Masters Thesis

Computing the Visibility Graph of Points Within a Polygon

Jesper Buch Hansen

December 22, 2005

Department of Computer Science, University of Aarhus

Student	Supervisor
Jesper Buch Hansen	Gerth Stølting Brodal

Contents

1 Introduction **9**
1.1 Previous Study 10
1.2 The Algorithms 11

2 Trapezoidal Graph **13**
2.1 Trapezoidal Map 13
2.2 Point Location Preprocessing 17

3 Decomposition and Factor Graph **23**
3.1 Decomposition of a Tree 23
3.2 Decomposition of the Trapezoidal Graph 29
3.3 Factor Graph 30

4 Query Structure **33**
4.1 Hourglasses 34
4.2 Convex Chains 37

5 Visibility Graph of a Simple Polygon **41**
5.1 Dual Plane 41
5.2 Association of Rays 42
5.3 Visibility Edge Computation 43

6 Visibility Graph of a Polygon with Holes **45**
6.1 Other Problems with Holes 45
6.2 Visibility Graph 46

7 Range-restricted Visibility Graph **49**
7.1 Segment Trees 50
7.2 Multi-Level Data Structures 51
7.3 Computing the Visibility Edges 52

8 Invisibility Graph **55**
8.1 Sector Range Algorithm 55
8.2 Improved algorithm 56

9 Future Work **59**

A Notation **61**

List of Figures

1.1 An example of a visibility graph of points within a polygon . . . 9
1.2 An example of a range-restricted visibility graph 10
1.3 Running times of known and new algorithms 11

2.1 A trapezoidal map with three types of polygon vertices 14
2.2 An example of a balanced decomposition 14
2.3 The merge algorithm 15
2.4 A trapezoidal graph after a merging process 16
2.5 Degenerate case: Two sites with the same x-coordinate 16
2.6 Degenerate case: Two polygon vertices with the same x-coordinate 17
2.7 The point location structure 18
2.8 The point location algorithm 18
2.9 The Kirkpatrick algorithm to build the data structure 20

3.1 The auxiliary tree algorithm 24
3.2 The build tree algorithm 24
3.3 Labels on a trapezoidal graph 25
3.4 An example of an auxiliary tree 28
3.5 An example of a factor graph 30

4.1 An example of an hourglass 33
4.2 Examples of an open and a closed concatenation 35
4.3 The hourglasses algorithm 36
4.4 A derived chain 37
4.5 Possible intersections of chain points 38
4.6 An example of chain intersections 39

5.1 A line through two sites 41
5.2 A diagonal not crossing the line between two sites 42
5.3 Ray intervals of a site 42
5.4 The ray association algorithm 43
5.5 The visibility algorithm 43

6.1 Junction triangles and corridors of a polygon 47
6.2 The ray association algorithm for polygons with holes 47
6.3 The visibility algorithm for polygons with holes 48

7.1 The sector range of a site through a part of a diagonal 49
7.2 The query algorithm of the segment tree 51

7.3 The insertion algorithm of the segment tree 51
7.4 The canonical segment of a site 52
7.5 The visibility algorithm of sites with restricted sight 53

8.1 The invisibility algorithm 56
8.2 An example of two half-plane range queries from a site 57

Abstract

The subject of this thesis is the problem of computing the visibility graph of m points P within a polygon Q. The vertices of the visibility graph are the points of P and the edges of the graph are the pairs of points in $P \times P$, where the line between them is fully contained inside Q. Known algorithms by Ben-Moshe et al. are presented for computing the visibility graph of a simple polygon, the visibility graph of a polygon with holes, the range-restricted visibility graph and the invisibility graph. Finally, a new improved algorithm to compute the invisibility graph is presented. This algorithm improves the running time by a factor $O(m^{1/2})$ of the, so far, best algorithm.

Chapter 1

Introduction

The visibility graph is very fundamental in computational geometry, and the problem of computing it has a lot of special cases depending on the type of polygon, the type of visibility, the application etc. It is applied mostly in computer graphics, for example illumination and rendering, but also in motion planning, where the problem is to find a route from A to B among a set of obstacles. Some less obvious applications include pattern recognition and sensor networks.

The problem of computing the visibility graph is defined as follows: Let Q be a polygon having n vertices V and consider a set P of m points (*sites*) inside Q or on the boundary of Q. Then the *visibility graph* $VG_Q(P)$ of P in Q is the graph with P as vertices and an edge between two sites s and t, if the line segment $\overline{st}$ lies within Q (Fig. 1.1). We then say that s and t *see* each other within Q. The number of edges in $VG_Q(P)$ is denoted k.

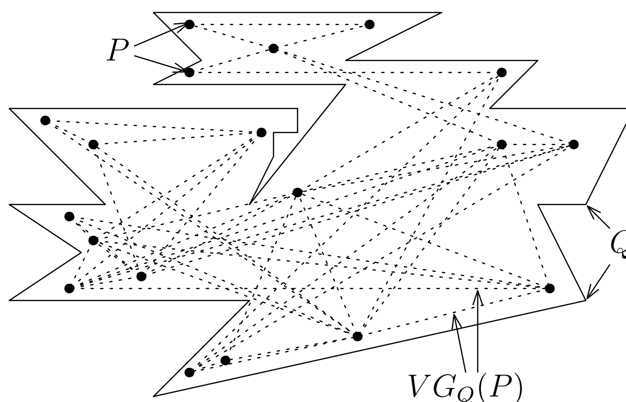

Figure 1.1: A visibility graph $VG_Q(P)$ of 19 points P within a polygon Q

Two variants of the visibility graph is the range-restricted visibility graph and the invisibility graph. The *range-restricted visibility graph* $\overline{VG}_Q(P)$ is a directed graph with the same vertices as the normal visibility graph. There is an edge from s to t if and only if s can see t and $\overline{st} \leq d_s$ where d_s is an associated range of sight of s (Fig. 1.2). The *invisibility graph* is the complement of the visibility graph. I.e., it has the same vertices as the visibility graph, but there

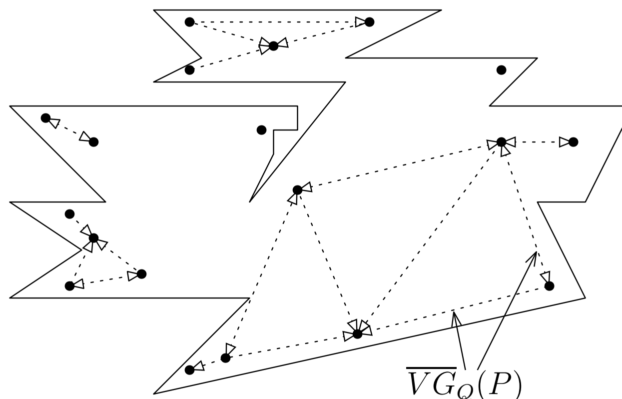

Figure 1.2: An example of a range-restricted visibility graph $\overline{VG}_Q(P)$

is and edge between s and t if and only if s can not see t.

Other related work with visibility graphs includes the problem of computing the visibility graph of a set of disjoint polygonal obstacles. This problem has an optimal solution running in $O(n \log n + k)$ time, where n is the total number of edges of the obstacles and k is the number of edges of the output graph [GM91].

Ben-Moshe et al. [BHKM04] present other cases of the visibility problem: The robust visibility problem, where two sites are robustly visible to each other if they can be moved within a given distance and still be visible, and the one-dimensional terrain problem, where it is detected if two sites are visible to each other for the case that Q is a one-dimensional terrain.

1.1 Previous Study

Many different visibility problems have been studied, and the algorithms of this thesis will be compared with the relevant ones.

One of the special cases of the problem is the computation of the visibility graph of the polygon vertices, i.e. $P = V$. This problem already has an optimal solution running in $O(n + k)$ time [H89], and we will compare our time complexity to this in the end of Chapter 5.

For the general problem of points within a simple polygon, there are two immediate solutions:

1. By looking at the points of P as polygon vertices of one vertex polygons, we can use an optimal algorithm [GM91] to compute the visibility graph of a set of polygonal obstacles. The vertices of the obstacles is then $V \cup P$, and the running time is $O((n+m)\log(n+m)+k')$, where k' is the number of visibility edges of $VG_Q(V \cup P)$. Then, only edges of $VG_Q(P)$ are reported. The problem is that too many edges could be over-reported, for example when k' is $O(n^2)$ and $|VG_Q(P)|$ is small.

2. This solution uses *ray shooting*: Given a set S of line segments and a query ray r—a half line—report the first line segment of S intersected by r. By shooting a ray between two sites, it can be detected whether

Type	Author(s)	Running time
simple	[GM91]	$O((n+m)\log(n+m) + \lvert VG_Q(P \cup V)\rvert)$
simple	[CG89]	$O(n\log n + m^2 \log n)$
simple	[BHKM04]	$O(n + m\log m \log mn + k)$
simple, $P = V$	[H89]	$O(n+k)$
h holes	[BHKM04]	$O(n + m(h\log mn + \log m \log mn) + k)$
range-restricted	[BHKM04]	$O(n + m\log m(m^{1/2}\log m + \log mn) + k)$
invisibility	[BHKM04]	$O(n + m\log m(m^{1/2}\log m + \log mn) + \overline{k})$
invisibility	**this thesis**	$O(n + m\log m \log mn + \overline{k})$

Figure 1.3: Running times of known and new algorithms

there is an edge of the polygon between the sites or not. The polygon Q is preprocessed for ray shooting queries in $O(n\log n)$ time, and the time for shooting a ray between all pairs of sites takes $O(m^2 \log n)$ time. So, this method requires $O(n\log n + m^2\log n)$ time on any output, but it is faster than the first method if $|VG_Q(V \cup P)|$ is $O(n^2)$.

Running times of known and new algorithms is presented in Fig. 1.3.

1.2 The Algorithms

This thesis gives a detailed presentation of algorithms from [BHKM04] by Ben-Moshe et al. First, we present an output-sensitive, divide-and-conquer algorithm that is nearly linear in m, n and k. The running time is $O(n+m\log m\log mn+k)$ for simple polygons and roughly multiplied by a factor of $O(h)$ when the polygon has h holes. The space complexity is $O(m+n)$ which is optimal and not output-sensitive, since visibility edges are only reported and not saved.

The main algorithm is covered in the following chapters of the thesis.

Chapter 2 Trapezoidal Map: Insertion of $O(n)$ diagonals (edges with endpoints on the polygon boundary) to decompose the polygon. Point Location Structure: A data structure to efficiently locate the region of the trapezoidal map containing a given site.

Chapter 3 Decomposition Tree: Recursive selection of diagonals that split sub-polygons in two. Factor Graph: Extension of the decomposition tree used by the query structure.

Chapter 4 Query Structure: The data structure used to detect the visible part of a diagonal from a given site. The Chain Data Structure: the main data structure used by the query structure.

Chapter 5 Ray Associations: Association of visibility rays from sites to diagonals. Visibility Edge Computation: Computation of the output edges using ray associations.

After this, **Chapter 6** describes an algorithm to compute the visibility graph of points within a polygon with holes. The idea of this algorithm is to divide

the polygon into $O(h)$ simple polygons, use the main algorithm to compute the visibility graph inside these simple polygons, and finally compute visibility edges across boundaries of the simple polygons.

In **Chapter 7**, an algorithm to compute the range-restricted visibility graph is presented. Here, the visibility graph is computed by answering range queries for each site. This algorithm runs in $O(n + m \log m(m^{1/2} \log m + \log mn) + k)$ time.

Finally, two algorithms to compute the *in*visibility graph is presented in **Chapter 8**. The first algorithm is from [BHKM04] and uses the method of the range-restricted visibility graph and thus has the same time complexity. The second algorithm is a new improved algorithm that has the same running time as the main algorithm, $O(n + m \log m \log mn + \overline{k})$. This algorithm improves the $O(n + m \log m(m^{1/2} \log m + \log mn) + \overline{k})$ time algorithm of [BHKM04] by roughly a factor $O(m^{1/2})$ for the case where $\overline{k}$ is $O(m \operatorname{polylog} m)$.

Preliminaries

The subtree of a node v is denoted $subtree(v)$. The *height* of a leaf is 0, and the height of an internal node v is one plus the maximum height of a child of v. The *depth* of the root is 0, and the depth of an internal node v is one plus the depth of the parent of v.

Chapter 2

Trapezoidal Graph

To compute $VG_Q(P)$, the polygon Q must first of all be represented as a data structure. We can then use a number of tools and other data structures to compute the visibility edges. It is assumed that Q is given as a list of vertices $(v_1, \ldots, v_n)$ in clockwise order around the polygon.

The contents of this chapter includes a trapezoidal map of the polygon, where diagonals are inserted to decompose the polygon, and modifications of the trapezoidal map by splittings and merging of trapezoids. The point location structure described afterwards is used to efficiently locate the region of the trapezoidal map containing a given site.

2.1 Trapezoidal Map

It turns out that a *trapezoidal map* of Q will suffice as a basic data structure for the polygon. In this data structure the polygon is decomposed into trapezoids or triangles by drawing vertical extensions from every vertex—upwards and downwards—inside the polygon until the extensions meet an edge or a vertex of the polygon (Fig. 2.1). If a vertex is incident to two edges on each (horizontal) side of the vertex, one extension is drawn inside the polygon until it meets an edge (case 1). A vertex incident to two edges on the same side of the vertex can occur in two ways: If the edges form an angle of the polygon of degree less than 180°, no extensions are drawn (case 2). If the angle is greater than 180°, two extensions are drawn (case 3). Because an extension has endpoints on the polygon edge, it will be called a *diagonal*. Diagonals—always vertical—are essential in the algorithm, because they hold information about which areas of the polygon a site can see.

For each trapezoid, let the left (resp. right) *definition vertex* be the vertex lying on the left (resp. right) diagonal of the trapezoid. If no diagonal is bounding the trapezoid to the left (resp. right), it is bounded by a vertex and this will be the definition vertex. See the end of this section, what happens in the degenerate case where more vertices lie on one of the diagonals.

Many algorithms build a trapezoidal map of n vertices in $O(n \log n)$ time. Chazelle shows in [C91] how the trapezoidal map can be computed in linear time. The algorithm is rather complicated and will not be described here, but the result is very important to our algorithm.

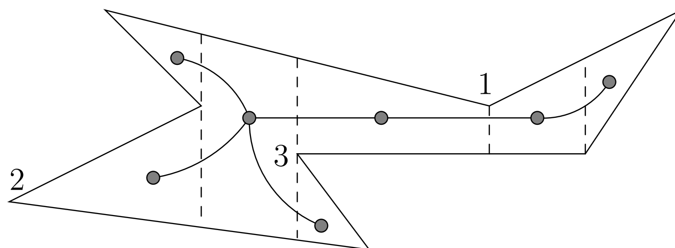

Figure 2.1: A trapezoidal map with three types of polygon vertices 1, 2 and 3

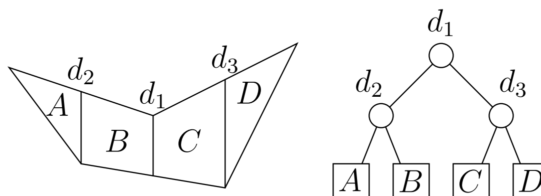

Figure 2.2: An example of a balanced decomposition

Theorem 2.1.1 *The trapezoidal map of a simple polygon with n vertices can be computed in $O(n)$ time.*

Furthermore, Chazelle showed how a triangulation of a polygon can be derived in linear time from its trapezoidal map.

