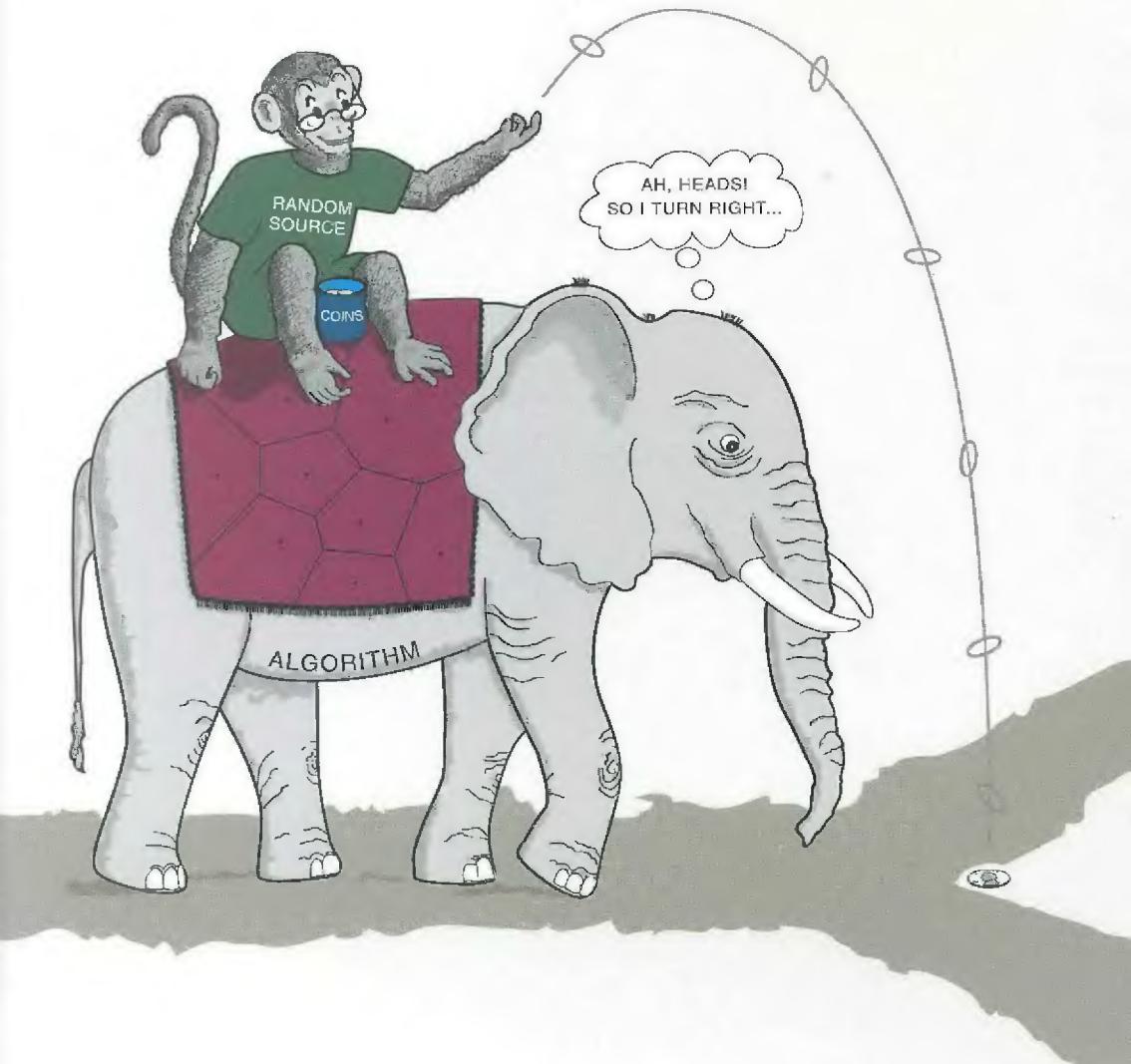


Computational Geometry

An Introduction Through
Randomized Algorithms

Ketan Mulmuley



Computational Geometry

An Introduction Through Randomized Algorithms

Ketan Mulmuley
The University of Chicago



PRENTICE HALL, Englewood Cliffs, NJ 07632

Library of Congress Cataloging-in-Publication Data

Mulmuley, Ketan.

Computational geometry: an introduction through randomized algorithms / Ketan Mulmuley.

p. cm.

Includes bibliographical references and index.

ISBN 0-13-336363-5

1. Geometry--Data processing. 2. Algorithms. I. Title.

QA448.M85 1994

516'.13--dc20

93-3138

CIP

Acquisitions editor: BILL ZOBRIST

Production editor: JOE SCORDATO

Copy editor: PETER J. ZURITA

Prepress buyer: LINDA BEHRENS

Manufacturing buyer: DAVID DICKEY

Editorial assistant: DANIELLE ROBINSON



© 1994 by Prentice-Hall, Inc.
A Simon & Schuster Company
Englewood Cliffs, New Jersey 07632

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-336363-5

Prentice-Hall International (UK) Limited, London

Prentice-Hall of Australia Pty. Limited, Sydney

Prentice-Hall Canada Inc., Toronto

Prentice-Hall Hispanoamericana, S.A., Mexico

Prentice-Hall of India Private Limited, New Delhi

Prentice-Hall of Japan, Inc., Tokyo

Simon & Schuster Asia Pte. Ltd., Singapore

Editora Prentice-Hall do Brasil, Ltda., Rio de Janeiro

For Sanchit

Contents

Preface	ix
Notation	xvi
I Basics	1
1 Quick-sort and search	3
1.1 Quick-sort	4
1.2 Another view of quick-sort	5
1.3 Randomized binary trees	7
1.3.1 Dynamization	11
1.3.2 Another interpretation of rotations	13
1.4 Skip lists	18
2 What is computational geometry?	26
2.1 Range queries	28
2.2 Arrangements	29
2.3 Trapezoidal decompositions	32
2.4 Convex polytopes	36
2.4.1 Duality	40
2.5 Voronoi diagrams	46
2.6 Hidden surface removal	51
2.7 Numerical precision and degeneracies	52
2.8 Early deterministic algorithms	55
2.8.1 Planar convex hulls	56
2.8.2 Arrangements of lines	58
2.8.3 Trapezoidal decompositions	61
2.8.4 Planar Voronoi diagrams	67
2.8.5 Planar point location	74
2.9 Deterministic vs. randomized algorithms	78

2.10	The model of randomness	78
3	Incremental algorithms	81
3.1	Trapezoidal decompositions	84
3.1.1	History	90
3.1.2	Planar graphs	94
3.1.3	Simple polygons*	94
3.2	Convex polytopes	96
3.2.1	Conflict maintenance	99
3.2.2	History	103
3.2.3	On-line linear programming	104
3.3	Voronoi diagrams	106
3.4	Configuration spaces	111
3.5	Tail estimates*	120
4	Dynamic algorithms	126
4.1	Trapezoidal decompositions	129
4.1.1	Point location	132
4.2	Voronoi diagrams	135
4.2.1	Conflict search	137
4.3	History and configuration spaces	140
4.3.1	Expected performance*	142
4.4	Rebuilding history	149
4.5	Deletions in history	151
4.5.1	3D convex polytopes	158
4.5.2	Trapezoidal decompositions	162
4.6	Dynamic shuffling	167
5	Random sampling	173
5.1	Configuration spaces with bounded valence	176
5.2	Top-down sampling	181
5.2.1	Arrangements of lines	182
5.3	Bottom-up sampling	184
5.3.1	Point location in arrangements	185
5.4	Dynamic sampling	192
5.4.1	Point location in arrangements	193
5.5	Average conflict size	197
5.6	More dynamic algorithms	201
5.6.1	Point location in trapezoidal decompositions	204
5.6.2	Point location in Voronoi diagrams	210
5.7	Range spaces and ϵ -nets	215
5.7.1	VC dimension of a range space	221

5.8 Comparisons	223
II Applications	227
6 Arrangements of hyperplanes	229
6.1 Incremental construction	232
6.2 Zone Theorem	234
6.3 Canonical triangulations	238
6.3.1 Cutting Lemma	241
6.4 Point location and ray shooting	242
6.4.1 Static setting	242
6.4.2 Dynamization	248
6.5 Point location and range queries	250
6.5.1 Dynamic maintenance	253
7 Convex polytopes	260
7.1 Linear programming	262
7.2 The number of faces	267
7.2.1 Dehn–Sommerville relations	268
7.2.2 Upper bound theorem: Asymptotic form	271
7.2.3 Upper bound theorem: Exact form	271
7.2.4 Cyclic polytopes	274
7.3 Incremental construction	276
7.3.1 Conflicts and linear programming	278
7.3.2 Conflicts and history	279
7.4 The expected structural and conflict change	280
7.5 Dynamic maintenance	284
7.6 Voronoi diagrams	285
7.7 Search problems	286
7.7.1 Vertical ray shooting	288
7.7.2 Half-space range queries	289
7.7.3 Nearest k -neighbor queries	291
7.7.4 Dynamization	292
7.8 Levels and Voronoi diagrams of order k	294
7.8.1 Incremental construction	301
7.8.2 Single level	306
8 Range search	311
8.1 Orthogonal intersection search	311
8.1.1 Randomized segment tree	312
8.1.2 Arbitrary dimension	317

8.2	Nonintersecting segments in the plane	322
8.3	Dynamic point location	327
8.4	Simplex range search	328
8.4.1	Partition theorem	332
8.4.2	Preprocessing time	336
8.5	Half-space range queries	338
8.5.1	Partition theorem	341
8.5.2	Half-space emptiness	343
8.6	Decomposable search problems	345
8.6.1	Dynamic range queries	348
8.7	Parametric search	349
9	Computer graphics	358
9.1	Hidden surface removal	361
9.1.1	Analysis	367
9.2	Binary Space Partitions	372
9.2.1	Dimension two	372
9.2.2	Dimension three	379
9.3	Moving viewpoint	383
9.3.1	Construction of a cylindrical partition	391
9.3.2	Randomized incremental construction	392
10	How crucial is randomness?	398
10.1	Pseudo-random sources	399
10.2	Derandomization	410
10.2.1	ϵ -approximations	412
10.2.2	The method of conditional probabilities	413
10.2.3	Divide and conquer	417
A	Tail estimates	422
A.1	Chernoff's technique	423
A.1.1	Binomial distribution	424
A.1.2	Geometric distribution	426
A.1.3	Harmonic distribution	428
A.2	Chebychev's technique	429
Bibliography		431
Index		442

Preface

This book is based on lectures given to graduate students at the University of Chicago. It is intended to provide a rapid and concise introduction to computational geometry. No prior familiarity with computational geometry is assumed. A modest undergraduate background in computer science or a related field should suffice.

My goal is to describe some basic problems in computational geometry and the simplest known algorithms for them. It so happens that several of these algorithms are randomized. That is why we have chosen randomized methods to provide an introduction to computational geometry. There is another feature of randomized methods that makes them ideal for this task: They are all based on a few basic principles, which can be applied systematically to a large number of apparently dissimilar problems. Thus, it becomes possible to provide through randomized algorithms a unified, broad perspective of computational geometry. I have tried to give an account that brings out this simplicity and unity of randomized algorithms and also their depth.

Randomization entered computational geometry with full force only in the 80s. Before that the algorithms in computational geometry were mostly deterministic. We do cover some of the very basic, early deterministic algorithms. Their study will provide the reader an historical perspective and familiarity with some deterministic paradigms that every student of computational geometry must know. It will also provide the reader an opportunity to study the relationship between these deterministic algorithms and their randomized counterparts. This relationship is quite akin to the relationship between quick-sort and deterministic sorting algorithms: Randomization yields simplicity and efficiency at the cost of losing determinism.

But randomization does more than just provide simpler alternatives to the deterministic algorithms. Its role in the later phase of computational geometry was pivotal. For several problems, there are no deterministic algorithms that match the performance of randomized algorithms. For several others, the only known deterministic algorithms are based on a technique called derandomization. Later in the book, we study the basic principles underlying

this technique. It consists of an ingenious simulation of a randomized algorithm in a deterministic fashion. It has made the study of randomized algorithms even more important.

Of course, there are also numerous problems in computational geometry for which randomization offers no help. We have not touched upon these problems in this book. Our goal is to provide a cohesive and unified account rather than a comprehensive one. Even the field of randomized algorithms has become so vast that it is impossible to cover it comprehensively within a book of this size. Finally, the choice of topics is subjective. But I hope that the book will provide the reader with a glimpse of the exciting developments in computational geometry. Once the vistas of this terrain are roughly illuminated, it is hoped that the reader will be provided with the perspective necessary to delve more deeply into the subfields of his or her choice.

Organization of the book

Figure 0.1 shows logical dependence among the various chapters.

The book is organized in two parts—basics and applications. In the first part, we describe the basic principles that underlie the randomized algorithms in this book. These principles are illustrated with the help of simple two-dimensional geometric problems. The first part uses only elementary planar geometry. It can also be easily read by those who are mainly interested in randomized algorithms rather than in computational geometry.

In the applications part of the book, we apply the basic principles developed in the first part to several higher-dimensional problems. Here the geometry becomes more interesting. The chapters in this part are independent of each other. Thus, one can freely select and read the chapters of interest.

If the reader wishes, Chapter 2 can be read after reading just the introduction of Chapter 1. But the rest of Chapter 1 should be read before proceeding further. Chapters 1, 2, 3, and 5 should be read before proceeding to the applications part. Chapter 4 is optional, and can be skipped in the first reading after reading its introduction. This requires skipping Section 7.5 and some exercises that depend on it. The other dynamic algorithms in the book are based on the principles described in Chapter 5.

It should be noted that all randomized algorithms in this book are simple, without any exception. But sometimes their analysis is not so simple. If you are not theoretically inclined, you may skip the probabilistic analysis—which is always separate—on the first reading. But, eventually, we hope, you will wish to know why these algorithms work so well and hence turn to their analysis.

Prerequisites

This book is meant to be understandable to those with only a modest undergraduate background in computer science or a related field. It is assumed that the reader is familiar with elementary data structures such as lists, trees, graphs, and arrays. Familiarity with at least one nontrivial data structure, such as a balanced search tree, will help. But strictly speaking even that is not required, because Chapter 1 gives a complete description of randomized search trees. The algorithms in this book are described in plain English. It is assumed that the reader can convert them into a suitable programming language, if necessary.

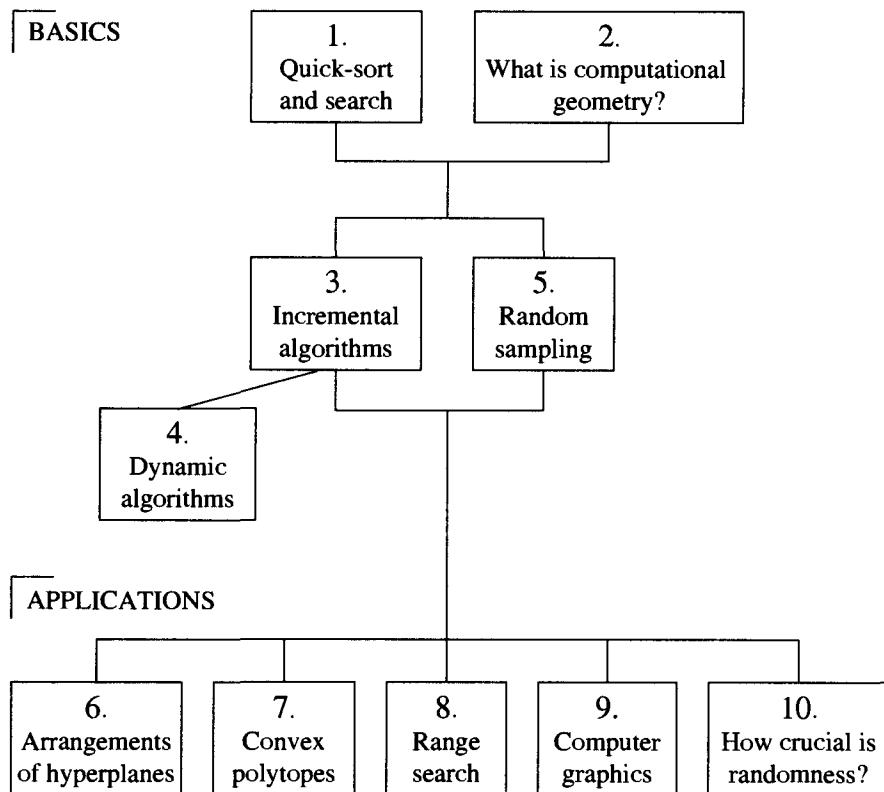


Figure 0.1: Logical dependence among the chapters.

It is also assumed that the reader is familiar with elementary notions of probability theory—random variables, their expectation, conditional expectation, and so on. We do not need anything deep from probability theory. The deeper aspects of randomized, geometric algorithms are generally geometric rather than probabilistic.

The first part of the book requires only planar geometry. The second part assumes nodding acquaintance with elementary notions in Euclidean geometry: closed set, open set, boundary of a set, linear function, and so on. Otherwise, it is meant to be self-contained.

To the teacher

The book can be used for a course on computational geometry to beginning graduate students. Such a course may begin with some basic problems and their deterministic solutions (Chapter 2) and then shift to randomized algorithms as the problems get more complex.

On the deterministic side, the book contains only some very basic algorithms, which can be used for the first course in computational geometry. In two dimensions, we cover roughly as much ground as in, say, the book by Preparata and Shamos [185], but in higher dimensions, we go little further to address some additional deterministic techniques: incremental construction (Section 6.1), parametric search (Section 8.7), dynamization of decomposable search problems (Section 8.6), and derandomization (Section 10.2). Also, the algorithms for orthogonal intersection search and dynamic planar point location in Chapter 8 are almost deterministic. The only difference is that I have substituted the weight-balanced trees in their solutions with skip lists. So, if you wish, you can cover the deterministic versions, too, after covering the simpler randomized versions.

On the side of randomized algorithms, the book contains more than what can be reasonably covered in one semester course. The choice of material would depend on the inclination of the students. In a leisurely course to students who are not theoretically inclined, you may wish to cover only the basic algorithms in the beginnings of Chapters 3 and 5. This assumes very little mathematical sophistication. In an intensive course to theoretically inclined graduate students, you can cover most of Part I, but perhaps skip Chapter 4, and cover selected topics in Part II. Since the chapters in Part II are independent, the choice is flexible.

The book can also be used as a supplement for a course on randomized algorithms.

Outline

Here is a brief outline of the book, chapter by chapter.

In Chapter 1, we begin with the simplest and the most well-known randomized algorithm: quick-sort. It is of special interest to us, because the randomized methods in computational geometry can be viewed as higher-dimensional generalizations of quick-sort. Quick-sort is known to be one of the simplest and the most efficient general-purpose algorithms for sorting. This indicates why its higher-dimensional analogs can be expected to be simple and efficient in practice. It is our hope that this book can help a practitioner get acquainted with some theoretical principles that may turn out to be useful in practice. With that in mind, we have chosen quick-sort as a starting point. Here, we analyze and interpret quick-sort in several ways, some of which are standard and some are not. These various interpretations are systematically extended later in the book to attack a large number of problems.

Chapter 2 gives a snapshot of computational geometry. It describes several of the basic motivating problems in the field. These problems are tackled in a unified fashion later in the book. We also cover here some early deterministic algorithms for very basic two-dimensional problems. This should provide the reader familiarity with some simple deterministic design principles and an opportunity to compare these deterministic algorithms with their randomized counterparts studied later.

Chapter 3 deals with randomized incremental algorithms in computational geometry. They solve a problem by adding the objects in the input, one at a time, in random order. This is exactly how quick-sort proceeds, if it is viewed appropriately. We describe this paradigm in general form and then illustrate it with several simple problems dealing with planar graphs, Voronoi diagrams, and so on.

Chapter 4 deals with dynamic problems. In a dynamic setting, the user is allowed to add or delete an object in the input in an on-line fashion. We develop some general principles and demonstrate these on the same two-dimensional problems considered in Chapter 3.

Chapter 5 is central to the book. It describes the principle of random sampling. This can be thought of as an extension of the randomized divide-and-conquer view of quick-sort. We describe the general principles and then illustrate them on the two-dimensional problems considered in the earlier chapters.

Chapter 6 deals with arrangements of hyperplanes. Arrangements are important in computational geometry because, among all geometric configurations, they have perhaps the simplest combinatorial structure. Thus, they

serve as a nice test-bed for the algorithmic principles developed earlier in the book.

Chapter 7 deals with convex polytopes, convex hulls, Voronoi diagrams, and related problems. Convex polytopes and Voronoi diagrams come up in many fields—signal processing, operations research, physics, and so on. There is no need to dwell on their importance here.

Chapter 8 deals with range searching problems. Range searching is an important theme that encompasses a large number of problems in computational geometry. These problems have the following form: We are given a set of geometric objects. The goal is to build a data structure so that, given a query region, one can quickly report or count the input objects that it intersects. Chapter 8 deals with several important problems of this form.

Chapter 9 deals with the applications of randomized methods to some basic problems in computer graphics.

Finally, Chapter 10 studies how crucial the randomness is for the performance of the algorithms studied earlier in the book. We shall see that the number of random bits used by most randomized incremental algorithms can be made logarithmic in the input size without changing their expected performance by more than a constant factor. Several of the algorithms based on random sampling can be made completely deterministic by “derandomizing” them. A comprehensive account of such deterministic algorithms is outside the scope of this book. However, we describe the basic principles underlying this technique.

There are several exercises throughout the book. The reader is encouraged to solve as many as possible. Some simple exercises are used in the book. These are mostly routine. Quite often, we defer the best known solution to a given problem to the exercises, if this solution is too technical. The exercises marked with * are difficult—the difficulty increases with the number of stars. The exercises marked with † are unsolved at the time of this writing. Some sections in the book are starred. They can be skipped on the first reading.

The bibliography at the end is far from being comprehensive. We have mainly confined ourselves to the references that bear directly on the methods covered in this book. For the related topics not covered in this book, we have only tried to provide a few references that can be used as a starting point; in addition, we also suggest for this purpose [63, 174, 180, 199, 231].

As for the exercises in the book, some of them are new and some are just routine or standard. For the remaining exercises, I have provided explicit references unless they directly continue or extend the material covered in the sections containing them (this should be apparent), in which case, the reference for the containing section is meant to be applicable for the exercise, too, unless mentioned otherwise.

Acknowledgments

I am very grateful to the Computer Science Department in the University of Chicago for providing a stimulating atmosphere for the work on this book. I wish to thank the numerous researchers and graduate students who shared their insights with me. I am especially grateful to the graduate students who took the courses based on this book and provided numerous criticisms, in particular, Stephen Crocker, L. Satyanarayana, Sundar Vishwanathan, and Victoria Zanko. Sundar provided a great help in the preparation of the appendix. I also wish to thank Bernard Chazelle, Alan Frieze, Jirka Matoušek, Joe Mitchell and Rajeev Motwani for their helpful criticisms. The work on this book was supported by a Packard Fellowship. I wish to thank the Packard Foundation for its generous support. It is a pleasure to thank Adam Harris who did the artwork, including the art on the cover, and provided every kind of support during the typesetting of this book. I wish to thank Sam Rebelsky for customizing LaTeX, Bill Zobrist and Joe Scordato of Prentice Hall for their enthusiasm and cooperation. Finally, I wish to thank my wife Manju for her endurance while the book was being written. I should also acknowledge the contribution of my son Sanchit; he insisted on having his favorite animals on the cover.

Ketan Mulmuley

Notation

$f(n) = O(g(n))$	$f(n) < cg(n)$, for some constant $c > 0$.
$f(n) = \Omega(g(n))$	$f(n) > cg(n)$, for some constant $c > 0$.
$f(n) \approx g(n)$	Asymptotic equality, i.e., $f(n) = O(g(n))$ and $g(n) = O(f(n))$.
$f(n) = \tilde{O}(g(n))$	$f(n) = O(g(n))$ with high probability. This means, for some constant $c > 0$, $f(n) < cg(n)$ with probability $1 - 1/p(n)$, where $p(n)$ is a polynomial whose degree depends on c , and this degree can be made arbitrarily high by choosing c large enough. Thus, $f(n) > cg(n)$ with probability $1/p(n)$, which is minuscule.
$X \subseteq Y$	X is a subset of Y .
$X \setminus Y$	The relative complement of Y in X .
R	The real line. (On a few occasions, R has a different meaning, but this will be clear from the context.)
R^d	The d -fold product of R , i.e., the d -dimensional Euclidean space.
$ X $	The size of X . If X is an ordinary set, then this is just its cardinality. If X is a geometric partition (complex), this is the total number of its faces of all dimensions (cf. Chapter 2). Finally, if X is a real number, $ X $ is its absolute value.
$\lceil x \rceil$	The smallest integer greater than x (the ceiling function).
$\lfloor x \rfloor$	The largest integer smaller than x (the floor function).
$E[X]$	Expected value of the random variable X .
$E[X \mid ..]$	Expected value of X subject to the specified conditions.
$\text{prob}\{..\}$	Probability of the specified event.
∂Z	Boundary of the set Z .
$\min\{\dots\}$	The minimum element in the set.
$\max\{\dots\}$	The maximum element in the set.
$[x]_d$	The falling factorial $x(x-1)\cdots(x-d+1)$.
$\text{polylog}(n)$	$\log^a n$, for a fixed constant $a > 0$.

We often refer to a number which is bounded by a constant as simply a bounded number. By a random sample of a set, we mean its random subset of the specified size.

Part I

Basics

Chapter 1

Quick-sort and search

In this book, we view computational geometry as a study of sorting and searching problems in higher dimensions. In a sense, these problems are just generalizations of the ubiquitous problem in computer science—how to search and sort lists of elements drawn from an ordered universe. This latter problem can be seen as the simplest one-dimensional form of sorting and searching. This becomes clear if we reformulate it in a geometric language as follows. We are given a set N of n points on the real line R . The goal is to sort them by their coordinates. This is equivalent to (Figure 1.1):

The sorting problem: Find the partition $H(N)$ of R formed by the given set of points N .

The partition $H(N)$ is formally specified by the points in N , the resulting (open) intervals within the line R —such as J in Figure 1.1—and adjacencies among the points and the intervals. A geometric formulation of the associated search problem is the following:

The search problem: Associate a search structure $\tilde{H}(N)$ with $H(N)$ so that, given any point $q \in R$, one can locate the interval in $H(N)$ containing q quickly, i.e., in logarithmic time.

In a dynamic variant of the search problem, the set N can be changed in an on-line fashion by addition or deletion of a point. We are required to update $\tilde{H}(N)$ quickly, i.e., in logarithmic time, during each such update.



Figure 1.1: $H(N)$.

In higher dimensions, the elements of the set N are not points any more, but, rather, hyperplanes or sites or polygons and so on, depending on the problem under consideration.

The simplest methods for sorting and searching linear lists are randomized. It is no surprise that the simplest known methods for several higher-dimensional sorting and searching problems are also randomized. These methods can be thought of as generalizations of the randomized methods for sorting and searching linear lists. Hence, we begin with a review of these latter methods. Actually, our goal is more than just a review—we want to *reformulate* these methods in a geometric language and analyze them in ways that can be generalized to higher dimensions. A great insight in higher-dimensional problems is obtained by just redoing the one-dimensional case.

Remark. In the above geometric formulation of sorting and searching in lists, we assumed that the points in N were points on the line. Strictly speaking, this is not necessary. The elements in N can be arbitrary, as long as they can be linearly ordered and the comparison between any two elements in N can be carried out in constant time. All methods described in this chapter can be translated to this more general setting trivially.

1.1 Quick-sort

A simple randomized method for sorting a list of points on the real line R is *quick-sort*. It is based on the randomized divide-and-conquer paradigm. Let N be the given set of n points in R . Pick a random point $S \in N$. It divides R into two halves. Let $N_1, N_2 \subseteq N$ be the subsets of points contained in these two halves. Sort N_1 and N_2 recursively.

Intuitively, we should expect the sizes of N_1 and N_2 to be roughly equal to $n/2$. Hence, the expected depth of recursion is $O(\log n)$. This means the expected running time of the algorithm should be $O(n \log n)$. We shall give a rigorous justification for this in a moment.

One issue remains to be addressed here. In the division step, how do we choose a random point in N ? In practice, this has to be done with the help of a random number generator. But the numbers generated by these so-called random number generators are not truly random. They only “appear” random. Fortunately, we shall see in Chapter 10 that all randomized algorithms in this book work very well even when the source of randomness is not perfectly random, but only “pseudo-random”. Until then, we shall assume, for the sake of simplicity, that all our algorithms have ability to make perfectly random choices whenever necessary.

1.2 Another view of quick-sort

If we unwind the recursion in the definition of quick-sort, we get its so-called randomized incremental version. Though this version is equivalent to the version in the previous section, in higher dimensions, these two paradigms lead to markedly different algorithms.

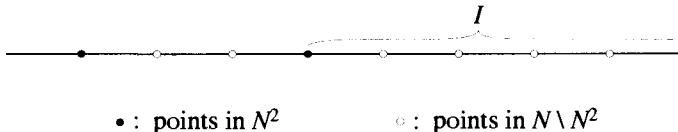


Figure 1.2: $H(N^2)$.

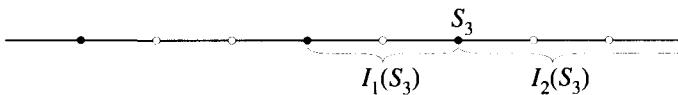


Figure 1.3: Addition of the third point.

The randomized incremental version of quick-sort works as follows. It constructs the required partition of the line incrementally by adding the points in N , one at a time, in random order. In other words, at any given time we choose a random point from the set of unadded points and add the chosen point to the existing partition. Let us elaborate this idea fully. Let N^i be the set of the first i added points. Let $H(N^i)$ be the partition of R formed by N^i . Starting with $H(N^0)$, the empty partition of R , we construct a sequence of partitions

$$H(N^0), H(N^1), H(N^2), \dots, H(N^n) = H(N).$$

At the i th stage of the algorithm, we also maintain, for each interval $I \in H(N^i)$, its *conflict list* $L(I)$. This is defined to be an *unordered* list of the points in $N \setminus N^i$ contained in I (Figure 1.2). Conversely, with each point in $N \setminus N^i$, we maintain a pointer to the conflicting interval in $H(N^i)$ containing it.

Addition of a randomly chosen point $S = S_{i+1}$ in $N \setminus N^i$ consists in splitting the interval I in $H(N^i)$ containing S , together with its conflict list.

Let $I_1(S)$ and $I_2(S)$ denote the intervals in $H(N^{i+1})$ adjacent to S (Figure 1.3). Let $l(I_1(S))$ and $l(I_2(S))$ denote their *conflict sizes*, i.e., the sizes of their conflict lists $L(I_1(S))$ and $L(I_2(S))$. It is easily seen that:

Fact 1.2.1 The cost of adding S is proportional to $l(I_1(S)) + l(I_2(S))$, ignoring an additive $O(1)$ term.

Analysis

Next, we shall estimate the expected running time of the whole algorithm. Before we do so, let us elaborate what we mean by the expected running time. Here the expectation is meant to be solely with respect to the randomization in the algorithm. No assumption is made about the distribution of the points in the input. This latter point is crucial to us. In the algorithms covered later in this book, the input set will get more complicated. The objects in the input set will not be points on the line, as is the case here. Rather, they will be higher-dimensional objects, which will depend on the problem under consideration. But the basic philosophy underlying quick-sort will prevail throughout this book: *We shall never make any assumption about how the objects in the input set are distributed in space.* The input has to be thought as user-given—or god-given if you wish—and the goal is to use randomization in the algorithm to ensure good expected performance.

Let us get back to estimating the expected running time of quick-sort. We assume that the points in N are added in random order. By symmetry, all possible sequences of addition are equally likely. We shall calculate the expected cost of the $(i+1)$ th addition. For this, we shall analyze the addition *backwards*: We shall *imagine*, just for the sake of analysis, going backwards from N^{i+1} to N^i , rather than forwards, as the algorithm does, from N^i to N^{i+1} . In other words, we shall bound the *conditional* expected cost of the $(i+1)$ th addition, for every fixed choice of N^{i+1} , not N^i . As one would expect, this bound will turn out to be dependent only on i and not on N^{i+1} . Hence, the bound will hold unconditionally as well.

So let us fix N^{i+1} arbitrarily. By the random nature of our additions, each point in N^{i+1} is equally likely to occur as S_{i+1} . In other words, one can imagine that N^i results by deleting a random point in N^{i+1} . Fact 1.2.1 implies that the expected cost of the $(i+1)$ th addition, conditional on a fixed N^{i+1} , is

$$\frac{1}{i+1} \sum_{S \in N^{i+1}} [l(I_1(S)) + l(I_2(S)) + 1] \leq \frac{2}{i+1} \sum_{J \in H(N^{i+1})} [l(J) + 1] = O\left(\frac{n}{i+1}\right).$$

Since this bound does not depend on N^{i+1} , it unconditionally bounds the expected cost of the $(i+1)$ th addition.

It follows that the expected cost of the whole algorithm is of the order of $n \sum_i 1/(i+1) = O(n \log n)$. Note that this expectation is solely with respect to the random choices made in the algorithm.

The reader should carefully examine why we estimated the expected cost of the $(i+1)$ th addition, conditional on a fixed N^{i+1} , not N^i . In other words, we imagined, *just for the sake of analysis*, going backwards from N^{i+1} to N^i by removing a random point in N^{i+1} . Each point in N^{i+1} is adjacent to

two intervals in $H(N^{i+1})$. Hence, the expected conflict size of the intervals adjacent to the removed point is proportional to the total conflict size of $H(N^{i+1})$ divided by $i + 1$. We shall use such *backward analysis* on several occasions later. Its main advantage is that S_{i+1} , the object involved in the $(i + 1)$ th step, is contained in N^{i+1} . Hence, estimating the expected cost of the $(i + 1)$ th step becomes easy, once N^{i+1} is fixed. If we were to condition the expectation on a fixed N^i instead, this advantage would be lost, because S_{i+1} does not belong to N^i .

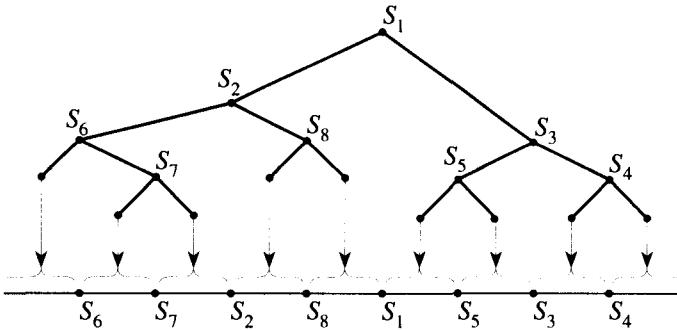


Figure 1.4: Randomized binary tree.

1.3 Randomized binary trees

Quick-sort can be readily extended so that, at the end of the algorithm, we have not just the partition $H(N)$, but also a randomized binary search tree $\tilde{H}(N)$. This can be used to locate any point $q \in R$ in $H(N)$ quickly.

If we view quick-sort as a divide-and-conquer algorithm, then we are led to the following definition of $\tilde{H}(N)$. The leaves of $\tilde{H}(N)$ will be in one-to-one correspondence with the intervals in $H(N)$.

If N is empty, $H(N)$ consists of one node, corresponding to the whole of R . Otherwise, the root of $\tilde{H}(N)$ is labeled with a randomly chosen point $S \in N$. The left and the right subtrees are defined to be the recursively defined trees $\tilde{H}(N_1)$ and $\tilde{H}(N_2)$. These trees correspond to the halves of R on the two sides of S (Figure 1.4).

Given a query point $q \in R$, we can locate the interval in $H(N)$ containing q as follows. We compare q with the point S labeling the root of $\tilde{H}(N)$. If $q = S$, we are done. If q has a smaller coordinate than S , we recur on $\tilde{H}(N_1)$. Otherwise, we recur on $\tilde{H}(N_2)$. Eventually, we will reach a leaf that corresponds to the interval in $H(N)$ containing q .

If we view quick-sort as a randomized incremental algorithm, we are led to a different definition of $\tilde{H}(N)$. This definition is equivalent to the previous

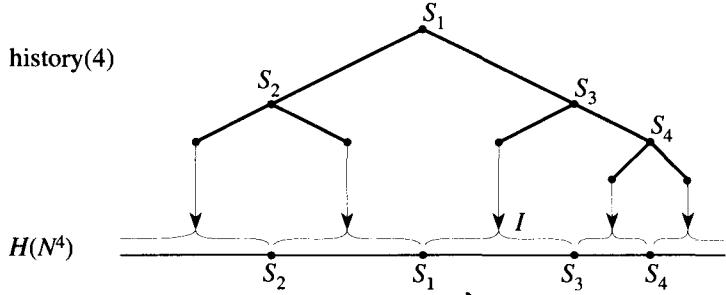


Figure 1.5: History after four additions.

one. But in higher dimensions, the two paradigms lead to quite different search structures.

So let us give an alternative definition of $\tilde{H}(N)$, viewing quick-sort as a randomized incremental algorithm. The idea is to just remember what one is doing! This leads us to define $\tilde{H}(N)$ as a certain *history* of the computed partitions

$$H(N^0), H(N^1), H(N^2), \dots, H(N^n) = H(N),$$

where N^i is the set of the first i randomly chosen points. Clearly, we cannot afford to remember each $H(N^i)$ separately. But notice that the difference between $H(N^i)$ and $H(N^{i+1})$ is small. So we only need to keep track of the differences between the successive partitions.

We shall define history(i), the history of the first i additions, by induction on i . Then, $\tilde{H}(N)$ is defined to be history(n). The leaves of history(i) will correspond to the intervals of $H(N^i)$, and its internal nodes will be labeled with the points in N^i (Figure 1.5).

Formally, history(0) will consist of just one node, which corresponds to the whole of R . Inductively, assume that history(i) has been defined. Let S_{i+1} be the $(i+1)$ th randomly chosen point to be added (Figure 1.5, Figure 1.6). Let I be the interval in $H(N^i)$ containing S_{i+1} . It is split into two intervals I_1 and I_2 by S_{i+1} . We say that the interval I is *killed* during the $(i+1)$ th addition. We also say that the intervals $I_1, I_2 \in H(N^{i+1})$ are *created* during the $(i+1)$ th addition. History($i+1$) is obtained by labeling the leaf of history(i), which corresponds to I , with S_{i+1} . We also give this leaf two *children*, one each for I_1 and I_2 (Figure 1.6).

Analysis

Plainly, the time required to build $\tilde{H}(N) = \text{history}(n)$ is of the same order as the running time of quick-sort. Its expected value is $O(n \log n)$.

Let us turn to the search cost, which is also called the query cost or the point location cost. We want to estimate the *expected* search cost. What

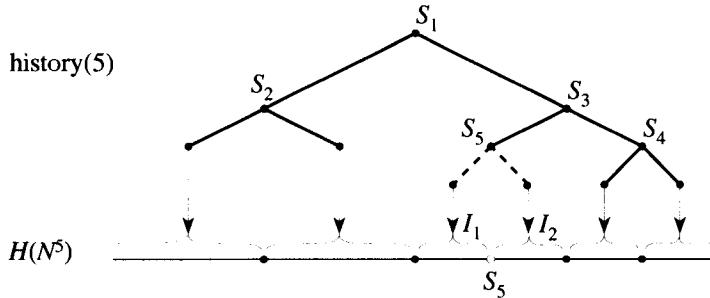


Figure 1.6: Addition of the fifth point.

does this mean? Note that the construction of $\text{history}(n)$ was randomized. It was based on the random order of additions of the elements in N . We assume that this random order is kept completely hidden from the outside world. So the user does not know the order actually used by the algorithm. He (or she) only knows that the order was random. By the expected search cost, we mean the search cost that the user will expect, only knowing that the order of additions was random.

Fix a query point $p \in R$. We shall use backward analysis. Imagine tracing p backwards in history. Starting with $H(N) = H(N^n)$, imagine deleting the points in N , one at a time, in the reverse order S_n, S_{n-1}, \dots, S_1 . This gives rise to a sequence of partitions $H(N^n), H(N^{n-1}), \dots, H(N^0)$. For $1 \leq j \leq n$, define a 0-1 random variable V_j such that $V_j=1$ iff the interval in $H(N^j)$ containing p is adjacent to the deleted point S_j . In this case, the interval is destroyed during the imaginary deletion of S_j from $H(N^j)$. Clearly, the length of the search path in $\text{history}(n)$ is $V = \sum_{j=1}^n V_j$. Due to the randomized construction of our search structure, each point in N^j is equally likely to occur as S_j . Hence, the probability that $V_j = 1$ is bounded by $2/j$, because at most two points in N^j are adjacent to the interval in $H(N^j)$ containing p . This means that the expected value of V_j is also bounded by $2/j$. By linearity of expectation, it follows that the expected value of V is $O(\sum 1/j) = O(\log n)$. Thus, for a fixed query point p , the expected length of the search path is $O(\log n)$.

Actually, one can show that the length of the search path is $\tilde{O}(\log n)$, i.e., the logarithmic bound holds with high probability.¹ This follows directly from the so-called Chernoff bound for harmonic random variables. The in-

¹ In this book, the \tilde{O} notation means the following. We say that $f(n) = \tilde{O}(g(n))$, if, for some positive constant c , $f(n) < cg(n)$, with probability $1 - 1/p(n)$, where $p(n)$ is a polynomial whose degree depends on c , and this degree can be made arbitrarily high by choosing the constant c large enough. Thus, $f(n) > cg(n)$ with probability $1/p(n)$, which is minuscule.

terested reader should refer to Appendix A.1.3. Chernoff's technique can be applied to V because each V_j is independent of V_i , for $i > j$.

Remark. If you do not wish to know what Chernoff's technique is, you can simply ignore the high-probability performance bounds in this book, such as the one above. Even if you are a theoretically inclined reader, it might make sense to take such bounds on trust on the first reading and come back to them later.

The preceding high-probability bound is for a fixed query point p . But it is possible to prove that the same bound holds regardless of the query point. First, notice that the search paths of all query points lying in the same interval of $H(N)$ are identical. Hence, the number of distinct search paths is $n + 1$. Second, each search path has $\tilde{O}(\log n)$ length; this follows if we choose a sample query point in each interval of $H(N)$. This implies the following: For each fixed interval $I \in H(N)$, the error probability that the length of the corresponding search path in $\tilde{H}(N)$ exceeds $c \log n$, for a positive constant c , is $O(1/h(n))$, where $h(n)$ is a polynomial whose degree can be made arbitrarily large by choosing c large enough. As the total number of intervals is $n + 1$, the probability that the search for any interval exceeds $c \log n$ is $O(n/h(n)) = O(1/h'(n))$, where the degree of $h'(n) = h(n)/n$ can be made arbitrarily large by choosing c large enough. It follows that the maximum length of a search path in $\tilde{H}(N) = \text{history}(n)$ is $\tilde{O}(\log n)$.

The above argument would have worked even if the number of distinct search paths were not $O(n)$, but rather $O(n^a)$, for a fixed constant $a > 0$. For future reference, we encapsulate this simple observation in an abstract form.

Observation 1.3.1 Suppose the length of the search path in the given randomized search structure is $\tilde{O}(\log n)$, for any fixed query point, where n is the number of objects in the search structure. Assume that the number of distinct search paths is bounded by a polynomial of fixed degree in n . More precisely, assume that the space under consideration (which is the real line in the case of randomized binary trees, but could be higher-dimensional Euclidean space, in general) can be partitioned into $O(n^a)$ regions, for a constant $a > 0$, so that (1) the partition depends only on the set N of objects in the search structure and not on the state of the search structure, and (2) the search paths for any two points in the same region of the partition are identical. Then the depth of the search structure, i.e., the maximum length of any search path is also $\tilde{O}(\log n)$.

As a by-product, the above result also implies that the running time of quick-sort is $\tilde{O}(n \log n)$. Let us see why. Let us amortize (i.e., distribute) the cost of adding $S = S_{i+1}$ in the randomized incremental version of quick-sort. This is done by charging $O(1)$ cost to each point in the conflict list

of the (destroyed) interval $I \in H(N^i)$ containing S (Figures 1.2 and 1.3). This suffices to cover the cost adding S , ignoring an $O(1)$ additive term. The total cost charged to a fixed point $Q \in N$, over the whole course of the algorithm, is clearly equal to the number of times the interval containing Q is destroyed during the additions. This is the same as the length of the search path for Q in $\text{history}(n)$, which is $\tilde{O}(\log n)$. Hence, the total cost charged to a fixed point in N is $\tilde{O}(\log n)$. Thus, the total running time of quick-sort is $\tilde{O}(n \log n)$.

We have already remarked that the cost of building history is of the same order as the running time of quick-sort. Hence, we have proved the following.

Theorem 1.3.2 *Given a set of points N on the real line R , the partition $H(N)$ and an associated randomized search tree can be built in $\tilde{O}(n \log n)$ time. The point location cost is $\tilde{O}(\log n)$.*

1.3.1 Dynamization

The search tree $\tilde{H}(N)$ defined in the previous section is static, in the sense that the whole set N is assumed to be known *a priori*. In this section, our goal is to make the search structure dynamic.

The basic idea is quite simple. If M denotes the set of points at any given time, then our search structure at that time will look as if it were constructed by applying the static procedure in Section 1.3 to the set M . This would automatically imply that the maximum query cost for the search structure existing at any given time is $\tilde{O}(\log m)$, where $m = |M|$ denotes the number of points existing at that time.

For the sake of dynamization, it is convenient to modify the static procedure in Section 1.3 slightly. Our data structure in the static setting is now defined as follows. First, for every point in M , we randomly choose a *priori*ty (a real number) from the uniform distribution on the interval $[0, 1]$. The ordering of the points in M according to the increasing priorities is called a *priority order* or a *shuffle*. Because the priorities of all points in M are chosen independently and in a similar fashion, it follows that all priority orders on M are equally likely. The state of the data structure that we will associate with $H(M)$ will be completely determined by the shuffle of M . For this reason, we shall denote it by $\text{shuffle}(M)$. Let S_k denote the k th point in the priority order on M . Let M^i denote the set of the first i points S_1, \dots, S_i . Our data structure is then simply the history of the sequence $H(M^1), \dots, H(M^m) = H(M)$, as defined in Section 1.3. In other words, we imagine adding the points in M in the increasing order of their priorities. $\text{Shuffle}(M)$ is then defined to be the history of this imaginary computation.

Another equivalent definition of $\text{shuffle}(M)$ is as follows. The root of $\text{shuffle}(M)$ is labeled by the point in M with the least priority. Let $M_1, M_2 \subseteq M$ be the subsets of points with lower and higher coordinates, respectively. The left and the right subtrees below the root are the recursively defined binary trees $\text{shuffle}(M_1)$ and $\text{shuffle}(M_2)$.

How do we delete a point from $\text{shuffle}(M)$? Let $S \in M$ be the point that we wish to delete. Let $M' = M \setminus \{S\}$, with the priority order naturally inherited from M . Thus, $\text{shuffle}(M')$ is uniquely defined. Our goal is to obtain $\text{shuffle}(M')$ from $\text{shuffle}(M)$ quickly. If S occurs at the bottom of $\text{shuffle}(M)$, as S_5 does in Figure 1.6, then this is easy. In this case, both children of S are empty. That is to say none of them is labeled with any point in M . So, we simply remove these children. For example, if we delete S_5 from the binary tree in Figure 1.6, we shall get the binary tree in Figure 1.5.

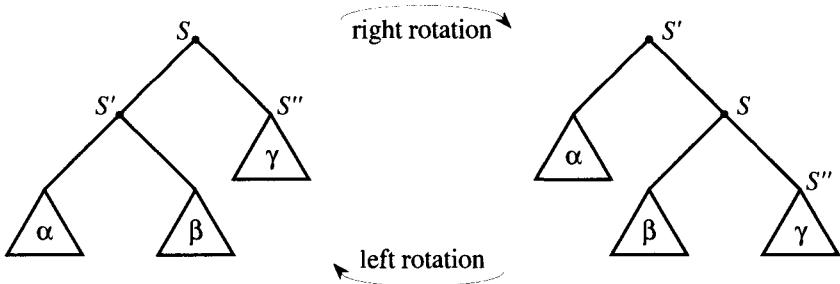


Figure 1.7: Rotations.

The interesting case occurs when S is not at the bottom of $\text{shuffle}(M)$. In this case, the idea is to increase the priority of S step by step to the highest possible (symbolic) value ∞ , thereby bringing it to the bottom of the tree. Consider the node in $\text{shuffle}(M)$ labeled with S (Figure 1.7). Let S' and S'' be the points labeling the left and right children of this node. Assume, without loss of generality, that S' has lower priority than S'' , the other case being symmetrical. Now imagine increasing the priority of S just above that of S' . The search tree has to be changed to reflect this change in the priority. So we need to bring S below S' in the tree, because the new priority of S is higher. With this in mind, we change our tree locally, using an operation called *right rotation*, and bring S below S' in the tree (Figure 1.7). The resulting tree correctly reflects the switch in the priority orders of S and S' , everything else being the same as before. We keep on performing rotations in this fashion until S is at the bottom of the tree. After this, we remove S , as described before.

Note that $\text{shuffle}(M')$ obtained at the end of the deletion operation is as if it were constructed by applying the previously described static procedure to the set M' .

As the depth of $\text{shuffle}(M)$ is $\tilde{O}(\log m)$ and the cost of each rotation is $O(1)$, it follows that the cost of deletion is $\tilde{O}(\log m)$.

Let us now turn to the addition operation. Suppose we want to add a new point S to $\text{shuffle}(M)$. We choose a priority for S from the interval $[0, 1]$ randomly and independently of the other points in M . This fixes a priority order on the set $M' = M \cup \{S\}$. Our goal is to obtain $\text{shuffle}(M')$ from $\text{shuffle}(M)$ quickly. At first pretend that the priority of S is the highest in M . In that case, addition of S to $\text{shuffle}(M)$ is easy: We locate the leaf of $\text{shuffle}(M)$ that corresponds to the interval in $H(M)$ containing S . We label this node with S and give it two children corresponding to the two intervals in $H(M')$ adjacent to S (Figures 1.5 and 1.6). Since the length of the search path is $\tilde{O}(\log m)$, this whole operation takes $\tilde{O}(\log m)$ time.

But what happens if the priority of S is not really the highest in M' ? In that case, we move S higher in the search tree through rotations until S is in the proper place. This operation is just the reverse of the analogous operation during deletion. Hence, its cost is also of the same order.

It follows that the cost of adding S to $\text{shuffle}(M)$ is $\tilde{O}(\log m)$. Note that $\text{shuffle}(M')$ obtained at the end of the addition operation is as if it were constructed by applying the previously described static procedure to the set M' .

1.3.2 Another interpretation of rotations

The rotation operation just described can be interpreted differently. The previous description of this operation was based on the fact that $\text{shuffle}(M)$ is a tree. Now, we wish to give an abstract interpretation that does not depend on the detailed structural properties of history. This alternative interpretation will turn out to be crucial later in the book.

Conceptually, deletion of S from $\text{shuffle}(M)$ can be carried out by moving S higher in the priority order by one position at a time. Let us elaborate. Let l be the number of points in M with priority higher than S . For $i \leq l$, let $M(i)$ denote the priority-ordered set obtained from M by moving S higher in the order by i places. Conceptually, $\text{shuffle}(M')$ can be obtained from $\text{shuffle}(M)$ as follows. Starting with $\text{shuffle}(M) = \text{shuffle}(M(0))$, we imagine computing the sequence

$$\text{shuffle}(M(0)), \text{shuffle}(M(1)), \text{shuffle}(M(2)), \dots, \text{shuffle}(M(l)).$$

Consider the basic operation of obtaining $\text{shuffle}(M(i))$ from $\text{shuffle}(M(i-1))$. Refer to Figure 1.8, where $B = M(i-1)$ and $C = M(i)$. Each arrow

in the figure symbolically represents an addition operation. The figure has two rows. The upper row represents a sequence of additions, whose history is recorded by $\text{shuffle}(B)$. The lower row represents a sequence of additions, whose history is recorded by $\text{shuffle}(C)$. In the figure, j denotes the priority order of S in B ; thus $j = m - l + i - 1$, where m is the size of M . Let S' denote the point just after S in the priority order B . The priority-ordered set C is obtained from B by switching the priorities of S and S' . $\text{Shuffle}(C)$ is obtained from $\text{shuffle}(B)$ by changing the history to reflect this switch.

If $\text{shuffle}(C) = \text{shuffle}(B)$, this operation comes for free, and we need not do anything. In such cases, we say that S and S' commute. This happens when S and S' are contained in the different intervals of $H(B^{j-1})$ (Figure 1.9).

Otherwise, the update of $\text{shuffle}(B)$ to $\text{shuffle}(C)$ exactly corresponds to the rotation operation described in the last section. Figure 1.10 illustrates why this is so. In the figure, subtrees α , β , and γ record the history, restricted within the corresponding intervals, of the additions of the points with priority higher than $j + 1$.

At the end of the above sequence of rotations, we obtain $\text{shuffle}(M(l))$. Now, S has the highest priority in $M(l)$. This means it occurs at the bottom of $\text{shuffle}(M(l))$ (like S_5 in Figure 1.6). Hence, $\text{shuffle}(M')$ is obtained easily, as described before, by removing the children of the node labeled with S .

Exercises

1.3.1 In this section, we assumed that the priority of each element in M is chosen from a uniform distribution on $[0, 1]$.

- (a) Show that the analysis goes through even when the priority is chosen from any continuous distribution, as long as all priorities are chosen independently from the same distribution.
- (b) What happens when the priorities are chosen from a discrete, rather than a continuous, distribution? (Hint: What happens if some elements in M are assigned the same priority? When the distribution is continuous, the probability of this event is zero.)
- (c) What happens when the priorities are drawn from a uniform discrete distribution on a sufficiently large universe?

$$\begin{array}{ccccccc} \longrightarrow & H(B^{j-1}) & \xrightarrow{S} & H(B^j) & \xrightarrow{S'} & H(B^{j+1}) & \longrightarrow \\ & = & & = & & = & \\ \longrightarrow & H(C^{j-1}) & \xrightarrow{S'} & H(C^j) & \xrightarrow{S} & H(C^{j+1}) & \longrightarrow \end{array}$$

Figure 1.8: Rotation diagram.

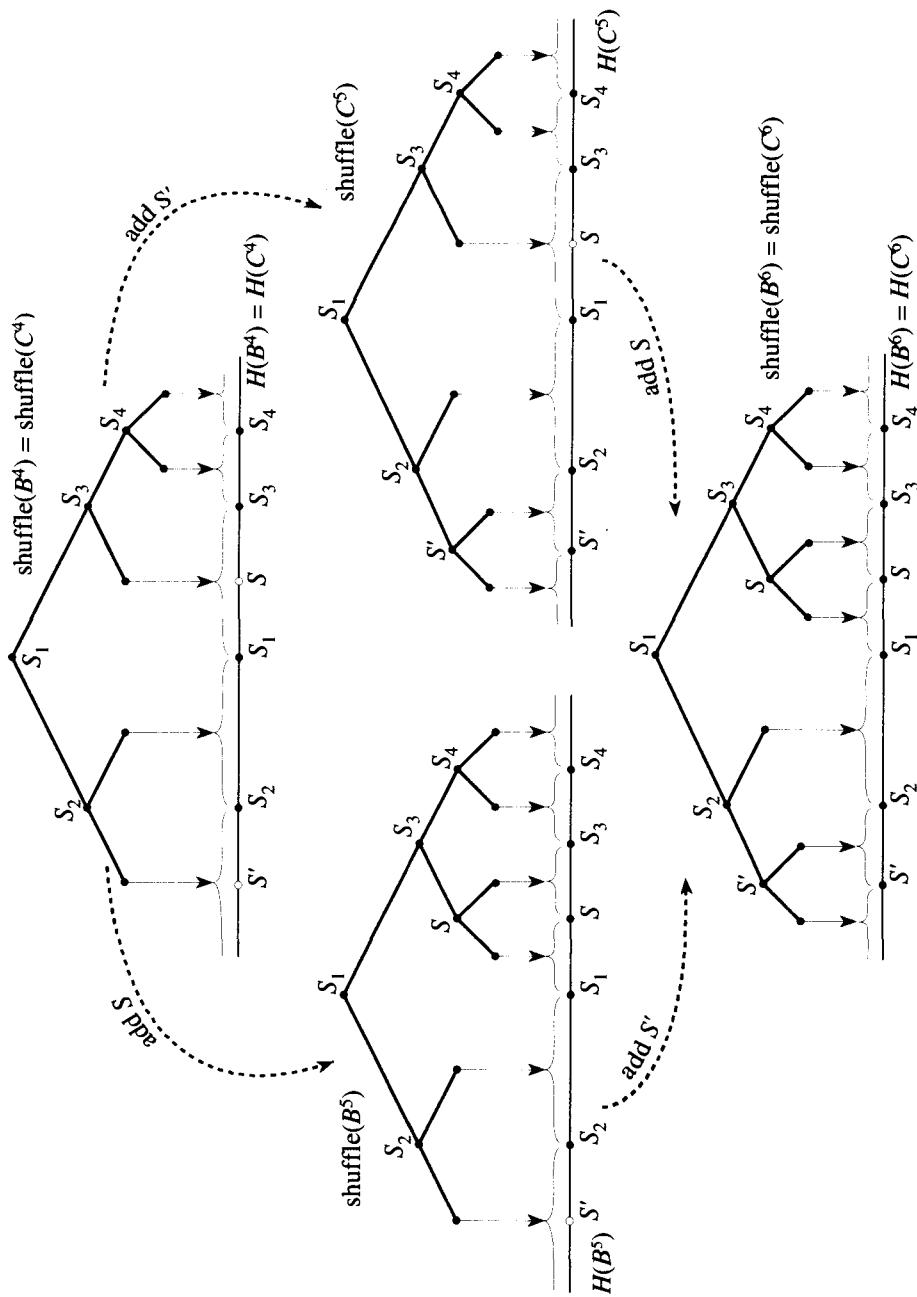


Figure 1.9: Commutativity.

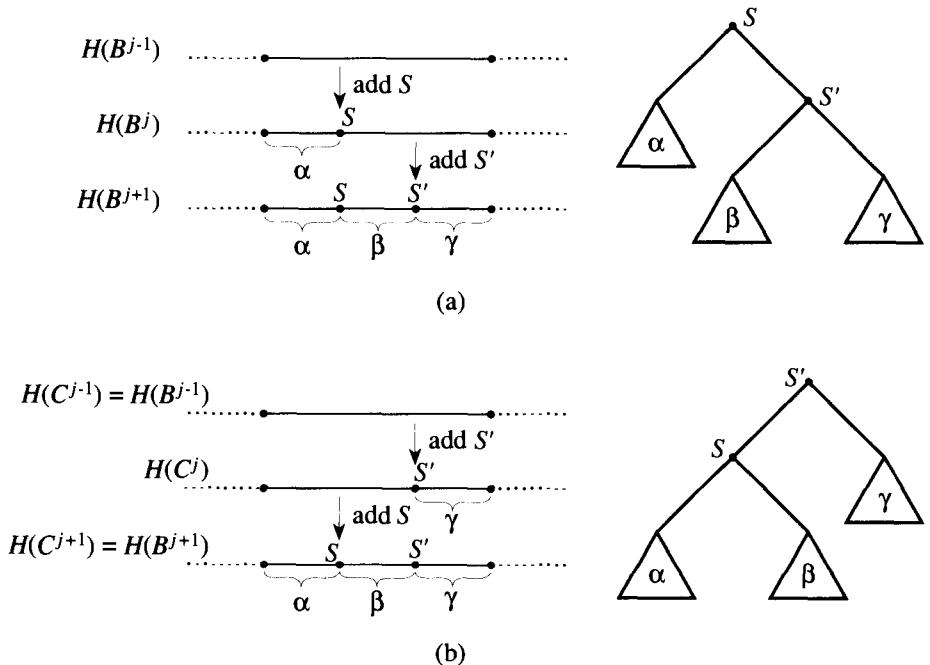


Figure 1.10: (a) Before rotation. (b) After rotation.

