Single Source Shortest Paths in Simple Polygons

Mateus Barros Rodrigues

Thesis presented
to the
Institute of Mathematics and Statistics
of the
University of São Paulo
in fulfillment
of the requirements of the degree
of
Master of Science

Program: Computer Science

Advisor: Prof. Dr. Carlos Eduardo Ferreira

During the course of this project the author was supported by CAPES

São Paulo, July 2019

Single Source Shortest Paths in Simple Polygons

This version of the thesis was revised based on the suggestions and corrections made by the Thesis Comittee during the original thesis' defense, held on the 11th of July of 2019. A copy of the original version is available at the Institute of Mathematics and Statistics of the University of São Paulo.

Thesis Committee:

- Prof. Dr. Carlos Eduardo Ferreira - IME - USP
- Prof. Dr. Álvaro Junio Pereira Franco - UFSC
- Prof. Dr. Luiz Henrique de Figueiredo - IMPA

Abstract

A classic problem Computational Geometry is finding all euclidean shortest paths in a simple polygon from a given source vertex to every other vertex in the boundary. In this text, we give a detailed description of the Visibility Graph and Shortest Path Tree structures that solve this problem and also present the Shortest Path Map structure that extends the solution to shortest paths to every point inside the polygon.

Resumo

Um problema clássico em Geometria Computacional é: encontrar todos os caminhos mínimos euclidianos dentro de um polígono simples a partir de um dado vértice fonte para todos os outros vértices da borda. Neste texto, apresentamos detalhadamente as estruturas de Grafo de Visibilidade e Árvore de Caminhos Mínimos que resolvem este problema e descrevemos também a estrutura Mapa de Caminhos Mínimos que estende a solução para todos os pontos contidos dentro do polígono.

Contents

Chapter 1

Introduction

Computational Geometry, also known as *algorithmic geometry*, is a branch of the field of algorithm design and analysis that deals with geometric objects as discrete entities. The term has first appeared in this context in 1975 [PS85]. Its development as a line of research can be credited not only to the elegance of its algorithms but also to its vast applications domains, such as computer graphics, geographic information systems, robotics and computer vision.

One of the fundamental problems in computational geometry is finding shortest paths that avoid obstacles in the plane. There are many applications of this problem in robotics, for example, in motion planning [Sha89]. In this text we will focus on finding *single source shortest paths*. This problem can be modeled as follows: Given a simple polygon P and a vertex s on the boundary of P, find all the euclidean shortest paths, that is the paths that have minimum total length under the euclidean norm, from s to every other vertex of P that are entirely contained within P (see figure 1.1). The algorithm we describe is due to Guibas et al. [Gui+87], who improved on the technique presented by Lee and Preparata [LP84]. We also extend this problem to the one of finding the shortest paths from s to any point inside of P. We do so by presenting the *shortest path map* data structure, together with the hierarchical subdivision search presented by Kirkpatrick [Kir83]. Our goal in this text is to provide a succinct yet detailed self-contained description of data structures related to this problem that can be found in the literature, that is, we provide no new results or algorithms, but we focus on explaining the ones that already available.

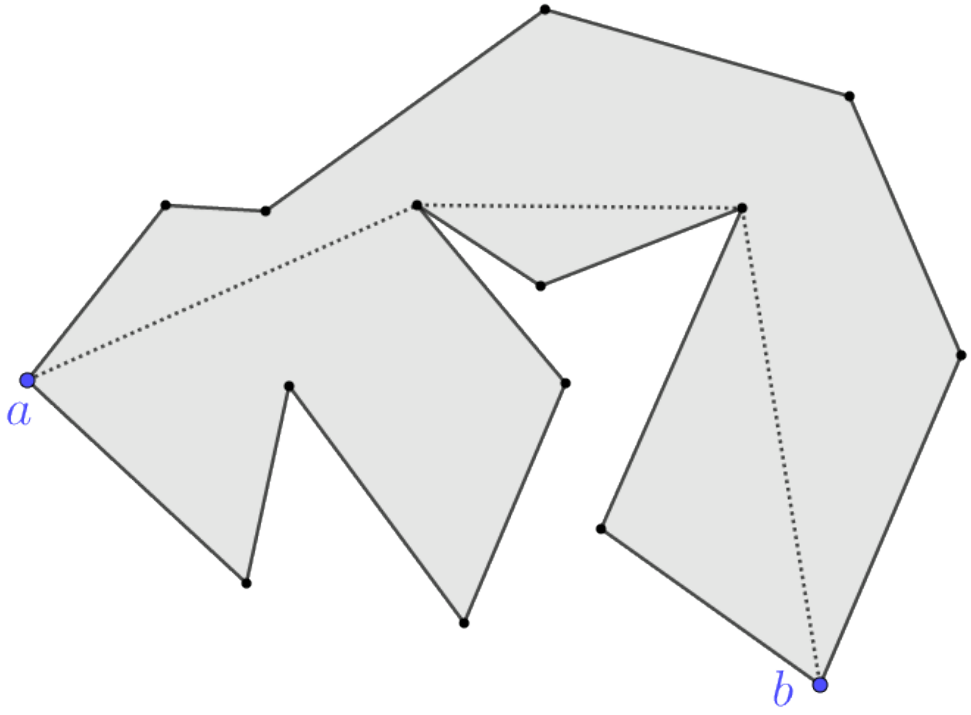


Figure 1.1: *Shortest path between a in b inside this polygon.*

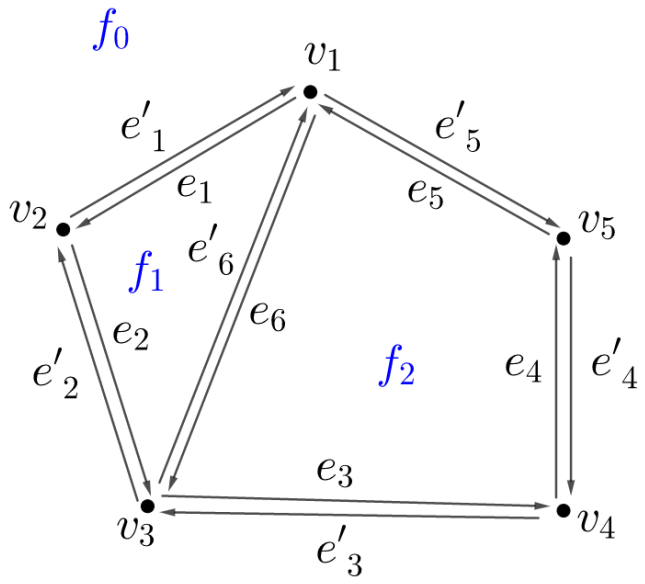
We begin by describing the *doubly-connected edge list* structure in chapter 2, which is used to represent planar graphs and simple polygons. We then present the concept of *triangulation* in chapter 3. We then describe the *visibility graph* data structure for finding the shortest paths between all pairs of vertices in a simple polygon in $O(n^2 \log n)$ construction and query time in chapter 4, which we can use as validation for the remaining algorithms, since it is very simple to understand. In chapter 5, we describe the *single source shortest path tree* data structure for finding the single source shortest path from a fixed vertex to every other on the boundary of the polygon in $O(n)$ construction and query time. And finally, in chapter 6 we show the *shortest path map* data structure for finding the shortest path from a fixed vertex to every point inside the polygon in $O(n)$ construction time and $O(\log n)$ query time.

Chapter 2

Doubly-Connected Edge List

In this chapter we will describe the *Doubly-Connected Edge List* (DCEL) data structure. It is also known as the *half-edge data structure* and can be used to represent the embedding of a planar graph in the plane. This data structure was originally presented by Muller and Preparata [MP78], and it allows us to easily traverse the faces of the graph, visit all edges surrounding a given vertex, among other interesting operations.

The DCEL structure is composed of three main elements: vertices, faces and half-edges. Suppose we are given a planar graph G. The DCEL of G, DCEL(G), is defined as follows: Every edge $uv \in E(G)$ has two corresponding directed half-edges $\overrightarrow{uv}$ and $\overrightarrow{vu}$; we say these half-edges are *twins* of one another. In this representation every half-edge belongs to exactly one face of G. Every face $f \in F(G)$ is composed of half-edges in counter-clockwise orientation, except for the exterior face which has clockwise orientation; every face has a pointer **edge** to an half-edge that forms it. Every half-edge has a pointer **face** to the face it belongs to, a pointer **twin** to its twin half-edge, a pointer **vertex** to the vertex it is incident to, and pointers **previous** and **next** which correspond to the previous and next half-edges following the orientation of the face it belongs to. This means that every face is represented by a doubly linked list of half-edges, which can be traversed by following the pointers in the half-edge it points to. Finally, every vertex $v \in V(G)$ has a pointer **edge** to a half-edge that points to it. Figure 2.1 shows the DCEL of a polygon composed of 5 vertices, 6 edges and 3 faces.



face	edge
f_0	e'_1
f_1	e_1
f_2	e_3

vertex	edge
v_1	e_5
v_2	e_1
v_3	e_2
v_4	e_3
v_5	e_4

edge	vertex	face	twin	next	previous
e_1	v_2	f_1	e'_1	e_2	e'_6
e'_1	v_1	f_0	e_1	e'_5	e'_2
e_2	v_3	f_1	e'_2	e'_6	e_1
e'_2	v_2	f_0	e_2	e'_1	e'_3
e_3	v_4	f_2	e'_3	e_4	e_6
e'_3	v_3	f_0	e_3	e'_2	e'_4
e_4	v_5	f_2	e'_4	e_5	e_3
e'_4	v_4	f_0	e_4	e'_3	e'_5
e_5	v_1	f_2	e'_5	e_6	e_4
e'_5	v_5	f_0	e_5	e'_4	e'
e_6	v_3	f_2	e'_6	e_3	e_5
e'_6	v_1	f_1	e_6	e_1	e_2

Figure 2.1: *Example of a DCEL of a certain planar graph and its corresponding pointers.*

The algorithms we will describe in the next chapters are all restricted to *simple polygons*. A simple polygon is a polygon composed by only straight non-intersecting edges, except at the vertices, which is usually represented as a circular list of vertices $[v_1,\ v_2, \ldots,\ v_n]$, where $\overline{v_i v_{i+1}}$, $v_{n+1} = v_1$, is an edge of the polygon. The vertices in a simple polygon all have degree 2. A simple polygon can be seen as an embedding of a planar graph in the

plane, so we can use the DCEL structure to represent it. Storing a polygon in a DCEL allows us to perform many modifications in the structure of the polygon very easily, namely, adding and deleting vertices and edges to the polygon. By using the **addVertex** and the **addEdge** operations we can represent every connected planar graph on at least two vertices, by starting from an embedding of the K_2 graph (composed of two vertices connected by an edge). We will now explain how to perform these operations.

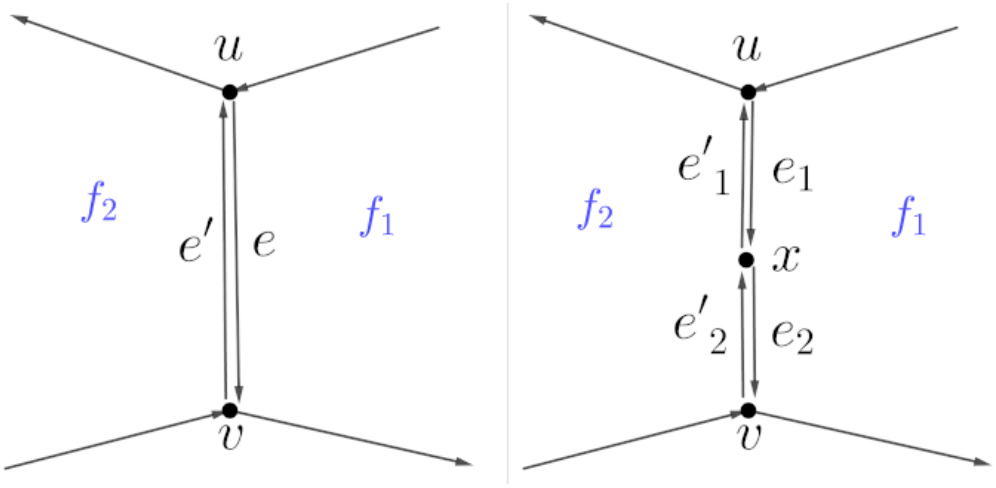


Figure 2.2: *Adding a vertex x to a certain edge uv in a DCEL.*

Whenever we add a vertex x to an edge $e = uv$ of G, we are dividing e into two new edges $e_1 = ux$ and $e_2 = xv$. Since every edge of G is represented by two half-edges in DCEL(G), we will end up with four half-edges. If uv (or its twin) was the edge that was pointed by the face it belongs to, we also need to update that pointer to one of the new edges created (see figure 2.2).

This procedure is described in algorithm 1:

Algorithm 1 Receives a vertex x and half-edge $e = \overrightarrow{uv}$ and inserts the vertex x to both e and its twin.

function ADDVERTEX(x, e):
    Let $e_1 = \overrightarrow{ux}$ and $e_2 = \overrightarrow{xv}$ be two new half-edges and $e'_1 = \overrightarrow{xu}$ and $e'_2 = \overrightarrow{vx}$ be their respective twin half-edges.
    $f_1 \leftarrow \text{face}(e)$
    $f_2 \leftarrow \text{face}(\text{twin}(e))$
    $\text{edge}(x) \leftarrow e_1$
    $\text{next}(e_1) \leftarrow e_2$
    $\text{previous}(e_1) \leftarrow \text{previous}(e)$
    $\text{face}(e_1) \leftarrow f_1$
    $\text{edge}(v) \leftarrow e_2$
    $\text{next}(e_2) \leftarrow \text{next}(e)$
    $\text{previous}(e_2) \leftarrow e_1$
    $\text{face}(e_2) \leftarrow f_1$
    $\text{next}(\text{previous}(e)) \leftarrow e_1$
    $\text{previous}(\text{next}(e)) \leftarrow e_2$
    $\text{next}(e'_1) \leftarrow \text{next}(\text{twin}(e))$
    $\text{previous}(e'_1) \leftarrow e'_2$
    $\text{face}(e'_1) \leftarrow f_2$
    $\text{next}(e'_2) \leftarrow e'_1$
    $\text{previous}(e'_2) \leftarrow \text{previous}(\text{twin}(e))$
    $\text{face}(e'_2) \leftarrow f_2$
    $\text{edge}(f_1) \leftarrow e_1$
    $\text{edge}(f_2) \leftarrow e'_1$
    delete e and $\text{twin}(e)$

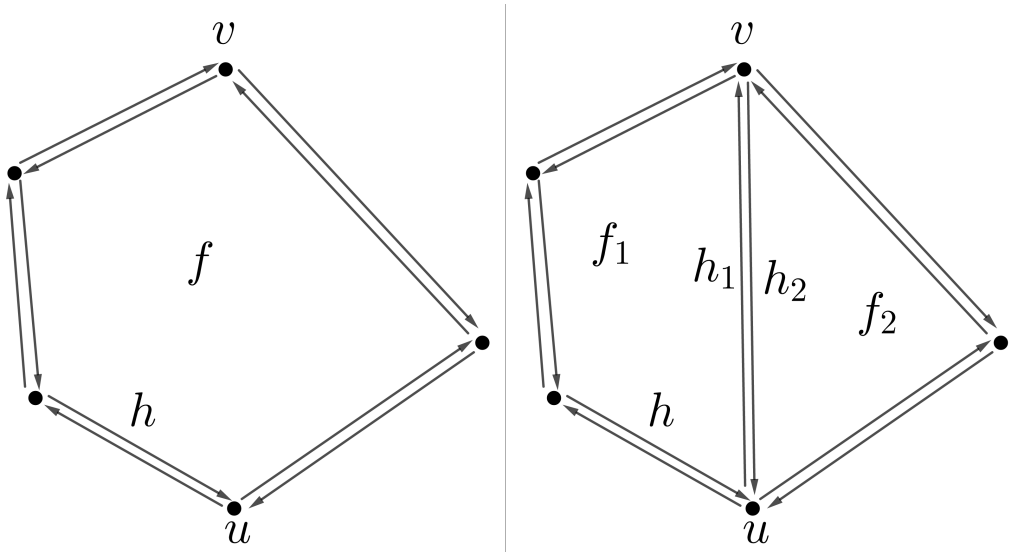


Figure 2.3: *Adding an edge uv to a certain face f in a DCEL.*

Whenever we add a new edge uv linking two vertices u and v in a certain face f in G, we are dividing the face f into two new faces f_1 and f_2. So we have to update the pointers of u and v and then we need to update the face pointer of the edges in f to the corresponding new faces (see figure 2.3). This is shown in Algorithm 2:

Algorithm 2 Receives a face f and two vertice u and v and divides f into two new face by adding the edge $\overline{uv}$.

```
 1: function ADDEDGE(f, u, v):
 2:     Let f_1 and f_2 be new faces.
 3:     Let h be an half-edge in f such that vertex(h) = u.
 4:     Let h_1 and h_2 be new half-edges.
 5:     edge(f_1) ← h_1
 6:     edge(f_2) ← h_2
 7:     twin(h_1) ← h_2
 8:     twin(h_2) ← h_1
 9:     vertex(h_1) ← v
10:     vertex(h_2) ← u
11:     next(h_2) ← next(h)
12:     previous(next(h_2)) ← h_2
13:     previous(h_1) ← h
14:     next(h) ← h_1
15:     i ← h_2
16:     while vertex(i) ≠ v do
17:         face(i) ← f_2
18:         i ← next(i)
19:     next(h_1) ← next(i)
20:     previous(next(h_1)) ← h_1
21:     next(i) ← h_2
22:     previous(h_2) ← i
23:     i ← h_1
24:     while vertex(i) ≠ u do
25:         face(i) ← f_1
26:         i ← next(i)
27:     delete f
```

To delete a vertex or an edge, one can simply apply the inverse of the operations we have described above. Joining a pair of consecutive half-edges into one or combining faces of the DCEL, respectively.