Theorem 2.1.2 *The triangulation of a simple polygon with n vertices can be computed in $O(n)$ time.*

The output of Chazelle's algorithm is a tree because every edge represents a diagonal that divides the polygon in two and because the polygon is simple. The tree has a node for each trapezoid and an edge for each diagonal between two neighbouring trapezoids (Fig. 2.1). This tree will be called a *trapezoidal graph*.

The purpose of the trapezoidal graph is to make a *balanced decomposition* $\mathcal{S}$ of the polygon. A balanced decomposition is a tree representing recursive splitting of a polygon, where each split results in two sub-polygons with approximately the same number of sites (Fig. 2.2). Hence, to ensure that the decomposition can be balanced, a trapezoid must not contain more than one site. A trapezoid must then be split if it contains more than one site, and two trapezoids must be merged into a simple polygon if one of them is empty. Eventually every sub-polygon contains exactly one site, and we can make the decomposition balanced.

To find out which trapezoids must be split, we must keep track of all sites inside a trapezoid. We do this by inserting the sites into a binary search tree of each trapezoid. Hence, we need a point location structure to efficiently find the trapezoid containing a given site. This is explained in the next subsection.

A triangulation of Q would be simpler than the trapezoidal map, but we now know that this type of partition has a drawback: If the edges of a triangle Δ in

```
Algorithm Merge(Node x, Node y)
for each node z incident to y and not visited do
  if z is empty and the degree of a merged node does not exceed 4 then
    Merge(x, z)
  else
    create a new edge (x, z)
for each node z incident to x by a new edge do
  Merge(z, z)
return
```

Figure 2.3: The merge algorithm

the triangulation are all diagonals, it would not be possible to create a diagonal inside Δ if it contained two sites, since a diagonal must have end vertices on the boundary of Q.

We use the point location structure to locate each site s in $O(\log n)$ time. If sites are already associated to that trapezoid, we use the binary search tree to locate the correct trapezoid in $O(\log m)$ time. This gives us a total running time of $O(m(\log m + \log n))$ for the splitting process.

Since the decomposition should be balanced according to the number of sites in the sub-polygons, we need to merge empty and non-empty sub-polygons to obtain a decomposition where each sub-polygon contains exactly one site. This can be done by running a merge algorithm (Fig. 2.3). Let x be the node associated with a non-empty trapezoid. Then the procedure call **Merge(x, x)** will merge all trapezoids correctly.

If a node of an empty sub-polygon is merged with a node of a non-empty sub-polygon, and the result is a node with degree greater than 4, we no longer have a bound on the number of children of nodes. To prevent this, we simply do not merge two nodes if the resulting node has degree greater than 4. If a node of an empty sub-polygon is not merged, it is treated as a non-empty sub-polygon. Hence, it will be merged with other nodes of empty sub-polygons. In this way no node of an empty sub-polygon is incident to a node of another empty sub-polygon, and the resulting graph will have at most m nodes of empty sub-polygons and m nodes of sub-polygons containing exactly one site.

Theorem 2.1.3 *The space complexity of the trapezoidal map/graph after the merge is $O(m)$.*

After running the merge algorithm, we end up with a division, where each region is a sub-polygon containing one site (Fig. 2.4). We call these sub-polygons *basic sub-polygons*.

Notice that after running the algorithm, nodes are associated with basic sub-polygons instead of trapezoids. Hence, pointers from nodes to diagonals should be updated accordingly. The algorithm runs in linear time, and the following result follows immediately. Notice that we still call the graph a trapezoidal graph even though regions are now sub-polygons and not trapezoids.

A query structure will be built on top of the trapezoidal graph $\mathcal{T}$ before the merging of nodes to answer visibility queries for each of the sites. So the structure of $\mathcal{T}$ is of great importance.

Lemma 2.1.4 *The trapezoidal graph $\mathcal{T}$ is a ternary tree.*

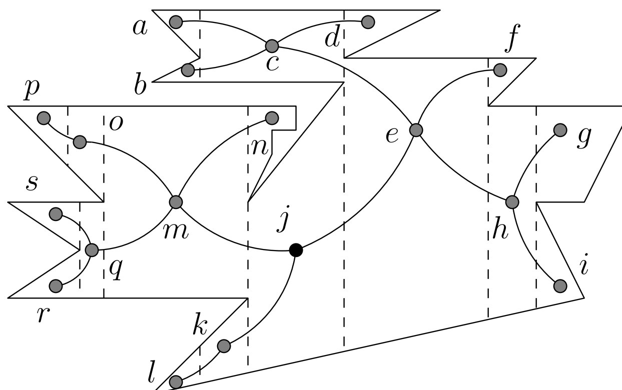

Figure 2.4: A trapezoidal graph after a merging process (See Fig. 1.1)

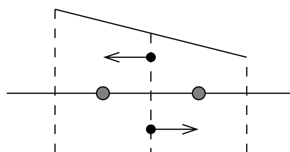

Figure 2.5: Degenerate case: Two sites with the same x-coordinate

Proof: First of all, the trapezoidal graph is a tree since every edge represents a diagonal that divides the polygon in two and since the polygon has no holes. Pick a root at random from the nodes of the graph.

Each node has degree at most 4 before running the merge algorithm, since every trapezoid can have only one upper left neighbour, one lower left neighbour, one upper right neighbour, and one lower right neighbour. The merge algorithm prevent that the degree of a node exceeds 4.

Hence, each node has at most three children. Thus the tree is ternary. □

Degenerate Cases

Will it always be possible to split every trapezoid with two sites? Yes, if two sites have different x-coordinates, a trapezoid will be correctly decomposed, since diagonals by definition are vertical. If two sites inside the same trapezoid have the same x-coordinate, we let the upper site belong to the left trapezoid and the lower site belong to the right trapezoid (Fig. 2.5). If more sites have the same x-coordinate, this procedure continues creating symbolic trapezoids with zero width. The thought of this approach is to slightly rotate the sites counter clockwise—a standard approach for degenerate cases in computational geometry.

If a polygon vertex a and a site s have the same x-coordinate, the site s may lie on the diagonal with a as endpoint. Then s is assumed to belong to the trapezoid to the left of this diagonal.

The degenerate case, where a trapezoid has more than one left definition vertex or more than one right definition vertex, is handled by inserting symbolic

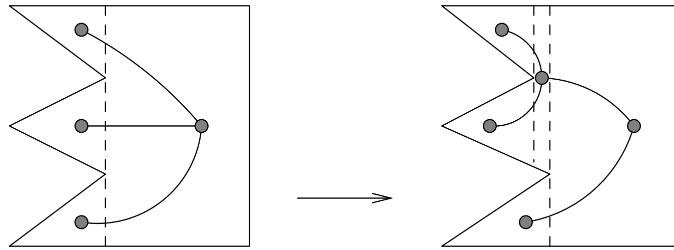

Figure 2.6: Degenerate case: Two polygon vertices with the same x-coordinate

trapezoids of zero width (Fig. 2.6). This can also be illustrated by slightly rotating the polygon vertices counter clockwise.

2.2 Point Location Preprocessing

In this section we consider the point location structure used to efficiently locate the trapezoid containing a given site. Identification of the trapezoid of a site is used to split trapezoids containing more than one site.

Let a *subdivision* be any partition of the plane into polygonal faces (regions). Then the trapezoidal map is a subdivision. Kirkpatrick [K83] showed how a subdivision S with n vertices can be preprocessed in $O(n)$ time in order to answer point location queries in $O(\log n)$ time: Given a query point p, determine which face of S contains p.

First, he shows that the time bound applies to point location in triangular subdivisions, after which he reduces the general (trapezoidal) subdivision problem to the point location problem for triangular subdivisions. The Kirkpatrick data structure is a general data structure and also applies to other problems. Hence, we will describe it out of the context of the visibility problem.

Triangular Subdivisions

Let S be a triangular subdivision with n vertices. Kirkpatrick's data structure is then a directed acyclic graph with faces of a subdivision as internal nodes and leaves. The graph represents a hierarchy of triangular subdivisions $S_1, \ldots, S_{h(n)}$, where $S = S_1$, $h(n)$ is the height of the hierarchy of subdivisions, and $|S_i| > |S_{i+1}|, 1 \leq i \leq h(n) - 1$. The size $|S_i|$ of a subdivision is the number of vertices, and the top level subdivision $S_{h(n)}$ is a triangle. Each face F of S_{i+1} has a pointer (downwards) to each face F' of S_i for which $F' \cap F \neq \emptyset$ (Fig. 2.7). The face F' satisfying this is called a *parent* of F in S_i.

When searching for the face containing the query point p, we locate for each level S_i (starting in $S_{h(n)}$ at the top) which face F' contains p, using the knowledge of which face F in S_{i+1} contained p (Fig. 2.8).

It can be tested in constant time whether a point p is inside a triangle. Let a, b, c be the vertices of a triangle. If the polygonal lines *abp*, *bcp* and *cap* all make the same turn (left or right), then the point is inside the triangle, whether a, b and c are in clockwise order or not.

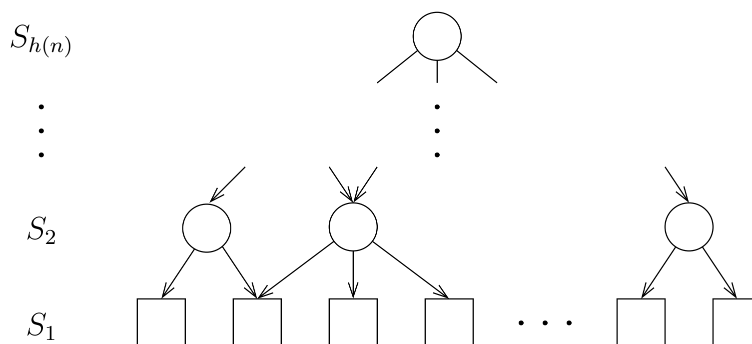

Figure 2.7: The point location structure

Algorithm PointLocation(Point p)
$candidates_{h(n)} \leftarrow$ faces of $S_{h(n)}$
$F \leftarrow$ face in $candidates_{h(n)}$ containing p
$i \leftarrow h(n) - 1$
while $i > 0$ **do**
  $candidates_i \leftarrow parents(F)$
  $F \leftarrow$ face in $candidates_i$ containing p
  $i \leftarrow i - 1$
return F

Figure 2.8: The point location algorithm

It takes $O(|S_{h(n)}|)$ time to find the face containing p in the top level of the hierarchy. In each lower level of the hierarchy, it takes $O(|parents(F)|)$ time to find the face containing p, where F is the face of the level above having most parents. Therefore, it is easy to see that the algorithm in Fig. 2.8 has time complexity

$$O\left(|S_{h(n)}| + \sum_{i=1}^{h(n)-1} \max_{f \in S_{i+1}} \{|parents(f)|\}\right) .$$

We need a few lemmas to determine how large this is.

Lemma 2.2.1 *An n-vertex connected, planar graph has at most $3n - 6$ edges.*

Proof: Euler's formula states that for a connected planar graph with n vertices, m edges, and f faces

$$n - m + f = 2 .$$

When $n \geq 3$ for a connected, planar graph, every face has at least three edges. So if every edge was only incident to one face, then $m \geq 3f$. But since every edge is incident to two faces

$$2m \geq 3f .$$

Together with Euler's formula this gives us:

$$\begin{aligned} & 3n - 3m + 3f = 6 \\ \Rightarrow\quad & 3n - 3m + 2m \geq 6 \\ \Rightarrow\quad & m \leq 3n - 6 \end{aligned}$$

which completes the proof. □

Let an *independent* set I of vertices in the graph $G(V, E)$ be a subset $I \subseteq V$ of vertices such that no pair of vertices in I is incident in $G(V, E)$.

Lemma 2.2.2 *There exists constants $c, d > 0$ such that every connected, planar graph with $n \geq 2$ vertices has at least n/c independent vertices of degree at most d. At least n/c of these can be identified in $O(n)$ time.*

Proof: We will show that $c = 24$ and $d = 11$ will satisfy it, although this may not be optimal.

By Lemma 2.2.1, the average vertex degree

$$d_{avg} = \frac{2m}{n} \leq \frac{2(3n-6)}{n} = 6 - \frac{12}{n} \leq 6 \,.$$

Since no vertex has degree 0, less than half of the vertices have degree greater than 11. Let V be the set of vertices of degree at most 11. Then

$$|V| \geq \frac{n}{2} \,.$$

By choosing elements from V for the independent subset, at most 12 elements are removed from the graph each time an element is chosen—the element itself and at most 11 dependent elements. In this way the independent subset will contain at least $|V|/12 \geq n/24$ elements, and these elements can easily be identified in $O(n)$ time. □

The algorithm for the construction of the point location data structure is described in Fig. 2.9. The data structure is built bottom-up starting with the lowest level S_1. For each level of the hierarchy, an independent set of vertices is removed, and regions of the resulting subdivision are triangulated. Pointers are added from faces of the subdivision before the triangulation to intersecting faces of the subdivision after the triangulation. The time complexity of this algorithm is $O(n)$, which is proved in Lemma 2.2.3.

Lemma 2.2.3 *There exist constants $c, d > 0$ such that, for any triangular subdivision S with n vertices, an associated subdivision hierarchy $S_1, \ldots, S_{h(n)}$ can be constructed in $O(n)$ time satisfying:*

1. $|S_{h(n)}| = 3$
2. $|S_{i+1}| \leq (1 - 1/c)|S_i|$
3. *each face of S_{i+1} has at most d parents in S_i*

```
Algorithm Kirkpatrick(Subdivision S)
i ← 1
S_1 ← enclose S in a large triangle Δ
triangulate the area in S_1 between S and Δ
create a node for each triangle in S_1
while |S_i| > 3 do
   I ← independent set of vertices in S_i with degree less than d
   i ← i + 1
   S_i ← remove I from S_{i−1}
   for each sub-polygon S' of S_i do
      triangulate S'
      create a node for each triangle in S_i
      add pointers from each node in S_i to intersecting nodes of S_{i−1}
return
```

Figure 2.9: The Kirkpatrick algorithm to build the data structure

Proof: Assume that $|S_i| > 3$, and let v be any internal vertex of S_i. Let the *neighbourhood* of v be the union of incident faces of v. If v and its $\deg(v)$ incident edges are removed and the neighbourhood of v is re-triangulated in linear time, the resulting triangulation will decrease by one. In this way, each new face intersects at most $\deg(v)$ faces of S_i.

If $v_1, \ldots, v_t$ form an independent set of vertices of S_i, then it is possible to do this process t times to get the subdivision S_{i+1} with the property that each of the faces intersects at most $\max\{\deg(v_j), 1 \le j \le t\}$ faces of S_i. The intersecting faces can be found in constant time for each face by looking at the constant number of pairs of triangles.

Now, by Lemma 2.2.2, there exist constants $c, d \ge 1$ such that the subdivision S_i has an independent set of size at least $|S_i|/c$ with $deg(v_i) \le d, 1 \le i \le t$. Furthermore, this independent set can be found in $O(|S_i|)$ time.

We get the following construction time for the subdivision hierarchy.

$$\sum_{i=1}^{h(n)} \left(1 - \frac{1}{c}\right)^i n \le \sum_{i=0}^{\infty} \left(1 - \frac{1}{c}\right)^i n = \frac{n}{1 - \left(1 - \frac{1}{c}\right)} = O(n)$$

which completes the proof. □

Theorem 2.2.4 *There is an $O(\log n)$ query time, $O(n)$ space and $O(n)$ preprocessing time algorithm for the triangular subdivision point location problem, where $O(n)$ is the complexity of the subdivision.*

Proof: If we use the data structure by Kirkpatrick to answer point location queries, Lemma 2.2.3 tells us that the preprocessing time is $O(n)$. Since $|S_1|, |S_2|, \ldots, |S_{h(n)}|$ is decreasing geometrically, $h(n) = O(\log n)$. Hence, the space used by the data structure is

$$O\left(\sum_{i=1}^{h(n)} |S_i|\right) = O(n)$$

plus the space used by the links between levels, but this is also $O(n)$, since each face has a constant number of parents.

