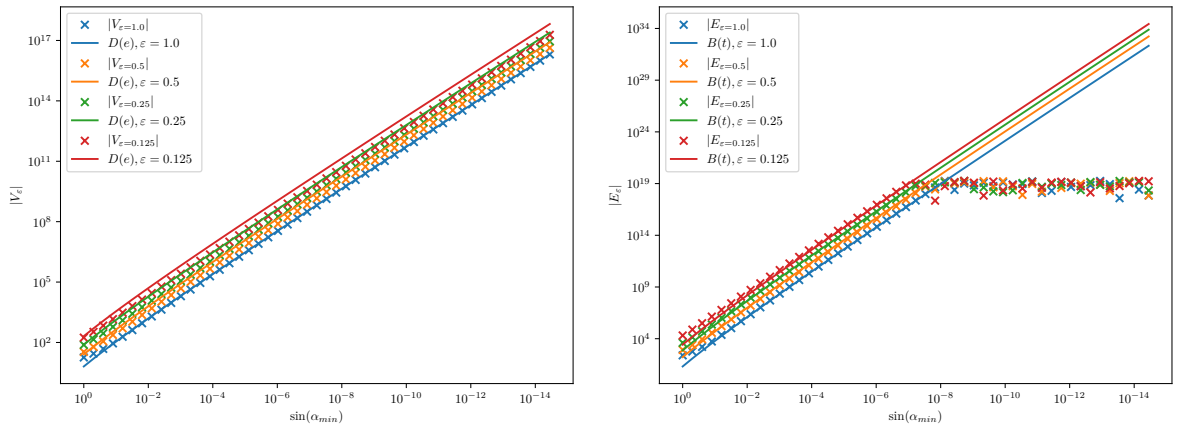
(a) *medi-ref*

ε	$\lvert V \rvert$	$\lvert E \rvert$	memory[KiB]
0	826.15×10^3	721.39×10^6	13.78×10^6

(b) *medi-vis*

Figure 4.6: exact graph sizes for *medi*

consider a few edges in *aegaeis-ref* at $\varepsilon = 0.5$ that have the largest number of points placed on them: There is 1 edge with $> 2^{24}$, 2 with $> 2^{23}$, 3 with $> 2^{22}$ and 3 with $> 2^{21}$ points. These already contribute at least $2^{24} + 2 \cdot 2^{23} + 2 \cdot 2^{22} + 3 \cdot 2^{21} \approx 48 \cdot 10^6$ nodes to V_ε. Assuming that two of these edges border the same triangle, $> (2^{22})^2 = 2^{44} \approx 1,8 \cdot 10^{13}$ face-crossing segments have to be expected for that triangle alone. These numbers already lie in the order of magnitude of the measured numbers. In consequence, the number of nodes and edges measured is mainly dominated by a few triangles with small inner angles.



(a) number of nodes placed **(b)** number of edges

Figure 4.7: graph sizes of the skinny triangle $\triangle_{\alpha_{min}}$

Further, with these few edges, the margin of error due to imprecise angle computations for $|V_\varepsilon|, |E_\varepsilon|$ is very high. Other methods of angle computation could potentially result in very different graph sizes (in particular the different numbers of edges).

4.2.1 Number of points inserted on skinny triangles

To examine how the number of steiner points scales with $\sin(\alpha_{min})$, as described in Equation 2.3, the following type of instances is used: A single triangle $\triangle_\alpha$ between the points $a = (0.0, 0.0), b = (1.0, 0.0), c = (0.0, \sin(\alpha))$.

For $\varepsilon \in \{0.125, 0.25, 0.5, 1.0\}$ and $\alpha = 2^{-i}, 0 \leq i < 48$, such instances are generated and the size of the graph is computed. This way, the results in Figure 4.7 are obtained. For this experiment, the limit on Steiner points per edge is set to $2^{63} \approx 9.223 \cdot 10^{18}$ and indices are stored in 64-bit integers.

Figure 4.7 shows the graph sizes and estimates derived using section 2.5 on the respective angle. For the examined values of ε, the number of nodes scales similarly to the estimate multiplied by some constant. As a result, graph sizes can be expected to scale with the inner angles as described in section 2.5. The same holds for the number of edges. As expected it scales roughly with the estimate, i.e. the square of the node count. At 10^{19}, overflows occur while counting and therefore these values can be ignored.