1.3.2 Show that the expected number of rotations per addition or deletion is $O(1)$; in fact, ≤ 2 . (Hint: First, consider deletion of a point $Q \in N$. Use backward analysis. Imagine deleting the points in N in the reverse random order S_n, S_{n-1}, \dots, S_1 . Define a random variable W_j as follows: $W_j = 1$ iff $Q \in N^j$ and S_j is adjacent to Q in $H(N^j)$. Using the interpretation of rotations in Section 1.3.2, show that the total number of rotations is $\sum_j W_j$. Moreover, $W_j = 1$ with probability $\leq (j/n)(2/j) = 2/n$. Hence, the expected number of rotations is bounded by 2. For addition, argue that the number of rotations in the addition of a new point P is equal to the number of rotations in deleting P from the resulting tree. Hence, the expected number of rotations during addition is also $O(1)$.)

In particular, it follows that the expected cost of deletion is $O(1)$. Similarly, show that the expected cost of addition is also $O(1)$ when we know where the item is going to be inserted. (This means we know the node storing the item before or after the inserted item, in the coordinate order.)

Show that a randomized binary tree for a list of n items, which is already in the sorted order, can be built in expected $O(n)$ time.

1.3.3 The assumption that each point in M can be assigned, as a priority, a real number from the uniform distribution on $[0, 1]$ is unrealistic. This is because real computers use finite precision. Instead, assume that the bits in the binary representation of a priority $x \in [0, 1]$ are generated one at a time as required, starting with the most significant bit. In the algorithm, we need only compare priorities.

For this, it is not necessary to know all bits of the two priorities involved in a comparison. We generate more bits only when the currently known bits in the binary representations are not sufficient to determine the outcome of the comparison.

Show that the expected number of bits required per update is $O(1)$. (Hint: Use the previous exercise.)

1.3.4 In some applications, it might be necessary to visit all nodes of the subtrees involved in a rotation, i.e., the subtrees α , β , and γ in Figure 1.7. This might be required to update the auxiliary application-dependent information stored at these nodes, which can change during a rotation. If the cost of rotating a subtree of size s is taken to be $O(s)$, show that the total expected cost of rotations during an addition or deletion is $O(\log m)$, where m is the number of points in the search tree. If the cost of rotating a subtree of size s is taken to be $O(s^a)$, where $a > 1$, then the total expected cost of rotations is $O(m^{a-1})$.

1.3.5 Given a point p on the real line and any set M of m points, let $M', M'' \subseteq M$ denote the subsets of points with smaller and higher coordinates than p , respectively. Show how $\text{shuffle}(M)$ can be split in expected $O(\log m)$ time to yield $\text{shuffle}(M')$ and $\text{shuffle}(M'')$. (Hint: Add p to M , after assigning to it $-\infty$ priority. It will then occur at the root of the new tree, after addition. The two subtrees of the root should provide the required answer.)

Give the reverse concatenation procedure. The input consists of two trees, $\text{shuffle}(M')$ and $\text{shuffle}(M'')$, where each point in M' has a smaller coordinate than every point in M'' . Show how to obtain $\text{shuffle}(M)$, $M = M' \cup M''$, in expected $O(\log m)$ time, where m is the size of M . (Hint: First, add a dummy root, labeled with a point r whose coordinate lies between those of the points in M' and M'' , and having $\text{shuffle}(M')$ and $\text{shuffle}(M'')$ as its two subtrees. The initial priority of r is $-\infty$. Next, increase the priority of r to $+\infty$.)

1.3.6 Given two points $p, q \in M$, let $l(p, q)$ denote the number of points in M whose coordinates lie between those of p and q . Show that the expected length of the path in the tree $\text{shuffle}(M)$ connecting the nodes labeled p and q is $O(\log l(p, q))$.

Given a “finger” (a pointer) to the node labeled with p , show how any point q can be searched in expected $O(l(p, q))$ time. This helps when we know a finger near the query point.

1.3.7 Given two points $p, q \in M$, let $M', \bar{M}, M'' \subseteq M$ denote the subsets of points to the left of p , between p and q , and to the right of q , respectively. Given pointers (fingers) to the nodes in $\text{shuffle}(M)$ labeled with p and q , show how $\text{shuffle}(M)$ can be split into $\text{shuffle}(M' \cup M'')$ and $\text{shuffle}(\bar{M})$ in expected $O(\log l(p, q))$ time. (Hint: Use the previous exercise.)

Give the reverse concatenation procedure.

***1.3.8** Assume that each point $p \in M$ is assigned a positive integer weight $w(p)$, which equals the number of times p has been accessed since its insertion. The weight $w(p)$ is initialized to 1 when p is inserted. After that, every time p is accessed, $w(p)$ is increased by 1. We want to ensure that the search paths of heavily accessed points have shorter expected length. For this, we modify the search procedure as follows. In the beginning, the point p is assigned a priority, as usual, from the uniform distribution on $[0, 1]$. After that, every time p is accessed, we independently choose

a priority from the uniform distribution on $[0, 1]$. If this priority is smaller than the current priority of p , we change the priority of p to this smaller value. $\text{Shuffle}(M)$ is updated, through rotations, to reflect this change in priority, thereby pushing p toward the root. Using backward analysis, show the following:

- (a) The expected time necessary to access a point $p \in M$ with weight $w(p)$ is $O(1 + \log(W/w(p)))$. Here W denotes the total weight of the points in M . (Hint: Imagine, for the purpose of analysis, inserting a new distinct copy of p every time it is accessed.)
- (b) The expected time required to add a point q with weight $w(q)$ is

$$O\left(1 + \log \frac{W + w(q)}{\min\{w(p), w(q), w(r)\}}\right),$$

where p and r are, respectively, the predecessor and successor of q in M , with respect to the coordinates. (Here we are assuming that the point is added with the given weight $w(q)$ in just one shot, rather than inserting it $w(q)$ times and incrementing the weight during each access.)

- (c) The expected time required to delete a point $q \in M$ with predecessor p and successor q is

$$O\left(1 + \log \frac{W}{\min\{w(p), w(q), w(r)\}}\right),$$

- (d) The expected number of rotations required during the insertion or deletion of q is

$$O(1 + \log[1 + w(q)/w(p)] + \log[1 + w(q)/w(r)]).$$

- (e) The expected time required to split $\text{shuffle}(M)$ in two trees $\text{shuffle}(M')$ and $\text{shuffle}(M'')$ (or to concatenate $\text{shuffle}(M')$ and $\text{shuffle}(M'')$) is

$$O(1 + \log[1 + W'/w(p)] + \log[1 + W''/w(q)]),$$

where p is the point in M' with the maximum coordinate, and q is the point in M'' with the minimum coordinate. W' and W'' are the total weights of the points in M' and M'' , respectively.

1.4 Skip lists

In this section, we describe a different randomized structure for dynamic search in sorted lists, called skip list. First, we shall describe the search structure in a static setting and address its dynamization later. Let M be any given set of m points on the real line R . We shall associate a search structure with M using a certain bottom-up random sampling technique. For this reason, the search structure will be denoted by $\text{sample}(M)$.

We assume that we are given a fair coin. This means that the probability of success (i.e., obtaining heads) is $1/2$. Our construction works even for a biased coin for which the *bias*, i.e., the probability of success, is a fixed

constant between 0 and 1. But this generalization is straightforward and hence we shall leave it to the reader.

Starting with the set M , we obtain a sequence of sets

$$M = M_1 \supseteq M_2 \supseteq \cdots \supseteq M_{r-1} \supseteq M_r = \emptyset, \quad (1.1)$$

where M_{i+1} is obtained from M_i by flipping the coin independently for each point in M_i and retaining only those points for which the toss was a success (heads). The sequence in (1.1) is called a *gradation* of M . Note that each M_i has roughly half the size of M_{i-1} . Hence, one would guess that the expected length of this gradation should be $O(\log m)$. We shall see later that this is indeed the case. The state of $\text{sample}(M)$ is going to be determined completely by the gradation of M . This property is analogous to the similar property in Section 1.3.1, wherein the state of $\text{shuffle}(M)$ was completely determined by the priority order on M .

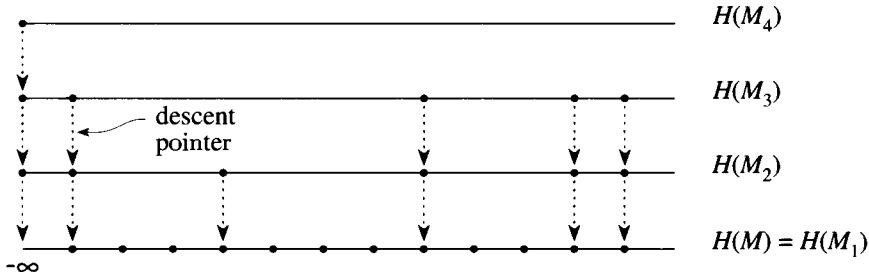


Figure 1.11: Skip list.

$\text{Sample}(M)$ consists of r levels, where r is the length of the above gradation. The set M_i , $1 \leq i \leq r$, can be thought of as the set of points “stored” in the i th level (Figure 1.11). We shall store the whole partition $H(M_i)$ in the form of a sorted linked list at the i th level of $\text{sample}(M)$. This means M_i has to be sorted. (The intervals of $H(M_i)$ are implicit in this representation.) With every point $S \in M_i$ stored in the i th level, we also associate a *descent pointer* to the occurrence of the same point in the $(i-1)$ th level. For the sake of simplicity, we assume that each M_i contains an additional point with coordinate $-\infty$. This completely specifies $\text{sample}(M)$.

Some terminology: By an interval in $\text{sample}(M)$, we mean an interval in any level of $\text{sample}(M)$. The parent interval of a given interval $I \in H(M_i)$ is defined to be the interval J in $H(M_{i+1})$ containing I ; we also say that I is a child of J . If we imagine pointers from parents to their children, the skip list can also be thought of as a search tree, as we shall do on several occasions. Of course, it is not represented in the memory as a tree because that is not its natural representation.

$\text{Sample}(M)$ can be used for searching as follows (Figure 1.12). Let S be the point that we wish to locate in $H(M)$. First we trivially locate S in $H(M_r)$, which consists of just one interval, namely, the whole of R . Inductively, assume that we have located S in $H(M_i)$, the partition stored in the i th level, where $1 < i \leq r$. Let $\Delta_i \in H(M_i)$ be the interval containing S . Using the descent pointer associated with the left endpoint of Δ_i , we can search through all children of Δ_i . (Not all children of Δ_i are visited if we search from the left to right—but never mind.) This yields the required interval Δ_{i-1} containing S in the $(i-1)$ th level. When we reach the first level, our search is complete. Note that the cost of the search is proportional to the number of children of the intervals on the search path. By the intervals on the search path, we mean the intervals containing S in all levels of $\text{sample}(M)$.

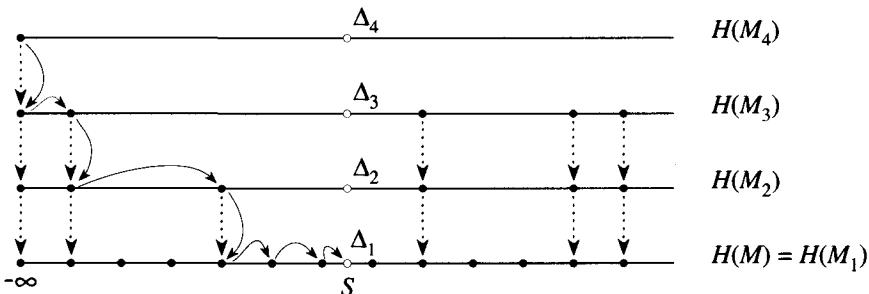


Figure 1.12: Search path.

Analysis

As a warm-up exercise, let us show that the expected number of levels in $\text{sample}(M)$ is indeed logarithmic. In fact, we shall show something stronger. We shall show that the number of levels is logarithmic with high probability. Fix a positive integer k . A fixed point $Q \in M$ is contained in more than k levels of $\text{sample}(M)$ iff, for each $1 < i \leq k + 1$, the coin toss for Q during the selection of M_i was successful. Since all coin tosses are independent, this happens with probability $1/2^k$. Hence, the probability that at least one point in M is stored in more than k levels of $\text{sample}(M)$ is trivially bounded by $m/2^k$. This is because the probability of the union of events is bounded by the sum of their probabilities. It follows that the number of levels in $\text{sample}(M)$ exceeds k with probability bounded by $m/2^k$. This bound is $1/m^{c-1}$ if we let $k = c \log_2 m$, for some constant $c > 1$. By choosing the constant c large enough, this bound can be made minuscule. This proves that the number of levels in $\text{sample}(M)$ is $\tilde{O}(\log m)$.

Next, let us estimate the space requirement of $\text{sample}(M)$. Fix a point $Q \in M$. Let $h(Q)$ denote the number of levels in $\text{sample}(M)$ containing Q . Note that $h(Q)$ is equal to one plus the number of successes obtained before

failure in a sequence of independent fair coin tosses for Q . In other words, $h(Q) = i$ iff the first $i - 1$ tosses for Q are successful and the i th toss is a failure. In the language of probability theory, $h(Q)$ is said to be distributed according to the geometric distribution² with a parameter $1/2$. Since the tosses are all independent, it is clear that $h(Q) = i$ with probability $1/2^i$. Hence, its expected value is

$$\sum_i \frac{i}{2^i} = O(1). \quad (1.2)$$

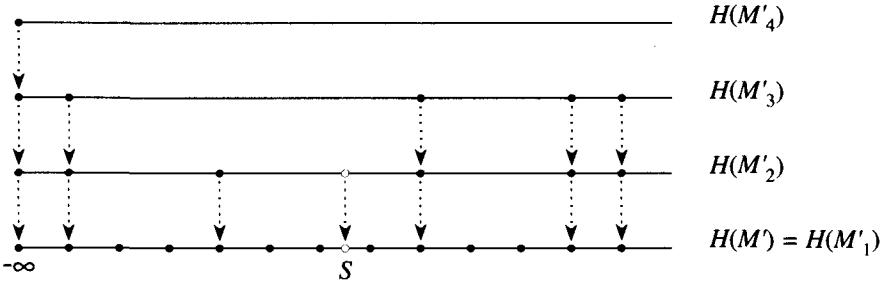
Thus, the expected number of levels in $\text{sample}(M)$ containing a fixed point is $O(1)$. The size of $\text{sample}(M)$ is obviously proportional to $\sum_Q h(Q)$, where Q ranges over all points in M . By linearity of expectation, it follows that the expected space requirement of $\text{sample}(M)$ is $O(m)$. In fact, one can show that this bound holds with high probability: Because all coin tosses are independent, the $h(Q)$'s are independent geometric distributions. The high-probability $\tilde{O}(m)$ bound then follows from the Chernoff bound for the sum of independent geometric distributions. We refer the interested reader to Appendix A.1.2.

Next, we shall estimate the cost of constructing $\text{sample}(M)$. The time taken in constructing the i th level of $\text{sample}(M)$ is dominated by the construction of $H(M_i)$, which amounts to sorting the set M_i . This can be done by quick-sort in $\tilde{O}(m_i \log m)$ time, where m_i is the size of M_i . Over all levels, this sums to $\tilde{O}(\sum_i m_i \log m)$ total cost. The sum $\sum_i m_i$ is essentially the size of $\text{sample}(M)$, which, as we already know, is $\tilde{O}(m)$. Thus, $\text{sample}(M)$ can be constructed in $\tilde{O}(m \log m)$ time.

Finally, let us turn to the search cost. We have already observed that the cost of locating a given point S using $\text{sample}(M)$ is proportional to the number of children of the intervals on the search path. By the intervals on the search path, we mean the intervals containing S in all levels of $\text{sample}(M)$. We now want to estimate the *expected* search cost. What do we mean by the expected search cost? Recall that our construction of the search structure was based on random coin tosses. We assume that the results of these coin tosses are kept completely hidden from the outside world. By the expected search cost, we mean the search cost that the user will expect, only knowing that the tosses were random.

For each interval Δ_i on the search path (Figure 1.12), let $l(\Delta_i)$ denote the number of its children. The cost of the search is clearly bounded by $O(\sum_i 1 + l(\Delta_i))$. Let $l_0(\Delta_i)$ and $l_1(\Delta_i)$ denote the number of points in $M_{i-1} \setminus M_i$ that are contained in Δ_i to the left and right of S , respectively.

²A geometric distribution with a parameter p is the number of successes obtained before failure in a sequence of Bernoulli trials (independent coin tosses) with a coin of bias p .

Figure 1.13: Addition of a new point S .

Thus, $l(\Delta_i) = l_0(\Delta_i) + l_1(\Delta_i)$. We shall estimate $\sum_i l_0(\Delta_i)$. The other sum, $\sum_i l_1(\Delta_i)$, can be estimated similarly.

See Figure 1.12. Note that once M_{i-1} is fixed, $l_0(\Delta_i)$ is distributed according to the geometric distribution with parameter 1/2, in the terminology introduced before. This is because $l_0(\Delta_i) = k$ iff, for exactly the k nearest points in M_{i-1} to the left of S , the tosses were all failures (tails). Hence, arguing as in (1.2), it follows that the expected value of $l_0(\Delta_i)$, conditional on a fixed M_{i-1} , is $O(1)$. As this bound does not depend on M_{i-1} , it provides an unconditional bound as well. We have already seen that the depth of $\text{sample}(M)$, i.e., the number of levels in it, is $\tilde{O}(\log m)$. Since the coin tosses at all levels are independent, it follows that the expected search cost is $O(\log m)$. In fact, the search cost is $\tilde{O}(\log m)$. For this, we observe that the coin tosses used at all levels are independent. Both $l_0(\Delta_i)$ and $l_1(\Delta_i)$ follow geometric distributions. Hence, the high-probability bound follows from the Chernoff bound for the sum of independent geometric distributions. This sum is also called the negative binomial distribution. We refer the interested reader to Appendix A.1.2.

The above high-probability bound is for a fixed query point S . But notice that the number of distinct search paths in $\text{sample}(M)$ is $O(m)$. Hence, arguing as in the previous section (Observation 1.3.1), it follows that the maximum search cost is also $\tilde{O}(\log m)$.

Updates

Let us now see how to add a new point S to $\text{sample}(M)$. We first toss the given coin repeatedly until we get failure (tails). Let j be the number of successes obtained before getting failure. Our goal is to add S to levels 1 through $j+1$. This is easy. The search path of S in $\text{sample}(M)$ tells us the interval $\Delta_i \in H(M_i)$ containing S , for each i (Figure 1.12). Hence, we only need to split each Δ_i , for $1 \leq i \leq j+1$, and link the occurrences of S in the successive levels by descent pointers; see Figures 1.13 and 1.12. When $j+1$

is higher than the current depth r , we need to create extra $j + 1 - r$ levels, each containing only S ; in practice, it should suffice to create only one extra level. The cost of the above procedure is clearly dominated by the search cost, which is $\tilde{O}(\log m)$.

Let $M' = M \cup \{S\}$. For $1 \leq l \leq j + 1$, let M'_l denote $M_l \cup \{S\}$; and for $l > j + 1$, let $M'_l = M_l$. At the end of the above addition procedure, we get the updated data structure $\text{sample}(M')$, which corresponds to the gradation

$$M' = M'_1 \supseteq M'_2 \supseteq \dots$$

It looks as if it were constructed by applying the static procedure in the beginning of this section to the set M' .

Deletion is even easier. To delete a point $S \in M$, we simply remove it from every level containing it. The cost of removal per level is $O(1)$. The expected number of levels containing S is $O(1)$ (see (1.2)). Hence, it follows that the expected cost of deletion is $O(1)$.

Exercises

1.4.1 Assume that the coin used in the construction of $\text{sample}(M)$ has a fixed bias (probability of success) p , where $0 < p < 1$.

- (a) Show that the expected number levels containing a fixed point in M is $1/p$.
(Hint: This number is distributed according to the geometric distribution with parameter p (Appendix A.1.2)). Show that the expected number of random bits used per update is $1/p$. Compare with Exercises 1.3.2 and 1.3.3.
- (b) Get a precise bound on the search cost and the cost of update in terms of p .
- (c) How should the bias p be chosen in practice? Discuss the trade-offs.

1.4.2 Show that the expected number of random bits used per addition is $O(1)$.
(Hint: The expected number of levels that a new point gets inserted into is $O(1)$.)

1.4.3 Assume that we are given M in a sorted order according to the coordinates of the points. In that case, show that $\text{sample}(M)$ can be constructed in $O(m)$ expected time. (Hint: Add points in the increasing order of coordinates. The search operation is no longer necessary during addition, because we know that each point is going to be added at the end.)

1.4.4 Define the global conflict list of an interval $\Delta_i \in H(M_i)$ as the set of points in $M \setminus M_i$ that lie within Δ_i . Consider the addition of S to a fixed level i of $\text{sample}(M)$. Let Δ_i be the interval in $H(M_i)$ containing S . In some applications, it might be necessary to visit all intervals in $H(M_j)$, $j < i$, that are contained within Δ_i . This might be required to update the auxiliary application-dependent information stored with these intervals. The expected number of such intervals is proportional to the global conflict size of Δ_i , i.e., the number of points in $M \setminus M_i$ contained in it. (Why?) Suppose that the cost of inserting S in a given level i is proportional to the global conflict size $s(i)$ of the interval in $H(M_i)$ containing S . Show that the total expected cost of addition or deletion is still $O(\log m)$. If the cost of insertion is $O(s(i)^a)$, for $a > 1$, show that the total expected cost of addition or deletion is $O(m^{a-1})$. Compare with Exercise 1.3.4.

1.4.5 Given a point p on the real line and any set M of points, let $M', M'' \subseteq M$ denote the subsets of points with smaller and larger coordinates than p , respectively. Show how $\text{sample}(M)$ can be split in $O(\log m)$ expected time to yield $\text{sample}(M')$ and $\text{sample}(M'')$. (Hint: Split all levels of $\text{sample}(M)$ at p .)

Give the reverse concatenation procedure. The input consists of $\text{sample}(M')$ and $\text{sample}(M'')$, where each point in M' has a smaller coordinate than every point in M'' . Show how to obtain $\text{sample}(M)$, where $M = M' \cup M''$, in $O(\log m)$ expected time, where m is the size of M . (Hint: Simply concatenate all levels.) Compare with Exercise 1.3.5.

1.4.6 Given two points p and q , let $l(p, q)$ denote the number of points in M whose coordinates lie between those of p and q . Let $j(p, q)$ denote the lowest level of $\text{sample}(M)$ containing no point between p and q . Show that the expected value of $j(p, q)$ is $O(\log(l(p, q)))$.

Given a “finger” p , i.e., a pointer to the occurrence of p in the lowermost level of $\text{sample}(M)$, show how any point q can be searched in expected $O(l(p, q))$ time. This helps when we know a finger near the query point.

Compare with Exercise 1.3.6.

1.4.7 Given two points $p, q \in M$, let $M', \bar{M}, M'' \subseteq M$ denote the subsets of points to the left of p , between p and q , and to the right of q , respectively. Given pointers (fingers) to the occurrences of p and q in the lowest level of $\text{sample}(M)$, show how $\text{sample}(M)$ can be split into $\text{sample}(M' \cup M'')$ and $\text{sample}(\bar{M})$ in expected $O(\log l(p, q))$ time. (Hint: Use the previous exercise.)

Give the reverse concatenation procedure.

***1.4.8** Assume that each point $p \in M$ is assigned a positive integer weight $w(p)$ which equals the number of times p has been accessed since its insertion. The weight $w(p)$ is initialized to 1 when p is inserted. After that, every time p is accessed, $w(p)$ is increased by 1. We want to ensure that the search costs for heavily accessed points have a shorter expected value. Toward this end, we modify the search procedure as follows. Every time p is accessed, we again independently toss a coin for p until we get failure. Let j be the number of successes obtained before getting failure. If $j + 1$ is bigger than the number j' of levels in $\text{sample}(M)$ currently containing p , we insert p in the levels $j' + 1, \dots, j + 1$. Show the following:

- (a) The expected time necessary to access a point $p \in M$ with weight $w(p)$ is $O(1 + \log(W/w(p)))$. (Hint: Imagine, for the purpose of analysis, inserting a new distinct copy of p every time it is accessed.)
- (b) The expected time required to add a point q with weight $w(q)$ is

$$O\left(1 + \log \frac{W + w(q)}{\min\{w(p), w(q), w(r)\}}\right),$$

where p and r are, respectively, the predecessor and the successor of q in M , with respect to the coordinates. (Here we are assuming that the point is added with the given weight $w(q)$ in just one shot, rather than inserting it $w(q)$ times and incrementing the weight during each access.)

- (c) The expected time required to delete a point $q \in M$ with predecessor p and successor r is

$$O\left(1 + \log \frac{W}{\min\{w(p), w(q), w(r)\}}\right).$$

- (d) The expected number of levels that are affected during the insertion or deletion of q is

$$O(1 + \log[1 + w(q)/w(p)] + \log[1 + w(q)/w(r)]).$$

- (e) The expected time required to split $\text{sample}(M)$ into two skip lists $\text{sample}(M')$ and $\text{sample}(M'')$ (or concatenate $\text{sample}(M')$ and $\text{sample}(M'')$) is

$$O(1 + \log[1 + W'/w(p)] + \log[1 + W''/w(q)]),$$

where p is the point in M' with the maximum coordinate, and q is the point in M'' with the minimum coordinate. W' and W'' are the total weights of the points in M' and M'' , respectively.

Compare with Exercise 1.3.8.

- 1.4.9** Compare this section with the previous section (together with the exercises). Between skip list and randomized binary tree, which one do you think will be simpler and more efficient in practice? Verify your answer by implementation.

Bibliographic notes

Quick-sort is due to Hoare [120]. The $O(n \log n)$ bound on sorting achieved by quick-sort is actually the best possible under a certain algebraic decision tree model [19, 213, 229]. Randomized binary trees have been investigated by many authors. A good reference is [130], which is also a classic reference for the searching and sorting of lists. The dynamic version of randomized binary trees (Section 1.3.1 and the exercises) is due to Aragon and Seidel [10]. The interpretation of rotations given in Section 1.3.2 is from Mulmuley [168]. Skip lists (Section 1.4) are due to Pugh [187].

Chapter 2

What is computational geometry?

In Chapter 1, we studied sorting and searching in the one-dimensional setting. In that setting, the problem could be succinctly stated as follows: Given a set N of points on the real line R , construct the resulting partition $H(N)$ and a search structure $\tilde{H}(N)$ that could be used for point location in $H(N)$. We also studied a dynamic variant of this problem, wherein the set N could be changed by adding or deleting points.

In a multidimensional generalization of the preceding problem, the objects in N are no longer points in R , but rather hyperplanes, sites, half-spaces, etc., in R^d , depending on the problem under consideration. Here R^d denotes the d -dimensional Euclidean space. The partition $H(N)$ now denotes a partition of the whole or of a part of R^d induced by N . This, too, depends on the problem under consideration. For example, when N is a set of lines in the plane, $H(N)$ can denote the resulting partition of the plane. In general, $H(N)$ need not stand for a partition of the whole of R^d . This happens, for example, when N stands for a set of half-spaces in R^3 and $H(N)$ stands for the convex polytope formed by intersecting these half-spaces—or, more precisely, $H(N)$ stands for the collection of all (sub)faces of this polytope, endowed with the adjacency relationships among them. For this reason, we shall quite often refer to $H(N)$ as a *geometric complex*. Of course, there is nothing wrong in calling $H(N)$ a *geometric partition*, as long as it is not misleading. We do not really need to know here exactly what a geometric complex means. All we need is a rough definition to evoke a mental picture. By a geometric complex, we roughly mean a collection of disjoint regions (sets) in R^d , of varying dimensions, together with adjacency relationships among them. The simplest example is provided by a planar graph, which is

defined by a collection of vertices, edges, and faces, along with the adjacencies among them. A region in a geometric complex with dimension j is called a *j-face*. Generally, a 0-face, i.e., a zero-dimensional face, is called a *vertex*, and a 1-face, i.e., a one-dimensional face, is called an *edge*. This terminology is motivated by planar graphs. A full d -dimensional face of a geometric complex is called a *cell* or a *d-cell*. We say that an i -face f is *adjacent* to a j -face g , where $i < j$, if f is contained in the boundary of g . In this case, we also say that f is a *subface* of g . By the *size* of a geometric complex, we mean the total number of its faces of all dimensions. By the *size* of a face, we mean the total number of its subfaces.

The higher-dimensional sorting and searching problems can now be stated as follows:

Higher-dimensional sorting problem: Given a set N of objects in R^d , construct the induced geometric complex $H(N)$ quickly. In other words, determine the faces of $H(N)$ and “sort out” their relationships (adjacencies) with one another. Here the word “sorting” has to be interpreted in a larger context.

Higher-dimensional search problem: Construct a search structure $\tilde{H}(N)$ that can be used to answer queries over N quickly. For example, the queries could be point location queries over the partition $H(N)$ induced by N . But the queries need not always refer to some partition over N ; we shall see some examples of such queries in a moment.

The problems above are stated in a *static* setting, wherein the set N is known *a priori*. In a *dynamic* setting, we are required to update $H(N)$ and $\tilde{H}(N)$ quickly when N is changed by adding or deleting an object. In a *semidynamic* setting, only additions of objects are allowed. Semidynamic and dynamic algorithms are also called *on-line* algorithms.

Abstractly speaking, computational geometry can be viewed as a study of the general sorting and searching problem stated above. Of course, not all problems in computational geometry can be stated in this form, but a large number of them can be. In this chapter, we shall describe a few basic problems of computational geometry in this common form. This will prepare us to deal with them in a unified fashion later in the book. We shall state only a few basic problems, because we only want to provide the reader with some motivating examples. There is a large number of even more important problems that are not stated here. Several of these will be treated later in the book.

In computational geometry, the dimension d of the problem is generally considered fixed. This assumption will be implicit throughout this book. It does not mean that the dependence of the running time on d is unimportant.

In fact, in several exercises we shall calculate this dependence explicitly. We shall address the problems in arbitrary fixed dimension in the second half of this book. In the first half we are mainly interested in planar or three-dimensional problems. Hence, if you wish, you can assume that $d = 2$ or 3 in this chapter.

Throughout this book, we assume that the total number of objects in N is finite. We shall denote this number by n .

2.1 Range queries

One basic search problem in computational geometry is the *range searching* problem. Here N is a set of geometric objects in R^d . The query is given in the form of a *range*, by which we mean a connected region in R^d . We are asked to report or count the objects in N that intersect the query range. This problem, in various forms, occurs in a wide range of fields: data bases, pattern or speech recognition, best-match information retrieval, computer graphics, robotics, and so on.

The goal is to build a search structure so that the queries of the above form can be answered quickly. By “quickly,” we mean that the report-mode search time should be $O(k g(n) + f(n))$, where k is the size of the answer, i.e., the number of objects intersecting the range, and $f(n)$ and $g(n)$ are slowly growing functions, such as polylogarithmic functions. The count-mode search time should be $O(f(n))$, where $f(n)$ is a slowly growing function as above.

Here we are only interested in the search problem. The sorting problem is irrelevant, because, quite often, there is no natural partition $H(N)$ that can be associated with N in this connection.

The general range searching problem is far too difficult to admit a single general purpose solution. Hence, one generally makes some simplifying assumptions about the nature of the objects and the shape of the query range.

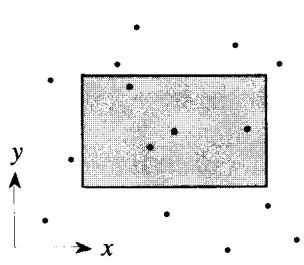


Figure 2.1: Rectangular range query.

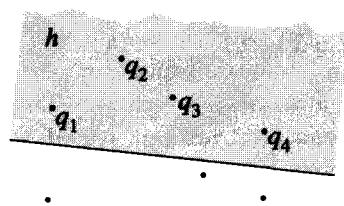


Figure 2.2: Half-space range query.

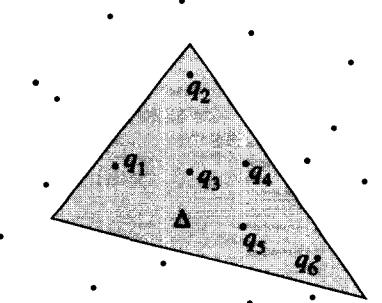


Figure 2.3: Simplex range query.

The simplest instance of the range searching problem is obtained when the objects in N are points and the query range is a hyperrectangle, i.e., a Cartesian product of d intervals (Figure 2.1). This problem is called an *orthogonal range searching* problem. Another simple instance is the so-called *half-space* range searching problem. Here the set N consists of points in R^d . The range is a half-space in R^d (Figure 2.2).

A generalization of a half-space range query is a *simplex range query*. In this problem, the range is a simplex (possibly unbounded) in R^d (Figure 2.3). Since a simplex can be thought of as the intersection of a bounded number of half-spaces, this problem may be regarded as a generalization of the half-space range query problem. (Here, as well as in the rest of this book, by a bounded number we mean a number that is bounded by a constant.)

2.2 Arrangements

An arrangement of lines in the plane is one of the simplest two-dimensional geometric structures (Figure 2.4). For this reason, it is often used as a test-bed for algorithmic ideas. Here the set N is a set of lines in the plane. These lines divide the plane into several convex regions. These regions are called the *cells* or *2-faces* of the arrangement (Figure 2.4). Each line in N is itself divided into several intervals by the remaining lines in N . These intervals are called *edges* or *1-faces* of the arrangement. Finally, the intersections among all lines in N are called *vertices* or *0-faces* of the arrangement. Clearly, the dimension of a j -face is j , which is the basis of this terminology.

The size of $H(N)$, i.e., the total number of its faces of all dimensions, is $O(n^2)$. This is because each line in N can be partitioned into at most n intervals by the remaining lines.

A generalization of an arrangement of lines is an arrangement of hyperplanes in R^d . Arrangements of hyperplanes in arbitrary dimensions are

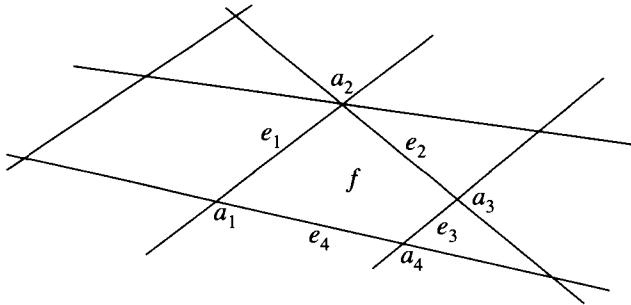


Figure 2.4: An arrangement.

studied in the second half of this book. Here we only state what they are. Each hyperplane is a set of points in R^d satisfying a linear equality of the form $a_1x_1 + \dots + a_dx_d = a_0$, where the x_j 's denote the coordinates in R^d , the coefficients a_j 's are arbitrary real numbers, and a_1, \dots, a_d are not all zero. When $d = 2$, a hyperplane is simply a line. When $d = 3$, a hyperplane is a plane.

Let N be a set of hyperplanes in R^d . The arrangement $H(N)$ formed by N is the natural partition of R^d into regions (faces) of varying dimensions together with the adjacencies among them. Formally, it is defined as follows. The hyperplanes in N partition R^d into several convex regions. These d -dimensional convex regions are called cells, or d -faces, of the arrangement $H(N)$. The intersections of the hyperplanes in N with every fixed hyperplane $S \in N$ give rise to a $(d - 1)$ -dimensional arrangement within S . The cells of this $(d - 1)$ -dimensional arrangement are called $(d - 1)$ -faces of $H(N)$. Proceeding inductively in this fashion, we define j -faces of $H(N)$, for all $j \leq d$. A 0-face of $H(N)$ is also called a vertex. A 1-face is also called an edge.

The arrangement $H(N)$ is said to be *simple* or *nondegenerate* if the intersection of any j hyperplanes in N is $(d - j)$ -dimensional. By convention, the set of negative dimension is defined to be empty. When $d = 2$, the previous condition merely says that no two lines in N are parallel and that no vertex in $H(N)$ is contained in more than two lines. An arrangement that is not simple is called *degenerate* or *nonsimple*. The arrangement in Figure 2.4 is degenerate, because a_2 is contained in three lines.

We can represent $H(N)$ in computer memory in several ways. When it is two-dimensional, it can be represented as a planar graph. Another method, which works in arbitrary dimension, is to represent $H(N)$ by its *facial lattice*. This lattice contains a node for each j -face of $H(N)$. Each node contains auxiliary information, such as pointers to the hyperplanes containing the corresponding face. A node for a j -face f is connected to a node for a

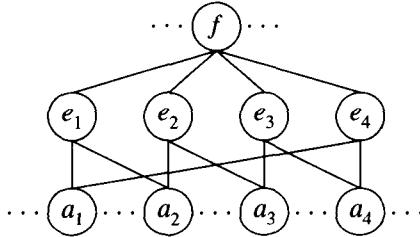


Figure 2.5: Facial lattice of an arrangement.

$(j - 1)$ -face g if f and g are adjacent, i.e., if g lies on the boundary of f . Figure 2.5 shows a part of the facial lattice for the arrangement in Figure 2.4.

Sorting and searching problems in this setting assume the following form:

The sorting problem: Given N , construct the arrangement $H(N)$.

The search problem: Construct a search structure $\tilde{H}(N)$ that can be used to locate the face of $H(N)$ containing a given query point quickly, say, in logarithmic time.

Exercises

2.2.1 Let $H(N)$ be a nonsimple arrangement of lines in the plane. Show that if the lines in N are perturbed infinitesimally, then the resulting arrangement is almost always simple. Formally, prove the following. Let ϵ be an arbitrarily small positive real number.

- (a) Rotate each line in N independently by an angle θ_i , $1 \leq i \leq n$, where $|\theta_i| \leq \epsilon$. Show that, for almost all rotations, no two of the resulting lines are parallel. This means the set of points $\{(\theta_1, \dots, \theta_n)\} \subseteq R^n$ for which this property is violated has zero volume.
- (b) Now translate each line in N independently in a fixed direction by distance ϵ_i , $1 \leq i \leq n$, where $|\epsilon_i| \leq \epsilon$. Show that, for almost all such translations, no vertex in the resulting arrangement is contained in more than two lines.

2.2.2 Denote the degree of a vertex v in the arrangement $H(N)$ by $\deg(v)$. It is always 4 when the arrangement is simple. Show that $\sum_v \deg^2(v) = O(n^2)$, where v ranges over all vertices in the arrangement $H(N)$. (Hint: Perturb all lines in N infinitesimally. The total degree of the resulting simple arrangement is $O(n^2)$.)

2.2.3 Show that the total number of faces in an arrangement of n hyperplanes in R^d is $O(n^d)$. (Hint: Use induction on d and the fact that each $(d - 1)$ -face is adjacent to two d -faces.)

Prove the following more precise bound on the number of d -cells. Define a function $\phi_d(n)$ by

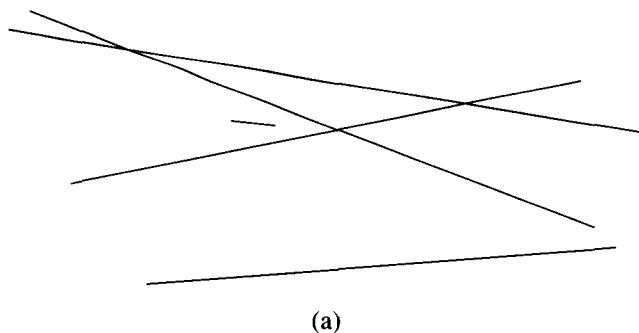
$$\phi_0(n) = 1, \phi_d(0) = 1, \text{ and } \phi_d(r) = \phi_d(r - 1) + \phi_{d-1}(r - 1), \text{ for } d, r \geq 1.$$

Show that the number of d -cells is bounded by $\phi_d(n)$. (Use induction on dimension.) Check that

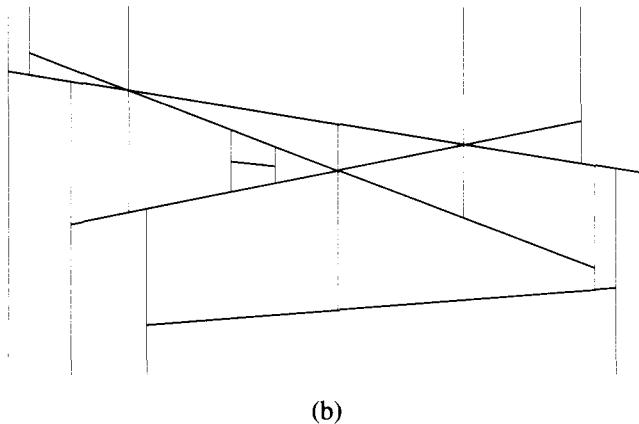
$$\phi_d(r) = \sum_{j=0}^d \binom{r}{j}, \text{ if } d < r. \text{ Otherwise, } \phi_d(r) = 2^r.$$

2.3 Trapezoidal decompositions

Another generalization of an arrangement of lines is a planar arrangement of segments (Figure 2.6(a)). The segments can be bounded or unbounded. Such arrangements arise in several applications, such as computer graphics. A very important special case arises when the segments are nonintersecting, except at the endpoints (Figure 2.7(a)). In this case, the resulting arrangement is nothing but a planar graph. Planar graphs are ubiquitous in combinatorial algorithms.



(a)



(b)

Figure 2.6: (a) An arrangement of segments. (b) Its trapezoidal decomposition.

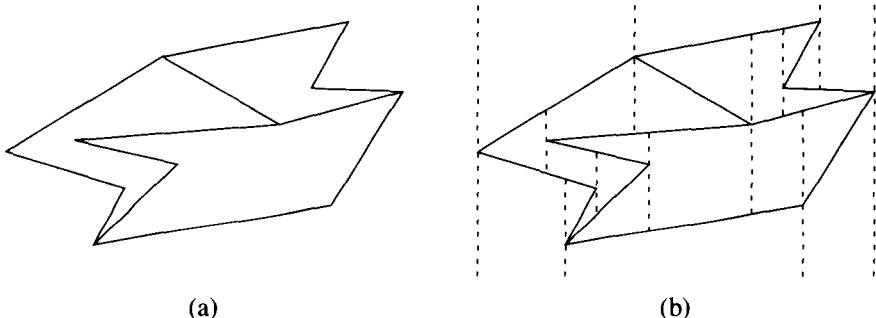


Figure 2.7: (a) A planar graph. (b) Its trapezoidal decomposition.

Let N denote the given set of segments. These segments decompose the plane into several regions. Unfortunately, the regions in the resulting decomposition can have complicated shapes. Hence, it is convenient to refine this partition further, as in Figures 2.6(b) and 2.7(b), by passing a vertical attachment through every endpoint or point of intersection. Each vertical attachment extends upwards and downwards until it hits another segment, and if no such segment exists, then it extends to infinity.¹ This yields a *trapezoidal decomposition* $H(N)$ of the plane. The regions in $H(N)$ are called trapezoids, even though they can be triangles, strictly speaking. Triangles are considered to be degenerate forms of trapezoids.

We can represent $H(N)$ in computer memory as a planar graph in an obvious fashion, i.e., by storing the specifications of its vertices, edges and regions together with the adjacency relationships among them. Each region is represented by a circular list of the vertices on its border. In the exercises, we shall describe other representations that can be more efficient in practice.

What is the size of $H(N)$? Let m denote the number of intersections among the segments in N . The segments in N are thus broken into $O(m+n)$ pieces (before passing any vertical attachments). This is assuming that no more than two segments intersect in a common point, other than an endpoint of a segment. Otherwise, m has to be replaced with the total degree of all intersections. Each vertical attachment gives rise to at most three vertices of $H(N)$. Hence, the insertion of one vertical attachment can increase the total number of vertices, edges, and regions by a constant. Since $m + 2n$ vertical attachments are inserted overall, the size of $H(N)$ is $O(m+n)$.

Sorting and searching assume the following form in the present setting.

¹In this book, the infinity is always to be taken in the symbolic sense. For example, a segment that is unbounded in one direction can be completely specified in a computer by the line containing it and its other endpoint.

The sorting problem: Given a set N of segments in the plane, construct the trapezoidal decomposition $H(N)$.

Note that once $H(N)$ is constructed, we automatically know all intersections among the segments in N . Hence, this latter intersection problem is a subproblem of the preceding construction problem.

The search problem: Construct a search structure $\tilde{H}(N)$ so that the face of $H(N)$ containing any query point can be located quickly.

A special case of this search problem arises when all segments in N are assumed to be nonintersecting, but they are allowed to share endpoints. In this case, the segments in N form a planar graph (Figure 2.7). Once we have located the trapezoid in $H(N)$ containing the given query point q , we automatically know the face in the planar graph containing q . This latter problem of locating the face in the planar graph containing the query point is also known as the planar point location problem. It arises in several applications, such as geographic applications, wherein a planar graph corresponds to a map.

The trapezoidal decompositions also serve as an important practical means of answering range searching queries when the object set in the range query problem consists of segments in the plane. Given any range, i.e., a connected region, one can locate in $H(N)$ a fixed point in the range using the point location structure $\tilde{H}(N)$, and then travel within that range in $H(N)$ starting at this fixed point (Figure 2.8). This discovers all segments intersecting the range. Traveling in $H(N)$ is easy, because all regions in $H(N)$ are trapezoids. Thus, given a trapezoid in $H(N)$ intersecting the range, we can easily determine all adjacent trapezoids that also intersect the range. This is not

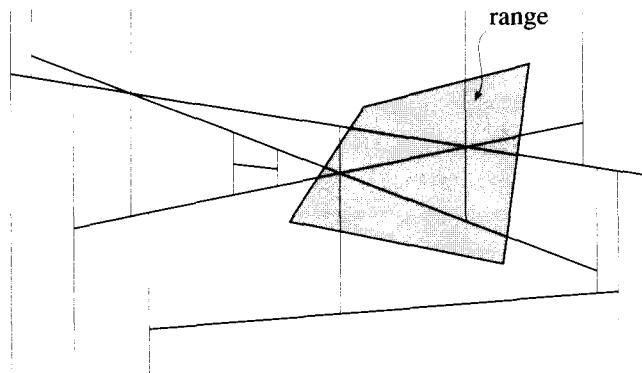


Figure 2.8: Trapezoidal decompositions in range queries.

easy when the regions in the partition have complicated shapes, as in the original unrefined arrangement of segments (Figure 2.6(a)).

In theory, one can come up with pathological ranges that intersect lots of vertical attachments in $H(N)$ but very few segments in N . In this case, one could justifiably say that the time invested in computing the intersections with the auxiliary vertical attachments is wasted. In practice, this does not cause a problem for two reasons. First, such pathological ranges are rare. Second, the number of vertical attachments that are intersected by a range is generally comparable to the number of intersected segments. The most important advantage is that this method works regardless of the shape of the range. This versatility makes it quite often preferable to the specialized methods that depend on the shape of the query range.

It is also interesting to investigate what happens when the objects in N are not segments but rather polygons in R^3 , or more generally, polyhedra in R^d . Such generalizations are very important in computer graphics and robotics. The three-dimensional instance of this problem is studied in Chapter 9.

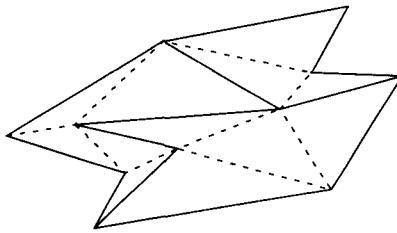


Figure 2.9: A triangulation of a planar graph.

Exercises

2.3.1 By a triangulation of a planar graph G , we mean a decomposition of its regions into triangles whose vertices are contained in the vertex set of G (Figure 2.9). A triangulation is not unique. Show that, given the trapezoidal decomposition of a planar graph, one can obtain its triangulation in linear time.

2.3.2 Consider the following opaque representation of a trapezoidal decomposition $H(N)$. Each trapezoid in $H(N)$ is represented by a circular list of its corners, which are at most four in number. We also store adjacencies between all vertices and trapezoids in $H(N)$. However, we consider a vertex adjacent to a trapezoid iff it is a corner of that trapezoid. Thus, the ends of the vertical attachments that rest on the outer side of a trapezoid are considered invisible inside that trapezoid.

Compare this opaque representation with the planar graph representation of $H(N)$. The sizes of both representations are of the same order, up to a constant factor. Which one is more space-efficient in practice? Show that one representation can be converted into another in time proportional to their size.

2.3.3 Let us call a trapezoidal decomposition $H(N)$ *simple* if each point of intersection is contained in exactly two segments and if no two vertical attachments intersect or meet.

Show that if the segments in N are perturbed infinitesimally, by translation and rotation (cf. Exercise 2.2.1), then the resulting trapezoidal decomposition is almost always simple.

2.3.4 (Euler's relation) Let G be a bounded, connected planar graph. Assume that there can be only one edge between two distinct vertices. Let μ, ϵ, ϕ denote the total number of vertices, edges, and regions in G , respectively. The graph G contains exactly one unbounded region.

- (a) Show that $\mu - \epsilon + \phi = 2$. (Hint: First, prove the relation when G is a connected tree. When G is a general connected graph, start with the relation for a maximal tree inside G . Then add the remaining edges of G , one at a time.)
- (b) Show that the total size, i.e., the total number of vertices, edges and faces, of a planar graph is $O(\mu)$. (Hint: $3(\phi - 1) \leq 2\epsilon$, because each edge is adjacent to two vertices and each bounded face is adjacent to at least three vertices.)
- (c) What is the relation when G is unbounded? (Hint: Consider a large square containing all vertices of G . Consider Euler's relation for the intersection of G with this square.)
- (d) Suppose G is bounded and all its bounded regions are triangles. Also assume that the outer boundary of G is a triangle. Show that $\epsilon = 3\mu - 6$.

***2.3.5** Let N be a set of algebraic curves of bounded degree. Let us call a point p on a curve C *critical* if it is a local x -maximum or minimum on C . Define an analogue of a trapezoidal decomposition in this case by passing vertical attachments through all critical points and points of intersection.

- (a) Show that each region in the resulting decomposition $H(N)$ is monotonic with respect to the y -coordinate. This means its intersection with any line parallel to the y -axis is connected.
- (b) Show that the maximum size of $H(N)$ is $O(n^2)$. (Hint: Use Bezout's theorem.)
- (c) Show that if the coefficients of the equations defining the curves in N are changed infinitesimally, then the resulting decomposition $H(N)$ is almost always simple. This means: (1) there are no singularities on any curve, (2) each point of intersection is contained in exactly two curves, and (3) no two vertical attachments intersect. (Hint: Use Sard's theorem.)

2.4 Convex polytopes

Convex polytopes are very special convex sets in R^d . They are central to several practical applications: operations research, image processing, pattern recognition and so on. A set in R^d is called convex if, for any two points in the set, the linear segment joining those two points is completely contained within the set. The simplest convex set in R^d is a *half-space*. It is a set of points satisfying a linear inequality of the form $a_1x_1 + \cdots + a_dx_d \geq a_0$, where the x_j 's denote the coordinates in R^d , the coefficients a_j are fixed real

numbers, and a_1, \dots, a_d are not all zero. The boundary of this half-space is the set of points satisfying the linear equality $a_1x_1 + \dots + a_dx_d = a_0$. It is called a hyperplane when d is arbitrary. When $d = 3$, it is a plane. When $d = 2$, it is a line.

Let N be any set of half-spaces in R^d . Their intersection $H(N)$, if nonempty, is called the *convex polytope* formed by N (Figure 2.10). It is indeed convex, because the intersection of convex sets is convex. In what follows, we shall assume that $H(N)$ is full-dimensional, i.e., its dimension is d . Otherwise, $H(N)$ is contained within the intersection of some hyperplanes bounding the half-spaces in N ; in this case $H(N)$ can be thought of as a full-dimensional polytope within this intersection. A convex polytope with dimension d is also called a d -polytope.

A special case of a convex polytope arises quite often, wherein all half-spaces in N extend to infinity in some fixed direction. We can assume that this direction is the positive x_d direction, changing the coordinates if necessary. We can also assume that each element in N is, in fact, a hyperplane that implicitly stands for the half-space bounded by it and extending in the positive x_d direction. We say that $H(N)$ is the *upper convex polytope* formed by the hyperplanes in N (Figure 2.10).

Figure 2.10(a) shows a two-dimensional convex polytope, which is just a convex polygon. Figure 2.10(b) shows an example of an upper convex polygon. The boundary of a convex polygon consists of several edges, also called 1-faces, and vertices, also called 0-faces. As a convention, by the 2-face of a convex polygon $H(N)$, we mean the whole polygon itself.

In general, a d -polytope $H(N)$ has one d -face, namely, itself. The intersection of $H(N)$ with the boundary ∂S of a half-space $S \in N$ is a convex polytope of lower dimension within ∂S . It is formed by the $(d - 1)$ -dimensional

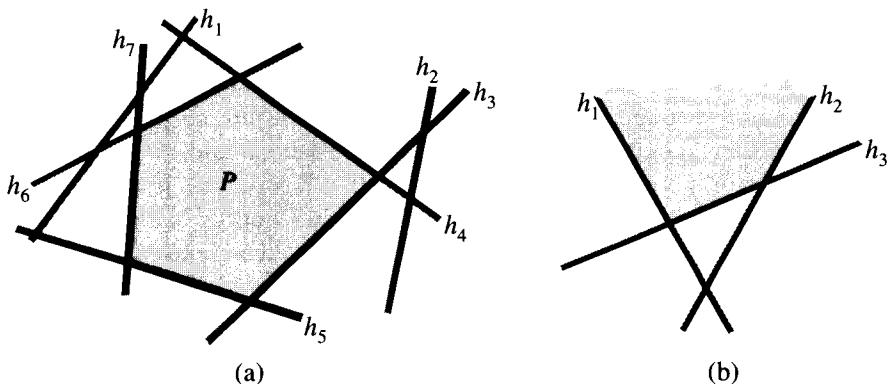


Figure 2.10: (a) A convex polytope. (b) An upper convex polytope.

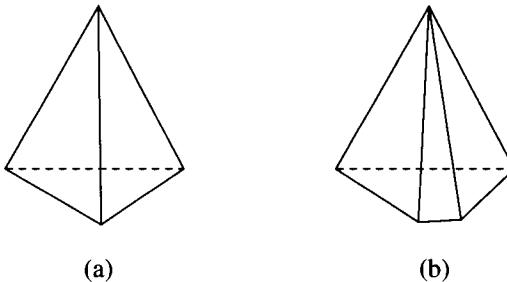


Figure 2.11: (a) A simple and simplicial polytope. (b) A nonsimple and nonsimplicial convex polytope.

half-spaces of the form $Q \cap \partial S$, where Q is any half-space in N other than S . If the dimension of $\partial S \cap H(N)$ is $d - 1$, it is called a $(d - 1)$ -face or a *facet* of $H(N)$. When $d = 3$, a facet is just a convex polygon. A $(d - 2)$ -face of $H(N)$ is a facet of its facet, i.e., a subfacet. Proceeding inductively in this fashion, we can define a j -face, i.e., a j -dimensional face, of $H(N)$ for every $0 \leq j \leq d$. A 0-face of $H(N)$ is also called a *vertex*. A 1-face is also called an *edge*. The *size* of $H(N)$ is defined to be the total number of its faces of all dimensions. By the size of a face of $H(N)$, we mean the total number of its subfaces.

A half-space in N is called *redundant* if $H(N)$ is unaffected by its removal. Otherwise, it is called *nonredundant*. In Figure 2.10(a), the half-spaces h_1 and h_2 are redundant. All other half-spaces are nonredundant. A hyperplane bounding a nonredundant half-space in N is called a *bounding* or a *nonredundant* hyperplane of $H(N)$. The polytope $H(N)$ is called *simple* if every j -face of $H(N)$ is contained in exactly $d - j$ bounding hyperplanes. The convex polytope in Figure 2.11(b) is not simple because its top vertex is contained in four bounding planes.

$H(N)$ can be represented in computer memory by its facial lattice. This contains a node for each face of $H(N)$. Each node contains auxiliary information such as pointers to the half-spaces whose bounding hyperplanes contain the corresponding face. A node for a j -face f and a $(j - 1)$ -face g are connected if f and g are adjacent, i.e., if g lies on the boundary of f . We shall study d -dimensional convex polytopes in Chapter 7. Until then, we shall be mainly interested in two- and three-dimensional convex polytopes.

The sorting problem in the present setting assumes the following form.

The sorting problem: Given a set N of half-spaces, construct the facial lattice of the convex polytope formed by them.

The most natural search problem associated with convex polytopes is the so-called linear programming problem (Figure 2.12). Here we are given a

set N of half-spaces in R^d . These half-spaces stand for linear constraints. We are also given a linear function z in R^d , which is also called an *objective function* in this context. The goal is to maximize the objective function over the convex polytope $H(N)$ formed by the half-spaces in N and determine the coordinates of a point where the maximum is achieved.

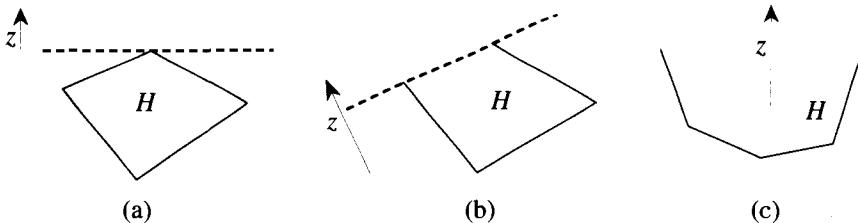


Figure 2.12: Objective function on a convex polytope: (a) A nondegenerate case. (b) A degenerate case. (c) An unbounded case.

For example, the variables x_1, \dots, x_d could stand for the production quantities of various goods produced in a factory. Each linear constraint in N could stand for a production constraint, which could arise for various reasons. The objective function $z = b_1x_1 + \dots + b_dx_d$ could stand for the total profit, where b_i stands for the profit over one unit of the i th good. The goal is to determine the production quantities x_i so that the profit is maximized.

The linear function z need not always have a maximum on the convex polytope $H = H(N)$. This happens if H is unbounded in the z -direction (Figure 2.12(c)). Even if a z -maximum exists, it need not be unique (Figure 2.12(b)). In this case, we say that z is degenerate over $H(N)$. However, we shall see later that a z -maximum, if it exists, is “generally” unique. In other words, z is generally nondegenerate (Figure 2.12(a)). The usual form of linear programming is the following:

The search problem (linear programming): Given a set N of half-spaces and a linear function z , find the coordinates of a z -maximum on $H(N)$, if any.

In this form of linear programming, we are only interested in computing the coordinates of a z -maximum. We do not care for the facial lattice of the whole polytope $H(N)$.

Another useful form of linear programming is the following:

The search problem (linear programming, second version): Given a set N of half-spaces, build a search structure $\bar{H}(N)$, so that given any linear function z , a z -maximum (if any) on $H(N)$ can be found quickly, say, in polylogarithmic time.

Exercises

2.4.1 Let Z be any set in R^d . Let x be any point outside Z . Define $\text{join}[Z, x]$ to be the union of all segments of the form $[y, x]$, where y ranges over all points in Z . Assume that Z is convex. Show the following:

- (a) $\text{Join}[Z, x]$ is convex and it is the smallest convex set containing Z and x .
- (b) If Z is a j -dimensional simplex, $j < d$, then $\text{join}[Z, x]$ is a $(j+1)$ -dimensional simplex, assuming x is not contained in the j -dimensional affine (linear) space containing Z .
- (c) Assume that Z is full d -dimensional. Let y be any point in ∂Z , the boundary of Z . Let $l(x, y)$ be the ray emanating from x and passing through y . Because Z is convex, $l(x, y) \cap Z$ is a connected segment or a point. We say that y belongs to the silhouette of Z (with respect to x) if $l(x, y) \cap Z$ is a point. Otherwise, we say that y belongs to the front of Z (with respect to x) if y is nearer to x than the other endpoint of $l(x, y) \cap Z$. When y is farther, we say that y belongs to the back of Z . Show that the boundary of $\text{join}[Z, x]$ is the union of the back of Z and $\text{join}[\hat{\partial}Z, x]$, where $\hat{\partial}Z$ denotes the silhouette of Z with respect to x .
- (d) If Z is a convex polytope, then $\text{join}[Z, x]$ is also a convex polytope. If Z is full d -dimensional, each i -face of $\text{join}[Z, x]$ is either an i -face that is contained in the union of the back and the silhouette of Z , or it is of the form $\text{join}[\sigma, x]$, where σ is an $(i-1)$ -face of Z contained in the silhouette of Z .