Chapter 3

Triangulation

3.1 Introduction

Often when dealing with polygons the need arises to partition them in simpler forms. The most common way to do this is with a *triangulation*. A triangulation will be required in the preprocessing step of the structure described in chapter 4. A triangulation of a polygon P is a partition of the inside area of P into triangles, particularly, we are interested in achieving this using *diagonals* (see figure 3.1). A diagonal of P is a line segment between two nonconsecutive vertices that does not intersect the exterior of P. We say that two diagonals are *non-crossing* if they share no interior points. If we add as many non-crossing diagonals as possible to a polygon P, we achieve a triangulation of P. The results shown in this section and the next are taken from a book by O'Rourke [ORo98].

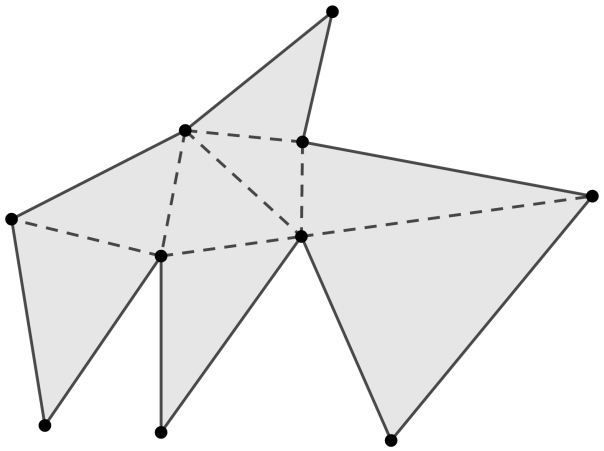


Figure 3.1: *Example of a polygon and a possible triangulation of using diagonals.*

Lemma 3.1.1. *Every polygon must have at least one strictly convex vertex.*

Proof: Suppose the vertices of P are given following a counter-clockwise ordering. While traversing around the boundary of P, when passing through a vertex with interior angle strictly less than π, that is, a strictly convex vertex, we would then make a left turn, when passing through a reflex vertex we would make a right turn. As we travel around the boundary, the interior of P is always to our left. Let L be a horizontal line passing through the lowest y-coordinate vertex v of P; if there are several lowest, pick the one with greatest x-coordinate. The interior of P must lie above L, and the edge that leaves v must lie above L. This implies that we must make a left turn when passing through v and therefore v is a strictly convex vertex (see figure 3.2).□

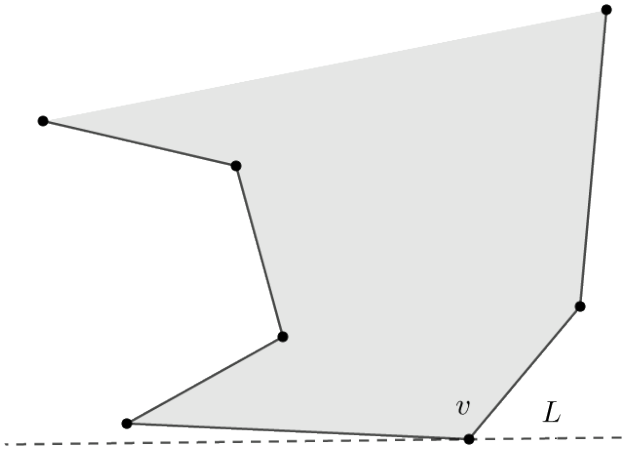


Figure 3.2: *Example where v is a strictly convex vertex.*

Lemma 3.1.2 (Meisters)**.** *Every polygon with $n \geq 4$ vertices has a diagonal.*

Proof: Let v be a strictly convex vertex, whose existence is guaranteed by the lemma above. Let a, b be the vertices of P adjacent to v. If ab is a diagonal, we are finished. So suppose ab is not a diagonal. This means that either ab lies outside of P or ab intersects the boundary of P. In either case, since there are at least 4 vertices in P, there is at least one vertex of P inside the region bounded by Δavb (the triangle with vertices a, v and b). Imagine we take a line L parallel to ab and move it from v towards ab. Let x be the first vertex encountered by L. We claim that vx is a diagonal. This follows

from the fact that the intersection of Δavb and the half-plane bounded by L passing through x does not contain any other vertex of P. Therefore vx cannot intersect any other vertex other than v and x and thus vx is a diagonal (see figure 3.3). $\square$

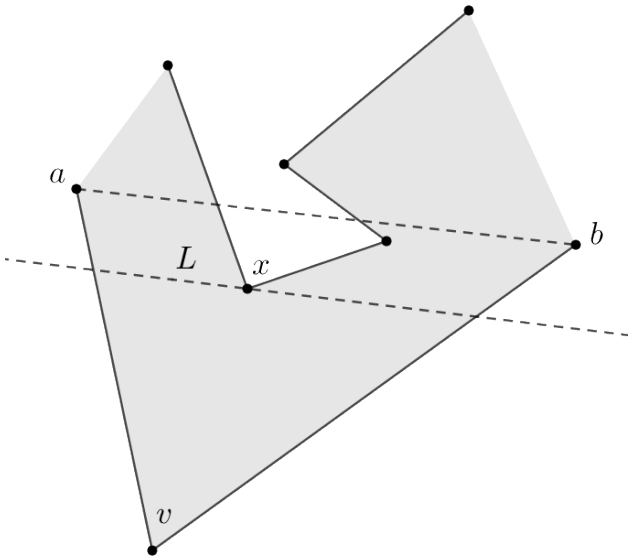


Figure 3.3: *Example where vx is a possible diagonal.*

Theorem 3.1.3. *Every triangulation of a polygon P with n vertices uses $n-3$ diagonals and consists of $n-2$ triangles.*

Proof: We shall prove this by induction. Both claims are trivially true for $n = 3$. Let $n \geq 4$. Divide P into two subpolygons P_1 and P_2 by adding a diagonal ab. Let the two subpolygons have n_1 and n_2 vertices respectively. $n_1 + n_2 = n + 2$, since a and b are each counted twice. Applying the induction hypothesis on each subpolygon we get that there are a total of $(n_1 - 3) + (n_2 - 3) + 1 = n - 3$ diagonals and $(n_1 - 2) + (n_2 - 2) = n - 2$ triangles. $\square$

Let $\mathcal{T}$ be the triangulation of the polygon P, represented as a planar graph embedded in the plane. The geometric dual of $\mathcal{T}$, denoted $\mathcal{T}^D$, is a planar graph where the set of faces of $\mathcal{T}$ form its vertex set and for every two

faces in $\mathcal{T}$ that share an edge, there is a corresponding edge in $\mathcal{T}^D$ between them. By purposely leaving the exterior face out of our representation, we are left with a planar tree over all interior faces of $\mathcal{T}$, since a cycle would mean there are internal vertices that do not belong to the boundary of P. Thus, for every vertex v of $\mathcal{T}$, there is an unique minimal path π in the dual from a triangle that contains v to one of the triangles that contains another vertex s, in $\mathcal{T}^{\mathcal{D}}$ (see figure 3.4).

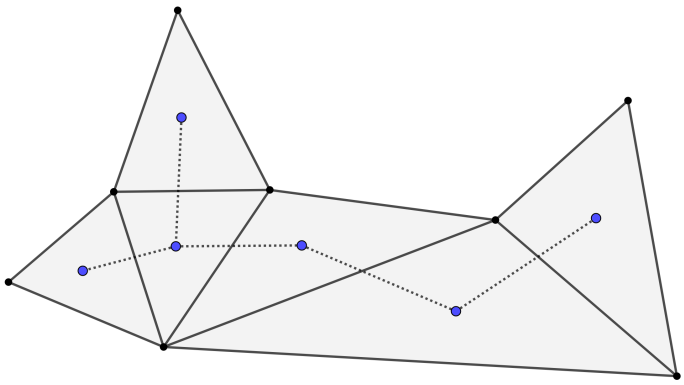


Figure 3.4: *Example of a triangulated polygon and its respective geometric dual.*

Theorem 3.1.3 gives a quadratic algorithm for triangulating an arbitrary polygon. For a long time there was the question of whether it was possible to triangulate an arbitrary simple polygon without holes with n vertice faster than $O(n \log n)$ time (there is a $\Omega(n \log n)$ lower bound for simple polygons with holes). This barrier was finally broke in [TJ88] by showing a $O(n \log \log n)$ algorithm (Their algorithm was later simplified in [KKT90]). This result was followed by several randomized Las Vegas approaches [CCT91; CTV89; Sei91], which means the output is correct but the running time is probabilistic, achieving expected $O(n \log^* n)$ time, thus being indistinguishable from linear time in practice. Finally, in 1990 there was a deterministic $O(n)$ algorithm described in [Cha91], which is considered to be too complicated and has a very large constant factor, making it not ideal for real-life implementations, in fact, none currently exist. We will focus our description on the simpler $O(n^2)$ ear-clipping method (Section 3.2) and the $O(n \log n)$ method by monotone decomposition (Section 3.3). Both

of these methods triangulate the polygon by adding new diagonals, which is guaranteed to yield a triangulation by theorem 3.1.3.

3.2 Ear-Clipping Method

As we have seen previously, one way to triangulate a polygon is to add as many diagonals as possible until you are left with a triangulation. This approach is simple but very costly, since there are $\binom{n}{2} = O(n^2)$ diagonal candidates and testing a candidate takes $O(n)$ time. Repeating this $O(n^3)$ procedure for each of the $n-3$ diagonals would lead us to a $O(n^4)$ total running time. Fortunately, this can be brought down to a $O(n^2)$ running time. To do so, first we need to define what is an *ear* of a polygon. We say three consecutive vertices u, v, w of P, with regards to the counter-clockwise ordering around P, form an ear if uw is a diagonal of P. If uvw is an ear, we say that v is the *ear tip*.

Theorem 3.2.1 (Meister's Two Ears Theorem)**.** *Every polygon with $n \geq 4$ vertices has at least 2 different ears.*

Proof: Every leaf in the geometric dual of a polygon corresponds to an ear. Since the geometric dual is a tree and every tree with at least 2 nodes must have at least 2 leaves, the polygon must have at least 2 different ears. □

Using the theorem above we can easily improve the $O(n^4)$ algorithm we have mentioned before to a $O(n^3)$ worst-case running time. We know that there must be a diagonal that closes off an ear of the polygon. We also know that there are only $O(n)$ such candidates of diagonals ($\overline{v_i, v_{i+2}}$, for $i \in [0, n-1]$). By adding one such diagonal at each step, we will have a new triangle, which is the ear we have added a diagonal to. And so, we have removed a vertex from P and we can repeat this process with the remaining vertices until we have a triangulation of P.

We can now improve on the above idea to finally reach the promised $O(n^2)$ running time. We will make use of the fact that when we remove an ear from P, we do not change P drastically, in fact, all but two vertices will keep their previous status as ear tip candidate. So first we will determine for each vertex v_i if it is a potential ear tip, by checking if $\overline{v_{i-1}v_{i+1}}$ is a diagonal. This will take total time $O(n^2)$, but we will not need to repeat this process.

Consider that P has at least five vertices. By lemma 3.3 we know there is a diagonal in P. Let $(v_1, v_2, v_3, v_4, v_5)$ be five consecutive vertices of P. Suppose v_2 is an ear tip. If we remove the ear $E = \Delta v_1 v_2 v_3$, only the status of v_1 and v_3 may change with regard to whether or not they were potential ear tips (see figure 3.5). Consider v_4, for example. For v_4 to be a potential ear tip, $\overline{v_3v_5}$ needs to be a diagonal. By adding the diagonal $\overline{v_1v_3}$ and essentially removing E from P, we leave the endpoints of the segment $\overline{v_3v_5}$ unchanged, so if the line between them was a diagonal, it will continue being so. Let us now show that the same is true when $\overline{v_3v_5}$ was not a diagonal of P. If $\overline{v_3v_5}$ was external to the boundary of P, clearly the removal of E cannot make it internal. Otherwise, $\Delta v_3v_4v_5$ must contain another vertex, which is reflex (the vertex corresponding to x in lemma 3.3). But the removal of E only removes a convex vertex from P. Therefore, the status of v_4 is unchanged. This means that whenever we remove an ear from P, we need only to update the status of the endpoints of the diagonal we added, leading us to algorithm 3:

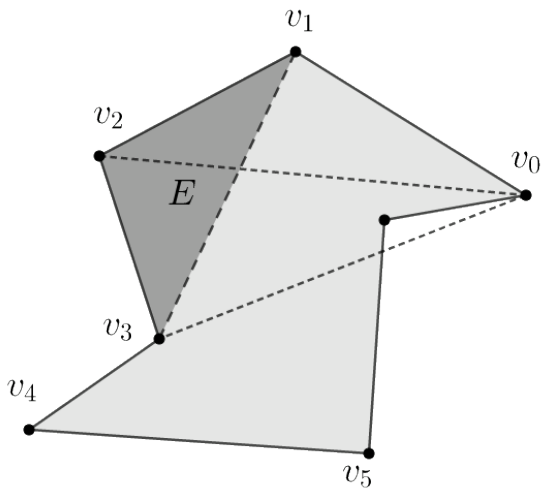


Figure 3.5: *In this example, after the removal of E, v_1 is no longer an ear tip candidate.*

Algorithm 3 Receives a simple polygon P given as a list of vertices in counter-clockwise order and returns a list of diagonals of P that represent the triangulation of P.

```
1:  function earClippingTriangulation(P):
2:      l ← ∅
3:      Consider P = [v_1, v_2, v_3, ..., v_n]
4:      for every v ∈ P do
5:          earTip(v) ← isEar(v, P)
6:      while |P| > 3 do
7:          Let v_2 be a vertex of P.
8:          while not isEar(v_2, P) do
9:              v_2 ← next(v_2)
10:         v_1 ← previous(v_2)
11:         v_3 ← next(v_2)
12:         l ← l ∪ {v_1v_3}
13:         P ← P \ {v_2}
14:         earTip(v_1)← isEar(v_1, P)
15:         earTip(v_3)← isEar(v_3, P)
16:     return l
```

3.3 Triangulation by Monotone Polygon Decomposition

3.3.1 Partitioning a Polygon into Monotone Pieces

While it is hard to triangulate an arbitrary simple polygon in $O(n)$ time, some classes of polygons are much easier to triangulate. One example is the class of convex polygons: Since all vertices in a convex polygon see each other, we can simply fix a vertex and add a diagonal from it to every other non-adjacent vertex (see figure 3.6). A possible solution for triangulating a non-convex polygon would then be to partition it in convex pieces and triangulate them, however, this is as difficult as triangulating [KS98]. Another such class of polygons which we can easily triangulate are *monotone polygons*. The algorithm we will describe now was presented by Lee and Preparata [LP76]. We will now properly define this class and describe how to triangulate an arbitrary polygon in $O(n \log n)$ time by dividing it in monotone polygons. The description we provide in this section and the next is taken from the book by De Berg et al. [Ber+08].