ε	nodes	edges	file size [MiB]
1	931.50×10^3	28.70×10^6	1.07×10^3
1/2	2.01×10^6	128.13×10^6	4.94×10^3
1/4	4.54×10^6	645.43×10^6	24.77×10^3

Figure 4.8: graph and file sizes for *milos-explicit-*ε

4.3 Performance and solution quality

As the performance and potentially the solution qualities depend on some implementation details of the pruned search, these are compared here. On the *milos* graph, a random set of 50 source vertices is selected from the boundary vertices and one-to-all computations performed with different configurations. These vary by the following parameters:

- pruning: either no pruning, pruning according to section 2.7 or pruning where neighbors are selected such that bending angles are minimal.
- graph representation: either implicit, semi-explicit (with coordinates stored) or fully explicit
- neighbor selection: selecting the steiner interval intersected by the current search direction can be performed using parameter representations(*param*), law of sines (*trig*) or binary search (*binary*)

These configurations are compared in subsection 4.3.1 and subsection 4.3.2.

To measure the performance on larger instances, one-to-all and one-to-one queries are performed on *aegaeis* and *medi*, see subsection 4.3.3 and subsection 4.3.4.

4.3.1 Graph representations

Due to the high number of nodes and edges of the discretization graphs, a smaller graph is chosen for evaluating explicit representations and comparing search configurations. This smaller graph (*milos-ref*) is extracted from the *aegaeis-triangle*-instance. For this subgraph (still represented by a triangulation), the information on Steiner point placement is computed as described in subsection 3.1.2. The resulting graph is then written to a text file containing the coordinates of all nodes and edges represented by the indices of their endpoints and length. These graphs are identical to their implicitly represented counterparts. Such a graph file can be read into an adjacency-list-based graph data structure for shortest path computations and memory measurement.

storage	ε	graph [MiB]	graph+tree [MiB]
explicit	0.25	12 427.50	12 499.30
explicit	0.5	2484.97	2520.59
explicit	1.0	599.84	602.00
implicit	0.25	17.20	215.07
implicit	0.5	17.24	123.65
implicit	1.0	17.12	81.43
semi-explicit	0.25	93.62	306.90
semi-explicit	0.5	54.68	168.96
semi-explicit	1.0	36.92	103.22

Figure 4.9: memory demand for the different graph representations

The graph sizes are summarized in Figure 4.8. The memory demand (Figure 4.9) is measured once with the graph datastructures initialized, and once with graph and search datastructures after a one-to-all computation. As already established, the memory demand of the implicit representation is constant in ε, with the measured demand being far lower than the ones of the explicit graph representations.

As can be seen from the tables, storing the full graph (i.e. nodes and edges) explicitly, requires far more memory than the fully implicit representation. For larger graphs, one should not expect explicit representations to be applicable. The semi-explicit representation (storing only the nodes) however, only moderately increases the memory requirement. This can also be seen for the *aegaeis-triangle* instance (Figure 4.4), where a fully explicit representation would not be practical.

The search performance of the different graph representations is compared in Figure 4.10. The measured timings show the performance benefits of relying on explicitly stored nodes and edges. The fully explicit graph can be searched about in about $\sim 30\%$ less time than for the implicit representation (without pruning). The semi-explicit graph representation provides an improvement of $\sim 15\%$ compared to the implicit one. With pruning, it is still $\sim 10\%$.

4.3.2 Pruning

The different methods of pruning and computing intersections are compared in Figure 4.10. The table shows the average query time and the number of edges in E_ε that is checked during the search.

storage	pruning	intersections	ε	edges checked	time[ms]
explicit	none		0.25	645.43×10^6	14 951.6
implicit	none	binary	0.25	643.82×10^6	23 238.4
implicit	none	param	0.25	643.82×10^6	22 703.3
implicit	none	trig.	0.25	643.82×10^6	23 337.5
implicit	prune	binary	0.25	43.67×10^6	6557.9
implicit	prune	param	0.25	28.54×10^6	3505.1
implicit	prune	trig.	0.25	21.18×10^6	4167.0
implicit	min-angle	binary	0.25	43.32×10^6	4717.3
implicit	min-angle	param	0.25	25.98×10^6	2266.6
implicit	min-angle	trig.	0.25	19.05×10^6	3168.4
semi-explicit	none	binary	0.25	643.82×10^6	19 253.9
semi-explicit	none	param	0.25	643.82×10^6	18 606.2
semi-explicit	none	trig.	0.25	643.82×10^6	19 291.4
semi-explicit	prune	binary	0.25	43.67×10^6	4469.2
semi-explicit	prune	param	0.25	28.54×10^6	2684.9
semi-explicit	prune	trig.	0.25	21.18×10^6	3540.0
semi-explicit	min-angle	binary	0.25	43.32×10^6	3566.8
semi-explicit	min-angle	param	0.25	25.98×10^6	1955.8
semi-explicit	min-angle	trig.	0.25	19.05×10^6	2976.2

