

June 1987

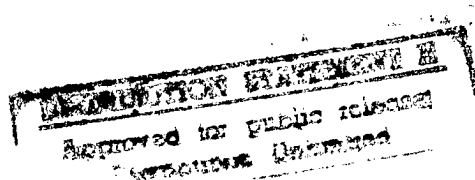
Report No. STAN-CS-87-1163



Efficient Algorithms for Shortest Path and Visibility Problems

by

John Edward Hershberger



Department of Computer Science

Stanford University
Stanford, CA 94305

100% QUALITY INSPECTION



19970516 033

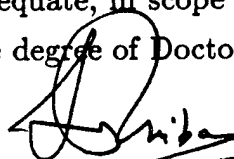
1997-02-17/2

EFFICIENT ALGORITHMS
FOR SHORTEST PATH AND VISIBILITY
PROBLEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
John Edward Hershberger
June 1987

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.



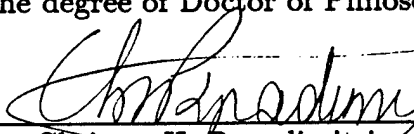
Leonidas J. Guibas
(Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.



Donald E. Knuth

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.



Christos H. Papadimitriou

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies

Abstract

Finding shortest paths and determining visibilities are problems encountered every day. Formal versions of these problems are important in computational geometry. Both kinds of problems specify a space and a set of opaque, impenetrable obstacles in the space. Visibility problems ask what an observer would see if placed in the space; shortest path problems seek the minimum length route for an object moving among the obstacles. This thesis provides algorithms for several visibility and shortest path problems, illuminating the relationship between the two classes.

The first part of the thesis considers problems in which the space is the plane and the obstacles are non-intersecting line segments. It presents a worst-case-optimal algorithm to find the visibility graph of the set of segments; that is, it computes what would be seen by observers standing at all the segment endpoints. It then uses this information to find shortest paths for a non-rotating convex body moving among the segments.

The second part of the thesis provides several optimal algorithms for shortest path and visibility problems inside simple polygons that have already been triangulated. In this setting, the polygon walls are the only obstacles. The most basic problem considered is that of finding all shortest paths from a particular vertex to other vertices. The solution to this problem can be applied to solve several visibility problems, including that of finding the visibility graph of a simple polygon in time proportional to its size. The thesis concludes by presenting geometric structures and corresponding data structures to solve several query problems efficiently, including the following: the *shooting problem*, which asks where a bullet fired inside a polygon would hit its boundary, and the *two-point shortest path* problem, which

asks for the length of the shortest path between two arbitrary query points.

Throughout the thesis, solutions to shortest path problems help solve visibility problems, and vice versa. The use of a single technique to solve problems of both classes demonstrates the close ties between the two.

Preface

All of the work described in this thesis has been or will be published separately. My thanks go to the holders of the copyrights for permission to use the material here. In accordance with their request, no part of this thesis should be reproduced for direct commercial advantage.

A version of Chapter 2 appeared as a joint paper with Takao Asano, Tetsuo Asano, Leo Guibas, and Hiroshi Imai. It appeared in the proceedings of the 1985 *IEEE Symposium on Foundations of Computer Science* [AAG*85] and was published in revised form in *Algorithmica* [AAG*86]. The joint paper was a result of simultaneous independent discovery. Leo Guibas and I found an $O(n^2)$ visibility graph algorithm in early 1985, which we described in the *Bulletin of the European Association for Theoretical Computer Science* [GH85]. The other three authors achieved the same result by similar means at about the same time, so we published our work together. The prose of Chapter 2 is my (substantial) revision of Imai's first draft.

Chapter 3 is mostly my own work. It appeared as a DEC/SRC technical report coauthored with Leo Guibas [HG86], and will be published in the *Journal of Algorithms*.

Chapter 4 is another case of simultaneous discovery. Leo Guibas and I found algorithms for all the problems discussed in that chapter except the convex rope problem of Section 4.1.1. When we learned that Micha Sharir, Daniel Leven, and Robert Tarjan had independently solved the same problems, we combined our work with theirs for publication. The resulting paper appeared in the 1986 *ACM Symposium on Computational Geometry* [GHL*86] and will be published in revised form

in *Algorithmica*. Most of the words in Chapter 4 come from Sharir's draft of the paper. The algorithm of Appendix A, which I developed, appeared in this paper.

The algorithms of Chapter 5 are my own. A paper describing them will be published in the 1987 *ACM Symposium on Computational Geometry* [Her87].

The results described in Chapter 6 will also appear in the 1987 *ACM Symposium on Computational Geometry* [GH87]. The work is chiefly my own, though Leo Guibas made important contributions.

Acknowledgements

The greatest debt I have to acknowledge is to my advisor, Leo Guibas, who introduced me to the field of computational geometry. I know no other field that provides such a fruitful opportunity to draw incomprehensible figures on restaurant napkins and placemats. Solving problems with Leo has been one of the greatest pleasures of the work recorded in this thesis. I've found few things as invigorating as our sessions of tossing wild ideas onto the blackboard, shooting them down, and molding them into something usable.

Several people contributed to this thesis by their comments on earlier versions of individual chapters. Chapters 3 and 5 improved significantly thanks to the careful reading of Jim Saxe and Jack Snoeyink. Don Knuth's many detailed comments improved both the technical content and the presentation of the results. He found a bug in Appendix A that had escaped the notice of all previous readers and reviewers. I owe a special debt to Cynthia Hibbard, whose comments on the prose style of Chapter 3 wilted me for weeks, but nonetheless helped me become a better writer. That which does not kill me makes me stronger, to use Nietzsche's words.

It is a good thing, I think, for grad students to have officemates. We need others around us to share our minor triumphs and disasters, to nod encouragingly while we bore them with our latest result, to bore us in turn with their own research. I thank all my officemates for their support through the years; with affection I single out for mention the other two NA apostates, my friends Dave Foulser and Jeff Naughton.

Finally, I thank my wife, Janie Hershberger, for her encouragement, advice, and general life-maintenance during the research and writing that went into this thesis. Few people could have stood the incessant explanations of my work as well as she

has. May I be as supportive during her thesis-writing.

Part of my financial support was provided by a National Science Foundation fellowship and by a US Army Research Office fellowship under agreement DAAG29-83-G-0020.

To Janie

Contents

Abstract	iii
Preface	v
Acknowledgements	vii
1 Introduction	1
2 The Visibility Graph of Segments	8
2.1 The visibility problem for a set of segments	10
2.2 Polar sorting and duality	13
2.3 Finding the visibility polygon: triangulation and set-union	14
2.4 Finding the visibility polygon: scanlines and segment splitting	17
2.5 Extensions	23
2.6 The visibility graph and the shortest-path problem	27
2.7 Concluding remarks	30
3 Moving a Non-Rotating Convex Body	31
3.1 Definitions	33
3.2 The path graph	34
3.3 Tangent-visibility	39
3.4 Construction of the augmented path graph	44
3.5 Pruning the path graph	52
3.6 Node coalescing in the pruned path graph	56
3.7 Conclusions and open questions	66

4	The Shortest Path Tree	68
4.1	Calculating a shortest path tree for a simple polygon	71
4.1.1	The convex rope algorithm	80
4.2	The shortest path map	81
4.3	Visibility within a simple polygon	84
4.4	Conclusion	92
5	The Visibility Graph of a Simple Polygon	93
5.1	Shortest path maps	95
5.2	Constructing shortest path maps	98
5.2.1	A sweeping-path algorithm	99
5.2.2	A depth-first search algorithm	105
5.3	Complexity bounds	108
5.4	Conclusion and open problems	110
6	Two-Point Shortest Path Queries	112
6.1	Balanced hierarchical decomposition of P	114
6.2	Shortest paths and the polygon decomposition	116
6.3	Hourglasses and shortest paths	119
6.3.1	Hourglass data structures: specifications and use	125
6.3.2	The chain data structure	129
6.3.3	Hourglass data structures: implementation	140
6.4	Improved query time bounds	143
6.5	Applications and extensions	146
7	Conclusions and Continuations	150
A	Balanced Decomposition of a Binary Tree	152
A.1	Constructing the auxiliary tree A	162
A.2	Decomposing the tree T	164
	Bibliography	168

List of Figures

1.1	Two definitions of visibility	4
2.1	The visibility problem for a set of segments	9
2.2	Solving the visibility problem	11
2.3	Constructing the directed graph $G(T_q)$	16
2.4	A set-union solution to the visibility problem	17
2.5	A right envelope	18
2.6	A scanline solution to the visibility problem	20
2.7	Subroutines used by the scanline algorithm	24
2.8	A parallel view	25
2.9	A visibility graph with $O(n)$ edges	29
3.1	Arcs bridge the difference in slopes between segments on the boundary of a fattened convex polygon	35
3.2	Although n barriers can intersect in $O(n^2)$ points, only a linear number of the intersections are on the boundary of the barriers	37
3.3	Barrier-connecting segments on a shortest path must be tangents	38
3.4	The visibility polygon	39
3.5	Tangent-visibility	40
3.6	Tangent-visibility notation	41
3.7	B^r hides B^p from B^q	42
3.8	Proof that $v_r(q, p)$ is an interval	42
3.9	Tangent directions and visibility directions	43
3.10	Five permutations satisfy Lemma 3.5	45
3.11	The tube $B^{\overline{p}r}$ and its internal parallelogram	47

3.12	Blocking intervals are merged	48
3.13	An algorithm to compute $TV(q)$	50
3.14	Proof that $\tilde{v}_T(q, p)$ is an interval	52
3.15	Edge e_2 cuts off a bay	53
3.16	Condition for pruning edge e_2	54
3.17	Pruning may remove many useless edges	55
3.18	Neighbor non-intersection implies global non-intersection	57
3.19	A forward-going subpath	58
3.20	Construction of G_d	58
3.21	The edge of G_c corresponding to the directed edge from a to b is assigned weight $ ab - a''a + bb' $	60
3.22	Forward-going paths are charged accurately in the coalesced graph .	61
3.23	Loops are undercharged, but not too much	61
3.24	Two loops connected by a single edge; (a) possible and (b) impossible configurations	63
3.25	A bound on the number of short edges in G_p	65
4.1	The shortest path tree	72
4.2	A triangulation and its dual	73
4.3	The funnel belonging to $\overline{uw}$	73
4.4	Splitting a funnel	74
4.5	The convex rope problem	81
4.6	Breaking a funnel region into triangles	83
4.7	Visibility of a polygon from a point	86
4.8	Visibility of a polygon from an edge	87
4.9	Visibility of one edge of a polygon from another, and the correspond- ing hourglass	89
5.1	An edge on the boundary of $Vis(\overline{st})$ belongs to $SPM(s)$ or $SPM(t)$.	98
5.2	Identifying edges on the boundary of $Vis(\overline{st})$	101
5.3	Vertices of $SPM(t)$ not in $SPM(s)$	102
5.4	Detecting vertices of $SPM(t)$	103
5.5	Multiple vertices on one extension edge	104
5.6	Region edges with trivial funnels	107

6.1	The hourglass of $\overline{AB}$ and $\overline{CD}$	120
6.2	Shortest paths in an open hourglass	122
6.3	Concatenating two closed hourglasses	122
6.4	Concatenating one closed and one open hourglass	123
6.5	Concatenating two open hourglasses	124
6.6	A chain is the convex hull of its subchains	130
6.7	A canonical example	132
6.8	Segment $\overline{ab}$ hides $\overline{cd}$	137
6.9	Using the hiding test at query time	138
6.10	Examples of fundamental strings	141
6.11	A derived string and its data structure	142
6.12	Definitions for query speedup	145
6.13	Computing the relative convex hull of some points in P	147
A.1	The ruler function tree	157
A.2	A more general tree	158

Chapter 1

Introduction

Learn everything; you will see afterwards that nothing is superfluous.

— Hugh of St. Victor (early twelfth century)

Visibility problems and shortest path problems occur frequently in everyday life. We often want to know what is visible from some point, say a corner office window, or how to move an object, say a case of Limburger cheese, from one place to another as quickly as possible. Related problems arise in computer science in application areas including graphics and robotics. In graphics, the *hidden surface removal* problem is a visibility problem of fundamental importance. In robotics, computer vision and robot motion planning pose a variety of visibility and shortest path problems. This thesis addresses formal versions of these problems, which are important in computational geometry.

Both types of problems specify a space and set of opaque, impenetrable obstacles in the space. Visibility problems then give a position or set of positions in the space and ask what part of the space is visible from those positions. Shortest path problems specify an object to be moved and its initial and final positions; the

desired solution is a path for the object between the given positions that avoids the obstacles and has minimum length, if any such path exists. There are many variants on the problems: the space may have two, three, or more dimensions; the obstacles may be subject to constraints (convexity, for example); the moving object may have constraints on its shape or motion; and any of several definitions of visibility and distance may be used. This thesis uses Euclidean length as the distance metric for shortest paths and straight-line visibility in its visibility problems. (A precise definition of visibility appears below.)

Even the simplest shortest path problems seem to be very difficult in three or more dimensions; the best algorithms known for a point moving among polyhedral obstacles require time exponential in the number of vertices of the obstacles. Therefore, this thesis considers shortest path problems in the plane, where the obstacles are disjoint simple polygons with a total of n vertices. The polygons are not allowed to intersect because intersections effectively increase the number of vertices. For precise characterization of algorithm running times, it is important to have all vertices be explicit.

The definition of visibility requires some care. At least two slightly different definitions are in current use. The first definition, which is used in this work, says that two points are *mutually visible* if the open segment connecting them does not intersect any obstacle. This is a natural definition: it corresponds to what we mean by visibility in daily life. We call this definition the *segment* definition. It has been used by several authors, including Asano et al. [AAG*86] and Chazelle and Guibas [CG85]. A competing definition has been used by El Gindy and Avis [EA81] and Guibas et al. [GHL*86], among others. This definition, which we call the *shortest path* definition, says that two points are mutually visible if the segment connecting them does not intersect the interior of any obstacle.

Both definitions have advantages and disadvantages. The chief advantage of the shortest path definition is its close relationship to shortest paths: two points are mutually visible if and only if the obstacle-avoiding shortest path between them is a straight segment. To understand the advantages of the segment definition, let us consider visibility from a point v . Let $visible(v, \theta)$ be the set of obstacle

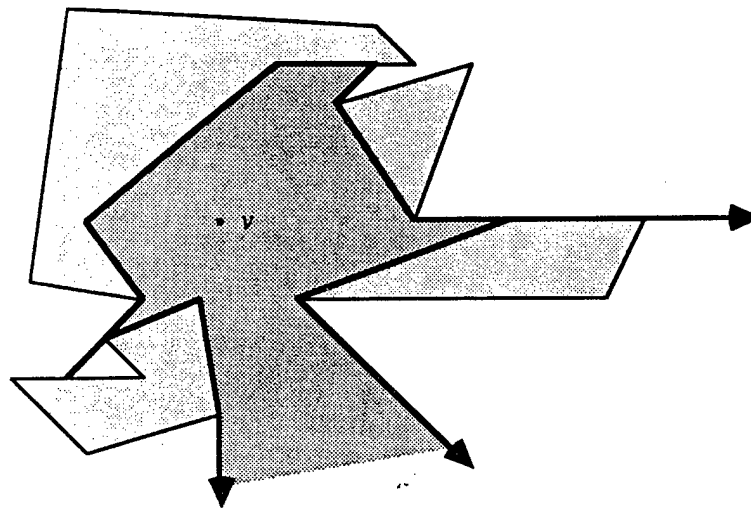
points visible from v in direction θ . Under the segment definition, $visible(v, \theta)$ is a function: there is at most one obstacle point visible in any direction. Under the shortest-path definition, $visible(v, \theta)$ can include arbitrarily many obstacle points; in fact, if the obstacles have degeneracies, $visible(v, \theta)$ can include arbitrarily many obstacle vertices. The set of all points visible from v is a connected region of the plane under both definitions, but under the shortest path definition it can have protruding rays and segments, as in Figure 1.1(a). A point on one of these rays has no visible points in its neighborhood except those on the ray. By contrast, the region visible from v under the segment definition, shown in Figure 1.1(b), has no such dangling segments or rays; however, it is not necessarily a closed region, as is the region visible from v under the shortest path definition.

Many authors minimize the differences between the two definitions by assuming that no three obstacle vertices are collinear (counting v among them). One of the goals of this thesis is to cope gracefully with degeneracies, so the algorithms presented here use the segment definition of visibility, which has fewer inherent liabilities in the presence of collinear points. When the algorithms need to compute the region visible from a point (which may not be closed), they calculate instead its boundary, which is always closed.

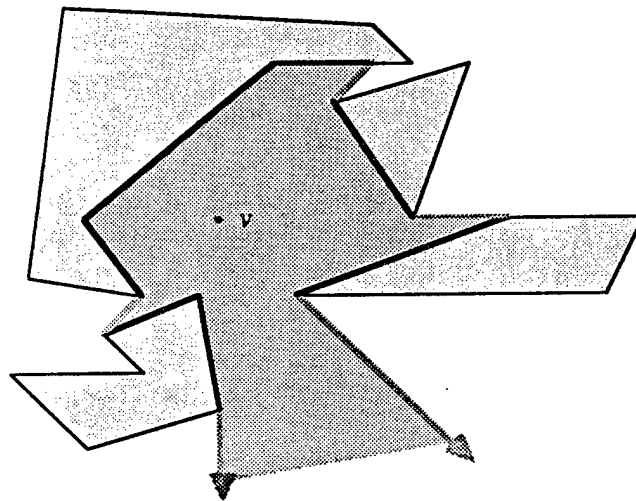
The algorithms presented in this thesis depend on the close relationship between shortest paths and visibility. If two points are mutually visible, the *visibility segment* connecting them is also the shortest obstacle-avoiding path between them. On the other hand, a shortest path for a point moving among polygonal obstacles is a polygonal path; each consecutive pair of vertices on the path is either mutually visible or connected by a polygon edge. The algorithms that follow exploit this connection in both directions: Chapters 2 and 3 use visibility information to find shortest paths, while Chapters 4 and 5 construct shortest paths, then use them to solve visibility problems.

The thesis is divided into two main sections. The first part addresses problems involving arbitrarily many polygonal obstacles, while the second part obtains improved bounds for the special case in which there is only one polygon.

Chapters 2 and 3 comprise the first part of the thesis. Chapter 2 solves the



(a)



(b)

Figure 1.1. Two definitions of visibility: (a) the shortest path definition, and (b) the segment definition.

visibility graph and *visibility polygon* problems for a set of n disjoint line segments (a slight generalization of disjoint polygons). The visibility polygon problem asks for the boundary of the region of the plane visible from a query point v . That is, the desired solution is the function $visible(v, \theta)$ described above. The chapter gives an algorithm to find the visibility polygon of an arbitrary query point in $O(n)$ time after $O(n^2)$ preprocessing. Clearly, the algorithm is most useful when many visibility polygons must be computed for the same set of obstacles. The visibility graph problem is an ideal application. The visibility graph is a combinatorial graph structure with segment endpoints as nodes and an edge between every pair of mutually visible vertices. It can be built by finding the visibility polygon from every vertex. Since the visibility graph can have $\Theta(n^2)$ edges, this construction algorithm is worst-case optimal if all edges must be listed.

As noted above, shortest paths between segment endpoints follow visibility graph edges and polygon edges. It is possible to find the shortest path between two vertices in $O(n^2)$ time by building the visibility graph, augmenting it with polygon edges, and then running Dijkstra's shortest path algorithm for graphs on it [AHU74, pages 207–209]. Dijkstra's algorithm runs in $O(n^2)$ time on this graph when implemented using the Fibonacci heaps of Fredman and Tarjan [FT84]. (This bound assumes that the Euclidean distance between two points can be computed in constant time, and also that sums of such distances can be compared in constant time.) Chapter 3 extends this approach to find shortest paths for a non-rotating convex object. The extension uses the “configuration space” technique of Lozano-Perez and Wesley [LW79] to transform the problem of moving a convex object among obstacles into that of moving the object's center among “fattened” versions of the obstacles. An analogue to the visibility graph, called the *path graph*, records all the sub-paths that belong to shortest paths among the fattened obstacles. The algorithm of Chapter 3 constructs this graph from the visibility graph of the polygons, then uses Dijkstra's algorithm to find the desired shortest path in $O(n^2)$ time.

The algorithms of Chapters 2 and 3 require $\Omega(n^2)$ time because they do not take into account the structure of the obstacles. Chapters 4 through 6 improve on these bounds when the only obstacle is a single simple polygon P with n vertices.

In this setting shortest paths and visibility segments lie inside the polygon. All of the algorithms presented here are triangulation-based: given a triangulation of the polygon (a partitioning of its interior into $n - 2$ disjoint triangles by $n - 3$ non-intersecting diagonals), the algorithms run in linear time, which is optimal. The best triangulation algorithm known is that of Tarjan and Van Wyk [TV86], which runs in $O(n \log \log n)$ time. (Triangulation-based linear algorithms like those presented here should, of course, be contrasted with linear-time algorithms on “raw” simple polygons, such as calculation of the convex hull of such a polygon P [GY83,MA79] and calculation of the subpolygon of P visible from a given point [EA81, Lee83].)

Chapter 4 presents an algorithm that, given a triangulation of a simple polygon P , computes all shortest paths from a source inside P to the vertices of the polygon. This structure, called the *shortest path tree*, is then used to solve a variety of shortest path and visibility problems. These problems include calculation of the subpolygon of P visible from a given segment within P , preprocessing P for fast “ray shooting” queries, and several related problems. Chapter 5 uses the shortest path tree plus some additional techniques to build the visibility graph of P in $O(m + n \log \log n)$ time, where m is the number of edges in the visibility graph. This improves substantially on the $O(n^2)$ algorithm of Chapter 2, since m can be as small as $O(n)$.

The shortest path tree of Chapter 4 can be enhanced to answer shortest path queries. After a linear amount of additional processing, it can be used to compute in $O(\log n)$ time the distance from the source to an arbitrary query point inside P . The path itself can be extracted in time proportional to the number of turns along it. A limitation of this approach is that the source is fixed; if the source is also arbitrary, queries may take linear time to answer. Chapter 6 corrects the flaw, introducing data structures that allow arbitrary shortest path queries. Once the structures are built (in linear time and space), the length of the shortest path between two arbitrary points can be computed in $O(\log n)$ time, and the path itself can be produced in time proportional to the number of turns along it. The data structure has several further applications: for example, it can be used to solve the all-farthest neighbors problem inside P in $O(n \log n)$ time, and to build the relative

convex hull of m points inside P in $O(m \log m + m \log n)$ time.

Chapter 2

Visibility Polygons and the Visibility Graph of Segments

I can see clearly now; the rain is gone.

I can see all obstacles in my way.

— Johnny Nash, “I Can See Clearly Now” (1972)

The visibility-polygon problem, or the hidden-line elimination problem in the plane, is of fundamental importance in computer graphics. It is also important in computational geometry, since it is often used as a subproblem in other geometric problems, such as the shortest-path problem in the plane with polygonal obstacles. The problem is formally stated as follows: Given a set of h disjoint polygons with n edges and a query point q , report in polar order all the boundary points of those polygons that are visible from q . For the case of a single simple polygon, El Gindy and Avis [EA81] and Lee [Lee83] presented $O(n)$ time algorithms. Asano [Asa84] gave an $O(n + h \log h)$ time algorithm for the case where the h disjoint polygons are convex, and an $O(n \log h)$ time algorithm for the general problem.

In this chapter we consider a canonical case of the visibility polygon problem, in which the obstacles are straight-line segments: Given a set of n segments with arbitrary slopes, non-intersecting except at their endpoints, and an arbitrary query point q , find the parts of those segments that are visible from q , where a point p is visible from q if the open line segment $\overline{pq}$ intersects no segment in the set (Figure 2.1). We use $\text{Vis}(S, q)$ to refer to the set of all points visible from q in the presence of S , and $\text{VP}(S, q)$, or just $\text{VP}(q)$, to denote the set of points of S visible from q . The set $\text{VP}(q)$ lies on the boundary of $\text{Vis}(S, q)$. This chapter presents two algorithms that compute $\text{VP}(q)$ in $O(n)$ time with $O(n^2)$ preprocessing time and $O(n^2)$ space. The algorithms can easily be modified to solve the visibility problem for a set of disjoint polygons with n edges in the same complexity bounds.

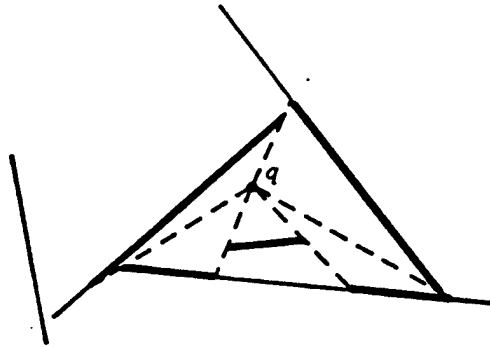


Figure 2.1. The visibility problem for a set of segments

Our methods are especially efficient when the visibility problem is solved repeatedly for the same set of polygons, which is the case in constructing the visibility graph. The visibility graph of disjoint polygons with n edges can be found by solving the visibility problem from each vertex of those polygons, and hence it can be constructed in $O(n^2)$ time and space. This problem had been previously solved in $O(n^2 \log n)$ time by Lee [Lee78]. Independently, Welzl [Wel85] has discovered a different $O(n^2)$ solution. Since the visibility graph can have size $\Omega(n^2)$, our methods and Welzl's are optimal to within a constant factor, if we assume that all the edges of the graph must be listed.

The visibility graph can be used to find shortest paths. The graph formed by adding all polygon edges to the visibility graph contains all shortest paths between polygon vertices. The shortest path between a particular pair of vertices can be found in $O(n^2)$ time by applying Dijkstra's algorithm [AHU74, pages 207–209][FT84] to this graph (Lee and Preparata [LP84], Sharir and Schorr [SS84]). This improves the $O(n^2 \log n)$ bound for the shortest-path problem due to Sharir and Schorr [SS84]. However, this is not the best bound known for the shortest path problem. When the obstacles consist of k disjoint polygons, the algorithm of Reif and Storer [RS85] solves the shortest path problem in $O(kn + n \log n)$ time.

The algorithms mentioned in the preceding paragraph find shortest paths for a point moving among polygonal obstacles. Chapter 3 uses the visibility graph to solve a more difficult problem, that of finding shortest paths for a non-rotating convex object.

2.1 The visibility problem for a set of segments

Let S be a set of n segments with arbitrary slopes allowed to intersect only at their endpoints, and let q be an arbitrary query point. We shall describe in top-down fashion two algorithms for finding the parts of the segments in S that are visible from q . Let p_i ($i = 1, \dots, N$) be all the endpoints of segments in S , where $N \leq 2n$. For simplicity, we assume that no two points in the set $\{q, p_1, \dots, p_N\}$ have the same x -coordinate (this assumption can be removed by techniques of Chazelle, Guibas and Lee [CGL85]). Our algorithms also work when the segment endpoints $p_1, \dots, p_N$ have distinct x -coordinates and $q \in \{p_1, \dots, p_N\}$. The algorithms solve the visibility problem by decomposing it into the following two subproblems.

(1) **Polar Sorting:** Consider a polar coordinate system with the point q as the origin and the vertical ray directed upward from q as the reference. Let us denote the polar angle of a point p by $\theta_q(p)$, where the polar angle increases counterclockwise around q . Problem (1) is to find a sorted list of p_i ($i = 1, \dots, N$) in increasing order of $\theta_q(p_i)$.

We define the rank $x(p_i)$ of point p_i ($i = 1, \dots, N$) to be its rank in the sorted list obtained in Problem (1). Points with the same θ_q are given the same rank. The maximum rank is $N_x \leq N$. For $0 \leq i \leq N_x + 1$, we use $\alpha(i)$ for the angle corresponding to rank i , where $\alpha(0) = 0$ and $\alpha(N_x + 1) = 2\pi$.

Before we introduce the second subproblem, let us perform certain normalizations on the set S . We first remove the segments whose endpoints and q are collinear, and then break into two separate parts each segment in S that intersects the vertical ray $\vec{r}$ emanating upward from q . The newly created endpoints all have the same x -coordinate, but this causes no trouble; our algorithms require only the *original* endpoints to have distinct x -coordinates. We denote the resulting set of segments by S_q , and let $n_q = |S_q|$. For two segments s and s' in S_q , we say that s is in front of s' , written $s \prec_q s'$, if there exists a non-vertical ray emanating from q that intersects both segments and hits s before s' . Chazelle [Cha83] shows that the relation $\prec_q$ can be embedded in a total order; see Figure 2.2(a). One of our two algorithms computes such a total order of the segments in S_q compatible with $\prec_q$. For each segment s in S_q , we define $y(s)$ to be the rank (from 1 to n_q) of s in this total order.

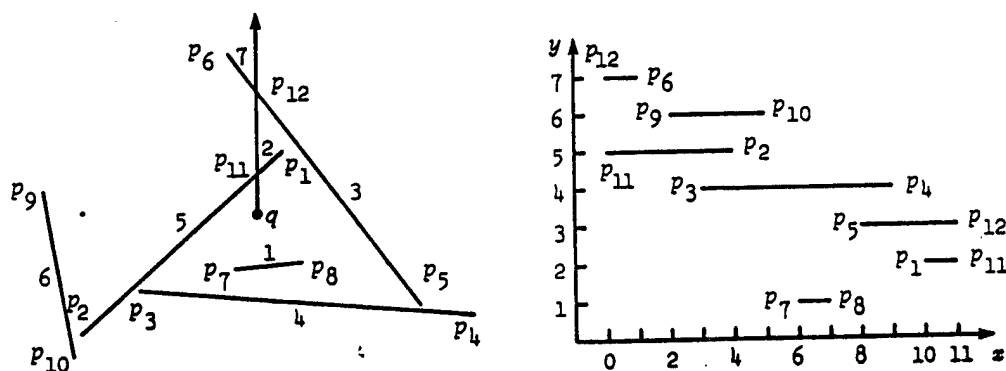


Figure 2.2. Solving the visibility problem. In (b), the x -coordinates correspond to the polar order of the segment endpoints, and the y -coordinates correspond to a total ordering of the segments based on the “in front of” relation.

We define the interval $I_x(s)$ of each segment in S_q as follows: Suppose the endpoints of s in S are u and v with $x(u) < x(v)$. If s does not intersect $\vec{r}$, then s is in S_q and we let $I_x(s)$ be $[x(u), x(v)]$. If on the other hand s intersects $\vec{r}$ at w , then it gives rise to two segments $\overline{uw}$ and $\overline{wv}$ in S_q , and we set $I_x(\overline{uw})$ to be $[0, x(u)]$ and $I_x(\overline{wv})$ to be $[x(v), N_x + 1]$.

The visibility polygon of q is defined by considering rays that emanate from q and proceed until one of the given line segments is encountered. Notice that all the rays in the angular range $(\alpha(k), \alpha(k+1))$, $0 \leq k \leq N_x$, have their first intersection with the same segment. We allow, of course, for the possibility that a ray may encounter no segment at all in some angular intervals.

(2) Visibility of Sorted Segments: For each $k = 0, 1, \dots, N_x$, we define $visible(k)$ to be the segment in S_q intersected by all rays from q in the angular interval $(\alpha(k), \alpha(k+1))$. If no such segment exists, we set $visible(k)$ to be $\emptyset$. Problem (2) is to find a polar view of the intervals, that is, to find $visible(k)$ for each $k = 0, 1, \dots, N_x$ (see Figure 2.2(b)).

This definition of *visible* is a variant of the definition used in the introduction. This is a discrete version of that continuous definition.

Even though S_q does not include segments lying on lines through q , such segments may occur in S . The visible portion of a segment lying on a ray from q is at most its endpoint p nearer q . Visibility can be checked by determining which is the nearest to q , on the ray emanating from q towards p , among p and segments $visible(k-1)$ and $visible(k)$, where $k = x(p)$. If q lies on a segment of S , nothing is visible from q in the directions of the segment endpoints. Thus, if we have already solved problems (1) and (2), we can find in polar order the parts of the segments of S that are visible from q in $O(n)$ time.

In the above problems, as depicted in Figure 2.2, we cut the plane along the ray emanating upward from q and spread it out according to the angular and radial orders $x(p_i)$ and $y(s)$. (Note that although it is convenient to think of embedding the “in front of” relation $\prec_q$ in a total order, it is not necessary to do so.) In this spread-out view, the lowest segment in interval $[k, k+1]$ is $visible(k)$; Problem (2) is

to find the lowest segment in all intervals, the *lower envelope* of the set of segments. It will be convenient in our second algorithm to think of Problem (2) in terms of the spread-out view.

In the next section we establish the following lemma:

Lemma 2.1. *Using $O(n^2)$ preprocessing time and $O(n^2)$ space for constructing an arrangement of lines, we can find the sorted list of p_i ($i = 1, \dots, N$) in increasing order of $\theta_q(p_i)$ in $O(n)$ time for each query point q .*

In Sections 2.3 and 2.4 we present two different linear solutions to Problem (2). The first approach uses a triangulation to construct the total order $y(s)$ explicitly in linear time, then applies the linear set-union algorithm of Gabow and Tarjan [GT83] to find $\text{visible}(k)$ for $0 \leq k \leq N_x$ in $O(n)$ time. The second method uses a scanline to sweep through the segment endpoints in polar order. By breaking each segment in S_q into two pieces, it produces regularized segments for which visibilities can be found in $O(n)$ time using bit operations or table lookup.

From the lemma and the preceding discussion, we obtain the following theorem:

Theorem 2.1. *For a set of n arbitrarily oriented segments and a query point q , the parts of the segments that are visible from q can be found in $O(n)$ time with $O(n^2)$ preprocessing time and $O(n^2)$ space.*

2.2 Polar sorting and duality

For the set of points p_i ($i = 1, \dots, N$), consider the following geometric transformation $\mathcal{T}$, as used by Brown [Bro80]: Let l be a non-vertical line in the plane whose points (x, y) satisfy $y = a_l x + b_l$, and let $p = (x_p, y_p)$ be a point in the plane. Transformation $\mathcal{T}$ maps l into the point $p_{\mathcal{T}}(l) = (a_l, b_l)$ and p into the line $l_{\mathcal{T}}(p)$ whose points (a, b) satisfy $b = -x_p a + y_p$. This transformation is an example of a duality mapping.

Geometric duality in two dimensions maps lines to points and points to lines while preserving their relationships. For example, if two lines map to a pair of

points in the dual space, their intersection point maps to the line connecting the dual points. Duality is a powerful tool; for examples of its use in computational geometry, see [CGL85,CG85,GRS83,Sto87]. We use duality again in Section 4.3.

Using the transformation $\mathcal{T}$, we can solve Problem (1) as follows: In the preprocessing step, we map the points p_i ($i = 1, \dots, N$) into lines by $\mathcal{T}$. These lines and their intersections induce a subdivision of the mapped plane, which we construct in $O(n^2)$ time and space using the algorithm of Chazelle, Guibas and Lee [CGL85] or Edelsbrunner, O'Rourke and Seidel [EOS83]. Such a subdivision is called the *arrangement* of the lines [Gru72]. Given a query point q , we map it by $\mathcal{T}$ into the line $l_{\mathcal{T}}(q)$ and insert it into the arrangement, which can be done in $O(n)$ time [CGL85,EOS83]. The line connecting points q and p_i in the original plane corresponds to the point of intersection of the two lines $l_{\mathcal{T}}(q)$ and $l_{\mathcal{T}}(p_i)$ in the mapped plane. (Because no two points in $\{q, p_1, \dots, p_N\}$ have the same x -coordinate, there are no parallel lines in the arrangement.) The ordering by slope of the lines connecting q and p_i ($i = 1, \dots, N$) corresponds to the ordering by x -coordinate of the points of intersection of the line $l_{\mathcal{T}}(q)$ with lines $l_{\mathcal{T}}(p_i)$ ($i = 1, \dots, N$) in the mapped plane. Since we have the arrangement at hand, this ordering can be obtained in $O(n)$ time.

For a set of points p_i with $0 \leq \theta_q(p_i) < \pi$ (resp. $\pi \leq \theta_q(p_i) < 2\pi$), the ordering of these points in increasing order of $\theta_q(p_i)$ is precisely the ordering of them in increasing order of the slopes of the lines joining q and p_i . Thus the ordering of the points p_i in increasing order of θ_q can be obtained from their ordering with respect to slope in $O(n)$ time. This proves Lemma 2.1.

2.3 Finding the visibility polygon: triangulation and set-union

In our first solution to Problem (2), we produce a total order $y(s)$ in which the “in front of” relation $\prec_q$ may be embedded. To do this, we triangulate the set S of segments in the preprocessing step; specifically, we triangulate the convex hull

of S so that all the segments in S are edges of the triangulation. This can be done in $O(n \log n)$ time and $O(n)$ space (El Gindy and Toussaint [ET84], Garey et al. [GJPT78]). We denote by T the set of edges of this triangulation, where we have $S \subseteq T$.

For a given q , we modify T to produce a new triangulation. To do this, we first stretch a line segment up from q to the boundary of the convex hull. When we add this segment, some faces become non-triangles. Adding new segments to such faces, we can obtain in $O(n)$ time a new triangulation whose edge set, denoted by T_q , contains S_q . Then we construct a directed graph $G(T_q)$ whose vertex set is T_q and whose arcs are obtained as follows: For each triangle composed of edges $e_1, e_2, e_3 \in T_q$ as depicted in Figure 2.3,

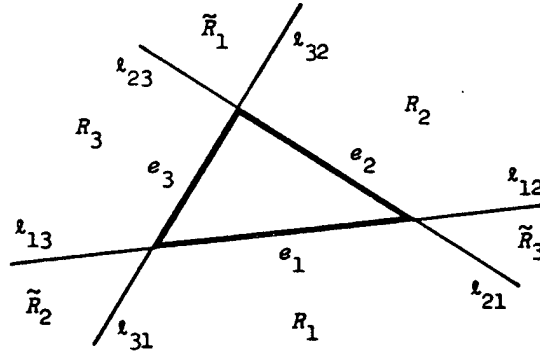
- if point q is in the interior of region R_1 , then add arcs from e_2 to e_1 and from e_3 to e_1 (the cases where q is in the interior of region R_2 or R_3 are similar);
- if point q is in the interior of $\tilde{R}_1$, then add arcs from e_1 to e_2 and from e_1 to e_3 (the cases where q is in the interior of region $\tilde{R}_2$ or $\tilde{R}_3$ are similar);
- if point q is on the half line l_{12} , but not a vertex of the triangle, add an arc from e_3 to e_2 (the cases where q is on l_{21} , l_{31} , l_{13} , l_{23} or l_{32} are similar);
- if point q is on some edge or in the interior of the triangle, do nothing.

For this graph $G(T_q)$, we have the following lemma.

Lemma 2.2. *The directed graph $G(T_q)$ is acyclic. Furthermore, for $s, s' \in S_q$ such that $s \prec_q s'$, there is a directed path from s' to s in $G(T_q)$.*

Proof: Let us consider extending the relation $\prec_q$ to all edges of T_q to get $\prec'_q$. The relation can still be embedded in a total order [Cha83], and since the arcs of $G(T_q)$ record a subset of $\prec'_q$, $G(T_q)$ must be acyclic.

Any two segments $s, s' \in S_q$ with $s \prec_q s'$ intersect a common ray from q . The $\prec'_q$ relations between edges of T_q that cross this ray at adjacent intersections are all recorded in $G(T_q)$, and hence there is a directed path from s' to s in $G(T_q)$. ■

Figure 2.3. Constructing the directed graph $G(T_q)$

A topological order of $G(T_q)$ can be found in $O(n)$ time (note that the number of vertices and arcs of $G(T_q)$ is $O(n)$). From the topological order, which is a total order on T_q , we can obtain in $O(n)$ time a total order on S_q in which the relation $\prec_q$ is embedded.

In order to obtain a linear-time algorithm for Problem (2), we employ the set-union algorithm of Gabow and Tarjan [GT83], which solves the following problem: Initially, there are $N_x + 2$ disjoint sets $\{0\}, \{1\}, \{2\}, \dots, \{N_x\}, \{N_x + 1\}$; two operations may be performed on this system of disjoint sets, $find(k)$ and $link(k)$ for $0 \leq k \leq N_x$. The function $find(k)$ returns the maximum element of the set containing k , and $link(k)$ unites the set containing k with the set containing $k + 1$. Gabow and Tarjan's algorithm executes, on-line, a sequence of $O(N_x + n_q)$ $find$ and $link$ operations in $O(N_x + n_q)$ time [GT83]. Using these operations, Problem (2) can be solved by the algorithm of Figure 2.4.

The algorithm keeps track of contiguous blocks of intervals for which $visible(k)$ is known. It maintains the following invariant: two elements k and k' ($k < k'$) belong to the same set if and only if $visible(j)$ has been found for every $k \leq j < k'$. Thus, for any k in the range $0 \leq k \leq N_x$, if $find(k) \neq N_x + 1$, then $visible(find(k))$ is unknown. This means that the algorithm sets $visible(k)$ correctly and at most once for each $0 \leq k \leq N_x$. Hence $find$ and $link$ operations are executed $O(n)$ times and the algorithm runs in linear time.

```

for  $j := 1$  to  $n_q$  do
  begin
    Let  $s$  be the segment in  $S_q$  with  $y(s) = j$ ;
    Let  $[l, r]$  be the interval  $I_x(s)$  of  $s$ ;
     $l := find(l)$ ;
    while  $l < r$  do
      begin
         $visible(l) := s$ ;  $link(l)$ ;  $l := find(l)$ 
      end
    end
  end
end

```

Figure 2.4. A set-union solution to the visibility problem

2.4 Finding the visibility polygon: scanlines and segment splitting

The second algorithm for Problem (2) does not compute the $y(s)$ order of the segments; it does not require either a triangulation or the set-union data structure. Instead, it uses a polar scanline to sweep through the sorted list of segments and find the visible sub-segments. It will be convenient in this algorithm to think of the segments as being spread out as in Figure 2.2(b); Problem (2) in this context is the problem of finding the lower envelope, and the scanline is vertical. We will describe the algorithm in two passes: first we give an easily-explained but slower method, and then we describe the modifications needed to make it linear.

If a segment s has endpoints p_i and p_j with $x(p_i) \leq x(p_j)$, let us define its left and right endpoint ranks $l(s) = x(p_i)$ and $r(s) = x(p_j)$. In what follows we will often identify an endpoint p with its rank $x(p)$.

The solution to Problem (1) gives us the segments' endpoints in sorted order; running through the list from left to right corresponds to sweeping a scanline through the segments. If the list gives each segment's length along with its left endpoint, an algorithm has enough information to compute the lower envelope to the right of the scanline (the *right envelope*) for those segments that start to the scanline's left. Since every segment that contributes to the right envelope must

cross the scanline, this envelope looks like a staircase, monotonically ascending to the right. Every segment in it is lowest just once, from its right endpoint part way back to the scanline. See Figure 2.5 for an example. The right envelope is important for the following reason: when the scanline moves from $\bar{x} = k$ to $\bar{x} = k + 1$ (all points p_i with $x(p_i) \leq k$ have been processed), the lowest segment of the current right envelope is *visible*(k).

