Dynamic and I/O-Efficient Algorithms for Computational Geometry and Graph Problems: Theoretical and Experimental Results

Yi-Jen Chiang

Ph.D. Dissertation

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-95-27
August 1995

Dynamic and I/O-Efficient Algorithms for Computational Geometry and Graph Problems: Theoretical and Experimental Results

Yi-Jen Chiang
Ph.D. Dissertation

Department of Computer Science
Brown University
Providence, Rhode Island 02912

August 1995

Dynamic and I/O-Efficient Algorithms for
Computational Geometry and Graph Problems:
Theoretical and Experimental Results

by
Yi-Jen Chiang
B. Sc., National Taiwan University, Taipei, Taiwan, June 1986
Sc. M., Brown University, May 1991

Thesis
Submitted in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy in the Department of Computer Science
at Brown University.

May 1996

Vita

Yi-Jen Chiang was born in Taipei, Taiwan on October 28, 1962. He received his B. Sc. degree in Computer Science and Information Engineering (CSIE) from National Taiwan University, Taipei, Taiwan in June 1986. After two years of compulsory military service, he became a research assistance in the CSIE Department at National Taiwan University. He entered the master's program in the Computer Science Department at Brown University in 1989, and received his Sc. M. degree in May 1991. Later that year, Yi-Jen entered the Ph.D. program in the same department. He completed his Ph.D. in August 1995.

Dedication

To my wife Wen-Pei Sophie Hsu
and my parents:
Wen-Jien Arthur Chiang
and
Ray-Wha Chiang

Abstract

As most important applications today are large-scale in nature, high-performance methods are becoming indispensable. Two promising computational paradigms for large-scale applications are dynamic and I/O-efficient computations. We give efficient dynamic data structures for several fundamental problems in computational geometry, including point location, ray shooting, shortest path, and minimum-link path. We also develop a collection of new techniques for designing and analyzing I/O-efficient algorithms for graph problems, and illustrate how these techniques can be applied to a wide variety of specific problems, including list ranking, Euler tour, expression-tree evaluation, least-common ancestors, connected and biconnected components, minimum spanning forest, ear decomposition, topological sorting, reachability, graph drawing, and visibility representation. Finally, we present an extensive experimental study comparing the practical I/O efficiency of four algorithms for the orthogonal segment intersection problem with large-scale test data. The experiments provide detailed quantitative evaluation of the performance of the four algorithms, and the observed behavior of the algorithms is consistent with their theoretical properties.

Acknowledgements

I am deeply grateful to my friend and advisor Roberto Tamassia, who taught me how to do research and also constantly gave me guidance and assistance. I am very lucky to have worked with him. All research in this thesis was performed jointly with Roberto (except Chapter 5, for which I also had a lot of useful discussions with him), and I really enjoyed working with him very much.

I would like to thank Franco Preparata for his kind assistance and encouragement. His insight and guidance on research directions also influenced my research a lot. The research in Chapter 2 was done jointly with him. Interestingly, I attended Franco's invited talk on point location in 1989 in Taiwan before I came to Brown, and that was the initiation of my interests in point location. (At that time Franco was with the University of Illinois and I did not expect to see him at Brown; he came to Brown in 1991—what a coincidence!)

Additionally, I appreciate the involvement of the following colleagues of mine: Michael Goodrich, Edward Grove, Darren Vengroff, and Jeff Vitter. The research in Chapter 4 was done jointly with them.

I thank my thesis committee, Franco Preparata and Michael Goodrich, for their helpful suggestions and comments. Many thanks to Jeff Vitter and Paris Kanellakis for their encouragement and kind help during my stay at Brown. I also thank Tom Doeppner and Peter Galvin for useful information about the computing systems of the Brown CS department while doing the experiments of Chapter 5.

I would like to thank the administrative staff, in particular Mary Andrade, Katrina Avery, Jennet Kirschenbaum, Dawn Nicholaus, and Susan Platt, for helping me in many day-to-day matters.

Many friends at Brown provided assistance and moral support: Jyh-Han Lin, Wen-Chun Ni, Hsueh-I Lu, Shieu-Hong Lin, Yi-Jing Lin, Tu-Hsin Tsai, Ashim Garg, Ramamurthy Ravi, Sairam Subramanian, P. Krishnan, Sridhar Ramaswamy, Manojit Sarkar, and Paul Howard.

I am deeply grateful to my wife Wen-Pei Sophie Hsu, for her constant support and comfort. Every day was sweet and wonderful since our marriage in 1991, and her company made the stay at Brown even more enjoyable.

Finally, I thank my father Wen-Jien Arthur Chiang and my mother Ray-Wha Chiang, both of whom passed away right before I entered college. They taught me everything, in particular how to think and how to live. They helped me in establishing the goal of my life, and made me love knowledge. They showed me by themselves that studying was a part of everyday life and was very enjoyable. In addition, my mother taught me how to play the piano, and my father taught me literature, English, and mathematics. I am who I am now because of them. I wish they could read this dissertation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As most important applications today are large-scale in nature, high-performance methods are becoming indispensable. Two promising computational paradigms for large-scale applications are dynamic and I/O-efficient computations. We give efficient dynamic data structures for several fundamental problems in computational geometry, and also develop a collection of new techniques for designing and analyzing I/O-efficient algorithms for graph problems. Finally, we present an extensive experimental study comparing the practical I/O efficiency of some geometric algorithms.

1.1 Dynamic Computational Geometry

The development of dynamic algorithms for geometric problems has acquired increasing interest, motivated by many practical applications in computer graphics, robotics, VLSI layout, and geographic information systems. Dynamic (or incremental) computation considers updating the solution of a problem when the problem instance is modified. Many applications are incremental (or operation-by-operation) in nature and the typical run involves *on-line* processing of a mixed sequence of *queries* and *updates* on some structure that evolves over time. Considerable savings can be achieved if the new solution need not be generated "from scratch," especially when the problem is large-scale.

An important task in spatial databases, computer-aided geometric design, and vehicle vulnerability assessment is geometric searching. Searches typically are for locating a specific point in a geometric environment (*point location*) and then possibly traversing a ray "shot" from that location until it strikes some object in the environment (*ray shooting*). Also, many motion planning applications require the computation of an obstacle-avoiding path between a source and a destination, and there are various criteria used to optimize the path, for example, *shortest* (a path minimizing the total length in an underlying metric), and *minimum-link* (a path minimizing the number of turns).

In my master's work [28], a fully dynamic data structure for point location in a monotone map is presented, based on the trapezoid method. The update operations supported are insertion and deletion of vertices and edges, and horizontal translation of vertices. This is the first fully dynamic point location data structure for monotone maps that achieves optimal query time.

Continuing the work along this direction, in this dissertation we describe in Chapter 2 a new technique for dynamically maintaining the trapezoidal decomposition of a connected planar map $\mathcal{M}$ with n vertices, and apply it to the development of a unified dynamic data structure that supports point-location, ray-shooting, and shortest-path queries in $\mathcal{M}$. The space requirement is $O(n \log n)$. Point-location queries take time $O(\log n)$. Ray-shooting and shortest-path queries take time $O(\log^3 n)$ (plus $O(k)$ time if the k edges of the shortest path are reported in addition to its length). Updates consist of insertions and deletions of vertices and edges, and take $O(\log^3 n)$ time (amortized for vertex updates). This is the first polylog-time dynamic data structure for shortest-path and ray-shooting queries. It is also the first dynamic point-location data structure for connected planar maps that achieves optimal query time.

Exploring further on the path optimization problem, we present in Chapter 3 efficient algorithms for shortest-path and minimum-link-path queries between two convex polygons inside a simple polygon P, which acts as an obstacle to be avoided. Let n be the number of vertices of P, and h the total number of vertices of the query polygons. We show that shortest-path queries can be performed optimally in time $O(\log h + \log n)$ (plus $O(k)$ time for reporting the k edges of the path) using a data structure with $O(n)$ space and preprocessing time, and that minimum-link-path queries can be performed in optimal time $O(\log h + \log n)$ (plus $O(k)$ to report the k links), with $O(n^3)$ space and preprocessing time.

We also extend our results to the dynamic case, and give a unified data structure that supports both queries for convex polygons in the same region of a connected planar map $\mathcal{M}$. The update operations consist of insertions and deletions of edges and vertices. Let n be the current number of vertices in $\mathcal{M}$. The data structure uses $O(n)$ space, supports updates in $O(\log^2 n)$ time, and performs shortest-path and minimum-link-path queries in times $O(\log h + \log^2 n)$ (plus $O(k)$ to report the k edges of the path) and $O(\log h + k \log^2 n)$, respectively. Performing shortest-path queries is a variation of the well-studied *separation* problem, which has not been efficiently solved before in the presence of obstacles, even in a static environment. Also, it was not previously known how to perform minimum-link-path queries in a dynamic environment, even for two-point queries.

1.2 I/O Efficient Computations

Input/Output (I/O) communication between fast internal memory and slower external memory is the major bottleneck in many large-scale applications. The significance of this bottleneck is increasing as internal computation gets faster, and especially as parallel computing gains popularity. Due to this important fact, more and more attention has been given to the development of I/O-efficient algorithms in recent years. Algorithms designed specifically to make efficient use of levels of memory are often called *external-memory algorithms* to emphasize the explicit use of memory beyond random access main memory. The major problem is how to design external-memory algorithms that are efficient in terms of I/O for significant applications.

In Chapter 4 we derive a collection of new techniques for designing and analyzing efficient external-memory algorithms for graph problems, which arise in many large-scale computations including those common in object-oriented and deductive databases, VLSI design and simulation programs, and geographic information systems. Our techniques apply to a number of specific problems, including list ranking, Euler tour, expression-tree evaluation, least-common ancestors, connected and biconnected components, minimum spanning forest, ear decomposition, topological sorting, reachability, graph drawing, and visibility representation. For all these problems considered, we give the first I/O-efficient algorithms, most of them being I/O-optimal.

Although there has been an increasing interest in the development of I/O-efficient techniques, most of the developed algorithms, however, are shown to be efficient only *in theory*, and their performance *in practice* is yet to be evaluated. In particular, all such algorithms assume that the internal computation is free compared to the I/O cost, which also has to be justified in practice. An important task in the research of I/O-efficient computation, therefore, is to evaluate the practical efficiency of the algorithms by judicious experimentation.

In Chapter 5 we present an extensive experimental study comparing the performance of four algorithms for the following *orthogonal segment intersection problem*: given a set of horizontal and vertical line segments in the plane, report all intersecting horizontal-vertical pairs. The problem has important applications in VLSI layout and graphics, which are large-scale in nature. The algorithms under evaluation are distribution sweep and three variations of plane sweep. Distribution sweep is specifically designed for the situations in which the problem is too large to be solved in internal memory, and theoretically has optimal I/O cost. Plane sweep is a well-known and powerful technique in computational geometry, and is optimal for this particular problem in terms of internal computation. The three variations of plane sweep differ by the sorting methods (external vs. internal sorting) used in the preprocessing phase and the dynamic data structures (B tree vs. 2-3-4 tree) used in the sweeping phase. We generate the test data by three programs that use a random number generator while producing some interesting properties that are predicted by our theoretical analysis. The sizes of the test data range from 250 thousand segments to 2.5 million segments. The experiments provide detailed quantitative evaluation of the performance of the four algorithms, and the observed behavior of the algorithms is consistent with their theoretical properties. This is the first experimental work comparing the practical performance between external-memory algorithms and conventional algorithms with large-scale test data.

Remark. Most of the results presented in this dissertation have been published by the Author in journals and/or proceedings of conferences: Chapters 2 and 3 are based on [26] and [29], and summaries of the results in Chapters 4 and 5 have respectively appeared in [25] and [24].

Chapter 2

Dynamic Point Location, Ray Shooting and Shortest Paths

2.1 Introduction

A number of operations within the context of planar maps (or subdivisions, as determined by a planar graph embedded in the plane) have long been regarded as important primitives in computational geometry. First and foremost among these operations is planar point-location, i.e., the identification of the map region containing a given query point; but also shortest-path and ray-shooting queries have been considered very prominently.

Starting with the pioneering work in planar point-location of the seventies [46, 75], over the years several techniques have been developed, culminating in asymptotically time- and space-optimal methods [49, 72, 103] that are also of sufficiently practical flavor. Such methods, however, refer to the *static* case where no alteration of the map is allowed during its use. Due to the obvious importance of the *dynamic* setting, in recent years considerable attention has been devoted to the development of dynamic point-location algorithms [8, 23, 28, 54, 57, 89, 94, 95, 112].

All the known dynamic point location results are for connected maps, since maintaining region names in a disconnected map would require solving half-planar range searching in a dynamic environment, for which no polylog-time algorithm is known. The best results to-date for dynamic point-location in an n-vertex connected map are due to Cheng-Janardan [23] and Baumgarten-Jung-Mehlhorn [8]. The technique of [23] achieves $O(\log^2 n)$ query time, $O(\log n)$ update time, and $O(n)$ space. The data structure of [8] has query and insertion time $O(\log n \log\log n)$, deletion time $O(\log^2 n)$, using $O(n)$ space, where the time bounds are amortized for the updates. In many real-time applications, point-location queries are executed more frequently than updates, so that it is often desirable to achieve optimal $O(\log n)$ query time in a dynamic setting. The only previous technique that supports $O(\log n)$-time queries in a dynamic environment is restricted to monotone maps [28]. For a survey of dynamic point-location techniques and other dynamic algorithms in computational geometry, see Chiang and Tamassia [27].

Algorithmic research on shortest-path and ray-shooting queries has also experienced steady progress, resulting in time-optimal techniques for the static setting [1, 21, 22, 63, 76]. In particular, the linear-space data structures of Chazelle-Guibas [21] and of Guibas-Hershberger [63] support in $O(\log n)$ time ray-shooting and shortest-path queries, respectively, in a simple polygon with n vertices. No polylog-time method was previously known in a dynamic setting, although a polylog-time ray-shooting technique by Reif and Sen [99], designed for monotone polygons, may be extensible to the general case. Sublinear-time techniques are known only for ray-shooting queries [1, 22], with

$O(\sqrt{n}\,\mathrm{polylog}(n))$ query/update time; they support ray-shooting in a set of possibly intersecting segments without taking advantage of the structure of planar maps.

A property that appears to greatly facilitate the development of dynamic point-location techniques is *monotonicity* ([28, 57, 94]). Whereas the restriction to monotone maps is quite adequate for many important applications, yet the exclusion of more general maps is a severe shortcoming. In the static case, a connected map can be reduced to monotone (or, as we say in this section, *normalized*) by the straightforward insertion of (auxiliary) diagonals. The same approach, when attempted for the dynamic setting, could lead to onerous updates, such as when the insertion of an edge causes the removal of a very large number of normalizing diagonals. A rather complicated and only partially documented technique due to Fries [53], is reported to assure that only a logarithmic number of normalizing diagonals be involved in any update.

In this chapter we combine the feature just stated with the underpinnings of the trapezoid method, whose search efficiency both in theory [16, 92] and practice [47] is well-established. This leads to the adoption of horizontal normalizing diagonals, called *lids*. The method rests on three major components:

1. A *normalization structure* that transforms a connected map into a monotone one by the addition of horizontal diagonals, while guaranteeing that no more than a logarithmic number of such diagonals are affected by insertions/deletions of edges/vertices.
2. A *hull structure* that stores the convex hulls of the chains and subchains of the monotone subregions, so that ray-shooting and shortest-path queries can be efficiently performed.
3. A *location structure* that represents a recursive decomposition of the normalized map into trapezoidal regions, and supports point-location queries in optimal time.

It is important to underscore that a single tree structure—the normalization structure—provides the unifying framework for the three applications considered. In fact, this structure, while ensuring efficient updates by controlling the size of the modifications, can be naturally augmented with node-appended secondary structures to support shortest-path and ray-shooting queries. It can also be supplemented with a distinct, but tightly coupled, location structure designed for efficient point-location. The main normalization structure and its two auxiliary components act in a tightly integrated fashion: point-location is crucially used in shortest-path and ray-shooting queries and in the update of the normalization structure.

The fundamental constituents of our data structures are monotone chains and trapezoids determined by edges and horizontal lines through vertices. This provides the unifying framework for the three applications mentioned earlier. Indeed, a simple augmentation of the normalization structure provides the right environment for all three queries, as we shall illustrate. It should be underscored that, although their linkings are obviously elaborate, the elementary data structures employed are particularly simple, so that not only asymptotic efficiency is established, but also practical potential is apparent.

Our main results in this chapter are outlined in the following theorem:

Theorem 2.1 *There exists a fully dynamic data structure that supports point-location, ray-shooting, and shortest-path queries in a connected planar map $\mathcal{M}$ with n vertices. The space requirement is $O(n \log n)$. Point-location queries take time $O(\log n)$. Ray-shooting and shortest-path queries take time $O(\log^3 n)$ (plus $O(k)$ time if the k edges of a shortest path are reported in addition to its length). Updates take $O(\log^3 n)$ time (amortized for vertex updates).*

As a corollary, we can also perform stabbing queries, i.e., determine the k edges of map $\mathcal{M}$ intersected by a query segment, in $O((k+1)\log^3 n)$ time.

The contributions of this chapter can be summarized in the following points:

- We present the first polylog-time dynamic data structure for shortest-path queries in connected planar maps. All previous data structures for shortest paths are static and take linear time for either queries or updates when used in a dynamic environment.
- We provide the first polylog-time dynamic data structure for ray-shooting queries in connected planar maps. The previous best result is $O(\sqrt{n}\,\mathrm{polylog}(n))$ query time.
- We present the first dynamic data structure for point location queries in connected planar maps with optimal $O(\log n)$ query time and polylog update time. The previous best result is $O(\log n \log\log n)$ query time.
- We provide the first dynamic point-location data structure that checks the validity of an edge insertion, i.e., whether the new edge does not intersect the current edges of the map. Previous dynamic point location data structures did not have such a capability due to the lack of an efficient dynamic ray-shooting technique.

The rest of this chapter is organized as follows. In Section 2.2 we briefly review the terminology of planar maps and the basic data structures used by our method. The mechanics of the dynamic maintenance of a normalized map are described in Section 2.3, while Sections 2.4, 2.5, and 2.6 are respectively devoted to shortest-path, point-location, and ray-shooting queries.

2.2 Review of Background

For the geometric terminology used in this chapter, see [93]. A *connected planar map* $\mathcal{M}$ is a subdivision of the plane into polygonal regions whose underlying planar graph is connected. The map is augmented with two vertical rays, one directed toward $y = +\infty$, the other toward $y = -\infty$, respectively issuing from the vertices of $\mathcal{M}$ with maximum and minimum y-coordinates. Thus, all but two regions of $\mathcal{M}$ are bounded simple polygons. In the following, n denotes the number of vertices of the planar map $\mathcal{M}$ currently being considered. Also, we assume that no two vertices of $\mathcal{M}$ have the same y-coordinate; the degenerate cases can be handled by standard techniques and will not be discussed in this chapter.

In the plane we have an orthogonal frame of reference (x, y). A polygonal chain γ is *monotone* if any horizontal line intersects it in a single point or in a single interval or not at all. A simple polygon r is *monotone* if its boundary consists of two monotone chains. A *cusp* of a polygon is a vertex v whose internal angle is greater than π and whose adjacent vertices are both strictly above (lower cusp) or strictly below (upper cusp) v. A polygon is monotone if and only if it has no cusps. A map is monotone if all its regions are monotone.

The *trapezoidal decomposition* of a connected map $\mathcal{M}$ is obtained by drawing from each vertex v of $\mathcal{M}$ two horizontal rays that either remain unbounded or terminate when they first meet edges of $\mathcal{M}$: the resulting segments are called *splitters*. It is easily verified that a region of $\mathcal{M}$ with s vertices is partitioned by the splitters into $s - 1$ trapezoids (see Fig. 2.1). The trapezoidal decomposition of $\mathcal{M}$ is geometrically dualized by mapping each of the obtained trapezoids τ to an arbitrary point $\delta(\tau)$ in the interior of τ: each of the splitters is mapped to an edge between images of trapezoids in the usual way. We let $\delta(\mathcal{M})$ denote the resulting dual graph, which is a forest of trees since the trapezoids of a single region $r \in \mathcal{M}$ dualize to a tree $\delta(r)$ (because r has no holes). Note that each node of $\delta(r)$ has degree at most four. Let s_i, $i = 1, 2$, denote either a splitter or an extreme vertex of region r. Then $\textsc{sleeve}(s_1, s_2)$ denotes the union of the trapezoids traversed by the shortest path within r between any point of s_1 and any point of s_2. (Note the duality between "sleeves" in region r and paths in tree $\delta(r)$.) In a notationally consistent manner, $\delta(s)$ denotes the edge of $\delta(r)$ that is the dual of splitter s.

Our data structures are based on a variety of balanced search trees. We observe that all the standard operations on balanced search trees (insertion, deletion, split, and join) can be performed by means of a logarithmic number of more basic primitives, which we call "elementary joins and splits", defined as follows:

- An *elementary join* of two binary trees T_1 and T_2 forms a new tree T by making T_1 and T_2 the left and right subtrees of a new root node.
- An *elementary split* yields the left and right subtrees T_1 and T_2 of T by removing its root.

In particular, a simple rotation can be viewed as a sequence of four elementary splits and joins.

Three special types of data structures will be used in this chapter: biased binary trees [10], $BB[\alpha]$-trees [81], and dynamic trees [104].

A *biased binary tree* [10] is a binary search tree whose leaves store weighted items. Let w be the sum of all weights. We have that the depth of a leaf with weight w_i is at most $\log(w/w_i) + 2$, and each of the following update operations can be done in $O(\log w)$ time: change of the weight of an item, insertion/deletion of an item, and split/splice of two biased trees [10].

A $BB[\alpha]$-tree [81] (where α is a fixed real, with $\frac{1}{4} < \alpha \leq 1 - \frac{\sqrt{2}}{2}$) is a binary search tree and has the following important properties (among others):

- A $BB[\alpha]$-tree with n nodes has height $O(\log n)$.
- Assume that we augment a $BB[\alpha]$-tree with secondary structures stored at its nodes. Let the subtree with root μ have ℓ leaves, and let the time for updating the secondary structures after a rotation at node μ be $O(\ell \log \ell)$. Then the amortized time of an update operation in a sequence of n insertions and deletions starting from an initially empty $BB[\alpha]$-tree is $O(\log^2 n)$.

Dynamic trees [104] are designed to represent a forest of rooted trees, with each edge directed toward the root of its tree (and called an *arc*). Some important operations (among others) supported by dynamic trees include:

link(μ, ν): Add an arc from μ to ν, thereby making μ a child of ν in the forest. This operation assumes that μ is the root of one tree and ν is a node of another tree.

cut(μ): Delete the arc from μ to its parent, thereby dividing the tree containing μ into two trees.

evert(μ): Make μ the root of its tree by reversing the path from μ to the original root.

Each arc of the trees is classified as *solid* or *dashed*, so that each tree is partitioned into a collection of *solid paths*, connected by dashed arcs. A solid path is maintained by a data structure called a *path tree*. Using biased binary trees [10] as the standard implementation of path trees, each of the above operations takes $O(\log n)$ time, where n is the size of the tree(s) in the forest involved.

2.3 The Dynamics of Trapezoidal Decompositions

Given a connected map $\mathcal{M}$, our objective is first to systematically transform *(normalize)* it into a monotone map, and then to illustrate how to efficiently maintain it under a dynamic regimen of edge and vertex insertions/deletions.

2.3.1 Normalization

We first address the problem of normalization. Each region r of $\mathcal{M}$ is handled individually. We refer to a region r, bounded or unbounded. In the following, we denote by m the current number of vertices in r.

We imagine to represent $\delta(r)$ as a dynamic tree $\Delta(r)$ [104] (see Fig. 2.1). We choose an arbitrary node of $\delta(r)$ as the *root*, which immediately forces a direction on each edge, referred to hereafter as an *arc* and directed toward the root. Since we have chosen to dualize each trapezoid to a point in its interior, the y-component of each arc has a well-defined sign. An arc is usually denoted either by a single letter or by an ordered pair (origin, destination). The arcs are classified as follows: letting $w(\mu)$, *weight* of μ, denote the number of nodes in the subtree rooted at node μ, an arc (μ, ν) is classified *heavy* if $w(\mu) \geq \frac{1}{2}w(\nu)$, and *light* otherwise. Consequently, at most one heavy arc enters a node of $\Delta(r)$. Note that the attributes {light, heavy} pertain uniquely to the weight structure of the dynamic tree $\Delta(r)$.

Arcs are also classified as *solid* or *dashed* to enforce the property that *at most one solid arc enters a node* of $\Delta(r)$. The maximal paths of consecutive solid arcs (possibly consisting of a single node) are called *solid paths*, and each corresponds to a sleeve of r. Note that the attributes {dashed, solid} pertain to a given, but otherwise arbitrary, decomposition of r into sleeves.

The weight structure and the sleeve decomposition are tied by the following *weight invariant*, which holds before and after the execution of data structure operations (queries or updates):

"heavy arcs are solid and light arcs are dashed."

However, during the execution of operations, we may change heavy arcs to dashed and light arcs to solid, and thus loose the original correspondence. The weight invariant is restored at the completion of each operation.

Region r contains a set of splitters, called *lids*, which are the duals of the following arcs:

Rule 1. All dashed arcs.

Rule 2. Any two consecutive solid arcs whose y-components have opposite signs.

Note that each lid is generated by a vertex of r. The set of lids *normalizes* r. Namely, we have

Lemma 2.1 *The set of lids partitions r into a collection of monotone polygons.*

Proof: Let c be a cusp of polygon r. We consider the two arcs of $\Delta(r)$ which are the duals of the two splitters issuing from c. If at least one of them is dashed (see Fig. 2.2(a)), then there is at least one lid issuing from cusp c corresponding to the dashed arc (Rule 1). If on the other hand both arcs are solid, then one must have a positive y-component and the other a negative one, or otherwise they would enter the same node of $\Delta(r)$ and thus would violate the property that at most one solid arc enters a node (see Fig. 2.2(b)). Then these two arcs are consecutive solid arcs with y-components of opposite signs, and there are two lids from c corresponding to these arcs (Rule 2). Hence there is always at least one horizontal lid issuing from each cusp c of r, thereby achieving a decomposition of r into monotone polygons. □

Lemma 2.2 *Each directed path of the dynamic tree $\Delta(r)$ contains at most $\log_2 m$ light arcs.*

Proof: Moving away from the root, each light arc traversed reduces the size of the current subtree by at least one half, since $w(child) < \frac{1}{2}w(parent)$. □

Corollary 2.1 *Any straight line drawn in region r crosses $O(\log m)$ lids.*

Proof: The weight invariant is always preserved before and after the execution of data structure operations. Each lid then corresponds to either (i) a light arc (Rule 1, since dashed arcs are light), or (ii) a solid arc at which the solid path containing this arc changes monotonicity with respect to the y-axis (Rule 2). By Lemma 2.2, any straight line l drawn in r crosses $O(\log m)$ lids of type (i). Now consider the lids of type (ii). Lemma 2.2 also implies that l goes through $O(\log m)$ solid paths. Observe that each solid path P can be partitioned into maximal monotone subpaths, and l

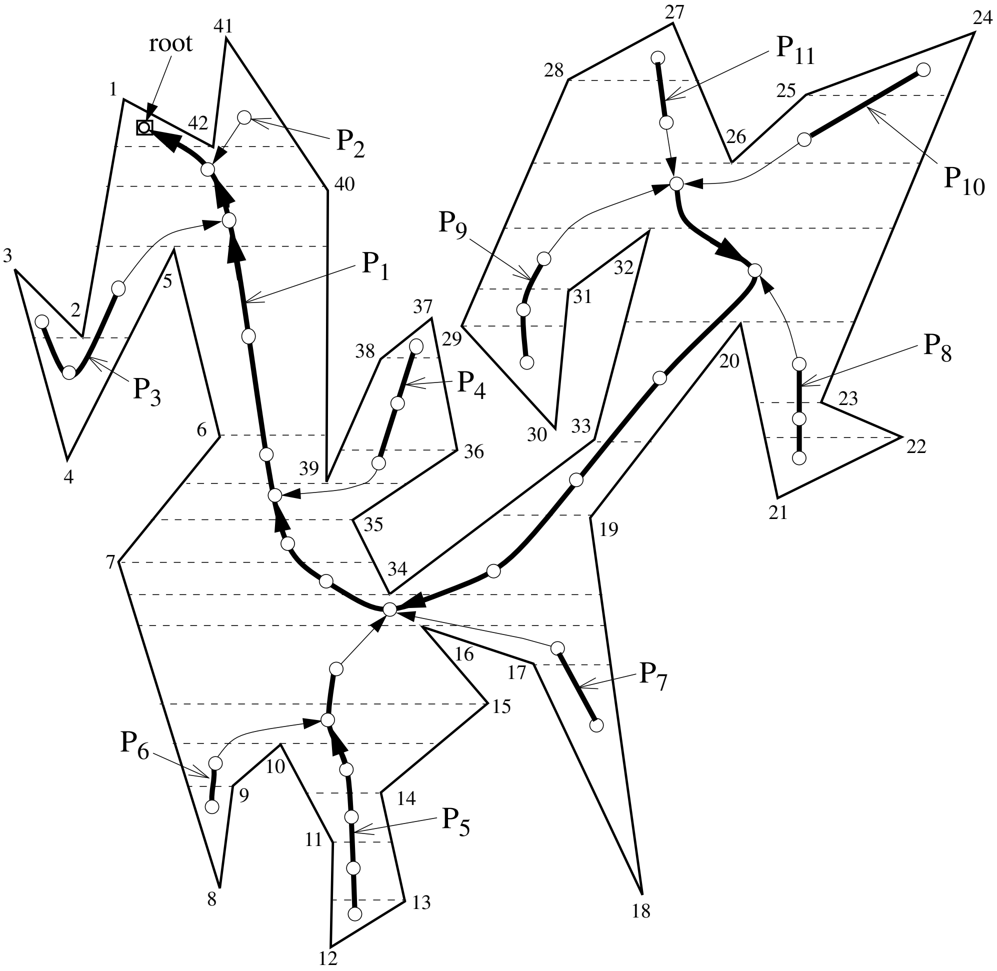


Figure 2.1: Example of a region r and its dynamic tree $\Delta(r)$ (P_1,...,P_{11} are solid paths).

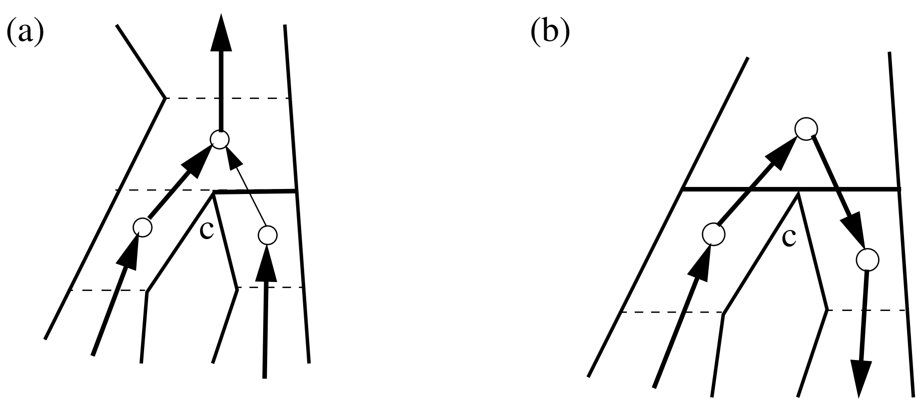


Figure 2.2: Proof of Lemma 2.1.

can go through at most one such monotone subpath, thus crossing at most two lids of solid arcs of P. It follows that the number of lids of type (ii) crossed by l is also $O(\log m)$. $\square$

2.3.2 The Double-Thread Data Structure

It is intuitively clear that insertion or deletion of an edge may substantially modify the set of trapezoids, whereas it alters only slightly the structure of region boundaries. For this reason we adopt a data structure that represents a solid path of $\Delta(r)$ by two "threads"; these two threads respectively correspond to two chains whose union is the boundary of the sleeve associated with the solid path. The proposed structure is referred to as *double-thread data structure* for region r, denoted by $DT(r)$.

Each arc α of $\Delta(r)$ can be drawn to intersect its dual splitter issuing from some vertex v of r. Therefore we associate α with v. Notice that each vertex v in $\mathcal{M}$ is associated with two arcs: if v is a cusp of some region r, then the two splitters issuing from v both lie in r and thus cross two arcs of $\Delta(r)$; otherwise, v belongs to two regions r_1 and r_2 and the two splitters issuing from v cross respectively an arc of $\Delta(r_1)$ and an arc of $\Delta(r_2)$. Instead of maintaining the nodes of $\Delta(r)$, we choose to maintain the arcs of $\Delta(r)$ using the vertices of r as their representatives, by associating each node of $\Delta(r)$ to the arc issuing from it. As a consequence, each solid path P is represented by two binary trees $lthread(P)$ and $rthread(P)$, referred to as *thread trees*, whose implementation is described below. Recall that each solid path is directed toward the root. Each vertex v associated with an arc on solid path P is classified as follows: walking along P toward the root, vertex v is classified *left* if it lies to the left of P, and *right* otherwise. Notice that if P is followed by a dashed arc α (every solid path except the one terminating at the root of $\Delta(r)$ has this property), then we also include α as an arc on solid path P in our representation.

The arcs of a solid path P can be partitioned into maximal monotone (on the basis of the signs of their y-components) subpaths $Q_1, Q_2, \cdots, Q_k$. Our thread trees $lthread(P)$ and $rthread(P)$ are each implemented as a two-level (called lower and upper) balanced binary tree (i.e., the roots of lower-level trees are leaves of the upper-level tree). Referring to $lthread(P)$, in the lower level, we have a balanced binary tree $ltree(Q_i)$ for each Q_i, where the leaves of $ltree(Q_i)$ store the left vertices of Q_i in their path order. Thread tree $rthread(P)$ is analogously organized, with $rtree(Q_i)$ storing the right vertices. The roots of $ltree(Q_i)$ and $rtree(Q_i)$ are bidirectionally linked. In the upper level, $lthread(P)$ (and analogously $rthread(P)$) has the roots of $ltree(Q_1)$, $ltree(Q_2)$, $\cdots$, $ltree(Q_k)$ as leaves in their path order. A bidirectional link also exists between the roots of $lthread(P)$ and $rthread(P)$. An example is shown in Fig. 2.5(a).

Any node on P might be pointed to (via dashed arcs) by some other solid paths in the dynamic tree $\Delta(r)$. Suppose that P' points to P via an arc α' associated with vertex v'. Two situations may now occur: (i) vertex v' is also associated with an arc of P (e.g., see paths P_2, P_3 and P_4 in Fig. 2.1 with $P = P_1$). Then v' is a left or right vertex of P (thus stored as a lower-level leaf of $lthread(P)$ or of $rthread(P)$). We establish a pointer from each root of $lthread(P')$ and $rthread(P')$ to that lower-level leaf v' (see Fig. 2.5(b)). The possible instances of this situation are illustrated in Figure 2.3(b, d). (ii) vertex v' is not associated with an arc of P (e.g., see paths P_5 and P_7 in Fig. 2.1 with $P = P_1$). This occurs if P changes monotonicity (by crossing both splitters of a cusp c) at the node reached by arc α'. In this case, in order to provide a destination for the pointers from the roots of $lthread(P')$ and $rthread(P')$, we introduce an auxiliary leaf, called a *coupler* (usually denoted by letter H), inserted between the two consecutive subtrees (both either $ltree$'s or $rtree$'s) of the thread tree not containing cusp c (see Fig. 2.5(b)). The possible instances of this situation are illustrated in Figure 2.3(c, e).

Note that a pointer destination may be needed when a solid path P begins (Fig. 2.4(b, c) and

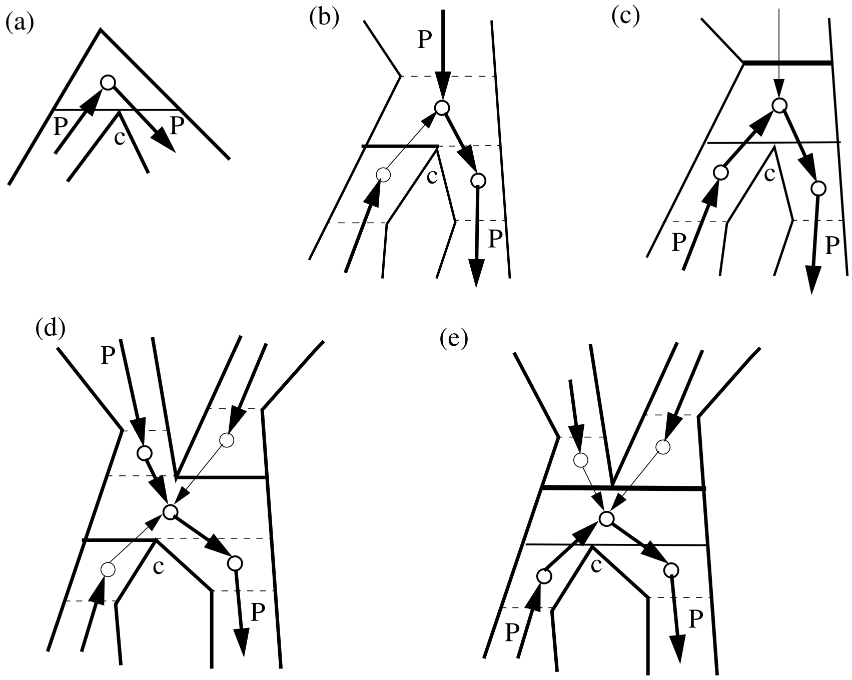


Figure 2.3: All possible cases in which a solid path P crosses a splitter issuing from a cusp c. Note that P does not change monotonicity (i.e., crosses only one splitter issuing from c) in (b) and (d), and P changes monotonicity (i.e., crosses both splitters issuing from c) in (a), (c) and (e).

Fig. 2.1 for $P = P_1$). In this case we adopt the convention to insert a coupler preceding either $ltree(Q_1)$ or $rtree(Q_1)$, where Q_1 is the initial monotone subpath of P (see Fig. 2.5(b)). The overall data structure $DT(r)$ consists therefore of two rooted trees of indegree at most 4 (see Fig. 2.5(b)).

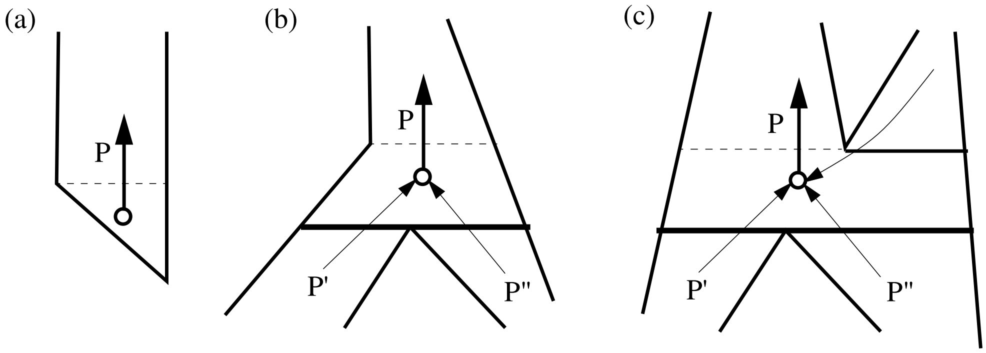


Figure 2.4: All possible cases in which a solid path P starts. Note that a coupler of P is needed to provide a destination of P' and P'' in (b) and (c).

We now define a new parameter of nodes of $DT(r)$ (*DT-nodes*), called *charge*, which will be used to maintain the weights of the nodes of the dynamic tree $\Delta(r)$. Each DT-node corresponding to a vertex of r (a leaf of a lower-level tree) is labelled *distinguished*; the charge of a DT-node is the number of the distinguished nodes in the subtree of which it is the root.

According to its definition, the weight $w(\mu)$ of a node μ of $\Delta(r)$ is the number of the nodes in the subtree of which it is the root, or, equivalently, the number of the arcs in this subtree plus the arc α issuing from μ. It is immediate that, denoting by v the vertex associated with arc α and by

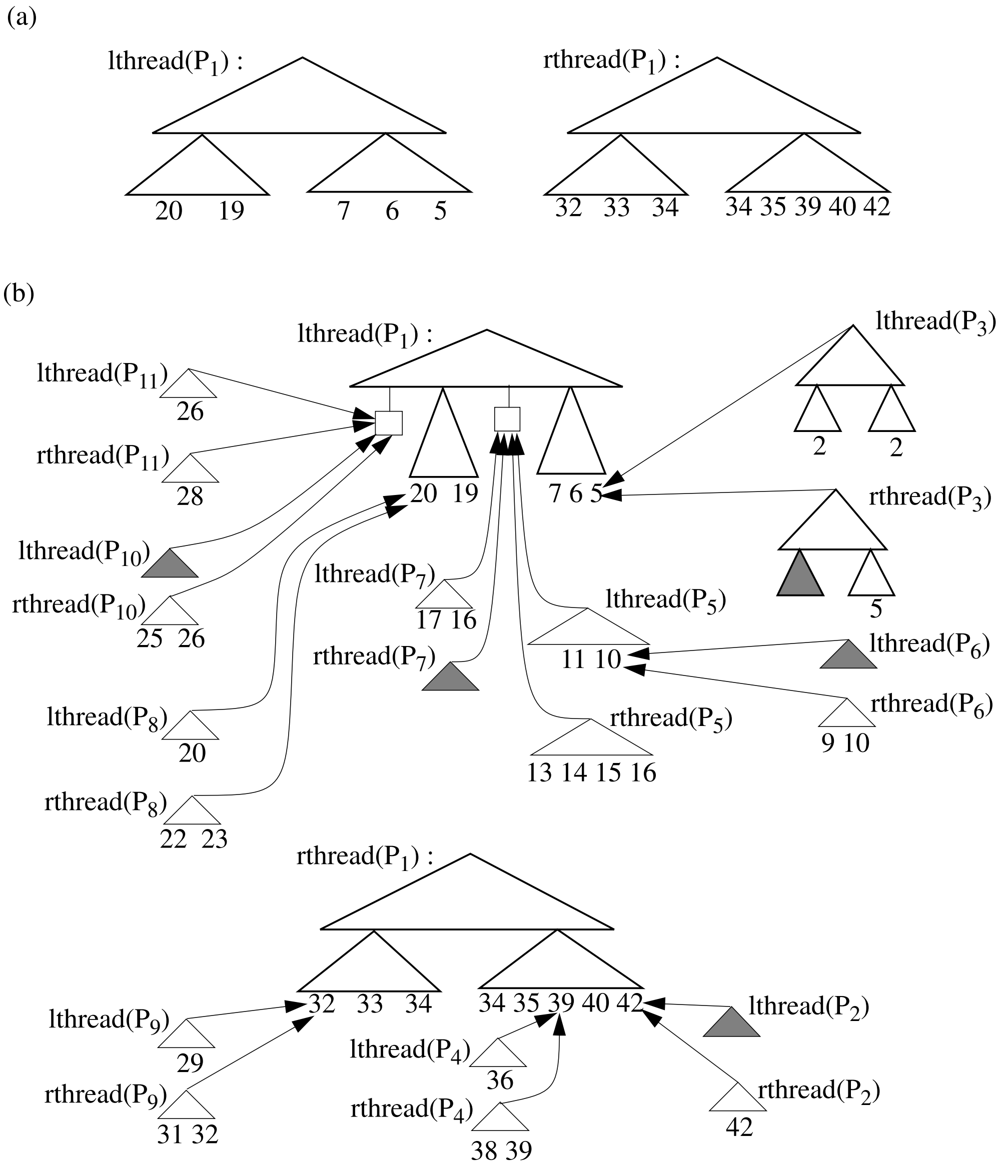


Figure 2.5: Double-thread data structure $DT(r)$ for region r in Fig. 2.1: (a) basic thread trees for P_1; (b) complete structure of $DT(r)$. The bidirectional pointers linking pairs of corresponding thread trees and thread subtrees are omitted.

