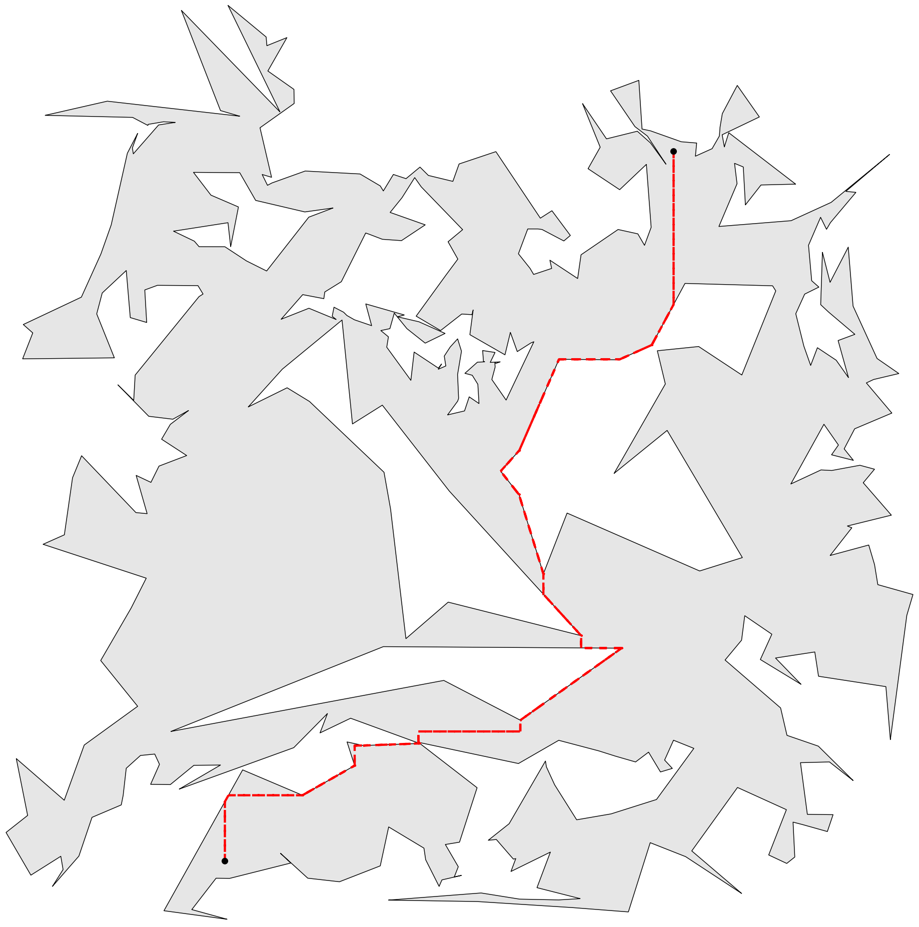
Datastructures for approximate shortest paths queries in polygons with holes

July 2022

Erwin Glazenburg

Supervisor: Frank Staals
Second supervisor: Maarten Löffler



Contents

1 Introduction **4**

2 Preliminaries **4**
- 2.1 Manhattan vs Euclidean distance . . . 4
- 2.2 Approximation algorithms . . . 4

3 Research Question **5**

4 Related work **5**
- 4.1 Algorithmic problem . . . 5
- 4.2 Euclidean shortest paths . . . 6
  - 4.2.1 Simple polygons . . . 6
  - 4.2.2 Polygons with holes . . . 6
  - 4.2.3 Sensitive to the number of holes . . . 7
  - 4.2.4 Approximate shortest paths . . . 7
- 4.3 Manhattan distance . . . 7
  - 4.3.1 Visibility graph . . . 7
  - 4.3.2 Sparse visibility graph . . . 8
  - 4.3.3 Preprocessing . . . 10
  - 4.3.4 Improving the algorithm . . . 10

5 Theory **10**
- 5.1 Overview . . . 10
- 5.2 Cones . . . 10
- 5.3 Approximation factor . . . 11
- 5.4 Sawtooth structure . . . 14
- 5.5 Cutline trees . . . 15
- 5.6 Steiner points . . . 16
  - 5.6.1 Type 1 Steiner points . . . 16
  - 5.6.2 Type 2 Steiner points . . . 17
- 5.7 Preprocessing . . . 17
- 5.8 Gateways . . . 18
- 5.9 Querying . . . 20
- 5.10 Final result . . . 21

6 Implementation **21**
- 6.1 Exact numbers . . . 21
  - 6.1.1 Non-rational values . . . 21
- 6.2 Ray shooting . . . 22
- 6.3 k-directional rayshooting . . . 22
- 6.4 Arbitrary direction rayshooting . . . 22
  - 6.4.1 Point line duality . . . 23
  - 6.4.2 Dual arrangement of polygon . . . 23
  - 6.4.3 Storing $E_{intersect}$ . . . 24
  - 6.4.4 Custom fully persistent binary search tree . . . 24
- 6.5 Suboptimal query implementation . . . 25
- 6.6 All pair shortest paths . . . 25
- 6.7 Optimization . . . 26
- 6.8 Naive datastructure . . . 26

7 Methods **27**

7.1 Dataset . . . 27

7.1.1 Fast Polygon Generator (FPG) . . . 27

7.1.2 Simple Polygon Generator (SPG) . . . 27

7.1.3 Super Random Polygon Generator (SRPG) . . . 27

7.1.4 Pacman polygon . . . 28

7.1.5 Query points . . . 28

7.2 Experiments . . . 28

7.2.1 Main experiments on Salzburg database and ARPs . . . 28

7.2.2 Breakdown of time usage . . . 30

8 Results and analysis **31**

8.1 Query times . . . 31

8.1.1 Non-trivial queries . . . 31

8.1.2 Naive oracle . . . 31

8.1.3 Sawtooth oracle . . . 31

8.1.4 Comparison . . . 32

8.2 Preprocessing . . . 34

8.2.1 Build time . . . 34

8.2.2 Build space . . . 34

8.3 Approximation factor . . . 35

9 Conclusion **38**

9.1 Future work . . . 38

Abstract

Given a polygon P, we might be interested in finding a $(1+\epsilon)$-approximate shortest path between two points in P. This thesis considers the problem of creating a datastructure that can quickly answer such queries. Our resulting datastructure is of size $O(\frac{n^2}{\epsilon^2} 4^{\sqrt{\log n}} \log n)$ and can answer queries in $O(\frac{\log n}{\epsilon^2})$ time. Along with a theoretical proof of these complexities we also implement our datastructure and conduct experiments in which we compare our datastructure to a naive one. We find that our datastructure has poor scalability with respect to preprocessing, but that it does give good approximations much faster than the naive approach, especially in the worst case.

1 Introduction

The shortest path problem is very well known: given a polygon and some starting location p and ending location q in that polygon, what is the shortest path between p and q that does not intersect any of the polygons edges? As an extension to this problem, we assume our polygon can have holes, and that we will not get one query pair (p, q) but many of them. In this case it would be inefficient to calculate every one of these paths from scratch. In this thesis, we focus on the problem of how to preprocess a polygon such that shortest path queries can be answered quickly. It turns out it is difficult to give an exact answer to this question [1]. Instead we can find an approximate shortest path, a path with a length close to the length of the optimal shortest path, more quickly. In particular we will adapt an existing technique from [2], used to calculate exact Manhattan shortest paths, to get a datastructure that can answer $(1+\epsilon)$-approximate Euclidean shortest paths queries.

This can be a useful datastructure in many scenarios, for example when doing a simulation study with a large number of agents that can move around. They will all want to calculate the path they need to take to their destination, so there will be a lot of shortest path queries which can be answered quickly using the datastructure. The fact that it returns an approximate shortest path and not an exact one might not a problem, if the increase in travel time of a few percent makes little difference in the simulation as a whole. One could also think of more real-world applications, such as sorting robots that move around in large warehouses. Depending on the size of the warehouse and the chosen ϵ the time saved on computation could be larger than the time lost in actual travel time.

First in Section 2 we will discuss some necessary background information. Then in Section 3 we will define the research questions to be answered by the thesis, and in Section 4 results from other relevant papers will be discussed. Section 5 introduces our new datastructure, and Section 6 elaborates on how this was implemented. The setup for our experiments is described in Section 7, and finally their results are shown and analysed in Section 8.

2 Preliminaries

2.1 Manhattan vs Euclidean distance

There are many methods of measuring distance between points, and the two methods relevant to this study are Euclidean distance (L_2) and Manhattan distance (L_1). The Euclidean distance between two points p and q is calculated as $\delta(p, q) = \sqrt{(p_x - q_x)^2 + (p_y + q_y)^2}$, and the Manhattan distance is simply $\delta(p, q) = |p_x - q_x| + |p_y - q_y|$.

Although the two are approximations of each other, it is possible that a Manhattan shortest path between two points is not a Euclidean shortest path, nor the other way around. An example of this is shown in Fig. 1, where the red line is a Manhattan shortest path and the blue line is a Euclidean shortest path.

2.2 Approximation algorithms

Some algorithms will give an exact answer to the shortest path problem, meaning there is no valid path shorter than the path such an algorithm returns. Other algorithms can give an approximation

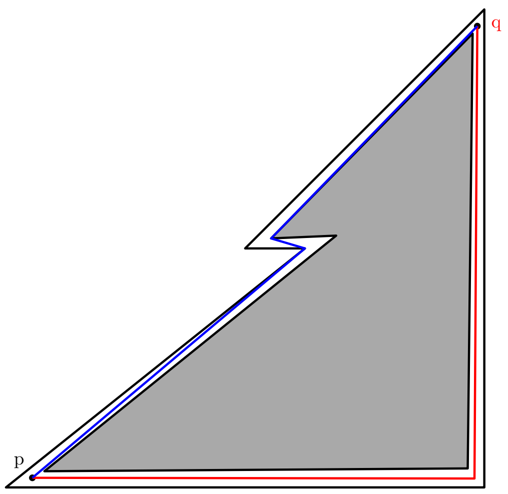


Figure 1: Manhattan shortest path vs Euclidean shortest path

of this shortest path. For some query (p, q) we say the length of an optimal shortest path between them is l^*. An *ϵ-approximate* shortest path datastructure will return a path with a length l that differs no more than a factor ϵ from the optimum l^*, so $l^* \leq l \leq \epsilon * l^*$.

3 Research Question

In this research we will be creating a new datastructure to answer approximate Eurlicdean shortest path queries, based on a datastructure to answer exact Manhattan shortest path queries. The main research question we are trying to answer is:

How efficient of a datastructure can we build to answer $(1+\epsilon)$-approximate Euclidean shortest path queries?

For this we will want to answer the following subquestions:

What space and time complexities can we achieve in theory?
What space and time usage can we achieve in practice?
What approximation factor does this datastructure have in practice?

4 Related work

In this section related work about shortest path queries will be discussed .

4.1 Algorithmic problem

In this study we are mostly concerned with how to preprocess a polygon to be able to quickly query shortest paths. However the algorithmic version of the problem, where we can not do any preprocessing and simply have to find the shortest path from scratch, is a good place to start. According to the Handbook of Computational Geometry [3], there are two main approaches to this problem.

The first is based on the visibility graph. This graph has the polygon vertices as nodes, and edges for all pairs of vertices that can see each other. We can also add the query points p and q to this graph by adding them as ‘point holes‘. We know that a shortest path in a polygon consists of straight lines between query points and polygon vertices. We can therefore use Dijkstra’s algorithm to find the shortest path between them in the visibility graph, which is also the shortest path in the polygon.

This visibility graph has $O(n^2)$ edges, since in the worst case all vertices can see all other vertices. However there are output sensitive algorithms to construct it: if the graph has E_{VG} edges, it is possible to construct it in optimal $O(E_{VG} + n \log n)$ time and $O(n)$ space [4, 5]. The shortest path can then be found in $O(E_{VG} + n \log n)$ time.

The second approach is the continuous Dijkstra method, which simulates a wavefront propagating from the query point such that all points on the wavefront have the same distance to the query point. This technique is similar to a sweepline. We start with the wavefront at distance 0, and at certain 'events' it changes structure, for example when it hits a vertex. When we hit the other query point we are done, since we know the distance of the wavefront at that point. With this technique we can find the distance in $O(n^{1.5+\epsilon})$ time and $O(n)$ space, or in $O(n \log n)$ time with $O(n \log n)$ space.

Very recently this problem has been solved optimally by Wang in [6], in $O(n + h \log h)$ time and $O(n)$ space using the continuous Dijkstra method, where h is the number of holes in the polygon.

4.2 Euclidean shortest paths

In this section we look at datastructures that can answer Euclidean shortest path queries.

4.2.1 Simple polygons

For simple polygons (polygons without holes) this problem is, unsurprisingly, much simpler than for polygon with holes. Guibas and Herschbringer [7] give a datastructure of size $O(n)$ that can answer shortest path queries in $O(\log n)$ time for simple polygons, which are optimal complexities. They first make a triangulation of the polygon by repeatedly splitting it into halves that are roughly the same size, creating a tree of subpolygons in the process. For each of these subpolygons they store information about shortest paths inside it. This information can then be used to answer queries by finding the subpolygons between the query points, and combining the subpaths from those subpolygons.

4.2.2 Polygons with holes

So for simple polygons we only need linear space to get logarithmic query times. If we introduce holes, the problem gets a lot more difficult. Chiang and Mitchell give several different datastructures with tradeoffs between space and query time in [1], and the one we look at here uses $O(n^{11})$ space and has $O(\log n)$ query time.

Their method uses *shortest path maps*, which is a way to store shortest paths from a single point to the rest of the whole polygon. For some point z, the shortest path map $SPM(z)$ of z is the division of the polygon into maximal regions such that for any two points p and q in that region, the combinatorial path from z to p is the same as from z to q. It makes sense that for some point z' that is close to z, $SPM(z')$ will be similar to $SPM(z)$. This is indeed what the authors try to exploit: they divide the polygon into regions, such that all points in the region have a similar shortest path map. 'Similar' here formally means that they are topologically equivalent, so both shortest path maps have the same number of regions in about the same place, but the actual geometry of those regions may differ depending on the exact location of its source point.

First the polygon is divided into those regions with similar shortest path maps. Chiang and Mitchell prove that $O(n^{10})$ is an upper bound to the number of such regions. The shortest path map is then stored for every region, but in a parametrized way: we can not just store the exact geometry of the map because it is different for every source point. They also created a parametrized point location datastructure, to quickly find in which region of such a parametrized map a point lies in $O(\log n)$ time. This leads to a total space complexity of $O(n^{11})$.

It is unknown if this complexity can still be improved on, as pointed out by the authors, for example by improving the upper bound $O(n^{10})$ on the number of SPM-similar regions. However, it is still unlikely to be usable in real world settings when the polygons are not extremely small.

4.2.3 Sensitive to the number of holes

A way to make more practical datastructures is by making them sensitive to the number of holes in the polygon. Let the number of holes be h. A polygon with very few holes ($h = O(1)$) can be easier to handle than one with very many holes ($h = O(n)$).

In the same paper as above [1], Chiang and Mitchell also provide a datastructure with $O(n+h^5)$ space, and $O(h * \log n)$ query time. For polygons with $O(1)$ holes this is optimal since it has linear space and logarithmic time, just like the algorithm in [7] for simple polygons. However for polygons with $O(n)$ holes it degenerates to $O(n \log n)$ query time which is very slow. Another datastructure with $O(h \log n)$ query time that uses $O(n^2)$ space is given by Guo et al. in [8].

4.2.4 Approximate shortest paths

The above algorithms give good results with few holes, but with a linear number of holes they might still not be very practical. A possible way to handle this is by accepting approximate answers instead of exact answers. This is indeed also what our new proposed algorithm will do.

In [9], Thorup proposes a $(1+\epsilon)$-approximate datastructure, which uses $O(\frac{n \log^2 n}{\epsilon})$ space and can answer queries in $O(\frac{1}{\epsilon^3} + \frac{\log n}{\epsilon * \log \log n})$. The user can freely pick the parameter ϵ as any number greater than 0. The smaller the ϵ, the more accurate the answer will be, but the larger the space and time requirements will be.