Figure 4.10: query times for different configurations

ε	vertices	segments	boundary vertices	segments
0.25	27.04×10^3	268.25×10^3	22.34×10^3	453.28×10^3
0.5	27.81×10^3	274.60×10^3	22.42×10^3	353.69×10^3
1.0	29.01×10^3	285.00×10^3	22.61×10^3	352.94×10^3

ε	Steiner points	segments
0.25	4.51×10^6	25.26×10^6
0.5	1.96×10^6	10.87×10^6
1.0	885.20×10^3	4.83×10^6

Figure 4.11: numbers of nodes of the different types and the respective outgoing segments that are searched with *min-angle*-pruning

pruning	ε	mean	max
prune	0.25	1.003 61	1.015 24
prune-min-angle	0.25	1.001 35	1.035 61
prune	0.5	1.007 89	1.032 25
prune-min-angle	0.5	1.002 96	1.055 60
prune	1.0	1.012 75	1.046 01
prune-min-angle	1.0	1.006 88	1.050 47

Figure 4.12: mean and max values over the qualities compared to the unpruned case

Pruning variants

The numbers of visited edges in the unpruned case correspond to the total number of edges in G_ε, as expected. These numbers show the benefit of pruning as it reduces this number to less than $\sim \frac{1}{15}$. Without pruning, the average number of outgoing edges that are checked per node is ~ 144.67. With pruning and the *param* neighbor selection, that number is reduced to ~ 6.44.

For smaller values of ε, one can expect even stronger reductions, as the degree of a Steiner point scales with $\varepsilon^{-1} \log \varepsilon^{-1}$, while the number of edges checked with pruning stays constant. Also, for instances that contain edges with larger values for $C(e)$, the pruning can be expected to reduce the number of edges more strongly. This is the case because the number of searched segments per node does not scale with its degree in G_ε, as seen in subsection 2.7.2.

Figure 4.11 lists the numbers of nodes and outgoing segments for the different types of nodes. For $\varepsilon = 0.25$, the average number of segments are ~ 10.10 for vertices, ~ 12.33 for boundary vertices and ~ 5.59 for Steiner points. This roughly matches the estimates from subsection 2.7.2: These numbers are a bit higher than the estimate, which can be explained by the way outgoing segments are selected: While the outgoing segments ending on one reachable edge are selected such that each lies in a different subcone of the ε-spanner, this is not the case for segments ending on different edges. Segments ending on different edges can actually belong to the same subcone.

The quality for the entire computed trees is checked too: For every query, the distance for each node is divided by the distance computed without pruning. The mean and maximum of these ratios is listed in Figure 4.12. Here, both pruning variants show a good solution quality, with the worst ratio being $1,05047$ at $\varepsilon = 1.0$, which is better than the theoretical ratio of $(1 + \pi\varepsilon/2)$. As a result, the pruning does not cost much in terms of quality for this graph. Further, the measured qualities are far better when pruning for

minimal bending angles, with the mean deviations being less half as large as for default pruning.

Neighbor selection

The different methods of selecting the neighbors on the current search direction also result in different numbers of visited edges as seen in Figure 4.10: This number is the largest for *binary* and lowest for *param*. The reason for these differences is that the checks for whether an edge is actually intersected work differently: Binary search simply adds neighbors for each visible edge in $E_{\mathcal{P}}$, without checking if that edge is intersected by the current search direction. For the alternatives, such a check is performed. Apparently, for the parameter representations, this check accepts more edges. It is not clear why that is the case.

The performance difference between the different methods of selecting the neighbors on the current search direction can be explained by these different numbers of edges and the differences in computational overhead. When comparing these for the unpruned case, the difference is negligible. When pruning for minimal bending angles, using the parameter representations performs better than the alternatives.

4.3.3 One-to-all queries

For evaluating the performance on larger instances, one-to-all dijkstra computations are performed on 5 randomly selected vertices in $\mathcal{P}$ and a few metrics collected. This smaller number of queries is chosen as results for one-to-all don't vary as much as for one-to-one computations and due to the higher computational effort.

As the *ref*- and *unref*- instances are larger (see Figure 4.4), the benchmarks are only run with $\varepsilon \geq 0.25$, while for *triangle* smaller values $\varepsilon \geq 0.0625$ are included.