Q_i the monotone subpath containing α, this number is obtained as the sum of two items: (1) the sum of the charges of all lower-level leaves (actually leaves or couplers) up to and including v in the thread tree containing v, and (2) in the other thread tree, the sum of the charges of all lower-level leaves *preceding* v^*, where v^* is the first vertex on monotone subpath Q_i whose splitter follows the splitter issuing from v, or if v^* does not exist (because Q_i terminates at v), the sum of the charges of all lower-level leaves up to and including the last leaf of the appropriate subtree of Q_i (either $ltree(Q_i)$ or $rtree(Q_i)$). For example, let us look at $w(\mu_1)$ and $w(\mu_2)$ in Figure 2.6. For $w(\mu_2)$, $v^* = s$, thus $w(\mu_2)$ is the sum of the charges of all lower-level leaves of $rthread(P)$ from left up to and including v which corresponds to μ_2, and the charges of all lower-level leaves of $lthread(P)$ up to and including coupler H; for $w(\mu_1)$, v^* does not exist, and thus $w(\mu_1)$ is the sum of the charges of all lower-level leaves of $rthread(P)$ up to and including v which corresponds to μ_1, and the charges of all lower-level leaves of $lthread(P)$ up to and including u. Clearly, we can locate v^* or decide its nonexistence in logarithmic time, using the y-coordinate of v to perform a binary search on either $ltree(Q_i)$ or $rtree(Q_i)$ of the thread tree that does not contain v.

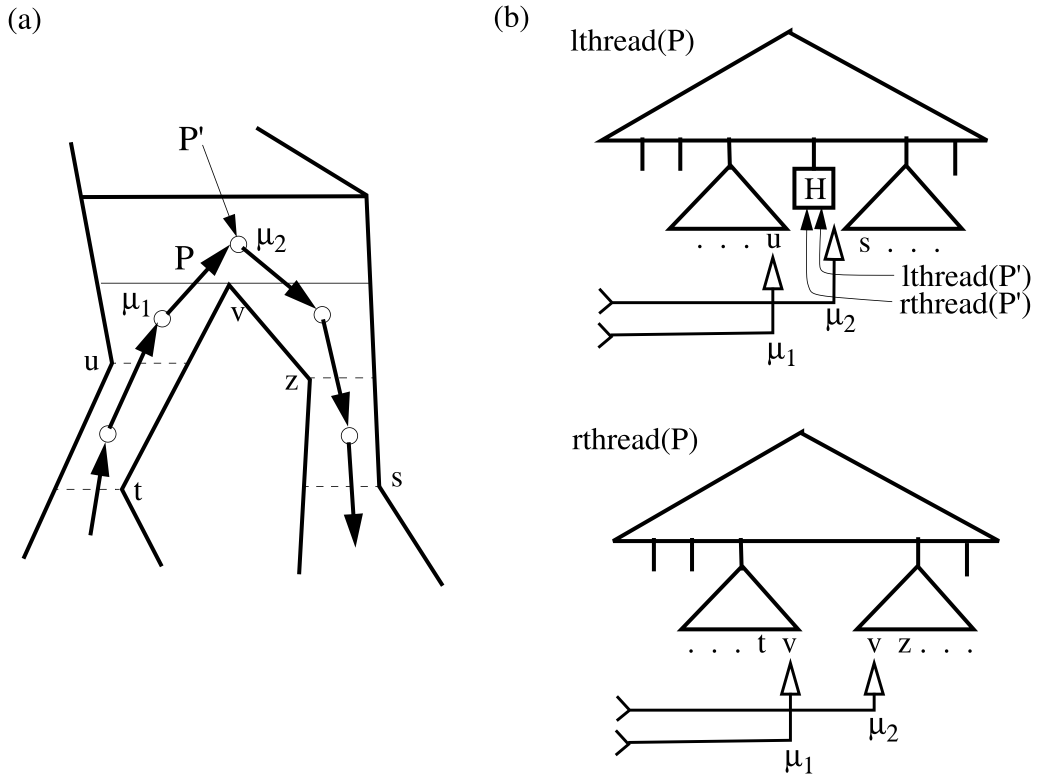


Figure 2.6: Weights $w(\mu_1), w(\mu_2)$ of nodes μ_1, μ_2 of the dynamic tree $\Delta(r)$: $w(\mu_2)$ is the sum of the charges of all lower-level leaves of $rthread(P)$ from left up to and including the occurrence of v which corresponds to μ_2, and the charges of all lower-level leaves of $lthread(P)$ up to and including coupler H; $w(\mu_1)$ is the sum of the charges of all lower-level leaves of $rthread(P)$ up to and including the occurrence of v which corresponds to μ_1, and the charges of all lower-level leaves of $lthread(P)$ up to and including u.

The preceding discussion establishes the following lemma.

Lemma 2.3 *The space complexity of the normalization structure for an n-vertex map is $O(n)$.*

Our data structure has an auxiliary component, called *dictionary*. The *dictionary* stores the names of vertices, edges and regions, so that their representatives occurring in various places in the normalization structure, hull structure and location structure (see Sections 2.4 and 2.5), etc., can be efficiently accessed. The edges of a region r are also maintained in the dictionary by a balanced binary tree according to their circular order, with the root of the tree storing the name of r. We store with each edge e two pointers respectively to its left and right representatives in such trees, so that given e, the region r to its left (resp. right) can be found by accessing its left (resp. right) representative and walking up to the root of the tree of r. It is easy to see that accessing and updating the dictionary can be performed in logarithmic time, and that the dictionary does not affect the space complexity of our data structure. Therefore we omit any further discussion of the dictionary in the rest of the chapter.

2.3.3 Update Operations

We define the following update operations on a connected map $\mathcal{M}$:

INSERTEDGE$(e, v_1, v_2, r; r_1, r_2)$: Insert edge $e = (v_1, v_2)$ into region r such that r is partitioned into two regions r_1 and r_2.

REMOVEEDGE$(e, v_1, v_2, r_1, r_2; r)$: Remove edge $e = (v_1, v_2)$ and merge the regions r_1 and r_2 formerly on the two sides of e into a new region r.

INSERTVERTEX$(v, e; e_1, e_2)$: Split the edge $e = (u, w)$ into two edges $e_1 = (u, v)$ and $e_2 = (v, w)$ by inserting vertex v along e.

REMOVEVERTEX$(v, e_1, e_2; e)$: Let v be a vertex with degree two such that its incident edges $e_1 = (u, v)$ and $e_2 = (v, w)$, are on the same straight line. Remove v and merge e_1 and e_2 into a single edge $e = (u, w)$.

ATTACHVERTEX$(v_1, e; v_2)$: Insert edge $e = (v_1, v_2)$ and degree-one vertex v_2 inside some region r, where v_1 is a vertex of r.

DETACHVERTEX(v, e): Remove a degree-one vertex v and edge e incident on v.

With the above repertory, the following theorem is immediate.

Theorem 2.2 *An arbitrary connected map $\mathcal{M}$ with n vertices can be assembled from the empty map, and disassembled to obtain the empty map, by a sequence of $O(n)$ operations drawn from the set* { *point-location query,* INSERTVERTEX, REMOVEVERTEX, INSERTEDGE, REMOVEEDGE, ATTACHVERTEX, DETACHVERTEX }.

Now we show that ATTACHVERTEX and DETACHVERTEX can be simulated by a sequence of $O(1)$ operations taken among the first four of the repertory and point-location query. Referring for simplicity to ATTACHVERTEX$(v_1, e; v_2)$, we have the following emulation routine: perform a point-location query of v_2 to obtain the region r containing it (which also provides the trapezoid containing v_2), compute the two horizontal projection points v' and v'' of v_2 on the boundary of r, insert vertices v' and v'', insert edge (v', v''), insert vertex v_2 on (v', v''), insert edge e, remove edges (v', v_2) and (v_2, v'') and finally remove vertices v' and v''.

In the rest of this section, we describe how to implement the first four operations of the above repertory on the dynamic tree $\Delta(r)$ of an arbitrary region r (a simple polygon), represented by the double-thread data structure described above.

Primitive dynamic tree operations

We begin by considering some elementary dynamic tree operations *expose, conceal* and *evert* introduced in [104], in terms of which the operations of the above repertory can be immediately expressed. In the course of some updates, we may change a solid arc to dashed and *vice versa* and thus violate the weight invariant; so we need the capability to restore such weight invariant. Such actions are effected by the operation *expose* and *conceal* introduced in [104]. Operation $expose(\mu)$, for some node μ of $\Delta(r)$, transforms the unique path P from node μ to the root of $\Delta(r)$ into a solid path, by changing the dashed arcs in P to solid and the solid arcs incident to P to dashed. Since this transformation may violate the weight invariant of dynamic trees, the inverse operation $conceal(P)$ is used to remove the violation, by identifying all the light arcs in P and making them dashed, and also identifying all heavy arcs (if any) among the arcs incident to P and making them solid.

The primitive operation used in *expose* and *conceal* is $splice(P_1, P_2; P', P'')$, acting on two given paths P_1 and P_2 to produce two new paths P' and P'' (see Fig. 2.7). Originally solid path P_2 points to node μ of solid path P_1 via a dashed arc α. Denoting by α' the (solid) arc of P_1 terminating at μ (if any), *splice* exchanges the roles of α and α', i.e., it creates two new solid paths P' and P'' with P'' pointing to P' via dashed arc α' (again, P'' and α' might be empty).

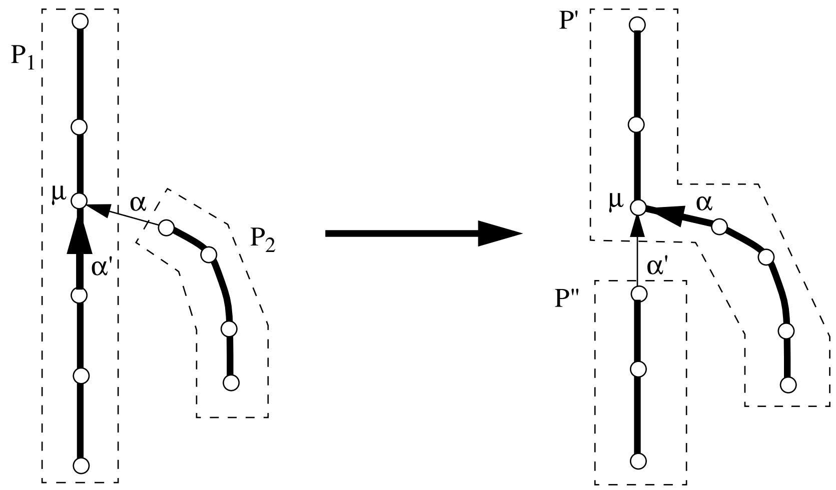


Figure 2.7: Example of $splice(P_1, P_2; P', P'')$.

Operation $splice(P_1, P_2; P', P'')$ essentially involves splitting and concatenating both threads of the paths concerned. Specifically, $lthread(P_1)$ is split into $lthread(P''')$ and $lthread(P'')$, and then $lthread(P_2)$ is concatenated with $lthread(P''')$ to form $lthread(P')$; analogously for *rthread*. Operation *splice* may require either the insertion or the deletion of a coupler (see, for example, splicing P_4 to P_1 (insertion) and P_5 to P_1 (deletion) in Fig. 2.1). Since a constant number of splits/concatenations have to be performed, we have the following lemma.

Lemma 2.4 *Operation splice can be executed in $O(\log m)$ time on the double-thread data structure.*

Since each directed path in $\Delta(r)$ contains at most $\log_2 m$ light arcs by Lemma 2.2 (each accessible by climbing to the root of an $O(\log m)$-depth thread tree), *expose* uses at most $\log_2 m$ *splice* operations, and therefore is executed in $O(\log^2 m)$ time.

Given a solid path P, operation $conceal(P)$ identifies the light arcs of P which have to be made dashed and the heavy arcs (if any) incident to P which have to be made solid in order to comply with the weight invariant of dynamic trees. It can be carried out by finding the topmost (i.e., closest to the root of $\Delta(r)$) light arc α, splitting P at α, removing the subpath from the root up to and including α, and then repeating the process for the remaining solid path, until no light arc is found. The heavy arcs incident to P can then be identified (and made solid) in a straightforward way: each time a light arc (μ, ν) is found, we check all (up to three) arcs incident to ν to see if any one of them is heavy; finally, we also apply this checking process to the arcs incident to the bottommost node of P. So the main issue for performing $conceal(P)$ is how to find the topmost light arc.

Before describing its adaptation to the double-thread data structure, we briefly review the standard implementation of operation *conceal* as proposed by Sleator and Tarjan [104]. Let the dynamic-tree-nodes of solid path P be stored left-to-right as the leaves of a balanced binary tree $T(P)$, called in [104] a *path tree*. Each leaf ζ of $T(P)$ stores $local\_weight(\zeta)$, defined as the sum of the local weights of all dashed-arc children (which are the roots of some other path trees) of

ζ, if any, plus one (to account for ζ itself). For each internal node η, $local\_weight(\eta)$ is defined as the sum of the local weights of its children. Note the similarlity between the *local weights* of nodes in path tree $T(P)$ and *charges* of nodes in thread trees $lthread(P)$ and $rthread(P)$ defined in Section 2.3.2. Actually, parameters local weight and charge are identical except for their usages in computing $w(\mu)$—the weight of a dynamic-tree-node μ: in $T(P)$, $w(\mu)$ is the left to right prefix-sum of the local weights of the leaves, whereas in thread trees, $w(\mu)$ is contributed by the prefix-sums of the charges of the leaves in *both* $lthread(P)$ and $rthread(P)$ (see Section 2.3.2). Let T_η be the subtree of $T(P)$ rooted at internal node η. Denoting by λ the rightmost leaf in T_η, and by ξ the leaf adjacent to λ on the left, variable $lefttilt(\eta)$ is defined by $lefttilt(\eta) \stackrel{\triangle}{=} w(\xi) - local\_weight(\lambda)$. We recall that arc (ξ, λ) of P is light if and only if $w(\xi) < \frac{1}{2}w(\lambda)$, i.e., $w(\xi) < \frac{1}{2}(w(\xi) + local\_weight(\lambda))$, which yields $lefttilt(\eta) < 0$.

Moreover, define $leftmin(\eta) \stackrel{\triangle}{=} min\{lefttilt(\theta) : \theta \text{ is an internal node of } T_\eta\}$. It follows that if $leftmin(\eta) \geq 0$, then there is no light arc between any two adjacent leaves of T_η. Also, variable $netleft(\eta)$ is defined as $leftmin(\eta)$ if η is the root of $T(P)$ and $leftmin(\eta) - leftmin(parent(\eta))$ otherwise. Correspondingly, variables $netright(\eta)$, $rightmin(\eta)$, and $righttilt(\eta)$ are defined symmetrically in a straightforward manner by summing the local weights from right to left for the purpose of reversing the path direction. In summary, each internal node η of $T(P)$ stores three values: $local\_weight(\eta)$, $netleft(\eta)$, and $netright(\eta)$.

To find the topmost light arc in P, we traverse a path from the root of $T(P)$ with the following advancing mechanism: Assume inductively that, for the current node η, parameter $leftmin(\eta)$ (< 0) is known. Let η' and η'' be the left and right children of η, respectively, and ξ be the leftmost leaf of $T_{\eta''}$. From the definition

$$netleft(\eta'') = leftmin(\eta'') - leftmin(\eta)$$

we obtain $leftmin(\eta'')$. If $leftmin(\eta'') < 0$, then we proceed to η''. Otherwise, we compare $local\_weight(\eta')$ and $local\_weight(\xi)$. If $local\_weight(\eta') < local\_weight(\xi)$, then the arc leading to ξ is the sought light arc; else, we compute $leftmin(\eta') = leftmin(\eta) + netleft(\eta')$ (which is necessarily < 0) and proceed to η' (this establishs the inductive step). By this process akin to binary search, the topmost light arc can be found in $O(\log m)$ time. Recall that by Lemma 2.2, there are at most $\log_2 m$ light arcs in $T(P)$.

We are now ready to consider the implementation of *conceal* for the double-thread data structure. We treat thread trees $lthread(P)$ and $rthread(P)$ independently as two path trees, with parameter *charge* playing the role of *local weight*. By the method just illustrated, we identify at most $\log_2 m$ light arcs from each of $lthread(P)$ and $rthread(P)$. Note that a light arc (ξ, λ) in P assures the existence of a light arc (ξ', λ) in either $lthread(P)$ or $rthread(P)$ that contains leaf λ, where ξ' is the left-neighboring leaf of λ. Indeed, in $T(P)$, the sum $w(\xi)$ of the local weights up to and including ξ satisfies $w(\xi) < local\_weight(\lambda)$; but in the appropriate thread tree (i.e., either $lthread(P)$ or $rthread(P)$ that contains λ), the sum w' of the charges up to and including ξ' is only a fraction of $w(\xi)$ ($w(\xi)$ is contributed by *both* $lthread(P)$ and $rthread(P)$), so that $w' \leq w(\xi) < local\_weight(\lambda) = charge(\lambda)$, and (ξ', λ) is light. Hence $2\log_2 m$ light arcs from $lthread(P)$ and $rthread(P)$ give all possible candidates for light arcs in P. For each such candidate (ξ', λ), we perform a binary search in the paired thread tree to locate the point just before λ, at the same time accumulate the total charge w'' up to and including this point in that tree, then compute $w(\xi)$ by adding w'' to w', and check if $w(\xi) < charge(\lambda)$ $(= local\_weight(\lambda))$. Therefore, we find $2\log_2 m$ candidates, perform $2\log_2 m$ binary searches for checking, identify at most $\log_2 m$ light arcs in P (and also at most $\log_2 m + 1$ heavy arcs incident to P), and then split and join P accordingly—each of these operations within $O(\log m)$ time. This leads to the following lemma.

Lemma 2.5 *The update of the double-thread data structure as required by the operation conceal can be performed in $O(\log^2 m)$ time.*

Operation $evert(\mu)$, for an arbitray node μ of $\Delta(r)$, moves the root of $\Delta(r)$ to μ while preserving the weight invariant. If we can reverse the direction of a solid path, then $evert(\mu)$ can be carried out as follows: we perform $expose(\mu)$ to obtain a solid path P from μ to the original root, reverse the direction of P (which effectively moves the root to μ), and then perform $conceal(P)$ to comply with the weight invariant. We add a "direction" bit to each node of the thread trees, so that when we reverse the direction of a solid path P, the direction bit of the root of $lthread(P)$ is complemented, indicating that the meanings of left and right subtrees of $lthread(P)$ are interchanged; similarly for the direction bit of the root of $rthread(P)$. Also, these two complemented bits indicate that $lthread(P)$ means $rthread(P)$ and *vice versa.* Given the direction bits and operations *expose* and *conceal*, we can perform *evert* in the double-thread data structure in $O(\log^2 m)$ time.

In the following, if μ is a node of $\Delta(r)$ and a an incoming arc of μ, the notations $expose(a)$ and $expose(\mu)$ are equivalent, and similarly for *evert*.

Lemma 2.6 *Given splitters s_1 and s_2 of region r with m vertices,* SLEEVE(s_1, s_2) *and the corresponding solid path between $\delta(s_1)$ and $\delta(s_2)$ can be constructed in $O(\log^2 m)$ time, by means of $O(\log^2 m)$ elementary splits/joins of thread trees.*

Proof: We obtain a solid path between $\delta(s_1)$ and $\delta(s_2)$ by $evert(\delta(s_1))$ and $expose(\delta(s_2))$. Each of operations *evert* and *expose* uses $O(\log^2 m)$ elementary splits/joins and takes $O(\log^2 m)$ time. □

The double-thread structure adds two new primitive operations to the original repertory of dynamic trees. Operation $part(P, e; P_1, P_2)$ on a solid monotone path P separates $lthread(P)$ and $rthread(P)$, and creates two new solid paths P_1 and P_2 by adjoining $lthread(P)$ and $rthread(P)$ to a new edge e. Namely, $lthread(P_1) = lthread(P)$, $rthread(P_1) = e$, $lthread(P_2) = e$, and $rthread(P_2) = rthread(P)$. The operation $pair(P_1, P_2; P, e)$ is the inverse operation of $part(P, e; P_1, P_2)$ and is implemented similarly.

Lemma 2.7 *Operations part and pair have time complexity $O(1)$.*

As we shall see in the next section, operations *part* and *pair* are crucial in the efficient execution of INSERTEDGE and REMOVEEDGE.

Insertion and deletion of edges and vertices

Operation INSERTEDGE$(e, v_1, v_2, r; r_1, r_2)$ is carried out as follows:

1. For $i = 1, 2$, if v_i is an extreme vertex of r, let $s_i = v_i$, else let s_i be a splitter of r induced by v_i. If v_i is a cusp of r then there are two such splitters; by viewing edge $e = (v_1, v_2)$ as issuing from v_i, s_i is taken as the left splitter of v_i if e goes toward left (and as the right splitter otherwise), so that SLEEVE(s_1, s_2) is the smallest monotone sleeve that contains e.
2. Construct SLEEVE(s_1, s_2) and the corresponding solid path P, by performing $evert(\delta(s_1))$ and then $expose(\delta(s_2))$.
3. Insert edge e by performing $part(P, e; P_1, P_2)$, so that there are new solid paths P_1 and P_2 respectively in new regions r_1 and r_2.
4. For each of the (up to three) solid paths previously pointing to the head of P, make it point to the head of P_1 if it lies in r_1, and to the head of P_2 if it lies in r_2; similarly for the solid paths previously pointing to the tail of P. Note that P_1 and P_2 have the same orientation as P.

5. Create a new dynamic tree $\Delta(r_1)$ for r_1, by putting the root at the end of P_1 that is closer to v_1 (which does not change the direction of P_1), then performing operation *conceal*(P_1); similarly create a new dynamic tree $\Delta(r_2)$. Note that the *conceal* operations readily splice the solid paths pointing to the heads and tails of P_1 and of P_2 if necessary.

We analyze the time complexity of the above operation. Steps 1, 3 and 4 take $O(1)$ time, and the other steps globally involve a fixed number of *evert*, *expose* and *conceal* operations, so that the total time required for updating the double-thread data structure is $O(\log^2 m)$.

Operation REMOVEEDGE$(e, v_1, v_2, r_1, r_2; r)$ is the inverse operation of INSERTEDGE. We first evert v_1 and then expose v_2 in both $\Delta(r_1)$ and $\Delta(r_2)$, pair the two solid paths into one, and conceal it. This can also be done in $O(\log^2 m)$ time.

Operation INSERTVERTEX$(v, e; e_1, e_2)$ is performed as follows. We insert v with $charge(v) = 1$ into $lthread(P)$ and $rthread(P')$ for some solid paths P and P' of different regions r and r', where both $lthread(P)$ and $rthread(P')$ contain two endpoints v_1 and v_2 of e. In dynamic tree $\Delta(r)$, we perform *expose* on the one of v_1 and v_2 that is farther from the root to obtain a solid path, and then perform *conceal* on this path; in $\Delta(r')$ we perform exactly the same operations. It is easy to see that operation INSERTVERTEX is executed in $O(\log^2 m)$ time. Operation REMOVEVERTEX is the inverse operation of INSERTVERTEX, and can be completed within the same time bound.

2.4 Shortest Path Queries

In this section we illustrate how the normalization data structure can be modified, by appending secondary data structures collectively called *hull structure*, to answer the following queries:

PATHLENGTH(q_1, q_2, r): Return the length of a shortest path inside region r between query points q_1 and q_2.

PATH(q_1, q_2, r): Return the shortest path inside region r between query points q_1 and q_2 as a chain of segments.

First, by point location (see Section 2.5) we can check whether q_1 and q_2 belong to r. Note that we need to specify within which region the shortest path is sought to avoid ambiguities when both q_1 and q_2 belong to edges of the map. We now show that the above queries can be supported in worst-case time $O(\log^3 n)$ and $O(\log^3 n + k)$, respectively, where k is the number of segments in the shortest path reported by PATH.

The notion of *hourglass* is central to our current problem. We adopt the terminology proposed by Guibas and Hershberger [63].

Consider two nonintersecting diagonals $s_1 = (a_1, b_1)$ and $s_2 = (a_2, b_2)$ of r, where the endpoints have been named so that the counterclockwise cyclic sequence of points in the boundary of r includes the subsequence (a_1, a_2, b_2, b_1). The *hourglass* of s_1 and s_2, denoted HOURGLASS(s_1, s_2), is the subregion of r formed by the union of all the shortest paths PATH(q_1, q_2, r) with $q_1 \in s_1$ and $q_2 \in s_2$ (see Fig. 2.8.b). It is known that the boundary of HOURGLASS(s_1, s_2) is the concatenation of s_1, PATH(a_1, a_2, r), s_2, and PATH(b_2, b_1, r). Let α be the subchain of r counterclockwise from a_1 to a_2, and define β similarly for b_2 and b_1. The hourglass has one of the following special structures (as analyzed by [63]):

Open hourglass: If the convex hulls inside r of α and β do not intersect, then PATH(a_1, a_2, r) is the convex hull of the subchain of α clockwise from a_1 to a_2, and similarly for PATH(b_2, b_1, r).

Closed hourglass: If the convex hulls of α and β intersect, then there exist vertices p_1 and p_2 of $\alpha \cup \beta$ such that PATH$(a_1, a_2, r) \cap$ PATH$(b_1, b_2, r) =$ PATH(p_1, p_2, r). Without loss of generality, assume that p_1 is in α. Then $\text{PATH}(a_1, p_1, r)$ is the convex hull inside r of the subchain of α from a_1 to p_1, while $\text{PATH}(b_1, p_1, r)$ is the union of segment (p_1, p'_1) and the convex hull inside r of the subchain of β from b_1 to p'_1, where p'_1 is the vertex of β closer to b_1 on the two tangents from p_1 to β. Similar arguments apply to p_2. The union of $\text{PATH}(a_i, p_i, r)$ and $\text{PATH}(b_i, p_i, r)$ ($i = 1$ or 2) is called a *funnel* [76]. Vertices p_1 and p_2 are called the *apices* of the hourglass, and the path between them the *string* of the hourglass (see Fig. 2.8.b).

If we represent an hourglass by its string and the (two to four) convex chains forming the rest of its boundary, and for each polygonal chain represented we also store its length, then given $\text{HOURGLASS}(s_1, s_2)$, it is possible to compute $\text{PATHLENGTH}(q_1, q_2, r)$ in $O(\log n)$ time for any two points $q_1 \in s_1$ and $q_2 \in s_2$ by means of $O(1)$ common tangent computations. Also, given $\text{HOURGLASS}(s_1, s_2)$ and $\text{HOURGLASS}(s_2, s_3)$ in r, with s_1 and s_3 on opposite sides of the line containing s_2, it is possible to compute $\text{HOURGLASS}(s_1, s_3)$ in time $O(\log n)$ by means of $O(1)$ common tangent computations and $O(1)$ split and join operations on the chains forming the two hourglasses.

We now consider the modifications of the normalization data structure that enable the support of the given path queries. As we shall see, only three items are needed, i.e.:

(i) The choice of an appropriate implementation of the trees *ltree* and *rtree* introduced in Section 2.3.2;

(ii) The appending of secondary data structures (collectively called "hull structure") to the nodes of *ltree*'s and *rtree*'s. The *hull structure* stores at the nodes of *ltree*'s and *rtree*'s the hourglasses of the corresponding sleeves; it establishes an implicit correspondence between the two chains of a monotone sleeve, allowing both efficient access to the hourglass of the sleeve, and fast pairing or parting of the two chains as required by edge insertion or deletion;

(iii) A separate BB[α]-tree $\mathcal{Y}$ (called *Y-tree*) that determines a hierarchical partition of the plane into horizontal strips, according to which *ltree*'s and *rtree*'s are implemented.

We first describe the adopted representation of polygonal chains. A concatenable queue, called *chain tree*, will be used to represent a polygonal chain γ. The chain tree T for γ is a balanced tree and has inorder thread pointers. Each node μ of T corresponds to a subchain γ_μ of γ and stores the endpoints of γ_μ, the common point of the subchains of the children of γ_μ, and the length of γ_μ. It should be clear that this information can be updated in $O(1)$ time per elementary join or split, so that splitting or splicing two chain trees takes logarithmic time. With this representation, it is possible to find the two tangents from a point to a convex chain and the four common tangents between two convex chains in logarithmic time [93].

We now give the details of our representation of hourglasses. An open hourglass is represented by storing its two convex chains into chain trees. A closed hourglass is represented by storing into separate substructures the four convex chains forming the funnels, and the string between the apices. The convex chains of the funnels and the string are each stored into a chain tree.

Without loss of generality we assume that the degree of each vertex of $\mathcal{M}$ is at most three. This is not restrictive since we can expand a vertex v with degree $d > 3$ into a chain of degree-3 vertices connected by edges of infinitesimal length. Since the sum of the degrees of all vertices of $\mathcal{M}$ is $O(n)$, the total number of vertices after the expansion is still $O(n)$. Every update operation in the original map $\mathcal{M}$ can be simulated with $O(1)$ operations in the modified map with bounded-degree vertices.

We consider the ordered sequence Y of the y-coordinates of the vertices of $\mathcal{M}$, and establish a one-to-one correspondence between Y and the leaves of a BB[α]-tree $\mathcal{Y}$, called *Y-tree*, which is added as a separate tree into the data structure. Tree $\mathcal{Y}$ determines a hierarchical partition of the plane into horizontal strips according to the well-known segment-tree scheme. Each node of

$\mathcal{Y}$ corresponds to a *canonical interval* of y-coordinates. A vertical interval (y', y'') with $y', y'' \in Y$ is uniquely partitioned into $O(\log n)$ canonical intervals, called the *fragments* of (y', y''), and their associated nodes in $\mathcal{Y}$ are called the *allocation nodes* of (y', y''). We extend this terminology to any geometric entity that is uniquely associated with a vertical interval, such as an edge, a monotone chain, or a monotone sleeve.

We now introduce the useful notion of "pruned tree". A *pruned tree* of a rooted tree T is a tree S that can be obtained from T by removing from it the subtrees rooted at a selected subset of its nodes. Pruned trees of a balanced tree T support the full repertory of concatenable queue operations. Each operation takes $O(\log n)$ time and is performed by means of $O(\log n)$ elementary joins and splits between pruned trees whose roots are associated with sibling nodes in T. A sequence I of k consecutive intervals with endpoints in Y will be stored in a pruned subtree of $\mathcal{Y}$, whose leaves are the allocation nodes of the intervals of I, and whose internal nodes are the ancestors of such leaves. It is easy to verify that the pruned tree associated with I has $O(k \log n)$ nodes and $O(\log n)$ height.

Now, we show how to modify the normalization structure so that hourglasses can be dynamically maintained (see Fig. 2.8). We denote with Q a maximal monotone subpath of a solid path P and specify the implementation of $ltree(Q)$ and $rtree(Q)$. We use pruned trees augmented with chain-trees as secondary structures. Our scheme uses ideas from [90] and [63].

- Trees $ltree(Q)$ and $rtree(Q)$ are implemented by means of pruned trees with respect to $\mathcal{Y}$.
- Let μ be a node of $ltree(Q)$ (nodes of $rtree(Q)$ are handled identically) and ν the parent of μ. Node μ has a pointer to the corresponding node y of $\mathcal{Y}$. Also, if μ is not a leaf, then we establish a back pointer from y to μ. We do not set up back pointers from y to leaves of $ltree(Q)$ (or of $rtree(Q)$) in order to obtain efficient updates, as we shall see later. Consider the subpath Q' of Q associated with the subtree of $ltree(Q)$ rooted at μ. We denote with $\textsc{sleeve}(\mu)$ the sleeve of Q', with s_1 and s_2 the splitters that delimit $\textsc{sleeve}(\mu)$, with $\textsc{hourglass}(\mu)$ the hourglass of s_1 and s_2 (namely, $\textsc{hourglass}(s_1, s_2)$), and with $\textsc{chain}(\mu)$ the "left chain" of $\textsc{sleeve}(\mu)$, i.e., the chain formed by the edge-fragments stored at the leaves of the subtree of $ltree(Q)$ rooted at μ.
  We distinguish several subcases:
  - If μ is a leaf of $ltree(Q)$, then μ stores the corresponding edge fragment.
  - If $\textsc{hourglass}(\mu)$ is open and $\textsc{hourglass}(\nu)$ is closed, then μ stores in a secondary data structure the right convex hull of $\textsc{chain}(\mu)$.
  - If both $\textsc{hourglass}(\mu)$ and $\textsc{hourglass}(\nu)$ are open, then μ stores in a secondary data structure only the endpoints of $\textsc{chain}(\mu)$ and the portion of the right hull of $\textsc{chain}(\mu)$ that is not stored at an ancestor of μ.
  - If $\textsc{hourglass}(\mu)$ is closed and μ is the root of $ltree(Q)$, then μ stores in secondary data structure the (up to five) components of $\textsc{hourglass}(\mu)$.
  - If $\textsc{hourglass}(\mu)$ is closed and μ is not the root, then μ stores the apices and the length of the string of $\textsc{hourglass}(\mu)$, plus the subchains of the funnels of $\textsc{hourglass}(\mu)$ that are not stored at the ancestors of μ.
- The upper levels (see Section 2.3.2) of thread trees $lthread(P)$ and $rthread(P)$ are essentially identical (except for the couplers). Also, an internal node μ in the upper level of $lthread(P)$ stores the length and the endpoints of the string of $\textsc{hourglass}(\mu)$. The corresponding node of $rthread(P)$ stores exactly the same information.

Lemma 2.8 *The space requirement of the hull structure is $O(n \log n)$.*

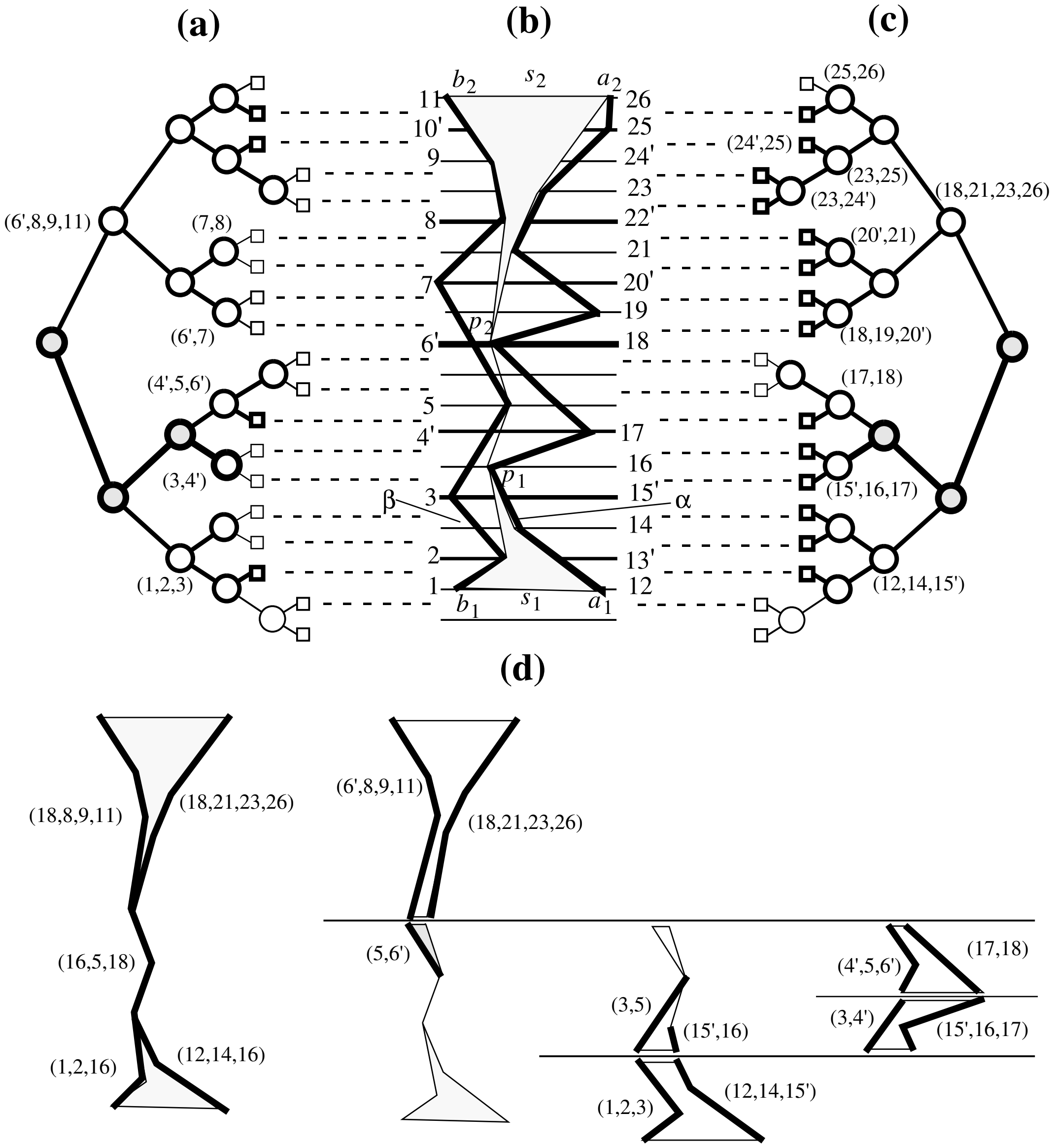


Figure 2.8: Example of representation of hourglasses in the nodes of *ltree*(Q) and *rtree*(Q) of a monotone path Q. **(b)** The sleeve of Q (directed from left to right): the parallel lines drawn on it represent set Y; the points on the sleeve with labels of the type i' delimit fragments of the same edge; the hourglass between the extreme splitters of the sleeve is shown grey-filled. **(a)** Pruned-tree *ltree*(Q): the nodes of *ltree*(Q) are those drawn with thick lines, while the nodes drawn with thin lines denote the subtrees of $\mathcal{Y}$ pruned away to construct *ltree*(Q); the grey-filled nodes are associated with closed sleeves, and the white-filled nodes are associated with open sleeves. Next to each white-filled node μ we show the subchain of HOURGLASS(μ) stored at μ. **(c)** Pruned-tree *rtree*(Q) (similar comments as in (a) apply). **(d)** Hourglasses of the grey-filled nodes and of their children. The subchains stored at each node are labeled and shown with thick lines.

Proof: We only need to determine the space used by the secondary structures (the chain-trees) that augment the *ltree*'s and *rtree*'s. Consider the set S of all segments s such that s is either an edge-fragment or the tangent segment in the hourglass of a node in *ltree* or *rtree*. We claim that the size of S is $O(n \log n)$. By standard segment-tree arguments, the number of edge-fragments in S is $O(n \log n)$. For the tangent segments, consider the hourglasses $\text{HOURGLASS}(\mu)$, $\text{HOURGLASS}(\mu')$ and $\text{HOURGLASS}(\mu'')$ of a node μ and its children μ' and μ''. Note that $\text{SLEEVE}(\mu')$ and $\text{SLEEVE}(\mu'')$ share a common splitter, say s_2, and the other splitters s_1 of $\text{SLEEVE}(\mu')$ and s_3 of $\text{SLEEVE}(\mu'')$ lie on opposite sides of s_2. It follows that $\text{HOURGLASS}(\mu) = \text{HOURGLASS}(s_1, s_3)$ is obtained from $\text{HOURGLASS}(\mu') = \text{HOURGLASS}(s_1, s_2)$ and $\text{HOURGLASS}(\mu'') = \text{HOURGLASS}(s_2, s_3)$ by $O(1)$ common tangent computations, and thus each node μ contains $O(1)$ tangent segments. Again, by segment-tree arguments, the total number of nodes in *ltree*'s and *rtree*'s is $O(n \log n)$, hence the total number of tangent segments in S is $O(n \log n)$. Also, each segment of S is stored $O(1)$ times in the data structure, since it can have representatives in an allocation node (for an edge fragment), in the the highest open hourglass, and in the highest monotone hourglass, and there may be two such nodes for each segment (recall that edges have two "sides", and the corresponding nodes in the paired *ltree* and *rtree* may have duplicate information). We conclude that the secondary structures are a collection of balanced trees with a total of $O(n \log n)$ nodes, and hence use total space $O(n \log n)$. □

Query operations $\text{PATHLENGTH}(q_1, q_2, r)$ and $\text{PATH}(q_1, q_2, r)$ are performed as follows:

1. Find the trapezoids τ_1 and τ_2 of the trapezoidal decomposition of r containing q_1 and q_2, using the point-location machinery of Section 2.5. Let s_1 and s_2 be the splitters on the boundary of τ_1 and τ_2, such that q_1 and q_2 are on opposite sides of $\text{SLEEVE}(s_1, s_2)$.
2. Create the solid path P for $\text{SLEEVE}(s_1, s_2)$ (P is the path between edges $\delta(s_1)$ and $\delta(s_2)$ of $\delta(r)$), by means of *evert*$(\delta(s_1))$ and *expose*$(\delta(s_2))$. The secondary structure stored at the root of *lthread*(P) (or *rthread*(P)) yields a representation of $\text{HOURGLASS}(s_1, s_2)$.

Given the representation of $\text{HOURGLASS}(s_1, s_2)$, after computing in time $O(\log n)$ the tangents from q_1 and q_2 to the appropriate funnels, we can answer $\text{PATHLENGTH}(q_1, q_2, r)$ and $\text{PATH}(q_1, q_2, r)$ in time $O(1)$ and $O(k)$, respectively (where k is the number of edges of the shortest path reported). Finally, we conceal the path exposed in step 2 to satisfy the weight invariant.

Regarding updates, we have (see the example in Fig. 2.9):

Lemma 2.9 *An elementary split or join of two thread trees in the normalization structure augmented with the hull structure takes time* $O(\log n)$.

Proof: After an elementary split or join of two solid thread trees, we need to update only the secondary data structures of their roots. Since such data structures represent the hourglasses of the corresponding sleeves, they can be updated in $O(\log n)$ time (see Fig. 2.9). Note that for a nonmonotone solid path the updates are limited to its leftmost or rightmost monotone subpath. For this reason it is sufficient to store only the length of the string in the nodes of the upper levels of the *lthread* and *rthread* trees. □

As a consequence, splitting a solid path or joining two solid paths takes time $O(\log^2 n)$. Note that parting or pairing *ltree*(Q) and *rtree*(Q) of a monotone path Q (because of an edge insertion or deletion in the corresponding sleeve) takes $O(1)$ time. The lemma below follows from Lemmas 2.6 and 2.9.