Their datastructure is based on dividing the view of points into k *cones* where k is $O(\frac{1}{\epsilon})$, meaning that the more accurate we want our answer the more cones we need. They want to know for every vertex and for every one of its cones, what the closest visible vertex in that cone is. They construct a *cone graph*, whose vertices are the obstacle vertices, and where vertices u and v are connected iff v is the closest visible vertex to u in some cone direction. A path between two vertices in this graph will then correspond to a path in the polygon, since only vertices that are visible to each other are connected. It is proven in [10] that the shortest path in the graph is a $(1 + O(\frac{1}{k}))$-approximation of the shortest path in the polygon.

This can be used to create a query method: in the preprocessing they create a Voronoi-diagram like structure, which can be used to find the closest vertex in each cone direction for some point in $O(\frac{\log n}{\log \log n})$ time. For a query (p, q) we can then find the k closest points, one in each direction, for both p and q. The shortest path through the cone graph goes through one of these points for both p and q. We will store all pair (approximate) distances between obstacle vertices, so we can look at all pairs of cone-points of p and q and find the pair that gives the shortest total distance between p and p. Finding the k closest points will take $O(k * \frac{\log n}{\log \log n}$ time, and looking at all pairs takes $O(k^2)$ time. The easiest way to store these all pair shortest distances is using $O(n^2)$ space by storing a shortest path tree for every vertex, but Thorup gives a data structure that can do so in $O(\frac{n \log n}{\epsilon})$ space, or $O(\frac{n \log^2 n}{\epsilon})$ if we also want to retrieve the actual path in addition to the its length. This datastructure has $O(k)$ query time, leading to the earlier mentioned $O(\frac{1}{\epsilon^3} + \frac{\log n}{\epsilon * \log \log n})$ query time (since $k = O(\frac{1}{\epsilon})$).

4.3 Manhattan distance

Creating an oracle for finding Manhattan shortest paths seems to be a simpler problem than for Euclidean shortest paths, as we will see in this section. Many of the techniques used in the following papers will be adapted to be used in our new proposed algorithm, so they are covered somewhat extensively.

4.3.1 Visibility graph

In [2], Clarkson et al. solve the offline version of the shortest path problem, meaning we are not allowed to do any preprocessing and simply have to find the shortest path between two query points in an unseen polygon. For this they first introduce their *visibility graph* $\overline{VIS}(\overline{V}, \overline{E})$. $\overline{V}$ is the set of all polygon vertices and $\overline{E}$ contains pairs of mutually visible vertices, but not all mutually visible vertices are in $\overline{E}$. Note that this makes it different from a 'standard' visibility graph where all mutually visible vertices are connected. Shortest paths in this graph should correspond to shortest paths in the polygon, so their idea is to construct the visibility graph and use Dijkstra's algorithm to find a shortest path.

For a point p we look at its four quadrants separately. In this explanation we will only look at the first quadrant, so at all points above and to the right of p, but the same holds for the other three quadrants. Point p will be connected to all vertices of its *staircase*, shown in Fig. 2 as the black dots. A vertex s is in p's staircase if it is not *dominated* by any other vertex, meaning there is no vertex in the rectangle between p and s. For each of the black dots this is true, while it is not for the red dot r since point s_1 is in the rectangle between it and p. This means we can get from p to r through s_1 without increasing the length of the path, in the Figure this means the blue line is the same length as the red line (recall that we are using the Manhattan distance). Now a shortest path in $\overline{VIS}(\overline{V}, \overline{E})$ is also a Manhattan shortest path in the polygon, since every part of the path can pass through the staircase vertices without changing the length of the path.

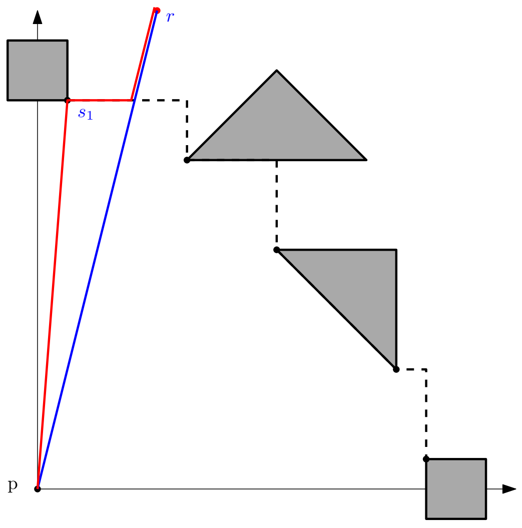


Figure 2: The staircase around a point p

With this definition a vertex can have $O(n)$ staircase vertices, for example in Fig. 3 all the black dots are staircase vertices of p. This also means that $\overline{E}$ can be size $O(n^2)$, and we would like it to be smaller. A sparse version of $\overline{VIS}(\overline{V}, \overline{E})$, $VIS(V, E)$, will be defined.

4.3.2 Sparse visibility graph

To make the visibility graph sparse, extra points called *Steiner points* are introduced. *Type 1 Steiner points*[1] are the easiest: we project each vertex in the four directions up, down, left and right onto the polygon, and these projections are type 1 Steiner points. Since every vertex introduces exactly 4 type 1 Steiner points, there are $O(n)$ of them.

For the *type 2 Steiner points* the *cutline tree* is introduced. The cutline tree is a binary tree, where a set of vertices and a cutline are associated with every node. The cutline is a vertical line through the median of the vertex set. This splits that set in a left and right half, which are the vertex sets of the left and right child in the cutline tree, respectively. The vertex set of the root of the tree is the set of all polygon vertices, meaning the tree will have height $O(\log n)$. Every node of the tree introduces the following type 2 Steiner points: for every vertex in the vertex set, if it is horizontally visible to the cutline, we project it horizontally onto the cutline and create a type 2 Steiner point there. Since every vertex is in exactly one vertex set on every level of the tree, and the tree has height $O(\log n)$, there are $O(n \log n)$ type 2 Steiner points.

Now $VIS(V, E)$ is created as follows. V is the set of all polygon vertices plus all Steiner points. Then every polygon vertex is connected to its 4 type 1 Steiner points, and its $O(\log n)$ type 2 Steiner points. Lastly for every cutline and obstacle edge, if a and b are two adjacent points visible

[1]note that the words 'type 1' and 'type 2' are interchanged in some papers; I use a consistent notation here

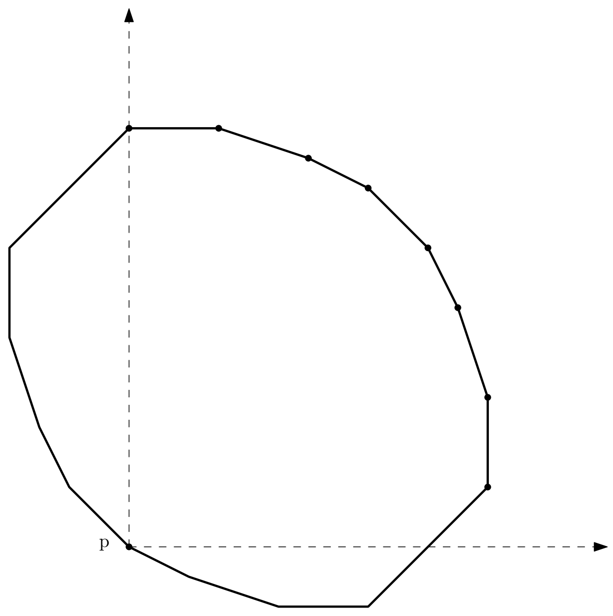


Figure 3: p has $O(n)$ staircase vertices

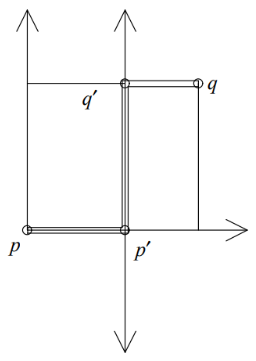
to each other on that line, we connect a and b. Each Steiner point created by a point p is thus connected to 2 other Steiner points. These are called the *gateways* of p.

Now for an arbitrary vertex p and one of its staircase vertices q, p and q are still indirectly connected with a shortest path, either via a cutline or via an obstacle edge. There are two cases:

Case 1: if no line segment intersects the rectangle between p and q they are connected through a cutline like on the left of Fig. 4. This is because there is a place in the cutline tree where p and q were in the same set on level i, but in seperate sets on level $i+1$. That means that the cutline associated with their node on level i separated them, so they will both have created type 2 Steiner points on this cutline. Since all adjacent points on the cutline are connected, p and q are then also connected with a shortest path.

Case 2: if a line segment does intersect the rectangle between p and q they are connected through that edge, since they both created type 1 Steiner points on that edge and all adjacent points on the edge are connected.

With that we know that shortest paths in $VIS(V, E)$ are also Manhattan shortest paths in the polygon.



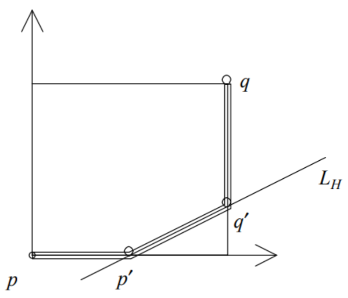


Figure 4: Paths from p to q. Left: case 1 through a cutline, right: case 2 over an edge

4.3.3 Preprocessing

In [2] the sparse visibility graph was only used to solve the offline problem, but in [11] Chen et al. use it to create a datastructure of size $O(n^2 \log^2 n)$ that can answer Manhattan shortest path queries in $O(\log^2 n)$ time. The preprocessing consists of calculating the sparse visibility graph of all obstacle vertices and storing it and the cutline tree used to create it. We also store the shortest path between all pairs of Steiner points in $O(n^2 \log^2 n)$ space, since there are $O(n \log n)$ total Steiner points. Then for a query (p, q) we 'insert' p and q into the graph and find their Steiner points, and with those their gateways (the points adjacent to their Steiner points). There are four type 1 Steiner points which can be found in $O(\log n)$ time, and $O(\log n)$ type 2 Steiner points which can be found in $O(\log n)$ time as well with a fractional cascading structure. We know that if p and q and their Steiner points would have been in the visibility graph, the shortest path in this graph would have corresponded with a shortest path between them in the polygon. That shortest path in the graph would pass through one p's Steiner points and then one of p's gateways. The same holds for q. This means we can find the shortest path between p and q by looking at the $O(\log^2 n)$ combinations of their gateways, and choosing the one combination that gives the smallest total distance.

4.3.4 Improving the algorithm

In [12] Chen et al. build on their algorithm given in [11], first by improving the query time from $O(\log^2 n)$ to $O(\log n)$. They do so by slicing the cutline tree into $\sqrt{\log n}$ horizontal slices, called *super levels*. Now instead of including one pair of gateway points per level of the cutline tree, they include one pair of gateway points per superlevel. This reduces the number of gateways of p and q to $O(\sqrt{\log n})$, and reduces the number of combinations to $O(\log n)$.

They also improve the space complexity from $O(n^2 \log^2 n)$ to $O(n + h^2 * \log^2 h * 4^{\sqrt{\log h}}) = O(n + h^{2+\delta})$ for some small $\delta > 0$.

5 Theory

In this section we will introduce our datastructure for answering approximate shortest path queries.

For a query we can first check if the query points are mutually visible in $O(\log n)$ time by shooting a single ray, in which case the shortest path is a straight line between them. Otherwise there must be at least one polygon vertex on the shortest path, which we assume throughout the proof.

5.1 Overview

Our datastructure is an adaptation of the datastructure used for the exact shortest Manhattan distance [2, 11, 12], which used cutline trees and a sparse visibility graph.

We divide the view of a point p into k *cones*, and for each such cone define a set of *sawtooth vertices* such that to any other point q there is an $f(k)$-approximate path that goes through a sawtooth vertex of p. We then introduce two types of *Steiner points*, and use them to make a set of *gateway* points for the query points. We will prove that we can get to all sawtooth vertices, and therefore to any point in the polygon, with $f(k)$-approximate paths through these gateway points.

The preprocessing will take $O(n^2 k^2 4^{\sqrt{\log n}} \log n)$ space and a similar amount of time to build. A query will take $O(k^2 \log n)$ time.

5.2 Cones

While in the Manhattan case we work with 4 quadrants, in the Euclidean case we will work with k *cones*, similar to the cones of Thorup in [9]. Let $\theta_k = \frac{360}{k}$, and for any integer $j \leq k$ let direction $d_j = j * \theta_k$. Then for some point p, cone i will span between directions $d_i = i * \theta_k$ and $d_{i+1} = (i + 1) * \theta_k$ from p. We can make these cones arbitrarily narrow by increasing the number of cones k. The number of cones k has to be even.

We say a point a *i-dominates* another point b if b is inside the $i'th$ cone from a. For example in Fig. 5 a i-dominates b (and c), while b i-dominates c. The i in 'i-dominates' can be omitted if the cone direction is clear from context.

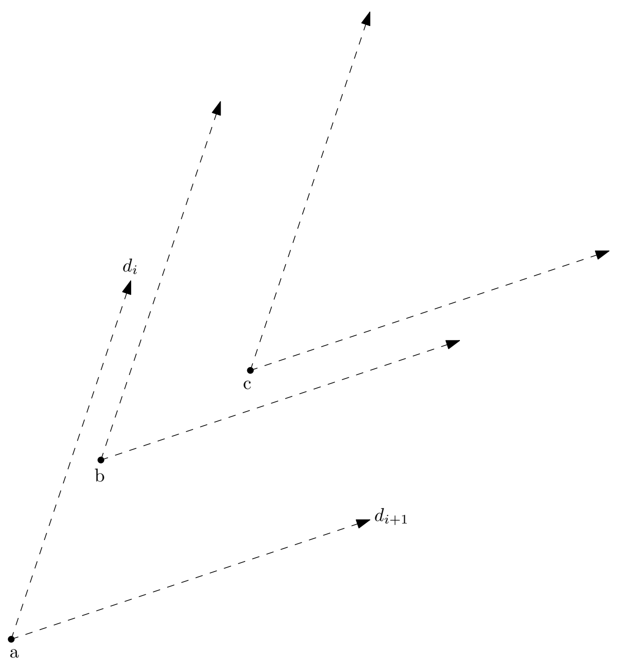


Figure 5: Three points with their cones in direction i

5.3 Approximation factor

An x-monotone path is a path that is either always non-decreasing or always non-increasing in x, a y-monotone path is defined similarly. In Fig. 6 we see from left to right an x-monotone path, a y-monotone path, and an xy-monotone path. A monotone path is *polygonal* if it only consists of straight segments; the first two paths in Fig. 6 are polygonal, the third is not.

Another way to define xy-monotone paths is as follows. Let a, b be any two points on the path, such that b is further along the path than a. If for all such pairs b is in the same quadrant of a, the path is xy-monotone. For example on the right of Fig. 6, the path from p to q is xy-monotone because any point b is always in the first quadrant of any earlier point a.

Similarly we can define an *i-monotone path.* Intuitively, a path is i-monotone if it only goes in directions between d_i and d_{i+1}. More formally, let a, b again be any two points on the path such that b is further along the path than a. If for all such pairs a i-dominates b (b is in cone i of a), the path is i-monotone. For example in Fig. 7 the blue path is i monotone, but the red path is not since r_2 is further along the path than r_1 but r_2 is not i-dominated by r_1.

