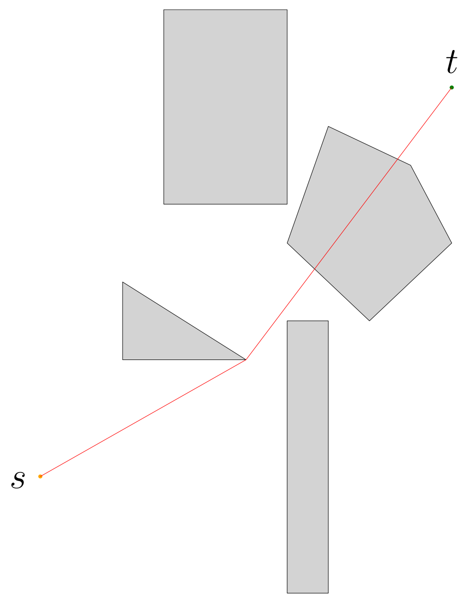


Aarhus University, Department of Computer Science

Master's Thesis
Shortest Paths in the Plane with Polygon Violations

Nick Bakkegaard - 20114270
Peter Burgaard - 201209175
Advisor: Gerth Stølting Brodal

June 15, 2018

Abstract

This thesis is dedicated to exploring a solution to the shortest path in the plane with polygonal obstacles violation problem. First, we explore and implement a naive $O(n^3)$ solution to introduce the problem and key concepts. Next, we move on to a $O(n \log n)$ algorithm which solves the shortest path in the plane without polygonal obstacle problem, which is due to Hershberger and Suri [6], where one build a shortest path map, in which one can find the shortest path without violations in time $O(\log n)$. This algorithm is extended to the main $O(k^2 n \log n)$ solution which is due to Hershberger, Kumar and Suri [5], in which the algorithm builds a shortest k-path map, from which the shortest path with k violations can be queried in $O(\log n)$ time.

Acknowledgement

We want to thank Niels Bross and Kresten Maigaard Axelsen for being great office buddies and also being up for a nice chat and a cup of coffee.

We also want to thank Andrease Malling Østergaard and Andreas Østergaard Nielsen for being great skribbl.io opponents.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

Given a starting point s, an endpoint t and a set of polygons $\mathcal{O}$ in the plane, we want to find the shortest path from s to t without traveling through the interior of any polygon in $\mathcal{O}$. This is an old and well studied problem, and historically there have been two conceptually different approaches to the problem, one using visibility graphs and another using the continuous Dijkstra method.

The visibility graph approach is to construct a graph of all paths between every pair of vertices in the plane which does not go through an interior of an obstacle. These paths are called legal paths. Then the shortest path is found by then running a single source shortest path algorithm on that graph. A problem of this approach is that the complexity of the graph can be $\Theta(n^2)$, with n begin the number of vertices of the polygons, which makes it difficult to get below this threshold.

The continuous Dijkstra method works by simulating a *wavefront propagation*. A wavefront is a circular arch which originates from a point and expands in at unit-speed (non-changing speed) such that all points of the boundary of the arch have equal distance d to the generator point from which the wavefront originated, at every time t. This point is the source (start point) s, see Figure 1.1.

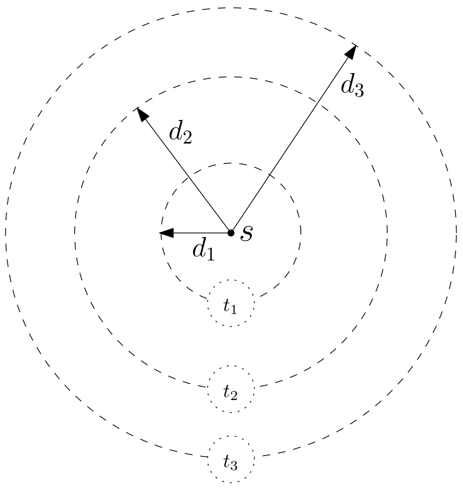


Figure 1.1: A right turn formed by three points

From the figure we can see that since the wavefront expands at unit-speed, the time distance d_i at time t_i are equal, are one therefore is only concerned with the time t. Every time a vertex v is reached a new wavefront starts emitting from v. This way the plane will be divided into different areas depending on which wavefront encapsulates it, see Figure 1.2.

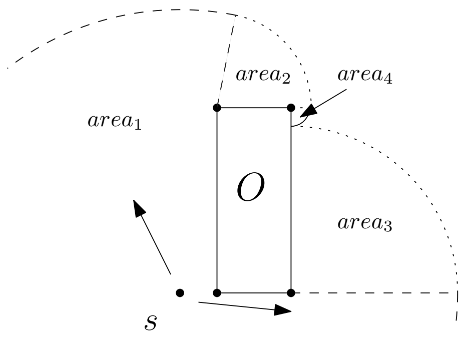


Figure 1.2: A simple example of the wavefront spreading in the plane around an obstacle O and encapsulating different areas.

In 1999 Hershberger and Suri published the paper "an optimal algorithm for euclidean shortest paths in the plane"[6] in which they presented an optimal $O(n \log n)$ time algorithm which matched the lower bound of the problem (see Chapter 8), using an implementation of the continuous Dijkstra method.

In 2017 they, together with Kumar, looked at a generalization of the problem: given the same setting as before, one is now allowed to go through up to k obstacles. See Figure 1.3.

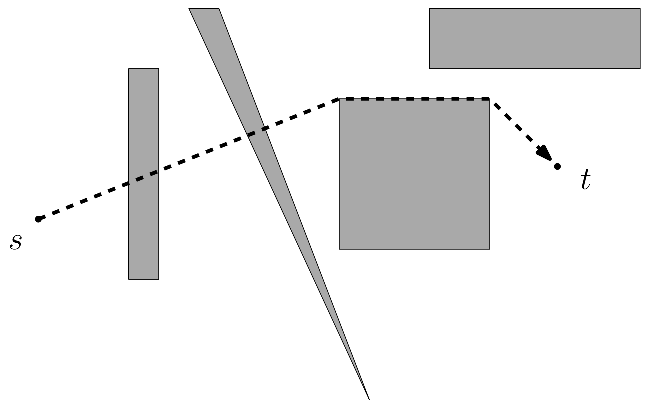


Figure 1.3: A shortest k-path from s to t with $k = 2$

The problem lies in which polygons you should pass through to minimize the distance from s to t. They presented an $O(k^2 n \log n)$ algorithm for this problem, which used a modified version of the 1999 algorithm[5]. Notice that if $k = 0$ it is identical to the Shortest path with no violations.

In this thesis we are going to present implementation details of both the original and the generalized problem. We start by describing and implementing a naive algorithm for solving the problem based on computing a visibility graph and then running the Dijkstra single source shortest path algorithm. Afterwards we are going to explain the theoretical results leading to the algorithm and the implementation details. Lastly we describe the algorithm from 2017 and its implementation details.

1.1 Formal problem description

Given two points in the plane $s, t \in \mathbb{R}^2$ and a list of polygons $\mathcal{O} = o_1, \dots, o_h$ where o_i is a list of points in polygon o_i starting at an arbitrary place and in clockwise order (note that sometimes we use $\mathcal{O}$ as a set e.g. polygon $o_i \in \mathcal{O}$). We say a *legal path* is a list of points where two adjacent points are mutually visible, i.e. you are able to draw a line from one to the other without crossing the interior of any polygon in $\mathcal{O}$. We want to find a shortest legal path from s to t.

1.2 Previous work

Year	Paper	Run time	Space	Visibility graph	SPM
	Naive[1]	$O(n^3)$	$O(\lvert E \rvert)$	x	
1978	Lee [11][2]	$O(n^2 \log n)$	?	x	
1985	Welzl [18]	$O(n^2)$	$O(n^2)$	x	
1991	Ghosh et al. [4][3]	$O(E + n \log n)$	$O(E + n)$	x	
1991	Mitchell [13][4]	$O(kn \log^2 n)$	$O(n)$		x
1996	Mitchell [14][5]	$O(n^{3/2+\varepsilon})$	$O(n)$		x
1999	Hershberger et al. [6]	$O(n \log n)$	$O(n \log n)$		x

Table 1.1: Shortest Paths in the Plane with Polygon obstacles algorithms

In 1978 Lee presented a $O(n^2 \log n)$ algorithm for constructing a visibility map in his Ph.d. thesis[11], we were unable to find the original paper, the running time is taken from [6].

Seven years later, in 1985, Welzl [18] published an $O(n^2)$ time algorithm consuming $O(n^2)$ for construction of a visibility map.

Six years later in 1991 Ghosh et al. [4] presented an $O(E+n \log n)$ algorithm consuming $O(E + n)$ space, for constructing a visibility graph where E is the number of edges in the visibility graph. Since the visibility graph can contain $O(n^2)$ the algorithm is output bound and people started making shortest path map algorithm instead, hoping to reach the $\Omega(n \log n)$ lower bound (see chapter 8).

That same year in 1991 Michell [13] published an algorithm for constructing the shortest path map, with running time $O(kn \log^2 n)$ and $O(n)$ space consumption. Where k is bounded by the number of different obstacles that touches any shortest path from s.

[1]See Chapter 2

[2]We were not able to obtain the original ph.d. thesis, the got the running time from [6]

[3]Where E is the number of edges in the visibility graph

[4]Where k is a number bounded by the number of different obstacles that touches any shortest path from s

[5]For any $\varepsilon > 0$ where the constant in the big-Oh notion depending on ε

In 1996 he improved the run time to $O(n^{3/2+\varepsilon})$, for any $\varepsilon > 0$ where the constant in big-Oh notion depends on ε keeping the linear space consumption [14].

Then finally in 1999 Hershberger et al. [6] revealed a $O(n \log n)$ algorithm for computing the shortest path map, matching the lower bound. The space consumption is $O(n \log n)$ and it is still an open problem if there exists an algorithm with run time $O(n \log n)$ and linear space consumption.

1.3 Overview of thesis

Chapter 2 We describe a simple $O(n^3)$ algorithm for constructing a visibility graph. Then we implement it and compare the run time to the theoretical time.

Chapter 3 Gives an overview of the Hershberger et al. [6] algorithm.

Chapter 4 Goes through the formal definition of the shortest path maps and its geometric properties including the complexity of the map.

Chapter 5 Explains what the conforming subdivision is and how it is constructed.

Chapter 6 Is dedicated to the wavefront propagation algorithm explaining how it works and its implementation details.

Chapter 7 Presents the Hershberger et al. [5] which shows how to extend the original algorithm to work with k violations.

Chapter 8 In here we prove the lower bound of the "shortest path in the plane with obstacles problem in The Algebraic Computation Tree Model. By making a reduction from number distinction to number sorting to shortest map in the plane with obstacles.

Chapter 9 We conclude the work we have done.

Chapter 2

Simple $O(n^3)$ implementation

In this section we describe an $O(n^3)$ implementation which solves the "Shortest Paths in Plane with Obstacles Violations"-problem using visibility graphs. Recall that n is the number of vertices in the polygons and k is the number of polygon violations allowed. In Section 2.1 we describe how we construct the visibility graph. Imagine a plane with a starting point s, an end point t, and $\mathcal{O}$ polygon obstacles, which is build of groups of connected points, each representing an obstacle. A visibility graph is a graph where for each set of points $p, q \in \mathcal{O} \cup \{s, t\}$ there is an edge between them if the two points can see each other without going (or looking) through any interior of an obstacle (see Figure 2.1). In Section 2.2 we explain Dijkstra's algorithm for finding the shortest path from s to t in the visibility graph and finally in Section 2.3 we test our implementation to verify that the actual running time is the same as the theoretically predicted one.

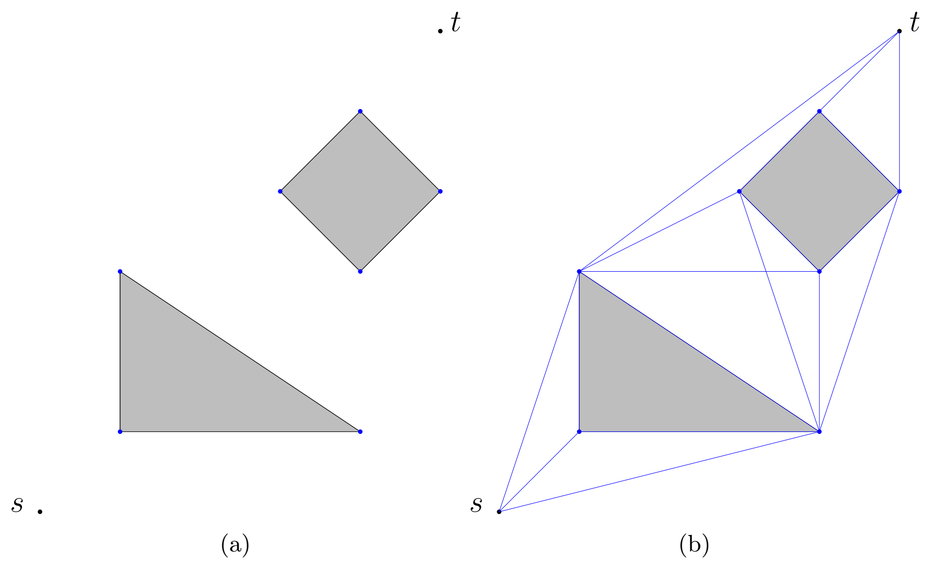


Figure 2.1: Example of visibility graph

2.1 Constructing the visibility graph

The naive way of constructing a visibility graph is to make a graph where every pair of points $p, q \in \mathcal{O} \cup \{s, t\}$ is connected to each other, then removing all edges that cross the interior of a polygon. But in this setting we are allowed to cross k polygons, so we construct the graph a bit differently. Given a set $\mathcal{O}$ consisting of all the polygons, where each polygon is a list of the points in the polygon we use the following algorithm 1. Create a graph $G_0 = (V, E)$, where V contains all the vertices in $\mathcal{O} \cup \{s, t\}$, and let E contain all possible connections between the vertices. Make k copies of the graph G_0 and name them $G_1, \ldots, G_k$. Algorithm 1 goes as follows: for each graph G_i, take each edge $e_j \in G_i$ and call NumberOfCrosses(e_j), which returns the number m of polygons from $\mathcal{O}$ that the line segment crosses, and connect the endpoint to the corresponding point in G_{i+m}, i.e. if you take an edge in the graph, that goes through m polygons, you travel m graphs up. If $i + m > k$ then delete the edge from G_i. We now have a graph that has k levels where every time you go through k polygons you go k levels up.

Algorithm 1 MakeVisibilityGraph($\mathcal{O}, s, t$)

```
1: for all $G_i = (V_i, E_i) \in G$ do
2:     for all $e \in E_i$ do
3:         $crosses = numberOfCrossings(e)$
4:         if $crosses + i \leq k$ then
5:             make $e$ go from $G_i$ to $G_{i+crosses}$
6:         else
7:             delete edge $e$
```

The only missing part is the numberOfCrossings function, which we define below.

2.1.1 Number of crossings

Calculating the number of polygons a line segment crosses is no trivial task, since there is a number of edge cases. We try to give a brief intuition of the edge cases, and then we present our algorithm. The first five cases (a-e) in Figure 2.2 are allowed intersections since it is only the interior of a polygon that we can not travel. The next five cases (f-j) are not allowed since they travel through the interior of the polygon.

So, given a polygon $o \in \mathcal{O}$, and a line segment l we want to determine if l crosses the polygon O. We start by making each obstacle (i.e. list of points $o = p_1, p_2, \ldots, p_i$) into a list of line segment $o' = (p_1, p_2), (p_2, p_3), \ldots, (p_{i-1}, p_i), (p_i, p_1)$. Then we observe that if a line segment crosses a line segment of a polygon, it counts as a crossing (cases f and g). The other three cases of crossing (cases h, i and j) all have in commonm that the line segment crosses four end points of the polygon. So we could say it is not allowed to cross four points of a polygons line segments. The problem is that it makes (cases a, c and d) illegal. But fortunately they all have in common that they are collinear (they lie on a common line) with a line segment of the polygon, so the algorithm is as follows:

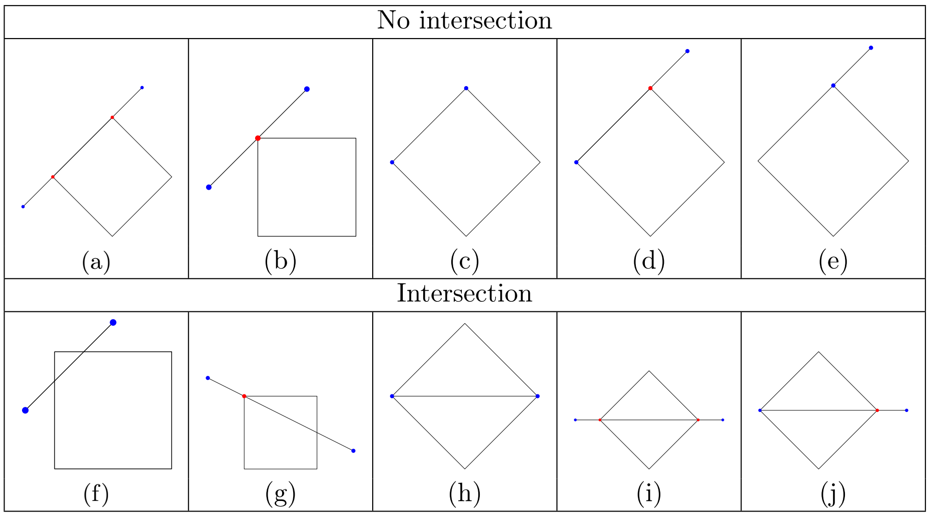


Figure 2.2: A collection of 10 different cases showing what we have defined as an intersection between a polygon and a line segment

1. if a line segment l_1 crosses another line segment l_2 of a polygon it crosses the polygon
2. if a line segment has four points in common with the polygon, it crosses the polygon, unless the line segment is collinear with a line segment of the polygon

This leads us to the following algorithm:

Algorithm 2 NumberOfCrossings(l, $\mathcal{O}$)

```
howManyCrosses=0
for each $o \in O$ do
    count= 0
    for each $l' \in o$ do
        result=crosses($l'$, $l$)
        if result==-1 then
            counter= 4
            Break
        else if result==0 then
            count=count+1
    if $counter > 3$ then
        howManyCrosses=howManyCrosses+1
return howManyCrosses
```

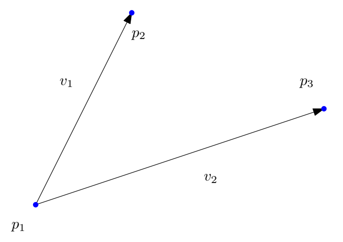
2.1.2 Crosses

To make a crosses function, we need a right turn function. Consider three points p_1, p_2, p_3 in the plane and make a line that goes through p_1 and p_2. Now if we stand at point p_1 and look in the direction of p_2 , if p_3 does not lie on the same line as p_1 and p_2 will it be on the right or the left of the line. Let $p_i.x$ and $p_i.y$ denote the x-coordinates and y-coordinates respectively. To find out whether the three points form a right turn, a left turn or are collinear we make the following two vectors.

$$v_1 = p_2 - p_1 = \langle p_2.x - p_1.x, p_2.y - p_1.y\rangle$$
$$v_2 = p_3 - p_1 = \langle p_3.x - p_1.x, p_3.y - p_1.y\rangle$$

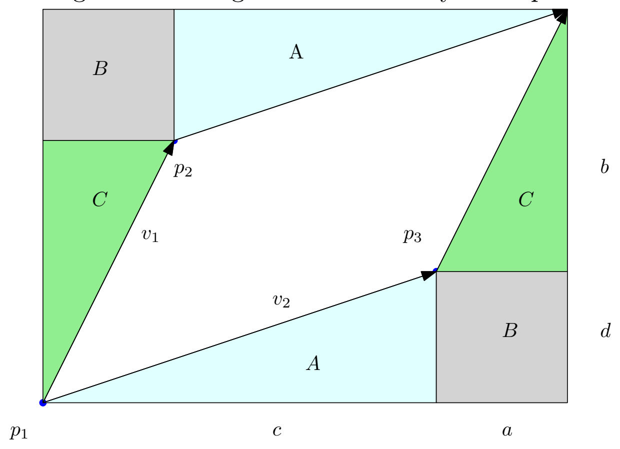
Let us denote $v_1 = \langle a, b\rangle$ and $v_2 = \langle c, d\rangle$ (see Figure 2.3a)

2.1.3 Right turn



(a)

Figure 2.3: A right turn formed by three points



(a)

Figure 2.4: Area are of a parallelogram given by two vectors

We claim that we can calculate the turn by calculating the signed area of the parallelogram spanned by the two vectors (see Figure 2.4a). The area of their parallelogram can be calculated as follows: calculate the area of the big rectangle, and take the two small triangles and the two small squares and subtract that area.

$$\begin{aligned} \text{area} &= (a+c)(d+b) - 2A - 2B - 2C \\ &= ad + ab + cd + bc - cd - 2ad - ab \\ &= bc - ad \\ &= (p_2.y - p_1.y)(p_3.x - p_1.x) - (p_2.x - p_1.x)(p_3.y - p_1.y) \qquad (2.1) \end{aligned}$$

Now we claim that the area between these two vectors is positive if the three points form a right turn, and negative if they form a left turn. We illustrate that by an example (see Figure 2.5)

Given our formula (formula 2.1) we get that the q_1, q_3, q_2 area is

$$\begin{aligned} & (q_3.y - q_1.y)(q_2.x - q_1.x) - (q_3.x - q_1.x)(q_2.y - q_1.y) \\ = \quad & (2-0)(-1-0) - (0-0)(1-0) \\ = \quad & 2 \cdot (-1) - 0 \cdot 1 \\ = & -2 - 0 \\ = & -2 \end{aligned}$$

And the area of q_1, q_3, q_4 is

$$\begin{aligned} & (q_3.y - q_1.y)(q_4.x - q_1.x) - (q_3.x - q_1.x)(q_4.y - q_1.y) \\ = & (2-0)(1-0) - (0-0)(1-0) \\ = & (2 \cdot 1 - 0 \cdot 1 \\ = & \, 2 - 0 \\ = & \, 2 \end{aligned}$$

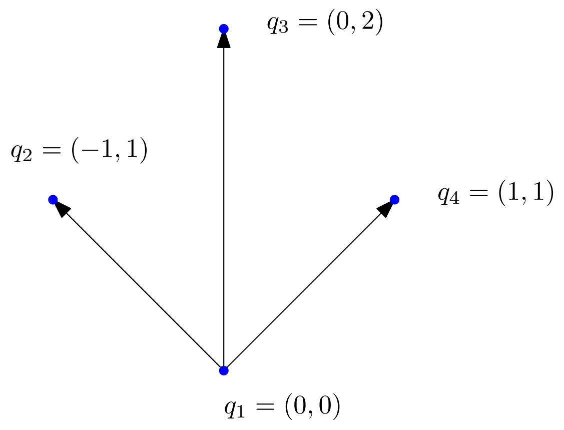


Figure 2.5: Right turn example

The righturn function below will return a negative number if the three points make a left turn, a positive number if it is a right turn and 0 if the three points are on a line.

Algorithm 3 rightTurn(p_1, p_2, p_3)

```
1: return $(p_2.x - p_1.x)(p_3.y - p_1.y) - (p_2.y - p_1.y)(p_3.x - p_1.x)$
```

2.1.4 Crossing of two line segments

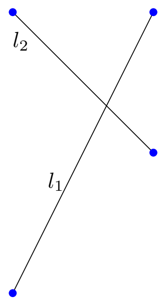


Figure 2.6: Two lines crossing

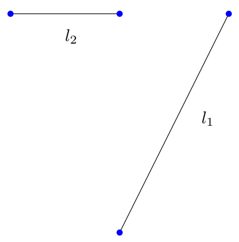


Figure 2.7: Two line which does not cross

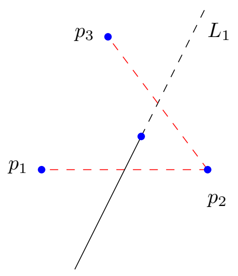


Figure 2.8: $\overline{p_1 p_2}$ passes both tests of 1, while $\overline{p_2 p_3}$ only passes one

Lemma 1. *Given two line segments, l_1 and l_2 we can decide whether they cross by first checking if the two end points of l_2, namely $l_2.p$ and $l_2.q$, lie on separate sides of the line which is collinear to l_1. Should this be the case, we do a similar check for l_2 on l_1. Should both cases be true we know l_1 crosses l_2, see Figure 2.6.*

Proof. Let L_1 and L_2 denote the lines collinear to l_1 and l_2 respectively. If both the end points of l_2 are on the same side of L_1, then the line segments cannot cross L_1, and therefore l_1 obviously(see Figure 2.7). If they lie on opposite sites, l_2 crosses L_1 and we have to determine if l_2 crosses L_1 between $l_1.p$ and $l_1.q$.

We know the line l_2 crosses L_1, the question is, if it is between the two end points. We determine this verifying that if the endpoints of l_1 lie on opposite sites of l_2, like before(see Figure 2.7). If they do, it must be the case that l_1 and l_2 cross, if not, it crosses L_1 in another place. □

2.1.5 Crosses algorithm

We can check if the endpoints of a line segment lie on opposite sites of another line segment by multiplying the right turn results, since, if they lie on opposite sites, they will have different signs, if they lie on the same side they will have the same sign, so the result will be negative if they are on opposite sites and positive if they are on the same side. If both foo and bar is negative, the segments cross and we return -1. If either foo and bar is 0 it means that a point from one segment touches the other line segment and we return 0. Otherwise they do not touch at all, and we return 1.

Algorithm 4 Crosses(l_1, l_2)

```
1: foo = rightTurn($l_1.p, l_1.q, l_2.p$) · rightTurn($l_1.p, l_1.q, l_2.q$)
2: bar = rightTurn($l_2.p, l_2.q, l_1.p$) · rightTurn($l_2.p, l_2.q, l_1.q$)
3: if foo< 0 and bar< 0 then
4:     return $-1$
5: else if foo= 0 or bar= 0 then
6:     return 0
7: else
8:     return 1
```

2.1.6 Run time

Calculating the number of crossing each set edge make takes $O(n^3)$ since there are $O(n^2)$ possible edges and they each can cross $O(n)$ possible polygon edges. Then constructing the k layers takes $O(kn^2)$. Since $k < n$ (if $k \geq n$ the shortest distance will just be the line from s to t) $O(n^3)$ will dominate and the running time will be $O(n^3)$.

2.2 Dijkstra

Dijkstra originally conceived the algorithm in his 1959 paper "A note on two problems in connexion with graphs" [2]. The following description is based on an "Introduction to Algorithms"[17], 24.3.

Dijkstra's algorithm solves the single-source shortest path problem for a weighted directed graph G. i.e. Given a graph $G = (V, E)$, where V is the vertices and E is the directed weighted edges and a start vertex $s \in V$, find the path where the sum of the weights is the smallest possible.

Let v_π either be a predecessor of null. v_d being the upper bound of the weight of a shortest path from source s to v.

Algorithm 5 Initialize-Single-Source(G,s)

```
1: for each vertex $v \in G.V$ do
2:     $v.d = \infty$
3:     $v.\pi$ = Null
4: $s.d = 0$
```

Relaxing an edge (u, v) consist of testing whether we can improve the shortest path to v found so far, by going through u and, if so, update $v.d$ and $v.\pi$. We define w as following for a path $p = \langle v_0, v_1, ..., v_k \rangle$

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

In the below algorithm Q acts as a min-priority queue to contain all the vertices in V. Naive implementation of Dijkstra's algorithm yields $O((V + E)\log V)$ which is $O(E \cdot \log V)$ if all vertices are reachable from the source. And can be $O(V^2)$ if $E = O(V^2/\log V)$. Extract-min runs in $O(\log V)$

Algorithm 6 Relax(u, v, w)

```
1: if $v.d > u.d + w(u, v)$ then
2:     $v.d = u.d + w(u, v)$
3:     $v.\pi = u$
4: $s.d = 0$
```

Algorithm 7 Dijkstra(G, w, s)

```
1: Initialize-Single-Source$(G, s)$
2: $S = \emptyset$
3: $Q = G.V$
4: while $Q \neq \emptyset$ do
5:     $u$ =Extract-Min(Q)
6:     $S = S \cup \{u\}$
7:     for each vertex $v \in G.Adj[u]$ do
8:         Relax$(u, v, w)$
```

2.3 Experiment

In this section we present the experiments we did on our $O(n^3)$ implementation, both for running time and test of correctness.