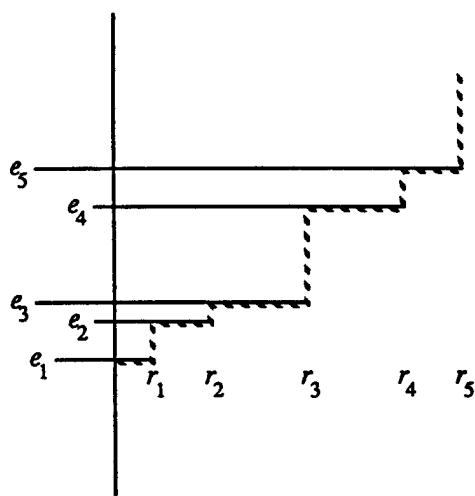


Figure 2.5. A right envelope

This observation suggests a way to build the lower envelope incrementally: we just need to maintain the current right envelope as the scanline sweeps across the segments. For a given position of the scanline $\bar{x}$, let the set of segments in the right envelope be $E = \{e_1, e_2, \dots, e_m\}$. This is a subset, possibly a proper one, of the segments whose intervals contain $\bar{x}$. Let the right endpoint rank of a segment $e_i \in E$ be $r_i = r(e_i)$. Number the elements of E so that $\bar{x} = r_0 < r_1 < r_2 < \dots < r_m < r_{m+1} = \infty$. The left endpoints of segments in E are all at or to the left of $\bar{x}$. The segments themselves are comparable under the $\prec_q$ relation, and $e_i \prec_q e_{i+1}$ for $1 \leq i < m$.

Let $\hat{s}$ be a segment whose left endpoint is encountered by the scanline at $\bar{x}$. The right endpoint $r(\hat{s})$ falls somewhere in the staircase pattern, that is, $r_i < r(\hat{s}) \leq r_{i+1}$

for some $0 \leq i \leq m$. Once the position of $r(\hat{s})$ in the endpoint list is known, the lower envelope can be updated in constant amortized time per segment. Let i be the index such that $r_i < r(\hat{s}) \leq r_{i+1}$. If $e_{i+1} \prec_q \hat{s}$, then $\hat{s}$ does not contribute to the lower envelope and may be dropped. On the other hand, if $\hat{s} \prec_q e_j$ for some $j \leq i$, then segment e_j should be dropped from the current right lower envelope. If $r(\hat{s}) = r_{i+1}$ and $\hat{s} \prec_q e_{i+1}$, then the segment e_{i+1} should also be removed from E .

When the scanline encounters a right endpoint, the corresponding segment may or may not appear in the current right envelope. If the segment is not in the right envelope, then no action is needed. If the segment does appear in the right envelope, then it is e_1 , the lowest segment in it. (It could not be e_i for $i > 1$, since that implies $r_1 > r_i$, a contradiction.) In this case e_1 must be removed from E and the segments below it advanced. Segment e_2 becomes e_1 , and so on.

The implementation of the algorithm in Figure 2.6 keeps E as a list ordered by $r(s)$, modifying it as the scanline sweeps across endpoints from left to right. In the code, X is a sorted list of right and left endpoints (two per segment in S_q), each associated with its segment. The program manipulates the list E using standard list operators *Delete*(s), *Insert*(s, t), *Pred*(s), and *Head*(E), which respectively delete s from E , insert s after t in E , return the predecessor of s in E , and return the element s in E with minimum $r(s)$. To insure that the *Insert* operator can always be applied, the list E is preceded by a dummy segment e_0 . The program also uses a nonstandard function *ShortestAsLongAs*(s), which returns the segment s^* in E , if any, ending immediately to the right of s . (Formally, $r(s^*) = \min \{r(s') \mid s' \in E, r(s') \geq r(s)\}$.) Note that in the algorithm of Figure 2.6, E never contains two segments s and t with $r(s) = r(t)$. This fact is important in the implementation of *Insert* given in Figure 2.7.

Except for the problem of locating $r(\hat{s})$ among the endpoints of E , the list operations in the program of Figure 2.6 can be performed in linear time overall using doubly-linked lists. Unfortunately, the one-dimensional point-location problem solved by the function *ShortestAsLongAs*(s) requires $\Omega(\log n)$ average time per query in the general case, as well as more complicated data structures, and hence the algorithm presented so far runs no faster than $\Omega(n \log n)$ in the worst case.

```

last_k := 0;
Initialize  $E$ , the current right envelope, to be empty except for the
dummy element  $e_0$ ;
while  $X$  is non-empty do
begin
  Remove the next endpoint  $p$  from  $X$ ;   $cur\_k := x(p)$ ;
  for  $k := last\_k$  to  $cur\_k - 1$  do
     $visible(k) := Head(E)$ ;
   $last\_k := cur\_k$ ;
  Let  $s$  be the segment associated with  $p$ ;
  if  $p$  is the left endpoint of  $s$  then
    begin
       $t := ShortestAsLongAs(s)$ ;
      /* Locates  $r(s)$  in  $(r_0, r_1, \dots, r_{m+1})$ . */
      if  $s \prec_q t$  then
        /*  $s$  is not obscured by  $t$ . */
        begin
           $p := Pred(t)$ ;
          if  $r(s) = r(t)$  then  $Delete(t)$ ;
           $Insert(s, p)$ ;
          /* Insert  $s$  after  $p$  in  $E$ . Note that  $E$  never contains two
             segments with equal right endpoints. */
          while  $p \neq \emptyset$  and  $s \prec_q p$  do
            begin
               $p' := Pred(p)$ ;   $Delete(p)$ ;   $p := p'$ ;
            end
          end
        end
      else
        /* This is a right endpoint. */
        if  $s = Head(E)$  then  $Delete(Head(E))$ ;
        /* else already deleted. */
      end
    end
  end
end

```

Figure 2.6. A scanline solution to the visibility problem

Modification of the segments can be used to construct a linear algorithm. As a first step, note that two lower envelopes (essentially segment lists with at most one segment in any interval $[k, k+1]$) can be merged into their common lower envelope in linear time. In particular, if an algorithm splits each segment into two parts and independently computes the lower envelopes of the left parts and the right parts, the resulting pair of lists can be merged in linear time. This observation will be useful if we can find a way of splitting the original segments that makes point location easy in the resulting subproblems. To this end we define the following function of two integers.

The *least common ancestor* $lca(a, b)$ of a pair of positive integers $a \leq b$ is the integer c , $a \leq c \leq b$, with a maximal number of trailing zeroes when written in binary. We can write a and b in binary as

$$\begin{aligned} a &= \alpha 0 \beta \\ b &= \alpha 1 \gamma, \end{aligned}$$

where the lengths of β and γ are both equal to some integer k . Then $lca(a, b) = \alpha 1 0^k$ unless $\beta = 0^k$, in which case $lca(a, b) = a$. For example, $lca(5, 15) = 8 = (1000)_2$ and $lca(14, 15) = 14$. The lemma below shows that $lca(a, b)$ is unique. A trivial but useful fact is that $lca(a, b) = lca(a, lca(a, b))$. The lca function can be computed in a constant number of operations under the RAM model, as shown by Harel [Har80].

Lemma 2.3. *If $lca(a, b) = c = d 2^i$ for some odd integer d , then no $c' \neq c$ in the interval $[a, b]$ is divisible by 2^i .*

Proof: Suppose to the contrary that there is some $c' \neq c$, $a \leq c' \leq b$, such that $c' = d' 2^i$ for some odd integer $d' \neq d$. Then there is an even integer $\hat{d}$ strictly between d and d' , $\hat{d} = \bar{d} 2^j$ for $j > 0$ and $\bar{d}$ odd. But then $\hat{c} = \hat{d} 2^i = \bar{d} 2^{i+j}$ is strictly between a and b and has more trailing zeroes than c , contradicting the fact that $c = lca(a, b)$. ■

Let us consider splitting segments at the point indexed by the least common ancestor of their endpoint ranks. That is, we split a segment s with $I_x(s) = [l, r]$ into

segments s_1 and s_2 with $I_x(s_1) = [l, lca(l, r)]$ and $I_x(s_2) = [lca(l, r), r]$. Note that $lca(l, r)$ may be equal to l or r , and thus the splitting may not divide the segment at all. Let $\mathcal{L}$ be the set of segment left portions and $\mathcal{R}$ be the symmetrically defined set of right portions. Since finding the lower envelope of $\mathcal{R}$ is the mirror image of finding that of $\mathcal{L}$, we will consider only the latter case. Let us classify segments in $\mathcal{L}$ by the number of trailing zeroes in the binary representation of their right endpoints. (This is just the number of trailing zeroes in the least common ancestor of the original segment endpoints.) To do this we introduce a zero-counting function $z(i)$ whose value is the integer j such that $i = d2^j$ with d odd; we use this function to classify each segment $s \in \mathcal{L}$ according to the value of $z(r(s))$. The following lemma is crucial to the linearity of our algorithm:

Lemma 2.4. *For any x and j , let U_j be the set of segments in $\mathcal{L}$ that touch x and have $z(r(s)) = j$, that is, $U_j = \{s \in \mathcal{L} \mid l(s) \leq x \leq r(s) \text{ and } z(r(s)) = j\}$. Then the number of distinct right endpoints of segments in U_j is at most one: $|\{i \mid i = r(s), s \in U_j\}| \leq 1$.*

Proof: Suppose that two segments $s_1, s_2 \in U_j$ have different right endpoints, $r(s_1) < r(s_2)$. Then $I_x(s_2)$ includes at least two integers, $r(s_1)$ and $r(s_2)$, that are divisible by 2^j . But since $r(s_2) = lca(l(s_2), r(s_2))$, this contradicts the uniqueness of the lca function. ■

This lemma means that in the scanline algorithm given above, there are only $O(\log n)$ right endpoints in the current right envelope at any given time. Furthermore, the right endpoints of segments that touch the scanline position $\bar{x}$ are ordered in x by the function $z(r(s))$. That is, for two segments s_1 and s_2 that include $\bar{x}$, $r(s_1) = r(s_2)$ if and only if $z(r(s_1)) = z(r(s_2))$, and $r(s_1) < r(s_2)$ if and only if $z(r(s_1)) < z(r(s_2))$. This is because no segment $s \in \mathcal{L}$ has an interval that includes an integer i with $z(i) > z(r(s))$.

The algorithm presented in Figure 2.6 keeps exactly one segment for each endpoint in the right envelope. Therefore it can use a $(\lceil \log n \rceil + 1)$ -bit vector to keep track of the segments that are present. Define the bit vector v such that $v[i] = 1$

if and only if a segment s with $z(r(s)) = i$ is present in the current right envelope. When the scanline encounters a new segment $\hat{s}$ with $z(r(\hat{s})) = i$, the segment with which $\hat{s}$ must be compared is the segment s^* in E terminating immediately to the right of $\hat{s}$. This is the segment corresponding to the smallest $j \geq i$ such that $v[j] = 1$. The index j can be found in constant time given v and i using bit operations or lookup in a table of size $O(n)$. (The table space can be reduced to $O(n^\epsilon)$ for any fixed ϵ , but the computation time increases to $O(1/\epsilon)$.) For use in the subroutines of Figure 2.7, let us define the function $f(v, i)$ to be the least integer $j \geq i$ such that $v[j] = 1$.

As noted above, a doubly-linked list of segments in E is a good structure for all operations of the algorithm except point location. To get from the index j discussed above to the segment s^* itself, the algorithm keeps a $(\lceil \log n \rceil + 1)$ -entry table of pointers into the linked list. The entry $seg[i]$ points to the segment s in E with $z(r(s)) = i$, if any. These pointers are maintained by the *Insert* and *Delete* functions defined in Figure 2.7.

2.5 Extensions

In this section, we discuss several extensions of our algorithms.

(1) The dynamic problem

Either of the algorithms can be easily extended to the dynamic problem, where the set of segments is updated by insertions and deletions. We can execute each update (inserting or deleting a segment) in $O(n)$ time without increasing the other complexity, since the line arrangement can be updated in linear time by the algorithms in [CGL85] and [EOS83], and the triangulation needed by the first algorithm can be updated in linear time by the algorithm of El Gindy and Toussaint [ET84].

(2) Visibility of a set of disjoint polygons

The visibility problem from a point for a set of disjoint polygons with n edges can be solved in the same complexity by applying the above algorithms to the edges of those polygons. In the visibility problem for disjoint polygons, it is considered

```

function ShortestAsLongAs(s)
  begin
    i := z(r(s));
    j := f(v, i);
    /* This uses bit operations or table lookup. */
    return(seg[j]);
  end

procedure Insert(s, t)
  begin
    Manipulate pointers to insert s after t in E;
    i := z(r(s));
    v[i] := 1;  seg[i] := s;
  end

procedure Delete(s)
  begin
    Manipulate pointers to cut s out of E;
    i := z(r(s));
    v[i] := 0;  seg[i] :=  $\emptyset$ ;
  end

```

Figure 2.7. Subroutines used by the scanline algorithm. Because E never contains two segments s and t with $r(s) = r(t)$, the *Insert* procedure never overwrites data. When *Insert*(s, t) is called, $v[z(r(s))] = 0$.

that two points are mutually visible if the open segment connecting them lies in the exterior of all the polygons (when a polygonal region with holes is concerned, two points are visible if their connecting segment lies in the interior of the boundary polygon), so the algorithms must be modified slightly.

(3) The parallel view

We have mainly considered the view from a point, that is, a *perspective view*. However, there is another interesting type of view, called a *parallel view*, which consists of the portions of the objects that are visible from a given direction. Figure 2.8 gives an example of a parallel view. (Our algorithms reduce the problem of finding a perspective view to that of finding a parallel view.) The problem of finding the parallel view from an arbitrary direction can be solved in $O(n)$ time with $O(n^2)$ preprocessing and space by either of the algorithms given above. Because the parallel and perspective view problems are equivalent under a projective transformation, we can map one problem into the other before applying the algorithm. This improves the best previously known algorithm, due to Edelsbrunner, Overmars, and Wood, which uses $O(n)$ search time, $O(n^2 \log n)$ preprocessing time, and $O(n^2 \log n)$ space [EOW83].

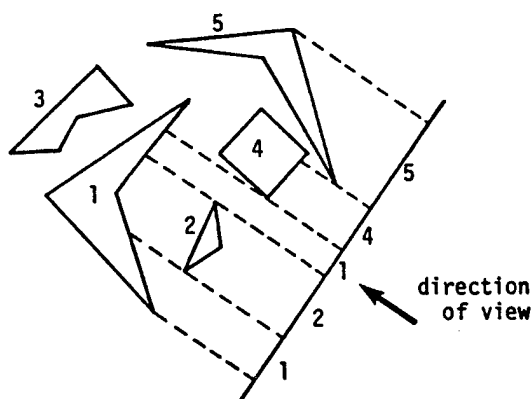


Figure 2.8. A parallel view

(4) A lower bound on the visibility-polygon search problem

There is no algorithm that uses $O(n)$ time and space (including preprocessing

steps) to find the visibility polygon from a point for h disjoint polygons with n edges. We can bound the time complexity of this problem from below by $\Omega(n + h \log h)$.

We first show that the problem of sorting h positive integers can be transformed in linear time into the problem of finding the visibility polygon from a point for a set of h polygons. Let $X = \{x_1, x_2, \dots, x_h\}$ be a set of h distinct positive integers. For each x_i , construct a triangle with vertices $p_{3i} = (2x_i + 2, 0)$, $p_{3i+1} = (2x_i + 1, x_i)$ and $p_{3i+2} = (2x_i + 1, -x_i)$ for $1 \leq i \leq h$. Consider the problem of finding the visibility polygon for these triangles from the point $q = (0, 0)$. Since the slope $y/(2y + 1)$ of the line connecting q and the point $(2y + 1, y)$ with $y > 0$ is strictly increasing with y , the points p_{3i+1} for $1 \leq i \leq h$ are visible from q . We can obtain the sorted list of the set X in $O(h)$ time once we have the visibility polygon from q . From the preceding argument and the fact that it takes $\Omega(n)$ time to read the polygon coordinates, we have the following theorem:

Theorem 2.2. *In computation models where $\Omega(h \log h)$ is a lower bound for sorting h integers, $\Omega(n + h \log h)$ is a lower bound on the time complexity of finding the visibility polygon from a point for h disjoint polygons with n edges altogether.*

Any algorithm that can find the visibility polygon for a set of n non-intersecting segments in $O(n)$ time requires $\Omega(n \log n)$ preprocessing time. Our algorithms find the visibility polygon in $O(n)$ time with $O(n^2)$ preprocessing time and space. It is a challenging open problem to devise an algorithm that finds the visibility polygon from an arbitrary point in $O(n)$ time with $o(n^2)$ preprocessing time.

In the case where all the polygons are convex, we can obtain an algorithm that meets the lower bound. Consider the problem of finding the visibility polygon from a point for a set of h (not necessarily convex) disjoint polygons with n edges. We first compute the visible portion of the boundary of each polygon from the point using linear-time algorithms [EA81], [Lee83]. The result is a sequence of edges from each polygon. Then we decompose each sequence into maximally continuous portions, where two consecutive edges on the visible portion are continuous if they were originally consecutive. Let M be the total number of such maximally continuous portions. (Note that no two such portions intersect.) We can apply a circular plane

sweep method to find the visible portions by sorting $2M$ endpoints in the order of their polar angles with q at the origin.

The algorithm outlined above runs in $O(n + M \log h)$ time in the worst case. Since M may be $\Omega(n)$, the worst-case time complexity is $\Omega(n \log h)$. On the other hand, if every polygon, like a convex polygon, has a constant number of maximally continuous portions, then the algorithm requires only $O(n + h \log h)$ time and is optimal within a constant factor.

It is not known whether the above approach can be extended to give an algorithm that finds the visibility polygon in $O(n)$ time with $O(n + h \log h)$ or even $O(n + h^2)$ preprocessing.

2.6 The visibility graph and the shortest-path problem

The visibility graph of n non-intersecting segments with arbitrary slopes is a graph whose vertices are endpoints of those segments and whose edges are the straight line segments joining vertices that are visible from each other. This graph can be constructed in $O(n^2)$ time and space by solving the visibility problem from each vertex for the given segments. This improves the $O(n^2 \log n)$ time bound due to Lee [Lee78] and matches the $O(n^2)$ result of Welzl [Wel85]. (Note that in the worst case, the visibility graph has $\Omega(n^2)$ edges.) As an application of this result, the shortest path between two points in the plane with polygonal obstacles having n edges can be computed in $O(n^2)$ time, improved from $O(n^2 \log n)$, since the problem can be solved in $O(n^2)$ time by Dijkstra's algorithm, provided that the visibility graph is available.

It is instructive to compare Welzl's $O(n^2)$ visibility graph construction with the algorithms presented here. He transforms endpoints into lines via the dual construction and computes their arrangement in $O(n^2)$ time, just as we do. For each endpoint p_i , he uses the arrangement to sort the other endpoints by the slopes of the lines they determine with p_i . Our algorithms use this information to find the

visibility edges from each p_i independently of the other points. In particular, our second algorithm sweeps the other points with a scanning ray from each segment endpoint.

Welzl's algorithm is a global one; it scans the directions from 0 to π once, maintaining parallel scanning rays from all segment endpoints. This is equivalent to sorting the $\binom{N}{2}$ pairs of endpoints by the slopes they determine and running through the list once. Welzl's method keeps track of the first segment encountered by the ray from each point. When a sweeping ray (with origin p) hits a point q , it is easy to update p 's visible segment. Let p 's current visible segment be s and let t be the segment with q as endpoint. If t starts at q and obscures s , then t becomes p 's new visible segment. If $s = t$, then s ends at q ; p 's new visible segment is q 's current one.

As we have described it, this algorithm requires sorting the $\binom{N}{2}$ endpoint pairs by slope. However, Welzl observes that the list of pairs does not need to be completely sorted; the method still succeeds as long as every two pairs that contain only three distinct points are correctly ordered. The slope-sorted list of lines through each p_i obtained in the first step correctly orders all endpoint pairs that include p_i . In $O(n^2)$ time Welzl produces a satisfactory order of all pairs by applying topological sort to the N ordered lists.

Welzl's algorithm suggests an open problem. He shows that the visibility graph application does not require sorting the $\binom{N}{2}$ directions determined by N points. Nonetheless, that problem is interesting and unsolved: can such a sorted list be produced in $o(n^2 \log n)$ time?

Our algorithms require $\Theta(n^2)$ time to compute the visibility graph, but the structure they construct may be as small as $O(n)$, as in Figure 2.9. In such situations it would be desirable to compute the visibility graph in time proportional to its size, or at least in $o(n^2)$. It is not known whether this is possible in general; however, Chapter 5 gives an almost-optimal algorithm for the special case in which the segments form a simple polygon.

For the case where the obstacles are h convex polygons with a total of n edges, several shortest-path algorithms that make use of the convexity of the polygons

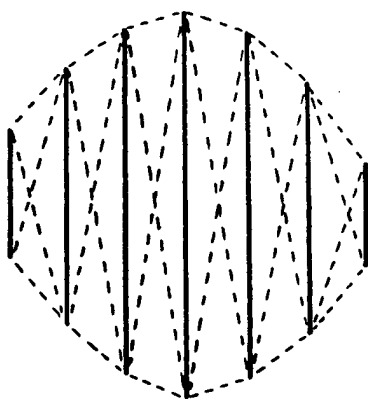


Figure 2.9. A visibility graph with $O(n)$ edges

are known. Rohnert [Roh85] showed that in this case the shortest-path problem between two arbitrary query points can be solved in $O(h^2 + n \log n)$ time with $O(n + h^2 \log n)$ preprocessing time and $O(n + h^2)$ space (of course, h can be $\Omega(n)$, and thus the time complexity is $\Omega(n^2 \log n)$ in the worst case). His algorithm is based on the fact that, when the polygonal obstacles are convex, shortest paths use edges of the polygons or supporting lines of pairs of polygons. Based on this fact, we can devise a shortest-path algorithm that takes $O(hn \log n)$ time and $O(n)$ space (a similar result appears independently in a paper by Papadimitriou [Pap85]). It is not known whether the shortest-path problem in this case can be solved in $O(n + h^2)$ time and space.

By using our algorithms, we can also find the shortest path between two *segments* in the plane with polygonal obstacles in $O(n^2)$ time. (Here, by a path between two segments, we mean a path between a point on one segment and a point on the other.) The shortest path between the source and target segments may start or end with edges connecting interior points of the segments to obstacle vertices. If we augment the visibility graph by including an edge corresponding to the shortest obstacle-avoiding segment from each obstacle vertex to the source and target segments, then we can proceed as above. These additional edges can easily be computed in $O(n^2)$ total time as part of the visibility computation from each vertex.

2.7 Concluding remarks

In this chapter we have given two algorithms to compute the visibility polygon from a point q for a set of n segments allowed to intersect only at their endpoints. Both algorithms use $O(n^2)$ preprocessing and storage to construct the arrangement of lines corresponding to the segment endpoints via a dual transformation. With this arrangement, the algorithms require linear time to find the polar order of the segment endpoints around the query point q . The first algorithm uses $O(n \log n)$ preprocessing time to triangulate the set of segments S ; the triangulation allows it to compute an order compatible with the visibility relation $\prec_q$ in linear time. It then uses the Gabow-Tarjan linear set-union algorithm to find the visibility polygon. The second algorithm splits each segment in S_q at the polar angle indexed by the least common ancestor of the indices of its endpoints. The regularized segments thus produced are processed by a scanline algorithm to find the visibility polygon.

Our algorithms can be used to construct the visibility graph of disjoint polygons with n edges in $O(n^2)$ time and space. This leads to an $O(n^2)$ algorithm for finding the shortest obstacle-avoiding path between two points in the plane. There are several possible ways to improve and extend this shortest path algorithm. One extension is presented in the following chapter, which uses the same tools as this chapter—the visibility graph and Dijkstra’s algorithm—to find shortest paths for a non-rotating convex object. By taking an alternative approach, Reif and Storer [RS85] are able to solve the shortest path problem for a point in $O(n \log n + nk)$ time, where k is the number of polygons formed by the obstacle segments. Given a fixed source, their algorithm builds a structure that can be used to find the length of the shortest path from the source to an arbitrary query point in $O(\log n)$ time. The path itself can be found in additional time proportional to the number of turns along it. Chapters 4 and 6 improve Reif and Storer’s bounds when the obstacle segments form a single simple polygon; Chapter 6 shows how to answer queries when both endpoints of the path are part of the query.

Chapter 3

Shortest Paths for a Non-Rotating Convex Body

Pooh Bear stretched out a paw, and Rabbit pulled and pulled and pulled. . . .

"Ow!" cried Pooh. "You're hurting!"

"The fact is," said Rabbit, "you're stuck."

"It all comes," said Pooh crossly, "of not having front doors big enough."

— A. A. Milne, *Winnie-the-Pooh* (1926)

In the last chapter we saw how to find shortest paths for a point moving among polygonal obstacles; the algorithm uses $O(n^2)$ time, where n is the number of polygon vertices. The problem of finding shortest paths for a moving body of finite size is more difficult. In the special case when the body is a non-rotating convex polygon of fixed complexity, a simple extension of the point-motion algorithms runs

in $O(n^2)$ time. Baker and Chew propose an $O(n^2 \log n)$ algorithm to find shortest paths for a disk moving among polygonal obstacles [Bak85,Che85]. An extension of Reif and Storer's algorithm (discussed in Section 2.7) gives an $O(n^2)$ solution to the same problem [RS85]. Their method exploits the geometry of the obstacle space to run faster than $O(n^2)$ under some conditions.

In this chapter we solve a generalization of the disk motion problem. We show how to find shortest paths for a non-rotating convex body. The obstacles we consider are disjoint simple polygons, as in the work cited above; our method is novel because the moving body is not constrained to be polygonal or round, but only convex. The arbitrary shape of the convex body is not without its costs, but we defer discussion of these costs until Section 3.4.

Our approach uses two ideas from the literature to simplify the problem of finding shortest paths. The first simplification reduces the problem of moving a non-rotating convex object among polygons to that of moving a point (the object's center) among "fattened" versions of the obstacles. This idea is due to Lozano-Perez and Wesley [LW79]. The second simplification, due to Baker and Chew [Bak85,Che85], reduces the problem of finding shortest paths for a point among fattened obstacles to that of finding a shortest path in a graph. Dijkstra's shortest path algorithm for graphs [AHU74, pages 207–209] can be used to solve the reduced problem.

The shortest path algorithm we present builds the graph of Baker and Chew, simplifies it, and then invokes Dijkstra's algorithm. Each step takes $O(n^2)$ time and space. The visibility graph of the polygons, obtained using the methods of the preceding chapter, forms the basis for constructing Baker and Chew's graph. If the graph so constructed has $\Omega(n^2)$ nodes, then running Dijkstra's algorithm on it will take $\Omega(n^2 \log n)$ time, causing a bottleneck. To circumvent this potential problem, the algorithm reduces the graph to an equivalent one with only $O(n)$ nodes before running Dijkstra's algorithm. Using the Fibonacci heaps of Fredman and Tarjan [FT84], Dijkstra's algorithm takes only $O(n^2)$ time when applied to the reduced graph.

3.1 Definitions

This section gives the special notations used in this chapter. It also explains the idea of “fattening” obstacles mentioned in the introduction.

We represent individual points by lowercase letters and sets of points by uppercase letters. The segment between points p and q is $\overline{pq}$, and the distance from p to q is $|pq|$. This distance is usually the straight-line distance between the points, though we sometimes use $|pq|$ to refer to the distance from p to q along the boundary of a convex region on which both points lie.

We refer to the moving convex body as the *robot* and represent it by the letter A . The robot must avoid obstacles as it moves. These obstacles are disjoint simple polygons with a total of n vertices; the robot may not touch the interior of any polygon. (The polygons must be disjoint to satisfy the preconditions of the visibility graph algorithms of Chapter 2.) The set of all polygon points, both vertices and points on segments, is S .

The robot A has a center, which is just the coordinate origin in its frame of reference. In that frame, B is the point-wise reflection of A through its center. (Note that the center of A need not lie within its boundary.) We denote the set of points covered by B when its center is placed at a point q by B^q (pronounced “ B at q ”).

To plan the motion of the robot among polygons, we solve the equivalent problem of moving the robot’s center among fattened versions of the polygons. To distinguish between the obstacles of the original problem and their fattened versions, we refer to the polygons that the robot avoids as *obstacles* and to the fattened polygons that the robot’s center avoids as *barriers*. Points on the boundaries of barriers are especially important to our algorithm; we will refer to these as *boundary points*.

We can draw the barriers by using B as a paintbrush, placing its center at every point of the polygon boundaries and interiors; the painted areas are the barriers that the robot’s center must avoid. The painted region is the Minkowski sum (or vector sum) of B with the polygons and their interiors.¹ The boundary of the painted area

¹The Minkowski sum of two regions X and Y consists of all points expressible as a vector

is closely related to the *convolution* of the boundary of B with the polygons in S , as defined by Guibas, Ramshaw, and Stolfi [GRS83].

Any point on the boundary of a barrier must be on the boundary of B^q for at least one polygon point q , and it may not lie inside $B^{q'}$ for any other q' in S . Given a barrier boundary point p , we refer to a polygon point q such that p is on the boundary of B^q as a *generator* of p and denote the set of generators of p by $g(p)$. Unless the boundary of B contains a segment parallel to one of the polygon segments, the number of generators $|g(p)|$ is finite for any point p on the boundary of a barrier. In fact, of those boundary points p with finitely many generators, only a finite number have $|g(p)| > 1$. (We sometimes give the term *generator* a slightly broader meaning; we say that q *generates* the barrier B^q .)

We can classify each barrier boundary point p by its generator set $g(p)$. We use the term *arc* to denote a maximal connected set of boundary points generated by a single polygon vertex. The barrier boundaries are composed of arcs, of straight segments whose generators all lie on a single polygon segment, and of intersection points of these elements.² Each arc copies a portion of the border of B . If an obstacle polygon is convex, then, in the fattened polygon, each arc bridges the difference in slopes between the segments that precede and follow it. (See Figure 3.1.) When two arcs or segments intersect other than by abutment, the point of intersection has multiple generators. (An arc and its adjacent segment intersect at their point of abutment, but that point has only a single polygon vertex as its generator.)

3.2 The path graph

Since combinatorial problems are often easier to solve than geometric ones, we reduce the shortest path problem to a combinatorial problem by introducing the

sum $x + y$, where $x \in X$ and $y \in Y$. The sum is symmetric in its arguments—it is the set $\{x + y \mid x \in X, y \in Y\}$ —but it can also be viewed asymmetrically as $\bigcup_{x \in X} Y^x = \bigcup_{y \in Y} X^y$.

²These definitions can break down if the boundary of B contains a segment parallel to one of the polygon segments. In this case, boundary arcs and segments may not be disjoint. To remedy the problem, we use the technique of ϵ -perturbation: we define the boundary arcs and segments as if the polygon segment were not parallel to the flat spot on B , but rotated clockwise by an infinitesimal amount ϵ . This results in a consistent definition of disjoint arcs and boundary segments.

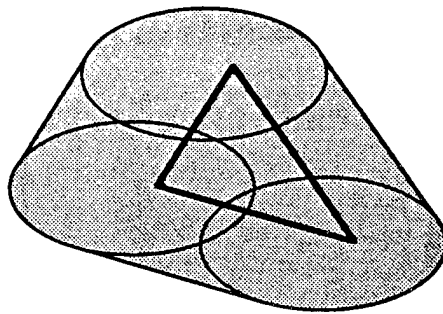


Figure 3.1. When a convex obstacle polygon is fattened by the inverted robot B , the outer boundary of the resulting barrier has arcs and segments. The tangent at each of an arc's endpoints matches the slope of the adjacent segment. The tangent's direction varies monotonically between these two extremes along the arc.

path graph. Our path graph is a combinatorial graph structure obtained from the positions of the barriers and is an extension of the one used by Baker and Chew. Each path graph edge corresponds to a path among the barriers. We show that except for its initial and final segments, any barrier-avoiding shortest path is a subset of the path graph edges.

The path graph of a set of barriers is closely related to the visibility graph of a set of polygons, which we discussed in Chapter 2. The visibility graph records pairs of mutually visible vertices; similarly, the path graph records pairs of barriers connected by common tangents that avoid all other barriers. Because shortest paths for the robot's center follow barrier boundaries as well as common tangents, the path graph has two kinds of edges: portions of the barrier boundaries and tangent segments.

Definition 3.1. *The nodes and edges of the path graph are defined as follows:*

- (1) *Every maximal straight boundary segment generated by a single polygon segment is an edge of the path graph, and its endpoints are path graph nodes. This implies that when a segment generated by a polygon segment intersects arcs or other boundary segments, each intersection is a path graph node. Intersections of boundary arcs are also path graph nodes.*

- (2) *If a line tangent to two arcs does not intersect any barrier between its points of tangency, the two tangent points are path graph nodes, and the segment connecting them is a path graph edge. If the tangent touches an arc at more than one point (the arc contains a straight segment), we use the tangent point that minimizes the length of the tangent segment. These edges are called tangent edges.*
- (3) *Minimal arc sections connecting nodes defined in (1) and (2) are path graph edges (no two such edges overlap). These edges are convex curves; some may be straight line segments, but only if the boundary of B has flat spots.*

Path graph edges defined in (1) and (3) are called *boundary edges*; their union includes all the boundary arcs and straight segments.

Recall that the path graph of a set of barriers is the analogue of the visibility graph of a set of polygons. Since the visibility graph has $O(n^2)$ edges, it is reasonable to assume that the path graph also has $O(n^2)$ nodes and edges. The following lemma shows that this assumption is true.

Lemma 3.1. *The path graph defined above has $O(n^2)$ nodes and edges.*

Proof: We begin by bounding the number of path graph tangent edges. Consider centering a copy of B at each polygon vertex. These n (possibly overlapping) convex regions have at most $4\binom{n}{2}$ common tangents, since two translated copies of B have at most four common tangents. Because tangent edges are common tangents of boundary arcs, and arcs are generated by polygon vertices, the path graph tangent edges are a subset of the $O(n^2)$ tangents. Each tangent edge contributes at most two endpoint nodes to the path graph.

Kedem et al. [KLPS86] show that there are only $O(n)$ nodes formed by intersections of segments and arcs on the boundary of the barriers. (As in Figure 3.2, there can be $O(n^2)$ intersections, but only $O(n)$ of them are on the boundary.) These are the nodes of type (1).

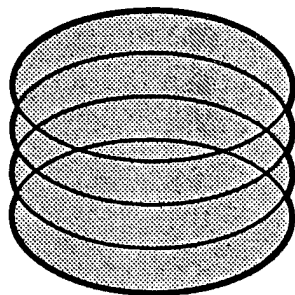


Figure 3.2. Although n barriers can intersect in $O(n^2)$ points, only a linear number of the intersections are on the boundary of the barriers

Every path graph node has exactly two incident boundary edges. Since there are $O(n^2)$ nodes, there are $O(n^2)$ boundary edges. This fact, in combination with the bounds on nodes and edges already given, proves the lemma. ■

An instance of the shortest path problem specifies the initial and final positions α and ω of A 's center, chosen so that A^α and A^ω avoid the interiors of the polygons. Shortest paths among the fattened polygons begin and end with edges from α and ω , but intermediate edges come from the path graph.

In order to use Dijkstra's algorithm to find the shortest path, we need a graph whose edges contain all shortest paths from α to ω . We produce such a graph, which we call the *augmented path graph*, G_a , by adding nodes and edges to the path graph. If the segment $\overline{\alpha\omega}$ does not intersect any barrier, it is the shortest path and belongs in G_a . If $\overline{\alpha\omega}$ is blocked, we add to the path graph all unobstructed segments passing through α or ω and tangent to some barrier. The endpoints of the added segments augment the set of path graph nodes. Some boundary edges are split in two by these additional nodes. Because there are at most $4n$ tangents from α and ω to barriers, the added nodes and edges leave the augmented path graph still of size $O(n^2)$. The following lemma shows that G_a is useful for motion planning.

Lemma 3.2. *Every shortest path from α to ω that avoids the barriers follows edges*

of the augmented path graph, G_a .

Proof: Any shortest path is composed of subpaths that alternately follow barrier boundaries and move along straight lines between barriers. If the endpoints of the segments that connect barriers are augmented path graph nodes, then the boundary subpaths are made up of boundary edges. Thus it suffices to show that every non-boundary segment on the shortest path is an edge of the augmented path graph.

Suppose that a segment $\overline{ab}$ lies on a shortest path from α to ω , touches barriers only at its endpoints, and is not an edge of the augmented path graph. At least one of $\overline{ab}$'s endpoints, say b , lies on a barrier boundary and is not α , ω , or a point of tangency of $\overline{ab}$ with the barrier. Because b is not α or ω , the path continues beyond b . Some point d on the continuation must be visible from a point c on $\overline{ab}$, since $\overline{ab}$ is not tangent at b . But this means that the supposed shortest path could be shortened by replacing the subpath from c to d via b by the segment $\overline{cd}$, which contradicts the assumption that $\overline{ab}$ lies on a shortest path. (See Figure 3.3.) ■

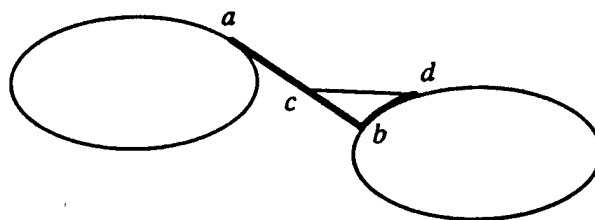


Figure 3.3. Barrier-connecting segments on a shortest path must be tangents. The dark path through a , b , and d cannot be a subpath of any shortest path. Because b is not a tangent point of $\overline{ab}$, the point d on the continuation of the path is visible from c . Taking the shortcut $\overline{cd}$ gives a path shorter than the dark one.

3.3 Tangent-visibility

This section deepens the connection between the path graph of a set of barriers and the visibility graph of the polygons that generate the barriers. It shows that the path graph tangent edges from an arc are closely related to the visibility graph edges from the vertex that generates the arc. The next section exploits this connection to construct the path graph quickly.

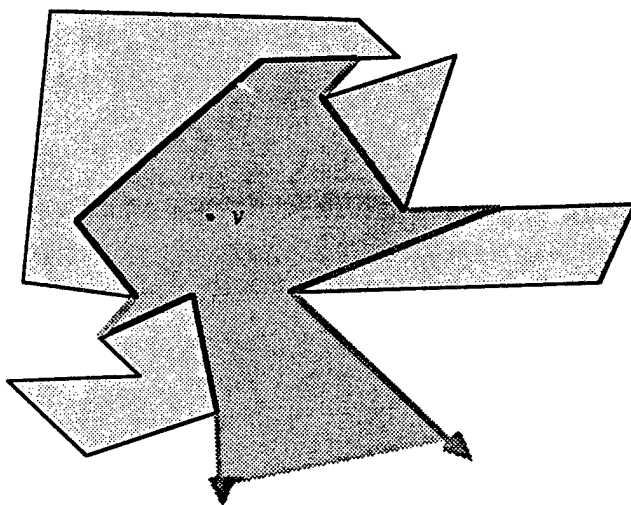


Figure 3.4. The visibility polygon $Vis(S, q)$ of query point q is the region visible from q in the presence of the polygonal obstacles in S . The circular sequence of points of S visible from q is $VP(q)$.

If we are given a set of simple polygons with n vertices and a query point q outside the polygons, the (uncountably many) polygon points visible from q , listed in polar order, form a cyclic sequence that we denote by $VP(q)$. The points of $VP(q)$ lie on the boundary of the visibility polygon $Vis(S, q)$. (See Figure 3.4.) We can define a similar concept for the fattened polygons by sweeping them with a ray tangent to B^q whose endpoint moves along the boundary of B^q . If the tangent point of the ray is not inside a barrier that overlaps B^q , we say that the first boundary point encountered by the ray is *tangent-visible* from B^q . See Figure 3.5 for an example. Each query point q has two sequences of tangent-visible points, since each boundary

point of B has both a clockwise and a counterclockwise tangent. Both sequences may contain points of tangency that are path graph nodes; since the two sequences are similar, we discuss only the sequence derived from the clockwise tangent. Let us denote the sequence of barrier boundary points swept by the clockwise tangent to B^q by $TV(q)$. The generators of the points in $TV(q)$ form a sequence we call $g(TV(q))$. (If a point p in $TV(q)$ has multiple generators, any one of them may appear in the generator sequence.)

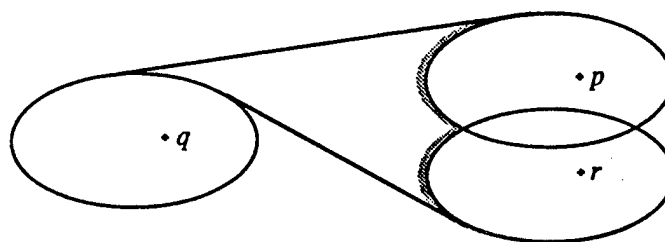


Figure 3.5. Tangent-visibility. The darkened section of the boundary of B^p and B^r is tangent-visible from B^q .

To help characterize the sequence of generators $g(TV(q))$, we introduce some special notation. As the tangent to B^q sweeps around, it points in each direction between 0 and 2π exactly once. Let p be a point and T a set of points. The point p generates the barrier B^p , and T generates barriers that are the Minkowski sum of T and B . We use the notation $v_T(q, p)$ to refer to the set of directions in $[0, 2\pi)$ for which the tangent from B^q hits B^p strictly before hitting any other barrier B^r for $r \neq p$ and r in T . (It doesn't matter whether p is in T or not.) In this notation $v_\emptyset(q, p)$ is an interval (allowing wraparound across 2π) of measure less than π . The set $v_T(q, p)$ shrinks as T grows; that is, $v_{T \cup \{r\}}(q, p) \subseteq v_T(q, p)$ for any r . The set $v_S(q, p)$ can be expressed as $\bigcap_{r \in S} v_{\{r\}}(q, p)$. For convenience, we let $v_r(q, p)$ stand for $v_{\{r\}}(q, p)$. (See Figure 3.6 for an example of the notation.) The following lemmas apply this notation to show that the generators of the tangent-visible sequence $TV(q)$ are a subsequence of $VP(q)$.