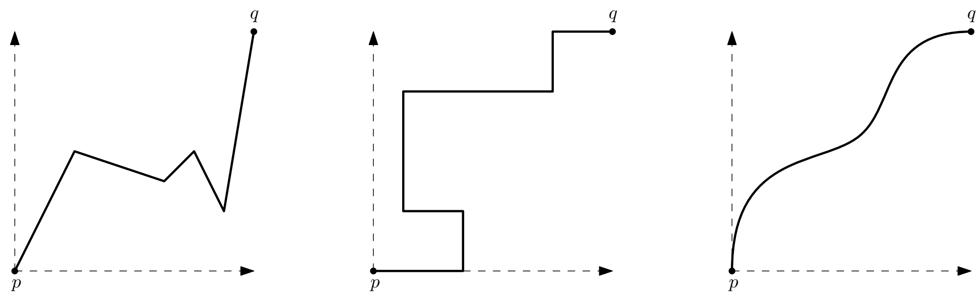


Figure 6: From left to right: an x-monotone path, a y-monotone path, and an xy-monotone path from p to q

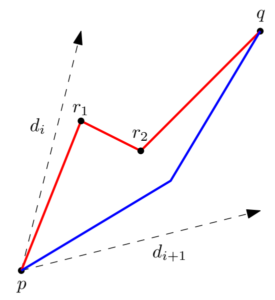


Figure 7: A polygonal i-monotone path in blue, and a polygonal non-i-monotone path in red

The more cones we have, the narrower they get, the more i-monotone paths start to resemble straight lines. This brings us to the key idea of the proposed datastructure:

Lemma 1. *Let a, b be two points such that a i-dominates b. Then any polygonal i-monotone path from a to b is an $f(k)$-approximation of the direct path $\langle a, b\rangle$, where $f(k) = \frac{1}{\cos(\frac{\theta_k}{2})}$.*

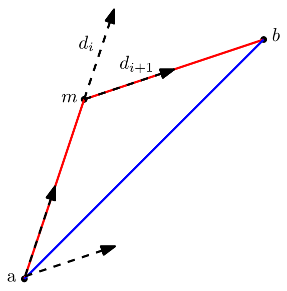


Figure 8: The worst case for points a, m, c when approximating the optimal path from a to c through m

Proof. To prove a bound for our approximation factor we need to consider the worst possible polygonal i-monotone path, such that all other polygonal i-monotone paths yield a better approximation factor. This worst case path goes an equal distance in directions d_i and d_{i+1}, and in no other direction, shown as a red path in Fig. 8. W.l.o.g. we assume this path consists of exactly one segment in direction d_i and one in direction d_{i+1}, because any polygonal i-monotone path with more segments in directions d_i and d_{i+1} can be rearranged to have two segments without changing the length of the path. This is the worst case because of two reasons:

- (1): for any two points a and b, the longest polygonal i-monotone path between them goes only in directions d_i and d_{i+1}. We can prove this by contradiction. Assume there is a pair of points a, b for which the longest polygonal i-monotone path contains at least one segment in a direction that is not d_i or d_{i+1}. Let the endpoints of this segment be c and d, as shown in blue in Fig. 9. We can replace this segment by a segment ce in direction d_i and a segment ed in direction d_{i+1}. This results in a path that is still polygonal and i-monotone, but with a larger length since $|\langle c, e, d\rangle| > |\langle c, d\rangle|$ by triangle inequality. This leads to a contradiction, since we assumed the path containing segment cd was the longest, proving that for any two points a and b, the longest polygonal i-monotone path between them goes only in directions d_i and d_{i+1}

- (2): we achieve the worst possible approximation factor if b is exactly in the middle of the i-th cone of a. To prove this we will formulate the approximation factor as a function of the angle α between d_i and $\overline{ab}$. Let the other angles of triangle amc be β and γ, and let the lengths of the sides be x, y and z, like in Fig. 10. Now our approximation factor as a function of α is $\epsilon(\alpha) = \frac{x+y}{z}$, so we need to find the lengths of x, y and z. We can use the *law of sines* here, which states that for any triangle the length of a side divided by the

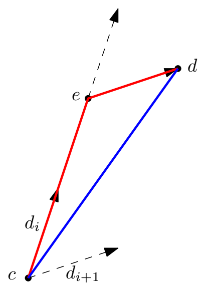


Figure 9: Any path between p and q that does not use angles d_i and d_{i+1}, like the blue path, results in a better approximation factor than the red path.

sine of the angle of the opposite corner is a constant, meaning that ratio is the same for all three corner/side pairs. In our case this means $\frac{\sin(\alpha)}{y} = \frac{\sin(\beta)}{x} = \frac{\sin(\gamma)}{z} = S$, where S is some constant. Rewriting this and filling it in our formula for ϵ gives:

$$\epsilon(\alpha) = \frac{x+y}{c} = \frac{\frac{\sin(\beta)}{S} + \frac{\sin(\alpha)}{S}}{\frac{\sin(\gamma)}{S}} = \frac{\sin(\beta) + \sin(\alpha)}{\sin(\gamma)}$$

Recall that we want to maximize our approximation factor ϵ. Recall Θ_k is the angle between d_i and d_{i+1}. Since $\gamma = 180 - \theta_k$ is constant with respect to α, we can leave out the denominator. Also, by z-angles we know that $\alpha + \beta = \theta_k$, so $\beta = \theta_k - \alpha$. This leaves us with the following function that we want to maximize:

$$f(\alpha) = \sin(\theta_k - \alpha) + \sin(\alpha)$$

We want to know for what value of α this function has a maximum value. Note that $0 \leq \alpha \leq \theta_k$, since otherwise the path is not i-monotone anymore. We find the first and second order derivatives:

$$f'(\alpha) = \cos(\alpha) - \cos(\theta_k - \alpha)$$

$$f''(\alpha) = -\sin(\alpha) - \sin(\theta_k - \alpha)$$

$f(\alpha)$ has a maximum whenever $f'(\alpha_{max}) = 0$ and $f''(\alpha_{max}) < 0$. We can work this out as follows:

$$f'(\alpha_{max}) = \cos(\alpha_{max}) - \cos(\theta_k - \alpha_{max}) = 0$$

$$\cos(\theta_k - \alpha_{max}) = \cos(\alpha_{max})$$

$$\theta_k - \alpha_{max} = \alpha_{max}$$

$$\alpha_{max} = \frac{\theta_k}{2}$$

Filling this in in the second derivative gives $f''(\frac{\theta_k}{2}) = -\sin(\frac{\theta_k}{2}) - \sin(\frac{\theta_k}{2})$, which is indeed negative for $k \geq 4$.

All in all, we have shown that $f(\alpha)$ reaches a maximum value for $\alpha = \frac{\theta_k}{2}$, and therefor b should be exactly in the middle of cone i from a to achieve the worst possible approximation factor.

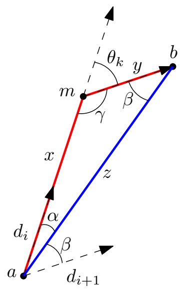


Figure 10: Angle $\alpha = \frac{\theta_k}{2}$ leads to the worst case for an i-monotone path

So, in the worst case we have an isosceles triangle (a triangle with two sides of equal length), shown on the left of Fig. 11. As mentioned before, $\gamma = 180° - \theta_k$. W.l.o.g. we assume $\overline{ab} = 1$. We will only look at the left half adm of the triangle as shown on the right of Fig. 11, since the right half dbm is symmetrical.

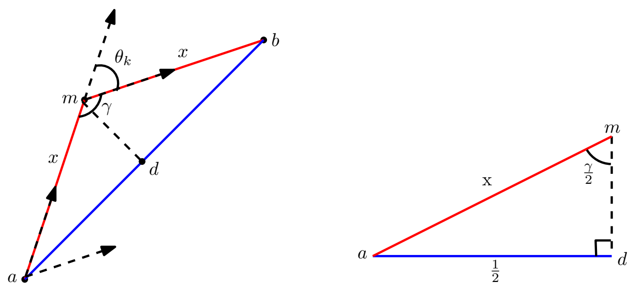


Figure 11: Left: worst case isosceles triangle. Right: zoom-in on triangle amb

Using the the law of sines again, we can say for the triangle in Fig. 11:

$$\frac{x}{\sin(90°)} = \frac{\frac{1}{2}}{\sin\frac{\alpha}{2}}$$

$$\frac{x}{1} = \frac{1}{2\sin\frac{\alpha}{2}}$$

$$x = \frac{1}{2\sin(90° - \frac{\theta_k}{2})} = \frac{1}{2\cos(\frac{\theta_k}{2})}$$

Since x is only half of the red path in Fig. 11, the total length of the approximated path is $2x = \frac{1}{\cos(\frac{\theta_k}{2})}$. This is is also directly the approximation factor since we chose the optimal path to have length 1, so we have proven that $f(k) = \frac{1}{\cos(\frac{\theta_k}{2})}$ $\square$

So, when we use k cones we will get a $\frac{1}{\cos(\frac{\theta_k}{2})}$-approximation of the optimal path. By rewriting this we find that if we want to have a certain ϵ as approximation factor, we have to choose $k \geq \frac{180°}{\cos^{-1}(\frac{1}{\epsilon})}$. For example choosing $k = 4$ leads to $\epsilon = \sqrt{2}$, and to get a maximum error of 1%, so $\epsilon \leq 1.01$, we need to choose $k \geq 22$.

5.4 Sawtooth structure

A point p has a *sawtooth* structure in every cone, similar to the staircase structure. The old staircase and new sawtooth are put side by side in Fig. 12. A vertex s is a sawtooth vertex of p if there is no vertex m such that p dominates m and m dominates s. This is the case when there are no other points in the *parallelogram* between p and s. This parallelogram is formed by the directions d_i and d_{i+1}, and is shown in Fig. 12 for vertex r using the red dashed lines. All the black vertices are sawtooth vertices, but r is not since it is dominated by s_1.

In between the sawtooth vertices are the *teeth* of the sawtooth structure, shown as a black dashed line in Fig. 12. A tooth connects two adjacent sawtooth vertices and consists of up to three segments: a segment in direction d_{i+1}, a segment on a polygon edge, and a segment in direction d_i. Each of these three segments can be null. The region outlined by these teeth is the *sawtooth region*. There can not be any vertices inside the sawtooth region, since they would have been sawtooth vertices too.

Now that sawtooth vertices have been defined, we can prove the following lemma:

Lemma 2. *For a point p and any vertex q, there exists a path from p to q which is an ϵ-approximation of the shortest path, where the first vertex on the path is a sawtooth vertex of p.*

Proof. Take some optimal path. Let the first vertex on this path be r, as in Fig. 12 where the optimal path is in blue. If r is a sawtooth vertex of p we are done, otherwise we find the segment of the sawtooth structure that the path intersects. Let this intersection point be j. This can not be a polygon edge segment since we can not intersect the polygon, so it has to be a segment in direction d_i or d_{i+1}. We adjust the path to first go through the sawtooth vertex incident to that segment, then to the intersection point j, then to r. By Lemma 1, $\langle p, s_1, j, r\rangle$ is an ϵ-approximation of $\langle p, r\rangle$, and thus the whole altered path is an ϵ-approximation of the shortest path from p to q. □

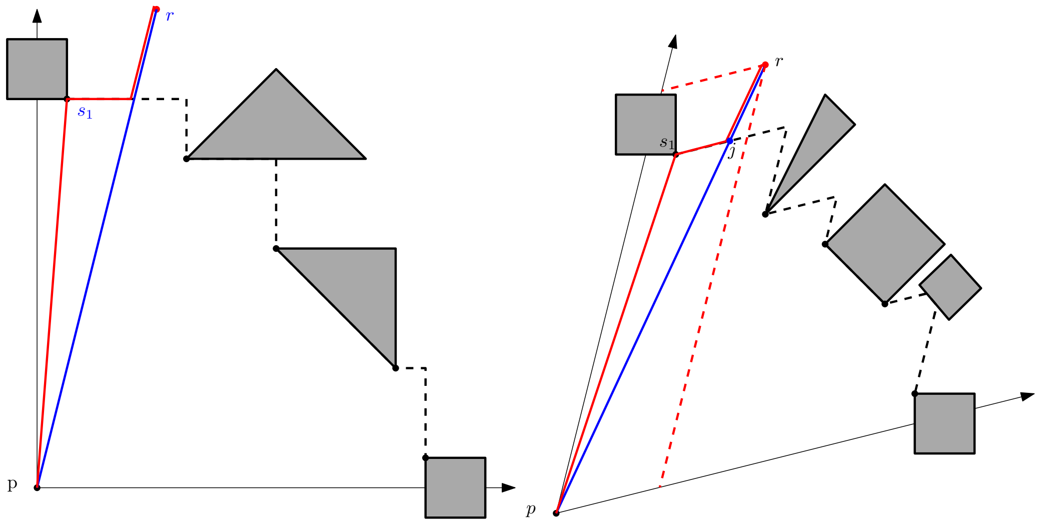


Figure 12: Left: the first quadrant of the old structure with its staircase. Right: a cone in the new structure with its sawtooth

5.5 Cutline trees

We will now introduce our *cutline trees*, as an adaptation to the ones used in [2, 12, 11]. A cutline tree is built for a specific direction d_i (and simultaneously for the direction opposite to it $d_{i+k/2}$). Each node u in the tree has a vertex set $S(u)$ and a cutline $l(u)$ associated with it. A polygon with its cutline tree for d_0 is drawn in Fig. 13.

As a base case, $S(root)$ is the set of all vertices of the polygon.

For a node u line $l(u)$ is a line in direction d_i. If we rotate the polygon such that d_i is parallel to the y-axis, we can find the vertex v_m with the median x coordinate. $l(u)$ is shot through v_m. The vertex set associated with the left child $S(left)$ is the set of all vertices on the left of v_m plus the median vertex v_m itself, and $S(right)$ is the set of all vertices on the right ofv_m. This means that $S(u)$ is divided in two (almost) equally large halves at every node, so the tree will have height $O(\log n)$.

We have one such cutline tree for each direction, meaning we will have $k/2$ trees since two opposite directions produce one cutline tree.

We also introduce the notion of *levels* and *super levels* of the tree. The level number $level(u)$ is $\lceil \log n \rceil$ if u is the root of the tree. Then if v is the child of u, $level(v) = level(u) - 1$. Note that this is defined inversely to [12] such that a node with a higher level number is higher in the tree, if we view the root as being the top node. Then we horizontally subdivide the cutline trees into $\sqrt{\log n}$ strips called *super levels* of height $\sqrt{\log n}$ each, such that super level i contains all nodes with level numbers from $(i-1) * \sqrt{\log n}$ to $i * \sqrt{\log n}$.

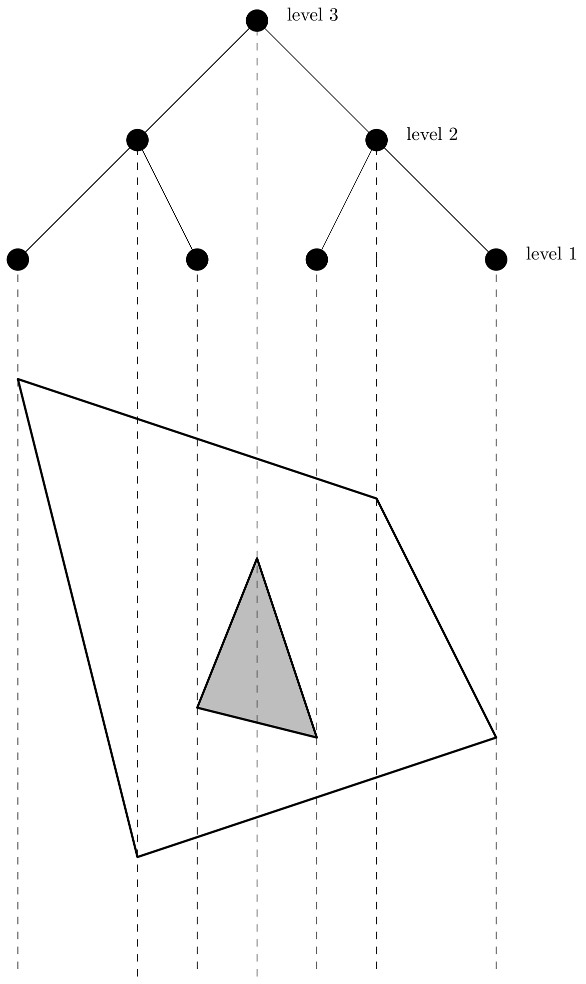


Figure 13: A cutline tree in direction d_0 for a small polygon

5.6 Steiner points

Just like in [2], a vertex can have $O(n)$ sawtooth vertices. Storing all these will take too up too much space, so again we will use *Steiner points* to connect a vertex to its sawtooth vertices indirectly. There are two types of Steiner points.