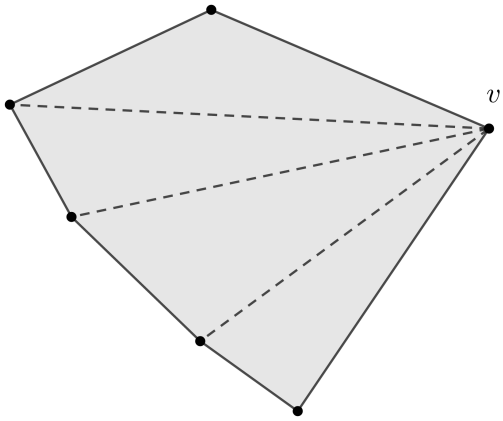


Figure 3.6: *Example of a convex polygon triangulated by extending diagonals from vertex v.*

A simple polygon P is said to be *monotone with respect to line* l if any line l' orthogonal to l intersects P at most twice. A polygon that is monotone with respect to the y-axis is called *y-monotone*. This means that if we follow along on a chain of vertices in the boundary of a y-monotone polygon, we always move downwards or horizontally, but never upwards (see figure 3.7).

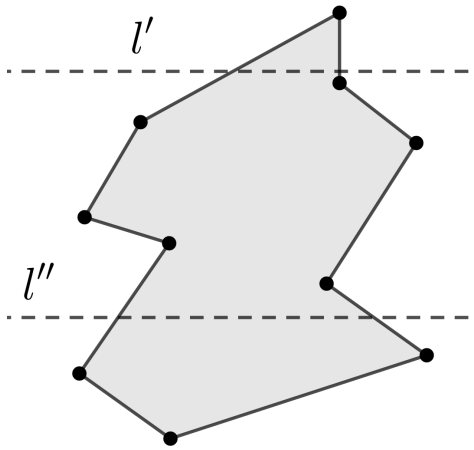


Figure 3.7: *Example of a y-monotone polygon.*

Our goal is then to first subdivide the polygon P into smaller y-monotone pieces, and then triangulate each of these pieces. This can be achieved by traversing along the boundary of P, starting at the topmost vertex and heading towards the bottommost vertex in either subchain of P. Some vertices along the way may force us to head back vertically as we travel through them, these will be referred to as *turn vertices*. To partition P into monotone pieces we have to get rid of all turn vertices, and we will do so by adding diagonals. We first will divide the vertices of P in five different categories, which will help us decide the pairs of vertices that we have to add diagonals to.

Before explaining the different categories of vertices, we need to specify a way to deal with different vertices with same y-coordinate. We do this by defining the notions of *above* and *below* for a pair of vertices. We say a vertex p is below another vertex q if $p_y < q_y$ or $p_y = q_y$ and $p_x > q_x$, and p is above q if $p_y > q_y$ or $p_y = q_y$ and $p_x < q_x$. This can be seen as slightly

tilting the plane in clockwise direction such that no two points have the same y-coordinates (see figure 3.8).

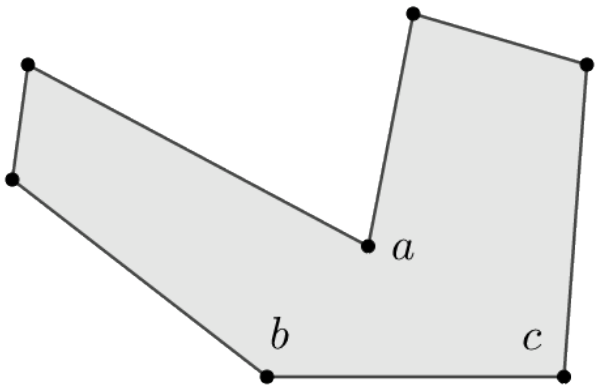


Figure 3.8: *In this example, a is above b and c, and b is above c, since $b_x < c_x$.*

The first category is *regular vertices*: if a vertex of P is not a turn vertex, it is a regular vertex; this means one of its neighbors lies above it and the other below it. The other four categories are for turn vertices and are defined as follows: a vertex v is a *start vertex* if its two neighbors are below it and the interior angle at v is less than π; if the interior angle is greater than π, it is considered a *split vertex*, a vertex v is a *end vertex* if its two neighbors are above it and the interior angle at v is less than π; if the interior angle at v is greater than π, it its considered a *merge vertex* (see figure 3.9).

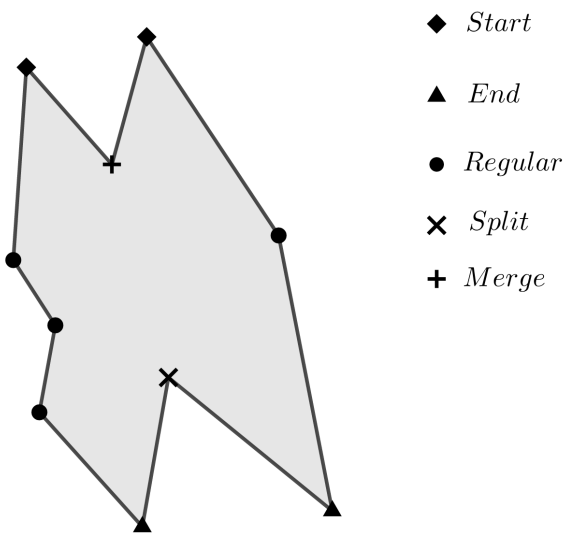


Figure 3.9: *Example of a non-monotone polygon with its vertices labeled into the five different categories.*

We will perform a downward plane sweep along P, maintaining the intersection of the sweep line with P. Whenever we reach a split vertex, a component of the intersection splits, whenever we reach a merge vertex, a component of the intersection merges, and hence these names were chosen in [Ber+08]. These split and merge vertices are the sources of non-monotonicity. This is reflected in the following lemma:

Lemma 3.3.1. *A polygon is y-monotone if it has no split and merge vertices.*

This lemma implies that P has been succesfully partitioned into monotone pieces if no split and merge vertices remain. We will do so by adding non-crossing diagonals between split and merge vertices. Once we have done this, P will be partitioned into y-monotone pieces.

Let us begin explaining how to add these diagonals by describing how to handle split vertices. We will use a plane sweep algorithm for this. Let $v_1, v_2, \ldots, v_n$ be the vertices of P given in counterclockwise order. Let $e_1, e_2, \ldots, e_n$ be the edges of P where $e_i = \overline{v_i v_{i+1}}$, for $1 \leq i < n$, and $e_n = \overline{v_n v_1}$. The plane sweep algorithm will move an imaginary horizontal line along P, stopping at certain *event points*. The event set will be the

vertex set of P sorted by y-coordinate, where if two vertices have the same y-coordinate, the one with smaller x-coordinate appears first.

Our objective is to add a diagonal from every split vertex to a vertex above it. The main idea is that when dealing with split vertex v_i we would like to link it to a nearby vertex, since it probably means we can link them without intersecting the boundary of P. Let us now explain this further. Let e_j be the edge of P immediately to the left of v_i on the sweep line and e_k the edge immediately to the right of v_i on the sweep line. We can always link v_i to the lowest vertex lying in-between e_j and e_k. If there is no such vertex, we can link v_i to the upper endpoint of either e_j or e_k. We will call this vertex the *helper* of e_j, denoted by helper(e_j). The helper of e_j will be the lowest vertex above the sweep line such that the horizontal segment connecting it to e_j does not intersect the boundary of P (see figure 3.10).

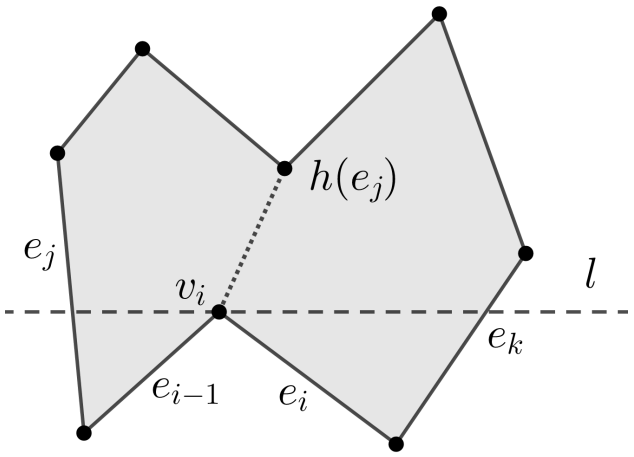


Figure 3.10: *In this example v_i is a split vertex and $h(e_j) \coloneqq$ helper(e_j).*

While we can find which diagonal to add at the moment we find a split vertex, we cannot do the same for a merge vertex since we would need to link it to a vertex that is below the sweep line. Fortunately, this is an easy problem to solve. Suppose that the sweep line reaches a certain merge vertex v_i. As we defined previously, let e_j be the edge of P immediately to the left of v_i on the sweep line and e_k the edge immediately to the right of v_i on

the sweep line. Note that v_i becomes helper(e_j) as soon as we reach it. We would like to be able to link v_i to the highest vertex below the sweep line, this is the exact opposite of what we wanted previously for split vertices. At this point in the algorithm we do not know which is the highest vertex below the sweep line, but as soon as we find another vertex v_m which replaces v_i as the helper of e_j, this vertex is the one we are looking for. So, whenever we replace the helper of an edge, we check if the previous helper was a merge vertex, and if so we add a diagonal to it. If the helper of e_j is never replaced below v_i we can link it to the lower endpoint of e_j (see figure 3.11).

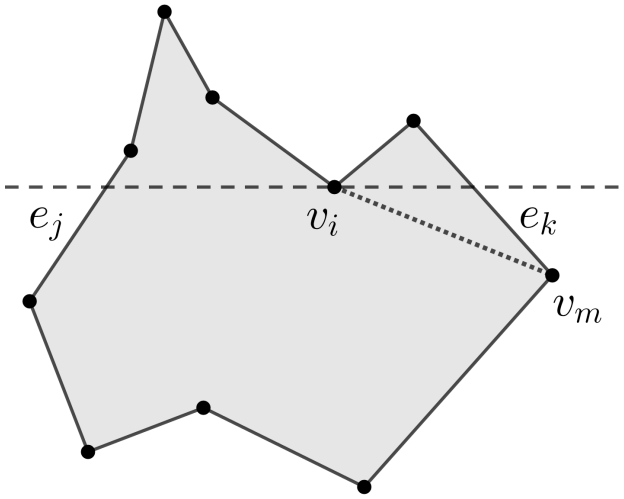


Figure 3.11: *In this example v_i is a merge vertex and the highlighted diagonal will be added when the sweep line reaches v_m.*

Along this sweep method we need to find the edge to the left of every vertex of P. We will do so by storing the edges of P that intersect the sweep line in a balanced binary search tree T. The leaves of T will be the edges of P that intersect the sweep line in order. Along with every edge we store the corresponding helper vertex. The tree T represents the status of the sweep line algorithm at a given event. As we move the line forward, edges can stop intersecting with the sweep line, or the helper vertices associated with them may change.

The algorithm partitions P into y-monotone subpolygons that will have to be processed individually when we get to triangulate them. To easily

access these polygons we will store them in a DCEL data structure.

We are now ready to show the main body of the algorithm:

Algorithm 4 Receives a simple polygon P given as a list of vertices in counter-clockwise order and returns a list of diagonals that divide P into monotone polygons.

```
1: function DIVIDEMONOTONE(P):
2:     P' ← the vertices of P, sorted by y-coordinate, with ties resolved by
   x-coordinate.
3:     Let T be an empty balanced binary search tree.
4:     l ← ∅
5:     for every v_i ∈ P' do
6:         P'', d ← handleVertex(P, v_i, T)
7:         l ← l ∪ d
8:         P ← P''
9:     return l
```

Line 6 of **divideMonotone** checks the vertex type of v_i and calls one of the five corresponding functions. We will now describe how to handle the different types of vertices. There are always two things we have to do when analyzing a vertex. The first is deciding if we must add a diagonal. This is the case when we have a split vertex or when we replace a helper vertex and the previous helper vertex was a merge vertex. The second is updating the status of the sweeping line, stored in the tree T.

Algorithm 5 Auxiliary function to handle the start vertex. Returns the rest of the polygon we have yet to divide and the diagonal added.

```
1: function HANDLESTARTVERTEX(P, v_i, T):
2:     Insert e_i in T and set v_i as its helper.
3:     return (P, ∅)
```

Algorithm 6 Auxiliary function to handle the end vertex. Returns the rest of the polygon we have yet to divide and the diagonal created.

```
1: function HANDLEENDVERTEX(P, v_i, T):
2:     P' ← P
3:     d ← ∅
4:     if helper(e_i) is a merge vertex then
5:         u ← helper(e_i)
6:         d ← d ∪ {v_i u}
7:         P' ← vertices of P from v_i to u.
8:     Delete e_i from T.
9:     return (P', d)
```

Algorithm 7 Auxiliary function to handle split vertices. Returns the rest of the polygon we have yet to divide and the diagonal created.

```
1: function HANDLESPLITVERTEX(P, v_i, T):
2:     d ← ∅
3:     Search in T for the edge e_j directly to the left of v_i.
4:     u ← helper(e_j)
5:     d ← d ∪ {v_i u}
6:     P' ← vertices of P from v_i to u.
7:     helper(e_j) ← v_i
8:     Insert e_i in T and set v_i as its helper.
9:     return (P', d)
```

Algorithm 8 Auxiliary function to handle merge vertices. Returns the rest of the polygon we have yet to divide and the diagonal created.

```
1: function HANDLEMERGEVERTEX(P, v_i, T):
2:     d ← ∅
3:     if helper(e_{i−1}) is a merge vertex then
4:         u ← helper(e_{i−1})
5:         d ← d ∪ {uv_i}
6:         P′ ← vertices of P from v_i to u.
7:     else P′ ← P
8:     P″ ← P′
9:     Delete e_{i−1} from T.
10:    Search in T for the edge e_j directly to the left of v_i.
11:    if helper(e_j) is a merge vertex then
12:        u ← helper(e_j)
13:        d ← d ∪ {uv_i}
14:        P″ ← vertices of P′ from v_i to u.
15:    helper(e_j) ← v_i
16:    return (P″, d)
```

Algorithm 9 Auxiliary function to handle regular vertices. Returns the rest of the polygon we have yet to divide and the diagonal created.

```
1: function HANDLEREGULARVERTEX(P, v_i, T):
2:     P′ ← P
3:     if the interior of P lies to the right of v_i then
4:         if helper(e_{i−1}) is a merge vertex then
5:             u ← helper(e_{i−1})
6:             d ← uv_i
7:             P′ ← vertices of P from v_i to u.
8:         Delete e_{i−1} from T.
9:         Insert e_i in T and set v_i as its helper.
10:    else Search in T for the edge e_j directly to the left of v_i.
11:        if helper(e_j) is a merge vertex then
12:            u ← helper(e_j)
13:            d ← uv_i
14:            P′ ← vertices of P from v_i to u.
15:        helper(e_j) ← v_i
16:    return (P′, d)
```

Lemma 3.3.2. *The algorithm* ***divideMonotone*** *adds a set of non-crossing diagonals that partition P into monotone subpolygons.*

Proof: It is easy to see that the subpolygons formed by **divideMonotone** have no split or merge vertices and thus are monotone by lemma 3.3.1. All that is left is for us to prove that the diagonals added by divideMonotone do not intersect the boundary of P and are non-crossing. It suffices to show that whenever we add a new segment, it does not intersect any other edge. We will prove this for the segment added in **handleSplitVertex**, the proof for the segments added in the other auxiliary functions is similar.