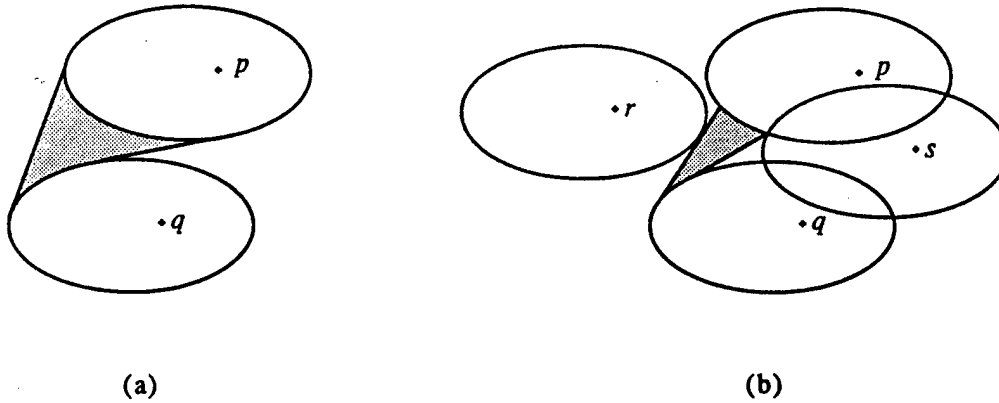


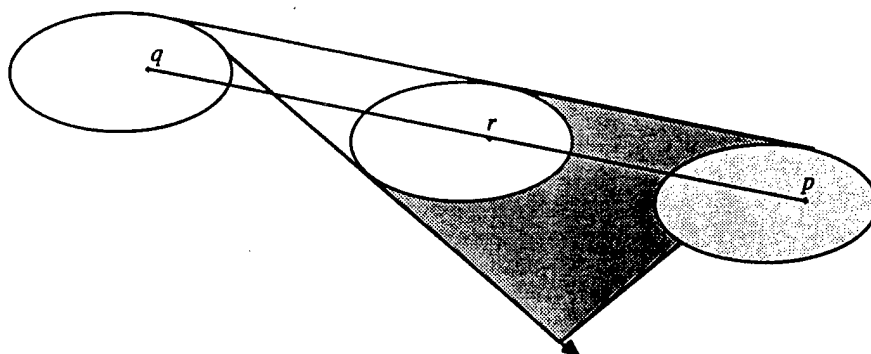
Figure 3.6. Tangent-visibility notation. When no other barriers are present, B^p is tangent-visible from B^q in the shaded sector shown in (a). The directions of the rays that bound the sector define the interval $v_0(q, p)$. When barriers B^r and B^s are added, the sector of tangent-visibility is reduced to the shaded region shown in (b). This sector's boundary rays define $v_{\{r,s\}}(q, p)$.

Lemma 3.3. *If p' is a point of the tangent-visible sequence $TV(q)$, all of its generators are in $VP(q)$.*

Proof: Let p be a generator of p' , that is, $p \in g(p')$. If $p \notin VP(q)$, a polygon point r lies on the segment connecting p and q . The object B^r hides every point of B^p from the sweeping tangent, as shown in Figure 3.7. Because $v_r(q, p) = \emptyset$, its subset $v_s(q, p)$ is also empty, and p' cannot be in $TV(q)$. ■

Lemma 3.4. *For any set of barrier generators T , $v_T(q, p)$ is an interval.*

Proof: Since the intersection of intervals of length less than π results in an interval, we need only show that $v_r(q, p)$ is an interval for all r . If $v_r(q, p)$ is not an interval for some r , then let $a < b < c$ be directions in the interval $v_0(q, p)$ such that a and c are in $v_r(q, p)$ and b is not. (We have assumed without loss of generality that $v_0(q, p)$ does not

Figure 3.7. B^r hides B^p from B^q

include 2π .) In Figure 3.8, the segments $\overline{u_a v_a}$, $\overline{u_b v_b}$, and $\overline{u_c v_c}$ have directions a , b , and c . Because directions a and c are part of $v_r(q, p)$, the barrier B^r cannot intersect the segments $\overline{u_a v_a}$ and $\overline{u_c v_c}$, even at their endpoints.

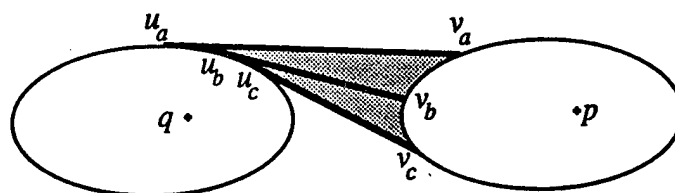


Figure 3.8. Proof that $v_r(q, p)$ is an interval. B^r cannot intersect $\overline{u_b v_b}$ (block tangent-visibility in direction b) without also intersecting either $\overline{u_a v_a}$ or $\overline{u_c v_c}$.

We show that B^r cannot intersect $\overline{u_b v_b}$, and hence b is in $v_r(q, p)$. Consider all possible placements of B^r that intersect the shaded region of Figure 3.8. Note that every placement that abuts the region without overlapping its interior touches $\overline{u_a v_a}$ or $\overline{u_c v_c}$. Since the shaded region is narrower than the parallelogram defined by the outer common tangents to B^q and B^p , any placement of B^r that overlaps the shaded region's interior must also intersect $\overline{u_a v_a}$ or $\overline{u_c v_c}$. Because $\overline{u_b v_b}$ is contained in

the shaded region, B^r cannot intersect it. ■

Lemma 3.5. *If points p , r , and s belong to the sequence $g(TV(q))$ and appear in clockwise order around q , then the intervals $v_S(q, p)$, $v_S(q, r)$, and $v_S(q, s)$ are disjoint and also appear in clockwise order.*

Proof: The intervals $v_S(q, p)$, $v_S(q, r)$, and $v_S(q, s)$ are disjoint because p , r , and s all belong to S .

The ray from q to p is parallel to the outer common tangents from B^q to B^p . Let p_t (t for tangent) be the direction of this ray, and let p_v (v for visible) be any direction in $v_S(q, p)$. We define r_t , r_v , s_t , and s_v similarly. These six directions occur in some clockwise order, though there may be degeneracies. If $p_t = p_v$, we order p_t and p_v such that p_v immediately follows p_t in clockwise order. If $p_v = r_t$, we order p_v and r_t such that r_t immediately follows p_v in clockwise order. (The first condition applies to r and s as well, and the second applies to all pairs taken from $\{p, r, s\}$.) See Figure 3.9 for an example of this notation. We want to show that if p_t , r_t , and s_t occur in clockwise order, so do p_v , r_v , and s_v .

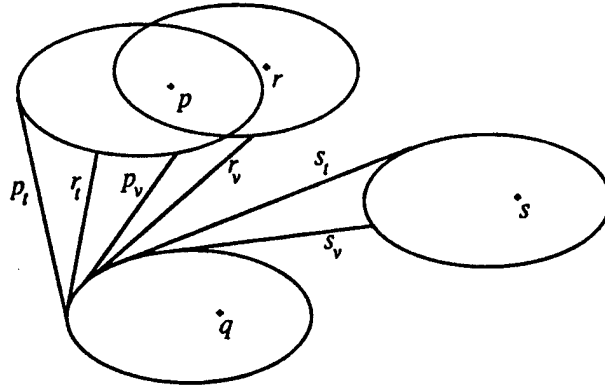


Figure 3.9. The tangent directions p_t , r_t , and s_t and the visibility directions p_v , r_v , and s_v . The cyclic permutation associated with these directions is $(p_t r_t p_v r_v s_t s_v)$.

The following simple condition on the clockwise sequence of directions is stated using p and r , but it applies to all pairs drawn from $\{p, r, s\}$: *At most one of r_t and r_v occurs after p_t and before p_v .* If $(p_t r_t r_v p_v)$ is a subsequence of the cyclic sequence of angles, then $v_r(q, p)$ is not an interval, since it includes both p_t and p_v , but not r_v , which lies between them. This contradicts Lemma 3.4. If $(p_t r_v r_t p_v)$ is a subsequence of the cyclic sequence of angles, then at least one of $v_\theta(p)$ and $v_\theta(r)$ is larger than π , also a contradiction.

Of the possible angle sequences, the only ones that satisfy the condition also have p_v , r_v and s_v in order. There are 120 cyclic permutations on $\{p_t, p_v, r_t, r_v, s_t, s_v\}$. Of these, 60 have p_t , r_t , and s_t in clockwise order. Of the 60, only eleven meet the condition given above; in all of those, p_v , r_v , and s_v appear in clockwise order. If we class together permutations that are equivalent under renaming of p , r , and s , there are really only five different permutations that satisfy the condition. They are given in Figure 3.10.³ This proves that the tangent-visible points generated by p , r , and s occur in the same clockwise order as p , r , and s themselves.

■

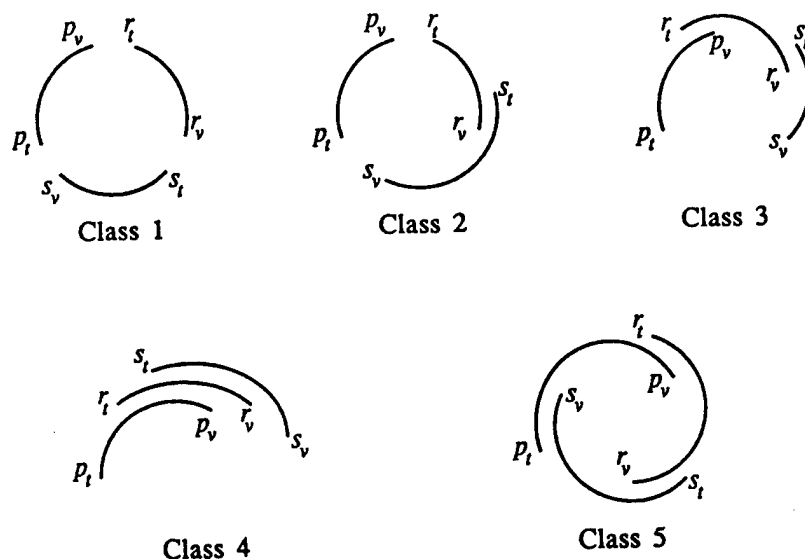
Lemmas 3.3 and 3.5, taken together, have the following consequence:

Theorem 3.1. *The sequence $g(TV(q))$ is a subsequence of $VP(q)$.*

3.4 Construction of the augmented path graph

Since any shortest path from α to ω follows edges of the augmented path graph, our first step in finding a shortest path is to construct that graph. The graph has two kinds of edges, boundary edges and tangent edges, which we find separately. (We need find only the edges, since the nodes of the graph are given by edge intersections.)

³By an elegant geometric argument, Jim Saxe has shown that there are no barrier configurations that give rise to permutations of Class 5. (Private communication, 1986.)



Example	Equivalent permutations	
Class 1: $(p_t p_v r_t r_v s_t s_v)$		
Class 2: $(p_t p_v r_t s_t r_v s_v)$	$(p_t s_v p_v r_t r_v s_t)$	$(p_t r_t p_v r_v s_t s_v)$
Class 3: $(p_t r_t p_v s_t r_v s_v)$	$(p_t s_v p_v r_t s_t r_v)$	$(p_t s_v r_t p_v r_v s_t)$
Class 4: $(p_t r_t s_t p_v r_v s_v)$	$(p_t r_v s_v p_v r_t s_t)$	$(p_t r_t s_v p_v r_v s_t)$
Class 5: $(p_t s_v r_t p_v s_t r_v)$		

Figure 3.10. Five essentially different cyclic permutations satisfy the condition of Lemma 3.5. The figure shows one example from each class, and the table lists its equivalent permutations, normalized to start with p_t .

To find the boundary edges, we construct the boundary of the barriers, that is, the border of the region where the robot's center may be placed. We compute this boundary using the divide-and-conquer method of Kedem et al. [KLPS86], which runs in time $O(\tau n \log^2 n)$; here τ is a constant that depends on B and which will be defined later.

To find the tangent edges, we construct the two tangent-visible sequences from each fattened polygon vertex B^q . The sequence $TV(q)$ and its counterclockwise counterpart give all path graph edges tangent to B^q .

The results of the preceding section provide a way to produce the tangent-visible sequence $TV(q)$ from $VP(q)$, which we can find using the methods of Chapter 2. After $O(n^2)$ preprocessing, those algorithms take $O(n)$ time to compute the visibility polygon for any particular vertex. (The algorithms require that the obstacle segments be disjoint, as does the boundary construction of Kedem et al. mentioned above.)

To bound the time it takes to construct $TV(q)$, we introduce some conditions on B . These conditions are not restrictions on the shape of B , but rather characterizations of it. Each of the five operations listed below is used in some phase of our shortest path algorithm; to bound the algorithm's performance, we need to bound the complexities of the basic operations. We therefore assume that each of the following operations can be performed in time τ , for some τ dependent on B :

To compute $TV(q)$ from $VP(q)$, we must be able to

- (1) Find intersection points of two translated copies of B ,
- (2) Find the intersection of a line with B , and
- (3) Compute the inner and outer common tangents of two translated copies of B .

To compute the boundary points visible from the source and destination points α and ω , we must be able to

- (4) Find the tangents to B through a point.

To compute the lengths of path graph edges quickly, we need to

- (5) Compute the perimeter distance between two arbitrary points on an arc of B .

(Note that preprocessing of B may be used to speed up distance computations.)

Because of Theorem 3.1, we are able to compute $TV(q)$ using an algorithm similar in spirit to the Graham scan convex hull algorithm [Gra72]. Roughly speaking, we replace each vertex and segment in $VP(q)$ by the barrier it generates, then find the tangent-visible points on the union of these barriers. Each visibility polygon vertex p contributes B^p to the union, and each segment $\overline{pr}$ in $VP(q)$ contributes a tube $\bigcup_{s \in \overline{pr}} B^s$, which we represent compactly by $B^{\overline{pr}}$. (See Figure 3.11.)

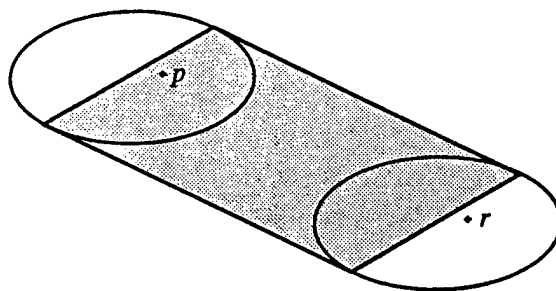


Figure 3.11. The tube $B^{\overline{pr}}$ and its internal parallelogram

It is convenient to replace the tube $B^{\overline{pr}}$ by the parallelogram defined by the endpoints of the outer common tangents of B^p and B^r . This choice removes a possible degeneracy from the construction. Each arc is generated just once, by a vertex, rather than three times: once by the vertex and twice by the vertex's incident segments. Computing the parallelogram requires operation (3) above.

Each barrier generated by an element i in $VP(q)$, either vertex or segment, has a range of angles in which it is tangent-visible from B^q if no other barriers are present. This range is $v_\theta(q, i)$ if i is a vertex; for convenience, we use the notation $v_\theta(q, i)$ to represent the range when i is a segment, too.

If B^i intersects B^q , the intersection blocks all tangent-visibilitys in the angular range it covers. The range in which the barrier generated by i blocks all visibilitys is called its *blocking interval* $b(i)$. If B^i and B^q do not intersect, $b(i)$ is empty. For clockwise-going tangents to B^q , the intersection of B^i and B^q is clockwise of the tangent-visible part of B^i . The blocking interval $b(i)$ immediately follows $v_\theta(q, i)$ in clockwise order.

When more than one barrier is present, each barrier B^i is visible in some subinterval of $v_\theta(q, p)$. An interval in which a barrier B^i is tangent-visible may be followed in clockwise order by a blocking interval. If two blocking intervals $b(i)$ and $b(j)$ overlap, they can be merged into a single joint blocking interval. (See Figure 3.12.) We associate this joint interval with the tangent-visible barrier B^i that precedes it in clockwise order; note that the blocking interval $b(i)$ contributes to the joint blocking interval.

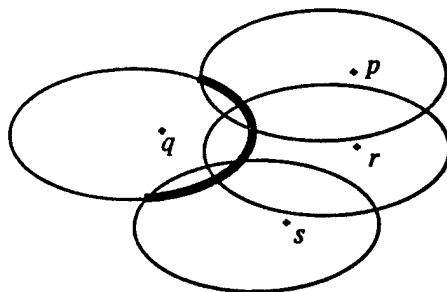


Figure 3.12. The blocking intervals of B^p , B^r and B^s are merged into a single joint blocking interval and associated with B^p .

Theorem 3.1 states that the generators of the tangent-visible sequence $TV(q)$ are a subsequence of $VP(q)$; we apply the theorem to construct $TV(q)$ efficiently. Let $L = (i_1, \dots, i_k)$ be a clockwise subsequence of $VP(q)$. Each element i in L has associated with it a blocking interval $b(i)$ that may be a joint interval (larger than the intersection of B^i with B^q). Suppose that when the barriers and blocking intervals given by the elements of L are present, every such barrier is tangent-visible from B^q . Let j be a successor of i_k in $VP(q)$; that is, $(i_1, \dots, i_k, j)$ is a subsequence of $VP(q)$. Theorem 3.1 implies that if B^j is hidden from B^q by barriers and blocking

intervals given by elements in L , it is hidden by the union of B^{i_1} , B^{i_k} , $b(i_1)$, and $b(i_k)$. On the other hand, if B^j hides some of the barriers generated by elements in L , it is visible itself. Furthermore, B^j and $b(j)$ hide a contiguous initial and final subsequence of L . That is, if they hide a part of B^x for some x in L , they also hide all the predecessors of x in L or all the successors.

The algorithm of Figure 3.13 exploits these observations to compute $TV(q)$. It maintains a double-ended linked list L of elements of $VP(q)$. Each element has associated with it an interval of current visibility $v(i)$ and a blocking interval $b(i)$. (As the algorithm runs, $v(i)$ shrinks and $b(i)$ grows.) The list corresponds to the subsequence L described above. At the top of each **for** loop, the following invariant holds: L contains a clockwise subsequence of $VP(q)$ such that when only the barriers and blocking intervals given by L are present, every such barrier is tangent-visible from B^q . (At the end of execution, L contains exactly the elements that generate $TV(q)$.) During each loop, a new element of $VP(q)$ is inserted at the end of L . The operations necessary to restore the invariant affect only the ends of the list.

The algorithm uses just one geometric primitive: Given two elements from the visibility polygon boundary i and j and their associated blocking intervals $b(i)$ and $b(j)$, the primitive determines all tangent visibilities that exist when only the barriers B^i and B^j and the blocking intervals $b(i)$ and $b(j)$ are present. If the barrier B^i is not tangent-visible under these circumstances, we say that $b(j)$ and B^j *hide* B^i . The primitive requires a constant number of the tangent- and intersection-finding operations (1)–(4), and therefore takes $O(\tau)$ time. (Operation (4) is needed because we replaced B^i by a parallelogram for each segment i in $VP(q)$.)

The algorithm runs in $O(\tau n)$ time. Each execution of the outer **for** loop takes $O(\tau)$ time, exclusive of the time spent in the two inner **while** loops. Each time one of those loops is executed, an element of L is deleted. No more than n elements are ever added to L , so the inner loops are executed at most n times altogether; each execution takes $O(\tau)$ time.

The algorithm and the preceding discussion constitute a proof of the following lemma.

Set L to be an empty double-ended queue. The functions $head(L)$ and $tail(L)$ return the elements at the ends of the queue.

for each i in $VP(q)$ in clockwise order do

begin

Let $b(i)$ be the interval given by the intersection of B^i and B^q

while B^i and $b(i)$ hide $B^{head(L)}$ in the interval $v(head(L))$ do

begin

Merge $b(head(L))$ into $b(i)$;

Delete $head(L)$ from L ;

end

while B^i and $b(i)$ hide $B^{tail(L)}$ in the interval $v(tail(L))$ do

begin

Merge $b(tail(L))$ into $b(i)$;

Delete $tail(L)$ from L ;

end

Let U be the set of barriers and blocking intervals $B^{head(L)}$, $B^{tail(L)}$, $b(head(L))$, $b(tail(L))$, and $b(i)$;

(The following test uses the primitive operation twice:)

if B^i is tangent-visible from B^q in the presence of U then

begin

Determine the endpoints of $v(i)$ by comparisons with U ;

Adjust the endpoints of $v(head(L))$ and $v(tail(L))$ if B^i and $b(i)$ affect them;

Insert i at the tail of L ;

(The barriers generated by i , $head(L)$, and $tail(L)$ are all visible and separate their associated blocking intervals, so no blocking intervals need to be merged.)

end

else (B^i is not tangent-visible)

Merge $b(i)$ into $b(tail(L))$;

end

Figure 3.13. An algorithm to compute $TV(q)$

Lemma 3.6. *If each of the operations (1), (2), (3), and (4) given above can be performed in time τ , then the tangent-visible sequence $TV(q)$ can be produced from $VP(q)$ in $O(\tau n)$ time.*

When the algorithm terminates, L contains $g(TV(q))$, the generators of the barriers that are clockwise tangent-visible from B^q . It is easy to find the path graph edges that are clockwise-tangent to B^q given the list L . For each element $i \in L$, if either end of the visible range $v(i)$ is delimited by a common tangent of B^q and B^i , then we add the tangent edge to the path graph.

We can construct the tangent edges of the path graph, but we still need to find the edges from α and ω that augment the path graph. However, the ideas used in Section 3.3 and the preceding lemma apply to this problem as well.

Lemma 3.7. *The edges from α and ω that augment the path graph can be found in $O(\tau n)$ time.*

Proof: Lemmas 3.3 through 3.5 discuss tangent-visibility of barriers. Similar lemmas are true for visibility of barriers from a point. For example, if r lies on the segment connecting α and p , no point of B^p is visible from α , which proves a lemma analogous to Lemma 3.3. Let $\tilde{v}_T(q, p)$ have the same meaning for visibility from q as $v_T(q, p)$ has for tangent-visibility from B^q . Then $\tilde{v}_T(q, p)$ is an interval for any T : as in the proof of Lemma 3.4, we consider $\tilde{v}_r(q, p)$ for any r and show that B^r cannot intersect $\overline{qv_b}$ in Figure 3.14 without touching $\overline{qv_a}$ or $\overline{qv_c}$.

The analogues of Lemma 3.5 and Theorem 3.1 follow directly from the modified Lemmas 3.3 and 3.4, and the Graham scan of Lemma 3.6 is easily modified to handle visibility from a point, given that B satisfies condition (4) given above. ■

This section has described three separate aspects of constructing the path graph. The method of Kedem et al. finds the boundary of the barriers in $O(n\tau \log^2 n)$ time. The algorithm given in Figure 3.13 produces the path graph tangent edges in $O(n^2\tau)$

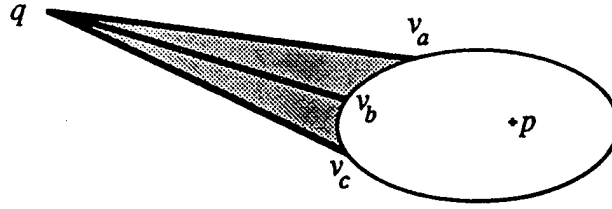


Figure 3.14. Proof that $\tilde{v}_T(q, p)$ is an interval; see Figure 3.8. B^r cannot intersect $\overline{qv_b}$ (block visibility in direction b) without also intersecting either $\overline{qv_a}$ or $\overline{qv_c}$.

time. A variant on the algorithm, sketched in the previous lemma, finds the edges from α and ω that augment the path graph in $O(n\tau)$ time. The endpoints of these edges, taken together with the barrier boundaries, give all the boundary edges of the augmented path graph. The following theorem summarizes these results:

Theorem 3.2. *The augmented path graph can be constructed in $O(\tau n^2)$ time.*

3.5 Pruning the path graph

It is possible to find a shortest path by running Dijkstra's algorithm on the augmented path graph, G_a . However, because the tangent edges in G_a can have $\Omega(n^2)$ endpoints altogether, the best implementation of Dijkstra's algorithm known may take $\Omega(n^2 \log n)$ time in the worst case [FT84]. To reduce the time to $O(n^2)$, we need a graph with fewer nodes. This section and the following section show how to produce a graph equivalent to G_a that has fewer nodes. This section deletes unusable tangent edges and their endpoints; the next section groups nodes together and modifies edges to link the groups.

We now show how to eliminate some edges of G_a that lie on no shortest path between α and ω ; we identify these edges using only local tests. After deleting the edges from G_a , we delete any resulting nodes of degree two and merge their incident edges. The result is a *pruned path graph*, G_p .

We have already noted that in the path graph there are both clockwise and

counterclockwise tangent edges from a barrier arc. There are inner and outer common tangent edges of both types. Altogether, the four possible combinations of clockwise/counterclockwise with inner/outer give four classes of path graph edges from each barrier arc. Within each class all tangents to an arc are cyclically ordered. The following lemma shows how to identify unusable edges of each class.

Lemma 3.8. Suppose e_1 and e_2 in G_a are consecutive clockwise outer tangents going from B^q to barriers B^{p_1} and B^{p_2} with p_2 to the right of the directed line from q to p_1 . If B^{p_1} and B^{p_2} intersect, the continuation of e_2 intersects B^{p_1} , thus defining a bay. (See Figure 3.15.) Then edge e_2 cannot appear on a shortest path unless α or ω lies in the bay.

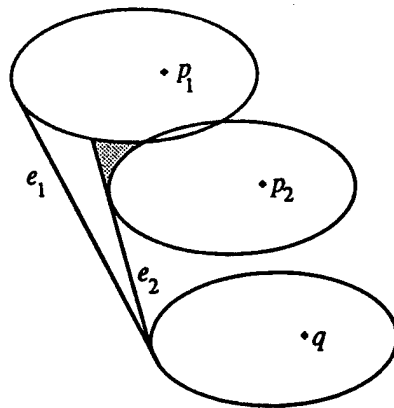


Figure 3.15. Because B^{p_1} and B^{p_2} intersect, edge e_2 cannot appear on a shortest path unless α or ω lies in the shaded bay.

Proof: The continuation of e_2 must intersect B^{p_1} ; otherwise it would separate B^{p_1} and B^{p_2} . If neither α nor ω lies in the bay, any path that enters the bay can be shortened by re-routing it to avoid the bay. Any path that uses e_2 has no forward continuation (wrapping around the corner), because all such continuations enter the bay. ■

Similar conclusions hold for pairs of consecutive edges from the other three classes: clockwise inner tangents, counterclockwise outer tangents, and counterclockwise inner tangents.

The preceding lemma gives a way of pruning the augmented path graph to remove unneeded edges. The following lemma shows that the pruning can be performed in $O(\tau n^2)$ time overall.

Lemma 3.9. *All prunable edges from arcs generated by a polygon vertex q can be found and discarded in $O(\tau n)$ time.*

Proof: Each pruning test takes only $O(\tau)$ time. Suppose that e_1 and e_2 are consecutive edges of the same class tangent to B^{p_1} and B^{p_2} . If B^{p_1} and B^{p_2} intersect, we can find the points a , b , and c shown in Figure 3.16 using $O(\tau)$ time. Since α and ω lie outside the barriers, α or ω lies in the shaded bay if and only if it lies in the triangle Δabc , which can be checked in constant time. If neither α nor ω lies in Δabc , then edge e_2 may be removed.

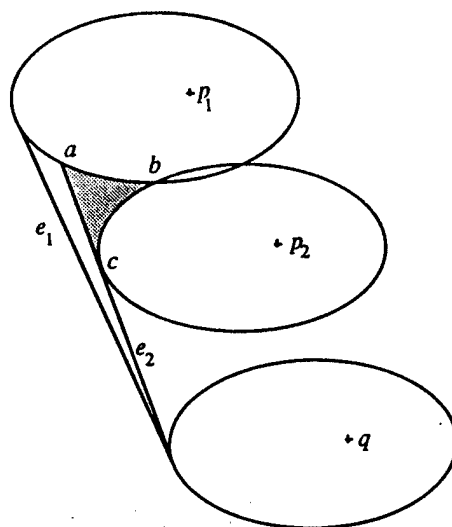


Figure 3.16. Point α or ω is in the bay if and only if it is in Δabc . If neither α nor ω lies in Δabc , then e_2 can be pruned.

To check whether an edge needs pruning, we need only test it against edges of the same class whose endpoints precede or follow it on the same arc. When an edge is pruned, two edges that were not neighbors become neighbors and must be tested. However, because there are $O(n)$ path graph edges of each class from arcs generated by q , only $O(n)$ tests are needed for each polygon vertex q . ■

If two successive edges of the same class go to overlapping translated copies of B and neither α nor ω lies in the bay reached by the inner edge (e_2 in Figure 3.16), the inner edge is pruned. In cases like the one shown in Figure 3.17, many unnecessary edges can be pruned. After pruning, at most two pairs of successive edges go to overlapping barriers. (Each unpruned inner edge defines a bay containing α or ω .) The following lemma, which is used in the next section, shows that if barriers reached by successive edges of the same class do not overlap, neither do barriers reached by non-successive edges.

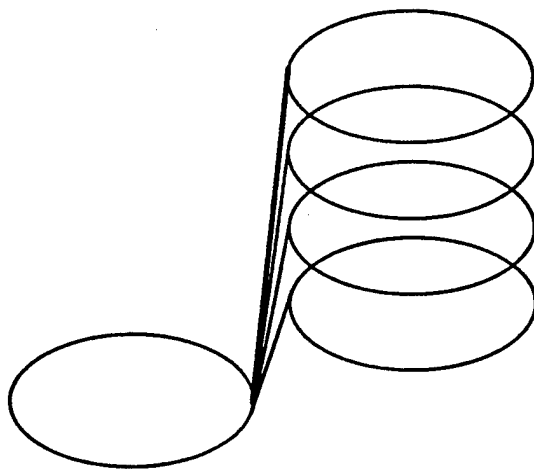


Figure 3.17. In this example, pruning removes many useless edges. Of the edges shown, only the leftmost can appear on a shortest path, since neither α nor ω lies in any of the bays formed by the other edges.

Lemma 3.10. *Let E be a subset of the edges in G_p that connect B^q to arcs of other barriers, chosen so that all edges in E have the same class. Let U be the set*

of polygon vertices that generate the arcs reached by edges of E . The angles of the edges in E give a cyclic ordering on the elements of U . If $B^p \cap B^s$ is empty for all consecutive generators p and s , then $B^p \cap B^s$ is empty for all distinct p and s in U .

Proof: Suppose to the contrary that $B^p \cap B^s$ is non-empty for some p and s in U . Choose p and s such that p precedes s and the size of the subset of U between p and s is minimized. By hypothesis this subset contains some element r . Since the common tangent from B^q to B^r reaches a visible point, the point of tangency is in the shaded region (see Figure 3.18). This implies that B^r extends beyond (in Figure 3.18, to the right of) any line passing through the intersection of B^p and B^s and tangent to B^q . To get out of the shaded region without overlapping B^p or B^s , B^r must cross the tangent from B^q to B^s , and hence that edge cannot be in the path graph, a contradiction. ■

3.6 Node coalescing in the pruned path graph

We have shown that any shortest path from α to ω follows edges of the pruned path graph, G_p . Because G_p may still have $\Omega(n^2)$ nodes, Dijkstra's algorithm may require $\Omega(n^2 \log n)$ time when applied to it. In this section, we show how to modify the graph so that Dijkstra's algorithm takes only $O(n^2)$ time when applied to the result. Because nodes of G_p are coalesced during the modification, we call the resulting graph the *coalesced graph*, G_c .

The modifications of this section group consecutive nodes on the boundaries of barriers. To better describe the modifications, we first characterize barrier boundaries. Every boundary between a maximal connected barrier region and the barrier-free region is a ring of path graph nodes and edges. (A barrier may have several such rings if it has holes.) Any subpath of a shortest path that uses edges from one of these rings follows them in a consistent direction, either clockwise or counterclockwise. A ring is entered and left via tangents to it, and these connections are consistent with the orientation of the ring. See Figure 3.19. Any path in G_c

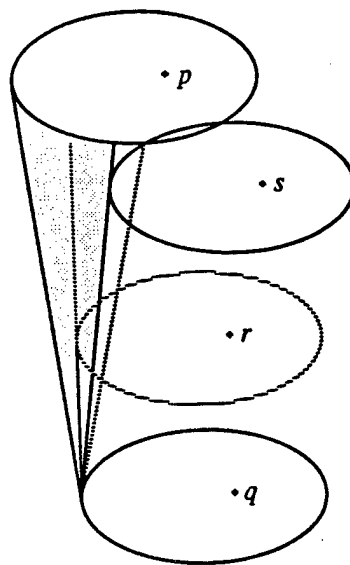


Figure 3.18. Neighbor non-intersection implies global non-intersection. The tangents from B^q to B^p , B^r , and B^s occur in clockwise order. If B^p and B^s intersect, but B^r intersects neither of them, then B^r blocks the tangent to B^s .

that traverses rings in a single direction and connects to them consistently is called *forward-going*.

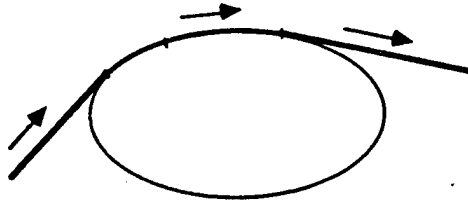


Figure 3.19. A forward-going subpath

An edge of the pruned path graph can be traversed in either direction, since the path graph is undirected. As the first step in forming the coalesced graph, we transform G_p into a directed graph, replacing each undirected barrier boundary ring by two oppositely directed copies of itself. Each tangent edge is replaced by two oppositely directed copies of itself, and each copy is connected to the ring consistent with its direction. The procedure of duplication and connection is depicted in Figure 3.20. Path graph edges from α and ω are replaced by directed edges (outgoing from α , incoming to ω) whose other endpoints are connected to the appropriate directed rings. In this directed graph, which we call G_d , only forward motion is possible. Every shortest path from α to ω is present in G_d .

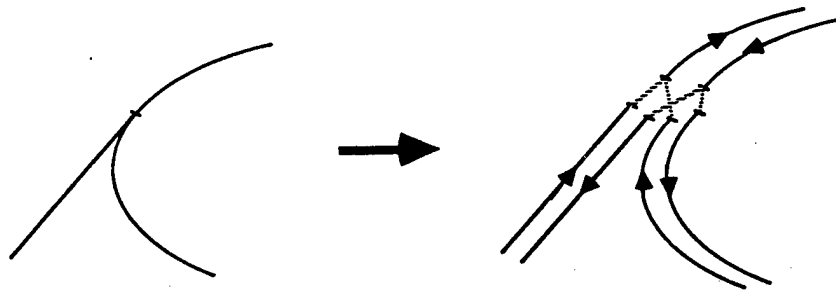


Figure 3.20. Construction of G_d . Each path graph edge is replaced by two oppositely directed edges; the edges are linked up to allow only forward-going paths.

To help define the nodes of the coalesced graph, we introduce *distinguished nodes*, which are a subset of the nodes of G_d . (We show how to select this subset later.) The distinguished nodes have three properties that we use presently. First, all arc endpoints are distinguished nodes. Second, the tangents to consecutive distinguished nodes on a single arc differ in direction by at most $\pi/2$. Third, every tangent edge that ends between two consecutive distinguished nodes on a single arc is at least as long as the interval between the distinguished nodes, measured along the barrier boundary.

The distinguished nodes break the barrier boundary rings into intervals. (Note that each point on the boundary of a barrier belongs to two intervals: one for each direction of traversal.) Somewhat unconventionally, we view the distinguished nodes as single-point closed intervals distinct from the open intervals they separate. The set of all such intervals (both open and closed) forms the node set of the coalesced graph, G_c .

The edges of the coalesced graph connect the intervals defined by the distinguished nodes. An edge linking two intervals is assigned a weight (roughly equivalent to length) as if it went from the forward point of one interval (as defined by the ring orientation) to the forward point of the other.

We first assign weights to tangent edges. Each tangent edge of G_d links intervals (either open or closed) delimited by distinguished nodes. In Figure 3.21, the edge from a to b links the intervals (a', a'') and (b', b'') . We assign the corresponding edge in G_c the weight $|ab| - |a''a| + |bb''|$, where distances $|a''a|$ and $|bb''|$ are measured along the barrier boundary. This is the net forward motion of a hypothetical car that backs up from a'' to a along the arc, goes forward along $\overline{ab}$, and then advances from b to b'' along that arc. (The fifth condition on B means that the length computation takes $O(\tau)$ time.) Since there is at most one edge of G_d from one arc to any other, there is at most one tangent edge connecting any pair of intervals. Because $|a'a''|$ and $|b'b''|$ are no greater than $|ab|$, the edge linking the intervals of a and b in the coalesced graph has non-negative weight.

We now assign weights to boundary edges. The distinguished nodes break each barrier boundary into a ring of intervals. These rings are a coarsening of the rings

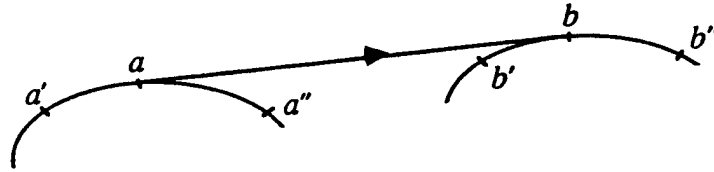


Figure 3.21. The edge of G_c corresponding to the directed edge from a to b is assigned weight $|ab| - |a''a| + |bb'|$

of boundary edges in G_d . From a single-point interval $[a', a']$ to its open successor (a', a'') we create an edge with weight $|a'a''|$. An edge of weight 0 goes from the open interval (a', a'') to its closed successor $[a'', a'']$. Note that each edge joining adjacent intervals has weight equal to the distance between the forward points of the intervals.

If we are to use the coalesced graph to find shortest paths, we must show that shortest paths in G_c correspond to shortest paths in the augmented path graph.

Lemma 3.11. *Any minimum weight path from α to ω through the coalesced graph corresponds to a shortest path of the same length in the augmented path graph.*

Proof: The proof has two parts. We first show that every path in G_d maps to a path in the coalesced graph with the same length. Next we show that every path in G_c that does not correspond to a forward-going path has weight greater than the length of the shortest forward-going path.

A shortest path through G_d is also a shortest path in the augmented path graph, since a shortest path is a forward-going path. When the nodes of G_d are coalesced into the intervals of G_c , the lengths of forward-going paths are preserved. The coalesced edge weights are calculated so that when a path travels forward along a barrier boundary, the perimeter length is accounted correctly. For example, in Figure 3.22, the forward-going path from s to f has length $|sa| + |ab| + |bf|$. In G_c the path has two edges, one from s to the interval (a', a'') and one from the interval

to f . These edges have weights $|sa| + |aa''|$ and $|bf| - |a''b|$, which sum to the correct value.

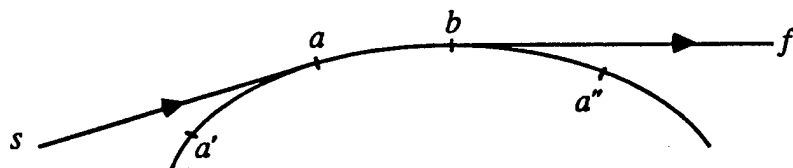


Figure 3.22. Forward-going paths are charged accurately in the coalesced graph. The path from s to f has weight $(|sa| + |aa''|) + (|bf| - |a''b|)$, which is equal to $|sa| + |ab| + |bf|$, the Euclidean length of the path.

Grouping edge endpoints into intervals allows some paths in G_c that are not derived from forward-going paths. These are subpaths that enter an interval, back up, and then leave, as in Figure 3.23. The edge weights of such a subpath in G_c add up to less than the Euclidean length of the path. However, we show that the total weight of a path in G_c from α to ω that includes such a subpath is still greater than the length of the shortest path in G_d .

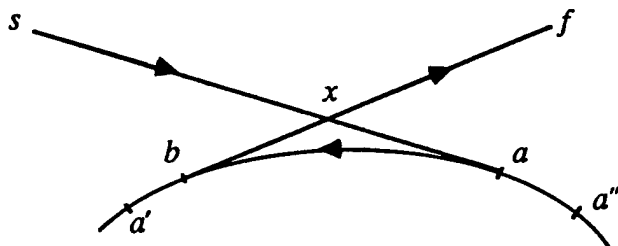


Figure 3.23. Loops are undercharged, but not too much. The path $s \rightarrow a \rightarrow b \rightarrow f$ has weight $|sa| + |bf| - |ab|$, which is less than its length, but more than the length of the path $s \rightarrow x \rightarrow f$. This latter path is longer than the shortest path from s to f , so the loop cannot be mistaken for a shortest path.

The subpath in Figure 3.23 has weight $(|sa| + |aa''|) + (|bf| - |a''b|) = |sa| + |bf| - |ab|$, which is less than $|sa| + |ab| + |bf|$, the actual length

of the subpath. However, because $|sa|$ and $|bf|$ are both greater than $|a'a''|$ and the difference between the tangent directions at b and a is at most $\pi/2$ ($\angle sxf \geq \pi/2$), the edges $\overline{sa}$ and $\overline{bf}$ intersect at x . Since $|ab|$ is measured along a convex barrier boundary, $|ax| + |xb| > |ab|$. Though the weight of the subpath is less than its actual length, it is still more than $|sx| + |xf|$, the length of the subpath that follows $\overline{sx}$ to x and then follows $\overline{xf}$. Because of the open corner at x , this second subpath cannot be part of any shortest path.

The preceding analysis assumes that the loop $x \rightarrow a \rightarrow b \rightarrow x$ is isolated from any other such loop. We now show that the assumption is valid. If an edge $\overline{ba}$ connects two loops as in Figure 3.24(a), the two loops are independent: the subpath in G_c that contains them has weight at least $|sc| + |cd| + |df|$, which is the length of an easily identified subpath. If intersections c and d appeared in opposite order, the two loops would not be independent: the lower bound would be $|sc| + |df| - |cd|$, which corresponds to no recognizable subpath. However, because $\angle sca$ and $\angle fdb$ are at least $\pi/2$, the interlocking loops of Figure 3.24(b) are impossible.