The extracted metrics include:

- total runtime
- memory demand after search tree is complete

Figure 4.13 lists the query times for the different instances of *aegaeis* and *medi*. The rows with $\varepsilon = 0$ represent the query times for the visibility graph, and $\varepsilon = \infty$ the respective triangulation without steiner points. As the tables show, the query times for $\varepsilon = 1.0$ already exceed the times on the visibility graph. This is to be expected due to the different graph sizes. However, the query times achieved here would not be possible without pruning due to the large number of edges (see Figure 4.4).

ε	mean[ms]	max[ms]
0	1240.0	1252.0
0.25	123 993.4	125 287.0
0.5	56 276.8	57 486.0
1	26 063.0	26 565.0
inf	1344.8	1352.0

(a) *aegaeis-ref*

ε	mean[ms]	max[ms]
0	1240.0	1252.0
0.25	311 756.1	318 039.0
0.5	137 586.0	138 967.0
1	62 162.8	63 418.0
inf	420.9	432.0

(b) *aegaeis-unref*

ε	mean[ms]	max[ms]
0	1240.0	1252.0
0.0625	141 999.2	143 807.0
0.125	62 270.2	63 472.0
0.25	26 755.2	27 047.0
0.5	12 190.0	12 400.0
1	5964.0	6057.0
inf	1362.0	1373.0

(c) *aegaeis-triangle*

ε	mean[ms]	max[ms]
0	2993.6	3272.0
0.25	154 767.2	155 662.0
0.5	69 498.0	70 552.0
1	32 784.4	33 063.0
inf	2208.8	2215.0

(d) *medi-ref*

Figure 4.13: one-to-all query times

ε	mean[MiB]	max[MiB]
0	6032.3	6032.3
0.25	10 020.5	10 074.9
0.5	4728.0	4772.7
1	2338.6	2359.7
inf	255.4	255.4

(a) *aegaeis-ref*

ε	mean[MiB]	max[MiB]
0	6032.3	6032.3
0.25	20 796.2	20 832.6
0.5	9534.6	9563.2
1	4384.4	4396.6
inf	89.9	90.1

(b) *aegaeis-unref*

ε	mean[MiB]	max[MiB]
0	6032.3	6032.3
0.0625	8264.9	8272.9
0.125	3685.4	3706.0
0.25	1702.8	1714.5
0.5	862.2	866.5
1	506.1	512.4
inf	242.7	242.9

(c) *aegaeis-triangle*

ε	mean[MiB]	max[MiB]
0	13 859.7	13 875.4
0.25	12 168.2	12 175.1
0.5	5730.9	5832.7
1	2868.3	2868.4
inf	362.1	362.1

(d) *medi-ref*

Figure 4.14: memory requirement for one-to-all searches

ε	mean	max
∞	1.044 00	1.117 20
1.0	1.018 00	1.074 90
0.5	1.010 00	1.079 90
0.25	1.005 00	1.065 70

Figure 4.15: approximation quality for *aegaeis-ref*

As can be expected, the computation on the visibility graph performs slightly worse on *medi* than on *aegaeis*, as the visibility graph has more edges. However, it still performs better than the approximation scheme.

Figure 4.14 lists the memory demand for the different instances of *aegaeis* and *medi*. For *ref* and *unref*, the memory demand after the search (in particular, with the shortest path tree over V_ε stored) quickly grows similarly large as for the computation on the visibilty graph. For *aegaeis-triangle*, memory demand grows more slowly, matching the smaller graph size.

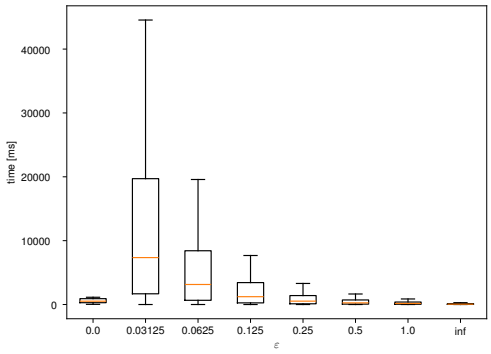
The quality of the entire trees is included only for the *ref* instance, as *triangle* and *unref* slightly differ from the visibility graph in some regions and therefore do not have comparable distances. The quality is computed by iterating over the entire tree and dividing each distance by the corresponding distance computed with the visibility graph. The mean and maximum over these ratios are kept.

For *aegaeis-ref* (see Figure 4.15) the qualities are far better than the theoretic guarantee. Without steiner points, distances are already approximated a factor of at most 1.1172.