Let $\overline{v_m v_i}$ be the segment added by handleSplitVertex when the sweep line reached vertex v_i. Let e_j be the edge to the left of v_i, and let e_k be the edge to the right of v_i. Thus helper$(e_j) = v_m$ when v_i is reached.

We first argue that an edge on the boundary of P cannot intersect $\overline{v_m v_i}$. To show this, let us consider the quadrilateral Q bounded by the horizontal lines through v_m and v_i and by the edges e_j and e_k. Since v_m is the helper of e_j, there cannot be any other vertices of P inside Q. Now suppose there is an edge of P intersecting $\overline{v_m v_i}$. Since it cannot have an endpoint inside Q, it must intersect either the horizontal segment connecting v_m to e_j or the one connecting v_i to e_j. Both are impossible since e_j is the edge directly to the left of both v_m and v_i.

Now let us consider a diagonal previously added by the algorithm. Since there are no vertices inside Q, and any diagonal added previously must have both endpoints lying above v_i, it cannot intersect $\overline{v_m v_i}$. □

We will now analyze the running time of the algorithm. Sorting the vertices of P takes $O(n \log n)$ time and initializing T takes constant time. To handle a certain vertex of P, we perform at most one query, one insertion and one deletion in T, and we add at most two new diagonals to P. Balanced search trees allow us to perform queries and updates in $O(\log n)$ time, and adding a diagonal takes constant time. And so, handling an event of the sweeping line takes $O(\log n)$ time, and the algorithm runs in $O(n \log n)$ time. The space used is clearly linear: we store a copy of every vertex of P in the sorted list, and every edge is stored at most once in T. This is summarized

in the following theorem:

Theorem 3.3.3. *A simple polygon with n vertices can be partitioned into y-monotone subpolygons in $O(n \log n)$ time with an algorithm that uses $O(n)$ space.*

3.3.2 Triangulating a Monotone Polygon

We have just shown how to subdivide a simple polygon into monotone parts in $O(n \log n)$ time. In this section we will show how a monotone polygon can be triangulated in $O(n)$ time. Since the total number of vertices in the monotone parts is exactly n, we get that it is possible to triangulate any simple polygon in $O(n \log n)$ time.

Let P be a y-monotone polygon, we can assume that P is *strictly y-monotone*, that is, that P has no horizontal edges, by applying the notions of above and below we have discussed previously. The algorithm we will describe is a greedy algorithm that goes from the top to the bottom of the boundary chains of P adding as many diagonals as it can at each step.

We will handle the vertices by decreasing y-coordinate. If two vertices share the same y-coordinate, the one with the smallest x-coordinate is handled first. We will store the vertices that we encounter along the execution that still need at least one diagonal added to them in a stack S. When we handle a new vertex we add as many diagonals as possible to the vertices that are in S. The vertices on the stack lie on the boundary of the part of P that has not been triangulated yet. The lowest of these vertices, which is the one visited last, is the top of the stack (the second lowest is the second from the top, and so on). The part of P that has yet to be triangulated has a very particular shape: it is a funnel facing towards the bottom of P. One of the boundary chains of this funnel is a single edge of P, and the other is a chain of reflex vertices, that is, vertices with interior angle greater than or equal to π. Only the highest vertex, which is on the bottom of the stack, is convex. This property remains when handling new vertices and so it is the invariant of this algorithm (see figure 3.12).

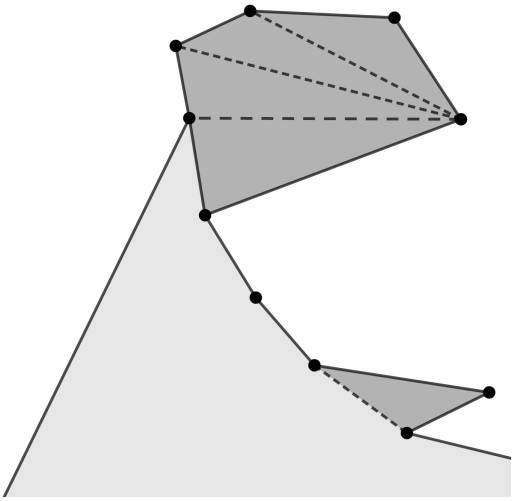


Figure 3.12: *Example of the funnel maintained by the algorithm. The darker regions represent the parts of the polygon that have already been triangulated.*

Now let us discuss which diagonals can be added when we are handling a new vertex v_j. There are two possible cases: either v_j lies on the edge side of the funnel, or on the chain with the reflex vertices. If v_j lies on the edge side it must belong to the lower endpoint of the single edge e bounding the funnel. We can add diagonals from v_j to all vertices on the stack, except the bottom one, since the last vertex in S is the upper endpoint of e and thus it is already connected to v_j. All the vertices that we have connected to v_j get popped from S. The untriangulated part of P will be bounded by the edge connecting v_j to the element previously on the top of the stack and the edge of P that extends downward from this vertex. So the funnel shape is preserved. This vertex and v_j are still on the untriangulated part of P and so we push them onto the stack (see figure 3.13).

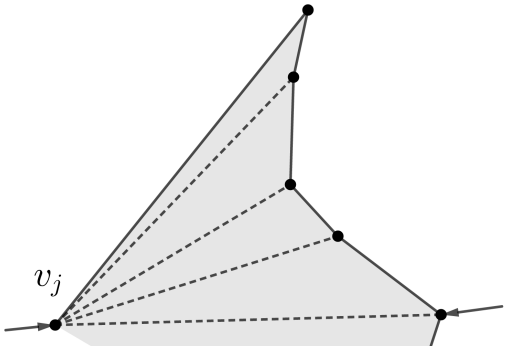


Figure 3.13: *Example when v_j lies on the edge side of the funnel. Both v_j and the lowest vertex on the opposite side get pushed into S at the end.*

If v_j lies on the chain with the reflex vertices, we may not be able to connect it to all the vertices on the stack. But the ones we can connect it to are all consecutive and are on the top of the stack. First, we pop an element from S since it is already connected to v_j by an edge in the boundary. Then, we pop vertices from S and connect them to v_j until we encounter one that we cannot connect it to. We can easily verify if we can add a diagonal from v_j to a vertex v_k by looking at v_j, v_k and the last popped vertex. When we find a vertex that we cannot connect to v_j, we push the last vertex that has been popped back to S. This will be either the last vertex to which we added a diagonal or, if we have added no diagonals, will be the neighbor of v_j in P. At the end of this proccess we push v_j back onto S (see figure 3.14).

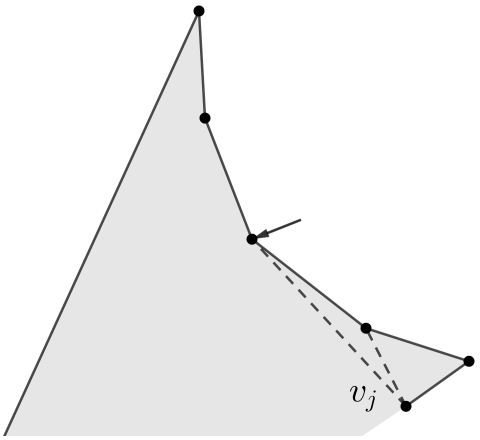


Figure 3.14: *Example when v_j lies on the chain side of the funnel. The highlighted vertex gets popped and then pushed back into S.*

In both cases the invariant is maintained: one side of the funnel is a single edge and the other a chain of reflex vertices. And so, we get the following algorithm:

Algorithm 10 Receives a y-monotone polygon P and returns a triangulation of P.

1: **function** TRIANGULATEMONOTONE(P):
2: Sort the vertices of by decreasing y-coordinate P by merging the left and right chains of P into one sequence. Let $u = u_1, u_2, \ldots, u_n$ be this sorted sequence.
3: Initialize an empty stack S and push u_1 and u_2 onto it.
4: **for** $j \leftarrow 3$ **to** $n - 1$ **do**
5: **if** u_j and top(S) lie on different chains **then**
6: Pop all vertices from S.
7: Add a diagonal from u_j to each popped vertex, except the last.
8: Push u_{j-1} and u_j onto S.
9: **else**
10: Pop one vertex from S.
11: Pop vertices from S as long as the diagonals from these vertices to v_j do not violate the boundary of P. Add these diagonals and push the last vertex that has been popped back onto S.
12: Push u_j onto S.
13: Add diagonals from u_n to all vertices in S, except the first and last.

Line 2 of this algorithm can be done in linear time and line 3 takes

constant time. Line 4 is executed a total of $n-3$ times. At every execution of the for-loop we push at most two vertices. And thus the total number of pushes, counting the ones on line 3 is bounded by $2n-4$. Since the number of pops cannot exceed the number of pushes, the total time for all executions of line 4 is $O(n)$. Line 13 is at worst linear time. So this algorithm runs in $O(n)$ total time. This is summarized in the theorem below:

Theorem 3.3.4. *A strictly y-monotone polygon with n vertices can be triangulated in linear time.*

By combining theorems 3.3.3 and 3.3.4, we get the following:

Theorem 3.3.5. *A simple polygon with n vertices can be triangulated in* $O(n \log n)$ *time using* $O(n)$ *space.*

Chapter 4

Visibility Graph

4.1 Introduction

We will begin our analysis on shortest path algorithms inside polygons with the *visibility graph algorithm* [Ber+08]. This is an algorithm that, given a polygon P (represented as a list of vertices in counter-clockwise order), finds the euclidean shortest path between any pair of vertices of P in $O(n^2 \log n)$ construction and query time. This is achieved by performing a search on the visibility graph data structure. This graph will have the same vertices as the polygon P, and for every pair a, b of vertices there is an edge between them if they can see each other, that is, if $\overline{ab}$ is entirely contained inside P (see figure 4.1). We identify the vertices of the visibility graph with the vertices of the polygon.

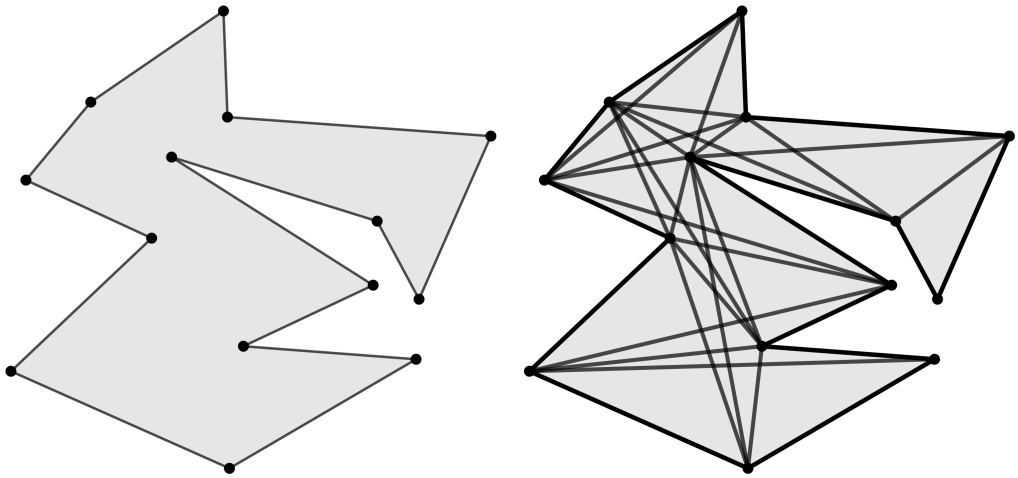


Figure 4.1: *Example of a polygon and its associated visibility graph.*

This problem can then be formulated as follows:

Visibility Graph
Input: Simple polygon P.
Output: The visibility graph over the vertices of P.

First off we need a characterization on the shape of the shortest paths we are interested in finding. Lemma 4.1.1 guarantees that the shortest path between two vertices of the polygon is entirely contained in the visibility graph, and so, by searching inside this graph we will find the shortest paths between vertices of P.

Lemma 4.1.1. *Any shortest path between two vertices inside a polygon is a polygonal path whose inner vertices are vertices of the polygon.*

Proof: Let s and t be vertices of P. Let π be a shortest path from s to t with minimal inner vertices that are not in the boundary of P. If no such vertices exist, we are done. Thus, suppose that there is $v \in V(\pi) \setminus V(P)$. Let e_1 and e_2 be the two incident edges of π in v. If they are collinear, v can be deleted from π, contradicting the fact that π has minimal number of inner vertices. There is a $\epsilon > 0$ such that the circle with radius ϵ around v intersects e_1 and e_2 and no vertices from P (see figure 4.2). Let w_1 and w_2 be the intersection points of the circle with e_1 and e_2, respectively. Since w_1 and w_2 are at distance ϵ from v, $\overline{w_1 w_2}$ is inside the polygon. Also, since e_1 and e_2 are not collinear, the segment $w_1 w_2$ is strictly shorter than the chain $w_1 v w_2$. Thus, by replacing $w_1 v w_2$ by $w_1 w_2$ in π, we get a strictly shorter path from s to t, contradicting the choice of π (see figure 4.2). □

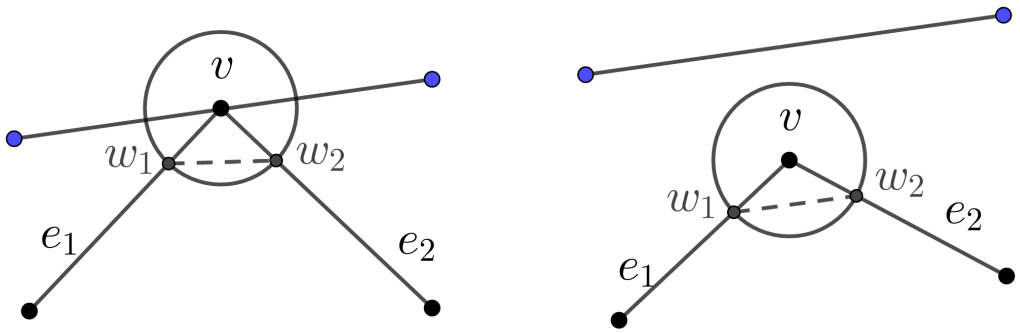


Figure 4.2: *By replacing w_1vw_2 by w_1w_2 we get a shorter path inside the polygon.*

4.2 Computing the Visibility Graph

To compute the visibility graph we need to determine for all pairs of vertices whether they can see each other. This means that for every pair of vertices we need to check whether the line segment connecting them is contained inside the polygon. If done naively, this test would take $O(n)$, leading to an overall $O(n^3)$ running time, since there are $\binom{n}{2} = O(n^2)$ such pairs. We will show that if we consider the vertices in a specific order, this time can be improved further. This procedure is summarized below:

Algorithm 11 Receives a polygon P as a list of vertices and returns the visibility graph G of P.

```
1: function visibilityGraph(P):
2:     Let G =(V(P),∅) be a graph.
3:     for every v ∈ V do
4:         W ← visibleVertices(v, P)
5:         for every w ∈ W do
6:             e ← (v, w)
7:             weight(e) ← euclidean distance from v to w.
8:             Add the edge e to E(G).
9:     return G
```

In the algorithm above, **visibleVertices**(v, P) is a procedure that returns

a list of vertices that can be seen by the vertex v in P (the pseudocode for it will be provided further ahead). If we need to test if a certain vertex w is visible from v, we have to test the segment $\overline{vw}$ against all edges of the boundary of P. Although this procedure is linear, it would lead to an overall cubic run time, as previously mentioned. We can use the fact that we need to find all vertices visible from v to our advantage, that is, we can use the information gained by testing a certain vertex to improve the tests on subsequent vertices.