Any path from α to ω in G_c that includes instances of doubling back, as in Figure 3.23, has weight greater than the length of the shortest path in G_d . Consider replacing each instance of doubling back by the shortcut corresponding to the subpath in Figure 3.23 that follows $\overline{sx}$ to x and then follows $\overline{xf}$. The length of the resulting barrier-avoiding path is less than the weight of the path through G_c from which it is derived. The resulting path is not a shortest path from α to ω , since short-cutting the corner corresponding to $\angle sxf$ gives a shorter path. This means that no path through G_c that is not forward-going can be mistaken for a shortest path. ■

The proof of the preceding lemma uses the three properties of distinguished nodes mentioned earlier. We now show how to select the distinguished nodes so

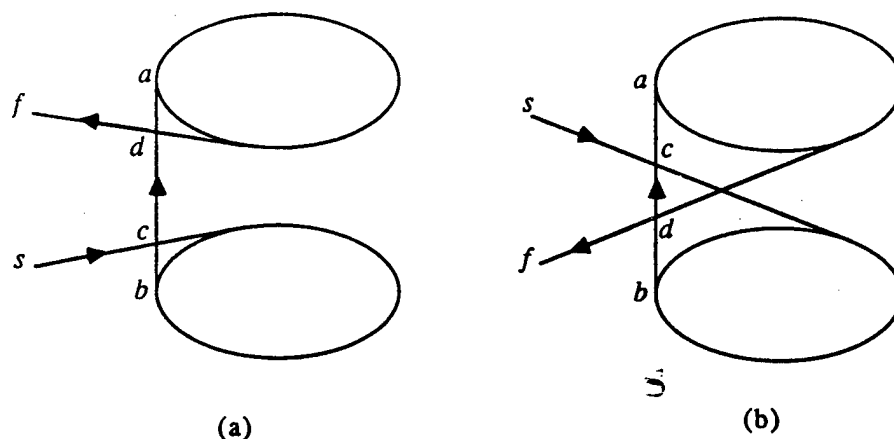


Figure 3.24. Two loops connected by a single edge; (a) possible and (b) impossible configurations. Because angles $\angle sca$ and $\angle fdb$ are at least $\pi/2$, the points a , d , c , and b occur in that order along $\overline{ab}$.

that these properties hold. (The properties, once again, are the following: all arc endpoints are distinguished nodes, the tangent directions at consecutive distinguished nodes differ by at most $\pi/2$, and every tangent edge is at least as long as the intervals containing its endpoints.) There are a number of ways of selecting distinguished nodes; the one that we adopt here is perhaps the simplest nontrivial method.

Our approach characterizes the robot with two parameters. We take l to be the *diameter* of B , that is, the maximum distance between two points of B . This is also equal to the maximum separation of two parallel supporting lines of B . The minimum separation w of two such lines is called the *width* of B .

The first distinguished nodes we choose are those path graph nodes that are intersections of arcs or boundary segments with each other, including the junctions between arcs and segments at their endpoints. Kedem et al. [KLPS86] show that there are only $O(n)$ such intersections. We add distinguished nodes to each arc so that no interval on an arc is longer than l . We then add distinguished nodes to guarantee that tangent directions at successive distinguished nodes differ by at most $\pi/2$. (Note that these two conditions on intervals need only be met if the

interval contains an endpoint of a tangent edge. If an interval is too long or turns by more than $\pi/2$, but has no tangent endpoints in it, there is no need to break it in two with a distinguished node. Therefore all distinguished nodes can be taken to be path graph nodes.) So far, we have chosen $O(n)$ distinguished nodes. Now we add to the set of distinguished nodes the endpoints of all tangent edges shorter than l . This guarantees the non-negativity of all edge weights in G_c . The following lemma bounds the number of distinguished nodes added in the final step.

Lemma 3.12. *Let w be the width and l the diameter of B . The pruned path graph has $O(nl/w)$ edges shorter than l .*

Proof: The proof concentrates on the barriers connected to B^q by tangent edges. It uses Lemma 3.10 to divide the barriers into four sets, each of them free of overlaps. For each set, the subset connected to B^q by edges shorter than l lies in a relatively small region close to B^q . Area arguments show that there are only $O(l/w)$ barriers in each such subset.

As in Lemma 3.10, let the set U contain the generators of the arcs reached from B^q by tangent edges of a single class in the pruned graph, G_p . The elements of U are cyclically ordered by the angles of the edges. At most two pairs of consecutive vertices in U generate overlapping barriers. If we remove from U any vertex (there are at most two) whose corresponding edge leads to a bay containing α or ω , Lemma 3.10 implies that the barriers generated by the remaining vertices are pairwise disjoint.

Let $\mathcal{R}$ be the region consisting of all points closer than $2l$ to B^q . The area of $\mathcal{R}$ is $O(l^2)$. Every vertex in U corresponding to a tangent edge shorter than l generates a barrier that is entirely contained in $\mathcal{R}$. (See Figure 3.25.) These barriers are non-overlapping, and each has area $\Omega(lw)$; hence there are at most $O(l/w)$ of them. This argument applies to all four tangent edge classes, and so for each polygon vertex q , there are $O(l/w)$ edges in G_p tangent to B^q and shorter than l . Summing over all vertices proves the lemma. ■

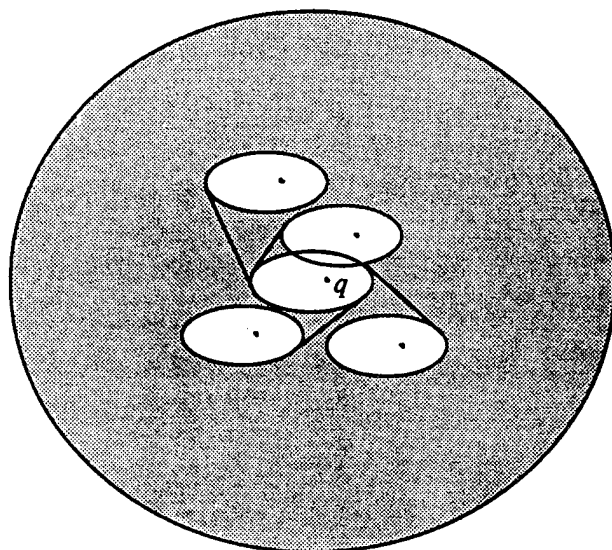


Figure 3.25. The barriers reached from B^q by tangent edges of a single class in the pruned path graph are non-overlapping (with at most two exceptions). Those connected to B^q by edges shorter than l lie completely inside the region closer than $2l$ to B^q (the shaded region). This region has area $O(l^2)$, and each barrier has area $\Omega(lw)$, so the number of short edges is $O(l/w)$.

Our algorithm finds shortest paths by producing a graph on which Dijkstra's single-source shortest path algorithm will run quickly, then invoking that algorithm. Sections 3.4 through 3.6 show how to construct the coalesced graph in $O(\tau n^2)$ time. Dijkstra's algorithm takes $O(n^2 + \min(n, l/w)n \log n)$ to find the shortest path in G_c ; this shortest path corresponds directly to the shortest path in the original setting. (The min function is present because there are at most $O(n^2)$ tangent edges shorter than l in the path graph.) Overall, our algorithm uses time $O(\tau n^2 + \min(n, l/w)n \log n)$ and space $O(n^2)$. If B is independent of n , as in the important special case in which B is a disk, the algorithm finds shortest paths in $O(n^2)$ time and space. If B depends on n but the aspect ratio l/w is $O(n/\log n)$, the running time of the algorithm is still $O(\tau n^2)$. This concludes the proof of the following theorem:

Theorem 3.3. *Given a non-rotating convex body A , a set S of simple polygons with a total of n vertices, and two points α and ω , it is possible to find a shortest polygon-avoiding path for A from α to ω in $O(n^2)$ time, where the implied constant depends on the complexity of A .*

3.7 Conclusions and open questions

In this chapter we have presented an algorithm to find shortest paths for a non-rotating convex body moving among polygonal obstacles. We first reduced the problem to the equivalent problem of moving a point among "fattened" obstacles [LW79,GRS83]. We then introduced the concept of tangent-visibility to describe the sequence of barrier boundary points swept by a tangent ray continuously rotating around a placement of B , the inverted robot. This concept allowed us to compute efficiently the path graph of the barriers, which contains the edges of all shortest paths. Unfortunately the path graph may contain $\Omega(n^2)$ nodes, so that Dijkstra's shortest path algorithm runs in $\Omega(n^2 \log n)$ time when applied to it. By pruning certain path graph edges and then coalescing the remaining nodes into distinguished nodes, we were able to reduce the total number of nodes to $O(n)$ and thus get a shortest path algorithm that runs in $O(n^2)$ time.

Perhaps the least satisfying feature of this algorithm is its dependence on the aspect ratio l/w of the object to be moved. According to our bounds, the algorithm could require $\Omega(n^2 \log n)$ time to find shortest paths for an ellipse whose aspect ratio is a linear function of n . This is disturbing, since in the limit such an ellipse becomes a segment, for which shortest paths are easy to find.

Our algorithm uses $O(nl/w)$ distinguished nodes in the construction of G_c . However, this upper bound on the number of necessary distinguished nodes is very crude. If we omit inner common tangents from the path graph, we can show that only $O(n \log(l/w))$ distinguished nodes are needed to construct G_c . We have not yet been able to extend this bound to allow inner common tangents, though we believe such an extension is possible.

Both upper bounds mentioned are local bounds: we prove that no fattened polygon vertex needs to have more than $O(l/w)$ or $O(\log(l/w))$ distinguished vertices on its boundary. Such bounds fail to consider the interactions of multiple barriers. It seems quite possible that a global argument could bound the number of necessary distinguished nodes by $O(n^2 / \log n)$ or something even smaller.

The gap between upper and lower bounds is large; we can construct an example using only outer common tangents in which $\Omega(\sqrt{\log(l/w)})$ distinguished nodes are needed on a particular fattened vertex, but we have no super-linear global lower bounds. Improvements to either our lower or upper bounds would be welcome.

The algorithm we have presented does not process S before α and ω are given. However, it might be possible to construct approximations to the whole progression of graphs from G to G_c before α and ω are given, then modify them in $O(rn)$ time once the source and destination are known. The most difficult preprocessing task would be pruning edges and simultaneously building a structure to identify the pruned edges that must be added back once α and ω are known. This is an interesting problem, but it is probably not worth working out the details so long as the final step of the algorithm has to run Dijkstra's algorithm, which may take $\Omega(n^2)$ time.

Chapter 4

Linear Time Algorithms for Visibility and Shortest Path Problems Inside Triangulated Simple Polygons

The essence of all good strategy is simplicity.

— Eric Ambler, *Journey into Fear* (1940)

Nothing is as simple as it ought to be.

— John Hershberger (1987)

The preceding two chapters consider visibility and shortest path problems in which the obstacles are disjoint line segments in the plane. Because they do not take

advantage of special relationships among the segments, the algorithms of Chapters 2 and 3 run in $\Theta(n^2)$ time. Algorithms that exploit the geometry of the obstacles can run faster than $O(n^2)$, as Reif and Storer show [RS85]. Bearing this in mind, we focus our attention on problems in which the segments form a simple polygon. We obtain algorithms that are optimal after the polygon has been triangulated.

All of the algorithms in the next three chapters run in triangulated simple polygons. (To triangulate a polygon with n vertices, we add $n - 3$ non-intersecting diagonals to the polygon interior, splitting it up into $n - 2$ disjoint triangles.) A recent algorithm of Tarjan and Van Wyk [TV86] for triangulating simple polygons runs in $O(n \log \log n)$ time, thereby improving the previous $O(n \log n)$ algorithm of Garey et al. [GJPT78]. Although this result falls short of the goal of achieving a linear time bound, it shows that triangulating simple polygons is a simpler problem than sorting, and raises the hope that linear-time triangulation might be possible. The result thus renews interest in linear-time algorithms on already-triangulated polygons, a considerable number of which have been recently developed (for lists of these see [FM84,TV86]). Our algorithms fall into this class.

In this chapter we present algorithms that solve the following problems in linear time, given a triangulated simple polygon P with n sides:

- (1) Given a fixed source point x inside P , calculate the shortest paths inside P from x to all vertices of P . (Our algorithm even provides a linear-time processing of P into a data structure from which the length of the shortest path inside P from x to any desired target point y can be found in time $O(\log n)$; the path itself can be found in time $O(\log n + k)$, where k is the number of segments along the path).
- (2) Given a fixed edge e of P , calculate the subpolygon $Vis(P, e)$ consisting of all points in P visible from (some point on) e .
- (3) Given a fixed edge e of P , preprocess P so that, given any query ray r emanating from e into P , the first point on the boundary of P hit by r can be found in $O(\log n)$ time.

- (4) Given a fixed edge e of P , preprocess P so that, given any point x inside P , the subsegment of e visible from x can be computed in $O(\log n)$ time.
- (5) Given a vertex x of P lying on its convex hull, calculate for all other vertices y of P the clockwise and counterclockwise convex ropes around P from x to y , when such paths exist. (These are polygonal paths in the exterior of P from x to y that wrap around P , always turning in a clockwise (resp. counterclockwise) direction; see Section 4.1.1.)

Our results improve previous algorithms for some of these problems (refer to [Cha82,CG85,PS85a]). All of the algorithms in this chapter are based on the solution to Problem (1) and exploit interesting relationships between visibility and shortest path problems for a simple polygon. Our algorithm for solving Problem (1) extends the technique of Lee and Preparata [LP84] for calculating the shortest path inside P between a single pair of points. To obtain linear-time performance, it uses *finger search trees*, a data structure for efficient access into an ordered list when there is locality of reference (see [GMPR77,HM82]).

The chapter contains three main sections. Section 4.1 presents a linear-time solution to the shortest path Problem (1), constructing all shortest paths from a source to the polygon vertices. We solve Problem (5) as an easy application of this result. Section 4.2 extends the result of Section 4.1 to answer shortest path queries, as specified in the second part of (1). This extension is important in Chapter 5, so Section 4.2 characterizes it in some detail. Section 4.3 concludes the chapter by solving the visibility problems (2)–(4).

4.1 Calculating a shortest path tree for a simple polygon

*Two roads diverged in a wood, and I—
I took the one less travelled by
And that has made all the difference.*

— Robert Frost, “The Road Not Taken” (1916)

Let P be a (triangulated) simple polygon having n vertices, and let s be a given source vertex of P . (Our algorithm will also apply, with some minor modifications, to the case in which s is an arbitrary point in the interior or on the boundary of P . For the sake of exposition, the algorithm below is described for the case in which s is a vertex of P , and we later comment on the modifications required to handle the case of an arbitrary source s .) For any two points p and q inside P or on its boundary, we denote by $\pi(p, q)$ the Euclidean shortest path from p to q that does not go outside P . It is well known (see [LP84]) that $\pi(p, q)$ is a unique polygonal path whose corners are vertices of P . Furthermore, the union $\bigcup_v \pi(s, v)$, taken over all vertices v of P , is a plane tree rooted at s . We call this tree the *shortest path tree* of P with respect to s and denote it by $SPT(P, s)$. (See Figure 4.1.) This tree has n nodes, namely the vertices of P , and its edges are straight segments connecting these nodes. Our goal is to calculate this tree in linear time, once a triangulation of P is given.

Let G be a triangulation of the interior of P . The planar dual of G (whose vertices are the triangles in G and whose edges join two such triangles if they share an edge) is a tree T , each of whose vertices has degree at most three. The counterclockwise order of the edges incident to a vertex of T with degree three is determined by the triangulation. For an example of a triangulation and its dual

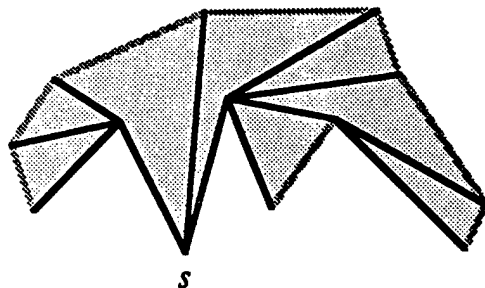


Figure 4.1. The shortest path tree is the union of the shortest paths from a source vertex s to all other vertices.

tree, see Figure 4.2. If a vertex t of P is incident to k diagonals, then it is incident to $k + 1$ triangles. We call the nodes of T that correspond to these triangles the *neighbors* of t , written $N(t)$. For each vertex t of P , there is a unique minimal path in T connecting a node in $N(s)$ to a node in $N(t)$. The edges of this path correspond to an ordered sequence of diagonals $d_1, d_2, \dots, d_l$ of P . (Choosing the minimal path between $N(s)$ and $N(t)$ ensures that the sequence $d_1, \dots, d_l$ contains no diagonal with s or t as an endpoint.) The diagonals d_i are exactly those that divide P into two parts, each containing one of s and t . Because any path that crosses a diagonal twice can be shortened by removing both crossings, $\pi(s, t)$ intersects only the diagonals d_i , $1 \leq i \leq l$, and each of them exactly once.

Let $d = \overline{uw}$ be a diagonal or an edge of P and let a be the nearest common ancestor of u and w in the shortest path tree $SPT(P, s)$. It is shown in [LP84] that $\pi(a, u)$ and $\pi(a, w)$ are both *inward convex*; that is, the convex hull of each of these subpaths lies outside the open region bounded by $\pi(a, u)$, $\pi(a, w)$, and the segment $\overline{uw}$. This implies that along each of $\pi(a, u)$ and $\pi(a, w)$ the slopes of the sides change in a monotonic fashion. Following [LP84], we call the union $F = F_{\overline{uw}} = \pi(a, u) \cup \pi(a, w)$ the *funnel* associated with $d = \overline{uw}$; we call a the *apex* of the funnel and $\overline{uw}$ the *base* of the funnel. See Figure 4.3 for an example of a funnel. Note that the apex of the funnel may be s , and that the funnel can degenerate into a single segment: if $a = u$, then $\pi(a, w) = \overline{uw}$.

Suppose next that d is a diagonal of P used by G , and let Δuwx be the unique

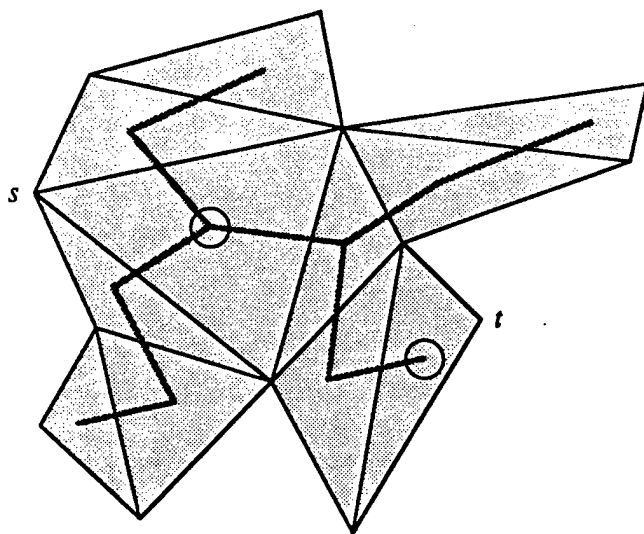


Figure 4.2. A triangulation and its dual. The shaded tree T inside the polygon is the dual of the triangulation. T has $n - 2$ nodes and $n - 3$ edges; each edge corresponds to a diagonal of the triangulation. Vertex s is incident to three triangles, vertex t to only one. These two sets of triangles correspond to two sets of nodes. The shortest path in T between the two sets connects the circled nodes of T ; the edges on this path correspond to the diagonals crossed by $\pi(s, t)$.

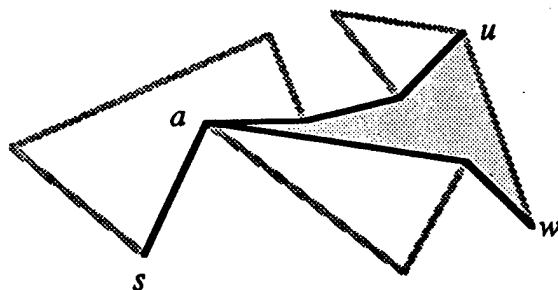


Figure 4.3. The two shortest paths from s to u and w define the funnel belonging to edge $\overline{uw}$. The vertex a at which the paths separate is the apex of the funnel, and the funnel itself is the union of the two shortest paths $\pi(a, u)$ and $\pi(a, w)$.

triangle in G having d as an edge that does not intersect the area bounded between F and d . Then the shortest path from s to x must start with $\pi(s, a)$ and then either continue along the straight segment $\overline{ax}$ if this segment does not intersect F , or else proceed along either $\pi(a, u)$ or $\pi(a, w)$ to a vertex v such that $\overline{vx}$ is a tangent to F at v , and then continue along the straight segment $\overline{vx}$ (see Figure 4.4). The tangent vertex is the unique funnel vertex v where the slope of the funnel and the slope of $\overline{vx}$ are equal. These observations form the basis for the algorithm of Lee and Preparata [LP84], and for ours.

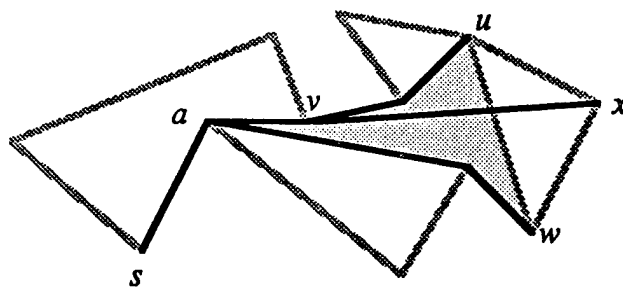


Figure 4.4. Splitting a funnel

We are now ready to describe our algorithm, which is essentially a depth-first search exploration of the triangulation, starting with a triangle incident to the source. Each step processes an unexplored triangle that shares a diagonal with one already explored. The algorithm maintains a shortest path tree in the subpolygon explored so far. It also maintains a funnel for each diagonal at the boundary of the explored region. When the algorithm crosses into a new triangle, it splits the funnel belonging to the crossed diagonal into two funnels, one for each of the triangle's other two sides. The splitting edge is the tangent from the triangle's third vertex to the interior of the funnel. When one of the new sides is a polygon edge rather than a diagonal, the algorithm has found a funnel of the final shortest path tree.

As the algorithm comes to process a diagonal $d = \overline{uw}$ of P , it maintains the current funnel $F = F_{\overline{uw}}$ as a sorted list $[u_l, u_{l-1}, \dots, u_1, a, w_1, \dots, w_k]$, where $a = u_0 = w_0$ is the apex of F , $\pi(a, u) = [u_0, \dots, u_l]$, $\pi(a, w) = [w_0, \dots, w_k]$ (either of

these sublists can be empty except for a), and $u_l = u, w_k = w$. In the algorithm, we denote the apex a of the present funnel F by $APEX(F)$.

The algorithm begins by placing s and an adjacent vertex v_1 in F and setting $APEX(F) = s$. It then proceeds recursively as follows.

ALGORITHM PATH(F)

Let u and w be the first and the last elements of F , and let $a = APEX(F)$ (thus $F = \pi(a, u) \cup \pi(a, w)$). Let Δuwx be the unique triangle in the triangulation G of P that has $\overline{uw}$ as an edge and that has not yet been processed (see Figure 4.4).

- (a) Search F for an element v for which $\overline{vx}$ is a tangent to F at v (if the straight line segment $\overline{ax}$ does not intersect F then $v = a$). The search is very similar to that for finding a tangent to a convex polygon from an exterior point (as described in [PS85b, page 115]). If we move a point v^* from one end of the funnel to the other, the difference between the slope of the funnel at v^* and the slope of $\overline{v^*x}$ varies monotonically (we must choose the reference direction so the funnel slope is a continuous function of v^*). We use binary search to find the zero point of the difference: each (unsuccessful) "comparison" performed at some node v^* during this search determines a unique side of v^* on which the desired v lies. We split $F \cup \{x\}$ into two new funnels $F_1 = [u, \dots, v, x]$ and $F_2 = [x, v, \dots, w]$. If v belongs to $\pi(a, u)$ then we set $APEX(F_1) := v, APEX(F_2) := a$. If, on the other hand, v belongs to $\pi(a, w)$ then $APEX(F_1) := a, APEX(F_2) := v$.
- (b) Set $\pi(s, x) := \pi(s, v) \cup \overline{vx}$. (Actually we just store a back pointer from x to v . The collection of all these pointers will constitute the required shortest path tree $SPT(P, s)$.)
- (c) If the line segment $\overline{ux}$ is a diagonal of P then we call $PATH(F_1)$ recursively.
- (d) If the line segment $\overline{wx}$ is a diagonal of P then we call $PATH(F_2)$ recursively.

The list representing the present funnel F is stored in a *finger search tree* (see [GMPR77, HM82, TV86]). This structure is essentially a search tree equipped

with *fingers* (which, in our application, are always placed at the first node and at the last node of the tree in symmetric order). To facilitate the search for the vertex v at which the tangent from x touches F , we store with each item v of F appearing in the search tree two pointers, to its predecessor and successor in F . This enables us to calculate the slopes of the two edges of F incident to v in constant time, and by comparing these slopes with that of $\overline{vx}$ we can tell in constant time on which side of v the binary search should continue. This structure therefore supports searching for a tangent from a point x in time $O(\log \delta)$, where δ is the distance from the point of tangency to the nearest finger; it also supports operations that split the tree into two subtrees at an item v in amortized time $O(\log \delta)$, with δ as above [TV86].

The correctness of our algorithm is a direct consequence of the correctness of the algorithm of Lee and Preparata [LP84].

The main difference between our algorithm and that of Lee and Preparata is in the techniques for representation, searching, and splitting of funnels. In [LP84] the search for the vertex v is a linear search starting at one designated endpoint of F . This is sufficient to guarantee the linearity of their procedure since in their case the vertices of F that are scanned during the search for v are no longer required, as the algorithm always continues with only one of the funnels F_1 or F_2 (depending on whether the next diagonal to be crossed is $\overline{xw}$ or $\overline{xu}$). In our case, however, the algorithm may have to continue recursively with both funnels, and thus it requires a funnel searching and splitting strategy that uses finger search trees (and is thus subtler than the simple linear list representation used in [LP84]), in order to obtain the desired linear time complexity.

To bound the time required by the algorithm we argue as follows. Let T be the dual tree of the triangulation of P . It is easily seen that T has $n - 2$ nodes. Without loss of generality, suppose s lies in just one triangle τ_0 of T , which we take to be the root of T . (If s is a vertex of P that lies in several triangles of the given triangulation, then at least one of them will be bounded by an edge of P incident to s , and we can start the algorithm from that triangle. The algorithm will then correctly propagate the funnel structure to all the other triangles containing s ; the funnels for the triangle edges containing s are all trivial.) Thus each node of T

(including the root) has 0, 1, or 2 children. Our algorithm is a depth-first traversal of T . With each node ζ of T we associate the parameter m_ζ , denoting the size (number of edges) of the funnel F just before ζ is processed. For example, $m_{\tau_0} = 1$.

When our algorithm processes the node ζ of T , it splits its funnel into two parts and then appends a new edge to both parts to form the funnels of the children ζ_1, ζ_2 of ζ in T . If the split parts of the funnel of ζ contain m_1 and $m_2 = m_\zeta - m_1$ edges respectively, then $m_{\zeta_1} = m_1 + 1, m_{\zeta_2} = m_2 + 1$, and the (amortized) cost of processing ζ using finger search trees is $K(\zeta) = O(\min(\log m_{\zeta_1}, \log m_{\zeta_2}))$. The complexity of our algorithm essentially depends only on the growth of the function m_ζ over the nodes $\zeta \in T$. If a node ζ has just one child, ζ' , then $m_{\zeta'}$ is at most $m_\zeta + 1$; if ζ has two children, then m_ζ is split into two parts, and each child inherits one part plus one.

We begin our analysis with the following observation: at nodes $\zeta \in T$ that have just one child or are leaves, the "direct costs" $K(\zeta)$ sum up to at most $O(n)$. Indeed, suppose $\zeta \in T$ has just one child ζ' , and that the funnel at ζ has been split into two parts having m' and $m_\zeta - m'$ edges respectively. Then the vertices of P lying in one of these parts will never be encountered again by the algorithm (by the same reasoning used in [LP84] to justify the linearity of their procedure). The number of such vertices is at least $\min(m', m_\zeta - m') - 1 \geq K(\zeta) - 2$. Thus the sum of all these $K(\zeta)$'s is proportional to at most $n + 2|T| \leq 3n$, as claimed.

Next consider the total direct costs at nodes having two children. We claim that it is sufficient to consider only cases in which m_ζ grows by exactly one at each node ζ of T having a single child, because these cases provide maximum growth of the function m down the tree T . Under this additional assumption, we have the following lemma:

Lemma 4.1. *For a node $\zeta \in T$, let the leaves of the subtree T_ζ of T rooted at ζ be $\eta_1, \dots, \eta_k$, and set $M_\zeta = \sum_{j=1}^k m_{\eta_j}$. We then have*

$$M_\zeta = m_\zeta + |T_\zeta|,$$

where $|T_\zeta|$ is the number of edges in T_ζ .

Proof: The lemma follows from summing the equations

$$\sum_{j=1}^t m_{\xi_j} = m_{\xi} + t$$

for each internal node ξ of T_{ζ} , where $\{\xi_j\}_{j=1}^t$ are the children of ξ ($t = 1$ or 2). ■

Corollary 4.1. *In particular, we have $m_{\zeta} \leq M_{\zeta}$.*

For each $\zeta \in T$ let $C(\zeta)$ denote the total cost of processing nodes with two children in the subtree of T rooted at ζ . Then

$$C(\zeta) = \begin{cases} 0 & \text{if } \zeta \text{ is a leaf} \\ C(\zeta') & \text{if } \zeta \text{ has just one child } \zeta' \\ C(\zeta_1) + C(\zeta_2) + O(\min(\log m_{\zeta_1}, \log m_{\zeta_2})) & \text{if } \zeta \text{ has two children } \zeta_1, \zeta_2. \end{cases}$$

In solving these recurrence formulas, we can assume without loss of generality that each node in T is either a leaf or has two children. Moreover, replacing m_{ζ} by M_{ζ} (using the preceding corollary) in these formulas, we obtain the recurrence formula

$$C(\zeta) = \begin{cases} 0 & \text{if } \zeta \text{ is a leaf} \\ C(\zeta_1) + C(\zeta_2) + O(\min(\log M_{\zeta_1}, \log M_{\zeta_2})) & \text{if } \zeta \text{ has two children } \zeta_1, \zeta_2. \end{cases}$$

But $M_{\zeta} = M_{\zeta_1} + M_{\zeta_2}$, if ζ has children ζ_1, ζ_2 , and $M_{\zeta} \geq 1$ for all nodes ζ . Hence if $C^*(k)$ is the maximum cost $C(\zeta)$ for any node ζ with $M_{\zeta} = k$, then we obtain the formula

$$C^*(m) = \max_{1 \leq k \leq m-1} \{C^*(k) + C^*(m-k) + O(\min(\log k, \log(m-k)))\},$$

whose solution is $C^*(m) = O(m)$ (see [Meh84, page 185]). Finally, by Lemma 4.1 we have for the root τ_0 of T

$$M_{\tau_0} = m_{\tau_0} + |T| = n - 2.$$

Thus the total complexity of the algorithm is

$$O(n) + O(M_{\tau_0}) = O(n).$$

Summarizing our analysis, we obtain the following theorem:

Theorem 4.1. *The shortest paths inside a triangulated simple polygon P from a fixed source vertex to all the other vertices of P can be calculated in linear time.*

Remark: As finger trees are complicated to implement, we could have obtained a simpler but less efficient version of the algorithm by maintaining funnels simply as doubly-linked linear lists (the same data structure as that used in [LP84]), and by performing each search through a funnel in a linear manner, starting simultaneously from both ends of the funnel. The complexity analysis of this modified procedure is almost identical to that given above, except that the direct costs at each triangle processed are now linear, rather than logarithmic, in the smaller of the subfunnel sizes. This leads to a recurrence formula for C^* of the form

$$C^*(m) = \max_{1 \leq k \leq m-1} (C^*(k) + C^*(m-k) + O(\min(k, m-k))),$$

whose solution is $C^*(n) = O(n \log n)$ (cf. [GK82, pp. 25–27] or [Knu72]).

Remark: Another possibility is to represent the funnels by self-adjusting search trees [ST85]. A weak form of the dynamic optimality conjecture of Sleator and Tarjan for such trees (see [ST85]) suggests that the resulting shortest path algorithm runs in linear time, although we do not know how to prove this.

Remark: If the source s is not a vertex of P , we can modify the algorithm as follows. Suppose s is internal to a single triangle $\Delta = \Delta uvw$ of G . Then we split Δ into three subtriangles Δsuv , Δsvw , Δswu , each having s as a vertex, and repeat the algorithm three times, each time starting at one of these triangles and propagating the funnel structure only through the edge of that triangle which is also an edge of Δ . Similar problem splitting can be employed when s lies on an edge of one or two of the triangles in G . It is easily checked that the modified algorithm produces the desired shortest path tree in linear time.

4.1.1 The convex rope algorithm

As an immediate and relatively simple application of the shortest path algorithm just presented, we consider the *convex rope problem*, posed by Peshkin and Sander-son [PS85a]: Let P be a simple polygon, let s be a vertex of P lying on the convex hull of P , and let v be another vertex of P . The *clockwise convex rope* from s to v is the shortest polygonal path ($s = p_0, \dots, p_m = v$) starting at s and ending at v that does not enter the interior of P and that is clockwise convex, in the sense that the clockwise angle from the directed segment $\overrightarrow{p_{i-1}p_i}$ to the directed segment $\overrightarrow{p_i p_{i+1}}$ is less than π , for $i = 1, \dots, m - 1$. The counterclockwise convex rope from s to v is defined in a symmetric manner (see Figure 4.5). Not all the vertices of P need have convex ropes from s . The vertices v that have both clockwise and counterclockwise ropes from any hull vertex s are precisely those that are “visible from infinity” (see [PS85a]); calculation of these convex ropes is required in [PS85a] to plan reachable grasps of P by a simple robot arm. In [PS85a], an $O(n^2)$ algorithm is presented. Using the shortest path algorithm given above, we obtain an improved algorithm running in linear time plus the time for triangulation.

The convex rope problem can be solved as follows. First compute the convex hull of P in linear time (cf. [PS85a, GY83]). The clockwise (respectively counterclockwise) convex rope from s to any vertex v on the convex hull can now be calculated by moving along the convex hull in a clockwise (respectively counterclockwise) direction from s to v . For each vertex v not on the convex hull, v lies inside a simple polygon Q (a “bay” of P) bounded by some subsequence of the sides of P and by an edge of the convex hull that is not a side of P (see Figure 4.5; note also that the collection of all these bays can be found in linear time).

Let v_1, v_2 be the endpoints of this edge such that v_1 is reached first from s when moving along the convex hull in a clockwise direction. The clockwise (respectively counterclockwise) convex rope from s to v is then the clockwise convex rope from s to v_1 (respectively the counterclockwise convex rope from s to v_2) followed by the shortest path from v_1 to v (respectively from v_2 to v) within Q , provided that this shortest path is clockwise (respectively counterclockwise) convex. (Otherwise the

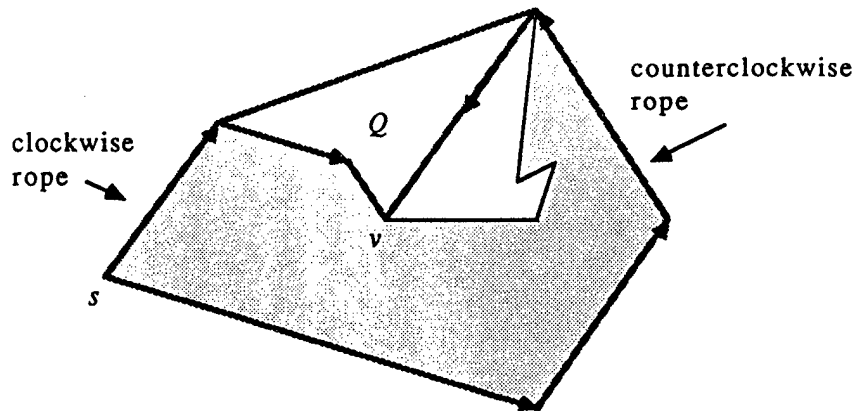


Figure 4.5. The convex rope problem

required convex rope does not exist.) Hence we can use the shortest path tree algorithm presented above to calculate the shortest paths from v_1 and from v_2 to all the vertices of Q (and also to check whether these paths are convex in the required directions) in time $O(|Q| \log \log |Q|)$. Since the sum of the sizes of all the “bays” Q of P is $O(n)$, it follows that we can solve the convex rope problem in $O(n \log \log n)$ time.

4.2 The shortest path map

This section defines the shortest path map, which is a representation of all shortest paths from a source vertex s to points inside P . The shortest path map partitions the interior of P into disjoint triangular regions, each with a distinguished vertex called the apex. The regions are chosen such that the shortest path from s to a point v passes through the apex of the region containing v .

We produce the shortest path map by partitioning the funnels defined in Section 4.1. The following lemma proves some properties of funnels necessary for the construction.

Lemma 4.2. *For each edge e of P , let $\Phi(e)$ denote the region bounded by e and by the funnel F_e . Then*

- (a) Let x be a point inside such a region $\Phi(e)$. Let v be a vertex in the corresponding funnel such that $\overline{vx}$ is tangent to the funnel (in the terminology of the algorithm of Section 4.1). Then the shortest path from s to x is the concatenation of the shortest path from s to v with the segment $\overline{vx}$.
- (b) The interiors of the regions $\Phi(e)$ are all disjoint, and the union of these regions is the entire polygon P . The total number of edges along their boundaries is $O(n)$.

Proof: The first part of the lemma follows by the same argument (taken from [LP84]) used to justify the correctness of our algorithm. As to the second part, note first that if $\Phi(e_1)$ and $\Phi(e_2)$ had a point x in common, then x would necessarily have two distinct shortest paths reaching it from s (one for each region Φ containing x), which is impossible for a simple polygon. As to the second claim, let x be any point inside P , and let Δ be the triangle in the triangulation of P that contains x . It is clear that after the above algorithm processes Δ , it will have produced some funnel F_e , where e is one of the sides of Δ , such that x lies between F_e and e . If e is an edge of P the claim is immediate; otherwise, follow the recursion of our algorithm from e further on. Each of these recursive steps takes a region bounded between some funnel F_i and its associated diagonal t , splits it into two subregions, and adds a portion of the current triangle to each of these subregions. If the input region for such a step contains x , one of the output regions must also contain x . Thus eventually our algorithm will have reached an edge e for which $\Phi(e)$ contains x . Finally, because $SPT(P, s)$ is a tree on n nodes, it has $n - 1$ edges. The regions $\Phi(e)$ are bounded by these $n - 1$ edges plus the n edges of P , which proves the last claim of (b). ■

To produce the shortest path map from the shortest path tree, we partition each funnel with a non-empty interior. Let e be an edge of P , and let $\Phi(e)$ be the corresponding region of P as defined in Lemma 4.2. Assume that the funnel F_e has the form $[u_l, u_{l-1}, \dots, u_1, a, w_1, \dots, w_k]$ with $a = u_0 = w_0$ as its apex (thus

$e = \overline{u_l w_k}$. Then, for each $i = 0, \dots, l-1$ (respectively for each $i = 0, \dots, k-1$), the ray emanating from u_i (respectively from w_i) and passing through u_{i+1} (respectively through w_{i+1}) hits e , and its portion between e and u_i (respectively w_i) is fully contained in $\Phi(e)$. These rays partition $\Phi(e)$ into $k + l - 1$ disjoint triangles, such that each triangle has two vertices lying on e , and its third vertex (called its apex) belongs to the funnel F_e . (See Figure 4.6.) The side of the triangle opposite its apex is its base, a subsegment of e . It follows immediately from Lemma 4.2(a) that if $x \in \Phi(e)$ belongs to the triangle with apex q , then $\overline{qx}$ is tangent to the funnel at q , and thus the shortest path from s to x is the concatenation of $\pi(s, q)$ and the segment $\overline{qx}$.

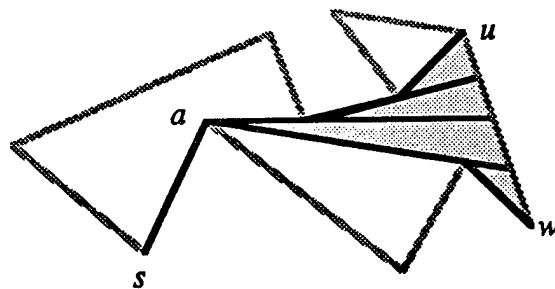


Figure 4.6. Each funnel region is broken into triangles by extending the edges on the sides of the funnel.

The collection of all triangles obtained this way for all regions $\Phi(e)$ yields a partitioning of P into disjoint triangles. This subdivision of the polygon interior is the shortest path map; we denote it by $SPM(P, s)$. Because no edge of $SPM(P, s)$ lies on the boundary of more than two regions $\Phi(e)$, Lemma 4.2(b) implies that the shortest path map has $O(n)$ triangles bounded by a total of $O(n)$ edges.¹ Although the shortest path map partitions the interior of P into triangles, it is generally not a triangulation. Polygon vertices may lie in the interior of edges of the triangular regions. Furthermore, although the apex of each triangular region is a vertex of P ,

¹By an argument that relates the number of single-edge funnels in the shortest path tree to the number of vertices on the sides of funnels, we can show that the shortest path map has at most $3n - 6$ edges. This bound is tight.

the other two vertices of the triangle need not be polygon vertices. They may instead be intersections of the funnel base with extended edges from the funnel sides.

We can use the shortest path map to answer shortest path queries for arbitrary points in P . To do so, we use one of the standard linear-time algorithms to preprocess $SPM(P, s)$ into a data structure that supports $O(\log n)$ -time point location queries [Kir83, EGS86]. Given a query point x in P , we can find the triangle or edge of $SPM(P, s)$ that contains it in $O(\log n)$ time, and thereby find the last vertex of P on the shortest path from s to x , the apex q of the containing triangle. If we store at each apex q the length of the shortest path from s to q , we can then calculate the length of the shortest path from s to x in $O(1)$ additional time; the path itself can be calculated in $O(1)$ additional time per segment on the path by simply traversing the path in the shortest path tree from q to s . Thus we have the following result:

Theorem 4.2. *Given a triangulated simple polygon P with n sides and some source point s within P , one can preprocess P in linear time so that, for any target point x in P , the length of the shortest path from s to x can be calculated in $O(\log n)$ time, and the path itself can be extracted in time $O(\log n + k)$, where k is the number of segments from which this path is composed.*

If we want to extract the path in time proportional to the number of turns it makes (rather than the number of segments along it), we change the shortest path tree pointers so that each vertex v stores a pointer to the vertex x on $\pi(s, v)$ farthest from v such that all vertices on $\pi(x, v)$ lie on the segment $\overline{xv}$.

Remark: Related work on shortest paths inside a simple polygon can be found in [El 85].

4.3 Visibility within a simple polygon

In this section we study a collection of problems involving visibility within a simple polygon. These problems have been studied in recent papers by several authors

[EA81, Lee83, AT81, CG85, El 84, LL84, Asa84], and various algorithms have been developed to solve them. Some of the simpler problems already have linear-time solutions, whereas others only have $O(n \log n)$ solutions. Here we present linear-time solutions for all these problems, again based on the availability of a triangulation of P . Our approach relies on an interesting relationship between visibility and shortest path problems.