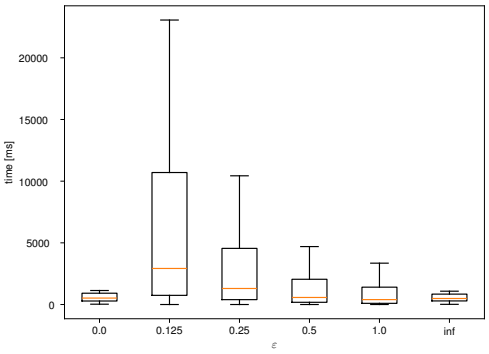
A possible explanation for these low values is that for paths lying between vertex vicinity, the quality bound of $1 + \varepsilon$ applies (when not considering the pruning). These paths can make up larger portions of the full paths and as a result the possible $1 + 2\varepsilon$ derivation can have a weaker effect. Further, due to the property of Delaunay-triangulations having a spanning factor of lower than 2, the graph without Steiner points already provides good results.

4.3.4 One-to-one queries

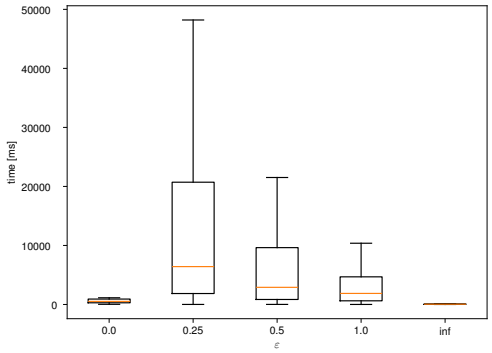
This experiment is intended to measure the performance of computing shortest paths instead of full shortest-path trees on larger instances. For each graph, a set of 100 source-target pairs is randomly selected from the boundary nodes. For these pairs, the shortest paths are computed with A^* based on the beeline distance. These computations are run with *min-angle* pruning and without storing coordinates explicitly.



(a) *aegaeis-triangle*



(b) *aegaeis-ref*



(c) *aegaeis-unref*

Figure 4.16: Boxplots over query time for different values of ε

For each query, the ratio $|\tilde{\Pi}^*|/|\Pi^*|$ is computed to examine the approximation quality. In contrast to one-to-all queries, where the distances are not comparable for all graph instances, the paths for the evaluated queries do not differ between the graph instances.

For 2 of these queries, the implementation did not find a path on the *unref* graph and for 3 on *ref*. It is not known why that is the case, possibly due to an error in the implementation. These appear as ∞ in the quality statistic.

Query times

Figure 4.16 shows distributions over query times for the different tested values of ε. These numbers are also listed in Figure 4.18. These boxplots consist of the median,

lower and upper quartile and the minima/maxima. The exact solution using the visibility graph is denoted by $\varepsilon = 0$, and the triangulation without steiner points by $\varepsilon = \infty$.

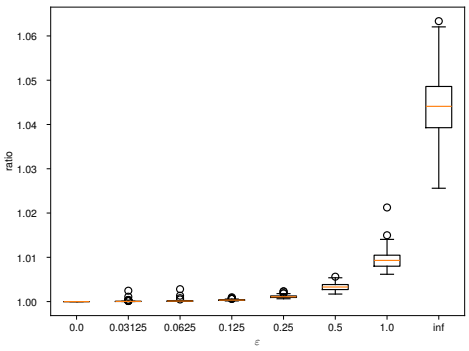
As can be seen, the times greatly increase with lower values for ε, similar to the one-to-all searches. However, the distributions also show that A^* can strongly reduce the query times. For a one-to-all search on *aegaeis-triangle* at $\varepsilon = 1/16$, the maximal time was $\sim 142s$, while for the one-to-one searches the maximum lies at $\sim 62s$, and the median at $\sim 3.14s$. These lower numbers can be explained by the lower numbers of nodes visited, and by the fact that high numbers of Steiner points concentrate on a few edges (as seen in section 4.2). For random shortest path queries and A^*, these edges are less likely to be visited. This shows that A^* can be applied to greatly reduce the size of the search and therefore improve the query times.

The query times also lie closer to the times for the visibility graph, with median query time for *aegaeis-triangle* at $\varepsilon = 0.25$ being almost equal to the median time required on the visibility graph.