2.4.2 Show that if A is any finite set of points in R^d , the smallest convex set containing A is a convex polytope whose vertices are contained in A . It is called the *convex hull* of A . (Hint: Use induction on the size of A and Exercise 2.4.1.)

2.4.3 Show that $Z \subseteq R^d$ is a convex set iff it contains every convex combination of any finite subset of points from Z . By a convex combination of points x_1, \dots, x_t , we mean $\sum_{j=1}^t \lambda_j x_j$, where $\lambda_j \geq 0$, $\sum_j \lambda_j = 1$. Here each point is considered as a vector of its coordinates, and the sum is the vector sum.

2.4.4 Let A be any finite set of points in R^d . Show that the set consisting of the convex combinations of all finite subsets of A is convex, and that it is the smallest convex set containing A , i.e., the convex hull of A . (Hint: Use Exercise 2.4.3.)

2.4.5 Let N be a set of half-spaces. Show that the convex polytope $H(N)$ is the convex hull of its vertices, assuming that $H(N)$ is bounded.

2.4.6 Let N be a set of half-spaces in R^d . If the half-spaces in N are independently perturbed by infinitesimal translations in a fixed direction (see Exercise 2.2.1), then the convex polytope formed by the resulting set of half-spaces is almost always simple and its number of j -faces, for any j , is greater than or equal to the number of j -faces of $H(N)$. (Hint: To prove the latter statement, perturb one half-space at a time.)

2.4.1 Duality

A notion that is closely related to a convex polytope is that of a *convex hull*. Given a set \hat{N} of points in R^d , its convex hull $H(\hat{N})$ is defined to be the

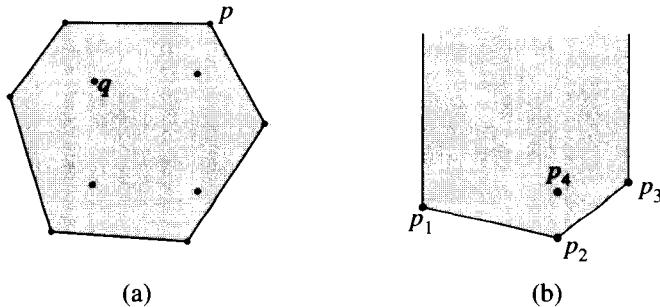


Figure 2.13: (a) A convex hull. (b) An upper convex hull.

smallest convex set containing all points in \hat{N} .² $H(\hat{N})$ is a convex polytope, whose vertices are contained in \hat{N} (Exercise 2.4.2). Not all points in \hat{N} necessarily occur as the vertices of $H(\hat{N})$. We call a point in \hat{N} *extremal* if it is a vertex of $H(\hat{N})$. Otherwise, it is called *nonextremal*. A simple case of a convex hull for $d = 2$ is shown in Figure 2.13(a). There point p is extremal, whereas q is nonextremal. The simplest example of a convex hull in R^d is a d -simplex formed by $(d + 1)$ -points that are not contained in a common hyperplane. A 2-simplex is just a triangle. A 1-simplex is just an edge. The convex hull is called *simplicial* if all its j -faces are simplices of dimension j . The convex hull in Figure 2.11(b) is not simplicial, because its bottom facet is not a simplex.

A special kind of a convex hull, called an *upper convex hull*, arises quite often in practice. Given a set \hat{N} of points in R^d , the upper convex hull of \hat{N} is defined to be the convex hull of the set $\hat{N} \cup \{(0, \dots, 0, \infty)\}$ (Figure 2.13(b)).

Convex polytopes and convex hulls are actually dual geometric structures. There is a very simple geometric map that transforms a convex polytope into a convex hull and vice versa. Such duality transformations play an important role in computational geometry because they allow us to transform one problem into another.

Let $H(N)$ be any d -polytope formed by a set N of half-spaces in R^d . Assume that the origin lies strictly in the interior of $H(N)$. Thus, we might as well assume that each element in N is a hyperplane, which implicitly stands for the half-space bounded by it containing the origin. Consider the transform T that maps a point $p = (p_1, \dots, p_d) \in R^d$ to the hyperplane $\langle p, x \rangle = p_1x_1 + \dots + p_dx_d = 1$, and vice versa. Here $\langle p, x \rangle$ denotes the vector inner product, and we assume that p_1, \dots, p_d are not all zero. (The transform

²In this section, $H(\hat{N})$ denotes the convex hull of \hat{N} , when \hat{N} is a set of points, and $H(N)$ denotes the convex polytope formed by N , when N is a set of half-spaces.

is not well defined on the hyperplanes passing through the origin.) Note that $T(T(p)) = p$. Let \hat{N} denote the set of points in R^d obtained by transforming the hyperplanes in N (Figure 2.14).

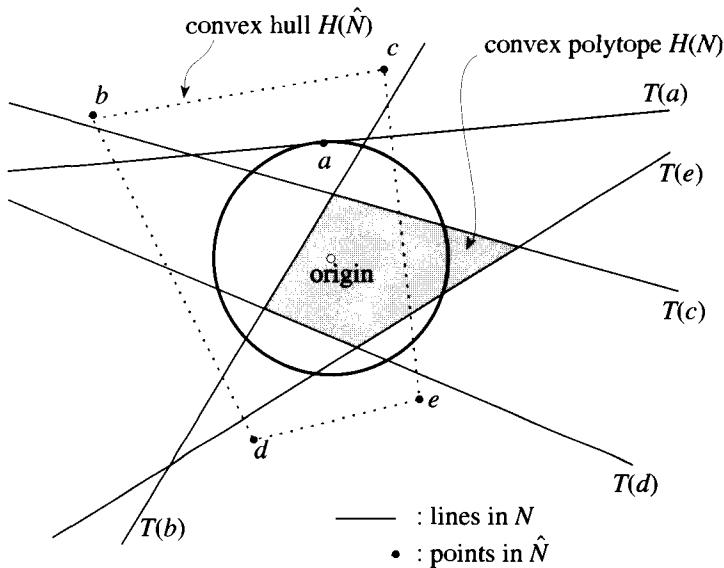


Figure 2.14: Dual transform T .

The transform T maps a point p lying on the unit sphere $U : x_1^2 + \dots + x_d^2 = 1$ to the hyperplane tangent to U at p . In general, it maps a point p at distance d from the origin to the hyperplane at distance $1/d$ from the origin which is perpendicular to the ray emanating from the origin and passing through p .

For a hyperplane h in R^d , let h^- denote the half-space bounded by this hyperplane containing the origin, and let h^+ denote the other half-space. If the hyperplane h is identified with the coefficient vector (h_1, \dots, h_d) in its normal-form equation $h_1x_1 + \dots + h_dx_d = 1$, then for any point $p = (p_1, \dots, p_d)$,

$$\langle p, h \rangle = \langle T(p), T(h) \rangle. \quad (2.1)$$

Moreover, a point p lies on the hyperplane h iff $\langle p, h \rangle = 1$. It lies in h^+ iff $\langle p, h \rangle \leq 1$. It follows from (2.1) that the transformation T has the following two important properties:

1. Incidence invariance: A point p lies on a hyperplane h iff the point $T(h)$ lies on the hyperplane $T(p)$.
 2. Inclusion invariance: A point p lies in a half-space h^+ (resp. h^-) iff $T(h)$ lies in $T(p)^+$ (resp. $T(p)^-$).

These properties imply the following (Figure 2.14):

1. A hyperplane h whose bounding half-space h^+ contains the points of \hat{N} strictly in its interior is transformed to a point in the interior of $H(N)$.
2. More generally, a hyperplane h that contains a subset $\hat{J} \subset \hat{N}$ of points and whose bounding half-space h^+ contains the remaining points of \hat{N} strictly in its interior is transformed to a point in the interior of the face $H(N) \cap \bigcap_{p \in \hat{J}} T(p)$.

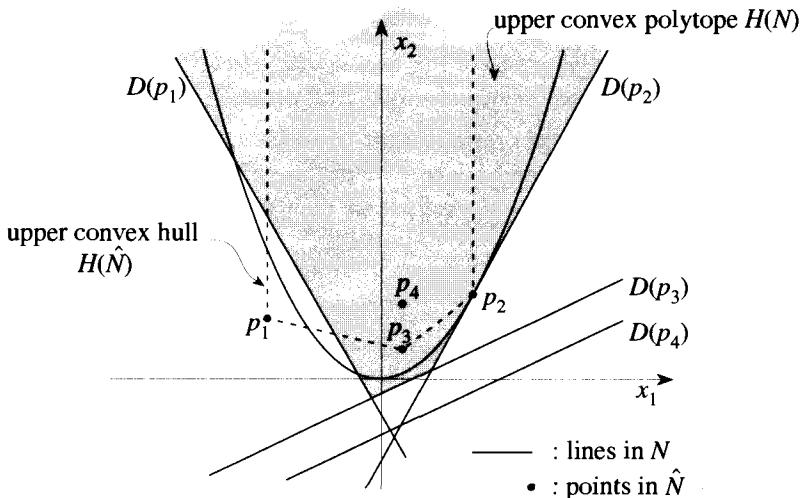
This means:

1. Redundant hyperplanes in N are transformed to nonextremal points in \hat{N} , and vice versa.
2. Nonredundant hyperplanes in N are transformed to extremal points in \hat{N} , and vice versa. Thus, the facets of $H(N)$ are in one-to-one correspondence with the vertices of $H(\hat{N})$.
3. More generally, there is a natural one-to-one correspondence between the j -faces of $H(N)$ and the $(d - j - 1)$ -faces of $H(\hat{N})$, for $0 \leq j \leq d - 1$. A j -face of $H(N)$ contained in the hyperplanes S_1, S_2, \dots in N corresponds to the face of $H(\hat{N})$ which is the convex hull of the points $\hat{S}_1, \hat{S}_2, \dots$.
4. If $H(N)$ is simple, $H(\hat{N})$ is simplicial, and vice versa.

Similarly, upper convex polytopes and upper convex hulls are dual geometric structures. A duality transformation between them is obtained by replacing the unit sphere in the transformation T by a paraboloid $x_d = x_1^2 + \dots + x_{d-1}^2$, and letting $(0, \dots, 0, \infty)$ play the role of the origin. Formally, consider the transformation D that maps each point (p_1, p_2, \dots, p_d) to the hyperplane $x_d = 2p_1x_1 + \dots + 2p_{d-1}x_{d-1} - p_d$, and vice versa. This maps a point p lying on the paraboloid $U : x_d = x_1^2 + \dots + x_{d-1}^2$ to the hyperplane tangent to U at p . (The transform is not well defined on the hyperplanes parallel to the x_d -axis.) Moreover, if two points p_1 and p_2 lie on a line parallel to the x_d -axis, then the transformed hyperplanes are parallel and have the same vertical distance between them (Figure 2.15).

Now let N be a set of hyperplanes in R^d . For any hyperplane $h \in N$, let h^+ denote the half-space bounded by h extending in the positive x_d -direction, and let h^- denote the other half-space. Transform D has the following properties, just like the transform T :

1. Incidence invariance: A point p lies on a hyperplane h iff the point $D(h)$ lies on the hyperplane $D(p)$.
2. Inclusion invariance: A point p lies in a half-space h^+ (resp. h^-) iff $D(h)$ lies in $D(p)^+$ (resp. $D(p)^-$).

Figure 2.15: Dual transform D .

Let \hat{N} be the set of points obtained by transforming the hyperplanes in N . The preceding properties immediately imply that the upper convex polytope of N and the upper convex hull of \hat{N} are dual objects (Figure 2.15). There is a natural one-to-one correspondence between the j -faces of $H(N)$ and the $(d - j - 1)$ -faces of $H(\hat{N})$, for $0 \leq j \leq d - 1$: A j -face of $H(N)$ contained in the intersection of the hyperplanes S_1, S_2, \dots in N corresponds to the $(d - j - 1)$ -face of $H(\hat{N})$ which is the convex hull of the points $\hat{S}_1, \hat{S}_2, \dots$.

The following is a dual of the sorting problem in the previous section:

The sorting problem: Given a set \hat{N} of points in R^d , construct its convex hull.

The following is a dual of the linear programming problem in the previous section:

The search problem (Ray shooting problem, first version): Let \hat{N} be a given set of points in R^d . Assume that $H(\hat{N})$ contains the origin. Given a ray emanating from the origin, find the facet of $H(\hat{N})$ intersected by this ray.

The search problem (Ray shooting problem, second version): Let \hat{N} be a given set of points in R^d . Build a search structure $\tilde{H}(\hat{N})$ so that given a ray emanating from the origin, one can locate the facet of $H(\hat{N})$ intersected by this ray quickly, say, in polylogarithmic time.

Convex polytopes and convex hulls in arbitrary dimension are studied in Chapter 7. Until then, we shall only be concerned with dimensions two

and three. The problems concerning convex polygons in the plane are generally simple, because the boundary of a convex polygon is one-dimensional. In fact, it can be represented by a sorted circular list of its vertices. For this reason, the problems dealing with convex polygons generally exhibit characteristics of the one-dimensional problems dealing with linear lists, like the ones treated in Chapter 1. Dimension three, in comparison, is more interesting and nontrivial. The problems connected with convex polytopes and convex hulls in R^3 exhibit, to a large extent, characteristics of planar problems. This is because the edges of a three-dimensional convex hull or polytope form a planar graph. This is evident when we remove the interior of any 2-face and then flatten out its remaining boundary. It follows from Euler's relation (Exercise 2.3.4) that the size of a three-dimensional convex hull is $O(m)$, where m is the number of its external points. Dually, the size of a three-dimensional convex polytope is also $O(m)$, where m now denotes the total number of nonredundant half-spaces.

The use of the duality transforms described in this section is not restricted to convex polytopes alone. Indeed, duality is a persistent theme in computational geometry, which is used to transform one problem into another. To emphasize this point, let us give another application of duality. Consider the half-space range query problem in Section 2.1. Here N is a set of points in R^d . Given a half-space range h , the goal is to report or count all points of N contained in h . If we use the transformation D described before to map the points in N to hyperplanes in R^d , and similarly map the hyperplane \bar{h} bounding h to the point $D(\bar{h})$, then the half-space query problem is seen to be equivalent to the following problem: Report (or count) all the hyperplanes in $D(N) = \{D(a) \mid a \in N\}$ that lie above or below the point $D(\bar{h})$, depending upon whether h lies below or above \bar{h} . Compare Figures 2.2 and 2.16. If we fix any d -cell Δ in the arrangement formed by the hyperplanes in $D(N)$,

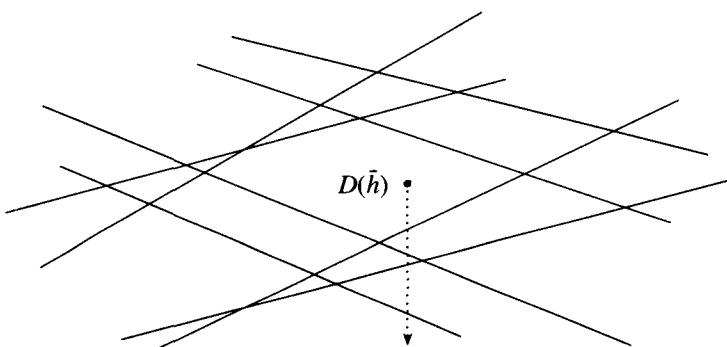


Figure 2.16: Half-space range queries in dual space.

then the set of hyperplanes above (or below) a point $p \in \Delta$ remains the same, regardless of where p is actually located in Δ . This naturally points out a close connection between half-space range queries and point location in arrangements of hyperplanes.

Exercises

2.4.7 Let \hat{N} be a set of points in R^d . If we perturb the points in \hat{N} infinitesimally, show that the convex hull of the resulting set of points is almost always simplicial. (Hint: Use duality and Exercise 2.4.6.)

2.4.8 Verify that linear programming and ray shooting are indeed dual problems. (Hint: The points on a ray emanating from the origin are transformed into a family of hyperplanes with the same normal direction.)

Verify that the maximum of a linear function on a convex polytope $H(N)$ corresponds to the facet of $H(\hat{N})$ intersected by the dual ray.

2.4.9 Consider the following search problem: Given a line l in R^d , find its intersection with the convex hull $H(\hat{N})$. What is the dual of this problem? (Hint: The points on l are transformed into a family of hyperplanes. How is this family parametrized?)

2.4.10 Consider the following query problem: Given a query point p , decide whether it is inside or outside the polytope $H(N)$. Show how this query problem can be reduced to the linear programming problem.

2.5 Voronoi diagrams

Suppose we are given a set N of sites in R^d . The following is an important search problem:

The nearest neighbor problem: Build a search structure so that, given any query point p , the site in N nearest to p can be located quickly.

As an example, the sites in question could correspond to locations of the post offices in a given area. This nearest neighbor problem is a problem of great practical interest in computational geometry. It occurs in various forms in several applications, such as speech or pattern recognition, best-match information retrieval, and so on. There is a simple partition of the plane induced by N , called its *Voronoi diagram*, which can be used to solve the above problem.

The *Voronoi region* $\text{vor}(S)$ of a site $S \in N$ is defined to be the region consisting of the points in R^d whose nearest neighbor in N is S (Figure 2.17). This region is always convex. This can be seen as follows. For each $Q \in N$ other than S , the points that are nearer to S than Q form a half-space $h(S, Q)$. This half-space is bounded by the hyperplane that is the perpendicular bisector of the segment (S, Q) . $\text{Vor}(S)$ is just the intersection of these half-spaces.

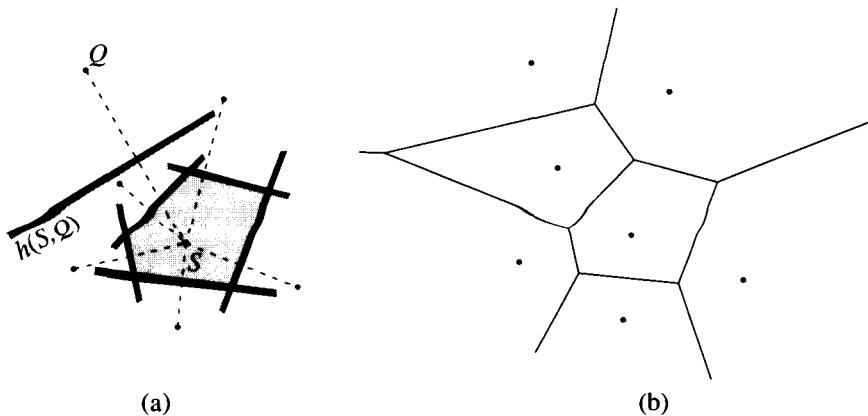


Figure 2.17: (a) A Voronoi region. (b) A Voronoi diagram.

The partition $H(N)$ of R^d into the regions $\text{vor}(S)$, $S \in N$, is called the Voronoi diagram of N (Figure 2.17). The Voronoi regions in $H(N)$ are called its d -faces or simply cells. Each Voronoi region is a convex polytope. By a j -face of $H(N)$, $j < d$, we mean a j -face of its Voronoi region. A planar Voronoi diagram can be represented in computer memory as a planar graph. In general, $H(N)$ can be represented by its facial lattice. This contains a node for every face of $H(N)$. A node for a j -face f and a $(j-1)$ -face g are connected if they are adjacent, i.e., if g is contained in the boundary of f .

To find the nearest neighbor of a given query point q , we need to locate the Voronoi region in $H(N)$ containing q . Thus sorting and searching in this context assumes the following form:

The sorting problem: Given a set of sites N , construct its Voronoi diagram $H(N)$.

The search problem: Construct a search structure $\tilde{H}(N)$ so that, given a query point p , the nearest neighbor of p in N can be located quickly.

If one looks at Figure 2.17(b) carefully, it seems as if it were obtained by projecting the boundary of some three-dimensional convex polytope. This is, in fact, always the case.

Every Voronoi diagram in R^d can be obtained by vertically projecting the boundary of a convex polytope in R^{d+1} (Figure 2.18). Let N be the given set of sites in R^d . Identify R^d with the hyperplane $\{x_{d+1} = 0\}$ in R^{d+1} . Vertically project every site $a \in N$ onto the point \bar{a} on the unit paraboloid

$$U : x_{d+1} = x_1^2 + \cdots + x_d^2.$$

Let $D(\bar{a})$ denote the hyperplane that is tangent to U at \bar{a} . Let $G(N)$ denote the upper convex polytope formed by the hyperplanes $D(\bar{a})$, $a \in N$. It is easy to see that the map from a to $D(\bar{a})$ has the following property (Figure 2.18):

Distance transformation: For any point $q \in R^d$ and any site $c \in N$, the square of the distance between q and c is the same as the vertical distance between \bar{q} and the hyperplane $D(\bar{c})$.

In Figure 2.18, $d(c, q) = d(q, c)$ denotes the distance between q and c and $d(\bar{q}, D(\bar{c}))$ denotes the vertical distance between \bar{q} and the hyperplane $D(\bar{c})$. In this figure, $d(q, c)$ is less than one. That is why $d(\bar{q}, D(\bar{c}))$ is smaller than $d(q, c)$. The distance transformation property implies that the nearest site in N to q corresponds to the nearest hyperplane in $D(\bar{N}) = \{D(\bar{a}) \mid a \in A\}$ below \bar{q} (in the negative x_{d+1} -direction). In other words, the Voronoi region of a site $a \in N$ is obtained by vertically projecting the facet of the upper convex polytope $G(N)$ supported by $D(\bar{a})$ onto the hyperplane $\{x_{d+1} = 0\}$. Thus, the Voronoi diagram of N can be obtained by vertically projecting the boundary of the upper convex polytope $G(N)$. Note that each hyperplane $D(\bar{a})$, $a \in N$, is nonredundant.

Figure 2.18 illustrates the connection between Voronoi diagrams and convex polytopes for the simplest case $d = 1$. Note that the vertices of the Voronoi diagram of a set of sites on a line are simply bisectors of the intervals between consecutive sites.

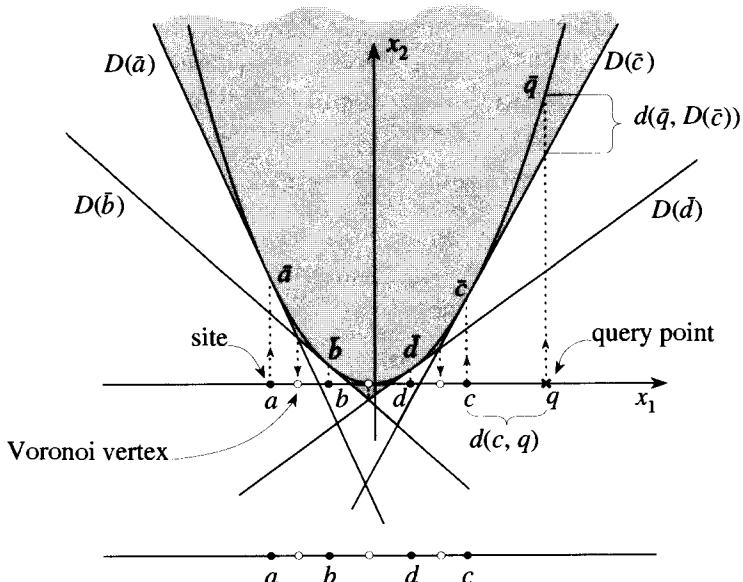


Figure 2.18: Voronoi diagram via projection.

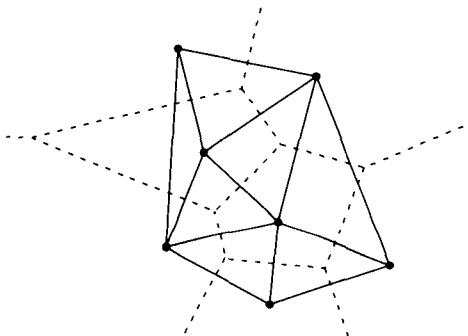


Figure 2.19: A Delaunay triangulation.

Since the size of a three-dimensional convex polytope is linear in the number of bounding planes (Section 2.4.1), it follows from the above connection that the size of a Voronoi diagram formed by n sites in the plane is $O(n)$.

The map D that transforms a projected point \bar{a} to the tangent hyperplane $D(\bar{a})$ coincides with the duality map used in Section 2.4.1. Hence, the dual of the upper convex polytope $G(N)$ is the upper convex hull of the points in $\bar{N} = \{\bar{a} \mid a \in N\}$. Let us vertically project the boundary of this convex hull onto R^d . This projection is called the *Delaunay triangulation* of the sites in N . The terminology arises from the fact that this projection is “generally” a triangulation in two dimensions (Exercise 2.5.2). Figure 2.19 shows the Delaunay triangulation of the sites in Figure 2.17. Duality between Voronoi diagrams and Delaunay triangulations implies that there is a Delaunay edge between two sites in the plane iff their Voronoi regions share an edge.

Since the Delaunay triangulation of a set N of sites in R^d is the projection of a convex hull in R^{d+1} , it must cover a convex region in R^d . Moreover, all vertices on the boundary of the Delaunay triangulation belong to N . It follows that the convex hull of N must coincide with the outer boundary of its Delaunay triangulation (Figure 2.19).

So far, we described Voronoi diagrams and Delaunay triangulations as a means of solving the nearest neighbor problem. We should remark here that in quite a few applications—in ecology, biochemistry, solid state physics, and so on—the construction of a Voronoi diagram or a Delaunay triangulation is an end in itself. For example, in biochemistry, the structure of a molecule can be determined by an interplay between various forces, which can be probed by constructing elaborate Voronoi diagrams. Thus, in a given application, one might be interested only in the sorting problem but not in the search problem. The opposite situation can also hold—one might only be interested

in answering nearest neighbor queries. Whether this is done by explicitly constructing a Voronoi diagram or not may be irrelevant.

In the exercises below, we describe several other applications of Voronoi diagrams.

Exercises

2.5.1 Verify the distance transformation property.

2.5.2 Call a Delaunay triangulation in R^d simplicial if all of its cells (d -faces) are simplices. We call a Voronoi diagram simple if the dual Delaunay triangulation is simplicial. Perturb the sites in N infinitesimally. Show that the Delaunay triangulation of the resulting set of sites is almost always simplicial. (Hint: Use Exercise 2.4.7.)

In particular, the regions in a planar Delaunay triangulation are almost always triangles. This explains the terminology.

2.5.3 Get an exact bound on the size of a two-dimensional Voronoi diagram. Show that the number of vertices in it is at most $2n - 5$ and that the number of edges is at most $3n - 6$. Show that this bound is actually attained.

2.5.4 Let N be a set of sites in R^d . Show that the sphere circumscribing any cell Δ in its Delaunay triangulation is empty, i.e., it does not contain a site that is not a vertex of that cell. Such a sphere is called a Delaunay sphere. (Hint: Show that the center of the sphere circumscribing Δ coincides with the vertex dual to Δ in the Voronoi diagram formed by N . This is the unique vertex that is incident to the Voronoi regions of the sites in question.)

More generally, let $a_1, a_2, \dots \in N$ be any subset of sites that define a unique sphere containing them. The number of such sites is generally $d + 1$. But it can be more than $d + 1$ if the input is degenerate. Let v the intersection of $D(\bar{a}_1), D(\bar{a}_2), \dots$. Show that it is a vertex. Show that a site $b \in N$ can lie in the interior of the sphere iff v is below the hyperplane $D(\bar{b})$. (Hint: Use the distance transformation property of the transform D .)

2.5.5 Let N be a set of sites in R^d . Show that the convex hull of N is the vertical projection of the silhouette with respect to $(0, \dots, 0, -\infty)$ of the convex hull of $\{\bar{a} \mid a \in N\}$ (Cf. Exercise 2.4.1.)

Show that a site in N is an extremal point on this convex hull iff its Voronoi region is unbounded.

2.5.6 Consider the following farthest neighbor problem. Let N be a set of points in R^d . Given a query point q , the problem is to find the site in N that is farthest from q . Give an analogue of a Voronoi diagram in this situation. (Hint: Consider the lower half-spaces bounded by the hyperplanes $D(\bar{a})$, $a \in N$. Their intersection is a lower convex polytope. Project it onto R^d .)

2.5.7 Given any partition N_1, N_2 of the set N of points in the plane, show that the shortest segment joining a point in N_1 and a point in N_2 is an edge of the Delaunay triangulation of N . (Hint: The circle whose diameter is the shortest segment $[p, q]$, $p \in N_1, q \in N_2$, cannot contain any other point in N . Hence, the nearest neighbor of every point in the interior of this segment is either p or q .)

2.5.8 (Euclidean minimum spanning trees) Given a set N of points in the plane, the problem is to find the tree of minimum total length whose vertices are the given points. For example, the problem can occur when one is asked to design a minimum-cost road network linking a set of towns. Consider the following algorithm, which maintains a queue of disjoint trees. The initial trees in the queue are the points in N .

While the queue contains more than two trees, pick the tree T in the front of the queue, and do:

- Find the shortest segment that connects a leaf of T to a point in $N \setminus T$. By the previous exercise, this segment is a Delaunay edge.
- Merge T with the tree containing the other endpoint of this segment. Put the merged tree at the back of the queue.

Assume that the Delaunay triangulation of N is known. So the first step is carried out by examining the Delaunay edges adjacent to the leaves of T . Show that the algorithm runs in $O(n \log n)$ time.

Modify the algorithm so that it runs in $O(n)$ time, once the Delaunay triangulation of N is known.

2.5.9 (Euclidean traveling salesman) The problem is to find the shortest closed path (tour) through a given set of points in the plane. Show how the Euclidean minimum-spanning tree can be used to obtain a tour whose length is less than twice the length of the shortest tour.

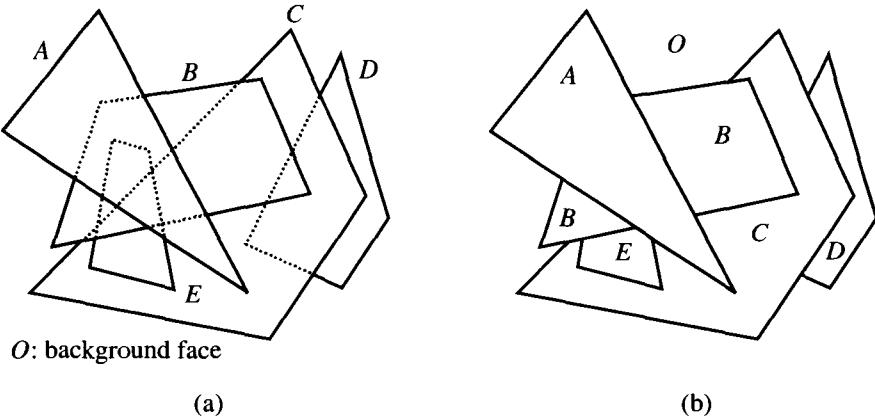


Figure 2.20: Hidden surface removal.

2.6 Hidden surface removal

Hidden surface removal is a basic problem in computer graphics. Let N be a set of opaque polygons in R^3 . Given a view point in R^3 , the problem is to determine the scene that is visible from the given view point (Figure 2.20).

Assume for the sake of simplicity that the polygons in N do not intersect, but we shall allow them to share edges. We shall assume that there is a

special background face behind all the polygons in N . We shall also assume that the view point is located at $(0, 0, -\infty)$; it turns out that the general problem is reducible to this special problem. Thus, we are dealing with simple orthographic projections now. Project visible parts of the scene edges onto the view plane $\{z = 0\}$. This gives rise to a certain planar graph in the view plane. Let us label each face of this planar graph with the unique polygon in N that is visible there. This labeled planar graph will be called the *visibility map (graph)³* of N . The sorting problem in this scenario is: Given N , find the visibility map $H(N)$ (Figure 2.20(b)).

2.7 Numerical precision and degeneracies

Let us add a few words regarding the exact model of computation that we shall be using in this book; in fact, we have already used it implicitly. In computational geometry, it is quite customary to use the so-called *real model of computation*, also called the *real-RAM model of computation*.⁴ In this model, it is assumed that each memory location or register in the computer can hold an arbitrarily long real number, and all basic operations of the computer, such as $+, -, \times, \div$, movement of an operand, are assigned a unit cost, regardless of the bit lengths of the operands. By the bit length of an operand, we mean the length of its binary representation. The advantage of this model of computation is that it frees us from the dull affair of keeping track of bit lengths. The disadvantage is that it can be misused. For example, one may come up with an algorithm that manages to keep the number of real operations low by performing them cleverly on integers with very large bit lengths. On the actual computer, such operations would need to be carried out using extended precision integer arithmetic. So it would be hardly fair to assign them unit costs. Fortunately, for all algorithms in this book, it is straightforward to show that the bit lengths of all intermediate operands are small enough. Keeping track of these bit lengths is easy and routine, but not illuminating. That is why it makes sense to use the real-RAM model of computation, and thereby ignore the complexity issues regarding bit lengths.

What cannot be ignored, however, is the issue of numerical stability. In the real-RAM model of computation, each arithmetic operation is assumed to be carried out with infinite precision. Unfortunately, real computers do

³The term *visibility map* is preferable because the term *visibility graph* has some other meanings in computational geometry.

⁴Here RAM stands for Random Access Memory, i.e., a memory whose any location can be accessed in almost the same amount of time. The word random here is somewhat misleading; it has nothing to do with the random nature of algorithms. That is why we prefer the first terminology, which, of course, has a drawback, too: The model is by no means faithful to the reality!

not follow the real model! Dealing with the finite nature of actual computers is an art that requires infinite patience. The problem of ensuring algorithm correctness in the presence of rounding errors is becoming increasingly important these days. A rule of thumb that is emerging is the following: The simpler the algorithm, the simpler it is to implement it in a numerically robust fashion. This might seem like a tautology. But if you ignore it, you might realize its truth the hard way. It is here that one aspect of randomized algorithms becomes very crucial, namely, that they are very simple.

The same aspect becomes crucial in dealing with another real life challenge posed by the so-called *degeneracies*. The simplest example is provided by a degenerate arrangement of lines, wherein a vertex in the arrangement may be contained in more than two lines. Degenerate configurations are rare. What this formally means is the following. Let N denote the set of geometric objects in the given problem. Think of each object in N as a solid geometric object in R^d , and perturb every object independently, through an infinitesimal translation and rotation. Then it can be easily proven, at least for all problems that arise in this book, that the resulting configuration contains no degeneracies, for almost all such perturbations—formally, this means that the set of perturbations that yield degenerate configurations has zero measure; e.g., see Exercises 2.2.1 and 2.4.6. The nondegenerate configurations are also sometimes called *simple*.

Since degenerate configurations are rare, it is customary in computational geometry to assume *a priori* that the objects in N are in *general position*; in other words, the positions of the objects are such that the configurations that arise in the course of the algorithm have no degeneracies. But what happens if the actual input to the algorithm gives rise to degenerate configurations during the execution of the algorithm? Of course, one could always imagine perturbing the objects in N infinitesimally before starting the algorithm. But imaginary perturbations would be useless. What one needs is a concrete way of perturbing the objects which would guarantee that the resulting object set N cannot give rise to any degeneracies. There are several symbolic ways for ensuring this. For example, one can fix a symbolic parameter ϵ to stand for a positive, infinitesimally small real number. One then adds a suitable power of ϵ to each coordinate in the specification of an object in N . One can arrange these powers in such a way that the configuration formed by the resulting object set is guaranteed to be nondegenerate, for almost all values of ϵ (e.g., see the exercise below). Now the quantities that arise, when we run the algorithm on this perturbed object set, would not be real numbers, but rather polynomials, or ratios of polynomials, in ϵ . We perform the arithmetic operations on these quantities following the usual rules of polynomial arithmetic. The only interesting situation occurs when one has

to execute a branch operation in the algorithm. A typical branch operation would depend on the sign of some quantity. How do we decide its sign? Since the quantity is a rational function in ϵ , we only need to be able to decide the signs of polynomials in ϵ . But that is quite easy: Since ϵ is supposed to stand for an infinitesimally small, positive number, the sign of a polynomial in ϵ is just the sign of its term with the smallest power in ϵ . As long as the dimension of the problem is fixed, one can show that such symbolic simulation of nondegeneracy can only increase the running time of the algorithm by at most a constant factor (at least for all algorithms in this book). Conceptually, this simplifies our task considerably, because we can now assume that the configurations that arise in our algorithms are always nondegenerate.

On the other hand, the constants within the Big-Oh notation for the above simulation can be quite large. So in practice degeneracies are quite often dealt with in a direct fashion. This requires a programmer to analyze all the kinds of degeneracies that could possibly arise in the algorithm, and then ensure that the algorithm will handle them correctly. The situation gets even worse when one has to ensure correct behaviour in the presence of degeneracies *and* rounding errors. Undoubtedly, this is a painstaking task. But quite often there is no way out when the speed is at stake. Here, too, the rule of thumb is the same as before: *The simpler the algorithm, the simpler it is to implement it correctly.* The importance of randomized algorithms stems from the fact that they can be simple as well as efficient.

The methods that are used to handle degeneracies and floating point errors are beyond the scope of this book, because the basic problems that arise in this connection are quite often algebraic rather than geometric. In this book, we shall use the real-RAM model of computation. We shall also assume, unless otherwise specified, that the objects in the input set N and also the objects involved in the queries are always in general position; degeneracies will be handled mostly in the exercises.

But we should always keep in mind the aforementioned rule of thumb. This will enable us to appreciate the simplicity and beauty of randomized algorithms fully.

Exercise

2.7.1 Let N be any set of hyperplanes in R^d , not necessarily in general position. Let S_0, \dots, S_{n-1} be any enumeration of these hyperplanes. Write the equation of each hyperplane S_i in the form $x_d = a_1^i x_1 + \dots + a_{d-1}^i x_{d-1} + a_d^i$. Let ϵ be a symbolic parameter that stands for an infinitesimally small, positive real number. To each coefficient a_j^i , add (2^{di+d-j}) th power of ϵ . Show that the resulting arrangement is simple, for every sufficiently small $\epsilon > 0$.

Let us denote the set of perturbed hyperplanes by N again. One primitive operation that is repeatedly needed in any algorithm dealing with N is the following: Given a subset $M \subseteq N$ of d hyperplanes and one additional hyperplane $S \in N \setminus M$, determine which side of S contains the intersection formed by the hyperplanes in M . Since the perturbation ensures that the arrangement formed by N is simple, we can assume that this intersection exists and that it does not lie on S . The above operation is equivalent to evaluating the sign of a determinant formed by the coefficients of the $d + 1$ hyperplanes in $M \cup \{S\}$, for a sufficiently small $\epsilon > 0$. Give a fast method for evaluating this sign without evaluating the entire determinant.

2.8 Early deterministic algorithms

The algorithms that were developed in the early phase of computational geometry were mostly deterministic. Randomization entered the field only later in the 80s. In this section, we shall present some of these early deterministic algorithms. Their study will provide the reader an historical perspective and familiarity with some simple deterministic paradigms, which every student of computational geometry must know. It will also provide the reader an opportunity to study the relationship between the deterministic algorithms given here and their randomized counterparts given later. This relationship is quite akin to the relationship between quick-sort and deterministic sorting algorithms: Randomization yields simplicity and efficiency at the cost of losing determinism.

Not surprisingly, the early algorithms in computational geometry dealt mostly with two-dimensional problems. We shall do the same here. The efficient deterministic algorithms for the higher-dimensional versions of these problems are considerably more complex than the comparable randomized algorithms given later in the book. Some of them are based on the deterministic simulation of their randomized counterparts (Chapter 10). The others are beyond the scope of this book. (Arrangement of lines is an exception. There is a simple deterministic algorithm for constructing higher-dimensional arrangements of hyperplanes. We shall study it in Chapter 6.)

For the sake of simplicity, we shall confine ourselves to the static setting in this section. Only Sections 2.8.1 and 2.8.2 are going to be needed later in this book. So if you wish, you can skip the remaining subsections. The algorithms for trapezoidal decompositions (Section 2.8.3) and planar Voronoi diagrams (Section 2.8.4) given here are somewhat more involved than their randomized counterparts, which will be described in Chapter 3.

2.8.1 Planar convex hulls

Perhaps the simplest two-dimensional problem in computational geometry is the construction of a planar convex hull. Given a set N of n points in the plane, the problem is to compute its convex hull (Section 2.4.1). The boundary of the convex hull of N can be trivially decomposed at its leftmost and rightmost vertices into *upper* and *lower* chains (Figure 2.21). By the leftmost and rightmost vertices, we mean the vertices with the lowest and the highest x -coordinate. We shall confine ourselves to the construction of the upper chain. The lower chain can be constructed similarly.

To begin with, we sort the points in N by their x -coordinates. This can be done in $O(n \log n)$ time using any sorting algorithm in the literature. Or it can be done in expected $O(n \log n)$ time using quick-sort (Chapter 1). Once this sorting is done, we shall show that the upper chain can be computed in additional $O(n)$ time.

The idea is to add the points in N , one at a time, in the increasing order of their x -coordinates. For each $i \geq 1$, let N^i denote the first i points in this order. Let $U(N^i)$ denote the upper chain formed by N^i . Let us say that we have constructed the upper chain $U(N^i)$, the basis case $i = 1$ being trivial. We assume that the upper chain is stored in an obvious fashion by linking its vertices in the increasing order of their x -coordinates.

Now consider the addition of the $(i + 1)$ th point $s = s_{i+1}$. Let r denote the point of $U(N^i)$ with the highest x -coordinate (Figure 2.22(a)). We travel on the boundary of $U(N^i)$ in the left direction, starting at r . We stop as soon as we reach the vertex q , whose left neighbor p lies below the line through s and q (Figure 2.22(a)). If we are lucky, q can coincide with r . It is clear that the upper chain $U(N^{i+1})$ is obtained by joining q to s , and discarding all points of $U(N^i)$ to the right of q (Figure 2.22(b)).

Analysis. Ignoring a constant factor, the cost of each addition is 1 plus the number of points that are discarded. The total number of points that are discarded over all additions is trivially less than n , because a point can be discarded only once. Hence, the algorithm takes $O(n)$ time. This is assuming

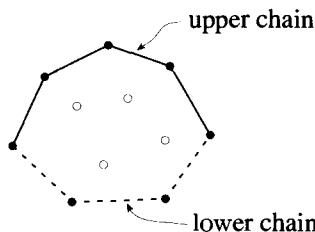
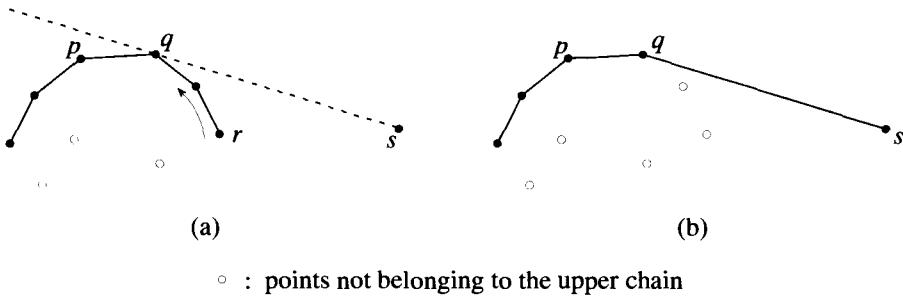


Figure 2.21: Upper and lower chains.

Figure 2.22: Addition of a new point. (a) $U(N^i)$. (b) $U(N^{i+1})$.

that the points in N are already sorted by their x -coordinates. Otherwise, we incur an additional $O(n \log n)$ cost in sorting them. This proves:

Theorem 2.8.1 *The convex hull of n points in the plane can be computed in $O(n \log n)$ time. If the points are given sorted by their x -coordinates, this takes $O(n)$ time.*

Exercises

2.8.1 Give an $O(n \log n)$ time algorithm for computing the intersection of n half-spaces in the plane. (Hint: Dualize the algorithm in this section.)

2.8.2 Make the algorithm in this section on-line (semidynamic). It should be possible to update the convex hull during the addition of each new point in $O(\log n)$ time. (Hint: Associate balanced search trees with the upper and lower chains. Show how the point q in Figure 2.22 can now be located in $O(\log n)$ time. Do not forget to update the search trees. You can use deterministic search trees in the literature, or the randomized ones in Chapter 1.)

***2.8.3** Modify the algorithm in Exercise 2.8.2 so that it becomes fully dynamic. The cost of update should be $O(\log^2 n)$. (Hint: Deletion of a point can “expose” some old points that were redundant so far, by making them nonredundant.)

† Can you make the cost of update logarithmic?

2.8.4 (Convex hulls and polygons in the plane)

(a) Let $H(\hat{N})$ be the convex hull of the set of points \hat{N} . Show how to construct a search structure $\tilde{H}(\hat{N})$ so that, given any ray emanating from the origin, its intersection(s) with $H(\hat{N})$ can be determined in logarithmic time. (Hint: Associate search trees with the upper and lower chains of $H(N)$ just as in Exercise 2.8.2.)

(b) Show how the above search structure can also be used to determine whether a given query point is inside or outside $H(\hat{N})$ in logarithmic time.

(c) Show how $H(\hat{N})$ and $\tilde{H}(\hat{N})$ can be updated in logarithmic time during the addition of a new point. (Hint: Very similar to Exercise 2.8.2.)

***(d)** Design a search structure that can be updated in polylogarithmic time even during the deletion of a point. (Hint: Similar to Exercise 2.8.3.)

- (e) Give direct solutions in the dual setting. Let $H(N)$ be the convex polygon formed by a set N of half-spaces in the plane. Show how to associate a search structure $\tilde{H}(N)$ so that given any linear function, its maximum on $H(N)$ can be located in logarithmic time. Show how to update $H(N)$ and this structure in logarithmic time during addition. Give an alternative search structure that can be updated efficiently during deletion.

2.8.5 Show that the convex hull of the union of two convex polygons can be found in time proportional to the total number of their vertices.

2.8.6 Show that any n -vertex convex polygon P can be decomposed into $n - 2$ triangles in $O(n)$ time, without introducing any extra vertices.

More generally, show that the same holds if P is star-shaped. This means that there is a point o in the interior of P such that the segment joining o with any vertex of P lies entirely within P . We assume that the point o is given to us. It is called the *center* of P . (Hint: Use induction on n . If $n > 3$, there are three consecutive vertices a , b , and c on P , such that the total angle spanned around o , as we go from a to b to c , is less than 180 degrees. Introduce the triangle joining a , b , and c . The remaining part of P remains star-shaped.)

In fact, it is possible to triangulate any n -vertex simple polygon in $O(n)$ time. But the proof of this fact is far more difficult. In Chapter 3, we shall give a simple randomized algorithm with almost linear expected running time.

***2.8.7** Suppose P is a simple polygon with n vertices. A polygon is said to be simple if its boundary does not intersect itself. Show that the convex hull of the vertices of P can be computed in $O(n)$ time.

2.8.2 Arrangements of lines

Perhaps the simplest geometric structure in the plane, after a planar convex hull, is an arrangement of lines (Section 2.2). In this section, we shall give an optimal algorithm for its construction. The algorithm in this section is also incremental like the planar convex hull algorithm given before. It adds the lines in the given set, one at a time.

Let N be any set of lines in the plane. Let $H(N)$ denote the arrangement formed by N (Section 2.2). An arrangement formed by a given set N of lines can be built incrementally as follows. We shall add the lines in N one at a time in *any* order. It turns out that an arrangement of lines is special; so any order is good enough. Let N^i denote the set of the first i added lines. Let $H(N^i)$ denote the arrangement formed by them. Addition of the $(i + 1)$ th line $S = S_{i+1}$ to $H(N^i)$ is achieved as follows (Figure 2.23).

Pick any line $L \in N^i$. Let u be the intersection of S with L . We locate u in $H(N^i)$ by simply searching all edges of $H(N^i)$ contained within L . Strictly speaking, it is not necessary to search all edges of L . For example, one can search L linearly beginning at its top edge, proceeding downwards until one visits the edge intersecting S . The cost of this search is obviously dominated

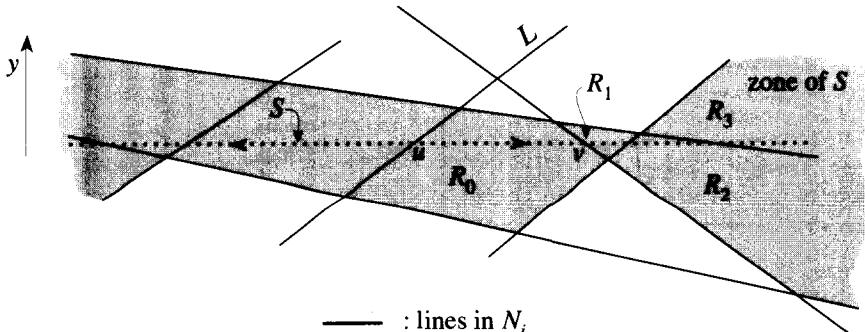


Figure 2.23: Addition of $S = S_{i+1}$ to $H(N^i)$.

by the total number edges on L , which is $O(|N^i|) = O(i)$. After this, we travel along S , starting at u and moving right, splitting the successive faces R_0, R_1, \dots that we encounter along S .

Let us explain what we mean by the splitting of a face. Consider the first face R_0 . We already know the edge containing u on the boundary of R_0 . Using the adjacency information, we can access the node for R_0 in the facial lattice of the arrangement. The adjacency information tells us all edges adjacent to R_0 . We now trivially split R_0 along S in two faces. This requires substituting the node for R_0 in the facial lattice with the nodes for its two split faces. We also need to introduce an additional edge for the intersection of S with R_0 . Similarly, we split the edges of R_0 intersecting S . Concurrently, we also need to update the adjacency information in the facial lattice involving the deleted and the newly created nodes. All this can be trivially done in time proportional to the face-length of R_0 . By the face-length of R_0 , we mean the total number of its vertices. The splitting of R_0 yields the point v (if any), other than u , where S intersects its boundary (Figure 2.23). The next face R_1 that we enter is adjacent to the edge containing v . Hence, we can access R_1 using the facial lattice in constant time. We now split R_1 similarly, and then R_2 , and so on.

We repeat the same procedure along S on the left side of u as well.

Analysis. We have already observed that the initial search along L costs $O(i)$. The cost of traveling and splitting along $S = S_{i+1}$ is clearly proportional to the total face-length of all faces in $H(N^i)$ intersecting S . The set of faces of $H(N^i)$ intersecting S , together with the edges and vertices adjacent to them, is called the *zone of S* in $H(N^i)$. It follows from the following zone theorem that the cost of adding S to $H(N^i)$ is $O(i)$.

Theorem 2.8.2 (Zone Theorem) Let M be any set of m lines in the plane. For any line $S \notin M$, the size of the zone of S in $H(M)$ is $O(m)$. In other words, the total face-length of all faces in $H(M)$ intersecting S is $O(m)$.

Proof. We can assume that S is horizontal, changing the coordinate system if necessary. For any face f in the zone, let us call the edges of f incident to its top or bottom vertex *trivial*. By the top vertex, we mean the vertex of f with the maximum y -coordinate. The bottom vertex is similarly defined. The total number of trivial edges in the zone of S is $O(m)$. Hence, it suffices to count only the nontrivial zone edges. We shall show that every line $S_k \in M$ can contribute only $O(1)$ nontrivial edges. This will prove the theorem. It obviously suffices to count the number of nontrivial edges on S_k above S ; the nontrivial edges below S can be counted similarly. We can also restrict ourselves to the nontrivial zone edges on S_k that are adjacent to the zone from the lower side of S_k ; the ones that are adjacent to the zone from the upper side of S_k can be handled similarly. Figure 2.24 shows one such edge e . We shall show that S_k can contain at most one such edge.

Suppose this were not the case. Let e be the closest such nontrivial edge to the intersection w between S_k and S (Figure 2.24). Let S_j , $j \neq k$, be the line adjacent to the upper end point of e . Let $\text{cone}(S_k, S_j)$ denote the cone obtained by intersecting the upper half-space bounded by S_j and the lower half-space bounded by S_k . Because e is nontrivial, it follows that $\text{cone}(S_k, S_j)$ lies completely above S . Hence, its interior is disjoint from the zone of S . It follows that there can be no nontrivial edge of the same kind on S_k that is farther than e from w . \square

Turning back to the incremental algorithm, it follows that the addition of the $(i+1)$ th line to $H(N^i)$ can be accomplished in $O(i)$ time. Thus, the cost of building $H(N)$ incrementally is $O(\sum i) = O(n^2)$. This proves:

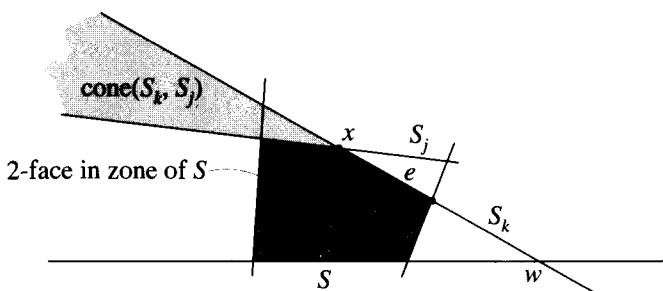


Figure 2.24: A nontrivial edge on the lower side of S_k .

Theorem 2.8.3 *The arrangement of n lines in the plane can be built in $O(n^2)$ time.*

The running time bound in this theorem is clearly optimal, because the size of the arrangement is $\Omega(n^2)$ if the lines are in general position.

Exercises

2.8.8 Modify the previous algorithm so that it works when the arrangement is degenerate. Prove that the running time remains $O(n^2)$.

2.8.9 Let $H(N)$ be any arrangement of lines in the plane. Prove that

$$\sum_f (\text{face-length}(f))^2 = O(n^2),$$

where f ranges over all faces in $H(N)$. (Hint: Show that the sum can be bounded in terms of the total zone size of all lines in N .)

***2.8.10** Let $H(N)$ be as in Exercise 2.8.9. Let L_1 and L_2 be any two parallel lines not in N . Without loss of generality, they can be assumed to be horizontal. Let $\text{slab}(L_1, L_2)$ denote the region in the plane between L_1 and L_2 . Let $G(N) = H(N) \cap \text{slab}(L_1, L_2)$. When L_1 and L_2 lie at infinity, $G(N) = H(N)$. Generalize the result in Exercise 2.8.9 as follows. Show that $\sum_f \text{face-length}^2(f) = O(n + k)$, where f ranges over all faces in $G(N)$ and k denotes the size of $G(N)$.

Show that the same result holds when $G(N) = H(N) \cap C$, where C is any convex polygon with a bounded number of sides.

†Can you show the same if C is a convex region bounded by a constant number of algebraic curves of bounded degree?

2.8.11 Let $H(N)$ be as in Exercise 2.8.9. Let P be any set of p points in the plane. Let $k(N, P) = \sum_f \text{face-length}(f)$, where f ranges over all faces in $H(N)$ containing at least one point in P . Show that $k(N, P) = O(p^{1/2}n)$. (Hint: By the Cauchy-Schwarz inequality, $k(N, P) \leq p^{1/2}(\sum_f \text{face-length}^2(f))^{1/2}$. Apply Exercise 2.8.9.)

Is the bound tight?

2.8.3 Trapezoidal decompositions

In this section, we shall give a deterministic algorithm for constructing the trapezoidal decomposition formed by a set of segments in the plane (Section 2.3). In particular, this also computes all intersections among the given segments. The running time of this algorithm is $O((k + n) \log n)$, where k is the number of intersections. The algorithm is based on a certain technique called *scanning*, which has wide applicability in computational geometry. (The algorithm in this section is also called a sweep-line or a plane-sweep algorithm.)

Let N be the given set of n segments in the plane (Figure 2.25). For the sake of simplicity, assume that the segments in N are bounded. We shall

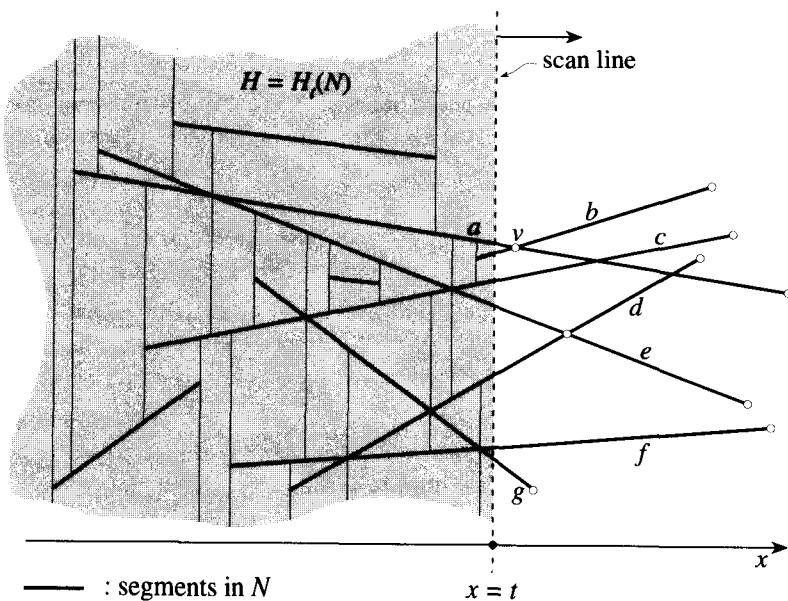


Figure 2.25: Scanning.

also assume, as usual, that they are in general position. In particular, this means that no two segments in N share an endpoint. All these assumptions are easy to remove, and this will be done in the exercises. Let $H(N)$ denote the trapezoidal decomposition formed by N (Section 2.3). Our goal is to compute $H(N)$.

The basic idea is to scan the segments in N by an imaginary vertical line moving in the positive x -direction (Figure 2.25). The x -coordinate will play the role of “time” during this scanning. Initially, we imagine that the scan line is at $x = -\infty$. At time t , the scan line will be at the coordinate $x = t$. Let us denote the intersection of $H(N)$ with the half-space $x \leq t$ by $H_t(N)$. The basic idea is to maintain the restricted partition $H_t(N)$ throughout the scanning process. The initial partition $H_{-\infty}(N)$ is empty. The final partition $H_\infty(N)$ is the partition $H(N)$ that we seek.

Of course, we cannot really increase the time continuously in the algorithm. So we shall somehow ensure that the time takes only discrete values in the algorithm. These values will coincide with the x -coordinates of the endpoints or the intersections of the segments in N . These time values will be called *critical*. An important observation is that the combinatorial structure of $H_t(N)$ changes only at the critical time values. Hence, the computer representation of $H_t(N)$ will need to be changed only at the critical values. One has to be a bit careful though about how $H_t(N)$ is stored in computer

memory. We store the adjacency relationships in $H_t(N)$ as usual. The only thing to remember is that we cannot store the coordinates of the junctions that lie on the right boundary of $H_t(N)$. This is because these coordinates change continuously during the scan. In any case, there is no need to do such a thing. All that we need is an implicit understanding that these junctions lie on the scan line. The adjacency information takes care of the rest.

In the algorithm, the time, i.e., the clock, will advance from one critical value to the next. To accomplish this, we need to be able to predict, at any given time, the next critical value that we will encounter. In other words, we need to be able to predict at any given time the next point—an endpoint or a point of intersection—that the scan line will encounter.

How do we do that? If we knew all endpoints and intersections to the right of the scan line, this will be no problem. Endpoints do not pose a problem, because we know all of them right in the beginning. On the other hand, we certainly cannot pretend to know all intersections to the right of the scan line. In fact, computing the intersections is one of our goals! Fortunately, we do not need to know all intersections to be able to predict the next critical value. The reason is quite simple. Consider a moment when the scan line is about to pass through an intersection; call it v . Let a and b denote the two segments in N containing v (Figure 2.25). Observe that the span on the scan line between a and b is about to shrink to a point at this moment. We can reformulate this observation as follows. At any given time t , let $L = L_t$ denote the list of the segments in N intersecting the scan line. It will be called the *frontier list* or simply the *frontier*. The reason behind this terminology is that the scan line can be thought of as the frontier, i.e., the right boundary of the partition $H_t(N)$. In Figure 2.25, the frontier list is $\{a, b, c, d, e, f, g\}$. At any given time, the segments in the frontier list can be linearly ordered in the obvious vertical fashion. This order corresponds to the vertical order among the intersections of the segments in L with the scan line. The previous observation now leads to the following:

Observation 2.8.4 When the scan line is about to pass an intersection $v = a \cap b$, the segments a and b must be consecutive in the linear order on the frontier list.