Lemma 2.10 *Queries* $\text{PATHLENGTH}(q_1, q_2, r)$ *and* $\text{PATH}(q_1, q_2, r)$ *are performed in time* $O(\log^3 n)$ *and* $O(\log^3 n + k)$*, respectively, where* k *is the number of edges of the shortest path reported.*

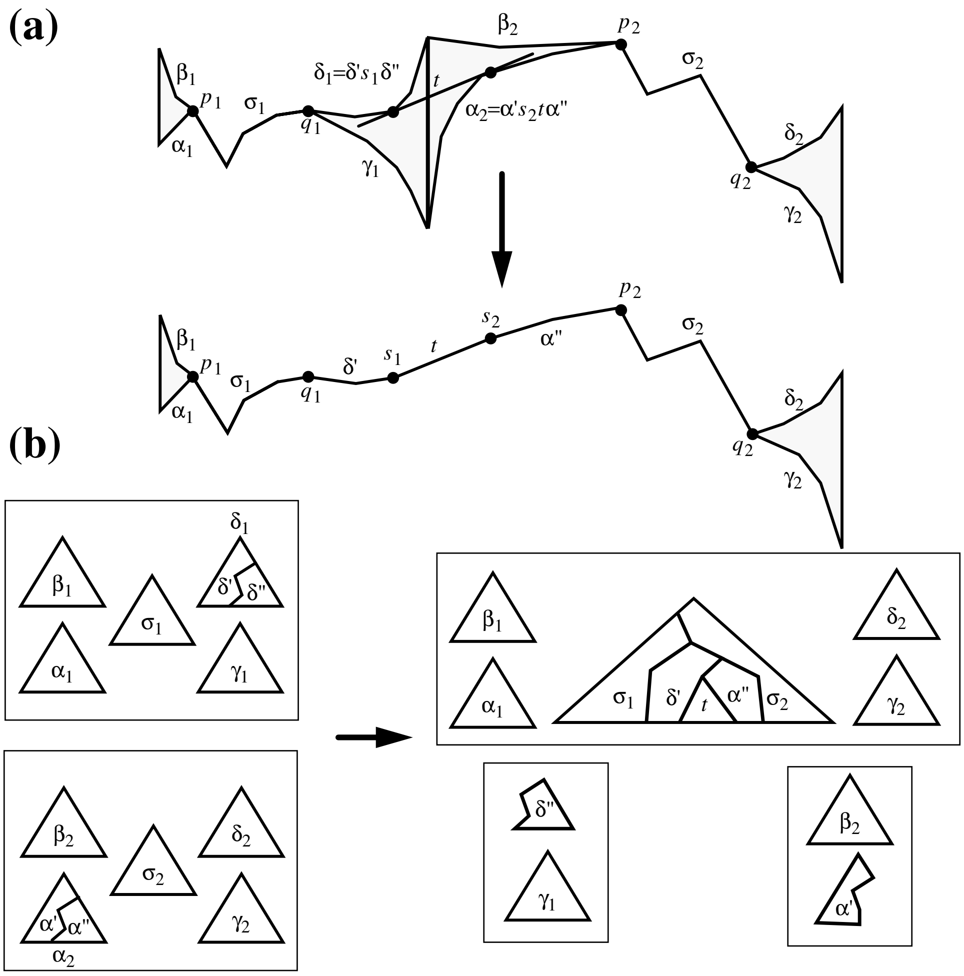


Figure 2.9: Example of update of the secondary structures in an elementary join of two solid paths. **(a)** Geometric construction of the hourglass. **(b)** Construction of the representation of the root hourglass by means of split and join operations on the chain-trees in the representation of the hourglasses of the children nodes.

Now, we discuss how operation INSERTVERTEX$(v, e; e_1, e_2)$ affects the new data structure. First, we insert a new node $y(v)$ into $\mathcal{Y}$. Let μ be one of the two nodes in the *ltree* and *rtree* that stores the fragment of edge e where v is inserted, and let y be the corresponding node of $\mathcal{Y}$. Before the insertion of v, there is a pointer from μ to y but no back pointer from y to μ, since μ is a leaf of a pruned tree. After the insertion of v, the fragment of e stored in μ is further partitioned into $O(\log n)$ fragments (but the total number of fragments of e is still $O(\log n)$) according to the subtree of $\mathcal{Y}$ rooted at y ($y(v)$ has already been inserted into this subtree); we allocate these edge fragments into a new tree T_μ, expand leaf μ to T_μ, establish a pointer from y to μ, and rename all fragments of e to e_1 or e_2 appropriately.

The insertion of $y(v)$ into $\mathcal{Y}$ may cause rebalancing operations in $\mathcal{Y}$ carried out by means of rotations. A rotation between a node y' and its child y'' implies that horizontal cuts at y'' now take priority over horizontal cuts at y'. It is easy to see that the rotation only affects the subtrees of the *ltree*'s and *rtree*'s rooted at the nodes pointed to by y'. We rebuild such subtrees from scratch, which can be done in time proportional to their size. Note that prior to the rotation, a leaf μ of *ltree* or *rtree* corresponding to y' stores an edge fragment that spans the canonical vertical interval I of y', and thus μ is not affected by the rotation (except that after the rotation we have to redirect the original pointer of μ to y' so that it now points to y'', since y'' now corresponds to I). Since there may be a large number of such leaves μ that do not require rebuilding, we do not establish a back pointer from y' to leaf μ in our data structure (as we have already seen), so that inefficient checking for the necessity of rebuilding is avoided. Also, the redirection of all pointers of leaves μ from y' to y'' can be done efficiently when we rotate y' with y'': we switch the contents of the *physical* nodes y' and y'' to interchange the roles of the physical nodes y' and y'' (and then carry out the rotation appropriately by $O(1)$ elementary splits and joins), so that all these pointers are effectively redirected, though no actual changes are made to the pointers. Now we show that the rebuilding of the subtrees of *ltree*'s and *rtree*'s caused by a rotation in $\mathcal{Y}$ can be performed efficiently.

Lemma 2.11 *Let y be a node of $\mathcal{Y}$ whose subtree has ℓ leaves. The subtrees of ltree's and rtree's with the hull structure appended whose roots are pointed to by node y have total size $O(\ell \log \ell)$, and can be built in time $O(\ell \log \ell)$.*

Proof: The subtree of $\mathcal{Y}$ rooted at y has exactly $2\ell - 1$ nodes. Thus there are $O(\ell)$ vertices inside the canonical vertical interval I of node y. The leaves of the subtree rooted at a node pointed by y store the edge fragments that are inside I but do not span I. Hence, the edges contributing to such fragments must be incident on some vertex inside I. Since each vertex has bounded degree, there are $O(\ell)$ such edges. Also, since the subtree of $\mathcal{Y}$ has height $O(\log \ell)$, each such edge has $O(\log \ell)$ fragments inside I. We conclude that the total number of leaves in the subtrees rooted at node pointed by y is $O(\ell \log \ell)$, and hence their total size is also $O(\ell \log \ell)$. □

By the properties of BB[α]-trees, we derive the following lemma.

Lemma 2.12 *The amortized rebalancing time of the Y-tree $\mathcal{Y}$ in a sequence of update operations is $O(\log^2 n)$.*

We conclude:

Theorem 2.3 *Shortest path queries* PATHLENGTH(q_1, q_2, r) *and* PATH(q_1, q_2, r) *in an n-vertex connected planar map can be performed in worst-case time $O(\log^3 n)$ and $O(\log^3 n + k)$, respectively (where k is the number of edges of the shortest path reported), using a fully dynamic data structure that uses space $O(n \log n)$ and supports updates of the map in time $O(\log^3 n)$ (amortized for vertex updates).*

Remark. In a concrete situation where vertices are a priori restricted to a fixed set of ordinates, tree $\mathcal{Y}$ is static; if we then implement the trees *ltree* and *rtree* by means of *contracted binary trees* [96] of depth $< \log |Y|$ (whose maintenance requires no rotation), then the update times become $O(\log^2 n \log |Y|)$, in the worst case.

The following are two additional types of queries that can be supported by the described data structure without any modification:

TRAILLENGTH$(q_1, q_2 | e_1, ..., e_\ell)$: allowing edges e_1,...,e_ℓ to be deleted, are points q_1 and q_2 reachable to each other? if so, then return the length of the shortest path.

TRAIL$(q_1, q_2 | e_1, ..., e_\ell)$: allowing edges e_1,...,e_ℓ to be deleted, are points q_1 and q_2 reachable to each other? if so, then return the shortest path.

An immediate application is that viewing the edges of the map as walls, we are allowed to put doors on edges e_1,...,e_ℓ; can a point-like robot at position q_1 reach position q_2? If so, then report the shortest path or its length.

Clearly, by using REMOVEEDGE, point-location query (see Section 2.5), PATHLENGTH or PATH, and INSERTEDGE operations, queries TRAILLENGTH and TRAIL can be answered in worst-case time $O((\ell+1)\log^3 n)$ and $O((\ell+1)\log^3 n + k)$, respectively, where k is the number of edges of the shortest path reported.

2.5 Point Location Queries

In this section we consider the problem of answering point-location queries:

LOCATE(q): Find the region containing query point q. If q is on an vertex or edge, then return that vertex or edge.

Our dynamic point-location data structure is inspired by the static trapezoid method [92] and its dynamic version for monotone maps [28]. It uses the normalization and hull structures as the underpinning of update operations. Queries are instead performed in a location structure, a binary tree called *trapezoid tree.*

The trapezoid tree defines a binary partition of the plane obtained by means of vertical and horizontal cuts. It differs in many substantial aspects from the trapezoid trees used in [28, 92], the most striking difference being that it is not balanced.

The trapezoid tree $\mathcal{T}$ for map $\mathcal{M}$ is based on the *Y-tree* $\mathcal{Y}$ (see Section 2.4) and on the normalization of $\mathcal{M}$ as reflected by the normalization structure (see Section 2.3). We view the unnormalized map $\mathcal{M}$ as a trapezoid with its sides at infinity. If a trapezoid τ contains more than a single edge fragment in its interior, we recursively decompose it into trapezoids whose vertical spans are canonical vertical intervals, according to the following rules (see Fig. 2.10):

Vertical cut: If τ is a coupler or is vertically spanned by a monotone subpath Q and the hourglass H of SLEEVE(Q) is open, we decompose τ by one of the supporting tangents t of H.

Horizontal cut: If no vertical cut is possible, then we decompose τ by cutting it along the horizontal line at the y-coordinate associated with the (unique) allocation node of τ in $\mathcal{Y}$.

Note that a vertical cut always takes priority over a horizontal cut. If more vertical cuts are possible, their order is arbitrary. We represent the above decomposition of $\mathcal{M}$ by means of a binary tree $\mathcal{T}$ (see Fig. 2.10). Each node of $\mathcal{T}$ is associated with a trapezoid τ of the decomposition and the partitioning object (a tangent or a horizontal line) of τ, and stores the representation of such object. Nodes of $\mathcal{T}$ are classified into three categories (and the association): a $\bigcirc$-*node* (a vertical cut), a ∇-*node* (a horizontal cut), and a $\square$-*node* (a terminal trapezoid of the decomposition and its edge fragment).

The above decomposition process is closely related to the one induced by the segment-tree. In particular, the leaves of $\mathcal{T}$ are in one-to-one correspondence with the fragments of the edges of $\mathcal{M}$, so that tree $\mathcal{T}$ has $O(n \log n)$ leaves. Since each node stores a constant amount of information, we have that the space requirement of the trapezoid tree $\mathcal{T}$ is $O(n \log n)$.

It is clear that a point location query LOCATE(q) can be performed by traversing a root-to-leaf path in $\mathcal{T}$, where at each internal node μ we branch left or right depending on the discrimination of the query point q with respect to the partitioning object stored at μ. Indeed, the leaf reached identifies an edge that is first hit by a horizontal ray through q. Since we did not impose any balance requirement on $\mathcal{T}$, the query time could be linear in the worst-case.

To speed-up queries, we implement $\mathcal{T}$ as a dynamic tree [104], i.e., $\mathcal{T}$ is decomposed into solid paths (which should not be confused with the solid paths in the normalization structure), connected by dashed arcs (see Fig. 2.11). Each solid path is associated with a path tree, implemented as a

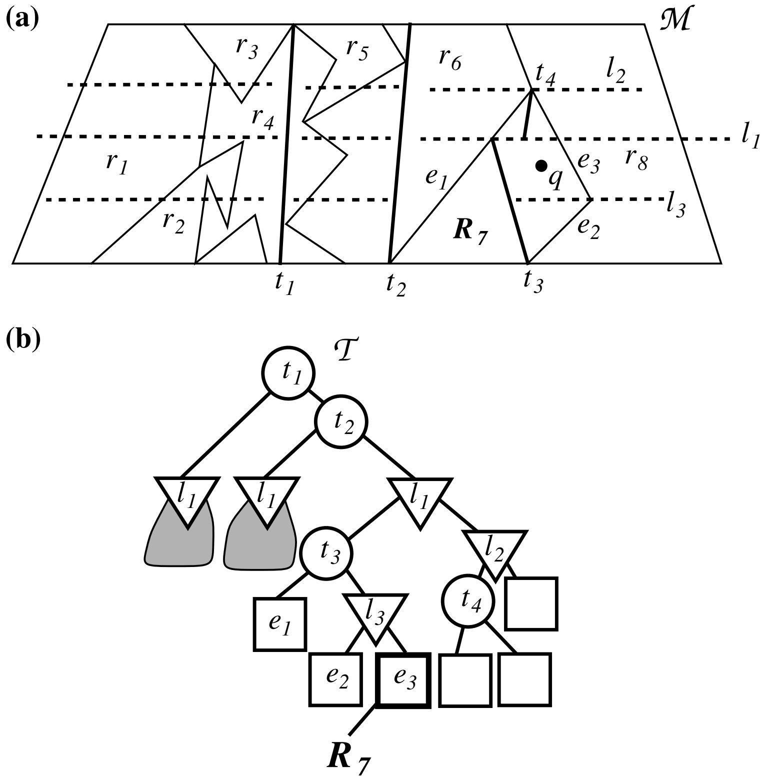


Figure 2.10: Example of the construction of trapezoid tree $\mathcal{T}$ for map $\mathcal{M}$. **(a)** Recursive decomposition of $\mathcal{M}$ by vertical and horizontal cuts. **(b)** Trapezoid tree $\mathcal{T}$ associated with the decomposition in part (a).

biased search tree [10]. Note that the sequence of nodes of a solid path of $\mathcal{T}$ identifies a sequence of nested trapezoids. For example, in path tree $T(P_1)$ of Fig. 2.11(c), leaf t_1 identifies the trapezoid of the entire map $\mathcal{M}$, and leaf t_4 identifies the trapezoid whose right side is at infinity and whose other sides are t_2, l_1 and l_2. A point-location query starts at the root of the path tree of the topmost solid path of $\mathcal{T}$ (e.g., the root of $T(P_1)$ in Fig. 2.11(c)). At a given internal node η of a path tree we consider the rightmost node ζ in the left subtree of η (readily available given thread pointers). We discriminate q against the trapezoid τ of ζ and go to the left or right child of η according to whether q is inside or outside τ (recall that a solid path is stored bottom-to-top in the left-to-right leaves of its path tree). When we reach a leaf of a path tree (which represents a node μ of $\mathcal{T}$), we always exit on a dashed arc, and we always know the exit except for the case of the last node of the solid path, in which case we go to its left or right child by discriminating q against the partitioning object of μ. For example, in Fig. 2.11(c), when we reach leaf l_1 of $T(P_1)$, we know that the next node to visit is the root of $T(P_2)$, since that is the only exit; when we reach leaf l_3 of $T(P_2)$, we discriminate q with l_3 and move down right to $T(P_5)$ by the fact that q is above l_3. By this process, we will finally reach a leaf of a path tree with no exit (representing a leaf of $\mathcal{T}$), which identifies an edge of the region containing q.

Using biased search trees [10] as the standard implementation of path trees, we have

Lemma 2.13 *The time complexity for a point location query is* $O(\log n)$.

Proof: Let $(\nu_1, \mu_1), \cdots, (\nu_\ell, \mu_\ell)$ be the sequence of dashed arcs traversed by the query algorithm, with ν_i the parent of μ_i. (Note that μ_ℓ is the leaf reached by the query algorithm.) Also, let

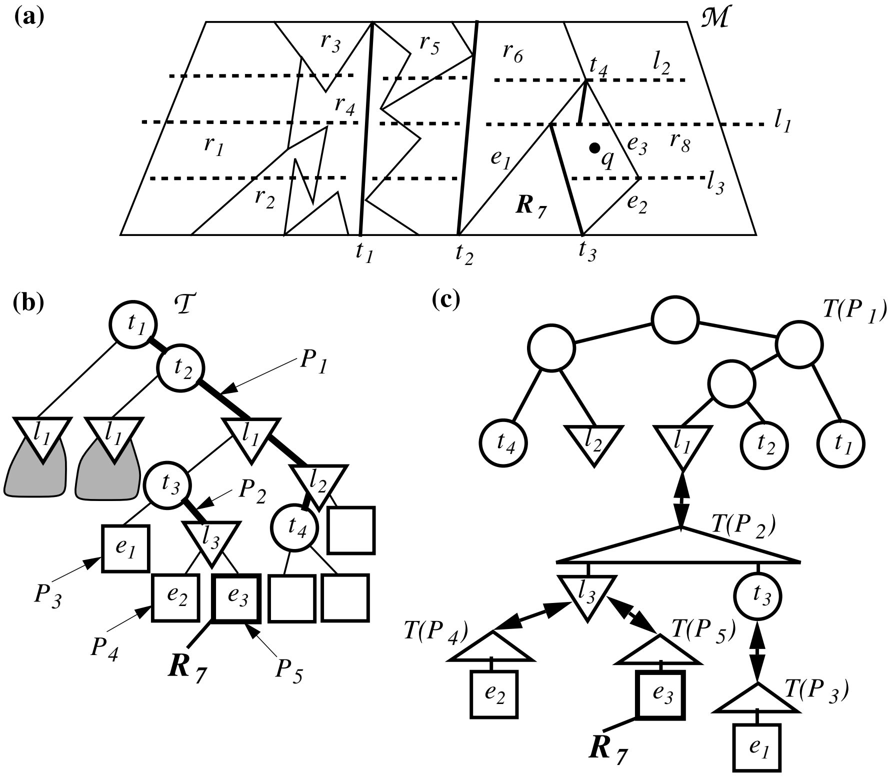


Figure 2.11: Representing trapezoid tree $\mathcal{T}$ by a dynamic tree. **(a)** The same decomposition of $\mathcal{M}$ as in Fig. 2.10(a). **(b)** Decomposing trapezoid tree $\mathcal{T}$ of Fig. 2.10(b) into solid paths $P_1, P_2, \cdots$. **(c)** Actual data structure representing $\mathcal{T}$, where $T(P_i)$ is the path tree for solid path P_i in $\mathcal{T}$. The left-to-right leaves of $T(P_i)$ represent bottom-to-top nodes of P_i, which in turn correspond to smaller-to-bigger nested trapezoids.

μ_0 be the root of $\mathcal{T}$. Since the path trees are implemented as biased search trees, we have that the number of nodes visited in the solid path of ν_i is at most $\log(weight(\mu_{i-1})/weight(\nu_i)) + 2$. Hence, the time complexity of a point location query is $O(\sum_{i=1}^{\ell} \log(weight(\mu_{i-1})/weight(\nu_i)))$. Since $weight(\mu_i) < weight(\nu_i)$, the above sum telescopes, and we have that a point location query takes time $O(\log n)$. □

To perform update operations, we establish bidirectional links between the trapezoid tree and the normalization structure. Let μ be a node of $\mathcal{T}$. We have

- If μ is a ○-node, let Q' be the subpath of a monotone path Q associated with the vertical cut at μ (i.e., the sleeve of Q' spans the trapezoid of μ and has an open hourglass). We establish pointers between μ and the nodes of $ltree(Q)$ and $rtree(Q)$ associated with Q'.
- If μ is a ∇-node, let Q' and R' be subpaths of monotone paths Q and R such that Q' and R' span the leftmost and rightmost regions in the trapezoid of μ. We establish pointers between μ and the nodes of $ltree(R)$ and $rtree(Q)$ associated with R' and Q', respectively. Also, we establish a back pointer from the allocation node y of μ in $\mathcal{Y}$ to μ.
- If μ is a □-node, we establish pointers between μ and the two nodes in the normalization structure associated with the same edge fragment.

Note that every node of a thread tree associated with an open hourglass is pointed to by exactly two nodes of $\mathcal{T}$.

Now, we discuss how update operations affect the trapezoid tree. Since the decomposition described by $\mathcal{T}$ is determined by the monotone paths, we update the trapezoid tree whenever monotone paths are changed in the normalization structure. We only need to consider the effects on the trapezoid tree of elementary splits, joins, partings, and pairings of monotone paths. Each such elementary operation in the normalization structure corresponds to performing $O(1)$ link and cut operations in the trapezoid tree. Details are shown in Figs. 2.12–2.13. Link and cut operations are performed in $O(\log n)$ time by standard dynamic tree algorithms. Regarding vertex insertions, a rotation at a node y in the *Y-tree* $\mathcal{Y}$ caused by a vertex update is handled by rebuilding the subtrees of $\mathcal{T}$ whose roots are ∇-nodes pointed by y. With an argument analogous to the one of Lemma 2.11, we can prove the following lemma.

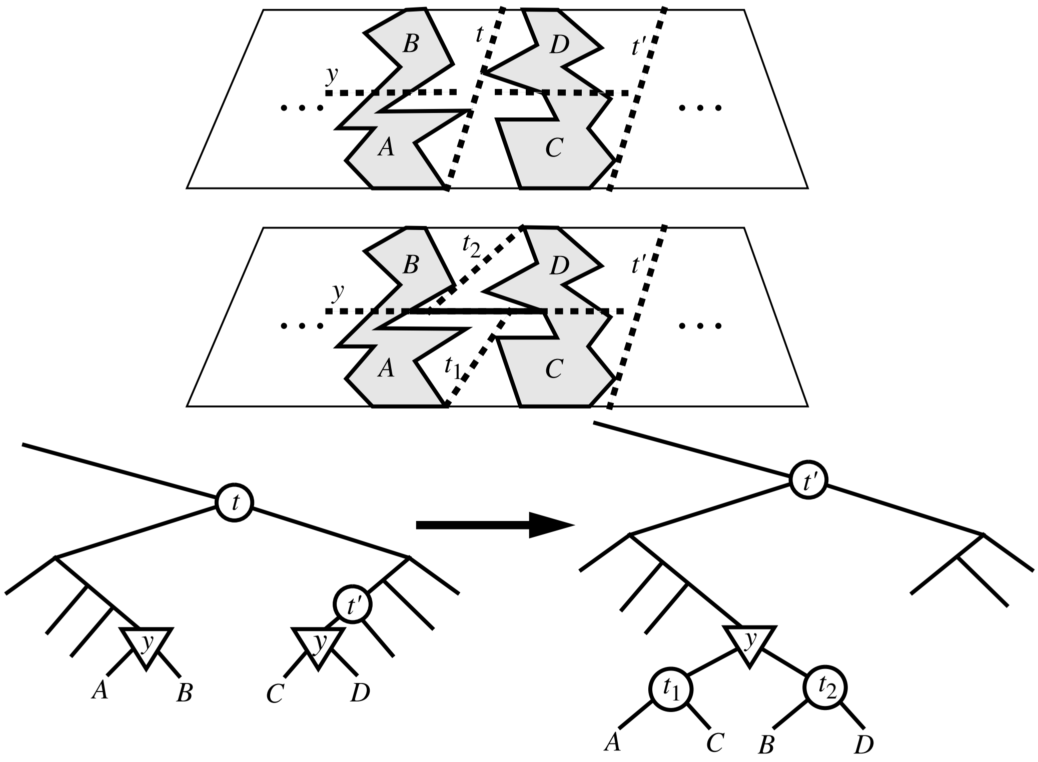


Figure 2.12: Update of the trapezoid tree in consequence of an elementary split of a monotone path in the normalization structure.

Lemma 2.14 *Let y be a node of $\mathcal{Y}$ whose subtree has ℓ leaves. Then the subtrees of $\mathcal{T}$ whose roots are pointed to by y have total size $O(\ell \log \ell)$, and can be built in time $O(\ell \log \ell)$.*

Hence the amortized cost of rebalancing $\mathcal{Y}$ in a sequence of updates is $O(\log^2 n)$. We conclude

Theorem 2.4 *Pont location queries* LOCATE(q) *in an n-vertex connected planar map can be performed in worst-case time $O(\log n)$ using a fully dynamic data structure that uses space $O(n \log n)$ and supports updates of the map in time $O(\log^3 n)$ (amortized for vertex updates).*

Note that query LOCATE(q) is used in the update of the hull structure.

2.6 Ray Shooting Queries

In this section we consider the problem of performing ray-shooting queries of the type:

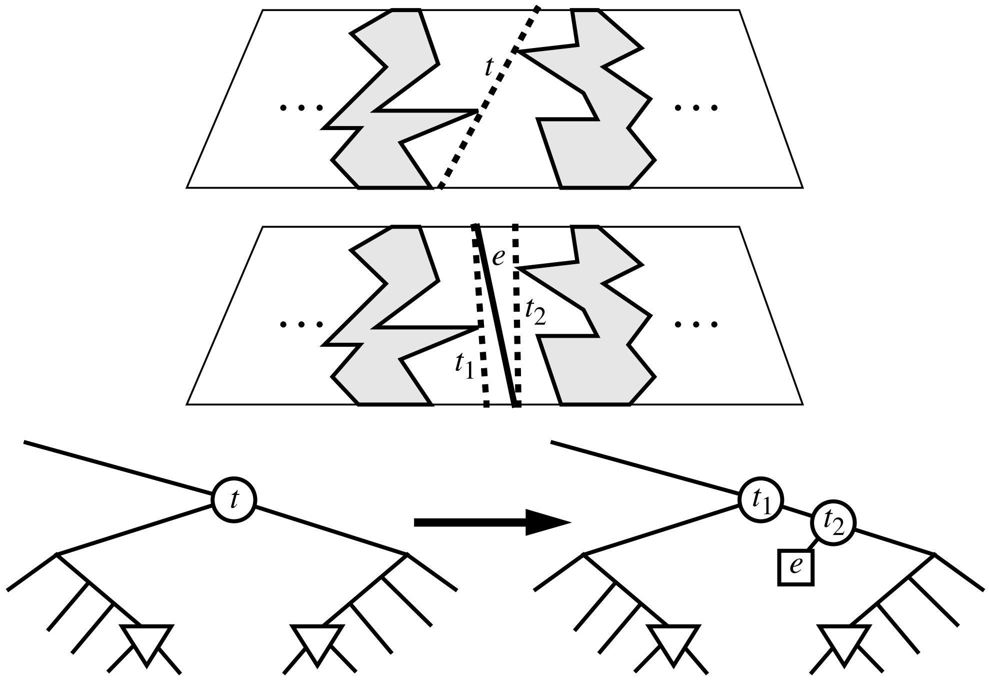


Figure 2.13: Update of the trapezoid tree caused by parting a monotone path in the normalization structure because of an edge insertion.

SHOOT(q, d): Find the first vertex or edge hit by a query ray (q, d) in direction d originating at point q.

We show that the dynamic point location data structure in the previous section also supports ray-shooting queries in worst-case time $O(\log^3 n)$. Without loss of generality, assume that (q, d) is oriented upwards. The ray-shooting algorithm is as follows:

First, we perform LOCATE(q) to determine the region r containing q. If q lies on an a vertex or edge, an infinitesimal perturbation of q in direction d enables us to find the first region r entered by the ray. Query LOCATE(q) also identifies the monotone sleeve SLEEVE(Q) of r containing q and the splitter s_1 of SLEEVE(Q) immediately below q. We find the first intersection q' of (q, d) with the boundary of SLEEVE(Q). If q' is on a vertex or edge of r, then we report q' and stop; else (q' is on a lid of SLEEVE(Q)) we apply the algorithm recursively to the new ray (q', d).

We find the first intersection q' of (q, d) with the boundary of SLEEVE(Q) by the process below:

1. Find the topmost splitter s_2 in SLEEVE(Q) such that HOURGLASS(s_1, s_2) is open, by means of $O(\log n)$ elementary splits and joins of subpaths of Q that yield a new monotone path R such that SLEEVE(s_1, s_2) = SLEEVE(R). Note that the boundary of SLEEVE(R) is part of the boundary of SLEEVE(Q) except for possibly s_1 and s_2, where s_2 is part of the boundary of SLEEVE(Q) if and only if s_2 is the top lid of SLEEVE(Q).
2. Find the first intersection p of (q, d) with the boundary of SLEEVE(R).
3. If p is not on s_2, or if p is on s_2 but s_2 is the top lid of SLEEVE(Q), then p is on the boundary of SLEEVE(Q) and thus the desired intersection q'. Return p and stop.
4. Else (p is on s_2 and s_2 is not the top lid of SLEEVE(Q)), set $s_1 := s_2$, $q := p$, and go to Step 1. Note that this situation can occur at most twice, since s_2 is the topmost splitter above s_1 such that HOURGLASS(s_1, s_2) is open, and any straight line can completely go through at most one

such hourglass, with the bottom and top portions of the line possibly in the two (below and above) adjacent hourglasses (see Fig. 2.14).

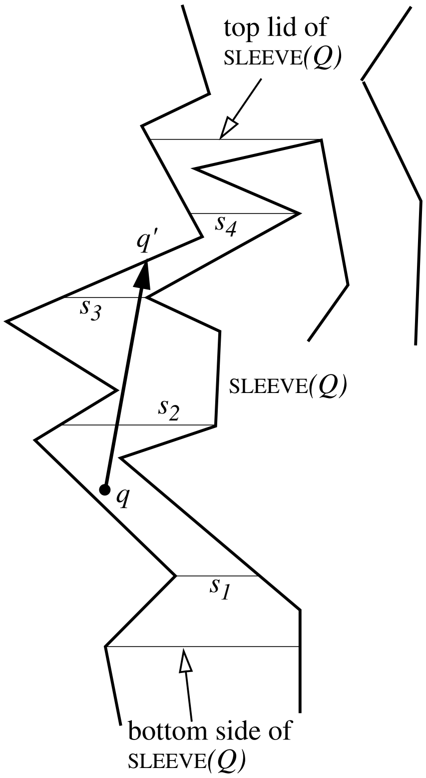


Figure 2.14: The situation in step 4 of the process for computing q' can occur at most twice. For $i = 1, 2, 3$, s_{i+1} is the topmost splitter above s_i such that HOURGLASS(s_i, s_{i+1}) is open. As shown, the situation of step 4 occurrs twice when (q, d) hits s_2 and s_3, respectively. Note that (q, d) can not reach s_4, or otherwise HOURGLASS(s_2, s_4) would be open and s_3 would not be the topmost splitter above s_2 such that HOURGLASS(s_2, s_3) is open.

In step 2, the first intersection p of (q, d) with the boundary of SLEEVE(R) can be found by a binary search in the trees *ltree*(R) and *rtree*(R) as follows: at a current node μ with children μ' and μ'', where CHAIN(μ') is below CHAIN(μ''), we determine the intersection of (q, d) with the convex hull of CHAIN(μ). If the intersection is on a real edge or on a lid then we are done; else it is on a (fictitious) convex hull edge: we then compute the complete convex hulls of CHAIN(μ') and CHAIN(μ''), and repeat the process on μ' or on μ'' depending on whether or not (q, d) intersects with the convex hull of CHAIN(μ'), respectively.

The computation of point q' can be done in $O(\log^2 n)$ time: step 1 performs $O(\log n)$ elementary joins and splits of solid subpaths of Q, each in $O(\log n)$ time by Lemma 2.9; step 2 takes $O(\log^2 n)$ time, with $O(\log n)$ time on each node visited during the binary search; finally, the steps are executed at most three times by step 4. The number of recursive calls to compute a sequence of such points q' is $O(\log n)$ since the query ray intersects $O(\log n)$ lids by Corollary 2.1. At the end, we conceal the path of $\Delta(r)$ traversed by the query ray to restore the weight invariant. We conclude with the following theorem.

Theorem 2.5 *Ray-shooting queries* SHOOT(q, d) *in an n-vertex connected planar map can be performed in worst-case time $O(\log^3 n)$ using a fully dynamic data structure that uses space $O(n \log n)$*

and supports updates of the map in time $O(\log^3 n)$ (amortized for vertex updates).

Theorem 2.5 also provides the capability of checking the validity of an edge insertion, i.e., whether the new edge does not intersect the current edges of the map. Moreover, as a corollary, we can perform stabbing queries, namely, determine the k edges of the map intersected by a query segment, in time $O((k+1)\log^3 n)$.

Chapter 3

Optimal Shortest Path and Minimum-Link Path Queries

3.1 Introduction

In this chapter, we present efficient algorithms for shortest-path and minimum-link-path queries between two convex polygons inside a simple polygon, which acts as an obstacle to be avoided. We give efficient techniques for both the static and dynamic versions of the problem.

Let R_1 and R_2 be two convex polygons with a total of h vertices that lie inside a simple polygon P with n vertices. The (*geodesic*) shortest path $\pi_G(R_1, R_2)$ is the polygonal chain with the shortest length among all polygonal chains joining a point of R_1 and a point of R_2 without crossing edges of P. A minimum-link path $\pi_L(R_1, R_2)$ is a polygonal chain with the minimum number of edges (called *links*) among all polygonal chains joining a point of R_1 and a point of R_2 without crossing edges of P. The number of links in $\pi_L(R_1, R_2)$ is called the *link distance* $d_L(R_1, R_2)$.

The related problem of computing the length of the shortest path between two polygons R_1 and R_2 *without obstacle* P has been extensively studied; this problem is also known as finding the *separation* of the two polygons [45], denoted by $\sigma(R_1, R_2)$. If both R_1 and R_2 are convex their separation can be computed in $O(\log h)$ time [31, 48, 20, 45]; if only one of them is convex an $O(h)$-time algorithm is given in [31]; if neither is convex, an optimal algorithm is recently given by Amato [3], who improves the previous result of Kirkpatrick [71] from $O(h \log h)$ to $O(h)$.

Although there has been a lot of work on the separation problem, the more general shortest-path problem for two objects *in the presence of obstacle* P has been previously studied only for the simple case when the objects are points, for which there exist efficient static [63] and dynamic [26, 59] solutions. The static technique of [63] supports two-point shortest-path queries in optimal $O(\log n)$ time (plus $O(k)$ if the k edges of the path are reported), employing a data structure that uses $O(n)$ space and can be built in linear time. The dynamic technique of [26], as already presented in Chapter 2, performs shortest-path queries between two points in the same region of a connected planar map $\mathcal{M}$ with n vertices in $O(\log^3 n)$ time (plus $O(k)$ to report the k edges of the path), using a data structure with $O(n \log n)$ space that can support updates (insertions and deletions of edges and vertices) of $\mathcal{M}$ each in $O(\log^3 n)$ time. The very recent result of [59] improves the query and update times to $O(\log^2 n)$, with space complexity also improved to $O(n)$.

The minimum-link path problem between two points has been extensively studied. In many applications, such as robotics, motion planning, VLSI and computer vision, the link distance often provides a more natural measure of path complexity than the Euclidean distance [67, 83, 100, 107, 109]. For example, in a robot system, a straight-line navigation is often much cheaper than rotation, thus it is desirable to minimize the number of turns in path planning [100, 109]. Also, in graph drawing, it is often desirable to minimize the number of bends [105, 110].

All previously known techniques for the minimum-link path problem are restricted to the static environment, where updates to the problem instance are not allowed. The method of [107] computes a minimum-link path between two *fixed* points inside a simple polygon in linear time. In [109], a scheme based on *window partition* can answer link distance queries from a *fixed* source in $O(\log n)$ time, after $O(n)$ time preprocessing. The best known results are due to Arkin, Mitchell and Suri [7]. Their data structure uses $O(n^3)$ space and preprocessing time, and supports minimum-link-path queries between two points and between two segments in optimal $O(\log n)$ time (plus $O(k)$ if the k links are reported). Their technique can also perform minimum-link-path queries between two convex polygons, however, in non-optimal $O(\log h \log n)$ time. Also, efficient parallel algorithms are given in [18].

There are other results on the variations of the minimum-link-path problem. Efficient algorithms for link diameter and link center are given in [44, 56, 78, 67, 85, 84, 108]. A minimum-link path between two fixed points in a multiply connected polygon can be computed efficiently [83]. Sequential and parallel algorithms for *rectilinear* link distance are respectively given by de Berg [39] and Lingas *et al.* [79]. De Berg *et al.* [40] study the problem of finding a shortest rectilinear path among rectilinear obstacles. Mitchell *et al.* [82] consider the problem of finding a shortest path with at most K links between two query points inside a simple polygon, where K is an input parameter.

Our main results in this chapter are outlined as follows.

- Let P be a simple polygon with n vertices. There exists an optimal data structure that supports shortest-path queries between two convex polygons with a total of h vertices inside P in time $O(\log h + \log n)$ (plus $O(k)$ if the k links of the path are reported), using $O(n)$ space and preprocessing time; all bounds are worst-case.
- Let P be a simple polygon with n vertices. There exists a data structure that supports minimum-link-path queries between two convex polygons with a total of h vertices inside P in optimal time $O(\log h + \log n)$ (plus $O(k)$ if the k links of the path are reported), using $O(n^3)$ space and preprocessing time; all bounds are worst-case.
- Let $\mathcal{M}$ be a connected planar map whose current number of vertices is n. Shortest-path and minimum-link-path queries between two convex polygons with a total of h vertices that lie in the same region of $\mathcal{M}$ can be performed in times $O(\log h + \log^2 n)$ (plus $O(k)$ to report the k links of the path) and $O(\log h + k \log^2 n)$, respectively, using a fully dynamic data structure that uses $O(n)$ space and supports insertions and deletions of vertices and edges of $\mathcal{M}$ each in $O(\log^2 n)$ time; all bounds are worst-case.

The contributions of this chapter can be summarized as follows:

- We provide the first optimal data structure for shortest-path queries between two convex polygons inside a simple polygon P that acts as an obstacle. No efficient data structure was known before to support such queries. All previous techniques either consider the case where P is not present or the case where the query objects are points.
- We provide the first data structure for minimum-link-path queries between two convex polygons inside a simple polygon P in optimal $O(\log h + \log n)$ time. The previous best result [7] has query time $O(\log h \log n)$ (and the same space and preprocessing time as ours).
- We provide the first fully dynamic data structure for shortest-path queries between two convex polygons in the same region of a connected planar map $\mathcal{M}$. No such data structure was known before even for the static version.
- We provide the first fully dynamic data structure for minimum-link-path queries between two

convex polygons in the same region of a connected planar map $\mathcal{M}$. No such data structure was known before even for two-point queries.

We summarize the comparisions of our results with the previous ones in Tables 3.1–3.4.

Static Shortest Paths	Query Type	Query	Space	Preprocessing
Guibas-Hershberger [63]	two query points	$\log n$ *	n *	n *
This chapter	two query convex polygons	$\log h + \log n$ *	n *	n *

* optimal

Table 3.1 Results for static shortest-path queries.

Dynamic Shortest Paths	Query Type	Query	Space	Update
Chiang-Preparata-Tamassia [26] (Chapter 2)	two query points	$\log^3 n$	$n \log n$	$\log^3 n$
Goodrich-Tamassia [59]	two query points	$\log^2 n$	n *	$\log^2 n$
This chapter	two query convex polygons	$\log h + \log^2 n$	n *	$\log^2 n$

* optimal

Table 3.2 Results for dynamic shortest-path queries.

Static Min-Link Paths	Query Type	Query	Space	Preprocessing
Suri [109]	one fixed point and one query point	$\log n$ *	n *	n *
Arkin-Mitchell-Suri [7]	two query points/segments	$\log n$ *	n^3	n^3
	two query convex polygons	$\log h \log n$	n^3	n^3
This chapter	two query convex polygons	$\log h + \log n$ *	n^3	n^3

* optimal

Table 3.3 Results for static minimum-link-path queries.

We briefly outline our techniques. Given the available static techniques with optimal query time for shortest paths and minimum-link paths between *two points*, our main task in performing the *two-polygon* queries is to find two points $p \in R_1$ and $q \in R_2$ such that their shortest path or minimum-link path gives the desired path between R_1 and R_2. As we shall see later, the notion of *geodesic hourglass* between R_1 and R_2 is central to our method. The geodesic hourglass is *open* if R_1 and R_2 are mutually visible, and *closed* otherwise. As for shortest-path queries, the case where R_1 and R_2 are mutually visible is a basic case that, surprisingly, turns out to be nontrivial (the complication comes from the fact that the shortest path in this case may still consist of more than one link), and our solution makes use of interesting geometric properties. If R_1 and R_2 are not visible, then the geodesic hourglass gives two points p_1 and p_2 that are respectively visible from R_1 and R_2 such that the shortest path between any point of R_1 and any point of R_2 must go through p_1 and p_2. Then the shortest path between R_1 and R_2 is the union of the shortest paths between R_1 and p_1, between p_2 and R_2 (both are basic cases), and between two points p_1 and p_2. The geodesic hourglass also gives useful information for minimum-link-path queries. When it is

Dynamic Min-Link Paths	Query Type	Query	Space	Update
This chapter	two query convex polygons	$\log h + k \log^2 n$	n *	$\log^2 n$

* optimal

Table 3.4 Results for dynamic minimum-link-path queries.

open, a minimum-link path is just a single segment; if it is closed, then it gives two edges such that extending them to intersect R_1 and R_2 gives the desired points p and q whose minimum-link path is a minimum-link path $\pi_L(R_1, R_2)$. However, it seems difficult to compute the geodesic hourglass in optimal time. Interestingly, we can get around this difficulty by computing a *pseudo hourglass* that gives all the information we need about the geodesic hourglass. We also extend these results to the dynamic case, by giving the first dynamic method for minimum-link-path queries between two points.

The rest of this chapter is organized as follows. In Section 3.2 we briefly review the basic geometric notions used by our method. Section 3.3 shows how to perform shortest-path queries in the static environment, in particular how to compute the pseudo hourglass and how to handle the nontrivial basic case where two query polygons are mutually visible. Sections 3.4, 3.5 and 3.6 are devoted to dynamic shortest-path, static minimum-link-path, and dynamic minimum-link-path queries, respectively.

3.2 Preliminaries

In this section we describe geometric notions needed in this chapter, some of which have already been introduced in Section 2.2 and are further explored here. A *connected planar map* $\mathcal{M}$ is a subdivision of the plane into polygonal regions whose underlying planar graph is connected. Thus each region of $\mathcal{M}$ is a simple polygon P. A polygonal chain γ is *monotone* if any horizontal line intersects it in a single point or in a single interval or not at all. A simple polygon P is *monotone* if its boundary consists of two monotone chains. A *cusp* of a polygon P is a vertex v whose interior angle is greater than π and whose adjacent vertices are both strictly above (lower cusp) or strictly below (upper cusp) v. If we draw from a cusp v of P two horizontal rays that terminate when they first meet the edges of P, the resulting segments to the left and right of v are called *left lid* and *right lid* of v, respectively. A polygon is monotone if and only if it has no cusps.

The notion of *window partition* was introduced in [109]. Given a point or a line segment s in region P, let $WP(s)$ denote the partition of P into maximally-connected subregions with the same link distance from s; $WP(s)$ is called the *window partition* of P with respect to s. Associated with $WP(s)$ is a set of *windows*, which are chords of P that serve as boundaries between adjacent subregions of the partition.

Given two points p and q that lie in the same region P of $\mathcal{M}$ (or in the same simple polygon P), it is well known that their shortest path $\pi_G(p, q)$ is unique and only turns at the vertices of P. On the contrary, a minimum-link path is not unique and may turn at any point inside P. Adopting the terminology of [109], we define the (unique) *greedy minimum-link path* $\pi_L(p, q)$ to be the minimum-link path whose first and last links are respectively the extensions of the first and last links of $\pi_G(p, q)$, and whose other links are the extensions of the windows of $WP(p)$. The number of links in $\pi_L(p, q)$ is then the link distance $d_L(p, q)$. In the following we use the term "window" to refer to both a window and its extension.

Given a shortest path $\pi_G(p, q)$, an edge $e \in \pi_G(p, q)$ is an *inflection edge* if its predecessor and its successor lie on opposite sides of e. It is easily seen that an edge $e \in \pi_G(p, q)$ is an inflection edge if and only if it is an internal common tangent of the boundaries of P.

Given two convex polygons R_1 and R_2 inside P, we say that R_1 and R_2 are *mutually visible* if there exists a line l connecting R_1 and R_2 without crossing any edge of P; we call such line l a *visibility link* between R_1 and R_2. Now we define the *left and right boundaries* B_L and B_R of P with respect to R_1 and R_2 when they are not mutually visible through a horizontal line. For $i = 1, 2$, let u_i and d_i be the highest and lowest vertices of R_i, respectively. Without loss of generality, we assume that $y(u_1) \geq y(u_2)$ (otherwise we exchange the roles of R_1 and R_2). We choose $q_1 \in \{u_1, d_1\}$ and $q_2 \in \{u_2, d_2\}$ such that (i) the subpolygon P' of P delimited by both e_1 and e_2 contains both R_1 and R_2, where e_i is a horizontal chord of P going through q_i, $i = 1, 2$, and (ii) among the four shortest paths $\pi_G(u_1, u_2)$, $\pi_G(u_1, d_2)$, $\pi_G(d_1, u_2)$ and $\pi_G(d_1, d_2)$, $\pi_G(q_1, q_2)$ has the largest number of cusps (see Fig. 3.1). Now P' is bounded by e_1, e_2 and two polygonal chains. We define B_L and B_R as these two polygonal chains of P': B_L is the one to the left of $\pi_G(q_1, q_2)$ when we walk along $\pi_G(q_1, q_2)$ from q_2 to q_1, and B_R is the one to the right (see Fig. 3.1). Clearly, any shortest path π between a point in R_1 and a point in R_2 can only touch the vertices of P on B_L and B_R, and the inflection edges of π are those edges that have one endpoint on B_L and the other endpoint on B_R.