The constant number of parents also means that the query time is

$$O\left(|S_{h(n)}| + \sum_{i=1}^{h(n)-1} \max_{f \in S_{i+1}} \{|parents(f)|\}\right) = O(\log n)$$

since $h(n) = O(\log n)$. □

Trapezoidal Subdivisions

Now that we have established a result for point location in triangular subdivisions, we need a way to use this result for trapezoidal subdivisions in order to identify the trapezoid containing a given point.

Theorem 2.2.5 *There is an $O(\log n)$ query time, $O(n)$ space and $O(n)$ preprocessing time algorithm for the trapezoidal subdivision point location problem, where $O(n)$ is the complexity of the subdivision.*

Proof: We can easily triangulate all faces of the trapezoidal subdivision S in linear time, thereby getting a new subdivision T. Since T is a refinement of S, the location of a triangle in T implies the location of a face in S. So after a triangulation of each trapezoid, Theorem 2.2.4 implies the desired result. □

So after computing the trapezoidal map in $O(n)$ time, preprocessing the trapezoidal map in $O(n)$ time, locating each site in $O(m \log n)$ time and splitting trapezoids in $O(m(\log m + \log n))$ time, we end up with an $O(m)$ sized trapezoidal graph $\mathcal{T}$ where each node represents a sub-polygon with one site inside. In the next chapter, a balanced decomposition of $\mathcal{T}$ and a factor graph will be built to support the query structure.

Chapter 3

Decomposition and Factor Graph

In order to compute visibility edges inside the polygon efficiently, a balanced decomposition of the trapezoidal graph is built. This data structure is then extended to form the *factor graph* that is the underlying structure of the query structure described in the next chapter.

In the first section of this chapter, the balanced decomposition will be described. This data structure is built by recursively removing edges of the trapezoidal graph $\mathcal{T}$ of the last chapter. Every removal constitutes a node of the balanced decomposition. Since the process is a general tree transform, it will be described out of the context of $\mathcal{T}$. In the second section, we will describe how the tree transform is used on $\mathcal{T}$. In the last section, the factor graph will be described. The factor graph is an extension of the balanced decomposition used by the query structure.

3.1 Decomposition of a Tree

Every removal of an edge in a tree T results in two subtrees. By repeatedly removing edges, we get a decomposition $\mathcal{S}$ of T, where nodes are edges of T and leaves are nodes of T. The decomposition is balanced if there exists a constant $\alpha > 0$ such that each time a subtree T is partitioned by the removal of an edge, the two subtrees remaining each have size at least $\alpha|T'|$.

Guibas et al. [GHLST86] showed how to build a balanced decomposition of a binary tree in $O(n)$ time, where n is the number of nodes in the tree. The same idea is used here to build a balanced decomposition of a ternary tree. Therefore, in our case $\alpha = 1/4$, contrary to [GHLST86] where $\alpha = 1/3$. It should be noted that the proof of the $O(n)$ time construction of the auxiliary tree is omitted in [GHLST86]. Hence, the proof of Theorem 3.1.6 and preceeding lemmas is my own work.

Auxiliary Tree

Construction of the balanced decomposition makes use of an *auxiliary tree*, denoted A, to find the centroid (split) edges of T. The auxiliary tree has the

Algorithm AuxiliaryTree(Tree T**)**
$L \leftarrow$ empty list
$V \leftarrow$ empty list
for each node v of T (in an Euler tour) **do**
  insert v into L
  if v has not been visited **then** insert v into V
  if v is a leaf **then**
    $b_v \leftarrow 1$
    $\gamma_v \leftarrow 0$
  else if all children of v have been visited **then**
    $b_v \leftarrow carryadd(b_x + b_y + b_z)$ for children x, y, z
    $\gamma_v \leftarrow \max\{n \in \mathbb{N} : 2^n \uparrow b_v\}$
$U \leftarrow$ tournament tree over V with γ_v as priority
return BuildTree(U**,** L**)**

Figure 3.1: The auxiliary tree algorithm

Algorithm BuildTree(TournamentTree U**, List** L**)**
$x \leftarrow$ remove max element from U
$\mathcal{U} \times \mathcal{L} \leftarrow$ split U at each instance of x in L
join the first and the last element of $\mathcal{U} \times \mathcal{L}$
for each element (U', L') in $\mathcal{U} \times \mathcal{L}$ **do**
  make **BuildTree(**U'**,** L'**)** a child of x
return x

Figure 3.2: The build tree algorithm

same nodes as T, and each node has at most four children. See Fig. 3.1 and Fig. 3.2 for the construction of A.

A label $b_v = carryadd(b_x + b_y + b_z)$ is computed for every node v of T with children x, y and z. The function $carryadd(b_x + b_y + b_z)$ is defined as follows. Let i be the position of the leftmost carry in the computation of $b_x+b_y+b_z+1$ as binary numbers and let n be the number of bits. Then $carryadd(b_x + b_y + b_z)$ is defined to be the binary number consisting of bits $(n-1)\ldots i$ from $b_x+b_y+b_z+1$ and bits $(i-1)\ldots 0$ being zero. Leaves v of T have labels $b_v = 1$. For example

$$carryadd(10 + 10 + 1) = 100$$

since the addition is $10 + 10 + 1 + 1 = 110$ and $i = 2$.

A label b_v of a node v is an approximation of the size of the subtree of v: $|v|/2 \leq b_v \leq |v|$ where $|v|$ is the size of the subtree of v. The first inequality comes from the fact that $b_x + b_y + b_z + 1 \leq 2carryadd(b_x + b_y + b_z)$. Since at most 1 is added to the label each time a node is added to a tree, the second inequality is proved.

A label is used to find the *priority* γ_v of v which is the number of times that 2 divides b_v. The priority is used to find the *strong* node of a tree in each step of the construction. Let the node v be balancing in a tree T if there exists a subtree T' after the removal of v such that $\alpha|T| \leq |T'| \leq (1-\alpha)|T|$. A node v is then strong if it is balancing, or the node of highest priority in a subtree after removal of v is balancing.

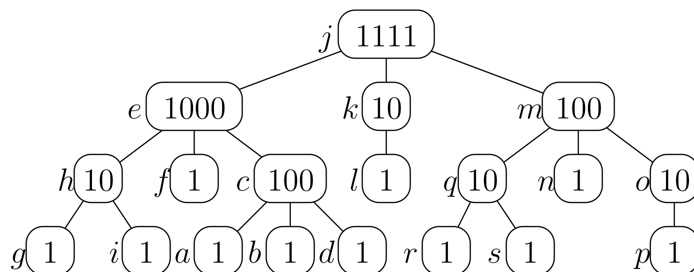

Figure 3.3: Labels on the trapezoidal graph from Fig. 2.4

See Fig. 3.3 for an example of labels on a graph. The graph is the trapezoidal graph from Fig. 2.4, where the black node is picked as the root of T. The priority γ_v is the number of successive zeroes in the right end of the binary representation of b_v.

Lemma 3.1.1 *There is a unique node with highest priority in every subtree of the tree T.*

Proof: Every two different numbers divisible by the same highest power of two have a number in between divisible by a higher power of two. This is an algebraic property of the natural numbers. The lemma is proved by showing that the path between every two nodes with labels divisible by the same power of two contain a node whose label is divisible by a higher power of two.

Suppose now that the node x and its descendant y are divisible by the highest power of two and have labels $b_x = c_1 2^i$ and $b_y = c_2 2^i, c_1 \geq c_2$. Then somewhere between x and y, there must be a node z with label $b_z = 2^{i+1}$, since it can not happen that on the path from y to x label(s) are added without producing the label b_z at some node.

Suppose now that y is not a descendant of x or vice versa, and that their labels are divisible by 2^i, the highest power of two. Assume without loss of generality that $b_x = c_1 2^i$ and $b_y = c_2 2^i$, where $c_1 \leq c_2$. Then the label of a node on the path from x to the common ancestor of x and y is divisible by 2^{i+1} by the same argument as above.

In either case there is a node in between with a label divisible by a higher power of two. Hence, there is a unique node whose label is divisible by the highest power of two. □

Let λ be the modification of a label b_v where the rightmost 1-bit is switched. I.e., if γ is the priority of v then $\lambda = b_v - 2^\gamma$. When a subtree of a tree rooted at v has been removed, the approximation of the size of the subtree of v changes to: $|v|/2 \leq b_v - \lambda \leq |v|$.

The following lemma shows that if the label b_v of the root v of a tree T satisfies that $\lambda + 110\ldots0 \leq b_v \leq \lambda + 111\ldots1$ where the lower and upper bit strings have the same length, then the node of T with highest priority is balancing.

Lemma 3.1.2 *In a tree with v as root, the node with highest priority γ is balancing if*

$$\lambda + 2^\gamma + 2^{\gamma-1} \leq b_v \leq \lambda + 2^{\gamma+1} - 1 \,.$$

Proof: Assume the condition of the lemma. We will prove that the part $T - subtree(m)$ of T above m satisfies that

$$\frac{1}{8} < \frac{|v| - |m|}{|v|} < \frac{7}{8}$$

and thus makes the node m balancing.

First inequality: Since $b_v - b_m$ is an approximation of the size of $T - subtree(m)$, we get that $b_v - b_m \leq |v| - |m|$. Together with the condition and the definition of λ we have that

$$\frac{|v| - |m|}{|v|} \geq \frac{b_v - b_m}{|v|} \geq \frac{\lambda + 2^\gamma + 2^{\gamma-1} - b_m}{|v|} \geq \frac{\lambda + 2^\gamma + 2^{\gamma-1} - (\lambda + 2^\gamma)}{2(b_v - \lambda)} = \frac{2^{\gamma-1}}{2(b_v - \lambda)} = \frac{2^{\gamma-2}}{b_v - \lambda} \geq \frac{2^{\gamma-2}}{2^{\gamma+1} - 1} \geq \frac{1}{8 - 2^{2-\gamma}} > \frac{1}{8}$$

for all $\gamma \geq 0$.

Likewise, we get the second inequality:

$$\frac{|v| - |m|}{|v|} \leq 1 - \frac{|m|}{|v|} \leq 1 - \frac{2^\gamma}{|v|} \leq 1 - \frac{2^\gamma}{2b_v} \leq 1 - \frac{2^{\gamma-1}}{b_v} \leq 1 - \frac{2^{\gamma-1}}{\lambda + 2^{\gamma+1} - 1} < \frac{1}{2} < \frac{7}{8} .$$

Hence, the node m is balancing, since the size of $T - subtree(m)$ is between $1/8$ and $7/8$ of the size of T. □

Lemma 3.1.3 *The node with the highest priority of a tree is a strong node and can be picked as the root of the auxiliary tree (or subtree).*

Proof: If the node m with highest priority γ of the tree T satisfies that

$$\lambda + 2^\gamma + 2^{\gamma-1} \leq b_v \leq \lambda + 2^{\gamma+1} - 1 .$$

then m is balancing by Lemma 3.1.2.

If m does not satisfy the above, then at least one of the remaining subtrees of T has a balancing node after the removal of m:

We will show that a root of one of the remaining subtrees satisfies that

$$\lambda + 2^\gamma + 2^{\gamma-1} \leq b_v \leq \lambda + 2^{\gamma+1} - 1$$

by assuming the negation. Hence some child x of m must satisfy

$$\lambda + 2^{\gamma-1} \leq b_x \leq \lambda + 2^\gamma + 2^{\gamma-1} - 1$$

for some γ. The node m can then only have priority γ if for another child y

$$2^{\gamma-2} \leq b_y .$$

Together this gives us that

$$2^{\gamma-2} \leq b_y \leq |y| < \frac{1}{8}$$

since the subtree of y must have size less then $|T|/8$. This equation fails for $\gamma \geq 0$. Hence, one of the remaining subtrees of T has a balancing node by Lemma 3.1.2. □

Lemma 3.1.4 *After the removal of a node in T, there is still a unique node with highest priority in each of the remaining four subtrees.*

Proof: Let x be the removed node of T. The same proof as the one of Lemma 3.1.1 can be used for the subtree of each of the children of x.

In $T - subtree(x)$ all pairs of nodes still have a node in between with label divisible by a higher power of two, since the path between two nodes in $T - subtree(x)$ is the same as before the removal of x. □

The implementation of the tournament tree in Fig. 3.1 and Fig. 3.2 must support efficient insert, remove, split and join operations. This is achieved using a *2-4 tree* [GT98] with the property that each internal node has 2, 3 or 4 children and all external nodes have the same depth. Each internal node with x children has $x-1$ keys. After inserting (resp. deleting) an element, the two properties are maintained by creating (resp. deleting) internal nodes and performing $O(\log n)$ key transfer operations and fusions of nodes when overflow (resp. underflow) of children occur.

Deletion of elements, key transfers and fusions of nodes are also used to maintain the properties after a split operation: Let x and y be the keys of the first and last element of the new tree. First, all keys outside the range $[x, y]$ are removed together with the respective children in each node from x (resp. y) to the common ancestor z of x and y. This causes underflow in some of the nodes from x (resp. y) to z which is fixed by $O(\log n)$ node fusions and key transfers. In the remaining part of the tree, the same operations are used to maintain the properties. The same technique is used for tree joins where operations instead are node splits and key transfers.

Theorem 3.1.5 *The auxiliary tree of a tree with n nodes has height $O(\log m)$.*

Proof: By Lemma 3.1.1 and Lemma 3.1.3, we can choose a unique, strong node for each subtree of the tree T. This node is balancing in the auxiliary tree A at least half of the times. Hence, A has height $O(\log m)$. □

Theorem 3.1.6 *It takes $O(n)$ time to construct the auxiliary tree of a tree with n nodes.*

Proof: Computing the Euler tour takes $O(n)$ time, since insertion into a list and priority computation is constant time operations. The tournament tree U on the Euler tour can be constructed bottom up in $O(n)$ time.

Removal of the node with highest priority in U and splitting of the tree takes $O(\log m)$ time. Let a be the removed node having subtrees b, c, d and e. After (at most) four split operations in $O(\log n)$ time, (at most) five parts of U remain corresponding to the following chunks from left to right in the Euler tour: first part of a, b, c, d and second part of a. The two parts of a are joined in $O(\log n)$ time.

Recursion proceeds to the four parts, where the sum of the size of these parts is $|L|$ and L is the Euler tour list. Hence, the complexity of the **AuxiliaryTree()** algorithm is

$$T(n) = \sum_{i=1}^{4} T(n_i) + \log n$$

which is $O(n)$, since $\sum_i n_i = n$ and $n_i \leq \epsilon n$ for some $\epsilon > 0$. □

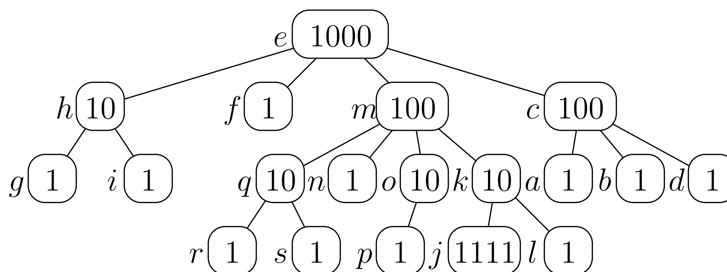

Figure 3.4: The auxiliary tree derived from the tree in Fig. 3.3

Lemma 3.1.7 *The auxiliary tree has at most* $O\left(\frac{m}{2^h}\right)$ *nodes of height* h.