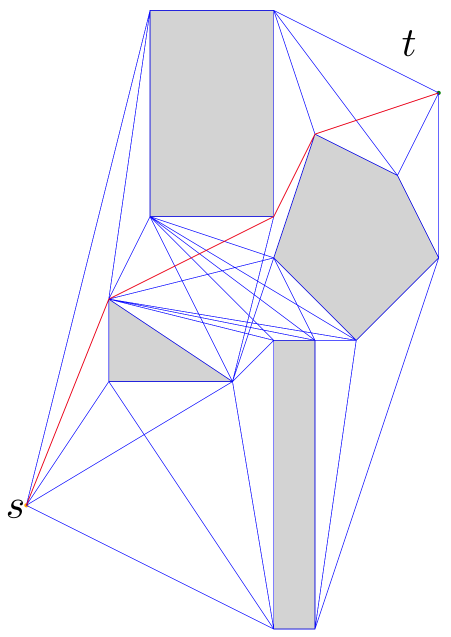
2.3.1 Computer specification

The test were run on a computer with the following specification

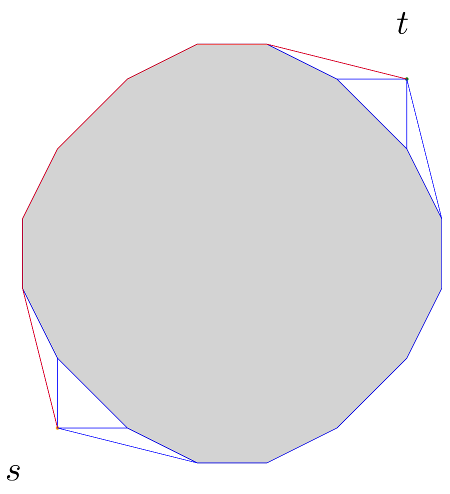
Model	Lenovo ThinkPad, x230
Operating system	Arch Linux
CPU	Intel(R) Core(TM) i7-3520M CPU @ 2.90GHz
Memory	8 GB

2.3.2 Correctness of algorithm

To verify the correctness of our implementation we run the code against a list of tests. We implemented the function to output a svg image of the polygons and route so we were able to confirm the algorithm made the correct visibility graph. (See Figure 2.9a and 2.9b)



(a)



(b)

Figure 2.9: Examples of figures for correctness

2.3.3 Running time of algorithm

To test the running time of our implementation we auto generated a map consisting of x times x squares and put s and t in opposite corners of the map (see Figure 2.10)

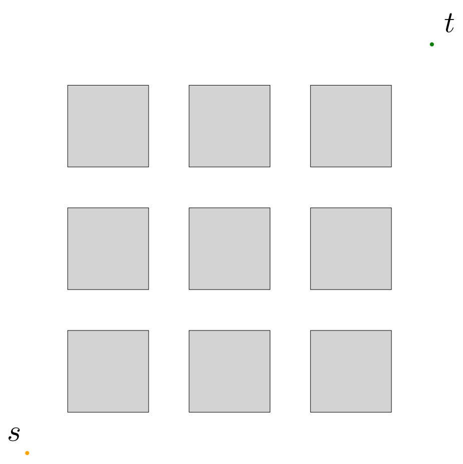
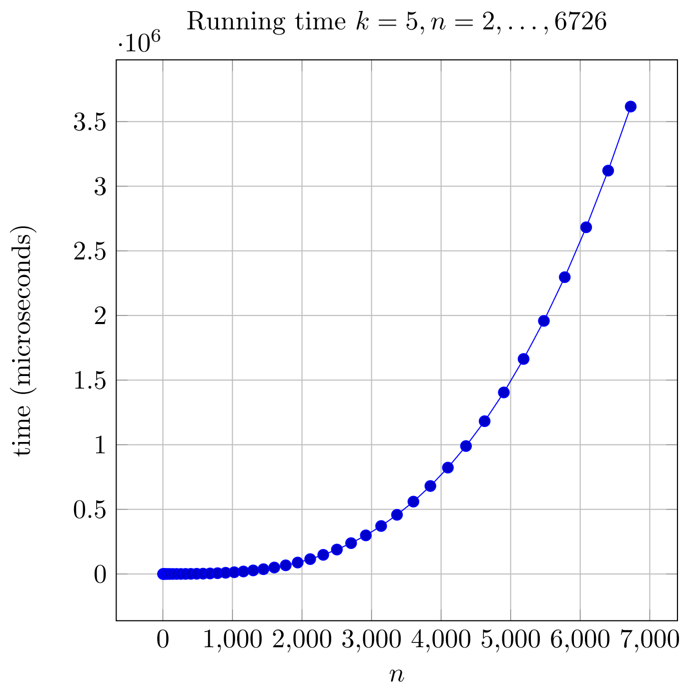
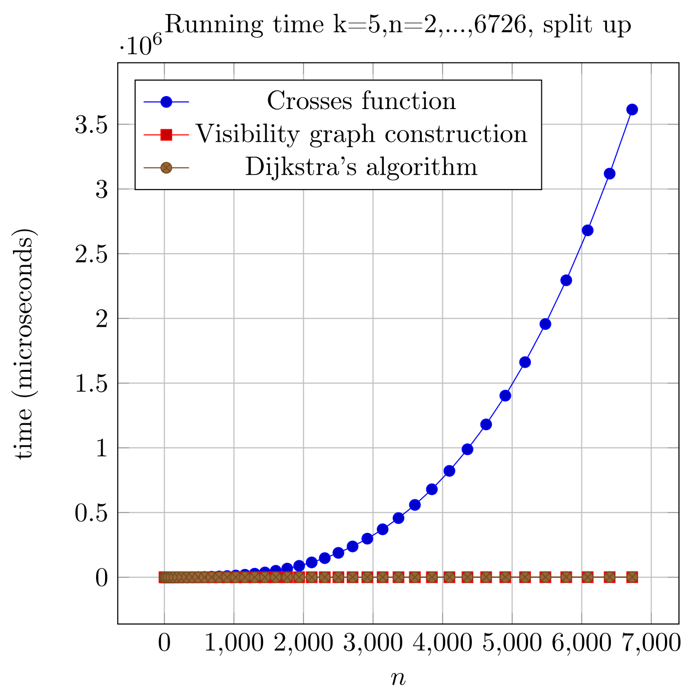


Figure 2.10: Example of how the run time test are constructed. Here is a 3 times 3 example

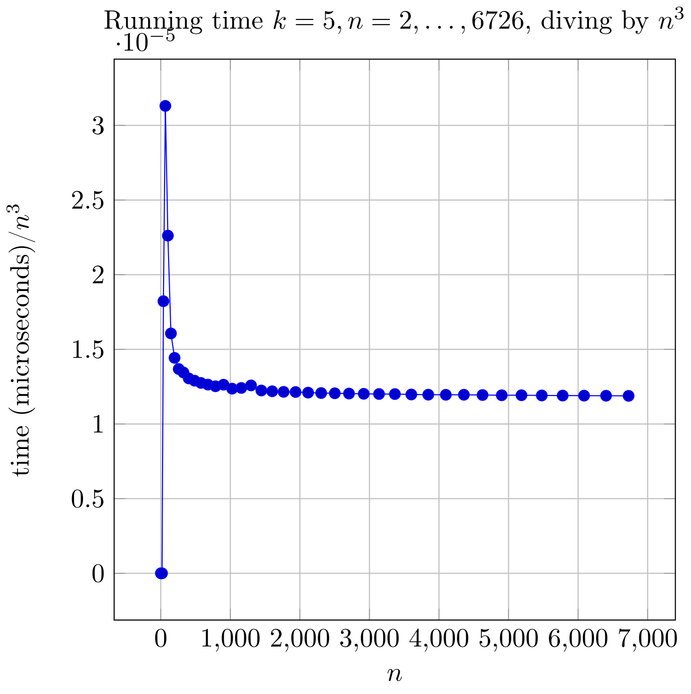
We ran the implementation with the number of violations being constant at 5 and the number of vertices was $n = 4 \cdot t^2$ for $t = 1, \ldots, 41$ and got the following graph



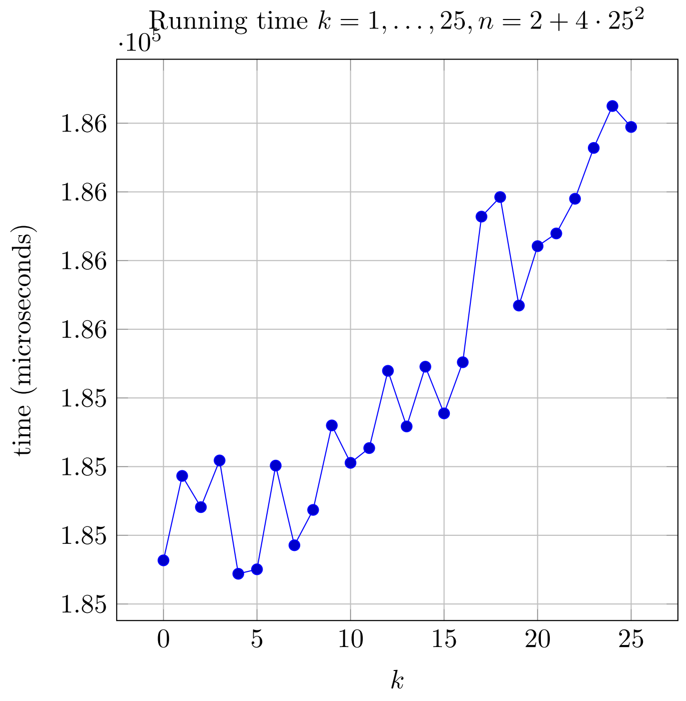
Then we tried to figure out where the time was mostly spent so we tried measuring the crossing function, the construction of visibility graph and Dijkstra's algorithm separately and we obtained the following result:



The implementation is totally dominated by the crossing calculation, which makes sense since the $O(n^3)$ is the most dominant time of the three, we tried dividing the first graph with n^3 and got the following



Lastly we tried to make a test where $n = 25^2$ and $k = 1, \ldots, 25$ (see Figure 2.10) to see if k would influence the runtime. We got an almost horizontal graph, meaning that given a bigger k does not seem to influence the overall run time that much, which is what we would expect.



In conclusion we have implemented a naive $O(n^3)$ algorithm and made tests both for confirming its correctness and its running time.

Chapter 3

Continuous Dijkstra - overview of $O(k^2 n \log n)$ algorithm

The following chapter is dedicated to giving the reader an overview of an $O(k^2 n \log n)$ time algorithm for solving the shortest problem path with polygonal obstacle violation. First, we present an overview of the Hershberger-Suri algorithm, which solves the shortest path with no violation in an optimal $O(n \log n)$ time [6]. This algorithm produces a shortest path map, which divides the free space (free space being the plane minus the interior of the obstacles) into regions where each region will have the same shortest sub-path from s to all points in that region. This structure can then be extended for the purpose of calculating shortest paths with violations, which we will present an intuitive idea on how this is done. Finally, we present an algorithm in chapter 7 which combines the Hershberger-Suri algorithm with a modified version of the same algorithm, to produce a shortest k-path map, a subdivision which has the shortest sub-path to all points in an area with k violations, in time $O(k^2 n \log n)$. This result is due to Hershberger, Kumar and Suri [5].

3.1 The Hershberger-Suri algorithm overview

The Hershberger-Suri algorithm is an algorithm for computing a shortest path map, a data structure from which we can query the shortest path in a plane in the presence of obstacles without violations. The Hershberger-Suri algorithm was proven to be optimal-time in [6], with its $O(n \log n)$ running time, where n is the total number of vertices of the obstacles located in the plane. This is done by computing a *shortest path map* which is a map containing the shortest paths from a fixed source point s, to all other points in the plane. This is done by subdividing the plane into a finite number of regions, where all the points in such a region have the same shortest sub-path from s. This map can be constructed in $O(n \log n)$ time and requires $O(n \log n)$ space [6]. A query for a shortest path can be processed in $O(\log n)$ time due to [8].

The first step in the Hershberger-Suri algorithm is to use an implementation of the continuous Dijkstra method, which purpose is to give a distance from the source s to each vertex and edge in the plane. The continuous Dijkstra method is a theoretical tool to simulate a propagation of a unit speed wavefront in a free space, where s sends out the first emission, which propagates through the plane and collides with obstacles. Upon collision between a wavefront and a vertex in an obstacle, then this vertex will also start emitting a wavefront of its own, with its weight equal to the time it took from s started to emit its wavefront until a contact (with any wavefront), see Figure 3.1.

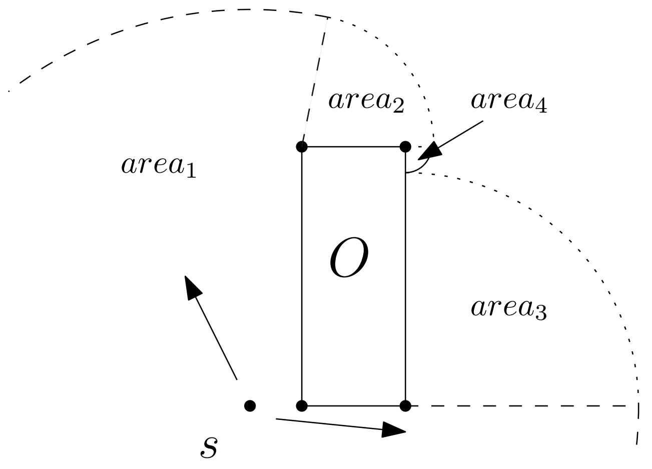


Figure 3.1: A simple example of the wavefront propagation from s and hitting vertices on obstacle O which starts new wavefronts, from these vertices.

Hershberger and Suri introduced two new ideas to speed up the implementation of the continuous Dijkstra method compared to previous attempts, the first being a quad-tree-like subdivision of the plane, which we will introduce in chapter 5 as a conforming subdivision, and the second being an approximate wavefront, which introduces a bit of slack in the calculation of the collision-time of the wavefronts, which we will discuss in chapter 6.

The first idea is grounded in the observation that a wavefront of the type in the continuous Dijkstra method, will be quite complicated to implement directly. A subdivision of the plane into well-behaved regions will be a way around this. This subdivision is constructed in such a way that it aids the propagation of the wavefronts. This is done by temporarily ignoring the line segments(edges) between the vertices in the obstacles, and subdividing the plane into a grid-like subdivision of size $O(n)$ around the vertices. Each cell in this subdivision, (the conforming subdivision) will only have a constant number of straight line edges, and will contain at most one obstacle vertex. This construction means that the subdivision satisfies the following crucial property: for any edge e of the subdivision, there are $O(1)$ cells within distance $2|e|$ of e, which will be crucial in bounding the overall complexity of the subdivision.

The obstacle line segments will then be inserted into the subdivision, while maintaining both the linear size of the subdivision and its conforming property, except now a non-obstacle edge e will have the property of having $O(1)$ cells within shortest path distance $2|e|$ of the edge.

These cells will then form the basis for the *unitspeed* propagation in the algorithm, which will act as the wavefront propagation in the algorithm. This means that at each step of the wavefront will propagated through one cell at a time. Since the descriptive complexity of each cell will be constant, the algorithm will perform efficiently in the propagation through each of the cells.

Inside each of these cells there will be two types of event: the first being a collision between a wavefront and an obstacle, which is quite easy to handle. The second type of event will be a collisions between two wavefronts which will be more complex problem to handle. There will be two different types of collisions between two wavefronts, the first being the collisions where the wavelets are neighbors in the wavefront, and the second being collisions between non-neighboring wavelets. Here two wavefronts are waves emitted from two different sources, and two wavelets being two parts of the wave arch emitted from one source. The case of colliding neighboring wavelets occurs when a wavelet would be engulfed by the expansion of wavelets of its two neighbors and should be quite easy to detect and process. The collision between non-neighboring wavelets, however are more troublesome, and to process these we make use of the second idea: *approximate wavefront*.

The idea of approximating wavefronts is the abandonment of trying to compute the exact time of collision and instead maintaining two separate wavefronts approaching the edge from opposite sides. Each of these wavefronts is an *approximate wavefront*, representing the wavefront that hits the edge from only one side. This leads to the other wavefronts which arrives at the edge after the first wavefront, wont be recorded by the edge, due to it being slower than the initial arriving wavefront.

As mentioned above, the Hershberger-Suri algorithm make use of timers to estimate the distance between two points in the plane but also to estimate when each edge in the subdivision would be engulfed by the wavefronts. A critical task of these timers is to ensure that the collision between two wavefronts which are used in the construction of the shortest path map, is measured in a small proximity of their actual collision, and therefore location.

At the end of the propagation phase, all the collision information is collected, and then a Voronoi diagram like technique is used in each cell to compute the collision events in that cell exactly. These collisions determines the edges of the final shortest path map, which will give us the shortest path to every point in the plane.

3.2 From no violations to k-violations

Previously, we have given an overview of the Hershberger-Suri algorithm which calculates a shortest path map (SPM) in $O(n \log n)$ time from a source point s [6]. By modifying the algorithm, and using it as a subroutine, Hershberger, Kumar and Suri showed an algorithm for calculating the shortest path map, where every route would violated at most k obstacles from s to an endpoint t [5]. This is done by calculating a shortest k-path map, which in essence would produce a subdivision of the plane into regions, as done by the Hershberger-Suri algorithm, but with the guarantee of every path violating at most k obstacles. A way to better understand how such a map would be calculated is by using the metaphor of a parking garage. Here every obstacle would be seen as an elevator from one floor i to the next floor $i + 1$. This would imply one could only take at most k elevator trips when taking the path from s to t. When thinking about the problem in this way, it seems quite natural to think of the construction of SPM_k iteratively, starting by making the SPM_0 map, which is done by the Hershberger-Suri algorithm, and then from this construct the SPM_1 map, and for each iteration going one floor up, until we reach SPM_k.

3.3 Construction of a shortest k-path map

The $O(k^2 n \log n)$ implementation of shortest path with obstacle violation, makes use both of an unchanged and a modified version of the Hershberger-Suri algorithm. The unchanged version is used to prepare the SPM_0 map, which the modified version then uses to iteratively calculate the SPM_k map. The difference is due to the unchanged algorithm starting the wave propagation from a source point s which then propagates through the plane.

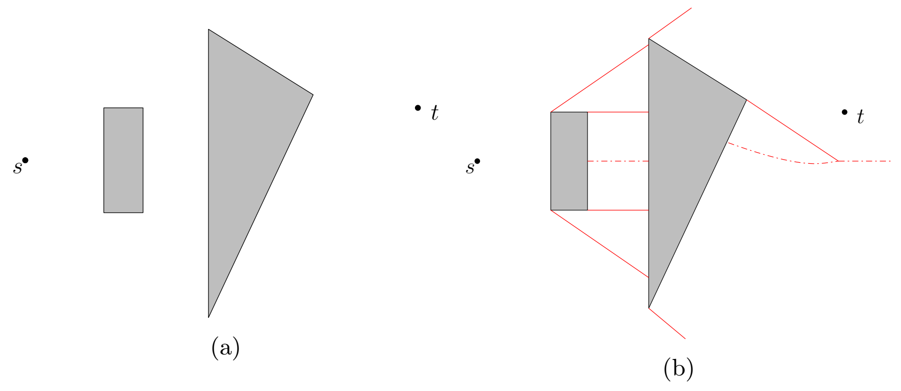


Figure 3.2: The left figure show a plane with source s and target t with two obstacles while the right a plane with the regions of SPM_0 drawn. The dash dotted line is the edge of two areas where there is two shortest paths of equal length.

Above we see in Figure 3.2a a plane with s and t and two obstacles. Figure 3.2b shows a drawing of the SPM_0, where each encapsulated area is a region of the SPM, which will be the output of the unmodified Hershberger-Suri algorithm.

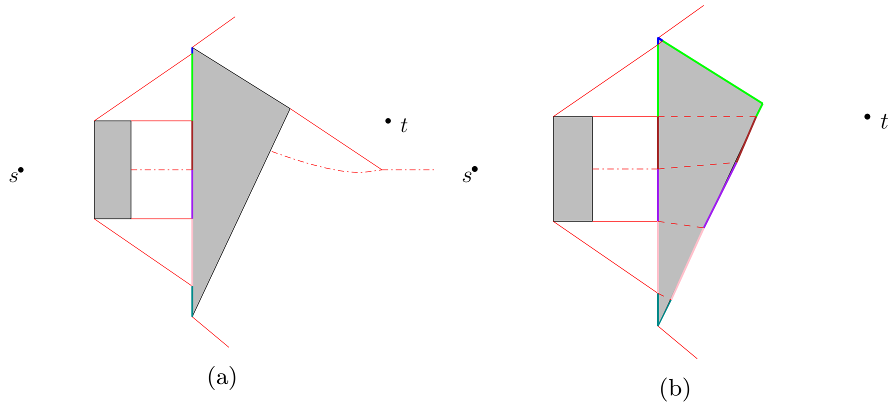


Figure 3.3: The left figure shows the preparation of the modified Hershberger-Suri, which will propagate through each color on the left most edge of the triangular obstacle, and the right the modified Hershberger-Suri algorithm propagates through the triangle to the opposite side, where the propagation into the free space will happen for each colored sub-edges as sources.

The modified algorithm needs to be able to propagate not from a source which is a point but from a sub-edge which can be seen in Figure 3.3a. Each of the colors represent a "sub-edge-source" from which wavefronts will propagate the obstacle. When the obstacles interior has been propagated, the modified algorithm is ready to propagate the free space towards t, which we see in Figure 3.3b. It is worth noting that the free area the modified algorithm needs to propagate in Figure 3.3b, is enclosed by the the two red line which are collinear with s and the top and bottom point of the triangle, with an angle of less than 180 degree. This is due to the area on the other side of the wedge could be reached faster by just going directly from s without violating an obstacle.

This process is then repeated when constructed the next level of the SPM_i map. Finally we will have SPM_k which will consist of an area where the path freely can traverse, since it haven't violated more than the allowed obstacles, and a $SPM_{=k}$ which consist of areas, where the path needs to make turns since it can't violate any more obstacles. These two areas are what SPM_k are made of, and from this we can use the map to look up the fastest path from s to t.

Chapter 4

Shortest path maps and their geometric properties

This chapter is dedicated to presenting some formal definition that will help us precisely discuss the theory in the rest of the thesis. These definition are very much inspired or borrowed from [6] section 3 and [5] section 2 and 3. We will also define the shortest path map and its near relatives the shortest k-path map, and some properties their properties. Finally we will present a Lemma which bounds some complexity of the shortest path map which will be usefull later in Chapter 6.

4.1 Definitions for shortest paths and shortest k-paths

We start with a trivial definition which we will expand upon.

Definition 2. ***Path:*** *Given a plane encapsulated by a polygon $\mathcal{P}$, let s and t be two points in the plane. We define a path between s and t to be a set of vertices and edges which forms a connection from s to t.*

Its trivial to see that a path with a minimum length in the case where s and t are the only entities in the plane will be $\overline{st}$. But we can imagine the space being occupied not only by two points s and t, but also with a set of polygons in which a path cannot pass through. We call such polygons obstacles and we assume through out the rest of this thesis that these obstacles will be simple polygons. Since we will represent these with graphs we give a definition of simples graphs.

Definition 3. ***Simple Graph:*** *A graph is simple if it has no loops and no two of its links join the same pair of vertices.*

Now we might be in a situation where the path with minimum length between two point s and t isn't just $\overline{st}$ due to obstacles being placed in the way. We further define what space we can create our path in, and which we cannot.

Definition 4. ***Free space:***
Let $\mathcal{O} = \{O_1, O_2, ..., O_k\}$ be a family of simple polygons which will act as obstacles in the interior of an encapsulating polygon $\mathcal{P}$. We define the free space to be $\mathcal{FS} = \mathcal{P} \setminus \mathcal{O}$, that is the plane of the encapsulating polygon minus the interiors of all obstacle polygons.

The final $O(k^2 \cdot n \log n)$ algorithm that we will be examining will need the obstacles to be convex, which are defined as follows:

Definition 5. ***Convex Polygons:***
A convex polygon is a simple polygon (not self-intersecting) in which no line segment between two points on the boundary ever goes outside the polygon. In a convex polygon, all interior angles are less than or equal to 180 degrees, while in a strictly convex polygon all interior angles are strictly less than 180 degrees. [10]

We are only allowed to make paths between s and t in the free space, which we will denote as legal paths.

Definition 6. ***Legal Path:***
We define a path between two vertices to be legal if it lie entirely in the free space. That is a legal path is disjoint from the interiors of all potential obstacle polygons in the plane.

Now we are ready to define out legal shortest path between our two points s and t

Definition 7. ***Euclidian shortest path:***
The legal path of minimum total length connecting the two endpoints is a shortest path.

We define a path which is not legal to be violating, or having a number of violations, equal to the number of obstacles in which the path will pass through. We will not only deal with shortest paths which have no violations, but also shortest path which allow up to k violations.

Definition 8. ***Shortest k-path*** *The path of minimum total length which violates at most k obstacles.*

Through out this thesis we will use use the notation of $\pi(s,t)$ to denote the shortest paths connecting two points s and t in the case where no obstacle violation is allowed. The length of any path in $\pi(s,t)$ is the shortest path distance between s and t, denoted $d(p,q)$. If the shortest path between s and t is the line segment $\overline{pq}$, then p and q are said to be visible.

Definition 9. ***Visibility between points:*** *We define two points s and t to be visible to each other if s and t are connectible with the path $\overline{st}$ which is either legal, or in the case of violations have less that the allowed k violations.*

The notion of shortest path distance between two sets of points X and Y is denoted as $d(X,Y)$ and is the minimum $d(x,y)$ over all pairs of points $x \in X$ and $y \in Y$. [6]

We call a path violating of at most k obstacles a *k-path*, generalizing on the traditional obstacle-free path, which is a 0-path. We use the notation of $\pi_k(p)$ to be the shortest path in this case where the path can pass through up to k obstacles from a *fixed source* s to the point p. When reasoning about a path with exactly k crossing we denote this as an $(=k)$-path. And equally we use $d_k(p)$ to denote the length of the shortest path from the fixed source s in the case of up to k obstacle violations. The reason we use a separate notation in the case of obstacle violation is that shortest 0-path problem with origination at a common source point s cannot intersect, by the triangle inequality[5]

Definition 10. ***Triangle Inequality:***
Let A and B be points in a $\mathbb{R}^n$ space and let $|AB|$ denote the distance between A and B. Then the triangle inequality states that for three points $A, B, C \in \mathbb{R}^n$ [7]

$$|AB| \leq |AC| + |BC|$$

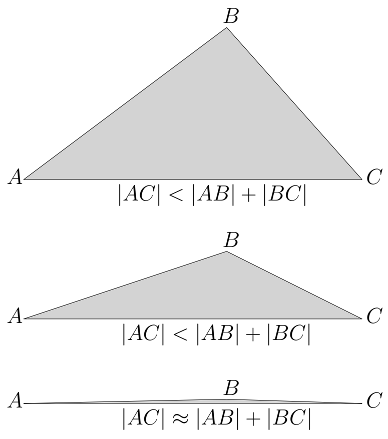


Figure 4.1: Triangular Inequality approaching equality

4.2 Shortest path map and shortest k-path map

This section will briefly introduce the reader to the concept of shortest path map and shortest k-path map and som basic properties of these.

We begin this section with a definition of a predecessor which is essential to understand what a shortest path map is

Definition 11. ***Predecessor:***
The predecessor of an arbitrary point p is defined as a vertex in the plane which is adjacent to p in $\pi(p, s)$. These vertices also include the source s. A predecessor of p is necessarily visible from p. If p and s are mutually visible, then s is a predecessor of p. [6]

Next we give the definition of a shortest path map

Definition 12. ***Shortest Path Map:***
The shortest path map of a particular source point s, denoted $SPM(s)$, is a subdivision of the plane into two-dimensional regions such that all the points in one region have the same, unique predecessor[6].

An example of a construction of SPM can be seen in Figure 4.2 blow

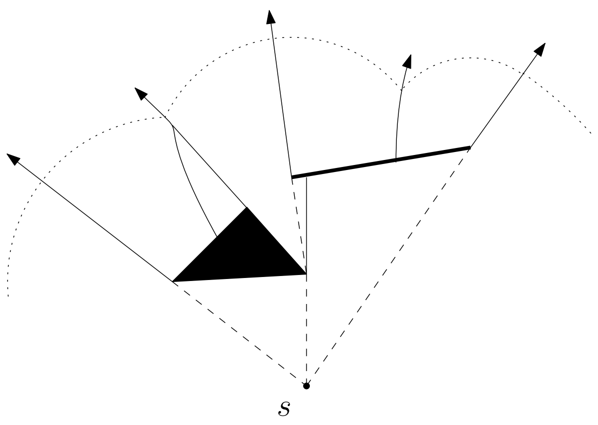


Figure 4.2: An example of an SPM build around s with where the area between the arrows and fully drawn lines shows the regions in the plane with the same predecessors[6].

The figures shows the SPM build around s as the source, and the different obstacles in the plane. Since a shortest path only needs to turn at the vertices of the obstacles, by triangular inequality, the vertices of obstacles naturally constitute the unique predecessors of the points within the different regions marked by the fully drawn lines. Here the lines will extend until the meet the encapsulating polygon $\mathcal{P}$ an then be fully enclosed. The dashed lines show the shortest path from a region s or to the preceding region. It should be noted that these fully drawn line constitutes bisectors (which will explained in later chapter) and there point on the line have a equal distance to s through either of the regions which the bisector acts a border between. It should therefore be noted that in the case of the SPM there may multiple shortest paths to points in the plane, in which case one just chooses one of them.

The distance to a point p in the SPM is calculated by finding the weight from s to p

Definition 13. ***Weight:***
We define weight of an vertex (including obstacle vertices) to be its shortest path distance to the source s. Given an arbitrary point p in free space, its weighted distance to a visible vertex v is defined as

$$d(s, v) + |\overline{vp}|$$

that is the straight-line distance from v to p plus the shortest path distance from s to v. [6]

Next we move on to the definition of k-predecessors and the shortest k-path map

Definition 14. ***k-predecessor***
Given a shortest k-path $\pi_k(p)$, we define the predecessor of p to be the vertex (including s) that is adjacent to p in $\pi_k(p)$[5].