Let us consider the set of all segments to vertices of P, starting from vertex v. We will analyze each of these segments following a cyclic order around the vertex v. Vertex w is visible from v if the segment $\overline{vw}$ does not intersect any boundary edge of P. Consider the half-line ρ that begins at v and passes through w (see figure 4.3). If w is not visible, ρ must intersect a boundary edge before reaching w. One way to efficiently find such intersection is to store the edges intersected by ρ in a search tree, we will describe this structure in details further ahead.

Analyzing the vertices of the polygon in cyclical order is the same as rotating the half-line ρ around v. So it is similar to the plane sweep approach used widely in other geometric problems, but instead of a vertical or horizontal line sweeping the plane, we use a rotating half-line. The status of the rotational plane sweep is the ordered sequence of edges intersected, stored in the search tree. The events in the sweep are the vertices of P. For each vertex w we have to decide if w is visible from v by searching in the search tree and we then have to update the tree by inserting and/or deleting edges incident to w.

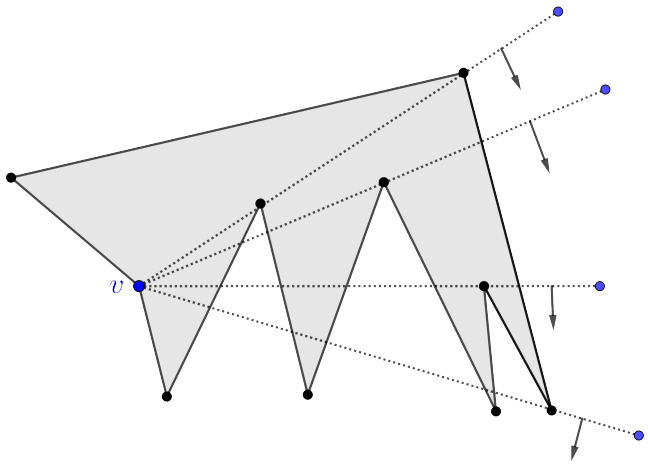


Figure 4.3: *Example of the rotational plane sweep around vertex v.*

The procedure **visibleVertices** performs this rotational plane sweep. The half-line ρ begins pointing in the positive x-direction from v, proceeding in clockwise direction. We then sort all the vertices by the clockwise angle formed between the segment from v to it with the positive x-axis. Ties are resolved by the distance from v to the vertices, that is, vertices closer to v are treated beforehand.

Algorithm 12 Returns the list of vertices visible from a given vertex v.

1: **function** VISIBLEVERTICES(v, P):
2: $V \leftarrow$ The vertices $[v_1, \ldots, v_n]$ of P sorted in clockwise order around vertex v.
3: Let T be a balanced binary search tree.
4: Find the edges intersected by the half-line from v to the positive x-axis and store them in T in order.
5: $W \leftarrow \emptyset$
6: **for** $i \leftarrow 1$ **to** n **do**
7: **if visible**(v, v_i) **then**
8: Add v_i to W
9: Add to T the edges incident to v_i that lie on the clockwise side of the half-line from v to v_i.
10: Remove from T the edges incident to v_i that lie on the counterclockwise side of the half-line from v to v_i.
11: **return** W

As we have mentioned before, to determine whether a certain vertex w is visible from v, we must determine if the half-line ρ from v and passing through w reaches w before hitting any boundary edge of P. We will do so by storing all the edges intersected by ρ in a balanced binary search tree (Line 3 and 4 of algorithm 12). Allowing us to check if w is behind the minimum element of the tree as seen from v (see figures 4.4 and 4.5).

The leaves of the search tree will store the edges intersected by ρ in order. The interior nodes also store edges and help guide the search. An interior node stores the rightmost edge in its left subtree, that is, all the edges in the left subtree will be smaller than or equal to the edge stored and all edges in the right subtree will be greater, with respect to the order along ρ (see figure 4.5).

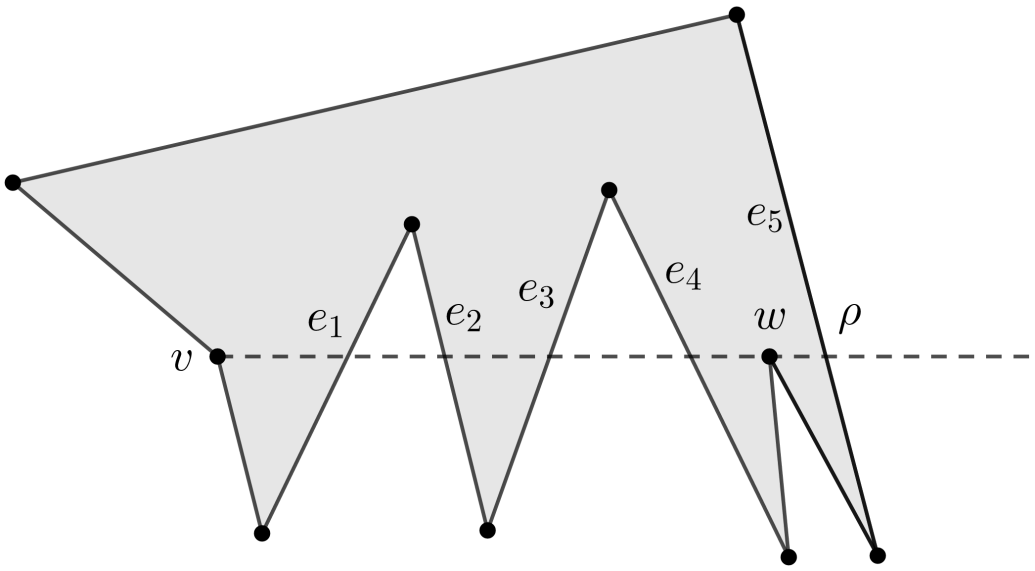


Figure 4.4: *The half-line ρ going from v and passing through w intersects the boundary edges e_1, e_2, e_3, e_4,and e_5.*

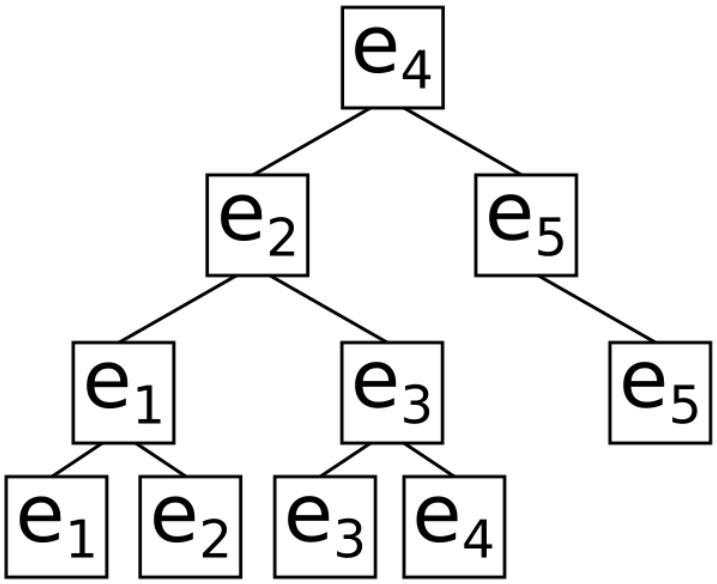


Figure 4.5: *Balanced binary search tree corresponding to the example in figure 4.4.*

Now, we have to decide wether a certain vertex v_i is visible from v (Line 7 of algorithm 12). At first glance it seems that we would only need to check whether the first edge intersected by the half-line is behind v_i, but we need to pay extra attention when the half-line contains other vertices. Figure 4.6 illustrates some of the cases where this occurs. Since $\overline{vv_i}$ may intersect the exterior of P depending on the configuration, it is natural to think that we need to check all the edges incident to all the vertices contained in $\overline{vv_i}$ to determine visibility. This would be very costly to check naively, but fortunately the vertices that are contained in $\overline{vv_i}$ have already been processed, since they are closer to v. Therefore, we determine if a certain v_i is visible as follows: if v_{i-1} is not visible, then v_i is also not visible. If v_{i-1} is visible, there are two possibilities where v_i is not visible. Either the entire segment $\overline{v_{i-1}v_i}$ is outside of P or $\overline{v_{i-1}v_i}$ is intersected by an edge in the tree. This follows from the fact that $\overline{vv_i} = \overline{vv_{i-1}} \cup \overline{v_{i-1}v_i}$ (If $i = 1$ there is no vertex between v and v_i). And so we get the following algorithm:

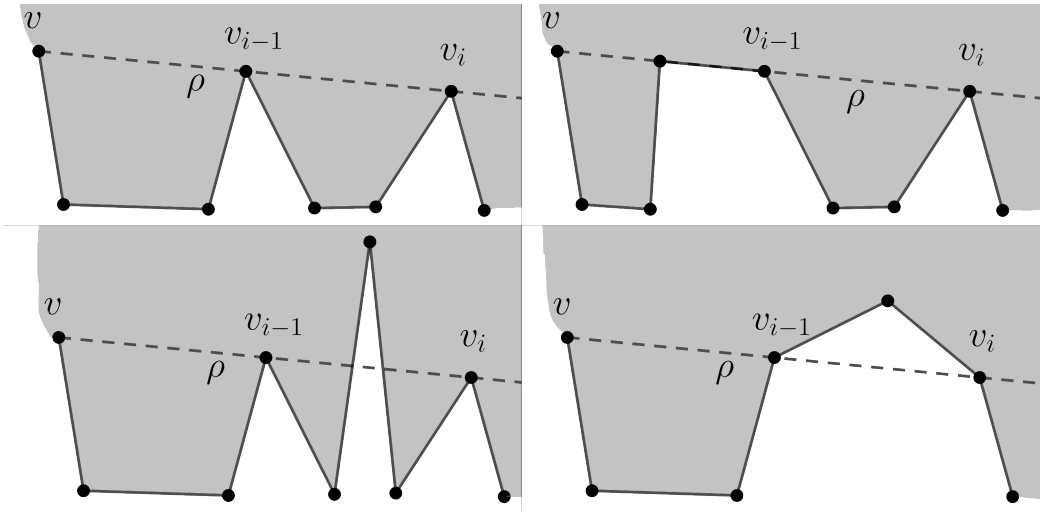


Figure 4.6: *Some of the cases where $\overline{vv_i}$ intersects other vertices. In the bottom cases v_{i-1} is visible from v, but v_i is not.*

Algorithm 13 Determines if a certain vertex v_i is visible from vertex v.

```
1:  function VISIBLE(v, v_i):
2:      if v_i v_i intersects the exterior of P, locally at v_i then
3:          return False
4:      else if  i = 1 or v_{i-1} is not contained in v_i v_i then
5:          Search in the tree for the edge e in the leftmost leaf.
6:          if e exists and v_i v_i intersects e then
7:              return False
8:          else return True
9:      else if v_{i-1} is not visible then
10:         return False
11:     else
12:         Search in the tree for the first edge e to the right of v_{i-1}.
13:         if e exists and e intersects v_{i-1} v_i then
14:             return False
15:         else return True
```

4.3 Complexity Analysis

We are now ready to discuss the running time of the algorithms we have described: the running time of **visibleVertices** (Algorithm 12) is dominated by the time to sort the vertices in cyclic order, which is $O(n \log n)$. Each execution of **visible** (Algorithm 13) takes time $O(\log n)$, since it is composed of operations in a balanced binary search tree and constant time geometric tests for each level. This means that the main loop of **visibleVertices** takes total time $O(\log n)$, leading to a total running time of $O(n \log n)$. We also call **visibleVertices** for each vertex in P in order to build the visibility graph in **visibilityGraph** (Algorithm 15). This is summarized in the following theorem:

Theorem 4.3.1. *The visibility graph of a given polygon P with n vertices can be computed in $O(n^2 \log n)$ time.*

With the visibility graph in hand we can then run Dijkstra's algorithm for shortest paths in weighted graphs [Dij59] to find the shortest path between any two vertices in the visibility graph. Lemma 4.1.1 assures us that any shortest path between two vertices in a polygon needs to be composed of only vertices of P, this, together with the fact that the visibility graph holds all the possible edges between vertices of P that do not intersect the exterior, means that the visibility graph contains all the shortest paths between its vertices. Thus, by applying Dijkstra's algorithm we are guaranteed to find this shortest path. Since the visibility graph has size $O(n^2)$ (the number of edges is bounded by $\binom{n+2}{2}$), we can find a shortest path in $O(n^2 \log n)$ time by running Dijkstra's algorithm. This is summarized in the following theorem:

Theorem 4.3.2. *A shortest path between all pairs of vertices in a polygon with n vertices can be computed in $O(n^2 \log n)$ preprocessing and query time.*

Chapter 5

Single Source Shortest Path Tree

5.1 Introduction

In the previous section we discussed the *visibility graph algorithm* for finding the shortest euclidean path between a chosen pair of vertices in a polygon. We will now show a linear time algorithm by Lee and Preparata [LP84] that, for a fixed source vertex s in a simple polygon P, represented as a list of vertices in counter-clockwise order, and with a given triangulation $\mathcal{T}$, represented as a planar graph stored in a *doubly-connected edge list* (DCEL) data structure (we have described this structure in details in chapter 2), computes a *shortest path tree* from s to every other vertex v.

The shortest path tree T is a planar tree rooted at s such that for all vertices v of P, the shortest path from s to v in P is the same as the shortest path from s to v in T, that is, the vertices of P form the vertex set of T and there will be an edge linking vertices u and v in T if the segment $\overline{uv}$ is contained inside the polygon P (see figure 5.1).

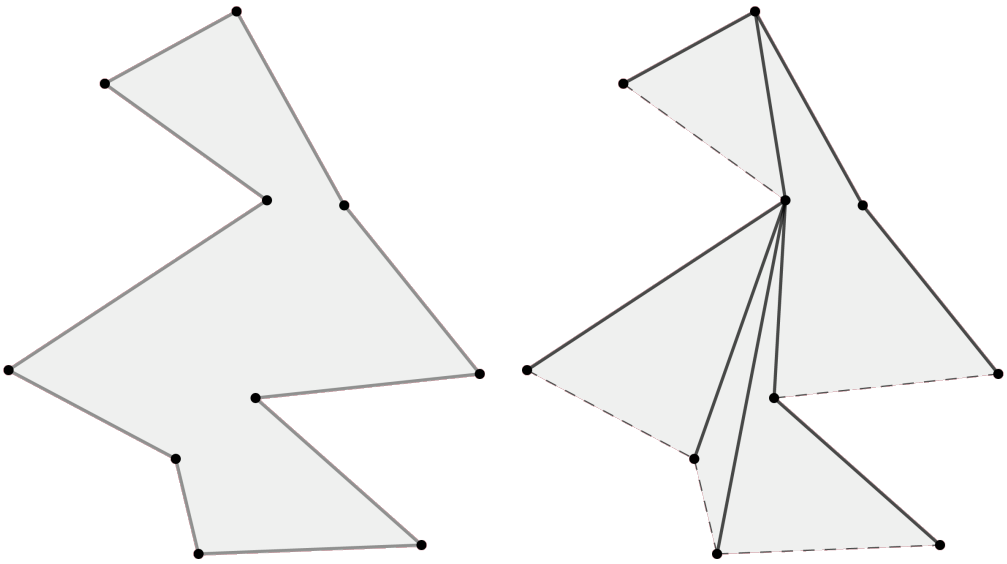


Figure 5.1: *Example of a polygon and a shortest path tree.*

We are now ready to properly formulate the problem as such:

Single Source Shortest Path Tree

Input: Simple polygon P, source vertex s of P and a triangulation $\mathcal{T}$ of P.

Output: The shortest path tree of P with source s.