Proof: Follows immediately from Theorem 3.1.5. □

The auxiliary tree of the trapezoidal graph in Fig. 3.3 is seen in Fig. 3.4 as an example.

Centroid Edge and The Decomposition

Lemma 3.1.8 *In any ternary tree with* m *nodes there is an edge (a centroid edge) whose removal leaves two components, each with at least* $\lfloor \frac{m+1}{4} \rfloor$ *nodes.*

Proof: The lemma will be proved by an algorithm as follows. Assume that it is known for each node how large the subtree is. Start at the root of the ternary tree T. In each node x of T, check whether any of the edges incident to x satisfy the condition. If none of the edges is a centroid edge, then go to the child where the size of the subtree is at least $m/4$. This child exists, since the part of T minus the subtree of x and the other two children of x must each be of size less than $m/4$. Furthermore, this child is unique, since otherwise there would be a centroid edge. Proceed in this way until the centroid edge is found.

The algorithm will eventually find a centroid edge, since the size of the subtree is at most decreased by a factor 2 each time a new child is visited.

Notice that the centroid edge need not be unique, but this algorithm will always find the centroid edge of lowest depth in T. □

We could find the centroid edges directly in T, but since T is not balanced, this could take more than $O(\log n)$ time, which we can not afford. Therefore, we use the auxiliary tree to find each centroid edge in $O(\log n)$ time.

A subtree in T has the same nodes as a subtree in A but possibly in another structure. Therefore, an edge in T incident to nodes x and y does not have to be incident to x and y in A.

If the size of the subtree of a node is computed for all nodes in T (linear time), nodes in A can also have information on the size of the component in A. This will be useful for the following lemma.

Lemma 3.1.9 *The centroid edge of an auxiliary tree with* n *nodes can be found in* $O(\log n)$ *time.*

Proof: By Lemma 3.1.8, there is an edge whose removal leaves two components, each with at least $\lfloor \frac{m+1}{4} \rfloor$ nodes. Thus, we can find the edge by searching through A starting at the root.

By looking at the size of the subtree that is stored in each node of A, it can be tested in constant time, whether one of the edges in T incident to the node in question is a centroid edge. If so, it is reported, otherwise the search continues to the child with the largest subtree.

Eventually the search will find the centroid edge, since it exists and it will be in the larger of two children's subtrees. The edge will be found in $O(\log n)$ time, since the height of A is $O(\log n)$.

The removal of a centroid edge e from a node x to its parent y results in two subtrees of A, one tree $A - subtree(y)$ above e and one tree $subtree(y)$ below e. In the subtree above e, the size of a subtree of a node can be updated in $O(\log n)$ time for all nodes from e to the root, since the height of A is $O(\log n)$. □

Lemma 3.1.10 *A balanced decomposition of a ternary tree with n nodes can be computed in $O(n)$ time.*

Proof: Four properties of the auxiliary tree are used to prove linearity of the construction:

1. Nodes in A can not increase in height, since no nodes are added to A.
2. There are at most $O(m/2^k)$ nodes of height k in A (Lemma 3.1.7).
3. By Lemma 3.1.9, it takes $O(h)$ time to split A where h is the height of the root in A.
4. No node can appear as the root more than $O(h)$ times, since A is balanced (priorities decrease for each level of the tree).

This means that decomposition of a subtree takes $O(k^2)$ time as long as a particular node with height k is the root. Since there are at most $m/2^k$ nodes of height k, the decomposition of the tree takes

$$O\left(\sum_{k=0}^{\lfloor \log m \rfloor} k^2 \frac{m}{2^k}\right) = O\left(m \sum_{k=0}^{\lfloor \log m \rfloor} \frac{k^2}{2^k}\right) = O(m)$$

time, since k^2 is a polynomial and 2^k increases exponentially. □

3.2 Decomposition of the Trapezoidal Graph

The decomposition of a ternary tree can be applied to the trapezoidal graph $\mathcal{T}$ of our polygon Q. In this way, the decomposition of $\mathcal{T}$ is a recursive splitting of Q into two with a diagonal.

Each node x in the auxiliary tree A has at most four children. The subtrees of these children form the sub-polygons bounding the trapezoid of x. The parent sub-polygon of x is adjacent to one of these sub-polygons.

Theorem 3.2.1 *A balanced decomposition of the trapezoidal graph can be computed in $O(m)$ time.*

Proof: Since by Lemma 2.1.4, the trapezoidal graph of $O(m)$ nodes is ternary, we have the desired result from Lemma 3.1.10. □

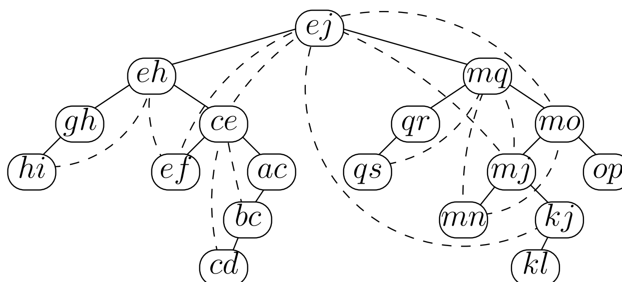

Figure 3.5: The factor graph of Fig. 2.4 where factor edges are dashed lines

3.3 Factor Graph

The last thing to do before applying the query structure, is to build the factor graph from the balanced decomposition.

The factor graph, introduced by Guibas and Hershberger [GH89], has information on the topology of the polygon and is used as a base of the query structure.

The polygon Q is recursively split into two sub-polygons until basic sub-polygons remain. Now, let R_d be the sub-polygon split by the diagonal d in the decomposition process, and let the *depth* of R_d be the depth of d in the decomposition $\mathcal{S}$.

Lemma 3.3.1 *R_d has at most $O(\log m)$ diagonals on its boundary.*

Proof: The diagonals bounding R_d have different depth. This follows by induction:

If d is the diagonal splitting the complete polygon, no diagonals are bounding R_d. Hence, they (none) are of different depth.

Assume now that diagonals bounding R_e have different depths and that e is the parent diagonal of d. Then R_d is bounded by e and some of the same diagonals as R_e. Either way, e has another depth than the other diagonals bounding R_e which in turn are of different depth. Hence, diagonals bounding R_d also have different depths.

Since the tree is balanced, it has height $O(\log m)$ and there is only a $O(\log m)$ number of different depths. Hence, only $O(\log m)$ diagonals are bounding R_d.□

The factor graph of $\mathcal{S}$ is denoted $\mathcal{S}^*$ and has the same nodes and edges as $\mathcal{S}$. Furthermore, $\mathcal{S}^*$ has an edge from each diagonal d to all the diagonals bounding R_d (Fig. 3.5).

Lemma 3.3.2 *The degree of a node in $\mathcal{S}^*$ having $O(m)$ vertices is $O(\log m)$.*

Proof: Each diagonal d has at most $depth(d)$ edges to diagonals of lesser depth, the diagonals bounding R_d. So by Lemma 3.3.1, the out-degree of d is $O(\log m)$.

A diagonal d is adjacent to two sub-polygons at each stage of the decomposition, so for each depth greater than $depth(d)$, d has at most two edges to other diagonals. Thus, the the total degree of a node in $\mathcal{S}^*$ is $O(\log m)$. □

Lemma 3.3.3 *R_d must contain at least*

$$\left\lfloor \left(\frac{4}{3}\right)^{h(d)-1} + 1 \right\rfloor$$

sites, where $h(d)$ is the height of d.

Proof: The lemma is proved inductively: If d is a leaf, then it contains one site which is at least as large as $\lfloor 7/4 \rfloor = 1$.

Assume now that d has two children e and f with heights $h(e)$ and $h(f)$, respectively. Since the decomposition is balanced, we have that

$$\begin{aligned} & \left(\frac{3}{4}\right)^{\max\{h(e),h(f)\}} \geq \frac{1}{2}\left(\frac{3}{4}\right)^{\min\{h(e),h(f)\}} \\ \Rightarrow \quad & 2\left(\frac{3}{4}\right)^{\max\{h(e),h(f)\}-\min\{h(e),h(f)\}} \geq 1 \, . \end{aligned}$$

Assume that the children contain at least $\lfloor (\frac{4}{3})^{height-1} + 1 \rfloor$ sites. Then R_d contains at least

$$\begin{aligned} & \left\lfloor \left(\frac{4}{3}\right)^{h(e)-1} + 1 \right\rfloor + \left\lfloor \left(\frac{4}{3}\right)^{h(f)-1} + 1 \right\rfloor \geq 2 \left\lfloor \left(\frac{4}{3}\right)^{\min\{h(e),h(f)\}-1} + 1 \right\rfloor \\ \geq \quad & 2\left(\frac{3}{4}\right)^{\max\{h(e),h(f)\}-\min\{h(e),h(f)\}} \left\lfloor \left(\frac{4}{3}\right)^{\max\{h(e),h(f)\}-1} + 1 \right\rfloor \\ \geq \quad & \left\lfloor \left(\frac{4}{3}\right)^{\max\{h(e),h(f)\}-1} + 1 \right\rfloor = \left\lfloor \left(\frac{4}{3}\right)^{h(d)-1} + 1 \right\rfloor \end{aligned}$$

sites. □

Theorem 3.3.4 *The size of the factor graph is $O(m)$.*

Proof: Sub-polygons split by diagonals with the same height in $\mathcal{S}$ are all disjoint, since they are separated by the diagonal of the lowest common ancestor. By Lemma 3.3.3, it can be proved like in Lemma 3.1.7 that there are only $O((\frac{3}{4})^k m)$ diagonals with height k in $\mathcal{S}$.

The graph $\mathcal{S}^*$ has $m - 3$ nodes and at most $2h(d)$ edges to descendants from each node d. Therefore by Lemma 3.3.2, $\mathcal{S}^*$ has at most

$$\sum_{d \in \mathcal{S}^*} 2h(d) = O\left(\sum_{k \leq 1+\log_{4/3} m} k \left(\frac{3}{4}\right)^k m \right) = O(m)$$

edges. □

Theorem 3.3.5 *The factor graph can be computed in $O(m)$ time.*

Proof: Two diagonals d and e will be connected in $\mathcal{S}^*$ if and only if they can be connected inside Q by a path that does not cross any diagonal of depth less than $\min\{depth(d), depth(e)\}$.

Hence, the factor edges in $\mathcal{S}^*$ for a given edge d in $\mathcal{T}$ can be computed by expanding a tree from d to paths that comply with the requirement above.

Since factor edges are only visited twice and no further computations are done, the complexity is $O(m)$ by Theorem 3.3.4. $\square$

Now a balanced decomposition $\mathcal{S}$ of the trapezoidal graph $\mathcal{T}$ has been computed in $O(m)$ time and afterwards used to build the factor graph $\mathcal{S}^*$, also in $O(m)$ time.

The last data structure to set up before using the visibility algorithm is the query structure that is applied on top of the factor graph.

Chapter 4

Query Structure

Guibas and Hershberger [GH89] introduced a data structure called an *hourglass* to represent shortest paths between two diagonals of a polygon. In this context, we will use an hourglass to represent visibility between two diagonals.

Let $\overline{AB}$ and $\overline{CD}$ be two diagonals in the polygon Q so that $ACDB$ is in clockwise order around the boundary of Q (Fig. 4.1). Let $\pi(A, C)$ represent the shortest path inside Q between points A and C that are endpoints of two diagonals. Then the hourglass of $\overline{AB}$ and $\overline{CD}$ is the union of the polygonal chains $\pi(A, C)$ and $\pi(D, B)$ and is denoted $H(\overline{AB}, \overline{CD})$.

We would like to answer queries like: For a given site s inside the polygon Q, determine the part I_s of a diagonal at one end of an hourglass that is visible to s (Fig. 4.1). In this way, rays from a site can be associated with diagonals in order to compute visibility edges of a diagonal. It turns out that hourglasses for all edges of the factor graph can be computed in $O(m + n)$ time and that queries can be answered in $O(\log mn)$ time.

The main data structure of the hourglass is the *convex chain*. The convex chain data structure represents a convex chain of edges and has two important properties: Two convex chains can efficiently be combined to form a new convex chain, and the tangent from a query point to a convex chain can be found efficiently. Details will be given later.

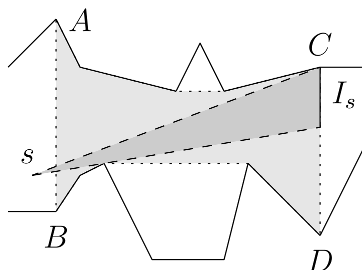

Figure 4.1: An hourglass of $\overline{AB}$ and $\overline{CD}$

4.1 Hourglasses

Let s and t be two sites inside Q and assume that the straight line $\overline{st}$ is contained in Q. Then s and t see each other and $\overline{st}$ will intersect a sequence of diagonals. Each of these separating diagonals will split Q into two parts, one containing s and one containing t. In this ordered sequence, adjacent diagonals have different depths. Furthermore by the definition of a splitting diagonal, there is always a diagonal of lesser depth between two diagonals of equal depth.

Let d_s be the deepest diagonal of S where the sub-polygon split by d_s contains s. Define d_t in the same way. Notice that $\overline{st}$ does not need to cross d_s or d_t. If s and t can see each other, $\overline{st}$ must cross a diagonal that is the lowest common ancestor of d_s and d_t. Let this diagonal be d'. We define d_s and d' to be part of the *principal diagonals* of s, denoted D_s. If the sequence of diagonals intersecting $\overline{st}$ is scanned from d_s to d', each diagonal with depth less than the minimum depth so far is also a principal diagonal. Define D_t in the same way. Since D_s is a subset of the ancestor diagonals of d_s, the idea is to only associate visibility rays from a site s to ancestor diagonals of d_s.

Since the depth of the principal diagonals is strictly decreasing from d_s to d', the length of the sequence is $O(\log n)$, since this is the height of $\mathcal{S}$.

Each diagonal in D_s is contained in the sub-polygon split by its successor in D_s, and hence is a descendant in $\mathcal{S}$ of that successor. This means that D_s is a sub-sequence of the path from d_s to d' in $\mathcal{S}$. This sub-sequence can be retrieved by following edges of $\mathcal{S}^*$. So instead of computing D_s by looking at all separating diagonals between s and t, only principal diagonals of $\mathcal{S}^*$ need to be considered. Hence, visibility between d_s (resp. d_t) and d' is computed by *concatenating* hourglasses of D_s (resp. D_t) found in the nodes of $\mathcal{S}^*$. Two hourglasses $H(\overline{AB}, d)$ and $H(d, \overline{CD})$ sharing a diagonal d are concatenated by combining them into $H(\overline{AB}, \overline{CD})$ such that this new hourglass is the union of the polygonal chains $\pi(A, C)$ and $\pi(D, B)$

The Data Structure

Given a site s in the sub-polygon split by $\overline{AB}$ in the decomposition process and the hourglass $H(\overline{AB}, \overline{CD})$, it is easy to find out which part of $\overline{CD}$ is visible to s (Fig. 4.1). Find the tangent from s to each side of $H(\overline{AB}, \overline{CD})$ and compute the intersections with $\overline{CD}$. These intersections will be the ends of the interval of $\overline{CD}$ to be seen from s.