5.6.1 Type 1 Steiner points

Every polygon vertex is projected along all k directions onto the polygon edges to get up to k type-1 Steiner points, as shown in Fig. 14. Since all n vertices define up to k type-1 Steiner points, there are $O(nk)$ of them.

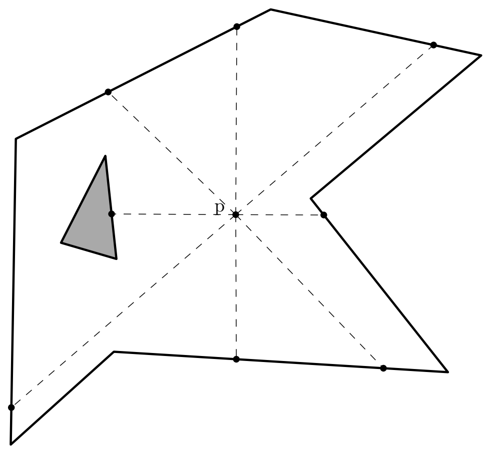


Figure 14: Type 1 Steiner points of a point p

5.6.2 Type 2 Steiner points

Type 2 Steiner points make use of the previously defined cutline trees. For some cutline tree in direction d_i, let u be a node that is at the highest level of its super level, meaning the parent of u is in a different super level or u is the root. Let $T(u)$ be the subtree of u while only including nodes at the same super level as u, as shown in Fig. 15. Then for each vertex $v \in S(u)$, for each node $c \in T(u)$, v defines a type 2 Steiner point at its projection in direction d_{i+1} on $l(c)$, if that projection point is visible to v. Fig. 16 shows some type-2 Steiner points.

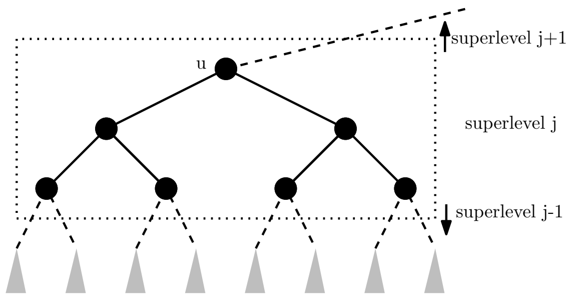


Figure 15: The subtree $T(u)$ in some cutline tree where $\sqrt{\log(n)} = 3$. All vertices in $S(u)$ are projected onto the cutlines of all black nodes to create type-2 Steiner points.

By the definition of super levels $T(u)$ has height $\sqrt{\log n}$, and therefor $2^{\sqrt{\log n}}$ nodes. This means that each vertex in $S(u)$ defines $O(2^{\sqrt{\log n}})$ type 2 Steiner points. Since there are n vertices on every level of the tree, one super level defines $O(n * 2^{\sqrt{\log n}})$ type 2 Steiner points. There are $O(\sqrt{\log n})$ super levels, so one cutline tree defines $O(\sqrt{\log n} * n * 2^{\sqrt{\log n}})$ type 2 Steiner points. All cutline trees together then define $O(k * \sqrt{\log n} * n * 2^{\sqrt{\log n}})$ points.

5.7 Preprocessing

Now that all Steiner points are defined, we can calculate them all in preprocessing. We will store all the cutline trees, and for all cutlines we store all the type 2 Steiner points on them in sorted order such that we can quickly binary search in them. We will use a fractional cascading structure between the nodes of the tree to speed up search. We also store all type 1 Steiner points in sorted order for each polygon edge.

We will also need to store shortest paths between all pairs of Steiner points. Note that this table does not need to store exact shortest paths, it is sufficient to store ϵ-approximate paths. Since we have $O(k * \sqrt{\log n} * n * 2^{\sqrt{\log n}})$ total Steiner points, we can store all pair shortest paths

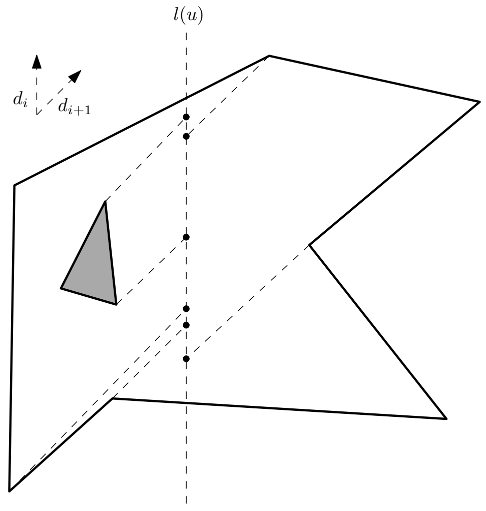


Figure 16: Type 2 Steiner points for direction i

between them using $O(k^2 * \log n * n^2 * 4^{\sqrt{\log n}})$ space. Note that $O(4^{\sqrt{\log n}}) = O(n^\delta)$ for any $\delta > 0$.

5.8 Gateways

We will now define a set of gateway points $G(p)$ for p, consisting of two parts $G_1(p)$ and $G_2(p)$. The idea is that there is always an ϵ-approximate path from p to any polygon vertex that goes through a gateway point.

For $G_1(p)$ we project p in all k directions onto the polygon edges. The type 1 Steiner points adjacent to these projections are all in $G_1(p)$, so $|G_1(p)| = O(k)$. Since a single ray can be shot in $O(\log n)$ time, it takes $O(k \log n)$ time to calculate $G_1(p)$.

To calculate $G_2(p)$ we use the cutline trees that we stored. For a tree in direction d_i we first find the path in the tree to point p. That is, we start at the root and go to the left (respectively right) child if p is on the left (respectively right) of the current cutline, and repeat this until we are in a leaf. For all nodes u on the path, if the projection from p in direction d_{i+1} onto $l(u)$ is visible to p it is a *projection cutline* of p. There are $O(\log n)$ projection cutlines. In Fig. 17 all the black nodes are potential projection cutlines of p, if p's projection onto their cutline is visible to p.

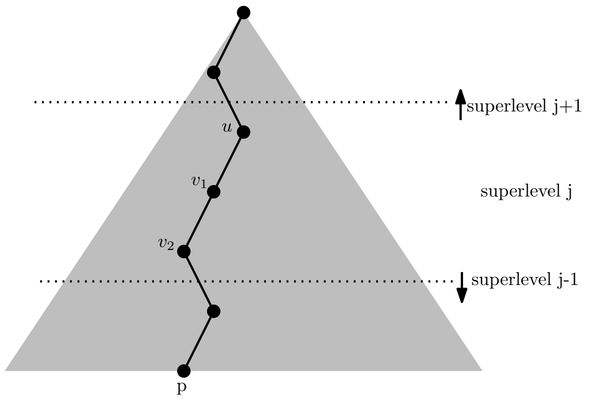


Figure 17: Possible projection cutlines and relevant projection cutlines of p

A projection cutline $l(u)$ of p can also be a *relevant projection cutline.* For this we look at the super level of u, and all other projection cutlines that are on the same side of the line in direction d_i through p. If $l(u)$ is the projection cutline with the lowest level number (the deepest in the tree) among all projection cutlines in the same super level, then $l(u)$ is a relevant projection cutline.

This means there are at most 2 relevant projection cutlines per superlevel, one on each side, so $O(\sqrt{\log n})$ per cutline tree.

In Fig. 17 superlevel j is shown. Here v_2 is the relevant cutline for superlevel j on the left of p, if it is visible to p. v_1 is the relevant cutline for superlevel j on the right of p if it is visible to p, otherwise u is (once again, only if it is visible to p).

$G_2(p)$ then consists of the two type 2 Steiner points adjacent to the projection point of p on its relevant projection cutlines. By this definition, $|G_2(p)| = O(\sqrt{\log n})$ per tree and therefor $G(p) = O(k\sqrt{\log n})$. It takes $O(k \log n)$ time to calculate $G_2(p)$.

With the gateways defined we can prove the following lemma:

Lemma 3. *For a point p with gateway points $G(p)$, for any of its sawtooth vertices s, there exists an approximate shortest path from p to s which goes through a gateway point $g \in G(p)$.*

Proof. Vertex s is in some cone i of p. Consider the parallelogram between p and s with directions d_i and d_{i+1}, shown with a red dashed outline in Figs. 18 and 19. There can not be any vertices in this parallelogram since s is a staircase vertex of p. However, there can be a part of a polygon edge inside the parallelogram. This gives us two cases, one with a part of an edge inside the parallelogram and one without.

Case 1: there is a polygon edge inside the parallelogram between p and s, intersecting the ray from p in direction d_{i+1} (or symmetrically d_i) as in Fig. 18. This edge must also intersect the ray from s in direction $d_{i+k/2}$, since otherwise there would be a vertex inside the paralellogram. This means that s has created a type 1 Steiner point on this edge, shown as s_1 in the figure. While building $G_1(p)$, we projected p in direction d_{i+1} to get point p_1. We then added the two Steiner points adjacent to p_1, g_1 and g_2 in Fig. 18, to $G_1(p)$. One of these two gateways, g_1 in the figure, lies between p_1 and s_1. Note that g_1 can be the same point as s_1. Now the path $\langle p, p_1, g_1, s_1, s\rangle$ is an ϵ-approximation of the shortest path from p to s by Lemma 1.

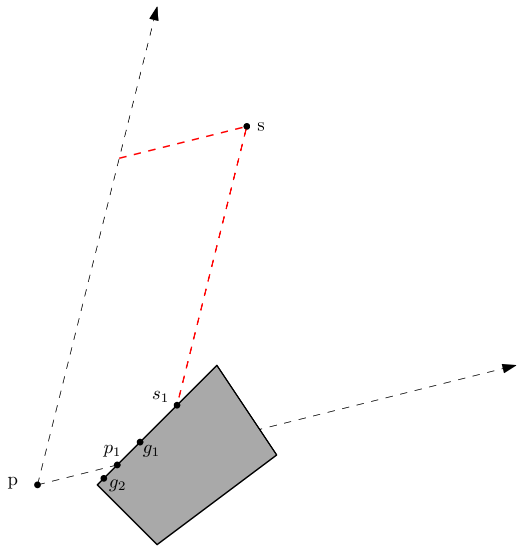


Figure 18: Case 1 where an edge intersects the parallelogram

Case 2: there are no polygon edges inside the parallelogram between p and s as in Fig. 19. Consider the cutline tree with cutlines in direction d_i, and consider the node u with the highest level number such that p and s are on other sides of $l(u)$, as illustrated in Fig. 20. We assume p is on the left of s with respect to direction d_i, the case where p is on the right of s is symmetric. $l(u)$ is a projection cutline of p since it is on the path in the tree to p, and its projection point on $l(u)$ is visible since there are no other edges in the parallelogram. Let a be the highest ancestor of u in the same super level, in Fig. 20 this is the same node as u. s will have projected type 2 Steiner points on the cutlines of all nodes in $T(a)$ (the subtree of a in the same superlevel). Exactly one of these cutlines is also a relevant cutline on the right of p. We call this relevant cutline $l(c)$, and the type 2 Steiner point s created on it s_2.

It could be that $l(c) = l(u)$. Otherwise, c must be somewhere in $T(u)$. In particular, it must be in the left subtree of $T(u)$ since p is on the left of $l(u)$. In either case $l(c)$ is between p and s. This means that while building $G_2(p)$, we projected p onto its relevant cutline $l(c)$ as point p_2. We then added the two adjacent Steiner points to $G_2(p)$. This means s_2, or another Steiner point on $l(c)$ between p_2 and p_2, is in $G_2(p)$. The path from p via this gateway point via s_2 to s is an ϵ-approximation of the shortest path from p to s by Lemma 1. □

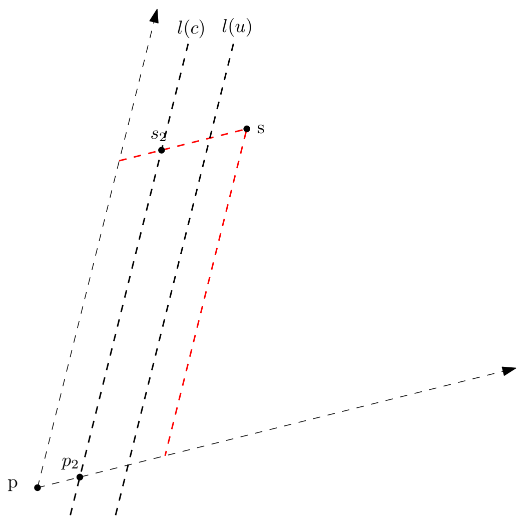


Figure 19: Case 2 where no edges intersect the parallelogram

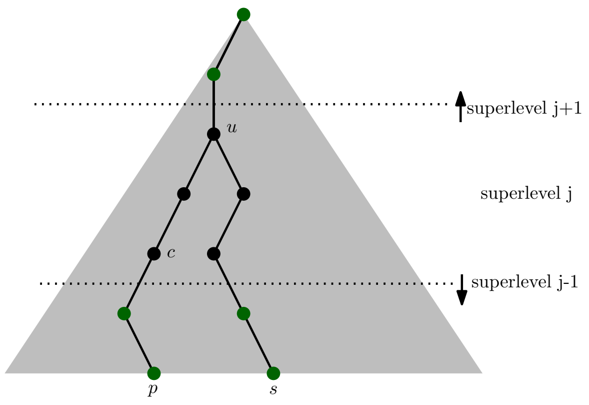


Figure 20: The cutline tree for case 2 where no edges intersect the parallelogram

5.9 Querying

Lemmas 2 and 3 together show that from a point p we can get to any vertex in the polygon with an ϵ-approximate shortest path through a gateway point in $G(p)$. Now we can prove our final lemma:

Lemma 4. *Let p and q be two query points. There always exists an ϵ-approximate path between them which goes through a gateway $g_p \in G(p)$ and a gateway $g_q \in G(q)$.*

Proof. As mentioned before we assume there is at least one polygon vertex in the shortest path between them, we call it v. We can get to v with an ϵ-approximate shortest path from both p and q that goes through a gateway in $G(p)$ and $G(q)$ respectively by Lemmas 2 and 3, and putting these paths together will result in a full ϵ-approximate path from p to q. □

This directly leads to the algorithm to find this path: simply go through all pairs (g_p, g_q) with $g_p \in G(p), g_q \in G(q)$. Since $|G(p)| = O(k\sqrt{\log n})$ and $|G(pq)| = O(k\sqrt{\log n})$ there are $O(k^2 \log n)$ such pairs. We can retrieve the length l_{g_p,g_q} of the shortest path between g_p and g_q from our preprocessed all-pair shortest path datastructure in $O(1)$ time. The total length of the path through this pair of points is then the distance from p to g_p plus l_{g_p,g_q} plus the distance from g_q to q. We choose the pair with the lowest total distance, which by Lemma 4 is then guaranteed to be an ϵ-approximation of the shortest path between p and q. The actual path can be retrieved in $O(z)$ time with z being the number of turns in the path between g_p and g_q.

5.10 Final result

All in all this leads to the following theorem:

Theorem 1. *Given a polygon P we can create a datastructure of size $O(n^2k^2 4^{\sqrt{\log n}} \log n)$, which can find $f(k)$-approximate shortest paths between any two points in P in $O(k^2 \log n)$ time.*

6 Implementation

In this section we will elaborate on how the datastructure described in Section 5 was implemented. Everything was implemented in C++using the Computational Geometry Algorithms Library (CGAL) [13], with the latest stable version 5.4. This library is widely used in the field of computational geometry, and gives a lot of freedom in how we use the algorithms and datastructures it provides.

6.1 Exact numbers

There are two types of number representations that we had to choose between when implementing this datastructure: rational numbers (exact) and floating point numbers (inexact). Rational numbers are stored as two integers of unbound size, the numerator and the denominator, resulting in unlimited precision. This has the advantage that it does not suffer from rounding errors, but it can make calculations slower and less space efficient. Floating point numbers have limited precision and are rounded off after every operation, but it is faster to do calculations with them.

Algorithms are often theoretically designed with the assumption that all used numbers are exact, so a loss in precision could lead to things not working as intended. For example if we shoot a ray from a polygon vertex, we might hit one of the edges adjacent to that vertex due to rounding errors, which is not what we want. This specific problem could be fixed with some extra checks in the code, but such checks would be required in multiple edge cases throughout the program. For this reason we decided to use an exact number representation, despite the fact that this slows down computations.