For each $v \in P$, the euclidean shortest path $\pi(s, v)$ is a polygonal chain with s and v as its endpoints, as can be seen in [LP84]. By taking the union $\cup_{v \in P}\ \pi(s, v)$ of all such paths we will have a plane tree spanning over all vertices of P, because if there were any cycles, some of these paths π could be further improved, contradicting them being a shortest path. This result is a consequence of the following lemma:

Lemma 5.1.1. *Every subpath of a shortest path between two vertices is itself a shortest path.*

Proof: Let $p = [v_1, \ldots, v_k]$ be a shortest path from v_1 to v_k inside a polygon. Suppose the shortest path between v_i and v_j, $1 \leq i \leq k$, $j > i + 1$, to be

$p' = [v_i, v'_1, \ldots, v'_m, v_j]$. The path $p'' = [v_1, \ldots, v_{i-1}] \cup p' \cup [v_{j+1}, \ldots, v_k]$ would be an even shorter path from v_1 to v_j, contradicting the choice of p. □

Let $d = ul$ be an arbitrary edge of $\mathcal{T}$, and let a be the deepest common vertex in the paths $\pi(s, u)$ and $\pi(s, l)$. As we shown on the lemma below, both $\pi(a, u)$ and $\pi(a, l)$, assumed to be non-empty, must be *outward convex* (see figure 5.2).

Lemma 5.1.2. *Every shortest path inside a polygon must be outward convex, that is, it must be a polygonal chain with its concavity facing outwards.*

Proof: Let $p = [v_1, v_2, \ldots, v_k]$ be a shortest path between v_1 and v_k inside a polygon. Suppose that the subpath $p' = [v_i, \ldots, v_j]$, $1 \leq i \leq k$, $j > i + 1$, of p is inward convex. Since p' is inward convex, the endpoints v_i and v_j can see one another, and then $p'' = [v_i, v_j]$ is the shortest path between them. Substituting p' with p'' in p, we would have a new path between v_1 and v_k that is shorter than p, contradicting the choice of p. □

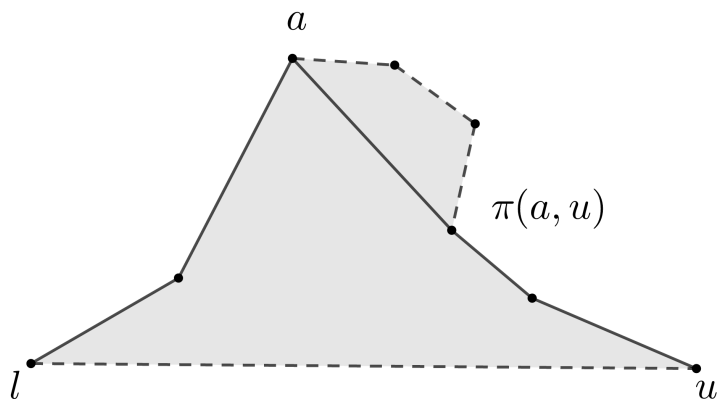


Figure 5.2: *Illustration of the outward convexity of* $\pi(a, u)$.

5.2 Algorithm Description

We define $F = F_{ul} = \pi(a, u) \cup \pi(a, l)$ as the *funnel* associated with ul and a as the *apex* of F. Suppose now $\pi(a, u)$ and $\pi(a, l)$ to be non-empty. Let d be a diagonal of $\mathcal{T}$ and let Δ_{ulx} be the unique triangle in $\mathcal{T}$ with d as one of its edges and that does not intersect the interior of $F \cup d$. We can then find the shortest path $\pi(s, x)$ by following $\pi(s, a)$ and then continuing along until we reach vertex v of F such that $\overline{vx}$ does not intersect the interior of F (see figure 5.3). By following this procedure, we have found the shortest path from s to x with the information of the shortest paths to u and l and have enough information to construct two new funnels, allowing us to find the shortest paths to two other vertices (see figure 5.4).

If either $\pi(a, u)$ or $\pi(a, l)$ is empty, we will have what we call *degenerate funnel*. Let $F' = [u, l]$ be a degenerate funnel, assume without loss of generality that u is the apex of F'. By looking at the triangle Δulx that has not yet been processed in our algorithm, we can extend the funnel F' to a non-degenerate one by adding the edge $\overline{ux}$ to our funnel, we would then have a new funnel $F'' = [x, a = u, l]$. We would then continue along with F'' and the funnel $F''' = [a = u, x]$

So at every step of our algorithm we will have the funnel associated with the edge we are currently processing, and by looking at the opposing vertex x we can proceed recursively with two new funnels.

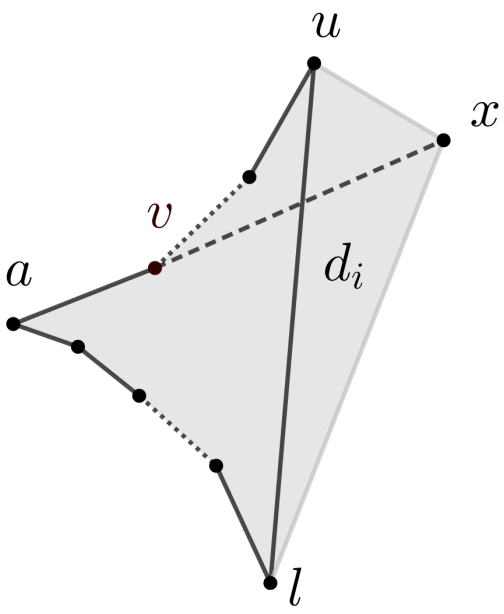


Figure 5.3: *Example of a non-degenerate funnel associated with* $\overline{ul}$ *and its opposing vertex* x*.*

As the algorithm comes to process a certain diagonal $d = ul$, we store the funnel $F_{ul} = [u = f_0, \ldots, f_{k-1}, a = f_k, f_{k+1}, \ldots, f_n = l]$ as a double-ended queue (deque) with a history stack, as described by Hershberger and Snoeyink [HS91]. We will refer to the vertices in the range $[0, k-1]$ as the *upper chain* of F_{ul} and vertices in the range $[k+1, n)$ as the *lower chain*. This structure allows the following linear time operations:

1. **Add(f,x)**: Adds a vertex x to the beginning of the deque.
2. **Add(b,x)**: Adds a vertex x to the end of the deque.
3. **split(f,i)**: Removes every element from the deque inside the range $(i, n]$.
4. **split(b,i)**: Removes every element from the deque inside the range $[0, i)$.
5. **undo()**: Undoes the last *add* or *split* operation on the deque.

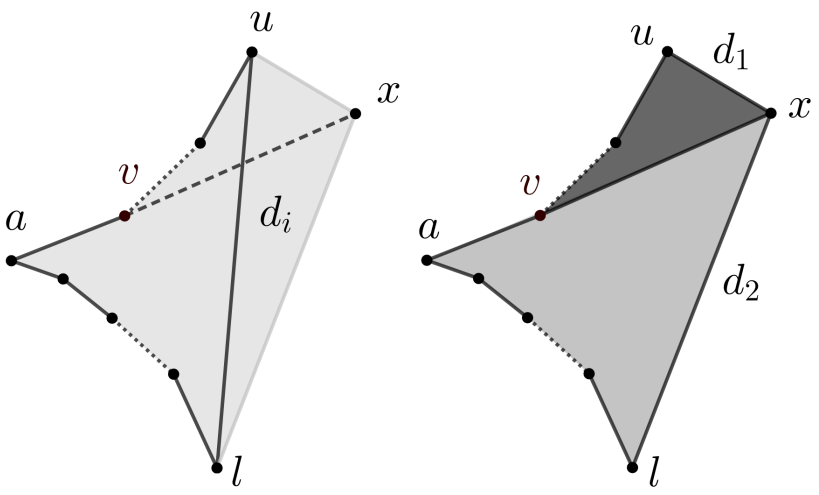


Figure 5.4: *Funnel associated with* $\overline{ul}$ *gets split into a funnel associated with* $d_1 = \overline{ux}$ *and another associated with* $d_2 = \overline{xl}$.

We begin by looking at a triangle Δsab in $\mathcal{T}$ that contains our source vertex s, $\overline{ab}$ will then be the first diagonal we look at, starting with a funnel $F = [a, s, b]$, with s as its apex. Our tree will be represented by the global pointer *predecessor* associated with each vertex of P, where **predecessor**(v) is the parent of v in the shortest path tree. We then initialize the predecessors of a and b as s and continue on recursively by calling **splitFunnel()**, which receives a diagonal d of P, a funnel F associated with d and the index i of the apex of F as parameters, to compute the predecessor field of the remaining vertices of P by splitting F into two new funnels (see figure 5.4).

Looking at the geometric dual tree $\mathcal{T}^D$ of the triangulation $\mathcal{T}$, by taking the triangle at which we have started the algorithm as the root, we can see that the algorithm will perform a *depth-first search* on $\mathcal{T}^D$, where for each node, we look at an opposing vertex, compute its predecessor, and then continues on until we hit the leaves (which are the vertices with no opposing vertex).

The initialization and its recursive procedure are shown below:

Algorithm 14 Receives source vertex s and triangulation $\mathcal{T}$ and calls the recursive function to compute the single source shortest path tree.

function RECURSIVESSSP(s, $\mathcal{T}$):
    $d \leftarrow$ edge of a triangle that contains s where s is not one of its endpoints.
    $a, b \leftarrow$ endpoints of d.
    $F \leftarrow [a, s, b]$
    predecessor(s) $\leftarrow \emptyset$ ▷ Since s is the source vertex.
    predecessor(a) $\leftarrow s$
    predecessor(b) $\leftarrow s$
    splitFunnel(d, F, 1) ▷ Since $f_1 = s$ is the apex of F.

Algorithm 15 Receives diagonal d, funnel F and index i of F to compute the predecessor of vertex x opposite to d.

function SPLITFUNNEL(d, F, i):
    if d is a boundary edge **then return**
    $u, l \leftarrow$ endpoints of d. ▷ $f_0 = u$ and $f_{n-1} = l$.
    $d_1 \leftarrow$ **previous**(d)
    $d_2 \leftarrow$ **next**(d)
    $x \leftarrow$ endpoint of the triangle Δuxl that does not intersect the interior of F.
    $k \leftarrow$ index of the closest vertex to f_i in F such that $\overline{f_k x}$ is tangent to the interior of the funnel at f_k.
    $v \leftarrow f_k$
    predecessor(x) $\leftarrow v$
    if $0 < k < i$ **then** ▷ v is inside the upper chain.
        $F \leftarrow [f_0, \ldots, f_k, x]$
        splitFunnel(d_1, F, k)
        $F \leftarrow [x, f_k, \ldots, f_{n-1}]$
        splitFunnel(d_2, F, $i - k + 1$)
    else if $i \leq k < n - 1$ **then** ▷ v is inside the lower chain.
        $F \leftarrow [f_0, \ldots, f_k, x]$
        splitFunnel(d_1, F, i)
        $F \leftarrow [x, f_k, \ldots, f_{n-1}]$
        splitFunnel(d_2, F, 1)

Algorithm 15 Part 2 of splitFunnel()

```
20:     else if k = 0 then                ▷ v is the first vertex of the upper chain.
21:         F ← [x, f_0, ..., f_{n-1}]
22:         splitFunnel(d_2, F, i + 1)
23:         F ← [f_0, x]
24:         splitFunnel(d_1, F, 0)
25:     else if k = n - 1 then            ▷ v is the last vertex of the lower chain.
26:         F ← [f_0, ..., f_{n-1}, x]
27:         splitFunnel(d_1, F, i)
28:         F ← [x, f_{n-1}]
29:         splitFunnel(d_2, F, 1)
```

Lines 10-19 of algorithm 15 refer to the case where our vertex $v = f_k$ is inside the upper or lower chains. If v is inside the upper chain, since we are removing every vertex in the range $[0, k-1]$ from the beginning of the funnel and replacing for x, the index of the apex gets shifted from i to $i-k+1$ (line 14). If v is in the lower chain, we remove vertices in the range $[k+1, n-1]$ and replace for x, and thus the index of the apex remains the same (line 17). Lines 20-29 refer to the cases where F gets split into a degenerate funnel or when F was already degenerate.

5.3 Correctness and Complexity Analysis

Firstly we will prove the correctness of our algorithm. We will do this by induction on the number of nodes of the geometric dual tree that have yet to be processed.

Theorem 5.3.1. *Algorithm 15 correctly computes a single source shortest path tree of P.*

Proof:

Basis step: When there are $n = 0$ nodes left to be processed in the dual

tree, this means all vertices have their predecessor field computed, and so we have a shortest-path tree.

Induction Hypothesis: There is an arbitrary integer value $k > 0$ such that for every dual tree with $n < k$ nodes the algorithm correctly computes the shortest-path tree.

Inductive Step: Assume that we are at a point where there are exactly k nodes left to be processed. We compute the predecessor of the opposing vertex associated with the current node and recursively call the function on the children of the node we just processed, each of these subtrees has less than k nodes left to process. Hence, by our induction hypothesis, the algorithm will compute the predecessor fields correctly for these subtrees, these predecessors along with the ones we already processed gives us a shortest-path tree for these k nodes. □

Let us now focus the discussion on the total time complexity of the algorithm, our approach is heavily influenced by the analysis found in [Gho07]. The lines where we would need to alter the funnel F before a recursive call are all easily handled with the deque structure we have previously discussed. For example, line 11 of algorithm 15 can be achieved by calling **split**(f,k) on F, followed by an **add**(b,x). We can then call **undo**() twice after the recursive call and are left with the original F.

Let us consider an amortized analysis of the total run time of the algorithm: lines 2–6 are $O(1)$ operations, so it is easy to see that if the total time to find the tangent vertex v (line 7) and split these funnels at the tangent vertex (before every recursive call) is $O(n)$ over all funnels then the algorithm itself will have total complexity $O(n)$. Let us then consider now some different approaches to finding the point of tangency of our funnel and their respective complexity.

Suppose that we find the index k by doing a linear search in F starting at one of its endpoints, the cost of finding index k would then be proportional to the number of vertices traversed in F. Suppose that the index of tagency k is $n-2$ for a certain funnel F, suppose further that our linear search begins at 0. The cost of finding k would be $n-2$. If this situation occurs repeatedly

along its execution, the total time for splitting all funnels would be $O(n^2)$ (see figure 5.5).

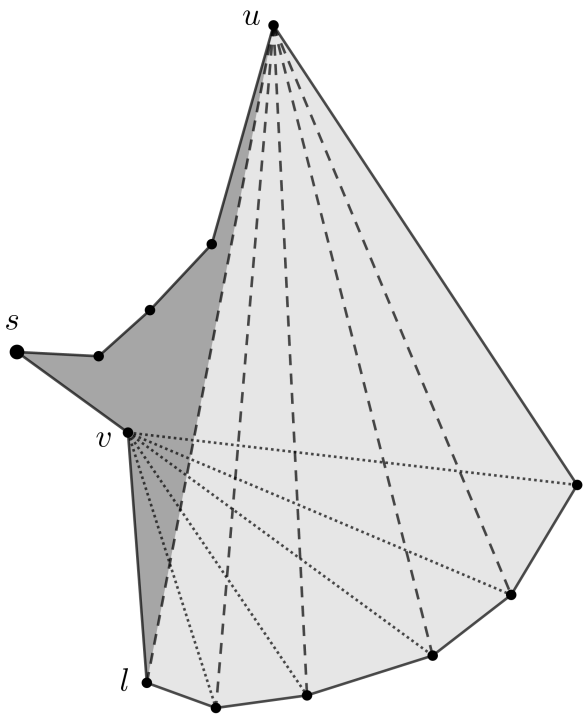


Figure 5.5: *Example of a polygon and triangulation where a naive linear search takes time* $O(n^2)$.