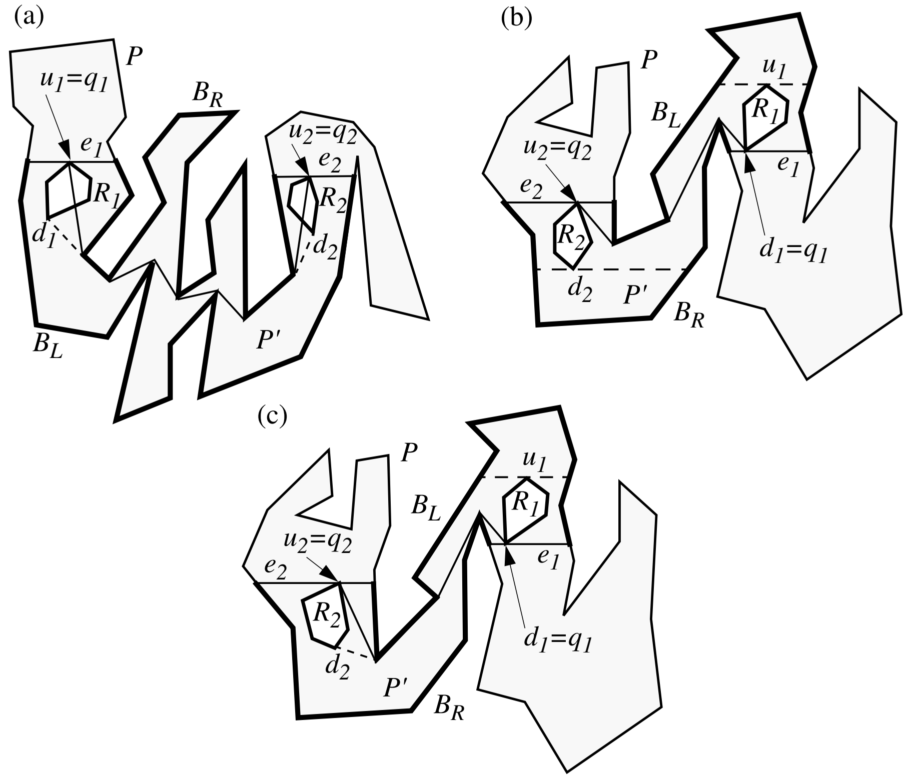


Figure 3.1: Left and right boundaries B_L and B_R of P: (a) several choices of (q_1, q_2) satisfy condition (ii) but only one satisfies (i); (b) several choices of (q_1, q_2) satisfy condition (i) (e.g., (u_1, d_2) is also valid) but only one satisfies (ii); (c) neither (i) nor (ii) alone enforces a unique choice of (q_1, q_2), but their conjunction does.

3.3 Static Shortest Path Queries

In this section we show how to compute the shortest path $\pi_G(R_1, R_2)$ between two convex polygons R_1 and R_2 with a total of h vertices inside an n-vertex simple polygon P. The data structure of

Guibas and Hershberger [63] computes the shortest path $\pi_G(p,q)$ between any two points p and q inside P in $O(\log n)$ time, where in $O(\log n)$ time we get an implicit representation (a balanced binary tree) and the length of $\pi_G(p,q)$, and using additional $O(k)$ time to retrieve the k links we get the actual path. Point-location queries can also be performed in $O(\log n)$ time. The data structure uses $O(n)$ space and can be built in $O(n)$ time after triangulating P (again in $O(n)$ time by Chazelle's linear-time triangulation algorithm [19]). We modify this data structure so that associated with the implicit representation of a shortest path π_G, there are two balanced binary trees respectively maintaining the inflection edges and the cusps on π_G in their path order. The balanced binary tree representing π_G and the two associated binary trees support split and splice operations, so that we can extract a portion of π_G in logarithmic time.

With this data structure, our task is to find points $p \in R_1$ and $q \in R_2$ such that $\pi_G(p,q) = \pi_G(R_1, R_2)$. We say that p and q *realize* $\pi_G(R_1,R_2)$. Note that p and q lie on the boundaries of R_1 and R_2 but are not necessarily vertices.

To obtain a better intuition, let us imagine surrounding R_1 and R_2 with a rubber band inside P. The resulting shape is called the *relative convex hull* of R_1 and R_2. It is formed by four pieces: shortest paths $\pi_1 = \pi_G(a_1,a_2)$, $\pi_2 = \pi_G(b_1,b_2)$ ($a_1, b_1 \in R_1$ and $a_2, b_2 \in R_2$), and the boundaries of R_1 and R_2 farther away from each other. We call a_1, b_1, a_2, and b_2 the *geodesic tangent points*, and π_1 and π_2 the *geodesic external tangents* of R_1 and R_2. Note that if π_1 consists of more than one link, then the first (resp. last) link of π_1 is a common tangent between R_1 (resp. R_2) and the convex hull inside P of a portion of the boundary of P (see Fig. 3.2), and similarly for π_2. Let $s_1 = (a_1, b_1)$ and $s_2 = (a_2, b_2)$. If we replace R_1 and R_2 with s_1 and s_2, then the relative convex hull of s_1 and s_2 is the *hourglass* $H(s_1,s_2)$ bounded by s_1, s_2, π_1, and π_2. Note that π_1 and π_2 stay unchanged. We call $H(s_1,s_2)$ the *geodesic hourglass* between R_1 and R_2. We say that $H(s_1,s_2)$ is *open* if π_1 and π_2 do not intersect, and *closed* otherwise. When $H(s_1,s_2)$ is closed, there is a vertex p_1 at which π_1 and π_2 join together, and a vertex p_2 at which the two paths separate (possibly $p_1 = p_2$); we call p_1 and p_2 the *apices* of $H(s_1,s_2)$ (see Fig. 3.2(b)). Also, we say that $\pi_G(a_1,p_1)$ and $\pi_G(b_1,p_1)$ form a *funnel* $F(s_1)$. The only internal common tangent ρ_1 of P among all edges of $F(s_1)$ is called the *penetration* of $F(s_1)$, and similarly for ρ_2 in funnel $F(s_2)$ (see Fig. 3.2(b)). Hereafter we use H_G to denote the geodesic hourglass, and a_1, b_1 ($\in R_1$), a_2, b_2 ($\in R_2$) to denote the geodesic tangent points.

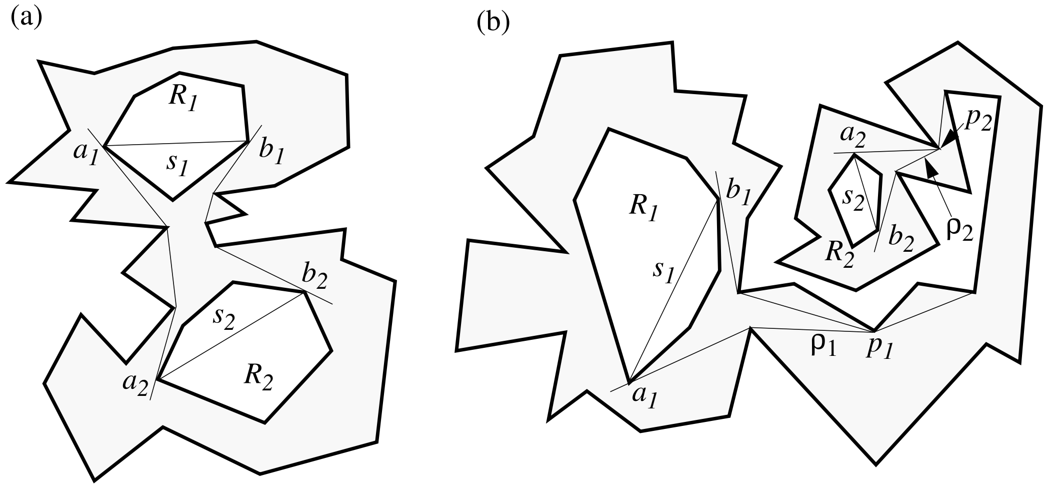


Figure 3.2: Geodesic hourglass H_G and geodesic external tangents: (a) H_G is open; (b) H_G is closed.

Observe that H_G is open if and only if R_1 and R_2 are mutually visible (see Fig. 3.2(a)). If H_G is closed, then $\pi_G(p', q')$ between any point $p' \in R_1$ and any point $q' \in R_2$ must go through p_1 and p_2 (see Fig. 3.2(b)). Thus $\pi_G(R_1, R_2)$ must go through p_1 and p_2, i.e., $\pi_G(R_1, R_2) = \pi_G(R_1, p_1) \cup \pi_G(p_1, p_2) \cup \pi_G(p_2, R_2)$. Since R_1 and p_1 are mutually visible, the algorithm for computing $\pi_G(R_1, R_2)$ when R_1 and R_2 are mutually visible can be used to compute $\pi_G(R_1, p_1)$ as well, and similarly for $\pi_G(p_2, R_2)$. In summary, we need to handle the following two main tasks: (i) deciding whether H_G is open or closed, and finding apices p_1 and p_2 when H_G is closed, and (ii) computing $\pi_G(R_1, R_2)$ when R_1 and R_2 are mutually visible.

3.3.1 The Pseudo Geodesic Hourglass

We first discuss how to compute the information about geodesic hourglass H_G in optimal $O(\log h + \log n)$ time. A straightforward method is to compute H_G directly. As shown in [7], we can compute the geodesic external tangents between R_1 and R_2 (and hence H_G) by a binary search mimicking the algorithm [90] for finding ordinary common tangents, where in each iteration we compute the shortest path between two chosen points rather than the segment joining them. However, this results in a computation of $O(\log h \log n)$ time. Also, it seems difficult to compute H_G in optimal time.

To overcome the difficulty, we notice that it is not necessary to compute H_G exactly. As for shortest-path queries, we only need to know whether H_G is open or closed, and the apices p_1 and p_2 of H_G when it is closed; as for minimum-link path queries (see Section 3.5), we only need to know a visibility link between R_1 and R_2 when H_G is open, and the penetrations ρ_1 and ρ_2 of H_G when it is closed. Interestingly, we can obtain the above information by computing a *pseudo hourglass* H'' with the property that if H'' is open then H_G is open, and if H'' is closed then H_G is closed with the same penetrations and apices. We first describe the algorithm and then justify its correctness.

Algorithm Pseudo-Hourglass

1. Ignore P and compute the ordinary external common tangents (a'_1, a'_2) and (b'_1, b'_2) between R_1 and R_2, using the algorithm of Overmars and van Leeuwen [90], where $a'_1, b'_1 \in R_1$ and $a'_2, b'_2 \in R_2$. Let $s'_1 = (a'_1, b'_1)$ and $s'_2 = (a'_2, b'_2)$. Compute shortest paths $\pi_1 = \pi_G(a'_1, a'_2)$ and $\pi_2 = \pi_G(b'_1, b'_2)$. If they are disjoint (i.e., neither has an inflection edge) then the hourglass $H' = H(s'_1, s'_2)$ is open. In this case s'_1 and s'_2 are mutually visible, implying that R_1 and R_2 are mutually visible. Use algorithm [90] to compute an internal common tangent l between π_1 and π_2, report {open with visibility link l} and stop.
2. Else (π_1 and π_2 are not disjoint) H' is closed. Now the geodesic external tangents (which constitute H_G) must go through vertices of P, and it is still possible that H_G is open. Let u_1 and d_1 be the highest and lowest vertices of R_1, respectively, and similarly for u_2 and d_2 in R_2. Assume that $y(u_1) \geq y(u_2)$ (otherwise exchange the roles of R_1 and R_2). Compute shortest paths $\pi_G(u_1, u_2)$, $\pi_G(u_1, d_2)$, $\pi_G(d_1, u_2)$ and $\pi_G(d_1, d_2)$. Take π as the one with the largest number of cusps (break ties arbitrarily). Consider π as *oriented from R_2 to R_1*.
3. From $R_i, i = 1, 2$, compute horizontal projection points l_i and r_i respectively on the left and right boundaries B_L and B_R of P, by discriminating the following cases.
   (a) π has no cusp at all.
   There are two subcases.
      i. $y(d_1) \leq y(u_2)$, i.e., there is a vertical overlap between horizontal projections of R_1 and R_2.
      In this case the line $l : y = y(u_2)$ connects R_1 and R_2 without being blocked (to be

proved in Lemma 3.1). Report {open with visibility link l} and stop.

ii. There is no vertical overlap (see Fig. 3.3).
Project u_1 horizontally to the left and right on the boundaries B_L and B_R of P to get points l_1 and r_1, respectively (via point location), and similarly project d_2 to the left and right to get l_2 and r_2.

(b) π has cusps.
Consider R_1 (and symmetrically for R_2). Look at cusp c_1 of π closest to R_1, and denote π' the portion of π from c_1 to the point on R_1. Without loss of generality, assume that c_1 is a lower cusp. There are two cases.

i. c_1 is lower than or as low as d_1 ($y(c_1) \le y(d_1)$).
This means that R_1 is entirely blocked by c_1. Project u_1 horizontally to the left and right to get l_1 and r_1, respectively.

ii. c_1 is higher than d_1 ($y(c_1) > y(d_1)$).
Then R_1 "stretches" beyond c_1. Consider the following subcases.

A. The first link of π' (oriented toward R_1) goes toward left (see Fig. 3.5(a)(b)).
Project both u_1 and d_1 to the right to get r_1 and l_1, respectively. Also a special-case checking is needed: if segment (d_1, l_1) intersects R_2 at v, then report {open with visibility link $l = (d_1, v)$} and stop.

B. The first link of π' goes toward right.
Project both u_1 and d_1 to the left to get l_1 and r_1, respectively. Again perform a special-case checking: if segment (d_1, r_1) intersects R_2 at v, then report {open with visibility link $l = (d_1, v)$} and stop.

4. Compute shortest paths $\pi_l = \pi_G(l_1, l_2)$ and $\pi_r = \pi_G(r_1, r_2)$. Extract the "left bounding convex chain" C_{L1} for R_1 as the portion of π_l from l_1 to x, where x is the first vertex v_1 on B_R or the first point c with $y(c) = y(l_1)$ or the second cusp c_2, whichever is closest to R_1, or $x = l_2$ if none of v_1, c and c_2 exists. Note that C_{L1} includes the first inflection edge if $x = v_1$. Similarly extract the "right bounding convex chain" C_{R1} of R_1 from π_r. The left and right bounding convex chains C_{L2} and C_{R2} of R_2 are computed analogously (see Fig. 3.4).

5. Compute *pseudo tangent points* $a_1'', b_1'' \in R_1$ and $a_2'', b_2'' \in R_2$ such that the *pseudo hourglass* H'' formed by $\pi_G(a_1'', a_2''), \pi_G(b_1'', b_2''), s_1'' = (a_1'', b_1'')$ and $s_2'' = (a_2'', b_2'')$ has the desired property. Point a_1'' is computed from R_1 and C_{L1} by the following steps (and analogously b_1'', a_2'' and b_2'' are computed from R_1 and C_{R1}, from R_2 and C_{L2}, and from R_2 and C_{R2}, respectively).

(a) Check whether R_1 intersects C_{L1} (viewing $C_{L1} = \pi_G(l_1, x)$ as a convex polygon with edge (l_1, x) added) using the algorithm [20], which runs in logarithmic time and also reports a common point g inside both R_1 and C_{L1} if they intersect. If $R_1 \cap C_{L1} = \emptyset$, then find the internal common tangent $t = (v, w)$ between R_1 and C_{L1}, $v \in R_1, w \in C_{L1}$, such that R_1 lies on the right side of t if t is directed from w to v (see Fig. 3.3). Note that only one of the two internal common tangents between R_1 and C_{L1} satisfies the criterion for t. Now check whether t intersects C_{R1} via a binary search on C_{R1}.

i. $t \cap C_{R1} = \emptyset$. Set $a_1'' := v$.

ii. $t \cap C_{R1} = \{y_1, y_2\}$. Let C'_{R1} be the portion of C_{R1} between points y_1 and y_2. Find the external common tangent $t' = (v', w')$ between R_1 and C'_{R1}, $v' \in R_1, w' \in C'_{R1}$, such that both R_1 and C'_{R1} lie on the right side of t' if t' is directed from w' to v'. Set $a_1'' := v'$. (See Fig. 3.3.)

(b) Else ($R_1 \cap C_{L1} \neq \emptyset$, with a common point g inside both R_1 and C_{L1}), then there is only one edge of C_{L1} intersecting R_1 (to be proved in Lemma 3.3). Compute this edge

(u, b) by applying Lemma 3.3. Suppose b is closer to R_2 than u; call b the *blocking point.* Consider the following two cases.

i. The blocking point b is on the left boundary B_L.
Compute a_1'' as the tangent point from b to R_1 such that R_1 is on the right side of (b, a_1'') when (b, a_1'') is directed toward a_1''. (See Fig. 3.7(a)(b).)

ii. The blocking point b is on the right boundary B_R.
Take C as the convex portion of π_l (oriented from R_1 to R_2) from b to z, where z is the first vertex v_1' on B_L again or the first point c' with $y(c') = y(b)$ or the second cusp c_2' after b, whichever is closest to R_1. Note that such v_1' always exists since $\pi_l = \pi_G(l_1, l_2)$ finally goes to $l_2 \in B_L$, and that C includes the first inflection edge after b if $z = v_1'$. Find the external common tangent $t'' = (v'', w'')$ between R_1 and C, $v'' \in R_1, w'' \in C$, such that both R_1 and C lie on the right side of t'' if t'' is directed from w'' to v''. Set $a_1'' := v''$. (See Fig. 3.7(c)–(f).)

6. Compute shortest paths $\pi_1 = \pi_G(a_1'', a_2'')$ and $\pi_2 = \pi_G(b_1'', b_2'')$ to form pseudo hourglass H''. Check whether H'' is open or closed.
   (a) H'' is open (neither π_1 nor π_2 has an inflection edge).
   Compute an internal common tangent l between π_1 and π_2, report {open with visibility link l} and stop.
   (b) H'' is closed.
   Penetration $\rho_1 = (w_1, p_1)$ is chosen from the first inflection edges of π_1 and of π_2 (one of such edges might be missing) as the one that is closer to R_1, and the endpoint p_1 of ρ_1 farther away from R_1 is an apex. The other penetration ρ_2 and apex p_2 are found similarly. Recall that an inflection edge has one endpoint on B_L and the other on B_R. To decide whether the first and last links of π_1 and π_2 are inflection edges, points a_1'' and a_2'' are viewed as on B_L, and b_1'' and b_2'' as on B_R. After computing ρ_1, ρ_2, p_1 and p_2, report {closed with penetrations ρ_1 and ρ_2 and apices p_1 and p_2}, and stop.

The correctness of the algorithm is justified by the following lemmas.

Lemma 3.1 *In step 3(a)i of Algorithm* Pseudo-Hourglass, *the line $l : y = y(u_2)$ connects R_1 and R_2 without being blocked.*

Proof: Recall that there is a vertical overlap between the horizontal projections of R_1 and R_2, i.e., $y(u_1) \geq y(u_2) \geq y(d_1)$. By the definition of π and the fact that π has no cusp, the shortest path between u_1 and u_2 must have no cusp. Thus any lower cusp c' of P in between R_1 and R_2 has $y(c') \geq y(u_2)$. Similarly, any upper cusp c'' of P in between R_1 and R_2 has $y(c'') \leq \max\{y(d_1), y(d_2)\}$. Note that $y(u_2) \geq \max\{y(d_1), y(d_2)\}$, therefore $y(c'') \leq y(u_2) \leq y(c')$, i.e., the line $l : y = y(u_2)$ connects R_1 and R_2 without being blocked. □

Lemma 3.2 *The projection points l_i and $r_i, i = 1, 2$ obtained in step 3 of Algorithm* Pseudo-Hourglass *lie on distinct boundaries of P, i.e., $l_i \in B_L$ and $r_i \in B_R$.*

Proof: The claim is obvious for steps 3(a)ii and 3(b)i since l_i and r_i are obtained by projecting the same point to the left and right. Now consider step 3(b)iiA (step 3(b)iiB is similar). It is clear that r_1 is on B_R, so we look at l_1. If all points on π are higher than d_1, then the horizontal line $y = y(d_1)$ is not blocked by π and thus is to the left of π (recall that π is oriented from R_2 to R_1). So l_1 is on B_L (see Fig. 3.5(a)). On the other hand, if π contains some point c' lower than d_1, then for c_1 to be a lower cusp, there must be an upper cusp on π between c_1 and c' that is higher than c_1 (see Fig. 3.5(b)). Let c'' be such upper cusp closest to c_1, then the line $y = y(d_1)$ is blocked by c'' and the projection point l_1 is on B_L. □

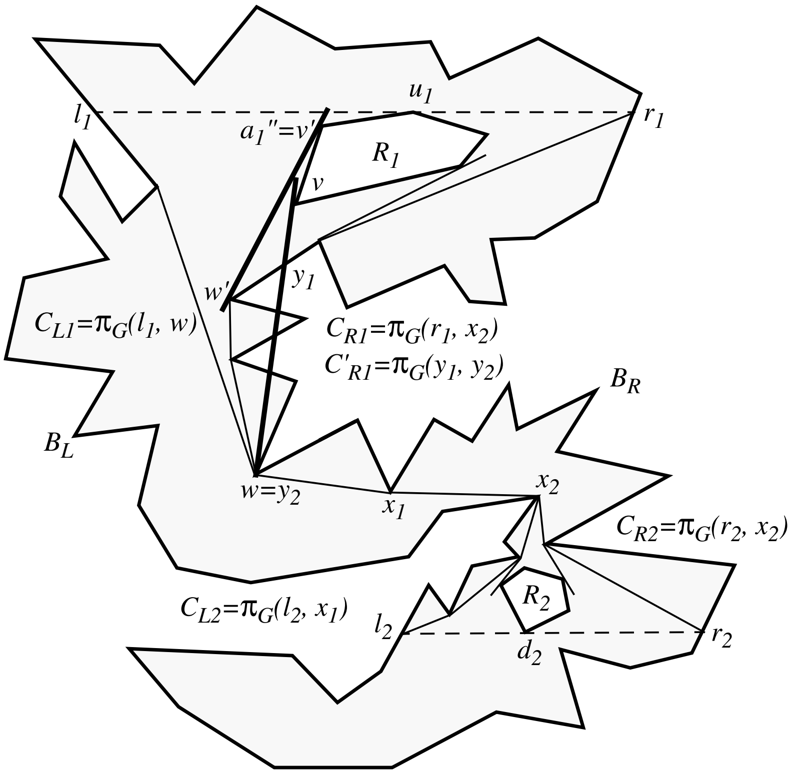


Figure 3.3: A running example for Algorithm *Pseudo-Hourglass* in the case where π has no cusps and $C_{L1} \cap R_1 = \emptyset$.

Lemma 3.3 *In step 5b of Algorithm* Pseudo-Hourglass, *where $R_1 \cap C_{L1} \neq \emptyset$ with a common point g inside both R_1 and C_{L1}, there is only one edge of C_{L1} intersecting R_1. Furthermore, this edge (u, b) can be computed in $O(\log n)$ time.*

Proof: We prove the first part by contradiction. If there were more than one edge of C_{L1} intersecting R_1, say (v_1, v_2) and (v_2, v_3) (see Fig. 3.6(a)), then v_2 would be inside R_1 and would also be a vertex of P, contradicting the fact that R_1 is in a free space of P.

Now we show how to compute (u, b) in $O(\log n)$ time. Assume that l_1 is obtained in step 3 of Algorithm *Pseudo-Hourglass* by projecting u_1. Then u_1 is inside R_1 but outside C_{L1}, thus segment $(g, u_1) \in R_1$ intersects the boundary of C_{L1} (see Fig. 3.6(b)). By the first part of this lemma, there is only one edge (u, b) of C_{L1} that can be intersected by a segment inside R_1. Performing a binary search on C_{L1} to identify the edge intersected by (g, u_1), (u, b) can be found in $O(\log n)$ time. $\square$

Lemma 3.4 *The pseudo hourglass H'' computed from steps 5 and 6 of Algorithm* Pseudo-Hourglass *has the property that if H'' is open then the geodesic hourglass H_G is open, and if H'' is closed with penetrations ρ_1 and ρ_2 and apices p_1 and p_2 then H_G is closed with the same penetrations and apices.*

Proof: Recall that $a_1, b_1 \in R_1$ and $a_2, b_2 \in R_2$ are the geodesic tangent points. We first consider the case in which the bounding convex chains C_{L1} and C_{R1} do no intersect R_1, and C_{L2} and C_{R2} do not intersect R_2 either (see Fig. 3.4). Define S_i, $i = 1, 2$, as follows. If l_i and r_i are obtained by projecting the same point of R_i then $S_i = (l_i, r_i)$; otherwise assuming without loss of generality that l_i is obtained from projecting d_i and r_i from u_i, then $S_i = (u_i, r_i) \cup (u_i, d_i) \cup (d_i, l_i)$. We observe that the area bounded by $S_1, S_2, \pi_l = \pi_G(l_1, l_2)$ and $\pi_r = \pi_G(r_1, r_2)$ properly contains H_G, therefore

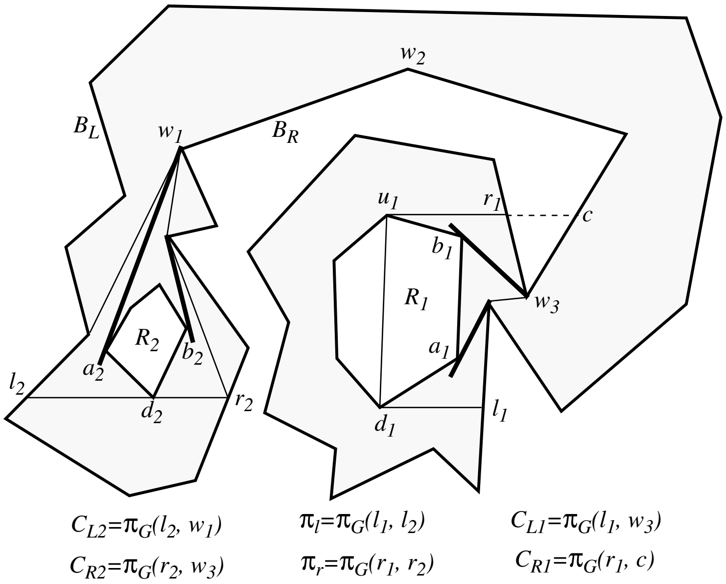


Figure 3.4: Step 4 of Algorithm *Pseudo-Hourglass* and proof of Lemma 3.4. As for step 4, notice how we get the bounding convex chains C_{L1}, C_{R1}, C_{L2} and C_{R2}, especially C_{R1} and C_{R2}; as for Lemma 3.4, note that R_1 and R_2 do not intersect any of the bounding convex chains, $S_1 = (u_1, r_1) \cup (u_1, d_1) \cup (d_1, l_1)$, $S_2 = (l_2, r_2)$, and $H_G = (a_1, b_1) \cup (a_2, b_2) \cup \pi_G(a_1, a_2) \cup \pi_G(b_1, b_2)$ is properly contained in $S_1 \cup S_2 \cup \pi_l \cup \pi_r$.

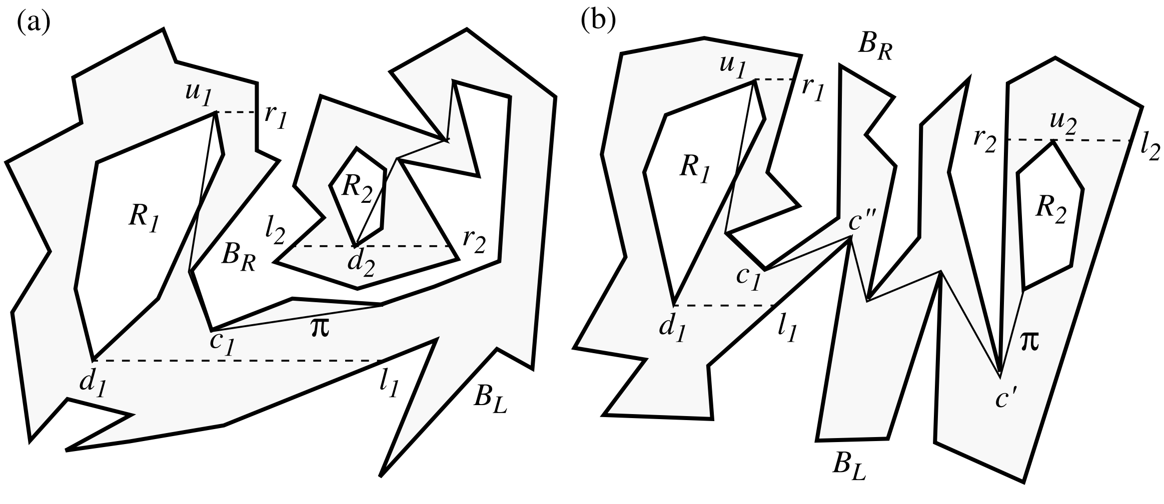


Figure 3.5: Step 3(b)iiA of Algorithm *Pseudo-Hourglass* and proof of Lemma 3.2: r_1 and l_1 are obtained by projecting u_1 and d_1 horizontally to the right; r_1 is on B_R and l_1 is on B_L.

a_1 and b_1 are computed from the common tangents between R_1 and C_{L1}/C_{R1}, and similarly for a_2 and b_2 (see Fig. 3.4, and also Fig. 3.3 for one more example). These are exactly what we compute in steps 5a–5(a)ii, i.e., $H'' = H_G$, and the lemma follows.

Next, look at the case where at least one of the bounding convex chains intersects R_1 or R_2. Since a_1'', a_2'', b_1'' and b_2'' are computed independently, we consider only a_1''; the same argument applies for the others. As we have already seen, $a_1'' = a_1$ when C_{L1} does not intersect R_1, so we consider a_1'' when C_{L1} intersects R_1.

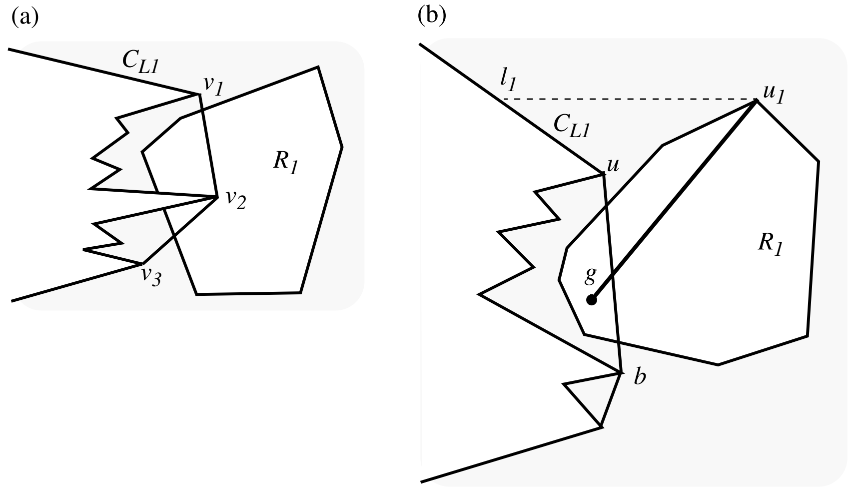


Figure 3.6: Proof of Lemma 3.3: (a) impossibility for C_{L1} to have more than one edge intersecting R_1; (b) finding edge (u, b).

We claim that in this case either $a_1'' = a_1$, or $\pi_G(a_1'', a_2'')$ and $\pi_G(a_1, a_2)$ join together at a point before their first inflection edge (if any) closest to R_1. This implies that if $\pi_G(a_1, a_2)$ has no inflection edge (a case where whether H_G is open or closed is decided by $\pi_G(b_1, b_2)$) then $\pi_G(a_1'', a_2'')$ has no inflection edge either, and if ρ_1' is the first inflection edge of $\pi_G(a_1, a_2)$ (a case where H_G is closed with ρ_1' a candidate for ρ_1) then ρ_1' is also the first inflection edge of $\pi_G(a_1'', a_2'')$, and thus the lemma follows.

We now give the details for proving the above claim. Note that $\pi_G(a_1, a_2)$ joins π_l at some point then leaves π_l later, and similarly for $\pi_G(a_1'', a_2'')$. First, look at the case where the blocking point b is on B_L (step 5(b)i) and refer to Fig. 3.7(a)(b) to visualize the proof. By the definition of C_{L1} and the fact that b is on B_L, $\pi_G(l_1, b) \subseteq C_{L1}$ is the convex hull inside P of the boundary of B_L from l_1 to b and it does not touch B_R, so no vertex of B_R lies to the left of $(u, b) \in \pi_G(l_1, b)$. But a_1'' is to the left of (u, b), thus $(a_1'', b) \cap B_R = \emptyset$. We classify two subcases: (i) $(a_1'', b) \cap (B_L - \{b\}) = \emptyset$ and (ii) $(a_1'', b) \cap (B_L - \{b\}) \neq \emptyset$. For (i), let q be the vertex on $C_{L1} = \pi_G(l_1, x)$ immediately after b. Such q always exists since $b \neq x$: for (u, b) to intersect R_1, b cannot be l_2 or the first point c with $y(c) = y(l_1)$ or the second cusp c_2, and b cannot be the first vertex v_1 on B_R either since $b \in B_L$. Because C_{L1} is convex toward right, the chain $(u, b, q) \subseteq C_{L1}$ is convex toward right, but then (a_1'', b, q) is also convex toward right (see Fig. 3.7(a)). This means that the shortest path $\pi_G(a_1, p') \subseteq \pi_G(a_1, a_2)$ from a_1 to any point p' on π_l beyond b must go through b. Then the first link of $\pi_G(a_1, a_2)$ is (a_1'', b) since (a_1'', b) is tangent to R_1 and does not cross any boundary of P. Therefore $a_1'' = a_1$. For (ii), let CH be the convex hull inside P of the boundary of B_L between b and b', where b' is the intersection of B_L and (a_1'', b) such that CH is as large as possible while not intersecting R_1. Clearly $\pi_G(a_1, a_2)$ goes through b, starting with a common tangent between R_1 and CH then following CH up to b; likewise, $\pi_G(a_1'', a_2'')$ goes through b starting with a tangent from a_1'' to CH then following CH up to b (see Fig. 3.7(b)). Observe that $\pi_G(a_1'', a_2'')$ and $\pi_G(a_1, a_2)$ join together at a point on CH that is before b, and neither path has an inflection edge before reaching b, so the claim holds.

Now look at the case where b is on B_R (step 5(b)ii). There are four subcases: (1) $w'' \in B_L$ and $(w'', a_1'') \cap (B_L - \{w''\}) = \emptyset$; (2) $w'' \in B_L$ and $(w'', a_1'') \cap (B_L - \{w''\}) \neq \emptyset$; (3) $w'' \in B_R$ and $(w'', a_1'') \cap B_L = \emptyset$; and (4) $w'' \in B_R$ and $(w'', a_1'') \cap B_L \neq \emptyset$. For (1), let x_1 and x_2 be the vertices on π_l immediately before and after w'' (see Fig. 3.7(c)). Note that (x_1, w'') is an inflection edge, so the chain (x_1, w'', x_2) is convex toward right (although $\pi_G(b, w'')$ is convex toward left). But the slope of (w'', a_1'') is even bigger than the slope of (w'', x_1), thus (a_1'', w'', x_2) is also convex toward right. Similar to case (i), this means that $\pi_G(a_1, p') \subseteq \pi_G(a_1, a_2)$ from a_1 to any point p' on π_l beyond w'' must go through w'', but (a_1'', w'') is a tangent to R_1 not blocked by P and hence the first link of $\pi_G(a_1, a_2)$, i.e., $a_1'' = a_1$. Case (2) is similar to case (ii) as w'' plays the role of b, i.e., both $\pi_G(a_1, a_2)$ and $\pi_G(a_1'', a_2'')$ go through a convex hull CH inside P of some portion of B_L then reach w'', with no inflection edge up to w'' (see Fig. 3.7(d)). For (3), it is clear that $\pi_G(a_1, p') \subseteq \pi_G(a_1, a_2)$ from a_1 to any point p' on π_l beyond w'' must go through w'', but (a_1'', w'') is a tangent to R_1 not blocked by P, so (a_1'', w'') is the first link of $\pi_G(a_1, a_2)$ and $a_1'' = a_1$ (see Fig. 3.7(e)). For (4), let CH' be the convex hull inside P of the boundary of B_L from q_1 to q_2, where q_1 is the intersection of (w'', a_1'') and B_L closest to w'', and q_2 is the intersection of (w'', a_1'') and B_L such that CH' is as large as possible while not intersecting R_1. Then $\pi_G(a_1, a_2)$ goes through w'', starting with a common tangent between R_1 and CH', followed by a portion of CH', a common tangent s between CH' and $C = \pi_G(b, z)$, then a portion C' of C up to w''; likewise, $\pi_G(a_1'', a_2'')$ goes through w'' starting with a tangent from a_1'' to CH', followed by a portion of CH' then s then C' up to w'' (see Fig. 3.7(f)). Clearly, the paths $\pi_G(a_1, a_2)$ and $\pi_G(a_1'', a_2'')$ join together at some point on CH' before reaching their first inflection edge s. This completes our proof of the claim. $\square$

We conclude with the following lemma.

Lemma 3.5 *Algorithm* Pseudo-Hourglass *correctly decides whether the geodesic hourglass H_G is open or closed, giving a visibility link when it is open or giving the penetrations and apices of H_G when it is closed, in $O(\log h + \log n)$ time, which is optimal.*

Proof: The correctness follows from Lemmas 3.1–3.4. As for time complexity, recall from our data structure (described at the beginning of Section 3.3) that we can extract a portion of a shortest path (*path extraction* for short) via split/splice operations in logarithmic time. Step 1 performs $O(1)$ tangent computations and shortest-path queries. Step 2 performs four shortest-path queries. Step 3(a)i can be done in $O(1)$ time, and step 3(a)ii involves $O(1)$ point-location queries to find projection points. In Step 3b, we perform a path extraction; in steps 3(b)i and 3(b)ii, we perform $O(1)$ point-location queries to project points and also binary searches for special-case checkings. We compute two shortest-path queries and extract four bounding convex chains in step 4. Step 5a invloves $O(1)$ calls to algorithm [20], and $O(1)$ tangent computations and binary searches. Step 5(a)i can be done in $O(1)$ time, and step 5(a)ii performs $O(1)$ path extractions and tangent computations. Step 5b applies the computation of Lemma 3.3, which is a binary search. Steps 5(b)i–5(b)ii invlove $O(1)$ tangent computations (5(b)i and 5(b)ii) and path extractions (5(b)ii). Finally, we perform $O(1)$ shortest-path queries, tangent computations and binary searches in step 6. In summary, we perform a constant number of logarithmic-time computations, and the time complexity follows. $\square$

3.3.2 The Case of Mutually Visible Query Polygons

We now discuss how to compute $\pi_G(R_1, R_2)$ when R_1 and R_2 are mutually visible, i.e., when the geodesic hourglass H_G is open. Surprisingly, this case turns out to be nontrivial, and its solution makes use of interesting geometric properties. Note that $\pi_G(R_1, R_2)$ in this case may still consist of more than one link (see, e.g., Fig. 3.8, where $\pi_G(R_1, R_2) = \pi_G(p, q)$).

Ignoring P and using any one of the methods for computing the separation of two convex polygons [31, 45, 48], we can find $p' \in R_1$ and $q' \in R_2$ with $length(p', q') = \sigma(R_1, R_2)$ in $O(\log h)$ time. Now we compute $\pi_G(p', q')$. If $\pi_G(p', q')$ has only one link, then (p', q') is not blocked by P and thus is the desired shortest path $\pi_G(R_1, R_2)$. Otherwise $\pi_G(p', q')$ must touch the boundary of P, and there are two cases: (1) $\pi_G(p', q')$ touches only one of the two geodesic external tangents

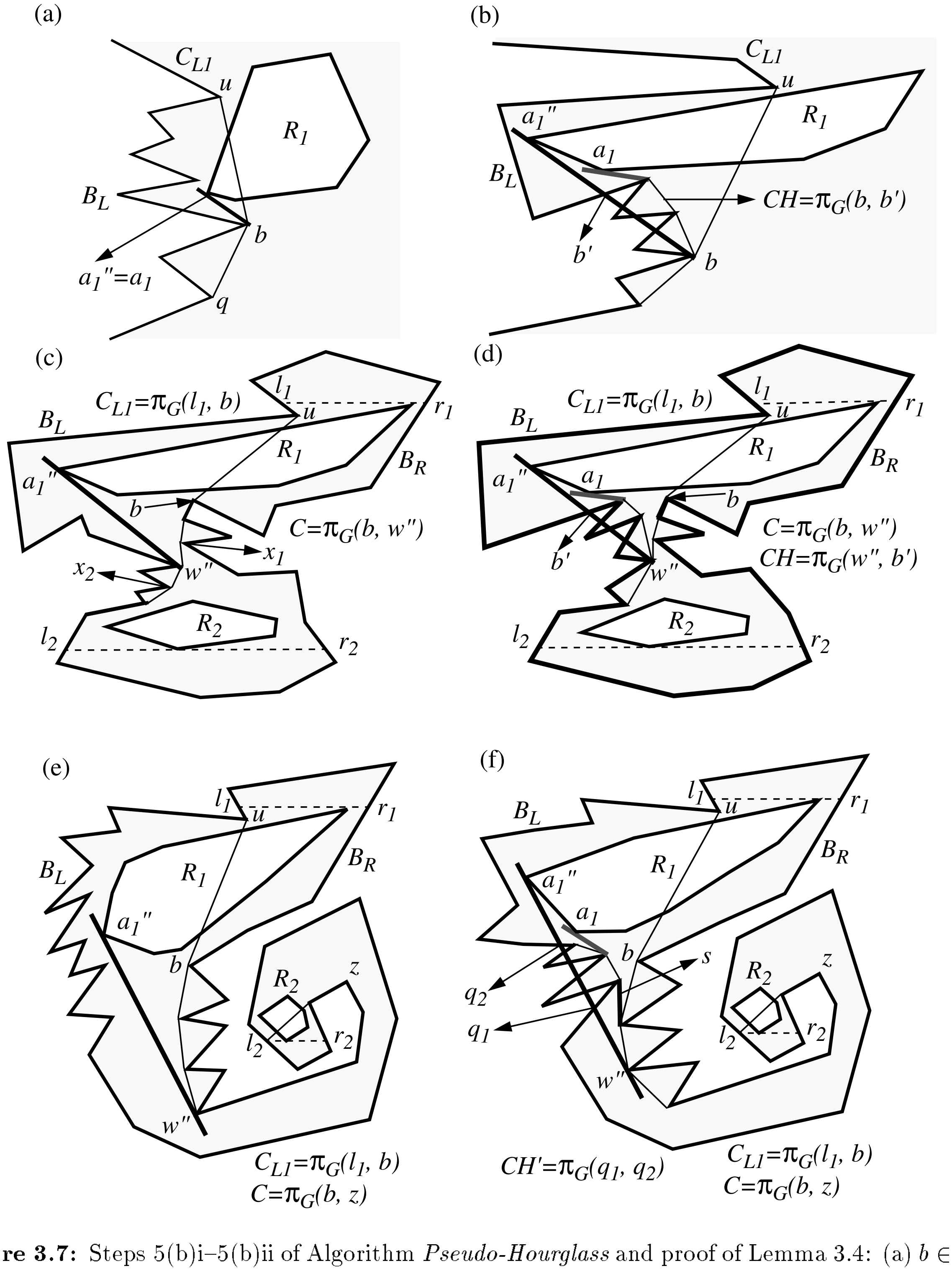


Figure 3.7: Steps 5(b)i–5(b)ii of Algorithm *Pseudo-Hourglass* and proof of Lemma 3.4: (a) $b \in B_L$ and $(b, a_1'') \cap (B_L - \{b\}) = \emptyset$; (b) $b \in B_L$ and $(b, a_1'') \cap (B_L - \{b\}) \neq \emptyset$; (c) $b \in B_R$, $w'' \in B_L$ and $(w'', a_1'') \cap (B_L - \{w''\}) = \emptyset$; (d) $b \in B_R$, $w'' \in B_L$ and $(w'', a_1'') \cap (B_L - \{w''\}) \neq \emptyset$; (e) $b \in B_R$, $w'' \in B_R$ and $(w'', a_1'') \cap B_L = \emptyset$; and (f) $b \in B_R$, $w'' \in B_R$ and $(w'', a_1'') \cap B_L \neq \emptyset$.