6.1.1 Non-rational values

There are two cases where we have to use non-rational values: when we want to do a rotation or shoot a ray at a certain angle we need to use the *sin* and *cos* functions, and when we want to calculate the distance between two points we need to calculate a square root. All three of these functions can give non-rational outputs. We have to decide on a way to deal with this.

For the *sin* and *cos* functions, we will use a rational approximation instead of the exact value. If we want to shoot a ray at some angle δ we first calculate $sin(\delta)$ and $cos(\delta)$ as floating point values, and then cast them to rational numbers. Because of the limited precision this means the rays will not be shot at exactly δ degrees, but slightly off. In our case this is not a problem however, as long as any calculations done with this ray are exact and the returned intersection point lies precisely on the boundary of the polygon.

When we need to calculate the distance between two points we can also just use an approximation. We can calculate the exact squared distance between them, and then approximate the square root of that as a floating point number. This does not cause any issues because if two paths are so similar in length that rounding errors can cause one or the other to be considered shorter, we do not care which of the two paths is returned.

6.2 Ray shooting

Being able to shoot rays is very important for building and using our datastructure: we shoot rays to create Steiner points, to check if points on a cutline are adjacent, to find gateways of a query point and to check if two query points are mutually visible.

Formally, given a point p and a direction d we want to know what is the first edge we hit if we travel from p in direction d. In our case d will almost always be one of the k directions defined earlier, except when checking if the two query points are mutually visible. In the following sections we will explore two approaches to solve this problem: the first can only answer ray-queries in the k predefined directions, the second in any direction. We ended up using the second approach.

6.3 k-directional rayshooting

In this section we will create a simple datastructure of size $O(k * n)$, that can answer rayshooting queries in k predefined directions in $O(\log n)$ time.

CGAL provides a datastructure to allow for $O(\log n)$ time vertical rayshooting queries [14], so shooting a ray straight up or down from a given point. We can use this to make our own k-directional rayshooting structure. In preprocessing we would rotate the original polygon by increments of θ_k, and build and the vertical rayshooting datastructure for all k of these rotated polygons. We store these datastructures in a simple list of length k.

A query consists of a point p and an integer dir, which means we want to shoot a ray from point p in direction d_{dir}. Direction d_{dir} corresponds to a vertical ray in the dir'th datastructure we stored, which we can access in $O(1)$ time. We will rotate p by $dir * \theta_k$ degrees to get p^*, as shown in Fig. 21. We shoot a vertical ray from p^* in $O(\log n)$ time to get intersection point i^*, and then rotate i^* back to find the original intersection point i.

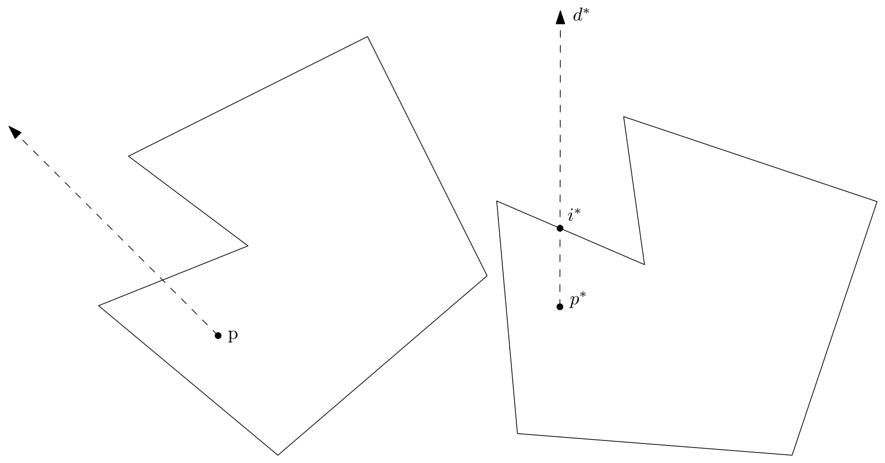


Figure 21: Left: the original polygon. Right: the preprocessed rotated polygon

We used this datastructure at the start of the implementation process, because it was simple to implement. It can not be used to find if two arbitrary query points are mutually visible however, which is a required operation. For that reason we implemented the datastructure from the next section, and ended up not using this datastructure altogether.

6.4 Arbitrary direction rayshooting

For this datastructure we followed a paper by Agarwal [15]. We will build a datastructure of size $O(n^2 \log n)$ that can answer rayshooting queries from any point in any direction in $O(\log n)$ time. We will use point-line duality to create a dual arrangement of our polygon, which has some nice properties to help us find what edges are intersected by a line. Using a persistent binary search tree we will store this information in a space efficient manner.

6.4.1 Point line duality

With point-line duality we can create a *dual space* from some *primal space*. If in our primal space we have a point $p = (p_x, p_y)$, then its dual in the dual space will be a line $p^* : y = p_x * x - p_y$. The inverse holds for lines: a primal line $l : y = ax + b$ is dualized into a dual point $l^* = (a, -b)$.

Since a primal line segment or edge e can be seen as infinitely many primal points next to each other, its dual e^* can be seen as infinitely many dual lines that are slightly rotated, also called a dual *wedge*, as shown in Fig. 22 for two edges. Dual wedge e^* consists of two dual lines e_1^* and e_2^*, which are the duals of the endpoints of the primal edge e_1 and e_2. All dual points between these dual lines are inside the wedge, and these regions are shaded blue and red for wedges e^* and d^* in the figure. A dual point q^* inside e^* corresponds to a primal line q that intersects e, and any dual point q^* outside e^* corresponds to a primal line q that does not intersect e. In Fig. 22 three examples are shown: q_1 intersects neither e or d and its dual point q_1^* is therefore outside both wedges, q_2 intersects only e so q_2^* is inside e^* but outside d^*, and q_3 intersects both e and d so q_3^* is inside both dual wedges.

If we take dual point q_3^* and start moving it towards q_2^*, primal line q_3 will start rotating to the right. At some point q_3^* will hit e_2^*, the outer boundary of dual wedge e^*. This corresponds to q_3 intersecting e_2 in the primal plane. If we move q_3^* slightly further we will go into the next dual face, and q_3 will stop intersecting e.

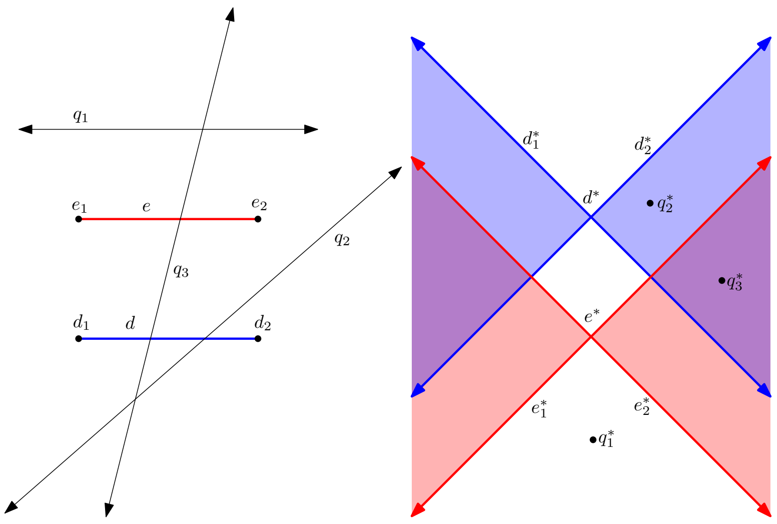


Figure 22: Left: a primal arrangement with line segments and lines. Right: the dual arrangement

6.4.2 Dual arrangement of polygon

We will dualize our polygon, which will result in a dual arrangement with n dual wedges, one for each primal edge of the polygon. Some dual face f^* lies inside some set of dual wedges $W_{inside}(f^*)$ and outside all others. This means that any dual point l^* inside f^* also lies inside all wedges in $W_{inside}(f^*)$. l^* corresponds to a primal line l that intersects some set of primal edges $E_{intersect}(f^*)$. For example for face $f_{q_3^*}^*$, the dual face containing q_3^*, $E_{intersect}(f_{q_3^*}^*) = \{d, e\}$. Now clearly $|W_{inside}(f^*)| = |E_{intersect}(f^*)|$, and each dual wedge $w \in W_{inside}(f^*)$ corresponds to a primal edge $e \in E_{intersect}(f^*)$. W_{inside} and $E_{intersect}(f^*)$ will be the same for any dual point l^* inside f^*, and the order that the edges in $E_{intersect}(f^*)$ are intersected will also be the same, because our primal edges are non-intersecting.

Knowing this, for each dual face f^* we would like to store the set of intersected primal edges $E_{intersect}(f^*)$, sorted by order of intersection. We also build a trapezoidal point-location datastructure [14] on the dual arrangement of size $O(n^2)$, such that we can find in what dual face a dual point lies in $O(\log n)$ time. We can then answer arbitrary direction rayshooting queries in $O(\log n)$ time as such: for a query from point p in direction d, first find the line l that d is on. We find its dual point l^*, and find what dual face f^* contains this dual point using our trapezoidal decomposition. In the sorted set $E_{intersect}(f^*)$ that is stored with f^* we can binary search for our query point in $O(\log n)$ time, and find its successor- or predecessor-edge depending on the direction of the query ray. This edge is the first edge hit by the ray, so we can return its intersection with l.

6.4.3 Storing $E_{intersect}$

We still need to figure out how exactly we store $E_{intersect}(f^*)$. Simply storing the full sorted set for every face will use too much space: since our polygon has n vertices the dual arrangement will have $O(n^2)$ faces, and since each line can intersect $O(n)$ edges this leads to a total space usage of $O(n^3)$. In this section we will bring this space usage down to $O(n^2 \log n)$, while keeping the $O(\log n)$ query time.

We first note that for two adjacent faces f_1^* and f_2^*, $W_{inside}(f_1^*)$ and $W_{inside}(f_2^*)$ are very similar. f_1^* and f_2^* are separated by a dual edge, which lies on a dual line v^* corresponding to some primal vertex v. The only difference between $W_{inside}(f_1^*)$ and $W_{inside}(f_2^*)$ are all dual wedges that use this dual line v^*: if f_1^* is inside such a wedge then f_2^* is outside, and the other way around. Recall the earlier example of moving dual point q_3^* around in Fig. 22, which was the same principle. Similarly, the only difference between $E_{intersect}(f_1^*)$ and $E_{intersect}(f_2^*)$ are all primal edges that are incident to v. Since we are working with polygons we know that every vertex has exactly two incident edges, so there are two edges that are only present in one of $E_{intersect}(f_1^*)$ and $E_{intersect}(f_2^*)$, and the rest of the sets are the same. We can store for each dual edge which two primal edges are associated with it.

Now we will step from dual face to dual face through the dual arrangement, while maintaining $E_{intersect}$ as a *fully persistent binary search tree.* This behaves just like a normal binary search tree, meaning you can insert, delete and query items in logarithmic time, but it keeps track of its history such that you can perform these operations on any previous version of the tree as well. In our case each insertion or deletion will use $O(\log n)$ more space. We have to start our path in a dual face f_{start} of which know $E_{intersect}(f_{start})$. We can easily find a dual face where $E_{intersect}(f_{start})$ is empty by taking the dual point l^* of a primal line l that lies fully below the whole polygon, and finding the face that contains l^*. From this starting face we will step through the dual arrangement in a depth-first-search fashion. At every step we move from the current dual face to a neighbouring dual face through some dual edge. With this dual edge we have stored the two primal edges associated with it, so we can delete those from the current binary search tree if they are already present, or add them if they are not. Finally, we have to store a pointer in the current face to the current version of the tree which we can use while querying. Since there are $O(n^2)$ faces we will perform $O(n^2)$ updates to the tree, resulting in a space usage of $O(n^2 \log n)$.

6.4.4 Custom fully persistent binary search tree

Our fully persistent binary search tree is based on [16] by Sarnak et al. and uses $O(\log n)$ extra space per update performed on the tree, while maintaining $O(\log n)$ time for all operations on all versions of the tree. Note that this space usage is not optimal, in [16] a solution is shown that uses only $O(1)$ space per update is shown. Due to time constraints we opted not to use this improved version, because the version that uses $O(\log n)$ space per update was easier to implement.

Our implementation builds on code from [17]. We implemented a standard red-black tree, which balances itself after every update (insertion or deletion). We made it persistent by modifying the updates such that they never actually change the values or structure of the current version of the tree, but instead create new nodes with pointers to older parts. This way a new root node containing the correct tree is created every update, but each version only takes $O(\log n)$ space since that is how many nodes are changed (or in the persistent case, created) during the operation. The time required for any operation is not changed by making the tree persistent.

As a second modification, our tree does not just store numbers with a predefined order. We need to sort edges in order of their intersection with some line. We can use any primal line l associated with a dual point l^* inside the dual face we are currently processing, since all these lines intersect the same edges in the same order. We call l the *reference line.* Every time we step to a new dual face f_{new}^* we need to update this reference line. For this we have to find a point inside f_{new}^*. This is easy if f_{new}^* is finite, we can just take three arbitrary dual vertices on its boundary and their midpoint will be inside f_{new}^* because all our dual faces are convex. When f_{new}^* is not finite we find the two directions d_1 and d_2 in which it stretches to infinity, take their average d_{avg} and go some distance in d_{avg} from any vertex on the boundary of f^*_{new}. This point will be inside f^*_{new} because it stretches infinitely in any direction between d_1 and d_2, including d_{avg}. Both these cases are illustrated in Fig. 23, which uses the same dual space as Fig. 22. In the figure m_1 is a point inside a finite face and m_2 in a non-finite face.

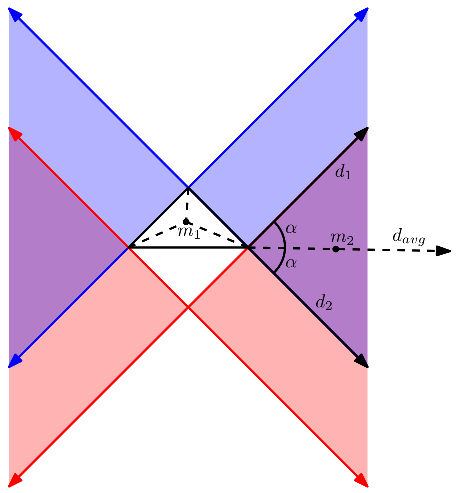


Figure 23: Two examples of finding a point inside a dual face

6.5 Suboptimal query implementation

As proven in Section 5 the optimal query time for the sawtooth datastructure is $O(k^2 \log n)$. Indeed the whole reason we introduced relevant cutlines was to reduce the number of gateways from $O(\log n)$ to $O(\sqrt{\log n})$, which reduces query time from $O(k^2 \log^2 n)$ to $O(k^2 \log n)$.

However our implementation is sadly not optimal, and has $O(k^2 \log n + k \log^2 n)$ query time. The $k^2 \log n$ part is still caused by going through all pairs of gateway points. The $k \log^2 n$ part is from finding these gateway points, because this was not implemented optimally. Firstly the fractional cascading datastructure was not implemented resulting in slower binary searches in the Steiner points. Secondly for some direction d_i we shoot a new ray at every level of the cutline tree, even though one ray in direction d_{i+1} and one ray in direction $d_{i+k/2+1}$ in total would suffice. Both these issues should be relatively easy to fix, but due to time constraints this was not done.

6.6 All pair shortest paths

During querying we need an all pair shortest path datastructure that can find the path between any two vertices or Steiner points in $O(1)$ time. This path does not have to be the exact shortest path, it can be an ϵ-approximate shortest path. For each pair p, q we store the distance between them, and the successor of p in this path. As mentioned before we have $O(k * \sqrt{\log n} * n * 2^{\sqrt{\log n}})$ total Steiner points, so we can store all pair shortest paths between them in a simple table using $O(k^2 * \log n * n^2 * 4^{\sqrt{\log n}})$ space.