Guibas and Hershberger use hourglasses to answer shortest path queries and divide them into two groups: open and closed hourglasses. In closed hourglasses, $\pi(A, C)$ and $\pi(D, B)$ share polygon vertices and $H(\overline{AB}, \overline{CD})$ consists of two funnels and a polygonal chain between them. Closed hourglasses can only be used to answer shortest path queries. If an hourglass in the context of visibility is closed, no points of $\overline{AB}$ can see anything of $\overline{CD}$, and the hourglass is useless. We will therefore only consider open hourglasses like in Fig. 4.1 and denote them simply as hourglasses. The polygonal chains of an open hourglass are convex.

If two hourglasses $H(d_1, d_2)$ and $H(d_2, d_3)$ are concatenated, the convex chains of the concatenated hourglass $H(d_1, d_3)$ will consist of a sub-chain of $H(d_1, d_2)$, a common tangent of the chains of $H(d_1, d_2)$ and $H(d_2, d_3)$ and a sub-chain of $H(d_2, d_3)$. Hence, concatenation reduces to finding common tan-

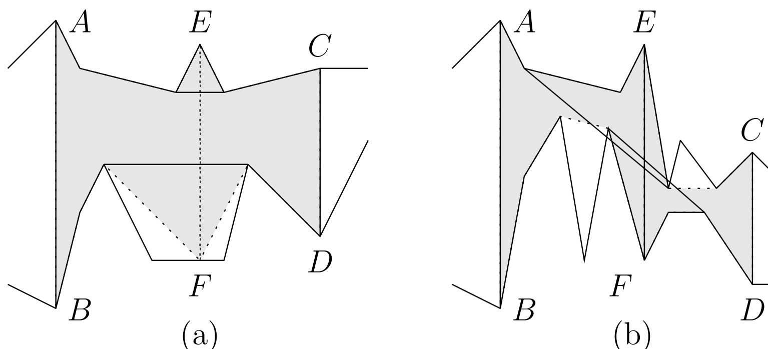

Figure 4.2: Examples of an open (a) and a closed (b) concatenation

gents between pairs of convex chains.

A *tangent* of two chains is the line that intersects one vertex of each chain and nothing else. A tangent of two chains sharing a vertex (one chain in continuation of the other) is called an *outer common tangent*. The common tangents of two hourglasses is defined as the upper outer common tangent and the lower outer common tangent.

Let d_1, d_2 and d_3 be diagonals, so that d_2 separates d_1 and d_3. If $H(d_1, d_2)$ and $H(d_2, d_3)$ are open, then $H(d_1, d_3)$ could be either open or closed. If both outer common tangents of $H(d_1, d_2)$ and $H(d_2, d_3)$ are unblocked with respect to the polygon boundary, the concatenation is open, otherwise closed (Fig. 4.2).

The first two parts of the following lemma will be used to prove that the construction of all hourglasses of the factor graph takes $O(m + n)$ time. The third part guarantees that the hourglass can answer queries in $O(\log mn)$ time. The proof of the lemma will be given in the next section.

Lemma 4.1.1 *If tangents from a point to hourglasses $H(d_1, d_2)$ and $H(d_2, d_3)$ can be found in times τ_1 and τ_2, respectively, and C is some constant, then*

1. *The common tangents of hourglasses $H(d_1, d_2)$ and $H(d_2, d_3)$ is computable in $O(\tau_1 + \tau_2)$ time.*

2. *The hourglass $H(d_1, d_3)$ can be constructed from $H(d_1, d_2)$ and $H(d_2, d_3)$ in time $O(\tau_1 + \tau_2)$. Furthermore, $H(d_1, d_2)$ and $H(d_2, d_3)$ are unaltered by the concatenation operation.*

3. *Tangents to $H(d_1, d_3)$ can be found in time* $\max\{\tau_1, \tau_2\} + C$.

Construction of Hourglasses

Hourglasses are constructed from hourglasses of greater depth (Fig. 4.3). In each outer loop of the algorithm, the depth is decreased and sub-polygons are merged to form new and larger sub-polygons. All hourglasses in these new sub-polygons re-use information of hourglasses of greater depth.

The input to the algorithm is the factor graph of the trapezoidal graph before the merging of nodes, where leaves represent trapezoids. Hourglasses

```
Algorithm Hourglasses(FactorGraph S*)
max ← maximum depth of a node in S*
for each k from max to 0 do
  for each diagonal d ∈ S* of depth k do
    for each d1 bounding Rl and d2 bounding Rr (d1, d2 ≠ d) do
      compute the common tangents of H(d1, d) and H(d, d2)
      construct the hourglass H(d1, d2)
return
```

Figure 4.3: The hourglasses algorithm

of trapezoids are computable in constant time contrary to hourglasses of basic sub-polygons.

The invariant of the algorithm is: After all rounds of the outer loop with k depth, all hourglasses in sub-polygons with greater depth than k have been computed. The invariant is satisfied after the first round, because every sub-polygon with greater depth than the maximum is a trapezoid. Hourglasses between bounding diagonals of a trapezoid are easy to construct, since trapezoids are convex polygons. In further steps of the algorithm, a diagonal d splits a sub-polygon R_d into R_l and R_r. Every hourglass of R_l and R_r has been computed in the last step. So to maintain the invariant, every hourglass $H(d_1, d)$ in R_l must be concatenated with every hourglass $H(d, d_2)$ in R_r.

This process computes hourglasses for exactly those pairs of diagonals linked by edges in the factor graph of the trapezoidal graph before merging. It requires only one concatenation for each hourglass generated.

If two diagonals d_1 and d_2 are linked by an edge in $\mathcal{S}^*$, they lie on the boundary of some sub-polygon. Let $\lambda(d_1, d_2)$ be the logarithm of the size of that sub-polygon.

Lemma 4.1.2 *If d_1 and d_2 are linked by an edge in the factor graph, the tangents through a site to the hourglass $H(d_1, d_2)$ can be found in time $O(\lambda(d_1, d_2))$.*

Proof: We will prove this by induction:

In the basic step, an hourglass of a trapezoid consists of two line segments connecting d_1 and d_2. Hence, tangents through a site to such an hourglass can be found in constant time, say D.

In the induction step, a diagonal d splits R_d into R_l and R_r. Let d_1 and d_2 be bounding diagonals of R_l and R_r, respectively, and let $|R|$ be the number of vertices of the sub-polygon R. We then concatenate $H(d_1, d)$ with $H(d, d_2)$. Since $\mathcal{S}^*$ is balanced, we have by Lemma 3.1.8 that $|R_d| \geq \frac{4}{3}\max\{|R_l|, |R_r|\}$.

By Lemma 4.1.1, tangents of $H(d_1, d_2)$ can be found in time

$$C' \max\{|R_l|, |R_r|\} + C \leq C' \frac{3}{4}|R_d| + C = O(\lambda(d_1, d_2))$$

which completes the proof. □

Theorem 4.1.3 *The hourglasses of all edges of the factor graph $\mathcal{S}^*$ can be constructed in $O(m + n)$ time using $O(m + n)$ space.*

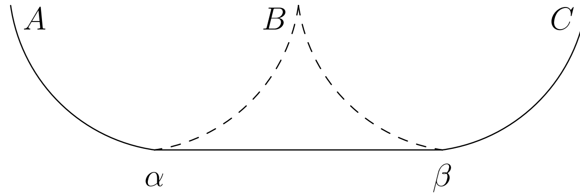

Figure 4.4: A derived chain

Proof: Lemma 4.1.1 part 2 and Lemma 4.1.2 implies that an hourglass obtained by concatenation of $H(d_1, d')$ and $H(d', d_2)$ can be built in $O(\lambda(d_1, d_2))$ time, which is proportional to the height $h(d')$ of d' in $\mathcal{S}^*$. Furthermore

$$h(d') < \max\{d_1, d_2\} .$$

Let $d = \max\{d_1, d_2\}$. Since d is joined with $O(h(d))$ edges to lower diagonals in $\mathcal{S}^*$, the hourglasses of these $O(h(d))$ edges can be built in $O((h(d))^2)$ time.

Thus, as in the proof of Theorem 3.3.4, the construction takes time and space proportional to

$$\sum_{d \in \mathcal{S}^*} (h(d))^2 = O\left(\sum_{k \leq 1+\log_{4/3} n} k^2 \left(\frac{3}{4}\right)^k (m+n) \right) = O(m+n)$$

since there are only $O((\frac{3}{4})^k(m+n))$ diagonals with height k in $\mathcal{S}$ and an hourglass crossing a diagonal of height k can be built in time $O(k)$. $\square$

In the next section, we will describe the chain data structure, prove Lemma 4.1.1 and show how the hourglass data structure can be used to answer queries in logarithmic time.

4.2 Convex Chains

The implementation of an hourglass uses convex chains. Inspired by Overmars and van Leeuwen [OL81], convex chains are represented as binary trees. This is the obvious way to find a tangent of a convex chain in logarithmic time.

A convex chain from A to B has the property that it is equal to its own convex hull minus $\overline{AB}$. A chain can be either *trivial* or *derived*. A trivial chain is just a polygon edge. A derived chain is the convex hull of two sub-chains, $\pi(A, B)$ and $\pi(B, C)$, minus $\{\overline{AB}, \overline{AC}, \overline{BC}\}$, where B is a polygon vertex between A and C (Fig. 4.4). Thus, a derived chain consists of a common tangent of two sub-chains and references to these. Notice that the common tangent can be of zero length, and that the common tangent can be equal to the convex chain.

So, in the binary tree implementation of a convex chain, a single edge is just a leaf, and a derived chain is represented by a node containing the common tangent and pointers to two nodes representing the sub-chains.

The basic query of the convex chains is: Given a site s outside the convex hull of the convex chain A, compute the two tangents from s to A. Given a node of A, it can be checked in constant time, which sub-chain of A a tangent

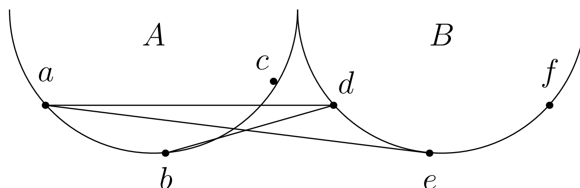

Figure 4.5: Possible intersections of chain points

touches. Since the height of the binary tree of A is logarithmic, the query time is also logarithmic.

Now that the chain data structure has been described, we are now able to prove the three parts of Lemma 4.1.1.

Proof of Lemma 4.1.1 part 1: We will now prove that the common tangent of two convex chains can be found in time $O(\tau_1 + \tau_2)$. The problem of finding the common tangent reduces to the problem of finding one point α on the first convex chain and one point β on the second convex chain, such that the straight line between them does not intersect the convex chains. Hence, we need to search in the convex chains for these points.

Assume that we are looking at the upper convex chains. The proof for the lower convex chains is similar. Assume furthermore that a point on the convex chain A is connected to a point on the convex chain B with a straight line. Then this straight line can essentially intersect the two convex chains in nine ways (Fig. 4.5). The point on A can lie on the side of A not facing B (Fig. 4.5:a), it can lie "on top" of A (Fig. 4.5:b), or it can lie on the side of A facing B (Fig. 4.5:c). Similarly, the point on B can have three possible positions..

Together, this gives us nine possible lines. We talk about the point p lying after q, if p is lying after q or on q in the counter-clockwise order of the chain. Similar for p lying before q. In the first case of the following, it is only possible to deduce that the search for β must continue after d for the segment $\overline{ad}$. This is so, because we can not say anything about the position of α. It could both lie before, on or after a. But certainly, β must lie after d if $\overline{ad}$ can not intersect B.

$\overline{ad}$ Search for β strictly after d.

$\overline{ae}$ Search for α strictly after a, and search for β after e.

$\overline{af}$ Further analysis is needed.

$\overline{bd}$ Search for α before b, and search for β strictly after d.

$\overline{be}$ Here $b = \alpha$ and $e = \beta$, and we are done.

$\overline{bf}$ Search for α before b, and search for β strictly before f.

$\overline{cd}$ Search for α strictly before c, and search for β strictly after d.

$\overline{ce}$ Search for α strictly before c, and search for β after e.

$\overline{cf}$ Search for α strictly before c.

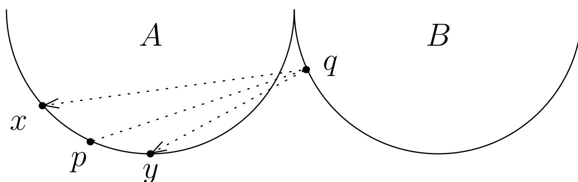

Figure 4.6: An example of $turn(pqm) \neq turn(pqn)$ and $turn(mqn) = right$.

Notice that the following pairs of cases are symmetrical: $\overline{ad}$ and $\overline{cf}$, $\overline{ae}$ and $\overline{bf}$, $\overline{bd}$ and $\overline{ce}$. In the case $\overline{af}$, the search for α proceeds after a, or the search for β proceeds before f, depending on the exact position of a and f on the convex chains.

Assume that p is a vertex of A, q is a vertex of B, and x and y are vertices incident to p where x is to the left of p and y is to the right of p. If the turn of pqx and pqy are the same, then p is at position b. If the turn of pqx and pqy are different and xqy is a right turn, then p is at position a (Fig. 4.6). Finally, if the turn of pqx and pqy are different and xqy is a left turn, then p is at position c. Similar conclusions can be made for q. Hence, the cases can be distinguished in constant time.

Let τ_1 and τ_2 be the (logarithmic) times to compute tangents from a point to A and B, respectively. Since, the search space is reduced to half the size for either α or β in each of the nine cases, this approach reduces the problem of finding a common tangent of A and B to the problem of finding a tangent from a point to A and a tangent from a point to B. Hence, the time to find the common tangent is $O(\tau_1 + \tau_2)$. □

Proof of Lemma 4.1.1 part 2: This part is a consequence of the previous one: If tangents from a point to hourglasses $H(d_1, d_2)$ and $H(d_2, d_3)$ can be found in times τ_1 and τ_2, respectively, then the common tangents of (the convex chains of) the two hourglasses is computable in $O(\tau_1 + \tau_2)$ time. The hourglass $H(d_1, d_3)$ is represented as two convex chains. The upper (resp. lower) convex chain is represented as a pointer to the common tangent of the upper (resp. lower) chains of $H(d_1, d_2)$ and $H(d_2, d_3)$, a pointer to the upper (resp. lower) chain of $H(d_1, d_2)$ and a pointer to the lower (resp. upper) chain of $H(d_2, d_3)$. Hence, $H(d_1, d_3)$ can be constructed in $O(\tau_1 + \tau_2)$ time, and the original two hourglasses are unaltered by the concatenation. □

Proof of Lemma 4.1.1 part 3: Let s be a site of P. When searching for the tangent from s to $H(d_1, d_3)$, we look at the edge stored at the binary tree node that represents a convex chain of $H(d_1, d_3)$. In constant time we determined which sub-chain of $H(d_1, d_2)$ and $H(d_2, d_3)$ the tangent touches and continue the search in that subtree. Hence, it takes $\max\{\tau_1, \tau_2\} + C$ time to compute the tangent from s to $H(d_1, d_3)$ for some constant C. □

Lemma 4.2.1 *The hourglasses of $\mathcal{S}^*$ can answer visibility queries from a site to one of its principal diagonals in $O(\log^2 mn)$ time.*

Proof: Before answering a query, hourglasses of $\mathcal{S}^*$ must be concatenated into the hourglass to be used for the query.