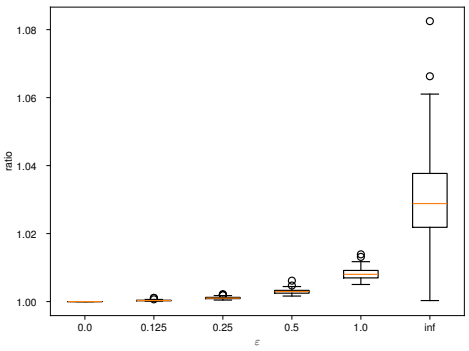
Solution quality

Figure 4.17 shows the distributions of the ratio of the approximated by exact cost $\frac{|\tilde{\Pi}^*|}{|\Pi^*|}$. The exact numbers are listed in Figure 4.19. The theoretical quality guarantee is $1 + 7\varepsilon$ but the measured approximation qualities are closer to 1. For *aegaeis-triangle*, the measured qualities with $\varepsilon = 1$ are better than $(1 + \frac{3}{100}\varepsilon)$, while for $\varepsilon = 0.03125$ the factor is still lower than $(1 + \frac{1}{10}\varepsilon)$. This suggests that the qualities more closely resemble the theoretical guarantee for smaller ε-values.

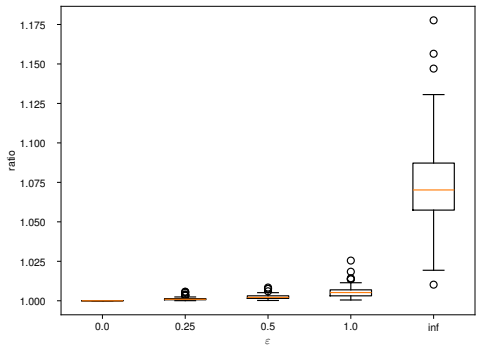
Between the different instances, a quality difference can be seen: For the unrefined graph, solutions are worse than for the refined graphs (with *aegaeis-triangle* delivering the best results).



(a) *aegaeis-triangle*



(b) *aegaeis-ref*



(c) *aegaeis-unref*

Figure 4.17: Boxplots over solution qualities for different values of ε

ε	mean	min	$q_{0.25}$	median	$q_{0.75}$	max
0.0	574.3	26.0	284.3	522.5	916.5	1136.0
0.25	17 682.2	7.0	1854.0	6415.5	20 715.3	154 263.0
0.5	7975.3	11.0	847.0	2903.0	9630.0	70 929.0
1.0	3899.9	5.0	620.5	1872.0	4682.0	32 716.0
inf	52.0	0.0	19.5	46.0	71.0	231.0

(a) *aegaeis-unref*

ε	mean	min	$q_{0.25}$	median	$q_{0.75}$	max
0.0	574.3	26.0	284.3	522.5	916.5	1136.0
0.125	13 765.1	0.0	743.0	2923.0	10 700.5	119 905.0
0.25	6446.5	0.0	395.0	1300.0	4551.5	57 041.0
0.5	2667.0	0.0	180.0	571.5	2047.0	24 920.0
1.0	1473.0	1.0	95.5	398.0	1406.8	11 779.0
inf	543.3	15.0	290.5	489.0	842.8	1080.0

(b) *aegaeis-ref*

ε	mean	min	$q_{0.25}$	median	$q_{0.75}$	max
0.0	574.3	26.0	284.3	522.5	916.5	1136.0
0.03125	17 227.3	0.0	1679.0	7349.0	19 708.0	147 744.0
0.0625	7065.6	1.0	660.3	3137.0	8419.8	62 385.0
0.125	3027.1	0.0	257.0	1226.5	3431.8	25 035.0
0.25	1236.0	0.0	113.0	528.0	1401.0	10 703.0
0.5	592.6	0.0	62.0	260.0	714.8	4733.0
1.0	300.2	0.0	38.3	165.0	386.3	2296.0
inf	93.7	1.0	15.3	56.5	131.8	530.0

(c) *aegaeis-triangle*

Figure 4.18: distributions of query times [ms]

ε	mean	min	$q_{0.25}$	median	$q_{0.75}$	max
0.0	1.000 00	1.000 00	1.000 00	1.000 00	1.000 00	1.000 00
0.25	inf	1.000 07	1.000 63	1.000 95	1.001 38	inf
0.5	inf	1.000 19	1.001 51	1.002 14	1.003 14	inf
1.0	inf	1.000 49	1.003 13	1.005 20	1.006 93	inf
inf	1.074 49	1.010 21	1.057 43	1.070 20	1.087 22	1.177 64