Let P be a triangulated simple polygon with n sides. The problems studied in this section and solved in linear time are

- I. Given a point x inside P , calculate the *visibility polygon* $Vis(P, x)$ consisting of all points $y \in P$ that are visible from x (that is, such that the open segment $\overline{xy}$ lies in the interior of P). (This problem is simpler than the subsequent ones, and there exist known linear-time algorithms for it [EA81, Lee83].)
- II. Given a segment e inside P , calculate the (weak) visibility polygon $Vis(P, e)$ consisting of all $y \in P$ that are visible from some point on e . (An $O(n \log n)$ solution is given in [CG85]; Avis and Toussaint [AT81] present a linear time algorithm for determining whether $Vis(P, e) = P$.)
- III. Given a segment e inside P , preprocess P so that for each query ray r emanating from some point on e into P , the first intersection of r with the boundary of P can be calculated in $O(\log n)$ time. (An $O(n \log n)$ solution is given in [CG85].)
- IV. Given a segment e inside P , preprocess P so that for each query point $x \in P$, the subsegment of e visible from x can be calculated in $O(\log n)$ time. (An $O(n \log n)$ solution is given in [CG85].)

We shall first consider Problem I; although this problem already has linear time solutions, it is worthwhile to sketch our solution as a preparatory step toward the solution of the more complicated problems II–IV. It is well known that the boundary of $Vis(P, x)$ is a simple polygon; each of its vertices is either a vertex of P that is visible from x or a “shadow” cast on the boundary of P by such a visible vertex. A shadow vertex y terminates the visible portion of a partially visible polygon

segment; that is, y is one endpoint of a subsegment whose entire interior is visible from x . The segment $\overline{xy}$ passes through a vertex of P ; see Figure 4.7.

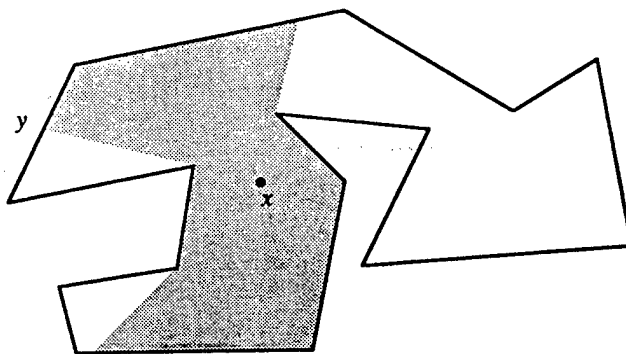


Figure 4.7. Visibility of a polygon from a point

Suppose without loss of generality that x is a vertex of P (if not, it is easy to construct in linear time a triangulation of the interior of P in which x is also a vertex of some triangle). Calculate the shortest path map $SPM(P, x)$ from x using the algorithm given in Section 4.2. In $SPM(P, x)$, the triangular regions with x as their apex include all the points inside P that are visible from x . The polygon vertices connected to x by edges of $SPM(P, x)$ include all the vertices visible from x . (Two of the neighbors of x in $SPM(P, x)$ are not visible from x : edges of $SPM(P, x)$ connect x to its neighbors on the boundary of P , but our definition of visibility does not count these as visible points.) To calculate $Vis(P, x)$, we concatenate in clockwise order the regions of $SPM(P, x)$ with x as their apex, thereby obtaining a single simple polygon that is the boundary of $Vis(P, x)$. Thus we have the following theorem:

Theorem 4.3. *The visibility polygon $Vis(P, x)$ can be calculated in linear time, given a triangulation of P .*

Next we consider Problem II. Let a, b be the endpoints of e . It is easily checked that in this case $Vis(P, e)$ is a simple polygon each of whose vertices is either

- (i) a vertex of P visible from e ; or

- (ii) a shadow cast on the boundary of P by a ray that emanates from a or from b and passes through a vertex of P visible from that endpoint; or
- (iii) a shadow cast by a ray r that emanates from some interior point on e and passes through two vertices x, y of P , such that the exterior of P lies on one side of r in the vicinity of x and on the other side of r in the vicinity of y .

We characterize $\text{Vis}(P, e)$ further in Section 5.1. See Figure 4.8 for an illustration of $\text{Vis}(P, e)$.

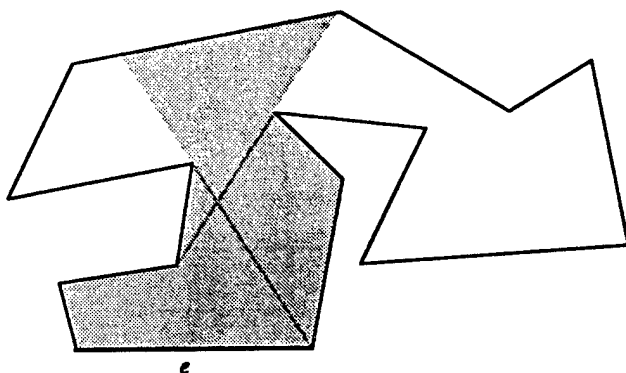


Figure 4.8. Visibility of a polygon from an edge

Let $e' = \overline{cd}$ be another edge of P . We say that the shortest path $\pi(a, c)$ is *inward convex* if, as in the case of funnels, the convex angles formed by successive segments of this path with the directed line $\overrightarrow{ab}$ keep increasing. A symmetric definition of inward convexity applies to $\pi(b, d)$.

Lemma 4.3. *If e' contains a point in its interior that is visible from e then (up to exchanging c and d) the two paths $\pi(a, c)$ and $\pi(b, d)$ are inward convex.*

Proof: Let $x \in e'$ be visible from some point z on e . Suppose without loss of generality that c is that endpoint of e' for which a and c lie on the same side of the line $\overleftrightarrow{xz}$. Then the shortest path $\pi(a, c)$ must lie entirely on one side of $\overleftrightarrow{xz}$, and indeed it does not cross the polygonal path $azxc$. Since the area R between $azxc$ and $\pi(a, c)$ is fully contained in P , it

follows that $\pi(a, c)$ must be inward convex, or else we could shortcut it by a segment contained in R , thus also in P . The claim concerning $\pi(b, d)$ follows by a symmetric argument. ■

In the situation described by the preceding lemma, we call the union of $\pi(a, c)$ and $\pi(b, d)$ the *hourglass* for the pair (e, e') .

Suppose we apply the shortest-path algorithm of Section 4.1 to the two source vertices a and b , and also compute, for each vertex c of P , whether the path $\pi(a, c)$ (respectively $\pi(b, c)$) is inward convex. (The latter calculations take $O(1)$ time per vertex.) Let $e' = \overline{cd}$ be another edge of P . If the two paths $\pi(a, c)$ and $\pi(b, d)$ are not both inward convex, and also the two paths $\pi(a, d)$ and $\pi(b, c)$ are not both inward convex, then by Lemma 4.3, e' is not visible from e . Thus suppose (without loss of generality) that the two paths $\pi(a, c)$ and $\pi(b, d)$ are inward convex. It is easy to see that the shortest path $\pi(a, d)$ must be the concatenation of three subpaths: a subpath $\pi(a, x)$ of $\pi(a, c)$, where x is a point lying on $\pi(a, c)$, a line segment $\overline{xy}$, where y is a point lying on $\pi(b, d)$, and the subpath $\pi(y, d)$ of $\pi(b, d)$. Moreover $\overline{xy}$ must be a common tangent to both of the paths $\pi(a, c)$ and $\pi(b, d)$ (see Figure 4.9). The path $\pi(b, c)$ has a symmetric structure of the form $\pi(b, w) \parallel \overline{wz} \parallel \pi(z, c)$ (where $\parallel$ denotes path concatenation), for appropriate points $w \in \pi(b, d)$, $z \in \pi(a, c)$. It follows that the subsegment of e' visible from e is that delimited by the intersections of e' with the two lines $\overline{xy}$ and $\overline{wz}$. (If $\overline{xy}$ and $\overline{wz}$ intersect e' in the same point, then no point on e' is visible from e .) Note also that, in the terminology of Section 4.1, when the shortest-path algorithm is run with a as the source, the funnel $F_{e'}$ associated with the segment e' has x as its apex, and y as an adjacent vertex. Similarly, when the algorithm runs with b as the source, w is the apex of the funnel $F_{e'}$ and z is an adjacent vertex in that funnel.

These observations suggest a straightforward method for calculating the points x, y, w and z . Namely, as we execute the shortest path algorithm with a as a source, and we reach an edge $e' = \overline{cd}$ of P for which the path $\pi(a, c)$ is inward convex, we take x to be the apex of the current funnel, and y to be the next vertex on the part of the funnel between x and d . The points w and z are found in a completely symmetric manner when running the shortest path algorithm with b as a source. To calculate

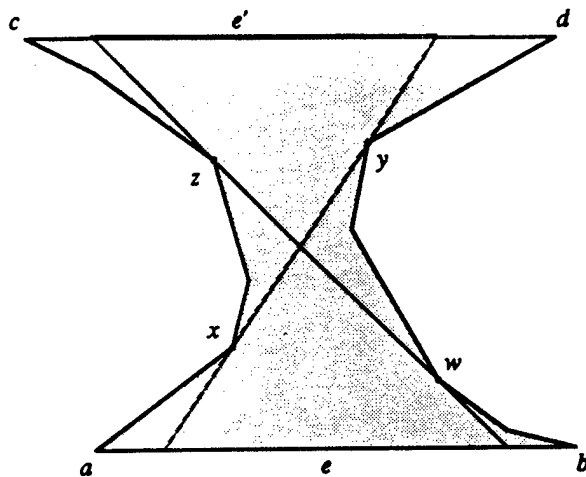


Figure 4.9. Visibility of one edge of a polygon from another, and the corresponding hourglass

$Vis(P, e)$ we simply traverse the boundary of P , say in clockwise order, collecting all visible subsegments along this boundary and replacing contiguous non-visible portions of the boundary by straight segments connecting visible vertices with their shadows. Thus we have the following theorem:

Theorem 4.4. *The visibility polygon $Vis(P, e)$ of P with respect to an edge e can be calculated in linear time (assuming a triangulation of P is given).*

Remark: Toussaint has independently obtained a similar connection between shortest paths and visibility inside a simple polygon, as well as some of the technical tools developed above [Tou85].

Next let us consider Problem IV. Let $x \in P$ be an arbitrary point that is visible from some point $z \in e$. An immediate generalization of Lemma 4.3 implies that both of the paths $\pi(a, x)$ and $\pi(b, x)$ must be inward convex. A slight variant of the converse statement is also true, as is easily checked: if both $\pi(a, x)$ and $\pi(b, x)$ are inward convex and intersect only at x , then x is visible from e . Moreover, let the last straight segment in $\pi(a, x)$ (respectively in $\pi(b, x)$) be $\overline{cx}$ (respectively $\overline{dx}$) for some vertex c (respectively d) of P . Then the portion of e visible from x is delimited

by the intersections of the rays $\overrightarrow{xc}$ and $\overrightarrow{xd}$ with e ; if the intersections coincide, e is not visible from x .

Thus, to preprocess P as required in Problem IV, we execute algorithm of Section 4.2 twice, once with a as a source, and once with b as a source. This yields two shortest path maps, $SPM(P, a)$ and $SPM(P, b)$, which we then further preprocess into two data structures that support $O(\log n)$ point location queries. This can be done in linear time, using the techniques in [Kir83] or [EGS86]. In addition, we store at each vertex c of P flags indicating whether the paths $\pi(a, c)$ and $\pi(b, c)$ are inward convex, and (as usual) also pointers to the two parents of c in the two shortest-path trees produced by the two runs of the algorithm.

Now let $x \in P$ be a given query point. First we locate x in the two subdivisions $SPM(P, a)$ and $SPM(P, b)$ and obtain the apexes of the two enclosing triangles, vertices c and d of P . Then, in additional constant time, we can test whether the paths $\pi(a, x) = \pi(a, c) \parallel \overrightarrow{cx}$, $\pi(b, x) = \pi(b, d) \parallel \overrightarrow{dx}$ are both inward convex, and if so, find the intersections of the rays $\overrightarrow{xc}$ and $\overrightarrow{xd}$ with e , thereby obtaining the desired subsegment of e that is visible from x .

Remark: Problem IV is more general than the corresponding problem P3 studied in [CG85], in that here the query point x can be any point inside P , whereas in [CG85] it is required to lie on the boundary of P .

Finally, we consider Problem III. Here we make use of the duality between rays emanating from e and points in the two-sided plane (2SP for short), as described in [GRS83, CG85]. Following [CG85], we will produce a partitioning Π of the 2SP into convex regions, each region containing the duals of all rays emanating from e and hitting the same edge of P . The same partitioning is obtained in [CG85] in $O(n \log n)$ time; we show here how to obtain it in linear time.

To this end we make use of the analysis of Problem II given above. Let $e' = \overline{cd}$ be another edge of P that contains points visible from e . As above, we know that the two paths $\pi(a, c)$, $\pi(b, d)$ are inward convex. Let $\overrightarrow{xy}$, $\overrightarrow{wz}$ be the two common tangents to these paths, where $x, z \in \pi(a, c)$ and $w, y \in \pi(b, d)$. Let $R(e')$ be the region in Π corresponding to e' . Then the boundary of $R(e')$ consists of points that are duals of rays r emanating from e and hitting points on e' , such that r passes

through a vertex of P (which can be either one of the endpoints a, b, c, d , or another vertex of P that r “grazes” on its way from e to e'). It follows from the preceding analysis that such a vertex must lie on one of the paths $\pi(a, c)$, $\pi(b, d)$, or, more precisely, on one of their subpaths $\pi(x, z)$, $\pi(w, y)$.

Moreover, each vertex of $R(e')$ corresponds either to a ray that passes through two vertices of P that are adjacent in one of the subpaths $\pi(x, z)$, $\pi(w, y)$, or to one of the two extreme rays $\overline{xy}$, $\overline{wz}$. It is also easy to establish adjacency of the vertices of $R(e')$ along its boundary. Specifically, two such vertices must correspond to two rays passing respectively through $\overline{fg}$ and $\overline{gh}$, where f, g, h are three vertices of P that are either adjacent along one of the paths $\pi(x, z)$, $\pi(w, y)$, or are such that g is one of w, x, y , or z , f is adjacent to g along $\overline{xy}$ or $\overline{wz}$, and h is adjacent to g along $\pi(x, z)$ or $\pi(w, y)$.

The preceding arguments imply that the total number of vertices in Π is at most proportional to the sum of the sizes of the funnels for the edges of P that are obtained during execution of the shortest path algorithm of Section 4.1. This sum is linear in n , which implies that Π has only $O(n)$ vertices (see also [CG85]).

It is also easy to calculate adjacency of regions in Π . Specifically, it suffices to consider adjacency of regions near a vertex τ of Π . By the preceding analysis, τ is the dual of a ray r emanating from e and passing through two vertices g, h of P . We can apply a simple local, though somewhat lengthy, case analysis (which requires only $O(1)$ time), to enumerate all possible pairs of edges e', e'' whose regions $R(e')$, $R(e'')$ in Π are adjacent near τ ; generally, each of these edges will either lie adjacent to g or h , or contain one of the endpoints of r , if this endpoint is different from g and h . We leave details of this case analysis to the reader.

All these observations imply that Π can be calculated in linear time from the output of the executions of the shortest path algorithm of Section 4.1 with a and b as sources. Having calculated Π , we next apply to it one of the linear-time preprocessing algorithms for point location [Kir83, EGS86], obtaining a data structure from which the region of Π containing (the dual of) any query ray r , and thus also the edge of P first hit by r , can be found in $O(\log n)$ time.

Remark: The case analysis in the construction of Π can be avoided if we work

in dual space from the beginning, as in [CG85]. The rays passing through edge e dualize to a convex region of the two-sided plane. We explore the triangulation one triangle at a time, as in Section 4.1; instead of splitting funnels, we subdivide convex polygonal regions in dual space. Finger search trees can be used as in Section 4.1 to make the overall construction time linear.

4.4 Conclusion

This chapter has presented a collection of linear-time algorithms for solving a variety of shortest path and visibility problems inside a triangulated simple polygon, exploiting interesting relationships between these two types of problems. All the algorithms are based on a single structure, the shortest path tree. One of the most useful derivatives of the shortest path tree is the shortest path map, which represents all the shortest paths from a fixed source vertex to other points in P . In the following chapter we use the shortest path map as a tool to build the visibility graph of a simple polygon.

Chapter 5

Finding the Visibility Graph of a Simple Polygon in Time Proportional to its Size

*Write the vision, and make it plain upon tables,
that he may run that readeth it.*

— Habakkuk 2:2

As described in Chapter 2, the visibility graph of a set S of n non-intersecting line segments in the plane records each pair of segment endpoints that can be connected by an open segment free of intersections with S . If we regard the segments as opaque walls, the visibility graph records, for every wall endpoint v , what other wall endpoints would be seen by an observer standing at v . As its name suggests, the visibility graph is a combinatorial graph structure; its nodes are the segment endpoints, and its edges connect mutually visible segment endpoints.

Although the visibility graph algorithms of Chapter 2 find the visibility graph in

$\Theta(n^2)$ time, the visibility graph itself may be as small as $O(n)$. This seems to suggest some inefficiency in the algorithms; a linear amount of work is being expended to find each visibility graph edge. It would be desirable to have an algorithm whose running time depended on the size of its output.

It is possible to improve on the $\Theta(n^2)$ algorithms when S has some special property. For example, when the segments in S form a collection of k convex polygons, it is not hard to find the visibility graph in time proportional to the output size plus $O(nk \log n)$. This gives an $\Omega(n^{3/2} \log^{1/2} n)$ best-case algorithm.

In this chapter we consider the case in which the segments form a simple polygon with no consecutive collinear edges. By convention, the visibility graph of a simple polygon P contains only the visibility edges inside P . If we need the complete visibility graph, we can construct it by computing the convex hull of P , then finding the visibility graph inside P and inside each of the bays cut off by non-polygon edges of the hull. We define m to be the number of visibility edges inside P . Because P can be triangulated, m is at least $n - 3$.

A recent algorithm of Suri finds the visibility graph of a simple polygon in time $O(m \log n)$ [Sur86b]. The method uses $O(m)$ “shooting” queries: given a direction d and a point q inside P , each query asks for the first point of P intersected by the ray from q with direction d . Each query takes $O(\log n)$ time to answer [CG85].

This chapter shows how to find the visibility graph of a simple polygon P in time $O(m + n \log \log n)$. The algorithm exploits the close relationship between the visibility graph and shortest paths inside P : shortest paths follow visibility edges, and visibility edges are shortest paths between the points they connect. The algorithm computes all the shortest paths from each polygon vertex to other vertices, then picks out the single-edge paths as visibility edges.

At first glance, this approach seems extraordinarily inefficient. Why should we compute shortest paths with many turns when only the single-edge paths are important? The answer is that the set of all shortest paths has more structure than the set of single-edge paths. Once we have found the set of all shortest paths from one vertex, we can exploit the structure to find similar sets for the other vertices efficiently.

For each polygon vertex v , we use the shortest path map $SPM(P, v)$ to represent the shortest paths from v to the other vertices. As noted in Section 4.3, the vertices visible from v are its neighbors in $SPM(P, v)$; given a reasonable representation of $SPM(P, v)$, we can find them in constant time apiece. If we constructed each shortest path map individually, we would spend $\Theta(n)$ time on each one, and it would take $\Theta(n^2)$ time to find the visibility graph. However, we can save time by building the shortest path map for an arbitrarily selected vertex v_1 , then transforming it to get all the other shortest path maps. Once we have $SPM(P, v_1)$, we modify it to produce $SPM(P, v_2)$, where v_2 is the counterclockwise neighbor of v_1 along the polygon boundary. We repeat this operation for all the vertices, working counterclockwise around the polygon.

We build the first shortest path map by triangulating P , then running the algorithm of the preceding chapter. The triangulation algorithm of Tarjan and Van Wyk takes $O(n \log \log n)$ time [TV86], which dominates the initialization cost.

In the rest of this chapter, we give algorithms to produce all the shortest path maps from $SPM(P, v_1)$ in $O(m)$ time. Section 5.1 reveals the similarity between shortest path maps for adjacent vertices by characterizing their differences. Section 5.2 gives two algorithms for obtaining one such shortest path map from the other in time proportional to the number of differences between them. Section 5.3 shows that the total number of differences between adjacent shortest path maps is $O(m)$, which implies that our algorithm takes $O(m + n \log \log n)$ time altogether. This is slightly suboptimal when $m = o(n \log \log n)$, but it is still a significant improvement over earlier methods.

5.1 Shortest path maps

Shortest path maps were introduced in Section 4.2; in this section we give additional properties of shortest path maps that we need for our visibility graph algorithm. Since our algorithm transforms one shortest path map into another, we characterize the differences between two shortest path maps whose sources are adjacent polygon vertices. Let us introduce some notation to help talk about shortest paths. We

have already defined $\pi(s, v)$ to be the unique shortest path from s to v that does not go outside P . We now pick out a key point on the path: $l(s, v)$ is the polygon vertex on $\pi(s, v)$ that is not equal to v but closest to it (l stands for the *last* vertex on the path). We assume in this chapter that s is a polygon vertex, so $l(s, v)$ is always well-defined. For all points v inside a triangular region of the shortest path map $SPM(P, v)$, $l(s, v)$ is constant and equal to the region's apex.

In the rest of this chapter we refer to shortest path maps repeatedly; since all our shortest path maps are subdivisions of P , we use the notation $SPM(s)$ in place of the notation $SPM(P, s)$. We represent $SPM(s)$ using a standard subdivision representation, such as the winged-edge data structure of Baumgart [Bau75] or the quad-edge data structure of Guibas and Stolfi [GS85]. The data structure lets us find the neighbors of s in $SPM(s)$ in constant time apiece, which gives us the visibility edges with s as an endpoint in constant time per edge. The algorithms of Section 5.2 must be able to find the base of a region in constant time, so we augment the data structure by storing at the apex of each region a pointer to its base. When the algorithms need to find the base of a region, they already know its apex.

Let s and t be adjacent polygon vertices, s clockwise of t . Given $SPM(s)$, the algorithms of the following section produce $SPM(t)$ by adding and deleting edges. The following lemma characterizes the part of the subdivision that they change.

Lemma 5.1. *Let s and t be adjacent vertices of P , and let e be an edge of $SPM(s)$ that does not belong to $SPM(t)$. Then there is a point v in the interior of e such that any neighborhood of v includes points that are visible from the interior of $\overline{st}$.*

Proof: The edge e separates two regions of $SPM(s)$, and hence $l(s, a) \neq l(s, b)$ for a and b on opposite sides of e . Because e does not belong to $SPM(t)$, we can pick a point v in the interior of e that is not part of an edge of $SPM(t)$. Then for a and b on opposite sides of e and sufficiently close to v , we have $l(t, a) = l(t, b)$. By possibly interchanging a and b , we can insure that $l(t, a) \neq l(s, a)$.

Now consider the shortest path tree of a . Since $l(t, a) \neq l(s, a)$, a

is the apex of the funnel associated with segment $\overline{st}$. The funnel is evidence that some part of the interior of $\overline{st}$ is visible from a . Any ray with source a and direction strictly in the angle formed by $l(s, a)$, a , and $l(t, a)$ must hit the interior of $\overline{st}$ before it hits any other part of P .

■

The part of P visible from at least one point in the interior of $\overline{st}$ is a connected subpolygon, the visibility polygon $Vis(P, \overline{st})$. Following our convention for shortest path maps, we drop P from the name and write $Vis(\overline{st})$ instead of $Vis(P, \overline{st})$. Lemma 5.1 shows that $SPM(s)$ and $SPM(t)$ differ only inside $Vis(\overline{st})$ and on its boundary. The following lemma characterizes the boundary of $Vis(\overline{st})$.

Lemma 5.2. *The boundary of $Vis(\overline{st})$ is made up of edges of $SPM(s)$ and $SPM(t)$. Any edge $\overline{ab}$ that is on the boundary of $Vis(\overline{st})$ but not on the boundary of P has one endpoint, say a , that lies on the paths to b from s and t : it is both $l(s, b)$ and $l(t, b)$. If a is clockwise of b on the boundary of P , then $\overline{ab}$ belongs to $SPM(s)$; otherwise, it belongs to $SPM(t)$.*

Proof: The edges of P belong to both $SPM(s)$ and $SPM(t)$; thus the polygon edges on the boundary of $Vis(\overline{st})$ belong to both shortest path maps. We therefore consider only boundary edges of $Vis(\overline{st})$ in the interior of P .

A point v in the interior of such an edge is not visible from $\overline{st}$, since it is not in the interior of $Vis(\overline{st})$. As noted in the proof of Lemma 5.1, we must have $l(s, v) = l(t, v)$. Let us define a to be $l(s, v)$. The point a is an endpoint of the edge that contains v ; let b be the other endpoint. The shortest paths from b to s and t must go through a , and so a is both $l(s, b)$ and $l(t, b)$.

Suppose that a is clockwise of b . This means that the interior of $Vis(\overline{st})$ lies to the left of $\overrightarrow{ab}$. Let v' be a point just to the left of v , inside $Vis(\overline{st})$. (See Figure 5.1.) Because v' is visible from $\overline{st}$, the two shortest paths $\pi(s, v')$ and $\pi(t, v')$ intersect only at v' . Vertex a is on or to the

right of $\pi(t, v')$, so $l(s, v') \neq a$. This holds for v' distinct from v but arbitrarily close to it. Since $l(s, v) = a$, the segment $\overline{ab}$ is an edge of $SPM(s)$; it separates two regions of $SPM(s)$.

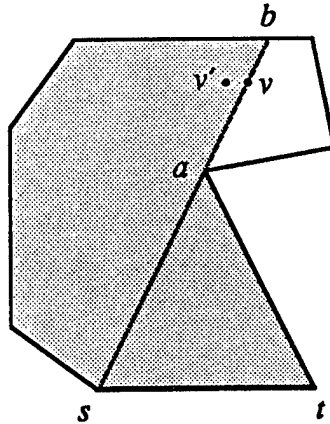


Figure 5.1. Because v' is visible from $\overline{st}$ and v is not, $l(s, v) \neq l(s, v')$. This implies that $\overline{ab}$ is an edge of $SPM(s)$.

If a is counterclockwise of b , then the interior of $Vis(\overline{st})$ lies to the right of $\overrightarrow{ab}$. An argument similar to the preceding one shows that $\overline{ab}$ is an edge of $SPM(t)$. ■

5.2 Constructing shortest path maps

This section presents two dissimilar ways to implement the general step of the algorithm, the construction of one shortest path map from that of an adjacent vertex. Both methods take time proportional to the number of differences between the two shortest path maps. Section 5.3 shows that the number of differences, summed over all neighbor pairs, is proportional to the size of the visibility graph.

5.2.1 A sweeping-path algorithm

The first transformation algorithm might be called a “sweeping-path” algorithm. It moves a point p from t to s along the boundary of the subpolygon $Vis(\overline{st})$, the region in which one shortest path map must be replaced by another. We think of the point as being connected to s and t by two rubber bands inside P , the shortest paths $\pi(s, p)$ and $\pi(t, p)$. As the point moves, it drags the two paths along with it, and the paths sweep over the subdivision inside $Vis(\overline{st})$. The algorithm uses the sweeping paths to change $SPM(s)$ into $SPM(t)$. The algorithm maintains the following invariant: on and to the right of $\pi(t, p)$, the subdivision represents $SPM(t)$; to the left of $\pi(s, p)$, the subdivision represents $SPM(s)$; between the two paths, there are no edges.

The algorithm updates the subdivision when p visits vertices of $SPM(s)$ or $SPM(t)$. If v is a vertex of $SPM(s)$, all the edges of $\pi(s, v)$ are edges of $SPM(s)$. When p visits v , the algorithm deletes the last edge of the path $\pi(s, v)$, unless it is also an edge of $SPM(t)$. (An edge may belong to both shortest path maps only if it is a boundary edge of $Vis(\overline{st})$.) Similarly, when p visits a vertex v of $SPM(t)$, the algorithm adds the last edge of $\pi(t, v)$ to the subdivision, unless it is already an edge of $SPM(s)$. This procedure considers just once each edge that must be changed.

When the algorithm adds or deletes an edge, it may have to make other changes to the subdivision. Suppose that the algorithm deletes an edge of $SPM(s)$ when p visits v . If v is a vertex of $SPM(s)$ but not of P , then v splits a polygon edge; the algorithm must merge the two pieces back into one. Similarly, when the algorithm adds an edge of $SPM(t)$ to the subdivision, it may have to split a polygon edge: if p is not at a polygon vertex, the algorithm splits the polygon edge on which p lies. Since adding and deleting edges changes regions in the current subdivision, we must update the base pointer in each changed region. This is not hard: whenever a pointer needs to be changed, the new base is incident to p , and the region’s apex is connected to p by a straight segment of $\pi(s, p)$ or $\pi(t, p)$.

How do we represent p and the two paths in a discrete algorithm? The point p does not move continuously along the boundary of $Vis(\overline{st})$, but in discrete jumps

from one vertex of $SPM(s)$ or $SPM(t)$ to the next such vertex. Because the shortest paths to intermediate vertices on $\pi(s, p)$ and $\pi(t, p)$ are unique, the movement of p changes $\pi(s, p)$ and $\pi(t, p)$ only at the ends nearer p . Thus we can represent the two paths as stacks of polygon vertices. The stacks do not include p : the top of the stack for $\pi(s, p)$ is $l(s, p)$, and the top of the stack for $\pi(t, p)$ is $l(t, p)$.

We have described the basic idea of the algorithm, as well as its data structures. We now describe the actions necessary to move p along the boundary of $Vis(\overline{st})$. These can be broken down into several interrelated tasks. The hardest two tasks are deciding which edge p should follow when it leaves a vertex and deciding when it should stop moving along the edge. By referring to the two shortest paths $\pi(s, p)$ and $\pi(t, p)$, the algorithm can make these decisions in constant time apiece. A simpler task is that of maintaining the stacks that represent the shortest paths. We show how to perform these tasks, taking into account the possibility of degeneracies in P . We then combine the costs of these operations to bound the algorithm's running time.

The region visible from the interior of $\overline{st}$, $Vis(\overline{st})$, is bounded by polygon edges and by extension edges of the types shown in Figure 5.2. (An extension edge is a shortest path map edge that does not belong to the shortest path tree. It is obtained by extending a funnel edge.) The algorithm for following the boundary of $Vis(\overline{st})$ is simple: p moves counterclockwise along the polygon boundary whenever doing so leaves both shortest paths $\pi(s, p)$ and $\pi(t, p)$ inward convex. When moving along the polygon boundary would cause one of these two paths to violate inward convexity, as at vertices v and w in Figure 5.2, p moves along an extension edge. It moves along edges of $SPM(s)$ when moving away from $\overline{st}$ and along edges of $SPM(t)$ when moving toward $\overline{st}$, as specified by Lemma 5.2. In Figure 5.2, when p advances from v , it moves along the extension of the edge from $l(s, v)$ through v . When p advances from w , it moves along the extension edge between w and $l(t, w)$.

The movement of p fits in with the edge addition and deletion scheme outlined above. Whenever p moves along an extension edge of $SPM(s)$, the edge still belongs to the subdivision. When p moves along an extension edge of $SPM(t)$, the edge has just been added to the subdivision.

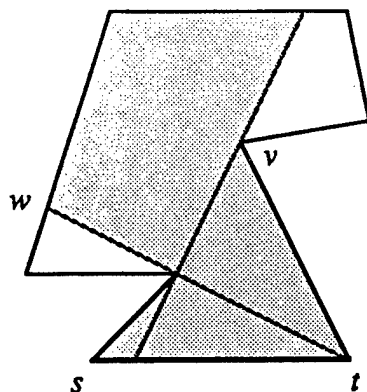


Figure 5.2. $Vis(\overline{st})$ is bounded by polygon edges and by extension edges. The left extension edge is an edge of $SPM(t)$; the right edge belongs to $SPM(s)$.

We can tell which edge p should follow when it leaves a vertex; we need to know when to stop its motion along the edge. We break the problem into two subproblems. We consider moving p along extension edges below; here we show how to move p along edges of P that are visible from $\overline{st}$. When moving along such an edge, p should stop at the first vertex of $SPM(s)$ or $SPM(t)$ on the edge. It is easy to detect vertices of $SPM(s)$: they are vertices of the subdivision left of $\pi(s, p)$. Recognizing vertices of $SPM(t)$ that are not vertices of $SPM(s)$ is more difficult.

The problem we need to solve is quite specific: given a point p on the boundary of $Vis(\overline{st})$ such that the polygon edge counterclockwise of p is visible from $\overline{st}$, find the first vertex of $SPM(s)$ or $SPM(t)$ counterclockwise of p along the edge. Let q be the first vertex of $SPM(s)$ counterclockwise of p ; it is part of the subdivision. We need to determine whether any vertices of $SPM(t)$ lie in the interior of edge $\overline{pq}$.

We begin by characterizing vertices of $SPM(t)$ that lie between p and q , assuming that at least one exists. Let v be such a vertex, chosen closest to p if there are several such vertices. Vertex v is an endpoint of an extension edge e . The edge separates two regions of $SPM(t)$; v lies on the base of both. The apexes of the two regions lie on the line containing e . Because v is not a polygon vertex, the two apexes cannot be coincident.

First consider the case in which the apex to the right of e is closer to v than the apex to the left, as in Figure 5.3(a). (We define right and left by assuming that e is directed away from t .) Because p lies on the boundary of the region on the right, the last two vertices of $\pi(t, p)$ lie on the segment $\overline{ab}$ of Figure 5.3(a). (In degenerate cases several polygon vertices may lie on $\overline{ab}$.) Hence v can be determined as the intersection of the ray through the last two vertices on $\pi(t, p)$ with the edge $\overline{pq}$.

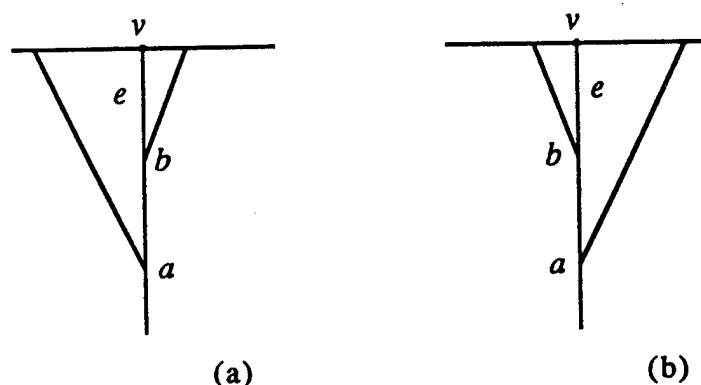


Figure 5.3. The point v is a vertex of $SPM(t)$ that lies between p and q , the vertex of $SPM(s)$ that follows p on the boundary of P . Figures (a) and (b) show the two configurations of $SPM(t)$ that give rise to such a vertex.

Now consider the case in which the apex to the left of e is closer to v than the apex to the right, as in Figure 5.3(b). Because $\pi(s, v)$ cannot lie to the left of the polygon vertex b or to the right of $\pi(t, v)$, it must go through b . Thus b is $l(s, v)$. The region of $SPM(t)$ to the right of e has a as its apex; that is, $l(t, p) = a$. Because there are no vertices of $SPM(s)$ between p and q , the last vertex on $\pi(s, p)$, $l(s, p)$, is either equal to b or to a polygon vertex in the interior of $\overline{ab}$. (Because $\overline{pq}$ is visible from $\overline{st}$, $l(s, p)$ cannot be a . It is not equal to b only in the case of degeneracies.) Hence v can be determined as the intersection of the ray through $l(t, p)$ and $l(s, p)$ with the edge $\overline{pq}$.

To detect vertices of $SPM(t)$ on the edge $\overline{pq}$, we need to compute only two intersections. If the ray through the last two vertices on $\pi(t, p)$ intersects $\overline{pq}$, or the ray through $l(t, p)$ and $l(s, p)$ intersects $\overline{pq}$, then we move p to the intersection

farthest clockwise; it is a vertex of $SPM(t)$. Otherwise we move p to q . (If $l(t, p) = t$, the first ray is undefined and its intersection does not exist. If either ray hits $\overline{pq}$, the ray farther clockwise hits $\overline{pq}$ without hitting any other part of P first; this is because the quadrilateral determined by p , q , $l(s, p)$, and $l(t, p)$ is free of polygon points.) See Figure 5.4 for an example of these intersections.

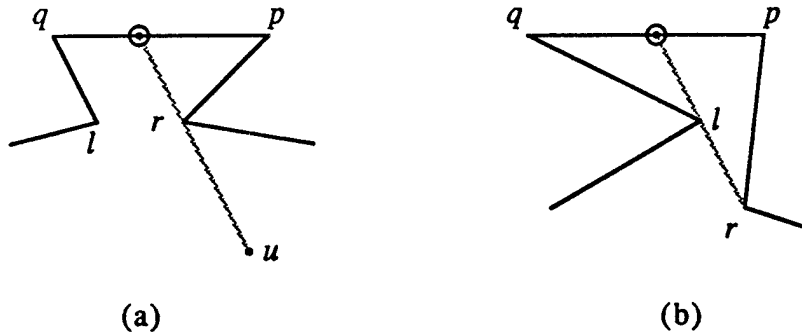


Figure 5.4. The circled points are vertices of $SPM(t)$ that do not appear in the subdivision left of $\pi(s, p)$. They are detected when the algorithm tries to move p along the polygon boundary to q . (Vertices l and r are the last polygon vertices on $\pi(s, p)$ and $\pi(t, p)$, respectively. Vertex u is the predecessor of r on $\pi(t, p)$.) In (a), the circled vertex is the intersection of $\overline{pq}$ with the extension of $\overline{ur}$. Vertex r is removed from $\pi(t, p)$. In (b), p cannot move past the intersection of $\overline{rl}$ with $\overline{pq}$ (circled) without violating the inward convexity of $\pi(t, p)$.

We have described how the algorithm moves p along an edge visible from $\overline{st}$; we now describe how it moves p along an extension edge. Extension edges require care because they may have multiple polygon vertices along them, even though no shortest path map edges need to be changed. This degenerate case is depicted in Figure 5.5. The algorithm handles extension edges specially to avoid having to visit many vertices when changing only a few shortest path map edges. It uses the base pointer stored at the apex of each region. The apex is the last vertex on $\pi(s, p)$ or $\pi(t, p)$; the algorithm follows its pointer to skip over extension edges, moving directly from the first vertex to the base, or vice versa. In Figure 5.5, p moves directly from v to u and from w to x . If there are intermediate vertices on

the extension edge (which can be checked in constant time), the edges along the extension are the same in both $SPM(s)$ and $SPM(t)$. If there are no intermediate vertices, the edge is added or deleted in constant time.

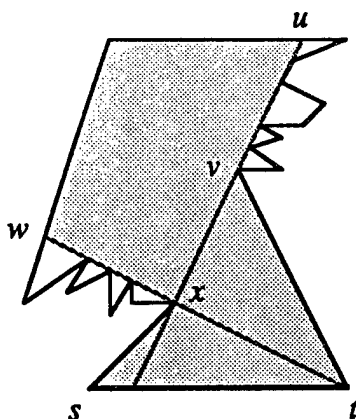


Figure 5.5. In degenerate cases, multiple polygon vertices may lie on one extension edge on the boundary of $Vis(\overline{st})$. None of the shortest path map edges along the extension edge needs to be changed, so the algorithm skips over them.

The shortest paths $\pi(s, p)$ and $\pi(t, p)$ are not hard to maintain. They change only when p passes through a vertex of $SPM(s)$ or $SPM(t)$. When p reaches a vertex of $SPM(t)$ of the type shown in Figure 5.4(a), then r , the last vertex of $\pi(t, p)$, is popped off the top of its stack. When p advances from v along edge $\overrightarrow{vq}$ and q is to the right of $\overrightarrow{rv}$, then v is pushed onto the stack for $\pi(t, p)$. Changes to the stack for $\pi(s, p)$ are made analogously. The two shortest paths change only in these four situations.

The algorithm runs in time proportional to the number of vertices p visits. The only vertices p visits where the algorithm does not modify the current subdivision are vertices v where $l(s, v) = l(t, v)$ and the edge from v to $l(s, v)$ belongs to both $SPM(s)$ and $SPM(t)$. Such a vertex is one endpoint of an extension edge on the boundary of $Vis(\overline{st})$; it is a “shadow” of a vertex visible from $\overline{st}$. The algorithm changes the subdivision at the vertex of which v is a shadow. Since each vertex where the algorithm does change the subdivision has at most one such shadow

vertex, we have the following theorem.

Theorem 5.1. *The sweeping-path algorithm computes $SPM(t)$ from $SPM(s)$ in time proportional to the number of differences between the two shortest path maps.*

5.2.2 A depth-first search algorithm

Our second transformation algorithm is a depth-first search algorithm; it is based on the shortest path map construction of Chapter 4. Because it uses finger search trees [GMPR77,HM82], it may be harder to implement than the sweeping-path algorithm. The idea behind the depth-first search algorithm is not difficult: the algorithm of Section 4.1, which runs in a triangulated simple polygon, can be modified to run using a shortest path map in place of the triangulation. We build $SPM(t)$ by running the modified algorithm in $SPM(s)$. The algorithm changes only the regions overlapping $Vis(\overline{st})$; it runs in time proportional to the number of differences between $SPM(s)$ and $SPM(t)$.

The algorithm of Chapter 4 is a depth-first exploration of a triangulation. It splits the current funnel each time it enters a new triangle; when it finishes, it has computed a funnel for each polygon edge. Extending the edges on the funnel sides gives the shortest path map decomposition of the polygon.

Because the polygon is subdivided into triangles, the algorithm splits a funnel in two when it enters a region. However, funnel-splitting works equally well if the polygon is subdivided into convex polygons with more than three sides. When the algorithm crosses a diagonal to enter a convex subpolygon with k sides, it splits the diagonal's funnel into $k - 1$ subfunnels.

Each region of a shortest path map is convex, so we can build $SPM(t)$ by running the algorithm of Chapter 4 in $SPM(s)$ instead of in a triangulation. Because $SPM(s)$ has vertices that do not belong to P , the resulting shortest path map has extra edges that we then have to delete. A second potential problem arises because regions of $SPM(s)$ may be degenerate. A region may have arbitrarily many edges on its boundary, even though the edges lie on only three lines. However, the exploration strategy outlined below ensures that degeneracies do not hurt the

algorithm's performance.

Our algorithm runs in time proportional to the number of differences between $SPM(s)$ and $SPM(t)$. To do this, it explores only the regions of $SPM(s)$ that overlap $Vis(\overline{st})$. It determines which regions to explore at run time. This is feasible because the algorithm of Section 4.1 is incremental: at each step the data structures contain the correct shortest path tree for the subpolygon explored so far.

The algorithm begins by exploring the region incident to $\overline{st}$. In the rest of the polygon, it crosses only diagonals that lie inside $Vis(\overline{st})$. We show that these diagonals are exactly those with nontrivial funnels in $SPT(t)$, the shortest path tree of t . Let $\overline{ab}$ be a diagonal of $SPM(s)$, and without loss of generality assume that $l(s, b) = a$. Then $\overline{ab}$ lies inside $Vis(\overline{st})$ if and only if $l(t, b) \neq a$, that is, the funnel associated with $\overline{ab}$ in $SPM(t)$ does not have a as its apex. Because b is the funnel apex only for $\overline{ab} = \overline{st}$, edge $\overline{ab}$ lies inside $Vis(\overline{st})$ if and only if its funnel in $SPT(t)$ is nontrivial.