While creating the Steiner points we simultaneously build a graph. This graph has as nodes all polygon vertices and Steiner points. A polygon vertex is connected to all the Steiner points it created, and to the closest type-1 Steiner points on its incident edges. Also, all adjacent Steiner points on the same edge or cutline are connected pairwise. In this graph all vertices are connected to all their sawtooth vertices with ϵ-approximate paths by Lemma 4, which means it contains ϵ-approximate shortest paths between all pairs of nodes that are not mutually visible.

After building the graph we simply run Dijkstra's algorithm [18], which calculates the shortest path from a single source node to all other nodes, on every node of the graph to create the all pair approximate-shortest path table we need.

6.7 Optimization

In Section 5 we have proven several worst-case space and time complexities for our datastructure. We can make small optimizations to make it work faster and use less space in practice.

The purpose of Steiner points is to connect vertices with their sawtooth vertices with ϵ-approximate paths. So the idea is that if a Steiner point can never be used for this, we do not need to store it. Fig. 24 illustrates this. In the figure $k = 8$ is used, and two type-1 and one type-2 Steiner points of p are shown.

For type-1 Steiner points (s_1 and s_2 in the figure), we know they can only be used to connect sawtooth vertices as in Fig. 18 when the angle α between the projection ray and the edge is small, specifically if $\alpha \leq \frac{360°}{k}$, or in the case of Fig. 24 with $k = 4$ if $\alpha \leq 45°$. Angle $\alpha_1 = 90° > 45°$ so s_1 does not need to be stored, but since $\alpha_2 = 45° \leq 45°$ Steiner point s_2 does need to be stored.

For type-2 Steiner points we only project in direction d_i from direction d_{i+1} or $d_{i+1+k/2}$, so the angle will always be small enough. This does not mean however that all type-2 Steiner points need to be stored. For example in the figure point s_3 can never be used to connect p to one of its sawtooth vertices: this would require a Steiner point on $l(u)$ below s_3 coming from the left of $l(u)$, as illustrated by the red arrow. Since such a Steiner point is not present, s_3 will never be used so we don't need to store it.

Both of these optimizations can also be used when finding the gateways for a query point.

The type-1 optimization has been implemented, while the type-2 optimization has not.

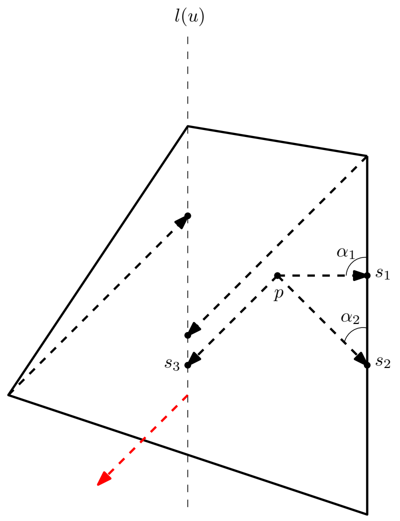


Figure 24: Examples of Steiner points that need not be stored

6.8 Naive datastructure

We also implemented a naive shortest path datastructure that finds the exact shortest path. This datastructure uses $O(n^2 \log n)$ space, and a query takes $O(n^2)$ time.

In preprocessing, we build the arbitrary direction rayshooting datastructure from Section 6.4 which uses $O(n^2 \log n)$ space. We use this to create a simple visibility graph, where the nodes are polygon vertices and two nodes are connected by an edge if they are mutually visible. The length of such an edge is the distance between the two nodes it connects. Again by running Dijkstra's single source shortest paths algorithm from every vertex we can build an all pair shortest path table, which can give us the shortest path between any two polygon vertices in $O(1)$.

Now for a query p, q the process is very simple. First check if p and q are mutually visible by shooting a ray between them, if so return the trivial path. Otherwise we know that there is at least one vertex on the shortest path. Let $Visible(p)$ be the set of vertices that p can see, which we can find by going through all vertices and shooting a ray towards them from p in $O(n \log n)$ time. The set $Visible(q)$ is defined similarly. Then for all pairs $v_p \in Visible(p)$ and $v_q \in Visible(q)$ we calculate the length of the shortest path from p to v_p to v_q to q. We can look up the shortest path between v_p and v_q in our all pair shortest path table, and simply add the length of the segments $\langle p, v_p >\rangle$ and $\langle v_q, q\rangle$. We find the v_p, v_q pair that gives the shortest total distance, giving us the exact shortest path.

7 Methods

With the datastructure designed and implemented we can test it in practice. This section will describe what data we used, and what experiments we ran exactly.

7.1 Dataset

An instance of our problem consists of two parts: a polygon, and a set of queries. We mostly used polygons from the Salzburg database [19], which contains both polygons with and without holes generated by different algorithms. Most of these polygons were far too large for our algorithms to handle, so we used only the ones with 500 or fewer vertices. There were 873 such polygons. We will give a short description of the different types of polygons below.

7.1.1 Fast Polygon Generator (FPG)

This method uses triangulation pertubation. We start of with a convex n-gon, and iteratively move its vertices in random directions while maintaining simplicity. This same technique works for polygons with holes as well. An example is shown on the left of Fig. 25.

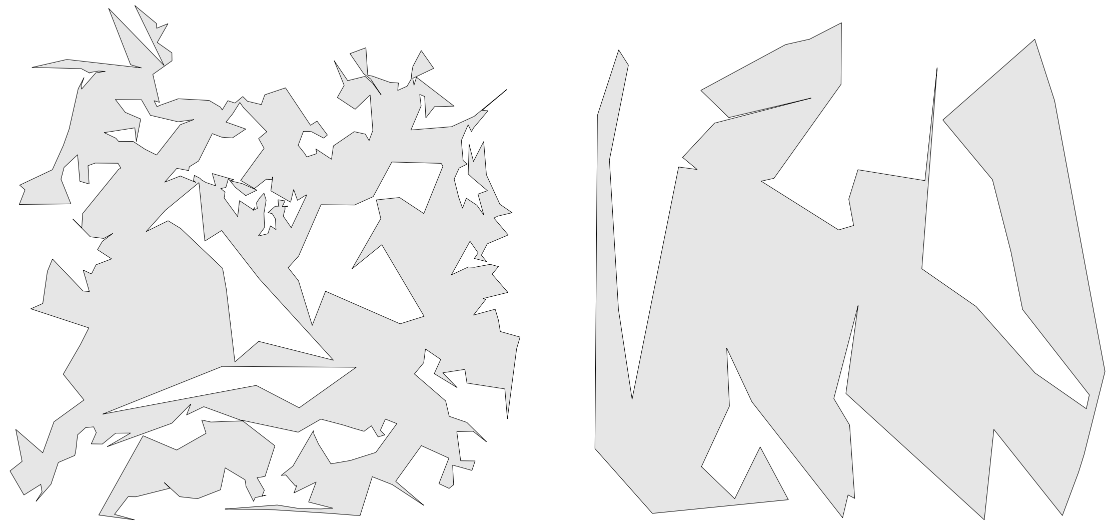


Figure 25: Left: an FPG polygon. Right: an SPG polygon

7.1.2 Simple Polygon Generator (SPG)

SPG takes a set of points as input, and produces a simple polygon whose vertices are the input points. In our case the points are created uniformly at random in a bounding box. Initially all points are put in a random order, which likely results in a polygon with self intersections. Using a line sweep we go through the polygon, detect these intersections, and fix them locally, repeating or reversing the line sweep if necessary.

SPG does not produce polygons with holes.

An example is shown on the right in Fig. 25.

7.1.3 Super Random Polygon Generator (SRPG)

Lastly we have SRPG, which most of our database consists of. This produces polygons using a regular grid of square cells, and has multiple variations:

- SRPG_iso: orthogonal polygons.
- SRPG_iso_aligned: orthogonal polygons on an integer grid.
- SRPG_octa: octagonal polygons.
- SRPG_perturbed: random polygons.
- SRPG_perturbed_smo: random polygons with smoothed corners.
- SRPG_perturbed_smr: random polygons with even smoother corners.

See Fig. 26 for examples of each of these variations.

7.1.4 Pacman polygon

In addition to the above polygons, which were all taken from the Salzburg database, we also generated some of our own *pacman polygons* to artificially create a situation where any query point has a lot of visible vertices. These polygons are regular polygons except for one vertex which almost cuts the polygon in half, as shown in Fig. 27. The reason for this split is such that two random query points have about a 50% chance to not be mutually visible, meaning the resulting query is not trivial.

7.1.5 Query points

To create random queries we simply choose two points independently and uniformly at random inside the polygon. This leaves the possibility that the two query points are mutually visible, which is a trivial but still valid query. Allowing trivial queries has advantages and disadvantages. An advantage is that we do not filter out any queries and allow all possible pairs of query points, keeping the query distribution uniform. A disadvantage is that polygons with large amounts of trivial queries will have very small query times and approximation factors, skewing the results.

To test the effect of these trivial queries we also did a small experiment without trivial queries, by discarding the chosen query points and trying again if they are mutually visible. This experiment was only done on the pacman polygons.

We find a point inside the polygon by generating a point inside the bounding box of the polygon first, and then checking if it is inside the polygon. If it is we return the point, otherwise we try again.

Due to a bug in the code the random seed was the same for every single polygon. This resulted in the exact same sequence of generated points for every polygon, relative to the bounding box of the polygon. For example the first point generated would always be in the lower left corner of the bounding box, and the second would always be in the middle. However this does not mean that the same queries were done for every polygon, since when a point in the generated sequence lies outside the polygon we would move on to its successor. Because of this, and because two equivalent queries can still lead to completely different paths in different polygons, this should not have affected the results. This bug was fixed before running the experiments on the pacman polygons.

7.2 Experiments

In this section we will elaborate on what experiments we ran exactly. We will refer to our new proposed datastructure from Section 5 as the sawtooth oracle, and to the naive datastructure from Section 6.8 as the naive oracle.

7.2.1 Main experiments on Salzburg database and ARPs

The main experiments were conducted on all 873 polygons in the Salzburg database with at most 500 vertices. For every polygon we built both the sawtooth oracle and the naive oracle. We then generated 1000 pairs of query points p, q, which could be trivial queries, which we queried in both oracles to obtain our ϵ-approximate shortest path from the sawtooth oracle and an optimal shortest path from the naive oracle.

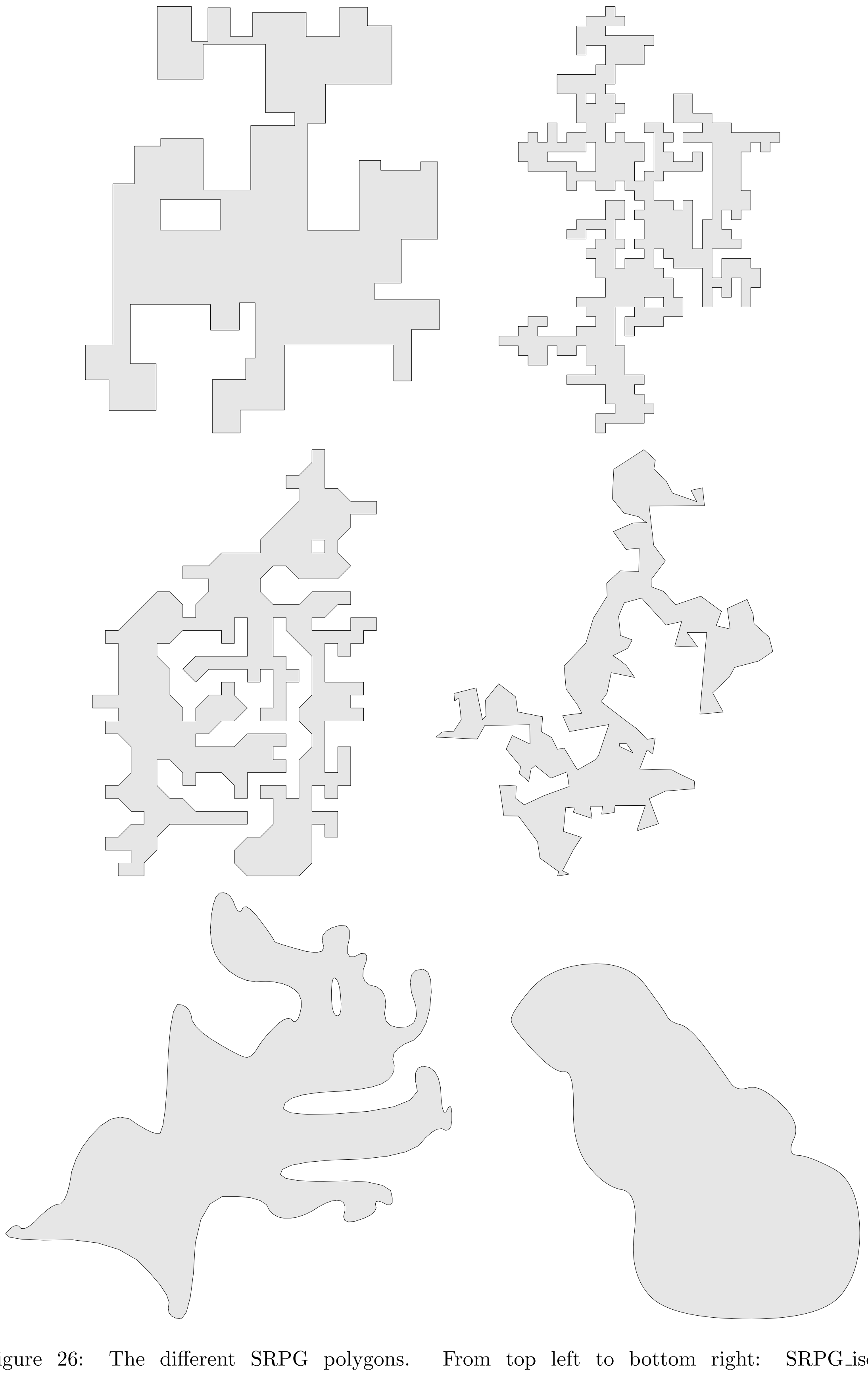


Figure 26: The different SRPG polygons. From top left to bottom right: SRPG_iso, SRPG_iso_aligned, SRPG_octa, SRPG_perturbed, SRPG_perturbed_smo, SRPG_perturbed_smr

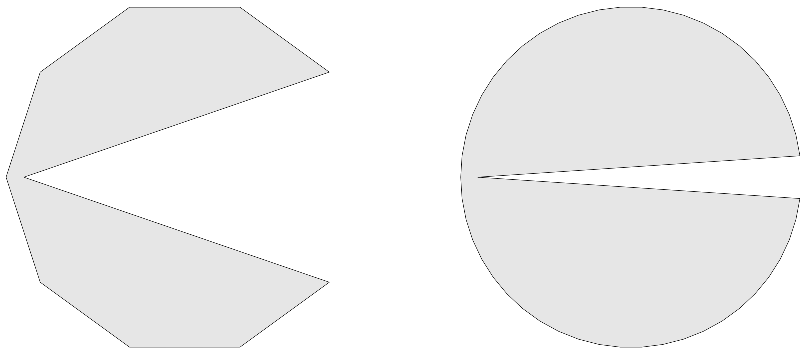


Figure 27: Two pacman polygons, with n=10 and n=50

We did the same for 50 pacman polygons with the number of vertices $n \in \{10, 20..490, 500\}$, once with queries that can be trivial and once with only nontrivial queries.

The following measurements were taken for each polygon:

- Preprocessing speed of both oracles, in microseconds.
- Preprocessing space of both oracles, in bytes.
- Average query speed of both oracles, in microseconds.
- Average approximation factor of sawtooth queries, by comparing the length of the path given by the sawtooth oracle to the length of the path given by the naive oracle.

All of this was done three times with three different values of $k \in \{4, 8, 16\}$. Lemma 1 tells us that we can calculate the worst-case approximation factor as $f(k) = \frac{1}{\cos(\frac{180^\circ}{k})}$, so we know our average approximation factors will be below $f(4) = \sqrt{2} = 1.414$, $f(8) = 1.082$ and $f(16) = 1.020$ respectively.

7.2.2 Breakdown of time usage

For the main experiments we only measured the total time used to creating the oracle or perform a query, but a more in-depth breakdown of time usage is also interesting. This was only done for a few polygons with close to 500 vertices, and only for the sawtooth oracle.