$\pi_G(a_1, a_2)$ and $\pi_G(b_1, b_2)$; or (2) $\pi_G(p', q')$ touches both $\pi_G(a_1, a_2)$ and $\pi_G(b_1, b_2)$.

Lemma 3.6 *Let the geodesic hourglass H_G be open and (p', q') with $p' \in R_1$ and $q' \in R_2$ be the shortest path between R_1 and R_2 without obstacle P. If $\pi_G(p', q')$ touches only one of $\pi_G(a_1, a_2)$ and $\pi_G(b_1, b_2)$, say $\pi_G(a_1, a_2)$, then $\pi_G(R_1, R_2)$ touches $\pi_G(a_1, a_2)$ but does not touch $\pi_G(b_1, b_2)$.*

Proof: We refer to Fig. 3.8 to visualize the proof. Let (w, z) be any segment tangent to the convex chain $\pi_G(p', q')$, where $w \in R_1$ and $z \in R_2$. *Without obstacles C and D*, the distance between a point on the boundary of R_1 and a point on the boundary of R_2 is a *bimodal function*, i.e., it decreases and then increases, with the minimum occurring at p' and q'. In particular, moving w downward along the boundary of R_1 to any point w' and/or moving z downward along the boundary of R_2 to any point z' will cause $(w', z') > (w, z)$, and $\pi_G(w', z') \geq (w', z')$ since $\pi_G(w', z')$ may have to avoid obstacles. Thus if $p \in R_1$ and $q \in R_2$ satisfy $\pi_G(p, q) = \pi_G(R_1, R_2)$, then p must lie on the boundary $(w, ..., p')$ of R_1 counterclockwise from w to p', and q must lie on the clockwise boundary $(z, ..., q')$ of R_2. It follows that $\pi_G(p, q)$ touches $\pi_G(a_1, a_2)$ but does not touch $\pi_G(b_1, b_2)$. □

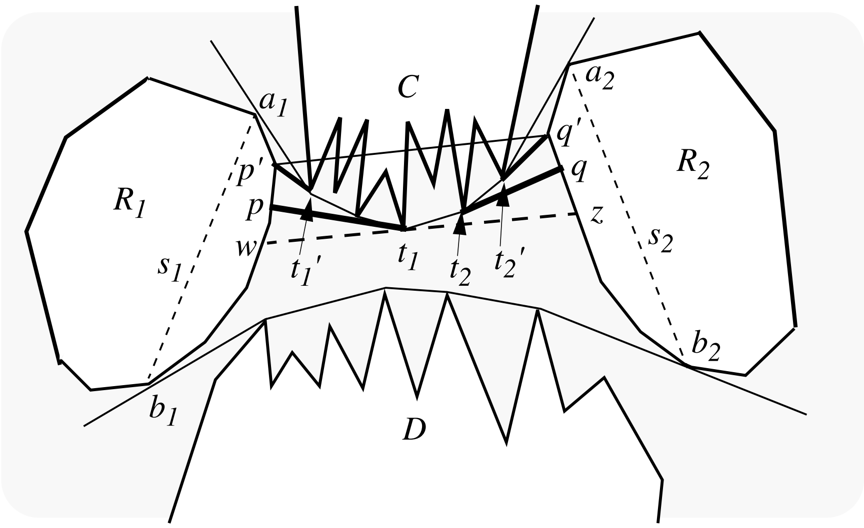


Figure 3.8: Lemma 3.6

Therefore in the above situation (see Fig. 3.8), if t_1' and t_2' are the points of obstacle C where $\pi_G(p', q')$ first touches C and finally leaves C, respectively, and t_1 and t_2 are the points of C where $\pi_G(p, q)$ first touches C and finally leaves C (recall that $\pi_G(p, q) = \pi_G(R_1, R_2)$), then t_2 is the point where the shortest path $\pi_G(t_1', R_2)$ from t_1' to R_2 finally leaves C, and similarly for t_1. We say that $t_2 \in C$ and $q \in R_2$ *realize* $\pi_G(t_1', R_2)$, and similarly for the other side. It is clear that $\pi_G(R_1, R_2)$ consists of $(p, t_1), \pi_G(t_1, t_2)$ (which is a portion of $\pi_G(p', q')$), and (t_2, q). So we only need to independently compute $t_2 \in C$ and $q \in R_2$ that realize $\pi_G(t_1', R_2)$, and by a similar algorithm to compute t_1 and p that realize $\pi_G(t_2', R_1)$.

Before describing how to compute t_2 and q (and similarly for t_1 and p), we first argue that the other case where $\pi_G(p', q')$ touches both $\pi_G(a_1, a_2)$ and $\pi_G(b_1, b_2)$ can be handled in the same way.

Lemma 3.7 *Let the geodesic hourglass H_G be open and (p', q') with $p' \in R_1$ and $q' \in R_2$ be the shortest path between R_1 and R_2 without obstacle P. If $\pi_G(p', q')$ touches both $\pi_G(a_1, a_2)$ and $\pi_G(b_1, b_2)$, say first $\pi_G(a_1, a_2)$ (entering at point t_1 and leaving at point t_3) and then $\pi_G(b_1, b_2)$ (entering at t_4 and leaving at t_2), then $\pi_G(R_1, R_2) = \pi_G(R_1, t_3) \cup (t_3, t_4) \cup \pi_G(t_4, R_2)$. (See Fig. 3.9.)*

Proof: We refer to Fig. 3.9. We extend (t_3, t_4) on both directions to intersect R_1 and R_2 at w and z, respectively. Notice that (w, z) is an internal common tangent of two convex chains $\pi_G(a_1, a_2)$

and $\pi_G(b_1, b_2)$. Again, without obstacles the distance between a point on R_1 and a point on R_2 is a bimodal function. In particular, moving z upward along the boundary of R_2 to any point z' and/or moving w downward along the boundary of R_1 to any point w' will make $(w', z') > (w, z)$. Observe that $\pi_G(w', z') \geq (w', z')$ since it may have to avoid the obstacles. Therefore the desired points $p \in R_1$ and $q \in R_2$ with $\pi_G(p, q) = \pi_G(R_1, R_2)$ must lie on the clockwise boundary $(p', ..., w)$ of R_1 and on the clockwise boundary $(q', ..., z)$ of R_2, respectively. It follows that $\pi_G(p, q)$ must be first tangent to $\pi_G(a_1, a_2)$ at some point, coincide with $\pi_G(a_1, a_2)$ from there to t_3, follow (t_3, t_4) to enter $\pi_G(b_1, b_2)$, join $\pi_G(b_1, b_2)$ from t_4 to some tangent point, which together with q are the two endpoints of the last link. Therefore $\pi_G(R_1, R_2) = \pi_G(R_1, t_3) \cup (t_3, t_4) \cup \pi_G(t_4, R_2)$. □

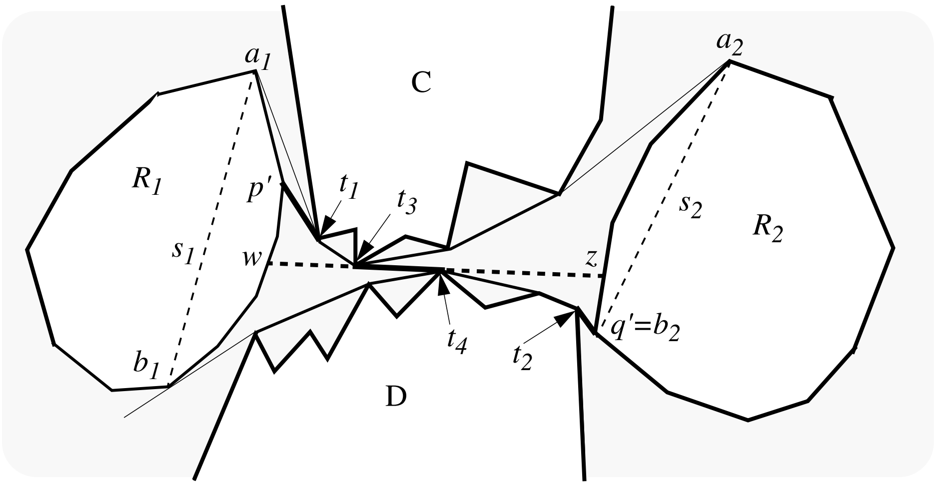


Figure 3.9: Lemma 3.7

It is clear that for the above situation, what we need to do is to independently compute the two points that realize $\pi_G(R_1, t_3)$ and two points that realize $\pi_G(t_4, R_2)$.

We now discuss how to compute two points $t_2 \in C$ and $q \in R_2$ that realize $\pi_G(t_1', R_2)$ in the situation of Fig. 3.8; the other case (Fig. 3.9) can be handled analogously. Note that we only need to consider the two convex chains $\pi_G(u, t_2')$ (denoted by C_1) and the clockwise boundary $(v, ..., q')$ of R_2 (denoted by C_2), where (u, v) is the external common tangent between the convex hull of C and R_2 with $u \in C$ and $v \in R_2$. Our algorithm is based on the following useful properties.

Lemma 3.8 *Let $v_1, v_2, ..., v_k$ be a sequence of points on C_2 in clockwise order, and e_i' and e_i'' be the two segments of C_2 incident on v_i with e_i' following e_i'' in clockwise order (e_i' and e_i'' are on the same straight line if v_i is not a vertex). From each v_i draw a line l_i tangent to C_1. Let θ_i be the angle formed by l_i and e_i' and measured from l_i clockwise to e_i', and ϕ_i be the angle formed by e_i'' and l_i and measured from e_i'' clockwise to l_i (see Fig. 3.10). Then $\theta_1 < \theta_2 < ... < \theta_k$ and $\phi_1 > \phi_2 > ... > \phi_k$. Also, if $\theta_i \geq \frac{\pi}{2}$ then $\phi_{i+1} < \frac{\pi}{2}$, and similarly if $\phi_{i+1} \geq \frac{\pi}{2}$ then $\theta_i < \frac{\pi}{2}$.*

Proof: We extend tangent l_{i+1} to intersect l_i at some point r, and also extend e_i' on both directions so that θ_{i+1}' and ϕ_i' are both exterior angles of $\triangle r v_i v_{i+1}$ (see Fig. 3.10). It follows that $\theta_{i+1} \geq \theta_{i+1}' > \theta_i$ (the equality holds if v_{i+1} is not a vertex), and $\phi_i \geq \phi_i' > \phi_{i+1}$ (the equality holds if v_i is not a vertex). For the last statement, consider $\triangle r v_i v_{i+1}$. It is clear that at most one of θ_i and ϕ_{i+1} can be larger than or equal to $\frac{\pi}{2}$. □

Lemma 3.9 *Let $v_1, v_2, ..., v_k$ and each θ_i and ϕ_i be as defined in Lemma 3.8. If $\phi_i \geq \frac{\pi}{2}$, then $\pi_G(v_i, t_1') < \pi_G(v_{i-1}, t_1')$. Similarly, if $\theta_i \geq \frac{\pi}{2}$, then $\pi_G(v_i, t_1') < \pi_G(v_{i+1}, t_1')$.*

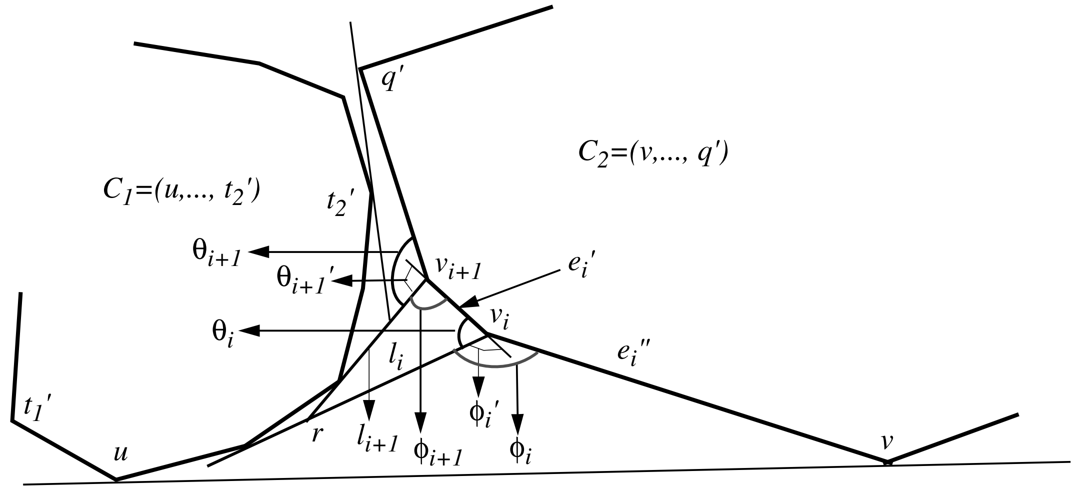


Figure 3.10: Lemma 3.8

Proof: We refer to Fig. 3.11 to visualize the proof. Let the tangent points on C_1 of l_i and of l_{i-1} be u_j and u_m, respectively, where $u_1, u_2, ...$ are the vertices of C_1 in counterclockwise order. We extend each of (u_s, u_{s+1}) to the right to intersect C_2 at some point u'_s, $s = m, m+1, ..., j-1$. In $\triangle v_i u_j u'_{j-1}$, $(u_j, u'_{j-1}) > (u_j, v_i)$ since $\phi_i \geq \frac{\pi}{2}$ is the biggest angle. Adding (u_j, u_{j-1}) to both sides of the inequality, we have $\pi_G(v_i, u_{j-1}) = (v_i, u_j) + (u_j, u_{j-1}) < (u'_{j-1}, u_j) + (u_j, u_{j-1}) = (u'_{j-1}, u_{j-1})$, thus $\pi_G(v_i, t'_1) = \pi_G(v_i, u_{j-1}) + \pi_G(u_{j-1}, t'_1) < (u'_{j-1}, u_{j-1}) + \pi_G(u_{j-1}, t'_1) = \pi_G(u'_{j-1}, t'_1)$, i.e., $\pi_G(v_i, t'_1) < \pi_G(u'_{j-1}, t'_1)$. Now, $\phi'_i = \angle u'_{j-2} u'_{j-1} u_j$ is an exterior angle of $\triangle u'_{j-1} v_i u_j$, so $\phi'_i > \phi_i \geq \frac{\pi}{2}$. By the previous argument, $\pi_G(u'_{j-1}, t'_1) < \pi_G(u'_{j-2}, t'_1)$. Applying this process repeatedly, we have $\pi_G(v_i, t'_1) < \pi_G(u'_{j-1}, t'_1) < \pi_G(u'_{j-2}, t'_1) < ... < \pi_G(v_{i-1}, t'_1)$. The other statement can be proved in the same way. □

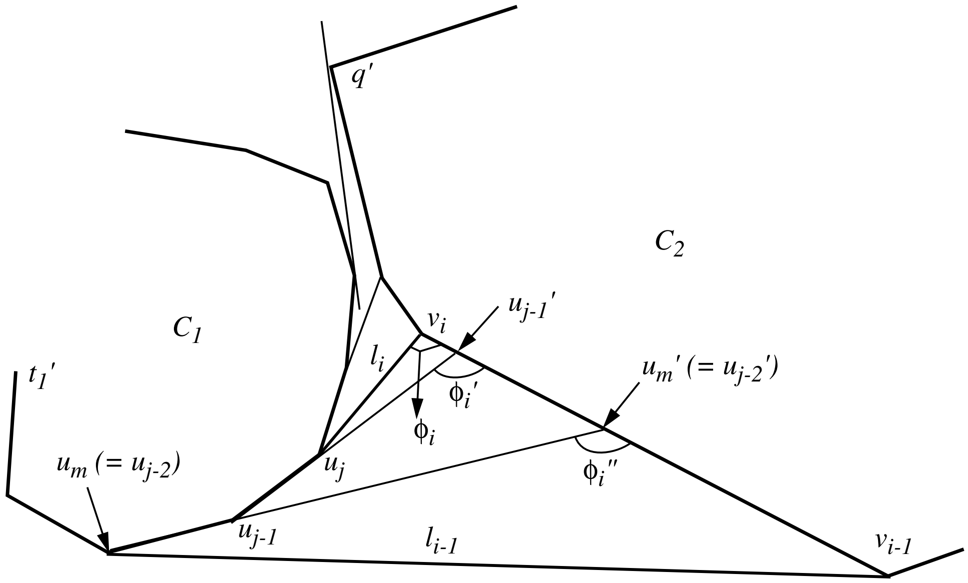


Figure 3.11: Lemma 3.9

Notice that for each $v_i \in C_2$, $\theta_i + \phi_i \geq \frac{\pi}{2}$ since C_2 is a convex chain (the equality holds when v_i

is not a vertex), thus either $\phi_i \geq \frac{\pi}{2}$ and $\pi_G(v_i, t_1') < \pi_G(v_{i-1}, t_1') < \pi_G(v_{i-2}, t_1') < ...$, or $\theta_i \geq \frac{\pi}{2}$ and $\pi_G(v_i, t_1') < \pi_G(v_{i+1}, t_1') < \pi_G(v_{i+2}, t_1') < ...$, by Lemmas 3.8 and 3.9. If both $\phi_i \geq \frac{\pi}{2}$ and $\theta_i \geq \frac{\pi}{2}$, then $v_i = q$, i.e., $\pi_G(v_i, t_1') = \pi_G(C_2, t_1')$. We summarize this result in the following lemma.

Lemma 3.10 *Let w be a point on C_2. Moving w along C_2, the length of $\pi_G(w, t_1')$ is a bimodal function, i.e., it decreases and then increases. In particular, the minimum value occurs at $w = v_i$ with $\phi_i \geq \frac{\pi}{2}$ and $\theta_i \geq \frac{\pi}{2}$. If this v_i is not a vertex, then $\phi_i = \theta_i = \frac{\pi}{2}$, namely, the line issuing from v_i and tangent to C_1 is perpendicular to the edge of C_2 containing v_i.*

Up to now we can compute $t_2 \in C_1$ and $q \in C_2$ that realize $\pi_G(t_1', C_2)$ by a binary search on the vertices of C_2, where at each step we compute a tangent of C_1 from the current vertex of C_2, check for angles θ and ϕ and then reduce the search space. Finally, we also have to take care of the case where q is not a vertex. Since tangent computation takes logarithmic time, this method has time complexity $O(\log h \log n)$. To speed up the algorithm, we appeal to the properties from C_1.

Lemma 3.11 *Let $u_1 = u, u_2, ..., u_k = t_2'$ be the vertices of C_1 in counterclockwise order. The extension of each edge (u_{i-1}, u_i) intersects C_2 at some point v_i, $i = 2, ...k$. Let v_i' and v_i'' be the two vertices of C_2 adjacent to v_i, with v_i' following v_i'' in clockwise order. Let $\theta_i = \angle u_i v_i v_i'$ and $\phi_i = \angle u_i v_i v_i''$ (see Fig. 3.12). Then $\theta_2 < \theta_3 < ... < \theta_k$, and $\phi_2 > \phi_3 > ... > \phi_k$.*

Proof: Since (q', t_2') is a tangent to C_1 (recall this from Fig. 3.8), its slope is larger than the slope of (u_{k-1}, t_2'), which shows that the extension of (u_{k-1}, t_2') is below q' and thus intersects C_2. Similar argument applies to the extension of (u_1, u_2), so all such extensions intersect C_2. We now prove that $\theta_i < \theta_{i+1}$; the proof of $\phi_i > \phi_{i+1}$ is similar. Let $w_1, ..., w_l$ be the vertices of C_2 between v_i and v_{i+1} in clockwise order. Draw a segment to connect u_i with each of $w_1, ..., w_l$ and define $\theta_i' = \angle u_i w_i w_{i+1}$ ($\theta_l' = \angle u_i w_l v_{i+1}$). Then $\theta_i < \theta_1' < ... < \theta_l' < \theta_{i+1}$ by the argument that an exterior angle of a triangle is larger than each of the two far interior angles. □

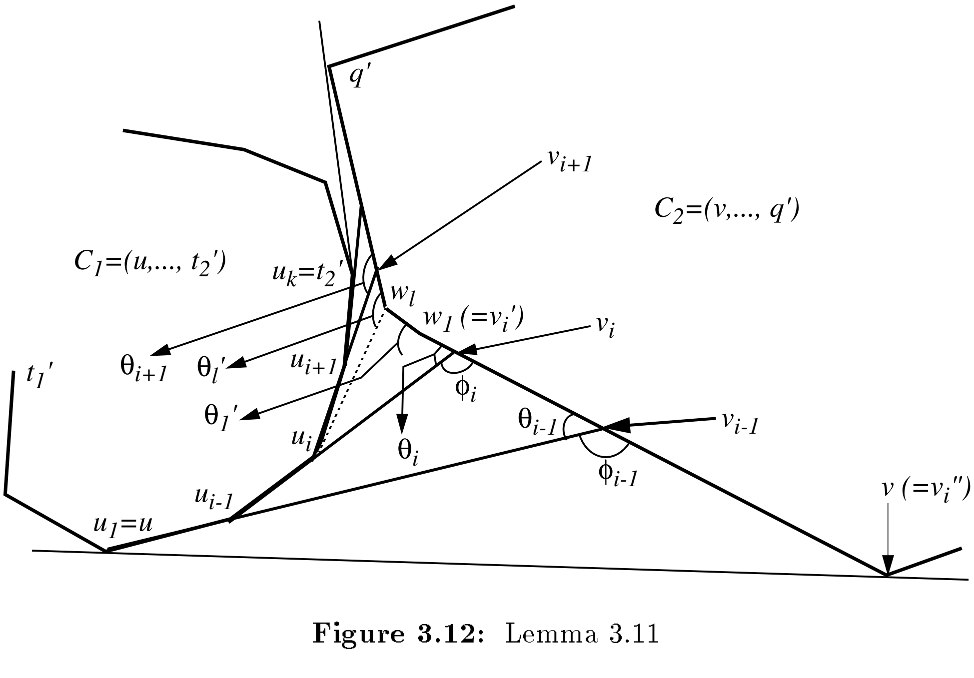


Figure 3.12: Lemma 3.11

Lemma 3.12 *Let $t_2 \in C_1$ and $q \in C_2$ realize $\pi_G(t_1', C_2)$, where t_2 is some vertex u_j. Let each θ_i be defined as in Lemma 3.11. Then $\theta_j < \frac{\pi}{2}$ and $\theta_{j+1} > \frac{\pi}{2}$.*

Proof: We refer to Fig. 3.13. Let v' and v'' be the two vertices of C_2 adjacent to point q, with v' following v'' in clockwise order. There are two cases. If q is not a vertex, then by Lemma 3.10, (u_j, q)

is perpendicular to (v', v'') (see Fig. 3.13(a)). We extend (v', v'') to intersect rays (u_{j-1}, u_j) and (u_j, u_{j+1}) respectively at r'' and r', and make angles θ'' and θ' as shown. We see that $\theta' > \angle u_j q r' = \frac{\pi}{2}$ since it is an exterior angle of $\triangle u_j q r'$, and $\theta_{j+1} \geq \theta'$ (the equality holds when ray (u_j, u_{j+1}) intersects *edge* (v', v'')), so $\theta_{j+1} > \frac{\pi}{2}$. Similarly $\theta'' < \frac{\pi}{2}$ (since in $\triangle u_j q r''$, $\angle u_j q r'' = \frac{\pi}{2}$) and $\theta_j \leq \theta''$ (again, the equality holds when ray (u_{j-1}, u_j) intersects (v', v'')), so $\theta_j < \frac{\pi}{2}$. In the other case where q is a vertex, by Lemma 3.10 $\angle u_j q v', \angle u_j q v'' \geq \frac{\pi}{2}$ (see Fig. 3.13(b)). Again we extend (q, v') to intersect ray (u_j, u_{j+1}) and make angle θ', and extend (q, v'') to intersect ray (u_{j-1}, u_j) and make angle θ'' as shown. By the same argument, we have that $\theta_{j+1} \geq \theta' > \angle u_j q v' \geq \frac{\pi}{2}$ and $\theta_j \leq \theta'' < \frac{\pi}{2}$. □

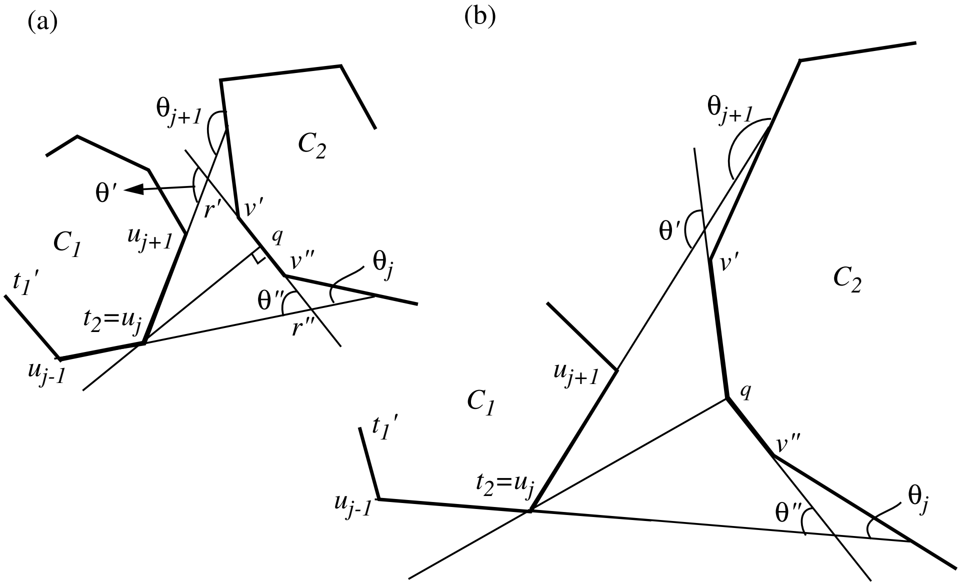


Figure 3.13: Lemma 3.12

Now we are ready to state the algorithm for computing $t_2 \in C_1$ and $q \in C_2$ that realize $\pi_G(t_1', C_2)$. This is actually a double-binary search.

Algorithm Double-Search

1. If either $| C_1 | = 1$ or $| C_2 | \leq 2$ then go to step 3.
2. Else, pick the median vertices v and w of current C_1 and C_2. Let v' be the vertex of C_1 that precedes v in counterclockwise order, and w' be the vertex of C_2 that follows w in clockwise order. Intersect the ray $r = (v', v)$ with the line extension l' of edge (w, w'). Let θ be the angle made by r and l' by measuring clockwise from r to l'. The actions (and the verification) depend on the following cases (see Fig. 3.14):
   (a) The intersection is below (w, w') and $\theta \geq \frac{\pi}{2}$ (Fig. 3.14(a)): prune the wiggly portion (not including w).
   Verification: Draw a line l from w parallel to (v', v). Since l is above (v', v), the tangent t from w to C_1 must make an angle $\theta' > \theta \geq \frac{\pi}{2}$. Thus the tangent of C_1 from any point in the wiggly portion will make an angle even bigger, so this portion can be pruned away by Lemma 3.10.
   (b) The intersection is below (w, w') and $\theta < \frac{\pi}{2}$ (Fig. 3.14(b)): prune the wiggly portion (including v').
   Verification: The real intersection between ray (v', v) and C_2 makes an angle $\theta' < \theta < \frac{\pi}{2}$.

By Lemmas 3.11 and 3.12, any edge in the wiggly portion will make an angle even smaller and thus this portion can be pruned away.

(c) The intersection is above (w, w') and $\theta \geq \frac{\pi}{2}$ (Fig. 3.14(c)): prune the wiggly portion (including v). This is symmetric to case (b).

(d) The intersection is above (w, w') and $\theta < \frac{\pi}{2}$ (Fig. 3.14(d)): prune the wiggly portion. This is symmetric to case (a). Note that w itself is not a candidate for q but w is not pruned away here, since q may still lie on (w, w') and thus w must be kept to retain (w, w').

(e) The intersection is on (w, w') and $\theta \geq \frac{\pi}{2}$ (Fig. 3.14(e)): prune the two wiggly portions (including v but not w' so that (w, w') is kept). This is a situation combining cases (a) and (c).

(f) The intersection is on (w, w') and $\theta < \frac{\pi}{2}$ (Fig. 3.14(f)): prune the two wiggly portions (including v' but not w so that (w, w') is kept). Again this is a situation combining cases (b) and (d).

After pruning the appropriate portions, go to step 1.

3. Now $| C_1 |= 1$ or $| C_2 | \leq 2$, a situation where the double-binary search in step 2 cannot proceed (either $| C_1 |= 1$ and $| C_2 | \neq constant$ or $| C_2 |= 1$ and $| C_1 | \neq constant$) or may not make any progress (case (d) with $| C_2 |= 2$ and $| C_1 | \neq constant$). The operations depend on the following cases:

   (a) $| C_2 |= 1$. The only vertex of C_2 is q. Compute the tangent from q to C_1 and take t_2 as the tangent point. Report q and t_2, and stop.

   (b) $| C_2 |= 2$. Let $C_2 = \{w_1, w_2\}$ such that walking from w_1 to w_2 the interior of R_2 is to the right of (w_1, w_2). From w_1 and w_2 compute tangents (w_1, v_1) and (w_2, v_2) of C_1, where $v_1, v_2 \in C_1$. Let $\theta_1 = \angle v_1 w_1 w_2$ and $\phi_2 = \angle v_2 w_2 w_1$. There are three subcases.

      i. $\theta_1 \geq \frac{\pi}{2}$. By Lemma 3.10, $q = w_1$ (and $t_2 = v_1$). Report q and t_2, and stop. Note that $\phi_2 < \frac{\pi}{2}$ by Lemma 3.8.

      ii. $\phi_2 \geq \frac{\pi}{2}$. Report $q = w_2$, $t_2 = v_2$, and stop. This is symmetric to case i.

      iii. $\theta_1 < \frac{\pi}{2}$ and $\phi_2 < \frac{\pi}{2}$ (and $q \neq w_1, w_2$). By Lemma 3.10, (t_2, q) is perpendicular to (w_1, w_2) and is tangent to C_1. Perform a binary search on subchain $(v_1, ..., v_2)$ of C_1 to find such vertex t_2: At each iteration with current vertex v, compute its projection point v' on (w_1, w_2), check whether vertex v on C_1 is concave, reflex or supporting with respect to (v, v') and branch appropriately. When v is supporting, report $t_2 = v$, $q = v'$ and stop.

   (c) $| C_1 |= 1$. The only vertex of C_1 is t_2. Now perform a binary search on C_2. Let $w_1, ..., w_k$ be the vertices of C_2 in clockwise order. At each step with current vertex w_i, let $\theta_i = \angle t_2 w_i w_{i+1}$ and $\phi_i = \angle t_2 w_i w_{i-1}$. Recall that $\theta_1 < \theta_2 < ... < \theta_k$ by Lemma 3.8, and if $\theta_i \geq \frac{\pi}{2}$ and $\phi_i \geq \frac{\pi}{2}$ then $w_i = q$ by Lemma 3.10. The binary search proceeds to find the smallest index i such that $\theta_i \geq \frac{\pi}{2}$. If also $\phi_i \geq \frac{\pi}{2}$, then $q = w_i$; report q and t_2, and stop. Else, both ϕ_i and θ_{i-1} are less than $\frac{\pi}{2}$, and thus t_2 has a projection q on (w_i, w_{i-1}). Report t_2 and q, and stop.

Note that the loop formed by steps 1 and 2 eventually makes either $| C_1 |= 1$ or $| C_2 | \leq 2$, and thus we finally exit the loop and go to step 3. Indeed, when C_1 is reduced (cases (b), (c), (e) and (f) of step 2), either v or v' is also pruned away, so that C_1 with $| C_1 |= 2$ is further reduced to $| C_1 |= 1$; when only C_2 is reduced (cases (a) and (d) of step 2), one of the two portions preceding and following w is pruned away, so that C_2 with $| C_2 |= 3$ is further reduced to $| C_2 |= 2$.

Lemma 3.13 *The time complexity of Algorithm* Double-Search *is* $O(\log h + \log n)$.

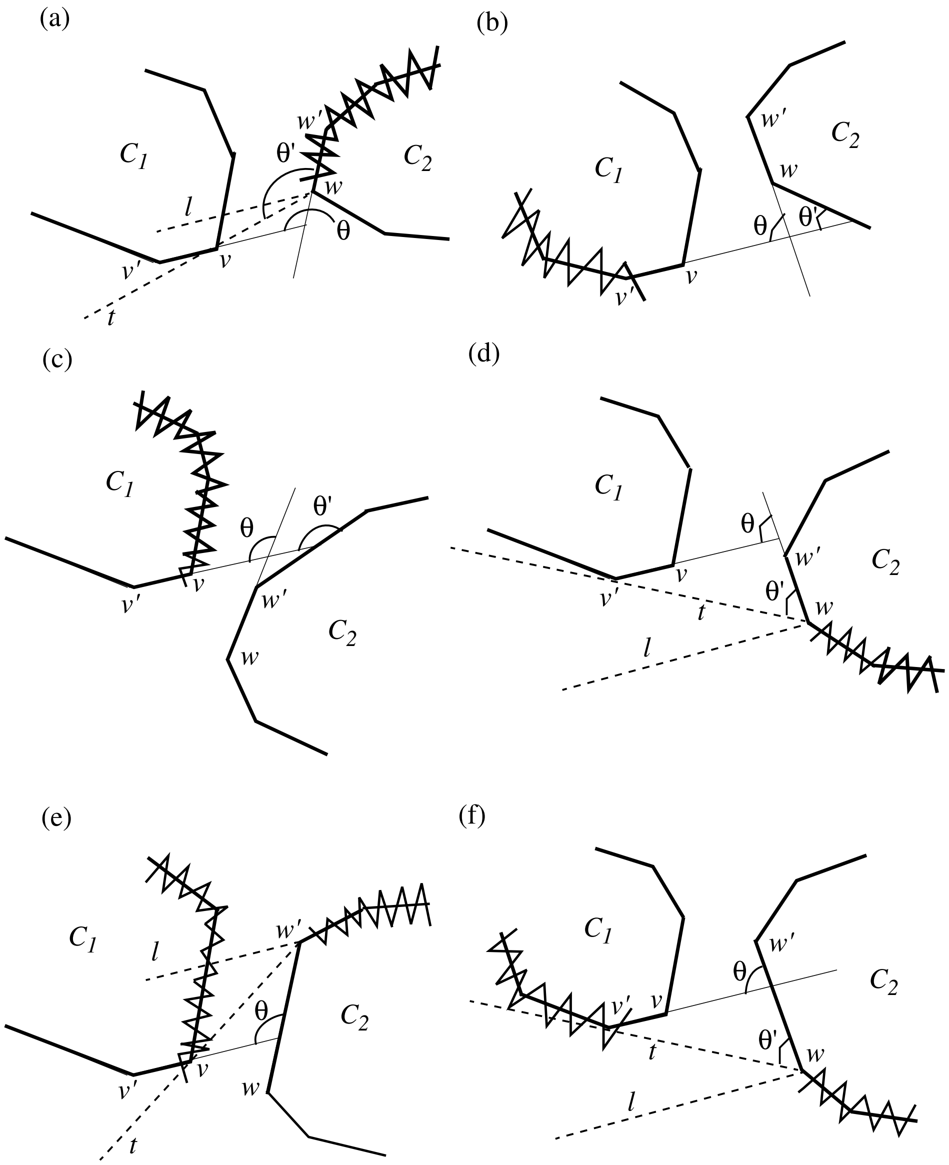


Figure 3.14: The cases (a)–(f) in step 2 of Algorithm *Double-Search*.

Proof: In each case of step 2, we always discard half of C_1 and/or half of C_2, so the loop formed by steps 1 and 2 takes $O(\log h + \log n)$ time. Step 3 also takes logarithmic time, since either $\mid C_1 \mid$ or $\mid C_2 \mid$ is a constant and a constant number of simple binary searches are performed on the other chain. □

We now give an algorithm for computing the shortest path $\pi_G(R_1, R_2)$ between R_1 and R_2 when they are mutually visible.

Algorithm Visible-Path

1. Ignore P and compute the separation $\sigma(R_1, R_2)$ of R_1 and R_2 by any one of the methods [31, 45, 48], which gives two points $p' \in R_1$ and $q' \in R_2$ such that $length(p', q') = \sigma(R_1, R_2)$.
2. Compute $\pi_G(p', q')$. If $\pi_G(p', q')$ has only one link, then (p', q') is not blocked by P; report $\pi_G(R_1, R_2) = (p', q')$ and stop.

3. Otherwise, $\pi_G(p', q')$ must touch the boundary of P. Let (p', t_1') and (t_2', q') be the first and last links of $\pi_G(p', q')$. Discriminate the two cases below:
   (a) There is no inflection edge in $\pi_G(p', q')$: this is the case of Lemma 3.6 (Fig. 3.8). Let $C = \pi_G(t_1', t_2')$. Find the external common tangent (u, v) between C and R_2, where $u \in C$ and $v \in R_2$; let C_1 be $\pi_G(u, t_2')$ and C_2 be the clockwise boundary $(v, ..., q')$ of R_2. Compute $t_2 \in C$ and $q \in R_2$ that realize $\pi_G(t_1', R_2)$ by performing Algorithm *Double-Search* on C_1 and C_2, and similarly compute $t_1 \in C$ and $p \in R_1$ that realize $\pi_G(t_2', R_1)$. Report $\pi_G(R_1, R_2) = (p, t_1) \cup \pi_G(t_1, t_2) \cup (t_2, q)$ and stop.
   (b) There is an inflection edge (t_3, t_4) in $\pi_G(p', q')$: this is the case of Lemma 3.7 (Fig. 3.9). Use Algorithm *Double-Search* to compute two pairs of points that respectively realize $\pi_G(R_1, t_3)$ and $\pi_G(t_4, R_2)$. Report $\pi_G(R_1, R_2) = \pi_G(R_1, t_3) \cup (t_3, t_4) \cup \pi_G(t_4, R_2)$ and stop.

Lemma 3.14 *The time complexity of Algorithm* Visible-Path *is $O(\log h + \log n)$ (plus $O(k)$ if the k links are reported).*

Proof: The separation computation in step 1 can be done in logarithmic time. Other computations involve a shortest-path query (step 2), two tangent computations and two calls of Algorithm *Double-Search* (step 3(a) or 3(b)), each taking logarithmic time. □

3.3.3 The Overall Algorithm

The overall algorithm for computing the shortest path $\pi_G(R_1, R_2)$ between R_1 and R_2 is as follows.

Algorithm Shortest-Path

1. Perform Algorithm *Pseudo-Hourglass* to decide whether the geodesic hourglass H_G is open or closed (with apices p_1 (closer to R_1) and p_2 (closer to R_2)).
2. If H_G is open, then apply Algorithm *Visible-Path* to report $\pi_G(R_1, R_2)$ and stop.
3. Otherwise (H_G is closed), apply Algorithm *Visible-Path* to find shortest paths $\pi_G(R_1, p_1)$ and $\pi_G(p_2, R_2)$ by treating p_1 and p_2 as "convex polygons" consisting of only one vertex. Compute shortest path $\pi_G(p_1, p_2)$, report $\pi_G(R_1, R_2) = \pi_G(R_1, p_1) \cup \pi_G(p_1, p_2) \cup \pi_G(p_2, R_2)$ and stop.

Lemma 3.15 *Algorithm* Shortest-Path *has time complexity $O(\log h + \log n)$ (plus $O(k)$ if the k links of the path are reported), which is optimal.*

Theorem 3.1 *Let P be a simple polygon with n vertices. There exists an optimal data structure that supports shortest-path queries between two convex polygons with a total of h vertices inside P in time $O(\log h + \log n)$ (plus $O(k)$ if the k links of the path are reported), using $O(n)$ space and preprocessing time; all bounds are worst-case.*

Remark. Although the case of mutually visible R_1 and R_2 is nontrivial, our algorithms (*Double-Search* and *Visible-Path*) turn out to involve only simple computations, by applying useful geometric properties. The other key technique, Algorithm *Pseudo-Hourglass*, to decide whether H_G between R_1 and R_2 is open (and compute a visibility link) or closed (and compute apices and penetrations), however, is more involved. We pose as an open problem whether there exist simpler techniques to perform the same operations in the same (optimal) time bound. Also, whether we can directly compute H_G in optimal time is an open problem, and may be of independent interest.

3.4 Dynamic Shortest Path Queries

In this section, we consider the shortest-path problem in a connected planar map $\mathcal{M}$ in a dynamic environment. The query operation is to compute the shortest path $\pi_G(R_1, R_2)$, where the two query convex polygons R_1 and R_2 are given in the same region P of $\mathcal{M}$. In addition, we support edge/vertex insertions and deletions on $\mathcal{M}$ in our data structure. Specifically, we define the following update operations on $\mathcal{M}$:

INSERTEDGE$(e, v, w, P; P_1, P_2)$: Insert edge $e = (v, w)$ into region P such that P is partitioned into two regions P_1 and P_2.

REMOVEEDGE$(e, v, w, P_1, P_2; P)$: Remove edge $e = (v, w)$ and merge the regions P_1 and P_2 formerly on the two sides of e into a new region P.

INSERTVERTEX$(v, e; e_1, e_2)$: Split the edge $e = (u, w)$ into two edges $e_1 = (u, v)$ and $e_2 = (v, w)$ by inserting vertex v along e.

REMOVEVERTEX$(v, e_1, e_2; e)$: Let v be a vertex with degree two such that its incident edges $e_1 = (u, v)$ and $e_2 = (v, w)$ are on the same straight line. Remove v and merge e_1 and e_2 into a single edge $e = (u, w)$.

ATTACHVERTEX$(v, e; w)$: Insert edge $e = (v, w)$ and degree-one vertex w inside some region P, where v is a vertex of P.

DETACHVERTEX(v, e): Remove a degree-one vertex v and edge e incident on v.

The above repertory of operations is complete for connected maps. That is, any connected map $\mathcal{M}$ can be constructed "from scratch" using only the above operations.

We make use of the dynamic data structure of Goodrich and Tamassia [59]. Their technique supports two-point shortest-path queries and *ray-shooting* queries, which consist of finding the first edge or vertex of $\mathcal{M}$ hit by a query ray. Their data structure is based on *geodesic triangulation* of each region of $\mathcal{M}$. Given three vertices u, v, and w of a region P (a simple polygon), which occur in that order, the *geodesic triangle* they determine is the union of the shortest paths $\pi_G(u, v)$, $\pi_G(v, w)$ and $\pi_G(w, u)$. A *geodesic triangulation* of P is a decomposition of P's interior into geodesic triangles whose boundaries do not cross. The technique [59] dynamically maintains such triangulations by viewing their dual trees as balanced trees. Also, rotations in these trees can be implemented via a simple "diagonal swapping" operation performed on the corresponding geodesic triangles, and edge insertion and deletion can be implemented on these trees using operations akin to the standard *split* and *splice* operations. Moreover, ray shooting queries are performed by first locating the ray's endpoint and then walking along the ray from geodesic triangle to geodesic triangle until hitting the boundary of some region of $\mathcal{M}$. The two-point shortest path is obtained by locating the two points and then walking from geodesic triangle to geodesic triangle either following a boundary or taking a shortcut through a common tangent [59].