When the algorithm enters a region by crossing an edge e , it can determine in constant time whether any particular edge of the region has a trivial funnel (equal to the edge itself). An edge has a trivial funnel if and only if the tangents from its endpoints to the interior of e 's funnel lie on the same line. In such a case the tangents both pass through the same endpoint of e .

The algorithm cannot afford to look at all the diagonals that it does not cross. Fortunately, it can recognize edges with trivial funnels without looking at them individually. The algorithm enters each region by crossing an edge e incident to the apex of the region. (This is because s and t are neighbors on P .) If there are any other edges on the side of the region that contains e , their associated funnels must be trivial. (See Figure 5.6.) Because the apex of each region in the subdivision has a pointer to the base of the region, we can check in constant time whether the base's funnel is trivial. If any edge on the side of the region opposite e has a trivial funnel, all such edges do. We can classify them all at once by checking the edge incident to the apex of the region. (In Figure 5.6, only the base has a nontrivial funnel.)

The algorithm constructs $SPM(t)$ inside the region without looking at the edges

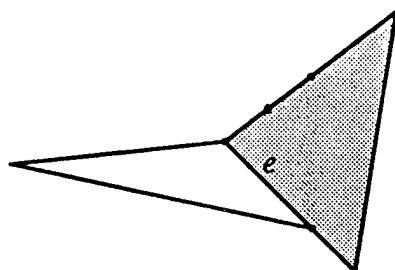


Figure 5.6. The algorithm enters the shaded region by crossing edge e . The funnels on both sides of the shaded region are trivial.

with trivial funnels. It splits up the funnel that belongs to e among the other edges. (This splitting is no more difficult if several of the edges with nontrivial funnels lie on a single side of the region.) If any of the edges with nontrivial funnels lies on the boundary of P , the algorithm divides its funnel up into regions of $SPM(t)$ and sets the base pointers at the apex of each region. If multiple edges with trivial funnels lie on a single side of a region of $SPM(s)$, then those edges also belong to $SPM(t)$. If only one such edge lies on a single side, it may not belong to $SPM(t)$. The cleanup phase described below deletes it if necessary.

When the algorithm finishes its exploration, it has constructed a refinement of $SPM(t)$ in a contiguous set of regions of $SPM(s)$. This subpolygon of P may have some vertices that do not belong to P , and hence the new shortest path map may have some vertices and edges that do not belong to $SPM(t)$. The additional vertices are vertices of $SPM(s)$ that split polygon edges; they lie on the bases of regions of $SPM(s)$. The additional edges are incident to the extraneous vertices. We clean up the shortest path map by visiting both endpoints of the base of each explored region of $SPM(s)$. If the diagonal edge incident to such a vertex separates two adjacent triangular regions of (the computed) $SPM(t)$ that share the same apex and whose bases lie on the same polygon edge, then the edge does not belong in $SPM(t)$; we delete the edge, merge the two regions into one, and set the base pointer at the apex of the region. This cleanup operation removes all the edges that do not belong to $SPM(t)$, both inside the changed subpolygon and on its boundary. It leaves

unchanged those edges that belong to both $SPM(s)$ and $SPM(t)$. It runs in time proportional to the number of regions of $SPM(s)$ explored, since it looks at at most two edges in each such region.

After the cleanup operation, the subpolygon that has been changed contains $SPM(t)$; the unchanged part of the polygon contains $SPM(s)$ (and hence $SPM(t)$). This concludes the transformation. We have computed $SPM(t)$, and the new subdivision is ready to be transformed again.

Like the algorithm of Chapter 4, this algorithm runs in time proportional to the total number of funnel edges it creates. Each funnel-splitting operation creates one new funnel edge. The number of funnel-splittings performed inside a region of $SPM(s)$ is proportional to the number of edges with nontrivial funnels on the boundary of the region. Since P does not have consecutive collinear edges, this quantity is in turn proportional to the number of regions adjacent to the current one that are explored after it. Thus the total number of funnel-splittings is proportional to the number of regions of $SPM(s)$ the algorithm explores.

The depth-first exploration and the cleanup phase both take time proportional to the number of regions of $SPM(s)$ the algorithm explores. Each region except the first is reached by deleting an edge of $SPM(s)$. Since the number of edges deleted is just part of the total number of edges changed, we have the following theorem.

Theorem 5.2. *The depth-first search algorithm computes $SPM(t)$ from $SPM(s)$ in time proportional to the number of differences between them.*

5.3 Complexity bounds

This section shows that each of the algorithms of the preceding section takes $O(m)$ time overall. The proof associates a constant number of shortest path map edges with each visibility edge or polygon edge; every edge that appears in a shortest path map has an associated visibility edge or polygon edge. As the source of the shortest path map moves around the boundary of P , edges are added to and deleted from the current shortest path map. We show that each edge is added and deleted a constant number of times, which gives the desired bound on the running time.

It is helpful to think of shortest path edges as being directed. The edge $\overrightarrow{pq}$ has two versions, $\overrightarrow{pq}$ and $\overleftarrow{pq}$. When $\overrightarrow{pq}$ appears in shortest path map $SPM(v)$, we write it as $\overrightarrow{pq}$ if p lies on the path from v to q and $\overleftarrow{pq}$ otherwise.

We associate every edge that appears in a shortest path map with a directed polygon edge or visibility edge. Shortest paths follow visibility edges and polygon edges, so we only need to account for the edges of the shortest path map that are not in the shortest path tree. These edges are extensions of funnel edges. Each funnel edge is a polygon edge or a visibility edge; we associate each extension edge with the version of its funnel edge that is directed toward it. Each directed edge has at most one extension.

The following lemma characterizes the shortest path maps in which each directed edge appears.

Lemma 5.3. *Let $\overrightarrow{pq}$ be a polygon edge or a visibility edge. The shortest path maps in which the directed edge $\overrightarrow{pq}$ appears have sources that form a contiguous subsequence of the polygon vertices.*

Proof: We exhibit polygon vertices l and r such that $\overrightarrow{pq}$ appears in $SPM(v)$ for all v strictly counterclockwise of l and clockwise of r in the sequence of polygon vertices, and $\overrightarrow{pq}$ appears in no other shortest path maps. We first note that $\overrightarrow{pq}$ must appear in $SPM(p)$. Let l be the first vertex clockwise of p for which $\overrightarrow{pq}$ is not in $SPM(l)$. Because shortest paths do not cross, $\overrightarrow{pq}$ cannot appear in any shortest path map whose source is between l and q , inclusive. A similar argument shows the existence of r and proves the lemma. ■

Now consider the shortest path map transformations of the previous section. If $\overrightarrow{st}$ is an edge of P , the shortest path map $SPM(t)$ can be constructed from $SPM(s)$ in time proportional to the number of edges that must be changed to get from one to the other. The preceding lemma lets us bound the time needed to produce shortest path maps for all the polygon vertices.

Theorem 5.3. *Given $SPM(v_1)$ for any polygon vertex v_1 , repeated application of either of the algorithms of Section 5.2 produces a shortest path map for every*

polygon vertex in $O(m)$ total time. Therefore, we can build the visibility graph of P in $O(m + n \log \log n)$ time.

Proof: As the source of the shortest path map moves around the perimeter of the polygon, each directed polygon or visibility edge is added to or deleted from the current shortest path map at most twice. (This follows from Lemma 5.3.) The extension edge associated with the directed edge, if any, is added and deleted along with it. There are $2n + 2m$ directed edges, and hence by either Theorem 5.1 or Theorem 5.2 the $n - 1$ applications of the transformation procedure take $O(m)$ time altogether. ■

5.4 Conclusion and open problems

This chapter has presented an algorithm to find the visibility graph of a simple polygon in time $O(m + n \log \log n)$, where m is the number of edges in the graph. The algorithm works by computing the shortest path map from each vertex of the polygon, then reading off the visibility edges incident to the source of the shortest path map. Rather than compute each shortest path map separately, the algorithm finds one shortest path map in $O(n \log \log n)$ time, then builds the others as modifications of the first one. In particular, the shortest path map with source t is built by modifying the one whose source is the clockwise neighbor of t along the polygon boundary. This procedure is fast because of the similarity between two shortest path maps whose sources are neighbors on the polygon.

Because it uses finger search trees to find the initial shortest path map, the algorithm may be difficult to implement. We can avoid the use of finger search trees (at the cost of slightly increased asymptotic complexity) by finding the initial shortest path map $SPM(v_1)$ in $O(n \log n)$ time. We triangulate P with the algorithm of Garey et al. [GJPT78]. By replacing the finger search trees in the algorithm of Section 4.1 with linked lists, we find the shortest path map in $O(n \log n)$ additional time. We use the sweeping-path algorithm to transform shortest path maps into

each other. This yields an $O(m + n \log n)$ algorithm that uses only simple data structures.

The algorithms of Section 5.2 can be simplified if we assume that P is non-degenerate. If no three points of P are collinear, then any triangular shortest path map region has at most five vertices on its boundary. In this case the sweeping path algorithm can afford to visit all the vertices on the boundary of $\text{Vis}(\overline{st})$: the algorithm adds or deletes an edge at every such vertex. Similarly, the depth-first search algorithm can afford to check every edge's funnel explicitly. The number of edges on the boundary of the explored subpolygon is proportional to the number of regions of $\text{SPM}(s)$ that make it up.

Even if degeneracies are allowed, alternate definitions of visibility can simplify our algorithms. If we define two points to be mutually visible when the segment connecting them does not intersect the exterior of P , the size of the visibility graph may increase. In a polygon with many collinear points, we may have $m = \Omega(n^2)$ under this definition and $m = O(n)$ under the standard definition. Under this broader definition of visibility, the sweeping-path algorithm will run in $O(m)$ time without using special cases to move p past the vertices on extension edges.

The presence of the $O(n \log \log n)$ term in our algorithm's running time suggests an open question: Is it possible to triangulate a simple polygon in $O(m)$ time? If it were, we would have an optimal visibility graph algorithm for simple polygons. (There are polygons with $m = O(n)$ for which the triangulation algorithm of Tarjan and Van Wyk takes $\Omega(n \log \log n)$ time.) This question fits into a framework of earlier work. Several authors have proposed triangulation algorithms that run in time $O(n \log k)$ for some parameter k related to the complexity of the polygon. For example, in the algorithm of Hertel and Mehlhorn, k is the number of reflex angles in the polygon [HM83]; in that of Chazelle and Incerpi, k is the "sinuosity" of the polygon [CI84]. In some sense, we ask less of our triangulation algorithm than these authors. We want it to run in $O(m)$, but this can be much larger than $O(n \log \log n)$. When m is smaller than $O(n \log \log n)$, the polygon is narrow and few triangulations are possible. This may make it easier to find one of them.

Chapter 6

Optimal Shortest Path Queries in a Simple Polygon

*How many miles to Babylon?
Threescore and ten.
Can I get there by candle-light?
Yes, and back again.*

— Nursery rhyme

This chapter, like Chapter 4, focusses on the problem of finding shortest paths for a point moving inside a simple polygon with n vertices. Several algorithms have been proposed to solve this problem. All the methods (including the one presented here) are based on a triangulation of the polygon. The algorithm of Lee and Preparata finds the shortest path between two points inside a simple polygon in linear time, once a triangulation is known [LP84]. Reif and Storer's [RS85] approach uses precomputation to speed up queries. Given a source point inside the polygon, their method produces a search structure so that the distance from a query

point to the source can be found in $O(\log n)$ time. The shortest path itself can be obtained in time proportional to the number of turns along it. Reif and Storer's method uses the Delaunay triangulation of the polygon and hence takes $O(n \log n)$ preprocessing time. Chapter 4 shows how to set up a similar query structure with less preprocessing. That algorithm takes linear time once a triangulation is known.

The sub-linear query algorithms mentioned above pre-select a fixed source point. If neither endpoint of the path is predetermined, each of these algorithms takes linear time to find the shortest path (or even its length). The present chapter removes the single-source restriction; it gives a data structure that supports sub-linear shortest path queries when both endpoints are part of the query. The data structure can be built in linear time once the polygon has been triangulated. After the data structure has been built, the query algorithm can find the length of the shortest path between two arbitrary points inside the polygon in logarithmic time. The path itself can be obtained in additional time proportional to the number of turns along it. These bounds are clearly best possible.

The idea underlying our method is not difficult. The preprocessing phase creates a hierarchy of nested subpolygons over an underlying triangulation, such that any shortest path crosses only a small number of subpolygons. We store information about shortest paths inside each such subpolygon. At query time, we obtain the shortest path between the query points by assembling path information from the subpolygons between them.

Section 6.1 describes the balanced hierarchical decomposition of P that we will use. (Appendix A tells how to construct it.) Section 6.2 characterizes shortest paths in terms of this decomposition. Section 6.3 describes the "funnels" and "hourglasses" associated with shortest paths, as well as the data structures for representing them. These structures allow linear preprocessing and $O(\log^2 n)$ query time. Finally, Section 6.4 shows how to use an additional $O(n)$ preprocessing time to enhance the structures of the previous section, thus yielding the desired logarithmic query time. Section 6.5 discusses various extensions and applications.

6.1 Balanced hierarchical decomposition of P

This section introduces a method, due to Chazelle and Guibas [CG85], of splitting the polygon P into subpolygons. The important property of the decomposition is that the shortest path in P between any pair of points passes through only a logarithmic number of subpolygons. The shortest path can then be derived from the sequence of subpolygons by referring to pre-computed information about shortest paths inside each subpolygon.

The polygon cutting theorem of Chazelle states that any simple polygon of at least four vertices has a diagonal that divides it into two subpolygons of roughly equal size [Cha82]. Specifically, if the polygon P has n sides, each of the two subpolygons P_1 and P_2 produced by splitting has at least $n/3 + 1$ sides and at most $2n/3 + 1$: the larger subpolygon is less than twice as big as the smaller. If we apply the theorem recursively to split each of the subpolygons, we get a balanced, hierarchical decomposition of P into triangles. These triangles form a triangulation of P .

The decomposition is easier to work with if we interpret it as a binary tree S whose nodes correspond to the splitting diagonals of P . At the root of the tree is the diagonal that splits P into P_1 and P_2 . The children of the root are the roots of the (recursively defined) decomposition trees corresponding to P_1 and P_2 . We can think of the decomposition as proceeding in stages: at time 0 the root diagonal splits P into two balanced subpolygons; the children of the root split these subpolygons at time 1, and so on. The time at which a particular diagonal is introduced corresponds to its depth in the tree S . We call the subpolygons produced during this process *cells*. These cells are the polygons for which we pre-compute and store shortest-path information. They are fundamental to our approach to the shortest path problem, and it is worth taking time to understand them.

For ease of reference, we associate with each diagonal d the cell it splits in two; we call the cell P_d . The diagonal d cuts P_d into two other cells; by the definition of S , the children of d in S are the diagonals associated with those cells. The *depth* of a cell P_d is equal to the depth of d in S . Triangles of the underlying triangulation are

cells not associated with a diagonal; we assign such a triangle a depth one greater than the depth of its deepest bounding diagonal. Because the decomposition is balanced, the depth of a diagonal d in S is related to the size of P_d . Let the depth of d be $\ell(d)$ (the depth of the root is 0). Then it follows from the splitting bounds that P_d has no more than $O((2/3)^{\ell(d)}n)$ vertices on its boundary. This fact implies that the tree S has height logarithmic in n .

The edges bounding a cell are of two kinds, edges of P and diagonals of P . Polygon edges act as walls: shortest paths in P_d cannot cross them. Diagonals act as doors: shortest paths that pass through P_d cross diagonals to enter and leave. As it happens, a cell has relatively few bounding diagonals. The diagonals bounding P_d must all be of different depths less than $\ell(d)$, as follows easily by induction. Therefore P_d has at most $\ell(d) = O(\log n)$ diagonals on its boundary.

Shortest paths that pass through a cell enter and leave by a particular pair of diagonals. To systematically refer to these pairs, we define a graph S^* , called the *factor graph*, based on the decomposition tree S . For each cell P_d , S has an edge connecting d to its parent, the deepest diagonal bounding P_d . We get S^* from S by adding edges connecting d not only to its parent, but to *all* the diagonals bounding P_d . By construction, S^* contains S as a subgraph. Each diagonal d has at most $\ell(d)$ edges in the factor graph connecting it to diagonals of lesser depth, one to each bounding diagonal of P_d . Furthermore, since a diagonal d is adjacent to only two cells at each stage of the decomposition, d has edges to at most two diagonals of each depth greater than $\ell(d)$. These observations give a logarithmic degree bound for the nodes of S^* .

Each edge of S^* represents a pair of diagonals for which we store information about shortest paths crossing the diagonals. To have any hope of storing the information in a linear-size structure, we must show that the factor graph has only linearly many edges.

Lemma 6.1. *The size of the factor graph S^* is $O(n)$.*

Proof: For a node d in S , the *height* of d , denoted by $h(d)$, is the length of the longest path to a leaf below d . If e denotes this deepest leaf, then

the height of d is $\ell(e) - \ell(d)$. Because S is balanced, there cannot be many diagonals with large height. In particular, if $h(d)$ is k , then it is easy to prove inductively that P_d must contain at least $\lfloor (3/2)^{k-1} + 1 \rfloor$ triangles of the underlying triangulation. The cells belonging to diagonals with the same height k in S are all disjoint; it follows that there are only $O((2/3)^k n)$ diagonals with height k in S . The graph S^* has $n - 3$ nodes and at most $2h(d)$ edges to descendants from each node d (one for each depth on each side). Overall therefore, S^* has at most

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

edges, which was to be proved. ■

We can compute the decomposition tree S and its extension S^* in the time it takes to triangulate P plus linear additional time. We use the algorithm of Tarjan and Van Wyk [TV86] to triangulate the polygon in $O(n \log \log n)$ time, then find a balanced hierarchical decomposition using the linear-time method described in Appendix A. From the decomposition we can easily get both S and S^* .¹

6.2 Shortest paths and the polygon decomposition

This section analyzes the sequence of diagonals crossed by a shortest path in P . It defines a logarithmic-size subsequence of those diagonals that helps determine the shortest path. The most important characteristic of the subsequence is that every pair of diagonals adjacent in it lies on the boundary of some cell of the decomposition, so the pair corresponds to an edge in the factor graph. We can discover this subsequence quickly, once we know the triangles containing the path

¹Note that the decomposition provided by the polygon cutting theorem is rarely unique; different decompositions result from different choices of splitting edges. When the decomposition is unique, there is only one triangulation of the polygon, and the algorithm of Tarjan and Van Wyk finds it.

endpoints.² By combining the pre-computed information stored for adjacent pairs of diagonals, we can then find the path itself.

For the remainder of this section, let us fix our attention on the shortest path connecting two particular points p and q in P . The triangles containing the points p and q in the underlying triangulation associated with S determine the sequence of diagonals crossed by the shortest path between the two points. These diagonals each split P into two parts, one containing p and the other q . We call these diagonals *separating* for p and q ; the shortest path from p to q crosses only these diagonals, and each of them exactly once. In fact, the shortest path between the points must cross these separating diagonals in a particular order. When the shortest path from p to q crosses a diagonal d , it must have already crossed all the separating diagonals on the side of d nearer p and none of those on the side nearer q . Note also that in this ordering adjacent separating diagonals must have different depths, and that between any two diagonals of the same depth there must be one of lesser depth (the “lacuna” property). These facts are easy to establish and their proofs are omitted.

We now define a subsequence of the separating diagonals that will be useful in finding the shortest path, the subsequence of the so-called *principal separating diagonals*. Which are the principal diagonals? Consider the lowest common ancestor $\hat{d}$ in S of all the separating diagonals. Clearly $\hat{d}$ itself is a separating diagonal and its depth is minimal among all separating diagonals. Indeed, the above remarks imply that $\hat{d}$ can be characterized by this minimal depth property. We define $\hat{d}$ to be a principal diagonal. The separating diagonals nearest p and q are also defined to be principal diagonals; we call them d_p and d_q , respectively. (Note that $\hat{d}$ can be d_p or d_q , or both, if they coincide.) We then define D_p to be the subsequence of the separating diagonals between d_p and $\hat{d}$ obtained as follows: Scan through this sequence from d_p to $\hat{d}$ and keep track of the minimum depth seen so far. During the scan, discard all diagonals with depth greater than the current minimum. The remaining diagonals are of strictly increasing depth (by the lacuna property) and form D_p . The sequence D_q is defined symmetrically, proceeding from d_q to $\hat{d}$. The

²The triangles can be found using any standard $O(\log n)$ point-location method, such as [EGS86].

principal diagonals are then just those in D_p together with those in D_q .

The strict monotonicity of depths in D_p has two important consequences. First, it implies that there are only logarithmically many principal diagonals in D_p . Second, each diagonal in D_p , except $\hat{d}$, is contained in the cell split by its successor in D_p , and hence is a descendant in S of that successor. Therefore D_p is a subsequence of the diagonals on the path in S from d_p to $\hat{d}$. Analogous statements hold for D_q .

This observation makes it easy to compute D_p in logarithmic time. Instead of scanning through all the separating diagonals between d_p and $\hat{d}$ in the polygon P , we can just look at the diagonals on the path from d_p to $\hat{d}$ in the tree S . If any of these diagonals is between $\hat{d}$ and d_p in P , it is a separating diagonal and belongs to D_p . Otherwise, it is not separating and not in D_p . The test of whether one diagonal lies between two other diagonals can be performed in constant time. Here is an outline of the method: Let T be the dual tree of the triangulation. Preprocess it by assigning to each node t (corresponding to a triangle) its preorder and postorder numbers $Pre(t)$ and $Post(t)$. Then a node s is the ancestor of node t if and only if both $Pre(s) < Pre(t)$ and $Post(s) > Post(t)$. Translate the query about diagonals of P (corresponding to edges of T) into one about nodes of T . The query asks whether node s lies on the path from r to t . The answer is yes if and only if one of two conditions holds: s is an ancestor of exactly one of r and t , or it is their lowest common ancestor.

Because all the separating diagonals between a pair e and f of adjacent principal diagonals in D_p have depth greater than e and f , the principal diagonals e and f must lie on the boundary of a single cell and be connected by an edge in the factor graph. These adjacent principal diagonals represent pairs for which the preprocessing phase of our algorithm stores a representation of all shortest paths between points on the diagonals in the pair. In conclusion, we know that in $O(\log n)$ time we can extract a sequence of $O(\log n)$ cells in whose "concatenation" the shortest path from p to q must lie.

6.3 Hourglasses and shortest paths

This section uses an enhanced version of the hourglasses introduced in Section 4.3 to represent shortest paths between two diagonals. An hourglass is associated with each edge of the factor graph; by combining the hourglasses of the edges that connect principal diagonals, we get a representation of all shortest paths between d_p and d_q . The previous sections have been purposefully vague about the notion of “combining” hourglasses; we now make the combining process more concrete. However, we postpone describing the data structures that implement hourglasses until Subsection 6.3.3.

In Chapter 4 we used funnels to represent all shortest paths from a source vertex to an edge. Because the source vertex in our problem is not fixed, we need richer structures than funnels. Hourglasses provide the additional flexibility we need. Let $\overline{AB}$ and $\overline{CD}$ be two diagonals in the triangulation of P , labelled so that $BACD$ is a subsequence of the vertex sequence of P . As in Section 4.3, we define the union of the two shortest paths $\pi(A, C)$ and $\pi(B, D)$ to be the hourglass of $\overline{AB}$ and $\overline{CD}$. (Please refer to Figure 6.1.) If the paths share any vertices, then the hourglass is essentially two funnels joined by a polygonal path between their apexes. We call this shortest path between the apexes the *string* and refer to the hourglass as *closed*. If the paths are disjoint, the hourglass consists of two inward convex chains. In this case, $\overline{AB}$ and $\overline{CD}$ are *mutually visible*; there is a segment with one endpoint on $\overline{AB}$ and the other on $\overline{CD}$ that avoids the polygon. We refer to the hourglass as *open*.

Given a point p on $\overline{AB}$ and a point q on $\overline{CD}$, we can use the hourglass to find the shortest path from p to q . If the hourglass is closed, the problem decomposes into two single-source queries. The complete shortest path is just the concatenation of the path from p to the apex of its funnel, the string, and the path from the apex of the other funnel to q . As noted in Section 4.1, the path from p (or q) to the apex of its funnel begins with the tangent from the point to the funnel, then follows funnel edges to the apex. When the hourglass is open, the situation is slightly more complicated. If p and q are mutually visible (segment $\overline{pq}$ avoids the hourglass), then $\overline{pq}$ is the shortest path. Otherwise, consider the lines of sight between $\overline{AB}$

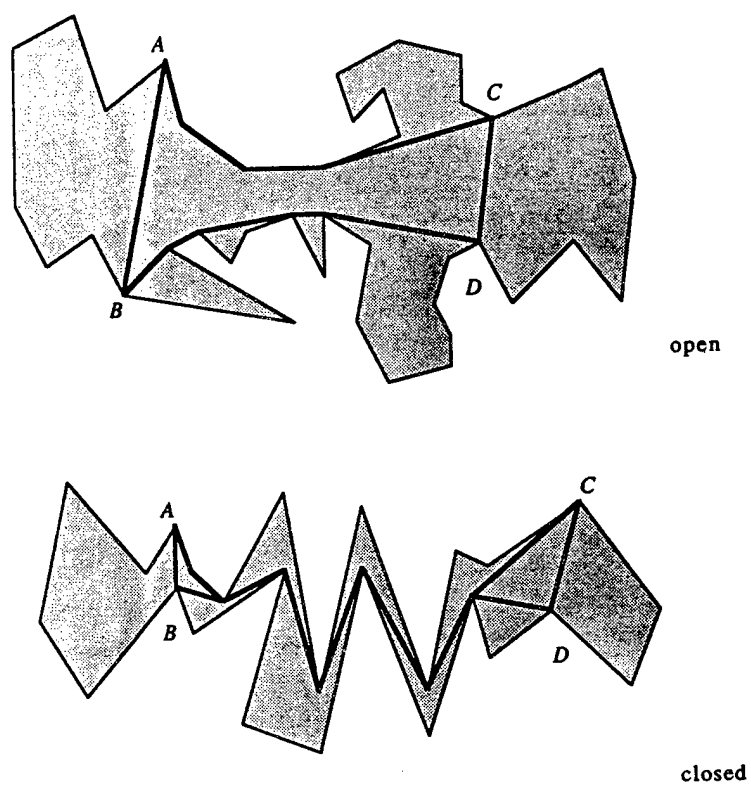


Figure 6.1. The hourglass of $\overline{AB}$ and $\overline{CD}$

and $\overline{CD}$ (segments that avoid the hourglass and have one endpoint on each of $\overline{AB}$ and $\overline{CD}$). Suppose one of the lines of sight has p and q on the same side, say the side where A and C lie. Then the shortest path includes the tangent from p to $\pi(A, C)$, the tangent from q to $\pi(A, C)$, and the portion of $\pi(A, C)$ connecting the tangents. See Figure 6.2(a). If every line of sight separates p from q , the shortest path includes an inner common tangent between the hourglass chains. In this case there is an unblocked tangent from p to one of the hourglass chains, say $\pi(A, C)$, but not to the other. Similarly, there is a tangent from q to $\pi(B, D)$. The shortest path between p and q includes these tangents, the inner common tangent between $\pi(A, C)$ and $\pi(B, D)$ that gives the shorter path, and the portions of the hourglass chains needed to connect the tangents. Please refer to Figure 6.2(b) for an example of this type of shortest path.

Hourglasses are especially well suited to our algorithm because they are *concatenable*. That is, if diagonal d_2 separates d_1 from d_3 , it is easy to compute the hourglass for the diagonal pair (d_1, d_3) from the hourglasses for (d_1, d_2) and (d_2, d_3) . There are several special cases, but the basic idea is simple: we just find the common tangents of the convex chains that make up the two hourglasses. The special cases arise because the concatenation of two open hourglasses can be closed.

- case A: If the hourglasses for (d_1, d_2) and (d_2, d_3) are both closed, so is that for (d_1, d_3) . It retains the funnels for d_1 and d_3 unchanged, but omits both funnels for d_2 . The shortest path $\pi(a, a')$ between the apexes of the two funnels for d_2 consists of the common tangent between the two funnels and the funnel portions that connect it to the apexes. The string for (d_1, d_3) incorporates the strings for (d_1, d_2) and (d_2, d_3) , linking them with $\pi(a, a')$. Please see Figure 6.3.
- case B: If one hourglass is closed, say that for (d_1, d_2) , and the other is open, the result is closed. Suppose a is the apex of the funnel for d_2 in the closed hourglass. Constructing the shortest paths from a to the endpoints of d_3 requires a constant number of common tangent computations. These shortest paths define a funnel on d_3 with apex a' . The new hourglass uses this funnel, the old funnel on d_1 , and augments the old string with the shortest path $\pi(a, a')$.

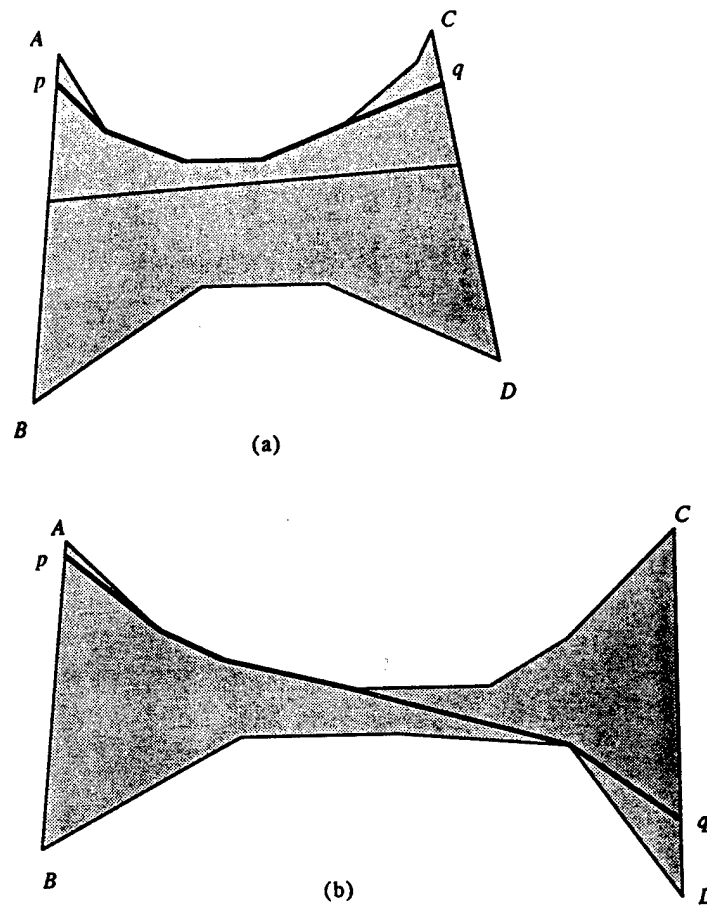


Figure 6.2. Shortest paths in an open hourglass

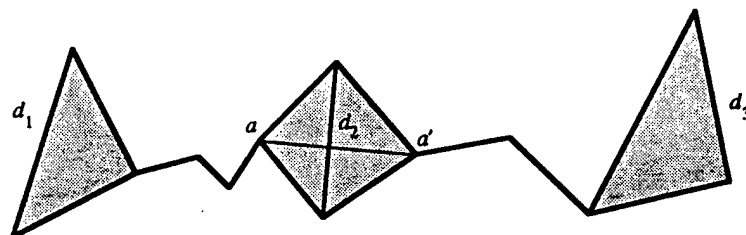


Figure 6.3. Concatenating two closed hourglasses

See Figure 6.4.

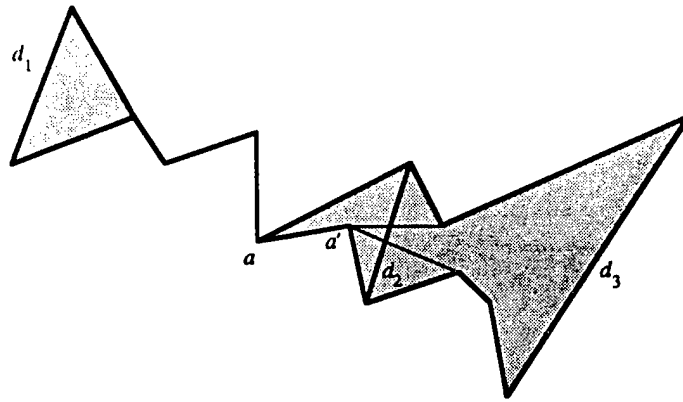


Figure 6.4. Concatenating one closed and one open hourglass

case C: If both hourglasses are open, their concatenation may be either open or closed.

The four common tangents that touch one chain from each hourglass determine the result. If both outer common tangents are unblocked, the new hourglass is open, as in Figure 6.5(a). Otherwise the result is closed, and the points of contact of the inner common tangents determine the apexes of the funnels. Refer to Figure 6.5(b).

We have not yet seen how to use hourglasses to find shortest paths if the path endpoints are not on the hourglass diagonals. However, this is not a difficult extension. Suppose d_p and d_q are the first and last diagonals in the sequence of separating diagonals between p and q , and that we are given the hourglass for (d_p, d_q) . The shortest path is determined by the tangents to the hourglass that pass through p and q . One way to think of this is to view p and q as tiny diagonals; then the triangle defined by p and d_p is the hourglass for (p, d_p) . Concatenating the hourglasses for (p, d_p) , (d_p, d_q) , and (d_q, q) gives a single closed hourglass with no funnels, only string. The string is the shortest path between p and q .

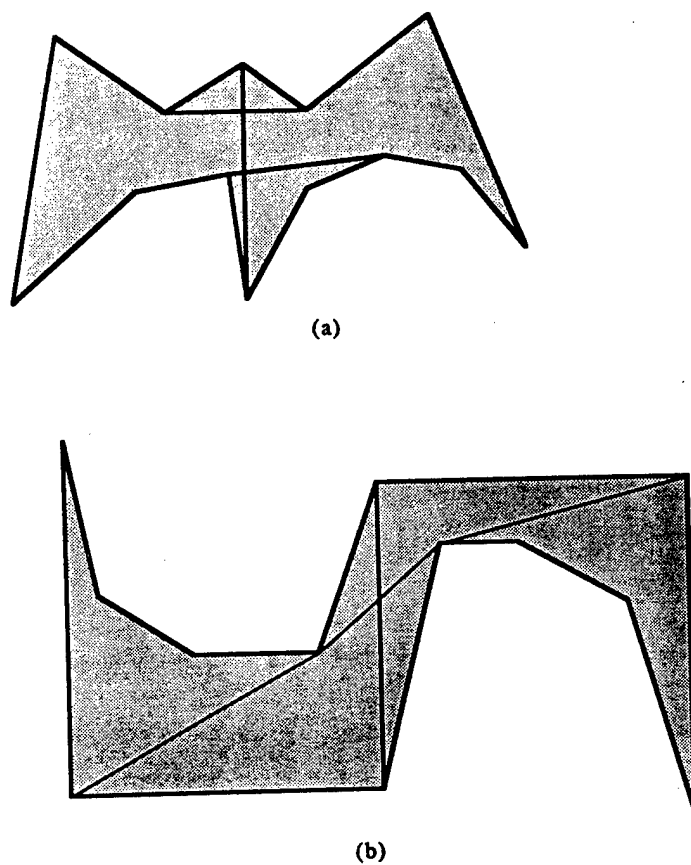


Figure 6.5. Concatenating two open hourglasses: (a) open result, and (b) closed result

6.3.1 Hourglass data structures: specifications and use

We have seen that concatenation of hourglasses gives a way to find shortest paths. However, the details of representing, constructing, and concatenating hourglasses have gone unmentioned so far. The sequel of this section corrects that omission, introducing a data structure that allows efficient representation and manipulation of hourglasses. This subsection specifies the operations that the structure must support, as well as the allowable space and time complexities of each. It demonstrates that any structure satisfying the specification can be used to answer shortest path queries efficiently. The following two subsections present a data structure that meets the specification.

Let us begin by giving a notation for our data structure. We represent the hourglass between diagonals d_1 and d_2 by the data structure $H(d_1, d_2)$. In all the specifications that follow, whenever diagonals d_1 , d_2 , and d_3 appear, we assume that d_2 lies between d_1 and d_3 in P . This means that hourglasses $H(d_1, d_2)$ and $H(d_2, d_3)$ can be concatenated.³

Concatenation is the fundamental hourglass operation. The previous section shows that concatenation reduces to finding common tangents between pairs of convex polygonal chains. A point is a degenerate convex chain, but our algorithms must be able to handle the general case about as well as the degenerate one. This is the import of the following specification requirement:

- (1) If our data structure allows tangent lines from a point to hourglasses $H(d_1, d_2)$ and $H(d_2, d_3)$ to be found in times τ_1 and τ_2 respectively, then the common tangents of the two hourglasses must be computable in time $O(\tau_1 + \tau_2)$.

This is not too restrictive a condition: the standard algorithms for finding a line through a point and tangent to a convex chain are related in the same way to the standard algorithms for finding common tangents between two convex chains [OvL81][PS85b, Section 3.3.6–7].

³The data structure for hourglass $H(d_1, d_2)$ need not be identical to that for $H(d_2, d_1)$, though the two are closely related. In this section we will assume that one hourglass can be obtained from the other in constant time. This happens to be true for the data structure introduced in the next two subsections; if it were not true, we could achieve the same effect (at twice the cost) just by computing both versions of every hourglass.

We want our data structure to reflect the close tie between finding tangents and concatenation. In particular, the structure must use space efficiently: we cannot afford to represent each hourglass as an explicit polygon. If an hourglass is formed by the concatenation of two others, we must record only incremental information. Furthermore, concatenation must leave its operands unchanged. Thus we impose the following requirement:

- (2) If the common tangents between $H(d_1, d_2)$ and $H(d_2, d_3)$ can be found in time τ , then the hourglass $H(d_1, d_3)$ can be constructed from the two constituent hourglasses in $O(\tau)$ time and additional space. Furthermore, the original two hourglasses are unaltered by the concatenation operation.

The following two properties of the hourglass specification guarantee that it can be used to answer queries quickly.

- (3) If tangents from a point to $H(d_1, d_2)$ and $H(d_2, d_3)$ can be found in times τ_1 and τ_2 , tangents to $H(d_1, d_3)$ can be found in time $\max(\tau_1, \tau_2) + C$ for some constant C .
- (4) If tangents from a point to $H(d_1, d_2)$ can be found in time τ , the length of the shortest path between query points on d_1 and d_2 can be found in $O(\tau)$ time. The path itself can be found in $O(\tau + m)$ time, where m is the number of turns in the path.

The shortest path from p to q crosses a sequence of $O(\log n)$ principal diagonals. Every pair of neighbors in that sequence is connected by an edge in the factor graph S^* . Thus if we store $H(d_1, d_2)$ for every pair of diagonals d_1 and d_2 linked by an edge in S^* , we will have all the hourglasses we need to compute shortest paths by concatenation. The next few paragraphs present an algorithm to construct the necessary hourglasses in linear time and space.

In Section 6.1 we imagined the decomposition of P as proceeding in a series of time steps, adding diagonals of depth k at time k . Now consider running time backwards, first deleting diagonals of greatest depth in S , then second greatest,

and so on. Cells are created roughly in increasing order of size; when a new cell appears, we will construct hourglasses between every pair of diagonals on the cell's boundary. When the last diagonal is finally removed, we will have constructed hourglasses corresponding to all the edges in S^* .

The inductive invariant of the construction is the following: just before the diagonals of depth k are deleted, all hourglasses in cells with depth greater than k have been computed. The basis is easy to establish: before deleting any diagonals, we construct trivial hourglasses between all pairs of diagonals that lie on the border of the same triangle in the underlying triangulation. As the algorithm runs, the deletion of a diagonal d of depth k merges two cells into one (the cell P_d). Each of the two cells has a set of hourglasses with d at one end (d has been paired with every other bounding diagonal of the cell). To maintain the invariant, we concatenate every hourglass from one side of d with every hourglass from the other side of d . This process computes hourglasses for exactly those pairs of diagonals linked by edges in S^* , and it requires only one concatenation for each hourglass generated.

The hourglasses built during the construction have special properties that we exploit to prove the linear time and space bounds. The procedure combines hourglasses in a balanced fashion, so that the cost of computing common tangents, which could be linear in the sizes of the chains involved, is instead logarithmic. To express this bound, let us introduce a logarithmic quantity to measure hourglass size. If two diagonals d_1 and d_2 are linked by an edge in S^* , they lie on the boundary of some cell. We define $\lambda(d_1, d_2)$ to be the logarithm of the size of that cell.

Lemma 6.2. *If d_1 and d_2 are linked by an edge in the factor graph, it is possible to find the tangents through a point to the hourglass $H(d_1, d_2)$ in time $O(\lambda(d_1, d_2))$.*

Proof: Let C' denote the constant implied by the " O " notation in the statement of the lemma. Like the hourglass construction, the proof proceeds by induction. The claim is certainly true for the hourglasses present before any diagonals are deleted. The cells they bridge are triangles, and so tangent computation takes only constant time, say time D .

Now consider deleting a diagonal d , creating cell P_d from left and right subcells. We concatenate every hourglass that links d to a diagonal on its left with every hourglass that links d to a diagonal on its right. The left and right cells are within a factor of two in size, so P_d is at least $3/2$ the size of the larger. By the third data structure requirement above, tangents to a newly created hourglass $H(d_1, d_2)$ can be found in time at most $C'\lambda(d_1, d_2) - C'\log(3/2) + C$. Choosing C' to be the maximum of $C/\log(3/2)$ and D completes the proof. ■

We now have the tools we need to prove the linearity of the construction. The preceding proposition, along with the first and second data structure requirements, implies that an hourglass obtained by concatenating two hourglasses across diagonal d' can be built from its constituent hourglasses in time and space proportional to the logarithm of the size of $P_{d'}$. This quantity is proportional to $h(d')$, the height of d' in S , which is in turn less than the height of the higher of the two diagonals the new hourglass links. Let this higher diagonal be d . Since there are $O(h(d))$ edges in S^* joining d to lower diagonals, the hourglasses corresponding to these edges can be built in $O((h(d))^2)$ time and space. Overall, the construction takes time and space proportional to

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

The hourglasses built by the construction are sufficient to answer shortest path queries in $O(\log^2 n)$ time: There are $O(\log n)$ hourglasses between p and q that link principal diagonals. These can be concatenated into a single long hourglass using $O(\log n)$ concatenations. The above proposition and the first three data structure requirements imply that each concatenation takes $O(\log n)$ time. Finally, by viewing p and q as diagonals, we can concatenate $H(p, d_p)$, $H(d_p, d_q)$, and $H(d_q, q)$ to get $H(p, q)$ in $O(\log n)$ additional time. This hourglass is a string equal to the shortest path from p to q . Since $H(p, q)$ is just a string, the length of the shortest path is available as soon as the concatenation is finished. By the fourth requirement, the path itself can be extracted in additional time proportional to the number of turns along it.