Let us now consider a special case. Assume that the geometric dual $\mathcal{T}^D$ of our triangulation $\mathcal{T}$ is such that the maximum degree in $\mathcal{T}^D$ is 2. This means that every face of $\mathcal{T}$ has a boundary edge forming it (see figure 5.6). And so, when splitting a funnel $F = [u = f_0, \ldots, a = f_i, \ldots, l = f_{n-1}]$, either $\overline{ux}$ or $\overline{lx}$ is a boundary edge, where x is the opposing vertex, and thus the funnel associated with it requires no further splitting (line 2 of algorithm 15). If $\overline{ux}$ is a boundary edge, we can begin the linear search at u. This search would then be proportional to the distance of u to $v = f_k$ in F, but since F_{ux} has no opposite vertex, the vertices of F_{ux} are not considered again in the execution of the algorithm. The same can be said if $\overline{lx}$ is a boundary edge, by beginning the search on l. By choosing the starting vertex as such, we can bound the total complexity to $O(n)$ in this case.

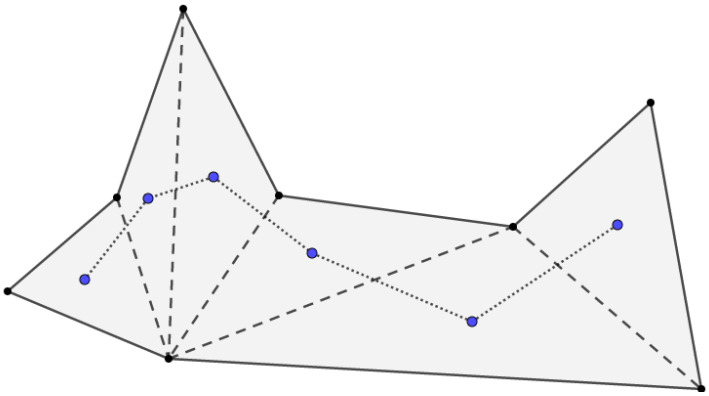


Figure 5.6: *Geometric dual of a triangulation with maximum degree 2.*

Finally, we are going to show that by doing a binary search to find the point of tangency we can achieve a linear time in arbirtrary simple polygons. This approach is heavily dependent on having a suitable representation of the funnel structure, more specifically having a structure that splits the funnel in a reasonable amount of time.

Suppose we had a list representing the funnel F stored in a search tree, where every vertex of F pointed to its neighbors in the list. We could perform a binary search to look for the vertex v such that $\overline{vx}$ is tangent to the funnel by looking at the slopes of the edges incident to a certain vertex f_i and comparing them to the slope of $\overline{f_i x}$, in constant time we would be able to decide in which side of f_i should our search continue looking for v (see figure 5.7). Therefore this structure would support searching for v in $O(\log |F|)$, since there are at most $O(n)$ splitting operations in P, the total time for splitting all funnels would be $O(n \log n)$.

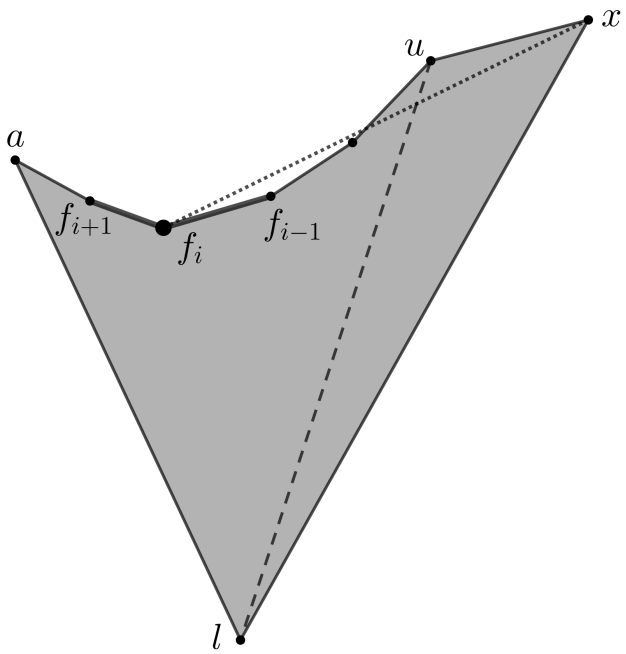


Figure 5.7: *Since the slopes of the edges incident at f_i are smaller than the slope of $\overline{f_i x}$, the binary search would continue on the left of f_i.*

As mentioned previously, in our implementation we are using a deque with history stack structure. This structure allow us to search for v from both ends of F at the same time by following the procedure we have described for the binary search. This brings to a $O(\min(\log d, \log(|F| - d)))$, where d is the size of the upper (or lower) chain, complexity for splitting the funnels. *Finger search trees* were used in [Gui+87] to achieve the same complexity, but we have decided against to using it avoid the extra pointer complexity required to implement it.

To bound the time for splitting all funnels with this approach, we argue the following: Let $\mathcal{T}^D$ be the geometric dual of the triangulation $\mathcal{T}$. Then $\mathcal{T}^D$ has $n - 2$ nodes. If the source vertex s lies in only one triangle of $\mathcal{T}$, we take this triangle as the root of $\mathcal{T}^D$. Otherwise, there exists at least one triangle incident to s which has a boundary edge forming it, we take this triangle as the root. Thus each node of $\mathcal{T}^D$ has 0, 1 or 2 children. Our algorithm performs a depth-first traversal of $\mathcal{T}^D$.

Note that the cost of processing a node is $O(\min(\log d, \log(m-d)))$, where

$|F| = m$. Let $C^*(m)$ denote the total cost of processing the subtree rooted at the node of the triangle containing x, the opposing vertex. So,

$$C^*(m) = C^*(d) + C^*(m-d) + O(\min(\log d, \log(m-d))).$$

Where $C^*(d)$ and $C^*(m-d)$ represent the cost of processing the trees rooted on the children of x. Taking all possible partitions of m, we get:

$$C^*(m) = \max_{1 \le d \le m-1} [C^*(d) + C^*(m-d) + O(\min(\log d, \log(m-d)))].$$

It can be show by induction on m that $C^*(m)$ is maximum for $d = \frac{m}{2}$. Therefore $C^*(m) \le C^*(m/2) + C^*(m/2) + O(\min(\log m/2, \log m/2)) = 2C^*(m/2) + O(\log m/2)$. So,

$$C^*(m) \le 2C^*(m/2) + O(\log m/2) \le 2C^*(m/2) + O(\log m).$$

By applying the master theorem in this last expression we get $C^*(m) = O(m)$. So the total time for splitting all funnels is $O(n)$. This is summarized in the following theorem:

Theorem 5.3.2. *The shortest path tree rooted at a point of a polygon P with n vertices can be computed in $O(n)$ time.*

Chapter 6

Single Source Shortest Path Map

6.1 Introduction

In the last section we have described the *Single Source Shortest Path Tree*: a data structure that can be built in linear time for finding the euclidean shortest paths from a given vertex to every other vertex in a simple polygon also in linear time. In this chapter we extend this idea with the *Single Source Shortest Path Map*, a data structure that helps us efficiently find the euclidean shortest path from a given source vertex s to every point inside a given polygon P (not only to its vertices).

The shortest path map with source s (SPM_s) is a planar subdivision of the polygon P to smaller convex polygons that we will refer to as *cells*. Every cell is associated with a *parent vertex*, which is the closest vertex to s inside that cell. Let v be the parent of a given cell C, every point p inside C has the property that the shortest path from s to p is composed by the shortest path from s to v ($\pi(s, v)$) and the segment $\overline{vp}$, that is, $\pi(s, p) = \pi(s, v) \cup \overline{vp}$ (see figure 6.1). Since we need to compute the shortest path from s to the parents of the cells, we will assume that we have already built a shortest path tree from s.

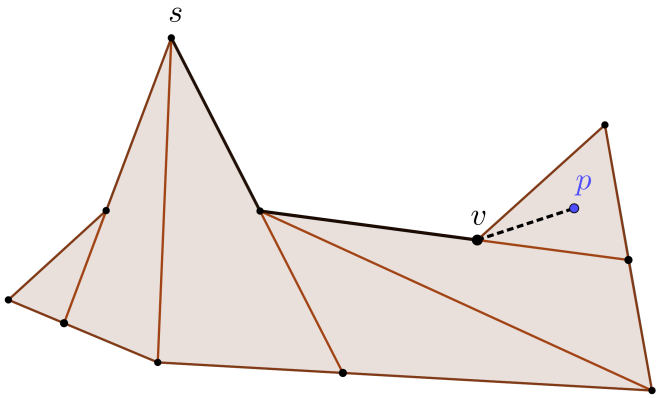


Figure 6.1: *In this example of a shortest path map v is the parent of the cell containing p, and the shortest path from the source s to p has been highlighted.*

We can then formulate this problem as such:

Single Source Shortest Path Map

Input: Simple polygon P with n vertices, source vertex s of P and the shortest path tree of s in P.

Output: The shortest path map of P with source s.

We will now describe how given the shortest path tree of the polygon P with n vertices we can divide the interior of P in $O(n)$ regions in $O(n)$ time. The construction is as follows: consider the polygon P given by a list of its edges in counter-clockwise order. For every edge $e = \overline{ul}$, we take the funnel $F_e = F_{ul} = [u = f_1, \ldots, f_{k-1}, f_k = a, f_{k+1}, \ldots, f_m = l]$, where a is the apex of F_e. This funnel can be easily constructed by following the predecessor fields of the shortest path tree. If the funnel F_e is degenerate we skip to the next edge, otherwise, we take the edge $\overline{f_i f_{i+1}}$ and calculate the intersection point x of $\overline{f_i f_{i+1}}$ and $\overline{ul}$. If $i < k$, that is, if f_i is on the upper chain, we add the edge $\overline{f_i x}$ to the map, otherwise, we add the edge $\overline{f_{i+1} x}$, in either case f_i will be the parent vertex of the triangular region created by the addition of the new edge. Particularly, when $i = m - 1$, the intersection point of $\overline{ul}$ and $\overline{f_{m-1} f_m}$ is $f_m = u$ and so we do not add a new edge, but we still identify f_{m-1} as the parent of the triangle with $\overline{f_{m-1} f_m}$ as one of its edges (see figure 6.2).

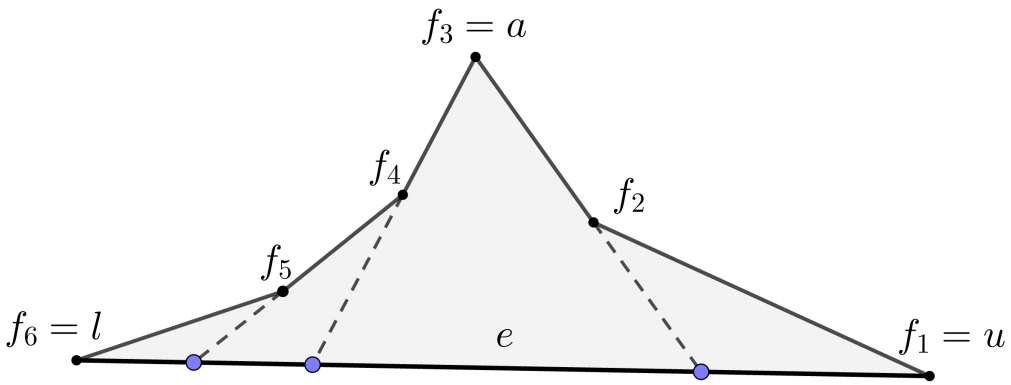


Figure 6.2: *One step in the construction of the shortest path map.*

At the end of these steps we will be left with $|F_{ul}| - 2$ triangular regions inside of F_{ul}. If a certain vertex p is inside a triangular region with parent v, the shortest path from source s to p will be tangent to the funnel at point v, so $\pi(s, p) = \pi(s, v) \cup \overline{vp}$. The polygon will be divided into $O(n)$ such triangular regions with the property we have described and thus these regions will make up the cells of the single source shortest path map.

Some of the triangular regions created by this method will be composed of 4 or 5 vertices (see triangles with parents f_3 and f_4 in figure 6.2), and hence the shortest path map would not be considered a proper triangulation. So to make our definitions consistent over the next section, we will divide these triangular regions into several triangles with the same parent vertex. So if we have a certain triangular region T with parent vertex p, we subdivide T into two triangles with parent vertex p each, if T had 4 vertices, and three triangles with parent vertex p each, if T had 5 vertices. Since these regions are convex, we can easily triangulate them by taking a vertex and adding the remaining edges from it to the other vertices (see figure 6.3).

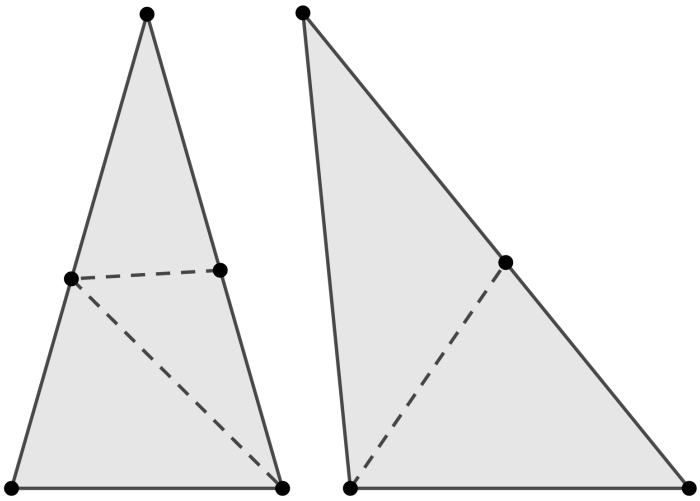


Figure 6.3: *Example of the subdivision of the triangular region into new triangles.*

6.2 Point Location

With the shortest path map in hand, we now need a way to organize these cells so that we can quickly determine to which cell a certain query point belongs. We will achieve this by performing a hierarchical subdivision search, as first described by Kirkpatrick [Kir83].

We begin by constructing a planar graph G with the same vertices as the shortest path map. If the shape of the boundary of P is not triangular, we add three new vertices a, b, and c such that P is entirely contained in the triangle Δabc and add this triangle to the planar graph (see figure 6.4). If the boundary of P is already triangular, we will refer to the 3 outermost vertices as a, b, and c. We will build a sequence $\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_k$ of triangulations such that $\mathcal{T}_0$ is the triangulation of G, $\mathcal{T}_k$ is composed of only the triangle Δabc, and every triangle in $\mathcal{T}_{i+1}$ intersects $O(1)$ triangles in $\mathcal{T}_i$.

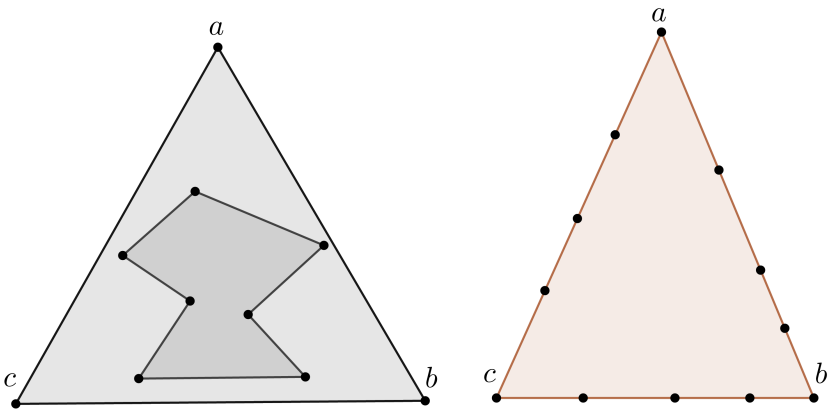


Figure 6.4: *Examples of polygons with and without triangular boundaries.*

We begin with the triangulation $\mathcal{T}_0$ with $O(n)$ triangles and we will make vertex deletions and retriangulations until we achieve $\mathcal{T}_k$. At every step we will remove a fraction of vertices from $\mathcal{T}_i$ and then will retriangulate the holes left by this deletion, leaving us with triangulation $\mathcal{T}_{i+1}$ (see figure 6.5).