With this observation in mind, we shall maintain the frontier throughout the scanning process. The segments in the frontier will be kept vertically ordered. For predicting the next critical value, we shall also need one additional list called an *event schedule*. Initially, this event schedule will contain all endpoints of the segments in N , ordered according to their x -coordinates. This corresponds to the situation when the scan line is at $t = -\infty$. At any given time, the event schedule will contain all endpoints to the right of the scan line; these are shown as bubbles in Figure 2.25. The event schedule

will also contain each intersection on the right side of the scan line that is contained in two consecutive segments of the frontier. Such intersections are shown as bubbles in Figure 2.25. The event schedule might contain at this time some additional points of intersection to the right of the scan line. But this fact is irrelevant as far as the prediction of the next critical value is concerned. Observation 2.8.4 implies that, at any given time, the next point to be encountered by the scan line is the one in the event schedule with the least x -coordinate.

We are now ready to give the algorithm for constructing $H(N)$. Throughout the algorithm we shall maintain:

1. A partially constructed partition $H_t(N)$, which we shall denote by H .
2. The vertically ordered frontier list of segments intersecting the current (imaginary) scan line.
3. The event schedule. The points in the event schedule are kept linearly ordered by their x -coordinates. With each point in the event schedule, we maintain pointers to the segment(s) in N containing it. If the point is an endpoint, there is one such segment. If the point is an intersection, there are two such segments.

We assume that one can add, delete, or search a segment in the frontier list in logarithmic time. This can be achieved by maintaining a balanced search tree for its linear order. One can use for this purpose a randomized search tree in Chapter 1 or any deterministic search tree in the literature. Similarly, we assume that one can add, delete, or search a point in the event schedule in logarithmic time.

Remark. The elements in the frontier list are not points, as was the case in Chapter 1. But, as we remarked in the beginning of Chapter 1, all that we need for the maintenance of a search tree is that any two elements in the linear order be comparable in constant time. That is certainly the case here: Given the current scan line, and any two segments intersecting it, we can decide in constant time which segment intersects the scan line above the other.

Here is a pseudo-code of the final algorithm for constructing $H(N)$.

Algorithm 2.8.5

Initially, the partition H and the frontier list are empty. The event schedule contains all endpoints of the segments in N , ordered according to their x -coordinates. (This initial state corresponds to the time $t = -\infty$.)

While the event schedule is nonempty, remove the point in it with the least x -coordinate. Denote this point by p . Imagine moving the scan line just past the point p . This amounts to doing the following (Figure 2.26).

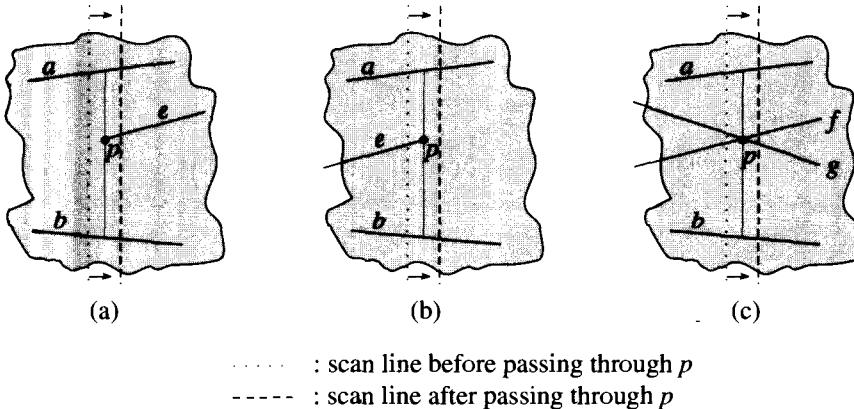


Figure 2.26: Scanning (a) a left endpoint, (b) a right endpoint, and (c) an intersection.

Main body:

1. Suppose p is the left endpoint of a segment $e \in N$: In this case, we add e to the frontier list; this includes adding e to the search structure associated with this list. Let a and b denote the segments in this list before and after e . We can assume that a and b always exist, by adding in the beginning of the algorithm two horizontal dummy segments that cover all segments in N from above and below. We update H at its right boundary by adding the segment e and the vertical attachment through p spanning between a and b . Now a and e become consecutive in the frontier list. If they intersect on the right side of p , we add their intersection to the event schedule, assuming that it is not present there already. We do the same for b and e .
2. Suppose p is the right endpoint of a segment $e \in N$: In this case, we delete e from the frontier list. Let a and b denote the segments in this list before and after e . We update H at its right boundary by adding the vertical attachment through p spanning between a and b . Now a and b become consecutive in the frontier list. If they intersect on the right side of p , we add their intersection to the event schedule, assuming that it is not present there already.
3. Suppose p is the intersection of two segments $f, g \in N$. In this case, we switch the orders of f and g in the frontier list. For the sake of concreteness, assume that f is above g after the switch. Let a and b denote the segments in the frontier list before and after f and g , respectively. We update the partition H at its right boundary so as to reflect that f and g intersect at p . We also add the vertical attachment

through p spanning between a and b . Now a and f become consecutive in the frontier list. If they intersect on the right side of p , we add their intersection to the event schedule, assuming that it is not present there already. We do the same for b and g .

When the event schedule becomes empty, the situation corresponds to the time $t = \infty$. The partition H at this time is clearly $H(N)$. This finishes the description of the algorithm.

Analysis. The construction of the initial event schedule requires sorting all endpoints by their x -coordinates and constructing a search tree for this sorted order. This takes $O(n \log n)$ time. The main body of the algorithm is executed once at each critical value. It consists in $O(1)$ updates and searches in the frontier list and the event schedule. This takes $O(\log n)$ time. The update of the partition H takes additional $O(1)$ time. The total number of critical values is $(k + 2n)$, one for each endpoint and point of intersection. Hence, the running time of the algorithm is $O((k + n) \log n)$.

We have thus proved the following.

Theorem 2.8.6 *The trapezoidal decomposition formed by n segments in the plane can be computed in $O((k + n) \log n)$ time, where k is the number of intersections. In particular, all intersections of the given n segments can be computed in $O((k + n) \log n)$ time.*

The bound achieved in this theorem is not the best possible. In Chapter 3, we shall give a much simpler randomized algorithm with expected $O(k + n \log n)$ running time. There also exists a deterministic algorithm with $O(k + n \log n)$ running time. But it is considerably involved and is beyond the scope of this book.

It follows from the previous theorem that, when the segments in N are nonintersecting, the resulting trapezoidal decomposition can be computed in $O(n \log n)$ time. This special case is very important because it arises when the segments in N form a planar graph. Strictly speaking, the preceding algorithm does not allow the segments in N to share endpoints. But this only requires easy modifications, which we shall leave to the reader.

Theorem 2.8.7 *Suppose G is a planar graph, whose planar embedding is given to us. Assume that all edges of G are line segments. Then the trapezoidal decomposition of G can be computed in $O(n \log n)$ time, where n is the size of G . (Because of Euler's relation (Exercise 2.3.4), n can also be taken as the number of vertices in G .)*

Exercises

2.8.12 Modify the algorithm in this section so as to allow unbounded segments. (Hint: Restrict within a large enough box.) Also allow sharing of endpoints among

the segments. Show that the bound in Theorem 2.8.6 continues to hold, where n now denotes the number of edges plus the number of vertices. Deduce Theorem 2.8.7.

***2.8.13** Use the sweep paradigm in this section to compute the intersection of two 3-dimensional convex polytopes. The running time of the algorithm should be $O((n + m) \log(n + m))$, where n and m denote the sizes of the two polytopes.

2.8.14 Suppose you want to decompose the regions of a given planar graph into triangles instead of trapezoids. One way of doing this is the following: First, we compute a trapezoidal decomposition as in Theorem 2.8.7. Then we decompose each trapezoid in this decomposition into triangles. However, the corners of the triangles in the resulting decomposition need not coincide with the vertices of the original planar graph. Suppose you wanted a triangulation of a planar graph, which does not introduce any extra vertices (Figure 2.9). Modify the sweep-line algorithm in this section so that it computes such a triangulation. The running time of the algorithm should remain $O(n \log n)$.

2.8.15 Suppose P is a simple polygon with n vertices. A polygon is said to be simple if its boundary does not intersect itself. Assume that we are given a triangulation of the interior of P . Suppose we are given a point p in the interior of P . A point q inside P is said to be visible from p if the segment joining p and q is completely contained within P . The set of visible points within P form a polygon, called the *visibility polygon* of p . Show how the visibility polygon of any point p within P can be computed in $O(n)$ time, where n is the size of P . (Hint: Consider the graph that contains a node for every triangle in P . Two nodes are connected by an edge iff the corresponding triangles share an edge. Show that this graph is a tree, by induction on the number of triangles. Traverse this tree starting at the node for the triangle containing p . Compute the visibility polygon concurrently.)

2.8.4 Planar Voronoi diagrams

The construction of planar Voronoi diagrams was one of the earliest problems studied in computational geometry because of its numerous applications. In this section, we shall give an $O(n \log n)$ time algorithm for constructing the Voronoi diagram of n sites in the plane. The algorithm is based on a deterministic divide-and-conquer principle, which has numerous other applications in computational geometry.

Let N be any set of n sites in the plane in general position. Let $H(N)$ denote its Voronoi diagram (Section 2.5). Our goal is to compute $H(N)$. To begin with, we shall assume that the sites in N are sorted by their x -coordinates. This can be done in $O(n \log n)$ time by any deterministic sorting algorithm in the literature. Or one can use quick-sort (Chapter 1).

The basic outline of our algorithm is as follows. We divide the sorted list N in two halves L and R of size $n/2$. We construct the Voronoi diagrams of L and R recursively. Then we “merge” these Voronoi diagrams in $O(n)$ time so

as to obtain the Voronoi diagram for N . The total cost of this construction satisfies the following recurrence relation:

$$T(n) = O(n) + T(n/2), \text{ and } T(1) = O(1).$$

The solution of this recurrence is $T(n) = O(n \log n)$. Thus, we shall get an $O(n \log n)$ time algorithm for constructing a planar Voronoi diagram, once we solve the following problem.

Merging problem: We are given two disjoint sets L and R of sites in the plane. We assume that these sets are linearly separated. This means that every site in L has a smaller x -coordinate than every site in R . Given the Voronoi diagrams of L and R , compute the Voronoi diagram of $N = L \cup R$ in $O(n)$ time. Here n is the size of N .

We now proceed to solve this problem. Without loss of generality, we can assume that we also have the convex hulls of L and R . This is because the convex hull of L is obtained by simply linking in a circular fashion its sites labeling the unbounded Voronoi regions in $H(L)$ (Section 2.5, Figure 2.19). And the same holds for R , too.

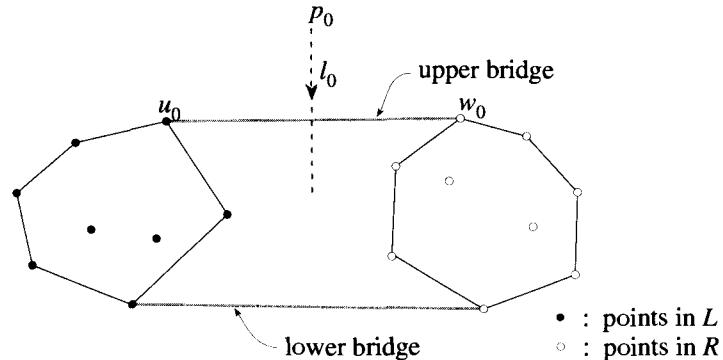


Figure 2.27: Upper and lower bridges.

As a warm-up exercise, we will show that the convex hull of N can be computed in linear time. Clearly, it suffices to compute its upper and lower chains (Figure 2.21). We shall only compute the upper chain, because the lower chain can be computed in a similar fashion. For the purpose of computing the upper chain, we need concern ourselves only with the sites that lie on the upper chains of L and R —the remaining sites cannot occur in the upper chain of N (Figure 2.27). Now observe the following: If we concatenate the upper chains of L and R , we get a list whose points are already sorted by their x -coordinates. Hence, the upper chain of these points can

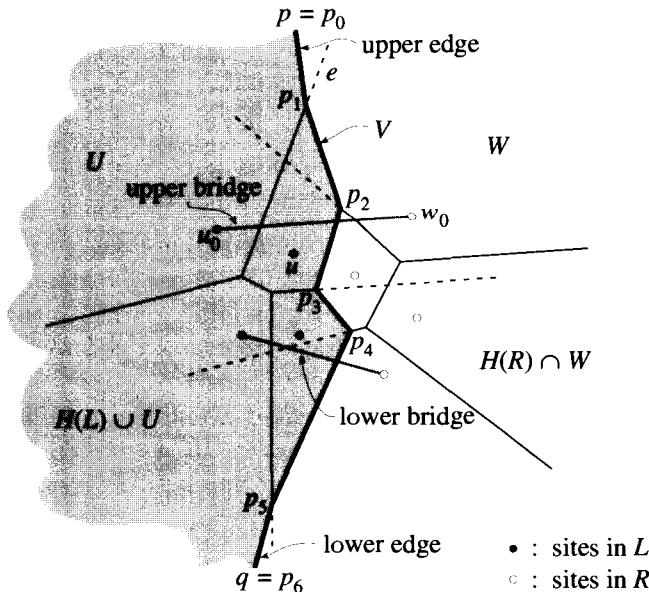


Figure 2.28: Merging of Voronoi diagrams.

be computed in $O(n)$ time (Section 2.8.1). What results is the upper chain of N . It contains exactly one new edge, which neither belongs to the upper chain of L nor to the upper chain of R . This edge will be called the *upper bridge* (Figure 2.27). One can compute the *lower bridge* similarly. As we shall soon see, these bridges can be used to make inroads into our problem literally.

Here is a snap-shot of the rest of our plan. We shall suitably partition the whole plane into two open regions U and W sharing a common boundary V . It will then turn out that $H(N)$ results by simply gluing the intersections $H(L) \cap U$ and $H(R) \cap W$ along V (Figure 2.28). The partition that we need is defined as follows. Let p denote any point in the plane. Define $d_{\min}(p, L)$ to be the minimum distance between p and any site in L . We define $d_{\min}(p, R)$ similarly. Then the whole plane can be partitioned into three disjoint sets mentioned above (Figure 2.28):

1. $U = \{p \mid d_{\min}(p, L) < d_{\min}(p, R)\}$. This is an open region containing L . If $p \in U$, then the nearest neighbor(s) of p in L must coincide with its nearest neighbor(s) in N . Hence, $H(N) \cap U = H(L) \cap U$.
2. $W = \{p \mid d_{\min}(p, L) > d_{\min}(p, R)\}$. This is an open region containing R . If $p \in W$, then the nearest neighbor(s) of p in R must coincide with its nearest neighbor(s) in N . Hence, $H(N) \cap W = H(R) \cap W$.

3. $V = \{p \mid d_{\min}(p, L) = d_{\min}(p, R)\}$. This is a common boundary of U and W . If $p \in V$, the nearest neighbor of p in L and the one in R are equidistant from p ; the converse also holds.

It thus follows that $H(N)$ results by gluing $H(L) \cap U$ and $H(R) \cap W$ along V (Figure 2.28). With this in mind, our next goal is to compute V . But before we do that, we must analyze its structure carefully.

Proposition 2.8.8 (1) V is a union of some Voronoi edges in $H(N)$. An edge in $H(N)$ belongs to V iff it is adjacent to a Voronoi region labeled with a site in L on its left side and a Voronoi region labeled with a site in R on its right side. Here the left and the right side of an edge are defined by orienting the edge upwards.

(2) V is a single, monotone chain of edges extending to infinity in both directions. Monotonicity means the following: If we direct each edge of V upwards, then the entire V gets oriented upwards; in other words, the arrowheads of no two adjacent edges in V meet.

Proof. Intuitively, the proposition should be clear because $H(N)$ results by gluing $H(L) \cap U$ and $H(R) \cap W$ along V and, moreover, L lies to the left of R . A rigorous justification is as follows.

Let p be any point in the plane. Let l and r be the nearest sites to p in L and R , respectively. If $p \in V$, then l and r are equidistant from p , and all other sites are at larger distances. Hence, p must lie on a Voronoi edge of $H(N)$, and this edge must be adjacent to the Voronoi regions in $H(N)$ labeled by l and r . By the Voronoi property, this edge—call it e —lies on the perpendicular bisector of the segment $[l, r]$. The site l has a smaller x -coordinate than r , because L and R are linearly separated. Hence, the Voronoi region labeled by $l \in L$ must lie on the left side of e , and the one labeled by $r \in R$ must lie on its right side. Conversely, if p lies on such an edge, then, by the Voronoi property, $l \in L$ and $r \in R$ are its nearest neighbors, and they are equidistant from p . Hence, $p \in V$. This proves (1).

We already know that V is the common boundary of the disjoint open regions U and W . By (1), each edge of V is adjacent to the region U on the left side and to the region W on the right side. Hence each vertex in

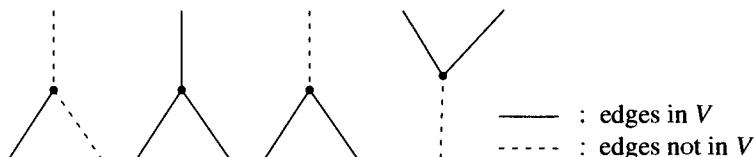


Figure 2.29: Various kinds of vertices forbidden in V (excluding some symmetric cases).

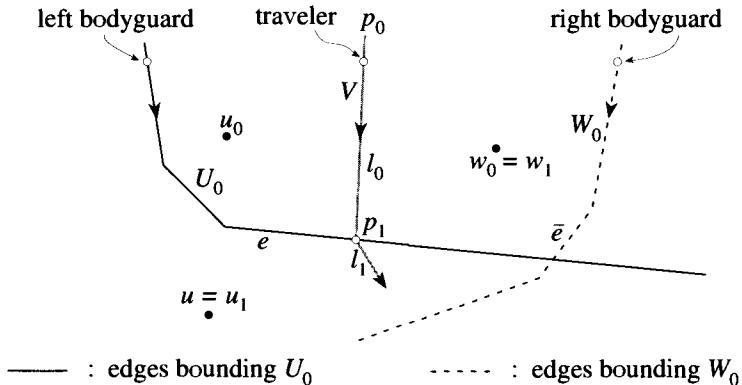
V must be adjacent to exactly two edges in V , and, in addition, V cannot loop down or up at this vertex; in other words, the possibilities shown in Figure 2.29 cannot arise. This implies that each connected component of V must be a monotone chain extending to infinity in both directions. In fact, V can contain a single such chain. This again follows because each edge of V is adjacent to the region U on the left side and to the region V on the right side. \square

In what follows, the topmost, unbounded edge of V will be called its *upper edge* and the lowermost, unbounded edge will be called its *lower edge* (Figure 2.28). For the sake of exposition, it will also be convenient to assume that V possesses endpoints at the upper and lower infinities. These will be denoted by p and q , respectively. This is just imaginary, of course. As far as the computer representation is concerned, p and q are implicitly determined by the lines containing the upper and lower edges.

The characterization of V given in the preceding proposition suggests a simple strategy for its construction. Let $p = p_0, p_1, \dots, p_{r-1}, p_r = q$ be the enumeration of the vertices of V in the vertical order. We are going to compute the vertices of V in this order (Figure 2.28).

To begin with, it is easy to compute the line containing the upper edge $[p_0, p_1]$. Let us see why. Let us denote the left and the right endpoints of the upper bridge by u_0 and w_0 , respectively (Figures 2.27). Since the upper edge belongs to the convex hull of N , its endpoints u_0 and w_0 must belong to the adjacent, unbounded regions in $H(N)$ (Figures 2.27 and 2.19). By Proposition 2.8.8, the edge shared by these Voronoi regions in $H(N)$ belongs to V . Since it is unbounded above, it must be the upper edge $[p_0, p_1]$. Because of the Voronoi property, the line containing this edge is just the perpendicular bisector of the upper bridge. Since we already know this bridge, the line can be computed easily. Call it l_0 (Figure 2.27). It provides us the beginning that we need.

Let U_0 denote the Voronoi region in $H(L)$ labeled by u_0 . We can access U_0 , because $H(L)$ is given to us. Similarly, let W_0 denote the Voronoi region in $H(R)$ labeled by w_0 . The sites u_0 and w_0 are equidistant nearest neighbors of any point on the upper edge. Hence, p_0 is contained in both U_0 and W_0 . Now let us imagine traveling along l_0 from p_0 to p_1 . Figure 2.30 illustrates a situation that is somewhat more complex than the one that arises in Figure 2.28. As long as we are completely in the interior of U_0 and W_0 , nothing can change: Our nearest neighbor in L continues to be u_0 , and our nearest neighbor in R continues to be w_0 , and both are equidistant from us. When we hit the boundary of either U_0 or W_0 , the situation changes. Let us say that we hit an edge of U_0 first, the other case being symmetric. We claim that the point where we hit is p_1 . Let us see why. Let e denote the edge of

Figure 2.30: Moving from p_0 to p_1 .

U_0 that is hit. Let u denote the site in L labeling the Voronoi region of $H(L)$ on the other side of e (Figures 2.30 and 2.28). At the point of hit, we are equidistant from the three sites u_0, w_0 , and u . Moreover, all the remaining sites in $N = L \cup R$ are at larger distances. Hence, the point of hit must be a Voronoi vertex in $H(N)$. Since it is adjacent to the Voronoi edge we are traveling on, it must be p_1 .

How do we proceed further from p_1 ? Well, the nearest neighbors of p_1 in N are u_0, w_0 , and u at equal distance from p . Since the next edge $[p_1, p_2]$ on V is a Voronoi edge of $H(N)$, the line through this edge must bisect either $[u_0, u]$ or $[u, w_0]$. Proposition 2.8.8 tells us that the latter must be the case. Hence, we are constrained to proceed down the line l_1 that bisects the segment $[u, w_0]$ perpendicularly (Figure 2.30). As we continue our journey on V downwards, our next nearest neighbor in L is going to be u , and not u_0 . So we let $u_1 = u$, and let U_1 be the Voronoi region in $H(L)$ labeled by u_1 . Our nearest neighbor in R continues to be w_0 . So we let $w_1 = w_0$ and let $W_1 = W_0$. Our new direction of travel is now going to be along l_1 .

Now repeat the same procedure: Given u_1 and w_1 , we access their Voronoi regions in $H(L)$ and $H(R)$ and then proceed from p_1 along l_1 until we hit p_2 . And then we proceed to p_3 and so on. We stop when we reach the last point $q = p_r$ on V . This happens when u_r and w_r coincide with the endpoints of the lower bridge, which we already know.

There is one point of detail that we have glossed over so far. When we are traveling from p_0 along l_0 , how do we know whether we are going to hit first the boundary of U_0 or the boundary of W_0 ? For this purpose, we imagine being accompanied by two bodyguards in our travel, one traveling on the border of U_0 on our left side and one traveling on the border of W_0 on our right side (Figure 2.30). Both bodyguards are required to maintain the

same y -coordinate as us during the travel. Of course, the bodyguards in the algorithm cannot walk in a continuous fashion. Instead, they will alternate their walks: First, one guard will travel forward by one edge. Then the second guard will travel forward, possibly by more than one edge until he catches up with the first guard, i.e., until he reaches an edge whose lower endpoint has a smaller y -coordinate than the lower endpoint of the edge containing the first guard. Once the second guard has traveled in this fashion, it is now the first guard's turn to catch up with the second guard. They keep on doing so until one of the guards discovers that he is walking on an edge that is intersected by l_0 . At this point, the other guard should advance, if necessary, to the edge that is intersected by the horizontal line through p_1 .⁵ (In Figure 2.30, there is no need for an advance, because the right bodyguard would already be at the edge \bar{e} .) Ignoring a constant factor, the total cost of these walks is clearly proportional to the total number of edges traversed by the bodyguards.

A similar procedure is used when we travel from p_1 to p_2 , from p_2 to p_3 , and so on. There is just one more thing that we need to be careful about. We need to make sure that a bodyguard does not visit the same edge more than once. Refer again to Figure 2.30. Let us examine more carefully what the bodyguards should do when we proceed from p_1 along l_1 . Since $U_1 \neq U_0$, the left bodyguard should carry out a preliminary survey of the Voronoi region $U_1 \in H(L)$ labeled by $u_1 = u$, and then he should position himself on its edge that intersects the (imaginary) horizontal line through p_1 . Since $W_1 = W_0$, the right bodyguard, on the other hand, should simply resume his travel downwards from the edge \bar{e} that he is currently at. In other words, he does not need to backtrack.

Analysis. If we ignore the preliminary surveys, our procedure ensures that no edge of $H(L)$ or $H(R)$ is visited more than once by a bodyguard. The preliminary survey is easy to take into account. It can happen for a Voronoi region in $H(L)$ or $H(R)$ only once, the first time it is visited (if at all). It follows that the total travel cost of the left bodyguard is bounded by the size of the Voronoi diagram $H(L)$. This is proportional to the size of L (Section 2.5). The total travel cost of the right bodyguard is similarly proportional to the size of R . Hence, the whole construction of V takes $O(n)$ time.

Final gluing. In the preceding construction of V , we also discover all Voronoi regions of $H(L)$ that are intersected by V . These regions are split, and only their parts to the left of V are retained. The remaining Voronoi regions in $H(L)$ are unaffected. What results is $H(L) \cap U$. We compute $H(R) \cap W$ similarly. Now, $H(N)$ is obtained by joining these two restricted Voronoi diagrams along V . The cost of this whole procedure is clearly dom-

⁵Strictly speaking, this step is not necessary.

inated by the total of size of $H(L)$, $H(R)$, and $H(N)$. This is $O(n)$, because the size of a Voronoi diagram is linear in the number of sites.

It follows that $H(L)$ and $H(R)$ can be merged to form $H(N)$ in $O(n)$ time. This fulfills the promise we made in the beginning of this section.

We have thus proved the following.

Theorem 2.8.9 *The Voronoi diagram of n sites in the plane can be constructed in $O(n \log n)$ time.*

Exercise

***2.8.16** Apply the divide-and-conquer paradigm in this section to the construction of three-dimensional convex hulls. Show that the convex hull of n sites in R^3 can be constructed in $O(n \log n)$ time. (We shall give a simpler randomized algorithm for this problem in Chapter 3.)

2.8.5 Planar point location

Another problem that has attracted a lot of attention in computational geometry since its earliest days is the planar point location problem. The problem is the following. We are given a planar graph G . We assume that we are given its explicit embedding in the plane, and we also assume that all of its edges are straight line segments. The goal is to build a point location structure so that, given any query point p , the region in G containing p can be located quickly—say, in logarithmic time. The planar graph G can arise in several ways. For example, it can stand for the trapezoidal decomposition that we built in Section 2.8.3 or it can stand for a planar Voronoi diagram. In the case of Voronoi diagrams, the point location problem is especially important. This is because the nearest neighbor queries can be reduced to point location queries over Voronoi diagrams (Section 2.5).

So let G be any n -vertex planar graph. For the sake of simplicity, we assume that G is bounded. This assumption can be easily removed; see the exercises below. We also assume that the outer boundary of G is a triangle. If not, we just add a large enough triangle. The unbounded region outside this triangle will be ignored in what follows. Finally, let us also assume that G is a triangulation. This means every bounded region of G is a triangle. This restriction can be easily enforced as follows. First, we decompose the regions of G into trapezoids, applying the algorithm in Section 2.8.3 to the edges of G . Next, we divide each trapezoid into triangles trivially.⁶ This

⁶If the trapezoid has m junctions on its border, it can be divided into $m - 2$ triangles. Note that m can be greater than 4, because there can be several junctions where the vertical attachments meet the trapezoid from its outside. It is also possible to triangulate G without introducing any new vertices (Exercise 2.8.14), but we do not need this property here.

yields a triangulation whose size is of the same order as that of G . The cost of this preprocessing is $O(n \log n)$.

So assume that G is an n -vertex planar triangulation. Our algorithm in this section is going to depend crucially on the following simple property of G . Let us call a set of vertices in G *independent*, if no two of these vertices are adjacent in G (Figure 2.31(a)). The vertices in G , other than the corners of the outer triangle, are said to be in the interior of G .

Proposition 2.8.10 *Fix any integer constant $c > 6$. Define a constant $d = (1/c)(1 - 6/c) > 0$. One can find in $O(n)$ time at least $dn - 1$ independent vertices of degree less than c in the interior of G .*

Proof. Let e denote the number of edges in G . By the specialization Euler's relation for planar triangulations (Exercise 2.3.4), we conclude that $e = 3n - 6$. Since each edge of G is adjacent to two vertices, this implies that the total degree of all vertices in G is $2e < 6n$. So G can contain at most $6n/c$ vertices of degree $\geq c$. In other words, the number of vertices of degree less than c in G is at least $(1 - 6/c)n$. At most three of them are not in the interior of G .

Now we proceed to choose a set of independent vertices of degree less than c in the interior of G . Choose any interior vertex of degree less than c , and mark its neighbors. Repeat: Again, choose any unmarked and unchosen vertex of degree less than c in the interior of G , and mark its neighbors. We proceed in this manner until we cannot continue any further. It is clear that the vertices chosen in this process are all independent. What is their number? Well, in each step, we choose one vertex and mark at most $c - 1$ vertices. Thus, each step eliminates at most c vertices from further consideration. The number of interior vertices of degree less than c is at least $(1 - 6/c)n - 3$. Hence, the number of steps, i.e., the number of chosen vertices, is at least $(1/c)[n(1 - 6/c) - 3] > dn - 1$. \blacksquare

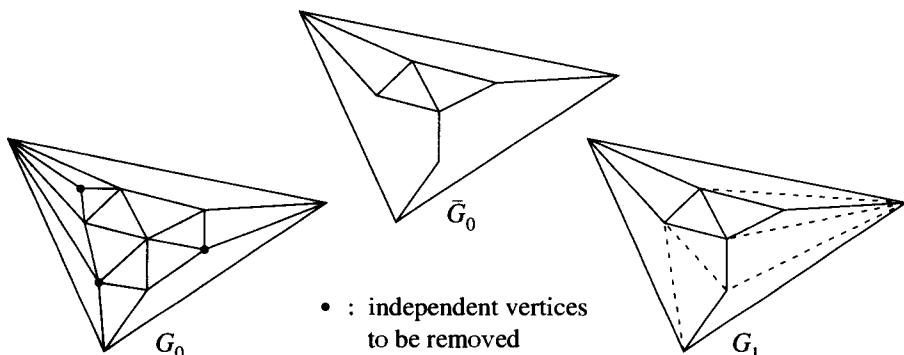


Figure 2.31: (1) $G = G_0$. (2) \bar{G}_0 . (3) G_1 .

A point location structure for G can be built as follows (Figure 2.31). Fix a constant $c > 6$, say, $c = 12$. Let $G_0 = G$. Proposition 2.8.10 provides us at least $n/24 - 1$ independent vertices of degree less than 12 in the interior of G_0 . We remove these vertices and the edges adjacent to them. Call the resulting graph \bar{G}_0 . The graph \bar{G}_0 need not be a triangulation, but it has a simple structure. Since the removed vertices are independent, each region in \bar{G}_0 is a star-shaped m -gon, $3 \leq m \leq 11$, with a removed vertex acting as its center. It can be trivially triangulated using $m - 3$ edges, and without introducing any new vertices (Exercise 2.8.6). When each polygon in \bar{G}_0 is triangulated in this fashion, we get a new triangulation G_1 , with a smaller number of vertices. We shall store G_1 in our search structure. With each triangle $\Delta \in G_1$, we shall also store pointers to its *children*. By this, we mean the triangles in G_0 intersecting Δ . These children must be contained in the same polygon in \bar{G}_0 as Δ . Hence, each Δ has at most 11 children, and they can be determined in $O(1)$ time.

Now we repeat the same process so as to get a sequence of triangulations $G = G_0, G_1, G_2, \dots$, until we end up with a triangulation G_r consisting of a single triangle. Our data structure will be a hierarchy of these triangulations, together with the parent-children pointers between the successive levels. The *root* of this data structure is the single triangle in G_r . This, in fact, must be the outer triangle of G . We shall denote it by Γ .

The point location is carried out as follows. Let p be any query point. If p lies outside the outer triangle Γ , we report so and stop. Otherwise, we begin at the root Γ . We determine the child of Γ in G_{r-1} containing p . We repeat this process until we end up with the triangle in $G = G_0$ containing p .

Analysis. The cost of the above search during point location is clearly proportional to r , the number of levels in the data structure. How large can r be? Let us denote the number of vertices in G_i , for $i \geq 0$, by n_i . By Proposition 2.8.10,

$$n_{i+1} \leq n_i - \left(\frac{n_i}{24} - 1\right) \approx \frac{23}{24}n_i, \text{ for each } i \geq 0.$$

Hence, it follows that $r \approx \log_{24/23} n = O(\log n)$. It also follows that the space requirement of our search structure is proportional to

$$\sum_i n_i \approx n \sum_i (23/24)^i = O(n).$$

The time cost is the same, ignoring the $O(n \log n)$ cost of producing the initial triangulation.

We have thus proved the following.

Theorem 2.8.11 For any n vertex planar graph, one can build a point location structure of $O(n)$ size in $O(n \log n)$ time, guaranteeing $O(\log n)$ query time.

The constant factors within the Big-Oh notation in this theorem are somewhat large. In Chapter 3, we shall give a randomized search structure with smaller constant factors.

If we apply the scheme in this section to planar Voronoi diagrams, we get a search structure for answering nearest neighbor queries. In conjunction with Theorem 2.8.9, we then get the following.

Theorem 2.8.12 Given n sites in the plane, one can build a search structure of $O(n)$ size in $O(n \log n)$ time so that nearest neighbor queries can be answered in $O(\log n)$ time.

Exercises

2.8.17 Modify the algorithm in this section so that it can handle unbounded planar graphs. (Hint: Restrict within a large enough box surrounding all vertices of G .)

2.8.18 In this exercise, we shall give another planar point location structure based on the so-called *persistent* search trees for sorted lists. A persistent search tree performs like an ordinary search tree for sorted lists, in the sense that one can add, delete, or search an item in logarithmic time. But, in addition, it also allows a search in the past. More precisely, let t be any positive integer that is less than or equal to the total number of updates performed so far. Let p be any item. The persistent search tree will allow us to determine in logarithmic time the item before (or after) p in the linear order that existed just prior to the t th update. We are assuming here that p is comparable to the items in this linear order. The space requirement of a persistent search tree is $O(1)$ per update in the amortized sense.

Assume that a persistent search tree is available to you. Use it to design a planar point location structure as follows. First, sort the vertices of G in the increasing order of their x -coordinates. Associate an ordinary search tree T with this sorted list. Now imagine scanning the given graph G just as in Section 2.8.3. The only difference now is that no intersections are encountered this time, but the sharing of endpoints is allowed. Maintain the frontier list just as in that section. But instead of associating an ordinary search tree with the frontier, associate a persistent search tree.

Point location queries are handled as follows. Let p be the query point. Using T , locate the vertex v of G with the smallest x -coordinate on the right side of p . Using the persistent search tree, locate the edge of G that was just above p in the frontier list, when the scan line was about to pass v . This must be the edge of G just above p . The region of G containing p is the one adjacent to the lower side of this edge.

Study the persistent search trees in the literature and fill in the details of this algorithm. Show that one gets a planar point location structure of $O(n)$ size in $O(n \log n)$ time, guaranteeing $O(\log n)$ query time.

2.9 Deterministic vs. randomized algorithms

Section 2.8 demonstrated several simple deterministic methods that are widely useful:

1. Incremental method (Section 2.8.1, Section 2.8.2): Add the objects in the given set, one at a time, in some *predetermined* order.
2. Divide and conquer (Section 2.8.4): Divide the object set in some *predetermined* fashion, and recur.
3. Search through a sequence of partitions computed in some *predetermined* fashion (Section 2.8.5).
4. Scanning (Section 2.8.3).

Three important randomized methods in this book result by just replacing the word *predetermined* with the word *randomized*:

1. Randomized incremental method (Chapter 3): Add the objects in the given set, one at a time, in a *random* order.
2. Randomized divide-and-conquer (Chapter 5): Divide the object set in a *random* fashion, and recur.
3. Search through a sequence of partitions computed in a *random* fashion (Chapters 3 and 5).

This simple substitution of words has dramatic consequences. The deterministic methods that are hard to generalize to higher dimensions become effortlessly generalizable in the randomized setting. Even in two dimensions, this leads to algorithms that are more efficient than their deterministic counterparts, sometimes in the asymptotic sense, sometimes in the sense of improved constant factors.

But there are also deterministic methods that have no randomized counterparts. A prime example is provided by the scanning method discussed before. The trapezoidal decomposition problem, which we solved by scanning in Section 2.8.3, can be handled more efficiently by a randomized incremental method (Chapter 3). But there are some problems which can be solved by scanning of some sort, but which do not seem to have simpler randomized solutions. We shall encounter some such situations later in the book.

2.10 The model of randomness

In the rest of this book, we are going to be occupied with randomized methods for the most part. The deterministic methods studied in the previous section will provide the reader an opportunity for comparison and also an historical perspective. The simplest known efficient algorithms for sorting and

searching linear lists are the randomized algorithms described in Chapter 1. It is no surprise that the simplest known efficient algorithms for several of the higher-dimensional sorting and searching problems stated in this chapter are also randomized.

We shall end this chapter by adding a few words regarding the model of randomness that we are going to be using in this book. Our model is basically the one used in quick-sort (Chapter 1). The algorithms make random choices. But we do not assume that the objects in N are chosen from any random distribution in space. This is just as in the case of quick-sort, where no assumption about the distribution of points is made. Such assumptions put severe restrictions on the validity of analysis for two reasons. First, it is very difficult to characterize the distributions that occur in reality. Second, the distributions differ from one application to another.

Thus, when we refer to an expected performance bound, the expectation is meant to be with respect to the random choices in the algorithm. These choices are kept hidden from the outside world. For example, if a search structure is built in a random fashion, the expected search cost refers to the search cost that the user will expect, only knowing that the choices made in the construction were random. The thing to remember is that we never make any assumptions about how the input objects are distributed in space.

Bibliographic notes

All problems stated in this chapter have been studied in computational and combinatorial geometry for a long time. For each stated problem, the reader should refer to the bibliographic notes for the later chapters, wherein the problem is studied in more detail. In addition, the reader may refer to the earlier books on computational geometry [87, 155, 185] and combinatorial geometry [29, 115] for more historical details.

The connection between Voronoi diagrams and convex polytopes given in Section 2.5 is due to Edelsbrunner and Seidel [93]. It is projectively equivalent to the earlier connection by Brown [30] based on a spherical transform. Exercise 2.3.1 is due to Chazelle and Incerti [52]. For a good survey of Euclidean minimum spanning trees and Euclidean traveling salesman problem, see [185]. Exercise 2.5.8 is due to Cheriton and Tarjan [59]. Exercise 2.5.9 is due to Rosenkrantz, Stearns, and Lewis [195]. For the issues of numerical stability and degeneracies (Section 2.7), see, for example, [83, 91, 99, 104, 118, 121, 159, 232]. Exercise 2.7.1 is from [91].

The deterministic construction of planar convex hulls in Section 2.8.1 is a variant of Graham's scan [114]. Its on-line construction (Exercise 2.8.2) is due to Preparata and Shamos [185, 208]; Exercise 2.8.5 is also due to them. An almost optimal algorithm for maintaining planar convex hulls dynamically (Exercise 2.8.3) was given by Overmars and Van Leeuwen [178]. A linear time algorithm for computing

the convex hull of a simple polygon (Exercise 2.8.7) was given by McCallum and Avis [149].

The incremental algorithm as well as the zone theorem for arrangements of lines (Section 2.8.2) is due to Edelsbrunner, O'Rourke, and Seidel [92], and independently to Chazelle, Guibas, and Lee [51]. Exercise 2.8.10 is due to Chazelle and Edelsbrunner [43]. Exercise 2.8.11 is due to Edelsbrunner and Welzl [98]. For more results on the total complexity of many cells in arrangements, see, for example, Aronov, Matoušek, and Sharir [12], and Edelsbrunner, Guibas, and Sharir [88].

The scanning algorithm for computing intersections of segments in the plane (Section 2.8.3) is due to Bentley and Ottmann [21]. An optimal deterministic algorithm for this problem was given by Chazelle and Edelsbrunner [43], after an earlier work by Chazelle [36]. Independently, Clarkson and Shor [72] and Mulmuley [160] gave optimal randomized algorithms for the same problem (see Chapter 3). A scan-line algorithm for triangulating a planar graph (Exercise 2.8.14) can be found in [110]. A linear time algorithm for the computation of a visibility polygon (Exercise 2.8.15) is due to Chazelle [33]. A scanning algorithm for computing the intersection of two 3D polytopes (Exercise 2.8.13), due to Hertel et al., can be found in [155]. An optimal linear time algorithm for this problem was given by Chazelle [42].

The optimal algorithm for constructing planar Voronoi diagrams in Section 2.8.4 is due to Shamos and Hoey [209]. Fortune [103] gives an alternative scan-line algorithm. A divide-and-conquer algorithm for constructing a 3D convex hull (Exercise 2.8.16) was given by Preparata and Hong [184].

The problem of planar point location (Section 2.8.5) has been extensively studied in computational geometry since its earliest days. The optimal solution in Section 2.8.5 is due to Kirkpatrick [127]. An earlier optimal solution can be found in Lipton and Tarjan [133]. Currently, there are several other optimal solutions. See, for example, Edelsbrunner, Guibas, and Stolfi [89], Sarnak and Tarjan [196]; Exercise 2.8.18 is taken from the latter source. For a survey of the earlier work on (static) planar point location problem, see Preparata and Shamos [185]. For dynamic planar point location, there are near-optimal deterministic solutions [58, 62, 106, 113, 186]; Chiang and Tamassia [63] provide a good survey. In this book, the dynamic planar point location problem is studied in Chapters 3 and 4, and Sections 5.6.1 and 8.3.

Chapter 3

Incremental algorithms

In Chapter 2, we saw that several higher-dimensional sorting problems in computational geometry can be stated in the following form: Given a set N of objects in R^d , construct the induced geometric complex (partition) $H(N)$ quickly. Both the object set N and the complex $H(N)$ depend on the problem under consideration.

Perhaps the simplest way of building $H(N)$ is incremental. By an *incremental* algorithm, we mean an algorithm that builds $H(N)$ by adding one object at a time. We have already used this approach for constructing planar convex hulls (Section 2.8.1) and arrangements of lines (Section 2.8.2). As one may expect, this approach is too simplistic to work efficiently in general. But we shall see that its subtle variation works efficiently for a large number of problems. The variation consists in adding the objects in N , one at a time, but in a *random* order. Such an algorithm is called a *randomized incremental* algorithm. The simplest randomized incremental algorithm is quick-sort (Section 1.2). A general randomized incremental algorithm can be thought of as a generalization of quick-sort to higher dimensions. It has the following abstract form:

1. Construct a random sequence (permutation) S_1, S_2, \dots, S_n of the objects in N .
2. Add the objects in N one at a time according to this sequence. Let N^i denote the set of the first i added objects. At the i th stage, the algorithm maintains the complex (partition) $H(N^i)$ formed by the set N^i . If necessary, it also maintains other auxiliary data structures. For example, in the case of quick-sort, we maintained conflict lists of all intervals in $H(N^i)$ (Section 1.2). Both $H(N^i)$ and the auxiliary data structures need to be updated efficiently during each addition.

In the following sections, we give several instances of incremental algorithms. If the sequence of additions is chosen randomly, these algorithms have a good expected running time, just like quick-sort.

We should remark that an incremental algorithm need not always be on-line (semidynamic). The reason is the following. Let M denote the set of objects that have been added up to any given time. An incremental algorithm can maintain, in addition to $H(M)$, auxiliary structures that depend on the objects in $N \setminus M$, i.e., the unadded objects. An example of such an auxiliary structure is the conflict list we maintained in the case of quick-sort (Section 1.2). In a static setting this is fine, because we are only interested in the final complex $H(N)$. But if we want a truly on-line algorithm, the auxiliary structures can depend only on M . We shall see that all incremental algorithms in this chapter can be converted into truly on-line algorithms.

How are we going to evaluate these on-line algorithms? The basic task of an on-line (semidynamic) algorithm is to maintain the complex $H(M)$ efficiently, where M denotes the set of added objects existing at any given time. We want to update $H(M)$ quickly during the addition of a new object. The best that we can hope for is that this update takes time proportional to the *structural change* in the underlying geometric complex $H(M)$. By the structural change during an update, we mean the total number of newly created and destroyed faces during that update.

For example, consider arrangements of lines (Section 2.2). In this case, M is a set of lines in the plane, and $H(M)$ is the resulting arrangement. It is easy to see that the structural change during any addition is of the order of m , the size of M , ignoring a constant factor. We have already seen that the addition can be accomplished in the same order of time (Section 2.8.2). This is clearly optimal.

Now consider the problem of maintaining a planar Voronoi diagram in an on-line fashion. The problem is to maintain the Voronoi diagram $H(M)$, where M denotes the set of added sites existing at any given time. Again, the best that we can expect from an on-line algorithm is that it adds a new site in time proportional to the structural change in $H(M)$. There is, however, a significant difference between the case Voronoi diagrams and the case of arrangements of lines considered above. In the case of arrangements of lines, the structural change during an update is always of the order of m , ignoring a constant factor and assuming that the lines are in general position. In the case of Voronoi diagrams, it can vary from $\Omega(1)$ to $O(m)$. Accordingly, the cost of an individual update can also vary a lot. But what about the total cost over a sequence of n additions? Can we prove that this total cost is relatively small even if the cost of an individual addition can vary a lot? For this, we need to prove that the total structural change during a sequence of n

additions is small, even if the structural change during a single update varies greatly.

Unfortunately, that is not the case. An adversary, i.e., a malicious user, can come up with an addition sequence that can slow down every on-line algorithm. For example, he can easily come up with a sequence of n additions so that the total structural change in the underlying Voronoi diagram is $\Omega(n^2)$ (how?). Thus, one cannot prove a bound that is better than the pessimistic $O(n^2)$ bound and is good for every update sequence. In such cases, it is quite often instructive to give better bounds that are valid for *most* update sequences, or, if one is less ambitious, for a *random* update sequence. What do we mean by that?

Let us call a sequence of additions an *N -sequence* if N is the set of objects that occur in the sequence. Fix N . Let n denote its size. Put a uniform distribution on all N -sequences. By a *random N -sequence*, we mean a sequence that is randomly chosen from this uniform distribution. This is the same as saying that all permutations of N are equally likely to occur. Now let N be a set of n sites in the plane. In this case, we shall see later that the expected total structural change during a random N -sequence is $O(n)$. It also turns out that the total structural change is $O(n \log n)$ during most N -sequences. By this, we mean that the fraction of N -sequences for which this property is violated is $O(1/g(n))$, where $g(n)$ is a large-degree polynomial in n . In our random model, this is equivalent to saying that the probability that the property is violated for a random N -sequence is $O(1/g(n))$. A similar phenomenon holds in other situations.

For all our on-line algorithms, we shall show that their expected running time on a random N -sequence is good. This justifies why they should work well in practice, even though in theory a malicious user can come up with an update sequence that is guaranteed to slow down any on-line algorithm.

A few remarks are in order here. When we evaluate the performance of an on-line algorithm on a random N -sequence, it does not mean that the algorithm knows the entire universe N . An on-line algorithm only knows about the objects that have been added so far. Here, N is to be thought of as a test universe. In contrast, a static incremental algorithm, whose task is to construct $H(N)$, knows the whole set N .

We should also emphasize that, in reality, the updates need not be random. We should use the above method of evaluating an on-line algorithm only when in a given problem, an adversary can come up with an update sequence that can slow down *any* on-line algorithm. In this situation, our model can justify why the on-line algorithm in question should perform well on a typical update sequence. The situation gets even better if one can prove performance bounds that hold for most N -sequences. (As we have

seen above, this is equivalent to proving performance bounds that hold with high probability in our model of randomness.) In this case, it becomes irrelevant whether or not the addition sequences in reality are random, because our model is used only as a vehicle for proving performance bounds that hold for most addition sequences.

In several problems, adversaries can cause no harm. We have already encountered one such problem before, when N is a set of points on the real line. In this case, the structural change in the underlying partition is $O(1)$ during any update. That is why we did not need to make any assumptions about the nature of updates. The same holds for arrangements of lines, too.

Notation. Throughout this book, we shall denote the size of the object set N by n . We shall use $\| \cdot \|$ to denote the size operation. If X is an ordinary set, then $|X|$ denotes its cardinality. If X is a geometric complex, such as $H(N)$ before, then $|X|$ denotes the total number of its faces of all dimension. In other words, $|H(N)|$ denotes the number of vertices, edges, 2-faces, and so on in $H(N)$.

3.1 Trapezoidal decompositions

Let N be any set of segments in the plane (Figure 3.1(a)). Let $H(N)$ be the resulting trapezoidal decomposition of the plane (cf. Section 2.3 and Figure 3.1(b)). The size of $H(N)$ is $O(k + n)$, where k is the number of intersections among the segments in N , and n is the size of N . In this section, we shall give a randomized incremental algorithm for constructing $H(N)$ whose running time is $O(k + n \log n)$. This improves the $O(k \log n + n \log n)$ bound for the scan-line algorithm (Section 2.8.3), at the cost of determinism. The problem under consideration is strictly harder than the one-dimensional sorting problem—indeed, the latter problem can be thought of as a degenerate instance wherein the segments become points on a line. Hence, this is the best running time one could expect.

We shall build $H(N)$ incrementally by adding the segments in N , one at a time, in random order. At the i th stage of the algorithm, we would have built $H(N^i)$, where N^i denotes the set of the first i randomly chosen segments (Figure 3.1(c)). We assume that $H(N^i)$ is represented in computer memory as a planar graph. Let $S = S_{i+1}$ be the $(i+1)$ th randomly chosen segment for addition. Let s_0 and s_1 denote its two endpoints. Addition of S_{i+1} to $H(N^i)$ is achieved as follows.

Travel and split

Assume at first that we somehow know the trapezoid in $H(N^i)$ containing an endpoint S , say, s_0 . In this case, we travel in $H(N^i)$ from s_0 to s_1 along S . In the process, we encounter the faces R_0, R_1, \dots in $H(N^i)$ one by one

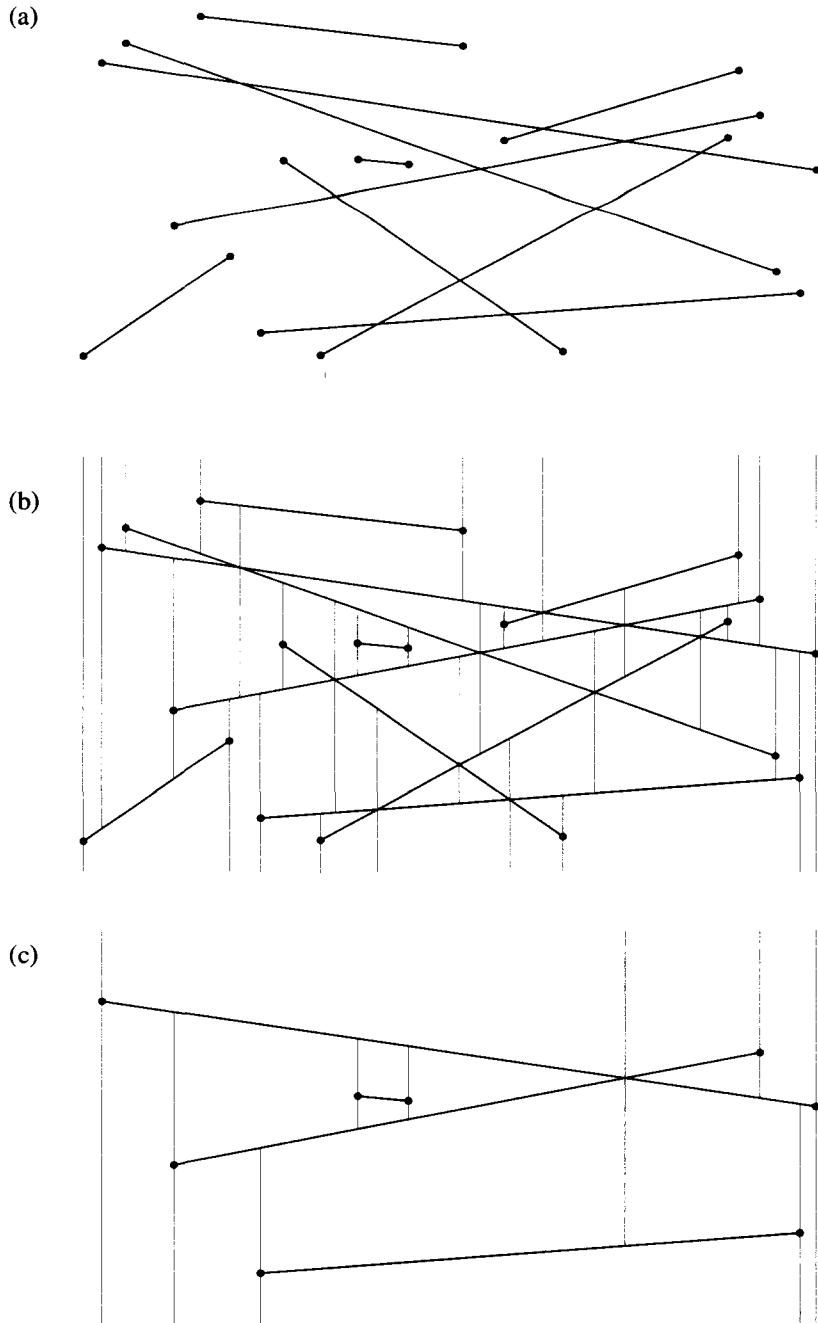


Figure 3.1: (a) N : a set of segments. (b) $H(N)$. (c) $H(N^4)$.

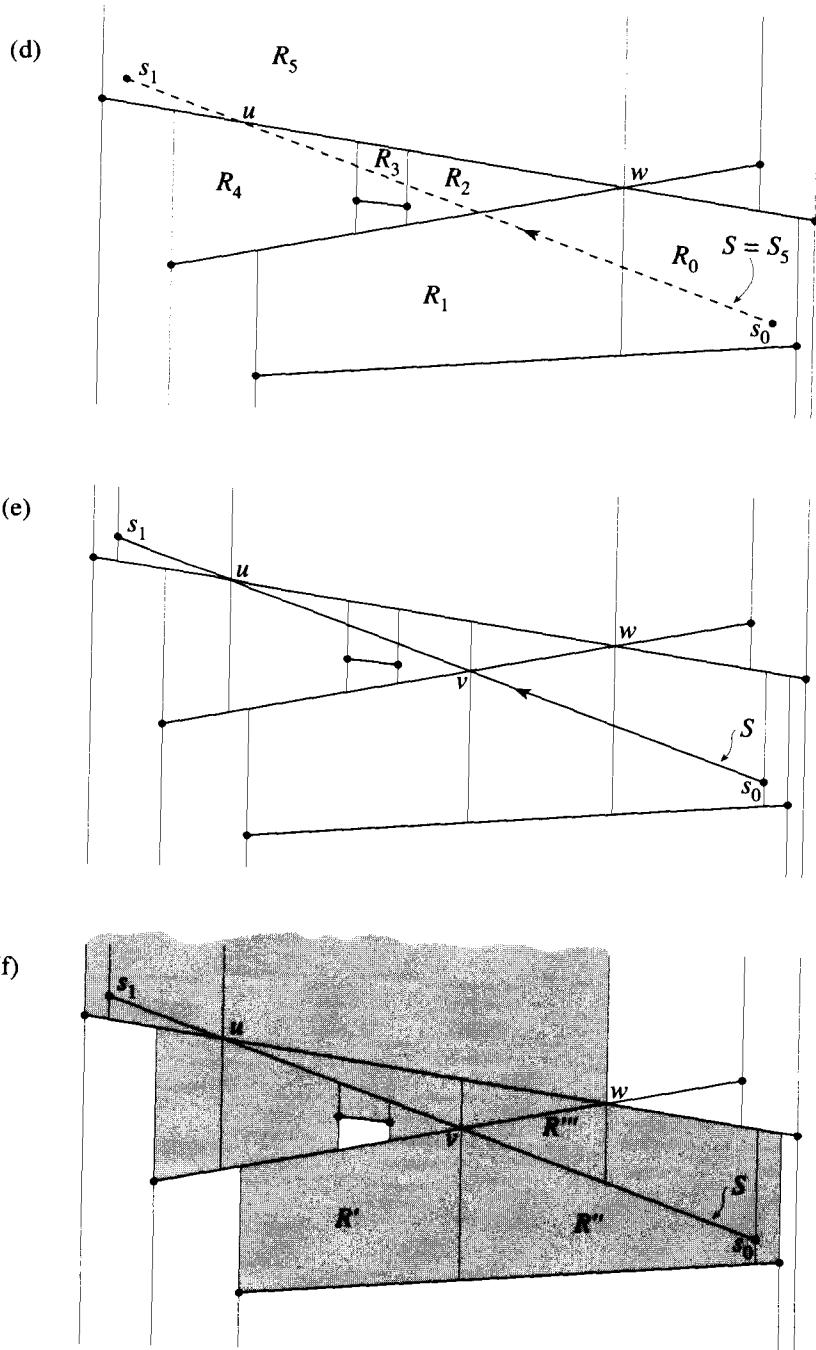


Figure 3.1: (d) Traveling along S_5 in $H(N^4)$. (e) $H'(N_4)$. (f) $H(N_5)$.

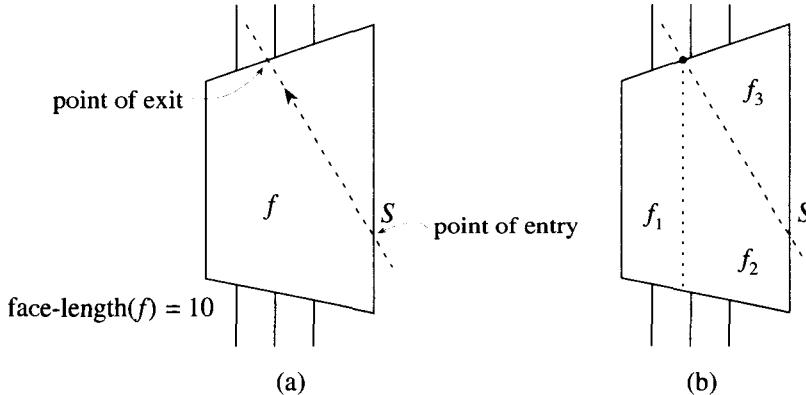


Figure 3.2: (a) Traversing a face f . (b) Splitting f .

(Figure 3.1(d)). If f is any face in $H(N^i)$ intersecting S , then traversing that face along S takes time proportional to $\text{face-length}(f)$. Here $\text{face-length}(f)$ denotes the number of vertices of $H(N^i)$ adjacent to f (Figure 3.2(a)). It can be arbitrarily large because many vertices in $H(N^i)$ can be sitting on the upper and lower sides of f .

We need to split every such traversed face f (Figure 3.2(b)). This means: If S intersects the upper or lower side of f , then we raise a vertical attachment through that intersection within f . If an endpoint of S lies within f , as in the case of R_0 and R_5 in Figure 3.1(d), then we raise a vertical attachment through that endpoint. It is clear that the required splitting of f , in at most four pieces, can also be done in time proportional to $\text{face-length}(f)$.

When we have split all faces that are encountered during our travel from s_0 to s_1 , we get a partition that we shall denote by $H'(N^i)$. Refer to Figure 3.1(e). Notice the added vertical attachments through the endpoints s_0, s_1 and also the newly discovered points of intersection u and v .

Merge

To get $H(N^{i+1})$ from $H'(N^i)$, we only need to contract the vertical attachments in $H(N^i)$ that are intersected by S . After contraction, these attachments rest on S . For example, notice how the intersected vertical attachments in Figure 3.1(e) are contracted in Figure 3.1(f). Contraction of any vertical attachment takes $O(1)$ time. Contracting a vertical attachment merges the faces adjacent to its erased part.

The discussion so far can be summarized as follows.