Let n be the current number of vertices in $\mathcal{M}$. Using the data structure of [59], we can perform each of the above update operations as well as ray-shooting and two-point shortest-path queries in $O(\log^2 n)$ time, using $O(n)$ space, where in $O(\log^2 n)$ time we get an implicit representation (a balanced binary tree) and the length of the queried shortest path, and using additional $O(k)$ time to retrieve the k links we get the actual path [59]. Again we enhance this data structure so that associated with the implicit representation of a shortest path π_G, there are two balanced binary trees respectively maintaining the inflection edges and the cusps on π_G in their path order. Moreover, we can extract a portion of π_G via split/splice operations in logarithmic time. Using this data structure to support two-point shortest-path queries as needed by Algorithm *Shortest-Path*, we get a dynamic technique for shortest-path queries between two convex polygons in $\mathcal{M}$.

Theorem 3.2 *Let $\mathcal{M}$ be a connected planar map whose current number of vertices is n. Shortest-path queries between two convex polygons with a total of h vertices that lie in the same region of $\mathcal{M}$ can be performed in time $O(\log h + \log^2 n)$ (plus $O(k)$ to report the k links of the path), using a fully dynamic data structure that uses $O(n)$ space and supports updates of $\mathcal{M}$ in $O(\log^2 n)$ time; all bounds are worst-case.*

Remark. Our update operations are, in the usual dynamic setting, allowed only on $\mathcal{M}$. If R_1 and/or R_2 are also updated, say, by inserting an edge (u, v) between vertices u and v of R_1 and removing the clockwise boundary of R_1 from u to v (or by an inverse operation while preserving the convexity of R_1), we can, of course, first update R_1 and/or R_2 and then re-compute $\pi_G(R_1, R_2)$ by our query algorithm, in $O(\log h + \log^2 n)$ time. An interesting open problem is whether we can support such updates on R_1 and R_2 while maintaining $\pi_G(R_1, R_2)$ in time $O(\text{polylog}(h))$.

3.5 Static Minimum-Link Path Queries

Given two convex polygons R_1 and R_2 with a total of h vertices inside an n-vertex simple polygon P, we want to compute their minimum-link path $\pi_L(R_1, R_2)$. The data structure given by Arkin, Mitchell and Suri [7] supports minimum-link-path queries between two points and between two segments inside P in optimal $O(\log n)$ time, and between two convex polygons R_1 and R_2 in time $O(\log h \log n)$ (plus $O(k)$ if the k links are reported), using $O(n^3)$ space and preprocessing time. We show in this section how to improve the two-polygon queries to optimal $O(\log h + \log n)$ time, using the same data structure.

Let H_G be the geodesic hourglass of R_1 and R_2, with geodesic tangent points $a_1, b_1 \in R_1$ and $a_2, b_2 \in R_2$. As shown in [7], a minimum-link path between the two segments $s_1 = (a_1, b_1)$ and $s_2 = (a_2, b_2)$ gives a desired minimum-link path between R_1 and R_2, i.e., $\pi_L(s_1, s_2) = \pi_L(R_1, R_2)$. Note that $H(s_1, s_2) = H_G$. Recall from Section 3.3.1 that when H_G is open (R_1 and R_2 are mutually visible) Algorithm *Pseudo-Hourglass* returns a visibility link l, which can serve as the desired link-one path $l = \pi_L(R_1, R_2)$. So we look at the case where H_G is closed. As we shall see in Lemma 3.20 (Section 3.6), if hourglass $H(s_1, s_2)$ is closed with penetrations ρ_1 (closer to s_1) and ρ_2 (closer to s_2), then there exists a minimum-link path $\pi_L(s_1, s_2)$ that uses ρ_1 and ρ_2 as the first and last links. This means that $\pi_L(p, q) = \pi_L(s_1, s_2) = \pi_L(R_1, R_2)$, where points p and q are obtained by extending ρ_1 and ρ_2 to intersect R_1 and R_2, respectively. Therefore the two-polygon queries can be reduced to the two-point queries. We summarize this result in the following lemma.

Lemma 3.16 *Let the geodesic hourglass H_G be closed with penetrations ρ_1 (closer to R_1) and ρ_2 (closer to R_2), and the line extensions of ρ_1 and ρ_2 intersect R_1 and R_2 at points p and q, respectively. Then $\pi_L(p, q)$ is a minimum-link path $\pi_L(R_1, R_2)$ between R_1 and R_2.*

We now give the algorithm for computing a minimum-link path $\pi_L(R_1, R_2)$ between R_1 and R_2.

Algorithm Min-Link-Path

1. Perform Algorithm *Pseudo-Hourglass* to decide whether the geodesic hourglass H_G is open (with a visibility link l) or closed (with penetrations ρ_1 (closer to R_1) and ρ_2 (closer to R_2)).
2. If H_G is open, then report $\pi_L(R_1, R_2) = l$, $d_L(R_1, R_2) = 1$ and stop.
3. Otherwise (H_G is closed), extend ρ_1 and ρ_2 to intersect R_1 and R_2 respectively at p and q via binary searches on R_1 and R_2. Compute $\pi_L(p, q)$ (and thus also $d_L(p, q)$) by the algorithm of [7]. Report $\pi_L(R_1, R_2) = \pi_L(p, q)$, $d_L(R_1, R_2) = d_L(p, q)$ and stop.

Lemma 3.17 *The time complexity of Algorithm* Min-Link-Path *is $O(\log h + \log n)$ (plus $O(k)$ if the k links are reported), which is optimal.*

Theorem 3.3 *Let P be a simple polygon with n vertices. There exists a data structure that supports minimum-link-path queries between two convex polygons with a total of h vertices inside P in optimal time $O(\log h + \log n)$ (plus $O(k)$ if the k links of the path are reported), using $O(n^3)$ space and preprocessing time; all bounds are worst-case.*

3.6 Dynamic Minimum-Link Path Queries

In this section we show that the dynamic data structure given in Section 3.4 can also support minimum-link-path queries between two convex polygons in the same region of a connected planar map $\mathcal{M}$. As we have already seen from the last section, we only need to support two-point queries and justify the correctness of Lemma 3.16, which in turn establishes the correctness of Algorithm *Min-Link-Path.*

3.6.1 Basic Properties

Let p and q be two points that lie in the same region P of $\mathcal{M}$, and (p, p') and (q', q) be the first and last links of the shortest path $\pi_G(p, q)$, respectively (see Fig. 3.15). If $\pi_G(p', q')$ is not a monotone chain, there are some cusps $c_1, \cdots, c_i$ such that $\pi_G(p', c_1)$, $\pi_G(c_1, c_2), \cdots, \pi_G(c_i, q')$ are the maximal monotone subchains of $\pi_G(p', q')$. For c_1, we draw a left or right lid l such that l and $\pi_G(p', c_1)$ lie on opposite (left and right) sides of c_1. Let $w_1 = (p, u)$ be the extension of (p, p'), where u is obtained by ray shooting (see Fig. 3.15). We consider the subregion P' of P delimited by w_1 and l. For each cusp v of P', we draw both lids of v if they do not intersect with $\pi_G(p', c_1)$, otherwise we draw left or right lid of v that does not intersect with $\pi_G(p', c_1)$. Then P' is partitioned into a collection of monotone polygons, among which we denote by $sleeve(w_1)$ the monotone sleeve that uses w_1 as its boundary and contains $\pi_G(p', c_1)$ (see Fig. 3.15). Excluding segment w_1, the boundary of $sleeve(w_1)$ consists of left and right monotone chains C_1 and C_2. We say that a line t is an *internal common tangent* of $sleeve(w_1)$ if t is locally tangent to two vertices a and b respectively on C_1 and C_2 (if t goes through u, then u is also considered as a tangent point, and similarly for p'). If t intersects with w_1 and a is closer to w_1 than b, we call t a *left tangent* of $sleeve(w_1)$; a *right tangent* is defined similarly.

Suppose that t' and t'' are two left (or right) tangents of $sleeve(w_1)$. Let $\pi'_G(p, q)$ be the set of points on $\pi_G(p, q)$ each of which is visible from some point of t', and v' be the point of $\pi'_G(p, q)$ that is closest to q; v'' is defined similarly with respect to t''. We say that t' *extends farther* than t'' if v' is closer to q than v'' on $\pi_G(p, q)$. Among the left tangents of $sleeve(w_1)$, the one that extends the farthest is called the *maximal left tangent* of $sleeve(w_1)$; similarly for the definition of *maximal right tangent.* By the definitions of $\pi_L(p, q)$ and of window partition, we have the following preliminary algorithm for computing $\pi_L(p, q)$, when the shortest path $\pi_G(p, q)$ is given (see Fig. 3.15).

Algorithm Prelim

1. If $\pi_G(p, q)$ has only one link then report $\pi_L(p, q) = (p, q)$ and stop; else if the extensions of the first and last links of $\pi_G(p, q)$ meet at some point v, then report $\pi_L(p, q) = (p, v, q)$ and stop.
2. Otherwise, perform the following steps.
   (a) Perform ray shooting to extend the first link of $\pi_G(p, q)$; this gives the first window w_1.
   (b) From w_1, compute the monotone sleeve $sleeve(w_1)$ as described above, and compute the maximal left tangent t_1 of $sleeve(w_1)$ and the maximal right tangent t_2. Choose t

from t_1 and t_2 as the one that extends farther. The second window w_2 is (p_1, v_2), where $p_1 = w_1 \cap t$, and v_2 is obtained by performing ray shooting from p_1 along t toward q.

(c) Repeat step 2b to compute subsequent windows, until the current window intersects with the extension of the last link of $\pi_G(p,q)$, which is the last window w_k.

(d) Let $p_i = w_i \cap w_{i+1}$. Report $\pi_L(p,q) = (p, p_1, \cdots, p_{k-1}, q)$ and stop.

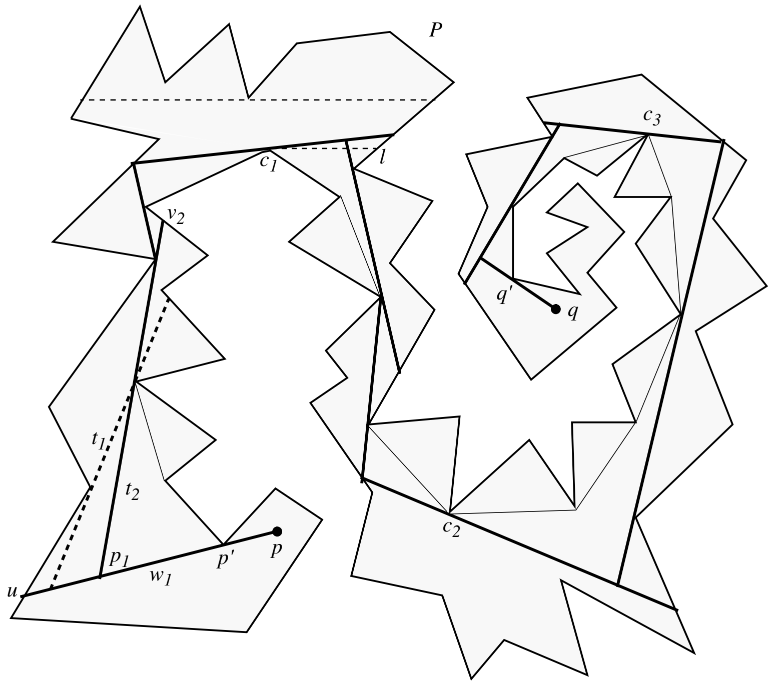


Figure 3.15: Computing $\pi_L(p,q)$ by Algorithm *Prelim*: the window w_2 following w_1 is chosen to be t_2 since it extends farther than t_1, and so on.

Let $e_1, \cdots, e_j$ be the inflection edges of $\pi_G(p,q)$. Then $e_1, \cdots, e_j$ partition $\pi_G(p,q)$ into subchains that are always left-turning or always right-turning, namely, into *inward convex* subchains (see Fig. 3.16). It is shown that every inflection edge $e \in \pi_G(p,q)$ must be contained in $\pi_L(p,q)$ [7, 18, 55]. Hence, extending each inflection edge of $\pi_G(p,q)$ by ray shooting on both sides, together with the extensions of the first and last links of $\pi_G(p,q)$ (where the first link extends towards q and the last toward p), we have fixed windows $W_1, \cdots, W_{j+2}$ (see Fig. 3.16). Now the task is how to connect consecutive fixed windows. In particular, each W_i has a portion $(u,v) \in \pi_G(p,q)$, with u closer to p than v in $\pi_G(p,q)$. Let the endpoints of W_i be u' and v' such that $W_i = (u', u, v, v')$ (note that $u' = u = p$ if $i = 1$ and $v' = v = q$ if $i = j+2$). We call (u', u) the *front* of W_i and (v, v') the *rear* of W_i. We want to connect the rear of W_i with the front of W_{i+1} for each $i = 1, \cdots, j+1$.

Lemma 3.18 *Let W_i and W_{i+1} be consecutive fixed windows, W the front of W_{i+1}, and w the rear of W_i or a window between the rear of W_i and the front of W_{i+1} as computed by Algorithm* Prelim. *If the hourglass $H(w, W)$ is closed, then the window w' following w is the penetration of funnel $F(w)$.*

Proof: For any local portion of P, the boundary of P consists of two bounding chains C_1 and C_2. Let $w = (a_1, b_1)$ and $W = (a_2, b_2)$, where a_1 and a_2 are on $\pi_G(p,q)$ (see Fig. 3.17). Then $\pi_G(a_1, a_2)$ is a convex hull inside P of a bounding chain, say C_1, of P. By Algorithm *Prelim*, there are two possible candidates for window w': the penetration of $F(w)$ and some internal common

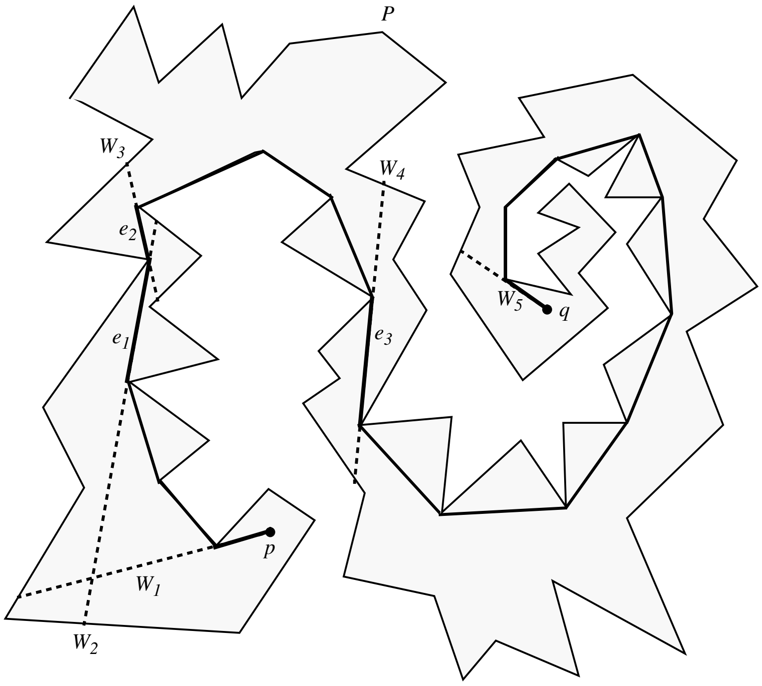


Figure 3.16: The shortest path $\pi_G(p, q)$ is partitioned by inflection edges e_1, e_2 and e_3. The fixed windows $W_1, \cdots, W_5$ are obtained by extending the inflection edges as well as the first and last links of $\pi_G(p, q)$.

tangent t intersecting with w. Let p_1 be the apex of funnel $F(w)$. Note that $p_1 \in C_1$ and thus the other tangent point of the penetration lies on C_2. Then t must be tangent to two vertices v_1 and v_2 with $v_1 \in \pi_G(a_1, p_1)$ and $v_2 \in C_2$, where v_1 is closer to w than v_2 when walking along t. While extending towards q, the penetration has a slope closer to $\pi_G(a_1, a_2)$ than t, i.e., anything blocking the penetration certainly blocks t (see Fig. 3.17). Thus the penetration extends farther than t towards q and is chosen as the next window w'. $\square$

3.6.2 Two Point Queries

The algorithm for computing $\pi_L(p, q)$ between two query points p and q is as follows.

Algorithm Point-Query

1. Compute the shortest path $\pi_G(p, q)$. If $\pi_G(p, q)$ has only one link, then report $\pi_L(p, q) = (p, q)$, $d_L(p, q) = 1$ and stop.
2. Else, perform ray-shooting queries to extend the first link of $\pi_G(p, q)$ in the direction toward q, and the last link of $\pi_G(p, q)$ in the direction toward p. If they intersect with each other at some point v, then report $\pi_L(p, q) = (p, v, q)$, $d_L(p, q) = 2$ (if p, v and q are collinear then $d_L(p, q) = 1$) and stop. Otherwise, also extend each inflection edge of $\pi_G(p, q)$ in both directions; together with the extensions of the first and last links of $\pi_G(p, q)$, this gives the fixed windows $W_1, \cdots, W_j$.
3. For each pair of consecutive fixed windows W_i and W_{i+1} that do not intersect with each other, repeat step 4 to compute the intermediate windows connecting the rear of W_i and the front of W_{i+1}.

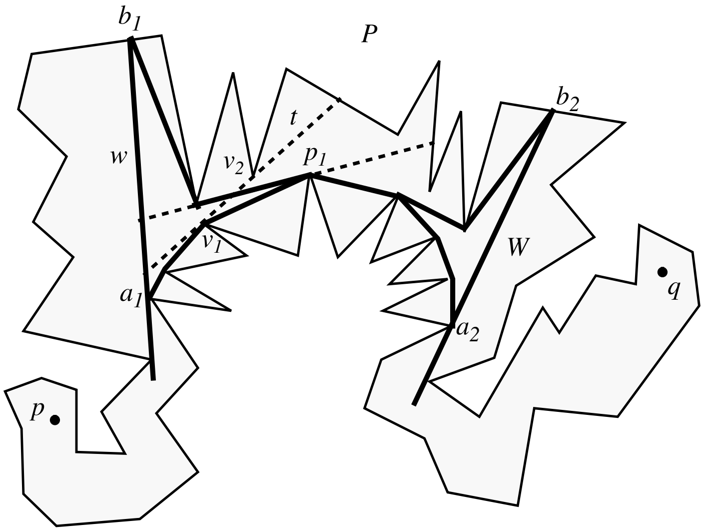


Figure 3.17: Proof of Lemma 3.18.

4. Initially, let w be the rear of W_i. Let $W = (a_2, b_2)$ be the front of W_{i+1} with $a_2 \in \pi_G(p, q)$.
   (a) Assume that $w = (a_1, b_1)$ with a_1 on $\pi_G(p, q)$. Compute the shortest path $\pi_G(b_1, b_2)$.
   (b) If there is no inflection edge in $\pi_G(b_1, b_2)$, then $H(w, W)$ is an open hourglass. Compute an internal common tangent t of the two inward convex chains $\pi_G(a_1, a_2)$ and $\pi_G(b_1, b_2)$. Note that t connects w and W. Set t to be the window following w and exit step 4.
   (c) Else (there are inflection edges in $\pi_G(b_1, b_2)$) let ρ be the first inflection edge of $\pi_G(b_1, b_2)$, then $H(w, W)$ is a closed hourglass: one endpoint p_1 of ρ is an apex and ρ is the penetration of funnel $F(w)$. Extend ρ in the direction toward b_2 by ray shooting, which hits the boundary of P at some point u; also intersect line ρ with w at some point v. Set (v, u) to be the window following w. Note that p_1 is in (v, u) and is a vertex of P on $\pi_G(p, q)$. Set $w := (p_1, u)$ and go to step 4(a).
5. Now there are windows $w_1, \cdots, w_k$ connecting p and q. Let $v_i = w_i \cap w_{i+1}$, $i = 1, \cdots, k-1$. Report $\pi_L(p, q) = (p, v_1, \cdots, v_{k-1}, q)$, $d_L(p, q) = k$ and stop.

It is easily seen that we perform $O(1)$ ray-shooting and shortest-path queries to compute each link of $\pi_L(p, q)$. Therefore, we have:

Lemma 3.19 *The time complexity of Algorithm* Point-Query *is $O(k \log^2 n)$, where k is the number of links in the reported path.*

Now we are ready to give the following lemma, which justifies the correctness of Lemma 3.16 and thus also Algorithm *Min-Link-Path* given in Section 3.5.

Lemma 3.20 *Suppose that two segments s_1 and s_2 inside a polygonal region P are not mutually visible, i.e., the hourglass $H(s_1, s_2)$ (containing funnels $F(s_1)$ and $F(s_2)$) is closed. Let ρ_1 be the penetration of $F(s_1)$ and ρ_2 the penetration of $F(s_2)$. Then there exists a minimum-link path $\pi_L(s_1, s_2)$ between s_1 and s_2 that uses ρ_1 and ρ_2 as the first and last links.*

Proof: To compute $\pi_L(s_1, s_2)$, we can view s_1 and s_2 as "fictitious windows" and apply the method for two-point queries. Let p_1 and p_2 be the apices of $F(s_1)$ and $F(s_2)$, respectively. If $p_1 = p_2$ then the lemma holds trivially. Otherwise, let t be the first internal common tangent in $\pi_G(p_1, p_2)$, and W be the extension of t. If there is no such t, then let $W = s_2$. Since the shortest path from any point of s_1 to any point of s_2 must go through p_1 and p_2, s_1 and W serve as *consecutive fixed* *windows* in $\pi_L(s_1, s_2)$. If $H(s_1, W)$ is an open hourglass, then the penetration ρ_1 is an internal common tangent connecting fixed windows s_1 and W, and thus is chosen as the window following s_1. If $H(s_1, W)$ is closed, then as computed by Lemma 3.18, ρ_1 is the window following s_1. In either case, ρ_1 is chosen as the first link of $\pi_L(s_1, s_2)$. Similarly ρ_2 "extends the farthest" from s_2 towards s_1. Suppose that $\pi_L(s_1, s_2)$ so computed does not use ρ_2 as the last link, and w and w' are the last two windows of $\pi_L(s_1, s_2)$. Since ρ_2 extends no worse than the last link w', ρ_2 can also catch w, i.e., replacing w' with ρ_2 still gives a minimum-link path between s_1 and s_2. □

Using Algorithm *Point-Query* to support two-point queries as needed by Algorithm *Min-Link-Path*, we are now able to perform two-polygon queries.

Theorem 3.4 *Let $\mathcal{M}$ be a connected planar map whose current number of vertices is n. Minimum-link-path queries between two convex polygons with a total of h vertices that lie in the same region of $\mathcal{M}$ can be performed in time $O(\log h + k \log^2 n)$ (where k is the number of links in the reported path), using a fully dynamic data structure that uses $O(n)$ space and supports updates of $\mathcal{M}$ in $O(\log^2 n)$ time; all bounds are worst-case.*

Chapter 4

External-Memory Graph Algorithms

4.1 Introduction

Graph-theoretic problems arise in many large-scale computations, including those common in object-oriented and deductive databases, VLSI design and simulation programs, and geographic information systems. Often, these problems are too large to fit into main memory, so the input/output (I/O) communication between main memory and external memory (such as disks) becomes a major bottleneck. The significance of this bottleneck is increasing as internal computation gets faster, and especially as parallel computing gains popularity.

Unfortunately, the overwhelming majority of the vast literature on graph algorithms ignores this bottleneck and simply assumes that data completely fits in main memory (as in the usual RAM model). Direct applications of the techniques used in these algorithms often do not yield efficient external-memory algorithms. Our goal is to present a collection of new techniques that take the I/O bottleneck into account and lead to the design and analysis of I/O-efficient graph algorithms.

4.1.1 The Computational Model

In contrast to solid state random-access memory, disks have extremely long access times. In order to amortize this access time over a large amount of data, typical disks read or write large blocks of contiguous data at once. An increasingly popular approach to further increase the throughput of I/O systems is to use a number of independent devices in parallel. In order to model the behavior of I/O systems, we use the following parameters:

$$\begin{aligned} N &= \text{\# of items in the problem instance} \\ M &= \text{\# of items that can fit into main memory} \\ B &= \text{\# of items per disk block} \\ D &= \text{\# of disks in the system} \end{aligned}$$

where $M < N$ and $1 \ll DB \leq M/2$. Here we deal with problems defined on graphs, so we also define

$$\begin{aligned} V &= \text{\# of vertices in the input graph} \\ E &= \text{\# of edges in the input graph.} \end{aligned}$$

Note that $N = V + E$. We assume that $E \geq V$. Depending on the size of the data items, typical values for workstations and file servers in production today are on the order of $M = 10^6$ to $M = 10^8$

and $B = 10^3$. Values of D can range up to 10^2 in current systems. Large-scale problem instances can be in the range $N = 10^{10}$ to $N = 10^{12}$.

Our measure of performance for external-memory algorithms is the standard notion of I/O complexity [122]. We define an *input/output operation* (or simply *I/O* for short) to be the process of simultaneously reading or writing D blocks of data, to or from each of the disks. The total amount of data transferred in an I/O is thus DB items. The I/O complexity of an algorithm is simply the number of I/Os it performs. For example, reading all of the input data requires N/DB I/Os, since we can read DB items in a single I/O.

Our algorithms make extensive use of two fundamental primitives, *scanning* and *sorting*. We therefore introduce the following shorthand notation to represent the I/O complexity of each of these primitives:

$$scan(x) = \frac{x}{DB},$$

which represents the number of I/Os needed to read x items, and

$$sort(x) = \frac{x}{DB} \log_{M/B} \frac{x}{B},$$

which is proportional to the optimal number of I/Os needed to sort x items [87].

4.1.2 Previous Work

Early work on external-memory algorithms concentrated largely on fundamental problems such as sorting, matrix multiplication, and FFT [2, 87, 122]. The main focus of this early work was therefore directed at problems that involved permutation at a basic level. Indeed, just the problem of implementing various classes of permutation has been a central theme in external-memory I/O research [2, 36, 37, 38, 122].

More recently, external-memory research has moved towards solving geometric and graph problems. For geometric problems, Goodrich *et al.* [60] study a number of problems in computational geometry and develop several paradigms for I/O-optimal geometric computations. Further results in this area have been obtained in [51, 125]. Also, Kanellakis *et al.* [65] and Ramaswamy and Subramanian [98, 106] give efficient data structures for performing range searching in external memory. Very recently, a new data structure called *buffer tree* and its applications are given in [5, 6], and an external-memory version of the directed topology tree ([52]) called *topology B-tree* is given in [17].

There has also been some work on selected graph problems, including the investigations by Ullman and Yannakakis [117] on transitive closure computations. This work, however, restricts its attention to problem instances where the set of vertices fits into main memory but the set of edges does not. Vishkin [121] uses PRAM simulation to facilitate prefetching for various problems, but without taking blocking issues into account. Also worth noting is recent work [58] on some graph traversal problems; this work primarily addresses the problem of storing graphs, however, not in performing specific computations on them. Related work [50] proposes a framework for studying memory management problems for maintaining connectivity information and paths on graphs. Other than these papers, we do not know of any previous work on I/O-efficient graph algorithms.

4.1.3 Our Results in This Chapter

We give a number of techniques for solving a host of graph problems in external memory. They are based on the following central methods:

- *PRAM simulation.* We give methods for efficiently simulating PRAM computations in external memory. Our simulation techniques explore the different flavors of locality in PRAM and in external-memory computations, and also take advantage of the "geometrically decreasing size" property common in many PRAM algorithms.
- *Deterministic 3-coloring of a cycle*—a problem central to list ranking and symmetry breaking in graph problems. Our methods for solving it go beyond simple PRAM simulation, and may be of independent interest. In particular, we give techniques to update scattered successor and predecessor colors as needed after re-coloring a group of nodes without sorting or scanning the entire list.

Combining these techniques, we are able to derive an I/O-optimal algorithm for list ranking, which is the most fundamental subroutine used in PRAM computations. We then apply our PRAM simulation and list-ranking algorithms to a wide variety of fundamental problems, including Euler-tour computation, expression-tree evaluation, least-common ancestors, connected and biconnected components, minimum spanning forest, ear decomposition, topological sorting, reachability, graph drawing, and visibility representation. For all these problems considered, we give the first I/O-efficient algorithms, most of them being I/O-optimal.

4.1.4 Organization of the Chapter

The rest of this chapter is organized as follows. In Section 4.2 we review the I/O lower bounds related to our problems. Sections 4.3 and 4.4 are respectively devoted to PRAM simulation and list ranking. Finally, we give applications of the developed techniques in Section 4.5.

4.2 Review of Lower Bounds: Linear vs. Permutation Times

In order to derive lower bounds for the number of I/Os required to solve a given problem it is often useful to look at the complexity of the problem in terms of the permutations that may have to be performed to solve it. In an ordinary RAM, any known permutation of N items can be produced in $O(N)$ time. In an N processor PRAM, it can be done in constant time. In both cases, the work is $O(N)$, which is no more than it would take us to examine all the input. In external memory, however, it is not generally possible to perform arbitrary permutations in a linear number ($O(scan(N))$) of I/Os. Instead, it is well-known that $\Theta(perm(N))$ I/Os are required in the worst case [2, 122] where

$$perm(N) = \min\left\{\frac{N}{D}, sort(N)\right\}.$$

When M or B is extremely small, $N/D = O(B \cdot scan(N))$ may be smaller than $sort(N)$. In the case where B and D are constants, the model is reduced to an ordinary RAM, and, as expected, permutation can be performed in linear time. However, for typical values in real I/O systems, the $sort(N)$ term is smaller than the N/D term. If we consider a machine with block size $B = 10^4$ and main memory size $M = 10^8$, for example, then $sort(N) \geq N/D$ only when $N \geq B \cdot (\frac{M}{B})^B = 10^{40{,}004}$, which is certainly not the case in any real-world applications.

The lower bound $\Omega(perm(N))$ holds even in some important cases when we are not required to perform all $N!$ possible permutations. In [118] (also appearing in [25]) a problem called *proximate neighbors*, which is a significantly-restricted form of permutation, is used to derive a number of non-trivial lower bounds. The proximate neighbors problem is defined as follows: Initially, we have N items in external memory, each with a key that is a positive integer $k \leq N/2$. Exactly two items have each possible key value k. The problem is to permute the items such that, for every k, both items with key value k are in the same block.

Lemma 4.1 ([25, 118]) *Solving the proximate neighbors problem requires $\Omega(perm(N))$ I/Os in the worst case.*

Corollary 4.1 ([25, 118]) *The following problems all have an I/O lower bound of $\Omega(perm(N))$: list ranking, Euler tours, expression-tree evaluation, and connected components in sparse graphs ($E = O(V)$).*

Upper bounds of $O(sort(N))$ for these problems are shown in Sections 4.4 and 4.5, giving optimal results whenever $perm(N) = \Theta(sort(N))$. As was mentioned above, this covers all practical I/O systems. The key to designing algorithms to match the lower bound of Corollary 4.1 is the fact that comparison-based sorting can also be performed in $\Theta(sort(N))$ I/Os. This suggests that in order to optimally solve a problem covered by Corollary 4.1 we can use sorting as a subroutine. Note that this strategy does not work in the ordinary RAM model, where the sorting takes $\Omega(n \log n)$ time, while many problems requiring arbitrary permutations can be solved in linear time.

4.3 PRAM Simulation

An intuitive way (but with a weaker computational power) to consider the external-memory computation is to view the main memory as a *local window*, and the contiguous data blocks on disks as a *tape*: first we put the window somewhere on the tape, read in what can be seen from the window, perform some computations and write them back, and then slide the window and repeat the process until the computing task is finished. Under this computational scheme, it is desirable that the computations performed be independent to one another (*goal 1*), and that the data needed be as localized as possible (*goal 2*). An ideal candidate to be used for achieving these goals is the paradigm of PRAM computations.

In this section, we present some simple yet general techniques for designing I/O efficient algorithms based on the simulation of PRAM algorithms. Observe that PRAM techniques already fulfill goal 1, and to achieve goal 2 we explore the different flavors of locality in PRAM and in external-memory computations. We also take advantage of the "geometrically decreasing size" property common in many PRAM algorithms to obtain I/O-optimal methods. In addition, since we do not need to simulate inactive processors, the PRAM algorithms we simulate are those described at a higher level (called the *work-time presentation framework* [64]) which ignore the more complicated processor scheduling issues, and thus our resulting I/O algorithms are simple and practical.

We show in subsequent sections how to combine these techniques with more sophisticated strategies to design efficient external-memory algorithms for a number of graph problems. Related work on simulating PRAM computations in external memory is done by Cormen [36]. The use of PRAM simulation for prefetching, without the important consideration of blocking, is explored by Vishkin [121].

4.3.1 Generic Simulation of a PRAM Algorithm

We begin by considering how to simulate a PRAM algorithm A. Under the work-time presentation framework, the *work* performed by A is defined to be the total number of operations used [64]. It is easy to see that a single step of A with $O(N)$ work which does not involve any pointer reference can be simulated by just scanning through the $O(N)$ data items and performing the operations, using $O(scan(N))$ I/Os.

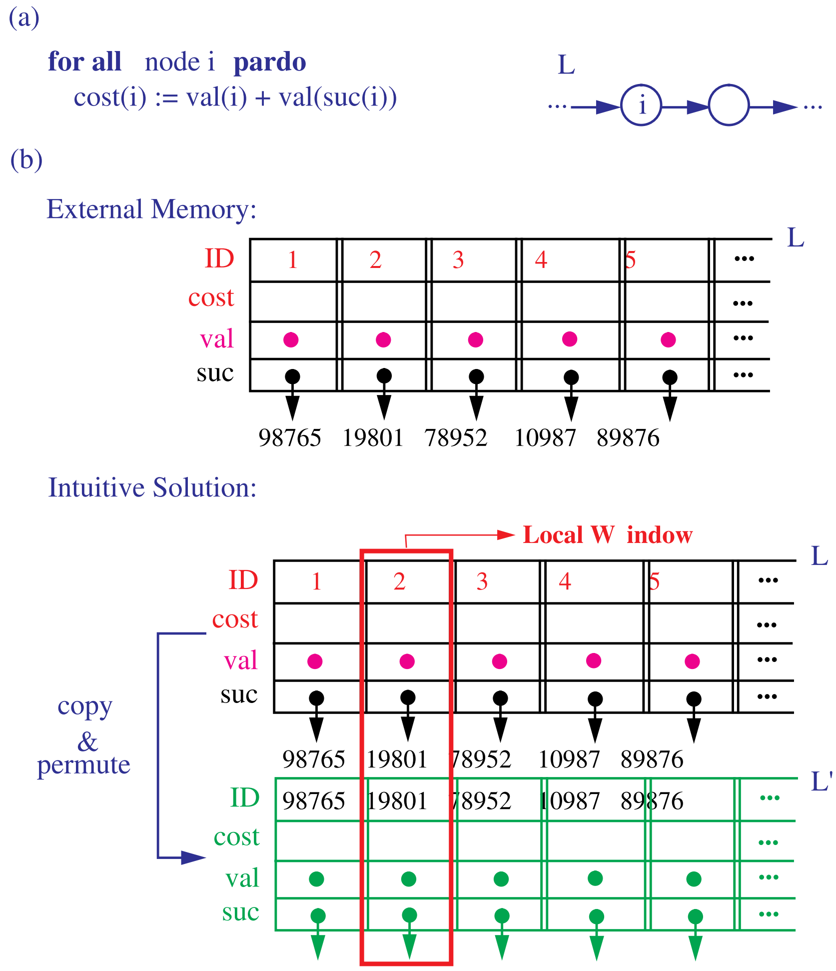


Figure 4.1: Example of simulating an $O(N)$-work single step of a PRAM algorithm A: (a) the step of A to be simulated; (b) intuitive simulation idea.

A more interesting case occurs when the PRAM step involves pointer references. Without loss of generality, we assume that each PRAM step does not have indirect memory references, since they can be removed by expanding the step into $O(1)$ steps. An example of a typical PRAM step using $O(N)$ operations is shown in Fig. 4.1(a). While this is a local computation for PRAM, it is not the case for external-memory computation, since the successor of node i, for each i being processed, may not currently reside in the main memory. If we just follow these pointers, then we may have to use $O(N)$ I/Os in the worst case. Our *intuitive solution* is simple: To process the node list L on disk (sorted by node ID), we copy L into another list L', "permute" L' by node ID according to the ordering given by the successor IDs of L, and align L and L'. Now the successor of each node in L is available, and we can perform the operations and write the results on L easily by a linear scan. This process is shown in Fig. 4.1(b). Notice that this intuitive solution needs to be refined, since the above "permute" step is not directly feasible. To make it work, we actually do the following. First we copy L into L', which is already sorted by node ID. Then we sort L by successor ID and align L on top of L' so that the successor of each node in L is available. We then perform the operations and write the results on L by a linear scan, and finally we sort L back to its original ordering by node ID. Since we use only $O(1)$ sorts and scans, this takes $O(sort(N))$ I/Os. We summarize this result in the following theorem.

Theorem 4.1 *Let A be a PRAM algorithm. Then a single step of A using $O(N)$ operations can be simulated in $O(sort(N))$ I/Os; if there is no pointer references, then the step can be simulated in $O(scan(N))$ I/Os.*

To simulate an entire algorithm, we merely have to simulate all of its steps.

Corollary 4.2 *Let A be a PRAM algorithm that runs in T steps, each using $O(N)$ operations. Then A can be simulated in $O(T \cdot sort(N))$ I/Os.*

4.3.2 Reduced Work Simulation for Geometrically Decreasing Computations

Many simple PRAM algorithms can be designed so as to have a "geometrically decreasing size" property, in that after each iteration or recursion the problem size (and thus the number of operations used) is reduced by a constant factor. This is a general algorithmic technique commonly used to derive work-efficient PRAM algorithms. As pointed out by [118], this technique is especially useful for us, since our technique of Theorem 4.1 already uses $O(sort(N))$ rather than $O(1)$ I/Os to simulate a single PRAM step. Without this geometrically decreasing size technique, it is almost impossible to simulate any powerful PRAM algorithms (i.e., with iterations and/or recursions) within an $O(sort(N))$ I/O bound optimal for many problems. We explicitly state the application of this technique to our PRAM simulation in the following theorem.

Theorem 4.2 *Let A be a PRAM algorithm that solves a problem of size N in $O(\log N)$ stages, each in T steps. The problem size is reduced by a constant factor after each stage, and the number of operations used in each single step is proportional to the problem size at that stage. Then A can be simulated in external memory in $O(T \cdot sort(N))$ I/O operations.*

Proof: The first stage consists of T steps, each of which can, by Theorem 4.1, be simulated in $O(sort(N))$ I/Os. Thus for some constant $0 < \alpha < 1$ the recurrence

$$\mathcal{I}(N) = O(T \cdot sort(N)) + \mathcal{I}(\alpha N)$$

characterizes the number of I/Os needed to simulate the algorithm, which is $O(T \cdot sort(N))$. □

We can make Theorem 4.2 even more powerful by applying the idea to the case of nested iterations/recursions, as will be presented in subsequent sections.

4.4 List Ranking

List ranking is the most fundamental subroutine used in PRAM computations. Deriving an I/O-efficient algorithm for list ranking will, combined with our PRAM simulation techniques, enable us to obtain efficient external-memory algorithms for a wide variety of important problems.

In this section, we demonstrate how the lower bound results of Section 4.2 and the PRAM simulation techniques of Section 4.3 can be put together to produce an I/O-optimal algorithm for list ranking. In particular, we give in Section 4.4.3 an efficient deterministic algorithm for 3-coloring of a cycle, which is central to list ranking and symmetry breaking in graph problems and our methods for solving it go beyond simple PRAM simulation.

The list ranking problem is as follows. We are given an N-node linked list L stored in external memory as an (unordered) sequence of nodes, each with a pointer *next* to the successor node in the list. Our goal is to determine, for each node v of L, the *rank* of v, which we denote *rank*(v) and define as the number of links from v to the end of the list. We assume that there is a dummy node ∞ at the end of the list, and thus the rank of the last node in the list is 1. We present algorithms that use an optimal $\Theta(sort(N))$ I/O operations. The lower bound for the problem comes from Corollary 4.1.

4.4.1 An Algorithmic Framework for List Ranking

Our algorithmic framework is adapted from the work of Anderson and Miller [4]. It has also been used by Cole and Vishkin [33], who developed a deterministic version of Anderson and Miller's randomized algorithm.

Initially, we assign $rank(v) = 1$ for each node v in list L. This can be done in $O(scan(N))$ I/Os. We then proceed recursively. First, we produce an independent set of $\Theta(N)$ nodes. The details of how this independent set is produced are what separate our algorithms from each other. Once we have a large independent set S, we use $O(1)$ sorts and scans to *bridge* each node v in the set, as described in [4]. We then recursively solve the problem on the remaining nodes. Finally, we use $O(1)$ sorts and scans to re-integrate the nodes in S into the final solution.

In order to analyze the I/O-complexity of an algorithm of the type just described, we first note that once the independent set has been produced, the algorithm uses $O(sort(N))$ I/Os and solves a single recursive instance of the problem. If the independent set can also be found in $O(sort(N))$ I/Os, then the total number of I/Os done in the nonrecursive parts of the algorithm is also $O(sort(N))$.

Since $\Theta(N)$ nodes are bridged out before recursion, the size of the recursive problem we are left with is at most a constant fraction of the size of our original problem. Thus, according to Theorem 4.2, the I/O-complexity of our overall algorithm is $O(sort(N))$. All that remains is to demonstrate how an independent set of size $\Theta(N)$ can be produced in $O(sort(N))$ I/Os.

4.4.2 Randomized Independent Set Construction

The simplest way to produce a large independent set is a randomized approach based on that first proposed by Anderson and Miller [4]. We scan along the input, flipping a fair coin for each vertex v. We then make two copies of the input, sorting one by vertex and the other by successor. Scanning down these two sorted lists in step, we produce an independent set consisting of those vertices whose coins turned up heads but whose successors coins turned up tails. The expected size of the independent set generated this way is $(N-1)/4$.

4.4.3 Deterministic Independent Set Construction via 3-Coloring

Our deterministic approach relies on the fact that the problem of finding an independent set of size $\Omega(N)$ in an N-node list L can be reduced to the problem of finding a 3-coloring of the list. We equate the independent set with the $\Omega(N)$ nodes colored by the most popular of the three colors.

In this section, we describe an external-memory algorithm for 3-coloring L that performs $O(sort(N))$ I/O operations. We make the simplifying assumption here (and also in the next section) that the block size B satisfies $B = O(N/\log^{(t)} N)$ for some fixed integer $t > 0$.[1] This assumption is clearly non-restrictive in practice. Furthermore, for simplicity, we restrict the discussion to the $D = 1$ case of one disk. The load balancing issues that arise with multiple disks are handled with balancing techniques akin to [86, 122].

The 3-coloring algorithm consists of three phases. Phase 1 sets up an N-coloring. Each remaining phase reduces the number of colors used. Colors and node IDs are represented by integers.

1. In this phase we construct an initial N-coloring of L by assigning a distinct color in the range $[0, \cdots, N-1]$ to each node. This phase takes $O(scan(N))$ I/Os.
2. Recall that $B = O(N/\log^{(t)} N)$ for some fixed integer $t > 0$. In this phase we produce a $(\log^{(t+1)} N)$-coloring.

[1] The notation $\log^{(k)} N$ is defined recursively as follows: $\log^{(1)} N = \log N$, and $\log^{(i+1)} N = \log\log^{(i)} N$, for $i \geq 1$.

(a) Construct a $(\log^{(t+1)} N)$-coloring of L with values in the range $[0, .. \log^{(t+1)} N - 1]$. This step can be done by performing $t+1$ times an external-memory variation of deterministic coin tossing (obtained by simulating the PRAM algorithm of [33]), and takes $O((t+1) \cdot \mathit{sort}(N))$ I/Os. We denote with N_i the number of nodes with color i.