In Section 6.4 we will see how to improve this bound to $O(\log n)$.

6.3.2 The chain data structure

Convex polygonal chains are the basic constituents of hourglasses. Because of the importance of such chains, we devote this subsection to discussion of the chain data structure and its properties. This data structure represents ordered convex chains by binary trees, in a manner similar to the dynamic convex hull structure of Overmars and van Leeuwen [OvL81]. Subsection 6.3.3 then uses this data structure to implement hourglasses. In order to ensure that the hourglasses built from chains satisfy the four conditions previously stated, we construct the chains to satisfy analogous conditions.

The convex chains that appear in hourglasses have special properties not found in general convex chains. Each chain is a shortest path inside P : if the chain has endpoints A and B , the chain is $\pi(A, B)$. The polygon touches only one side of the chain, and hence the chain is convex. The chain bulges toward the polygon interior. Because the chain belongs to an open hourglass or funnel, it cannot spiral: the chain is equal to its own convex hull minus the segment $\overline{AB}$.

By convention we assume that the chain $\pi(A, B)$ uses vertices from the boundary of P counterclockwise from A and clockwise from B . We think of the chain as convex upward, with A at the left end.

In our representation, a chain can be one of two types: a *trivial* chain, which is a single polygon edge, or a *derived* chain, which is the convex hull of two subchains. In other words, if $\overline{AC}$ is not a polygon edge, the chain $\pi(A, C)$ is represented as the convex hull of $\pi(A, B)$ and $\pi(B, C)$ for some B along the polygon boundary between A and C . The chain contains the outer common tangent of $\pi(A, B)$ and $\pi(B, C)$, along with the parts of the subchains that connect the tangent to A and C . Either subchain may contribute nothing to the shortest path, or the tangent may be null. Figure 6.6 gives examples of these cases.

A tangent edge may consist of a single point; however, we ensure that there are not many such null edges in any chain. If the combination of two chains would put

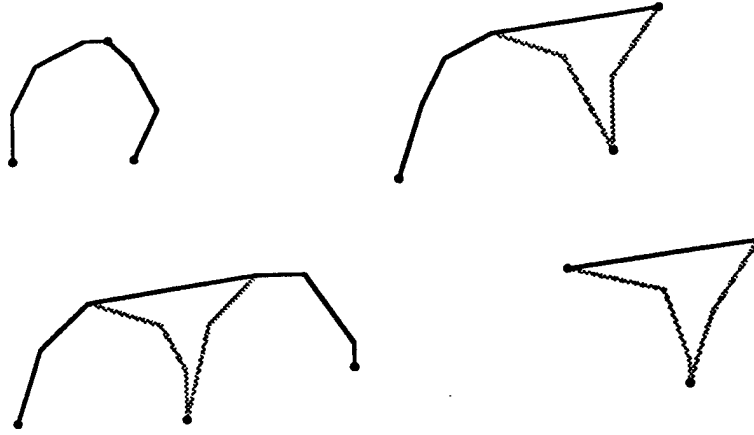


Figure 6.6. A chain is the convex hull of its subchains. Either subchain may contribute no edges to the result, or the tangent edge may be null.

two null edges next to each other, we consider the older of the two to be deleted by the addition of the new one. This means that every chain has fewer null edges than non-null ones. Similarly, we guarantee that no chain has adjacent collinear edges by extending the tangent edge to delete any subchain edge with which it is collinear.

The two most important operations supported by the chain data structure are tangent-finding and construction of a chain from its subchains. These are the same operations needed in the dynamic convex hull algorithm of Overmars and van Leeuwen [OvL81]. Like the dynamic convex hull structure, the chain data structure is basically a binary tree. A trivial chain is represented by a leaf node that contains the edge. A derived chain $\pi(A, C)$ is represented by a node that contains the common tangent of the subchains $\pi(A, B)$ and $\pi(B, C)$, along with pointers to the nodes that represent those subchains.

The data structure must support two types of tangent queries. In the first, we are given a chain $\pi(A, C)$ and a query point X outside the convex hull of the chain. We want to find the two tangents from X to the chain. We can find each tangent by looking at the edge stored at the node v that represents $\pi(A, C)$. In constant time we determine which subchain the tangent touches, then find the tangent by recursively searching in the subchain's tree. This search takes time proportional to the height of the tree rooted at v . The second type of tangent query asks for the

inner and outer common tangents of two chains. Neither chain is contained in the convex hull of the other. Using the techniques of Overmars and van Leeuwen, the tangents can be found in time proportional to the sum of the heights of the trees that represent $\pi(A, B)$ and $\pi(C, D)$.

In the hourglass application of the chain data structure, we do not need to compute all the tangents for each operation. We modify the tangent-finding algorithms to produce only the desired tangent. A tangent query just needs to specify, for each chain, whether the tangent segment heads clockwise or counterclockwise from the tangent point.

Our data structure is more complicated than that of Overmars and van Leeuwen because it must support more operations; however, the additional structures described below do not slow down the tangent-finding operations. Thus we have established the following analogues to conditions (1) and (3):

- (C1) If the lines passing through a given query point and tangent to chains $\pi(A, B)$ and $\pi(C, D)$ can be found in times τ_1 and τ_2 , the common tangents of the two chains can be found in time $O(\tau_1 + \tau_2)$.
- (C3) Suppose chain $\pi(A, C)$ is built from subchains $\pi(A, B)$ and $\pi(B, C)$. If tangents from a point to chains $\pi(A, B)$ and $\pi(B, C)$ can be found in times τ_1 and τ_2 , tangents to their combined chain $\pi(A, C)$ can be found in time $\max(\tau_1, \tau_2) + \langle \text{constant} \rangle$.

Condition (4) requires us to compute path lengths in hourglasses. To help find path lengths, we store chain lengths in the chain data structure. Consider a chain $\pi(A, E)$ built from two subchains $\pi(A, C)$ and $\pi(C, E)$. The subchains are joined by the tangent edge $\overline{BD}$, as shown in Figure 6.7. At the node representing $\pi(A, E)$, we store the lengths of its components $\pi(A, B)$, $\overline{BD}$, and $\pi(D, E)$. The sum of these lengths is the length of $\pi(A, E)$.

For ease of description, let us give names to some of the fields in the chain data structure. If node v represents the chain of Figure 6.7, $v.tan$ is the tangent edge $\overline{BD}$. The lengths of $\pi(A, B)$ and $\pi(D, E)$ are $v.llen$ and $v.rlen$. The tangent

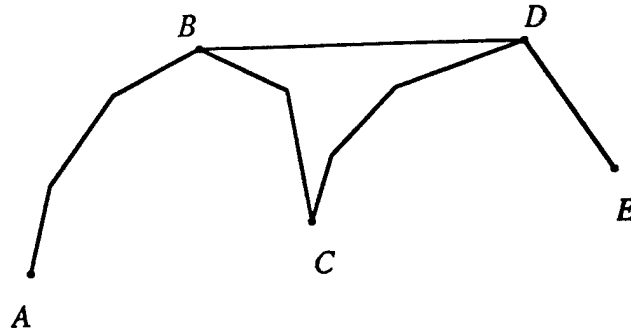


Figure 6.7. A canonical example of a chain formed by combining two subchains.

edge has length $v.tlen$. For convenience we define $v.len$ to be the length of the whole chain $\pi(A, E)$, the sum of the other length fields. If v 's chain is just a single edge, then $v.llen = v.rlen = 0$ and $v.len = v.tlen$.

The following lemma helps us fill in the length fields of v .

Lemma 6.3. *Let v be the root of the tree representing the chain $\pi(A, E)$ of Figure 6.7. Let u be a node in the tree whose edge $u.tan$ appears in $\pi(A, E)$, and let X be the right endpoint of $u.tan$. The length of the chain $\pi(A, X)$ can be determined by looking at the lengths stored at the nodes along the path from v to u in the tree.*

Proof: The proof is by induction on the length of the path from v to u . If the path is of length zero, then $u = v$, and $u.tan = \overline{BD}$. The length of $\pi(A, X)$ is just $u.llen + u.tlen$, the length of $\pi(A, B)$ plus the length of $\overline{BD}$.

If $u \neq v$, the proof has two cases, depending on whether u is in the left or right subtree of v . If u is left of v , then only edges from the subchain $\pi(A, B)$ contribute to $\pi(A, X)$. The right subchain $\pi(C, E)$ has no effect on the length of $\pi(A, X)$. By induction, we can find the length of $\pi(A, X)$ by looking at the lengths stored along the path from the left child of v down to u .

If u is right of v , then $\pi(A, X)$ consists of $\pi(A, B)$, $\overline{BD}$, and some prefix of $\pi(D, E)$. Thus the length of $\pi(A, X)$ is $v.llen + v.tlen$ plus the

length of $\pi(D, X)$. This last term is the same as the length of $\pi(C, X)$ minus the length of $\pi(C, D)$. If t is the right child of v , the node associated with $\pi(C, E)$, then the length of $\pi(A, X)$ is $v.llen + v.tlen + (v.rlen - t.len)$ plus the length of $\pi(C, X)$. (This is just $v.len - t.len$ plus the length of $\pi(C, X)$.) By induction, we can find the length of $\pi(C, X)$ by looking at the nodes on the path from t to u . ■

When we build chain $\pi(A, E)$ out of subchains $\pi(A, C)$ and $\pi(C, E)$, we find the outer common tangent of the two subchains, $\overline{BD}$ in Figure 6.7. Let v be the node that represents $\pi(A, E)$, and let e be the edge of $\pi(A, C)$ that is just to the left of the tangent $\overline{BD}$. Edge e is $u.tan$ for some node u in the tree representing the left subchain. We find u during the tangent-finding operation. The preceding lemma shows that we can determine $v.llen$ by looking at the nodes along the path from u up to the root of the left subtree. We can find $v.rlen$ in a similar manner. Thus all the length fields can be computed in time proportional to the height of v . This completes our discussion of the length fields.

The hardest data structure requirement to satisfy is the edge retrieval one: given a chain data structure, we must be able to retrieve the edges of the chain in constant time apiece. Retrieval in constant time per edge is especially difficult because the data structure for a chain must not modify the corresponding structures for its left and right subchains.

Our solution uses a compact representation of the differences between the chain and its left and right subchains. The representation is based on storing the two paths from the root to the nodes u and y whose edges $u.tan$ and $y.tan$ are adjacent on either side to the tangent between the subchains.

To be more concrete, consider the node v that represents the chain in Figure 6.7. Node v has two lists associated with it, a left and a right *edge retrieval list*, or *retrieval list* for short. Each record r in the left retrieval list has two pointers, one to a node in v 's left subtree, and one to a record in another retrieval list, called the *partner* of r . The right retrieval list is analogous to the left retrieval list.

We define retrieval lists in two stages: we specify the node pointers here and the *partner* fields below. Let w be the left child of v , and let u be the node in the

subtree rooted at w whose edge $u.tan$ appears in $\pi(A, E)$ and is incident to B . The records in v 's left retrieval list point to the nodes on the path from w to u , inclusive, whose associated edges belong to $\pi(A, E)$. The retrieval list keeps the nodes in the same order as the path from w to u . We have specified the node pointers of the left retrieval list; those of the right retrieval list are defined analogously.

The following lemma partially characterizes edge retrieval lists, and thereby enables us to specify the *partner* fields of list records.

Lemma 6.4. *Let s and t be two nodes pointed to by consecutive records in the left retrieval list of v . Then these two nodes are in an ancestor/descendant relationship. If s is the ancestor, then t is in the right subtree of s , and some record in the right retrieval list of s points to t . Analogous claims hold for records in the right retrieval list of v .*

Proof: We first prove that t is in the right subtree of s . Because $u.tan$ is adjacent to $v.tan$, it must lie to the right of $s.tan$, and hence u is in the right subtree of s . Since the path from u to v goes through t , t must be in the right subtree of s .

Because t 's associated edge $t.tan$ appears in v 's chain $\pi(A, E)$, it also appears in the chain $\hat{\pi}$ represented by s . All the edges between $s.tan$ and $t.tan$ in $\pi(A, E)$ also belong to $\hat{\pi}$.

Let x be the descendant of s such that $x.tan$ is immediately to the right of $s.tan$ in $\hat{\pi}$. Either $x = t$ or x is to the left of the path from s to t . Let y be the lowest common ancestor of x and t . Since x and t are both to the right of s , y is not equal to s .

In any chain in which $y.tan$ appears, it must lie between $x.tan$ and $t.tan$, inclusive; otherwise x and t would lie in the same subtree of y . Edges $x.tan$ and $t.tan$ both belong to $\pi(A, E)$, and hence so does $y.tan$. Node y lies on the path from u to v 's left child w , so it must belong to v 's left retrieval list. Because $s \neq y$, it must be that $y = t$. Hence t is an ancestor of x . The edge $t.tan$ belongs to $\hat{\pi}$, and so some record in the right retrieval list of s must point to t . ■

We can now specify the *partner* fields of retrieval list records. The *partner* field of the first record in a retrieval list is null. Let r and r' be two consecutive records in v 's left retrieval list that point to nodes s and t , with s an ancestor of t . Then the *partner* field of r' points to the record in the right retrieval list of s that points to t . The *partner* fields of records in the right retrieval list are defined analogously.

The following two lemmas show that edge retrieval lists allow us to report the edges of a chain in constant time apiece.

Lemma 6.5. *Let r be a record in the left retrieval list of v , and let s be the node it points to. Using the part of the retrieval list r heads, we can retrieve all the edges of v 's chain strictly between $s.tan$ and $v.tan$ in left-to-right order and in constant time per edge. A similar claim holds for right retrieval lists.*

Proof: The proof is by double induction on the height of v and the length of the retrieval list tail r heads.

If v has height zero, that is, it has no child trees, its retrieval lists are null and the claim is vacuously true. The claim also holds any time the retrieval list tail is empty: there are no edges to retrieve.

If r is the last record in v 's left retrieval list, the claim is true. There are no edges in v 's chain between $s.tan$ and $t.tan$.

Now assume the claim holds for all nodes with height less than that of v . Further assume that it holds for nodes with v 's height and retrieval list tails shorter than the one r heads. Let r' be the successor of r in v 's left retrieval list, and let t be the node pointed to by r' . We need to retrieve all the edges of v 's chain between $s.tan$ and $t.tan$.

Since both $s.tan$ and $t.tan$ appear in v 's chain, the edges between them in v 's chain are also edges of the chain represented by s . As noted in Lemma 6.4, some record in the right retrieval list of s points to t . The *partner* field of r' points to that record. By the induction hypothesis, we can retrieve all the edges between $s.tan$ and $t.tan$ using the tail of the right retrieval list of s that r' points to, since the height

of s is less than that of v . In constant time we report $t.tan$. Again by the induction hypothesis, we can report all the edges between $t.tan$ and $v.tan$ in constant time apiece, since the list r' heads is shorter than the one r heads. ■

Lemma 6.6. *Using v 's left edge retrieval list, we can report all the edges of v 's chain that are to the left of $v.tan$. We produce the edges in left-to-right order and at constant cost per edge. A similar claim holds for v 's right retrieval list.*

Proof: We consider only the edges to the left of v ; the other case is analogous. The proof is by induction on the height of v . If v has no children, its retrieval lists are null and the claim is trivially true. The claim is similarly true whenever the left retrieval list is null.

If the left retrieval list is not null, let r be the head of the list, and let t be the node to which r points. We prove that t is in the left subtree of each of its ancestors on the path to v . If t were to the right of any of its ancestors, the edge belonging to the highest such ancestor would belong to v 's chain, since all the edges belonging to higher nodes would be to the right of $t.tan$. Since none of t 's ancestors has an edge that belongs to v 's chain, t is to the left of all its ancestors.

The left part of t 's chain is not hidden by any of t 's ancestors, and so it belongs to v 's chain. By induction, we can report the edges left of $t.tan$ using t 's left retrieval list. We report $t.tan$ in constant time. By Lemma 6.5, we can report the edges between $t.tan$ and $v.tan$ in order and in constant time apiece using v 's left retrieval list. ■

We have shown that edge retrieval lists can be used to report the edges of a chain in constant time apiece. The lists for a node v are small: their size is bounded by a constant times the height of v . In the remainder of this subsection, we show how to construct edge retrieval lists efficiently.

Let u be the node in v 's left subtree such that $u.tan$ is adjacent to $v.tan$ in v 's chain. The left retrieval list of v points to nodes on the path from v to u . To

construct the retrieval list, we must be able to tell which nodes on the path have edges in v 's chain. We give a method for recognizing these nodes below.

Once we know which nodes on the path to u have edges in v 's chain, we can build a retrieval list with pointers to these nodes. We then fill in the *partner* fields of the records in the list. The *partner* field of the first record in the retrieval list is null. For each pair of nodes s and t pointed to by consecutive records r and r' in v 's left retrieval list, we fill in the *partner* field of r' by walking down the right retrieval list of s to the record that points to t . That record is the *partner* of r' . Setting all the *partner* fields takes time proportional to the height of v . When we walk down the right retrieval list of s looking for a record that points to t , we see records that point to nodes with distinct heights less than that of s and no smaller than that of t . Overall, the number of records we examine while setting *partner* fields is less than the length of the path from v to u .

We now tell how to recognize the nodes on the path from v to u that have edges in v 's chain. Our approach is based on comparing pairs of diagonals or polygon edges using the *hiding test* described below. Let t be a node in the tree rooted at v . If $t.tan$ does not appear in v 's chain, there is some edge $\overline{ab}$ in v 's chain such that $t.tan$ is contained in the pocket defined by $\overline{ab}$ and the polygon boundary between a and b . We say that $\overline{ab}$ *hides* $t.tan$. See Figure 6.8.

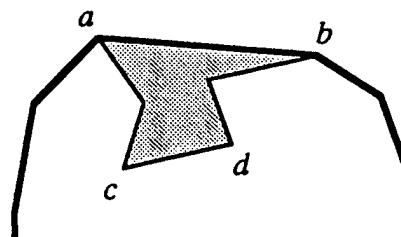


Figure 6.8. Segment $\overline{ab}$ hides $\overline{cd}$

We can test whether one edge hides another by comparing the indices of their endpoints, assuming all the endpoints are polygon vertices. Suppose that the edge connecting vertices i and l belongs to v 's chain, and i is left of l on the chain. If

$i < l$, the polygon vertices are numbered consecutively along the boundary of the pocket cut off by $\overline{il}$. In this case, an edge $\overline{jk}$ is hidden by $\overline{il}$ if and only if $\overline{jk} \neq \overline{il}$ and both integers j and k lie in the interval $[i, l]$. If $i > l$, an edge $\overline{jk}$ is hidden by $\overline{il}$ if and only if $\overline{jk} \neq \overline{il}$ and both j and k are outside the interval $[l + 1, i - 1]$.

At query time, not all chain vertices are polygon vertices: the query points p and q appear in some chains. Points p and q are not necessarily polygon vertices, so we must assign them vertex numbers before using the hiding test. Suppose that source point p lies in triangle Δijk and that d_p is diagonal $\overline{ik}$. (See Figure 6.9.) We construct the hourglass from p to q by pretending p is a polygon vertex (it is as if the darkly shaded part of the polygon were absent). We assign to p the vertex number j . This gives a consistent counterclockwise numbering to the vertices of the unshaded subpolygon and allows us to test hiding as described above.

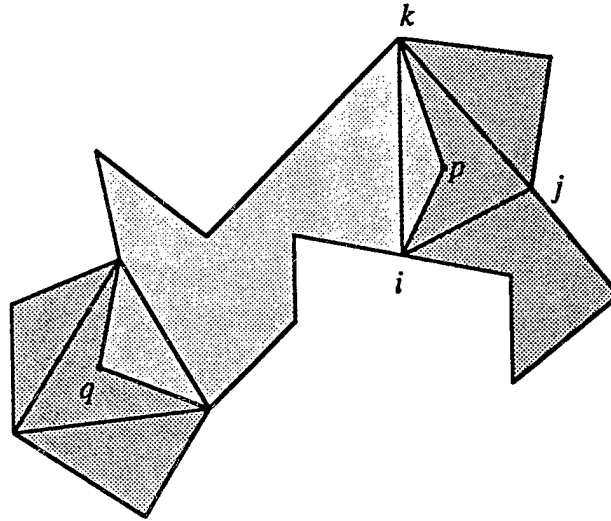


Figure 6.9. We assign vertex numbers to p and q , and hence are able to use the hiding test at query time.

The following lemma shows how to use the hiding test to construct retrieval lists.

Lemma 6.7. *Let u be the node in the left subtree of v such that $u.tan$ is adjacent to $v.tan$ in v 's chain, and let t be a descendant of v on the path from v to u . Let s*

be the lowest proper ancestor of t such that $s.tan$ appears in v 's chain. Either $t.tan$ is hidden by $s.tan$ or by $v.tan$, or it appears in v 's chain.

Proof: If $t.tan$ does not appear in v 's chain, it is hidden by an edge belonging to one of its ancestors. If $s = v$, then $v.tan$ is the only edge that can hide $t.tan$. If $s \neq v$, Lemma 6.4 implies that t is in the right subtree of s . By the same lemma, s is in the right subtree of all its ancestors that appear in v 's left retrieval list. The edges of those ancestors cannot hide $t.tan$: they are left of $s.tan$ and $t.tan$ is not. The only remaining ancestors of t whose edges may hide $t.tan$ are s and v .

■

To determine which nodes on the path from v to u have edges in v 's chain, we walk down the path testing each one. Suppose t is the current node and s is its lowest proper ancestor such that $s.tan$ appears in v 's chain. If $t.tan$ is not hidden by $s.tan$ or $v.tan$, then it appears in v 's chain.

We have now described all the elements of the chain data structure. A chain is represented by the root node of a binary tree; the node is augmented with edge retrieval lists to help report the edges of the chain. Including its retrieval lists, each node uses storage proportional to the height of its tree. We have shown how to construct a chain's tree from its subchains' trees in time proportional to the height of the new tree. The new chain's data structure does not modify any of the old chains' data structures. Taken together, these facts establish the following analogues to requirements (2) and (4):

- (C2) If the outer common tangent of chains $\pi(A, B)$ and $\pi(B, C)$ can be found in time τ , the data structure for their combined chain $\pi(A, C)$ can be built in $O(\tau)$ time and additional space. Furthermore, the data structures for the original two chains are unaltered by the construction.
- (C4) The length of a chain can be found in constant time. The edge sequence of the chain can be extracted in constant time per edge.

6.3.3 Hourglass data structures: implementation

This subsection uses the chains of Section 6.3.2 to implement the *hourglass* data structure. The hourglass data structure satisfies the requirements enunciated at the beginning of this section, and hence it can be used to answer shortest path queries efficiently. Hourglasses have two main components: chains and strings. We have described the chain data structure in the preceding subsection; we begin this subsection with a discussion of strings.

If we were concerned only with the lengths of shortest paths and not with the paths themselves, we could represent strings by their lengths alone. However, we need additional structures to represent the edges of a string. Our representation uses two types of strings, *fundamental* and *derived*. Fundamental strings are formed when the concatenation of two hourglasses collapses the multiple paths represented by funnels or open hourglasses into a single path. A fundamental string consists of two convex chains linked by a tangent edge; any of the pieces may be null. See Figure 6.10 for examples. Derived strings are obtained by concatenating two or more previously defined strings (either fundamental or derived). A derived string is produced when a closed hourglass is concatenated with another hourglass.

The chains that appear in strings are obtained by truncating hourglass chains. We can represent truncated chains using the data structure of Section 6.3.2. To clip off the right end of a chain, we simply create a new chain in which the old chain is the left subchain, the right subchain is null, and the tangent edge is the new chain's right endpoint.

A derived string, when fully expanded, is a sequence of fundamental strings. The data structure for a derived string represents this sequence as the leaves of a tree. A derived string is represented by an internal node whose children are its constituent strings. The tree has two or three children per node. See Figure 6.11.

We construct the derived string trees so that the non-empty fundamental strings that make up a derived string can be extracted in constant time apiece. When concatenation would cause us to include an empty string in a derived string, we drop that string. We also rule out nodes in the tree representation of derived

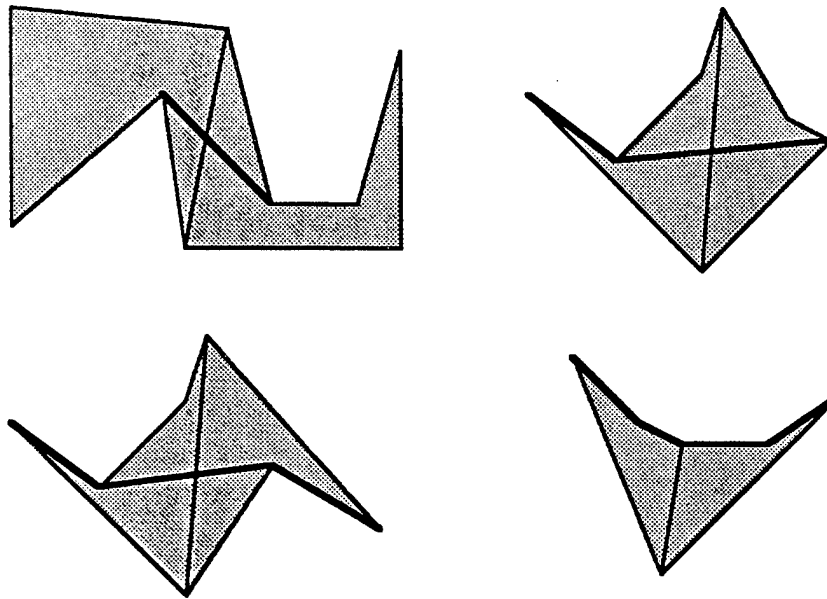


Figure 6.10. Examples of fundamental strings

strings that have only one child. That is, when two hourglasses are concatenated and the resulting string is just the string belonging to one of the two, we set the new hourglass to point to the old string, rather than creating a new derived string node with only one child. This strategy means that the internal nodes of a derived string tree have at least two children. Traversing the tree takes time proportional to the number of leaves (non-empty fundamental strings). Since the edges of each fundamental string can be extracted in constant time per edge, the edges of any string can be extracted within the same time bound. This concludes our discussion of the string data structure.

An hourglass is built of at most four convex chains and one string. The hourglass data structure has pointers to these five components. If any of the parts is missing, as when the hourglass is open, the corresponding pointer is null. To concatenate two hourglasses, we find the relevant common tangents of their chains, build the appropriate chains and strings, and set the pointer fields of a new hourglass structure. Each new chain is either an old chain or a convex hull of two old chains,

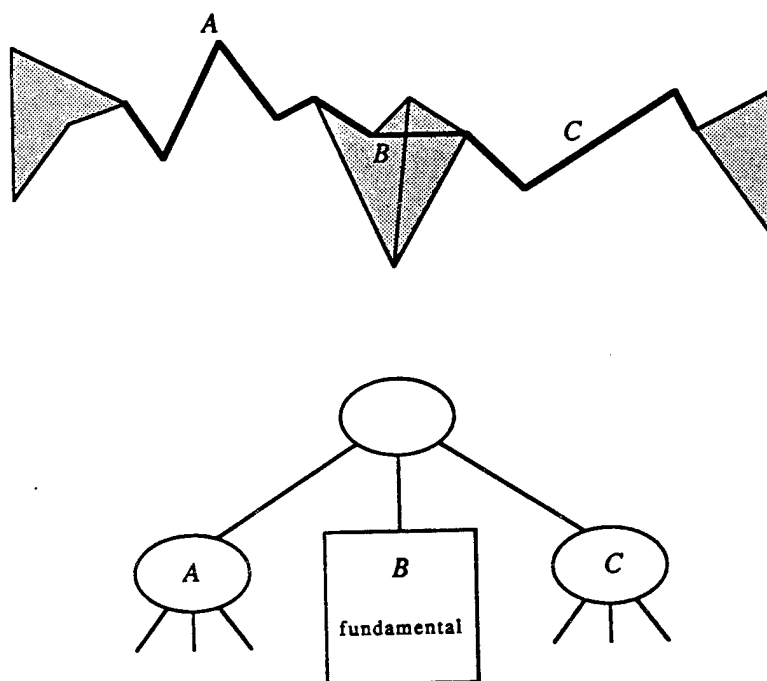


Figure 6.11. A derived string and its data structure

possibly with an end clipped off as described above. The new string, if any, is either a fundamental string or a derived string that points to one new fundamental string and one or two old strings.

We can conclude from the preceding discussion that the hourglass data structure meets its specification. Each hourglass and string record uses a constant number of chains, and each concatenation requires a constant number of chain operations. Because chains satisfy conditions (C1)—(C4), the hourglasses built from them satisfy requirements (1)—(4).

6.4 Improved query time bounds

Bypasses are devices that allow some people to dash from point A to point B very fast while other people dash from point B to point A very fast. People living at point C, being a point directly in between, are often given to wonder what's so great about point A that so many people from point B are so keen to get there, and what's so great about point B that so many people from point A are so keen to get there. They often wish that people would just once and for all work out where the hell they wanted to be.

— Douglas Adams, *The Hitchhiker's Guide to the Galaxy* (1979)

The data structures described above give an $O(\log^2 n)$ time bound for shortest path queries. This section shows how to cut a logarithmic factor off that bound. The idea behind the improvement comes from the following observation: if many shortest path queries are answered, hourglasses corresponding to edges high up in the factor graph are used many times; furthermore, the cost of concatenating them is the major contribution to query time. If we could find a way to bypass most of

the computation high up in S^* , we could reduce query costs.

Finding the shortest path from p to q has four main steps: (1) locating the points in the triangulation, (2) finding the subsequences D_p and D_q of the separating diagonals, (3) finding the hourglass between d_p and d_q , and (4) concatenating $H(p, d_p)$ and $H(d_q, q)$ with $H(d_p, d_q)$. Of these steps, only the third takes more than logarithmic time. To speed up step (3), we precompute some of the intermediate results it uses.

The decomposition tree S is balanced, and hence there are only $O(n/\log^2 n)$ nodes in S with at least $\alpha \log^2 n$ descendants. (The constant α is a parameter that can be adjusted to trade increased preprocessing for decreased query time.) Let the set of these upper nodes be U . Modify the graph S^* by adding to it all edges from nodes of U to their ancestors. Each node of U has $O(\log n)$ ancestors, so the modification adds $O(n/\log n)$ edges to S^* . The hourglasses corresponding to the additional edges can be computed in $O(\log n)$ time apiece, since the data structures that represent them have logarithmic height. (Each node d of U must be linked to all its ancestors in S . Linking to the ancestors in order along the path from d to the root of S gives the claimed bounds.) Overall, the additional hourglasses take linear time and space to construct.

These additional hourglasses provide the necessary bypass structures to cut a logarithmic factor off the shortest-path query time. Let p and q be the query points. In logarithmic time we find the subsequences D_p and D_q of the separating diagonals. The diagonal $\hat{d}$ is the least common ancestor in S of d_p and d_q . If $\hat{d}$ is not in U , then the cells between d_p and d_q have $O(\log^2 n)$ edges altogether, since they are contained in $P_{\hat{d}}$. There are only $O(\log(\log^2 n)) = O(\log \log n)$ hourglasses that need to be concatenated in order to produce $H(d_p, d_q)$, and each concatenation takes $O(\log \log n)$ time. Thus step (3) takes $O((\log \log n)^2)$ time, which is dominated by the time of the other steps.

If $\hat{d}$ is in U , then the situation is slightly more complex. Let d_p^- be the highest diagonal in D_p that is not in U , and let d_p^+ be the next higher diagonal in D_p (the successor of d_p^- in D_p). Define d_q^- and d_q^+ similarly. (See Figure 6.12.) As in the case when $\hat{d} \notin U$, we can find $H(d_p, d_p^-)$ and $H(d_q^-, d_q)$ in $O((\log \log n)^2)$

time. The hourglasses $H(d_p^-, d_p^+)$ and $H(d_q^+, d_q^-)$ correspond to edges of S^* , and hence have been precomputed. Since $\hat{d}$ is an ancestor in S of d_p^+ and d_q^+ , and all three diagonals are in U , the hourglasses $H(d_p^+, \hat{d})$ and $H(\hat{d}, d_q^+)$ also already exist. Concatenating these six hourglasses to get $H(d_p, d_q)$ takes logarithmic time, and therefore the length of the shortest path from p to q can be found in $O(\log n)$ time.

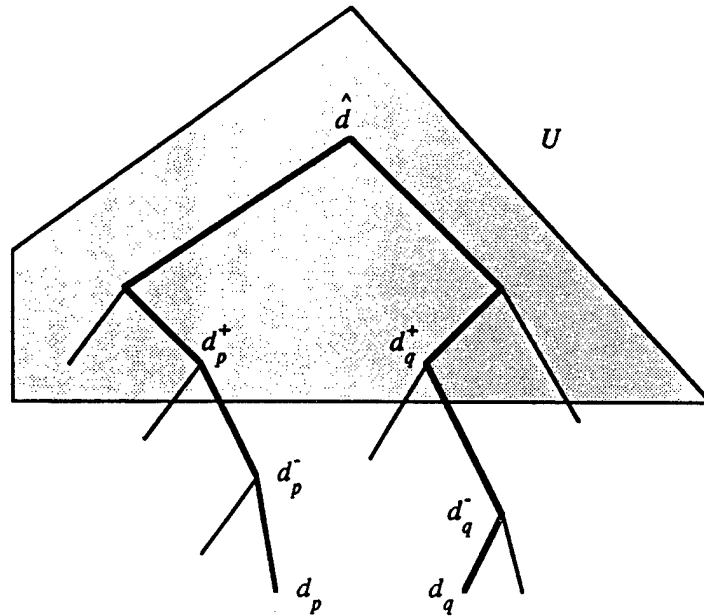


Figure 6.12. Definitions for query speedup

Theorem 6.1. *There is a linear space data structure for storing a simple polygon P of size n so that given any two points inside P , the length of the shortest path joining them can be computed in $O(\log n)$ time. The actual path can be extracted in additional time proportional to the number of turns along it. Furthermore, this shortest-path query structure for P can be computed in linear time once a triangulation of P is available.*

6.5 Applications and extensions

Our method can yield improved bounds for various questions related to relative convex hulls [BT86]. For example, given a simple polygon P of n sides, we can preprocess its exterior in $O(n \log \log n)$ time and build the factor graph for each bay cut off by the convex hull of P . Now, given another simple polygon Q of m sides and disjoint from P , we can compute an implicit representation of the relative convex hull of Q with respect to P in time $O(m \log n)$. This implicit relative convex hull can be used to answer the moving separability question of Bhattacharya and Toussaint [BT86]. An explicit representation of the relative convex hull can be extracted in additional time proportional to the size of the hull.

A related question has to do with computing the relative convex hull of m points in a simple polygon P of size n . After preprocessing the polygon for shortest path queries, we can solve this problem in additional time $O(m \log m + m \log n)$. The important observation is that we can define a “relative” notion of the counterclockwise test for three points a , b , and c in P that is testable in $O(\log n)$ time using our structures. The test looks at the (implicit) description of the paths $\pi(a, b)$ and $\pi(a, c)$ and finds the place where these paths diverge; the relative directions of these paths at the juncture tell the sign of the test for (a, b, c) . One can prove that the outcome of this test is the same no matter which of the three points is used as the path origin. Note also that this test and the ordinary counterclockwise test (with no reference to P) can have opposite results on the same three points. With this tool at our disposal, we can proceed as follows to determine the relative convex hull: Locate all the m points in the triangles of the triangulation underlying the preprocessing. For each group of points in the same triangle compute its standard convex hull and discard all points not on the hull. Suppose a total of m' points remain. The cost of what we have done so far is the cost of the preprocessing for P , plus $O(m \log m)$ for all the convex hull computations. We can now connect these hulls by (implicit) non-crossing paths lying in P . See Figure 6.13. By doubling up these paths we can connect all m' points into one simple polygon Q lying inside P . The edges of this polygon are shortest paths in P ; also, some vertices may occur

up to three times along the boundary of Q . However, we can still use a “linear” convex hull algorithm on such a polygon, for the only tool such an algorithm needs is the counterclockwise test on three points discussed above. Of course each test now costs $O(\log n)$, for a total of $O(m \log n)$ for this part of the computation.

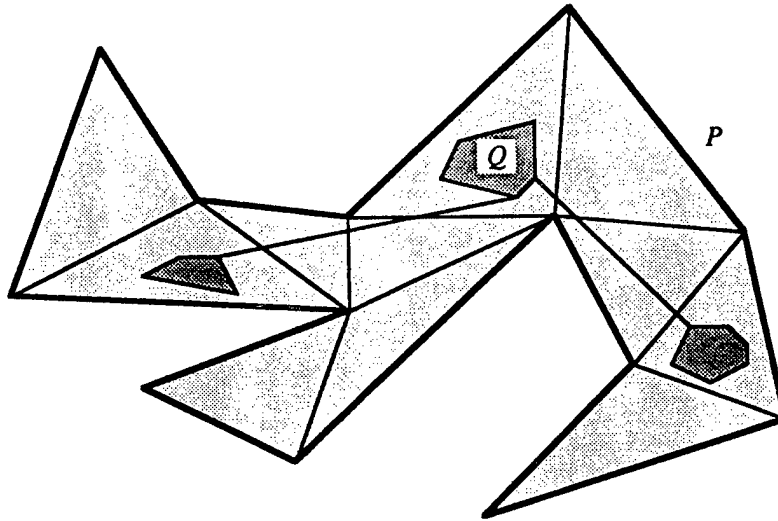


Figure 6.13. Computing the relative convex hull of some points in P

Another interesting application has to do with computing the furthest neighbor for all vertices of a simple polygon. This is analogous to the same question for a convex polygon considered in [AKM*86], only now we measure distance between vertices by the shortest path inside the polygon. By combining the approaches of [AKM*86] and the present chapter, we can perform this computation for a polygon of size n in total time $O(n \log n)$. A recent paper of Suri [Sur87] achieves the same bound using a different approach.

As a final observation, we note that the preprocessing described in Section 6.3.1 can be adapted to solve the *shooting problem* for P , as defined by Chazelle and Guibas [CG85, Section 3]. This problem calls for preprocessing P so that, given any point x inside P and any direction u , we can quickly compute the point $hit(x, u)$ where the ray emanating from x in direction u hits the boundary of P for the first time. If the polygon P has n sides, the method presented in [CG85] solves the

shooting problem in $O(\log n)$ time per query, after building a structure in $O(n \log n)$ time that requires $O(n)$ space. We adapt our algorithm to build the shooting structure used by [CG85] in linear time.

The shooting structure of Chazelle and Guibas uses hourglasses similar to those of this chapter to represent all lines of visibility between two diagonals of P . Because shooting is a visibility problem, only open hourglasses are important. In fact, not all edges of an open hourglass are used. The structure of [CG85] stores only those hourglass edges that affect the lines of visibility passing through the hourglass, that is, the hourglass edges that are visible from both ends of the hourglass. The shooting structure contains exactly the same hourglasses as our $O(\log^2 n)$ query data structure: it has an hourglass corresponding to each edge in the factor graph S^* . However, the hourglasses are not represented in the same way as ours. Instead of storing only the new tangent edges with an hourglass formed by concatenating two old ones, the shooting structure stores some old edges with the new hourglass. Each hourglass edge is allocated to, and stored only at, that hourglass in S^* that has it as an edge and is the last one formed during the construction algorithm of Section 6.3.1. This means that at each stage, as we compute the hourglass of (d_1, d_3) from the hourglasses of (d_1, d_2) and (d_2, d_3) , we need to find the common tangents and then split the old hourglasses where the common tangents touch them. The extremal hourglass pieces get joined by the tangent to form the new hourglass, while the inner pieces are left associated with the old hourglasses. See Figure 6.5(a) for an illustration.

As in Section 6.3.1, the general step of the merging process concatenates an hourglass H with several others. If some edges of H had to be duplicated to appear in more than one of the resulting hourglasses, the shooting structure might be superlinear in size. However, the restriction that hourglasses contain only edges visible from both bounding diagonals means that each edge of H belongs to at most one of the resultant hourglasses, so this problem does not arise. The splitting and joining operations can be implemented in time of the same order of magnitude as that of the common tangent computation, if a balanced tree structure is used to represent the hourglasses. In this way, continuing exactly as in [CG85], the entire

shooting structure can be computed in linear time and space. We therefore have the following result:

Theorem 6.2. *Given a triangulated simple polygon P of n sides, it is possible to construct, in linear time and space, an auxiliary structure with which the shooting problem for P can be solved in $O(\log n)$ time per query. These bounds are optimal.*

*I shot an arrow into the air,
It fell to earth, I knew not where;
For, so swiftly it flew, the sight
Could not follow it in its flight.*

— Henry Wadsworth Longfellow, “The Arrow and the Song”

*I shot an arrow into the air,
It fell to earth, I knew not where;
Until next day, with rage profound,
The man it fell on came around.
He showed me where that arrow fell
In less time than it takes to tell.
And now I do not greatly care
To shoot more arrows in the air.*

— Anonymous

Chapter 7

Conclusions and Continuations

In my end is my beginning.

— T. S. Eliot, *Four Quartets* (1943)

In this chapter we discuss possible extensions and further applications of the ideas used in the preceding chapters.

The foundation for many of the results of this thesis is the close relationship between shortest path and visibility problems. This connection is a fundamental one, similar to geometric duality [CGL85] in some respects. Algorithms that solve one type of problem often have solutions to the other type as easy corollaries. Chapters 4 and 5 especially illustrate the use of this connection. I expect this relationship to bear fruit in future work.

A theme of increasing importance in computational geometry is the idea of measuring algorithmic complexity in a way that depends on the data. The algorithm of Chapter 2 builds the visibility graph of segments in $\Theta(n^2)$ time and space. When the algorithm is used to solve shortest path problems, this gives a $\Theta(n^2)$ shortest path algorithm. Algorithms such as that of Reif and Storer [RS85] or those of

Chapters 4 and 6, which take into account the geometric relationships between the segments, can dramatically improve this bound. As noted in Chapter 2, the visibility graph itself can be as small as $O(n)$. It is therefore worthwhile to look for algorithms that build it in time proportional to its size. Chapter 5 solves the problem for the special case in which the segments form a simple polygon.

Chapters 4 and 6 enlarge the collection of problems solvable in linear time for triangulated simple polygons; I expect many additional problems for such polygons to have linear-time solutions. For example, Suri [Sur86a] has recently extended the techniques of Chapter 4 to solve in linear time the k -visibility problem, in which, given a simple polygon P and an edge e of P , we wish to partition P into disjoint subparts $P_1, P_2, \dots$ such that P_1 contains all points in P directly visible from some point on e , P_2 contains all points in P visible from some point in P_1 but not from e , and so on. This result and those of Chapters 4 through 6 emphasize the centrality to planar computational geometry of the question of whether there exists a linear-time triangulation algorithm for simple polygons.