(a) *aegaeis-unref*

ε	mean	min	$q_{0.25}$	median	$q_{0.75}$	max
0.0	1.000 00	1.000 00	1.000 00	1.000 00	1.000 00	1.000 00
0.125	1.000 34	1.000 11	1.000 24	1.000 31	1.000 40	1.001 09
0.25	1.001 07	1.000 45	1.000 87	1.001 02	1.001 25	1.002 19
0.5	1.002 97	1.001 63	1.002 48	1.002 96	1.003 30	1.006 15
1.0	1.008 24	1.005 03	1.006 97	1.008 01	1.009 16	1.013 90
inf	inf	1.000 30	1.021 85	1.028 84	1.037 70	inf

(b) *aegaeis-ref*

ε	mean	min	$q_{0.25}$	median	$q_{0.75}$	max
0.0	1.000 00	1.000 00	1.000 00	1.000 00	1.000 00	1.000 00
0.03125	1.000 09	1.000 01	1.000 02	1.000 04	1.000 06	1.002 46
0.0625	1.000 17	1.000 05	1.000 09	1.000 11	1.000 16	1.002 81
0.125	1.000 36	1.000 15	1.000 29	1.000 34	1.000 41	1.000 98
0.25	1.001 14	1.000 64	1.000 93	1.001 09	1.001 28	1.002 33
0.5	1.003 31	1.001 70	1.002 72	1.003 29	1.003 85	1.005 62
1.0	1.009 57	1.006 17	1.007 99	1.009 30	1.010 48	1.021 26
inf	1.043 80	1.025 60	1.039 28	1.044 10	1.048 59	1.063 34

(c) *aegaeis-triangle*

Figure 4.19: distributions of solution qualities

5 Future work

The presented implementation is developed purely for the 2D Euclidean shortest path problem, but could be extended to more general cases.

Firstly, the implementation could be extended to support planar subdivisions consisting of faces other than triangles. As described in subsection 2.6.3, these could be more well suited for the described approximation scheme. This would require some modifications to the presented implementation: For a face with a higher number of sides, the middle point M as described in the theory section is not defined uniquely. One would have to compute the two outermost points M, M' and place Steiner points equidistantly between them. Further, modifications to the data structure for the edge-to-edge relationships would be necessary. As seen in subsection 4.2.1, inner angles have a great effect on the graph sizes, and as stated in section 2.6, faces with more sides can have larger inner angles. Therefore, this approach could drastically reduce graph sizes.

Further, the implementation could be extended to use weighted faces, which would require the simplifications of section 2.7 to be adapted and a data structure for the face weights to be added. On a weighted domain, the approach of using a visibility graph is not directly applicable and therefore this approximation scheme might be more relevant for that problem.

Approximating shortest paths in 3D space is possible too, [AMS00] present a way to derive a discretization from a tetrahedralization, also supporting weighted tetrahedra. The discretization places Steiner points on faces, leaving vertex and edge-vicinities empty. As they state, the number of points on a face scales with $\varepsilon^{-3} \ln \varepsilon^{-1}$, and in consequence, an implicit graph representation would likely be required for an implementation with reasonable memory demand.

As the approximation scheme mainly relies on a Dijkstra search, the known speed-up techniques can potentially be applied too. Guided search with A^* has already been applied in this implementation. To reduce the sizes of search trees, one could also use A^* with landmarks. A^* with landmarks cannot be trivially applied to a discretization G_ε, because that would require precomputing landmark distances for all Steiner points and therefore require too much memory. One would have to come up with a way to compute heuristic values for Steiner points from the landmark distances of vertices of $V_\mathcal{P}$, and ensure that these do not overestimate distances.

6 Conclusion

This thesis provides an implementation for the approximation scheme described by [AMS00]. The general definitions have been translated into an implicit graph representation. Estimates for the scaling of the discretization in properties of $\mathcal{P}$ (mainly the inner angles) have been provided and experimentally confirmed. Further, an existing method of pruning the search is adapted and simplified using properties of Euclidean shortest paths in an unweighted domain. The experiments have shown that such pruning is necessary for obtaining acceptable query times. The experiments have also shown that the implicit graph representation poses a $\sim 30\%$ overhead for the presented implementation.

The approximation quality on the evaluated problem instances has been shown to be far better than the theoretical guarantee of $(1 + 7\varepsilon)$. By more thorough analysis, one could possibly show better quality guarantees for the unweighted problem. Having a better quality guarantee would allow for using smaller discretizations.

In the following, the main advantages and disadvantages are summarized.