Let s be a site of P. There are $O(\log mn)$ principal diagonals of D_s holding $O(\log mn)$ hourglasses. By Lemma 4.1.2 and Lemma 4.1.1 part 2, these hourglasses can be concatenated in $O(\log^2 mn)$ time into a single hourglass. By Lemma 4.1.1 part 3, this hourglass can answer queries in $O(\log mn)$ time. Hence, the total query time is $O(\log^2 mn)$. □

Logarithmic Time Queries

If some hourglasses of $\mathcal{S}^*$ answer many queries, it might be a good idea to compute these hourglasses once and for all instead of re-computing them each time they are used. Obviously hourglasses of the upper part of $\mathcal{S}^*$ are used the most. We would like to exploit this fact to reduce the query time of Lemma 4.2.1 to $O(\log mn)$.

Lemma 4.2.2 *In a balanced tree S with n nodes, there are only $O(n/\log^2 n)$ nodes with at least $\alpha \log^2 n$ descendants.*

Proof: Since S is balanced, the number of descendants is on average multiplied by a factor α when moving from a node to its parent. Therefore, each node of height larger than $O(\log\log^2 n)$ has at least $\alpha \log^2 n$ descendants. Since S is balanced, the number of nodes of height greater than h is $O(n/2^h)$ (similar proof like in Lemma 3.1.7). This means that there is at least

$$O(n/2^{\log\log^2 n}) = O(n/\log^2 n)$$

nodes with at least $\alpha \log^2 n$ descendants. □

Let U be the set of upper nodes of $\mathcal{S}^*$ with more than $\alpha \log^2 mn$ descendants. Add to $\mathcal{S}^*$ edges from nodes of U to all their ancestors. Since $\mathcal{S}^*$ is balanced, the set U has $O((m+n)/\log^2 mn)$ nodes by Lemma 4.2.2. These nodes have $O(\log mn)$ ancestors each, resulting in $O((m+n)/\log mn)$ new edges of $\mathcal{S}^*$. A construction time of $O(\log mn)$ for each hourglass gives us a $O(m+n)$ time precomputation of the additional hourglasses.

Theorem 4.2.3 *Visibility queries in $\mathcal{S}^*$ can be performed in $O(\log mn)$ time.*

Proof: Let R_d be the minimum sub-polygon containing the query site s and let d' be an ancestor of d in $\mathcal{S}^*$. Then d and d' is separated by a sequence D of diagonals.

If $d' \notin U$ then d' has at most $O(\log^2 mn)$ descendants, and there are only $O(\log(\log^2 mn)) = O(\log\log mn)$ hourglasses to be concatenated between d and d'. Therefore, the construction time of $H(d, d')$ is $O((\log\log mn)^2)$, which in turn is $O(\log mn)$.

Assume now that $d' \in U$. Let d^- be the highest diagonal in D that is not in U, and let d^+ be the successor of d^- in D. Since $d^- \notin U$, we can compute $H(d, d^-)$ in $O((\log\log mn)^2)$ time. The hourglass $H(d^-, d^+)$ can be computed in $O(\log mn)$ time, and $H(d^+, d')$ has been precomputed. Concatenation of $H(d, d^-)$, $H(d^-, d^+)$ and $H(d^+, d')$ takes $O(\log mn)$ time. □

We now have a data structure to determine which part of a diagonal is visible from any given site. All data structures of the algorithm have now been described, and we are ready to apply the main algorithm.

Chapter 5

Simple Polygon

All of the required data structures have now been set up, and the visibility graph can be computed by the main algorithm. The main algorithm consists of two parts. In the first part, rays from each site through the visible area of a diagonal d are associated to d. Visibility edges are computed in the second part using the associations of rays from the first part. Since the visibility edges are computed in the dual plane, the first section is about the duality transform.

5.1 Dual Plane

Let i_s be the part of d visible from s, and let L_s be the set of lines passing through s and a point in i_s. Then L_s forms a wedge.

In the standard point-line duality transformation, a point (a, b) is transformed into a line $y = ax - b$, and a line $y = ax + b$ is transformed into a point $(a, -b)$. Hence, the set L_s of lines is transformed into a set L_s^* of points, i.e. a line segment.

A site s is then visible to a site t if and only if $L_s \cap L_t \neq \emptyset$, in which case $l \in L_s \cap L_t$ is unique (Fig. 5.1). The line l is transformed into the point l^* in the dual. Hence, visibility between s and t can be discovered by computing the intersection point l^* between L_s^* and L_t^*.

Notice that the diagonal d computing the visibility between sites s and t does not have to cross $\overline{st}$. The sites s and t can see each other if $L_s^* \cap L_t^* \neq \emptyset$,

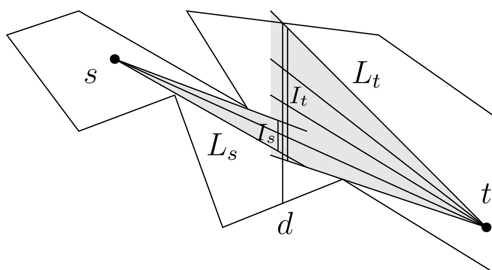

Figure 5.1: Site s sees site t if there is a line in common to L_s and L_t

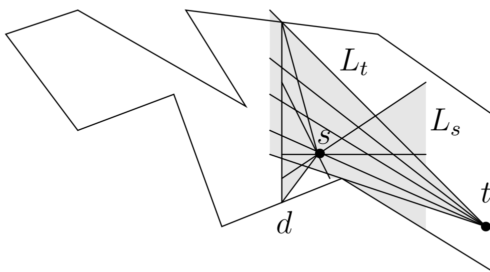

Figure 5.2: The diagonal d does not have to be between sites s and t

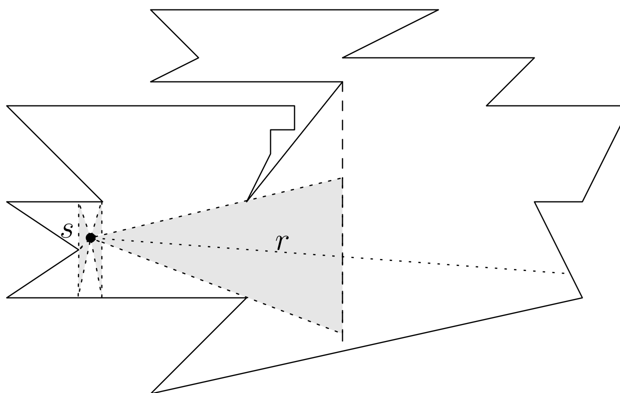

Figure 5.3: Ray intervals of a site

whether the sites are on the same side of d or not (Fig. 5.2), since then there will be a line through s and t.

5.2 Association of Rays

The first part of the main algorithm associate rays from each site to diagonals of the polygon.

Consider a ray r from a site s inside the polygon Q. The part of r inside Q will intersect some set D of the diagonals of Q. Then r is associated to the diagonal $d \in D$ of minimum depth. If this is done for all rays starting in s, we will get a partition of the rays into intervals (Fig. 5.3). For each interval, all rays are associated to the same diagonal, i.e. the diagonal of minimum depth.

It is essential that a ray from s is only associated to one diagonal. Otherwise, a visibility edge could be discovered more than a constant number of times. Hence, we must start by associating rays to the diagonal of lowest depth, hold information about how much of the ray-space has been covered so far, and then walk down the factor graph (Fig. 5.4). The variable J holds the information about how much of the ray-space is covered. Notice that there can be one or two line segments in the variable m of the algorithm, since $J_s \subset I_s$ causes I_s to be split into two parts.

Algorithm RayAssociation(Sites P)
for each site s in P **do**
  $d \leftarrow$ root of the factor graph
  $A \leftarrow$ smallest sub-polygon containing s
  $e \leftarrow$ splitting diagonal of A
  $J_s \leftarrow \emptyset$
  while d is not e **do**
    $I_s \leftarrow$ ray interval for the visibility of d from s
    $I_s \leftarrow I_s - J_s$
    $J_s \leftarrow J_s + I_s$
    if I_s is not empty **then**
      $m \leftarrow$ line segment(s) in the dual corresp. to lines from s through I_s
      insert m into d's list of line segments
    $d \leftarrow$ child of d containing e
return

Figure 5.4: The ray association algorithm

Algorithm Visibility(FactorGraph $\mathcal{S}^*$)
for each diagonal d of $\mathcal{S}^*$ **do**
  $I \leftarrow$ intersection points of all line segments associated with d
  for each point p in I **do**
    $(l, m) \leftarrow$ line segments creating the intersection point p
    $(a, b) \leftarrow$ vertex origins of the ray intervals l^* and m^*
    create visibility edge (a, b)
return

Figure 5.5: The visibility algorithm

Theorem 5.2.1 *Rays from all sites of P can be associated with diagonals of Q in $O(m \log m \log mn)$ time.*

Proof: Each point associates rays to $O(\log m)$ diagonals. Before each association, the visible area of a diagonal is computed in $O(\log mn)$ time by Theorem 4.2.3. Since there are m points, the total time is $O(m \log m \log mn)$. □

5.3 Visibility Edge Computation

Every diagonal of the factor graph now holds a list of line segments in the dual plane. It only remains to compute the intersections of all line segments for each diagonal (Fig. 5.5).

Theorem 5.3.1 *The running time of the intersection computation algorithm is $O(m \log^2 m + k)$, where k is the number of intersections.*

Proof: Let m_d be the number of sites in the sub-polygon split by the diagonal d. Then $\sum_{d \in D} m_d = m$ where D is the set of all diagonals of a given depth. The k_d intersections between m_d line segments can be computed in $O(m_d \log m_d + k_d)$ time using the optimal intersection algorithm of Balaban [B95]. So the total

time for finding the intersections in all of the diagonals in $\mathcal{S}^*$ is

$$\begin{aligned} O\left(\sum_{d\in\mathcal{S}^*}(m_d\log m_d + k_d)\right) &= O\left(\sum_{d\in\mathcal{S}^*}(m_d\log m_d) + \sum_{d\in\mathcal{S}^*}k_d\right) \\ &= O\left(m\log^2 m + k\right) \end{aligned}$$

since it takes $O(m\log m)$ time for the intersection algorithm in each of the $O(\log m)$ depths of $\mathcal{S}^*$. If two sites are visible to each other, then they have an overlapping ray interval over a diagonal of minimal depth. Hence, each visibility edge is reported only once, and $\sum_{d\in\mathcal{S}^*}k_d = k$ which is the total number of visibility edges. □

Theorem 5.3.2 *The visibility graph $VG_Q(P)$ of m sites P within a simple polygon Q having n vertices can be computed in $O(n+m\log m\log mn+k)$ time, where k is the number of edges of $VG_Q(P)$. The space complexity is $O(m+n)$.*

Proof: By Theorem 2.1.1, 2.2.5, 3.2.1, 3.3.5, 4.1.3, 5.2.1 and 5.3.1, we get the following time complexities for each step of the algorithm.

Trapezoidal Map	$O(n)$
Point Location Structure	$O(n)$
Trapezoid Splitting and Merging	$O(m(\log m + \log n))$
Decomposition Tree	$O(m)$
Factor Graph	$O(m)$
Hourglass Query Structures	$O(m+n)$
Ray Associations	$O(m\log m\log mn)$
Visibility Edge Computation	$O(m\log^2 m + k)$

All together, this results in an overall $O(n+m\log m\log mn+k)$ time visibility graph algorithm. By Theorem 2.2.5, 3.3.5 and 4.1.3, the space used for the point location structure, the factor graph and the query structure is $O(m+n)$. □

One of the special cases of the visibility graph computation is the case where $P = V$, i.e. the nodes of the visibility graph are the nodes of the polygon. The optimal algorithm for this problem runs in $O(n+k)$ time [H89]. If we replace m by n in Theorem 5.3.2, we get a running time of $O(n\log^2 n + k)$, which is close to the optimal solution.

Chapter 6

Polygon with Holes

Often, polygons are not simple as they were in the computation of the visibility graph in the previous chapter. Motion planning, for example, is the problem of finding a route among a set of obstacles. Here, the obstacles are the holes of the polygon. Since the polygon becomes more complex, we can not use the algorithm of the previous chapter. Instead, we present an algorithm used to compute the visibility graph of a polygon with holes.

6.1 Other Problems with Holes

Before presenting the algorithm of polygon with holes, let us look at how other known computational geometry problems become harder when holes are added to the polygon.

Polygon Triangulation

As we saw in Section 2.1, a simple polygon can be triangulated in linear time by Chazelle's algorithm. The linearity is achieved by computing an approximation of the triangulation in a bottom-up phase and refining the approximation in a top-down phase using information of the bottom-up phase.

Bar-Yehuda and Chazelle showed in [BC94] how a polygon with h holes can be triangulated in $O(n + h \log^{1+\epsilon} h)$ time. The idea of this algorithm is to connect every hole to another hole or the polygon boundary in $O(h \log^{1+\epsilon} h)$ time. This results in a simple polygon that can be triangulated in $O(n)$ time.

Here, the number of holes h does not affect the number of vertices n at all. Let us look at how many holes the polygon can have without affecting the running time of the linear algorithm. For

$$O(h) = O\left(\frac{n}{\log^{1+\epsilon} n}\right)$$

we have that

$$O(h \log^{1+\epsilon} h) = O(n) \; .$$

So if ϵ is small, h can be very close to n before this algorithm is asymptotically slower than running the algorithm for a simple polygon of the same complexity. Hence, the addition of holes almost never influences the running time of triangulation.

Shortest Path

The problem of finding the shortest path from s to t inside a simple polygon with n vertices can be solved in $O(n)$ time [GHLST86]. This algorithm has a $O(n)$ time preprocessing of the polygon Q given a fixed vertex of Q. The preprocessing exploits the $O(n)$ time triangulation of Chazelle [C91]. The query time for a target point inside Q is then logarithmic.

Kapoor et al. [KMM97] showed how the shortest path inside a polygon Q with h holes can be computed in $O(n + h^2 \log n)$ time. Thus, the addition of holes has no effect on the running time for

$$h \leq \sqrt{\frac{n}{\log n}}$$

when using this algorithm instead of the one in [GHLST86].

This algorithm also starts by computing the triangulation of Q. As we saw, this can be done in $O(n + h \log^{1+\epsilon} h)$ time for polygons with holes. The $O(h^2 \log n)$ part of the time complexity is a result of a sweep algorithm to compute a visibility graph of a set of polygons with $O(h)$ convex chains and $O(h)$ reflex vertices. Here, tangent segments are computed in $O(\log n)$ time. The final search for the shortest path is done in a graph with $O(h^2)$ nodes and edges. Therefore, h does not influence the linear factor n.

6.2 Visibility Graph

The idea of the algorithm in [BHKM04] to compute the visibility graph of points within a polygon with holes is to preprocess Q, so that it is divided into $O(h)$ simple polygons. The visibility graph of the points inside these $O(h)$ polygons can then be computed using the algorithm of the previous chapter. It remains to compute the visibility edges across boundaries of the simple polygons.

Preprocessing

The preprocessing of the visibility algorithm is the same as in the shortest path algorithm. We start by triangulating the polygon Q, which by [BC94] can be done in $O(n + h \log^{1+\epsilon} h)$ time. Let T be the *dual graph* of the triangulation where nodes are faces of the triangulation and edges represent adjacent faces. Then the nodes of T have degree 1, 2 or 3. The graph is now modified in the following way.