Definition 15. ***Shortest k-path map****(Definition 9 in [5])*
The partition of free space into connected regions with the same k-predecessor is

called the shortest k-path map, and is denoted by SPM_k. The subset of SPM_k for which the shortest path $\pi_k(p)$ to every point p has exactly k crossings is called the shortest $(=k)$-path map and denoted by $SPM_{=k}$.

It is quite easy to see that a SPM is the same as an $SPM_{=0}$, we will therefore use SPM when dealing with the Hershberger-Suri algorithm in chapters 5 and 6, and $SPM_{=0}$ when dealing with the computing of SPM_k in chapter 7.

We saw with the SPM_0 map that each predecessor to an region always where on the boundary on said region, this isn't necessarily the case for a SPM_k. Further more multiple regions in SPM_k may have the same predecessor see Figure 4.3.

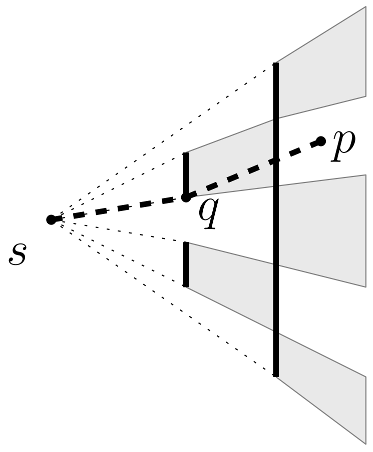


Figure 4.3: Here we present a SPM_1 map, where we the fat lines are obstacles.

Here we see that the shortest path from s to p goes through point q which lies outside the region in which p is. So we need to maintain additional information with polygon vertices to disambiguate the predecessor relation. So suppose we have a line segment $\overline{vp}$ between to vertices which crosses $(k-1)$ obstacles for some $0 \leq i \leq k$, then the length $d_k(p)$ of $\pi_k(p)$, is defined as the sum of the length of the i-path to v and the length of segment $\overline{vp}$. So in the context of Figure 4.3 we have $\overline{qp}$ crosses $(k-1)$ obstacle in the relation of $0 \leq 1 \leq k = 1$, which leaves the $k-1 = 0$ obstacles left which we can cross. The 0-path to q is the direct path $\overline{sq}$, where we have the total shortest path.

So for a point p in $SPM_{=k}$, we identify the k-predecessor of p by the pair (v, i), where v is a vertex of $\mathcal{P}$ and $i \in \{0, 1, .., k\}$ such that $d_k(o) = d_i(v) + |\overline{vp}|$ and the segment $\overline{vp}$ crosses $(k-i)$ obstacles [5].

Neat property of the SPM_k is we devide it into two parts, a V_{k-1} path which is the region consisting of $k-1$-visible points, which is star-shaped (Definition 16), and the $SPM_{=k}$ part. The concept of V areas can be seen in Figure 4.4.

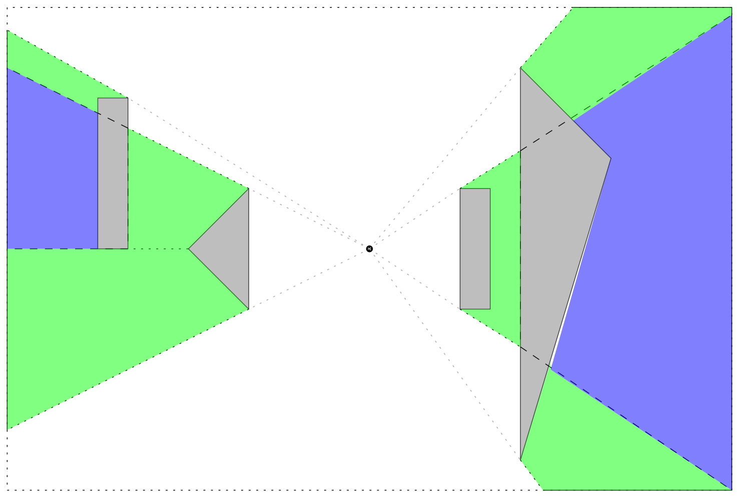


Figure 4.4: Here the boundary of V_1 is marked with dashed lines, while the region of V_0 is shown with dotted lines. V_1 is further shown with blue and the $V_1 \setminus V_0$ is shown with green.

4.3 Complexity of SPM map

Here follows a Lemma which will be usefull for bounding the number of hyperbolic arcs in the proof of Lemma 46 in Chapter 6 but first we define what a star-shaped polygon is.

Definition 16 (Star-shaped polygon)**.** *[15] A simple polygon P is star-shaped if there exists a point z not external to $\mathcal{P}$ such that for all points p of $\mathcal{P}$ the line segment $\overline{zp}$ lies entirely within P. The locus of the points z having the above property is the* ***kernel*** *of $\mathcal{P}$.*

Lemma 17 (Lemma 3.2)**.** *The shortest path map $SPM(s)$ has $O(n)$ vertices, edges and faces. Each edge is a segment of a line or a hyperbola*

Proof. Note that each face $SPM(s)$ is star-shaped (see Definition 16) with the unique predecessor vertex for the face, and the predecessor is in the kernel of the face. The idea behind this proof is to show that each obstacle vertex is a predecessor vertex for at most one face in $SPM(s)$. Consider a vertex u that is the predecessor of a face F and let $pred(u)$ be the set of predecessors of u, this is a set because there can be multiple predecessors. Observe that $d(s, u) = d(s, v) + |\overline{uv}|$ for any $v \in pred(u)$, since the distance $d(s, u)$ can always be rewritten as the distance to from s to u's predecessor, and a straight line from the predecessor to u since your predecessor is always visible from a point.

If a point p is visible from a vertex $v \in pred(u)$ with v, u, p not being collinear, then p cannot have u as its predecessor. This is due to the triangle inequality, where it is always shorter to take the direct line instead going by another point.

Consider the subset of the free space that is visible from u but not visible from $v \in pred(u)$. Let $R(u, v)$ denote the component of this subset that is incident to u. Then $R(u, v)$ lies in an angular angle around u of less than 180°. Define

$$R(u) = \bigcap_{v \in pred(u)} R(u, v) \tag{4.1}$$

Clearly $F \subseteq R(u)$, since the area $R(u)$ is the area that is incident to u, and not visible from any predecessor $v \in pred(u)$

Then the claim is that there is at most one face of $SPM(s)$ in $R(u)$ with u as its predecessor.

We do this by contradiction: Suppose there were two faces F_1 and F_2 both having u as their unique predecessor. The faces F_1 and F_2 must have exactly one point in common, the vertex u. In the space between F_1 and F_2 there is a point p, there have to be a point here, otherwise they would be the same face. The point p is arbitrarily close to u with predecessor z such that z is distinct from both u and $pred(u)$. In other words $d(s, u) + |\overline{up}| > d(s, z) + |\overline{zp}|$. However as p moves towards u the difference in the distance shrinks and finally $d(s, u) = d(s, z) + |\overline{zu}|$. But then z must be a predecessor of u. This means that F_1 and F_2 is part of the same face, contradicting the hypothesis. Thus a vertex u is a predecessor of at most one face in the shortest path map.

Finally to prove the linear upper bound on the size of the shortest path map, recall that the number of obstacle vertices is n the remaining vertices border at least three faces of $SPM(s)$ (for this argument we count the obstacle polygons as faces of the shortest path map). Since the number of faces is $O(n)$, Eulers formula for planar graphs implies that the total number of vertices is also $O(n)$. This completes the proof □

Chapter 5

Conforming subdivision

The following chapters results are due to Hershberger and Suri[6] section 2 and 6. We will present the main theory behind a conforming subdivision, and an algorithm for computing it, and The implementation details for an $O(n \log n)$ implementation.

Given a plane with obstacles, we could view this as a plane with holes in it, which we cannot enter. These holes are made of the obstacles occupying the space. So here the notion of free space is the plane minus the interior of the obstacles. This free space is where the unmodified Hershberger-Suri would find a shortest path, that is a shortest path without violations. One of the key ideas for calculating the shortest path map, which we need to find the shortest path, is the notion of a conforming subdivision of the free space. This is a subdivision of the free space into squares, which we will call *cells*, where a cell has a constant descriptive complexity.

This construction is done i two steps: the first step constructs a subdivision while only considering the vertices of the obstacle polygons. The second step will then insert the obstacle edges into the subdivision, which will have a taken a grid-like structure. This structure is build bottom up, such that every vertex in the plane is contained in the interior of a cell. The algorithm then proceeds to simulate *growth-process* which will make the cells grow until then entire plane is covered by these cells. The way this growth is facilitated is by defining a equivalence class of when cells overlap, and can be merged together. This is the reason it's called a conforming subdivision, since the grid grows, and conforms to the vertices in the plane.

When this grid of orthogonal cells has been produced we insert the obstacle edges intro the grid, giving us two types of edges. The edges that we have grown, which we will call *transparent* edges, since our wavefronts will be able to pass through them, and the obstacles edges which we will call *opaque* edges, which the wavefront will be blocked by. The transparent edges will obey the claim that they will be well-covered, which we will define in the next section, but this helps us bind the overall complexity of the subdivision, and secure that there are $O(1)$ of cells within a distance of $2|e|$ for every transparent edge e. It is this well-covering property that the shortest path algorithm relies heavy on, in the unmodified Hershberger-Suri algorithm. This subdivision can be built in $O(n \log n)$ time shown by Lemma 34 [6].

5.1 Defining well covering of regions

A crucial property of the quad-like subdivision is the subdivision being *well-covering* on its internal edges. The following section outlines the different definitions and properties that we mean by well-covering, This section is very much inspired by Hershberger and Suri definition of the same concepts in [6].

We give the following definition for well-covering:

Definition 18. ***Well-covering with parameter*** α***:***
Given a straight line subdivision $\mathcal{S}$[1] *of the plane, an edge* $e \in \mathcal{S}$ *is said to be well-covered with parameter* α *if the following three conditions hold:*

W1. There exists a set of cells $\mathcal{C}(e) \subseteq \mathcal{S}$ *such that e lies in the interior of their union. The union is denoted* $\mathcal{U}(e) = \{c \mid c \in \mathcal{C}(e)\}$.

W2. The total complexity of all the cells in $\mathcal{C}(e)$ *is* $O(\alpha)$.

W3. If f *is an edge on the boundary of the union* $\mathcal{U}(e)$*, then the Euclidean distance between* e *and* f *is at least* $\alpha \cdot max(|e|, |f|)$.

The edge is said to be strongly well-covered if the stronger condition W3' holds:

W3'. If f *is an edge on or outside the boundary of the union* $\mathcal{U}(e)$*, then the Euclidean distance between* e *and* f *is at least* $\alpha \cdot max(|e|, |f|)$.

In either of the two cases, we will say the region $\mathcal{U}(e)$ is the *well-covering region of e.* The Hershberger-Suri algorithm focuses solely on the distance from the boundary of $\mathcal{U}(e)$ to e, which means we only require a the region to be well-covered, and not strongly well-covered. The reason for a definition of strongly well-coveredness is due to the definition being used later for proving correctness of algorithm.

Definition 19. α***-conforming subdivision:***
Let V *denote the set of vertices of the obstacle polygons, plus the source vertex* s*. A subdivision* $\mathcal{S}$ *is called a (strong)* α*-conforming subdivision for* V *if:*

C1. Each cell of $\mathcal{S}$ *contains at most one point of* V *in its closure*[2].

C2. Each edge of $\mathcal{S}$ *is (strongly) well-covered with parameter* α.

C3. The well-covering region of every edge of $\mathcal{S}$ *contains at most one vertex of* V.

The reason for the naming of definition 19 being conforming, is due to condition C1 and C3, will force the cell structure to *conform* around the distribution of the vertices of V. An example of this can be seen in Figure 5.1. Since the Hershberger-Suri constructs a 2-conforming subdivision V, we will the rest of the thesis denote *conforming* to mean 2-conforming, and explicitly state the parameter if it is not 2.

[1]a subdivision composed of straight lines
[2]its interior plus boundary

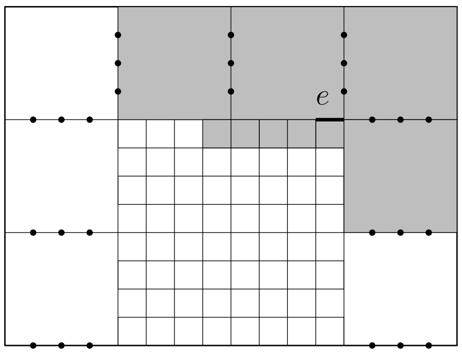


Figure 5.1: An example of part of an strong 1-conforming subdivision. The shaded region in the figure is the union of cell $\mathcal{U}(e)$, a well-covering of edge e [6].

As mentioned earlier in the overview of the Hershberger-Suri algorithm, the subdivision of $\mathcal{S}$ is similar to a quad-tree in that all its edges are horizontal or vertical. However, as we will see, the cells of $\mathcal{S}$ may not always be convex and the subdivision itself can be disconnected. As will shown later, each cell is still reasonably well-behaved, and there are at most one hole per cell. To give a more precise definition, each cell is either a square or a square-annulus.

Definition 20. ***(Square-annulus:)***
We will define it as a square A which is missing a square B internally such that the internal square B is at least $1/4$ the side length of the outer square A

The boundary of these cell may be subdivide into a constant number of edges. We require also that they have the following minimum clearance property:

Definition 21. ***(Minimum clearance property:)***
The minimum width of an annulus in the subdivision (the minimum distance from the inner square to the outer square) is at least one quarter of the side length of the outer square. See Figure 5.2

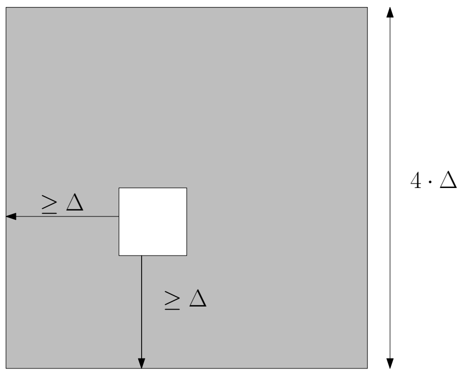


Figure 5.2: A square-annulus, where the distance from the inner square to the outer square, which is Δ, is at least $1/4$ the side length of the outer square which is $4 \cdot \Delta$

Both the annuli and square faces are subject to the uniform edge property:

Definition 22. ***Uniform edge property:***

- *Every edge on the outer square of an annulus has length $1/(4\lceil\alpha\rceil)$ times the side length of the outer square.*
- *Every edge on the inner square has length $1/(4\lceil\alpha\rceil)$ times the side length of the inner square.*
- *The lengths of edges on the boundary of a square cell differ by at most a factor of 4.*

5.2 Conforming subdivision theorem

The conforming subdivision theorem precisely states the properties we expect of the subdivision. It is also why this theorem will be proven by construction of the conforming subdivision algorithm which we will present in section 5.6.

Theorem 23. *(Theorem 2.1 in [6])* ***Conforming Subdivision Theorem:*** *For any $\alpha \geq 1$, every set of n points in the plane admits a strong α-conforming subdivision of $O(\alpha n)$ size satisfying the following additional properties:*

1. *All edges of the subdivision are horizontal or vertical,*
2. *Each face is either a square of a square-annulus, with subdivided boundary,*
3. *Each annulus has the minimum clearance property,*
4. *Each face has the uniform edge property, and*
5. *Every data point is contained in the interior of a square face*

Such a subdivision can be computed in time $O(\alpha n + n \log n)$.

This theorem implies that we need to make modifications to the strong conforming subdivision of V to accommodate for the edges of the obstacles, because our goal is to produce a *conforming subdivision of the free space*. This is done by modifying the edges present in the subdivision, s.t. we differentiate between the edges introduced by the subdivision construction and the edges of obstacles. We mentioned the difference between these before, but for completeness we here present a formal definition of these differences.

Definition 24. ***Transparent and opaque edges:***
Let the edges in a conforming subdivision of the free space, which are introduced by the subdivision, be transparent edges. Equally let the edges in a conforming subdivision of the free space, which are introduced by the obstacles be opaque edges.

The reason, as mentioned before, we the need to differentiate between these, is due to the fact that the algorithm allows wavefronts to pass through transparent edges, but are blocked by the opaque edges. We also require that the transparent edges are well-covered in the conforming subdivision of the free space, even though they don't need to be strongly covered. Due to these requirement, we will slightly alter definition 18 first and third requirement as such:

W1$_{fs}$. Let e be a tranparent edge of $\mathcal{S}$. There exists a set of cells $\mathcal{C}(e) \subseteq \mathcal{S}$ such that e is contained in the closure of the uinion of cells $\mathcal{U}(e) = \{c \mid c \in \mathcal{C}(e)\}$.

W3fs. Let e and f be two transparent edges of s such that f lies on the boundary of the well-covering region $\mathcal{U}(e)$. Then the shortest path distance between e and f is at least $\alpha \cdot \max(|e|, |f|)$.

It is worth noting that condition $W3_{fs}$. ensures e does not touch any transparent boundary edge of $\mathcal{U}(e)$, although it may touch opaque boundary edges.

5.3 Construction of the conforming subdivision

In this section we go through the basic building blocks used for computing the conforming subdivision, i-boxes and i-quads, and some nice properties and behavior of them. We will go through the overlap relation, which is used to make set of equivalence classes of i-quads. These equivalence classes are the main component for the two algorithms which will calculate the conforming subdivision. We will also briefly show a lemma for transforming a 1-conforming subdivision to a α-conforming subdivision, for a constant α. Finally we will discuss the invariants of the of the conforming subdivision algorithms, before moving on to discuss the algorithm in the next section.

5.3.1 Definitions of i-boxes and i-quads

Before going into the algorithm for constructing the conforming subdivision, wee need preliminary terminology and definitions.

To make things easy for ourselves, we fix a Cartesian coordinate system in the plane we are working with. We say for any integer j and l, the i'th-order grid in the coordinate system is the arrangement of all lines $x = j \cdot 2^i$ and $y = l \cdot 2^i$. This makes a grid where each cell (face), is a square of size $2^i \times 2^i$, whose lower-left corner lies at the point $(k \cdot 2^i, l \cdot 2^i)$, for any pair of integers j and l. We will refer to such a cell as an i-box. Any array of size 4×4 is called an i- *quad*, see Figure 5.3.

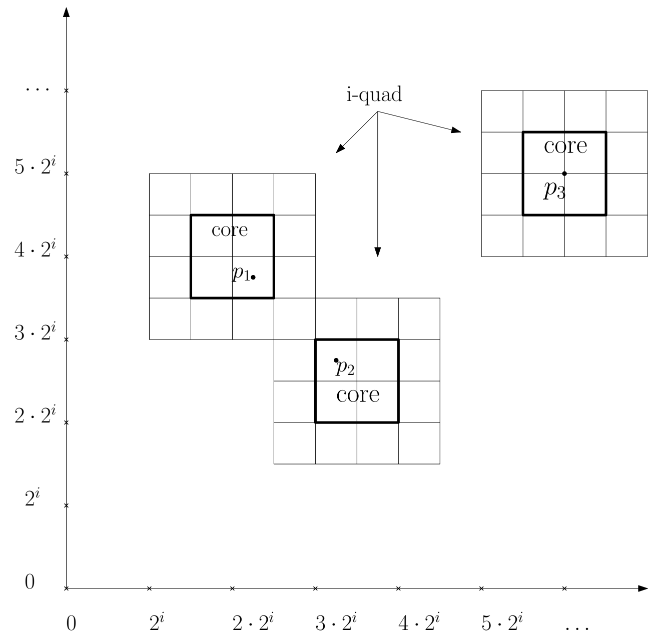


Figure 5.3: An example of how i-quads would be grown around the point p_1, p_2 and p_3. Here we also see that in this particular i'th stage of growth, that p_1 and p_3 belong to the same equivalence class, while p_3 does not.

One could note that, while it is true that the size of an i-quad is the same as an $(i + 2)$-box, an $(i + 2)$-box may not be a cell in the $(i + 2)$-order grid. So these are not always equal. We also refer to the four non-boundary i-boxes of an i-quad as the core of the i-quad, see Figure 5.3. By this definition the core is always an 2×2 array in the i-boxes. One can also observe that an i-box b may have up to four i-quads that contain b in their cores.

The algorithm for building a 1-conforming subdivision, is in a quad-tree-like fashion where we build a partition around the set of points in the plane in a bottom up procedure. This i done by *growing* a square box around each data point, until the entire plane is covered by these boxes. This is done in a number of discrete *stages* numbered $-2, 0, 2, 4, ...$ The end goal is to produce a 1-conforming partition of the points, where the subdivision will be a grid with orthogonal cells. Key idea behind the growth process is, each data point p in stage i is in the core of an i-quad. And since we grow the initial box, the data point p will remain in the i-quads core, and the following lemma holds inductively, by definition of the process we have described.

Lemma 25. *Each* $(i - 2)$*-quad constructed in stage* $(i - 2)$ *lies in the core of some* i*-quad constructed in stage* i*.*

To lower overhead of the algorithm, we only maintain a minimal set of quads at any given i-stage. We denote the set of quads in stage i with $\mathcal{Q}(i)$. This set is partitioned into equivalence classes under the transitive closure of an *overlap* relation.

Definition 26. ***(Overlap relation:)***
Given any two quads q and q', we say these are in the same equivalence class, by the overlap relation, if and only if there is a sequence of quads $q = q_0, q_1, ..., q_m = q' \in \mathcal{Q}(i)$, s.t. q_j and q_{j+1} overlap (have common interior point) for all $j = 0, 1, ..., m-1$. Further more, let $\{S_1(i), ..., S_l(i)\}$ denote the partition of $\mathcal{Q}(i)$ into these equivalence classes in the ith stage, and let $\equiv_i$ denote the transitive equivalence relation.

We denote a region, or to be more exact the partition of the plane, covered by the quads of one class a *component*. By previous definitions we know that a component in stage i either is a single i-quad, of the a union of i-quads where the points in each i-quads core, belong to the same equivalence class. We differentiate between two types of classes. The first being a *simple* component. A component at stage i is simple if

1. Its outer boundary is an i-quad.
2. It contains exactly one $(i-2)$-quad of $\mathcal{Q}(i-2)$ in its interior.

The second type is a complex component. A complex component is complex if it is not simple.

5.3.2 Merging of i-quad

In this subsection we show some distance properties that is satisfied by points of the same equivalence class at stage i, which will be useful in the final algorithm. We say that a quad q is a *containing i-quad* of a point[3] $u \in V$ if $q \in \mathcal{Q}(i)$ and u lies in q's core. We also say that a point u *belongs* to an equivalence class $S \in \mathcal{Q}(i)$ if there is a containing i-quad of u in S.

Lemma 27. *(Lemma 6.6 in [6])*
Let u be a point of V and let $q \in \mathcal{Q}(i)$ be a containing i-quad of u. Then the minimum distance between u and the outer boundary of q is 2^i.

Proof. The key idea is the property that u lies in the core of q, which we know the size of. Since q has side length 2^{i+2}, and u lies at least a quarter of this distance away from the outer boundary, the lemma trivially follows. $\square$

As used earlier in the thesis, we use the notation $d(u, v)$ to denote the distance between the points u and v.

Lemma 28. *(Lemma 6.7 in [6])*
Let u and v be two points of in the plane that belong to two different equivalence classes of $\mathcal{Q}(i)$. Then $d(u, v) > 2 \cdot 2^i$.

[3]its worth noting that in this section and generally in this context of conforming subdivision, that points and vertices are equivalent

Proof. Let q_u and q_v be two containing i-quads for u and v, respectively. Since u and v lie in different equivalence classes, these i-quads cannot intersect. By Lemma 27, each of the points lies at least a distance 2^i away from the outer boundaries of their i-quads, which immediately gives a lower bound of $d(u, v) > 2 \times 2^i$, which proofs the lemma. □

Lemma 29. *(Lemma 6.8 in [6])*
Let u and v be two points in the plane and let q_u and q_v, respectively, be the two i-quads of $\mathcal{Q}(i)$ containing them. If $q_u \cap q_v \neq \emptyset$, then $d(u, v) < 6 \cdot 2^i$.

Proof. By Lemma 27, the maximum distance between u and the outer boundary of q_u is at most $3 \cdot 2^i$. The same holds for v and q_v, which implies the upper boundary of $d(u, v) < 6 \cdot 2^i$. See Figure 5.4. □

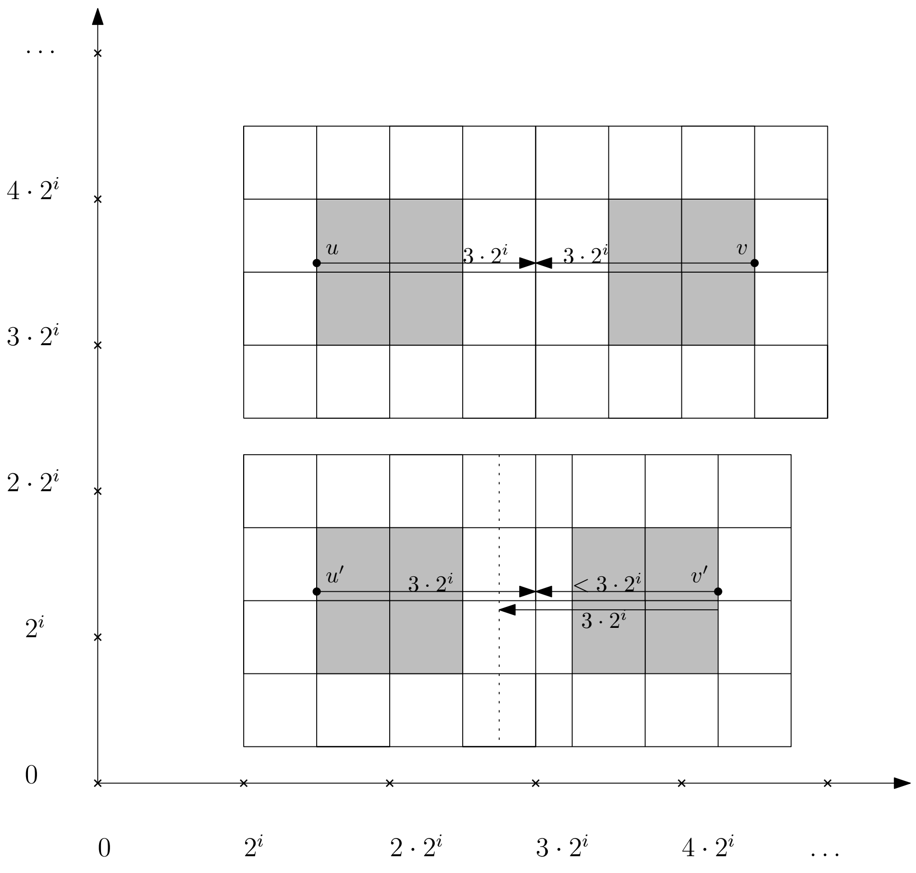


Figure 5.4: The two top i-quads with points u and v are as close as they can be without belonging to the same equivalence class, that is not overlapping, and therefore have $d(u, v) = 6 \cdot 2^i$. The two lower i-quads with points u' and v' overlap, and therefore have $d(u, v) < 6 \cdot 2^i$.

5.3.3 Transforming 1-conforming subdivision to α-conforming subdivision

The following lemma shows how to transform a 1-conforming subdivision into an α-conforming subdivision of size $O(\alpha \cdot n$ in $O(\alpha \cdot n)$ time. This is quite important for the correctness of our algorithm, since we will need the ability to transform the 1-conforming subdivision to an α-conforming subdivision.

Lemma 30. *(Lemma 6.1 from [6])*
Let V be a set of n points, and let $\mathcal{S}_1$ be a 1-conforming subdivision for V of size $O(n)$. For any $\alpha > 1$, we can build an α-conforming subdivision $\mathcal{S}_\alpha$ for V with complexity

Proof. Subdivide each edge of $\mathcal{S}_1$ into $\lceil \alpha \rceil$ equal-length pieces. Define the well-covering region of each edge e in $\mathcal{S}_\alpha$ to be the same as the well-covering region in $\mathcal{S}_1$ of which e is a fragment. These operations can performed in $O(\alpha \cdot n)$ time.

We show below that the subdivision thus defined satisfies properties from definition 19.

1. $\mathcal{S}_\alpha$ has the same set of cells as $\mathcal{S}_1$, so each cell of $\mathcal{S}_\alpha$ contains at most one point of V in its closure.
2. Each internal edge e_α of $\mathcal{S}_\alpha$ is well-covered with parameter α, since it satisfies the conditions stated in 18. Let e_1 be the edge of $\mathcal{S}_1$ of which e_α is a fragment. Let $C_\alpha(e_\alpha)$ be the set of cells of $\mathcal{S}_\alpha$ whose union $\mathcal{U}_\alpha(e_\alpha)$ is the well- covering region of e_α. Define $\mathcal{C}_1(e_1)$ and $\mathcal{U}(e_1)$ analogously.
   a) $\mathcal{U}_\alpha(e_\alpha)$ covers the same area as $\mathcal{U}_1(e_1)$, so e_α is contained in its interior.
   b) Each edge of each cell in $\mathcal{C}_1(e_1)$ is divided in $\lceil \alpha \rceil$ pieces in $\mathcal{C}_\alpha(e_\alpha)$ is $O(\alpha)$.
   c) Let f_α be an edge of $\mathcal{S}_\alpha$ on (or outside in the case of strongly 1-conforming) the boundary of $\mathcal{U}_\alpha(e_\alpha)$, and let f_1 be the edge of $\mathcal{S}_1$ from which it is derived. The Euclidean distance between e_α and f_α is at least as large as the distance between e_1 and f_1, which is at least $\max(|e_1|, |f_1|) \geq \max(\alpha \cdot |e_\alpha|, \alpha \cdot |f_\alpha|)$.
3. Well-covering regions in $\mathcal{S}_\alpha$ are the same as in $\mathcal{S}_1$, so each contains at most one vertex of V.