6.1 Advantages and shortcomings of the approximation scheme

The size of the discretization has been shown to have a strong dependency on inner angles of $\mathcal{P}$. Because of that, for suboptimal graph instances (like the unrefined graphs), the size of the discretization graph can grow too fast for low query times to be possible. These high graph sizes also prevent explicit graph representations to be possible with acceptable memory demands. In consequence, the presented approximation scheme requires triangulations to provide lower bounds on inner angles.

For the presented instances, performing searches on an explicitly stored visibility graph can provide far lower query times, at the expense of higher space demand. On the other hand, the possibility of using a space-efficient implicit graph representation can make the approximation scheme more suitable for certain use cases, e.g. when a graph has a large number of pairwise visible vertices.

Another advantage of the approximation scheme is that it can be applied to more general problems than a visibility graph, as seen in the previous section. In particular, for problems where a visibility graph cannot be used, this approximation scheme might be more relevant.

6.2 Final remarks

As shown in this thesis, the approximation scheme can be implemented and applied to moderately sized planar subdivisions. The implementation provided here shows far inferior performance to computations on visibility graphs especially when computing entire shortest-path-trees. Even one-to-one query times on the examined instances quickly exceeding the 10s mark, in contrast to query times below 0.5s for a visibility graph.

This low performance is mainly due to the large size of the discretizations. A promising approach for reducing these graph sizes is the usage of more general polyhedral surfaces, with faces other than triangles, as these could greatly increase the inner angles and thus reduce graph sizes. Further, if better quality guarantees for the unweighted Euclidean shortest path problem could be shown, smaller discretizations could be used.

The approximation scheme's lightweight implicit graph representation can make it applicable for instances where a visibility graph would grow too large to be stored explicitly. Also, this approximation scheme can be applied to more general problems where a visibility graph does not suffice. For these problems (mainly the weighted and three-dimensional variants), this approximation scheme might be more relevant for further research.

Bibliography

[ACG+12] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, M. Protasi. *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer Science & Business Media, 2012 (cit. on p. 6).

[AMS00] L. Aleksandrov, A. Maheshwari, J.-R. Sack. "Approximation algorithms for geometric shortest path problems." In: *Proceedings of the thirty-second annual ACM symposium on Theory of computing*. 2000, pp. 286–295 (cit. on pp. 5, 7, 8, 10–18, 21, 22, 24, 25, 61, 62).

[GM91] S. K. Ghosh, D. M. Mount. "An Output-Sensitive Algorithm for Computing Visibility Graphs." In: *SIAM Journal on Computing* 20.5 (1991), pp. 888–910. DOI: 10.1137/0220055. eprint: https://doi.org/10.1137/0220055. URL: https://doi.org/10.1137/0220055 (cit. on p. 6).

[HS99] J. Hershberger, S. Suri. "An Optimal Algorithm for Euclidean Shortest Paths in the Plane." In: *SIAM Journal on Computing* 28.6 (1999), pp. 2215–2256. DOI: 10.1137/S0097539795289604. eprint: https://doi.org/10.1137/S0097539795289604. URL: https://doi.org/10.1137/S0097539795289604 (cit. on p. 7).

[MS04] J. S. B. Mitchell, M. Sharir. "New results on shortest paths in three dimensions." In: *Proceedings of the Twentieth Annual Symposium on Computational Geometry*. SCG '04. Brooklyn, New York, USA: Association for Computing Machinery, 2004, pp. 124–133. ISBN: 1581138857. DOI: 10.1145/997817.997839. URL: https://doi.org/10.1145/997817.997839 (cit. on p. 7).

[Sch24] A. Schneewind. *Euclidean shortest path approximation*. Mar. 2024. URL: https://github.com/AxelSchneewind/euclidean-shopa-approximation.git (cit. on p. 27).

[She96] J. R. Shewchuk. "Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator." In: *Applied Computational Geometry: Towards Geometric Engineering*. Ed. by M. C. Lin, D. Manocha. Vol. 1148. Lecture Notes in Computer Science. From the First ACM Workshop on Applied Computational Geometry. Springer-Verlag, May 1996, pp. 203–222 (cit. on p. 40).

[Xia11] G. Xia. “The Stretch Factor of the Delaunay Triangulation Is Less Than 1.998.” In: *CoRR* abs/1103.4361 (2011). arXiv: 1103.4361. URL: http://arxiv.org/abs/1103.4361 (cit. on p. 20).

All links were last followed on May 15, 2024.

Declaration

I hereby declare that the work presented in this thesis is entirely my own. I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted hard copies.

---

place, date, signature