Delete every degree 1 node of T and its incident edge. Assume that $h \geq 2$. Then there must be some node of degree 3. Since there must always be an even number of odd-degree vertices, there must be at least two nodes of degree 3. Now, replace every node of degree 2 and its incident edges with a single edge. In the resulting graph G, all nodes have degree 3, and multi edges may occur. It is obvious that the graph has h faces—one for each hole of the polygon. Euler's formula then tells us that there are $O(h)$ nodes and edges. Each node of G corresponds to a triangle in T and is called a *junction triangle* (Fig. 6.1). Removal of the junction triangles from Q results in $O(h)$ simple polygons called *corridors*. The border line between a junction triangle and a corridor is called a *door*. The process of deleting all degree 1 nodes and merging all degree 2 nodes can be done in $O(n)$ time by scanning through the nodes.

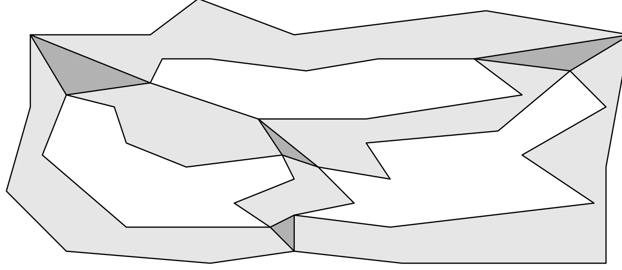

Figure 6.1: Holes (white), Junction triangles (dark) and corridors (light)

Algorithm RayAssociation(Sites P)
for each site s in P **do**
  for each door d of Q **do**
    $i \leftarrow$ interval of d visible from s
    if i is not empty **then**
      $m \leftarrow$ line in the dual plane corresponding to lines from s through i
      insert m into d's list of lines
return

Figure 6.2: The ray association algorithm for polygons with holes

Computing Visibility Edges

After the triangulation step and the above process, the polygon is divided into $O(h)$ corridors and junction triangles. Since the computation of the visibility graph of each of the corridors are completely isolated, the visibility graph of the $O(h)$ corridors of Q can be computed in

$$O\left(\sum_{i=1}^{O(h)} (n_i + m_i \log m_i \log m_i n_i + k_i)\right) = O(n + m \log m \log mn + k)$$

time, since $\sum_{i=1}^{O(h)} m_i = m$. Thus, the visibility graph of the corridors can be computed in $O(n + m \log m \log mn + h \log^{1+\epsilon} h + k)$ time, where h is the number of holes in Q and ϵ is small.

Visibility edges crossing doors still need to be computed. As for the simple polygon case, rays are associated with doors of Q, and visibility edges are computed for each door.

As in the simple polygon case, we iterate through all sites s in P (Fig. 6.2). Since, potentially, all doors of Q can be seen from s, we have to iterate through all $O(h)$ doors. Each visibility query in the $O(m+n)$ sized factor graph is done in $O(\log mn)$ time. Hence, the running time of the ray association algorithm is $O(hm \log mn)$.

Similarly, the algorithm to compute the visibility edges looks a lot like the one for simple polygons. All $O(h)$ doors of Q are iterated (Fig. 6.3), and the intersection points of the lines associated with a door d are computed in $O(m \log m + k_d)$ time. Summing up results in an $O(hm \log m + k)$ total running time.

Algorithm Visibility(Polygon Q)
for each door d of Q **do**
  $I \leftarrow$ intersection points of all lines associated with d
  for each point p in I **do**
    $(l, m) \leftarrow$ lines creating the intersection point p
    $(a, b) \leftarrow$ vertex origins of the ray intervals l^* and m^*
    create visibility edge (a, b)
return

Figure 6.3: The visibility algorithm for polygons with holes

Theorem 6.2.1 *The visibility graph $VG_Q(P)$ of m sites P within a non-simple polygon Q having n vertices and h holes can be computed in*

$$O(n + m(h \log mn + \log m \log mn) + k)$$

time, where k is the number of edges of $VG_Q(P)$.

Proof: The preprocessing of Q takes $O(n + h \log^{1+\epsilon} h)$ time. Computing the visibility edges of all corridors takes $O(n + m \log m \log mn + k_1)$ time, where k_1 is the number of visibility edges that do not cross doors. The association of rays to the doors takes $O(hm \log mn)$ time. Finally, the visibility edges across doors can be found in $O(hm \log m + k_2)$ time, where k_2 is the number of visibility edges crossing doors. The number of output edges is then $k = k_1 + k_2$. □

Notice that this algorithm uses the same amount of space as the algorithm for the simple polygon. The computation of the visibility graphs of all corridors take up $O(m + n)$ space. Since the ray association and visibility edge computation of the doors between corridors does not take up more space, the total amount of space is $O(m + n)$.

Note that the visibility graph problem almost becomes harder by a factor of $O(h)$ when the polygon contains holes, contrary to the addition of $O(h \log^{1+\epsilon} h)$ and $O(h^2 \log n)$ for the triangulation and shortest path problems, respectively.

Chapter 7

Range Restriction

Another special case of the visibility problem is to compute the range-restricted visibility graph $\overline{VG}_Q(P)$ of P within the simple polygon Q. In this scenario, each site s has a restricted range of sight $d_s > 0$, such that s only can see the site t if $\overline{st} \subseteq Q$ and $|st| \leq d_s$ (Fig. 7.1). Since $VG_Q(P)$ can be computed as a range-restricted visibility graph where $d_s = \infty$ for all sites s, $\overline{VG}_Q(P) \subseteq VG_Q(P)$. Since it might be that $d_s \neq d_t$ for two sites s and t, $\overline{VG}_Q(P)$ is a directed graph.

The visibility algorithm uses range queries to compute visibility edges. Let s be a site in Q with a range of sight d_s, and let σ_s be the part of a diagonal visible from s. Then s can only see sites inside the *sector* through σ_s of range d_s (Fig. 7.1). A sector is the intersection between two half-planes and a disc.

The preprocessing of the polygon for this algorithm is the same as for the normal visibility algorithm described in Chapter 2-4, i.e. computation of the trapezoidal map, the point location structure, the balanced decomposition, the factor graph and the query structure.

Like the normal visibility algorithm, this algorithm is based on the divide-and-conquer paradigm. Visibility edges are computed for each sub-polygon of the balanced decomposition $\mathcal{S}$. Before the visibility edges are computed, we must compute for each site s of a sub-polygon, which interval of a diagonal is visible from s. Like before, the interval is computed by two queries to determine the end points of the interval. Thus, it takes $O(m \log m \log n)$ time to compute intervals for the m points in each of the $O(\log m)$ levels of the balanced decomposition. Every interval is inserted into a *segment tree* [BKOS98] of the diagonal used to efficiently report the intervals containing a given query point. Segment

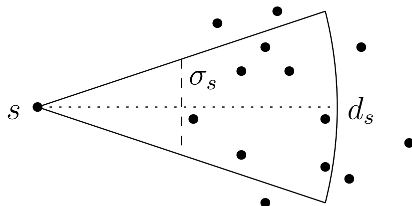

Figure 7.1: The sector range of a site s through σ_s

trees are described in the first section of this chapter.

In the second section, I make essential use of a result on multi-level data structures by Agarwal and Matoušek without proof. In the last section, the computation of the visibility graph is described.

7.1 Segment Trees

Let $I = \{[x_1, x_1'], \ldots, [x_m, x_m']\}$ be a set of m intervals, and let $p_1, \ldots, p_k$ be the sorted list of distinct interval endpoints. Then these interval endpoints induce a partitioning of the real line. The intervals in this partitioning are called *elementary intervals*:

$$(-\infty, p_1), [p_1, p_1], (p_1, p_2), \ldots, (p_{k-1}, p_k), [p_k, p_k], (p_k, \infty) .$$

Points are treated as intervals, because the answer to a query of an endpoint is different from the answer to a query of a point between two endpoints.

A segment tree is a balanced binary tree T. The leaves of T are the sorted elementary intervals, and the elementary interval corresponding to leaf μ is denoted $Int(\mu)$. The internal nodes of T stores intervals that are the union of elementary intervals. The interval $Int(v)$ of an internal node v is the union of the elementary intervals of leaves in the subtree rooted at v. Finally, a node v contains the *canonical subset* $I(v)$ consisting of intervals $[x, x'] \in I$ such that $Int(v) \subseteq [x, x']$ and $Int(parent(v)) \subsetneq [x, x']$.

Intervals of I are stored in the nodes of T instead of the leaves, thereby saving space. Otherwise the amount of space could become quadratic. This is the basic principle of segment trees.

Theorem 7.1.1 *A segment tree on a set of m intervals uses $O(m \log m)$ space.*

Proof: Let v_1, v_2, v_3 be nodes of the same depth of the segment tree T, such that they are numbered from left to right. Assume that $[x, x']$ is stored at v_1 and v_3. Then $[x, x']$ spans the interval from the left endpoint of $Int(v_1)$ to the right endpoint of $Int(v_3)$. Because v_2 lies between v_1 and v_3, $Int(parent(v_2)) \subset [x, x']$ and $[x, x']$ will not be stored at v_2.

Hence, an interval is at most stored in two nodes of each level of T. This means that $O(m \log m)$ space is used, since the height of T is $O(\log m)$. □

Theorem 7.1.2 *Using a segment tree, the intervals containing a query point can be reported in $O(\log m + k)$ time, where m is the number of intervals in the segment tree and k is the number of reported intervals.*

Proof: The QuerySegmentTree algorithm (Fig. 7.2) is called with $v = root(T)$ to report all intervals containing the query point q_x. The functions $lc(v)$ and $rc(v)$ returns the left and right child of v, respectively. The invariant of the algorithm is that $q_x \in Int(v)$ and that intervals of ancestors of v have been reported.

In the first call of the algorithm, the invariant is satisfied since q_x is in $Int(root(T))$ and $root(T)$ has no ancestors. In subsequent calls $q_x \in Int(v)$, since this is the requirement of the branch in the algorithm. Finally, all intervals of the parent of v have been reported, since this is done in the first line of the algorithm before branching.

Algorithm QuerySegmentTree(Node v, QueryPoint q_x)
report all the intervals in $I(v)$
if v is not a leaf **then**
  if $q_x \in Int(lc(v))$ **then**
    QuerySegmentTree($lc(v), q_x$**)**
  else
    QuerySegmentTree($rc(v), q_x$**)**
return

Figure 7.2: The query algorithm of the segment tree

Algorithm InsertSegmentTree(Node v, Interval $[x, x']$)
if $Int(v) \subseteq [x, x']$ **then**
  store $[x, x']$ at v
else
  if $Int(lc(v)) \cap [x, x'] \neq \emptyset$ **then**
    InsertSegmentTree($lc(v), [x, x']$**)**
  if $Int(rc(v)) \cap [x, x'] \neq \emptyset$ **then**
    InsertSegmentTree($rc(v), [x, x']$**)**
return

Figure 7.3: The insertion algorithm of the segment tree

Thus, when a leaf is encountered, all nodes (ancestors) with intervals containing q_x have been visited, and the intervals have been reported only once, since an interval is only associated with one node along a path from the root to a leaf.

The running time is $O(\log m + k)$, since the height of T is $O(\log m)$, and k intervals are reported. □

Theorem 7.1.3 *A segment tree of m intervals can be built in $O(m \log m)$ time.*

Proof: As mentioned before, the endpoints of the intervals must be sorted. This takes $O(m \log m)$ time. The balanced binary search tree is then constructed bottom-up on the sorted elementary intervals, which takes $O(m)$ time. In order to compute the canonical subsets, each interval is inserted one by one by calling the InsertSegmentTree algorithm (Fig. 7.3) with $v = root$.

As we saw in the proof of Theorem 7.1.1, an interval is stored at most twice in each level of T. In addition, only one node of each level has an interval containing x and one node has an interval containing x'. Hence, at most 4 nodes are visited at each level, and the total insertion time is $O(m \log m)$ for the m intervals. □

7.2 Multi-Level Data Structures

Given a set of points in the plane, we want to report the points lying inside a sector. This can be done by two half-plane queries and a disc query in a multi-level data structure described by Agarwal and Matoušek [AM94].

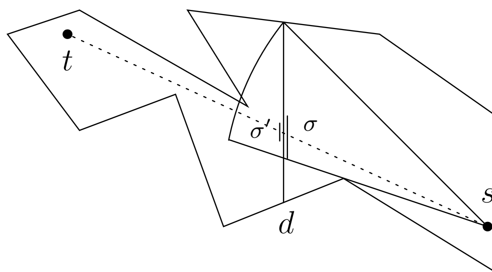

Figure 7.4: The canonical segment σ in which $o = \overline{st} \cap d$ lies

Theorem 7.2.1 *Let P be a set of m points in the plane. For any $\epsilon > 0$, there is a multi-level data structure for P, such that the points from P lying inside a query sector can be reported in $O(m^{1/2+\epsilon} + k)$ time, where k is the number of reported points.*

Furthermore, the time to preprocess the point set P to answer sector range queries is $O(m \log m)$.

7.3 Computing the Visibility Edges

Like mentioned in the beginning of this chapter, visibility intervals of each site of R_d is inserted into the segment tree of the diagonal d. By Theorem 7.1.3, this takes $O(m \log m)$ time for the m points. This is done for each of the $O(\log m)$ levels of the balanced decomposition $\mathcal{S}$, resulting in an $O(m \log^2 m)$ running time.

The visibility edges will now be computed for each level of $\mathcal{S}$. Suppose that we are looking at the diagonal d. We can assume that d is vertical, since we can rotate Q to achieve this. Let P_l (resp. P_r) be the sites of P to the left (resp. right) of d. Insert all of the m segments of visibility into the segment tree T of d. At each node v of T we divide $I(v)$ into S_v^l and S_v^r so that S_v^l (resp. S_v^r) includes all of the segments stored in v that are associated with sites in P_l (resp. P_r).

Assume that s and t are sites on each side of the diagonal d. Let σ_s and σ_t be visibility intervals of s and t, respectively, and let $\sigma \subseteq \sigma_s$ and $\sigma' \subseteq \sigma_t$ be the canonical segments in which $o = \overline{st} \cap d$ lies (Fig. 7.4).

Lemma 7.3.1 *Let $s \in P_l$ and $t \in P_r$. Then s and t see each other through o if and only if t lies in the sector from s through σ and s lies in the sector from t through σ'.*

Proof: First, assume that s and t see each other through o. The segment σ_s (resp. σ_t) is stored in $O(\log m)$ nodes of T, and the union of the canonical segment corresponding to these nodes is σ_s (resp. σ_t). Exactly one of these canonical segments contains o. Let this be σ (resp. σ'). Then t (resp. s) lies in the sector from s (resp. t) through σ.

Suppose now that the segment σ (resp. σ') is a canonical segment of the segment σ_s (resp. σ_t), and that t (resp. s) lies in the sector from s (resp. t). Then s and t see each other through a point $o \in \sigma$. □

Algorithm Visibility(FactorGraph $\mathcal{S}^*$)
for each diagonal d of $\mathcal{S}^*$ **do**
  $T_d \leftarrow$ segment tree for the visible parts of the sites in R_d
  for each node v of T_d **do**
    preprocess the subset of P_r corresponding to S_v^r
    for each segment $\sigma_i \in S_v^l$ **do**
      perform a range query with the sector from s_i through σ_i
      report sites from the output of the query to be visible from s_i
    preprocess the subset of P_l corresponding to S_v^l
    for each segment $\sigma_i \in S_v^r$ **do**
      query with the sector sector from s_i through σ_i
      report sites from the output of the query to be visible from s_i
return

Figure 7.5: The visibility algorithm of sites with restricted sight

Thus by Lemma 7.3.1, to find all visibility edges, a sector range query must be performed for each diagonal d of Q and each canonical segment of the segment tree of d (Fig. 7.5).