(b) Determine the predecessor and successor of each node of L, and create two copies of L, denoted L_1 and L_2, in external memory, where the nodes of L_1 store also the predecessor node, and the nodes of L_2 store also the successor node. The nodes of L_1 are stored sorted first by color and then by predecessor color. The nodes of L_2 are stored sorted first by color and then by successor color. By sorting, this step can be done with $O(\mathit{sort}(N))$ I/Os.

(c) Set up a table C^1 (C^2) of $(\log^{(t+1)} N)^2$ pointers to external-memory blocks, where $C^1_{i,j}$ ($C^2_{i,j}$) points to the first block of L_1 (L_2) storing a node with color i and predecessor (successor) color j. This step uses $O(\mathit{scan}(N) + (\log^{(t+1)} N)^2)$ I/Os.

The total number of I/Os performed in this phase is $O(t \cdot \mathit{sort}(N) + (\log^{(t+1)} N)^2)$.

3. In the final phase, for each $i = 3, .., \log^{(t+1)} N - 1$, we re-color the nodes with color i by assigning them a new color in the range $[0, 1, 2]$. This is done as follows:

(a) Determine the predecessor and successor of each node with color i. This step is carried out sorting by ID the nodes in L_1 and L_2 with color i, and uses $O(\mathit{sort}(N_i))$ I/Os.

(b) Re-color each node of color i with the smallest color in the range $[0, 1, 2]$ that is not currently assigned to its predecessor or successor. This step uses $O(\mathit{scan}(N_i))$ I/Os, where N_i is the number of vertices with color i (from the previous phase).

(c) Update the nodes of L_1 (L_2) with color $> i$ whose predecessor (successor) has been recolored. This is done by accessing the affected nodes of L_1 and L_2 by means of pointers $C^1_{i,j}$ and $C^2_{i,j}$, for $j > i$, and $O(1)$ sorts. This step uses $O(\log^{(t+1)} N + \mathit{sort}(N_i))$ I/Os.

The total number of I/Os performed in this phase is

$$\sum_{i=0}^{\log^{(t+1)} N - 1} O(\log^{(t+1)} N + \mathit{sort}(N_i)) = O(\mathit{sort}(N) + (\log^{(t+1)} N)^2).$$

The overall time complexity of the 3-coloring algorithm is thus $O(t \cdot \mathit{sort}(N) + (\log^{(t+1)} N)^2)$. Since t is a constant and $B = O(N/\log^{(t)} N)$, we get the following time bound:

Lemma 4.2 *The N nodes of a list L can be 3-colored with $O(\mathit{sort}(N))$ I/O operations.*

Recalling the algorithmic framework for list ranking of Section 4.4.1, we obtain the following result:

Theorem 4.3 *The N nodes of a list L can be ranked with optimal $O(\mathit{sort}(N))$ I/O operations.*

4.5 Applications

In this section we show that the techniques presented in Sections 4.3–4.4 can be used to solve a variety of fundamental problems on trees, undirected graphs and planar digraphs. We believe that many more problems are amenable to these techniques.

4.5.1 Tree Algorithms

Our results on tree problems are summarized in Table 4.1. Lower bounds are given in Corollary 4.1.

Problem	Notes	Lower Bound	Upper Bound
Euler Tour		$\Omega(sort(N))$	$O(sort(N))$
Expression Tree Evaluation	Bounded Degree Operators	$\Omega(sort(N))$	$O(sort(N))$
Least Common Ancestor	K Queries		$O((1+K/N)sort(N))$

Table 4.1 I/O-efficient algorithms for problems on trees. The problem size is $N = V = E + 1$.

Euler Tours Let T be a tree, and T' a directed graph obtained from T by replacing each edge (i, j) of T with two arcs $\langle i, j\rangle$ and $\langle j, i\rangle$. The *Euler Tour* of T is a directed circuit that traverses each arc of T' exactly once. By PRAM simulation, we obtain:

Theorem 4.4 *The Euler Tour of an N-node tree can be computed using optimal $O(sort(N))$ I/O operations.*

Expression Tree Evaluation A *general expression tree* T is a bounded-degree tree whose internal nodes are labeled by arbitrary functions with scalar arguments computable in $O(1)$ time. The *value* of an internal node is the result of applying its function to the values of its children. The value of each leaf is a scalar defined in the input. The *expression tree evaluation problem* is to determine the values at all internal nodes of T.

Theorem 4.5 *An N-node general expression tree can be evaluated with optimal $O(sort(N))$ I/Os.*

Proof: We give two algorithms; the first one is based on [62] (also appearing in the conference version [25] of this chapter) and the second one is new.

The first algorithm works as follows. By using Euler Tour and list ranking twice, we compute the depth and also the preorder numbering of each node. We then sort all nodes into a list first by nonincreasing depths and then by increasing preorder numbers. Observe that the nodes are grouped by levels, bottom up, and the nodes in the same level appear left to right. We then keep two pointers, one to the next node to be computed and the other to the next input value needed. The two pointers move sequentially through the list of nodes, so all of the nodes in the tree can be computed with $O(scan(N))$ additional I/Os.

Our alternative algorithm makes use of the external-memory priority queue based on the buffer tree [5], where *Insert* and *DeleteMin* operations can each be performed using amortized $O(\frac{1}{DB}\log_{M/B}\frac{N}{B})$ I/Os [5]. We first compute the depth $d(v)$ of each node v by Euler Tour and list ranking, and then sort all nodes into a list L first by nonincreasing depths and then by node IDs. The nodes in L are computed sequentially. As long as we obtain the value of a node v, we generate an item consisting of the value, the parent depth ($= d(v) - 1$), and the parent ID of v. The item is inserted into a priority queue which maintains all such items ordered first by parent depth and then by parent ID. In this way, all children of the current node in L have already been computed and their values are available by performing consecutive *DeleteMin* operations on the priority queue. Since we perform $O(N)$ *Insert* and *DeleteMin* operations, the total number of I/Os is $O(N \cdot \frac{1}{DB}\log_{M/B}\frac{N}{B}) = O(sort(N))$. □

Least Common Ancestors Given a rooted tree T, a *least common ancestor* query for a given pair (u, v) of nodes of T is to find a node that is an ancestor of both u and v, and is farthest from the root. Note that T is not necessarily binary.

Theorem 4.6 *Given a rooted tree T with N nodes, one can answer a batch of K least common ancestor queries on T using $O((1+K/N)sort(N))$ I/Os. This also builds an external-memory search structure with $O(D \cdot sort(N))$ blocks that answers a single query in $O(scan(M/B))$ I/Os. Alternatively, one can answer a batch of K queries using $O((1+K/N)sort(N) \cdot \log_{(DB)} \frac{M}{B})$ I/Os. This also builds an external-memory search structure with $O(D \cdot sort(N) \cdot \log_{(DB)} \frac{M}{B})$ blocks that answers a single query in $O(1)$ I/Os.*

Proof: The problem can be reduced to the range minima problem using Euler Tour and list ranking [15]. We construct a search tree S with $O(N/B)$ leaves, each a block storing B data items. Tree S is a complete (M/B)-ary tree with $O(\log_{M/B}(N/B))$ levels, where each internal node v of S corresponds to the items in the subtree S_v rooted at v. Each internal node v stores two lists maintaining prefix and suffix minima of the items in the leaves of S_v, respectively, and a third list maintaining M/B items, each a minimum of the leaf items of the subtree rooted at a child of v. Note that the total size of the prefix (suffix) minima lists of the nodes at the same level is $O(N/B)$ blocks, so that the total number of blocks used by S is $O(\frac{N}{B} \cdot \log_{M/B}(N/B)) = O(D \cdot sort(N))$.

A range minimum query for pair (i, j) is performed as follows. We first identify the least common ancestor w of i and j in S (by numerical computation, since S is a complete (M/B)-ary tree), from which we also know the two children w_1 and w_2 of w whose subtrees contain i and j. Suppose that w_1 is to the left of w_2. To obtain the range minimum, we check the suffix minima list of w_1 at the position of i, the prefix minima list of w_2 at the position of j, and the children minima list of w at all intermediate positions between w_1 and w_2, and take their minimum. Since w has at most M/B children, this takes $O(scan(M/B))$ I/Os.

The K batched queries are performed by sorting them first, so that all queries can be performed by scanning S $O(1)$ times when $K \leq N$. If $K > N$ we process the queries in batches of N at a time. The alternative method is to make S a complete (DB)-ary tree rather than an (M/B)-ary tree. □

4.5.2 Algorithms for Undirected Graphs

In this section we consider a set of problems on undirected graphs. Our results are summarized in Table 4.2. Lower bounds are similar to Corollary 4.1.

Problem	Notes	Lower Bound	Upper Bound
Connected Components, Biconnected Components, Minimum Spanning Forest, and Ear Decomposition			$O(\min\{sort(V^2), \log(V/M) \cdot sort(E)\})$
	Sparse graphs $(E = O(V))$ closed under edge contraction	$\Omega(sort(V))$	$O(sort(V))$
	Randomized, with probability $1 - \exp(-E/\log^{O(1)} E)$		$O((E/V)sort(V))$

Table 4.2 I/O-efficient algorithms for problems on undirected graphs.

Connectivity and Minimum Spanning Forest Let $G = (V, E)$ be an undirected graph. Two vertices u and v of G are *connected* if $u = v$ or there exists a path $P = (u = x_1, x_2, ..., x_k = v)$ such that $(x_i, x_{i+1}) \in E$, for all $1 \leq i \leq k - 1$. This is an equivalence relation on V, and partitions V into equivalence classes. Each equivalence class V_i of V together with all edges incident on vertices in V_i is a *connected component* of G. If G is connected, we define an equivalence relation R on E as follows: eRg if two edges e and g are on a common simple cycle. Each equivalence class E_i partitioned by R together with all vertices that are endpoints of edges of E_i is a *biconnected component* of G. Let P_0 be an arbitrary simple cycle of of G. An *ear decomposition* of G starting with P_0 is an ordered partition of the set of edge $E = P_0 \cup P_1 \cup ... \cup P_k$, such that, for each $1 \leq i \leq k$, P_i is a simple path whose endpoints belong to $P_0 \cup P_1 \cup ... \cup P_{i-1}$, but none of whose internal vertices does. Each simple path P_i is called an *ear*. It is well known that G has an ear decomposition if and only if it is *bridgeless*, that is, there exists no edge whose removal disconnects the graph.

Theorem 4.7 *Given a graph G with V vertices and E edges, the connected components, biconnected components, minimum spanning forest, and ear decomposition of G (if it exists) can be computed using $O(\min\{\log(V/M) \cdot sort(E), sort(V^2)\})$ I/Os.*

Proof: For connected components and minimum spanning forest, our algorithm is based on that of Chin *et al.* [32]. Each iteration performs a constant number of sorts on current edges and one list ranking to reduce the number of vertices by a constant factor. After $O(\log(V/M))$ iterations we fit the remaining M vertices to the main memory and solve the problem easily.

For biconnected components, we adapt the PRAM algorithm of Tarjan and Vishkin [116], which requires generating an arbitrary spanning tree (by our connected component algorithm), evaluating an expression tree (by Theorem 4.5), and computing connected components of a newly created graph.

For ear decomposition, we modify the PRAM algorithm of Maon *et al.* [80], which requires generating an arbitrary spanning tree (by our connected component algorithm), performing batched lowest common ancestor queries (by Theorem 4.6), and evaluating a general expression tree (by Theorem 4.5). $\square$

Observe that all problems in Theorem 4.7 can be solved within the bound of computing minimum spanning forest. For the latter problem, one major bottleneck in the algorithm of Theorem 4.7 is that each iteration reduces by a constant factor *only* the number of vertices. We can actually do better than that, decreasing the numbers of *both* edges and vertices by a constant factor in each iteration, by using an external-memory variation of the random sampling technique of [66, 74]. The verification step in [66, 74] to check whether an edge forming a cycle is heavy enough to be removed is carried out by the $O((E/V)sort(V))$-I/O minimum spanning tree verification method of [61] (also appearing in [25]) which is based on that of King [70]. This gives the following theorem.

Theorem 4.8 *Given a graph G with V vertices and E edges, the connected components, biconnected components, minimum spanning forest, and ear decomposition of G (if it exists) can be computed by a randomized algorithm using $O((E/V)sort(V))$ I/Os with probability $1 - \exp(-E/\log^{O(1)} E)$. For sparse graphs ($E = O(V)$) this is optimal.*

4.5.3 Algorithms for Planar Digraphs

A *planar st-digraph* is a planar acyclic digraph with exactly one source vertex s and exactly one sink vertex t, which is embedded in the plane such that s and t are on the boundary of the external face. Planar *st*-graphs were first introduced by Lempel, Even, and Cederbaum [77] in connection with a planarity testing algorithm, and have subsequently been used in a host of applications, dealing with partial orders [68], planar graph embedding [30, 42, 111], graph planarization [91], graph drawing [41, 43], floor planning [123], planar point location [49, 94], visibility [73, 88, 102, 113, 114, 124], motion planning [101], and VLSI layout compaction [123].

In this section, we consider a planar *st*-digraph G with V vertices, and recall that G has $O(V)$ edges. The results in this section are summarized in Table 4.3. Lower bounds are similar to Corollary 4.1. We obtain the given upper bounds by modifying the PRAM algorithms of Tamassia and Vitter [115], and applying the list ranking and the PRAM simulation techniques.

Problem	Notes	Lower Bound	Upper Bound
Reachability	K queries		$O((1+K/V)sort(V))$
Topological Sorting		$\Omega(sort(V))$	$O(sort(V))$
Drawing, and Visibility Representation	$2V-5$ bends $O(V^2)$ area	$\Omega(sort(V))$	$O(sort(V))$

Table 4.3 I/O-efficient algorithms for problems on planar *st*-graphs. Note that $E = O(V)$ for these graphs.

Reachability and Topological Sorting Given a pair (u, v) of vertices of digraph G, a *reachability query* consists of determining whether G has a directed path from u to v.

Theorem 4.9 *Given a planar st-digraph G with V vertices, one can perform a batch of K reachability queries on G using $O((1+K/V)sort(V))$ I/Os. This also builds an external-memory query structure with $O(V/B)$ blocks that can support an individual reachability query using $O(1)$ I/O operations.*

Corollary 4.3 *Given a planar st-digraph G with V vertices, one can compute a topological ordering of G using $O(sort(V))$ I/O operations.*

Drawing and Visibility Representation A *drawing* of a graph G maps each vertex of G into a point of the plane, and each edge (u, v) of G into a simple open curve between the points associated with the vertices u and v. A *planar drawing* has no crossing edges. In a *polyline drawing*, every edge is drawn as a polygonal chain. A *grid drawing* is a polyline drawing such that the vertices and bends of the edges have integer coordinates. An *upward drawing* for an acyclic digraph is such that every edge is drawn as a curve monotonically increasing in the vertical direction. A *visibility representation* Γ for a directed graph G maps each vertex v of G to a horizontal segment $\Gamma(v)$ and each edge (u, v) of G to a vertical segment $\Gamma(u, v)$ that has its lower endpoint on $\Gamma(u)$, its upper endpoint on $\Gamma(v)$, and does not intersect any other horizontal segment.

Theorem 4.10 *Given a planar st-digraph G with V vertices, one can compute a planar upward polyline grid drawing for G with $2V-5$ bends and $O(V^2)$ area, and a visibility representation for G with integer coordinates and $O(V^2)$ area, using $O(sort(V))$ I/O operations.*

Chapter 5

Experiments: Practical I/O Efficiency of Geometric Algorithms

5.1 Introduction

Although there has been an increasing interest in the development of I/O-efficient algorithms in recent years, most of the developed algorithms, however, are shown to be efficient only *in theory*, and their performance *in practice* is yet to be evaluated. In particular, all such algorithms assume that the internal computation is free compared to the I/O cost, which also has to be justified in practice. In this chapter, we establish the practical efficiency of one such algorithm by an extensive experimental study.

5.1.1 Previous Related Work

As mentioned above, most of the previous work on I/O-efficient computation is theoretical. In Chapter 4, we have presented a collection of new techniques for designing and analyzing I/O-efficient graph algorithms, and apply these techniques to a wide variety of specific problems. Other related theoretical results are reviewed in Section 4.1.2.

For excellent examples of experimental work in computational geometry, see Bentley [11, 12, 13, 14]. As for experimental work on I/O-efficient computation, very recently Vengroff has built an environment called TPIE for programming external-memory algorithms as he proposed earlier in [120], and also Vengroff and Vitter [119] have reported some benchmarks of TPIE on sorting and matrix multiplication. This work, however, is mainly on providing a programming environment and not on performance comparisons between external-memory algorithms and conventional algorithms. Also worth noting is the work by Ramaswamy and Kanellakis [97], who study the problem of indexing a class hierarchy in Object Oriented Databases. Based on the insight of Kanellakis *et al.* [65] that the problem is a special form of two-dimensional dynamic range searching in external memory, they propose a technique called *class-division*, and show by experiments that in the average case, class-division performs far less I/Os than the *class hierarchy index* technique most popular today. Their experimental setting is somewhat different from ours, however, in that the numbers of I/O operations rather than the total running times are measured.

5.1.2 Our Results in This Chapter

We present an extensive experimental study comparing the performance of four algorithms for the following *orthogonal segment intersection problem*: given a set of horizontal and vertical line segments in the plane, report all intersecting horizontal-vertical pairs. The problem has important applications in VLSI layout and graphics, which are large-scale in nature. The algorithms under evaluation are distribution sweep of Goodrich *et al.* [60] and three variations of plane sweep [93]. Distribution sweep theoretically has optimal I/O cost [60]. Plane sweep is a well-known and powerful technique in computational geometry, and is optimal for this particular problem in terms of internal computation [93]. The three variations of plane sweep differ by the sorting methods (external merge sort [2] vs. internal merge sort) used in the preprocessing phase and the dynamic data structures (B tree [9, 34, 35] vs. 2-3-4 tree [35]) used in the sweeping phase. We generate the test data by three programs that use a random number generator while producing some interesting properties that are predicted by our theoretical analysis. The sizes of the test data range from 250 thousand segments to 2.5 million segments. The experiments provide detailed quantitative evaluation of the performance of the four algorithms, and the observed behavior of the algorithms is consistent with their theoretical properties.

The contribution of this chapter can be summarized as follows:

- We have presented the first experimental work comparing the practical performance between external-memory algorithms and conventional algorithms with large-scale test data.
- We have generated test data with interesting properties that are predicted by our theoretical analysis. In particular, we give techniques for analyzing the expected number of intersections and the average number of vertical overlaps among vertical segments in the data sets generated, which may be of independent interest.
- We have implemented distribution sweep, three variations of plane sweep and external merge sort under a uniform experimental framework so that some resource usage can be parameterized and various statistic information related to performance can be obtained and analyzed. The implementations handle all degeneracies and are robust.
- We have presented the first experimental study on the four algorithms for the important orthogonal segment intersection problem with large-scale test data, and established the practical efficiency of distribution sweep.

5.1.3 Organization of the Chapter

The rest of this chapter is organized as follows. In Section 5.2 we overview the four algorithms under evaluation. Details on the experimental setting are given in Section 5.3. In Section 5.4, we summarize our experimental results in nine charts and give a comparative analysis of the performance of the four algorithms. Finally, we conclude the chapter in Section 5.5.

5.2 The Algorithms Under Evaluation

The four algorithms considered in this chapter are distribution sweep, denoted `Distribution`, and three variations of plane sweep, denoted `B-Tree`, `234-Tree`, and `234-Tree-Core`, described next. To discuss the time complexity, let N be the total number of segments in the given input, K the number of intersecting pairs that must be reported, and M and B the numbers of segments that can fit into the main memory and into a page, respectively. Each I/O operation transfers one page of data.

5.2.1 Three Variations of Plane Sweep

The well-known *plane sweep* paradigm [93] is a powerful technique in computational geometry, and is optimal for the orthogonal segment intersection problem in terms of internal computation. The method consists of preprocessing and sweeping phases. In the preprocessing phase, we sort all endpoints by the y-coordinates in non-decreasing order. In the sweeping phase, we (conceptually) use a horizontal sweep line to sweep the plane from bottom to top, and use a dynamic data structure, typically a search tree, to keep the objects currently intersecting the sweep line. Operations (object insertions/deletions or queries) are performed only when some *events* are encountered by the sweep line. In our problem, the objects being maintained are vertical segments, and the events are the endpoints of all segments. When a bottom endpoint of a vertical segment is encountered, it is inserted to the data structure; the segment remains intersecting the sweep line until its top endpoint is encountered, at which time it is deleted from the data structure. When an endpoint of a horizontal segment s is encountered, we search the data structure to find and report all vertical segments intersecting s, i.e., those segments whose x-coordinates are contained in the x-interval spanned by s. The sequence of events during the sweeping process is given by sorting in the preprocessing phase. Using any dynamic balanced tree, plane sweep takes optimal $O(N \log N)$ time in terms of internal computation.

Our three variations of plane sweep differ by the sorting methods and the dynamic data structures used. The first variation, `B-Tree`, uses external merge sort [2] and a B tree [9, 34, 35]; this is a direct way to implement plane sweep in secondary memory. The number of I/O operations performed in the first phase is optimal $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ [2], and in the second phase is $O(N \log_B \frac{N}{B} + \frac{K}{B})$. The second variation, `234-Tree`, uses external merge sort and a 2-3-4 tree [35], viewing the internal memory as having an infinite size and letting the virtual memory feature of the OS handle page faults during the second (sweeping) phase. It has the same I/O cost in the first phase and $O(N \log N + \frac{K}{B})$ I/O cost in the second phase. Finally, the third variation, `234-Tree-Core`, uses internal merge sort and a 2-3-4 tree, letting the OS handle page faults all the time. The I/O costs in the first and second phases are $O(N \log N)$ and $O(N \log N + \frac{K}{B})$, respectively. Viewing the internal memory as virtually having an infinite size is conceptually the simplest, and is actually the most commonly used strategy today in practice.

5.2.2 Distribution Sweep

Distribution sweep [60] is an external-memory version of plane sweep based on the subdivision technique used in the "distribution sort" algorithms of [2, 86, 122]. When applied to the orthogonal segment intersection problem, it works as follows.

In the preprocessing phase, we sort the endpoints of all segments into two lists, one by x and the other by y. Again we use external merge sort. The list sorted by x is used to locate the medians which we will use to split the input into $\lfloor \frac{M}{B} \rfloor$ vertical strips. The list sorted by y is used to perform the sweep, moving bottom up. Associated with each strip γ_i is an *active list* A_i that maintains the vertical segments inside γ_i that intersect the horizontal sweep line.

During the sweep, if the bottom endpoint of a vertical segment is encountered, it is inserted into the active list A_i of the strip in which the segment lies, and later it is deleted from A_i when its top endpoint is encountered. When an endpoint of a horizontal segment s is met, we locate the two strips γ_i and γ_j containing the two endpoints of s, $i <= j$, and report all vertical segments currently in the active lists $A_{i+1}, ..., A_{j-1}$. Note that strips $\gamma_{i+1}, ..., \gamma_{j-1}$ are completely spanned horizontally by s. This reports all vertical segments intersecting s except for those lying in γ_i and γ_j, which will be processed in the next level of recursion. After the sweep is complete in the top level, we recursively perform the same procedure to each strip γ_i. The recursive process continues until the size of the subproblem falls below M, at which time we simply solve the problem in main memory.

Based on the above idea, distribution sweep actually uses a "lazy deletion" policy. Each active list A_i can be viewed as a stack with the topmost block maintained in the internal memory. Whenever the internal block is full, it is written to the external memory. Now, the top endpoints of vertical segments become "no action" events, and the deletions are performed during reporting: in the reporting process, instead of just reporting each vertical segment t in active list A_i, we check the top endpoint of t to see if its deletion time has passed. If so we delete t, otherwise we report t and retain t in A_i. The total I/O cost is $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}+\frac{K}{B})$, which is optimal. Notice that distribution sweep needs two sortings as opposed to just one sorting in plane sweep.

5.3 Experimental Setting

5.3.1 Generation and Analysis of the Test Data

We use three programs to generate our test data; all of them use a random number generator that gives a uniform distribution. The programs randomly generate several attributes of a segment such as its length, position, and whether it is horizontal or vertical, and also maintain certain simple structures to make the test data interesting.

We must use the random number generator carefully to obtain interesting test data. Observe that if we just randomly generate segments with lengths uniformly distributed over $[0, N]$, place them randomly (and uniformly in each dimension) in a square with side length N, and make horizontal and vertical segments equally likely to occur, then the number K of intersections is $\Theta(N^2)$ (obtained by the analysis given below). In this case, any algorithm has $\Omega(\frac{N^2}{B})$ reporting I/O cost, which dominates the searching I/O costs in all four algorithms under evaluation. In fact, the following brute-force algorithm performs equally well: for each segment, check all the other $N-1$ segments for intersections; the I/O cost is $O(N\cdot\frac{N}{B})=O(\frac{N^2}{B})$. Certainly this kind of test data is undesirable.

Our three programs are denoted `gen-short`, `gen-long`, and `gen-rect`, and the data sets generated are correspondingly denoted `data-short`, `data-long` and `data-rect`. We try to generate test data with small number of intersections so that the searching I/O cost dominates the reporting cost. Also, it is conceivable that the number of vertical overlaps among vertical segments at a given time decides the tree size in that moment of plane sweep and also the total size of the active lists at that time of distribution sweep. Thus the number of vertical overlaps may affect the performance of the four algorithms. Our three programs generate test data with distinct structures regarding the number of intersections and the number of vertical overlaps. Also, all three programs decide whether the current segment being generated is horizontal or vertical by tossing a fair coin (simulated by using the random number generator).

Program `gen-short` generates short segments whose lengths are uniformly distributed over $[0, \sqrt{N}]$. The segments are randomly placed in an $N\times N$ square Q. More specifically, for the left endpoints of horizontal segments, the distances to the left and bottom sides of Q are uniformly distributed over $[0, N-\sqrt{N}]$ and over $[0, N]$, respectively. Similarly, for the bottom endpoints of vertical segments, the distances to the left and bottom sides of Q are uniformly distributed over $[0, N]$ and over $[0, N-\sqrt{N}]$, respectively. To simplify the discussion, we assume that the coordinate of the lower-left corner of Q is $(0, 0)$ (in the actual program this coordinate is $(-0.5N, -0.5N)$).

We now analyze the expected number of horizontal-vertical intersecting pairs in `data-short`. Let K be a random variable for the number of such pairs. We can express K by $K=\sum_{1\leq i,j\leq N,\ i\neq j} K_{ij}$, where for $1 \leq i, j \leq N$, $i \neq j$, K_{ij} is a 0-1 random variable defined by

$$K_{ij} = \begin{cases} 1 & \text{if segment } s_i \text{ is horizontal, segment } s_j \text{ is vertical, and } s_i \cap s_j \neq \emptyset \\ 0 & \text{otherwise.} \end{cases}$$

Clearly, we have

$$\Pr\{K_{ij} = 1\} = \Pr\{s_i \text{ is horizontal and } s_j \text{ is vertical}\} \cdot \Pr\{\text{horizontal } s_i \cap \text{vertical } s_j \neq \emptyset\}.$$

The first term is $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$, so let us focus on computing the second term. In the following, s_i is horizontal and s_j is vertical. For s_i and s_j to intersect, the y-coordinate of s_i is contained in the y-interval spanned by s_j (denoted by $y(s_i) \in I_y(s_j)$), and similarly for the x-dimension. Define P_0 to be the conditional probability $\Pr\{y(s_i) \in I_y(s_j) \mid y(b(s_j)) = t, |s_j| = h\}$, where $y(b(s_j))$ is the y-coordinate of the bottom endpoint of s_j, and $|s_j|$ is the length of s_j. For any $t \in [0, N - \sqrt{N}]$ and $h \in [0, \sqrt{N}]$ we have $P_0 = \Pr\{y(s_i) \in [t, t+h] \mid y(b(s_j)) = t, |s_j| = h\} = \frac{h}{N}$, since $y(s_i)$ is uniformly distributed over $[0, N]$. Let $f(t)$ be the probability density function of $y(b(s_j))$. Since $P_0 = \frac{h}{N}$ is independent of t, we have $\Pr\{y(s_i) \in I_y(s_j) \mid |s_j| = h\} = \int_{-\infty}^{+\infty} P_0 \cdot f(t)\, dt = P_0 \cdot \int_{-\infty}^{+\infty} f(t)\, dt = \frac{h}{N}$. Moreover, $\Pr\{y(s_i) \in I_y(s_j) \mid |s_j| = h, |s_i| = w\} = \Pr\{y(s_i) \in I_y(s_j) \mid |s_j| = h\} = \frac{h}{N}$, since events $\{y(s_i) \in I_y(s_j)\}$ and $\{|s_i| = w\}$ are conditionally independent given $|s_j| = h$. By a similar argument, we have $\Pr\{x(s_j) \in I_x(s_i) \mid |s_j| = h, |s_i| = w\} = \frac{w}{N}$, and therefore $\Pr\{s_i \cap s_j \neq \emptyset \mid |s_j| = h, |s_i| = w\} = \frac{h}{N} \cdot \frac{w}{N}$. Now $|s_j|$ is uniformly distributed over $[0, \sqrt{N}]$, namely,

$$g(h) = \begin{cases} \frac{1}{\sqrt{N}} & \text{if } 0 \leq h \leq \sqrt{N} \\ 0 & \text{otherwise} \end{cases}$$

is the probability density function of $|s_j|$; similarly for $|s_i|$. Thus

$$\Pr\{\text{horizontal } s_i \cap \text{vertical } s_j \neq \emptyset\} = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} \frac{h}{N} \cdot \frac{w}{N} \cdot g(h) \cdot g(w)\, dh\, dw = \frac{1}{4N}.$$

Therefore we have $\Pr\{K_{ij} = 1\} = \frac{1}{16N}$. It follows that $E[K] = N(N-1) \cdot E[K_{ij}] = \frac{1}{16}(N-1)$. Figure 5.1 shows the actual numbers of intersections with respect to data size N, for all three data sets generated. The observed K values are indeed $\frac{1}{16}N$ for `data-short`.

Now we proceed to analyze for `data-short` the average number of vertical overlaps among vertical segments, that is, the average number of vertical segments "cut" by the horizontal sweep line l *when l is passing through an event.* The average is taken over all sweeping events. Notice that this average number is exactly the average number of items stored in the data structure when an update/query operation is performed during plane sweep. A related problem has been studied in [69]. Intuitively, we would estimate this average number to be proportional to the average length of vertical segments, which is $\Theta(\sqrt{N})$. A rigorous analysis is given next.

Let V be a random variable for the number of vertical segments cut by l for an event. Our goal is to compute $E[V]$. We can express V by $V = V_1 + V_2 + \cdots + V_N$, where for $1 \leq j \leq N$, V_j is a 0-1 random variable defined by

$$V_j = \begin{cases} 1 & \text{if segment } s_j \text{ is vertical and } s_j \cap l \neq \emptyset \\ 0 & \text{otherwise.} \end{cases}$$

Clearly, we have

$$\Pr\{V_j = 1\} = \Pr\{s_j \text{ is vertical}\} \cdot \Pr\{\text{vertical } s_j \cap l \neq \emptyset\}. \tag{5.1}$$

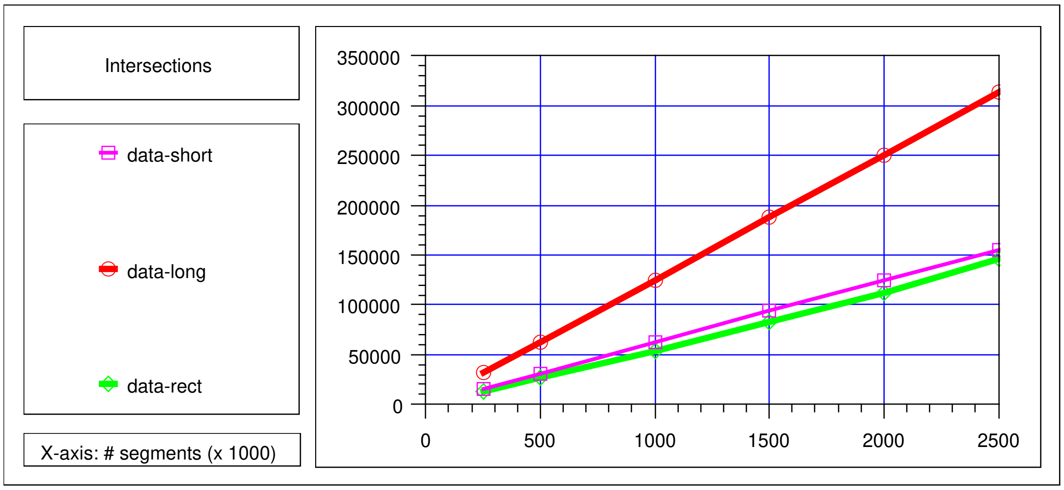


Figure 5.1: The actual numbers of intersections with respect to the number of segments in data sets `data-short`, `data-long` and `data-rect`.

The first term is $\frac{1}{2}$, so let us compute the second term. In the following s_j is vertical. If $y(l)$ were uniformly distributed over $[0, N]$, then letting l play the role of a horizontal segment s_i and applying the previous method for $E[K]$, we would have $\Pr\{s_j \cap l \neq \emptyset \mid |s_j| = h\} = \Pr\{y(l) \in I_y(s_j) \mid |s_j| = h\} = \frac{h}{N}$, and $\Pr\{V_j = 1\}$ could be computed accordingly. But this is not the case.

Recall that the positions of l depend on the positions of the events. Let the probability density function of $y(l)$ be defined by

$$k(u) = \begin{cases} k_1(u) & \text{if } 0 \leq u < \sqrt{N} \\ k_2(u) & \text{if } \sqrt{N} \leq u \leq N - \sqrt{N} \\ k_3(u) & \text{if } N - \sqrt{N} < u \leq N \\ 0 & \text{else.} \end{cases}$$

If there were only horizontal segments, then we would have $k_1(u) = k_2(u) = k_3(u) = \frac{1}{N}$. Now consider including also the vertical segments. There is no bottom endpoint q with $y(q) \in (N - \sqrt{N}, N]$, so $k_3(u) < \frac{1}{N}$. Also, for a top endpoint q to have $y(q) = u$ for some $u \in [0, \sqrt{N})$ (e.g., $u = \frac{1}{2}\sqrt{N}$), the length of the corresponding vertical segment must not exceed u, while there is no such restriction on a top endpoint with y-coordinate in $[\sqrt{N}, N - \sqrt{N}]$. Therefore we have $k_1(u) < \frac{1}{N}$ and $k_2(u) \geq \frac{1}{N}$. To compute $\Pr\{V_j = 1\}$, we proceed in a different way.

Define P to be the conditional probability $\Pr\{s_j \cap l \neq \emptyset \mid y(l) = u, |s_j| = h\}$. By the fact that $y(b(s_j))$ is uniformly distributed over $[0, N - \sqrt{N}]$ and the analysis shown in Fig. 5.2, we have that $P = \frac{h}{N - \sqrt{N}}$ for $\sqrt{N} \leq u \leq N - \sqrt{N}$ (see Fig. 5.2(b)) and $P = \frac{\min\{u,h\}}{N - \sqrt{N}}$ for $0 \leq u < \sqrt{N}$ (see Fig. 5.2(a)). As for $N - \sqrt{N} < u \leq N$ (see Fig. 5.2(c)), note that $P = \frac{h-t}{N-\sqrt{N}}$ if $h \geq t$ and $P = 0$ if $h < t$, and thus $P = \frac{len}{N - \sqrt{N}}$ where $len = \max\{h - t, 0\} = \max\{h - u + N - \sqrt{N}, 0\}$. In summary, we have

$$P \stackrel{\text{def}}{=} \Pr\{s_j \cap l \neq \emptyset \mid y(l) = u, |s_j| = h\} = \begin{cases} \frac{\min\{u,h\}}{N-\sqrt{N}} \stackrel{\text{def}}{=} P_1 & \text{if } 0 \leq u < \sqrt{N} \\ \frac{h}{N-\sqrt{N}} \stackrel{\text{def}}{=} P_2 & \text{if } \sqrt{N} \leq u \leq N - \sqrt{N} \\ \frac{\max\{h-u+N-\sqrt{N},0\}}{N-\sqrt{N}} \stackrel{\text{def}}{=} P_3 & \text{if } N - \sqrt{N} < u \leq N \\ 0 & \text{else.} \end{cases}$$

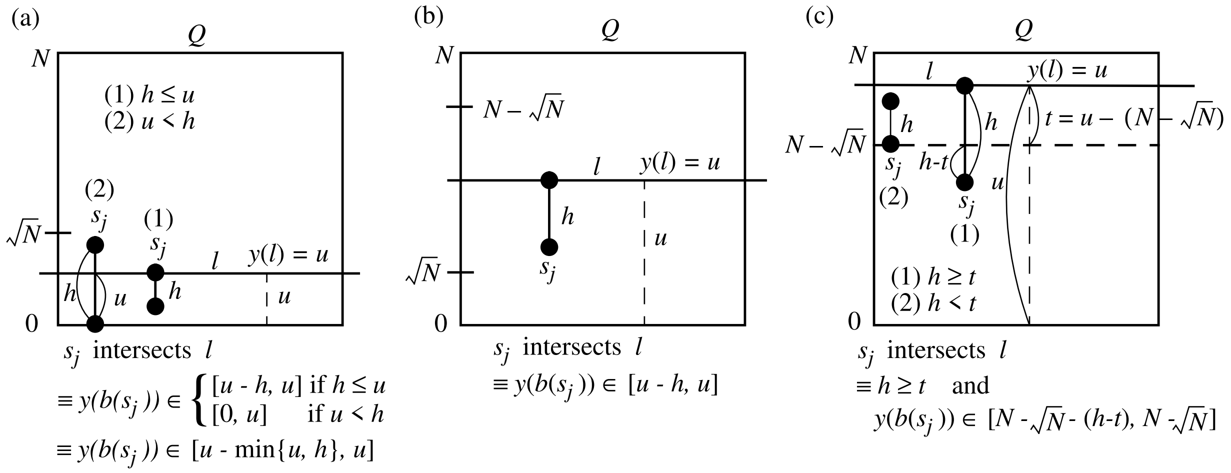


Figure 5.2: Computing $\Pr\{s_j \cap l \neq \emptyset \mid y(l) = u, |s_j| = h\}$ for `data-short`: (a) $0 \le u < \sqrt{N}$; (b) $\sqrt{N} \le u \le N - \sqrt{N}$; (c) $N - \sqrt{N} < u \le N$.

Observe that $\min\{u, h\} \le h$ and that $\max\{h - u + N - \sqrt{N}, 0\} \le h$ for $u > N - \sqrt{N}$, so we have $P \le \frac{h}{N - \sqrt{N}}$ for $u \in [0, N]$. Recall that the probability density function of $|s_j|$ is $g(h)$ given before. Let $\widetilde{P}$ denote the probability $\Pr\{\text{vertical } s_j \cap l \neq \emptyset\}$. Then

$$\begin{aligned}
\widetilde{P} &= \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} P \cdot k(u)\, g(h)\, du\, dh \\
&\le \int_0^{\sqrt{N}} \int_0^N \frac{h}{N - \sqrt{N}}\, k(u)\, \frac{1}{\sqrt{N}}\, du\, dh = \int_0^{\sqrt{N}} \frac{h}{N - \sqrt{N}} \frac{1}{\sqrt{N}} \left(\int_0^N k(u)\, du \right) dh = \frac{1}{2} \frac{1}{\sqrt{N} - 1},
\end{aligned}$$

where the last equality follows from the fact that $\int_0^N k(u) du = 1$. Also,

$$\begin{aligned}
\widetilde{P} &= \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} P \cdot k(u)\, g(h)\, dh\, du \\
&= \int_0^{\sqrt{N}} \int_{-\infty}^{+\infty} P_1 \cdot k_1(u)\, g(h)\, dh\, du + \int_{\sqrt{N}}^{N - \sqrt{N}} \int_{-\infty}^{+\infty} P_2 \cdot k_2(u)\, g(h)\, dh\, du \\
&\quad + \int_{N - \sqrt{N}}^{N} \int_{-\infty}^{+\infty} P_3 \cdot k_3(u)\, g(h)\, dh\, du \\
&\ge \int_{\sqrt{N}}^{N - \sqrt{N}} \int_0^{\sqrt{N}} \frac{h}{N - \sqrt{N}} \frac{1}{N} \frac{1}{\sqrt{N}}\, dh\, du = \frac{1}{2} \frac{1}{\sqrt{N} - 1} \frac{\sqrt{N} - 2}{\sqrt{N}},
\end{aligned}$$

where the inequality follows from that fact that the first and the third additive terms are both non-negative and that $k_2(u) \ge \frac{1}{N}$. Now we have lower and upper bounds for $\widetilde{P}$, and recall from equation (5.1) that $\Pr\{V_j = 1\} = \frac{1}{2} \cdot \widetilde{P}$. It follows that $E[V] = N \cdot E[V_j] \le \frac{1}{4}\sqrt{N}(\frac{\sqrt{N}}{\sqrt{N}-1}) = \frac{1}{4}\sqrt{N} + \frac{1}{4} + \frac{1}{4(\sqrt{N}-1)}$ and also $E[V] \ge \frac{1}{4}\sqrt{N}(\frac{\sqrt{N}-2}{\sqrt{N}-1}) = \frac{1}{4}\sqrt{N} - \frac{1}{4} - \frac{1}{4(\sqrt{N}-1)}$, that is, $E[V] = \frac{1}{4}\sqrt{N} + O(1) \sim \frac{1}{4}\sqrt{N}$. It is interesting to see that $\frac{1}{4}\sqrt{N}$ can be interpreted as the product of the probability of a segment being vertical ($\frac{1}{2}$) and the average length of vertical segments ($\frac{1}{2}\sqrt{N}$). Figure 5.3 shows the actual values of the average number of vertical overlaps with respect to data size N, for all three data sets generated. We also compare for `data-short` the actual values and the analyzed values in Table 5.1, which shows that the observed values are indeed $\frac{1}{4}\sqrt{N}$.

N : # segments ($\times 10^3$)	250	500	1000	1500	2000	2500
Actual value	125.23	176.74	249.98	306.40	353.58	395.60
$\frac{1}{4}\sqrt{N}$ ($\sim E[V]$)	125	176.78	250	306.19	353.55	395.28

Table 5.1 The actual and analyzed values of the average number of vertical overlaps in data set `data-short`.

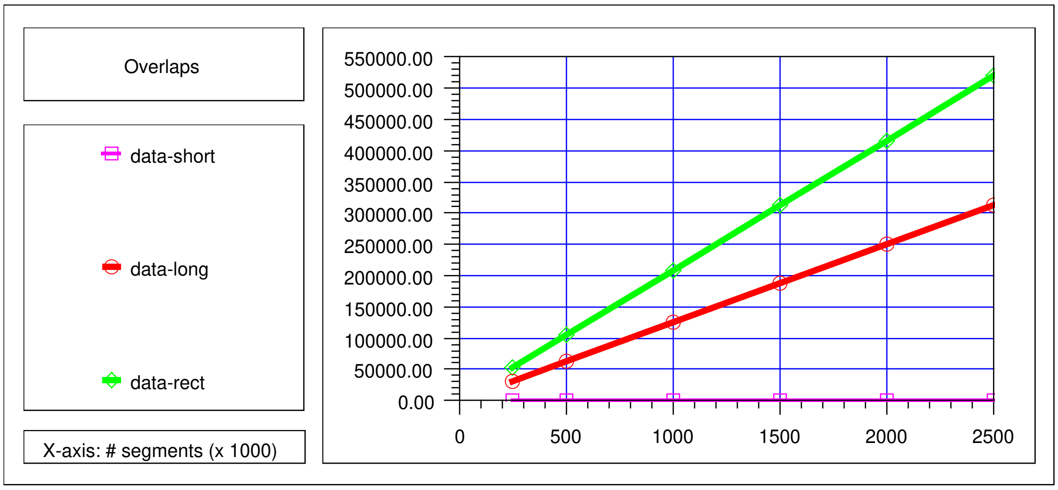


Figure 5.3: The actual values of the average number of vertical overlaps with respect to the number of segments in data sets `data-short`, `data-long` and `data-rect`.