8 Results and analysis

In this section we show the results of the experiments described in the previous section, and analyse them.

The experiments were ran on a machine with Windows 10, an AMD Ryzen 7 5800 8-core processor, and 16 GB memory. Note that the whole program is single-threaded. The main experiments were ran over the course of about 10 days, 15 hours per day. During part of this time the machine was also running other programs, resulting in slight variations between some measurements.

All graphs are scatterplots, where each dot is a polygon colored by its class. The x-axis shows the number of vertices of the polygon, the y-axis shows one of the described measures.

8.1 Query times

First we will look at the query times, shown in Figs. 28 and 29. The naive query time is shown twice for each value of k, once with the pacman polygons and once without. This was done because the pacman polygons had significantly higher query time with the naive oracle, making the other points hard to distinguish.

8.1.1 Non-trivial queries

In every figure depicting query time we can see that the non-trivial pacman queries have a remarkably long query time. This is because trivial queries are significantly faster to answer than non-trivial ones, namely in $O(\log n)$ time for both oracles: after shooting a single ray and finding the two query points are mutually visible, the query algorithms terminate. By removing the trivial queries we essentially forced the oracles to go through their whole query algorithm every time, increasing the average run-time.

In particular the nontrivial pacman queries are almost exactly twice as slow as the regular pacman queries, because roughly half the regular queries were mutually visible. The same trend can be seen in the approximation factor in Fig. 33, where the approximation factor of the nontrivial queries is about twice as high as that of the regular queries.

8.1.2 Naive oracle

The query times for the naive oracle are shown in Fig. 28. As shown in Section 6.8, the query time is $O(n^2)$ in the worst case. However many polygons seem to display a fairly linear relation between number of vertices and query time, with some outliers that take more time. This near-linear lower bound is caused by polygons that have relatively few visible vertices for query points, leading to much fewer g_p, g_q pairs that need to be considered than the worst-case $O(n^2)$. The reason that there are no outliers with a smaller query time is the fact that both query points first have to go through all vertices to calculate if it is visible, which takes $O(n \log n)$ time in any case.

The outliers take more than linear time are mostly of the classes SRPG_perturbed_smo and SRPG_perturbed_smr, and of course the pacman polygons. All these polygons generally consist of smooth curves with a lot of vertices very close to each other, as can be seen in Fig. 26. This results in a large number of visible vertices from any query point, and thus a large number of potential g_p, g_q pairs to consider. On the left of Fig. 28 we see that the pacman polygons, which were designed to have $O(n)$ visible vertices for any query point, indeed seems to follows a quadratic curve.

8.1.3 Sawtooth oracle

In Fig. 29 we can see the logarithmic nature of the sawtooth query time. Most polygons with a similar number of vertices have a very similar query time with fairly few outliers, barring the non trivial queries. In Section 6.5 we proved that our implementation has an $O(k^2 \log n + k \log^2 n)$ upper bound on the query time, but there is also an $\Omega(k \log^2 n)$ lower bound of finding the gateway sets $G(p)$ and $G(q)$. Because this upper and lower bound are quite close together, especially for small values of k like we used. This causes there to be few outliers.

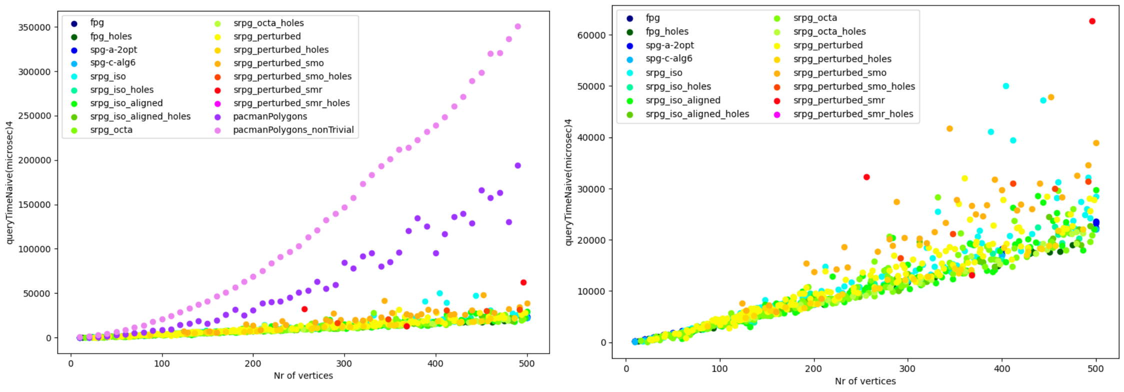


Figure 28: Query times for the naive oracle

A remarkable measurement is the polygon of class SPF_perturbed_smr with around 380 vertices that has an extremely small query time, with all values of k. This is actually the polygon shown on the bottom right in Fig. 26. Note that the same polygon also has an approximation ration very close to 1. In this polygon, two random points have a very high chance to be mutually visible. This means almost all generated queries will be trivial, making the average query time very small.

The pacman polygons also have a large chance for trivial queries. However the regular pacman polygon queries are not particularly fast, and for $k = 16$ they are even slower than average. This might be caused by the fact that these have a large number of gateway points per query point, because the projection from the query point to a cutline is unlikely to be obstructed.

Lastly, for $k = 8$ or $k = 16$ we see that both types of SPG polygons have a shorter query time than average. This was unexpected as these polygons do not appear to have a large chance for trivial queries, nor a very small amount of gateways per query point. If more time was available to investigate this, we would calculate these two statistics and compare them to other classes. If SPG polygons are not outliers in both statistics, we would know to look elsewhere for an explanation.

8.1.4 Comparison

Just from their worst-case time complexities we would expect the sawtooth oracle to easily outperform the naive oracle in query time. However in the figures we can see that they actually have relatively similar query times in practice, for the polygons we used. For $k = 4$ the sawtooth oracle starts to have lower query times for some polygons around 200 vertices, and for $k = 16$ this only starts around 500 vertices. Clearly for every value of k there will be a value of n where the sawtooth oracle outperforms the naive oracle, since that is what our worst-case time complexities tell us, but this value of n can quickly get quite large.

Importantly, the sawtooth oracle does perform much better in the worst case: the pacman polygons are only slightly slower than most other classes, while for the naive oracle the query time quickly rises to unreasonable levels.

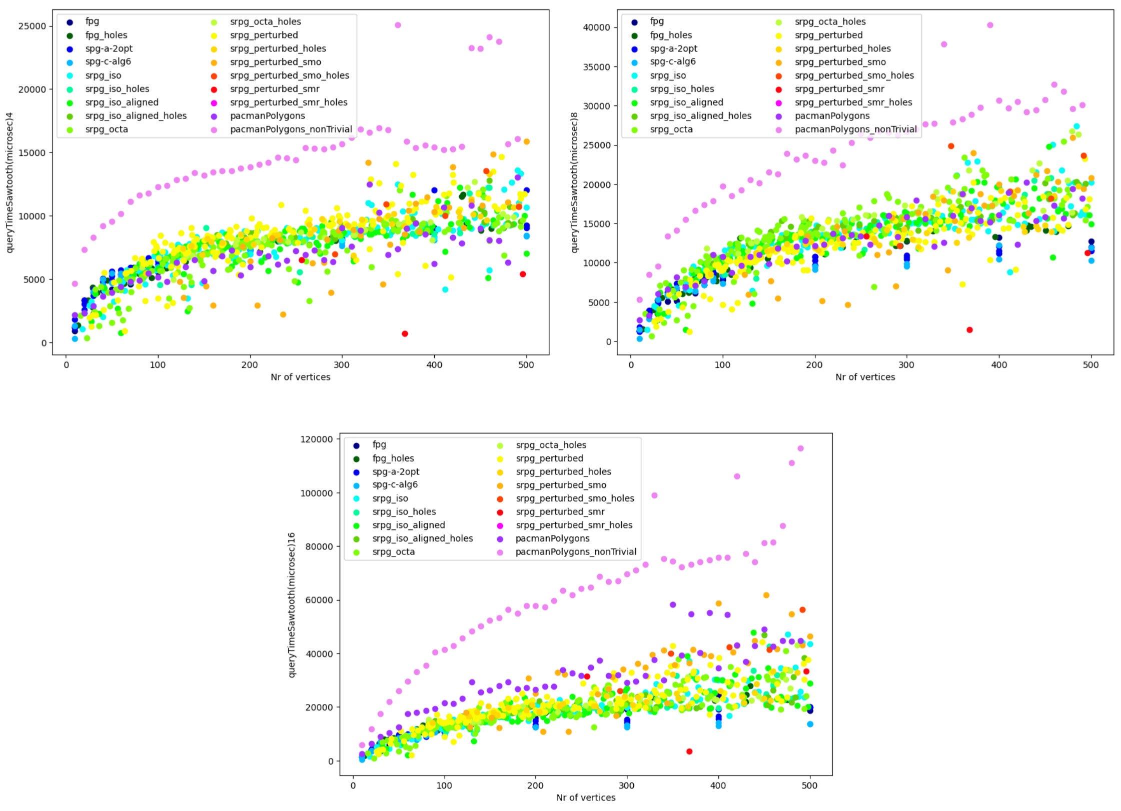


Figure 29: Query times for the sawtooth oracle

8.2 Preprocessing

Next we look at the time and space usage of the preprocessing, where we create the oracles.

8.2.1 Build time

Table 1 shows a breakdown of the time usage when building the sawtooth oracle. These values are the average over 6 polygons from the FPG class with between 400 and 500 vertices, using $k = 4$. In the table we can see that over 95% of the preprocessing time is spent on creating the rayshooting datastructure. This seems excessively long, considering that the worst-case complexities of creating the rayshooting datastructure and calculating the all pair shortest paths table 1 very similar. This might indicate there is something wrong in the implementation of the persistent binary trees. If more time was available we would investigate this by comparing the current implementation to a more naive one that uses $O(n^3)$ space, in which we replace the persistent binary trees by regular sorted lists

Process	Time used (microsec)	Time used (%)
Rayshooting: creating dual arrangement	339185	0.2%
Rayshooting: creating dual point location	8754713	5.1%
Rayshooting: creating binary search trees	156304887	91.2%
Calculating type-1 Steiner points	115040	<0.1%
Calculating type-2 Steiner points	1120573	0.6%
Calculating all pair shortest paths	4680115	2.7%

Table 1: A breakdown of the time used in creating the sawtooth oracle

Additionally, in Fig. 30 we can see that the naive oracle and the sawtooth oracle have almost identical build times. This is because both oracles require the rayshooting datastructure, which turns out to be relatively costly to build. As k grows the sawtooth oracle will eventually take longer to build, because the larger value of k leads to more Steiner points which makes calculating the all pair shortest paths slower. For the values we used in our experiment this was hardly noticeable however.

In the figure we can clearly see that the SRPG_iso_aligned and SRPG_octa polygons have a significantly smaller build time than the other classes. The reason for this is that these polygons contain many edges that are on the same supporting line, because all edges are aligned on a regular grid. This causes the dual space to have a lower complexity: when multiple primal edges are on the same supporting line, their dual wedges will intersect only in their middle point, creating fewer dual faces. Having fewer dual faces makes all steps in the creation of the rayshooting datastructure faster, and since that is the most time consuming step this has a relatively large impact on the build time as a whole.

8.2.2 Build space

In Fig. 31 we can see there is actually a large difference in space usage between the naive oracle and the sawtooth oracle. This difference gets larger as k grows. This is caused by the all pair shortest path table which takes $O(n^2k^2 4^{\sqrt{\log n}} \log n)$ space.

We can also see that the sawtooth oracles for SRPG_perturbed_smo and SRPG_perturbed_smr polygons and most notably the pacman polygons require more space than the other classes. These classes have a larger amount of Steiner points, similar to why their query time was larger for both oracles, which causes the table to grow.

For the naive oracle the RPG_iso_aligned and SRPG_octa polygons use less space, just like they used less preprocessing time. Since most of the space usage of the naive oracle is in the rayshooting datastructure, the reduced complexity of the dual space also reduces the total space usage significantly.

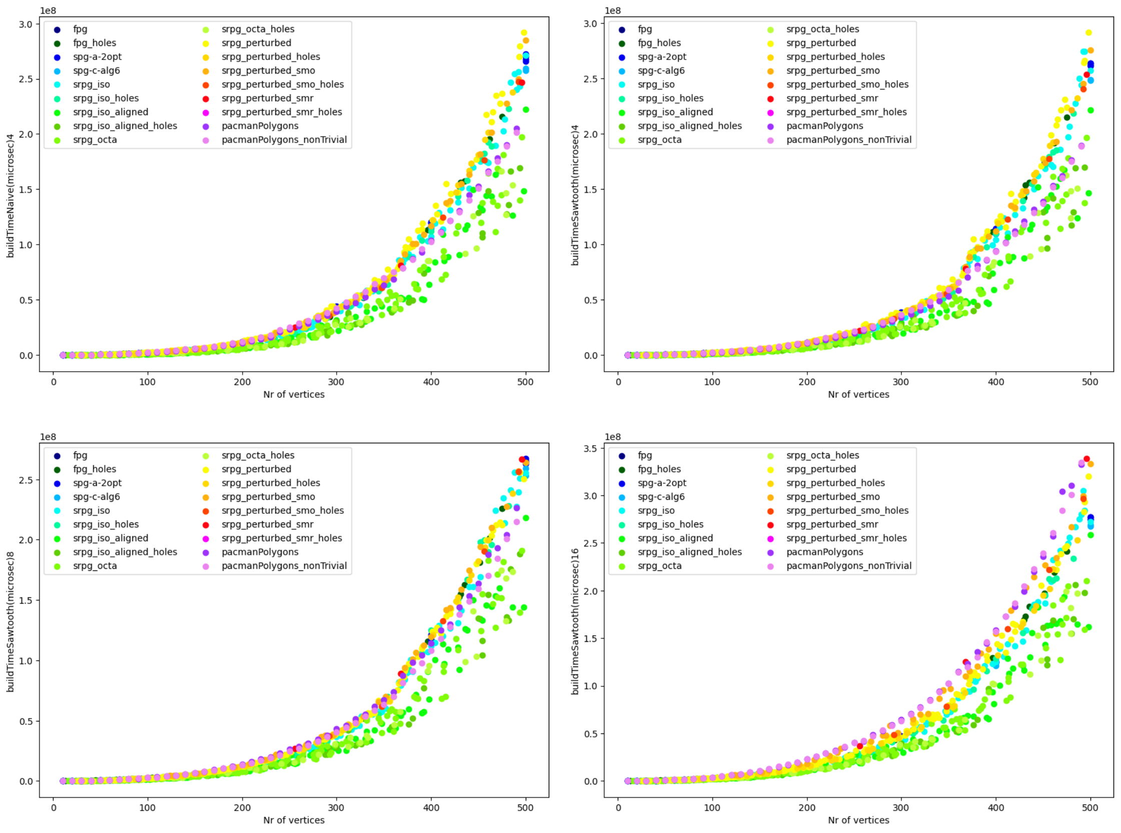


Figure 30: Build times for the naive oracle (topleft), and the sawtooth oracles with $k = 4$ (topright), $k = 8$ (bottom left) and $k = 16$ (bottom right)

8.3 Approximation factor

Fig. 33 shows the approximation factors for our polygons. For polygons with a low vertex count the approximation factor tends to be closer to 1, since there will be more trivial paths due to there being less edges to obstruct a direct path between query points. From about 200 vertices and on, the vertex count has little impact on the approximation factor.

Recall that our worst-case approximation factors for $k \in \{4, 8, 16\}$ are $f(4) = \sqrt{2} = 1.414$, $f(8) = 1.082$ and $f(16) = 1.020$ respectively. We see that the average approximation factor is about half as bad as it would be worst-case, for example for $k = 4$ most measurements lie between 1.1 and 1.2.