Which establishes the lemma. □

5.3.4 The invariants

The main objective of the algorithm is to draw the boundaries of certain components, which in the end will give the correct subdivision with the properties of the conforming subdivision theorem. Each of these edges will be straight line segments, all parallel to one the axes, and will be subdividing the plane into orthogonal cells. The critical property of our subdivision is the following *conforming property:*

Invariant 1: For any edge e and cell c of the subdivision, c has an interior point within distance $|e|$ of e if and only if c and e are incident (their closures intersect). Thus There are at most six cells within distance $|e|$ of any edge e.

The algorithm will only draw edges of increasing lengths, and so we never need to subdivide previously drawn edges inside a component. In order to maintain Invariant 1, the algorithm will also enforce the following auxiliary invariant:

Invariant 2: The boundary of each complex component in stage i is subdivided into edges of length 2^i that are aligned with the ith-order grid [4].

Through the algorithm the outer boundary of simple components, wont be drawn until just before they merge with other components to form complex components. This will show itself very valuable, since this helps to ensure the upper bound of the size for the final subdivision of $O(n)$.

The algorithm consists of two main sub-algorithms. The first procedure ***growth***, will take care of simulating the growth of the $(i-2)$-quads to i-quads at a stage i. The second procedure ***build-subdivision***, will compute and maintain the equivalence classes, and will also draw the subdivision edges (which will satisfy invariant 1 and 2).

First we will present **build-subdivision**, and then move on to presenting **growth**. All we need to know about **growth** for now is, given an i-quad q, the procedure **growth**(q) will produce a $(i+2)$-quad containing q inside its core. For a family S of i-quads, **growth**(S) is a minimal set of $(i+2)$-quads satisfying the following:

$$\forall q \in S, \quad \exists \bar{q} \in \mathbf{growth}(S) \quad \text{s.t.} \quad \bar{q} = \mathbf{growth}(q)$$

As mentioned earlier, up to four $(i+2)$-quads may contain the i-quad q in their cores. So to not complicate matter, we will postpone the discussion of how the procedure **growth** chooses **growth**(q). For now we will be content with **growth**(q) being a unique $(i+2)$-quad returned by the procedure **growth**. We also use the notation $\bar{q}$ to denote **growth**(q).

5.4 Pseudo code for build-subdivision

To assure that all components will be simple and disjoint at the initial state, and not complex (overlap) we will scale the plane in such a way that either the horizontal or the vertical distance between any two points in the plane is at least 1, and no points has a coordinate which is a multiple of 1/4. We compute a (-2)-quad for every point p in the plane, with p in the upper left corner of the (-2)-boxes core. These quads form the initial set of quads in $\mathcal{Q}(-2)$. Since no i-quad overlaps, they all belong to their own equivalence class, and can in this context be regarded as singletons. We proceed to draw the (-2)-box around each point p, which will be contained in the core of the (-2)-box, which we won't draw now. From this initial setup, one can easily

[4]both invariants are as defined in [6] section 6.2

see that the invariants are satisfied, and we are ready to proceed with the **build-subdivision** algorithm:

Algorithm 8 Algorithm **build-subdivision**

```
1: while $|\mathcal{Q}(i)| > 1$ do
2:     $i = i + 2$
3:     Initialize $\mathcal{Q}(i) = \emptyset$
4:     for each equivalence class $S$ of $\mathcal{Q}(i-2)$ do
5:         $\mathcal{Q}(i) = \mathcal{Q}(i) \cup \mathbf{growth}(S)$.
6:     for each pair of $i$-quads $q, q' \in \mathcal{Q}(i-2)$ do
7:         if $q \cap q' \neq \emptyset$ then
8:             Set $q \equiv_i q'$.
9:     Extend $\equiv_i$ to an equivalence relation by transitive closure, and compute
       the equivalence class
10:    for each $q \in \mathcal{Q}(i-2)$ do
11:        Let $\bar{q} = \mathbf{growth}(q)$ as computed in step $2-8$
12:        if $q$ is a simple component of $\mathcal{Q}(i-2)$ but $\bar{q}$ isn't a simple component
   of $\mathcal{Q}(i)(*)$ then
13:            Draw the boundary box of $q$ and subdivide each of its sides into
               four edges at the $(i-2)$-order grid lines.
14:    for each equivalence class $S$ of $\mathcal{Q}(i)$ do
15:        Let $S' = \{q \in \mathcal{Q}(i-2) \quad s.t. \quad \mathbf{growth}(s) \in S\}$.
16:        if $|S| > 1$ then
17:            Let $R_1 = \cup_{q \in S'}\{$the core of $\mathbf{growth}(q)\}$.
18:            Let $R_2 = \cup_{q \in S'}\{$the region covered by $q\}$.
19:            Draw $(i-2)$-boxes to fill the region between the boundaries of
               $R_1$ and $R_2$.
20:            Draw $i$-boxes to fill the region between the boundaries of $R_1$ and
               $S$; break each cell boundary with an endpoint incident to $R_1$ into
               four edges of length $2^{i-2}$, to satisfy Invariant 1.
```

As we explained earlier, the algorithm runs in discrete stages of $-2, 0, 2, 4, ...$, which we see in the increment step of step 2. Step 3 to step 8 computes the $\mathcal{Q}(i)$ from $\mathcal{Q}(i-2)$. This is done by growing the previous squares from $\mathcal{Q}(i-2)$, one equivalence class at a time, and the see if any of then newly grown squares overlap. If this is the case, thye belong to the same equivalence class, and should be marked as such in $\mathcal{Q}(i)$. Next in step 10 to 13 we process the simple components of $\equiv_{i-2}$ that are about the merge with other components. This is done by checking if the square q which was simple before the growth process still would be simple after the growth. If not we draw the boundary box of q before the growth, and subdivide its sides into edges of equal length, each a quarter of the total side length. The last steps 14 to 20 are dedicated to processing the complex components. Here we compute a S' which consists of the $q \in \mathcal{Q}(i-2)$ which will grow into the equivalence class S in $\mathcal{Q}(i)$. We will only process if $|S'| > 1$ which means S would be complex. Here create two set R_1 being the cores of the i-quads in the complex equivalence class S and R_2 being the region covered by the q in S'. Should these not overlap we fill the region between R_1's and R_2's boundaries with $(i-2)$-boxes. In the last step we basically draw the outer boundaries of the *growth*(q) square, to fill the space between the cores, R_1, and the outer boundary of S, with edges that satisfy the invariants.

This is the overall idea behind the **build-subdivision** algorithm. This pseudo code, while not being efficient enough, gives a good understanding of what we want it to do. We will visit this algorithm again in section 5.6, at improve it to an $O(n \log n)$ implementation, with the needed supporting data structures and more.

5.5 Pseudo code for growth

The overall idea behind the algorithm for **growth**(S) is to build a graph on the quads in S.

Algorithm 9 Algorithm **growth**(S)

```
1: Set growth$(S) = \emptyset$
2: for each pair of quads $q_1, q_2 \in S$ do
3:     if $q_1 \cup q_2$ can be contained in a $2 \times 2$ array of $(i+2)$-boxes then
4:         Put an edge between $q_1$ and $q_2$.
5: Compute a maximal matching in the graph computed in Step 1
6: for each edge $(q_1, q_2)$ in the maximal matching do
7:     Choose an $(i+2)$-quad $\bar{q}$ containing $q_1, q_2$ in its core.
8:     Set growth$(q_1)$ = growth$(q_2) = \bar{q}$, and add $\bar{q}$ to growth$(S)$.
9: for each unmatched quad $q \in S$ do
10:    Set growth$(q) = \bar{q}$, where $\bar{q}$ is an $(i+2)$-quad containing $q$ in its core.
11:    Add $\bar{q}$ to growth$(S)$.
```

Initially we set **growth**(S) to be empty. Step 2 to step 4 builds a graph whose nodes are the i-quads of S, with the property that their collective area can be contained in a grid of 2×2 $(i+2)$-boxes. If this is the case we connect the two nodes. In step 5 we compute a maximal matching of the graph.

Definition 31. *Maximal matching*
Given a graph $G = (V, E)$, we define a matching M in G to be the set of pairwise non-adjacent edges; that is, no two edges will have a vertex in common. A maximal matching is then defined as a matching M of G with the property that if any edge not in M is added to M, then M will no longer be a matching. By this definition, we see that a maximal matching M is a superset of all other matchings of G, where further M can't be a subset of the other matching of G. See Figure 5.5 and 5.6[17].

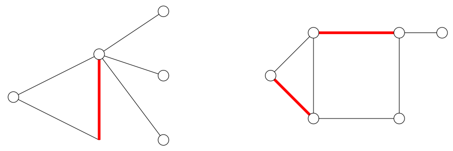


Figure 5.5: The figure shows two examples of non maximal matching, and one maximal matching [19]

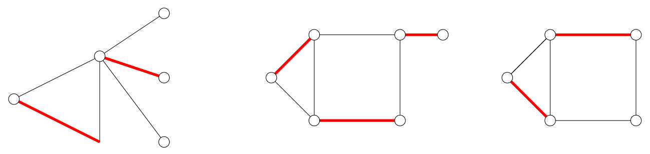


Figure 5.6: The figure shows three examples of maximal matching, one should notice that the last figure have multiple maximum matching, each with two edges[19].

An implementation of a maximal matching algorithm could be the approx-vertex-cover from [17].

The proof of correctness of the **growth** algorithm can be found in appendix C, but it is worth noting that the maximum node degree of the graph build in step 2 to 4, is of constant size, $O(1)$. This is due to the fact that only a constant number of i-quads can touch any i-quad q. This implies the maximal matching in this graph has $\Theta(|E|)$ edges. Since **growth** basically maps an i-quad to its larger counter part in the next stage, each i-quad at stage i maps to an $(i+2)$-quad in stage $(i+2)$. And since each matching edge, that is the edges marked in step 5, corresponds to two i-quads that map to the same $(i+2)$-quad, it follows that:

$$|\mathbf{growth}(S)| = |S| - |\Theta(|E|)$$

Later we will show that $|E|$ is a constant fraction of $|S|$ which leads to $|S|$ gradually becoming smaller and smaller, which is why the algorithm terminates. A formal proof for this fact can be found in appendix C, but we are content with the fact that for each iteration step 5 will compute a maximal matching on gradually smaller and smaller graphs.

Step 5 to 8 constructs a new larger $(i+2)$-quad if two point q_1 and q_2 would be in it's core, and assigns this quad to the equivalence class S. The remaining *unmatched* quad q are just grown individually and added to the equivalence class S.

The fact that any two quads $q, q' \in S$ are contained in the same grown quad $\mathbf{growth}(q) = \mathbf{growth}(q')$ if their closure intersect is one of the main facts why $\mathbf{growth}(S)$ runs in time $O(|S| \log |S|)$, which is the overall running time for **growth**.

5.6 An $O(n \log n)$ implementation for computing a 1-conforming subdivision

The following section presents an $O(n \log n)$ implementation for building a 1-conforming subdivision of the free space. This is done by maintaining the different equivalent classes in $\mathcal{Q}(i)$ for each discrete stage i. This is done by making a Delaunay triangulation of the vertices in the plane. When we have this triangulation we can compute the minimum spanning forest by using Kruskal's algorithm, and connect each tree if the distance between them is close enough to make the equivalence class merge together. This minimum spanning forest is maintained through each growth stage until all trees in the minimum spanning forest have merged into one tree.

5.6.1 Minimum spanning trees

The minimum spanning tree problem is based on the problem of connecting n points, with $n-1$ edges in such a way that the total weight of these edges remain minimal. More formally, given a graph $G = (E, V)$, we can let $w(u, v)$ be a weight function for any two vertices u and v in the graph G which returns the weight of the edge between u and v if such an edge exists, and ∞ if no such edge exists. Then the minimum spanning tree problem is to find a acyclic subset $T \subset E$ that connect all vertices, such that the total weight

$$w(T) = \sum_{(u,v) \in E} w(u, v)$$

is minimized [17]. The minimum spanning tree would then be the solution to this problem.

We further say, if the graph G is made up of multiple components then the minimum spanning tree for each component, will together form a minimum spanning forest of G.

We recall from section 5.3.2 by Lemma 29, if two i-quads q_u and q_v overlap (and therefore at stage i belong to the same equivalence class $S \in \mathcal{Q}(i)$) the distance between the two point u and v, contained in respectively q_u's and q_v's core, has the following property $d(u, v) < 6 \cdot 2^i$. The $O(n \log n)$ implementation of **build-subdivision** is based upon the fact that, given V_S which is the set of points in the core of some equivalence class $S \in \mathcal{Q}(i)$, then the longest edge of a minimum spanning tree of V_S has length less than $6 \cdot 2^i$.

Let V be the set of all vertices in the plane we want to build a conforming subdivision around. We then define $G(i)$ to be the graph on V which contains exactly those edges whose weight is at most $6 \cdot 2^i$, and define $MSF(i)$ to be minimum spanning forest of $G(i)$. Here the forest consist of each minimum spanning from each component $S \in \mathcal{Q}(i)$.

To show the validity of this idea, we briefly present two Lemma for the correctness of this the above assumption. First we show that each point at a stage i only will belong to a single minimum spanning tree.

Lemma 32. *(Lemma 6.9 in [6]) The points contained in any component S of $\mathcal{Q}(i)$ belong to a single tree of $MSF(i)$.*

Proof. Let S be a random component of $\mathcal{Q}(i)$. By Lemma 29, the points contained in S can be linked by a tree with edges shorter than $6 \cdot 2^i$. This implies that any bipartition[5] of the points of V_S, has a minimum weight edge linking the two subsets together which is shorter than $6 \cdot 2^i$. The minimum spanning tree of V_S has all edges shorter than $6 \cdot 2^i$, and therefore V_S belongs to a single tree of $MSF(i)$. □

Next we show that if two i-quads at stage i don't overlap, then their points will belong to different minimum spanning trees in stage $i - 2$.

Lemma 33. *(Lemma 6.10 in [6]) If i-quads q_1 and q_2 belong to different components of $\mathcal{Q}(i)$, then their points belong to different tree of* $\mathbf{MSF}(i-2)$.

[5]bipartition is the grouping of vertices into two groups

Proof. By Lemma 28 we know that every edge from a point in q_1's core to any point outside that core has length greater than $2 \cdot 2^i$. The points of quads q_1 and q_2 components are in the same tree of $MSF(i-2)$ only if every bipartition of V that separates the points of q_1 from those of q_2 is bridged by an edge of length less than $6 \cdot 2^{i-2}$, this is due to lemma 28. But the bipartition separating the points of q_1's component of $\mathcal{Q}(i)$ from the rest of V has bridge length greater than $2 \cdot 2^i$, which is due to lemma 28. Since $2 \cdot 2^i > 6 \cdot 2^{i-2}$. the points of q_1 and q_2 must belong to different trees of $\mathbf{MSF}(i-2)$. □

5.6.2 build-subdivision implementation

The final implementation of the **build-subdivision** procedure is based on an efficient construction of the $MSF(i)$ for all i such that $MSF(i) \neq MSF(i-2)$. One way to go about this is to compute a Delaunay triangulation of V. To understand what a Delaunay triangulation is, we start by understanding what a Voronoi diagram is. The Voronoi diagram is build around points in a plane, where the plane partition into a set of cells, where each cell has exactly one point in its interior. The special property for each of these cells is that each edge in their border is placed between two points, in such a way that the distance from the two points to any point on the edge is the same. The Voronoi diagram can be computed in $O(n \log n)$ time [12].

Delaunay Triangulation can then be understood as the dual graph of the Voronoi diagram. Such a dual graph is build as follow: each vertex in the dual graph corresponds to a cell of the Voronoi diagram. Each pair of vertices in the dual graph is connected with a edge if the vertices corresponds to neighboring cell in the Voronoi diagram. This gives us a triangulation of the all the points in the plane, where no edge overlaps.

The Delaunay triangulation of a plane with points can be done in $O(n \log n)$ time[12]. Then for finding the minimum spanning tree of this triangulation we can run Kruskal's MST algorithm[17].

Kruskal's algorithm will insert the $O(n)$ edges, made in the Delaunay triangulation, into the, at stage i, current minimum spanning forest in sorted order from shortest to longest. Any edge that might join two trees of the forest is retained, and all other edges are dropped.

For each edge e added to the forest, we compute $\mathcal{K} = 2\lceil \frac{1}{2} \log_2(|e|/6) \rceil$, which determines the stage k at which e is added to $MSF(k)$. This can be seen by knowing the inclusion of e in the forest happens when $|e| < 6 \cdot 2^i$. By inserting $2 \cdot 2^i$ into k we see that.

$$\begin{aligned} 2\lceil \frac{1}{2} \log_2(|e|/6) \rceil &= 2\lceil \frac{1}{2} \log_2(6 \cdot 2^i/6) \rceil \\ &= 2\lceil \frac{1}{2} \log_2(2^i) \rceil \\ &= 2\lceil \frac{1}{2} i \rceil \qquad\qquad = i \end{aligned}$$

Which is the stage in which e is added.

By stopping just before each stage change, we produce $MSF(i)$ for each even i such that $MSF(i) \neq MSF(i-2)$ in $O(n \log n)$ total time.

Algorithm 10 Implementation of **build-subdivision**

For each $T \in \mathbf{MSF}(i)$, maintain the corresponding set of i-quads in $\mathcal{Q}(i)$ that are the containing quads for the vertices of T. Call this set $\mathcal{Q}(i, T)$.

Initialize $i = -2$. Initialize $\mathbf{MSF}(-2)$ to be a forest of singleton vertices. For each vertex $v \in V$, $\mathcal{Q}(-2, \{v\})$ is a singleton quad with v in its core.

Maintain a set $\mathcal{N}$ of trees in $\mathbf{MSF}(i)$ such that for each $T \in \mathcal{N}$, $|\mathcal{Q}(i, T)| > 1$; that is, T's component is not a singleton quad. Initialize $\mathcal{N} = \emptyset$.

```
 1: while $|\mathcal{Q}(i)| > 1$ do
 2:     $i_{old} = i$;
 3:     if $|\mathcal{N}| > 0$ then
 4:         $i = i + 2$
 5:     else
 6:         Set $i$ to the smallest even $i' > i$ such that $\mathbf{MSF}(i') \neq \mathbf{MSF}(i)$
 7:     for each edge $e$ of $\mathbf{MSF}(i)$ not in $\mathbf{MSF}(i_{old})$ do
 8:         Let $T_1$ and $T_2$ be the trees linked by $e$.
 9:         for each $T_x \in \{T_1, T_2\}$ do
10:             if $T_x \in \mathcal{N}$ then
11:                 Remove $T_x$ from $\mathcal{N}$.
12:             else
13:                 compute the singleton $(i-2)$-quad in $\mathcal{Q}(i-2, T_x)$.
14:         Join $T_1$ and $T_2$ to get $T'$, and put $T'$ in $\mathcal{N}$.
15:         Set $\mathcal{Q}(i-2, T') = \mathcal{Q}(i-2, T_1) \cup \mathcal{Q}(i-2, T_2)$
16:     for each $T \in \mathcal{N}$ do
17:         Initialize $\mathcal{Q}(i, T) = \emptyset$.
18:         for each equivalence class $S$ of $\mathcal{Q}(i-2, T)$ do
19:             $\mathcal{Q}(i, T) = \mathcal{Q}(i, T) \cup \mathbf{growth}(S)$.
20:         Compute the equivalence classes of $\mathcal{Q}(i, T)$ by plane sweep.
21:         perform Steps 10 through 20 of algorithm 8 on $\mathcal{Q}(i, T)$.
22:         if $|\mathcal{Q}(i, T) = 1$ then
23:             Delete $T$ from $\mathcal{N}$.
```

To give a better overview, we include step 10 through 20 from algorithm 8, see algorithm 8 below.

Algorithm 11 step 10 to 20 from Algorithm 8

```
1: for each $q \in \mathcal{Q}(i-2)$ do
2:     Let $\bar{q} = \mathbf{growth}(q)$
3:     if $q$ is a simple component of $\mathcal{Q}(i-2)$ but $\bar{q}$ isn't a simple component
       of $\mathcal{Q}(i)(*)$ then
4:         Draw the boundary box of $q$ and subdivide each of its sides into four
           edges at the $(i-2)$-order grid lines.
5: for each equivalence class $S$ of $\mathcal{Q}(i)$ do
6:     Let $S' = \{q \in \mathcal{Q}(i-2) \quad s.t. \quad \mathbf{growth}(s) \in S\}$.
7:     if $|S| > 1$ then
8:         Let $R_1 = \cup_{q \in S'}\{$the core of $\mathbf{growth}(q)\}$.
9:         Let $R_2 = \cup_{q \in S'}\{$the region covered by $q\}$.
10:        Draw $(i-2)$-boxes to fill the region between the boundaries of $R_1$
           and $R_2$.
11:        Draw $i$-boxes to fill the region between the boundaries of $R_1$ and $S$;
           break each cell boundary with an endpoint incident to $R_1$ into four
           edges of length $2^{i-2}$, to satisfy Invariant 1.
```

There are a couple of things worth noting about algorithm 10. For once we only process stages in which something happens, indicated by the choice of i in step 2 to step 6. These cases are if $MSF(i)$ changes, that is two trees merge into one, or there are complex components of $\mathcal{Q}(i)$ whose **growth** computation is nontrivial. By this we mean we only compute $growth(S)$ for complex components and for simple components that will merge with other components soon, and compute the equivalence classes of $\mathcal{Q}(i)$ only for this same set of quads. Simple components that are well-separated from others are not involved in these computations since they by nature are quite trivial.

The running time of this algorithm is dominated by the $O(\mathcal{K} \log \mathcal{K})$ required for a plane sweep [12] of $k = |\mathcal{Q}(i,t)|$ quads in step 20. There are $O(k)$ quads in complex components either in $\mathcal{Q}(i,T)$ or in $\mathbf{Q}(i+2,T)$, so there are $O(\mathcal{K})$ edges drawn for these quads at stage i or $i+2$. We amortize this cost by charging $O(\log \mathcal{K})$ per edge of the subdivision getting $O(n \log n)$ time overall. The computation of the Delaunay triangulation and the minimum spanning forest contributes a term of the same asymptotic magnitude.

We have established the following lemma.

Lemma 34. *(Lemma 6.11 in [6])*
Algorithm build-subdivision can be implemented to run using $O(n \log n)$ standard operations on a real RAM, plus $O(n)$ floor and base$_2$ logarithm operation.

Chapter 6

Wavefront propagation

This chapter presents the results of Hershberger and Suris wavefront propagation, and is therefore heavily inspired by section 4 and 5 of [6].

Here we will present wavefront propagation with the unmodified Hershberger-Suri algorithm in mind. One of the only differences in the regards to the modified Hershberger- Suri, is the sources of the wavefronts, can be sub-edges inn the modified algorithm, instead of points in the unmodified.

When the conforming subdivision has been constructed we are ready to actually simulate the continuous Dijkstra method, by propagating through the subdivision with a wavefront expanding at a unit-speed spreading among the obstacles and cells. At simulation time t, we say that a wavefront consists of all the points whose shortest-path distance to the source is t. See Figure 6.1. Such a wavefront is a set of disjoint paths and closed cycles. Each path or cycle is a sequence of circular arcs, called *wavelets*. Each of these wavelets are centered on a obstacle vertex that is covered by the wavefront. These vertices are called *generators* of the wavelets. This is the reason that Figure 6.1 has multiple sources, since if both the dashed and dotted arches had source at s, their paths from s to g would overlap (the same for s to g'). The obstacle vertices g and g' are engulfed by the wavefront with source s, and becomes generators for their own wavelets.

As the wavefront expands, the meeting point of two adjacent wavelets sweeps along a *bisector curve*, and divides the area between them with a hyperbolic bisector of the two wavelets generators[1]. These ideas can be seen on Figure 6.2. Here g and g' are generators, who each starts a wavelet marked by the dotted line. These two wavelets meets, and for every point they meet, the shortest distance from this point has equal length to both g and g'. This is what creates the horizontal line between g and g', which is the hyperbolic bisector. The wavefronts creates paths, which have their endpoints when the wavelet meet obstacles, or the planes outer boundary. These endpoints sweeps along the obstacle boundaries as the wavefront expands.

From the above we see that the topology, or "shape", of wavefronts during the simulation changes in the case of two different event: wavefront-wavefront collisions and wavefront-obstacles collisions.

[1]see appendix A

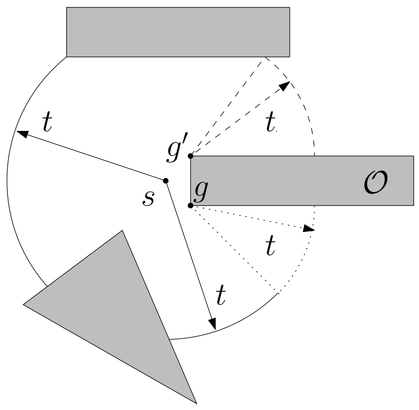


Figure 6.1: An example of a wavefront propagation from s with distance t to all points of its circular arch. Since a path into the dashed and dotted area would require a turn at O, these areas are propagated by g and g'. Since we look at the propagation at time t both generator would have propagated a distance t.

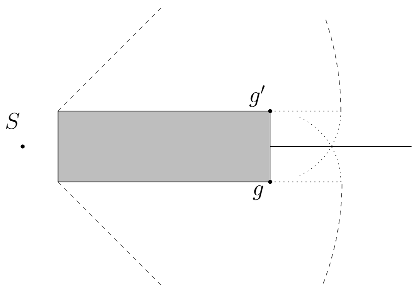


Figure 6.2: The adjacent generator g and g' each produce a wavelets which propagates the space, these are represented by the dotted circular arcs. These will overlap, and the split the area between them into two equal sized regions. The fully drawn line segment between them then represent the splitting point between the two generators, where any point on the line segment, has equal length shortest path to both of the generators.

6.1 Overview of propagation algorithm

The wavefront propagation algorithm operates in two phases: first a wavefront propagation phase, and second a map computation phase. The wavefront propagation phase, simulates the wavefront, and thereby determines the approximate locations of the different wavefront collision events. We remind that there are two different kind of wavefront collisions, the first being the collisions where wavelets are neighbors in the wavefronts, that is adjacent wavelets and the collision between them. The second being collisions between non-neighboring wavelets. This propagation happens through adjacent cells, and only across their transparent edges. We remind the reader that transparent edges are the edges established by the subdivision, and opaque edges, are the edges of the obstacle polygons. The map computation phase uses the information of wavefront collisions to build a shortest path map in each cell in the conforming subdivision.

One of the main idea behind the algorithm, is the idea of calculating two "*single-sided approximate wavefronts*" where each approximate wavefronts will approach the transparent edge from their own side. This idea comes from the fact that literally translating the idea of wavefront propagation into an implementation would be very hard. Instead we are contempt with calculating for each transparent edge, two *approximate wavefronts*, which will pass through the transparent edge, one from each side. So the job of the wavefronts is to assign an value t to each point p on an transparent edge. Here t is the time it takes to travers at unit speed from the source s to p. Each p would have two such values, one from each side, where the minimum or these would the one we would use for a shortest path. In some cases we determine if a portion of a wavelet w arrives after the wavelet w' from the other side has fully engulfed an edge, that there is no need to record the time of w since it is so much later than w'. This is also a reason why we refer to it as approximate, since the approximate wavefront might not necessarily give a complete view of all wavelets time to an edge.

6.1.1 Definitions and terminology for propagation algorithm

By visiting the cells in a correct order, and going cell by cell we can calculate the correct time in which a wavefront hits a transparent edge, within a giving approximation which will be enough for our purpose. To do that we formalize the following to sets of edges for an edge e, $\mathit{input}(e)$ and $\mathit{output}(e)$.

By $\mathit{input}(e)$ we mean the set of edges whose approximate wavefronts we use when computing the approximate wavefront collision with e, and as such the distance to e. This set consists of transparent edges on the boundary of $\mathcal{U}(e)$ which is the well covering region of e. This is described in section 5.1. Computing the approximate wavefront at e then consist of propagating the approximate wavefronts from $\mathit{input}(e)$ to e inside $\mathcal{U}(e)$. It is quite clear that a shortest path without violation only needs to bend in the case getting around an obstacle, and the same is true for the wavefront. Because $\mathcal{U}(e)$ neither needs to convex, or even simply connected, nonconvexity of $\mathcal{U}(e)$ can block the wavefronts from some edges of $\mathit{input}(e)$ from reaching e. Typically, paths corresponding to blocked wavefronts either pass through free space outside $\mathcal{U}(e)$ and re-enter through other edges of $\mathit{input}(e)$ of simply run into obstacles outside $\mathcal{U}(e)$.

By $\mathit{output}(e)$ we refer to the set of edges where e influences the approximate wavefront. Formally we define $\mathit{output}(e)$ as

$$\mathit{output}(e) = \mathit{input}(e) \cup \{f | e \in \mathit{input}(f)\}$$

The reason $\mathit{output}(e)$ contains $\mathit{input}(e)$ is the algorithm is depending on $\mathit{output}(e)$ having a cyclic enclosing of e for detecting wavefront collision events.