If a certain vertex v has degree d in a triangulation, the deletion of v creates a hole that can be filled with $d-2$ new triangles that will intersect at most d triangles of the previous triangulation $\mathcal{T}_i$. By choosing an independent vertex set for deletion, the holes created will also be independent from each other.

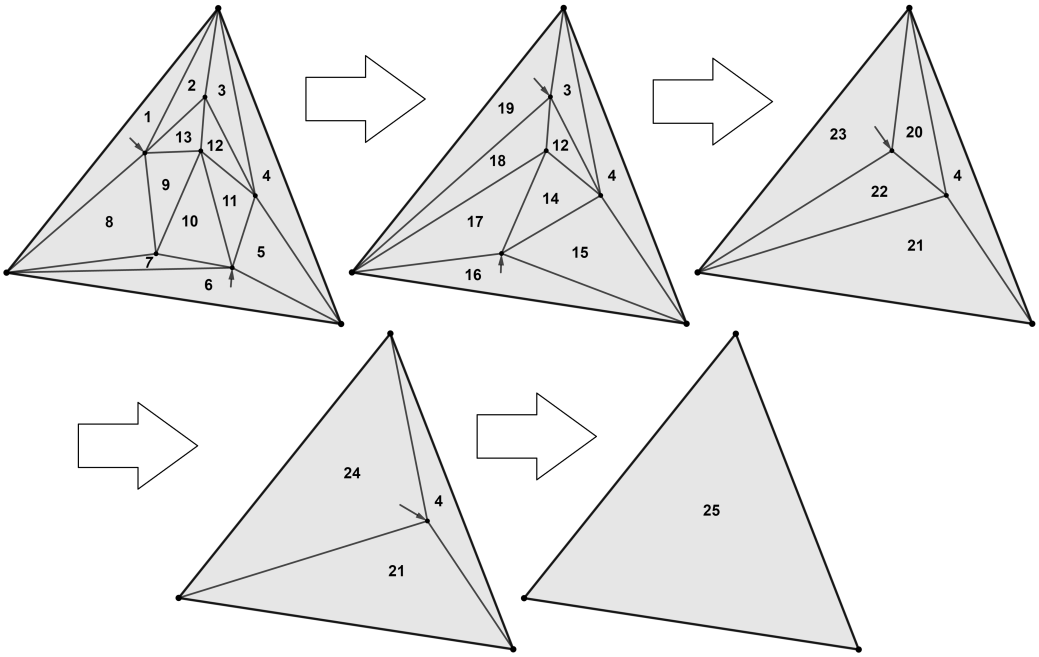


Figure 6.5: *Succession of vertex deletions and their resulting retriangulations.*

We are now ready to explain the general step of this subdivision: with triangulation $\mathcal{T}_i$ in hand, we find an independent set of vertices and remove them from the graph. Each deletion generates a cycle of length given by the degree of the deleted vertex. We call it a *hole*. We then retriangulate each of these holes. For each new triangle created this way in $\mathcal{T}_{i+1}$, we add a pointer from it to every triangle of the previous triangulation $\mathcal{T}_i$ that it intersects, this can be verified in $O(1)$ time. Ideally, we would like to find the largest independent set possible, to minimize the number of iterations in the process, but this problem is known to be NP-hard [GJ90]. So instead we will just use the largest independent set we can find with limited degree vertices (the procedure is explained in lemma 6.3.2). Since the vertices we will remove have limited degree, checking the intersection between old and new triangles will take constant time, which otherwise would be quadratic in regards to the degree.

At the end of this process, that is, after we reach triangulation $\mathcal{T}_k$, we will insert the information we have gathered in a *directed acyclic graph* (DAG). The root of the DAG will be the triangle Δabc of triangulation $\mathcal{T}_k$, the nodes

of the next level will be the triangles of triangulation $\mathcal{T}_{k-1}$, continuing similarly for the remaining levels. Every node in the level corresponding to the triangulation $\mathcal{T}_i$ has a pointer to a node in the next level if the triangles they represent intersect each other (see figures 6.5 and 6.6).

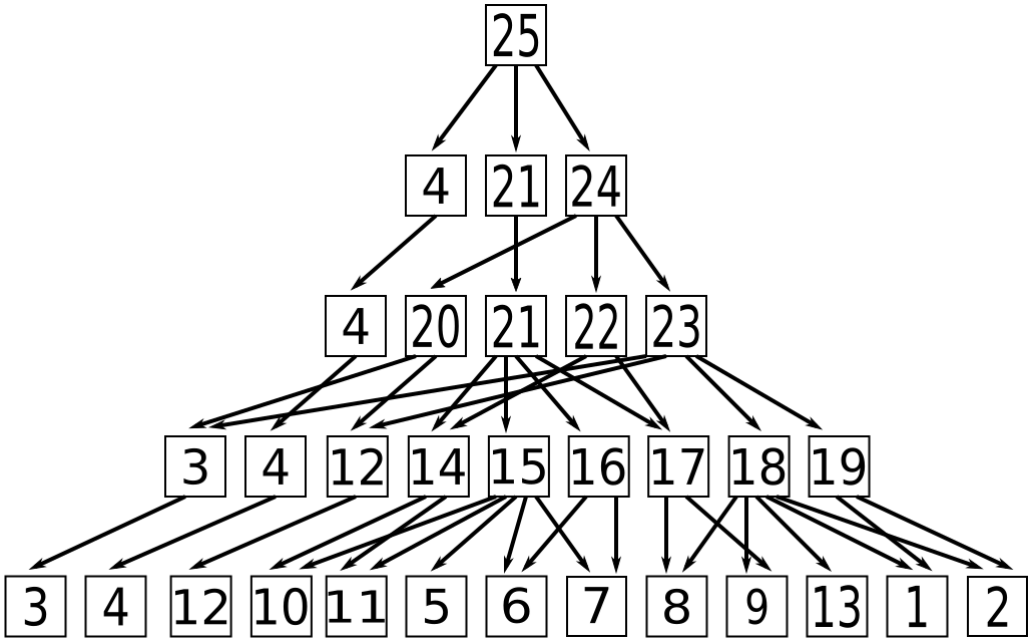


Figure 6.6: *DAG representation of the example in figure 6.5.*

Finally, we will explain how to find the cell that contains a certain point x in this DAG. First we check if x belongs to the root triangle, if it does not then x also does not belong to P. Then, at each level we know to which triangle x belongs in this level. We then have to check for the triangles in the next level, adjacent to this triangle. Note that the number of triangles to be checked in a given level is limited by the size of the independent set removed in the construction times the maximum degree in it. We do so until we reach a leaf node which will be the cell of the map that contains x. The time complexity of the search is then proportional to the total height of the DAG, we will show in the next section that by removing an independent set of small degree vertices we will end up with a logarithmic height (this is shown in theorem 6.3.3), making the total search time $O(\log n)$. Thus, if we keep the information of the distance to the source at each parent vertex,

we can find in $O(\log n)$ time the distance of the query point to the source. If instead we are interested in the path itself, the total complexity would be $O(\log n + k)$, where k is the total number of vertices in the path.

6.3 Complexity Analysis

Lemma 6.3.1. *In a plane triangulation, at least $\frac{|V|}{2}$ vertices have degree less than 12.*

Proof: Let E be the edge set of the triangulation. Suppose for contradiction that at most $\frac{|V|}{2}$ vertices have degree less than 12. Then,

$$|E| = \frac{1}{2}\sum_{v \in V} \text{degree}(v) \geq \frac{1}{2}\left(\frac{|V|}{2}12\right) = 3|V| \implies 3|V| \leq |E|.$$

Applying Euler's formula to a planar triangulation we get that $|E| = 3|V| - 6$. Then,

$$3|V| \leq |E| = 3|V| - 6,$$

which is a contradiction. This means that at least $\frac{|V|}{2}$ vertices must have degree less than 12. □

Lemma 6.3.2. *In a plane triangulation, we can find $\frac{|V|}{24}$ independent vertices with degree less than 12 in linear time.*

Proof: We will do so with a greedy algorithm. Initially the independent set is empty and all vertices are unmarked. For each vertex, if it is not on the set and it has degree less than 12, we add it to the set and mark its neighbors. If a vertex is marked or has degree 12 or higher, we do nothing. Clearly, this procedure builds an independent set, since any time we add a vertex we exclude its adjacent vertices. It is also clear from the construction that all of the vertices in the set have degree less than 12. By the preceding lemma, there are $\frac{|V|}{2}$ vertices with degree less than 12 and each time we add a vertex

to the set, we exclude at most 11 others. Thus, we will be able to find $\frac{|V|}{24}$ such vertices. □

Theorem 6.3.3. *We can find the distance of a point to the source by searching in the DAG in $O(\log n)$ time and find the path itself in $O(\log n + k)$ time, where k is the number of vertices in the path.*

Proof: Let n_i be the number of vertices at level i of the DAG. The root, which is the level representing $\mathcal{T}_k$ has $n_0 = O(1)$. Each subsequent level grows by $\frac{24}{23}$ until size reaches $O(n)$. Thus, the total height is $\log_{24/23} O(n) = O(\log n)$. This means that the total time to find the cell containing a query point is $O(\log n)$, so by adding the distance from the parent vertex to the query point to the distance from the parent to the source, we have found the distance from source to the query point in time $O(\log n)$, by following back the predecessor pointers from the parent vertex to the source, we can find the path itself in $O(\log n + k)$, where k is the size of the path. □

Theorem 6.3.4. *The preprocessing time and space are $O(n)$.*

Proof: We can construct the planar graph G and triangulate it in linear time. In theorem 6.3.3 we have shown that each level of the DAG increases in size by $\frac{24}{23}$. So, it suffices to prove that the time required to build a level of the DAG is linear to the number of vertices in that level. At each step in the construction of the DAG we perform the following tasks:

1. We find an independent set of low degree vertices. This is proven to be $O(n)$ by lemma 6.3.2.

2. We retriangulate the holes left by the deletion of the vertices in the independent set. This is $O(n)$ since each hole has constant size and there are $O(n)$ of them, since each hole originates from a vertex deletion.

3. We check the intersections and add the corresponding pointers to the DAG. This is also $O(n)$ because each new triangle has constant degree and there are $O(n)$ of them.

It follows that the total time and total space are $O(n)$. □

Chapter 7

Conclusion

The development of this text was motivated initially by the work of Maheshwari et al. [Mah+18], which deals with single source shortest paths that violate the boundary of a given polygon a limited amount of times. Their algorithm for this relaxation of the problem is very intricate since it uses very complex structures, like the *hourglass* structure they describe. Even so, it still builds upon the classical form of this problem, so we have deemed the classical problem to have enough depth on its own, having interesting algorithmic solutions. Hopefully that is clear from the descriptions we have provided throughout the text. Especially the single source shortest path tree, which elegantly computes these paths recursively by extending a funnel structure at each step. We have also described how this structure can easily be extended to the single source shortest path map, which allows us to query paths to points inside the boundary in logarithmic time, as shown in [LP84], albeit in a more complicated way, since it deals with obstacles inside the polygon. Moreover, we have found that many of the papers that present these algorithms have the tendency to present pseudocode descriptions that require a lot of work in its translation to computer code, so helping alleviate these issues became our main purpose throughout the text.

Let us now highlight some ideas for topics of study that could be derived from our text. First off, one could study the variant of the problem we have mentioned above, where violations of the boundary are allowed. Along the text we always dealt with simple polygons, so another extension would be to consider polygons with holes. One could also consider these shortest path problems in association with an additional cost function when you do violate the polygon: is it always "cheaper" to violate it? It would depend on the cost function chosen. Another natural application would be to consider robot motion planning, where given a mobile robot we would like to find a

path from a source point to a specified destination while avoiding obstacles, including variations where we consider limited turning angles or have to consider the dimensions of the robot when finding a path.

All the algorithms described in this text have been implemented in *Python*. The source code is available at the repository mlordx/masters-implementations on GitHub.

References

[Ber+08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. 3rd ed. Springer-Verlag TELOS, 2008 (cited on pages 17, 20, 32).

[CCT91] Kenneth L. Clarkson, Richard Cole, and Robert E. Tarjan. "Randomized Parallel Algorithms for Trapezoidal Diagrams". In: *Proceedings of the Seventh Annual Symposium on Computational Geometry*. SCG '91. New York, NY, USA: ACM, 1991, pages 152–161 (cited on page 13).

[Cha91] Bernard Chazelle. "Triangulating a simple polygon in linear time". In: *Discrete & Computational Geometry* 6.3 (September 1991), pages 485–524 (cited on page 13).

[CTV89] Kenneth L. Clarkson, Robert E. Tarjan, and Christopher J. Van Wyk. "A fast Las Vegas algorithm for triangulating a simple polygon". In: *Discrete & Computational Geometry* 4.5 (October 1989), pages 423–432 (cited on page 13).

[Dij59] E. W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1.1 (December 1959), pages 269–271 (cited on page 40).

[Gho07] Subir Kumar Ghosh. *Visibility Algorithms in the Plane*. Cambridge University Press, 2007 (cited on page 49).

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990 (cited on page 59).

[Gui+87] Leonidas Guibas, John Hershberger, Daniel Leven, Micha Sharir, and Robert E. Tarjan. "Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons". In:

Algorithmica 2.1 (November 1987), pages 209–233 (cited on pages 2, 52).

[HS91] John Hershberger and Jack Snoeyink. "Computing minimum length paths of a given homotopy class". In: *Algorithms and Data Structures*. Springer, 1991, pages 331–342 (cited on page 45).

[Kir83] David Kirkpatrick. "Optimal Search in Planar Subdivisions". In: *SIAM Journal on Computing* 12.1 (1983), pages 28–35 (cited on pages 2, 57).

[KKT90] David G. Kirkpatrick, Maria M. Klawe, and Robert E. Tarjan. "Polygon Triangulation in $O(n \log \log n)$ Time with Simple Data-structures". In: *Proceedings of the Sixth Annual Symposium on Computational Geometry*. SCG '90. ACM, 1990, pages 34–43 (cited on page 13).

[KS98] J. Mark Keil and Jack Snoeyink. "On the Time Bound for Convex Decomposition of Simple Polygons". In: *Int. J. Comput. Geometry Appl.* 12 (1998), pages 181–192 (cited on page 17).

[LP76] Der-Tsai Lee and Franco P. Preparata. "Location of a Point in a Planar Subdivision and Its Applications". In: *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*. STOC '76. Hershey, Pennsylvania, USA: ACM, 1976, pages 231–235 (cited on page 17).

[LP84] Der-Tsai Lee and Franco P. Preparata. "Euclidean shortest paths in the presence of rectilinear barriers". In: *Networks* 14.3 (1984), pages 393–410 (cited on pages 2, 41, 42, 63).

[Mah+18] Anil Maheshwari, Subhas C. Nandy, Drimit Pattanayak, Sasanka Roy, and Michiel Smid. "Geometric Path Problems with Violations". In: *Algorithmica* 80.2 (February 2018), pages 448–471 (cited on page 63).

[MP78] D.E. Muller and F.P. Preparata. "Finding the intersection of two convex polyhedra". In: *Theoretical Computer Science* 7.2 (1978), pages 217–236 (cited on page 4).

[ORo98] Joseph O'Rourke. *Computational Geometry in C*. 2nd edition. Cambridge University Press, 1998 (cited on page 10).

[PS85] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction*. Berlin, Heidelberg: Springer-Verlag, 1985 (cited on page 2).

[Sei91] Raimund Seidel. "A Simple and Fast Incremental Randomized Algorithm for Computing Trapezoidal Decompositions and for Triangulating Polygons". In: *Comput. Geom.* 1 (1991), pages 51–64 (cited on page 13).

[Sha89] Micha Sharir. "Algorithmic Motion Planning in Robotics". In: *Computer* 22.3 (March 1989), pages 9–20 (cited on page 2).

[TJ88] Robert Tarjan and Christopher J. Van Wyk. "An $O(n \log \log n)$-Time Algorithm for Triangulating a Simple Polygon". In: *SIAM J. Comput.* 17 (February 1988), pages 143–178 (cited on page 13).