Theorem 7.3.2 *Let Q be a simple polygon with n vertices, and let P be a set of m points in Q with associated ranges of sight. Then the range-restricted visibility graph $\overline{VG}_Q(P)$ is constructed in*

$$O(n + m \log m(m^{1/2} \log m + \log mn) + k)$$

time using $O(n + m \log m)$ space.

Proof: By Theorem 2.1.1, 2.2.5 and 3.2.1, the preprocessing steps take time $O(n + m \log mn)$.

Each level of the factor graph $\mathcal{S}^*$ has $O(m)$ points. Thus, computing the visible parts of the diagonals take $O(m \log m \log mn)$ time.

By Theorem 7.1.1, each level of $\mathcal{S}$ has $O(m \log m)$ nodes of a segment tree. Each of these nodes v has $O(\log m)$ segments in its canonical subset $I(v)$. Since, by Theorem 7.2.1, a sector range query takes $O(m^{1/2})$ time for each segment, the total running time is $O(n + m \log m(m^{1/2} \log m + \log mn) + k)$.

By Theorem 7.1.1, the segment trees take up $O(m \log m)$ space. Since the point location structure uses $O(m + n)$ space, the total amount of space is then $O(n + m \log m)$. □

It turns out that a segment tree with m intervals can be implemented using only $O(m)$ space. Thus, the space complexity of $\overline{VG}_Q(P)$ can be reduced to $O(m + n)$.

Chapter 8

Invisibility Graph

Until now, we have only considered the visibility graph. Since the algorithm to compute this is output sensitive, the running time of the algorithm for dense graphs will be dominated by k, the number of visibility edges.

Hence, it will sometimes be faster to compute the complementary graph, the *invisibility graph*. The algorithm to compute this must of course also be output sensitive.

The algorithm of [BHKM04] uses sector range searching and is presented briefly in the first section of this chapter. In the second section, an improved algorithm using half-plane range queries is presented.

8.1 Sector Range Algorithm

In the previous algorithm of range-restricted visibility, a sector range query was answered for each site s in each level of the balanced decomposition. The sector range was a combination of two half-plane queries and a circle query. If the sector range query is replaced by the complement of the two half-plane queries and no circle query, and the search space is a subset of sites on the other side of the diagonal, all reported sites are not visible from s.

Thus, as observed in [BHKM04], the idea of the algorithm to compute the range-restricted visibility graph can be used to compute the invisibility graph. The time complexity of the algorithm will then be the same as for the range-restricted algorithm.

Theorem 8.1.1 *Let Q be a simple polygon with n vertices, and let P be a set of m points in the polygon Q. Then the invisibility graph can be constructed in time $O(n + m \log m(m^{1/2} \log m + \log mn) + \overline{k})$ using $O(n + m \log m)$ space, where $\overline{k}$ is the number of invisible pairs of sites.*

Proof: Even though sector ranges have changed, we can still use Theorem 7.2.1 to answer sector ranges in $O(m^{1/2})$ time for each segment of the canonical subset in the segment tree of a diagonal. Thus, the total running time of the algorithm is $O(n + m \log m(m^{1/2} \log m + \log mn) + \overline{k})$.

Since the only change of the algorithm as compared with the range-restricted algorithm is the sector ranges, the space used is still $O(n + m \log m)$. □

Algorithm Invisibility(FactorGraph $\mathcal{S}^*$)
for each diagonal d of $\mathcal{S}^*$ **do**
  preprocess P_r (resp. P_l) for efficient half-plane range queries
  for each site s of P_l (resp. P_r) **do**
    $(u, l) \leftarrow$ boundary lines of sight from s through d
    $U \leftarrow$ half-plane range query with u in P_r (resp. P_l)
    create invisibility edges between s and all sites of U
    $L \leftarrow$ half-plane range query with l in P_r (resp. P_l)
    create invisibility edges between s and all sites of L
return

Figure 8.1: The invisibility algorithm

8.2 Improved algorithm

It turns out that when a sector range query is replaced by two separate half-plane queries, the problem is easier to solve. Using half-plane range reporting twice to determine which points lie above or below the wedge, the invisibility graph can be computed in $O(n + m \log m \log mn + \overline{k})$ time.

Let d be a diagonal of the factor graph $\mathcal{S}^*$, and let R_d be the sub-polygon that is split by d in the decomposition process. Preprocess the sites $P_r \subseteq R_d$ to the right of d for efficient half-plane range reporting. This takes $O(m_d \log m_d)$ time, where m_d is the number of sites in R_d [C85].

Now for each site $s \in P_l \subseteq R_d$ to the left of d, use the query structure of Guibas and Hershberger [GH89] to determine in $O(\log mn)$ time the boundary lines of sight through d visible from s. By [CGL85], half-plane range queries for the two boundary lines can be answered in $O(\log m_d + k)$ time to report k invisible pairs of sites between s and sites in P_r.

Now preprocess P_l and repeat the above steps for each site in P_r. Every invisible pair of sites (s, t) with d as the lowest common ancestor of s and t in $\mathcal{S}^*$ has now been computed. Repeat all of the above for each diagonal d of $\mathcal{S}^*$ (Fig. 8.1).

See Fig. 8.2 for an example of two half-plane range queries through a site s. The upper half-plane query detects that s is invisible to a, while the lower half-plane query detects that s is invisible to b. Notice that even though c is invisible to s, it is not detected in this step of the algorithm. However, it will be detected when s is one of the sites being preprocessed and half-plane queries are answered from c.

Theorem 8.2.1 *Let Q be a simple polygon with n vertices, and let P be a set of m points in Q with associated ranges of sights. Then the invisibility graph is constructed in $O(n + m \log m \log mn + \overline{k})$ time using $O(n + m)$ space.*

Proof: If $\overline{st} \cap d = \emptyset$ then the invisibility between s and t will be discovered from both s and t, since both sites lie in the respective half-planes. If $\overline{st} \cap d \neq \emptyset$ then let $o = \overline{st} \cap d$ and let d be the lowest common ancestor of s and t. Then the invisibility of s and t will be reported from the site that can not see o. Assume that s can not see o. Then t will lie in one of the half-plane queries answered at s.

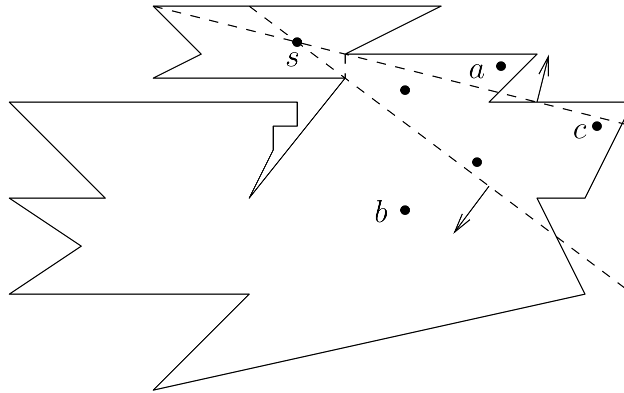

Figure 8.2: An example of two half-plane range queries from the site s

Thus, invisibility will be reported either once or twice. The time complexity of the above algorithm is then

$$O\left(\sum_{d\in\mathcal{S}^*}(m_d(\log m_d + \log mn)) + \overline{k}\right) = O\left(m \log m \log mn + \overline{k}\right) .$$

since $\sum_{d\in D} m_d$ is $O(m)$ where D is the set of all diagonals of a given depth. The space used for the point location structure, the factor graph and the query structure is still $O(m+n)$. Since the half-plane range query structure uses $O(m)$ space, the total amount of space used is $O(m+n)$. □

This result improves the running time of Theorem 8.1.1 by a factor of roughly $O(m^{1/2})$ and uses the same amount of time and space as the visibility algorithm.

Chapter 9

Future Work

After working with some special cases of the visibility graph problem, it would be interesting to know if some of the running times can be improved, since none of the described algorithms are optimal.

It seems hard to improve the main algorithm to compute the visibility graph within a simple polygon. The only thing to reduce is the $\log m \log mn$ factor on m. Apparently, the lower bound of the running time is poly-logarithmic in m, so it might be reduced to $O(n + m \log^2 m + k)$ or $O(n + m \log mn + k)$. It is, however, not obvious how to do this.

As noted at the end of Chapter 6, the visibility graph problem becomes harder by almost a factor $O(h)$ when the polygon contains h holes, contrary to other problems. This fact suggests that it might be possible to improve this algorithm. In that case, a different approach would probably be necessary.

Since it was possible to reduce the running time of the invisibility algorithm to poly-logarithmic time, there might be a possibility of improving the range-restricted algorithm to something similar. The approach with ray association might not work for this algorithm, since the range-restricted visibility graph is directed. Hence, we have to somehow preprocess P_r and query with each site in P_l and its range of sight. Then the preprocessing of m_d sites takes $O(m_d \operatorname{polylog} mn)$ time and the queries can be answered in poly-logarithmic time.

There are no known results on the 3-dimensional version of the visibility problem, where the visibility graph of points in space within a polyhedron is computed. It would be interesting to explore this special case, since it is widely used in computer graphics applications. It should be noted though, that this problem is expected to be much harder than the 2-dimensional problem.

Acknowledgements

First of all, I would like to thank my supervisor Gerth Stølting Brodal for excellent supervision, commitment and ideas for my thesis. I would also like to thank Niels Døssing and Torben Madsen for proofreading.

Appendix A

Notation

$|R|$ the number of vertices of the sub-polygon R, 36

$|S_i|$ the number of vertices of the subdivision S_i, 17

$\overline{AB}$ diagonal between A and B, 33

Δ a triangle of the triangulation, 14

d, e, f diagonal nodes in $\mathcal{S}^*$, 31

D_s the principal diagonals of s, 34

$depth(e)$ the depth of e in S or $\mathcal{S}^*$, 30

F a face of S, 17

γ_v the priority of the node $v \in \mathcal{T}$, 24

h the number of holes in Q, 11

$h(e)$ the height of e in S or $\mathcal{S}^*$, 31

i_s the part of a diagonal visible from the site s, 41

$H(d, e)$ the hourglass between diagonals d and e, 33

$I(v)$ the canonical subset of $v \in T$, 50

$Int(\mu)$ the elementary interval of the leaf μ, 50

k the number of edges in $VG_Q(P)$, 9

l^* the dual of the line l, 41

L_s the set of lines passing through s and a point in i_s, 41

L_s^* the dual of the line set, 41

m the number of sites in P, 7

μ a leaf of the segment tree T, 50

n the number of vertices in V, 9

p, q query points associated with sites in P, 17

P the sites within Q, 7

P_l the points in R_l, 52

P_r the points in R_r, 52

R_d the sub-polygon split by the diagonal d, 30

R_l the sub-polygon of R_d to the left of the diagonal d, 36

R_r the sub-polygon of R_d to the right of the diagonal d, 36

$\pi(A, B)$ the shortest path between A and B, 33

Q the polygon, 7

s, t sites in P, 9

S a subdivision of the plane, 17

S_i the subdivision of level i in the point location structure, 17

S_v^l segments stored in v that are associated with points in P_l, 52

S_v^r segments stored in v that are associated with points in P_r, 52

$\mathcal{S}$ the balanced decomposition, 14, 23

$\mathcal{S}^*$ the factor graph, 30

σ canonical segment of the segment tree T, 52

T a tree, 23, a segment tree, 50

$\mathcal{T}$ the trapezoidal graph, 15

U tournament tree, 24

V the vertices of Q, 9

$VG_Q(P)$ the visibility graph of P within Q, 9

Bibliography

[AM94] P. K. Agarwal and J. Matoušek. On Range Searching With Semialgebraic Sets (**Section 1**), *Discrete Comput. Geom.*, 11:393–418, 1994.

[B95] I. J. Balaban. An optimal algorithm for finding segment intersections (**Introduction**), *Proceedings of the 11th annual ACM Symposium on Computational Geometry*, 211–219, 1995.

[BC94] R. Bar-Yehuda and B. Chazelle. Triangulating Disjoint Jordan Chains (**Section 1**), *Internat. J. Comput. Geom. Appl.*, 4:475–481, 1994.

[BHKM04] B. Ben-Moshe, O. Hall-Holt, M. Katz, and J. Mitchell. Computing the Visibility Graph of Points Within a Polygon (**Sections 1–3,7**), *Proceedings of the 20th annual ACM Symposium on Computational Geometry*, 27–35, 2004.

[BKOS98] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. Computational Geometry - Algorithms and Application (**Section 10.3**), Springer-Verlag, 1998.

[C85] B. Chazelle. On the Convex Layers of a Planar Set (**Section 1**), *IEEE Transactions on Information Theory*, 509–517, 1985.

[CG89] B. Chazelle and L. Guibas. Visibility and Intersection Problems in Plane Geometry (**Section 1**), Discr. Comput. Geom., 4:551-581, 1989.

[C91] B. Chazelle. Triangulating a Simple Polygon in Linear Time (**Section 1**), *Discrete & Computational Geometry*, 6:485–524, 1991.

[CGL85] B. Chazelle, L. Guibas, and D. T. Lee. The Power of Geometric Duality (**Section 1**), *BIT, Vol. 25*, 76–90, 1985.

[GH89] L. Guibas and J. Hershberger. Optimal Shortest Path Queries in a Simple Polygon (**Sections 1–5**), *Journal of Computer and System Sciences*, 39(2):126–152, 1989.

[GHLST86] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan. Linear Time Algorithms for Visibility and Shortest Path Problems Inside Simple Polygons (**Sections 1,4**), *Proceedings of the 2nd annual ACM Symposium on Computational Geometry*, 1–13, 1986.

[GM91] S. K. Ghosh and D. M. Mount. An output-sensitive algorithm for computing visibility graphs (**Section 1**), *SIAM J. Comput.*, 20:888–910, 1991.

[GT98] M. T. Goodrich and R. Tamassia. Data Structures and Algorithms in Java (**Section 13.2**), John Wiley & Sons, 1998.

[H89] J. Hershberger. An optimal visibility graph algorithm for triangulated simple polygons (**Section 1**), *Algorithmica*, 4:141–155, 1989.

[K83] D. Kirkpatrick. Optimal Search in Planar Subdivisions (**Sections 1–4**), *SICOMP, Volume 12 Issue 1*, 28–35, 1983.

[KMM97] S. Kapoor, S. N. Maheshwari and J. S. B. Mitchell. An Efficient Algorithm for Euclidian Shortest Paths Among Polygonal Obstacles in the Place (**Section 1**), *Discrete & Computational Geometry*, 18:377–383, 1997.

[OL81] M. Overmars and H. van Leeuwen. Maintenance of Configurations in the Plane (**Sections 1,3**), *Journal of Computer and System Sciences*, 23:166–204, 1981.

Index

2-4 tree, 27

auxiliary tree, 23

balanced decomposition, 14
basic sub-polygon, 15

canonical subset, 50
common tangents of hourglasses, 35
concatenating hourglasses, 34
convex chain, 33
corridor, 46

definition vertex, 13
depth, 12
depth of a sub-polygon, 30
derived chain, 37
diagonal, 13
door, 46
dual graph, 46

elementary interval, 50

factor graph, 23

height, 12
hourglass, 33

independent set of vertices, 19
invisibility graph, 9, 55

junction triangle, 46

neighbourhood of a site, 20

outer common tangent of chains, 35

parent of a face, 17
principal diagonal, 34
priority, 24

range-restricted visibility graph, 9
ray shooting, 10

sector, 49
see, 9
segment tree, 49
site, 9
strong node, 24
subdivision, 17

tangent, 35
trapezoidal graph, 14
trapezoidal map, 13
trivial chain, 37

visibility graph, 9