Lemma 35 (Lemma 4.1 in [6]).
For any transparent edge e, $\mathit{output}(e)$ contains a constant number of edges.

Proof. Due to $|\mathcal{U}(f)| = O(1)$ for all edge f, and each $\mathcal{U}(f)$ being a connected set of cells of $\mathcal{S}'$, no edge e can belong to $input(f)$ for more than $O(1)$ edges f. □

The implementation of the wavefront propagation is loosely synchronized. A main idea of approximate wavefront propagation being approximate lies in the following implementation: For a transparent edge $e = \overline{ab}$ we define

$$\tilde{d}(s,e) = \min(d(s,a), d(s,b))$$

This estimates $d(s,e)$ because if the wavefront hits a or b then $\tilde{d}(s,e) = d(s,e)$, with $d(s,e)$ being the real distance from s to e. Should the wavefront hit right in the middle, between the endpoints of $\overline{ab} = e$, then the distance to a and b from the point the wavefronts collides with would be $(s,e) = d(s,e) + \frac{1}{2}|e|$. Since if we, in the later case, move the point of collision in any direction the distance becomes smaller, since it must hold that $d(e,s) \leq \tilde{d}(e,s) \leq d(e,s) + \frac{1}{2}|e|$.

Since we want to compute the covering time of each e, i.e. the time at which e is completely covered by the wavefront. We set the time to $\tilde{d}(s,e) + |e|$. It is obviously a conservative estimate of when the whole edge is fully covered. This time can easily be calculated on the fly and only be looking at the $input(e)$. We denote the time where the edge e is fully covered by $covertime(e)$.

6.1.2 The propagation algorithm, main loop

Initially we look at every e that is in the well-covering region $\mathcal{U}(e)$ (which also includes the source point s). We proceed to calculate an upper bound on $\tilde{d}(s,e)$ considering only straight-line paths inside $\mathcal{U}(e)$ and set the $covertime(e) = \tilde{d}(s,e) + |e|$. For all other edges e we initialize $covertime(e) = \infty$. This implies if the $covertime(e)$ is set to ∞, then the shortest path $\pi(s,a)$ or $\pi(s,b)$ must exit the boundary of $\mathcal{U}(e)$.

The algorithm for simulation, maintains a time parameter t and processes each edge in order of its covertime. The main loop of the simulation is as follows:

Algorithm 12 Propagation Algorithm

```
1: while there is an unprocessed transparent edge do
2:     Select edge $e$ with minimum $covertime(e)$
3:     Set time $t$ to $covertime(e)$
4:     compute the approximate wavefronts at $e$ based on the approximate
       wavefronts from all edges $f \in input(e)$ satisfying $covertime(f) <
       covertime(e)$
5:     Compute $d(s,v)$ exactly for each endpoint $v$ of $e$.
6:     for each edge $g \in output(e)$ do
7:         Compute time $t_g$ when approximate wavefront from $e$ first engulfs
           an endpoint of $g$
8:         Set $covertime(g)$ to $\min(covertime(g), t_g + |g|)$.
```

Lemma 36 provides a proof of the propagation algorithms consistency, by showing *covertime*() is correctly maintained and the edges needed for processing e would already have been processed.

Lemma 36 (Lemma 4.2 in [6]). *During the wavefront propagation the following invariants hold:*

(a) If a wavefront of an edge $f \in input(e)$ contributes to an approximate wavefront of e then $\tilde{d}(s,f) + |f| < \tilde{d}(s,e) + |e|$.

(b) The value of covertime$(\cdot)$ is updated a constant number of times.

(c) The final value of covertime(e) is $\tilde{d}(s,e) + |e|$. This value is reached no later than the simulation clock reaches that time.

(d) Edge e is processed at simulation time $\tilde{d}(s,e) + |e|$

Proof. The parts of the lemma are proven individually

(a) If a wavelet is able to contribute to the approximate wavefront at e it must be the case that it reaches e at some time t_e where $d(e,s) \leq t_e \leq \tilde{d}(s,e) + |e|$. On the way from s to e the wavelet either goes straight from s in side $\mathcal{U}(e)$ or by going through another transparent edge $f \in inpute(e)$ at an earlier time t_f with $d(s,f) \leq t_f < \tilde{d}(s,f) + |f|$ and $t_e \geq t_f + d(f,e)$. Since we know from $(W3_{fs})$ of a well-covering region with parameter 2, $d(f,e) \geq 2|f|$ and so $t_e \geq d(s,f) + 2|f|$. Since $\tilde{d}(s,f) + \frac{1}{2}|f|$, it must be the case that $\tilde{d}(s,f) + |f| < \tilde{d}(s,e) + |e|$.

(b) The value of *covertime*(e) is only updated when an edge f is processed from either $f \in input(e)$ or $e \in input(f)$. There are $O(1)$ such edges by Lemma 35

(c),(d) These are proven by induction on the simulation clock. (c) and (d) holds for the edges e whose initial *covertime*(e) values are not infinite. The wavelets that first reaches an endpoint of e, at $t_e = \tilde{d}(s,e)$ passes through some $f \in input(e)$. Because of the base-case in the induction we know that f has has already been visited before the simulation clock reaches t_e and so $covertime_e$ is set to $\tilde{d}(s,e) + |e|$ no later than $t_e = \tilde{d}(e,s)$. The variable $covertime_e$ cannot be set to any smaller value, because no approximate wavefront can reach the endpoints of e earlier than $\tilde{d}(s,e)$. It follows that e will be processed at simulation time $\tilde{d}(s,e) + |e|$. □

Lemma 37 (Lemma 4.3 in [6]). *For every vertex v of our conforming subdivision, the propagation algorithm correctly determines the distance $d(s,v)$ before v is used as a generator in any wavefront.*

Proof. In a conforming subdivision, every vertex v is an endpoint of a transparent edge e. The wavefront that creates the distance $d(s,v)$, either reaches v by only traveling within the boundary of $\mathcal{U}(e)$ or exists through an edge $f \in input(e)$ s.t. $covertime(f) < covertime(e)$. In the case of not leaving $\mathcal{U}(e)$, the initialization trivially computes $d(s,v)$ correctly. The case of leaving $\mathcal{U}(e)$, step 4 and 5 in algorithm 12 implies $d(s,v)$ would be correctly computed. Should v be an obstacle vertex, it may appear as a generator in a wavefront, but it will not be used until $d(s,v)$ is computed at time $\tilde{d}(s,e)+|e|$ (Lemma 36 (d)) $\square$

Even though a well-covering union of cells $\mathcal{U}(e)$ has constant complexity, it might not be a simple connected component. One could consider the case of a square annulus. Consequently there might be multiple topologically distinct paths from a boundary edge $f \in \mathit{input}(e)$ to e. But we're not interested in comparing paths of different topologies, so to avoid comparisons of different topological paths we split the wavefront $W(e)$ into topologically equivalent pieces.

For this purpose, let $W(e)$ denote one of the approximate wavefront passing through e. Now when we will compute $W(e)$ from the set $\{W(f) \mid f \in \mathit{input}(e)\}$, we will use topologically constrained versions of the two incoming wavefronts, which we will denote $W(f,e)$. In this context a wavefront $W(f,e)$ will be a portion of $W(f)$ that follows a single topological path inside $\mathcal{U}(e)$ from f to e.

To further extend this notation, we can consider a $\mathcal{U}(e)$ which contains holes. In this cell there will therefore be multiple topologically distinct paths from an edge $f \in \mathit{input}(e)$ to e. When distinguishing between the multiple topologically different wavefronts from a single edge f to e, we will use a primed notation $W(f,e)$, $W(f',e)$ etc.

Lets assume that two point $p,q \in e$ are hit by a single topologically constrained wavefront $W(f,e)$. The segment of e which has p and q as endpoints then all of the segments points has among their predecessor the generator vertices in $W(f)$, which intersects f and e. Also the quadrilateral[2] bounded by the segments of f and e, which is a subset of $\mathcal{U}(e)$. Such paths are not always segments. We can imaging an obstacle vertex v which lies in a well-covering region of e, and the path from f to p turns at v. This would then imply that the predecessor of p in $W(f,e)$ may be v. Should this be the case, then the paths from p and q to f can be continuously deformed (there is no obstacles between the two paths) to each inside $\mathcal{U}(e)$.

Unless source $s \in \mathcal{U}(e)$, then for any points $p \in e$ the shortest path $\pi(s,p)$ would pass through some $f \in \mathit{input}(e)$, an so constrain the source wavefronts to pass through $\mathit{input}(e)$, and by doing so not lose any essential information for the path.

6.1.3 The artificial wavefronts

As mentioned earlier, conceptually when calculating the distance to a transparent edge e, we can get to situation where one wavefront will consume the edge way before the other edge even reaches e. In such a situation we would want to discard the wavefront arriving later, because we don't need it since its being dominated. The concept for this is the artificial wavefronts. This mechanism will also be our only mechanism for pruning the wavefront that arrives second at a transparent edge. The easiest way to understand artificial wavefronts is by an example, see Figure 6.3 below.

[2]A figure consisting of 4 vertices in a euclidean space

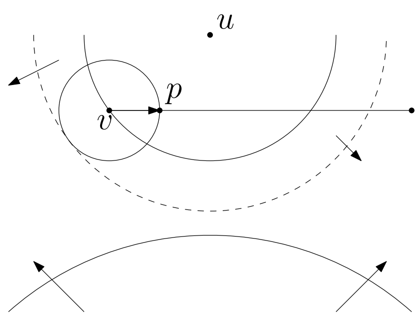


Figure 6.3: An example of an artificial wavefront from v reaching point p on edge e[6]

Here u's wavefront engulfs the left side of the transparent edge below it, lets call it e. The left endpoint of e is v. When v is engulfed, v will generate a artificial wavefront, which will run along e. In the figure we see that v's artificial wavefront engulfs p before the wavefront from the other side even reaches e. By the triangular inequality we have $d(s,p) \leq d(s,v) + |\overline{vp}|$ for any point $p \in e$. This surely means that the upper wavefront reaches p first, and there is therefore no need to continue the propagation of the lower wavefront through p.

So when computing the approximate wavefront passing through e from below, the contributing wavefronts are the following:

1. All wavefront $W(f,e)$ for $f \in \mathit{input}(e)$ and f below the line supporting e. As mentioned before, we differentiate between paths of topological difference, so if f intersect the line supporting e, we then split $W(f,e)$ into two, and keep only the portion $W(f',e)$ that comes from the part of f below e.

2. An artificial wavefront expanding from each endpoint of e. These generator, e.g. v from Figure 6.3 has weight $d(v,s)$.

So in essence, the artificial wavefront is a convenient mechanism for discarding parts of the actual wavefront which will be completely dominated by other parts of the wavefront. Since we only use of the artificial wavefronts to discard parts of incoming wavefronts, their generators will not be passed on to $\mathit{output}(e)$ as part of the approximate wavefront, unless it's also a vertex of the set of obstacles $\mathcal{O}$.

Proof for artificial wavefronts

The following proofs are taken from [6], and included for completeness.

Consider a set of wavefronts that reach e from the same side. We say that a contributing wavefront $W(f)$ claims a point $p \in e$ if $W(f)$ reaches p before any other contributor from the same side of e.

Lemma 38 (Lemma 4.4 in [6])**.** *Let e be horizontal and let $W(f,e)$ and $W(g,e)$ be two contributors to the approximate wavefront that passes through e from below. Let p and p' be points on e claimed by $W(f,e)$ and let q be a point e claimed by $W(g,e)$. The q cannot lie between p and p'.*

Proof. Consider the the shortest paths $\pi(s,p)$, $\pi(s,p')$, and $\pi(s,q)$ in the modified environment in which e has been replaced by an open, opaque segment. These paths connect p and p' to f and q to g, inside $\mathcal{U}(e)$. Shortest paths $\pi(s,p)$, $\pi(s,p')$, and $\pi(s,q)$ do not cross. The subpaths of $\pi(s,p)$ and $\pi(s,p')$ inside $\mathcal{U}(e)$ can be continuously deformed to each other inside $\mathcal{U}(e)$, so g is not between them. It follows that q is not between them, either. □

Lemma 39 (Lemma 4.5 in [6]). *Let u and v be two obstacle vertices, both generating wavelets that are considered when the approximate wavefront passing through an edge e from below is computed. Then the bisector generated by u and v intersects e at most once in SPM(s).*

Proof. Suppose the bisector intersects e twice. Without lose of generality assume u lies inside the loop formed by the bisector and e. If the bisector intersects e twice in u lies inside the loop formed by the bisector and e. If the bisector intersects e twice in $SPM(s)$, then the segment from u to its predecessor must intersect e between the two bisector intersections. The means that $d(e,s) < d(u,s)$, in fact, $d(e,s) + 2|e| \leq d(u,s)$. Hence $\tilde{d}(e,d) + |e| < d(u,s)$ and u cannot contribute to the approximate wavefront at e: it does not become a generator until after e is processed, contradicting the assumption that both u and v contribute to the approximate wavefront at e. □

Lemma 40 (Lemma 4.6 in [6]). *Given $W(f,e)$ for each f below e that contributes to $W(e)$ we can compute the interval of e claimed by each $W(f,e)$ in $O(1+m)$ total time, where m is the total number of generators in all wavefronts $W(f,e)$ that are absent from $W(e)$.*

Proof. For each contributing wavefront $W(f,e)$, we show how to determine the portion of e claimed by $W(f,e)$ if only one other contributing wavefront $W(g,e)$ is present. Lemma 4.4 implies that this portion is contiguous. The intersection of these claimed portions taken over all other contributors $W(g,e)$, is part of e claimed by $W(f,e)$ in $W(e)$.

In constant time we determine whether the claim of $W(f,e)$ is left or right of that of $W(g,e)$. If both $W(f,e)$ and $W(g,e)$ reach the left endpoint of e in constant time, check which one reaches it sooner. Otherwise one of $W(f,e)$ and $W(g,e)$ reaches a point on e that is left of any point reached by the other, and this point determines the ordering. Without loss of generality, assume that the claim of $W(f,e)$ is left of that of $W(g,e)$

By Lemma 4.4 we can combine the two wavefronts using only local operations. Let a denote the generator in $W(f,e)$ claiming the rightmost point on e. Let p_a be the left endpoint of a's interval on e. Similarly, let b denote the generator in $W(g,e)$ claiming the leftmost point on e and let p_b be the right endpoint of b's interval on e. Compute the bisector of a and b, and let its intersection with e be the point x. (By Lemma 4.5 there is only one intersection point in $SPM(s)$. If the hyperbola generated by a and b intersects e twice, then a is to the left of b at only one of the intersections, and we use that intersection as x.) See Figure 4.2. If x is to the left of p_a, then delete a from $W(f,e)$ if x is to the right of p_b, then delete b from $W(g,e)$ in either case redefine a, b, p_a, p_b recompute x, and repeat this test. If p_a is left of p_b and x lies between them then x is the right endpoint of $W(f,e)$'s claim in the presence of $W(g,e)$

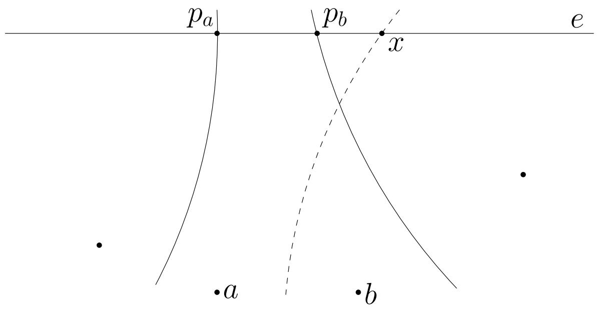


Figure 6.4: The contribution of b to $W(e)$ is constrained to be left of p_b and right of x and therefore does not exist[6].

By combining the claimed regions for all contributors $W(f, e)$ we construct the approximate wavefront e. The time bound follows since spend constant time per generator that is deleted for each pair of wavefronts, and he total number of wavefronts $W(f, e)$ to be merged is also constant. This finishes the proof □

Lemma 41. *Any generator deleted during the construction of an approximate wavefront at edge e does not contribute to the true wavefront at e. Every generator that contributes to the true wavefront at e either is s or belongs to one of the approximate wavefronts at e.*

Proof. The first part is clear since every deleted generator is dominated by some other generator at e. The second part follows by induction from two facts: any wavelet that contributes to the true wavefront at e must come either from s inside $\mathcal{U}(e)$ or through one of the edges in *input*(e) (by the definition of well-covering). The approximate wavefronts at *input*(e) are ready before they are needed to construct $W(e)$ (by Lemma 4.2) □

6.1.4 The bisector events

First we remind the reader that when we speak of bisectors, we refer to the meeting points of two adjacent wavelets along a bisector curve, which divides the area between them with a hyperbolic bisector, see Figure 6.2. When we talk about bisector events we mean the event of intersections of bisectors with each other or with obstacles. These may happen when we propagate an approximate wavefront $W(e)$ to *output*(e). These bisector event may be detected in two ways:

1. During the computation of $W(e, g)$ from $W(e)$ for some $g \in$ *output*(e). This kind of bisector event is detected when simulating the propagation of wavefront from e to g to compute $W(e, g)$. In particular, when two generators u and v are non-adjacent in $W(e)$, but at any time in the propagation from e to g become adjacent, then there is a bisector event involving u and v.

2. During the merging process described in Lemma 40. Let a generator v be contributing to one of the input wavefronts $W(e, g)$, but not the the

merged wavefront $W(g)$ at g, then v will be involved in the bisector event on the way from e to g.

The algorithm for detecting bisector events, detects these in a small proximity to their actual location in $SPM(s)$. The process of doing this is by *marking* the generators that participate in a bisector event in $O(1)$ cell near where the event is detected. So if a generator v is involved in a bisector event in a cell c, then v is guaranteed to belong to a set of marked generators for c. It may however be the case that the set of marked generators for a cell c is a super set of generators that actually participate in bisector events in c. The proof of showing that this total number of generator marked in all the cell in $O(n)$ time will be shown in Lemma 43

We here state the rules for Marking generators as given in [6].

1. If a generator v lies in a cell c, them mark v in c.

2. Let e be a transparent edge, and let $W(e)$ be the approximate wavefront coming from some generator v's side of e.

   a. If v claims an endpoint of e in $W(e)$, or if it would do so except for an artificial wavefront, then mark v in all cells incident to the claimed endpoint.

   b. If v's claim in $W(e)$ is shortened or eliminated by an artificial wavefront, then mark v in the cell on v's side of e.

3. Let e and f be two transparent edges with $f \in \mathit{output}(e)$. Mark v in both the cells that have e as an edge if one of the following event occurs:

   a. v claims an endpoint of f in $W(e, f)$;

   b. v participates in a bisector event detected either during the computation of $W(e, f)$ from $W(e)$, or during the merging step at f. We also mark v as having a bisector event if v's claim of $W(f)$ is shortened by an artificial wavefront.

4. If v claims part of an opaque edge when it is propagated from an edge e toward $\mathit{output}(e)$, mark v in both cells with e on their boundary.

From the above we see that both rule 2a and 3a apply when a wavefronts claims an endpoints of an edge e. The difference lies in 2a marking cells near the claimed end point while 3a marks generators in cells near the source edge of the wavefront.

Proof for bisector events

The following proofs are taken from [6], and are included for completeness.

A generator may contribute to a wavefront more than once in the wavefront sequence; each mark applies to only one instance of the generator in the sequence. The following technical lemma is used in the proof of Lemma 43 to establish the correctness of the marking rules.

Lemma 42 (Lemma 4.8 in [6])**.** *Let v be a generator that contributes to an approximate wavefront $W(e)$ suppose there is a point $p \in e$ that is claimed by v in $W(e)$ but not in $SPM(S)$ (because a wave from the other side of e reaches p first) Then v is marked in the cell c on v's side of e.*

Proof. If v is unmarked in c there must be generators u and w such that u, v, w are consecutive in $W(e)$ — otherwise Rule 2 would apply. The bisectors (u, v) and (v, w) must exit from $\mathcal{U}(e)$ though the same transparent edge h — otherwise Rule 3 or 4 would apply. For the same reason, the region bounded by (u, v), (v, w), h, and e is a subset of $\mathcal{U}(e)$ — if the region contained a non-$\mathcal{U}(e)$ island, v would claim an endpoint of a boundary edge of that island. Edge h is by definition part of $input(e)$. Consider the point $p \in e$ that is claimed by v in the approximate wavefront $W(e)$ but not in the true wavefront at e, and suppose that the true predecessor of p is $z \neq v$. The vertex z is either an obstacle vertex or the source s. In the former case, z lies outside $\mathcal{U}(e)$ or on its boundary $\partial\mathcal{U}(e)$ — by condition (C3), $\mathcal{U}(e)$ contains at most one obstacle vertex, so any vertex not strictly outside $\mathcal{U}(e)$ must be connected to points outside $\mathcal{U}(e)$ by opaque edge. Vertex z may lie strictly inside $\mathcal{U}(e)$ only if $z = s$.

Let us first assume that z lies outside the well-covering region $\mathcal{U}(e)$ — the proof simplifies in the other case, which is considered below. Let q denote the intersection point between $\overline{zp}$ and $input(e)$ closet to p (recall that $input(e) \subset output(e)$ and $input(e) \subset \partial\mathcal{U}(e)$). Based on the position of q relative to the bisector (u, v) and (v, w) we argue that v must have been involved in a bisector event detected by our algorithm and thus marked in cell c.

First consider the case in which q lies between the bisectors (u, v) and (v, w) on the edge h. Now, since $|\overline{qp}| \geq |h|$ (by the well-covering property), the endpoints of h are engulfed by a wavefront from z or from some other generator before the wavefront from z reaches p at time $d(s, z) + |\overline{zp}|$. The artificial wavefront from h's endpoints will cover h before time $d(s, z) + |\overline{zp}| + |h|$. By assumption we have $d(s, v) + |\overline{vp}| > d(s, z) + |\overline{zp}|$. The wavefront from v cannot reach e earlier than $d(s, v) + |\overline{vp}| - |e|$. By well-covering with parameter 2, $d(e, h)$ is at least $|e| + |h|$ and so the wavefront from v reaches h no earlier than $d(s, v) + |\overline{vp}| + |h| > d(s, z) + |\overline{zp}| + |h|$, at which time h is already covered by the artificial wavefront. The claim of v on h is shortened by the artificial wavefront (in fact a's claim is eliminated completely), and so it must be marked by Rule 3b.

In the second case, q is not between the bisectors (u, v) and (v, w) on h. The segment $\overline{qp}$ must intersect one of the bisectors. Without loss of generality, assume $\overline{qp}$ intersects bisector (u, v). Since every point on $\overline{qp}$ has z as its predecessor in $SPM(s)$, the bisector (u, v) does not reach $\partial\mathcal{U}(e)$ in $SPM(s)$. We show that our propagation and merging algorithms will detect a bisector event for (u, v). Let r be the intersection point between the bisector (u, v) and the edge h. As noted in the discussion after Lemma 37, the triangle defined by the segments $\overline{ur}$, $\overline{vr}$ and e is a subset of $\mathcal{U}(e)$. Bisector (u, v) crosses the triangle boundary on e and at r, but nowhere else. The larger region R bounded by e, h, $\overline{ur}$ and bisector (v, w) also is a subset of $\mathcal{U}(e)$ and it contains point p. Because $\overline{qp}$ crosses into R to intersect (u, v) and it does not intersect the (v, w) or h sides of R, $\overline{qp}$ must intersect $\overline{ur}$ let x be the point of intersection. The wavelet from z reaches x before the one from u, so the path $z \to x \to r$

starting at time $d(s,z)$ reaches r before the path $u \to r$ starting at time $d(u,s)$. Observe also that the path $z \to x \to r$ is a legal path — it lies in free space. Now, consider the shortest path from z to r inside the triangle $\triangle zxr$ that does not cross h or any obstacle edge (see Figure 6.5).

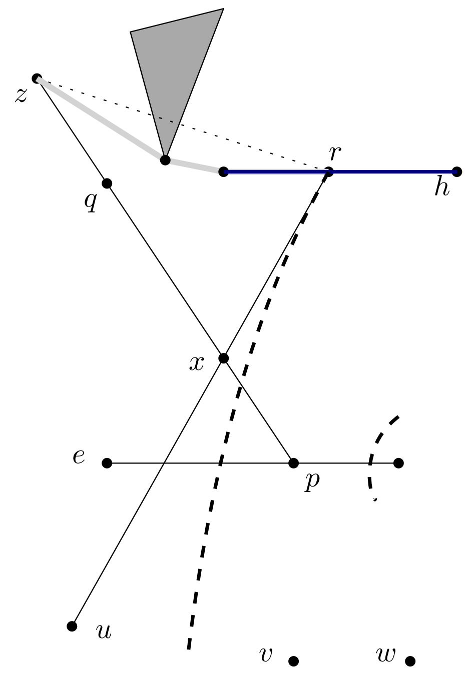


Figure 6.5: The shaded path from z to r claims r before the wavelet from u, and from the same side of h as u[6].

Because $z \to x \to r$ lies in free space, such a path exists and is shorter than $z \to x \to r$. This path claims r from the same side as u before the wavelet from u reaches r. (If the path passes through an endpoint of h, then an artificial wavefront claims r otherwise the last obstacle vertex on the path claims r.) Thus a bisector event for (u,v) is detected during the computation of $W(e,h)$ or $W(h)$ and v is marked by Rule 3b.

Next consider what happens if the predecessor vertex z lies on the boundary of the well-covering $\mathcal{U}(e)$. Let h be a boundary edge $\mathcal{U}(e)$ incident to z. In this case we detect a bisector event involving v when we advance the wavefront from e to *output*(e): if z lies between the bisectors (u,v) and (v,w) then v is marked by Rule 3a or 4 if z is not between the bisectors, the segment zp intersects one of the bisectors, say (u,v) and we detect a bisector event for (u,v) in advancing the wavefront from e to *output*(e).

Finally consider the case in which $z = s$ lies inside $\mathcal{U}(e)$. If z is not between the bisectors (u,v) and (v,w) segment $\overline{zp}$ intersects one of them and the proof is as above. Let r be the intersection of (u,v) with h and let t be the intersection of (v,w) with h. The convex quadrilateral bounded by subsegments of e, $\overline{ur}$, h, and $\overline{tw}$ is contained inside $\mathcal{U}(e)$. Hence if z is between the bisectors (u,v) and (v,w), the entire segment $\overline{rt}$ is visible from z (that is $\triangle zrt$ is empty) and so v's claim on h is eliminated by z. Therefore v is marked by Rule 3b. This completes the proof.

□

Lemma 43 (Lemmma 4.9 in [6]). *If a generator v participates in a bisector event of $SPM(s)$ in a cell c, then v is marked in c.*

Proof. If a bisector has an endpoint on an opaque edge of c, it either emanates from an obstacle vertex on the edge, or it is defined by two generators that claim part of the opaque edge. Rule 1 and 4 guarantee that all such generators are marked in c. If a generator v that contributes to an approximate wavefront in c is unmarked then by Rule 2a there must be transparent edges e and f on the boundary of c such that $W(e)$ and $W(f)$ both contain the generator subsequence u, v, w for some u and w. Without loss of generality assume $W(e)$ enters c and $W(f)$ leaves c. If v participates in a bisector event of $SPM(s)$ in c, then at least one point p inside the region R bounded by e, f, (u, v) and (v, w) is not claimed by v in $SPM(s)$. Let z be the true predecessor of p. Let r and t be the intersections of (u, v) and (v, w) with f, respectively. Region R is contained in the convex quadrilateral Q bounded by $\overline{ur}$, $\overline{rt}$, $\overline{tw}$, and the line supporting e. Because u, v, w is a subsequence of $W(e)$, no vertex on the same side of e as v claims any point of the side of Q collinear with e, that is, $\overline{zp}$ does not cross that side of Q. If r and t are both claimed by v in $SPM(s)$ then $\overline{ur} \in \pi(s, r)$ and $\overline{wt} \in \pi(s, t)$. In this case $\pi(s, p)$ cannot cross $\overline{ur}$ or $\overline{wt}$ and hence it must cross $\overline{rt}$. The intersection of $\overline{zp}$ with $\overline{rt}$ is a point q that satisfies the hypothesis of Lemma 4.8, and so v is marked in c. On the other hand if either r or t is not claimed by v in $SPM(s)$ that vertex satisfies the hypothesis of Lemma 4.8 and so v is marked in c. □

The following technical lemma shows that the approximate wavefront are not too different from the true wavefronts this lets us bound the number of marks made by the marking rules.

Lemma 44 (Lemma 4.10 in [6]). *Let B be the set of pairs (e,b) of transparent edges e and bisectors b such that b crosses e in some approximate wavefront but the same crossing does not occur in SPM(s). Then $|B| = O(n)$*

Proof. Let (e, b) be a pair in B. Bisector b is defined by two generators u and v. The proof of Lemma 4.8 notes that each generator (except possibly s) is outside or on the boundary of $\mathcal{U}(e)$. That proof also shows that b's intersection with e in some approximate wavefront (that is, the presence of u and v in $W(e)$) is proof that u and v claim points on the boundary of $\mathcal{U}(e)$ (in *input*(e)) in $SPM(s)$. Let $p = b \cap e$. Because (e, b) is not an incident pair in $SPM(s)$, there must be at least one bisector event in $SPM(s)$ that lies in the interior of $\mathcal{U}(e)$ between the line segment $\overline{up}$ and $\overline{vp}$. We can charge the early demise of b to any one of these bisector events.