Program `gen-long` generates short as well as long segments while keeping the number of intersections small. For a horizontal segment, the length is assigned $\sqrt{N}$ (short segment); for a vertical segment, the program tosses a coin, giving length $\sqrt{N}$ (short segment) if the outcome is a head and N (long segment) otherwise. The horizontal and vertical short segments are randomly placed in the $N \times N$ sqaure Q in the same way as described in `gen-short`. As for vertical long segments, the bottom endpoints are placed randomly in an $N \times N$ square Q' whose lower-right corner coincides with the lower-left corner of Q, such that the distances from the bottom endpoints to the left and bottom sides of Q' are both uniformly distributed over $[0, N]$. Thus we have about $\frac{1}{2}N$ short horizontal segments, $\frac{1}{4}N$ short vertical segments, and $\frac{1}{4}N$ long vertical segments which cause no intersections. Using similar analysis methods as given before, we have that $E[K] = \frac{1}{8}(N-1)$ and $E[V] = \Theta(N)$; the latter bound shows that $E[V]$ is asymptotically proportional to the length of the *long* vertical segments. The observed K values of `data-long` are indeed $\frac{1}{8}N$ (see Fig. 5.1), and the observed values of the average number of vertical overlaps are also $\frac{1}{8}N$ (see Fig. 5.3).

In program `gen-rect`, we generate horizontal and vertical segments with lengths uniformly distributed over $[20, 60]$ and over $[0, 2N]$, respectively. The left endpoints of horizontal segments are randomly placed inside an $80N \times N$ rectangle R (with horizontal side length $80N$), such that the distances to the left and bottom sides of R are uniformly distributed over $[0, 80N]$ and over $[0, N]$, respectively. The vertical segments are placed as follows: in the x-direction, the distance between the left side of R and the *i-th vertical* segment is $(i-1) \times 160$, $i = 1, 2, \cdots$; in the y-direction, the distances from the bottom endpoints to the bottom side of R are uniformly distributed over $[0, N]$. It is easily seen that each horizontal segment can intersect at most one vertical segment, so that $K = O(N)$. Using similar analysis methods as given before, we have that $E[K] = \Theta(N)$ and also $E[V] = \Theta(N)$. Again the latter bound shows that $E[V]$ is asymptotically proportional to the average length of vertical segments. Figures 5.1 and 5.3 show that the actual K values are $\frac{1}{18}N$, and the actual values of the average number of vertical overlaps are $\frac{1}{4.8}N$.

5.3.2 Computing Environment and Performance Measures

We perform the experiments on a Sun Sparc-10 workstation, which is running under Solaris 2.4 and is a multi-user distributed system. The main memory size is 32Mb and one page is of size 4Kb. Our performance measures are running time, number of I/O operations performed (i.e., number of pages read and written by the process), and number of page faults occurred.

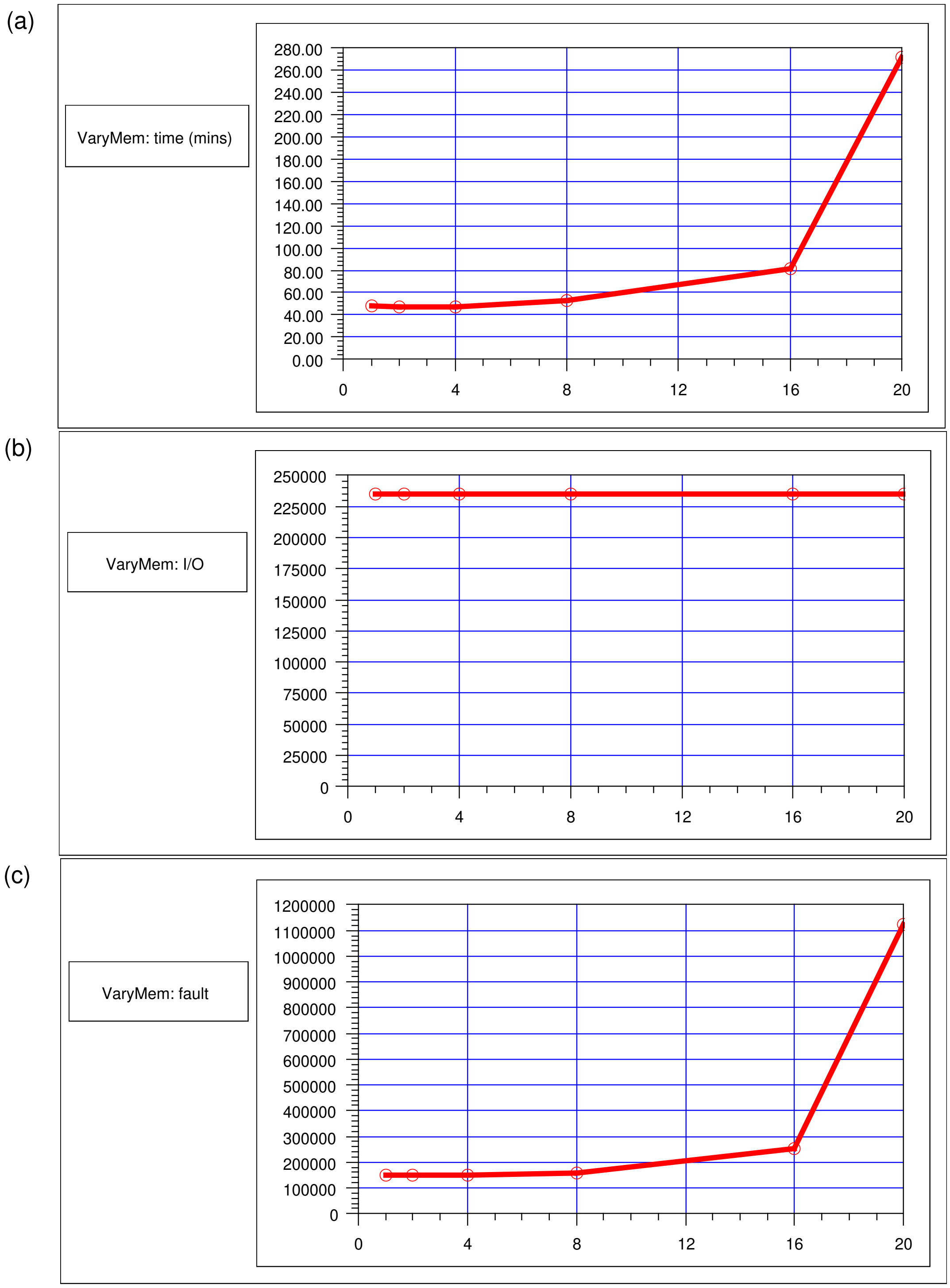
Notice that the running time is our ultimate concern. Unlike previous experimental work, the CPU time does not correctly reflect the performance of the algorithms we want to measure, since our main concern is the amount of time in which the CPU is sleeping waiting for the I/O or page faults. To overcome the difficulty, we perform all experiments by running the processes in the *real-time* class with the *highest priority* and measure the elapsed time. Also, the secondary memory used is the local disk so that the performance is not affected by the network file servers.

We are surprised to find that the system does not fully support performance statistic information. For example, it is claimed that user commands `time` and `timex` give CPU and elapsed times as well as numbers of I/O and page faults, etc., but it turns out that only the information regarding times are available. By using a system call in our program to retrieve the information in the `/proc` file system, we are able to obtain the number of page faults that require physical I/O's, yet the numbers of pages read and written by the process are still unavailable. Therefore, we also keep track of the numbers of times the `read` and `write` system calls are executed, where each time the size of the data being transferred is one page by our implementation.

We implement the algorithms so that the page size and the amount of main memory used can be parameterized. We set the page size to be 4Kb, to be consistent with the system we are using. As for the main memory size, it is surprising that the main memory size available for use is typically much smaller than what we thought. When we run `Distribution` on `data-long` of 1.5×10^6 segments with various sizes of main memory used (see Fig. 5.4), in theory we would expect that using more main memory results in a better performance according to the I/O cost bound $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}+\frac{K}{B})$, but the experiments show that using 4Mb gives the best performance (average running time 47.64 minutes), and using 20Mb gives a significantly worse performance (average running time 271.97 minutes)! Going from 1Mb to 4Mb, the number of I/O operations slightly decreases, and the number of page faults increases slightly yet negligibly; the net effect is to make the running time decrease slightly (see Fig. 5.4). Going from 4Mb to 20Mb, the number of I/O operations again decreases only slightly, but the number of page faults increases significantly, thus resulting in a much worse performance (see Fig. 5.4). This is actually a system issue. Using the `top` user command, we see that the "real" main memory size in the system configuration is only 26Mb rather than 32Mb and that processes (including their text, data, and stack portions) are never fully loaded into the main memory. The process loading behavior is decided by the OS and the user has no control over it. In the following, all the algorithms are running with the parameters of the main memory size set to 4Mb.

5.4 Analysis of the Experimental Results

Algorithms `Distribution`, `B-Tree`, `234-Tree`, and `234-Tree-Core` have been executed on data sets `data-short`, `data-long`, and `data-rect`, with data sizes ranging from 250 thousand segments to 2.5 million segments. While running times and numbers of page faults may differ between runs of



* X-axis: size of the main memory used (Mb)

Figure 5.4: Running `Distribution` on data set `data-long` of 1.5×10^6 segments with various sizes of the main memory used: (a) average running times in minutes; (b) exact numbers of I/O operations; (c) average numbers of page faults.

the same example, the numbers of I/O operations are always the same. We run each example three times, and find that the variation among runs is at most 5%. More importantly, these differences among runs do not affect the performance ranking of the four algorithms, that is, the slowest run of a higher-ranked algorithm is still faster than the fastest run of a lower-ranked algorithm. Figures 5.5–5.7 show the corresponding values of average running times, exact numbers of I/O operations, and average numbers of page faults of the four algorithms.

Our experimental results show that while the performance of the three variations of plane sweep depends heavily on the average number of vertical overlaps, the performance of distribution sweep is both steady and efficient. Also, distribution sweep does not require a large amount of main memory to perform well: using 4Mb is enough. We make more detailed observations as follows:

- `234-Tree-Core` performs the best for small input ($N = 250 \times 10^3$) in all three data sets (see Figs. 5.5–5.7), but as input size grows, the performance becomes considerably worse, and up to $N = 10^6$ its running times are already out of comparison.
- Consider data set `data-short` (see Fig. 5.5). Excluding `234-Tree-Core`, `234-Tree` always runs the fastest and `Distribution` always runs the slowliest. This can be explained by the small numbers of vertical overlaps which results in small tree sizes that still fit into the main memory. Also, `Distribution` performs two sortings, while all the others perform only one sorting.
- For data set `data-long` (see Fig. 5.6), `Distribution` runs much faster than all the others for $N \geq 1.5 \times 10^6$. `234-Tree`, following `234-Tree-Core` after $N \geq 10^6$, is out of comparison for its running times and numbers of page faults after $N \geq 1.7 \times 10^6$. The running times and numbers of I/O operations of `B-Tree` are still more or less linear, and are always worse than those of `Distribution`.
- For data set `data-rec` (see Fig. 5.7) with $N \geq 10^6$, `Distribution` performs the fastest, and the running times of the four algorithms differ significantly. For example, for $N = 1.37 \times 10^6$ and on average, `Distribution` runs for 45.29 minutes, `B-Tree` runs for 74.54 minutes, but `234-Tree` runs for more than 10.5 hours. Also, for $N = 2.5 \times 10^6$, `Distribution` always runs for less than 1.5 hours, but `B-Tree` always runs for more than 8.5 hours.
- For all three data sets, `234-Tree` and `234-Tree-Core` always have small numbers of I/O operations (see Figs. 5.5(b), 5.6(b), and 5.7(b)). This is because in the sweeping phase both of them only perform `read` for reading the input once and `write` for writing the output once, and all other I/O activities are page faults caused by the assumption of an infinite-size virtual memory. Also, `Distribution` always has much less I/O operations than `B-Tree` (see Figs. 5.5(b), 5.6(b), and 5.7(b)). Recall that the I/O cost bounds for `Distribution` and `B-Tree` are $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B} + \frac{K}{B})$ and $O(N \log_B \frac{N}{B} + \frac{K}{B})$, respectively. With the parameters of the main memory size set to 4Mb, the two logarithmic terms in these bounds are almost the same, and it is the $\frac{1}{B}$ term that makes the difference significant.
- Page faults seem to be more time-consuming than I/O operations (see Figs. 5.5–5.7). This is because the number of I/O operations is obtained by counting the number of times the `read` and `write` system calls are executed, and thus some of them might be executed for pages that are still residing in main memory (i.e., in the system buffer cache), while the number of page faults only counts for those that actually require physical I/O's. We hope that the actual number of I/O operations can be available by a better system support in the future.

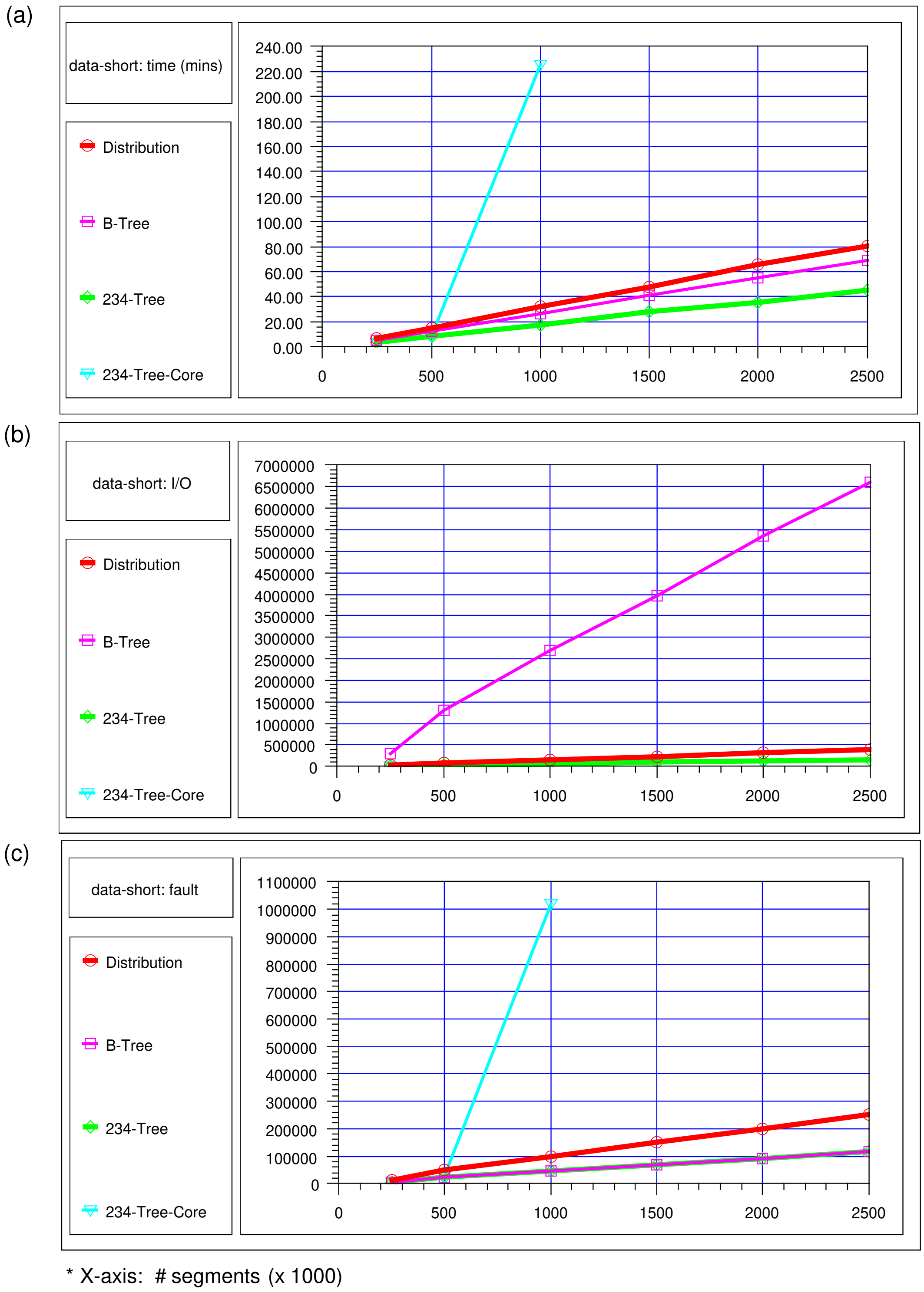
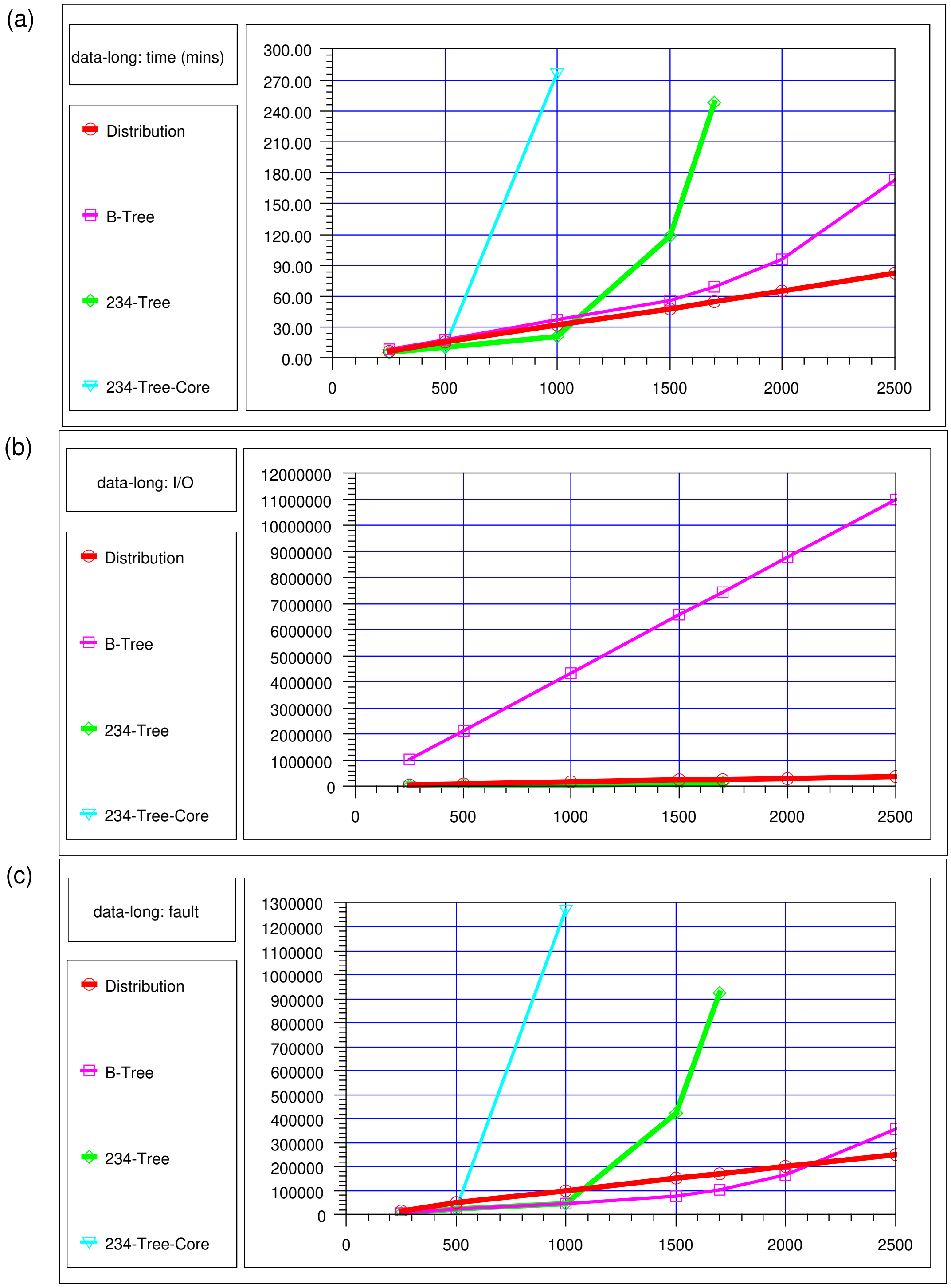


Figure 5.5: The results for the algorithms running on data set `data-short`: (a) average running times in minutes; (b) exact numbers of I/O operations; (c) average numbers of page faults. We run `234-Tree-Core` only up to $N = 10^6$ since at this point it already takes time much longer than the others even at $N = 2.5 \times 10^6$.



* X-axis: # segments (x 1000)

Figure 5.6: The results for the algorithms running on data set `data-long`: (a) average running times in minutes; (b) exact numbers of I/O operations; (c) average numbers of page faults. We run `234-Tree-Core` only up to $N = 10^6$ and `234-Tree` only up to $N = 1.7 \times 10^6$ since at these points they already take times much longer than the others even at $N = 2.5 \times 10^6$.

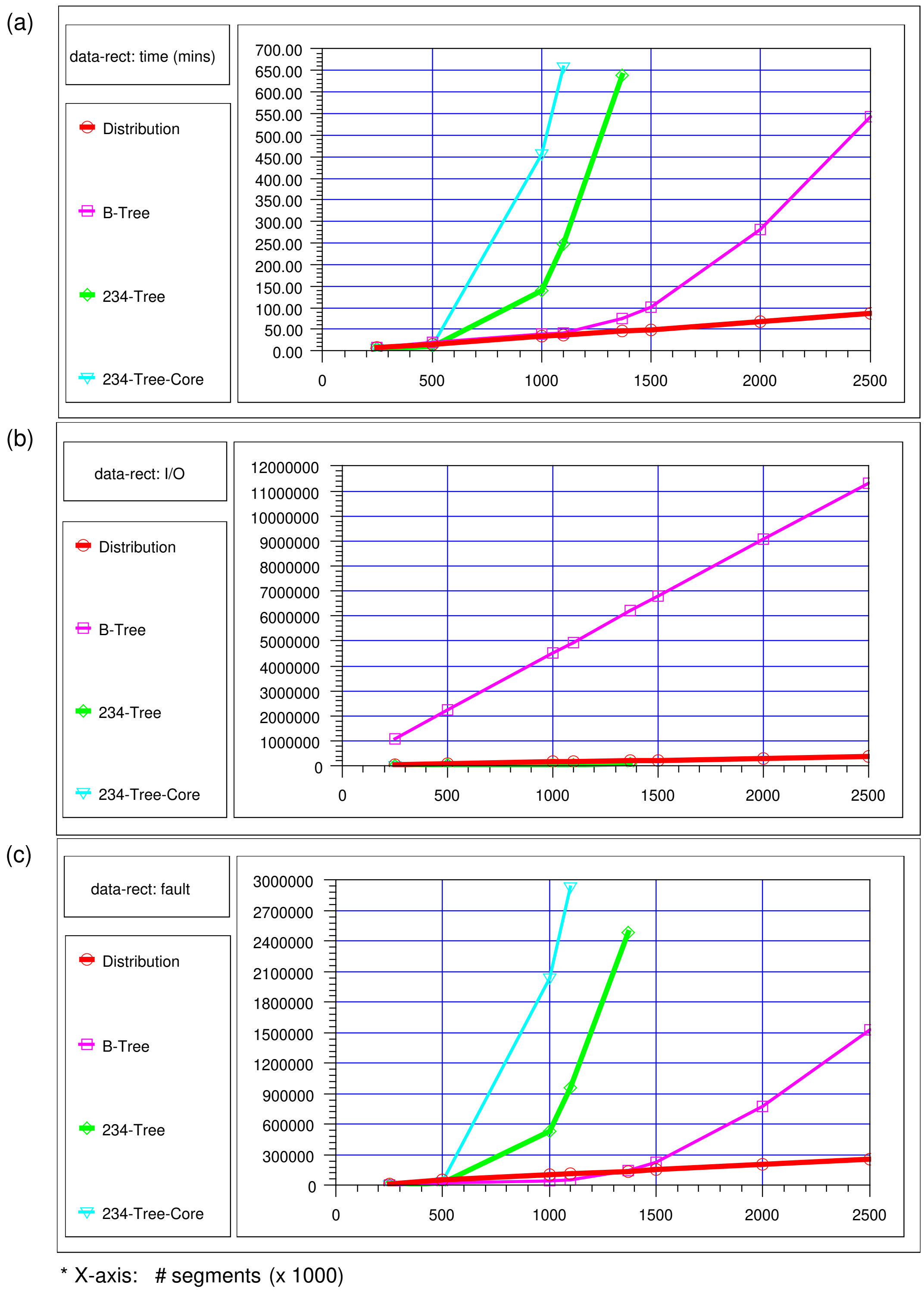


Figure 5.7: The results for the algorithms running on data set `data-rect`: (a) average running times in minutes; (b) exact numbers of I/O operations; (c) average numbers of page faults. We run `234-Tree-Core` only up to $N = 1.1 \times 10^6$ and `234-Tree` only up to $N = 1.37 \times 10^6$ since at these points they already take times much longer than the others even at $N = 2.5 \times 10^6$.

5.5 Conclusion

We have presented an experimental study comparing the performance of four algorithms for the orthogonal segment intersection problem. The observed behavior of the algorithms shows that the performance of distribution sweep is both steady and efficient. Also, it does not require a large amount of main memory to perform well. In addition, it is shown that the performance resulting from the strategy of letting the OS handle page faults and not explicitly considering the I/O cost is very undesirable. We conclude that one should make a full control of the I/O behavior of the program to obtain a good performance guarantee.

Bibliography

[1] P. K. Agarwal and M. Sharir. Applications of a new partition scheme. *Discrete Comput. Geom.*, 9:11–38, 1993.

[2] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] Nancy M. Amato. An optimal algorithm for finding the separation of simple polygons. In *Proc. 3rd Workshop Algorithms Data Struct.*, volume 709 of *Lecture Notes in Computer Science*, pages 48–59. Springer-Verlag, 1993.

[4] R. J. Anderson and G. L. Miller. A simple randomized parallel algorithm for list-ranking. *Info. Proc. Letters*, 33(5):269–273, 1990.

[5] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures (to appear)*, 1995.

[6] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. In *Proc. European Symp. Algorithms (to appear)*, 1995.

[7] E. M. Arkin, J. S. B. Mitchell, and S. Suri. Optimal link path queries in a simple polygon. In *Proc. 3rd ACM-SIAM Sympos. Discrete Algorithms*, pages 269–279, 1992.

[8] H. Baumgarten, H. Jung, and K. Mehlhorn. Dynamic point location in general subdivisions. In *Proc. 3rd ACM-SIAM Sympos. Discrete Algorithms*, pages 250–258, 1992.

[9] R. Bayer and E. McCreight. Organization of large ordered indexes. *Acta Inform.*, 1:173–189, 1972.

[10] S. W. Bent, D. D. Sleator, and R. E. Tarjan. Biased search trees. *SIAM J. Comput.*, 14:545–568, 1985.

[11] J. L. Bentley. Experiments on traveling salesman heuristics. In *Proc. 1st ACM-SIAM Sympos. Discrete Algorithms*, pages 91–99, 1990.

[12] J. L. Bentley. K-d trees for semidynamic point sets. In *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, pages 187–197, 1990.

[13] J. L. Bentley. Tools for experiments on algorithms. In *Proc. CMU 25th Anniversary Symp.*, 1990.

[14] J. L. Bentley. Fast algorithms for geometric traveling salesman problems. *ORSA J. Comput.*, 4(4):387–411, 1992.

[15] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. Technical report, Institue for Advanced Computer Studies, Univ. of Maryland, College Park, 1990.

[16] G. Bilardi and F. P. Preparata. Probabilistic analysis of a new geometric searching technique. *unpublished manuscript*, 1981.

[17] P. Callahan, M. T. Goodrich, and K. Ramaiyer. Topology B-trees and their applications. In *Proc. Workshop on Algorithms and Data Structures (to appear)*, 1995.

[18] V. Chandru, S. K. Ghosh, A. Maheshwari, V. T. Rajan, and S. Saluja. NC-algorithms for minimum link path and related problems. Technical Report CS-90/3, TATA inst., Bombay, India, 1990.

[19] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.*, 6:485–524, 1991.

[20] B. Chazelle and D. P. Dobkin. Intersection of convex objects in two and three dimensions. *J. ACM*, 34:1–27, 1987.

[21] B. Chazelle and L. J. Guibas. Visibility and intersection problems in plane geometry. *Discrete Comput. Geom.*, 4:551–581, 1989.

[22] S. W. Cheng and R. Janardan. Space-efficient ray shooting and intersection searching: algorithms, dynamization and applications. In *Proc. 2nd ACM-SIAM Sympos. Discrete Algorithms*, pages 7–16, 1991.

[23] S. W. Cheng and R. Janardan. New results on dynamic planar point location. *SIAM J. Comput.*, 21:972–999, 1992.

[24] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. In *Proc. Workshop on Algorithms and Data Structures (to appear)*, 1995.

[25] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. 6th ACM-SIAM Symp. on Discrete Algorithms*, pages 139–149, 1995.

[26] Y.-J. Chiang, F. P. Preparata, and R. Tamassia. A unified approach to dynamic point location, ray shooting, and shortest paths in planar maps. *SIAM J. Comput.*, to appear. *Prelim. version*: In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 44–53, 1993.

[27] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proc. IEEE*, 80(9):1412–1434, September 1992.

[28] Y.-J. Chiang and R. Tamassia. Dynamization of the trapezoid method for planar point location in monotone subdivisions. *Internat. J. Comput. Geom. Appl.*, 2(3):311–333, 1992. *Prelim. version*: In *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, pages 61–70, 1991.

[29] Y.-J. Chiang and R. Tamassia. Optimal shortest path and minimum-link path queries between two convex polygons inside a simple polygonal obstacle. *Internat. J. Comput. Geom. Appl.*, 1995. *Prelim. version*: In *Proc. 2nd Annu. European Sympos. Algorithms (ESA '94)*, volume 855 of *Lecture Notes in Computer Science*, pages 266–277. Springer-Verlag, 1994.

[30] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using PQ-trees. *J. Comput. Syst. Sci.*, 30(1):54–76, 1985.

[31] F. Chin and C. A. Wang. Optimal algorithms for the intersection and the minimum distance problems between planar polygons. *IEEE Trans. Comput.*, C-32(12):1203–1207, 1983.

[32] F. Y. Chin, J. Lam, and I. Chen. Efficient parallel algorithms for some graph problems. *Comm. of the ACM*, 25(9):659–665, 1982.

[33] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal list-ranking. *Information and Control*, 70(1):32–53, 1986.

[34] D. Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11:121–137, 1979.

[35] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press,

Cambridge, Mass., 1990.

[36] Thomas H. Cormen. *Virtual Memory for Data Parallel Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992.

[37] Thomas H. Cormen. Fast permuting in disk arrays. *Journal of Parallel and Distributed Computing*, 17(1–2):41–57, Jan./Feb. 1993.

[38] Thomas H. Cormen, Thomas Sundquist, and Leonard F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. Technical Report PCS-TR94-223, Dartmouth College Dept. of Computer Science, July 1994.

[39] M. de Berg. On rectilinear link distance. *Comput. Geom. Theory Appl.*, 1(1):13–34, July 1991.

[40] M. de Berg, M. van Kreveld, B. J. Nilsson, and M. H. Overmars. Finding shortest paths in the presence of orthogonal obstacles using a combined L_1 and link metric. In *Proc. 2nd Scand. Workshop Algorithm Theory*, volume 447 of *Lecture Notes in Computer Science*, pages 213–224. Springer-Verlag, 1990.

[41] G. Di Battista and R. Tamassia. Algorithms for plane representations of acyclic digraphs. *Theoret. Comput. Sci.*, 61:175–198, 1988.

[42] G. Di Battista and R. Tamassia. Incremental planarity testing. In *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 436–441, 1989.

[43] G. Di Battista, R. Tamassia, and I. G. Tollis. Area requirement and symmetry display of planar upward drawings. *Discrete Comput. Geom.*, 7:381–401, 1992.

[44] H. N. Djidjev, A. Lingas, and J.-R. Sack. An $O(n \log n)$ algorithm for computing the link center of a simple polygon. *Discrete Comput. Geom.*, 8:131–152, 1992.

[45] D. P. Dobkin and D. G. Kirkpatrick. Determining the separation of preprocessed polyhedra – a unified approach. In *Proc. 17th Internat. Colloq. Automata Lang. Program.*, volume 443 of *Lecture Notes in Computer Science*, pages 400–413. Springer-Verlag, 1990.

[46] D. P. Dobkin and R. J. Lipton. Multidimensional searching problems. *SIAM J. Comput.*, 5:181–186, 1976.

[47] M. Edahiro, I. Kokubo, and Ta. Asano. A new point-location algorithm and its practical efficiency: comparison with existing algorithms. *ACM Trans. Graph.*, 3:86–109, 1984.

[48] H. Edelsbrunner. Computing the extreme distances between two convex polygons. *J. Algorithms*, 6:213–224, 1985.

[49] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15:317–340, 1986.

[50] Esteban Feuerstein and Alberto Marchetti-Spaccamela. Memory paging for connectivity and path problems in graphs. In *Proc. Int. Symp. on Algorithms and Comp.*, 1993.

[51] P. G. Franciosa and M. Talamo. Orders, implicit k-sets representation and fast halfplane searching. In *Proc. Workshop on Orders, Algorithms and Applications (ORDAL'94)*, pages 117–127, 1994.

[52] G. N. Frederickson. A data structure for dynamically maintaining rooted trees. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 175–184, 1993.

[53] O. Fries. Zerlegung einer planaren Unterteilung der Ebene und ihre Anwendungen. M.S. thesis, Inst. Angew. Math. Inform., Univ. Saarlandes, Saarbrücken, West Germany, 1985.

[54] O. Fries, K. Mehlhorn, and S. Näher. Dynamization of geometric data structures. In *Proc.*

1st Annu. ACM Sympos. Comput. Geom., pages 168–176, 1985.

[55] S. K. Ghosh. Computing visibility polygon from a convex set and related problems. *J. Algorithms*, 12:75–95, 1991.

[56] S. K. Ghosh and A. Maheshwari. Parallel algorithms for all minimum link paths and link center problems. In *Proc. 3rd Scand. Workshop Algorithm Theory*, volume 621 of *Lecture Notes in Computer Science*, pages 106–117. Springer-Verlag, 1992.

[57] M. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. In *Proc. 23rd Annu. ACM Sympos. Theory Comput.*, pages 523–533, 1991.

[58] M. T. Goodrich, M. H. Nodine, and J. S. Vitter. Blocking for external graph searching. In *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Sys.*, pages 222–232, 1993.

[59] M. T. Goodrich and R. Tamassia. Dynamic ray shooting and shortest paths via balanced geodesic triangulations. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 318–327, 1993.

[60] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *IEEE Foundations of Comp. Sci.*, pages 714–723, 1993.

[61] M. T. Goodrich, D. E. Vengroff, and J. S. Vitter. Personal communication, 1994.

[62] E. F. Grove. Personal communication, 1994.

[63] L. J. Guibas and J. Hershberger. Optimal shortest path queries in a simple polygon. *J. Comput. Syst. Sci.*, 39:126–152, 1989.

[64] J. JáJá. *An Introduction to Parallel Algorithms.* Addison Wesley, New York, NY, 1992.

[65] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for data models with constraints and classes. In *Proc. ACM Symp. on Principles of Database Sys.*, pages 233–243, 1993.

[66] D. R. Karger. Global min-cuts in RNC and other ramifications of a simple mincut algorithm. In *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms*, pages 21–30, 1993.

[67] Y. Ke. An efficient algorithm for link-distance problems. In *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pages 69–78, 1989.

[68] D. Kelly and I. Rival. Planar lattices. *Canad. J. Math.*, 27(3):636–665, 1975.

[69] C. M. Kenyon-Mathieu and J. S. Vitter. The maximum size of dynamic data structures. *SIAM J. Comput.*, 20:807–823, 1991.

[70] V. King. A simpler minimum spanning tree verification algorithm. In *Proc. Workshop on Algorithms and Data Structures (to appear)*, 1995.

[71] D. G. Kirkpatrick. Efficient computation of continuous skeletons. In *Proc. 20th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 18–27, 1979.

[72] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12:28–35, 1983.

[73] D. G. Kirkpatrick and S. K. Wismath. Weighted visibility graphs of bars and related flow problems. In *Proc. 1st Workshop Algorithms Data Struct.*, volume 382 of *Lecture Notes in Computer Science*, pages 325–334. Springer-Verlag, 1989.

[74] P. Klein and R.E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. In *Proc. ACM Symp. on Theory of Computing*, 1994.

[75] D. T. Lee and F. P. Preparata. Location of a point in a planar subdivision and its applications. *SIAM J. Comput.*, 6:594–606, 1977.

[76] D. T. Lee and F. P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *Networks*, 14:393–410, 1984.

[77] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In *Theory of Graphs, Int. Symp. (Rome, 1966)*, pages 215–232. Gordon and Breach, New York, 1967.

[78] W. Lenhart, R. Pollack, J.-R. Sack, R. Seidel, M. Sharir, S. Suri, G. T. Toussaint, S. Whitesides, and C. K. Yap. Computing the link center of a simple polygon. *Discrete Comput. Geom.*, 3:281–293, 1988.

[79] A. Lingas, A. Maheshwari, and J.-R. Sack. Parallel algorithms for rectilinear link distance problems. In *Proc. 7th IEEE Internat. Parallel Process. Sympos.* IEEE Computer Society, 1993.

[80] Y. Maon, B. Schieber, and U. Vishkin. Parallel ear decomposition search and st-numbering in graphs. *Theoretical Computer Science*, 47(3):277–296, 1986.

[81] K. Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, Heidelberg, Germany, 1984.

[82] J. S. B. Mitchell, C. Piatko, and E. M. Arkin. Computing a shortest k-link path in a polygon. In *Proc. 33rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 573–582, 1992.

[83] J. S. B. Mitchell, G. Rote, and G. Woeginger. Minimum-link paths among obstacles in the plane. *Algorithmica*, 8:431–459, 1992.

[84] B. J. Nilsson and S. Schuierer. An optimal algorithm for the rectilinear link center of a rectilinear polygon. In *Proc. 2nd Workshop Algorithms Data Struct.*, volume 519 of *Lecture Notes in Computer Science*, pages 249–260. Springer-Verlag, 1991.

[85] Bengt J. Nilsson and Sven Schuierer. Computing the rectilinear link diameter of a polygon. In *Computational Geometry — Methods, Algorithms and Applications: Proc. Internat. Workshop Comput. Geom. CG '91*, volume 553 of *Lecture Notes in Computer Science*, pages 203–215. Springer-Verlag, 1991.

[86] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, June 1993.

[87] M. H. Nodine and J. S. Vitter. Paradigms for optimal sorting with multiple disks. In *Proc. of the 26th Hawaii Int. Conf. on Systems Sciences*, January 1993.

[88] R. H. J. M. Otten and J. G. van Wijk. Graph representations in interactive layout design. In *Proc. IEEE Internat. Sympos. on Circuits and Systems*, pages 914–918, 1978.

[89] M. H. Overmars. Range searching in a set of line segments. In *Proc. 1st Annu. ACM Sympos. Comput. Geom.*, pages 177–185, 1985.

[90] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.*, 23:166–204, 1981.

[91] T. Ozawa and H. Takahashi. A graph-planarization algorithm and its applications to random graphs. In *Graph Theory and Algorithms*, volume 108 of *Lecture Notes in Computer Science*, pages 95–107. Springer-Verlag, Berlin, 1981.

[92] F. P. Preparata. A new approach to planar point location. *SIAM J. Comput.*, 10:473–482, 1981.

[93] F. P. Preparata and M. I. Shamos. *Computational Geometry: an Introduction*. Springer-Verlag, New York, NY, 1985.

[94] F. P. Preparata and R. Tamassia. Fully dynamic point location in a monotone subdivision. *SIAM J. Comput.*, 18:811–830, 1989.

[95] F. P. Preparata and R. Tamassia. Dynamic planar point location with optimal query time. *Theoret. Comput. Sci.*, 74:95–114, 1990.

[96] F. P. Preparata, J. S. Vitter, and M. Yvinec. Computation of the axial view of a set of isothetic parallelepipeds. *ACM Trans. Graph.*, 9:278–300, 1990.

[97] S. Ramaswamy and P. C. Kanellakis. OODB indexing by class-division. In *Proc. ACM SIGMOD Internat. Conf. on Management of Data*, pages 139–150, 1995.

[98] S. Ramaswamy and S. Subramanian. Path caching: A technique for optimal external searching. In *Proc. ACM Symp. on Principles of Database Sys.*, pages 25–35, 1994.

[99] J. H. Reif and S. Sen. An efficient output-sensitive hidden-surface removal algorithms and its parallelization. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pages 193–200, 1988.

[100] J. H. Reif and J. A. Storer. Minimizing turns for discrete movement in the interior of a polygon. *IEEE J. Robot. Autom.*, pages 182–193, 1987.

[101] I. Rival and J. Urrutia. Representing orders by translating convex figures in the plane. *Order*, 4:319–339, 1988.

[102] P. Rosenstiehl and R. E. Tarjan. Rectilinear planar layouts and bipolar orientations of planar graphs. *Discrete Comput. Geom.*, 1(4):343–353, 1986.

[103] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29:669–679, 1986.

[104] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 24:362–381, 1983.

[105] J. A. Storer. On minimal node-cost planar embeddings. *Networks*, 14:181–212, 1984.

[106] S. Subramanian and S. Ramaswamy. The P-range tree: A new data structure for range searching in secondary memory. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 378–387, 1995.

[107] S. Suri. A linear time algorithm for minimum link paths inside a simple polygon. *Comput. Vision Graph. Image Process.*, 35:99–110, 1986.

[108] S. Suri. *Minimum link paths in polygons and related problems.* Ph.D. thesis, Dept. Comput. Sci., Johns Hopkins Univ., Baltimore, MD, 1987.

[109] S. Suri. On some link distance problems in a simple polygon. *IEEE Trans. Robot. Autom.*, 6:108–113, 1990.

[110] R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, 1987.

[111] R. Tamassia. A dynamic data structure for planar graph embedding. In T. Lepisto and A. Salomaa, editors, *Automata, Languages and Programming (Proc. 15th ICALP)*, volume 317 of *Lecture Notes in Computer Science*, pages 576–590. Springer-Verlag, 1988.

[112] R. Tamassia. An incremental reconstruction method for dynamic planar point location. *Inform. Process. Lett.*, 37:79–83, 1991.

[113] R. Tamassia and I. G. Tollis. A unified approach to visibility representations of planar graphs. *Discrete Comput. Geom.*, 1(4):321–341, 1986.

[114] R. Tamassia and I. G. Tollis. Representations of graphs on a cylinder. *SIAM J. Discrete Math.*, 4(1):139–149, 1991.

[115] R. Tamassia and J. S. Vitter. Parallel transitive closure and point location in planar structures. *SIAM J. Comput.*, 20(4):708–725, 1991.

[116] R.E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. *SIAM J. Computing*, 14(4):862–874, 1985.

[117] J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intellegence*, 3:331–360, 1991.

[118] D. E. Vengroff. Personal communication, 1994.

[119] D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. Manuscript, 1995.

[120] Darren Erik Vengroff. A transparent parallel I/O environment. In *Proc. 1994 DAGS Symposium on Parallel Computation*, July 1994.

[121] U. Vishkin. Personal communication, 1992.

[122] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2), 1994.

[123] S. Wimer, I. Koren, and I. Cederbaum. Floorplans, planar graphs, and layouts. *IEEE Trans. on Circuits and Systems*, 35(3):267–278, 1988.

[124] S. K. Wismath. Characterizing bar line-of-sight graphs. In *Proc. 1st Annu. ACM Sympos. Comput. Geom.*, pages 147–152, 1985.

[125] B. Zhu. Further computational geometry in secondary memory. In *Proc. Int. Symp. on Algorithms and Computation*, 1994.