Finally, I expect the techniques of Chapter 6 to find wider application. The decomposition provided by Chazelle's polygon cutting theorem [Cha82] is ideally suited to answering queries that relate one part of a simple polygon to another, possibly quite distant part. I believe that shortest path queries and shooting queries are only the first two query types discovered that fit this pattern. The edge retrieval lists of Section 6.3.2 may also be applicable in other settings.

Appendix A

Balanced Decomposition of a Binary Tree in Linear Time

The woods decay, the woods decay and fall.

— Alfred, Lord Tennyson, *Tithonus*

This appendix shows how to compute the balanced hierarchical decomposition of the triangulated polygon P that was described in Section 6.1. The decomposition is based on the polygon cutting theorem of Chazelle, which states that any polygon has a diagonal that splits it into two subpolygons of roughly equal size [Cha82]. The algorithm presented here finds a decomposition of P by operating on the tree T that is dual to the triangulation of the polygon. Deleting an edge of T splits T in two and corresponds to adding a particular diagonal to P . If we can find an edge whose removal splits T into subtrees of roughly equal size, this will give us the desired splitting diagonal for P . In the remainder of this appendix we focus our attention on decomposing the tree T ; we can translate a decomposition of T directly into a decomposition of the polygon P .

We now restate some of these ideas formally. Let T be a rooted binary tree with n nodes. (When T is the dual of a triangulation, we can pick any node with at most two neighbors to be the root.) If an edge e of T is removed, then T is partitioned into two subtrees. If we now similarly partition each of these subtrees and continue doing this recursively until the fragments left are single nodes, we obtain another tree structure, which is known as a *decomposition* of T . Such a decomposition is called *balanced* if there is a positive constant α such that each time a subtree F is partitioned by the removal of an edge, each of the two fragments obtained has size at least $\alpha|F|$, where $|F|$ denotes the size (number of nodes) of F . We make two remarks on such decompositions. First, it is clear that in a balanced decomposition the fragment containing a particular node v can be split only $O(\log n)$ times. Second, it is known that in any binary tree there is an edge whose removal leaves two components, each with at least $\lfloor (n+1)/3 \rfloor$ nodes. There is always such an edge incident to a *centroid* of the tree. (If we define the weight of a vertex v to be the size of the largest subtree that remains after v and its incident edges are deleted, a centroid is a vertex of minimum weight. There are at most two centroids, and if there are two, they must be adjacent [Knu73, pages 387–388].) A splitting edge incident to a centroid, called a *centroid edge*, can be found in linear time [Cha82]. As a result, we can find a balanced decomposition of T with $\alpha = 1/4$ in $O(n \log n)$ time total by recursively applying the centroid partition to each fragment. Note that $1/4$ is just a lower bound on α . In general, the balance parameter will be close to $1/3$: for $n \geq 4$, $\alpha \geq 1/3 - 1/(3n)$.

We show in this appendix how a centroid edge for partitioning each fragment can be computed in less than linear time, after some appropriate data structures have been set up. As a result, we are able to reduce the overall time for obtaining a balanced decomposition to $O(n)$. Our method makes use of an auxiliary ternary tree A^T , for brevity written A , to facilitate the splitting. The auxiliary tree A lets us find in $O(\log^2 n)$ time an edge at which T may be split. Moreover, in the same time bound, we can break A into two auxiliary trees, one for each of the two fragments of T resulting from the split. By applying this process recursively we obtain the desired balanced decomposition of T .

The remainder of this appendix has three parts. The first part defines and characterizes the auxiliary tree A associated with a binary tree T . Section A.1 shows how to construct the auxiliary tree in linear time, given T as input. Finally, Section A.2 explains how to use the auxiliary tree to find a balanced decomposition of T .

To describe our approach we adopt the following conventions: The notation T_v refers to the subtree of T rooted at v . The descendants of v include v itself. Also, all logarithms are base two.

The auxiliary tree A has the same nodes as T , but different edges. It provides a way to search for a node in T without traversing many edges of T . An analogy will make the idea clear: If we are given an ordered list and want to have fast access to its elements, we can build a symmetrically ordered balanced binary tree with the list elements stored at the nodes. The tree A plays the same rôle for T as the binary search tree does for the list; we now compare and contrast the two search trees. The height of A , like the height of the binary tree, is proportional to the logarithm of its size. Because each element in the list case has at most two neighbors, a binary search tree gives fast access to the list; each node in T has up to three neighbors, so in our case A is a ternary tree. The binary tree implements a form of binary search: to locate a list element, we look at the element r at the root of the tree, decide whether the desired element lies to the right or left of r (if it is not equal to r), and recursively search in one of the subtrees if necessary. The ternary tree A facilitates a similar search: to locate a particular node of T , we look at the node a at the root of A , decide which of the three subtrees of T incident to a contains the desired node (if a is not it), and recursively search in the auxiliary tree for that subtree of T . We can define the binary search tree based on numbering the list elements consecutively from 1 to n . The element whose number is divisible by the highest power of two goes at the root of the search tree. The subtrees for its left and right sublists are built recursively using the same rule. (See Figure A.1.) The construction of A described below is based on the same idea, but the numbering scheme is more complicated. Because T is not necessarily a linear list, we cannot simply number its nodes from 1 to n . Instead, we assign numbers to each node v

roughly based on the size of the subtree T_v . These numbers (the labels b_v described below) are calculated during a postorder traversal of T .

To define the auxiliary tree A , we attach pairs of integers to the nodes of T . We associate with each node v of T an integer index i_v and an integer label b_v . The index i_v is the height of v in A and also equals the number of trailing zeroes in b_v written in binary. Formally, b_v is defined by the following rules, which we call the *carry algorithm*: If v is a leaf of T , then $b_v = 1$. If v has only one child in T , say w , then $b_v = b_w + 1$. If v has two children in T , say w and z , the label b_v is constructed as follows: Let i be the bit position of the leftmost carry in the binary computation of $b_w + b_z + 1$ (bits are numbered right-to-left, with bit 0 as the rightmost bit). One can easily check that the addition order does not matter. If no carry occurs, as when $b_w = 4$ and $b_z = 2$, then we set $i = 0$, as if the added 1 were a carry. The bits of b_v are equal to those of $b_w + b_z + 1$ at and to the left of position i . To the right of position i all bits of b_v are 0. Note that since no carry arises to the left of position i , the bits of b_v left of position i can be obtained by adding the corresponding portions of b_w and b_z . Note also that the i -th bit of b_v must be 1, and so i_v is equal to i .

An example will help make the definition clear. Suppose that v has two children w and z , and that $b_w = 45$ and $b_z = 71$. Then b_v is obtained as follows:

$$\begin{array}{rcl}
 & & \text{leftmost carry} \\
 b_w & = & 45 = (0101101)_2 \\
 b_z & = & 71 = (1000111)_2 \\
 & & + \quad \quad \quad 1 \\
 \hline
 b_w + b_z + 1 & = & 117 = (1110101)_2 \\
 b_v & = & 112 = (1110000)_2
 \end{array}$$

Note that this definition also applies when v has one or zero children: we simply treat the labels of the missing children as zero.

If we use bitwise logical operations, we can compute b_v in constant time. However, bitwise operations are not necessary: Section A.1 shows how to compute labels for all the nodes of T in linear time without using bitwise operations.

When T is a path with the root at one end, as in Figure A.1, the labels run consecutively from 1 to n , and the indices give the “ruler function.” A key property

of the ruler function is that two nodes with equal indices are separated by a node with a higher index. We say that the indices assigned by the ruler function have the *separation property*:

The separation property: *If $i_v = i_w$ for $v \neq w$, then there is a node u on the path from v to w with $i_u > i_v$.*

Note that except for notational differences, the separation property is identical to the *lacuna property* used in Chapter 6. We show below that the indices assigned by the carry algorithm have the separation property not just when T is a path, but also when it is an arbitrary binary tree. However, before we prove that the indices have the separation property, we show why the separation property is desirable.

The separation property is closely related to decomposition by vertex-deletion. The following lemma makes the connection between separation and decomposition:

Lemma A.1. *Let T be a binary tree whose nodes have been assigned indices that satisfy the separation property. If all nodes with indices greater than some positive j are removed from T along with their incident edges, each remaining subtree of T has exactly one node with maximum index.*

Proof: We prove the lemma by proving a stronger result: If we break T into subtrees by deleting any collection of edges, each subtree has a unique node with maximum index. Suppose to the contrary that some subtree has two nodes v and w with maximum index. By the separation property there is a node u on the path between them with $i_u > i_v$. Every node on the path belongs to the subtree, so i_v cannot be the maximum index. ■

Because the indices assigned by the carry algorithm have the separation property, we can use node indices to build the auxiliary tree as follows: The single node a with highest index is the root of A . Removal of this node and its incident edges from T generates at most three subtrees in T , which recursively define the (at most) three subtrees in A of the root node. We can use the resulting tree A to search in T because the nodes of any subtree A_v are contiguous in T . Figures A.1 and A.2 give

two examples of the indices and labels assigned by the carry algorithm and their use in building the auxiliary tree A.

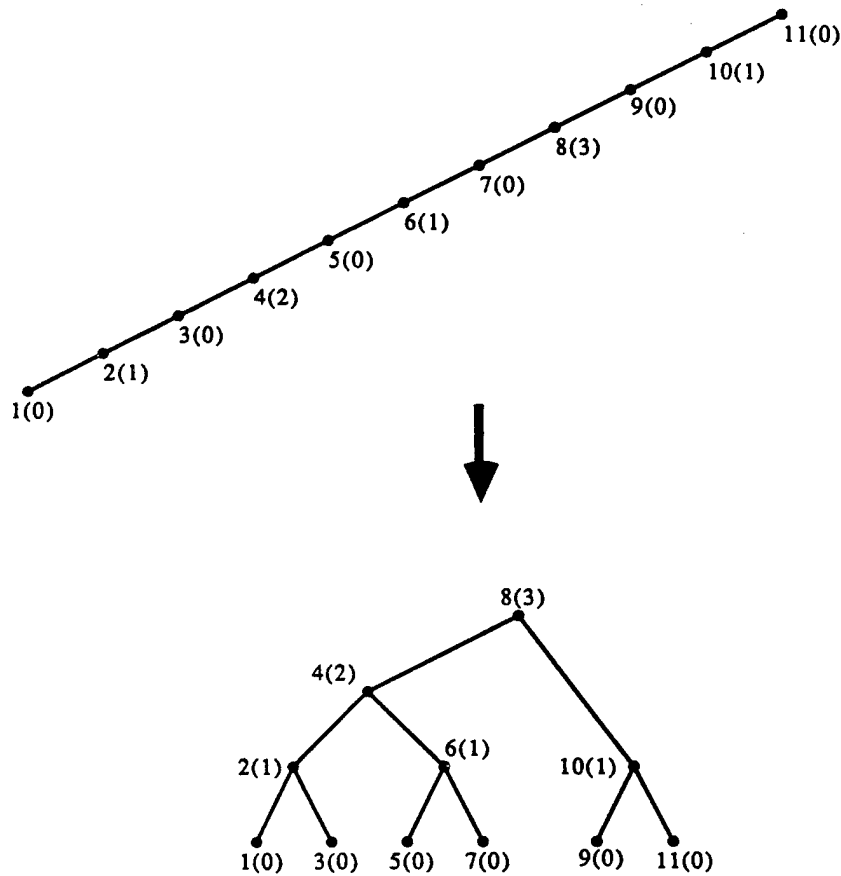


Figure A.1. The nodes of the upper tree (a single path) are labelled with $b_v(i_v)$. The indices i_v give the “ruler function.” The lower tree is the auxiliary tree built using the labels.

We now show that the indices assigned by the carry algorithm have the separation property. To do this, we attach an interpretation to the labels and indices. We begin by treating b_v as a bit vector: $b_v = \sum_{j \geq 0} 2^j b_v[j]$, where $b_v[j] = 0$ or 1 . If v is a leaf of T , then $b_v = 1$; that is, $b_v[0] = 1$ and $b_v[j] = 0$ for $j > 0$. The index i_v is 0 . If v has children w and z , then the “leftmost carry” definition of i_v is equivalent to

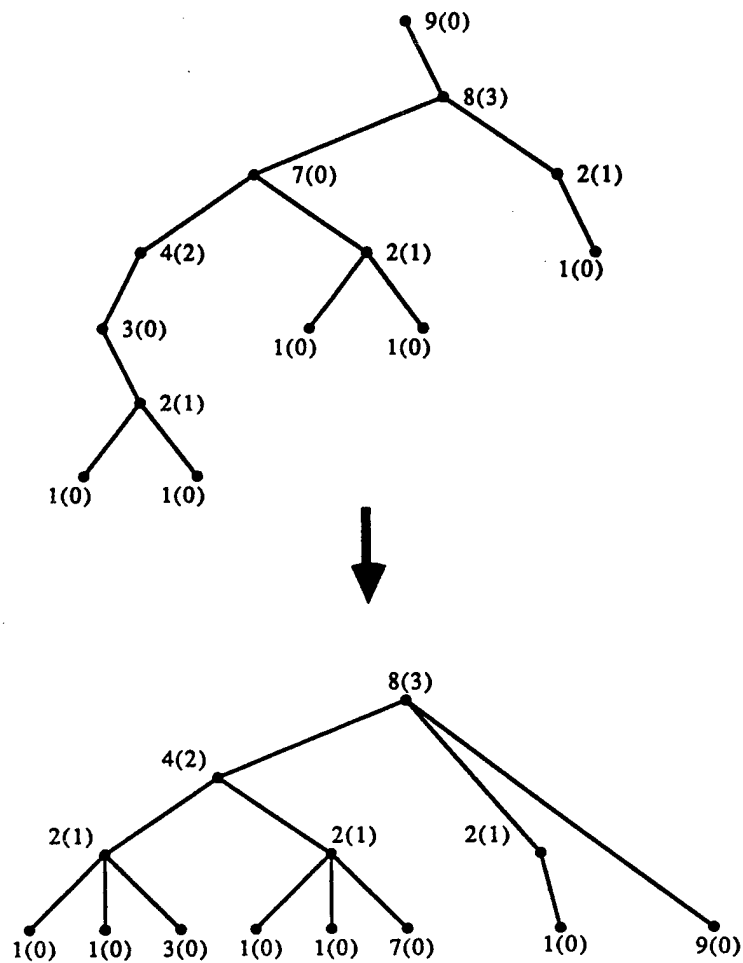


Figure A.2. The nodes of the upper tree (an unbalanced binary tree) are labelled with $b_v(i_v)$. The lower tree is the auxiliary tree built using the labels.

the following definition:

$$i_v = \min \{j \geq 0 \mid b_w[j] = b_z[j] = 0 \text{ and } b_w[k] \cdot b_z[k] = 0 \text{ for all } k > j\}. \quad (\text{A.1})$$

(If v has only one child, say w , then $b_z[j] = b_z[k] = 0$ in this expression.) Once i_v is known, b_v has a simple definition:

$$b_v[j] = \begin{cases} b_w[j] + b_z[j] & \text{if } j > i_v \\ 1 & \text{if } j = i_v \\ 0 & \text{if } j < i_v. \end{cases} \quad (\text{A.2})$$

To characterize the labels and indices we introduce the idea of *path indices*. The index of a path in T is defined to be the maximum index of all interior nodes on the path (that is, excluding the two end nodes of the path), or -1 if the path contains no interior nodes (it consists of a single edge or even of a single vertex). We define $\text{ind}(u, v)$ to be the index of the path joining two nodes u and v . The separation property can be restated in terms of path indices as follows: Any two nodes of equal index are connected by a path with greater index. The following lemma characterizes the labels b_v using path indices. The lemma refers to descendants of v ; recall that the descendants of v include v itself.

Lemma A.2. *Let T be a binary tree labelled by the the carry algorithm. For each node v and each $j \geq i_v$, v has at most one descendant u such that $i_u = j$ and $\text{ind}(u, v) < j$. Furthermore, $b_v[j] = 1$ if and only if such a descendant exists.*

Proof: The proof is by induction on the height of v . When v is a leaf, $b_v = 1$ and $i_v = 0$. The only descendant of v is v itself, and the path index $\text{ind}(v, v)$ is -1 , which is less than i_v . To prove the second statement, note that $b_v[j] = 1$ only for $j = i_v = 0$.

Now suppose that v has two children, w and z . We consider $b_v[j]$ for $j \geq i_v$. Whenever $b_w[j] = b_z[j] = 0$ for $j \geq i_v$, v has no strict descendant u with $i_u = j$ and $\text{ind}(v, u) < j$. To see this, let u be a descendant of w with $i_u = j$; because $b_w[j] = 0$, we have $u \neq w$ and $j \neq i_w$. Hence $\text{ind}(v, u) = \max(i_w, \text{ind}(w, u))$. If $j < i_w$, we have $\text{ind}(v, u) \geq$

$i_w > j$. On the other hand, if $j > i_w$, the induction hypothesis implies that $\text{ind}(w, u) \geq j$, and hence $\text{ind}(v, u) \geq j$.

Whenever $b_v[j] = 0$ for $j > i_v$, equation A.2 above implies that $b_w[j] = b_z[j] = 0$. Therefore v has no descendant u such that $i_u = j$ and $\text{ind}(v, u) < j$. If $j = i_v$, we have $b_v[j] = 1$, but $b_w[j] = b_z[j] = 0$. In this case v has no strict descendant u with $i_u = i_v$ and $\text{ind}(u, v) < i_v$, but v itself is a descendant of v that meets these conditions.

If $b_v[j] = 1$ for some $j > i_v$, then exactly one of $b_w[j]$ and $b_z[j]$ is 1; without loss of generality suppose $b_w[j] = 1$. By induction, every descendant x of z with $i_x = j$ has $\text{ind}(z, x) \geq j$, and therefore $\text{ind}(v, x) \geq j$. Also by induction, exactly one descendant u of w has $i_u = j$ and $\text{ind}(w, u) < j$. If $u = w$, then $\text{ind}(v, u) = -1$, which is less than j . On the other hand, if $u \neq w$, then $j = i_u > i_w$ and $\text{ind}(v, u) = \max(i_w, \text{ind}(w, u)) < j$. In either case $\text{ind}(v, u) < j$. Node u is the only descendant of v with $i_u = j$ and $\text{ind}(v, u) < j$.

If v has only one child, say w , the definition of i_v in equation A.1 reduces to $i_v = \min \{j \geq 0 \mid b_w[j] = 0\}$. The arguments needed to prove the lemma in this case are similar to those used above and are left to the reader. ■

We use the preceding lemma to show that the indices produced by the carry algorithm have the separation property.

Lemma A.3. *Let T be a binary tree labelled using the carry algorithm. The indices of the nodes of T have the separation property.*

Proof: Let x and y be two nodes of T such that $i_x = i_y$. The proof splits into two cases depending on whether one of the two nodes is an ancestor of the other.

First suppose that one of the nodes, say x , is an ancestor of the other. If any interior nodes on the path joining x and y have index i_x , let u be

the one closest to x ; otherwise let $u = y$. Because $\text{ind}(x, x) = -1$ is less than i_x , Lemma A.2 implies that $\text{ind}(x, u)$ is at least i_x . Since no node with index i_x lies strictly between x and u , we have $\text{ind}(x, u) > i_x$, and hence $\text{ind}(x, y) > i_x$.

If neither x nor y is an ancestor of the other, let v be their lowest common ancestor. We can write $\text{ind}(x, y) = \max(\text{ind}(x, v), i_v, \text{ind}(y, v))$. If $i_v > i_x$, we are done. If $i_v = i_x$, the argument of the first case shows that $\text{ind}(x, v) > i_x$. If $i_v < i_x$, we show that at least one of $\text{ind}(x, v)$ and $\text{ind}(y, v)$ is greater than i_x . Let w and z be the children of v , with w an ancestor of x and z an ancestor of y . Since $i_x > i_v$, at most one of $b_w[i_x]$ and $b_z[i_x]$ is 1; without loss of generality suppose that $b_w[i_x] = 0$. For any node u in T_w with $i_u = i_x$, we have $\text{ind}(w, u) \geq i_x$. An argument similar to the one above implies that $\text{ind}(w, u) > i_x$ for any such node u . Therefore $\text{ind}(x, w) > i_x$. The path from x to w is a subpath of the one from x to y , and so $\text{ind}(x, y) > i_x$. ■

The next lemma proves two more facts about the labels produced by the carry algorithm. It shows that the decomposition represented by A is height-bounded, although not necessarily balanced.

Lemma A.4. *Let T be a binary tree labelled by the carry algorithm. Then*

- (a) *for each node v , $|T_v| \geq b_v$, and*
- (b) *for each index j , there are at most $\lfloor |T|/2^j \rfloor$ nodes with that index in T .*

Proof: The proofs of both statements are inductive. The first follows easily from the carry definition of b_v . For a leaf v , $|T_v| = b_v = 1$. Since $b_v \leq b_w + b_z + 1$, induction implies that $|T_v| = |T_w| + |T_z| + 1 \geq b_v$.

To prove (b), let us define $D(v)$ to be the set of the descendants of v whose indices are at most i_v and which are joined to v by paths of index less than i_v . Because of the separation property, the sets $D(v)$ and $D(w)$ are disjoint for distinct v and w with $i_v = i_w$: If $T_v \cap T_w$ is empty, then so

is $D(v) \cap D(w)$. If $T_v \cap T_w$ is nonempty, without loss of generality assume that v is an ancestor of w . By the separation property, $\text{ind}(v, w) > i_v$, and hence $\text{ind}(v, u) > i_v$ for any $u \in D(w)$.

We next show by an inductive argument that $|D(v)| \geq 2^{i_v}$. This is clearly true for v a leaf: $|D(v)| = 1 \geq 2^0$. If v has children w and z , then a node q is in $D(v)$ if either $q = v$ or q is a descendant of w or of z , say for definiteness a descendant of w , such that $i_q < i_v$, $i_w < i_v$, and $\text{ind}(w, q) < i_v$. In this latter case, let $j < i_v$ be the maximum node index along the path from w to q (including these two nodes), and let u be the unique node along this path with index j . Then clearly $q \in D(u)$ and $b_w[j] = 1$. It is also easy to check the converse statement, namely that $D(v)$ contains each set $D(u)$ for a descendant u of w or of z that causes $b_w[j]$ or $b_z[j]$ to be 1 for any $j < i_v$. But the sets $D(u)$ are all disjoint. Suppose to the contrary that $q \in D(u) \cap D(u')$; then without loss of generality we can assume that u is a strict descendant of u' and both are descendants of w . If $i_u \leq i_{u'}$, then u cannot have caused $b_w[i_u]$ to be 1, and if $i_u > i_{u'}$, then q cannot belong to $D(u')$. Thus, by the induction hypothesis,

$$|D(v)| \geq 1 + \sum_{j < i_v} 2^j (b_w[j] + b_z[j]) \geq 2^{i_v}$$

(because $1 + b_w + b_z$ has a carry at the i_v -th bit); and, since all the sets $D(v)$ for nodes with the same index i_v are disjoint, (b) follows. ■

Since i_a is the height of A , part (b) of the preceding lemma shows that A has height at most $\lfloor \log n \rfloor$.

A.1 Constructing the auxiliary tree A

It is possible to determine the indices of the nodes and build A during a single postorder (depth-first) traversal of T . The two trees T and A have the same node set. To distinguish the edge sets, we speak of the *edges* of T and of the *links* of A .

For each label b_v , the construction requires a vector p_v of pointers to nodes. If $b_v[j]$ is 1, then $p_v[j]$ points to the (unique) descendant of v in T that has index j and is reachable by a path of index less than j . The stack implicit in the depth-first search defines a path π from the root of T to the current node. The construction maintains b_v and p_v for each node v that is a child of a node on π but is not on π itself. Each such v is the root of a subtree T_v . The vectors b_v and p_v take $O(\log |T_v|)$ space. Since the subtrees T_v are all disjoint, the total space taken by all the vectors b_v and p_v is linear.

When the postorder traversal visits a node v , the algorithm constructs b_v and p_v from the vectors stored at the children w and z of v . At the same time, it builds auxiliary tree links for nodes that appear in p_w and p_z but not in p_v . Such a node (suppose it is u in T_w) has an index less than i_v , so its parent in A is either v or a node of T_w . In fact, its parent in A is the node with minimal index $j > i_u$ reachable from u by a path in T of index less than i_u . These observations imply that if j is the largest integer less than i_v such that $b_w[j] = 1$, then v is the parent of $p_w[j]$ in A . Similarly, if j and k are integers less than i_v such that $j < k$, $b_w[j] = b_w[k] = 1$, and $b_w[l] = 0$ for all l strictly between j and k , then $p_w[k]$ is the parent of $p_w[j]$ in A . Linking these nodes to their parents in A takes time proportional to i_v .

When the algorithm reaches the root t of T , it links the nodes pointed to by p_t as if t were the child of a node with index greater than that of any node in T . The root a of A is the highest-indexed node appearing in p_t .

Even without logical operations on the b_v vectors, the construction of A requires only linear time. When the traversal visits v (with children w and z), the algorithm takes time proportional to the number of links made plus the logarithm of the smaller of $|T_w|$ and $|T_z|$, which gives a linear overall bound. In linear additional time, a traversal of A can be used to compute $|A_v|$ and store it at each node v ; this information is needed in order to split the auxiliary tree later on.

A.2 Decomposing the tree T

Once we have constructed the auxiliary tree A , only a linear amount of additional work is needed to find a balanced decomposition of T . The algorithm uses A to split T into balanced subtrees, then builds an auxiliary tree for each fragment, and finally decomposes each fragment recursively. The general step of the algorithm has two parts, which we describe in turn: The first step uses the auxiliary tree to find an edge of T whose removal splits T into pieces of nearly equal size. The second step splits A into two new auxiliary trees, one for each piece of T .

The algorithm finds a splitting edge for T in time proportional to the square of the height of A . This edge is incident to the centroid of T ; we call the edge e_c . The algorithm finds e_c by first finding the centroid of T (one of the centroids if there are two), then picking the edge incident to it whose deletion from T would leave fragments of most nearly equal size.

Before we describe the search algorithm, we describe one of its component operations, finding the relative locations of two nodes of T . Removing a node v from T leaves at most three subtrees. Given a query node w not equal to v , we want to determine quickly which subtree contains w . If we compute preorder and postorder numbers for the nodes of T at the same time as we compute the labels and indices, we can answer such queries in constant time. For any two nodes x and y , x is an ancestor of y if and only if $Pre(x) \leq Pre(y)$ and $Post(x) \geq Post(y)$. The query node w is in the left (right) subtree of v if and only if it is a descendant of the left (right) child of v . If v is not an ancestor of w , then w lies in the third subtree, the “up” subtree of v .

To find the centroid of T , we search in A using the subtree sizes $|A_v|$. Deleting a node v and its incident edges in T splits T into at most three trees L , R , and U . If the centroid search visits v , it must determine the sizes of L , R , and U . If all the sizes are at most $|T|/2$, then v is the centroid. If one subtree is larger than $|T|/2$, then the centroid lies in A_w , where w is the child of v in A that belongs to the large subtree. (This is easy to prove inductively.) When the search descends from v to w , it eliminates from consideration the nodes of A_v not in A_w . We associate this set

with v and call it $\text{elim}(v)$. The size of this set, $|\text{elim}(v)|$, is $|A_v| - |A_w|$.

We use the eliminated sets to find the sizes of L , R , and U . The eliminated sets associated with the strict ancestors of v in A , along with A_v , partition the nodes of T into disjoint sets of contiguous nodes. For each node u that is an ancestor of v in A , $\text{elim}(u)$ lies in exactly one of L , R , and U . Let x be a child of v in A , and suppose that x lies in L . Then the size of L is just $|A_x|$ plus $|\text{elim}(u)|$ for each u in L that is a strict ancestor of v in A . (Using preorder and postorder numbers, we can determine in constant time whether a node lies in L .) The search time at v is proportional to the number of ancestors of v in A ; the total time to find a centroid is proportional to the square of the height of A . Once we have found a centroid c , the desired splitting edge e_c is the one that connects c to the largest subtree that would remain if c were deleted.

Deleting e_c from T results in two subtrees R and B of roughly equal size. For ease of exposition, let us assume that the nodes of these two subtrees are painted red and black, respectively. To allow recursive splitting, R and B must have auxiliary trees A^R and A^B built for them. The indices of nodes in R and B satisfy the separation property, and so we can define the auxiliary trees A^R and A^B just as A was defined on page 156. That is, the root r of A^R is the node in R with maximum index, and its subtrees are the auxiliary trees for the fragments of R created by deleting r ; A^B is defined similarly. These auxiliary trees do not have the structural characteristics described in Lemmas A.2 and A.4, but they are quite adequate for searching in R and B . We create A^R and A^B by modifying A ; because the construction does not start from scratch, it takes time proportional to the height of A , rather than to $|A|$.

The construction of A^R and A^B is based on the following lemma:

Lemma A.5. *The parent in A^R or A^B of a node v is its nearest strict ancestor in A with the same color.*

Proof: Suppose without loss of generality that v is red. Consider deleting the nodes of R in decreasing index order. Let u be the node whose deletion leaves v the node of maximum index in its component, and let C^R be the component in which v has maximum index. Then u is the

parent of v in A^R . Define C to be the component of T that contains v after all nodes with indices at least i_u are deleted. The nodes of C belong to A_u , so u is an ancestor of v in A . The interior nodes of the path in A from u to v lie in C but not in C^R , so they must all be black. ■

Splitting A to form the two new auxiliary trees is relatively straightforward. By Lemma A.1, each of the subtrees R and B has a unique node with maximum index. These nodes, one of which is a , are the roots of A^R and A^B . Let v be the endpoint of e_c with smaller index. The path π in A from a to v includes the other endpoint of e_c . Because the nodes are augmented with their preorder and postorder numbers in T , a constant-time test can determine the color of a node. (If l is the endpoint of e_c that is lower in T , then T_l contains all the nodes of one color and none of the other color. Note that the definitions of l and v are unrelated.) The root of A^R is the highest red node on π ; the root of A^B is the highest black node. The new auxiliary trees are constructed by dividing π into two monochromatic paths. In A^R and A^B , every node w on π has as its successor the next node on π with the same color as w . The subtree sizes $|A_x|$ are still valid in A^R and A^B except at nodes x on π , where they must be recomputed. These changes take time proportional to the length of the path π . To see that these modifications of A are sufficient, observe that at most one tree A_v is dichromatic for nodes v of a given depth in A (because a dichromatic tree includes both ends of e_c), and that after π is modified, each node has auxiliary tree links only to nodes of its own color.

To analyze the cost of this construction, we note that auxiliary tree nodes cannot increase in height as the decomposition proceeds, and that the total number of nodes in A of height k is at most $n/2^k$. The cost of splitting an auxiliary tree whose root has height k is $O(k^2)$. Furthermore, a node v with height k in A appears at the root of an auxiliary tree at most $O(k)$ times total during the decomposition, since it takes only $O(k)$ splittings to reduce the size of the auxiliary tree rooted at v from $O(3^k)$ to 1. Therefore the whole decomposition takes time

$$O\left(\sum_{k=0}^{\lfloor \log n \rfloor} k^3 \frac{n}{2^k}\right) = O(n).$$

The preceding discussion constitutes a proof of the following theorem:

Theorem A.1. *It is possible to find a balanced decomposition of an arbitrary binary tree in linear time.*

As noted at the beginning of the appendix, this linear algorithm for decomposing an arbitrary binary tree also provides a linear-time method for finding a balanced decomposition of a triangulated simple polygon.

Remark: This algorithm can easily be extended to decompose trees in which each node has bounded degree.

Remark: An alternative $O(n)$ -time technique for finding a balanced tree decomposition is obtained by using a simplified version of the dynamic tree data structure (as described in [Tar83, Ch. 5]), which supports logarithmic-cost tree-splitting operations.

*You have endured worse things,
God will grant an end even to these.*

— Virgil, *Aeneid*, I. 199

Bibliography

- [AAG*85] Takao Asano, Tetsuo Asano, Leo Guibas, John Hershberger, and Hiroshi Imai. Visibility-polygon search and Euclidean shortest paths. In *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, pages 155–164, IEEE, 1985. Also appeared in *Algorithmica* [AAG*86].
- [AAG*86] Takao Asano, Tetsuo Asano, Leo Guibas, John Hershberger, and Hiroshi Imai. Visibility of disjoint polygons. *Algorithmica*, 1(1):49–63, 1986. Revised version of a conference paper [AAG*85].
- [AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [AKM*86] A. Aggarwal, M. Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix searching algorithm. In *Proceedings of the 2nd ACM Computational Geometry Conference*, pages 285–292, ACM, June 1986.
- [Asa84] T. Asano. Efficient algorithms for finding the visibility polygons for a polygonal region with holes. 1984. Manuscript, Department of Electrical Engineering and Computer Science, University of California at Berkeley.
- [AT81] D. Avis and G. T. Toussaint. An optimal algorithm for determining the visibility of a polygon from an edge. *IEEE Trans. on Computers*, C-30:910–914, 1981.

- [Bak85] B. Baker. Shortest paths with unit clearance among polygonal obstacles. In *SIAM Conference on Geometric Modeling and Robotics*, SIAM, 1985. No proceedings; paper presented at conference.
- [Bau75] B. Baumgart. A polyhedral representation for computer vision. In *Proceedings of the AFIPS National Computer Conference*, pages 589–596, 1975.
- [Bro80] K. Q. Brown. *Geometric Transforms for Fast Geometric Algorithms*. PhD thesis, Carnegie-Mellon University, 1980.
- [BT86] B. Bhattacharya and G. Toussaint. *A Linear Algorithm for Determining Translation Separability of Two Simple Polygons*. Technical Report SOCS-86.1, School of Computer Science, McGill University, Montreal, 1986.
- [CG85] B. Chazelle and L. Guibas. Visibility and intersection problems in plane geometry. In *Proceedings of the ACM Symposium on Computational Geometry*, pages 135–146, ACM, 1985. Submitted to *Discrete and Computational Geometry*.
- [CGL85] B. Chazelle, L. J. Guibas, and D. T. Lee. The power of geometric duality. *BIT*, 25:76–90, 1985.
- [Cha82] B. Chazelle. A theorem on polygon cutting with applications. In *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science*, pages 339–349, IEEE, 1982.
- [Cha83] B. Chazelle. Filtering search: a new approach to query-answering. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 122–132, IEEE, 1983.
- [Che85] L. Paul Chew. Planning the shortest path for a disc in $O(n^2 \log n)$ time. In *Proceedings of the ACM Symposium on Computational Geometry*, pages 214–220, ACM, 1985.

- [CI84] B. Chazelle and J. Incerpi. Triangulation and shape complexity. *ACM Transactions on Graphics*, 135–152, 1984.
- [EA81] H. El Gindy and D. Avis. A linear algorithm for computing the visibility polygon from a point. *Journal of Algorithms*, 2:186–197, 1981.
- [EGS86] H. Edelsbrunner, L. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15:317–340, 1986. Published earlier as DEC/SRC report number 2.
- [El 84] H. A. El Gindy. An efficient algorithm for computing the weak visibility polygon from an edge in simple polygons. 1984. Manuscript, McGill University.
- [El 85] H. A. El Gindy. *Hierarchical Decomposition of Polygons with Applications*. PhD thesis, School of Computer Science, McGill University, 1985.
- [EOS83] H. Edelsbrunner, J. O'Rourke, and R. Seidel. Constructing arrangements of lines and hyperplanes with applications. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 83–91, IEEE, 1983.
- [EOW83] H. Edelsbrunner, M. H. Overmars, and D. Wood. Graphics in flatland: a case study. In F. P. Preparata, editor, *Advances in Computing Research*, pages 35–59, JAI Press Inc., 1983.
- [ET84] H. A. El Gindy and G. T. Toussaint. Efficient algorithms for inserting and deleting edges from triangulations. 1984. Manuscript, School of Computer Science, McGill University.
- [FM84] A. Fournier and D. Y. Montuno. Triangulating simple polygons and equivalent problems. *ACM Transactions on Graphics*, 3(2):153–174, 1984.

- [FT84] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science*, pages 338–346, IEEE, 1984.
- [GH85] Leo Guibas and John Hershberger. Computing the visibility graph of n line segments in $O(n^2)$ time. *Bulletin of the European Association for Theoretical Computer Science*, July 1985.
- [GH87] Leo Guibas and John Hershberger. Optimal shortest path queries in a simple polygon. In *Proceedings of the 3rd ACM Symposium on Computational Geometry*, ACM, June 1987.
- [GHL*86] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan. Linear time algorithms for visibility and shortest path problems inside simple polygons. In *Proceedings of the Second ACM Symposium on Computational Geometry*, pages 1–13, ACM, June 1986. To be published in *Algorithmica* under the title “Linear Time Algorithms for Visibility and Shortest Path Problems inside Triangulated Simple Polygons.”
- [GJPT78] M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan. Triangulating a simple polygon. *Information Processing Letters*, 7(4):175–179, 1978.
- [GK82] D. H. Greene and D. E. Knuth. *Mathematics for the Analysis of Algorithms*. Birkhauser, Boston, 1982.
- [GMPR77] L. Guibas, E. McCreight, M. Plass, and J. Roberts. A new representation for linear lists. In *Proceedings of the 9th ACM Symposium on Theory of Computing*, pages 49–60, ACM, 1977.
- [Gra72] R. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–133, 1972.

- [GRS83] L. Guibas, L. Ramshaw, and J. Stolfi. A kinetic framework for computational geometry. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 100–111, IEEE, 1983.
- [Gru72] Branko Grünbaum. *Arrangements and Spreads. Regional Conference Series in Mathematics*, American Mathematical Society, Providence, 1972.
- [GS85] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, 1985.
- [GT83] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pages 246–251, ACM, 1983.
- [GY83] R. L. Graham and F. F. Yao. Finding the convex hull of a simple polygon. *Journal of Algorithms*, 4:324–331, 1983.
- [Har80] D. Harel. A linear time algorithm for the lowest common ancestors problem. In *Proceedings of the 21st Annual IEEE Symposium on Foundations of Computer Science*, pages 308–319, IEEE, 1980.
- [Her87] John Hershberger. Finding the visibility graph of a simple polygon in time proportional to its size. In *Proceedings of the 3rd ACM Symposium on Computational Geometry*, ACM, June 1987.
- [HG86] John Hershberger and Leo Guibas. An $O(n^2)$ Shortest Path Algorithm for a Non-Rotating Convex Body. Technical Report 14, DEC/SRC, November 1986. To appear in *Journal of Algorithms*.
- [HM82] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

- [HM83] S. Hertel and K. Mehlhorn. Fast triangulation of a simple polygon. In *Proceedings of the Conference on Foundations of Computing Theory*, pages 207–218, Springer-Verlag, Berlin, 1983.
- [Kir83] David Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12:28–35, 1983.
- [KLPS86] K. Kedem, R. Livne, J. Pach, and M. Sharir. On the union of Jordan regions and collision-free translational motion amidst polygonal obstacles. *Discrete and Computational Geometry*, 1(1):59–71, 1986.
- [Knu72] Donald E. Knuth. Mathematical analysis of algorithms. In C. V. Freiman, editor, *Information Processing 71*, pages 19–27, International Federation for Information Processing, North-Holland Publishing Company, 1972. Proceedings of the 1971 IFIP Congress.
- [Knu73] Donald E. Knuth. *Fundamental Algorithms*. Volume 1 of *The Art of Computer Programming*, Addison-Wesley, second edition, 1973.
- [Lee78] D. T. Lee. *Proximity and Reachability in the Plane*. PhD thesis, University of Illinois at Urbana-Champaign, 1978.
- [Lee83] D. T. Lee. Visibility of a simple polygon. *Computer Vision, Graphics, and Image Processing*, 22:207–221, 1983.
- [LL84] D. T. Lee and A. Lin. Computing the visibility polygon from an edge. 1984. Manuscript, Northwestern University.
- [LP84] D. T. Lee and F. P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *Networks*, 14(3):393–410, 1984.
- [LW79] T. Lozano-Perez and M. A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*, 22:560–570, 1979.

- [MA79] D. McCallum and D. Avis. A linear algorithm for finding the convex hull of a simple polygon. *Information Processing Letters*, 9:201–206, 1979.
- [Meh84] K. Mehlhorn. *Data Structures and Efficient Algorithms 1: Sorting and Searching*. Springer Verlag, Berlin, 1984.
- [OvL81] M. Overmars and H. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.
- [Pap85] C. H. Papadimitriou. An algorithm for shortest-path motion in three dimensions. *Information Processing Letters*, 20:259–263, 1985.
- [PS85a] M. A. Peshkin and A. C. Sanderson. *Reachable Grasps on a Polygon: The Convex Rope Algorithm*. Technical Report CMU-RI-TR-85-6, Carnegie-Mellon University, 1985. To appear in *IEEE Transactions on Robotics and Automation*.
- [PS85b] F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer Verlag, New York, 1985.
- [Roh85] H. Rohnert. *Shortest Paths in the Plane with Convex Polygonal Obstacles*. Technical Report A 85/06, University of Saarbrücken, 1985.
- [RS85] J. Reif and J. Storer. *Shortest Paths in Euclidean Space with Polyhedral Obstacles*. Computer Science Department Report CS-85-121, Brandeis University, 1985. Submitted to *Journal of the ACM*.
- [SS84] M. Sharir and A. Schorr. On shortest paths in polyhedral spaces. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 144–153, ACM, 1984.
- [ST85] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.

- [Sto87] Jorge Stolfi. Oriented projective geometry. In *Proceedings of the 3rd ACM Symposium on Computational Geometry*, ACM, June 1987.
- [Sur86a] Subhash Suri. Finding minimum link paths inside a simple polygon. 1986. To appear in *Computer Vision, Graphics and Image Processing*.
- [Sur86b] Subhash Suri. Private communication. 1986. An output-size sensitive algorithm to find the visibility graph of a simple polygon, not published.
- [Sur87] S. Suri. The all-geodesic-furthest neighbor problem for simple polygons. In *Proceedings of the 3rd ACM Symposium on Computational Geometry*, ACM, June 1987.
- [Tar83] R.E. Tarjan. *Data Structures and Network Algorithms*. CBMS-NSF Regional Conference Series in Applied Mathematics, Society for Industrial and Applied Mathematics, Philadelphia, 1983.
- [Tou85] G. Toussaint. *Shortest Path Solves Edge-To-Edge Visibility in a Polygon*. Technical Report SOCS-85.19, McGill University, 1985.
- [TV86] R. E. Tarjan and C. Van Wyk. An $O(n \log \log n)$ -time algorithm for triangulating simple polygons. 1986. Manuscript, submitted to *SIAM Journal on Computing*.
- [Wel85] E. Welzl. Constructing the visibility graph for n line segments in $O(n^2)$ time. *Information Processing Letters*, 20:167–171, 1985.