The segments $\overline{pu}$ and $\overline{pv}$ are disjoint inside $\mathcal{U}(e)$ from the corresponding segments defined by any other pair $(e, b') \in B$ in the modified shortest path problem in which the obstacles are $O \cup \{e\}$ the segments $\overline{pv}$ and $\overline{pu}$ belong to $\pi(s, p)$ and hence they are disjoint from any other such segments. Thus the sector bounded by $\overline{pu}$ and $\overline{pv}$ is disjoint inside $\mathcal{U}(e)$ from the sector defined by any other pair $(e, b') \in B$ so each bisector event inside $\sqcap(e)$ is charged at most once for all pairs in B that have e as the first element of the pair. Each cell in the conforming subdivision belongs to $O(1)$ well-covering regions $\mathcal{U}(e)$. Hence the sum over all transparent edges e of the number of bisector events in $\mathcal{U}(e)$ is only $O(n)$. This total is an upper bound on $|B|$. □

Lemma 45 (Lemma 4.11 in [6]). *The total number of marked generators over all cells is $O(n)$*

Proof. We begin by defining a propagation region for each edge e. For any transparent edge e let $P(e)$ be the collection of cells through which wavefronts propagate on the way from e to all edges $f \in output(e)$. Clearly $P(e) \subseteq \mathcal{U}(e) \cap \{\mathcal{U}(f)|f \in output(e)\}$. The number of cells in $P(e)$ is constant since $|output(e)|$ is constant and so is the number of cells in $\mathcal{U}(f)$ for any f. Furthermore since every cell of $P(e)$ is within a constant number of cells of e, each cell c belongs to $P(e')$ for only a constant number of edges e'.

The total number of generator-cell marks made under Rule 1 is clearly $O(n)$

Each $P(e)$ has constant complexity so there are $O(n)$ edge pairs (e, f) where e is transparent and f is either transparent and in *output*(e) or opaque and inside or on the boundary of $P(e)$. From this it follows that the number of marks made by Rules 2a and 3a is $O(n)$. similarly there are $O(n)$ Rule 4 marks in which the wavelet from v claims an endpoint of the opaque edge or is the first or last nonartificial wavelet in $W(e)$.

Any Rule 4 mark not yet counted involves a generator v that does not reach any opaque edge endpoint when propagated forward from e. Because v is not the first or last nonartificial wavelet in $W(e)$, there is a generator u such that v's claim on e in $W(e)$ is bounded on the left of bisector (u, v). We can assume that (u, v) intersects e in $SPM(s)$ by Lemma 4.10 there are only $O(n)$ bisector-edge pairs that intersect in approximate wavefronts but not in $SPM(s)$. Bisector (u, v) terminates in $P(e)$ either on the opaque edge or in a bisector event before the opaque edge. Let us charge the marking of v at e to this endpoint of (u, v) in $SPM(s)$. Because each cell belongs to $P(e')$ for a constant number of edges e', each vertex of $SPM(s)$ is charged $O(1)$ times. Since $|SPM(s)| = O(n)$, the number of RULE 4 marks is $O(n)$.

The proof for rule 2b and 3b are similar to that Rule 4. We begin with the proof for Rule 3b. We can assume that the interval claimed by v on e in $W(e)$ is bounded by two bisectors (v, u) and (v, w) for two nonartificial generator u and w the first and last generators in $W(e)$ counted separately sum to at most $O(n)$ overall. Furthermore we can assume that (u, v) and (v, w) both intersect e in $SPM(s)$ there are only $O(n)$ bisector-edge pairs that appear in some approximate wavefront but not in $SPM(s)$ (Lemma 4.10). At least one of the two bisector fails to reach the boundary of $P(e)$ in $SPM(s)$ because Rule 3b applies and a detected bisector event implies the existence of an actual bisector event no later than the point of detection we charge the marking of v to that bisector endpoint. Each bisector event gets charged $O(1)$ times and there are $O(n)$ bisector events in $SPM(s)$.

To bound the number of rule 2b marks, consider where the generator v lies. There is at most one generator v inside $\mathcal{U}(e)$ and so $O(n)$ marks for such generators overall. If v lies outside $\mathcal{U}(e)$ there is at least one edge in *input*(e) where v is marked by Rule 3b because of the shortening of v's claim on e.

Charge the Rule 2b mark at e to this Rule 3b mark. There are $O(n)$ Rule 3b marks and hence $O(n)$ Rule 2b marks □

We defer the finer details of the propagation algorithm to section 6.3 and instead describe the second phase of the algorithm next, namely the shortest path map computation.

6.1.5 Computing the shortest path map

So far in this section we have gone through the propagation phase. In this phase we have propagated the plane with wavefronts and wavelets from generator, and we have used approximate wavefronts for each transparent edge to sort out dominated wavefronts. We also marked generators in every cell c, where each marked generator is in the approximate wavefront of one of the boundary edges of c, an all but $O(1)$ of them contribute to a bisector event either in c or in one of $O(1)$ nearby cells. Here the bisector event was the intersections of bisectors with each other or with obstacles. The sketch of an algorithm from the proof of Lemma 40, will be presented in section 6.3, which will let us compute the marked generators in $O(\log n)$ time a piece.

This section is dedicated to show how we can break the interior area of a cell c into *active* and *inactive* regions. This splitting will be done on the basis of whether a vertex of $SPM(s)$ lie in a region. If it does we denote the area as active, and if no vertex is present in the area we denote it as inactive. We saw in section 6.1.4, lemma 43 that only marked generator would contribute to a bisector event in c.

The bisectors made by a marked generators and tehri unmarked neighbors generators will be drawn in the $SPM(s)$. All such bisectors are disjoint, and we will see the partition of c into active and inactive regions, will be done such that active regions are claimed only by unmarked generators and inactive regions by marked generators. This can be seen in Figure 6.6.

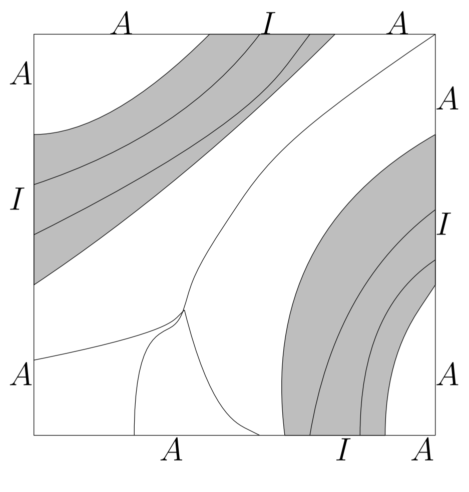


Figure 6.6: Here the white regions are the active regions and the shaded regions are the inactive. The boundaries between the regions encapsulating active and inactive region, are defined by being the bisector of one marked and one unmarked generator[6].

The algorithm will compute the active regions, which can be done in a time that is proportional to the number of marked generators in the cell c. Overall can this be done since we know the order of the generator along the boundary of c, which will help us finding the marked generators with unmarked neighboring generators. The calculated boundary of these active regions will have $O(1)$ segments. These segments will either be a transparent edge fragment, an opaque edge, of a bisector in $SPM(s)$.

We now let e be a transparent edge fragment that's bounding an active region. Further let $W(e)$ be the wavefront that enter the active region by crossing edge e. Should $W(e)$ be the only wavefront entering through e, it will partition the active region into piece which we call S-faces. Each S-face has a unique predecessor in the $W(e)$ partitioning the area. The S faces of an active region may not cover all of it, since each point in an S-face must be connected to its predecessor by a segment that intersect e. We define $S(e)$ to be this partition. So basically $S(e)$ are the building blocks of a shortest path map, which restricts the active regions and considers only generators in $W(e)$. Let assume a point p lies in an S-face of $S(e)$ with predecessor v, then $S(e)$ assigns weight $d(s, v) + |\overline{vp}|$ to p. Points outside any S-face are assigned infinite weight by $S(e)$.

We can compute $S(e)$ in $O(m \log m)$ time, where $m = |W(e)|$, by using the propagation algorithm and the auxiliary data structures which will be introduced in section 6.3.

Proof of correctness for computation of SPM

The following proofs are taken from [6], and included for completeness.

The following lemma shows how to combine the wavefronts incident to different boundary edges of an active region.

Lemma 46 (Lemma 4.12 in [6]). *Given the approximate wavefronts on the boundary of a cell c and a set of g marked generators in those wavefronts, we can compute the vertices of $SPM(s)$ inside c in time $O(g \log g)$.*

Proof. Consider an active region inside c and two transparent edge fragments e and f on the boundary of this active region. We can use the merge step from a standard divide-and-conquer Voronoi diagram algorithm[12] to compute the portion of the region nearer to $W(e)$ than to $W(f)$ using weighted distance in time $O(|W(e)| + |W(f)|)$. More specifically assume that $S(e)$ and $S(f)$ have both been computed. Let $m = |W(e)| + |W(f)|$. Each of $S(e)$ defines a distance function on the points of the active region. The pointwise minimum of these two functions determines which points are nearer to $W(e)$ than to $W(f)$ under weighted distance. Consider a point p in the S-face for some generator $v \in W(e)$. Point p belongs to v's S-face in $SPM(s)$ only if all of the segment $\overline{pv}$ is closer to v than to any generator in $|W(f)|$. The set of points p such that the entire segment from p to its predecessor is closer to $W(e)$ than to $W(f)$ is bounded by a single chain Γ of $O(m)$ hyperbolic arcs (see appendix A. (The number of arcs follows from Lemma 17.) To find Γ, first trace along a ray emanating from some generator $v \in W(e)$ marching through $S(e)$ and $S(f)$ simultaneously until the ray reaches the boundary of c or reaches a point whose weight in $S(f)$ equals its weight in $S(e)$. This takes $O(m)$ times since a line cuts $O(m)$ edges of $S(e)$ and $S(f)$ containing the current point trace along the hyperbola until it leaves one of the two S-faces then follow the hyperbola determined by the next pair of S-faces etc. This procedure takes $O(1)$ time per arc of Γ or $O(m)$ time altogether (see Figure 6.7).

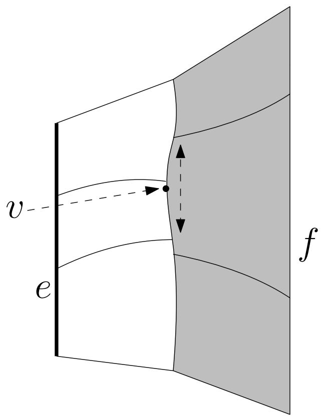


Figure 6.7: To find the region close to $W(e)$ than to $W(f)$ under weighted distance, trace a ray from some $v \in W(e)$ through $S(e)$ and $S(f)$ until it hits a point equidistant from the two wavefronts, then trace outward from the point along the bisector Γ[6].

The tracing procedure computes region closer to $W(e)$ than to $W(f)$ for one edge f. Intersecting the results for all such edges f on the boundary of the active region produces the region $R(e)$ claimed by $W(e)$ in $SPM(s)$. Intersecting $R(e)$ with $S(e)$ gives vertices of $SPM(s)$ to which $W(e)$ contributes. We repeat this computation for each transparent edge fragment to find all the vertices of $SPM(s)$ in the active region. Applying this algorithm to all active regions finds all vertices of $SPM(s)$ inside c.

The partition $S(e)$ determined by each edge fragment e participates $O(1)$ times in a Voronoi-style merge, so the total cost of merging is $O(g)$. Hence the running time is dominated by the propagation algorithm which takes $O(g \log g)$ times altogether. □

Lemma 47 (Lemma 4.13 in [6]). *The shortest path map vertices computed cell-by-cell can be combined to build $SPM(s)$ in additional $O(n \log n)$ time.*

Proof. To compute $SPM(s)$ we compute all its edges separately then use a standard plane sweep to assemble them as follows. Create a list of the bisector endpoints discovered in the computation of Lemma 46, each identified by a key consisting of two generators. Put each three-bisector endpoint into the list three times, once for each bisector. Put each bisector/edge collision in once labeled with the generators of the bisector. Now sort the list to group together endpoints belonging to each bisector. Take the endpoints belonging to the bisector of a generator pair (v, w) and sort them along the hyperbola determined by the weighted generators of v and w. This determines all edges of $SPM(s)$ on the hyperbola. Doing this for all pairs that appear as keys in the sorted list gives all $O(n)$ hyperbolic arcs of $SPM(s)$. Finally with a standard plane sweep, we can combine these arcs with the edges of $\mathcal{O}$ to build the subdivision $SPM(s)$. □

6.2 The list based data structure for wavefront propagation

To keep track of our approximate wavefront, we use a list to keep track of the generators coming from obstacle vertices. For this list we need a data structure which supports two types of operations:

1. *Standard list operations:* these are insert, delete, concatenate, split, find previous and next element, and search. Here the search operation locates the position of a query point. This position is found in the list of bisectors defined by the generators wavefront at a particular time.

2. *Priority queue operations:* these operations are used on the generators, to which we assign a priority. The data structure should be able to update priorities and find the minimum priority in the list.

These two types of operations should be implemented in a flavor of self balancing binary trees. One could implement the operations e.g. in a red and black search tree[17]. Here the generators will be located at the leaves, and each node should have a priority field which records the minimum priority of the leaves in it subtree.

The only requirements to the flavor of self balancing binary tree is the list operations should each take $O(\log n)$ time to process on a list of length $O(n)$. Each priority operation should likewise take $O(\log n)$ time each.

Beside these operations we also need the tree to be fully persistent, s.t. we can operate on past version of any list. Each kind of operation uses $O(1)$ storage per node of the binary tree, which mean we can make the data structure fully persistent by path copying. The effect of using an operation affects $O(\log n)$ nodes of the tree, which also includes the ancestors of every affected node. We simple, before an operation modifies the tree, copy all the nodes that will be affected, and then modify the copies. This way we create a new version of the tree while leaving the old version unchanged. Since the data structure uses $O(\log n)$ for each operation, and we save a copy of size $O(\log n)$, the total storage usage of the data structure is $O(m \log n)$ where m is total number of operations used. The above gives us the following lemma:

Lemma 48 (Lemma 5.1 in [6]). *There is a linear-space data structure that represents an approximate wavefront an supports list operations and priority queue operations in $O(\log n)$ time per operation. The data structure can be made fully persistent at the expense of an additional $O(\log n)$ space per operation.*

6.3 An implementation of the wavefront propagation

Concretely the idea of wavefront propagation is to propagate an approximate wavefront from an edge on the boundary of the cell containing the edge we want to propagate to. In particular, we want to propagate the wavefront $W(e)$, and compute $W(e, g)$ for every edge $g \in output(e)$, and in this process, assign the time (weight) of first contact between $W(e, g)$ and the endpoints of g. In this section we will show how to do this computation of $W(e, g)$ for all transparent edges g on the boundary of e's cell.

We know from lemma 35 that there is a constant number of edges in $output(e)$, which means they can only belong to a constant number of different cells. We will use this fact to compute $W(e, g)$ for all $g \in output(e)$. When propagating the wavefront cell-by-cell, one can think of this as splitting the wavefront $W(e, g)$ into multiple piece. Each piece is labeled by the sequence of crossings of the transparent edges it follows from e to g. The $W(e, g)$ is then made of these components wavefronts by concatenating the piece that are topological equivalent paths inside $\mathcal{U}(e)$.

Since each of these component wavefronts is a list of generators, we may find that adjacent wavefront piece can contain duplicate generator that claims the common endpoint. One of these duplicates are to be deleted before the lists are concatenated.

As an example of the above see Figure 6.8. Here $W(e, g)$ is assembled from $W(e', g)$ and $W(e'', g)$, where e' and e'' are two edges on the boundary of g's cell.

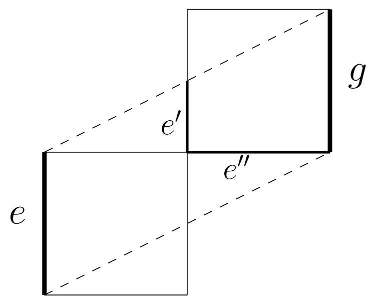


Figure 6.8: $W(e, g)$ is assembled from $W(e', g)$ and $W(e'', g)$, where e' and e'' are two edges on the boundary of g's cell[6].

The propagation algorithm assumes that each cell c is convex. Since we assumed the conforming subdivision could be build with square annulus's, this isn't satisfied immediately. For this we temporarily break a cell which isn't convex into subcells by adding transparent edges, which are *parallel to* e through the points of nonconvexity. An example of this i shown in Figure 6.9.

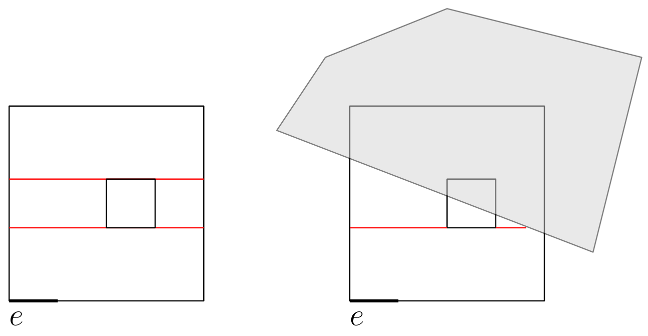


Figure 6.9: Insertion of transparent edges parallel to e to fix the cells non convexity. The right figure is the cases of an obstacle overlaying the cell[6].

Let f be a transparent edge of the boundary of c such that $f \neq e$. Then the propagation algorithm has the following invariant.

Propagation Invariant: When a wavefront $W(e, f)$ is propagated for distance $2 \cdot |f|$ beyond f, it intersects only a constant number of cells of the conforming subdivision of the free space.

We already saw in the former chapter that the conforming subdivision S' already satisfy the Propagation Invariant, since, by the propagation invariant, each edge f is well-covered with parameter 2. But since we just added some temporary transparent edges to non convex cells, we need to fix these in a way that abides to this invariant. This is done by dividing these added edges into $O(1)$ pieces, each no longer than the edges of $\mathcal{S}$ on the boundary of the annulus's outer boundary, one-eight the side length of the outer square, which we know is bounded by the minimum clearance property and uniform edge property of the conforming subdivision. These are explained in section 5.1.

We now let H denote the convex of e and the inner square of the annulus as shown in Figure 6.10. A convex hull is the smallest set of points which makes the set convex [12]. Should H intersect an added transparent edge f, we further subdivide the edge segment $f \cap H$ into pieces of length equal to the inner square of the annulus, as also shown on Figure 6.10.

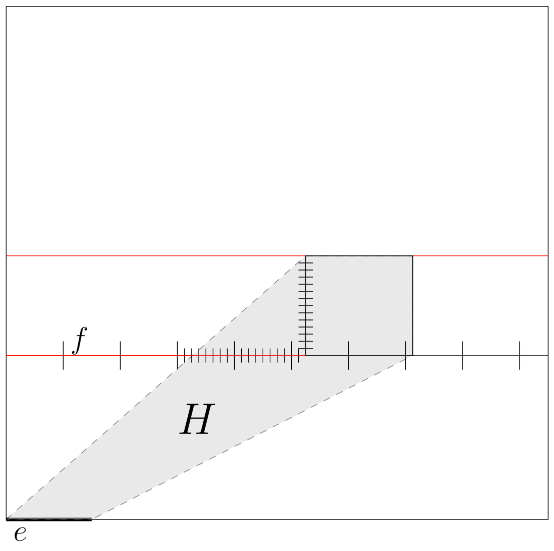


Figure 6.10: A subdivision of a cell, with a convex hull around a boundary edge e and the inner square of the annulus.

Due to f being parallel with edge e and the inner boundary of the annulus being well separated from the outer boundary (see again the minimum clearance property) the total length of f's edge segment inside H is proportional to the side length of the inner square. By this it follows that the partition of f only creates $O(1)$ edges. We can see that these subdivided edges satisfy the propagation invariant, by the following example. Given any such f, let g' be an edge of c such $W(e, f)$ leaves c by passing through g'. Since g' is an edge from the conforming subdivision of the free space, $\mathcal{S}'$, we know it is a fragment of an edge g in the conforming subdivision of the obstacles vertices and s, $\mathcal{S}$. We know we from lemma 62 (see appendix C) that there are $O(1)$ cells of S' within shortest path distance $2|g|$ of g' by construction. Since the new transparent edge which makes the cell c convex is subdivided into piece such that $|f| \leq |g|$, then this implies that the propagation invariant holds for the edge f.

6.3.1 Dynamic wavefront propagation

Up until now we have, in this chapter, kept the simulation of wavefront propagation rather static, in the sense that we have look at the atomic examples of how to handle different cases at different time in the propagation process. Now we take these ideas and see what happens when we let them operate in a dynamic setting. This means e.g. since a wavefront is a collection of generators, which one of the generators waves do we calculate first, and how do we drop a generator when it has served its purpose, and so on. So we see what happens to the wavefront propagation when we add the element of time to the process.

So what happens to the combinatorical structure of the wavefront as it sweeps across the (convex) cells, and how does the wavefront $W(e)$ behave as it sweeps across a cell c after entering it through the edge e? The simulation should detect and process each bisector event involving the generators from each wavefront, e.g. $W(e)$ that may occur inside c. The events are processed in an order of increasing distance to s. This is to simulate the element of time since the weight and therefore the distance to each generator is assigned by the arrival of the wavefront in unit speed, so distance and time are equivalent in this sense. So generator marked as event are process.

Let W denote the current wavefront we currently are simulating, at any time in its simulation process. In the beginning of the simulation we have $W = W(e)$ as the approximate wavefront which passes through e to compute cell c. It might be the case that the edge e is claimed by multiple waves, and therefore W is a list of generators, each claiming a portion of e. Every generator $v \in W$ defines a pair of bisectors with its neighbors in the list. If we let v be the first generator in the list, then v claims one of the endpoints of e, and its first bisector represented as the ray from v through the end point of e. We define the last generator v' of the generators similarly. Should $v = v'$ and claim the whole edge e, then there are no bisectors in the list, and the wave is the ray propagated from the generator through e. See figures 6.12, 6.13 and 6.14.

Figure 6.11: Crossing of two line segments

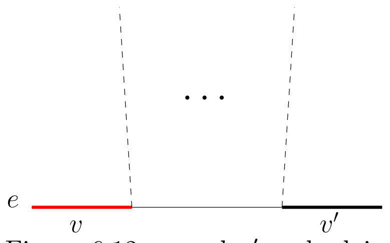


Figure 6.12: v and v' each claim the end points of e, with their bisectors being shared with other generators claiming the middle part of e, represented by the dots.

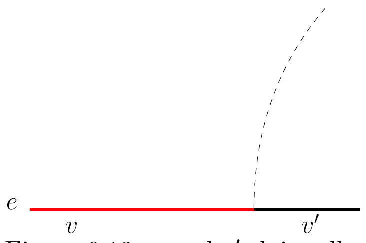


Figure 6.13: v and v' claim all of e and only have one bisector between them

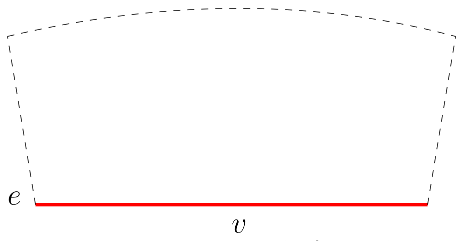


Figure 6.14: v claims all of e and there are therefore no bisectors other than the ray projected from v through e.

In order to process the bisectors event in the correct order, we will maintain the generators of W in the priority queue which was describe in section 6.2. The priority field which each node of the priority queue tree has, is assigned with its weighted distance to the point at which the two bisectors that defines v and its neighbors intersect, beyond e. An example of this is given in Figure 6.15. Here the bisectors defining v and its neighbours intersects after passing through e, and meeting at the point p. This means that v's priority value $priority(v) = d(s, v) + |\overline{vp}|$.

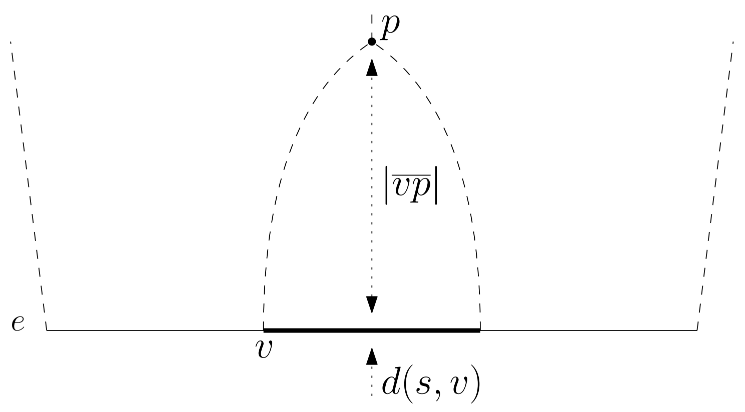


Figure 6.15: A visualization of what a generator v's priority value, $|\overline{vp}|+d(s,v)$, could be with the dashed lines being bisectors of the generators claiming e.

The processing of the approximate wavefront propagation will be done in event of increasing priority up to some maximum priority t_{stop}. The t_{stop} is calculated from the shape of the cell c, rules for this will be presented later in this section. The limit of t_{stop} is calculated from the individual values of $t_{stop}(f)$ for each transparent edge f of c. Initially the values for all f is $t_{stop}(f) = \infty$ and therefore $t_{stop} = \infty$. Also to keep track of the priorities through out the simulation, we initialize an empty set T which will be cleared after each simulation.

At each step of the simulation, the generator v with the lowest priority from the queue is processed, where one of following scenarios can happen.

1. If the event (the intersection of the v's bisectors) happens inside the cell c, then v is deleted, since it can no longer propagates the free space, and we recompute the priorities for v's neighbours. We mark v in $W(e)$ for the cell c by the marking rules explained in section 6.1.4, which also implies marking it in $O(1)$ number of neighbouring cells to c accord to the marking rules.

2. If v's event happens outside c, then we set $priority(v) = \infty$ and add v to the set T. In this case the generator list is not changed since v still has more free space outside of c to propagate.

If we where to process each bisector event of W in their initialize time order, and not update them in the first of the above cases, then the neighboring generators of v could participate in bisector event outside of cell c before all bisector events inside c were fully processed.

So far we have seen the intersection of v's bisectors if its neighbours engulfs v, as seen in Figure 6.15, in which case we would mark v by rule 3 of the generator marking rules. But it's also important to be aware that the bisector event or v could also happen if either the intersection lies on a opaque edge of if they lie on different transparent edges with an opaque edge between them. In that case rule 1 above, should still do as stated, but would use rule 4 to mark v for cell c and its neighbours. Should the intersection point p lie on a transparent edge, e.g. f, then we update our t_{stop}'s as follows:

$$\begin{aligned} t_{stop}(f) &= \min(t_{stop}(f), d(s,v) + |\overline{vp}| + |f|) \\ t_{stop} &= \min(t_{stop}, t_{stop}(f)) \end{aligned}$$

The update of $t_{stop}(f)$ is either the current $t_{stop}(f)$ or the weight as calculated the same way as in Figure 6.15 plus the length of the transparent edge f. This way we assure, that overestimate the priority since we would have swept all of f. By doing it this way, we also don't overshoot the estimate since it is still not $2 \cdot |f|$ times greater than the time at which W first comes to contact with f.

The t_{stop} priority, which is in the priority queue, is either $t_{stop} = \infty$ in which case all events inside c have been processed, or $t_{stop} < \infty$, which happens in the above update if there is a transparent edge f on the boundary of c with $t_{stop} = t_{stop}(f)$. We see that by updating $t_{stop}(f)$ this way, we ensure that all bisector events needed to produce $W(e, f)$ have indeed been processed. So now we move one to explain how to process $W(e, f)$. The wavefront $W(e, f)$ is calculated from W in the following way. First we locate the endpoints of f in W, which can be done by can picking a bisector in W, and following its neighbors outwards, there is at least on such bisector (in the case of one, this bisector claims all of f). Mark the endpoint claiming generators by rule 2. From here we split the generator list into 3 parts. Those generators whose bisectors are between the two endpoint claiming generators. Those are the one who will go through f, which after the reset of priorities will be the $W(e, f)$ wavefront, and the other two sets are those who pass left and those who pass right of f, see figurue 6.16. The simulation process continues with the left and right passing sets independently after the t_{stop} has been reset in each set to be the minimum of $t_{stop}(g)$ over the transparent edges g for that group.

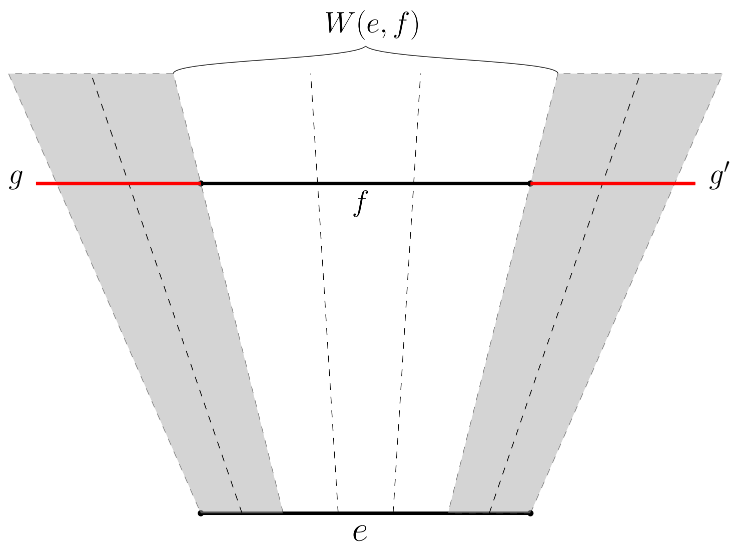


Figure 6.16: A visualisation of the three groups of bisectors that can occur in the calculation of $W(e, f)$. One going left of f, one going right of f (these two groups are marked with gray) and the middle group which is the group $W(e, f)$ consists of.