Proposition 3.1.1 Once we know the trapezoid in $H(N^i)$ containing one endpoint of $S = S_{i+1}$, $H(N^i)$ can be updated to $H(N^{i+1})$ in time propor-

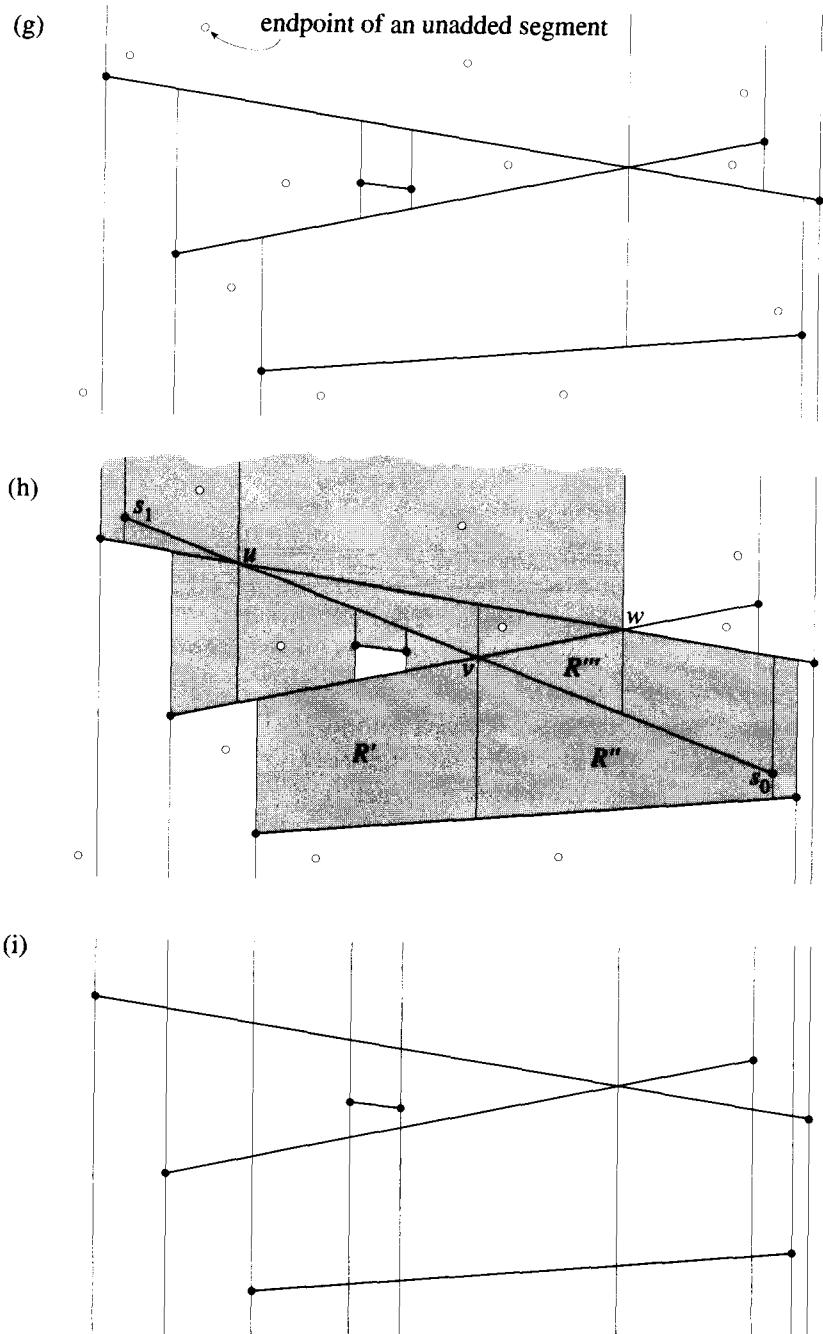


Figure 3.3: (Continuation of Figure 3.1): (a) $H(N^4)$ with conflicts.
 (b) $H(N^5)$ with conflicts. (c) $\bar{H}(N^4)$.

tional to $\sum_f \text{face-length}(f)$, where f ranges over all trapezoids in $H(N^i)$ intersecting S .

One issue still remains. How do we locate an endpoint of S in $H(N^i)$? The solution is surprisingly simple. At the i th stage of the algorithm, we maintain an unordered *conflict list* $L(f)$ with each trapezoid f in $H(N^i)$ (Figure 3.3(a)). This list contains endpoints within f of the unadded segments, i.e., the segments in $N \setminus N^i$. These lists are analogous to the conflict lists we maintained in the randomized incremental version of quick-sort (Section 1.2). With each endpoint of a segment in $N \setminus N^i$, we also maintain a *conflict pointer* to the trapezoid in $H(N^i)$ containing that endpoint. Strictly speaking, it suffices to maintain conflicts of only the left endpoints of unadded segments. We shall ignore this implementation detail in what follows.

Locating an endpoint of the newly added segment $S = S_{i+1}$ is now easy: The conflict pointer associated with each endpoint of S tells us the trapezoid in $H(N^i)$ containing it. This takes $O(1)$ time. On the other hand, we are now faced with an added task of updating the conflict lists during the addition of S (Figure 3.3(b)). This means when we split and merge faces during the addition of S , their conflicts lists should also be split and merged accordingly. The splitting of a conflict list is done trivially in time proportional to its size: If a face f is split in, say, two parts, f_1 and f_2 , then for each point in the conflict list of f , we decide whether it lies in f_1 or f_2 . Merging of conflict lists is done by simply concatenating them, which takes $O(1)$ time per merge. Thus:

Proposition 3.1.2 *The cost of updating conflict lists is $O(\sum_f l(f))$, where f ranges over all trapezoids in $H(N^i)$ intersecting S , and $l(f)$ denotes the conflict size of f , i.e., the size of the conflict list $L(f)$.*

Analysis

We shall now generalize the analysis of quick-sort (Section 1.2) to the preceding algorithm. It follows from Propositions 3.1.1 and 3.1.2 that the total cost of adding S_{i+1} to $H(N^i)$ is $O(\sum_f \text{face-length}(f) + l(f))$, where f ranges over all trapezoids in $H(N^i)$ intersecting S . Our analysis will work backwards—instead of adding S_{i+1} to N^i , we imagine deleting it from N^{i+1} . For this reason, it will be convenient to rewrite the above cost of addition as $O(\sum_g \text{face-length}(g) + l(g))$, where g ranges over all trapezoids in $H(N^{i+1})$ adjacent to S_{i+1} . In Figure 3.3(b), all such trapezoids are shown shaded. The advantage of such rewriting is that the expected cost of the $(i+1)$ th addition, conditional on $N^{i+1} \subseteq N$ being fixed, is now easy to calculate. Remember that each segment in N^{i+1} is equally likely to be involved in the

$(i + 1)$ th addition. Hence, this conditional expected cost is proportional to

$$\frac{1}{i+1} \sum_{S \in N^{i+1}} \sum_g \text{face-length}(g) + l(g),$$

where g ranges over the trapezoids in $H(N^{i+1})$ adjacent to S . Note that each trapezoid in $H(N^{i+1})$ is adjacent to at most four segments in N^{i+1} , and the total conflict size of all faces in $H(N^{i+1})$ is $2(n - i)$. Hence, the above expression is proportional to

$$\frac{n - i + |H(N^{i+1})|}{i+1} = O\left(\frac{n + k_{i+1}}{i+1}\right), \quad (3.1)$$

where k_{i+1} denotes the number of intersections among the segments in N^{i+1} and $|H(N^{i+1})|$ denotes the total size of $H(N^{i+1})$. Now we are ready to use the fact that N^{i+1} is a random sample (subset) of N of size $i + 1$.

Lemma 3.1.3 Fix $j > 0$. The expected value of k_j , assuming that N^j is a random sample of N of size j , is $O(kj^2/n^2)$.

Proof. Consider the set of all intersections among the segments in N . For any fixed intersection v in this set, define a 0–1 random variable I_v such that $I_v = 1$ iff v occurs in $H(N^j)$. Clearly, $k_j = \sum I_v$, where v ranges over all intersections among the segments in N . But note that v occurs in $H(N^j)$ iff both segments containing v are in N^j . This happens with probability $O(j^2/n^2)$. Hence, the expected value of I_v is $O(j^2/n^2)$. The lemma follows by linearity of expectation. \square

It follows from the above lemma and (3.1) that the expected cost of the $(i + 1)$ th addition, conditional on a fixed N^{i+1} , is $O(n/j + kj/n^2)$, where $j = i + 1$. Since this bound does not depend on N^{i+1} , it holds unconditionally as well. Hence, the expected cost of the whole algorithm is $O(n \sum 1/j + k \sum j/n^2)$, that is, $O(k + n \log n)$.

We have thus proved the following.

Theorem 3.1.4 A trapezoidal decomposition formed by n segments in the plane can be constructed in $O(k + n \log n)$ expected time. Here k denotes the total number of intersections among the segments.

3.1.1 History

The algorithm in the previous section is incremental but not on-line. This is because it maintains conflicts, which depend, at the i th stage of the algorithm, on the segments in $N \setminus N^i$. Conflicts help us locate an endpoint of S_{i+1} in $H(N^i)$. If we could associate with $H(N^i)$ some on-line search

structure that allowed us to do the same job efficiently, the maintenance of conflicts would no longer be necessary.

It is possible to associate such a point location structure with $H(N^i)$ simply by remembering the history of the previous additions. We shall denote this search structure by $\text{history}(i)$. It is analogous to the binary search tree (Section 1.3) that was obtained by remembering the history of quick-sort.

At the i th stage of the algorithm, we shall now maintain $H(N^i)$ as well as the search structure $\text{history}(i)$. The leaves of $\text{history}(i)$ will point to the trapezoids in $H(N^i)$. Given any query point p , we shall be able to locate the trapezoid in $H(N^i)$ containing p in $O(\log i)$ time with high probability. This is all that is needed to make the previous randomized incremental algorithm on-line. The expected total cost of point location in this on-line version is $O(\sum_i \log i) = O(n \log n)$. Thus, the expected total running time of the algorithm remains $O(n \log n + k)$, where k is the number of intersections among the segments in N .

Let us now specify $\text{history}(i)$ formally. $\text{History}(0)$ consists of one node, called the *root*, which corresponds to the whole plane. Inductively, assume that $\text{history}(i)$ has been specified. $\text{History}(i+1)$ is obtained from $\text{history}(i)$ as follows. As before, let $S = S_{i+1}$ denote the $(i+1)$ th segment to be added. If a trapezoid $f \in H(N^i)$ intersects S , we say that it is *killed* during the $(i+1)$ th addition. Accordingly, we mark the leaf in $\text{history}(i)$ pointing to f as killed by the segment S . We associate with this leaf a pointer to the specification of S . In Figure 3.1(d), the leaves of $\text{history}(4)$ corresponding to the trapezoids R_0, \dots, R_5 would be marked as killed. Note that we do not remove these leaves. We simply mark them as killed by S_5 .

Similarly, we allocate new nodes for the trapezoids in $H(N^{i+1})$ that are newly created during the $(i+1)$ th addition. We also store the specifications of these newly created trapezoids at the corresponding nodes. Thus, in Figure 3.1(f), new nodes are allocated for the shaded regions. Given a killed trapezoid f in $H(N^i)$ and a newly created trapezoid g in $H(N^{i+1})$, we say that g is a *child* of f if $f \cap g$ is nonempty. In this case, we associate a pointer with the node in $\text{history}(i)$ corresponding to f pointing toward the newly created node (leaf) for g . This completely specifies $\text{history}(i+1)$ given $\text{history}(i)$. Referring to Figure 3.1(d)–(f), the children of $R^1 \in H(N^4)$ are $R', R'', R''' \in H(N^5)$. Note that the number of children of any killed leaf in $\text{history}(i)$ is bounded; in fact, it is at most 4.

$\text{History}(i)$ is a directed acyclic graph with one root, wherein the directed edges correspond to the pointers from parents to children. The out-degree of each node in this graph is at most 4. Let us now see how $\text{history}(i)$ can be used for locating a query point in $H(N^i)$. Let us denote the query point by p . Our goal is to locate the trapezoid in $H(N^i)$ containing the query point.

We begin at the root of history(i). Precisely one of its children corresponds to a trapezoid containing p . Because the number of children is at most four, this child can be determined in $O(1)$ time. We descend to this child in the history and keep on repeating this process until we reach a leaf. The leaf corresponds to the trapezoid in $H(N^i)$ containing p .

Analysis

The cost of locating the query point p is clearly proportional to the length of the search path in history(i). The following lemma proves that this length is $\tilde{O}(\log i)$.

Lemma 3.1.5 *For a fixed query point, the length of the search path in history(i) is $\tilde{O}(\log i)$.*

Proof. The basic idea is the same as in the case of randomized binary trees (Section 1.3). Let p be a fixed query point. We follow the search path backwards in time. Towards this end, we imagine removing the segments from N^i , one at a time, in the reverse order S_i, S_{i-1}, \dots, S_1 . For $1 \leq j \leq i$, define a 0–1 random variable V_j so that $V_j=1$ iff the trapezoid in $H(N^j)$ containing p is adjacent to S_j . Note that this trapezoid will be destroyed during the imaginary deletion of S_j from N^j . For example, if p were to lie in any of the shaded trapezoids in Figure 3.1(f), then V_5 would be one. Clearly, the length of the search path in history(i) is $\sum_{j=1}^i V_j$. But note that every segment in N^j is equally likely to occur as S_j . Hence, the probability that $V_j = 1$ is $\leq 4/j$. This is because at most four segments in N^j are adjacent to the trapezoid in $H(N^j)$ containing p . This means that the expected value of V_j is $O(1/j)$. By linearity of expectation, it follows that the expected length of the search path is $O(\sum 1/j) = O(\log n)$. To show that the length of the search path is $\tilde{O}(\log n)$, one applies Chernoff's technique as in Section 1.3. \square

The following simple generalization of the previous lemma will be useful later.

Lemma 3.1.6 *For any $k \leq i$, if we know the node in history(k) that corresponds to the trapezoid in $H(N^k)$ containing the query point p , then we can locate the trapezoid in $H(N^i)$ containing p in $O(1 + \log[i/k])$ expected time.*

Proof. The length of the search path joining the nodes, which correspond to the trapezoids in $H(N^k)$ and $H(N^i)$ containing p , is $\sum_{j=k}^i V_j$. Hence, its expected value is $O(\sum_{j=k}^i 1/j) = O(\log[i/k])$. \square

A careful examination of the proof of Lemma 3.1.5 reveals that it rests on the following single property.

Bounded degree property: The answer to every query over $H(N^i)$ is defined by a bounded number¹ of objects in N^i .

In the present case, the answer to a query over $H(N^i)$ is the trapezoid containing the query point p . It is clearly defined by a bounded number of segments in N^i that are adjacent to the trapezoid. In a general case, Lemma 3.1.5 continues to hold as long as the bounded degree property holds. This observation will turn out to be crucial in later applications.

One subtle issue still remains. Lemma 3.1.5 proves a high-probability bound on the length of the search path for a fixed query point. What can we say about an arbitrary search path in $\text{history}(i)$? First, we shall show that the total number of search paths in $\text{history}(i)$ is bounded by a polynomial of a fixed degree in i ; $O(i^2)$, to be precise. Consider the refinement $\bar{H}(N^i)$ obtained by extending the vertical attachments in $H(N^i)$ to infinity in both directions (Figure 3.3(c)). The search paths of all points lying in a fixed trapezoid of $\bar{H}(N^i)$ are identical. (In contrast, the search paths of all points in a fixed trapezoid of $H(N^i)$ need not be identical: If two points belong to different trapezoids in $H(N^j)$, for some $j < i$, then their search paths are not identical, even if they belong to the same trapezoid in $H(N^i)$.) It follows that the total number of search paths in $\text{history}(i)$ is bounded by the size of $\bar{H}(N^i)$, which is $O(i^2)$. Arguing as we did in the case of randomized binary trees (Observation 1.3.1), it follows that the depth of $\text{history}(i)$ is $\tilde{O}(\log i)$. By the depth of $\text{history}(i)$, we mean the maximum length of any search path in it.

Exercises

3.1.1 Modify the algorithm in this section so that degeneracies in the input are allowed. This means that any number of segments are allowed to share the endpoints, and more than two segments are allowed to share a point of intersection. Show that the bound on the expected running time remains the same as before.

3.1.2 In this section, we assumed that the trapezoidal decomposition $H(N)$ is represented as a planar graph. Consider the following *opaque representation* of $H(N)$. In this representation, a vertex v in $H(N)$ is considered adjacent to a trapezoid Δ iff v is a corner of Δ . Thus, each trapezoid is adjacent to at most four vertices. For each trapezoid, we maintain a circular list of its adjacent vertices. For each vertex, we maintain a list of adjacent trapezoids.

- (a) Modify the randomized incremental algorithm in this section assuming that each $H(N^i)$ is represented in the opaque fashion. (Hint: How do you go from one face to another during the travel on S_{i+1} ?)
- (b) Prove that the expected running time of this modified algorithm remains $O(k + n \log n)$.
- (c) Compare the opaque and planar graph representations.

¹By a bounded number, we mean a number bounded by a constant.

3.1.3 Let N be a set of n monotone chains. By a chain, we mean a sequence of segments wherein each segment shares an endpoint with the segment before and after it. The chain is called monotone if any line parallel to the y -axis can intersect it in at most one point. Let \bar{N} be the set of segments in all the chains in N . Let \bar{n} be its size. Let k be the total number of intersections among the segments. Give a randomized algorithm for constructing $H(\bar{N})$ in expected $O(\bar{n} + k + n \log n)$ time. (Hint: Add one chain at a time.)

***3.1.4** Let N be a set of algebraic segments in the plane of bounded degree. Let $H(N)$ be the resulting trapezoidal decomposition (Exercise 2.3.5). Give a randomized incremental algorithm for constructing $H(N)$. Show that its expected running time is $O(d^2(k + n \log n))$, where d is the degree bound. (Hint: Break the algebraic segments into monotonic pieces by splitting them at the *critical* points, where tangents become parallel to the y -axis. Add monotonic pieces one at a time.)

3.1.2 Planar graphs

If the segments in N do not intersect, except at their endpoints, then they form a planar graph. The previous algorithm can be used to construct a trapezoidal decomposition of this planar graph (Figure 2.7) and also a point location structure for this trapezoidal decomposition. This search structure can be used for locating a trapezoid containing a given query point. Once we know the trapezoid containing the query point, we immediately know the face of the planar graph containing that point. Thus, we get a point location structure for a planar graph. We shall give another point location structure in Section 8.3. The present point location structure is not fully dynamic like the one in Section 8.3. We shall make it fully dynamic in Chapter 4.

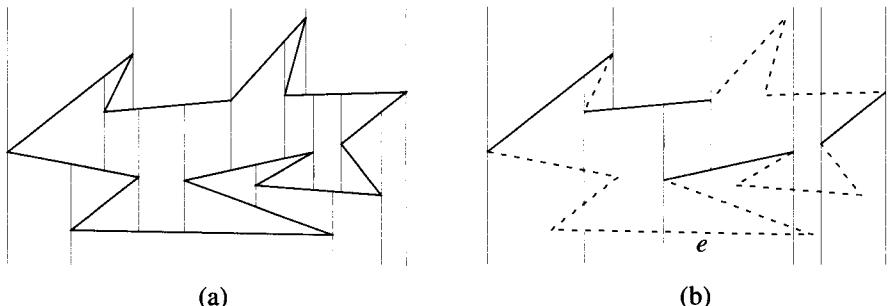


Figure 3.4: (a) Trapezoidal decomposition of a simple polygon. (b) Tracing a simple polygon.

3.1.3 Simple polygons*

Another interesting case arises when the segments in N form a simple, nonintersecting closed polygon (Figure 3.4(a)). In this case, the expected running

time of our trapezoidal decomposition algorithm becomes $O(n \log n)$. It is of interest to know whether this running time can be reduced further. That is indeed the case. In this section, we shall modify the algorithm so that its expected running time becomes $O(n \log^* n)$, when the segments form a simple polygon. Here $\log^* n$ denotes the number of log operations that need to be performed if we start with the number n and want to end up with a number less than 1. In other words, it is the least positive integer k such that the k th iterated log of n is less than 1.

The basic algorithm is the same as before. We add the segments in N one at a time in random order. We maintain $H(N^i)$ as well as $\text{history}(i)$ at all times. If we ignore the cost of locating an endpoint of the segment being added, then the expected running time of the algorithm is $O(n)$. This follows because k , the number of intersections, is zero; the $O(n \log n)$ term in the expected running time of the on-line trapezoidal decomposition algorithm (Section 3.1.1) arises due to the point location cost. We only need to change the procedure for locating an endpoint.

Let

$$j_1 = n / \log n, j_2 = n / \log^{(2)} n, j_3 = n / \log^{(3)} n, \dots$$

Here $\log^{(k)} n$ denotes the function obtained by iterating the log function k times. By convention, we let $j_0 = 1$. Let r be the least integer such that $j_r > n$. Obviously, $r = \log^* n$. Consider any fixed time i . Let k be the largest integer such that $j_k \leq i$. We shall ensure that the following invariant holds at every time i .

Invariant: For each segment in $N \setminus N^i$, we know the node in $\text{history}(j_k)$ that corresponds to the trapezoid in $H(N^{j_k})$ containing its, say, left endpoint.

Lemma 3.1.6 then implies that the cost of endpoint location during the i th addition, for $i \in [j_k, j_{k+1}]$, is $O(1 + \log[j_{k+1}/j_k]) = O(\log^{(k+1)} n)$. Hence, the total expected cost of point location over all $i \in [j_k, j_{k+1}]$ is

$$O(j_{k+1} \log^{(k+1)} n) = O(n).$$

This implies that the expected total cost of point location over the whole algorithm is $O(nr) = O(n \log^* n)$.

It remains to ensure the invariant. This will be done as follows. For each $k \geq 1$, we stop at time j_k and trace the whole polygon in $H(N^{j_k})$ (Figure 3.4(b)). At the end of the tracing, we know, for each vertex of the polygon, the trapezoid in $H(N^{j_k})$ containing that vertex. This also tells us the corresponding leaf of $\text{history}(j_k)$. Tracing is done in two steps: (1) We locate in $H(N^{j_k})$ one vertex of the polygon by using $\text{history}(j_k)$. This takes $O(\log j_k)$ expected time. (2) We travel along the entire polygon in $H(N^{j_k})$

(Figure 3.4(b)). The second step takes time proportional to

$$C(j_k) = \sum_f c(f),$$

where f ranges over all trapezoids in $H(N^{j_k})$, and $c(f)$ denotes the conflict size of f , i.e., the number of segments in $N \setminus N^{j_k}$ intersecting f . Note that $c(f)$ also bounds the number of times f is visited during the tracing. (Caution: This definition of the conflict size is different from the previous definition.)

How large can $C(j_k)$ be? One might think it is always $O(n)$. That is not correct, because a fixed segment in the given polygon can intersect several trapezoids in $H(N^{j_k})$ (e.g., segment e in Figure 3.4(b)). But notice that N^{j_k} is a random subset of N of size j_k . Hence, one might expect that the expected value of $C(j_k)$ is $O(n)$, or in other words, the average conflict size of the trapezoids in $H(N^{j_k})$ is $O(n/j_k)$. This is indeed the case. But it is a nontrivial fact. It follows from a general result on average conflict size to be proved in Chapter 5 (cf. Theorem 5.5.5 and Example 5.5.6). At the moment, we shall take its truth for granted. With this assumption, it follows that the expected cost of tracing at time j_k is $O(n)$. Hence, the expected total cost of tracing over the whole algorithm is $O(nr) = O(n \log^* n)$.

We have thus proved the following.

Theorem 3.1.7 *Trapezoidal decomposition of a simple polygon and an associated point location structure can be constructed in $O(n \log^* n)$ expected time. Here n is the size of the polygon. The cost of point location is $\tilde{O}(\log n)$.*

In additional $O(n)$ time, we can also construct a triangulation of the polygon (Exercise 2.3.1).

3.2 Convex polytopes

Let N be a set of n half-spaces in R^d , where $d = 2$ or 3 . Let $H(N)$ be the convex polytope formed by intersecting these half-spaces. (When $d = 2$, $H(N)$ is just a convex polygon.) In this section, we shall give a randomized incremental algorithm for constructing $H(N)$ with $O(n \log n)$ expected running time. We shall also show later how this incremental algorithm can be made on-line. Our algorithm will build $H(N)$ by adding the half spaces in N one at a time in random order. Figure 3.5 illustrates the algorithm for $d = 2$.

Let N^i denote the set of the first i randomly added half-spaces. For the sake of convenience, we shall assume, in what follows, that $H(N^i)$ is always bounded. There is a very simple way to ensure this using a certain *bounding*

box trick, which we shall use on several other occasions. The trick is the following. We add to N^0 , and hence to every N^i , half-spaces enclosing a huge symbolic box approaching infinity. More precisely, we let N^0 contain the half-spaces $x_i \geq -\delta$ and the half-spaces $x_i \leq \delta$, for each $i \leq d$. Here δ is a huge constant. This makes every $H(N^i)$ bounded, conceptually speaking. At the same time, no information is lost, because the vertices of the original $H(N^i)$ are contained within this box if δ is large enough. But how do we choose δ ? The cleanest solution is to treat δ as just a formal symbol that stands for a constant approaching ∞ . Thus, we have $\delta \times a = \delta$, for any real $a > 0$, $\delta + b = \delta$, for any real b , $b/\delta = 0$, and so on, asymptotically speaking. The controversial quantities like $\delta - \delta$ will never arise because every quantity that arises in the algorithm must make sense geometrically. All vertices of the new $H(N^i)$ which lie on the boundary of the bounding box approach infinity in a fixed way as δ approaches infinity. But $H(N^i)$ can be treated as bounded in the algorithm, because it is indeed bounded for every fixed value of δ .

Strictly speaking, it is not even necessary to ensure that each $H(N^i)$ is bounded. The following algorithm can be directly modified to handle the unbounded case. We shall leave the required simple modifications to the reader.

We assume that $H(N^i)$ is represented in computer memory by its facial lattice (Section 2.4). Now consider the addition of the $(i+1)$ th half-space $S = S_{i+1}$ to $H(N^i)$ (Figure 3.5). We shall obtain $H(N^{i+1})$ from $H(N^i)$ by splitting off the “cap” given by $\text{cap}(S_{i+1}) = H(N^i) \cap \bar{S}$, where \bar{S} denotes the complement of S (Figure 3.5(b)). Details of this operation are as follows. Assume at first that we are given some vertex $p \in H(N^i)$ that *conflicts* with S . By this, we mean that p is contained in the complement \bar{S} . If a conflicting

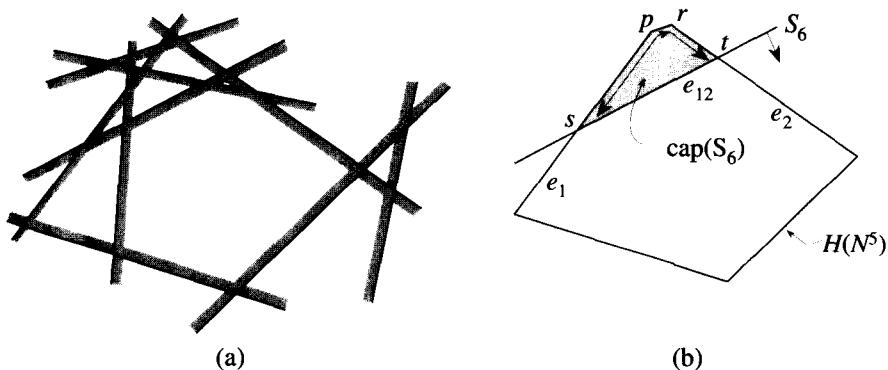


Figure 3.5: (a) N : a set of half-spaces. (b) Updating $H(N^5)$ to $H(N^6)$.

vertex does not exist, we are told that too. In the latter case, S must be redundant, because $H(N^i)$ is bounded, so it can be thrown away. Otherwise, consider the edge skeleton of $H(N^i)$. By this, we mean the obvious graph formed by the vertices and the edges of $H(N^i)$. The edges of $H(N^i)$ intersecting \bar{S} form a connected subgraph of this skeleton and the subgraph contains p . Hence, all these edges can be detected by performing a simple search on the edge skeleton of $H(N^i)$. We start this search at the vertex p and take care not to enter the half-space S at any time. At the end of this search, we shall have visited all edges of $H(N^i)$ intersecting \bar{S} . The 2-faces of $H(N^i)$ intersecting \bar{S} are precisely the ones adjacent to the visited edges. Let f be any such 2-face intersecting \bar{S} . If f lies completely within \bar{S} , we remove f and the adjacent edges and vertices from the facial lattice. If f intersects \bar{S} partially, we split f . This means: We replace f in the facial lattice with $f \cap S$. The edges of f contained within \bar{S} are removed. The two edges of f that partially intersect \bar{S} are split. We create a new edge $f \cap \partial S$, where ∂S denotes the boundary of S .

Finally, we also need to create a new face $H(N^i) \cap \partial S$. The adjacency information is also appropriately updated. This yields $H(N^{i+1})$ as required.

Analysis

Assume that we are given an initial conflicting vertex p in $H(N^i)$ for free. Then the whole procedure of adding $S = S_{i+1}$ takes time that is proportional to the number of destroyed vertices in $H(N^i)$ plus the number of newly created vertices in $H(N^{i+1})$. Each created vertex can be destroyed only once. Hence, we can simply ignore the number of destroyed vertices as far as the asymptotic analysis of the algorithm is concerned. In effect, this means that the *amortized*² cost of the $(i+1)$ th addition can be taken to be proportional to the number of newly created vertices in $H(N^{i+1})$. These are the vertices in $H(N^{i+1})$ that are contained in ∂S_{i+1} . We shall denote their number by $m(S_{i+1}, N^{i+1})$.

We shall analyze the algorithm in a backward fashion. We shall estimate the expected cost of adding S_{i+1} , conditional on a fixed N^{i+1} . In other words, we imagine deleting S_{i+1} from N^{i+1} . By the preceding discussion, and the fact that each half-space in N^{i+1} is equally likely to occur as S_{i+1} , it follows that this conditional expected cost is proportional to

$$\frac{1}{i+1} \sum_{S \in N^{i+1}} m(S, N^{i+1}).$$

²Amortization is just a convenient bookkeeping procedure. Amortized cost of a step can be much less than its actual cost. However, the sum of the amortized costs of all steps in the whole algorithm should be of the same order (within a constant factor) as the sum of the actual costs.

By our usual general position assumption, each vertex in $H(N^{i+1})$ is contained in two bounding lines, for $d = 2$, and at most three bounding planes, for $d = 3$. In other words, each vertex of $H(N^{i+1})$ is adjacent to a bounded number of objects in N^{i+1} . Hence, the sum in the above expression is bounded by d times the number of active vertices at time $i + 1$. By the active vertices at time $i + 1$, we mean the vertices on the convex polytope $H(N^{i+1})$. Their number is $O(i)$, for $d = 2$ or 3 (Section 2.4.1). Thus, the expected amortized cost of the $(i + 1)$ th addition, conditional on a fixed N^{i+1} , is $O(1)$. As N^{i+1} is arbitrary, the bound holds unconditionally as well.

We summarize the preceding discussion, letting $j = i + 1$.

Lemma 3.2.1 *The expected number of newly created vertices in the j th random addition, conditional on a fixed N^j , is bounded by d/j times the number of active vertices over N^j (i.e., the ones in $H(N^j)$). Since N^j is itself the random sample of N , the expected value, without any conditioning, is bounded by $d e(j)/j$, where $e(j)$ denotes the expected number of active vertices at time j . For $d = 2, 3$, this expected number is $O(j)$. Hence, the expected amortized cost of the j th addition is $O(1)$, assuming that we are given a vertex of $H(N^{j-1})$ in conflict with S_j .*

3.2.1 Conflict maintenance

One issue still remains. How do we locate a vertex of $H(N^i)$ in conflict with S_{i+1} ? Our basic idea is similar to the one we used in the trapezoidal decomposition algorithm (Section 3.1). At every time i , we maintain, for every half-space $I \in N \setminus N^i$, a pointer to one vertex in $H(N^i)$ in conflict with it, if there is one. The pointer is assumed to be a two-way pointer. If one wishes, one can associate with I a set of pointers to all conflicting vertices in $H(N^i)$. But it will unnecessarily increase the space requirement.

During the addition of $S = S_{i+1}$, we now need to update the conflict information accordingly (refer to Figures 3.5(c)–(e)). Suppose the conflict pointer from some half-space $I \in N \setminus N^{i+1}$ points to a vertex r in $\text{cap}(S_{i+1})$. We need to find another vertex in $H(N^{i+1})$ in conflict with I . In this case, some newly created vertex of $H(N^{i+1})$, i.e., a vertex contained in ∂S_{i+1} , must conflict with I . When, $d = 2$, there are only two newly created vertices. So this step takes $O(1)$ time. For $d = 3$, a different procedure is required, because the number of newly created vertices can be large. In the three-dimensional case, we note that the edge skeleton of the convex polytope $\text{cap}(S_{i+1}) \cap \bar{I}$, where \bar{I} denotes the complement of I , forms a connected subgraph of the edge skeleton of $\text{cap}(S_{i+1})$. This means if we perform a search on the edge skeleton of $\text{cap}(S_{i+1})$, starting at r and taking care not to enter the half-space I at any time, we shall eventually discover a newly

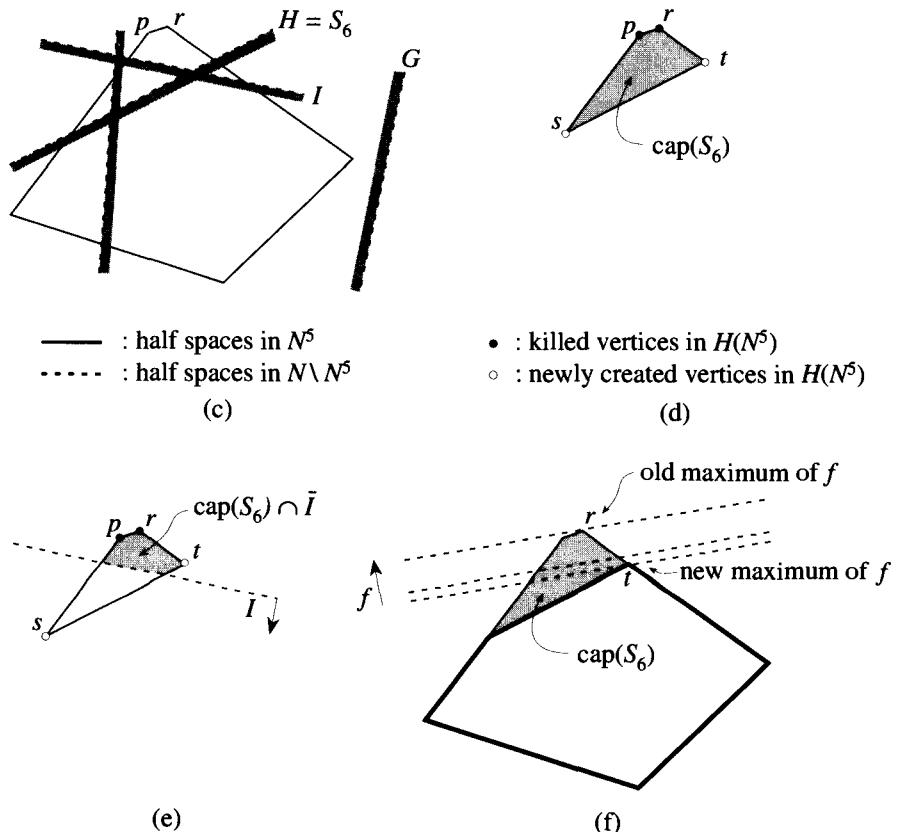


Figure 3.5: (continued) (c) Conflicts in $H(N^5)$. (d) $\text{Cap}(S_6)$. (e) Conflict relocation. (f) Relocating the maximum of a linear function.

created vertex in conflict with I , which is what we seek. Several of the newly created vertices can conflict with I . We just need one. So we stop our search as soon as we find one. In Figure 3.5(e), if we start at r , we shall reach a vertex t that is in conflict with I .

Analysis

We now turn to estimating the expected cost of the above search. Observe that the edge skeleton under consideration is a bounded degree graph because each vertex of $H(N^i)$ is contained in exactly three bounding planes, for $d = 3$, and two bounding lines, for $d = 2$. Hence, in the worst case, the cost of the above search is proportional to the number of vertices in $\text{cap}(S_{i+1}) \cap \bar{I}$. This is just the number of destroyed vertices in $H(N^i)$ in conflict with I plus the number of newly created vertices in $H(N^{i+1})$ in conflict with I . Each

created vertex can be destroyed only once. Hence, the number of destroyed vertices can be simply ignored in our asymptotic cost estimate. In effect, this means that the amortized cost of the search is proportional to the number $k(N^{i+1}, S_{i+1}, I)$ of the newly created vertices in $H(N^{i+1})$ in conflict with I . These are the conflicting vertices in $H(N^{i+1})$ that are adjacent to S^{i+1} , i.e., they are contained in ∂S_{i+1} .

In what follows, we shall let $j = i + 1$ to simplify the expressions a bit. Summing over all $I \in N \setminus N^j$, it follows that the amortized cost of updating the conflict information during the j th addition is proportional to $\sum_v l(v)$, where v ranges over the vertices in $H(N^j)$ contained in ∂S_j , and $l(v)$ denotes the *conflict size* of v , i.e., the number of half-spaces in $N \setminus N^j$ in conflict with v . In other words, the amortized cost of conflict relocation in the j th addition is proportional to the total conflict size of the newly created vertices at time j . What do we expect the conflict size of a vertex created at time j to be? Certainly $O(n)$, if $j = 1$, and zero, if $j = n$. Relying on the magical powers of randomization, one can hope to interpolate between these two extremes linearly. In other words, the expected conflict size of the vertices created at time j should be $O([n - j]/j)$, roughly speaking. That is almost correct! In what follows, we shall make this naive analysis rigorous.

First, we shall estimate the expected value of $k(N^j, S_j, I)$, conditional on a fixed N^j , assuming that S_j is randomly selected from N^j . In other words, each half-space in N^j is equally likely to occur as S_j . Each vertex in $H(N^j)$ is adjacent to d half-spaces in N^j . Hence, this expected value is proportional to

$$\frac{1}{j} \sum_{S \in N^j} k(N^j, S, I) \leq \frac{d k(N^j, I)}{j},$$

where $k(N^j, I)$ denotes the number of vertices in $H(N^j)$ in conflict with I .

Summing over I , the expected total conflict size of the newly created vertices during the j th addition, conditional on a fixed N^j , is bounded by

$$\frac{d}{j} \sum_{I \in N \setminus N^j} k(N^j, I). \quad (3.2)$$

On the other hand, assuming that each half-space in $N \setminus N^j$ is equally likely to occur as S_{j+1} , the expected value of $k(N^j, S_{j+1})$ is

$$E[k(N^j, S_{j+1})] = \frac{1}{n-j} \sum_{I \in N \setminus N^j} k(N^j, I).$$

Hence, the expression in (3.2) can be rewritten as

$$d \frac{n-j}{j} E[k(N^j, S_{j+1})].$$

But $k(N^j, S_{j+1})$ can also be interpreted as the number of vertices of $H(N^j)$ that are destroyed during the $(j+1)$ th addition. Summing over all j , it follows that the expected total conflict size of all vertices created in the course of the algorithm is bounded by

$$\sum_{j=1}^n d \frac{n-j}{j} \times \text{the expected number of vertices destroyed at time } j+1. \quad (3.3)$$

By linearity of expectation, this is the same as the expected value of

$$\sum_{\sigma} d \frac{n - [j(\sigma) - 1]}{j(\sigma) - 1}, \quad (3.4)$$

where σ ranges over all vertices created in the course of the algorithm, and $j(\sigma)$ denotes the time when σ is destroyed. Let $i(\sigma)$ denote the time when σ is created. Clearly, $i(\sigma) \leq j(\sigma) - 1$. Hence,

$$\frac{n - [j(\sigma) - 1]}{j(\sigma) - 1} = \frac{n}{j(\sigma) - 1} - 1 \leq \frac{n}{i(\sigma)} - 1 = \frac{n - i(\sigma)}{i(\sigma)}.$$

Substituting this in (3.4) and rearranging, the expression therein is seen to be bounded by

$$\sum_{j=1}^n d \frac{n-j}{j} \times \text{the number of vertices created at time } j. \quad (3.5)$$

Using linearity of expectation again, it follows that the expected total conflict size of all vertices created in the course of the algorithm is bounded by

$$\sum_{j=1}^n d \frac{n-j}{j} \times \text{the expected number of vertices created at time } j. \quad (3.6)$$

In other words, the vertices created at time j have average conflict size $O([n-j]/j)$, in some rough amortized sense. As already remarked, this is exactly what one would expect in a naive sense.

Using Lemma 3.2.1, the expression in (3.6) becomes

$$\sum_{j=1}^n d^2 \frac{n-j}{j} \frac{e(j)}{j}. \quad (3.7)$$

Here $e(j)$ denotes the expected number of active vertices at time j . By active vertices at time j , we mean the vertices on the convex polytope $H(N^j)$. Their

number, for $d = 2, 3$, is always $O(j)$ (Section 2.4.1). Hence, $e(j)$ is trivially $O(j)$. It follows that the total expected cost of conflict maintenance over the whole algorithm is $O(\sum_j (n - j)/j) = O(n \log n)$.

Combining with Lemma 3.2.1, we have proved the following.

Theorem 3.2.2 *Given a set of n half-spaces in R^d , for $d = 2$ or 3 , the convex polytope formed by the intersection of these half-spaces can be constructed in expected $O(n \log n)$ time.*

3.2.2 History

The algorithm in the last section is incremental but not on-line. This is because it maintains, at every stage, conflicts of the half-spaces not yet added. The conflicts are needed so that during the addition of S_{i+1} we know a vertex of $H(N^i)$ contained in the complement \bar{S}_{i+1} . It turns out that one can maintain an on-line search structure that will let us do this job efficiently. The basic idea is the same as in Section 3.1.1. This search structure at time i will be the history of the previous i additions. We shall denote it by $\text{history}(i)$. The leaves of $\text{history}(i)$ will point to the vertices in $H(N^i)$.

Formally, $\text{history}(i)$ is defined as follows. We assume, as before, that N^0 contains half-spaces containing a large box approaching infinity. Accordingly, $\text{history}(0)$ will contain nodes that correspond to the corners of this symbolic cube. Inductively, assume that $\text{history}(i)$ is defined. $\text{History}(i+1)$ is obtained from $\text{history}(i)$ as follows. We shall say that a vertex in $H(N^i)$ is *killed* at time $i+1$ if it is in conflict with S_{i+1} . Accordingly, we mark the corresponding leaf in $\text{history}(i)$ as killed by the half-space S_{i+1} . We also allocate new nodes in the history for the newly created vertices in $H(N^{i+1})$. These newly created nodes are in one-to-one correspondence with the vertices in the bottom of $\text{cap}(S_{i+1})$; by the bottom of $\text{cap}(S_{i+1})$ we mean its facet contained in ∂S_{i+1} . Similarly, the killed nodes in $\text{history}(i)$ are in one-to-one correspondence with the remaining vertices in $\text{cap}(S_{i+1})$. Hence, we link these killed and newly created nodes in a way that corresponds to adjacencies among the vertices of $\text{cap}(S_{i+1})$ (Figure 3.5(d)). We shall call this linkage structure $\text{link}(S_{i+1})$. It can be thought of as the edge skeleton of $\text{cap}(S_{i+1})$.

Given a half-space $I \notin N^i$, $\text{history}(i)$ can be used to find a vertex of $H(N^i)$ conflicting with I as follows. First, we find a node τ in the history that corresponds to a vertex of $H(N^0)$ in conflict with I . This takes $O(1)$ time. Let S_j , $j \leq i$, be the half-space, if any, that killed τ . Using $\text{link}(S_j)$, we find a node τ' that corresponds to a vertex in $H(N^j)$ in conflict with I . $\text{Link}(S_j)$ corresponds to the edge skeleton of $\text{cap}(S_j)$. Hence, this can be done by searching in this edge skeleton just as in Section 3.2.1 (see Figure 3.5(e)).

We repeat this process until we reach a leaf of history(i) that corresponds to a vertex in $H(N^i)$ in conflict with I .

History lets us convert the previous randomized incremental algorithm for convex polytopes into a randomized on-line algorithm. At the i th stage of the algorithm, we now maintain $H(N^i)$ as well as history(i). We locate a vertex of $H(N^i)$ in conflict with S_{i+1} using history(i). The rest of the algorithm is the same as before.

Analysis

It should be clear that the above conversion does not change the total running time of the algorithm by more than a constant factor: History only postpones the conflict relocation work. What does that mean? Consider a fixed $j \leq n$. In the original algorithm, we relocated all conflicts within $\text{cap}(S_j)$ during the addition of S_j . In the new algorithm, if $\text{cap}(S^j)$ is searched during the addition S^{i+1} , $i \geq j$, this, in effect, corresponds to relocating a conflict of S_{i+1} within $\text{cap}(S_j)$. The only difference is that we are now doing this relocation work at time $i + 1$ instead of at time j .

It follows that the expected running time of the above on-line algorithm over a sequence of n random additions remains $O(n \log n)$.

3.2.3 On-line linear programming

Finally, we shall show that, with some modifications, history(i) can be used to locate the maximum (or minimum) of any linear function on $H(N^i)$ in $\tilde{O}(\log^2 i)$ time, for $d = 3$. The idea is to augment $\text{link}(S_j)$ so that the maximum of any linear function on the bottom of $\text{cap}(S_j)$ can be located in logarithmic time. By the bottom of $\text{cap}(S_j)$, we mean its facet contained in ∂S_j . It is a convex polygon. Hence, we can easily construct a search structure that allows us locate the optimum of any linear function on it in logarithmic time³ (Exercise 2.8.4).

Now let us see how this augmented history can be used to locate the maximum of any linear function on $H(N^i)$. Let f be a query linear function. We first locate in the history the node τ that corresponds to the f -maximum on $H(N^0)$. This trivially takes $O(1)$ time. Let us denote this f -maximum by r . Let S_j be the half-space, if any, that kills τ (or equivalently r). Clearly, r is also present in $H(N^{j-1})$, because every half-space added before S_j contains it. It also must be the f -maximum on $H(N^{j-1})$, because the addition of a half-space does not disturb the maximum if it contains the maximum. The new f -maximum on $H(N^j)$ must lie on the bottom of $\text{cap}(S_j)$ (Figure 3.5(f)). Using the auxiliary search structure associated with the bottom of $\text{cap}(S_j)$,

³To be precise, the time bound is logarithmic only with high probability if the search structure is randomized. This will make no difference in what follows.

we can locate the node τ' in the history that corresponds to the f -maximum on $H(N^j)$. This takes logarithmic time. We repeat this process until we reach a leaf of $\text{history}(i)$. This leaf corresponds to the f -maximum on $H(N^i)$.

Analysis

The cost of the above search is proportional to the length of the search path, within a logarithmic factor. We will show that the length of the search path in $\text{history}(i)$, for any fixed query function, is $\tilde{O}(\log i)$. First, notice that each vertex in $H(N^i)$ is defined, i.e., completely determined, by three half-spaces in N^i . These are precisely the half spaces whose bounding planes contain that vertex. This means Lemma 3.1.5 is applicable because the bounded degree property required in its proof is satisfied (see the discussion after Lemma 3.1.6).

Thus, it follows that the search cost for any fixed query function is $\tilde{O}(\log i)$. It can be shown that the total number of distinct search paths in $\text{history}(i)$ is polynomial in i . We leave this as an exercise to the reader. Arguing as in the case of randomized binary trees (Observation 1.3.1), it then follows that the depth of our search structure is $\tilde{O}(\log i)$.

The expected size of $\text{history}(i)$ is $O(i)$. This is because the expected number of newly created nodes during each random addition is $O(1)$. This follows from Lemma 3.2.1.

Let us summarize the discussion so far.

Theorem 3.2.3 *The cost of locating the optimum of a linear function using $\text{history}(i)$ is $\tilde{O}(\log^2 i)$. The expected size of $\text{history}(i)$ is $O(i)$.*

Exercises

3.2.1 We assumed in this section that $H(N^i)$, for each i , is bounded. The simplest way to ensure this is to assume that N^0 contains half-spaces $x_i \leq \delta$ and $x_i \geq -\delta$, where $1 \leq i \leq d$, and δ is a large constant. There are two ways to handle δ :

- (a) Given N , calculate an upper bound on the absolute value of the coordinates of any vertex in $H(N^i)$, for any i . The bound should be in terms of n and the bit size of the input. Let δ be any constant greater than this value.
- (b) The above solution requires that N be known *a priori*. In an on-line algorithm, this assumption can be restrictive. Now let δ be a symbolic constant that stands for an arbitrarily large constant. Elaborate what this actually means as far as the implementation details are concerned. (This is the solution we used in this section.)

Give an alternative direct modification to the algorithms in this section so that the boundedness assumption is no longer necessary.

3.2.2 Modify the algorithms in this section so that they can handle all possible degeneracies. What happens when the query linear function is degenerate? Make

sure that the on-line algorithm in Section 3.2.2 still reports one optimum vertex correctly.

3.2.3 Prove that the number of distinct search paths in history(i) is bounded by a polynomial in i of fixed degree.

****3.2.4** Give an algorithm for constructing a convex polytope $H(N)$, for $d = 2, 3$, in expected $O(n \log m)$ time. Here m is the number of nonredundant half-spaces in N .

3.2.5 Translate the algorithms in this section in a dual setting. Given n points in R^3 , give dual incremental and on-line algorithms for constructing their convex hull. Show how the history can be used for answering ray shooting queries.

3.2.6 Construct an addition sequence of n half-spaces in R^3 so that the total structural change in the underlying convex polytope is $\Omega(n^2)$. Conclude that the running time of the on-line algorithm in this section can be $\Omega(n^2)$ in the worst case; thus, randomization is essential for its efficiency.

3.3 Voronoi diagrams

A Voronoi diagram in the plane can be obtained by projecting an appropriate three-dimensional convex polytope (Section 2.5). Hence, the algorithm in Section 3.2.2 yields a randomized on-line algorithm for constructing Voronoi diagrams in the plane. The expected running time of this algorithm is $O(n \log n)$, where n is the number of sites.

Let us now turn to the point location in Voronoi diagrams. This is an important problem because the nearest neighbor query problem can be reduced to it (Section 2.5). The history, as defined in Section 3.2.2, cannot be used for point location in Voronoi diagrams. This is because the point location problem for Voronoi diagrams does not get translated into the linear programming problem for convex polytopes. For this reason, we shall explicitly translate the randomized on-line algorithm in Section 3.2.2 so that it deals with Voronoi diagrams directly without transforming them to three-dimensional convex polytopes. Later we shall modify the algorithm further so that the history of the modified on-line algorithm can be used for point location in Voronoi diagrams. The dual Delaunay triangulations will be dealt with in the exercises.

So let N be a set of sites in the plane. Let $G(N)$ denote the Voronoi diagram formed by the sites in N . The randomized on-line algorithm in Section 3.2, when translated to deal with Voronoi diagrams directly, works as follows. It adds the sites in N one at a time in random order. Let N^i denote the set of the first i randomly added sites. At every stage i , the algorithm maintains the Voronoi diagram $G(N^i)$ and the history of the preceding additions. The Voronoi diagram $G(N^{i+1})$ is obtained from $G(N^i)$

as follows (Figure 3.6(a)–(b)). Let $S = S_{i+1}$ denote the $(i+1)$ th added site. First, we locate a vertex p of $G(N^i)$ which conflicts with S , i.e., which lies in the Voronoi region of S . This happens iff S is nearer to p than any of the sites labeling the regions in $G(N^i)$ adjacent to p . Clearly every such vertex p is destroyed during the addition of S . Note that several vertices in $G(N^i)$ can conflict with S . We just need to locate one. This can be done using the history at time i . We shall specify the details of this operation in a moment. Once one conflicting vertex—call it p —is located, all vertices conflicting with S are identified by a search within $G(N^i)$. This search starts at p . It works successfully because the conflicting vertices and the adjacent edges form a connected graph. The edges adjacent to the conflicting vertices are split or removed depending upon whether they are adjacent to one or two conflicting vertices. Finally, we also compute the edges of $\text{vor}(S)$, the Voronoi region of S . These connect the endpoints of the split edges in a cyclic order. The edges of $\text{vor}(S)$ are easy to compute because they are contained in the perpendicular bisectors of the segments joining S and the sites labeling the adjacent regions. At the end of this step, we have $G(N^{i+1})$ as desired.

It only remains to specify the history. It is a straightforward translation of the history in the case of convex polytopes (Section 3.2.2). The history at time i contains a unique node for every vertex that is created as a vertex of some Voronoi diagram $G(N^j)$, $j \leq i$. The link structure $\text{link}(S_{i+1})$ created

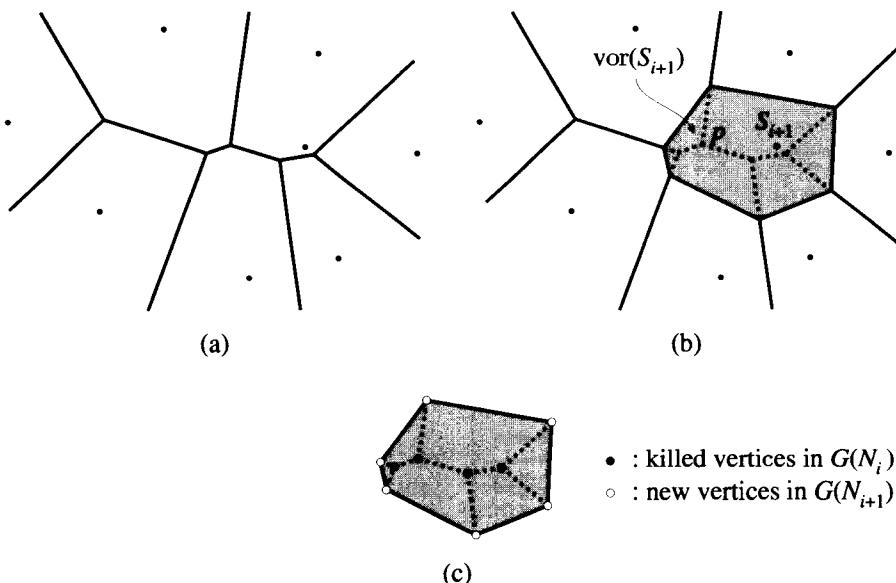


Figure 3.6: (a) $G(N^i)$. (b) $G(N^{i+1})$. (c) $\text{link}(S_{i+1})$.

during the above $(i+1)$ th addition corresponds to adjacencies among the new vertices on the boundary of $\text{vor}(S_{i+1})$ and the old vertices within $\text{vor}(S_{i+1})$ (Figure 3.6(c)).

The history is used in conflict search just as in the case of convex polytopes. Suppose we are given a site $I \notin N^i$. We want to locate a vertex of the Voronoi diagram $G(N^i)$ in conflict with it. First, we trivially find a node in the history that corresponds to a vertex of $G(N^0)$ in conflict with I . (Using the usual bounding box trick, we may assume that the single infinite region in $G(N^0)$ is a box approaching infinity.) Inductively, assume that we are at some node τ in the history that conflicts with I . Assume that τ is killed at time j . We want to descend from τ to a conflicting node created at time j . In other words, we want to find a node that corresponds to a vertex in $G(N^j)$ conflicting with I . This can be done using $\text{link}(S_j)$ as follows. First, notice that the nodes of $\text{link}(S_j)$ in conflict with I form its connected subgraph. (Why? This corresponds to the following connectivity property in the case of convex polytopes. The property is that $\text{cap}(S_{i+1}) \cap \bar{I}$ is connected, as shown in Figure 3.5(e).) The connectivity property enables us to locate a node that corresponds to a conflicting vertex in $H(N^j)$ by a simple search. We start the search at τ and restrict it to the above connected subgraph of $\text{link}(S_j)$. We stop the search as soon as we find a node σ that corresponds to a conflicting vertex in $G(N^j)$. Next, we descend from σ in the history in a similar fashion. We keep on repeating this process until we reach a leaf of the history at time i . This corresponds to a conflicting vertex in $G(N^i)$.

This finishes the description of our on-line algorithm for maintaining Voronoi diagrams. Since it is just a translation of the on-line algorithm for convex polytopes, it follows that its expected running time on a random sequence of n additions is $O(n \log n)$. Thus we have proved the following.

Theorem 3.3.1 *A Voronoi diagram of any n sites in the plane can be constructed in an on-line fashion in expected $O(n \log n)$ time.*

Next, we turn to point location in Voronoi diagrams. For the purpose of point location, we shall modify the above algorithm so that it maintains at time i not the Voronoi diagram $G(N^i)$ but, rather, a certain *radial triangulation* $H(N^i)$. The radial triangulation $H(N^i)$ is obtained by joining the vertices of every Voronoi region to the site labeling that region (Figure 3.6(d)). We shall call the regions in $H(N^i)$ triangles, though the unbounded regions are, strictly speaking, not triangles. What only matters is that $H(N^i)$ has the following bounded degree property, which $G(N^i)$ did not have: Every region Δ in $H(N^i)$ is defined by at most four sites in N^i . By a defining site, we mean a site whose Voronoi region is either adjacent to Δ or contains Δ . The bounded degree property is required to keep the depth of the history logarithmic (cf. the discussion after Lemma 3.1.6).

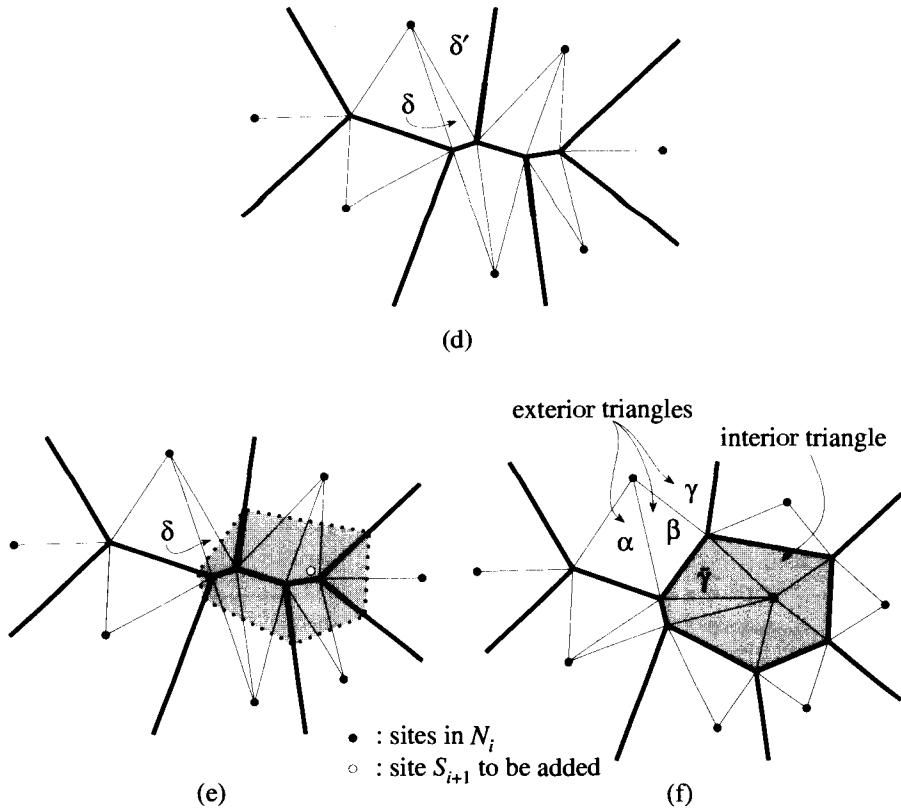


Figure 3.6: (continued) (d) $H(N^i)$. (e) Addition of S_{i+1} to $H(N^i)$. (f) $H(N^{i+1})$.

History(i) will now be the history of the preceding sequence of triangulations $H(N^0), H(N^1), \dots, H(N^i)$. Its leaves will point to the triangles in $H(N^i)$.