One major influence on the average approximation factor of a polygon is illustrated in Fig. 32 for $k = 4$. In the figure we see two paths from p to q, one over a cutline and one over a polygon edge. We know that a path over a cutline always uses worst-case angles: it travels to the cutline in direction d_{i+1}, then over the cutline in direction d_i, then away from the cutline in direction d_{i+1} again. A path over an edge also travels in some direction between d_i and d_{i+1} over that edge, which is a shortcut by triangle inequality. This generally means that the more a path uses type-1 Steiner points, the better its approximation factor is. This creates differences between classes. For example the RPG_iso_aligned polygons consist of only vertical or horizontal edges, meaning type-1 paths over edges are actually never shorter in these polygons; for all our values of k they have the exact same length as type-2 paths via cutlines. For this reason the RPG_iso_aligned polygons have a higher approximation factor than average for all values of k we used. The SRPG_octa polygons also have diagonal edges. These can act as shortcuts for $k = 4$ so their approximation factor for this value of k is quite average, but for $k = 8$ and larger they no longer provide shortcuts and therefore have a worse than average approximation factor. The larger the value of k used the less noticeable this shortcut-effect becomes.

The other major influence is the percentage of trivial queries, as mentioned before. A trivial

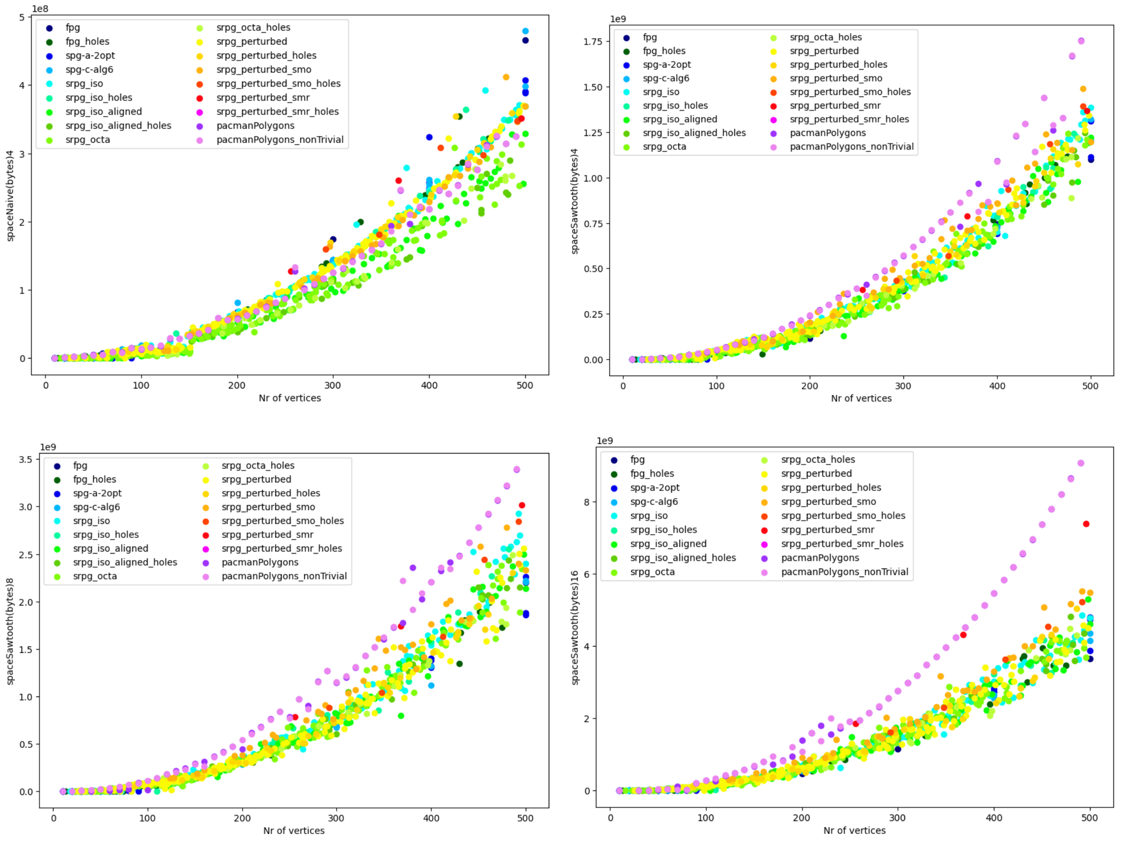


Figure 31: Space usage for the naive oracle (topleft), and the sawtooth oracles with $k = 4$ (topright), $k = 8$ (bottom left) and $k = 16$ (bottom right)

query will always be answered with an approximation factor of 1, since the straight line between the two query points is clearly the shortest possible path.

The SRPG_perturbed_smo and SRPG_perturbed_smr are examples of polygons with a below average approximation factor for all values of k. These polygons have smooth edges allowing for easy type-1 shortcuts, and are also made from a lot of open space which leads to a larger number of trivial queries.

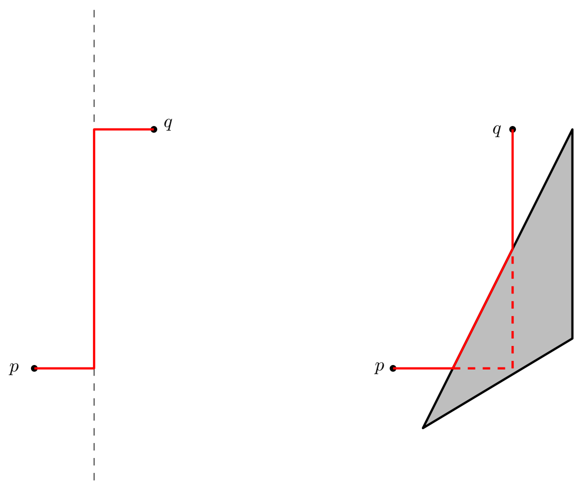


Figure 32: A path through a type-1 Steiner point over a polygon edge (right) often has a better approximation factor than a path through a type-2 Steiner point over a cutline (left)

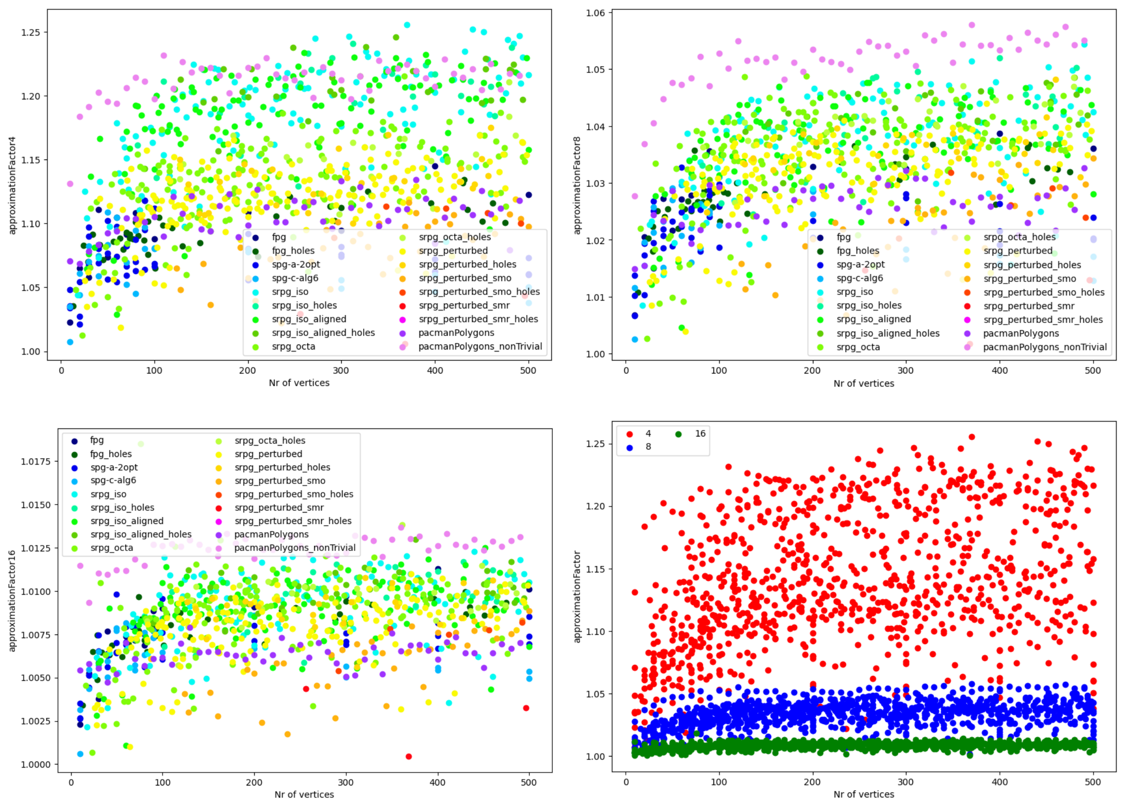


Figure 33: Approximation factors for the sawtooth oracle with $k = 4$ (topleft), $k = 8$ (topright), $k = 16$ (bottom left) and all three together (bottom right)

9 Conclusion

In this thesis we have proposed a datastructure to answer approximate shortest path queries, resulting in Theorem 1. We implemented our datastructure along with a naive one, and compared their performance on a set of random polygons.

As proven in Lemma 1 we need to choose $k \geq \frac{180^\circ}{\cos^{-1}(\frac{1}{\epsilon})}$ to get get a certain ϵ-approximation, and our experiments showed the average approximation factor was around half as bad the theoretical worst-case. In general, we do not need to use a very large value of k to get an approximation factor that is satisfactory for many use-cases. This is good news because both the query time and the preprocessing time and space grow quadratically with increasing k, due to the increased number of Steiner points and gateways.

We also saw that the sawtooth oracle has lower query times than the naive oracle for larger polygons. However, the space and time usage of the preprocessing also increases rapidly with the number of vertices. Even for relatively small polygons of up to 500 vertices like the ones used in our experiments, the preprocessing can already take up to 5 minutes. This gives the sawtooth oracle as a whole somewhat poor scalability: even though it would be able to quickly answer queries on very large polygons, the space and time required to build it can often make the oracle impractical to use.

9.1 Future work

It would be interesting to compare our datastructure to Thorups datastructure from [9]. These both provide ϵ-approximate shortest paths in logarithmic time, but Thorups datastructure uses only $O(\frac{n \log^2 n}{\epsilon})$ space while ours uses $O(\frac{n^2}{\epsilon^2} 4^{\sqrt{\log n}} \log n)$ space. Comparing how they perform in experiments might give more meaningful insight in how relevant our datastructure can be in practice.

The preprocessing time and space of our datastructure might also still be improved. Firstly the rayshooting datastructure could be investigated, to see if the implementation is correct. Secondly the techniques used in [12] to improve the space usage of the exact Manhattan oracle to $O(n+h^{2+\delta})$ might be applicable to our oracle as well; this would first have to be proven in theory, and then implemented.

Lastly there might be a different way to store all pair shortest paths in less than $O(n^2)$ or even less than $O(h^2)$ space. We have tried to come up with such a datastructure, but this has so far been unsuccessful. Currently a simple quadratic table is used to store the (approximate) shortest distance between all pairs of Steiner points. However the cutline trees and Steiner points also contain some information about shortest paths. We hoped to be able to use this information to find the distance between two points, without storing it explicitly. This turned out to be difficult, because two paths might cross the same cutlines while still being completely different. For example in Fig. 34 we see four paths from p to $q_1..q_4$. Although these paths all cross the same cutlines $l_1..l_3$ in the same order, they are topologically very different. Another complicating factor is that we have k different cutline trees, so a path through multiple Steiner points can traverse many different cutline trees. Despite these difficulties it would be interesting to investigate this more thoroughly.

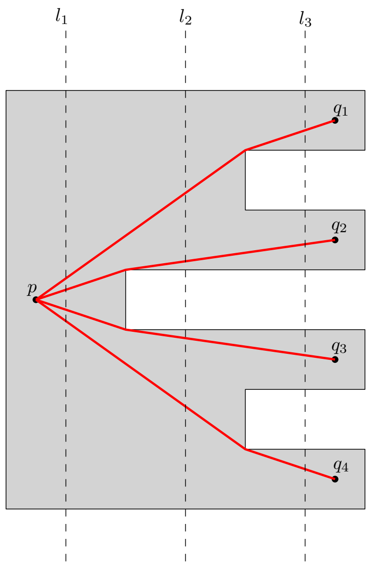


Figure 34: Four very different paths through the same cutlines

References

[1] Yi-Jen Chiang and Joseph S. B. Mitchell. Two-point euclidean shortest path queries in the plane. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '99, page 215–224, USA, 1999. Society for Industrial and Applied Mathematics.

[2] K. Clarkson, S. Kapoor, and P. Vaidya. Rectilinear shortest paths through polygonal obstacles in $o(n \log^{3/2} n)$ time. In *Proceedings of the Third Annual Symposium on Computational Geometry*, SCG '87, page 251–257, New York, NY, USA, 1987. Association for Computing Machinery.

[3] J.-R. Sack and J. Urrutia, editors. *Handbook of Computational Geometry.* North-Holland Publishing Co., NLD, 2000.

[4] Michel Pocchiola and Gert Vegter. Computing the visibility graph via pseudo-triangulations. In *Proceedings of the Eleventh Annual Symposium on Computational Geometry*, SCG '95, page 248–257, New York, NY, USA, 1995. Association for Computing Machinery.

[5] Stéphane Rivière. Topologically sweeping the visibility complex of polygonal scenes. In *11th Annual ACM Symposium on Computational Geometry*, pages 436–437, Vancouver, Canada, 1995.

[6] Haitao Wang. A new algorithm for euclidean shortest paths in the plane. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2021, page 975–988, New York, NY, USA, 2021. Association for Computing Machinery.

[7] L. J. Guibas and J. Hershberger. Optimal shortest path queries in a simple polygon. In *Proceedings of the Third Annual Symposium on Computational Geometry*, SCG '87, page 50–63, New York, NY, USA, 1987. Association for Computing Machinery.

[8] Hua Guo, Anil Maheshwari, and Jörg-Rüdiger Sack. Shortest path queries in polygonal domains. In *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management*, AAIM '08, page 200–211, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] Mikkel Thorup. Compact oracles for approximate distances around obstacles in the plane. In *Proceedings of the 15th European Symposium on Algorithms (ESA), LNCS 4698*, Lecture notes in computer science, pages 383–394. Springer, 2007.

[10] K. Clarkson. Approximation algorithms for shortest path motion planning. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, page 56–65, New York, NY, USA, 1987. Association for Computing Machinery.

[11] Danny Z. Chen, Kevin S. Klenk, and Hung-Yi T. Tu. Shortest path queries among weighted obstacles in the rectilinear plane. In *Proceedings of the Eleventh Annual Symposium on Computational Geometry*, SCG '95, page 370–379, New York, NY, USA, 1995. Association for Computing Machinery.

[12] Danny Z. Chen, Rajasekhar Inkulu, and Haitao Wang. Two-point l1 shortest path queries in the plane. In *Proceedings of the Thirtieth Annual Symposium on Computational Geometry*, SOCG'14, page 406–415, New York, NY, USA, 2014. Association for Computing Machinery.

[13] The CGAL Project. *CGAL User and Reference Manual.* CGAL Editorial Board, 5.4.1 edition, 2022.

[14] Ron Wein, Eric Berberich, Efi Fogel, Dan Halperin, Michael Hemmer, Oren Salzman, and Baruch Zukerman. 2D arrangements. In *CGAL User and Reference Manual.* CGAL Editorial Board, 5.4.1 edition, 2022.

[15] P. K. Agarwal. Ray shooting and other applications of spanning trees with low stabbing number. In *Proceedings of the Fifth Annual Symposium on Computational Geometry*, SCG '89, page 315–325, New York, NY, USA, 1989. Association for Computing Machinery.

[16] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, jul 1986.

[17] Bartosz Milewski. Functional data structures in c++: Trees, 2013. Available at https://bartoszmilewski.com/2013/11/25/functional-data-structures-in-c-trees/, last accessed on 01-07-2022.

[18] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[19] Günther Eder, Martin Held, Steinfor Jasonarson, Philipp Mayer, and Peter Palfrader. Salzburg database of polygonal data: Polygons and their generators. *Data in Brief*, 31:105984, 2020.