So the above is the case of $t_{stop} < \infty$. Should we stop because we reach t_{stop} and $t_{stop} = \infty$ then we split the current generator list at all the transparent edge endpoints, which will produce $W(e, f)$ for each transparent edge f, and some wavefront pieces that hit only opaque edges.

Should no transparent edges remain in some piece, then all bisectors in the piece hits an opaque edge, in which case we mark all the generators in that piece for cell c and a $O(1)$ of neighboring cell by marking rule 4. We also make the necessary marking of rule 2 and rule 3.

When finished the priority of each vertex in T is reset, based on the bisectors it defines with its neighbours in the new list. This is done to ensure each wavefront fragment $W(e, f)$ has the proper priorities. Now we have computer $W(e, f)$ and we can determine the time the wavefront first makes contact, which is $d(s, v) + |\overline{vp}|$.

6.3.2 Analysis of the wavefront propagation

The propagation algorithm calls the self balancing binary priority queue $O(1)$ times with priority and list operations per bisector event processed and $O(1)$ for each edge in the conforming subdivision (transparent edge). Each operation takes by construction of the data structure $O(\log n)$ time and space. Since the data structure by construction is fully persistent, all the modifications we do when computing a single wavefront list $W(e)$ are independent.

The main result for section 6.3.1 can be summarized in the following lemma:

Lemma 49 (Lemma 5.2 in [6])**.** *Every bisector event processed in the procedure described in section 6.3.1 either:*

1. *Lies inside cell c.*
2. *Involves a generator whose region is truncated by an opaque edge of c.*
3. *Is associated with $t_{stop}(f)$ being set to a finite value for the first time for some transparent edge f of c, or*
4. *Lies within shortest path distance $2 \cdot |f|$ of a transparent edge f of c.*

If the number of event is m, then the procedure takes $O(m \log n)$ time.

This Lemma, together with the results of chapter 4 and 5 gives us the following theorem due to Hershberger and Suri [6].

Theorem 50 (Theorem 5.3 in [6])**.** *Let $\mathcal{O}$ be a family of polygonal obstacles in the plane with pairwise disjoint interiors and a total of n vertices. Given a point s, we can construct the shortest path map from s with respect to $\mathcal{O}$ in time $O(n \log n)$ and space $O(n \log n)$.*

By this theorem we see that, one can compute the $SPM(s)$ by processing the point locations in the plane, where after one can make a shortest path query from s to any point t in the plane, which can be answered in time $O(\log n)$ due to [8]. A shortest path $\pi(s, t)$ can be computed in additional $O(h)$ time, where h is the number of edges in $\pi(s, t)$.

Chapter 7

Algorithm for shortest path with obstacle violations

This chapter is dedicated to showing the extension of the Hershberger Suri algorithm which was presented in Chapter 5 and 4 for calculating a shortest k-path map, SPM_k in time $O(k^2 \cdot n \log n)$. This extension is due to Hershberger, Kumar and Suri and presented in [5] section 3.2 and 4. This is done by extending the continuous Dijkstra method into a k-garage structure. This way we can enter each level i by going through an obstacle polygon in level $i-1$, and leave it into the next layer i. This can then be done up to k times, which is equivalent to violating k obstacles. So more precisely, when a wavefront hit an obstacle $O \in \mathcal{O}$, it claims the sub edges of the outer boundary of O, and then is reemitted into the interior of O, therefore also claiming the interior space of O. When reaching the opposite side of O from which it entered, it is reemitted into the free space free space a level higher than when entering. Therefore this "vertical" movement in the interior of O adds no time delay. So in this extension a wavefront at time t contains all points at all levels which has t distance to s.

Another change has to be made to the Hershberger Suri algorithm which involves how we identifies generators i each level. Before we had a generator g known by which vertex v it start emitting from, and the time t in which it starts to emit. When using the elevator we pass through some subedges of O, which leads us to define the generators in term of a triplet, which now also involves the sub edge on the border of O through which we enter this new level, see Figure 7.1.

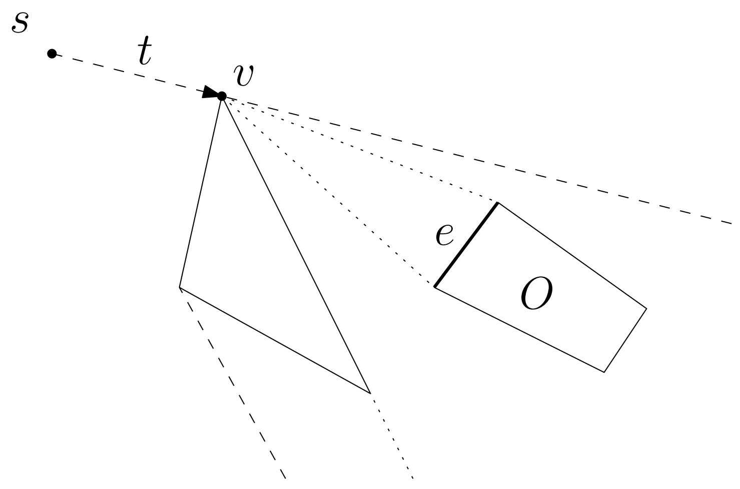


Figure 7.1: An example of a triplet generator where v starts to emit at time t, and enter obstacle O through edge e, which creates the new tiples generator (v, t, e).

We can now imaging that entering the next level through e is the same as emitting a wave from v through the interior "triangular flap", shown in Figure 7.1 with dotted lines, which is connected to e and from there enter the free space at the next floor. Algorithmically we ignore the triangular flap, and start emitting directly from e, which is another difference compared to the unmodified Hershberger Suri algorithm where we only emitted from points, and not edges. So we do the following for each edge e of the conforming sub division, as described in [5]:

1. Find all boundary sources (v, t, e) such that the well-covering region of $\mathcal{U}(f)$ which contains e.

2. Initialize *covertime*(f), which is the time at which f would be engulfed by the wavefront minimizing over all boundary sources (v, t, e) with $e \in \mathcal{U}(f)$ and for each such source considering paths from v with delay t, constrained to pass through e.

3. For each source (v, t, e) with $e \in \mathcal{U}(f)$, propagate its wavelet to e inside $\mathcal{U}(f)$.

By using the modified Hershberger Suri algorithm described above we get the following lemma, which we will use without proof.

Lemma 51 (Lemma 22 in [5]). *Given m boundary sources in a polygonal domain with n vertices, we can compute the exit claims of the sources in $O((m+n) \cdot \log(m+n))$ time and space.*

New we are ready to present the algorithm for constructing the SPM_k. The algorithm takes as a polygon $\mathcal{P}$ which encapsulate all the vertices in the plane we want in our SPM_k. The obstacles should be convex obstacles, which is a tighter condition than the unmodified Hershberger Suri algorithm, which also will work for non convex obstacles. Let M denote the set of boundary sources, which will be passed to the modified Hershberger Suri algorithm. The algorithm computes two different things, namely the $(k-1)$-visibility region V and the k =-path map $SPM_{=k}$, which together forms the SPM_k. The length from s to a point p in the plane by first locating the region in the SPM_k which contains p and then follow the k-predecessor of the region, back to s adding their length.

Algorithm 13 Construct SPM_k

```
1:  Set $M = \{s\}$
2:  call the Hershberger-Suri algorithm on $\mathcal{P}$ and computer $SPM_0$
3:  Let $V = \emptyset$
4:  **for** $i = 1$ to $k$ **do**
5:      Using the modified Hershberger-Suri algorithm propagate the sources
        in $SPM_{i-1}$ the obstacles in $\mathcal{P}$ and compute the set of boundary sources
        $M_{new}$ for $SPM_{=i}$
6:      Identify all the regions in $SPM_{=(i-1)}$ for which the predecessor is $s$.
        Observe that this is precisely the region $V' = V_{i-1} \setminus V_{i-2}$. Set $\mathcal{P}$ to be
        the new polygon domain with this region removed.
7:      **if** $V = \emptyset$ then **then**
8:          set $V = V'$
9:      **else**
10:         Merge $V$ with $V'$ at the common vertices.
11:     Set $M = M_{new}$
12:     Call the modified Hershberger-Suri algorithm to compute $SPM_{=i}$ for
        input $P$
13: Merge $SPM_{=k}$ with $v$ at the boundary of regions of $SPM_{=k}$ that have $s$
    as predecessor, i.e. $V' = V_k \setminus V_{k-1}$ to obtain $SPM_k$
```

When end this chapter with the theorem for the running time of algorithm above

Theorem 52 (Theorem 23 in [5]). *If P is a polygonal domain bounded by convex obstacles with a total of n vertices, the shortest k-path map for P with respect to a srouce point s can be computed in $O(k^2 \cdot n \cdot \log n)$ time and $O(k \cdot n \log n)$ space.*

Chapter 8

Lower bound of the "shortest path in the plane with polygonal obstacles" problem

In this chapter we will show the shortest path without violation has a $\Omega(n \log n)$ lower bound in the algebraic computation tree model, therefore affirming that the Hershberger Suri algorithm is optimal. After defining the algebraic computation tree model, we firstly introduce the element distinction problem, then we present a lower bound for that problem. Afterwards, we present the sorting problem, and we make a reduction to the element distinction problem and thereby showing, that sorting must have at least the same lower bound as element distinction and lastly showing that the shortest path in the plane without violations can be used to sort number and thereby showing that the shortest path problem has a lower bound which is at least the same.

8.1 The algebraic computation tree model

This section is based on [1]. The idea of the The Algebraic Computation Tree Model is to construct a rooted tree where each path from the root to a leaf is a membership test. Let $W \subseteq \mathbb{R}^n$ be any set. The membership problem for W is the following:

$$\text{Given } x = (x_1, \dots, x_n) \in \mathbb{R}^n \text{ determine if } x \in W \tag{8.1}$$

Formally a computation tree T for each vertex v it holds that

- If v has one son it computes one of the following computations

$$f_v := f_{v_1} \circ f_{v_2} \quad \text{or} \quad f_v := c \circ f_{v_1} \quad \text{or} \quad f_v := \sqrt{f_{v_1}} \tag{8.2}$$

  where v_i is an ancestor of v in the tree T or $f_{v_i} \in \{x_1, \dots, x_n\}$, $\circ \in \{+, -, \cdot, /\}$

- If v has two children it is a test instruction on the form

$$f_{v_1} > 0 \quad \text{or} \quad f_{v_1} \geq 0 \quad \text{or} \quad f_{v_1} = 0 \tag{8.3}$$

- If v is a leaf it is either labeled YES or NO, depending on weather the path down the tree T makes $(x_1, \ldots, x_n) \in W$

So given an input $x \in \mathbb{R}^n$ we can build a tree T and find the depth of the tree to see what the lower bound of the computation is. The important thing to note is that we are allowed to make comparisons (formula 8.3) and computations (formula 8.3).

8.2 Element distinction problem

The element distinctness problem is as follows

Definition 53. ***(Element distinctness problem:)***
Given n elements $x_1, \ldots, x_n \in \mathbb{R}$ is there a pair $x_i = x_j$ where $i \neq j$?

The following theorem is due to Michael Ben-Or 1983 [1]

Theorem 54. *Any algebraic computation tree that solves the n-element distinctness problem must have complexity of at least $\Omega(n \log n)$*

With that settled, we will move on to sorting.

8.2.1 Sorting of numbers

Definition 55 (Section 2.1 [17])**.** ***(Sorting:)***
Given a sequence of n numbers $x_1, \ldots, x_n$ find er permutation $x'_1, \ldots, x'_n$ such that $x'_1 \leq x'_2 \leq \cdots \leq x'_n$

Now we make a reduction from distinction problem (Definition 53) to the sorting problem (Definition 55). We are given $x_1, \ldots, x_n$, and would like to see if there exists a pair $x_i = x_j$ where $i \neq j$. We do this by sorting the values and afterwards make a linear scan to see if $x'_i = x'_{i+1}$ since the elements is in sorted order, two element that are the same will appear next to each other in the sorted sequence, so we will find it using this approach. Since the reduction takes $O(n)$ and the element distinction can be solved using sorting that means that sorting must have at least the same lower bound, $\Omega(n \log n)$, as element distinction.

8.2.2 Shortest path in the plane

Now we make a reduction from sorting numbers to calculating the shortest path in the plane with obstacles (See section 1.1 for the formal definition). Given numbers $x_1, \ldots, x_n \in \mathbb{N}$, and let us for simplicity say that they are all positive (i.e. $x_i > 0$ for $i = 1, \ldots, n$) we construct the obstacle as follows: Take each point x_i and construct a rectangle with the following points $(x_i, x_i^2), (x_i - 1, x_i^2), (x_i - 1, \max), (x_i, \max)$, where max is the maximum value (see Figure 8.1)

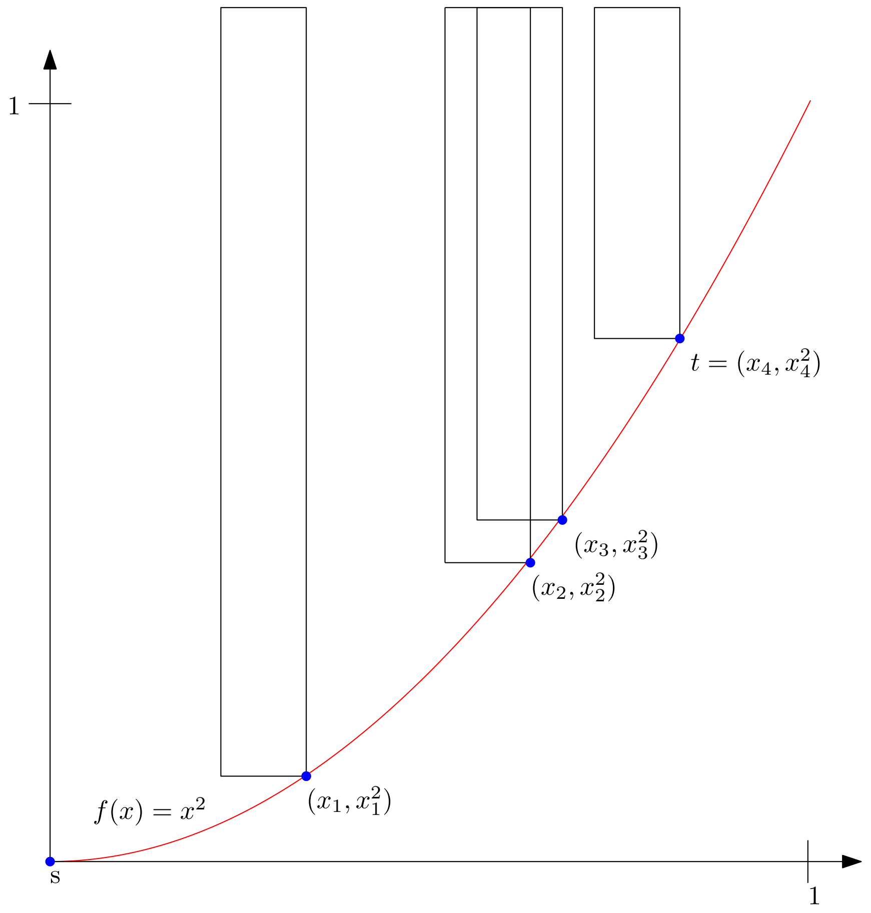


Figure 8.1: An example of the reduction from sorting numbers to shortest path

Set $s = (0, 0)$ and let $i_{max} = \arg\max_i(x_i)$ and set $t = (x_{i_{max}}, x_{i_{max}}^2)$. This gives us an instance of shortest path problem in the plane with obstacles where each rectangle has the lower right corner laying on the function $f(x) = x^2$.

This function is convex which results in every lower right corner of the rectangles to be visited on the way from s to t. Now we can run our SPM algorithm, to get a path which will be the sorted range of the numbers.

Given that the reduction (constructing the rectangles and finding t) takes $O(n)$ we can conclude that we can make a reduction from number sorting to finding the shortest path in the plane with obstacles. This mean that we have shown an $\Omega(n \log n)$ lower bound on shortest path in the plane with obstacles.

Chapter 9

Conclusion

In this thesis we have studied the problem of finding the shortest path in a plane with k polygonal obstacle violations. First we showed a naive way for computing this problem, by building a visibility graph and using Dijkstra to find the shortest path with a maximum of k allowed obstacle violation. This could be done in $O(n^3 + k \cdot n^2 + \text{Dijkstra})$, where Dijkstra runs in time $O((V + E)\log V)$. We implemented our naive algorithm and found our implementation ran in the expected time.

Next, we presented an optimal solution of solving the shortest path problem without obstacle violation, which is due to Hershberger and Suri[6], and presented an extension of this solution into one which could handle paths with obstacle violation, which is due to Hershberger, Kumar and Suri [5].

We began our discussion of the Hershberger-Suri algorithm with the construction of a conforming subdivision which builds a grid on top of the plane which gives us some strong properties for subdividing it into a shortest path math. Here we considered both theoretical results and implementation details. Next, we presented the wavefront propagation which uses the conforming subdivision to construct the shortest path map, from which we can query the shortest path between a source point s and an endpoint t in time $O(\log n)$ [8].

Next, we presented an extension of the Hershberger-Suri algorithm, with a needed modification of the Hershberger-Suri algorithm and the overall algorithm which solves the shortest path with k polygonal obstacle violations in $k^2 \cdot n \log n)$ time. We also gave an overview of the theory behind the solution to give a better understanding of the main ideas on which the algorithm is constructed.

Finally we gave a reduction of finding a shortest path without obstacle violation to sorting numbers and proved the optimally of the Hershberger-Suri algorithm.

9.1 Future work

Implementing the Hershberger Suri algorithm as described through chapters 5 and 6 would seem quite natural as the next step in this process, though the sheer complexity of it seems to make it a non trivial task. With such an implementation, it would be interesting to measure its performance and compare it to the naive $O(n^3)$ solution.

Also of interest would be an expansion of the bound of complexity for SPM_k and general lower bound of a shortest k-path computation

Appendix A

Definition for a hyperbola and bisection

A.1 Hyperbola definition

Definition 56. ***Hyperbola:***
A hyperbola is a set of points, such that for any point P of the set, the absolute difference of the distance $|PF_1|$, $|PF_2|$ to two fixed points F_1, F_2, also known as the foci, is constant, usually denoted by $2a, a > 0$[16]:

$$H = \{P \mid ||PF_1| - |PF_2|| = 2a\}$$

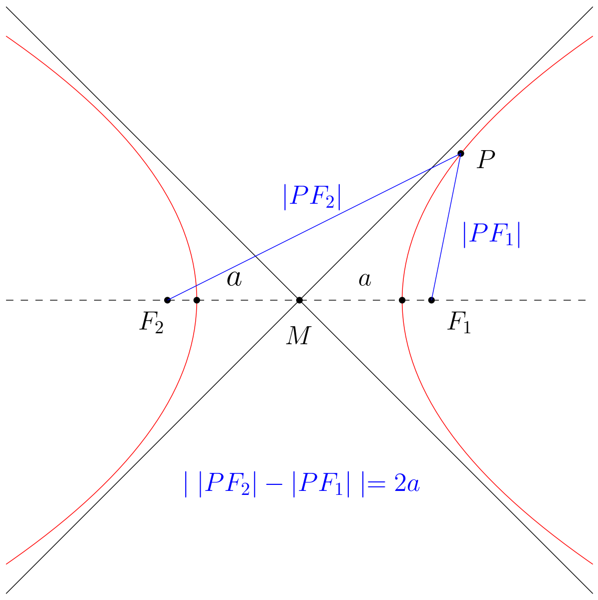


Figure A.1: Example of a hyperbola

A.2 Bisection

Definition 57. ***Bisection:***
Bisection is the division of something into two equal or congruent parts, usually by a line, which is then called a bisector. The most often considered types of bisectors are the segment bisector (a line that passes through the midpoint of a given segment) and the angle bisector (a line that passes through the apex of an angle, that divides it into two equal angles).[10]

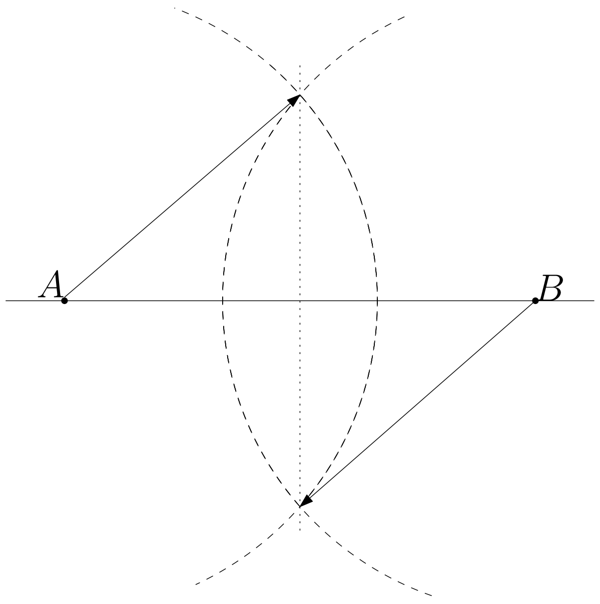


Figure A.2: Example of Segment Bisection

Appendix B

Guide to running the code

The code is located at `https://github.com/bakkegaard/Thesis`

It is compiled using the make file, and running make from the terminal. The program takes data from stdin. If the -p flag is given the program prints svg code to draw the graph to stdout, if the flag -k <number> is given k is set to <number> There a example files in the test folder. The program generateBig.py can by used to generate test files descriped in Chapter 2.3.3, it takes a number corresponding to t.

Appendix C

Proof of correctness for Chapter 5

The following section includes all the proofs for correctness of the discussed topics in the chapter. These are taken from [6], and included for completeness.

C.1 Correctness for build-subdivision

Lemma 58. *(Lemma 6.2 in [6])*
The subdivision computed by the algorithm ***build-subdivision*** *satisfies Invariants 1 and 2.*

Proof. We prove by induction that the invariants hold inside the family of quads $\mathcal{Q}(i)$, for all i. The initial family of quads $\mathcal{Q}(-2)$ clearly satisfies the two invariants. We show that no step of the algorithm **build-subdivision** ever violates these invariants. Step $2-8$ compute **growth**(S) for each equivalence class of $\mathcal{Q}(i)$, and then computes $\mathcal{Q}$. No new edges are drawn in this step.

The only edges drawn in step $9-12$ are on the boundaries of simple components. Let q be $(i-2)$-quad that is a simple component of $\mathcal{Q}(i-2)$. By definition, the single $(i-4)$-quad of $\mathcal{Q}(i-4)$ contained in q lies in its core, and thus is separated from the outer boundary of q by a gap of at least 2^{i-2} on all sides. Hence the edge already drawn in the core satisfy Invariant 1: they have length no more than 2^{i-2} (actually 2^{i-4}, except when $i=0$), and are separated from the boundary of q by a gap of at least 2^{i-2}. We draw the boundary of q in step $9-12$; since any previously drawn edges within q withing q lie in its core, the new edges satisfy Invariant 1. Invariant 2 holds vacuously.

Steps $13-19$ subdivides the region covered by each complex component S. Again, the boundary of S is separated from any components of $\mathcal{Q}(i-2)$ contained in it by a gap at least the width of an i-box. Step 18 add $(i-2)$-boxes to pad the region covered by $\mathcal{Q}(i-2)$ out to the boundaries of i-boxes. By Invariant 2, the newly drawn boxes satisfy Invariant 1 with respect to the previously drawn edges; they clearly satisfy Invariant 1 with respect to each others edges. Step 19 pacts the area between the core and the boundary of S with i-boxes, and breaks the segments incident to previously drawn cells into four pieces to guarantee Invariant 1 with respect to those cells. (The previously drawn edges on the core boundary have length 2^{i-2}, so by induction the cells incident to them have side lengths at least 2^{i-2}. It follows that the cells inside the core satisfy Invariant 1 with respect to the newly drawn segments of length 2^{i-2}.) The segments of the boundary of S are unbroken, so Invariant 2 holds at the next stage of the algorithm. This completes the proof. □

Lemma 59. *(Lemma 6.3 from [6])*
The subdivision produced by ***build-subdivision*** *has size* $O(n)$.

Proof. We show that the algorithm draws a linear number of edges altogether. The number of edges drawn in step $9-12$ is proportional to the number drawn in $13-19$, so we draw a constant number of edges in step $9-12$ for each simple component that merges to form a complex component at the next stage. The number of edges drawn in step $13-19$ for a complex component S if $O(|S'|)$, the number of $(i-2)$-quads whose growths constitute S. The key observation in proving the linear bound is that the total size of $\mathcal{Q}$ decreases every two stages by an amount proportional to the total number of quads in complex components. This fact, which we prove in Lemma 61, can be expressed as follows: If e_i edges are drawn in stage i, then

$$|\mathcal{Q}(i+2)| \le |\mathcal{Q}(i-2)| - \Theta(e_i)$$

That is, there exists an absolute constant β such that

$$\beta \cdot e_i \le |\mathcal{Q}(i-2)| - |\mathcal{Q}(i+2)|$$

If we sum this inequality over all even $i \ge 0$, the right hand side telescopes, and we obtain

$$\beta \cdot \sum_i e_i \le |\mathcal{Q}(-2)| + |\mathcal{Q}(0)| - 2$$

Since $|\mathcal{Q}(-2)| = n$, we have $\sum_i e_i \le (2n-2)/\beta$. The number of edges in the subdivision is $O(n)$. □

Lemma 60. *(Lemma 6.4 in [6])*
The subdivision produced by ***build-subdivision*** *is strongly 1-conforming and satisfies the following additional properties:*

1. *all edges of the subdivision are horizontal or vertical*
2. *each face is either a square or a square-annulus (with subdivided boundary)*
3. *each annulus has the minimum clearance property*
4. *each face has the uniform edge property, and*
5. *every point of V is contained in a square face.*

Proof. Strong 1-conformity is a consequence of Invariant 1, as we now show. Condition C1. from definition 18 is trivially true, since each point is initially enclosed by a square.

To establish well-covering, condition C2., let $I(e)$ be the union of the (at most 6) cells incident to an edge e. By Invariant 1, the distance from e to any edge outside or on the boundary of $I(e)$ is at least $|e|$. Edge e may be collinear with other edges of the two cells on whose boundary it lies. We define $\mathcal{C}(e)$ to be the set of cells incident to any of these collinear edges; $\mathcal{U}(e)$, the union of these cells, is a super-set of $I(e)$. See Figure 5.1. Because the two cells with e as a boundary edge meet only along edges collinear with e, this definition of $\mathcal{U}(e)$ means that for any edge f on or outside the boundary of $\mathcal{U}(e)$, if $I(f)$ does not contain both cells incident to e. But this implies, by Invariant 1, that e is on or outside the boundary of $I(f)$, and hence the distance from e to f is at least $|f|$. Edge e certainly lies in the interior of $\mathcal{U}(e)$ (condition W1. from definition 19).

Condition W2. follows because $\mathcal{C}(e)$ is the union of $I(e')$ for $O(1)$ edges e' collinear with e, $|I(e')| \leq 6$ for each e', and each cell has constant complexity. As noted above, the minimum distance between e and any edge f on or outside the boundary of $\mathcal{U}(e)$ is at least $\max(|e|, |f|)$, which establishes condition W3'.

Condition C3. follows from the observation that a well-covering region $\mathcal{U}(e)$ includes a vertex v of V if and only if e is an edge of the square containing v. This is because each vertex-containing square is the inner square of a square annulus in the subdivision. No edge belongs to two such squares, so condition C3 holds.

Properties (1)-(5) holds by construction. this completes the proof. □

Conforming Subdivision Theorem:
For any $\alpha \geq 1$, every set of n points in the plane admits a strong α-conforming subdivision of $O(\alpha n)$ size satisfying the following additional properties:

1. All edges of the subdivision are horizontal or vertical,
2. Each face is either a square of a square-annulus, with subdivided boundary,
3. Each annulus has the minimum clearance property,
4. Each face has the uniform edge property, and
5. Every data point is contained in the interior of a square face

Such a subdivision can be computed in time $O(\alpha n + n \log n)$.

Proof. Lemma 30, 59, 60 and 34 establish the theorem. □

C.2 Correctness for growth

Lemma 61. *(Lemma 6.5 in [6])*
Let $S \subset \mathcal{Q}(i)$ be a set of two or more i-quads such that $\mathbf{growth}(S)$ *is a complex component under the equivalence relation $\equiv_{i+2}$. Then* $|\mathbf{growth}(\mathbf{growth}(S))| \leq \kappa|S|$, *for an absolute constant $0 < \kappa < 1$.*

Proof. We show that either $|\mathbf{growth}(S)| < (3/4)|S|$, or at least half of the quads of $\mathbf{growth}(S)$ can be contained in a 2×2 array of $(i+2)$-boxes with some other quad of $\mathbf{growth}(S)$.