Let us reconsider the addition of the $(i+1)$ th site $S = S_{i+1}$. We already know how to update $G(N^i)$ to $G(N^{i+1})$. With minor modifications, which we shall leave to the reader, $H(N^i)$ can be updated to $H(N^{i+1})$ in the same order of time. Extending history(i) to history($i+1$) is more interesting. As usual, we mark the leaves of history(i) that correspond to the destroyed triangles in $H(N^i)$ as killed (dead). In Figure 3.6(e), all triangles intersecting the shaded region are marked as killed. We also allocate nodes for the newly created triangles in $H(N^{i+1})$. In Figure 3.6(f), these are the triangles that are adjacent to or are contained in the shaded region. These triangles can be classified in two types (Figure 3.6(f)): (1) the triangles that are interior to the new Voronoi region $\text{vor}(S)$ and (2) the triangles that are exterior to

$\text{vor}(S)$. Any Voronoi region adjacent to $\text{vor}(S)$ can contain at most three of the newly created exterior triangles. Given a destroyed triangle $\delta \in H(N^i)$ and a newly created triangle $\gamma \in H(N^{i+1})$, we say that γ is a *child* of δ if $\gamma \cap \delta$ is nonempty. If γ is an exterior triangle, we say that it is an exterior child of δ ; otherwise, it is called an interior child of δ . In Figure 3.6(f), α, β , and γ are exterior children of δ , and $\bar{\gamma}$ is an interior child of δ . A destroyed triangle in $H(N^i)$ can have at most three exterior children. On the other hand, the number of its interior children can be arbitrarily large. For this reason, we cannot afford to associate with a leaf of history(i) pointers to all its children among the newly created leaves of history($i+1$). We shall associate pointers only to its exterior children.

We also link all leaves for the newly created interior triangles in $H(N^{i+1})$ in a circular fashion around S . We build a search tree (using either a skip list or a randomized binary tree) that corresponds to the clockwise angular ordering of this circular list. This search tree will be called the *radial tree* associated with S . Given any point p lying in the region $\text{vor}(S)$ in $H(N^{i+1})$, one can now determine the interior triangle in $\text{vor}(S)$ containing p in logarithmic time.⁴ This can be done in an obvious fashion using the radial tree.

To summarize, $\text{link}(S_{i+1})$, the link structure created at time $i+1$, consists of (1) the pointers from the killed leaves of history(i) to their exterior children and (2) the radial tree for the newly created nodes of history($i+1$).

Let us turn to point location. Let p be any query point. We want to locate the triangle in $H(N^i)$ containing p . We begin at the root of history(i), which corresponds to the single infinite region in $H(N^0)$, and descend to its child containing the query point. We keep repeating this process until we reach a leaf of the history that corresponds to the triangle in $H(N^i)$ containing p . Let us elaborate the descent operation a bit more. Suppose we are given a node in the history that corresponds to a triangle δ containing p that is killed by a site S_j , $j \leq i$. How does one descend to the child of Δ containing p ? This can be done using $\text{link}(S_j)$ as follows. If an exterior child of δ contains p , we are lucky: δ has pointers to all of its exterior children (at most three in number). Otherwise, the interior child of Δ containing p can be located using the radial tree associated with S_j in logarithmic time.

Analysis

It follows from the preceding discussion that the cost of point location is proportional to the length of the search path, up to a logarithmic factor. Because radial triangulations have the bounded degree property, Lemma 3.1.5 becomes applicable. It implies that, for a fixed query point, the length of the search path in history(i) is $\tilde{O}(\log i)$. It can be shown that the number of

⁴With high probability, strictly speaking; but this will make no difference in what follows.

distinct search paths in history(i) is bounded by a polynomial of fixed degree in i . We leave this as an exercise to the reader. Arguing as in the case of randomized binary trees (Observation 1.3.1), it then follows that the depth of history(i) is $\tilde{O}(\log i)$. Hence, the cost of point location using history(i) is $\tilde{O}(\log^2 i)$. The nearest neighbor in N^i to the query point p is the one labeling the Voronoi region containing p . Hence, we have proved the following.

Theorem 3.3.2 Nearest neighbor queries can be answered using history(i) in $\tilde{O}(\log^2 i)$ time.

Exercises

3.3.1 Show that the number of distinct search paths in history(i) is bounded by a polynomial in i of a fixed degree.

3.3.2 Prove that the expected size of history(i) is $O(i)$. (Hint: Prove an analogue of Lemma 3.2.1.)

3.3.3 Give the dual algorithm for constructing Delaunay triangulations in the plane. Specify the link structures in the history in detail.

3.3.4 Construct an addition sequence of n sites in the plane so that the total structural change in the underlying Voronoi diagram is $\Omega(n^2)$. Conclude that the running time of the on-line algorithm in this section can be $\Omega(n^2)$ in the worst case; thus, randomization is essential for its efficiency.

3.4 Configuration spaces

The reader must have noticed that the incremental algorithms in this chapter have a lot in common. We want to distill these common principles so that they can be readily used in diverse applications later. This will be done using a general framework of certain combinatorial systems called *configuration spaces*. The same framework will be used in later chapters to develop other common principles in the design of randomized algorithms, such as randomized divide-and-conquer.

Let us assume that we are given an abstract set (universe) N of objects. The objects depend on the problem under consideration. For example, they are segments when we are dealing with trapezoidal decompositions and half-spaces when we are dealing with convex polytopes.

A configuration σ over N is a pair $(D, L) = (D(\sigma), L(\sigma))$, where D and L are disjoint subsets of N . The objects in D are called the *triggers* associated with σ . They are also called the objects that *define* σ . We also say that σ is *adjacent* to the objects in $D(\sigma)$. The objects in L are called the *stoppers* associated with σ . They are also called the objects that *conflict* with σ . The *degree* $d(\sigma)$ is defined to be the cardinality of $D(\sigma)$. The *level* or the (*absolute*) *conflict size* $l(\sigma)$ is defined to be the cardinality of $L(\sigma)$. (Thus

“d” stands for degree and “l” for level.) A configuration space $\Pi(N)$ over the universe N is just a set of configurations over N with the following property:

Bounded degree property: The degree of each configuration in $\Pi(N)$ is bounded (by a constant).

The reader probably recalls that the same property turned out to be crucial on several earlier occasions (Sections 3.1.1, 3.2.2, and 3.3). We shall allow $\Pi(N)$ to be a multiset. This means that several “distinct” configurations in $\Pi(N)$ can have the same trigger and stopper set. By the *size* of $\Pi(N)$, we mean the total number of elements in this multiset. Here the distinct configurations having the same trigger and stopper sets are counted separately. The number that is obtained when the configurations with the same trigger and stopper set are not counted separately is called the *reduced size* of $\Pi(N)$. We denote the size of $\Pi(N)$ by either $|\Pi(N)|$ or $\pi(N)$. The reduced size is denoted by $\tilde{\pi}(N)$. For each integer $i \geq 0$, we define $\Pi^i(N)$ to be the set of configurations in $\Pi(N)$ with level i . We denote its size by $\pi^i(N)$. The configurations in $\Pi^0(N)$ are said to be *active* over N . The configurations in $\Pi^i(N)$ are said to be *partially active* over N with level i .

Example 3.4.1 (Trapezoidal decompositions)

Let N be a set of segments in the plane (Figure 3.7). Let us call a trapezoid σ in the plane *feasible* or *possible* over N if σ occurs in the trapezoidal decomposition $H(R)$, for some subset $R \subseteq N$. Note that the trapezoids that can possibly arise in the course of the incremental construction of $H(N)$ (Section 3.1) are all feasible. Conversely, each feasible trapezoid can actually arise in the incremental computation, if one were to add the segments in R before the remaining segments. Thus, feasible trapezoids are precisely the ones that can possibly arise in the course of incremental computation of $H(N)$.

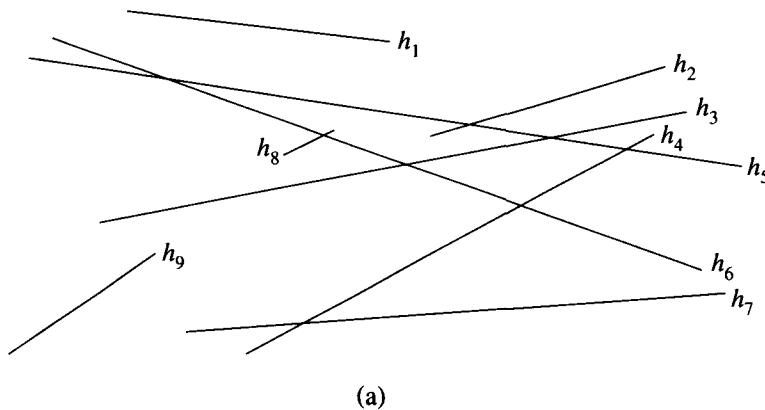
For a feasible trapezoid σ , we define the trigger set $D(\sigma)$ to be the set of segments in N that are adjacent to the boundary of σ . We define the conflict set $L(\sigma)$ to be the set of segments in $N \setminus D(\sigma)$ intersecting σ . For example, in Figure 3.7, the trigger set of the feasible trapezoid σ is $\{h_3, h_5, h_7\}$. The stopper set is $\{h_4, h_6\}$. In this fashion, we can associate an abstract configuration $(D(\sigma), L(\sigma))$ with every feasible trapezoid σ . It is easily seen that the size of $D(\sigma)$ is less than or equal to four, assuming that the segments in N are in general position. Thus, we get a configuration space $\Pi(N)$ of all feasible trapezoids over N . The general position assumption is crucial here, as well as in all other examples, to ensure bounded degree property.

Note that the definition of a conflict set given above is different from the definition of a conflict list used in the algorithm of Section 3.1. There, for the sake of speed, we only maintained conflicts of the endpoints of the unadded

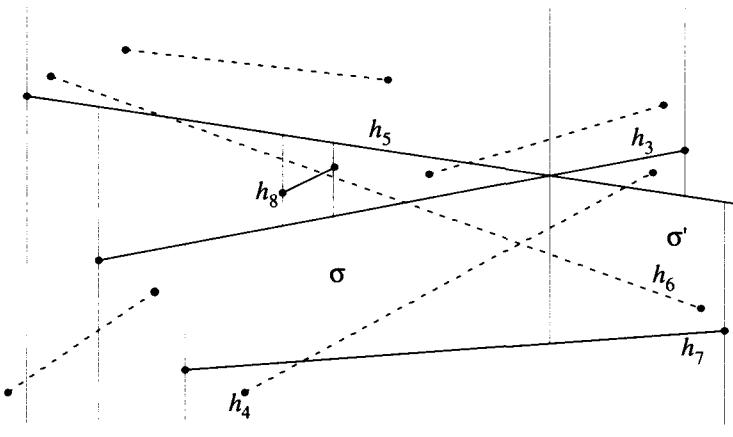
segments. In the definition of a configuration space, it is necessary to define the conflict sets as above. We shall soon see why.

Also note that $\Pi(N)$ is actually a multiset. For example, the feasible trapezoids σ and σ' in Figure 3.7(b) are associated with the same trigger and stopper sets. Also note that the configurations in $\Pi^0(N)$ are precisely the trapezoids in the trapezoidal decomposition $H(N)$.

Let us return to the abstract setting and make a few more definitions. Let $R \subseteq N$ be a subset of N . We define the subspace $\Pi(R)$ as follows. Given a configuration $\sigma \in \Pi(N)$ such that $D(\sigma) \subseteq R$, we define the restriction $\sigma \downarrow R$



(a)

Figure 3.7: (a) A set N of segments. (b) $H(R)$.

to be the configuration over the universe R whose trigger set is $D(\sigma)$ and whose conflict set is $L(\sigma) \cap R$. We let

$$\Pi(R) = \{\sigma \downarrow R \mid \sigma \in \Pi(N), D(\sigma) \subseteq R\}.$$

Note that two distinct configurations in $\Pi(N)$ may restrict to the configurations with the same trigger and stopper sets in $\Pi(R)$. But these two restricted configurations have to be considered different. For this reason, it is important to consider configuration spaces as multisets.

Note that the restriction $\sigma \downarrow R$ is defined only when R contains the triggers associated with σ . As a misuse of notation, we shall generally refer to the restriction $\sigma \downarrow R$ by σ itself. Thus, when we say that $\sigma \in \Pi(N)$ is *active* over R what we actually mean is that $\sigma \downarrow R$ belongs to $\Pi^0(R)$. This happens iff R contains all triggers associated with σ but none of the stoppers. The misuse of notation should cause no confusion as long as one distinguishes between the two conflict lists: the conflict list relative to N , namely, $L(\sigma)$, and the conflict list relative to R , namely, $L(\sigma) \cap R$. The size of the first list is called the *conflict size* of σ relative to N and the size of the latter is called the *conflict size* of σ relative to R . Thus, the conflict size of a configuration in $\Pi(R)$ is always relative to a set containing R . If we do not mention this set, it is generally clear from the context. As a convention, we always use the alternative word for the conflict size—namely, *level*—in the following restrictive sense: When we refer to the level of a configuration in $\Pi(R)$, we always mean its conflict size relative to R .

For example, consider the previous example of segments in the plane. In Figure 3.7, the conflict set of the feasible trapezoid σ relative to N is $\{h_4, h_6\}$. Hence, its conflict size relative to N is 2. The conflict list of σ relative to R is empty. Hence, its conflict size relative to R is zero. In other words, σ considered as a configuration in $\Pi(R)$ has zero level. The set $\Pi^0(R)$ consists of the configurations in $\Pi(R)$ with zero level. These correspond to the trapezoids in $H(R)$, the trapezoidal decomposition formed by R . Moreover, the conflict size of any trapezoid in $H(R)$ relative N is simply the number of lines in $N \setminus R$ which intersect that trapezoid.

Remark. The trapezoids in $\Pi^0(R)$ would not coincide with the ones in $H(R)$ if we were to consider only the endpoint conflicts as in Section 3.1. That is because a segment can intersect a trapezoid, even if none of its endpoints is contained in the trapezoid.

We shall now give some more examples of configuration spaces. These are the configuration spaces that underlie the remaining randomized incremental algorithms in the previous sections. As per our usual assumption, the objects are always assumed to be in general position.

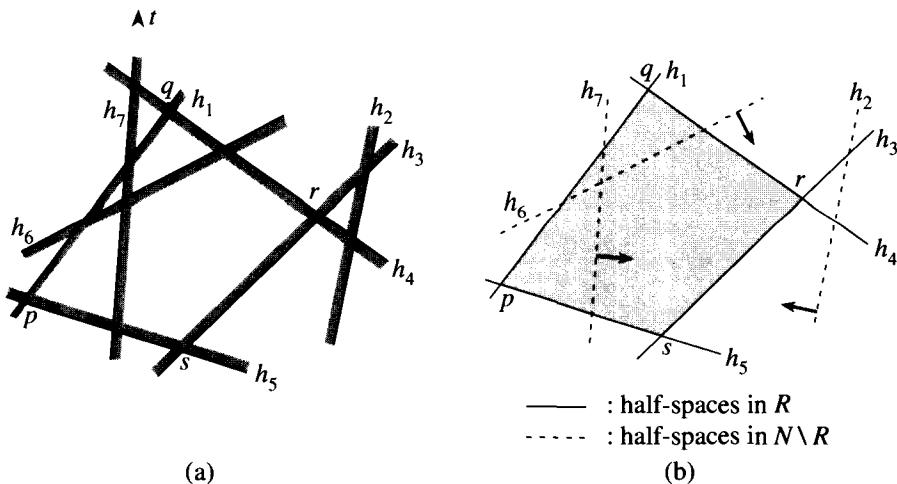


Figure 3.8: (a) A set N of half-spaces. (b) $R = \{h_1, h_4, h_5\}$.

Example 3.4.2 (Convex polytopes)

Let N be a set of half-spaces in \mathbb{R}^d (Figure 3.8). Let $G(N)$ be the arrangement formed by the hyperplanes bounding these half-spaces. We can associate an abstract configuration with each vertex σ of $G(N)$ as follows (Figure 3.8(a)). Let the trigger set $D(\sigma)$ be defined as the set of half-spaces in N whose bounding hyperplanes contain σ . The size of $D(\sigma)$ is clearly d , assuming that the half-spaces are in general position. Let the conflict set $L(\sigma)$ be the set of half-spaces in N whose complements contain σ . In Figure 3.8(a), the trigger set of the vertex q is $\{h_1, h_4\}$. Its stopper set is $\{h_6\}$.

In this fashion, we obtain a configuration space $\Pi(N)$ of the vertices in the arrangement $G(N)$. We shall find it convenient to let $\Pi(N)$ contain the “vertices” of $G(N)$ that lie at infinity. By this, we mean the endpoints at infinity of the unbounded edges (1-faces) of $G(N)$. In Figure 3.8(a), t is one such vertex at infinity. Its trigger set is $\{h_7\}$ and the conflict set is $\{h_1, h_4, h_6\}$.

Observe that, for every $R \subseteq N$, a vertex σ in $\Pi(N)$ is active (as a configuration) over R , i.e., it occurs in $\Pi^0(R)$, iff it is a vertex of the convex polytope $H(R)$ formed by intersecting the half-spaces in R . Thus, $\Pi(N)$ consists of all vertices that can possibly occur in the incremental computation of $H(N)$.

Example 3.4.3 (Voronoi diagrams)

Let N be a set of sites in the plane (Figure 3.9). Let $G(N)$ denote the Voronoi diagram formed by N . Let $H(N)$ denote its radial triangulation

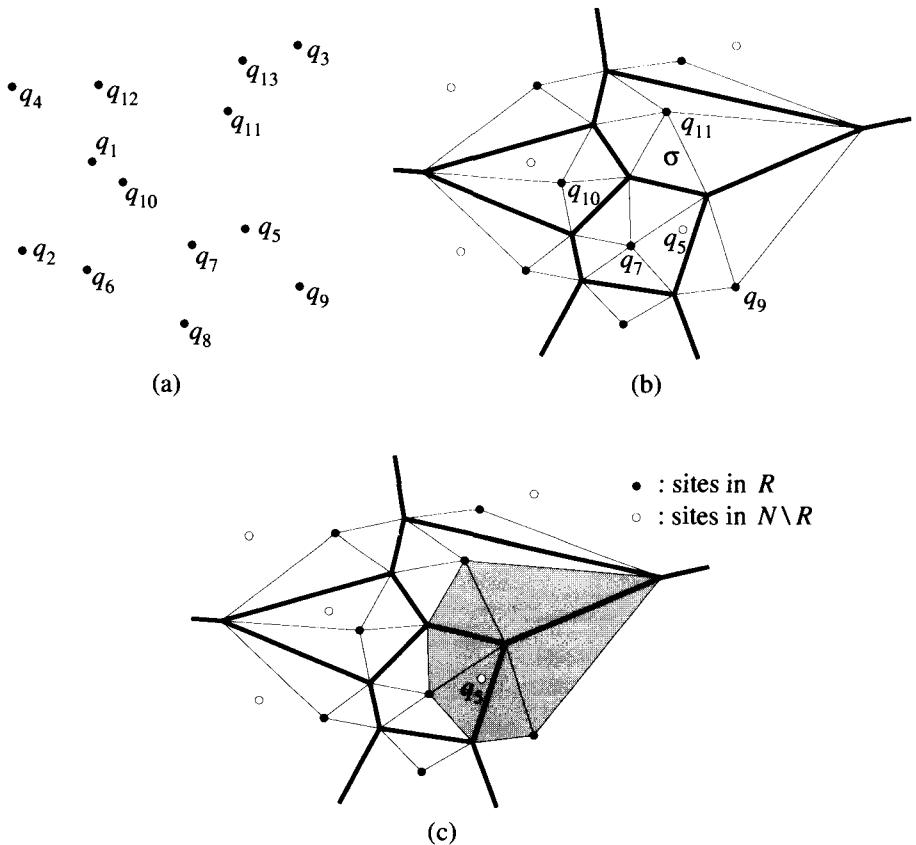


Figure 3.9: (a) A set N of sites. (b) Radial triangulation $H(R)$ for a subset R . (c) Triangles in $H(R)$ conflicting with q_5 (shown shaded).

(Section 3.3). Let σ be a triangle or an edge or a vertex in the plane. We say that σ is *feasible* over N if it occurs in the radial triangulation $H(R)$, for some $R \subseteq N$. For example, the triangle σ in Figure 3.9(b) is feasible. Feasible triangles are precisely the ones that can arise in the incremental computation of $H(N)$. A site $q \in N \setminus R$ is said to be in *conflict* with a triangle $\tau \in H(R)$ if the imaginary addition of q to R will destroy τ . What this means is that τ does not occur in $H(R \cup \{q\})$. This happens iff q is nearer to some point within τ than the site labeling the Voronoi region containing τ (Figure 3.9(c)). For any feasible triangle σ in $H(R)$, let the trigger set $D(\sigma)$ be defined as the set of sites in R that label the regions of $G(R)$ containing or adjacent to σ . Let the conflict set $L(\sigma)$ be defined as the set of sites in $N \setminus R$ in conflict with σ . For example, the trigger set of the triangle σ in

Figure 3.9(b) is $\{q_{11}, q_{10}, q_7, q_9\}$. It is easily seen that $D(\sigma)$ and $L(\sigma)$ do not depend on R . In this fashion, we can associate an abstract configuration $(D(\sigma), L(\sigma))$ with every feasible triangle σ .

It is also easily seen that a feasible triangle σ is active (as a configuration) over a subset $R \subseteq N$, i.e., it occurs in $\Pi^0(R)$, iff σ occurs in the radial triangulation $H(R)$. In a similar fashion, we can associate a trigger set and a conflict set with every feasible edge or a vertex over the universe N . It is clear that the size of the trigger set is always bounded, assuming that the sites in N are in general position. In this fashion, we get a configuration space $\Pi(N)$ of feasible triangles, edges, and vertices over N .

Example 3.4.4 (Trapezoidal decompositions again)

Let N be again a set of segments in the plane. Let $H(N)$ denote the resulting trapezoidal decomposition. We shall now define another configuration space over N . It is akin to the configuration space of feasible trapezoids over N (Example 3.4.1).

Define a racquet in a trapezoidal decomposition $H(M)$ as a pair $\tau = (\sigma, e)$, where σ is a trapezoid in $H(M)$, and e is a vertical attachment that is adjacent to σ (Figure 3.10). We say that σ is the face of the racquet. It could be possibly unbounded. The attachment e is called the handle. Figure 3.10 shows a racquet in $H(R)$, for a subset $R \subseteq N$.

Now let N be the given set of segments in the plane. Define a *feasible racquet* τ over N as a racquet that occurs in a trapezoidal decomposition

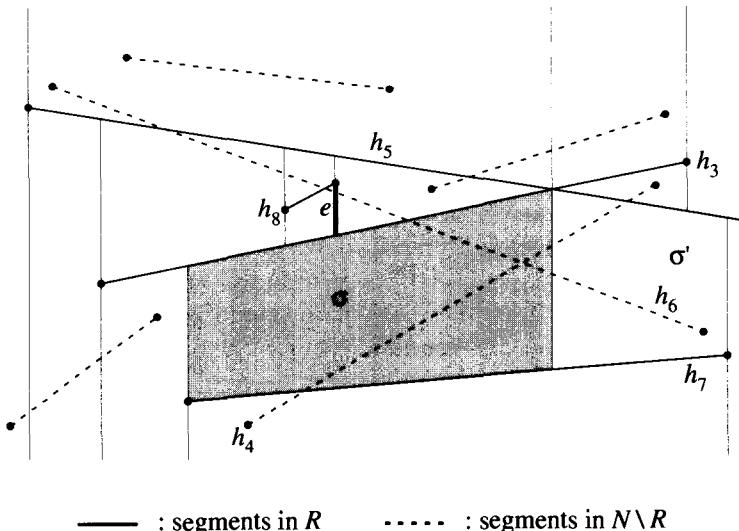


Figure 3.10: A racquet.

$H(R)$, for some $R \subseteq N$. We define the trigger set $D(\tau)$ to be the set of segments in N that define τ , i.e., the segments in N that are adjacent to the face or the handle of τ . We define the stopper set $L(\tau)$ to be the set of remaining segments in N that intersect this racquet, i.e, either its face or its handle. For example, in Figure 3.10, the trigger set of the racquet $\tau = (\sigma, e)$ is $\{h_3, h_7, h_5, h_8\}$. The stopper set is $\{h_4, h_6\}$.

The degree of any racquet is clearly bounded. In this fashion, given a set of segments N , we get a configuration space $\Sigma(N)$ of feasible racquets over N . This configuration space is naturally related to the incremental algorithm in Section 3.1 for constructing trapezoidal decompositions. We saw in Section 3.1 that, ignoring the cost of endpoint location, the cost of adding a new segment S to $H(N^i)$ is proportional to $\sum_f \text{face-length}(f)$, where f ranges over all destroyed trapezoids in $H(N^i)$. It is easy to see that $\text{face-length}(f)$ is essentially equal to the number of racquets in $H(N^i)$ with face f . Hence, the above cost is also proportional to the number of racquets in $H(N^i)$ that are destroyed during the addition of a new segment.

Note that, for any subset $R \subseteq N$, the active racquets over R , i.e., the racquets in $\Sigma^0(R)$ correspond to the racquets in $H(R)$. The total number of such racquets is proportional to the size of $H(R)$. Let us see why. By the previous observation, the number of racquets in $H(R)$ is essentially $\sum_f \text{face-length}(f)$, where f ranges over all trapezoids in $H(R)$. This in turn is proportional to the size of $H(R)$. That is because each vertex in $H(R)$ is counted a bounded number of times, once for each face that it is adjacent to. It follows that the size of $\Sigma^0(R)$ is $O(|H(R)|)$.

Now we return to abstract configuration spaces. Let $\Pi(N)$ be any configuration space. A randomized incremental algorithm, of the kind considered in this chapter, can be expressed in the following abstract form.

1. Construct a random sequence (permutation) S_1, S_2, \dots, S_n of the objects in N .
2. Add the objects in N one at a time according to this sequence. Let N^i denote the set of the first i added objects. At the i th stage, the algorithm maintains the set $\Pi^0(N^i)$ of active configurations. For each configuration $\sigma \in \Pi^0(N^i)$, we also maintain its conflict set $L(\sigma)$ relative to N . Its size $l(\sigma)$ is the conflict size of σ relative to N .

By destroyed configurations at time i , we mean the configurations that were present in $\Pi^0(N^{i-1})$ but not in $\Pi^0(N^i)$. By newly created configurations at time i , we mean the configurations that are present in $\Pi^0(N^i)$ but not in $\Pi^0(N^{i-1})$. Let us define the *structural change* at time i as the number of newly created and destroyed configurations at time i . Let us define the *conflict change* at time i as the sum of the conflict sizes (relative to N) of the newly created and destroyed configurations at time i .

The sum of the structural changes at time i , over all i , is called *the total structural change* during the preceding incremental construction. The *total conflict change* is defined similarly. As we have seen in this chapter, the performance of a randomized incremental construction depends on the expected values of the total structural and conflict change: The total structural change bounds the size of the history, and the total conflict change bounds the running time of the algorithm. The following theorem bounds these values in a general setting.

For each $j \leq n$, let $e(j)$ denote the expected size of $\Pi^0(N^j)$, assuming that the N^j is a random sample (subset) of N of size j . By this, we mean that each element in N is equally likely to be present in N^j . Let $d = d(\Pi)$ denote the maximum degree of a configuration in $\Pi(N)$. By the definition of a configuration space, d is bounded by a constant.

Theorem 3.4.5 *The expected value of the total structural change in the randomized incremental construction is bounded by $\sum_j d e(j)/j$. (Intuitively, the total structural change measures the size of the history of this construction.) The expected value of the total conflict change is bounded by $\sum_j d^2[(n-j)/j](e(j)/j)$.*

Proof. Lemma 3.2.1 and (3.7) prove the theorem for the configuration space of vertices described in Example 3.4.2. A careful examination of the proofs of Lemma 3.2.1 and (3.7) will reveal that they only use the following fact regarding this configuration space of vertices: Every vertex is adjacent to at most d half-spaces. By definition, the configuration space $\Pi(N)$ has the analogous bounded degree property: Every configuration in $\Pi(N)$ is adjacent to at most d objects in N . Hence, the proofs of Lemma 3.2.1 and (3.7) can be translated to the present general setting *verbatim*: Just replace the words “vertices” and “half-spaces” by the words “configurations” and “objects,” respectively. \square

We have already seen several implicit applications of this theorem to the randomized incremental algorithms in the previous sections. We end with one trivial application.

Example 3.4.6 (Trivial configuration space)

We say that a configuration space $\Pi(N)$ over a given object set N is *trivial*, if for every subset $M \subseteq N$, the number of configurations in $\Pi(N)$ active over M is $O(1)$. We give a simple example of a trivial configuration space. Let $\Sigma(N)$ be the configuration space of feasible trapezoids in Example 3.4.1. We shall show that, for every fixed point p in the plane, $\Sigma(N)$ contains a natural trivial configuration space defined as follows. Let $\Sigma_p(N)$ be the subspace of all feasible trapezoids in $\Sigma(N)$ that contain p . It is clear that $\Sigma_p(N)$ is

trivial. This is because, for any $M \subseteq N$, exactly one trapezoid in $H(M)$ can contain p .

When the configuration space is trivial, the bound in Theorem 3.4.5 for the expected structural change becomes $d \sum_j O(1/j) = O(\log n)$. What does this mean? Consider the trivial configuration space $\Sigma_p(N)$ of feasible trapezoids containing a fixed query point p . The structural change over this configuration space is equal to the number of trapezoids created in the course of the algorithm that contain p . This is nothing but the length of the search path for p in the history (Section 3.1.1). We had already seen that its expected length is $O(\log n)$ (Lemma 3.1.5). In fact, the specialization of Theorem 3.4.5 to trivial configurations spaces is the same as Lemma 3.1.5 in its generalized form (see the discussion after Lemma 3.1.6). (Lemma 3.1.5 actually gives the stronger high-probability bound.)

Exercises

3.4.1 Prove the following complement to Theorem 3.4.5. Show that the expected value of the total structural change in the randomized incremental construction is bounded below by $\sum_j e(j)/j$. (Hint: The expected number of newly created configurations in the i th addition is bounded below by $e(j)/j$.) Also show that the expected value of the total conflict change is bounded below by $\sum_j [(n-j)/j](e(j)/j)$.

3.4.2 In the randomized incremental construction of trapezoidal decompositions, we only maintained conflicts of the unadded endpoints rather than the whole segments. Give an algorithm that maintains conflict lists as defined in Example 3.4.1. Show that its expected running time is still $O(k + n \log n)$, where k is the number of intersections. (Hint: Bound the new expected conflict change by applying Theorem 3.4.5 to the configuration space of feasible trapezoids over N (Example 3.4.1). We have already seen in Section 3.1 that $e(r)$, the expected size of $H(N^r)$, is $O(r + (r/n)^2 k)$.)

***3.4.3** For $c \geq 0$, define the c th order conflict change at time j as $\sum_\sigma l^c(\sigma)$, where σ ranges over the newly created and destroyed configurations at time j . Generalize Theorem 3.4.5 to get a bound on the expected value of the total c th order conflict change.

3.5 Tail estimates*

Theorem 3.4.5 estimates the expected structural change in a randomized incremental construction. In this section, we shall give a high-probability bound for the same. Obtaining a high-probability bound for the total conflict change seems more difficult.

Let $\Pi(N)$ be any configuration space. Consider an abstract incremental construction as in Theorem 3.4.5. Let $X(N)$ denote the total number of configurations that get created during a random N -sequence of additions.

By an N -sequence of additions, we mean a sequence involving additions of the objects in N . The total structural change during this sequence is at most two times $X(N)$. This is because a created configuration can be destroyed at most once. So it suffices to bound $X(N)$. We can express it as

$$X(N) = \sum_{i=1}^n X_i, \quad (3.8)$$

where X_i is the number of configurations that get created during the i th addition. In other words, X_i is the number of configurations in $\Pi^0(N^i)$ that are adjacent to S_i . We have seen that the expected value of X_i , conditional on a fixed N^i , is bounded by $d\pi^0(N^i)/i$ (cf. Lemma 3.2.1 and the proof of Theorem 3.4.5). Suppose we are given a function μ such that $d\pi^0(I) \leq \mu(i)$, for every subset $I \subseteq N$ of size i . (For example, in the case of planar Voronoi diagrams (Example 3.4.3), we can let $\mu(i) = bi$ for a large enough constant b .) Then it is clear that

$$E[X(N)] \leq \sum_i \mu(i)/i.$$

We shall be interested in two special cases:

1. Suppose $\mu(i)$ is bounded by a constant μ . This happens in the case of trivial configurations spaces described before. In this case, the expected value of $X(N)$ is bounded by μH_n , where $H_n = \sum_i 1/i$ is the n th harmonic number.
2. Suppose we can choose $\mu(i)$ so that $\mu(i)/i$ is a nondecreasing function of i . In this case, the expected value of $X(N)$ is bounded by $n(\mu(n)/n) = \mu(n)$. For example, this is the case for planar Voronoi diagrams (Example 3.4.3) for the choice of $\mu(i)$ described previously.

For these two special cases, we shall be able to show that $X(N)$ deviates from the corresponding bound on its expected value with a very low probability.

Theorem 3.5.1

1. If $\mu(i)$ is bounded by a constant μ , then

$$\text{prob}\{X(N) \geq c\mu H_n\} \leq e^{-H_n(1+c\ln[c/e])}, \text{ for } c > 1.$$

2. If $\mu(i)/i$ is a nondecreasing function of i , then

$$\text{prob}\{X(N) \geq c\mu(n)\} \leq \frac{1}{e} \left(\frac{e}{c}\right)^c, \text{ for } c > 1.$$

We shall prove the theorem using the Chernoff Technique (Appendix A.1). The first step in this technique is to estimate the characteristic function of

$X(N)$. This is defined as the expected value of $e^{tX(N)}$ for a parameter $t \geq 0$. The following lemma estimates this expected value.

Lemma 3.5.2 $E[e^{tX(N)}]$ is bounded by $p(n) = \prod_{i=1}^n [1 + \frac{1}{i}(e^{t\mu(i)} - 1)]$.

Proof. We use induction on the size of N . Let $S = S_n$ denote the object involved in the n th addition. Let $x(N, S)$ denote the number of configurations in $\Pi^0(N)$ adjacent to S . Clearly, $X_n = x(N, S)$. Each object in N is equally likely to be S . Also, $X(N) = X(N \setminus \{S\}) + x(N, S)$, by (3.8). Hence, for a random addition sequence,

$$E[e^{tX(N)}] = \frac{1}{n} \sum_{S \in N} e^{tx(N, S)} E[e^{tX(N \setminus \{S\})}].$$

By the induction hypothesis, this is bounded by

$$p(n-1) \frac{1}{n} \sum_{S \in N} e^{tx(N, S)}. \quad (3.9)$$

We have already seen that

$$\sum_{S \in N} x(N, S) \leq d\pi^0(N) \leq \mu(i).$$

This is because each configuration in $\Pi^0(N)$ is adjacent to at most d objects. Hence, the above power sum is maximized when $x(N, S) = \mu(i)$, for some S , and zero for the remaining objects. In other words,

$$\frac{1}{n} \sum_{S \in N} e^{tx(N, S)} \leq \frac{1}{n} (e^{t\mu(n)} + n - 1) = 1 + \frac{1}{n} (e^{t\mu(n)} - 1).$$

Hence, the lemma follows from the bound in (3.9). \square

Notice that the bound in the above lemma coincides with a characteristic function for a generalized harmonic random variable (see (A.10) in Appendix A.1.3). Hence, Theorem 3.5.1 follows by arguing just as in Appendix A.1.3.

Corollary 3.5.3 If $\mu(i)/i$ is a nondecreasing function of i , then for almost all N -sequences of additions, the total structural change is $O(\mu(n) \log n)$. More precisely: Fix a large enough constant $a > 0$. For all but $(1/n^a)$ th fraction of the addition sequences, the total structural change is bounded by $a \mu(n) \ln n$.

Proof. Let $c = a \ln n$, in Theorem 3.5.1.2. \square

We shall now give a few applications of the above tail estimate.

Example 3.5.4 (Nonintersecting segments in the plane)

Suppose N is a set of n nonintersecting segments in the plane. We allow sharing of the endpoints among these segments. Thus, the segments in N form a planar graph. Consider the performance of the on-line algorithm in Section 3.1.1 on an N -sequence of additions. Let $\Pi(N)$ be the configuration space of feasible racquets over N , as in Example 3.4.4. We have seen in that example that the cost of each addition is proportional to the total number of racquets that are newly created (or destroyed) in that addition. This ignores $\tilde{O}(\log n)$ cost of endpoint location. Let us apply Corollary 3.5.3 to the configuration space $\Pi(N)$. We can let $\mu(i) = bi$, for a large enough constant b . This is because the size of a trapezoidal decomposition formed by any i nonintersecting segments is $O(i)$. It follows that:

Theorem 3.5.5 *Let N be any set of nonintersecting segments in the plane.⁵ The running time of the on-line algorithm in Section 3.1.1 is $\tilde{O}(n \log n)$ on almost all N -sequences of additions. The cost of point location at time i is $\tilde{O}(\log i)$.*

Example 3.5.6 (Point location in planar Voronoi diagrams)

Let N be a set of sites in the plane. Consider the on-line point location structure for Voronoi diagrams described in Section 3.3. We have already seen that its running time is proportional to the total structural change in the underlying radial triangulation $H(N^i)$. This ignores the cost of locating a vertex of $H(N^i)$ in conflict with the added site $S = S_{i+1}$. In Section 3.3, we located such a vertex by conflict search through the history. Since it seems difficult to bound the cost of conflict search with high probability, let us apply another method for the same purpose. Conceptually, this alternative method is even simpler. We simply locate a triangle in $H(N^i)$ containing S_{i+1} . This can be done using the point location structure that we are maintaining. At least one vertex of this triangle must be in conflict with S_{i+1} . This takes $\tilde{O}(\log^2 i)$ time (Section 3.3).

It only remains to estimate the structural change in the underlying radial triangulation. This can be done by applying Corollary 3.5.3 to the configuration space of feasible triangles over N (Example 3.4.3). We can let $\mu(i) = bi$, for a large enough constant b . This is because the size of the radial triangulation formed by any i sites is $O(i)$. It follows that:

Theorem 3.5.7 *The cost of maintaining the on-line point location structure for Voronoi diagrams, as in Section 3.3, over a sequence of n additions is $\tilde{O}(n \log^2 n)$. The cost of point location at time i is $\tilde{O}(\log^2 i)$.*

⁵Sharing of the endpoints is allowed (Exercise 3.1.1).

Exercises

†3.5.1 Derive an optimal high-probability bound for the total conflict change during an incremental construction in the general setting of configuration spaces.

3.5.2 Prove $\tilde{O}(n \log^2 n)$ high-probability bound on the running times of the algorithms in Section 3.2.

Prove $\tilde{O}(n \log n)$ bound on the size of $\text{history}(n)$. Can you improve this bound?

****3.5.3** Show that the running time of the trapezoidal decomposition algorithm in Section 3.1 is $O(k + n \log n)$ with high probability, if $k > cn \log n \log^{(3)} n$, for a large enough constant c .

Bibliographic notes

Incremental geometric algorithms have been favorite in practice for a long time because of their simplicity. But it was observed only later that a randomized incremental algorithm can sometimes be (provably) efficient; for example, see Clarkson and Shor [72] and Mulmuley [160]). In the recent years, the paradigm has been extensively studied by several authors, e.g., Boissonnat et al. [25], Boissonnat and Teillaud [27], Chazelle et al. [46], Clarkson et al. [70], Guibas et al. [117], Mehlhorn et al. [156], Mulmuley [161, 165, 168, 169], Schwarzkopf [200], Seidel [205, 206, 207], and Welzl [225].

The randomized incremental algorithm for trapezoidal decompositions in Section 3.1 and Exercise 3.1.2 is from Mulmuley [160], with minor variations, and its on-line version along with the associated planar point location structure (Section 3.1.1) is due to Seidel [206]. Another randomized incremental algorithm for trapezoidal decomposition (Exercise 3.4.2) was independently given by Clarkson and Shor [72]. Independently, Chazelle and Edelsbrunner [43] have given an optimal deterministic algorithm for the same problem. For the earlier work on trapezoidal decompositions, see, e.g., [21, 36]. Exercises 3.1.3 and 3.1.4 are from [163].

The randomized algorithm for triangulating simple polygons in Section 3.1.3 is due to Seidel [206]. See [73] for an earlier randomized algorithm for simple polygons, and [80] for more related work. Chazelle [41] gives an optimal $O(n)$ time deterministic algorithm for triangulating simple polygons. Earlier, Tarjan and Van Wyk [215] had given an $O(n \log \log n)$ time algorithm, breaking the $O(n \log n)$ barrier attained in [110]; see also [128].

The randomized, incremental algorithm for convex hulls in Section 3.2 as well as the output-sensitive algorithm in Exercise 3.2.4 are due to Clarkson and Shor [72]. Earlier, a deterministic $O(n \log n)$ algorithm for constructing 3D convex hulls was given by Preparata and Hong [184]. For a deterministic, output-sensitive algorithm for 3D convex hulls, see Edelsbrunner and Shi [95], and Chazelle and Matoušek [53]. For an earlier output-sensitive, deterministic algorithm for planar convex hulls, see Kirkpatrick and Seidel [129]. The on-line algorithm for Voronoi diagrams in Section 3.3 is due to Guibas, Knuth, and Sharir [117]. For an earlier *Delaunay tree*, also based on history, see Boissonnat and Teillaud [27]. The on-line algorithm for 3D convex polytopes and linear optimization in Section 3.2.1 is taken from Mulmuley [168]; see Guibas, Knuth, and Sharir [117] for an earlier, similar

history-based algorithm for maintaining 3D convex polytopes. Actually, [168] gives a dynamic linear optimization algorithm for dimensions up to 4. Recently, Matoušek and Schwarzkopf [142] have given a deterministic, quasi-optimal, dynamic search structure for answering linear optimization queries in all dimensions. See also Eppstein [100] for a related result in dimension three. For more on convex polytopes and linear programming, see Chapter 7.

The earlier analyses of randomized incremental algorithm were complex. The general theorem on randomized incremental construction (Theorem 3.4.5, Exercise 3.4.3) was proved by Clarkson and Shor [72] using a somewhat complex argument. Mulmuley [160, 161, 165] used for such analyses certain probabilistic games and a combinatorial series. This kind of analysis was also complex; later, after much simplification, it led to analysis in the the more realistic pseudo-random setting [169] (cf. Chapter 10). The trick that substantially simplified the proofs of randomized incremental algorithms was the idea of analyzing them backwards, first used in computational geometry extensively by Seidel [206, 205, 207]; also see Chew [61]. We have used this trick throughout the chapter. A much simpler proof of the bound for the expected conflict change (Theorem 3.4.5) has been recently given by Clarkson, Mehlhorn, and Seidel [70]. We have given in this chapter yet another proof for this bound, which is, hopefully, even simpler.

The notion of configuration spaces, under various forms and names, has been used by several authors. Clarkson and Shor call such configurations “regions”; we prefer the neutral term configuration, because, quite often, it does not have any resemblance to a region—for example, when a configuration is a vertex. Clarkson [66] and Haussler and Welzl [119] use similar spaces, but with more restrictions, in their important work on random sampling. Haussler and Welzl call configurations with empty defining (trigger) sets “ranges”. The set systems underlying the probabilistic games of Mulmuley [160, 161] consisted of “configurations”, characterized by sets of “stoppers” and “triggers”. The notion of “frames” due to Chazelle and Friedman [48] is also related, and so also the notion of “set systems” in Chazelle and Matoušek [54]. Thus, there is no unanimous choice for the terminology. In this book, we have formulated all randomized principles in one unifying framework of configuration spaces. However, it must be kept in mind that there is nothing very special about our particular choice of framework, since the underlying concept is extremely simple: a system of pairs of disjoint sets, with one set in the pair having constant size, and with additional restrictions, when necessary.

The notion of history used in this chapter has been used and developed by several authors [27, 25, 70, 81, 117, 168, 200, 206]. For the use of history in a dynamic setting, see Chapter 4. The tail estimates in Section 3.5 are due to Clarkson, Mehlhorn, and Seidel [70]. Exercise 3.5.3 is due to Mehlhorn, Sharir, and Welzl [158].

Chapter 4

Dynamic algorithms

In Chapter 3, we gave static and semidynamic algorithms for several problems: trapezoidal decompositions, Voronoi diagrams, three-dimensional convex polytopes, etc. By a semidynamic algorithm, we mean an algorithm that can only deal with additions. In this chapter, we want to generalize these algorithms to a fully dynamic setting wherein deletions of objects are allowed. Abstractly speaking, our problem is as follows. Let M denote the set of undeleted objects existing at any given moment. Our goal is to maintain a geometric complex (partition) $H(M)$ in a dynamic fashion. The complex $H(M)$ depends on the problem under consideration. The user can add or delete an object in M . Our goal is to update $H(M)$ quickly. If there is a search structure associated with $H(M)$, we also want to update that search structure quickly.

How does one evaluate a dynamic algorithm? The best we can hope for is that it performs an update in time proportional to the structural change in the underlying geometric complex $H(M)$. The dominant term in the running times of several of our algorithms will essentially turn out to be proportional to this structural change, ignoring a small factor. But in several problems, such as maintenance of Voronoi diagrams, the structural change can vary a lot. We have already encountered this problem in the semidynamic setting. In such cases, it is illuminating to know how the given dynamic algorithm performs on a “typical” update sequence. By a typical update sequence, we mean a random update sequence. We already know what is meant by a random sequence of additions. Let us now specify what is meant by a random update sequence, when even deletions are allowed.

Consider any update sequence \bar{u} . Define its signature $\delta = \delta(\bar{u})$ to be the string of + and – symbols such that the i th symbol in $\delta(u)$ is + (or –) iff the i th update in \bar{u} is addition (resp. deletion). Let N be the set of objects

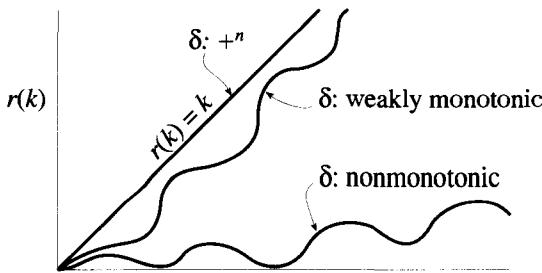


Figure 4.1: Examples of signatures.

involved in \bar{u} . We say that \bar{u} is an (N, δ) -sequence. We shall always denote the size of N by n .

Not every sequence δ of $+$ and $-$ symbols can occur as a signature of an update sequence. To see this, let us define $r(\delta, i)$, the *rank* of δ at time i , to be the number of $+$ symbols minus the number of $-$ symbols that occur in the initial subsequence of δ of length i . If δ is a signature of an update sequence \bar{u} , then $r(\delta, i)$ just denotes the number of current (i.e., undeleted) objects at time i . It is easy to see that δ is a valid signature, iff $r(\delta, i) \geq 0$, for all i . We denote $r(\delta, i)$ by just $r(i)$ when the signature is implicit.

When $\delta = +^n$, the update sequence involves only additions. Notice that $r(+^n, i) = i$. On a few occasions, we shall be interested in a class of nice signatures, called *weakly monotonic* signatures. These are akin to the signature $+^n$ in some sense (Figure 4.1). We say that a signature δ is weakly monotonic if $r(i) = \Omega(i)$. Since $r(i) \leq i$, this is equivalent to saying that there exists a constant $b \geq 1$ such that $r(i) \geq i/b$, for all i .

Now fix N and δ . We say that \bar{u} is a *random* (N, δ) -sequence if it is chosen from a uniform distribution on all valid (N, δ) -sequences. If $\delta = +^n$, this just means that all permutations of N are equally likely. Thus, a random $(N, +^n)$ -sequence is just a usual random sequence of additions. In general, a random (N, δ) -sequence has the following two properties. Let $|\bar{u}|$ denote the length of \bar{u} . For each $i \leq |\bar{u}|$, let S^i denote the object involved in the i th update. Let N^i denote the set of undeleted objects existing at time i during the execution of \bar{u} .

If \bar{u} is a random (N, δ) -sequence, then:

1. Each object in N is equally likely to be S^i , for any i .
2. Each N^i is a random subset of N of size $r(i)$.

Both of these properties follow by symmetry considerations, because each object in N is treated equally in our model. These are the only properties of random update sequences that we shall use.

Let us stipulate, for the sake of simplicity, that an object, once deleted during \bar{u} , cannot be added again. Strictly speaking, we do not need this assumption, because the above two properties are the only ones that we shall use in our analysis. But for conceptual simplicity, let us make this assumption any way. Then another way to think about a random (N, δ) -sequence would be: Read the signature δ from the left to right. If the symbol is $+$, randomly choose an unadded object from N and add it. If the symbol is $-$, randomly delete one of the earlier added objects.

In the above model of randomness, an adversary can choose the objects in N as well as the signature δ any way he likes. He must, however, treat all objects in N equally as far as the relative order of their additions and deletions is concerned. In that sense, the adversary is extremely fair: He treats all objects (people) equally regardless of their spatial positions.

We must emphasize that, in reality, updates need not be random. So we should use this model judiciously. It is very reasonable when, in a given dynamic problem, an adversary can come up with an update sequence that can cause an unusually large total structural change. This sequence will slow down *any* dynamic algorithm. In this situation, our model can justify why the dynamic algorithm in question should perform well on a typical update sequence.

The situation gets even better if one can prove good performance on most update sequences. Let us elaborate what we mean by this. When we say that a property holds for most (N, δ) -sequences, we mean that it holds for all but a $[1/g(n)]$ th fraction of the (N, δ) -sequences, where $g(n)$ is a large degree polynomial in n . Here both N and δ can be chosen by the adversary arbitrarily. One nice feature of our model of randomness is that it can serve as a means of proving a performance bound that holds for almost all update sequences. Let us see how. Since the probability distribution on the (N, δ) -sequences is uniform in our model, saying that the property holds on most (N, δ) -sequences is the same thing as saying that the probability of error (i.e., the probability that the property does not hold) is inversely proportional to a large degree polynomial in n . Once this is proven, one can assume, for all practical purposes, that the algorithm in question will demonstrate that performance except on some very rare occasions. *In fact, it becomes irrelevant now if the update sequences in reality are random or not, because the model is used only as a vehicle for proving the above property.* This last remark is crucial. It recommends that, for every algorithm that is analyzed in this model of randomness, our long term goal should be to prove performance

bounds that hold with high probability. At present, this is possible only in a few cases; overall, it remains a challenging open question in this area.

We should like to point out that in several problems, adversaries can cause no harm. As we have already remarked, the simplest such problem is encountered when N is a set of points on the real line. In this case, the structural change in the underlying partition is $O(1)$ during any update. In such cases, we do not need to make any assumption about the nature of updates. The same also holds for arrangements of lines. In this case, the structural change during any update is always of the order of m , where m is the number of currently existing lines.

As a convention in the rest of this book, by an update, we always mean any update. If we explicitly say a random update, we mean a random update in the above model. In this chapter, we shall be mainly interested in random updates. This is because for all the specific problems considered in this chapter—dynamic maintenance of trapezoidal decompositions, convex polytopes, and Voronoi diagrams—the adversary can come up with pathological update sequences. In later chapters, we shall deal with problems wherein no assumption about the update sequences needs to be made.

In the rest of this chapter, our goal is to extend the semidynamic algorithms given in Chapter 3 to the fully dynamic setting.

Notation. We shall denote the update sequence in a given dynamic problem by \bar{u} . We shall denote its signature by $\delta = \delta(\bar{u})$. We shall denote the set of objects that occur in \bar{u} by N . The size of N will be denoted by n . A dynamic algorithm does not know N or δ . At any given time, it only knows the objects that have been added so far. Let $|\bar{u}|$ denote the length of \bar{u} . For each $k \leq |\bar{u}|$, we shall denote the set of undeleted objects existing at time k by N^k . We shall denote the size of N^k by $r(k)$. It solely depends on the signature δ . In fact, it is precisely the rank of δ at time k . We shall denote the object that is involved in the k th update by S_k .

4.1 Trapezoidal decompositions

In Section 3.1.1, we considered semidynamic maintenance of trapezoidal decomposition. In this section, we shall extend the algorithm given there to the fully dynamic setting. Let N^k denote the set of undeleted segments existing at time k . Let $H(N^k)$ denote the resulting trapezoidal decomposition. Our goal is to maintain $H(N^k)$. Let $S = S_{k+1}$ be the segment involved in the $(k+1)$ th update. Our goal is to update $H(N^k)$ to $H(N^{k+1})$ quickly.

Addition: First, consider the case when the $(k+1)$ th update involves addition of S to N^k . Let us assume for a moment that we know the trapezoid in $H(N^k)$ containing an endpoint of S . In that case, we can add S to

$H(N^k)$ just as in the semidynamic case (Section 3.1). But how do we know the trapezoid in $H(N^k)$ containing an endpoint of S ? In the semidynamic setting, this was done with the help of history. We shall soon see that the same can be done in the fully dynamic setting as well. At present, let us just pretend that some oracle gives us this information for free.

Deletion: Next, consider the case when the $(k+1)$ th update involves deletion of S from N^k . In this case, $H(N^k)$ is updated to $H(N^{k+1})$ as follows (Figure 4.2). First, we make the segment S transparent to all vertical attachments adjacent to it. In other words, all attachments resting on S are made to pass through it. As far as these attachments are concerned, S does not exist. Let $\hat{H}(N^k)$ be the resulting partition. Then, $H(N^{k+1})$ is obtained from $\hat{H}(N^k)$ simply by erasing S and also the vertical attachments passing through the endpoints of S and the intersections on S . It is easy to see that this whole update can be done in time proportional to $\sum_f \text{face-length}(f)$, where f ranges over all trapezoids in $H(N^k)$ adjacent to S .

Analysis

Let us now calculate the expected cost of the $(k+1)$ th update. Whether the $(k+1)$ th update is addition or deletion is completely determined by the signature δ . We shall treat the two cases separately.

First, consider the case when the $(k+1)$ th update involves addition of S to N^k . We shall ignore at present the cost of locating an endpoint of S . Since the addition is done just as in the semidynamic case (Section 3.1), its expected cost, conditional on a fixed N^{k+1} (not N^k), is proportional to

$$\frac{|H(N^{k+1})|}{|N^{k+1}|} = O\left(\frac{|N^{k+1}| + I_{k+1}}{|N^{k+1}|}\right) = O\left(1 + \frac{I_{k+1}}{r(k+1)}\right), \quad (4.1)$$

where I_{k+1} is the number of intersections among the segments in N^{k+1} . This equation is analogous to (3.1). But (3.1) also takes into account the additional cost of maintaining conflicting lists, which is absent here.

Next, consider the case when the $(k+1)$ th update involves deletion of S from N^k . We have already observed that this can be done in time proportional to $\sum_f \text{face-length}(f)$, where f ranges over all trapezoids in $H(N^k)$ adjacent to S . We claim that its expected cost, conditional on a fixed N^k , is proportional to

$$\frac{|H(N^k)|}{|N^k|} = O\left(\frac{|N^k| + I_k}{|N^k|}\right) = O\left(1 + \frac{I_k}{r(k)}\right), \quad (4.2)$$

where I_k is the number of intersections among the segments in N^k . This equation is analogous to (4.1). It can be proved in exactly the same way.

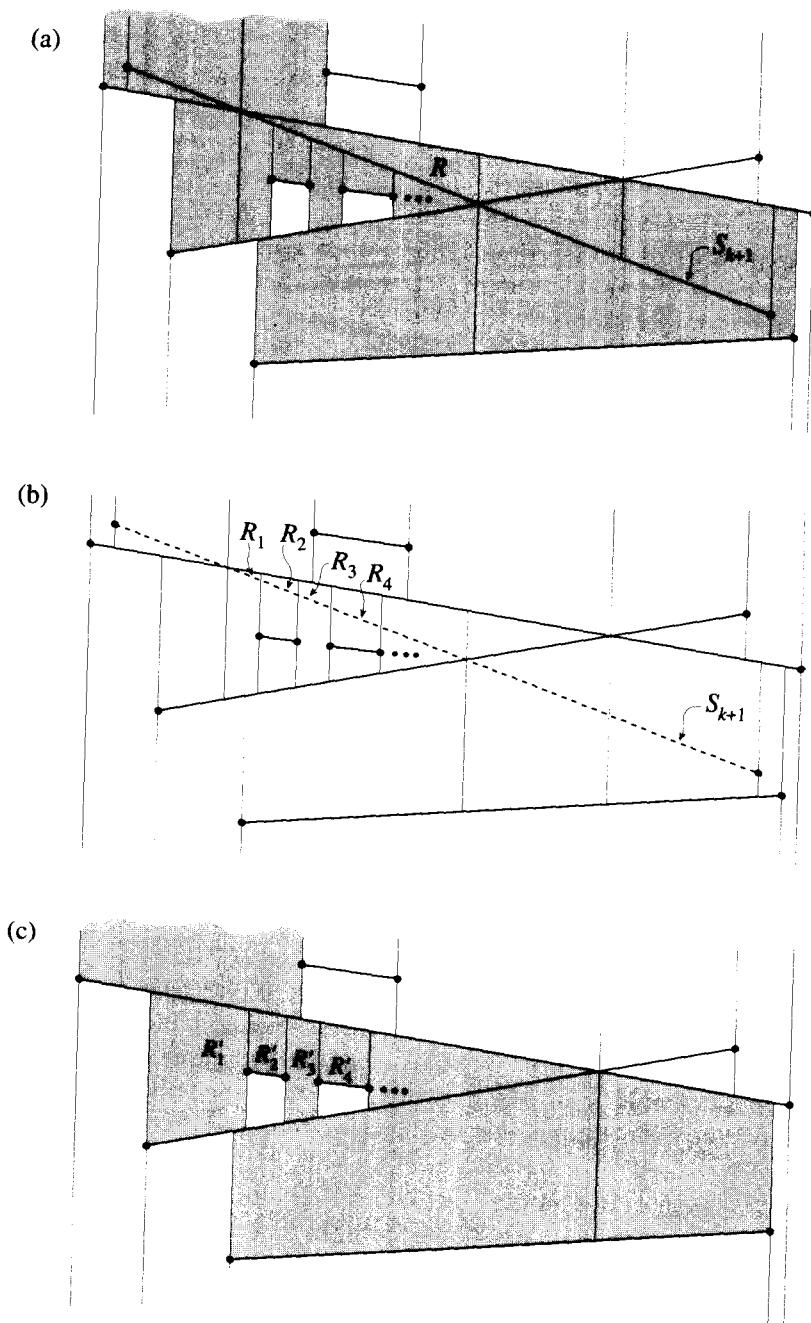


Figure 4.2: Removal of a segment: (a) $H(N^k)$. (b) $\hat{H}(N^k)$. (c) $H(N^{k+1})$.

If the reader remembers, (4.1) was proved by backward analysis, i.e., by imagining a deletion of a random segment from N^{k+1} . This time the situation is even simpler. There is no need for backward analysis, because we are dealing with a deletion in reality.

Now let us turn to the expected total cost of the algorithm over a random (N, δ) -sequence. We shall use the fact that, for each k , N^k is a random subset of N of size $r(k)$. In that case, the expected value of I_k is $O(r(k)^2 I/n^2)$, where I is the number of intersections among the segments in N (Lemma 3.1.3). Hence, it follows from (4.1) and (4.2) that the expected total cost is bounded, within a constant factor, by

$$\sum_{k=1}^{|\bar{u}|} 1 + \frac{r(k)I}{n^2} = n + \frac{I}{n^2} \sum_{i=1}^{|\bar{u}|} r(k) = O(n + I). \quad (4.3)$$

Note that $O(n + I)$ is not as tight a bound as the second bound in this equation. This is because an adversary can come up with signatures for which $\sum r(k)$ is much less than $O(n^2)$, in fact, as small as $O(n)$.

The discussion so far can be summarized as follows:

Proposition 4.1.1 *The expected cost of maintaining a trapezoidal decomposition over a random (N, δ) -sequence is of the order of*

$$n + \frac{I}{n^2} \sum_{i=1}^{|\bar{u}|} r(k) = O(n + I),$$

where I is the number of intersections among the segments in N . This ignores the cost of endpoint location during addition.

4.1.1 Point location

One main issue still remains unresolved. How do we locate an endpoint of a new segment during its addition? In the semidynamic setting, this was done using history. It turns out that a similar technique can be used in the dynamic setting as well. For this, we need to extend the earlier definition of history in the semidynamic setting (Section 3.1.1) to the fully dynamic setting.