If $|\mathbf{growth}(S)| < (3/4)|S|$, then we are done, because the following inequality obviously holds: $|\mathbf{growth}(\mathbf{growth}(S))| \leq |\mathbf{growth}(S)| \leq (3/4)|S|$. Therefore, suppose that $|\mathbf{growth}(S)| \geq (3/4)|S|$. Then at least half the i-quads of S are not matched in step 5 of the function $\mathbf{growth}(S)$, and their growths contribute more than half of the $(i+2)$-quads of $\mathbf{growth}(S)$. Consider on such i-quad $q \in S$. Since S is non-singleton equivalence class, there exists another i-quad $q' \in S$ that overlaps q Let $\bar{q} = \mathbf{growth}(q)$ and $\bar{q}' = \mathbf{growth}(q')$. By assumption $\bar{q} \neq \bar{q}'$. The cores of $\bar{q}$ and $\bar{q}'$ both contain the overlap region $q \cap q'$, so the cores must overlap. Therefore both cores are contained within a 3×3 array of $(i+2)$-boxes, and both the $(i+2)$-quads $\bar{q}$ and $\bar{q}'$ are contained within a 5×5 array of $(i+2)$-boxes. This ensures that $\bar{q}$ and $\bar{q}'$ are joined by an edge in the graph of $\mathbf{growth}(S)$: any two $(i+2)$-quads whose bounding box is contained in a 5×5 array of $(i+2)$-boxes can be covered by an 2×2 array of $(i+4)$-boxes. Hence the number of edges in the maximal matching of $\mathbf{growth}(S)$ is $\Omega(|S|)$, which proves the inequality $|\mathbf{growth}(\mathbf{growth}(S))| \leq \kappa|S|$ for some $\kappa < 1$. □

C.3 Proofs for correctness of the conforming subdivision

The following lemma proves that one can modify a strong conforming subdivision of obstacle vertices to obtain a conforming subdivision of the free space. This subdivision of free space has the additional property that each obstacle vertex is incident to a transparent edge. It should be noted that the shortest path algorithm of [6] computes the distance from the source of the endpoints of all the transparent edges. The condition that each obstacle vertex is incident to a transparent edge ensures the distance to each obstacle vertex is correctly computed.

Lemma 62. *(Lemma 2.2 in [6])*
Every family of disjoint simple polygons with a total of n vertices admits a 2-conforming subdivision of the free space with size $O(n)$ in which each obstacle vertex is incident to a transparent edge.

Proof. Let V by the set consisting of the source vertex s and the vertices from the family of obstacles $\mathcal{O}$'s polygons. Let $\mathcal{S}$ be a strong 2-conforming subdivision constructed as described in Theorem 23. This would imply that $\mathcal{S}$ has $O(n)$ vertices, edges and cells[1]. We note that by the theorem each cell is either a square of a square annulus.

Let $\mathcal{S}_{overlay}$ be the subdivision $\mathcal{S}$ with the obstacle edges on top. This overlay will cut the plane into $O(n^2)$ cells. We will call a cell in $\mathcal{S}_{overlay}$ *interesting* if its boundary contains an obstacle vertex or a vertex of from $\mathcal{S}$. We see that each vertex from $\mathcal{O}$ and $\mathcal{S}$ we keep intact the cells in $\mathcal{S}_{overlay}$ that these vertices are incident. This implies at most four cell for each vertex of $\mathcal{S}$ and two cell for each vertex of $\mathcal{O}$.

Every edge fragment of $\mathcal{S}$ not on the boundary of an interesting cell is deleted.

For each cell, if the cell contains an obstacle vertex v, we extend edges vertically up and down from v, if the resulting edges do not enter the interior of the obstacle it self, as can be seen in Figure C.1. This cuts the cell into at

[1]we use the term cells instead of face here on out when talking about the subdivision

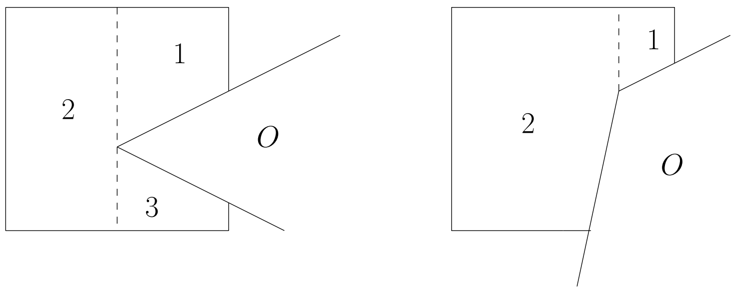


Figure C.1: Left example of v's added vertical edges splitting the cell into three piece, and right example of the cell being split into two[6].

most three convex pieces, due to the cells being derived from a square in $\mathcal{S}$. See Figure C.1.

For each such cell c in $\mathcal{S}$ that contains such a v, let δ be the length of the shortest edge on the boundary of c (recall that these can be subdivided). The newly added vertical edges are then subdivided into pieces of length at most δ, which produces $O(1)$ vertical edge fragments. This is due to the boundary of c consists of $O(1)$ edges, all with approximately same length and with the uniform edge property.

We let $\mathcal{S}'$ be the result of such subdivision of cells of $\mathcal{S}$. A result of this is that all cells are convex excepts those derived from a square-annuli.

Every nonconvexity in $\mathcal{S}_{overlay}$ is derived either from nonconvexity in $\mathcal{S}$ of $\mathcal{O}$, this is because every cell in $\mathcal{S}_{overlay}$ is derived from the intersection of a cell in $\mathcal{S}$ with a face of $\mathcal{O}$. This implies that all nonconvex cell of $\mathcal{S}_{overlay}$ are interesting cells. So any cell in $\mathcal{S}_{overlay}$ with an obstacles vertex inside its boundary is cut into convex pieces by the addition of vertical edges as was shown in Figure C.1. So the only other nonconvex vertices in $\mathcal{S}_{overlay}$ are the annulus vertices. Therefore each edge fragment that is deleted lie on the common boundary of two uninteresting face, and its deletion therefore creates no new nonconvexity.

Lets assume that a cell c of $\mathcal{S}$ has p edges on its boundary. Then for each subcell of c in $\mathcal{S}'$ which contains one of c's vertices will have size at most $2 \cdot p + O(1)$, since each convex corner of c may be cut off by an obstacle edge, adding an extra edge, and two obstacle edges may enter and exit through the same edge, leaving and obstacle vertex in the cell.

Adding vertical edges through each obstacle vertex splits a cell into at most three subcells, with at most $O(1)$ additional edges shared between them. Because each cell of $\mathcal{S}$ has constant complexity, then the same will be true for the interesting cells of $\mathcal{S}'$. From this it follows that the total complexity of the interesting cells is $O(n)$. Each uninteresting cell of $\mathcal{S}'$, that is those without a vertex of $\mathcal{S}$ or V, has at most eight edges. Them being four fragments from $\mathcal{S}$ and four from $\mathcal{O}$.

Since $\mathcal{S}'$ has $O(n)$ vertices, and each of these vertices are a vertex of an interesting cell, then by planarity, $\mathcal{S}'$ has $O(n)$ faces. Figure C.2 shows a simplified example of such a construction of $\mathcal{S}'$.

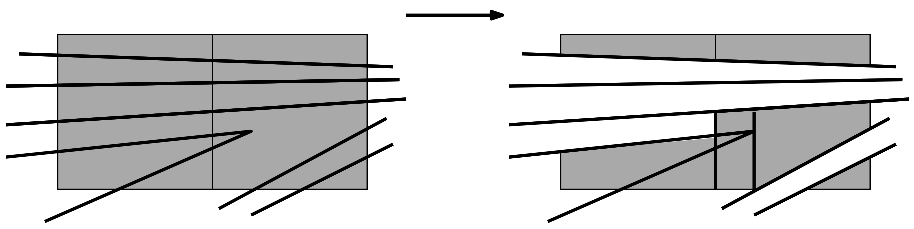


Figure C.2: Constructing a conforming subdivision of the free space, given a strong conforming subdivision for the obstacle vertices. The shaded cells on the right are interesting cells[6].

We will dedicate the remainder of the proof to show that the portion of $\mathcal{S}'$ outside all obstacles in $\mathcal{O}$ is a *conforming subdivision of the free space.* So all we to do is check the portion of $\mathcal{S}'$ outside all the obstacles in $\mathcal{O}$ behaves as defined by the 3 conditions from Definition 19

From the above it can be seen that the first condition from Definition 19 is satisfied: that is each vertex of V lies in its own square cell in $\mathcal{S}$. Each of these cells are interesting, and therefore they are either retained as they are or subdivided in $\mathcal{S}'$. Each cell of $\mathcal{S}_{overlay}$ therefore contains at most one vertex of V in its closure.

To show the next condition of Definition 19: that is all transparent edges of $\mathcal{S}'$ are well-covered, which would mean they will behave as defined in Definition 18, we consider such an edge e'. This edge e' can be one of three cases:

1. it may be a fragment of an edge $e \in \mathcal{S}$
2. or it may be all of e s.t. $e = e'$
3. or it may be a fragment of a vertical edge added incident to an obstacle vertex.

We will treat the first two cases as one, and for this purpose we will for the purpose of simplifying the notation have $\mathcal{U} = \mathcal{U}(e)$. We remind the reader that $\mathcal{U}(e) = \{c|c = \mathcal{C}(e)\}$ with $\mathcal{C}(e)$ being the a set of cells with e in its interior, and c being a cell.

In the third case where e' being a fragment of a vertical edge added incident to an obstacle vertex, e' is inside a square c of $\mathcal{S}$, we will define $\mathcal{U} = \cup_{e\in\partial c}\mathcal{U}(e)$, that is $\mathcal{U}$ is the union of $\mathcal{U}(e)$ over all edges of c's boundary. It is worth noting that the boundary of $\mathcal{U}$ is covered by edge fragments from $\mathcal{S}$, and therefore also from $\mathcal{S}_{overlay}$, but not necessarily edge fragments from $\mathcal{S}'$. Some edge fragments on the boundary of $\mathcal{U}$ may be erased in the construction of $\mathcal{S}'$. That is, $\mathcal{U}$ is union of cells of $\mathcal{S}$ and therefore also of $\mathcal{S}_{overlay}$, but not necessarily of $\mathcal{S}'$.

Region $\mathcal{U}$ satisfies conditions 1_{fs} by construction and 3_{fs} because $\mathcal{U}$ satisfies condition 3 in Definition 18 for the transparent edges of $\mathcal{S}$, and therefore for those of $\mathcal{S}_{overlay}$. However, it is not necessarily true that $\mathcal{U}$ is made up of a union of cells from $\mathcal{S}'$. This will lead to $\mathcal{U}$ being cut into a non-constant number of pieces by obstacle polygons. This implies we cannot use $\mathcal{U}$ directly as the well-covering region of e' in $\mathcal{S}'$. Instead we intersect $\mathcal{U}$ with the free space, which will partition $\mathcal{U}$ into a connected set of components $R_1, R_2, \ldots\ldots$ Exactly one of these component, e.g. R_1, contains e'.

Next, we show that each of the R_i's which are unions of cells of $\mathcal{S}_{overlay}$ is of constant size. This will bound the total complexity to be constant. We argue that for each cell c in $\mathcal{S}$, a cell c contains a number of $\mathcal{S}_{overlay}$ subcells, of which only a constant number belongs to R_i. "If two subcells of c in $\mathcal{S}_{overlay}$ both belong to R_i, then the obstacle edges separating them must have endpoints either inside $\mathcal{U}$, or contained in one or more holes of $\mathcal{U}$ if $\mathcal{U}$ is multiply connected, see Figure C.3".

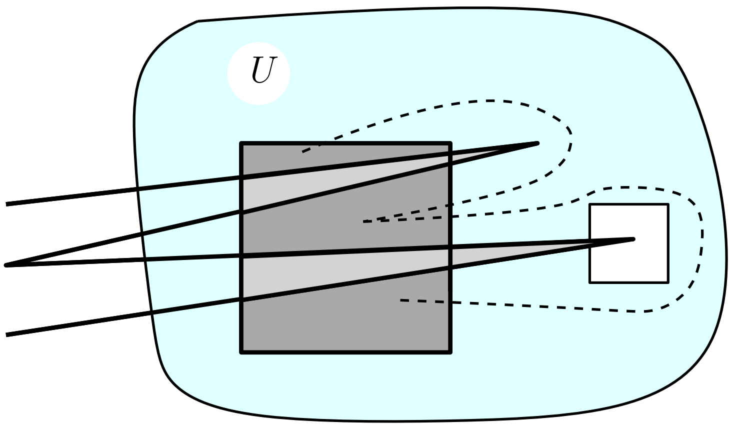


Figure C.3: A cell of $\mathcal{U}$ may be partitioned into many subcells in $\mathcal{S}_{overlay}$, but only $O(1)$ of them belong to any one R_i. [6]

Were we to traverse the boundary of R_i, one would visit the subcells of c repeatedly. Between each pair of different subcells of c, one would traverse either

1. A different hole of $\mathcal{U}$,
2. The outer boundary of $\mathcal{U}$, or
3. The unique obstacle vertex inside $\mathcal{U}$.

This is due to the fact that $\mathcal{U}$ only has $O(1)$ holes, only $O(1)$ subcells of c belong to R_i.

For any given component R_i, let $c(R_i)$ be the cells of $\mathcal{S}_{overlay}$ in R_i, we see this mean $|c(R_i)| = O(1)$. For each cell $c \in c(R_i)$, we will have a unique cell c' in $\mathcal{S}'$ such that $c \subseteq c'$. The cell c will be strict subset of c' if and only if some edge of c was erased during the construction of $\mathcal{S}'$. In case that $c \subsetneq c'$, then c' will be an uninteresting cell, and therefore have at most eight edges. This implies that both c and c' have constant complexity. If we define

$$c'(R_i) = \{c' | c' \in \mathcal{S} \text{ and } c \subseteq c' \text{ for some } c \in c(R_i)\}$$

then we have $|c'(R_i)| = O(|c(R_i)|) = O(1)$.

If $\mathcal{U}$ is nonconvex, it may be the case that some cell c' of $\mathcal{S}'$ that intersects R_i also intersects another component R_j, that is $c'(R_i) \cap c'(R_j) \neq \emptyset$, see Figure C.4.

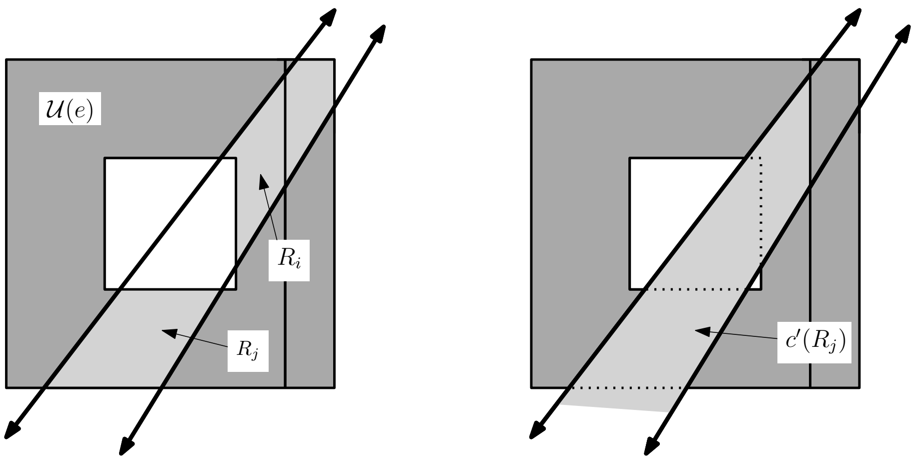


Figure C.4: R_i and R_j are disjoint components of $\mathcal{U}(e)$ in $\mathcal{S}_{overlay}$. R_i is partitioned by a vertical line inside $\mathcal{U}(e)$, so $c(R_i)$ consists of two cells; $c(R_j)$ is a single cell. $c'(R_j)$ intersects both R_i and R_j, so $R_i \sim R_j$. Note that $c'(r_j)$ may have transparent edges outside $\mathcal{U}(e)$. [6]

Let us say that two components are connected, $R_i \sim R_j$, if and only if $c'(R_i) \cap c'(R_j) \neq \emptyset$m and extend $\sim$ to an equivalence relation by transitive closure.

We define $\mathcal{U}' = \mathcal{U}(e')$, the well-covering region for e' in $\mathcal{S}'$, to be the union of $c'(R_i)$ for all R_i in the equivalence class R_1 under the $\sim$ relation. We argue that $\mathcal{U}'$ has constant complexity. Let $\overline{R}$ be the set of R_i that contain a vertex of $\mathcal{S}$ or $\mathcal{O}$. The set of cells $c'(\overline{R}) = \cup_{R_i \in \overline{R}} c'(R_i)$ has $O(1)$ total complexity. Further, if $R_i \notin \overline{R}$, then $c'(R_i)$ is a single convex cell with $O(1)$ complexity, because all transparent edges of $c(R_i)$ inside $\mathcal{U}$ have been deleted. If such a cell $c' = c'(R_i)$ does not intersect any component in $\overline{R}$, then the union of $c'(R_j)$ for all $R_j \sim R_i$ is just the single cell c'. On the other hand, if c' does intersect some $R_j \in \overline{R}$, $c' \cup c'(R_j)$ is identical to $c'(R_j)$. Because edge e' was not deleted, $R_1 \in \overline{R}$. It follows that $\mathcal{U}' \subseteq c'(\overline{R})$, and hence $\mathcal{U}'$ satisfies condition 2 of definition 22.

The definition of $\mathcal{U}(e')$ implies that every transparent edge f' on the boundary of $\mathcal{U}(e')$ is outside or on the boundary of $\mathcal{U}$. Edge f' is a subset of some edge f of $\mathcal{S}$, so the Euclidean distance from e' to f' is at least $2 \cdot \max(|e'|, |f'|)$. It follows that condition 3_{fs} holds. Condition 1_{fs} holds by construction.

As the last thing let us establish condition 3 of definition 19. A well-covering region $\mathcal{U}(e')$ in $\mathcal{S}'$ contains no obstacle vertex that lies outside the well-covering region $\mathcal{U}in\mathcal{S}$ from which $\mathcal{U}(e')$ is derived, since no edges of $\mathcal{S}$ that bound vertex-containing cells are deleted. If e' is a fragment of an edge e of $\mathcal{S}$ then its well-covering region $\mathcal{U}(e')$ in $\mathcal{S}'$ contains at most one obstacle vertex, since the same is true for $\mathcal{U} = \mathcal{U}(e)$ in $\mathcal{S}$. If e' is one of the edges added to $\mathcal{S}'$ inside a vertex-containing square, its well-covering region $\mathcal{U}$ is the union of $O(1)$ well-covering regions of $\mathcal{S}$. Each component region contains the square and its vertex, and no other vertex, hence the well-covering region of e' in $\mathcal{S}'$ also satisfies condition 3 of definition 19.

This complete the proof that $\mathcal{S}'$ is a conforming subdivision of the free space corresponding to the set of obstacles $\mathcal{O}$. □

C.3.1 Lemma for efficient construction time of conforming subdivision of the free space

The last lemma of this section shows that the conforming subdivision described above can be computed in $O(n \log n)$ time.

Lemma 63. *(Lemma 2.3 in [6])*
The linear-size conforming subdivision of free space described in Lemma 62 can be built in time $O(n \log n)$.

Proof. We start with a string 2-conforming subdivision $\mathcal{S}$ of the obstacle vertices; $\mathcal{S}$ is computed in $O(n \log n)$ time, by theorem 23. In $O(n \log n)$ additional time, we build a point-location data-structure for the obstacle polygons, so that given a query point q, we can in $O(\log n)$ times find the obstacle edge immediately to the left, right, above, or below q [3][9]. The edges of $\mathcal{S}'$ are obstacle edges, transparent edges on the boundary of kept cell, and transparent edges incident to obstacle vertices. To identify the second kind of edges, we trace the boundary of each kept cell separately.

Each kept cell is contained in a single cell of $\mathcal{S}$ and has at least one vertex on its boundary, so we trace starting from each vertex. Tracing along an obstacle edge is easy, since the next transparent edge intersected is one of the $O(1)$ edges on the boundary of the current cell in $\mathcal{S}$. We use the point-location structure to trace along transparent edges: the next cell vertex is either a vertex of $\mathcal{S}$, or it is the first obstacle point hit by the ray that the current point and edge define. This tracing takes $O(n \log n)$ time altogether. The third kind of edges can be computed in $O(n)$ total time by local operations in each cell containing and obstacle vertex. To stitch the three kinds of edges into a single adjacency structure $\mathcal{S}'$, we use an $O(n \log n)$ time plane sweep algorithm [12]. □

This complete the chapter of the conforming subdivision.

Appendix D

Survival guide

n	number of vertices
k	number of violation
l_i, l	edge segments
e, e_i, f, f_i	edges
i, j	counting variables
p, p_i, q, q_i, a, b, r	points
s	stars point, source
t	end point
V	set of vertices, in the context of $G(E, V)$, else all vertices in the plane including s
E	set of edges
G	Graph
$p_{i.x}, q_{i.x}$	x coordinate of points
$p_{i.y}, q_{i.y}$	4 coordinate of points
v_i	vectors or vertices
A, B, C	Figures, like triangles and square, or corners of triangles
L_i	lines
v_π	predecessor
s_d, v_d	upper bound of weight of shortest paths
$w(\cdot,\cdot), w(\cdot)$	weight function with one or two inputs
Q	min priority queue
$\mathcal{S}$	straight line subdivision
$\mathcal{K}$	$2\lceil\frac{1}{2}\log_2(\lvert e\rvert/6)\rceil$

α	well covering parameter
$C(e)$ $\mathcal{C}_i(e_i)$	set of cells with e in its interior set of cells of $\mathcal{S}_i$ whose union $\mathcal{U}_\alpha(e_\alpha)$ the the well covering region e_α
$\mathcal{U}(e)$	$\{c \mid c \in \mathcal{C}(e)\}$
c	cell
Δ	side length of figure
x, y	$= j \cdot 2^i$ coordinate of i-quad
b	i-box
$\mathcal{Q}(i)$	i-quads at stage i
$S, S', S_j(i)$	equivalence class of $\mathcal{Q}(i)$
$\equiv_i$	transitive equivalence relation of overlap
$input(e)$	edges who's wavefront is used for computing distance to e from $\mathcal{U}(e)$'s boundary
$output(e)$	$input(e) \cup \{f \mid e \in input(f)\}$
∂	boundary
V_S	points in the cores S quads
$\overline{pq}$	line segment between point p and q
$covertime(e)$	time where e is fully covered
w, w'	wavelet
O, O_i	obstacle
$\mathcal{O}$	the set of obstacles in the plane
g, g'	generators
$\pi(p)$	set of shortest path from s to p
$\pi_k(p)$	the shortest path from s to p
SPM	shortest path map
SPM_k	shortest k-path map
T, T_i	minimum spanning tree or subset of edges
T'	joined tree of T_1 and T_2
T_x	one of $\{T_1, T_2\}$
$\mathcal{N}$	set of minimum spanning trees
$G(i)$	graph with set of vertices V with edges weight of less than $6 \cdot 2^i$
$\lvert \cdot \rvert$	length of edges of size of set
R_1	$\bigcup_{q \in S'} \{$the core of $growth(q)\}$
R_2	$\bigcup_{q \in S'} \{$the region covered by $q\}$
M	matching in graph
q_u, q_v	i-quad q containing vertex v or u in its core
$MSF(\cdot)$	minimum spanning forest at stage i
$d(\cdot, \cdot)$	distance function
$\tilde{d}(s, e)$	$\min(d(s,a), d(s,b))$
$W(e)$	approximate wavefront passing through edge e
$W(f, e)$	$\{w(f) \mid f \in input(e)\}$
$W(f', e)$	topologically different than $W(f, e)$
$S(e)$	S-face which segment intersects e
S-face	piece of active region

Appendix E

Tables

n	time (in milliseconds)
2	0
6	0
18	0
38	1
66	9
102	24
146	50
198	112
258	235
326	466
402	848
486	1481
578	2462
678	3939
786	6083
902	9271
1026	13357
1158	19278
1298	27520
1446	37018
1602	50129
1766	66930
1938	88378
2118	114971
2306	148055
2502	188859
2706	238534
2918	298675
3138	371052
3366	457771
3602	559747
3846	680632
4098	822848
4358	989762
4626	1182116
4902	1404667
5186	1663503
5478	1958169
5778	2296425
6086	2682430
6402	3120744
6726	3616376

n	crossing	visibility	Dijkstra
2	0	0	0
6	0	0	0
18	0	0	0
38	1	0	0
66	8	0	1
102	22	0	2
146	45	1	4
198	104	2	6
258	222	3	10
326	447	4	15
402	821	6	21
486	1444	9	28
578	2415	11	36
678	3878	15	46
786	6006	18	59
902	9178	22	71
1026	13244	28	85
1158	19144	33	101
1298	27359	40	121
1446	36821	47	150
1602	49895	56	178
1766	66658	65	207
1938	88071	75	232
2118	114616	87	268
2306	147650	100	305
2502	188395	115	349
2706	237995	131	408
2918	298074	148	453
3138	370383	167	502
3366	457025	190	556
3602	558914	213	620
3846	679704	238	690
4098	821822	264	762
4358	988634	298	830
4626	1180886	328	902
4902	1403311	363	993
5186	1662007	400	1096
5478	1956547	440	1182
5778	2294661	481	1283
6086	2680522	530	1378
6402	3118674	574	1496
6726	3614138	628	1610

Table E.2: Data from graph 2

n	time in miliseconds dividet by n^3
2	0
6	0
18	0
38	1.82242309374544E-05
66	3.13047833708991E-05
102	2.26157360291291E-05
146	1.60661359272217E-05
198	1.44285421297971E-05
258	1.36838638480003E-05
326	1.34503354733029E-05
402	1.3053221060855E-05
486	1.29016795495295E-05
578	1.27498340864401E-05
678	1.26385397648696E-05
786	1.25270894447943E-05
902	1.26330137388433E-05
1026	1.2367070702209E-05
1158	1.24147019560424E-05
1298	1.25841634982224E-05
1446	1.2243570102851E-05
1602	1.21927454179994E-05
1766	1.2152027041557E-05
1938	1.21417937429069E-05
2118	1.21006985351175E-05
2306	1.20738331437455E-05
2502	1.20580136097767E-05
2706	1.20383485707373E-05
2918	1.20210667777948E-05
3138	1.20081459851104E-05
3366	1.20034459584254E-05
3602	1.19773474781993E-05
3846	1.19642236814143E-05
4098	1.19564913967957E-05
4358	1.19582904652676E-05
4626	1.19410690659842E-05
4902	1.19248646628465E-05
5186	1.19268580687987E-05
5478	1.19119836093996E-05
5778	1.19047328401354E-05
6086	1.1899605218836E-05
6402	1.18935399250374E-05
6726	1.18851040850164E-05

Table E.3: Data from graph 3

k	time in miliseconds
0	184927
1	185173
2	185082
3	185218
4	184888
5	184901
6	185203
7	184971
8	185074
9	185320
10	185211
11	185254
12	185479
13	185317
14	185491
15	185355
16	185504
17	185928
18	185985
19	185669
20	185842
21	185879
22	185980
23	186128
24	186250
25	186189

Table E.4: Data from graph 4

Bibliography

[1] Michael Ben-Or. Lower bounds for algebraic computation trees (preliminary report). In David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo, and Joel I. Seiferas, editors, *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*, pages 80–86. ACM, 1983.

[2] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.

[3] Herbert Edelsbrunner, Leonidas J. Guibas, and Jorge Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15(2):317–340, 1986.

[4] Subir Kumar Ghosh and David M. Mount. An output-sensitive algorithm for computing visibility graphs. *SIAM J. Comput.*, 20(5):888–910, 1991.

[5] John Hershberger, Neeraj Kumar, and Subhash Suri. Shortest paths in the plane with obstacle violations (all of it). In Kirk Pruhs and Christian Sohler, editors, *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria*, volume 87 of *LIPIcs*, pages 49:1–49:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[6] John Hershberger and Subhash Suri. An optimal algorithm for euclidean shortest paths in the plane (all of it). *SIAM J. Comput.*, 28(6):2215–2256, 1999.

[7] Mohamed A. Khamsi and William A. Kirk. *An Introduction to Metric Spaces and Fixed Point Theory.* Wiley-interscience Publication, 2001.

[8] David G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.

[9] David G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.

[10] Serge Lang and Gene Murrow. *Geometry.* Springer, 2000.

[11] D. T. Lee. *Proximity and reachability in the plane.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 1978.

[12] Marc van Kreveld Mark de Berg, Otfried Cheong and Mark Overmars. *Computational Geometry.* Springer, 2008.

[13] Joseph S. B. Mitchell. A new algorithm for shortest paths among obstacles in the plane. *Ann. Math. Artif. Intell.*, 3(1):83–105, 1991.

[14] Joseph S. B. Mitchell. Shortest paths among obstacles in the plane. *Int. J. Comput. Geometry Appl.*, 6(3):309–332, 1996.

[15] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry - An Introduction.* Texts and Monographs in Computer Science. Springer, 1985.

[16] James Stewart. *Calculus.* Thomson, 2006.

[17] Ronald L. Rivest Thomas H. Cormen, Charles E. LeiserSon and Clifford Stein. *Introduction to Algorithms.* The MIT press, 2009.

[18] Emo Welzl. Constructing the visibility graph for n-line segments in $o(n^2)$ time. *Inf. Process. Lett.*, 20(4):167–171, 1985.

[19] Wikipedia. Matching (graph theory), 2018. [Online; accessed 12-June-2018.

The thesis is mainly build on [6] and [5] which are read and understood in their entirety. The rest of the literature have been used "secondary litterateur" and we have therefore only read their abstract, or main results.