Let \bar{u} be any (N, δ) -sequence. Let N^k denote the set of undeleted segments existing at time k . We shall denote the history at time k by $\text{history}(k)$. The leaves of $\text{history}(k)$ will point to the trapezoids in $H(N^k)$. Formally, $\text{history}(0)$ consists of one node that corresponds to the whole plane. Inductively, assume that $\text{history}(k)$ has been defined. Let S_{k+1} denote the segment involved in the $(k + 1)$ th update. $\text{History}(k + 1)$ is obtained from $\text{history}(k)$ as follows:

1. We mark the leaves of history(k) corresponding to the trapezoids in $H(N^k)$ that are destroyed during the $(k+1)$ th update as *killed*.
2. We allocate new nodes for the newly created trapezoids in $H(N^{k+1})$.
3. We build a certain data structure $\text{link}(S_{k+1})$, which serves as a link between the decompositions $H(N^k)$ and $H(N^{k+1})$. It is used in point location in a manner to be described soon.

Let us turn to the link structures. Given a destroyed trapezoid $f \in H(N^k)$ and a newly created trapezoid $g \in H(N^{k+1})$, let us say that g is a *child* of f if it intersects f .

If the $(k+1)$ th update is addition, every destroyed trapezoid in $H(N^k)$ has at most four children. In this case, $\text{link}(S_{k+1})$ simply consists of pointers from the killed leaves of history(k) to their children. This is exactly the link structure that was used in the semidynamic setting (Section 3.1.1).

If the $(k+1)$ th update is deletion, the number of children of a killed trapezoid in $H(N^k)$ need not be bounded by a constant. Refer to Figure 4.2 again. The children of the killed trapezoid $R \in H(N^k)$ (Figure 4.2(a)) are R'_1, R'_2, \dots , (Figure 4.2(c)), and their number can be arbitrarily large in principle. The children of any killed trapezoid in $H(N^k)$ can be linearly ordered from the left to right. We let $\text{link}(S_{k+1})$ consist of linearly ordered lists of children, one list for each killed leaf of history(k). With each such linearly ordered list, we maintain a search tree (say, a skip list or a randomized binary tree). This will allow us to search within this linear order in logarithmic time (with high probability, strictly speaking, but this will make no difference in what follows). Since we already know the linear order among the children of each leaf, each such search tree can be constructed in expected linear time (Exercises 1.3.2 and 1.4.3). Given any point p lying within a killed trapezoid in $H(N^k)$, $\text{link}(S_{k+1})$ allows us to locate its child containing p in $O(\log |N^k|) = O(\log r(k))$ time.

Let us now see how history(k) can be used for point location. Let p be any fixed query point. To locate p in $H(N^k)$, we begin at the root of history(k), which corresponds to the whole plane. We descend to its unique child containing p and keep on repeating this process until we reach a leaf of history(k). This leaf corresponds to the trapezoid in $H(N^k)$ containing p . Each descent from a node to its child is accomplished using a link structure associated with that node. This takes at most logarithmic time. It follows that the cost of locating any point in $H(N^k)$ is proportional to the length of the search path in history(k), ignoring a logarithmic factor.

Analysis

Now let us estimate the expected length of the search path assuming that the update sequence is random. This can be done very much as in the

semidynamic setting (Lemma 3.1.5). For each $j \leq k$, define a 0–1 random variable V_j as follows: V_j is defined to be one if the trapezoid in $H(N^{j-1})$ containing the query point p is destroyed during the j th update. Otherwise, it is defined to be zero. Clearly, the length of the search path is $\sum_{j \leq k} V_j$. If the j th update is deletion, $V_j = 1$ iff S_j is one of the at most four segments adjacent to the trapezoid in $H(N^{j-1})$ containing p . Each segment in N^{j-1} is equally likely to be deleted. Hence, this happens with probability at most $4/|N^{j-1}| = 4/r(j-1)$. If the j th update is addition, $V_j = 1$ iff S_j is one of the at most four segments adjacent to the newly created trapezoid in $H(N^j)$ containing p . Each segment in N^j is equally likely to be S_j . Hence, this happens with probability at most $4/|N^j| = 4/r(j)$. By summing over all j and a bit of rewriting, it follows that the expected length of the search path is of the order of

$$\sum_{j=1}^{|\bar{u}|} \frac{1}{r(j)}.$$

This can be as high as $O(k)$ for a pathological signature. The problem here is that the history at time k contains all objects encountered so far, even the deleted ones. This can cause the search time to be too long. We shall rectify the shortcoming later in Section 4.4. At present, let us note that the expected search path length is not too bad if the signature δ is weakly monotonic (Figure 4.1). In this case $r(j) = \Omega(j)$. Hence, the expected length becomes $O(\log k)$. In fact, the length is $O(\log k)$ for most update sequences. This follows from the Chernoff technique just as in Lemma 3.1.5. We leave this as an exercise to the reader.

What about the cost of maintaining history? This maintenance cannot increase the total running time of our algorithm by more than a constant factor. This is because the link structure for every update can be built in time proportional to the structural change in the underlying trapezoidal decomposition. Hence, the bound in Proposition 4.1.1 continues to hold.

To summarize:

Theorem 4.1.2 *The expected cost of maintaining a trapezoidal decomposition and the history over a random (N, δ) -sequence is of the order of*

$$n + \frac{I}{n^2} \sum_{i=1}^{|\bar{u}|} r(k) = O(n + I),$$

where I is the number of intersections among the segments in N . This ignores the cost of endpoint location during addition. The expected cost of point location at time k is $O(\log k \sum_{j \leq k} 1/r(j))$. If δ is weakly monotonic, it is $O(\log^2 k)$ for most (N, δ) -sequences.

As the reader must have noticed, the performance of our algorithm is not satisfactory if δ is not weakly monotonic. The main problem is that the history can become long and dirty if it contains too many segments that have already been deleted. We shall rectify this shortcoming in Section 4.4.

Exercise

4.1.1 Verify that the length of any search path in $\text{history}(k)$ is $O(\log k)$, for most (N, δ) -sequences, assuming that the signature δ is weakly monotonic. (Hint: Follow the proof of Lemma 3.1.5. Do not forget to verify that the number of distinct search paths in $\text{history}(k)$ is polynomial in k .)

4.2 Voronoi diagrams

In Section 3.3, we considered semidynamic maintenance of planar Voronoi diagrams. Let us now consider the dynamic maintenance problem. Let N^k denote the set of undeleted sites existing at time k . Let $G(N^k)$ denote their Voronoi diagram. Our goal is to maintain $G(N^k)$. Consider the $(k+1)$ th update. Let $S = S_{k+1}$ be the site involved in that update. Our goal is to update $G(N^k)$ to $G(N^{k+1})$ quickly.

Addition: First, consider the case when the $(k+1)$ th update involves addition of S to $G(N^k)$. Let us assume for a moment that we know a vertex of $G(N^k)$ in conflict with S , such as p in Figure 4.3. (By a conflicting vertex, we mean a vertex that is destroyed during the addition of S .) In that case, $G(N^k)$ can be updated to $G(N^{k+1})$ in time proportional to the structural change, just as in the semidynamic setting (Section 3.3). But how do we find a vertex of $G(N^k)$ in conflict with S ? In the semidynamic setting, such a vertex was determined with the help of history. We shall soon see that history can be maintained in the fully dynamic setting as well. For a moment, let us just assume that some oracle provides us this information for free.

Deletion: Next, consider the case when the $(k+1)$ th update involves deletion of S from N^k . Thus, $N^{k+1} = N^k \setminus \{S\}$. Let Q_1, \dots, Q_j be the sites in N^k labeling the regions adjacent to the Voronoi region $\text{vor}(S)$ (Figure 4.3). The Voronoi diagram of N^{k+1} can be obtained from the Voronoi diagram of N^k as follows. We separately compute the Voronoi diagram of the sites Q_1, \dots, Q_j . This can be done in $O(j \log j)$ time (Section 2.8.4). (Even the randomized incremental algorithm in Section 3.3 will suffice here.) Let us restrict this Voronoi diagram to the region $\text{vor}(S)$. This decomposes $\text{vor}(S)$, shown shaded in Figure 4.3, into $O(j)$ convex regions. After this decomposition of $\text{vor}(S)$, we remove the j edges bounding $\text{vor}(S)$. What results is the Voronoi diagram of N^{k+1} . Note that the structural change in the Voronoi diagram, as we pass from N^k to N^{k+1} , is proportional to j .

By structural change, we mean the number of newly created and destroyed edges, vertices, etc. Thus, the cost of deletion is proportional to the resulting structural change, ignoring a logarithmic factor.

Analysis

From the preceding discussion, the performance of our dynamic algorithm can be summarized as follows:

Proposition 4.2.1 The cost of any update is proportional to the structural change during that update. This ignores a logarithmic factor in the case of

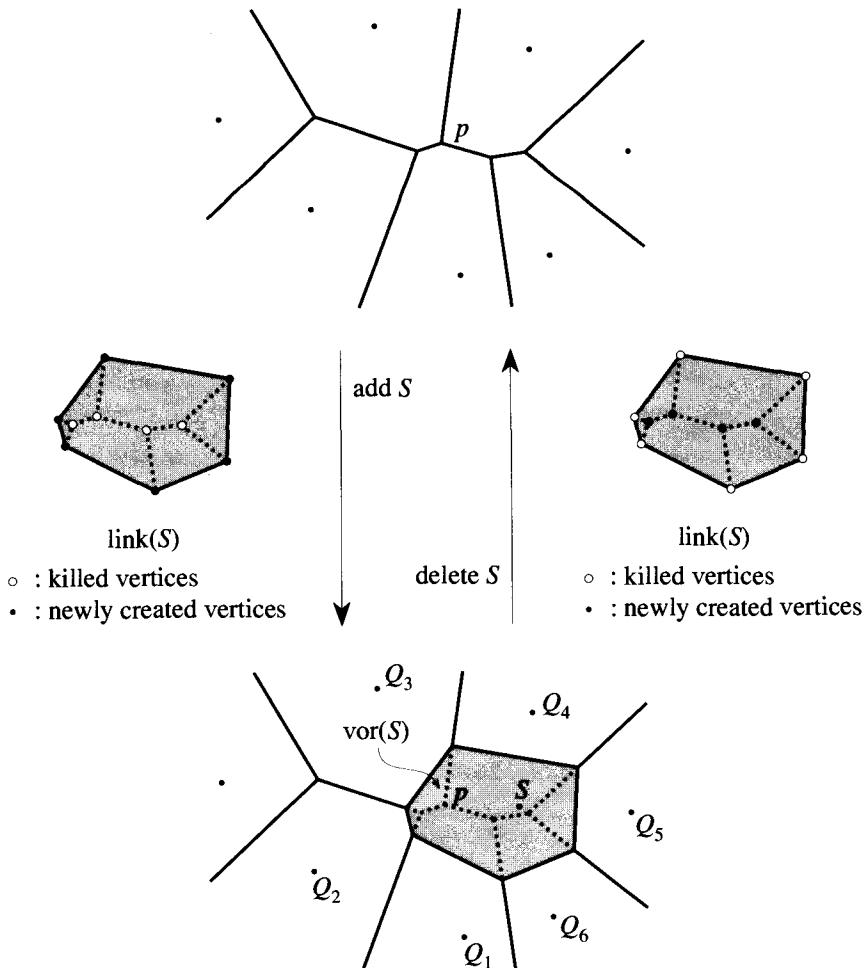


Figure 4.3: Addition or deletion of a site.

deletion. It also assumes that, during the addition of a new site, a conflicting vertex is provided to us for free by some oracle.

As we have already seen, the structural change during an update can vary a lot. Hence, the cost of a single update can also vary a lot. Now let us calculate the expected total cost over a random (N, δ) -sequence, ignoring the cost of the oracle. By the above proposition, we only need to estimate the expected structural change over a random (N, δ) -sequence. Fix a time k . Whether the k th update is addition or deletion is completely determined by the signature δ . We shall treat the two cases separately. First, consider the case when the k th update is deletion. We have already seen that the structural change during the deletion of a site $S \in N^k$ is $O(j)$. Here j is the size of $\text{vor}(S)$, which, in turn, is proportional to the number of vertices in $G(N^k)$ that are destroyed during the deletion. A fixed vertex v in the Voronoi diagram $G(N^k)$ is destroyed during deletion, iff the deleted site is one of the three sites defining v . By a defining site, we mean a site whose Voronoi region is adjacent to v . If we sum the structural change, by letting the deleted site S^{k+1} range over all sites in N^k , then each vertex in $G(N^k)$ contributes to this sum precisely thrice. It follows that this sum, which we shall denote by ϕ , is proportional to the size of $G(N^k)$. Since the size of a planar Voronoi diagram is linear in the number of sites (Section 2.5), it follows that $\phi = O(|N^k|)$. Since the deleted site S_{k+1} is randomly chosen from N^k , the expected structural change is $\phi/|N^k| = O(1)$.

If the $(k+1)$ th update were addition, we use backward analysis: The expected structural change in $G(N^k)$ during a random addition is the same as the expected structural change in $G(N^{k+1})$, if one were to delete a random site from N^{k+1} . This is because each site in N^{k+1} is equally likely to be involved in the $(k+1)$ th update.

Thus, the expected structural change during each update in a random (N, δ) -sequence is $O(1)$. Hence, the total expected structural change is $O(n)$.

To summarize:

Proposition 4.2.2 *The total cost of maintaining a Voronoi diagram in a given update sequence is proportional to the total structural change during that sequence. This ignores a logarithmic factor in the cost of a deletion. It also ignores the cost of locating a conflicting vertex during addition. The expected (total) structural change during a random (N, δ) -sequence is $O(n)$.*

4.2.1 Conflict search

One issue is still unresolved. How do we locate a conflicting vertex during the addition of a new site? In the semidynamic setting, this was done using history (Section 3.3). A similar thing can be done in the dynamic setting as

well. For this, we have to extend the semidynamic history, as described in Section 3.3, to the fully dynamic setting.

Let \bar{u} be any (N, δ) -sequence. Let N^k denote the set of undeleted sites existing at time k . We shall denote the history at time k by $\text{history}(k)$. The leaves of $\text{history}(k)$ will point to the vertices of the Voronoi diagram $G(N^k)$. Intuitively, $\text{history}(k)$ is obtained by recording the history of the sequence of Voronoi diagrams $G(N^1), G(N^2), \dots, G(N^k)$. As a convention, we assume that the single infinite Voronoi region in $G(N^1)$ is a (symbolic) triangle approaching infinity (refer to the bounding box trick in Section 3.2). Formally, $\text{history}(1)$ contains nodes for the vertices of this triangle. Inductively, assume that $\text{history}(k)$ is defined. $\text{History}(k+1)$ is obtained from $\text{history}(k)$ as follows:

1. We mark the leaves of $\text{history}(k)$ pointing to the destroyed vertices in $G(N^k)$ as *killed*.
2. We allocate new nodes for the newly created vertices of $G(N^{k+1})$.
3. We link the killed and the newly created nodes according to their adjacencies, as in Figure 4.3. We shall denote this link structure by $\text{link}(S_{k+1})$.

Let us make one definition. Consider any node \hat{v} in $\text{history}(k)$ created at time j , $1 \leq j \leq k$. Let v be the corresponding vertex in the Voronoi diagram $G(N^j)$. Given a site $I \notin N^j$, we say that \hat{v} is in conflict with I if v is in conflict with I . This means that the imaginary addition of I to the Voronoi diagram $G(N^j)$ would destroy v . This is the same as saying that I is nearer to v than any of the three sites defining v . By a defining site, we mean a site labeling one of the Voronoi regions in $G(N^j)$ adjacent to v .

The link structure $\text{link}(S_{k+1})$ has the following property: Suppose we know a leaf of $\text{history}(k)$ in conflict with I . In that case, we can descend to a leaf of $\text{history}(k+1)$ in conflict with I as follows. The killed and the newly created nodes of $\text{link}(S_{k+1})$ in conflict with I form its connected subgraph. If the $(k+1)$ th update is addition, this follows just as in the semidynamic case (Section 3.3). For the case of deletion, it follows by symmetry. Connectivity allows us to locate a conflicting leaf of $\text{history}(k+1)$ by a simple search within $\text{link}(S_{k+1})$. We start the search at the known conflicting leaf of $\text{history}(k)$ and restrict the search to the above connected subgraph of $\text{link}(S_{k+1})$. We stop the search as soon as we visit a newly created node in conflict with I . In the worst case, the cost of the search is proportional to the size of this connected subgraph. This size is equal to the total number of newly created or destroyed nodes during the $(k+1)$ th update that are in conflict with I .

The use of history during conflict search should now be clear. Suppose S is any site not in N^k . Let us see how to locate a vertex of the Voronoi diagram $G(N^k)$ in conflict with S . First, we trivially locate a node of $\text{history}(1)$ in

conflict with S . This takes $O(1)$ time. Assume that this node is killed at time j . Using $\text{link}(S_j)$, as described before, we locate a node created at time j in conflict with S . We repeat this process until we reach a leaf of $\text{history}(k)$. It corresponds to a vertex of $G(N^k)$ in conflict with S . It is easy to see that, in the worst case, the cost of this conflict search is proportional to the total number of nodes in $\text{history}(k)$ in conflict with S .

We are now in position to state our dynamic algorithm for maintaining Voronoi diagrams in a final form. At each time k , we maintain the Voronoi diagram $G(N^k)$ and also $\text{history}(k)$. If the $(k+1)$ th update is addition, we locate a vertex of $G(N^k)$ in conflict with S_{k+1} . After this, we update $G(N^k)$ to $G(N^{k+1})$ as in the semidynamic setting. We have already described how to update $G(N^k)$ if the update is deletion. As we update $G(N^k)$, $\text{link}(S_{k+1})$ can be concurrently constructed in the same order of time.

Analysis

Proposition 4.2.2 bounds the expected running time of our algorithm on a random (N, δ) -sequence, except for the cost of conflict search during additions. We have already seen that the cost of the conflict search during the $(k+1)$ th update, assuming that it is addition, is proportional to the number of nodes in $\text{history}(k)$ in conflict with S_{k+1} .

Claim 4.2.3 *Assume that the update sequence under consideration is a random (N, δ) -sequence. Then the expected number of nodes in $\text{history}(k)$ in conflict with S_{k+1} is $O(\log k)$, assuming that the signature δ is weakly monotonic.*

This follows from a general result to be proven soon (Theorem 4.3.5). For a moment, let us take this for granted.

We have then proved the following.

Theorem 4.2.4 *Assume that δ is weakly monotonic. Then the expected cost of maintaining a Voronoi diagram over a random (N, δ) -sequence is $O(n \log n)$. The expected cost of conflict search through $\text{history}(k)$ during the addition of S_{k+1} is $O(\log k)$.*

This theorem generalizes the previous result in the semidynamic setting (Theorem 3.3.1) to weakly monotonic signatures. We cannot prove a good bound on the performance of our algorithm if the signature is not weakly monotonic. This is because the history at time k depends on the sites that have already been deleted. If the number of such deleted sites is too large in comparison with the number of undeleted sites, then the history becomes long and dirty. We shall rectify this shortcoming in Section 4.4.

So far we have not addressed the point location issue for Voronoi diagrams. In Section 3.3, we had given an efficient semidynamic point location structure

for Voronoi diagrams. It can be easily extended to the fully dynamic setting very much as in this section. This is done in the exercises.

Exercises

4.2.1 Extend the point location structure in Section 3.3 to the fully dynamic setting as follows. Let $H(N^k)$ denote the radial triangulation of the Voronoi diagram at time k . As in Section 3.3, remember the history of the radial triangulations. We have already specified in that section the link structure that is constructed during addition. The link structure during deletion can be constructed as follows. Suppose the $(k+1)$ th update involves deletion of $S = S_{k+1}$ from N^k . Consider the planar graph formed by the union of all newly created triangles in $H(N^{k+1})$. Let the link structure be the point location structure for this planar graph. It can be constructed using the algorithm in Section 3.1.2 in expected time proportional to its size, ignoring a logarithmic factor. Given a destroyed triangle in $H(N^k)$ containing a query point p , the newly created triangle in $H(N^{k+1})$ containing p can be found in expected logarithmic time using this link structure.

Describe the algorithm and the point location procedure in detail. Analyze the performance of this algorithm on a random (N, δ) -sequence. For a fixed query point p , show that the expected cost of locating p in $H(N^k)$ is proportional to $\log k \sum_{j \leq k} 1/r(j)$. If δ is weakly monotonic, this is $O(\log^2 k)$. Show that the expected cost of maintaining the point location structure over a random (N, δ) -sequence is $O(n \log n)$, if the signature is weakly monotonic.

4.2.2 What problems occur if we try to maintain three-dimensional convex polytopes dynamically using the approach in this section? (Hint: Let N^k denote the set of half-spaces at time k . Consider the deletion of a half-space from N^k . This can cause a redundant half-space in N^k to become nonredundant in N^{k+1} . How would you detect such half-spaces?)

4.3 History and configuration spaces

We now formalize the notion of history, as illustrated in the previous sections, in the general setting of configuration spaces. We shall also prove some general results, which will be useful later.

Let us say that we are interested in dynamically maintaining a geometric complex (partition) $H(M)$. Here M denotes the set of objects existing at any given time. The complex $H(M)$ could be a Voronoi diagram, a trapezoidal decomposition, or whatever, depending upon the problem under consideration. Let \bar{u} be any (N, δ) -sequence. Intuitively, the history at time k during the execution of \bar{u} is a data structure that results by remembering the differences between the successive partitions in the sequence $H(N^1), H(N^2), \dots, H(N^k)$. Here N^k denotes the set of objects existing at time k . The leaves of $\text{history}(k)$ correspond to the faces of $H(N^k)$ of all or

some specified dimensions. We assume that these faces are configurations that are defined by a bounded number of objects in N^k .

More formally, consider any configuration space $\Pi(N)$ over the set N of objects occurring in \bar{u} . Let $\Pi(N^k)$ be the subspace induced by the set N^k . Let $\Pi^0(N^k)$ be the set of active configurations at time k , i.e., the configurations active over N^k . Consider an abstract dynamic algorithm whose task is to maintain $\Pi^0(N^k)$. We wish to define its history. The configuration space $\Pi(N)$ depends upon the problem under consideration. In the case of trapezoidal decompositions, it is the configuration space of feasible trapezoids over N (Example 3.4.1). In the case of Voronoi diagrams, it is the configuration space of feasible triangles over N (Example 3.4.3). The earlier histories for trapezoidal decompositions and Voronoi diagrams (Sections 4.1 and 4.2) were defined over these configuration spaces.

We say that a configuration in $\Pi^0(N^k)$ is *destroyed* (killed, deactivated) during the $(k + 1)$ th update if it is no longer present in $\Pi^0(N^{k+1})$. We say that a configuration in $\Pi^0(N^{k+1})$ is newly created during the $(k + 1)$ th update if it was not present in $\Pi^0(N^k)$. The *structural change* during the $(k + 1)$ th update is defined as the number of destroyed configurations plus the number of newly created configurations during this update. The total structural change during \bar{u} is defined to be the sum of the structural changes over all updates in \bar{u} .

Intuitively, history is just a record of the structural changes together with auxiliary link structures. We shall denote the history at time k during the execution of \bar{u} by $\text{history}(k)$. The leaves of $\text{history}(k)$ will correspond to the configurations in $\Pi^0(N^k)$. These are the active configurations in $\Pi(N^k)$. Formally, $\text{history}(1)$ contains a node for every configuration of $\Pi^0(N^1)$. Inductively, $\text{history}(k)$ is obtained from $\text{history}(k - 1)$ as follows: Let S_k denote the object that is added or deleted during the k th update.

1. For every newly created configuration σ in $\Pi^0(N^k)$, we allocate a new node $\hat{\sigma}$ in the history. We say that $\hat{\sigma}$ is *created* at time k .
2. For every killed (deactivated) configuration $\tau \in \Pi^0(N^{k-1})$, we mark the corresponding leaf $\hat{\tau}$ of $\text{history}(k - 1)$ as *killed* at time k by S_k . If necessary, we stamp $\hat{\tau}$ with the time of its death.
3. Create a *link* structure $\text{link}(S_k)$, which is meant to provide a link between $\Pi^0(N^{k-1})$ and $\Pi^0(N^{k+1})$. It is used during the search through the history.

The nature of the link structure and how it is actually used depends on the problem under consideration. In the above definition, history at time k depends on all objects that have been encountered so far, even the deleted ones. This has an obvious drawback: The history can become too long. This

in turn can slow down the search through it. We shall rectify this drawback in Section 4.4.

There is one difference between the dynamic history, as defined here, and the semidynamic history, as defined in Chapter 3. In the semidynamic setting, a killed configuration cannot become active again, and hence, the history contains just one node for every configuration that becomes active during the execution of \bar{u} . This need not be the case in the present fully dynamic setting.

4.3.1 Expected performance*

The total number of nodes in the history of \bar{u} is bounded by the total structural change during \bar{u} . This leads us to the problem of estimating the expected structural change during a random (N, δ) -sequence. We have already done this implicitly for the configurations spaces that arise in connection with trapezoidal decompositions and Voronoi diagrams (Sections 4.1 and 4.2). We now wish to extend the argument used in these sections to arbitrary configuration spaces.

We have already dealt with arbitrary configurations spaces in the semidynamic setting, i.e., when the signature $\delta = +^n$. By Theorem 3.4.5, the expected total structural change over a random $(N, +^n)$ -sequence of additions is bounded by

$$\sum_{k=1}^{|\bar{u}|} d \frac{e(k)}{k}.$$

Here $e(k)$ denotes the expected size of $\Pi^0(N^k)$, assuming that N^k is a random sample (subset) of N of size k . The constant d is the maximum degree of a configuration in $\Pi(N)$. Note that in the semidynamic case, $r(k) = k$.

We shall now generalize the above result in a slightly weaker form to arbitrary signatures. Consider a random (N, δ) -sequence \bar{u} . We wish to bound the expected total structural change during this sequence. Fix any $i \leq |\bar{u}|$. Consider the $(i+1)$ th update. Whether this update is an addition or a deletion depends solely on the signature δ . Consider, first, the case when it is deletion. We shall analyze addition later.

Let us first estimate the expected number of configurations in $\Pi^0(N^i)$ that are destroyed during this deletion. First, we shall calculate the expected value, conditional on a fixed N^i . This is done by a verbatim translation of the argument that we used in the case of Voronoi diagrams (Section 4.2). A configuration $\sigma \in \Pi^0(N^i)$ can be destroyed iff the deleted object is one of the at most d triggers associated with σ . Hence, if we sum the number of destroyed configurations by letting the deleted object S range over all of N^i ,

then the sum is at most $d\pi^0(N^i)$. Here $\pi^0(N^i)$ is the size of $\Pi^0(N^i)$. Each object in N^i is equally likely to be deleted. Hence, the expected number of destroyed configurations in $\Pi^0(N^i)$, conditional on a fixed N^i , is at most

$$d\pi^0(N^i)/|N^i| = d\pi^0(N^i)/r(i). \quad (4.4)$$

Now let us turn to the number of newly created configurations. In the case of Voronoi diagrams (Section 4.2), the number of newly created configurations was proportional to the number of destroyed configurations. In general, this is not the case. So a separate argument is needed.

For example, let N be a set of half-spaces and let $\Pi(N)$ be the configuration space of feasible vertices over N (Example 3.4.2). We have already seen there that the configurations in $\Pi(R)$ can be identified with the vertices of the arrangement formed by the boundaries of the half-spaces in R . For any subset $R \subseteq N$, $\Pi^0(R)$ can be identified with the set of vertices of the convex polytope formed by intersecting the half-spaces in R . In general, for any integer $c \geq 0$, $\Pi^c(R)$ is defined to be the set of configurations in $\Pi(R)$ with level c . In the above example, it consists of the vertices in the arrangement formed by R that are in conflict with exactly c half-spaces in R . Figure 4.4 gives an illustration for $R = N^i$. Now, consider the deletion of a half-space $S \in N^i$. Let $N^{i+1} = N^i \setminus \{S\}$. The newly created vertices in $\Pi^0(N^{i+1})$ can be identified with the vertices in $\Pi^1(N^i)$ in conflict with S . For example, in Figure 4.4, v is one such vertex.

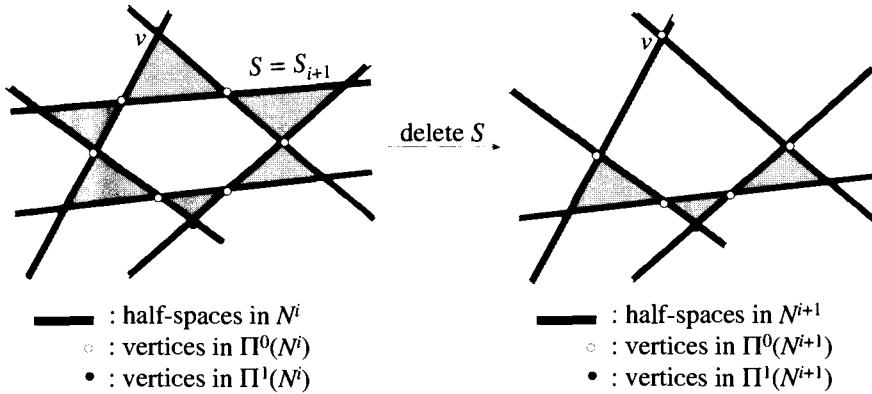


Figure 4.4: Deletion of a half-space.

The same thing also holds for general configuration spaces. The newly created vertices in $\Pi^0(N^{i+1})$ can be identified with the configurations in $\Pi^1(N^i)$ that are in conflict with the deleted object $S = S_{i+1}$. Hence, if we sum the number of such configurations by letting the deleted object S range

over all of N^i , the sum is $\pi^1(N^i)$. Here $\pi^1(N^i)$ denotes the size of $\Pi^1(N^i)$. Each object in N^i is equally likely to be deleted. Hence, the expected number of newly created configurations, conditional on a fixed N^i , is

$$\pi^1(N^i)/|N^i| = \pi^1(N^i)/r(i). \quad (4.5)$$

Now we use the fact that N^i is a random sample of N of size $r(i)$. For $c \geq 0$, let $e^c(r)$ denote the expected value of $\pi^c(R)$, where R is a random sample of N of size r . Here $\pi^c(R)$ denotes the size of $\Pi^c(R)$, the set of configurations in $\Pi(R)$ with level c . Note that $e^0(r)$ is just $e(r)$ as defined before.

It follows from (4.4) and (4.5) that the expected structural change during the $(i+1)$ th update of a random (N, δ) -sequence, assuming that it is deletion, is bounded by

$$\frac{d e(r(i)) + e^1(r(i))}{r(i)}. \quad (4.6)$$

Next, we turn to the case when the $(i+1)$ th update is addition. We shall analyze this addition backwards. The structural change during the addition of an object to N^i is the same as the structural change during an imaginary deletion of the same object from N^{i+1} . Because the update sequence is random, each object in N^{i+1} is equally likely to be involved in the $(i+1)$ th update. This means that the imaginary deletion is random too. Arguing as before, it follows that the expected structural change, conditional on a fixed N^{i+1} (not N^i), is bounded by

$$\frac{d e(r(i+1)) + e^1(r(i+1))}{r(i+1)}. \quad (4.7)$$

The total expected structural change follows by summing (4.6) and (4.7) over all i . After a bit of rewriting, it follows that:

Lemma 4.3.1 *The expected value of the (total) structural change during a random (N, δ) -sequence is*

$$\sum_{k=1}^{|u|} \frac{d e(r(k)) + e^1(r(k))}{r(k)}.$$

Here d is the maximum degree of a configuration in $\Pi(N)$.

The bound in this lemma is not useful on its own. It becomes useful in conjunction with the fact that, for every constant $c \geq 0$,

$$e^c(r(k)) = O(e(r(k)/2)).^1 \quad (4.8)$$

This follows from the following more general lemma.

¹Strictly speaking we should be writing $e(\lfloor r(k)/2 \rfloor)$.

Lemma 4.3.2 Let $a > 0$ be a fixed integer. Consider any configuration space $\Pi(R)$. Let d be the maximum degree of a configuration in $\Pi(R)$. Let r be the size of R . Let Q be a random sample of R of size $r/2$. Then

$$\pi^a(R) = O(2^{a+d} E[\pi^0(Q)]).$$

The same result also holds if Q is alternatively chosen as follows: Flip a fair coin for each object in R and put it in Q iff the toss is successful.

Before turning to the proof of the lemma, let us deduce (4.8) from it. It follows from the lemma that, for a constant $c \geq 0$, $\Pi^c(N^k)$ is bounded, within a constant factor, by the expected value of $\pi^0(Q)$, where Q is a random sample of N^k of size $|N^k|/2 = r(k)/2$. If N^k is itself a random sample of N , then Q can also be thought of as a random sample of N of size $r(k)/2$. Hence, $e^c(r(k)) = O(e(r(k)/2))$, as claimed.

Proof of the lemma: Fix a configuration $\sigma \in \Pi^a(R)$. Let $d(\sigma)$ be its degree. The configuration σ occurs in $\Pi^0(Q)$ iff Q contains all $d(\sigma)$ triggers associated with σ but none of the a stoppers associated with it. The random sample Q has half the size of R . Hence, the probability that any given element in R occurs in Q is $1/2$. It follows that

$$\text{prob}\{\sigma \in \Pi^0(Q)\} \approx \frac{1}{2^{d(\sigma)+a}}, \quad (4.9)$$

where \approx denotes asymptotic equality with a constant factor ignored. This equation is exact if Q is chosen by flipping a fair coin. If Q is a random subset of size exactly $r/2$, then one can calculate the exact expression in terms of binomial coefficients. We do not need the exact value here.

From (4.9), it follows that

$$E[\Pi^0(Q)] = \sum_{\sigma \in \Pi(R)} \text{prob}\{\sigma \in \Pi^0(Q)\} \geq \sum_{\sigma \in \Pi^a(R)} \text{prob}\{\sigma \in \Pi^0(Q)\} \geq \frac{\pi^a(R)}{2^{a+d}}.$$

The lemma follows from this inequality. \square

Define $\bar{e}(k) = \max\{e(j) \mid j \leq k\}$. Combining Lemma 4.3.1 and (4.8), it follows that:

Theorem 4.3.3 The expected structural change during a random (N, δ) -sequence is bounded, within a constant factor, by

$$\sum_{k=1}^{|u|} \frac{e(r(k)) + e(r(k)/2)}{r(k)} = O\left(\sum_k \frac{\bar{e}(r(k))}{r(k)}\right).$$

In particular, this is $O(\sum_k \bar{e}(k)/k)$, if δ is weakly monotonic (Figure 4.1). This is because $k \geq r(k) = \Omega(k)$ in that case.

Next, we briefly turn to a special class of configurations spaces, namely, trivial configuration spaces (Example 3.4.6). Configuration space $\Pi(N)$ is trivial iff $\pi^0(R) = O(1)$, for every $R \subseteq N$. We had seen in Example 3.4.6 that the length of the search path in history is often bounded by the structural change in a trivial configuration space determined by the query. For trivial configuration spaces, Theorem 4.3.3 assumes a very simple form.

Corollary 4.3.4 *If $\Pi(N)$ is a trivial configuration space then the expected structural change during a random (N, δ) -sequence is $O(\sum_k 1/r(k))$. In particular, it is $O(\log n)$ if δ is weakly monotonic. In the latter case, the bound is, in fact, $O(\log n)$ for most update sequences.*

Proof. This first part follows from Theorem 4.3.3, because in this case, $e(r) = O(1)$, for all r . We invite the reader to give a simple direct proof along the lines in Section 4.1. There we estimated the expected length of the search path in the history of trapezoidal decompositions. The stronger high-probability bound follows from the Chernoff technique, as indicated there. □

Next, we turn to the cost of the conflict search through history(k). In the case of Voronoi diagrams (Section 4.2.1), we saw that the cost of locating a vertex in the Voronoi diagram $G(N^k)$, which conflicts with a site S , is proportional to the number of nodes in history(k) in conflict with S . Its expected value can be bounded using the following general theorem.

Theorem 4.3.5 *Consider a random (N, δ) -sequence. Suppose the $(k+1)$ th update during this sequence is the addition of S_{k+1} . Then the expected number of nodes in history(k) in conflict with S_{k+1} is bounded, within a constant factor, by $\sum_{i \leq k} \bar{e}(r(i))/r(i)^2$. In particular, this is of the order of $\sum_{i \leq k} \bar{e}(i)/i^2$, if δ is weakly monotonic. This follows because $i \geq r(i) = \Omega(i)$ in that case.*

Before we proceed to the proof, let us see why Claim 4.2.3 follows from this theorem. Since the size of a planar Voronoi diagram is linear in the number of sites, $\bar{e}(i) = O(i)$. Hence, it follows that the expected number of nodes conflicting with S_{k+1} is of the order of $\sum_{i \leq k} 1/i = O(\log k)$ if the signature is weakly monotonic.

Proof. In what follows, we shall let d denote the maximum degree of a configuration in $\Pi(N)$. Consider a fixed $i < k$. Assume that the $(i+1)$ th update is deletion. Addition can be treated backwards as usual.

Let S_{i+1} denote the deleted object. Let N^i denote the set of objects existing before deletion. Let $\tilde{N}^i = N^i \cup \{S_{k+1}\}$. In other words, \tilde{N}^i contains the set of objects existing at time i and the object added at time $k+1$. Let us estimate the expected number of nodes killed at time i that are in conflict with S_{k+1} . First, we shall estimate this expected value, conditional on a

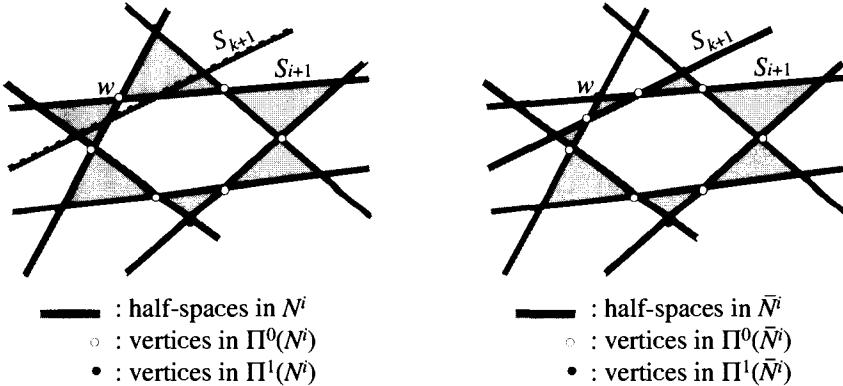


Figure 4.5: Deletion of a half-space (continued).

fixed \bar{N}^i . Consider any node, killed at time i , which conflicts with S_{k+1} . It corresponds to a configuration σ in $\Pi^0(N^i)$ which is destroyed during the deletion of S_{i+1} and which is in conflict with S_{k+1} . Figure 4.5 gives an illustration for the configuration space over half-spaces considered earlier. The vertex $w \in \Pi^0(N^i)$ is killed at time i during the deletion of S_{i+1} , and it is also in conflict with S_{k+1} . Note that w can also be considered as a vertex in $\Pi^1(\bar{N}^i)$. Moreover, S_{i+1} is one of the triggers associated with it and S_{k+1} is the unique stopper in \bar{N}^i associated with it.

This is true in general. A configuration in $\Pi^0(N^i)$ which is killed at time i and which is in conflict with S_{k+1} can be thought of as a configuration in $\Pi^1(\bar{N}^i)$; S_{i+1} is one of the at most d triggers associated with this configuration and S_{k+1} is the unique stopper in \bar{N}^i associated with it. If we sum the number of such configurations, by letting S_{i+1} and S_{k+1} range over all possible pairs of objects in \bar{N}^i , then the resulting sum is bounded by $d\pi^1(\bar{N}^i)$, where $\pi^1(\bar{N}^i)$ is the size of $\Pi^1(\bar{N}^i)$. This is because any configuration in $\Pi^1(\bar{N}^i)$ can be counted at most d times, once each for every trigger associated with it. By the random nature of the update sequence, each pair of objects in \bar{N}^i is equally likely to occur as S_{i+1} and S_{k+1} . The size of \bar{N}^i is $r(i) + 1$. Hence, it follows that, conditional on a fixed \bar{N}^i , the expected number of nodes in history(k) which are killed at time i and which are in conflict with S_{k+1} is bounded by

$$\frac{d\pi^1(\bar{N}^i)}{r(i)[r(i) + 1]}.$$

Because of the random nature of the update sequence, \bar{N}^i is a random subset of N of size $r(i) + 1$. Hence, it follows that the expected number of

nodes killed at time i that are in conflict with S_{k+1} is bounded by

$$\frac{de^1(r(i) + 1)}{r(i)[r(i) + 1]} = O\left(\frac{\bar{e}(r(i))}{r(i)^2}\right). \quad (4.10)$$

Here the last equality follows from Lemma 4.3.2, just as (4.8) followed from it.

In an analogous fashion, one can estimate the expected number of newly created nodes at time i that are in conflict with S_{k+1} . We have already seen in the proof of Theorem 4.3.3 that a newly created configuration during the $(i+1)$ th deletion can be thought of as a configuration in $\Pi^1(N^i)$ (Figure 4.4); S_{i+1} is its unique stopper in N^i . If, in addition, it is in conflict with S_{k+1} , then it can be thought of as a configuration in $\Pi^2(\bar{N}^i)$, and S_{i+1} and S_{k+1} are its only stoppers in \bar{N}^i . Proceeding as before, one concludes that the expected number of such configurations is bounded within a constant factor by

$$\frac{e^2(r(i) + 1)}{r(i)[r(i) + 1]} = O\left(\frac{\bar{e}(r(i))}{r(i)^2}\right). \quad (4.11)$$

It follows from (4.10) and (4.11) that the expected number of killed or newly created nodes at time i , which are in conflict with S_{k+1} , is $O(\bar{e}(r(i))/r(i)^2)$. This was assuming that the i th update is deletion. If it were addition, the similar bound can be proved by considering addition as an imaginary backward deletion.

The theorem now follows by summing over all i . □

Exercises

4.3.1 We did not give much attention to the constant factors in this section. Show that $e^1(r) = O(d\bar{e}(r))$, where d is the maximum degree of a configuration in $\Pi(N)$; the constant factor within the Big-Oh notation should not depend on d . (Hint: Instead of taking a random sample of size $r/2$, as in Lemma 4.3.2, take a random sample of size $dr/(d+1)$. Do precise calculations using binomial coefficients and Stirling's approximation.)

Calculate the constant factors precisely in all bounds in this section.

4.3.2 Let $\Pi(M)$ be any configuration space. For an integer $c \geq 0$, consider the set $\Pi^c(M)$ of configurations with level c . Remove a random object S from M . We say that a configuration in $\Pi^c(M)$ is destroyed during this deletion if it is adjacent to S , i.e., if S is one of the triggers associated with it. We say that the level of a configuration $\sigma \in \Pi^c(M)$ decreases during this deletion if S conflicts with σ . This is because σ becomes a configuration in $\Pi^{c-1}(M \setminus \{S\})$ after this deletion.

Consider any configuration space $\Pi(M)$. Let m be the size of M . Remove a random object in M . Show the following:

- (a) The expected number of configurations in $\Pi^c(M)$ that get destroyed is bounded by $d\pi^c(M)/m$. Here d is the maximum degree of a configuration in $\Pi(M)$.

- (b) The expected number of configurations in $\Pi^c(M)$ whose level decreases during the deletion is bounded by $c\pi^c(M)/m$.

4.3.3 Prove the high-probability bound in Corollary 4.3.4. (Hint: Revisit the proof of Lemma 3.1.5.)

4.3.4 For $\delta = +^n$, improve the bounds in Theorems 4.3.3 and 4.3.5 to $O(\sum_k e(k)/k)$ and $O(\sum_k e(k)/k^2)$, respectively. Show that Theorem 3.4.5 also follows from this, ignoring a constant factor.

4.4 Rebuilding history

As pointed out on several occasions, the performance of the previous history-based algorithms is not satisfactory enough if the signature of the update sequence is not weakly monotonic. The source of this problem is that $\text{history}(k)$ can depend on several dead objects in the past whose number, in general, can far exceed the number of active (undeleted) objects at time k . An object occurring in $\text{history}(k)$ is called *dead* if the deletion of that object has already occurred in the update sequence \bar{u} . It occurs in the history nevertheless.

We now wish to rectify this shortcoming. Our basic idea is very simple. Every time the number of dead objects in the history exceeds, say, twice the number of active objects in the history, we rebuild the history from scratch. Rebuilding is done by just adding the currently active objects in the same order in which they were added previously. (In practice, it might be preferable to add them in random order.) We use the semidynamic special case of our algorithm for this purpose. The resulting history is as if the dead objects were never encountered during the previous updates. After this, we proceed with the remaining updates pretending as if nothing happened, completely forgetting the dead objects. We keep on rebuilding the history in this fashion whenever required.

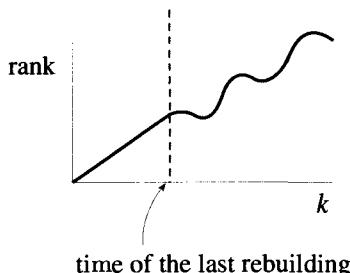


Figure 4.6: Signature of the filtered update sequence.

Analysis

Let \bar{u} be any update sequence. For any fixed time k , let \bar{u}_k denote the initial subsequence of \bar{u} of length k . Define the *filtered* update sequence \bar{w}_k at time k as follows (Figure 4.6). It is the subsequence of \bar{u}_k obtained by removing the updates involving all dead objects that were removed in the previous rebuilding operations. It has an initial sequence of additions in the last rebuilding operation, followed by the later updates in \bar{u}_k . In our new scheme, the history that exists at time k is actually the history of \bar{w}_k . By the very nature of our rebuilding strategy, \bar{w}_k has a weakly monotonic signature: If we scan \bar{w}_k from the last rebuilding operation, then the number of dead objects at any given time cannot exceed twice the number of active objects. The signature of \bar{w}_k can be completely determined from the signature of \bar{u}_k . If \bar{u}_k is a random sequence of updates, then \bar{w}_k is also a random sequence of updates. Hence, if we ignore the cost of rebuildings, the remaining cost of the algorithm can be estimated as in the previous sections simply by pretending that the sequence of updates under consideration has a weakly monotonic signature. In particular, the expected cost of the search through the rebuilt history is $O(\log^2 k)$ in the case of trapezoidal decompositions (Theorem 4.1.2) and $O(\log k)$ in the case of Voronoi diagrams (Theorem 4.2.4).

It remains to estimate the total cost of rebuildings. Recall that we rebuild the history every time the number of dead objects in it exceeds twice the number of currently active objects. Rebuilding is done by adding the currently active objects in the same order in which they were added previously. The number of these additions is clearly less than the number of dead objects. All these dead objects are removed. Hence, the cost of rebuilding can be equally distributed (amortized) among the dead objects. Since an object can die only once, it follows, in particular, that the total number of additions during all rebuildings is less than the length of \bar{u} . In other words:

Observation 4.4.1 *If $a(l)$ denotes the number of active objects at the time of the l th rebuilding, then $\sum_l a(l) \leq |\bar{u}|$.²*

This simple observation is basically the essence of our rebuilding strategy. If our original update sequence is random, the additions involved in any rebuilding operation are also random. Hence, the expected cost of a single rebuilding can be estimated as in the semidynamic setting of Chapter 3.

For example, consider the dynamic maintenance of Voronoi diagrams (Section 4.2). The l th rebuilding involves $a(l)$ additions of sites. Its expected cost is $O(a(l) \log a(l))$ by the semidynamic considerations (Section 3.3). By Observation 4.4.1, the total expected cost of rebuildings is $O(n \log n)$. In con-

²Notice that $|\bar{u}| \leq 2n$, if we do not allow repetitive addition of an object that has already been deleted. But the observation holds even if we allowed repetitive additions.

junction with Proposition 4.2.2 and Theorem 4.2.4, this implies the following bound for the dynamic scheme involving rebuildings.

Theorem 4.4.2 *The expected cost of maintaining a planar Voronoi diagram over a random (N, δ) -sequence is $O(n \log n)$.*

Next, we shall consider dynamic maintenance of trapezoidal decompositions. Let $I(l)$ denote the number of intersections among the $a(l)$ segments that are added during the l th rebuilding. The expected cost of the l th rebuilding is $O(I(l) + a(l) \log a(l))$ by the semidynamic considerations (Section 3.1). Because our original update sequence is random, the set of segments involved in the l th rebuilding is a random sample of N of size $a(l)$. Hence, the expected value of $I(l)$ is $O(a(l)^2 I/n^2)$, where I is the number of intersections among the segments in N (Lemma 3.1.3). By summing over l and combining with Observation 4.4.1, it follows that the total expected cost of rebuildings is $O(n \log n + I)$. In conjunction with Theorem 4.1.2, this implies the following.

Theorem 4.4.3 *The expected cost of maintaining a trapezoidal decomposition over a random (N, δ) -sequence is $O(n \log n + I)$.*

Exercises

4.4.1 Apply the rebuilding strategy to the point location structure for Voronoi diagrams described in Exercise 4.2.1.

4.4.2 Let $\text{span}(l)$ denote the total number of updates between the l th and the $(l+1)$ th rebuilding, if any. Show that $a(l) \leq c \text{span}(l)$, for some $c > 1$, for all but the highest l .

4.4.3 One disadvantage of our current rebuilding strategy is that the cost of the particular update that triggered the rebuilding operation can be high. This problem can be easily taken care of by using the following partial rebuilding method: Spread the execution of the $a(l)$ additions, in the l th rebuilding, uniformly and in parallel with, the next $a(l)/c \leq \text{span}(l)$ updates (Exercise 4.4.2), continuing to use the old history in the meanwhile. Elaborate this scheme in detail. Apply it in the dynamic maintenance of Voronoi diagrams and trapezoidal decompositions.

4.5 Deletions in history

The history, as maintained in the previous sections, needs to be rebuilt periodically to remove the dead objects in it. We describe an alternative method in this section which always maintains the history in a clean form. By this, we mean that the history never contains dead objects at any time. Whenever we encounter a deletion operation, we also delete the object from the history. Once this is done, the history should look as if the deleted object was never added in the first place.

We shall now elaborate this idea in a general setting. Let M denote the set of active, i.e., undeleted, objects at any given time. We shall denote the size of M by m . The set M can be naturally ordered according to the order in which the objects in M were actually added in the update sequence. Let S_j denote the j th object in this order. Let M^j denote the set of the first j objects. Now imagine adding the objects in M , one at a time, in this order. Intuitively, the history that we shall maintain will be the history of this sequence of additions.

A more formal definition of the history in the setting of configuration spaces can be given as in Section 4.3. We assume that we can associate with M a configuration space $\Pi(M)$. This configuration space depends on the problem under consideration. We shall denote the history associated with M by $\text{history}(M)$. Imagine adding the objects in M one at a time in the order described above. $\text{History}(M)$ is then defined just as in Section 4.3. But this time, there are no deletions in the sequence whose history is recorded. Hence, $\text{history}(M)$ looks just like the history in the semidynamic setting, as in Chapter 3. Intuitively, it is defined as follows. We consider the sequence $\Pi^0(M^1), \dots, \Pi^0(M^m) = \Pi^0(M)$. Here $\Pi^0(M^j)$ denotes the set of active configurations in the subspace $\Pi(M^j)$. $\text{History}(M)$ is essentially the history of the structural change encountered during this sequence, together with auxiliary link structures $\text{link}(S_k)$. It contains a *unique* node for each configuration in $\Pi(M)$ that becomes active during this sequence of additions. We say that $\sigma \in \Pi(M)$ becomes active during the sequence if it is active over some M^j . In that case σ , or more precisely its restriction, belongs to $\Pi^0(M^j)$. Sometimes we schematically denote the history by the following diagram:

$$\Pi^0(M^1) \xrightarrow{S_2} \Pi^0(M^2) \cdots \Pi^0(M^{k-1}) \xrightarrow{S_k} \Pi^0(M^k) \longrightarrow \cdots \Pi^0(M^m) = \Pi^0(M)$$

Here \rightarrow denotes the addition operation. $\text{History}(M)$ is uniquely defined once the order on M is fixed. Hence, in what follows, we shall think of M as an ordered set.

As a piece of notation, we define the *antecedents* of S_k to be the nodes in $\text{history}(M)$ that are killed by S_k . These correspond to the configurations in $\Pi^0(M^{k-1})$ that are destroyed by the addition of S_k (refer to the above schematic representation of history). We say that S_k is the *killer* of these nodes. We define the *descendants* of S_k to be the nodes in $\text{history}(M)$ created during the addition of S_k . These correspond to the newly created configurations in $\Pi^0(M^k)$ during the addition of S_k to $\Pi^0(M^{k-1})$. We also say that these nodes are created (born) by S_k and that S_k is the *creator* of these nodes. We label each node with its creator and killer (if any).

We have already remarked that $\text{history}(M)$ looks just like the history in the semidynamic setting, as in Chapter 3. In other words, it is as if only

additions were encountered in the past. It is to be used during the search just as in the semidynamic setting. In particular, when M is a set of points on the line, $\text{history}(M)$ becomes the binary tree, as defined in Section 1.3. It is a randomized binary tree if the objects in M were added in random order. The state of $\text{history}(M)$ is not independent of the past, as was the case for randomized binary trees in Section 1.3.1. We shall describe search structures with this property in Section 4.6.

Addition of a new object to $\text{history}(M)$ is conceptually the same as the addition in the semidynamic setting as in Chapter 3. Therefore, we only need to describe the deletion of an object from $\text{history}(M)$. This will be done through a sequence of operations called *rotations*. The reader will find it helpful to remember that this is a natural generalization of what we did in the case of randomized binary trees (Section 1.3.1).

Let $S \in M$ be the object that we wish to delete from $\text{history}(M)$. Let M' denote the set $M \setminus \{S\}$, with the naturally inherited order. Our goal is to obtain $\text{history}(M')$ from $\text{history}(M)$ in an efficient manner.

Conceptually, deletion of S from $\text{history}(M)$ is carried out as follows. Let l be the number of objects in M with order higher than S . For $i \leq l$, define $M(i)$ to be the ordered set obtained from M by moving S higher in the order by i places. Conceptually, we obtain $\text{history}(M')$ from $\text{history}(M)$ by moving S higher in the order step by step until it has the highest order. After this, deleting S is easy. We elaborate these two steps further:

1. Starting with $\text{history}(M) = \text{history}(M(0))$, we successively obtain a sequence

$$\text{history}(M(0)), \text{history}(M(1)), \dots, \text{history}(M(l)).$$

The basic operation of obtaining $\text{history}(M(i))$ from $\text{history}(M(i-1))$ is called a *rotation*. See Figure 4.7. In the figure, $B = M(i-1)$ and $C = M(i)$. Observe that C is obtained from B by switching the orders of S and S' . Here S' is the object next to S in the order on B . The history is accordingly changed to reflect this switch in the order.

2. Let $D = M(l)$. The object S has the highest order in D . Conceptually, D can be obtained from M' by adding S at the highest end of its order. This means we can think of $\text{history}(D)$ as being obtained from $\text{history}(M')$ by adding S after all objects in M' . So to obtain $\text{history}(M')$ from $\text{history}(D)$, we just have to imagine *undoing* this last (imaginary) addition of S . This is done by removing the descendants of S and $\text{link}(S)$. This last step is called *end deletion*.

There is one obvious snag in this scheme, namely, that it always needs l distinct rotations. This implies an $\Omega(l)$ lower bound on the cost of a deletion. This can be bad in the problems where one is looking for a polylogarithmic

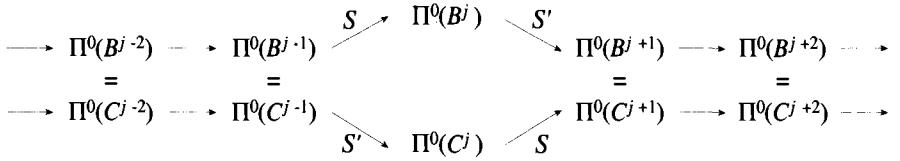


Figure 4.7: Rotation diagram.

bound on the expected update time (otherwise, it is fine). We can get around this difficulty as follows. Refer again to Figure 4.7. Let $B = M(i)$, for some $0 \leq i \leq l$. Let S' be the object just higher than S in the order on B . Let $C = M(i+1)$ be the order obtained by switching the orders of S and S' in B . We say that S and S' *commute* if $\text{history}(B)$ and $\text{history}(C)$ are identical. We want to get the rotations involving commuting pairs of objects for free. With this in mind, we refine our deletion scheme as follows.

Begin with $\text{history}(M)$. Let S' be the first object commuting with S with higher order than S . Assume that there is an oracle that tells us S' for free. We shall construct this oracle in a moment. The object S commutes with all objects between S and S' in the order. Hence, we can assume, without loss of generality, that S' is next to S in the order. Rotate S and S' , and keep on repeating this process until there is no noncommuting object with higher order than S . At this stage, we perform end deletion.

Let's turn to the construction of the oracle mentioned above. For this we stipulate that, for any ordered set B , the data structure $\text{history}(B)$ should satisfy a certain commutativity property. Let S and S' be any two objects in B such that S immediately precedes S' in the order. Let $C = B(S, S')$ be the order obtained from B by switching the orders of S and S' .

We make the following stipulation:

Commutativity Condition: $\text{History}(B)$ and $\text{history}(C)$ are identical data structures (i.e., S and S' *commute*) if no descendent of S in $\text{history}(B)$ is killed by S' . (Refer to the top row in Figure 4.7; we want that S and S' should commute when no configuration in $\Pi^0(B^j)$, which is created during the addition of S to $\Pi^0(B^{j-1})$, is killed during the addition of S' .)

Assuming this condition, the deletion algorithm can be further refined as follows. Suppose we want to delete an object S from $\text{history}(M)$. Initially, let Γ be the set of descendants of S in $\text{history}(M)$. For example, in the case of binary trees, Γ will consist of precisely two nodes that correspond to the two intervals in the linear partition $H(M^{j-1})$ adjacent to S . Here j denotes the order of S in M .

Let $\text{killer}(\Gamma)$ be the priority queue of the killers of the nodes in Γ . By a priority queue, we just mean a list sorted according to the ranks (orders) of

the objects in it. If the priority queue is maintained in the form of a skip list or a randomized binary tree, an item can be deleted or inserted in it in logarithmic time (with high probability, strictly speaking; but this makes no difference in what follows). With each object in $\text{killer}(\Gamma)$, we associate a list of pointers to the nodes in Γ that it kills.

In the example of binary trees, the size of $\text{killer}(\Gamma)$ is at most two. In general, it can be much larger.

We are now ready to give the general deletion algorithm. In the case of binary trees, the reader should check that it coincides with the deletion operation, as described in Section 1.3.1.

Algorithm 4.5.1 (Deletion)

1. Remove the first object S' in $\text{killer}(\Gamma)$. By the commutativity condition, S commutes with all objects in M , whose current ranks lie between S and S' . Hence, without loss of generality, we can assume that S' immediately succeeds S in the order.
2. Let $\Gamma(S')$ be the subset of Γ killed by S' . The subset $\Gamma(S')$ is available to us at this stage. Using this information, switch the orders of S and S' in M . Update $\text{history}(M)$ via rotation accordingly. Intuitively, the rotation takes place in the vicinity of $\Gamma(S')$. The exact nature of the rotation depends on the problem under consideration.
3. Replace the subset $\Gamma(S')$ of Γ by the new descendants of S in $\text{history}(M)$. Accordingly, update the priority queue $\text{killer}(\Gamma)$.
4. If $\text{killer}(\Gamma)$ is not empty, go to step 1.
5. *End deletion:* Now the order of S in M can be assumed to be the highest. Remove $\text{link}(S)$ and the descendants of S in $\text{history}(M)$.

At the end of the above algorithm, we get $\text{history}(M')$, where $M' = M \setminus \{S\}$.

Analysis

In the above scheme, the exact nature of rotation depends on the problem under consideration. The cost of rotation is generally governed by the *structural change* in that rotation. Let us elaborate what we mean by this. Refer again to the rotation diagram in Figure 4.7. There C is an ordered set obtained from B by switching the orders of S and S' . A node $\hat{\sigma}$ in $\text{history}(C)$ is called a newly formed antecedent of S (during this rotation) if it was not the antecedent of S in $\text{history}(B)$. There are two possibilities here. Either $\hat{\sigma}$ was absent in $\text{history}(B)$ or, even if it was present, it was not the antecedent of S in $\text{history}(B)$. In the first case, we certainly have to spend time in creating that node. In the latter case, we still need to visit $\hat{\sigma}$ and change the antecedent information associated with it. In both cases, $\hat{\sigma}$ must be considered

as contributing to the structural change in the rotation. Similarly, a node \hat{r} in $\text{history}(B)$ is called an old antecedent of S if it is no longer an antecedent of S in $\text{history}(C)$. Here, too, there are two possibilities. Either this node is absent in $\text{history}(C)$ or, even if it is present, it is no longer an antecedent of S . In the first case, we certainly need to spend time in destroying the node. In the second case, we need to spend time in changing the antecedent label associated with the node. Thus, a node of this kind also contributes to the structural change. We define the new and old descendants of S similarly. We also make similar definitions for S' . We define the structural change during a rotation to be the number of new or old antecedents or descendants of S plus the similarly defined number for S' .

The total cost of deletion is governed by the total structural change. This is defined as the sum of the structural changes during all rotations. Its expected value is bounded in the following theorem.

Let $e(i, M)$ denote the expected size of $\Pi^0(B)$, where B is a random sample of M of size i .

Theorem 4.5.2 Suppose the ordering of the objects in M is random. Then the total expected structural change during a deletion of a random object from $\text{history}(M)$ is bounded, within a constant factor, by $\frac{1}{m} \sum_{i=1}^m e(i, M)/i$.

Remark. Intuitively the theorem should be clear. The sum $\sum_{i=1}^m e(i, M)/i$ is essentially the expected size of $\text{history}(M)$ (Theorem 3.4.5). What the theorem is saying is that the average structural change during a random deletion from M is approximately $(1/m)$ th fraction of the expected size of the history.

Proof. If you wish, you can skip this proof on the first reading.

Let S denote the deleted object. Fix an integer $j < m$. Denote S_{j+1} by S' . There can be at most one rotation involving S and S' , if any. Let us analyze the cost of this rotation between S and S' . By the cost, we mean the structural change in that rotation. The cost is defined to be zero if there is no rotation between S and S' , which happens if S has higher order than S' in M .

So let us now consider the case when $S' = S_{j+1}$ is higher than S in the order—otherwise, the cost is zero by definition. The cost is also zero if the rotation between S and S' comes for free, i.e., if S and S' commute, when S moves past S' in the order. Refer to Figure 4.7, which illustrates the rotation between S and S' . In this diagram, B denotes the order obtained from M by moving S up in the order just below S' , and C denotes the order obtained from B by switching the orders of S and S' . Note that

$$B^i = C^i, \text{ for } i \neq j, \text{ and } B^i = C^i = M^i, \text{ for } i > j,$$

if we ignore the orderings on these sets.

The rotation transforms history(B) to history(C). Consider a node $\hat{\sigma}$ in either history(B) or history(C). Denote the corresponding configuration by σ . The node $\hat{\sigma}$ contributes to the structural change during the rotation iff one of the following four possibilities holds (Figure 4.7):

1. $\sigma \in \Pi^0(B^{j+1})$ and both S and S' conflict with σ . In this case, the antecedent label of $\hat{\sigma}$ changes from S to S' . We can also think of σ as a configuration in $\Pi^2(B^{j+1}) = \Pi^2(M^{j+1})$, where S and S' are its only stoppers in M^{j+1} . Here $\Pi^2(M^{j+1})$ denotes the set of configurations in $\Pi(M^{j+1})$ with level 2.
2. $\sigma \in \Pi^0(B^j)$, and, in addition, it is a descendent of S and an antecedent of S' . This implies that S is one of the triggers associated with σ and S' conflicts with σ . In this case, the node $\hat{\sigma}$ is to be removed, because it is no longer present in history(C). The configuration σ can also be thought of as a configuration in $\Pi^1(B^{j+1}) = \Pi^1(M^{j+1})$; S is one of its triggers and S' is its only stopper in M^{j+1} .
3. $\sigma \in \Pi^0(C^j)$, and, in addition, it is a descendent of S' and an antecedent of S . This implies that S' is one of the triggers associated with σ and S conflicts with σ . In this case, the node $\hat{\sigma}$ has to be newly created, because it was not present in history(B). The configuration σ can also be thought of as a configuration in $\Pi^1(C^{j+1}) = \Pi^1(M^{j+1})$; S' is one of its triggers and S is its only stopper in M^{j+1} .
4. $\sigma \in \Pi^0(B^{j+1}) = \Pi^0(C^{j+1}) = \Pi^0(M^{j+1})$ and S , as well as S' , is its trigger. In this case, the descendent label of $\hat{\sigma}$ changes from S' to S .

Note that in each case, σ can be thought of as a configuration in $\Pi^c(M^{j+1})$, with $c = 0, 1$, or 2 . Fix M^{j+1} . If we sum the number of configurations satisfying any of the previous four conditions, by letting S and S' range over all pairs of objects in M^{j+1} , then the resulting sum is bounded, within a constant factor, by $\pi^0(M^{j+1}) + \pi^1(M^{j+1}) + \pi^2(M^{j+1})$. This is because the number of triggers is bounded and the number of stoppers of any such configuration is at most 2. Hence, each configuration in $\Pi^c(M^{j+1})$, $c \leq 2$, is counted a bounded number of times. By our assumption, the ordering on M is random. The deleted object is also chosen randomly from M^{j+1} . If we assume that the deleted object belongs to M^{j+1} , then each pair of objects in M^{j+1} is equally likely to occur as S and S' . In that case, the expected structural change in a rotation involving the $(j+1)$ th object in M is of the order of

$$\frac{\pi^0(M^{j+1}) + \pi^1(M^{j+1}) + \pi^2(M^{j+1})}{(j+1)j}.$$

This is conditional on a fixed M^{j+1} and the assumption that the deleted object indeed belongs to M^{j+1} . Since the deleted object is randomly chosen

from M , the probability that it belongs to a given M^{j+1} is $(j+1)/m$. Hence, the expected structural change, without this latter assumption, is obtained by multiplying the above expression by $(j+1)/m$. It yields

$$\frac{\pi^0(M^{j+1}) + \pi^1(M^{j+1}) + \pi^2(M^{j+1})}{mj}.$$

Now we use the fact that M^{j+1} is a random sample of M of size $j+1$. Hence, the expected cost of the rotation involving the $(j+1)$ -object in M is

$$\frac{e^0(j+1, M) + e^1(j+1, M) + e^2(j+1, M)}{mj}.$$

Here $e^c(i, M)$ denotes the expected size of $\pi^c(B)$, where B is a random sample of M of size i . If c is bounded, as in the present case, it is $O(e(i/2, M))$ (Lemma 4.3.2). Hence, the previous expression is of the order of

$$e\left(\frac{j+1}{2}, M\right) / \left(m \frac{j+1}{2}\right).$$

The theorem follows by summing over all j and changing the summation index. \square

In the next few sections, we give a few applications of the preceding deletion technique.

4.5.1 3D convex polytopes

In Section 3.2.2, we gave a semidynamic algorithm for maintaining three-dimensional convex polytopes. In this section, we shall extend the algorithm to the fully dynamic setting.

Let M denote the set of undeleted half-spaces existing at any given time. Let $H(M)$ denote the convex polytope formed by intersecting these half-spaces. Our goal is to maintain $H(M)$ in a dynamic fashion. In addition, we shall also maintain $\text{history}(M)$. $\text{History}(M)$ records the history of the additions of the half-spaces in M , in the order they were added, ignoring the deleted half-spaces altogether. It is defined exactly as in the semidynamic setting (Section 3.2.2). It contains a unique node for each vertex that comes into existence during the computation of the sequence $H(M^1), H(M^2), \dots, H(M^m) = H(M)$. Here M^i denotes the set of the first i half-spaces in M . Let S_k denote the k th half-space in M . The killed and the newly created nodes at time k can be identified with the vertices of the cap defined as: $\text{cap}(S_k) = H(M^{k-1}) \cap \bar{S}_k$ (Figure 3.5). Here \bar{S}_k denotes the complement of S_k . Recall that $\text{link}(S_k)$ just corresponds to the adjacencies among these vertices.

Addition of a new half-space to M is done exactly as in the semidynamic setting (Section 3.2.2): (1) Find a vertex of $H(M)$ in conflict with the new half-space by search through the history. (2) Update $H(M)$. (3) Add the new half-space at the end of the history.

Let us turn to the deletion operation. Suppose we want to delete a half-space $S \in M$. This will be done by applying the general deletion scheme given before (Algorithm 4.5.1). First, we bring S to the end of the ordering on M via rotations. After this, we just perform end deletion.

Note that, at any given time, M can contain several redundant half-spaces. These can become nonredundant during the deletion. This is what made it difficult to apply the earlier dynamic scheme in Section 4.3 (Exercise 4.2.2). In the present scheme, this will not be a problem because, intuitively, once S is brought to the end of the order on M , the polytope $H(M \setminus \{S\})$ is implicitly present in the history just “behind” $H(M)$. That is why updating $H(M)$ to $H(M \setminus \{S\})$ is no problem.

It is easy to verify that $\text{history}(M)$ satisfies the commutativity condition. So it only remains to describe the rotation operation.

For the sake of clarity, we shall describe the rotation operation in general terms without referring to M or S . Let A be any ordered set of half-spaces. Let R and R' be any two consecutive half-spaces in the ordering on A . Assume that they do not commute. Let \bar{A} be the order obtained from A by switching the orders of R and R' . We shall show how to update $\text{history}(A)$ to $\text{history}(\bar{A})$. This is done in the following steps. (Figure 4.8 illustrates the rotation operation for dimension two.)

Let j be the order of R in A . The order of R' is $j + 1$. Assume that $H(A^{j-1})$ is bounded. We have already seen how to ensure boundedness in Section 3.2. Let $\Gamma(R')$ denote the set of the descendants of R in $\text{history}(A)$ that are killed by R' . The deletion Algorithm 4.5.1 ensures that $\Gamma(R')$ is available to us prior to the rotation (refer to its second step). The nodes in $\Gamma(R')$ correspond to the vertices of $\text{cap}(R)$ that are adjacent to R and are in conflict with R' . In other words, these vertices lie on the boundary ∂R and they also lie within the complement \bar{R}' . The vertex v in Figure 4.8(b) is one such example.

We rotate R and R' as follows (Figure 4.8):

1. Consider the intersection $I(R, R') = \text{cap}(R) \cap \bar{R}' = H(A^{j-1}) \cap \bar{R} \cap \bar{R}'$.

Assuming that $H(A^{j-1})$ is bounded, the edge skeleton of $I(R, R')$ is a connected subgraph of the edge skeleton of $\text{cap}(R)$. We know $\text{cap}(R)$ because it is the link structure associated with R . Hence, $I(R, R')$ can be determined as follows by a search within the edge skeleton of $\text{cap}(R)$.

We start the search at the vertices in the set $\Gamma(R')$, which we already know. We take care of not crossing into the half-space R' at any time.

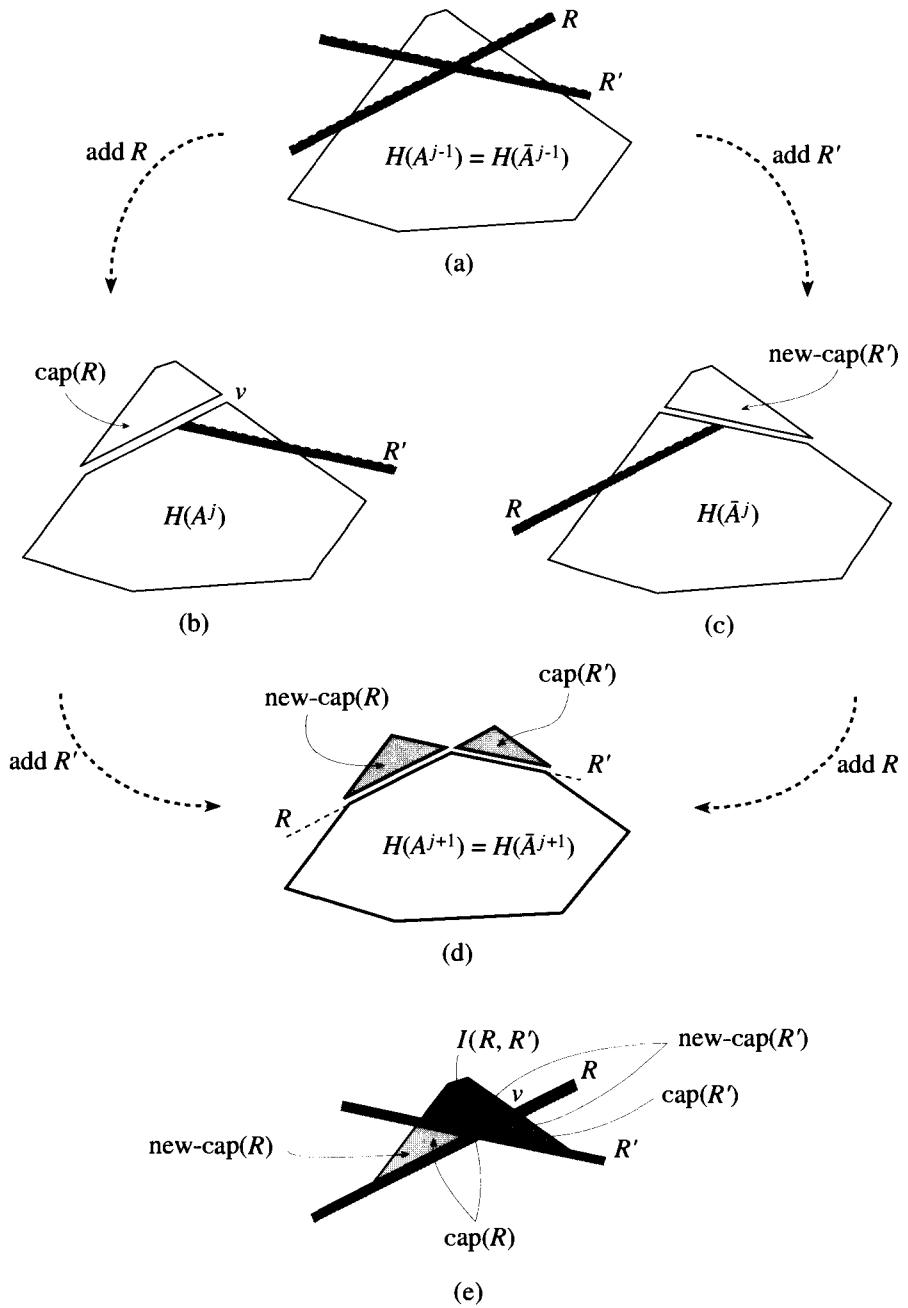


Figure 4.8: Rotation.

This search visits all edges and vertices of $I(R, R')$ in $O(|I(R, R')|)$ time. Here $|I(R, R')|$ denotes the size of $I(R, R')$.

2. Update $\text{cap}(R)$ and $\text{cap}(R')$: Let $\text{new-cap}(R)$ and $\text{new-cap}(R')$ denote the new caps that get associated with R and R' after the rotation as their new link structures. It is easy to see that $\text{new-cap}(R)$ is obtained from $\text{cap}(R)$ by “spitting off” the intersection $I(R, R')$ from it (Figure 4.8(e)). Similarly, $\text{new-cap}(R')$ is obtained from $\text{cap}(R')$ by “gluing” $I(R, R')$ to it and erasing the common boundary. We leave the routine details of this operation to the reader.

Analysis

It is easy to see that the time taken in rotating a noncommuting pair R, R' of half-spaces is proportional to $|I(R, R')|$. This does not take into account the cost of maintaining the priority queue of killers in the deletion Algorithm 4.5.1. Since the addition or deletion of any item into this queue takes logarithmic time, this adds another log factor to the above cost.

Note that $|I(R, R')|$ is essentially the structural change during the rotation. Its expected value can be estimated by applying Theorem 4.5.2 to the configuration space of feasible vertices over the half-spaces in M (Example 3.4.2). It then follows that the total expected structural change during a random deletion from $\text{history}(M)$ is

$$\frac{1}{m} \sum_{i=1}^m \frac{e(i, M)}{i}. \quad (4.12)$$

Here $e(i, M)$ denotes the expected number of vertices of the convex polytope $H(B)$, where $B \subseteq M$ is a random sample of size i . Since the size of a three-dimensional convex polytope is linear in the number of bounding half-spaces (Section 2.4.1), $e(i, M) = O(i)$. Hence, it follows from (4.12) that the expected total structural change during a random deletion is $O(1)$. We need to add an extra log factor to take into account the maintenance of the priority queue of killers. It follows that the expected cost of a random deletion is $O(\log m)$.

Similarly, the expected cost of a random addition is $O(\log m)$. This is proven just as in the semidynamic setting. It is already implicit in Section 3.2.2. (Alternatively, this also follows from Theorem 4.3.5 because it implies that the expected cost of conflict search during an addition is $O(\log m)$.)

$\text{History}(M)$ can be augmented as in Section 3.2.3 so that it can also be used for locating the optimum of a linear function on $H(M)$. As we saw there, the query cost is $O(\log^2 m)$ for most orderings of the objects in M . The augmentation consists in associating a radial tree with the bottom of $\text{cap}(S_k)$, for each k . By the bottom of $\text{cap}(S_k)$, we mean its facet supported

by ∂S_k . We leave it to the reader to check that during a rotation involving S_k , this radial tree can be updated along with $\text{cap}(S_k)$ in the same order of time.

To summarize:

Theorem 4.5.3 *The expected cost of a random update on $\text{history}(M)$ and the convex polytope $H(M)$ is $O(\log m)$. The cost of locating the optimum of a linear function on $H(M)$ is $O(\log^2 m)$ for most update sequences.*

Exercise

4.5.1 Check that during a rotation involving S_k , the associated radial tree can be updated along with $\text{cap}(S_k)$ in the same order of time. (Hint: Use the concatenate and split operations (Exercises 1.3.7 and 1.4.7).)

4.5.2 Trapezoidal decompositions

We have already given in this chapter one dynamic point location structure for trapezoidal decompositions. In this section, we shall give another. It is based on the technique of deletion via rotations. Incidentally, its query time is better by a log factor.

Let M denote the set of undeleted segments existing at any given time. Let $H(M)$ denote the trapezoidal decomposition formed by them. Our goal is to maintain $H(M)$. In addition, we shall also maintain $\text{history}(M)$. $\text{History}(M)$ records the history of the additions of the segments in M , in the order in which they were added, ignoring the deleted segments altogether. It is defined exactly as in the semidynamic setting (Section 3.1.1). It contains a unique node for each trapezoid that was created during the computation of the sequence $H(M^1), H(M^2), \dots, H(M)$. Here M^i denotes the set of the first i segments in M . Let S_k denote the k th segment in M . For each k , the linkage structure $\text{link}(S_k)$ is defined as in the semidynamic setting (Section 3.1.1). It associates with every killed node at time k pointers to its children. We assume that these pointers are bidirectional. For the sake of dynamization, we shall need to augment this link structure as follows. Define the left (right) *sibling* of a trapezoid Δ in $H(M^k)$ to be the trapezoid that shares the left (resp. right) vertical side of Δ (if any). We assume that each killed node at time k is given bidirectional pointers to the nodes that correspond to its siblings in $H(M^k)$.

Addition of a new segment S to M is done exactly as in the semidynamic setting (Section 3.1.1): (1) Find a trapezoid of $H(M)$ containing an endpoint of S . This is done by a search through $\text{history}(M)$. (2) Update $H(M)$. (3) Add the new segment to the end of the history. The only new thing that

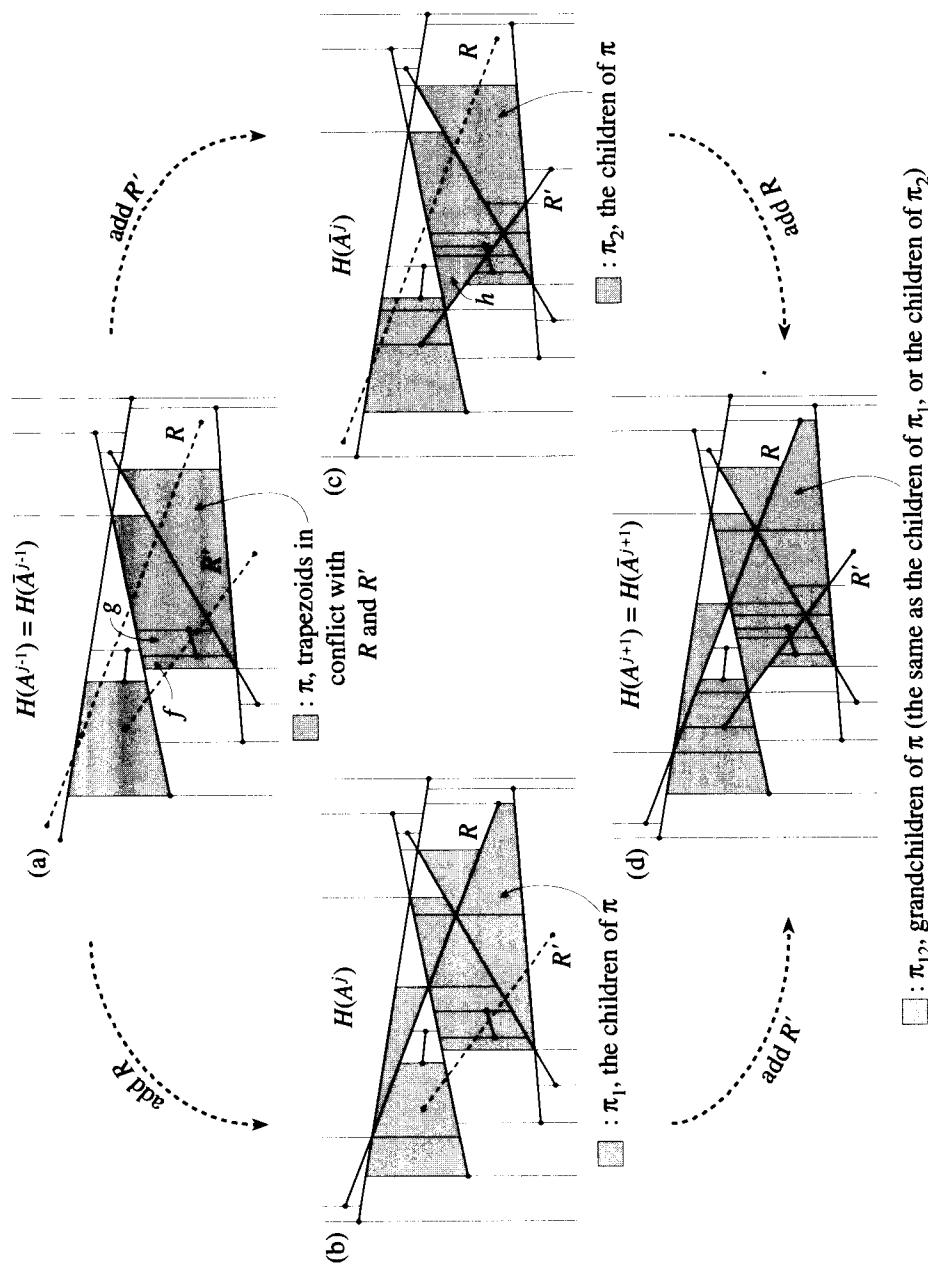


Figure 4.9: Rotation.

we need to worry about is the pointers among the siblings. But this only requires minor extensions, which we shall leave to the reader.

Let us turn to the deletion operation. Suppose we want to delete a segment $S \in M$. This can be done by applying the general deletion scheme given in Algorithm 4.5.1. First, we bring S to the end of the ordering on M via rotations. After this, we simply perform end deletion.

It is easy to verify that $\text{history}(M)$ satisfies the commutativity condition. So it only remains to describe the rotation operation.

For the sake of clarity, we shall describe the rotation operation in general terms without referring to M or S . Let A be any ordered set of segments. Let R and R' be any two consecutive segments in the ordering on A . Assume that they do not commute. Let \bar{A} be the order obtained from A by switching the orders of R and R' . We shall show how to update $\text{history}(A)$ to $\text{history}(\bar{A})$. This is done in the following steps (Figure 4.9)

Let j be the order of R in A . The order of R' is $j+1$. Let $\Gamma(R')$ denote the set of the descendants of R in $\text{history}(A)$ that are killed by R' . The deletion Algorithm 4.5.1 ensures that $\Gamma(R')$ is available to us prior to the rotation (refer to its second step). In Figure 4.9(b) the nodes in $\Gamma(R')$ correspond to the shaded trapezoids intersecting R' . Let π denote the nodes corresponding to the trapezoids in $H(A^{j-1})$ in conflict with both R and R' . The nodes in π can be accessed through the parent pointers associated with the trapezoids in $\Gamma(R')$. This takes time proportional to $\sum_{\Delta} \text{face-length}(\Delta)$, where Δ ranges over all trapezoids in $\Gamma(R')$.

Let us first examine the effect of rotation, confining our attention to a single trapezoid $f \in \pi$ (Figure 4.10). Our goal is to replace the subtree within $\text{history}(A)$ which corresponds to the “local” history of adding R and R' in that order (Figure 4.10(e)) with the new subtree which corresponds to the “local” history of adding R' and R in this reverse order (Figure 4.10(f)). Note that the sizes of these local histories are bounded by a constant. This is because the number of children of any node is bounded. Also note that the grandchildren of f do not change during the rotation. It is only the children which can change. Accordingly, we have to allocate new nodes for the newly formed children. For each $f \in \pi$, the local nature of rotation can be determined in $O(1)$ time. However, there remains the problem of patching up these local effects globally. The main problem is that the trapezoids in π can share children. In Figure 4.10(a), f shares its child g_2 (Figure 4.10(c)) with its left sibling. The trapezoids f and g in Figure 4.9(a) share a common child h (Figure 4.9(c)). In general, several trapezoids in π can share a common child and one has to be careful to allocate just one node for each such newly created shared child. This means we need to identify the common children among the newly formed children of the trapezoids in π . For this we separately

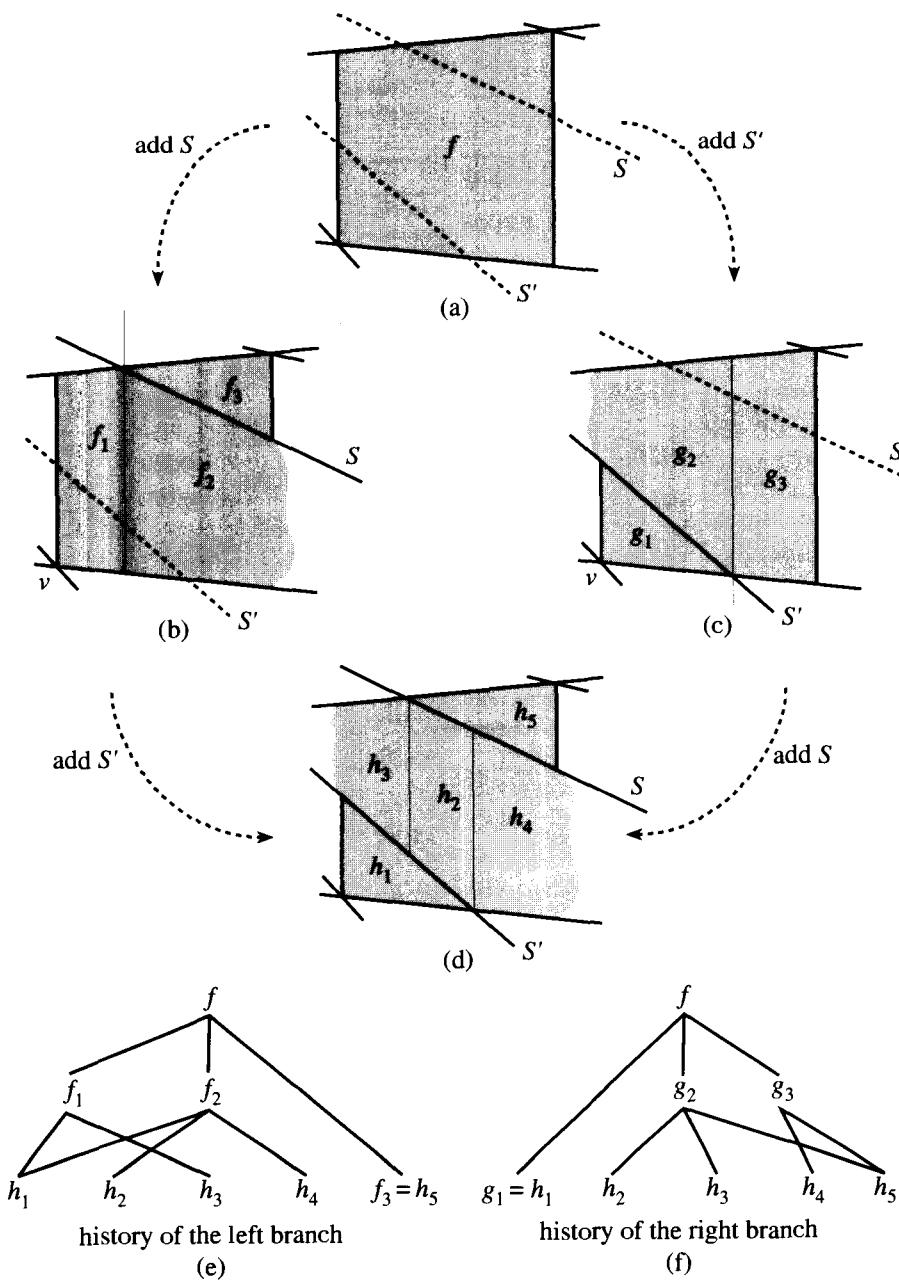


Figure 4.10: Effect of a rotation within a single trapezoid.

determine, for each trapezoid in π , which of its children are shared by its siblings. We then merge this information. This takes $O(1)$ time per trapezoid in π .

Finally, we deallocate all nodes in $\Gamma(R')$ because they will no longer be present in the new history.

Analysis

It follows from the preceding discussion that the cost of rotating R and R' is proportional to

$$|\pi| + \sum_{\Delta} \text{face-length}(\Delta) = O\left(\sum_{\Delta} \text{face-length}(\Delta)\right),$$

where Δ ranges over all trapezoids in the set $\Gamma(R')$. If the latter sum were proportional to $|\Gamma(R')|$, we would be lucky. The size $|\Gamma(R')|$ is certainly bounded by the structural change during this rotation. Thus, Theorem 4.5.2 will be directly applicable. Unfortunately, the above sum need not be proportional to $|\Gamma(R')|$ because the face-length of a trapezoid can be arbitrarily large in principle. We can get around this difficulty as follows. Let $\Sigma(M)$ be the configuration space of feasible racquets over M (Example 3.4.4). Just for the purpose of analysis, consider the imaginary history of the sequence of configuration spaces $\Sigma(M^1), \Sigma(M^2), \dots, \Sigma(M^m) = \Sigma(M)$. Then notice that $\sum_{\Delta} \text{face-length}(\Delta)$ is bounded by the number of racquets whose faces belong to $\Gamma(R')$. All these racquets are killed by R' . In other words, the cost of actual rotation is bounded by the structural change during the accompanying rotation on this imaginary history. Thus, we are in a position to apply Theorem 4.5.2 to the configuration space $\Sigma(M)$. It implies that the total expected cost of rotations during a random deletion is proportional to

$$\frac{1}{m} \sum_i \frac{e(i, \Sigma(M))}{i}, \quad (4.13)$$

ignoring a logarithmic factor that can arise due to the maintenance of the priority queue. Here $e(i, \Sigma(M))$ denotes the number of racquets active over a random sample $B \subseteq M$ of size i . The number of racquets active over B is proportional to the size of the trapezoidal decomposition $H(B)$ (Example 3.4.4). This in turn is proportional to i plus the number of intersections among the segments in B . Since B is a random sample of M of size i , the expected number of such intersections is $O([i^2/m^2]I(M))$, where $I(M)$ is the number of intersections among the segments in M (Lemma 3.1.3). It follows that

$$e(i, \Sigma(M)) = O\left(i + \frac{i^2}{m^2} I(M)\right).$$

From (4.13), it follows that the expected cost of rotations during a random deletion is bounded within a logarithmic factor by

$$\frac{1}{m} \sum_i 1 + \frac{i}{m^2} I(M) \leq 1 + \frac{I(M)}{m}.$$

The expected cost of a random addition to M is bounded just as in the semidynamic setting. It is $O(\log m + I(M')/m)$, where M' is the resulting set of segments after addition, and $I(M')$ denotes the number of intersections among the segments in M' . This bound is implicit in Section 3.1.1. The logarithmic term reflects the cost of locating an endpoint of the added segment. We have already seen (Section 3.1.1) that the cost of point location using history is logarithmic for most orderings on M .

To summarize:

Theorem 4.5.4 *The expected cost of a random deletion from history(M) is $O([1+I(M)/m] \log m)$. The expected cost of a random addition to history(M) is $O(\log m + I(M')/m)$, where M' is the resulting set of segments after addition. In particular, it follows that the expected cost of executing a random (N, δ) -sequence is $O(n \log n + I \log n)$, where I is the number of intersections among the segments in N .*

The cost of point location using history(M) is $O(\log m)$ for most update sequences.

Exercise

4.5.2 A special case arises when all segments in M are infinite lines. In this case, Theorem 4.5.4 implies that the expected cost of a random update on history(M) is $O(m)$. Show that in this special case, the expected cost of any update, not just a random one, is $O(m)$. Here the expectation is solely with respect to randomization in the data structure. (Hint: There are at most m rotations. Hence, it suffices to show that the expected cost of each rotation is $O(1)$. There is no need for priority queue in this case. Why? We have already seen that the cost of rotating R and R' in history(A) is proportional to $\sum_f \text{face-length}(f)$, where f ranges over all trapezoids in $\Gamma(R')$. Show that its expected value is $O(1)$. You will need to use Lemma 4.3.2.)

4.6 Dynamic shuffling

We have already seen how history can be used for answering point location queries. One drawback of the techniques so far is that they provide a guarantee on the query time on most but not all update sequences. Let us elaborate what we mean by that. For example, consider point location in trapezoidal decompositions. Let M denote the set of undeleted segments existing at any

given time. We have seen that the cost of point location using $\text{history}(M)$ is $O(\log m)$ on most update sequences. What this means is that if the order of additions of the segments in M were random, then the cost of point location would be $O(\log m)$ with very high probability. The order of additions is governed by the actual update sequences. The update sequences in our model are drawn from a uniform model, which treats all objects equally. Hence, this is the same as saying that the cost of point location is $O(\log m)$ for most update sequences—i.e., for all but a $(1/g(m))$ th fraction of the update sequences, where $g(m)$ is a large-degree polynomial. But what if we want a good query time on every update sequence? By a good query time, we generally mean (poly)logarithmic query time. We shall be satisfied if the query time is (poly)logarithmic with high probability. But here the high probability should solely be with respect to randomization in the data structure. No assumption about the update sequence should be made, just as in the case of skip lists or randomized binary trees (Sections 1.4 and 1.3.1). There, too, the cost of point location was logarithmic with high probability with respect to the randomization in the data structure, and no assumption about the update sequence was made. In this section, we shall give a technique, called *dynamic shuffling*, which generalizes the dynamization technique underlying randomized binary trees (Section 1.3.1). Just as in the case of randomized binary trees, it will ensure that the query cost is good with high probability for every update sequence. The probability is solely with respect to randomization in the data structure.

What about the cost of updates? This is something that is not entirely in our hands. We have already seen that in several problems, an adversary can come up with an update sequence that forces unusually large structural change, thereby slowing down any dynamic algorithm. Hence, for the same reasons as before, we shall be able to prove good bounds only on the costs of random updates. If the cost for the actual update deviates from the expected cost of a random update, it is quite often tolerable. On the other hand, query times are generally more critical. That is why we want to ensure that they are small regardless of the update sequence.

Let us now describe the dynamic shuffling technique. It consists in a simple modification to the technique used in Section 4.5. Let M denote the set of undeleted objects existing at any given time. Recall how $\text{history}(M)$ was defined in Section 4.5: We imagined adding the objects in M in the same order as in the actual update sequence. $\text{History}(M)$ recorded the history of this sequence of addition. In contrast, our new search structure is defined as follows: We choose a random ordering (permutation) of the objects in M . This order is completely independent of the actual order in which the objects in M were added. We now record the history of this imaginary sequence of

additions. The state of the resulting search structure is independent of the actual past (history). It would be certainly misleading to denote the resulting search structure by $\text{history}(M)$, because it has nothing do with the actual history. For this reason, we shall denote it by $\text{shuffle}(M)$. The randomized binary trees in Section 1.3.1 were built by the exact same technique.

The only difference between $\text{history}(M)$ and $\text{shuffle}(M)$ is that the ordering on M is the actual ordering in the first case and an imaginary ordering in the second case. But this difference is all that is needed to ensure a good query time on every sequence of updates. The reason is the following. The construction of $\text{shuffle}(M)$ is based on a sequence of additions that is random regardless of the actual update sequence. Hence, the high-probability bound for the query time remains valid for every update sequence. Moreover, this probability is solely with respect to the randomization in the data structure. Thus, in the case of trapezoidal decompositions, it will follow that the cost of point location using $\text{shuffle}(M)$ is $\tilde{O}(\log m)$ for any update sequence. In the case of convex polytopes, it will follow that the cost of locating the optimum of a linear function using $\text{shuffle}(M)$ is $\tilde{O}(\log^2 m)$ for any update sequence.

We now give a more formal definition of $\text{shuffle}(M)$ along the lines of randomized binary trees (Section 1.3.1). For every object in M , randomly choose a *priori*ty (a real number) from the uniform distribution on the interval $[0, 1]$. This naturally puts a priority order on M . Since all objects are treated equally, every ordering of M is equally likely. Let S_k denote the k th object in the priority order on M . Let M^i denote the set of the first i objects S_1, \dots, S_i . We imagine adding the objects in M in the increasing order of their priorities. $\text{Shuffle}(M)$ is defined to be the history of this imaginary sequence of additions. It is defined for any configuration space $\Pi(M)$ just as we defined $\text{history}(M)$ earlier.

Let us turn to the updates. Deletion of an object S from $\text{shuffle}(M)$ is just like deletion from $\text{history}(M)$: We first bring S to the end of the order by successive rotations. After this, it is deleted by end deletion.

The addition operation is different. Consider addition of a new object S to $\text{shuffle}(M)$. Let $M' = M \cup \{S\}$. We choose a priority for S randomly and independently from the uniform distribution on the interval $[0, 1]$. After this, the addition operation is conceptually just the reverse of the deletion operation. First, we imagine that S has the highest priority in M . Accordingly, we add S to the end just like we add S to $\text{history}(M)$. This step of adding S to the end of the priority order on M is called *end addition*. It yields $\text{shuffle}(M')$, wherein the priority of S is assumed to be the highest. After this we use rotations, just as in the deletion operation, but in the backward direction until S is in its proper place in the priority order.

We now give a more detailed description of the addition algorithm. It is the opposite of the deletion Algorithm 4.5.1. The reader will find it illuminating to compare the two. Let $\text{shuffle}(M')$ be obtained as before by adding S to the end of $\text{shuffle}(M)$. We want to move S to its proper place in the priority order. Initially, let Γ be the set of antecedents of S in $\text{shuffle}(M')$. For example, in the case of binary trees, Γ will consist of one node that corresponds to the interval containing S in the linear partition $H(M)$.

Let $\text{creator}(\Gamma)$ be the priority queue of the creators of the nodes in Γ . With each object in $\text{creator}(\Gamma)$, we associate a list of pointers to the nodes in Γ born (created) by it.

Let us again consider the case of binary trees. In this case, we have seen that initially Γ contains just one node. Let I be the interval in $H(M)$ that corresponds to this node. The creator of I is its endpoint that was added last, i.e., after the other endpoint. For binary trees, $\text{creator}(\Gamma)$ will initially consist of just this endpoint. In general, the size of $\text{creator}(\Gamma)$ can be large.

We are now ready to describe backward rotations in the case of addition. For randomized binary trees, the reader should check that this coincides with the procedure defined in Section 1.3.1.

Algorithm 4.6.1 (*Rotations during addition*)

1. Remove the last object S' in $\text{creator}(\Gamma)$, i.e., the object with the highest priority. By the commutativity condition, S commutes with all objects in M' which lie between S and S' in the current priority order. Hence, without loss of generality, we can assume that S' immediately precedes S in the priority order.
2. Let $\Gamma(S')$ be the subset of Γ born (created) by S' . The subset $\Gamma(S')$ is available to us at this stage. Using this information, switch the orders of S and S' in M' . Update the history using rotation to reflect this switch. Intuitively, this rotation takes place in the vicinity of $\Gamma(S')$. The exact nature of rotation will depend on the problem under consideration.
3. Replace the subset $\Gamma(S')$ of Γ by the new antecedents of S in $\text{shuffle}(M')$. Accordingly, update the priority queue $\text{creator}(\Gamma)$.
4. If the actual priority of S (chosen in the beginning of the addition procedure) is not higher than all objects in $\text{creator}(\Gamma)$, go to step 1.

At the end of the above algorithm, we obtain $\text{shuffle}(M')$ reflecting the correct priority of S .

Analysis

Let us turn to the expected cost of maintaining this structure during a random sequence of updates. Since $\text{shuffle}(M)$ records the history of a random sequence of additions, it follows from Theorem 4.5.2 that:

Theorem 4.6.2 *The total expected structural change during the deletion of a random object from $\text{shuffle}(M)$ is bounded, within a constant factor, by $(1/m) \sum_{i=1}^m e(i, M)/i$.*

Now, it only remains to analyze the cost of rotations during an addition. This is governed by the total structural change during the rotations, i.e., the sum of the structural changes during all rotations. Its expected value can be estimated backwards as follows. Let S denote the object that is added to $\text{shuffle}(M)$. Let $M' = M \cup \{S\}$. Now imagine deleting S from $\text{shuffle}(M')$. Note that the total structural change that takes place, as we transform $\text{shuffle}(M)$ to $\text{shuffle}(M')$, is the same as the total structural change that takes place, as we imagine transforming $\text{shuffle}(M')$ back to $\text{shuffle}(M)$. Assume that the update sequence under consideration is random. Then, conditional on a fixed M' (not M), each object in M' is equally likely to occur as S . Hence, applying Theorem 4.5.2 to this imaginary random deletion, we get the following.

Theorem 4.6.3 *The expected total structural change in rotations, as we pass from $\text{shuffle}(M)$ to $\text{shuffle}(M')$, is bounded, within a constant factor, by $(1/m') \sum_{i=1}^{m'} e(i, M')/i$. Here it is assumed that each object in M' is equally likely to occur as an added object.*

Let us now axiomatize the cost of addition or deletion in $\text{shuffle}(M)$ to be proportional to the accompanying total structural change. Combining the above theorem with Theorem 4.6.2, we get the following.

Theorem 4.6.4 *The cost of maintaining $\text{shuffle}(M)$ on a random (N, δ) -sequence is*

$$\sum_{j=1}^{|\bar{u}|} \frac{1}{r(j)} \sum_{i=1}^{r(j)} \frac{e(i)}{i}, \text{ where } e(i) = e(i, N).$$

Proof. Just observe that M existing at any time j during the execution of \bar{u} is itself a random sample of N of size $r(j)$. Hence, a random sample of M of size i can also be regarded as a random sample of N of size i . \square

Let us return to the three-dimensional convex polytopes, as treated in Section 4.5.1. Let M denote the set of half-spaces existing at any given time. Let us bound the expected cost of maintaining $\text{shuffle}(M)$ over a random (N, δ) -sequence. The expected cost of a deletion in this sequence is $O(\log m)$, where m denotes the number of undeleted half-spaces at that time. The same bound also holds for addition. This follows as in Section 4.5.1 by applying Theorem 4.6.3 instead of Theorem 4.5.2. As we have already remarked, the cost of locating the optimum of a linear function is now $\tilde{O}(\log^2 m)$, regardless of the update sequence.

Finally, let us reconsider point location in trapezoidal decompositions, treated in Section 4.5.2. Let M denote the set of segments existing at any given time. Let us bound the expected cost of maintaining $\text{shuffle}(M)$ over a random (N, δ) -sequence. It follows as in Section 4.5.2 that this cost is $O(n \log n + I \log n)$, where I is the number of intersections among the segments in N . We only need to take into account the additional cost of rotations during addition. But this can be handled just like the cost of rotations during deletion, using Theorem 4.6.3 instead of Theorem 4.5.2. As we have already remarked, the cost of point location is now $\tilde{O}(\log m)$, regardless of the update sequence.

Exercises

4.6.1 Extend the semidynamic point location structure for Voronoi diagrams (Section 3.3) to the fully dynamic setting using dynamic shuffling. Show that the cost of random update is $O(\log m)$, where m is the number of undeleted sites existing at any given time. Show that the query time is $\tilde{O}(\log^2 m)$ regardless of the update sequence.

4.6.2 Design a point location structure for planar Delaunay triangulations using the dynamic shuffling technique. Show that the expected cost of a random update is $O(\log m)$, where m is the number of undeleted sites existing at any given time. Show that the query time is $\tilde{O}(\log^2 m)$ regardless of the update sequence.

Bibliographic notes

This chapter is based on Mulmuley [166, 168]. The results on lazy maintenance of history (Sections 4.1, 4.2, and 4.3) were obtained independently by Schwarzkopf [200]. Independently, Devillers, Meiser, and Teillaud [81] give another dynamic algorithm for maintaining planar Delaunay triangulations. For more work on dynamic algorithms, see Clarkson, Mehlhorn, and Seidel [70], who also investigate dynamic maintenance of convex hulls further. Exercise 4.3.1 is based on [72]. Exercise 4.4.3 is based on [200].

For other techniques for dynamization, see Sections 5.4 and 8.6. Also see the bibliographic notes for these sections and for the sections wherein the techniques in this chapter are applied to specific problems.

Chapter 5

Random sampling

We remarked in Chapter 1 that quick-sort can be viewed in two different ways. It can be thought of as a randomized incremental algorithm or as a randomized divide-and-conquer algorithm. In Chapter 3, we gave several randomized incremental algorithms, which can be thought of as extending the first view of quick-sort. In this chapter, we extend the second view of quick-sort, namely, as a randomized divide-and-conquer algorithm.

The randomized divide-and-conquer principle underlying quick-sort is the following: Let N be any set of points on the real line. If we pick a random element S from N , then it divides the line into two regions of roughly equal size. By the size of a region, we mean the number of unchosen points lying in that region.

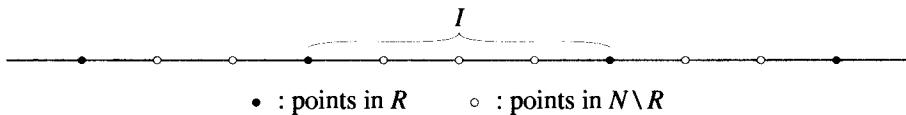


Figure 5.1: A random sample of points on the line.

More generally, consider the following situation. Let R be a random sample (subset) of N of size r (Figure 5.1).¹ By this, we mean that every element in N is equally likely to be in R . One way of choosing a random sample of size r is the following. We choose the first element in R randomly from N . We choose the second element of R from the remaining $n-1$ elements independently and randomly. We repeat the process until r elements from N are chosen. This process is called random sampling without replacement. We shall see other ways of taking random samples later.

¹In this chapter, we often let R denote a random sample instead of the real line, as in the earlier chapters. The meaning should be clear from the context.

We can now ask the following question: Does R divide the real line into regions of roughly equal size? What we mean is the following. Let $H(R)$ denote the partition of the line formed by R . Let us define the *conflict size* of any interval I in $H(R)$ as the number of points in $N \setminus R$ lying in I . Let n be the size of N . We are asking whether the conflict size of each interval in $H(R)$ is $O(n/r)$ with high probability. We cannot prove a result that is so strong. However, the following weaker result holds.

Theorem 5.0.1 *Let N be a set of n points on the real line. Let $R \subseteq N$ be a random sample of size r . With probability greater than $1/2$, the conflict size of each interval in $H(R)$ is $O([n/r] \log r)$. More generally, fix any $c > 2$. For any $s \geq r > 2$, with probability $1 - O(1/s^{c-2})$, the conflict size of each interval in $H(R)$ is less than $c(n \ln s)/(r-2)$. In other words, the probability of some conflict size exceeding $c(n \ln s)/(r-2)$ is small, $O(1/s^{c-2})$ to be precise.*

Proof. Let $\Pi = \Pi(N)$ be the set of all pairs of the form (p, q) , where p , as well as q , is a point in N or a point at infinity. By a point at infinity, we mean either $-\infty$ or $+\infty$. Let $\sigma = (p, q)$ be any such pair in Π . It defines an interval on the real line. Let $D(\sigma) = \{p, q\} \cap N$. In other words, $D(\sigma)$ consists of the endpoints of σ not at infinity. We say that the points in $D(\sigma)$ define σ . We define the degree $d(\sigma)$ to be the size of $D(\sigma)$. It is either two, one or zero in the present case. We define $L(\sigma)$ to be the set of points in N that lie in the interior of σ . We say that the points in $L(\sigma)$ *conflict* with σ . The number of points in $L(\sigma)$ is called the conflict size of σ . It is denoted by $l(\sigma)$.

Thus, $\Pi = \Pi(N)$ can be thought of as a configuration space (Section 3.4) over the given set of points N . It is one of the simplest configuration spaces. (The notion of a configuration space is the only concept from Chapter 3 that we need in this chapter.) We say that an interval $\sigma \in \Pi$ is *active* over a subset $R \subseteq N$ if it occurs as an interval of $H(R)$. This happens iff R contains all points defining σ , but no point in conflict with σ .

Now let $R \subseteq N$ denote a random sample of size r . Let $p(\sigma, r)$ denote the conditional probability that R contains no point in conflict with σ , given that it contains the points defining σ . We claim that

$$p(\sigma, r) \leq \left(1 - \frac{l(\sigma)}{n}\right)^{r-d(\sigma)}. \quad (5.1)$$

The intuition behind this is the following. Ignore the draws that result in points defining σ . The remaining $r - d(\sigma)$ can still be thought of as resulting from independent random draws. The probability of choosing a conflicting point in any such draw is greater than or equal to $l(\sigma)/n$. This should imply (5.1). A rigorous justification follows.

Let $R' = R \setminus D(\sigma)$. Then R' is a random sample of the set $N' = N \setminus D(\sigma)$ of size $n - d(\sigma)$. Conceptually, it is obtained from N' by $r - d(\sigma)$ successive random draws without replacement. For each $j \geq 1$, the probability that the point chosen in the j th draw does not conflict with σ , given that no point chosen in any previous draw conflicts with σ , is

$$1 - \frac{l(\sigma)}{n - d(\sigma) - j} \leq 1 - \frac{l(\sigma)}{n}.$$

This immediately implies (5.1), because the draws are independent.

Since $1 - l(\sigma)/n \leq \exp(-l(\sigma)/n)$, (5.1) implies that

$$p(\sigma, r) \leq \exp\left(-\frac{l(\sigma)}{n}(r - d(\sigma))\right), \quad (5.2)$$

where $\exp(x)$ denotes e^x . But $d(\sigma) \leq 2$. Hence, if $l(\sigma) \geq c(n \ln s)/(r - 2)$, for some $c > 1$, we get

$$p(\sigma, r) \leq \exp(-c \ln s) = \frac{1}{s^c}. \quad (5.3)$$

Let $q(\sigma, r)$ denote the probability that R contains all points defining σ . The probability that σ is active over the random sample R is precisely $p(\sigma, r) q(\sigma, r)$. This follows from the law of conditional probability. Hence, the probability that some σ , with $l(\sigma) > c(n \ln s)/(r - 2)$, is active over the random sample R is

$$\sum_{\sigma \in \Pi: l(\sigma) > \frac{cn \ln s}{r-2}} p(\sigma, r) q(\sigma, r) \leq \sum_{\sigma \in \Pi: l(\sigma) > \frac{cn \ln s}{r-2}} q(\sigma, r)/s^c \leq \frac{1}{s^c} \sum_{\sigma \in \Pi} q(\sigma, r). \quad (5.4)$$

Here we are using (5.3) and the fact that the probability of a union of events is bounded by the sum of the probabilities of respective events.

Let $\pi(R)$ denote the number of intervals in Π whose defining points are contained in R . Notice that the last sum in (5.4) is just the expected value of $\pi(R)$. Hence, the probability that some σ , with $l(\sigma) > c(n \ln s)/(r - 2)$, is active over the random sample R is bounded by

$$\frac{1}{s^c} E[\pi(R)]. \quad (5.5)$$

But R has r points. It is easy to check that $\pi(R) = \binom{r}{2} + 2r + 1$. If we choose c to be greater than 2, the theorem follows. \square

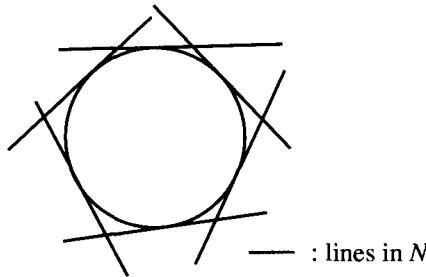


Figure 5.2: A counter-example.

5.1 Configuration spaces with bounded valence

What should be the analogue of Theorem 5.0.1 when N is not a set of points, but rather a set of objects? Consider a concrete example. Let N be a set of lines in the plane. Choose a random sample $R \subseteq N$ of size r (Figure 5.3). Let $G(R)$ denote the arrangement formed by R . Define the conflict size (relative to N) of any 2-face in the arrangement $G(R)$ as the number of lines in $N \setminus R$ intersecting it (Figure 5.3(b)). Is the conflict size of each such 2-face $O([n/r] \log r)$ with a high probability? In general, the answer is no. A counterexample is obtained if all lines in N are tangent to a unit circle (Figure 5.2). In this case, for any sample $R \subseteq N$ of size r , the conflict size of the 2-face in $G(R)$ enclosing the unit circle is always $n - r$.

The main problem in this counter-example is that a 2-face in $G(R)$ can have an arbitrarily large number of sides. What happens if we refine $G(R)$ so that each 2-face in the refined partition $H(R)$ has a bounded number of sides? For example, let $H(R)$ be the trapezoidal decomposition (Section 2.3) of $G(R)$ (Figure 5.3(c)). Then it turns out that, with high probability, the conflict size of every trapezoid in $H(R)$ is indeed $O([n/r] \log r)$. A proof is obtained by generalizing the proof of Theorem 5.0.1 in a simple way. However, this kind of an argument is going to be used so many times that we shall try to prove the analogue of Theorem 5.0.1 in as general setting as possible.

The reader has probably guessed what this general setting is going to be. The crucial difference between $G(R)$ and $H(R)$ above is that $H(R)$ has the bounded degree property: Each trapezoid in $H(R)$ is defined by a bounded number of segments. We have already introduced in Section 3.4 the general combinatorial systems, namely, configurations spaces, which have this property. So one might think that an analogue of Theorem 5.0.1 should hold for general configuration spaces, too. That is not quite true. We need

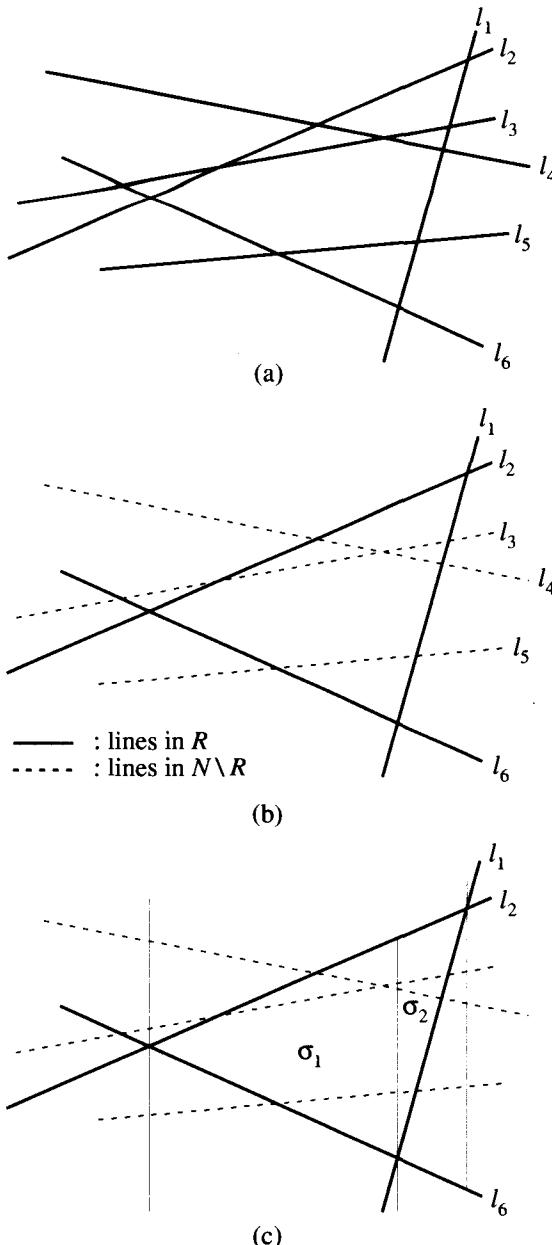


Figure 5.3: (a) A set N of lines. (b) $G(R)$. (c) $H(R)$ defined as a trapezoidal decomposition.

to impose one more restriction on the configuration spaces. The simplest such restriction is stipulated in the following definition.

Definition 5.1.1 A configuration space $\Pi(N)$ is said to have bounded valence if the number of configurations in $\Pi(N)$ sharing the same trigger set is bounded (by a constant).

All configuration spaces described in Chapter 3 have bounded valence. For example, consider the previous example of lines in the plane. Let N be a set of lines in the plane. Let $\Pi(N)$ be the configuration space of feasible trapezoids over N (Example 3.4.1). Then all feasible trapezoids in $\Pi(N)$ having the same trigger set T can be identified with the trapezoids in the trapezoidal decomposition formed by T . Since the size of T is bounded, the number of such trapezoids is also bounded.

The following theorem is the analogue of Theorem 5.0.1 for configuration spaces of bounded valence.

Theorem 5.1.2 Let $\Pi(N)$ be any configuration space of bounded valence. Let n be the size of N . Let d be the maximum degree of a configuration in $\Pi(N)$. (By the definition of a configuration space, d is bounded.) Let R be a random sample of N of size r . With probability greater than $1/2$, for each active configuration $\sigma \in \Pi^0(R)$, the conflict size of σ relative to N is at most $c(n/r) \log r$, where c is a large enough constant. More generally, fix any $c > d$. For any $s \geq r > d$, with probability $1 - O(1/s^{c-d})$ the conflict size of each configuration in $\Pi^0(R)$ (relative to N) is less than $c(n \log s)/(r-d)$. In other words, the probability of some conflict size exceeding $c(n \log s)/(r-d)$ is small, $O(1/s^{c-d})$ to be precise. (The constant within the Big-Oh notation depends exponentially on d .)

Proof. In the proof of (5.5), we never used the fact that the elements in N were points on the line. The only fact used in its proof was that $d(\sigma)$ is bounded (the bound was 2 in that case). This holds in the present case, too, because $\Pi(N)$ has bounded degree property, by definition. Translating the proof of (5.5) *verbatim*, we conclude the following.

Fact: The probability that some $\sigma \in \Pi(N)$, with $l(\sigma) \geq c(n \ln s)/(r-d)$, is active over a random sample R is bounded by $E[\pi(R)]/s^c$.

Here $\pi(R)$ denotes the number of configurations in $\Pi(N)$ whose defining elements, i.e., triggers, are contained in R . In other words, $\pi(R)$ is just the size of the subspace $\Pi(R)$. And $E[\pi(R)]$ denotes the expected value of $\pi(R)$. For each $b \leq d$, there are at most $\binom{r}{b} \leq r^b$ distinct trigger sets contained in R . Since $\Pi(N)$ has bounded valence, only a constant number of configurations in $\Pi(N)$ can share the same trigger set. It follows that $\pi(R) = O(r^d)$. Hence, $E[\pi(R)]$ is also $O(r^d)$. The theorem follows from the above fact. \square

Example 5.1.3 (Lines in the plane)

Let us return to the set N of lines in the plane. Apply Theorem 5.1.2 to the configuration space of feasible trapezoids over N (Example 3.4.1). It then follows that, for any random sample $R \subseteq N$, each trapezoid in the trapezoidal decomposition $H(R)$ has $O([n/r] \log r)$ conflict size with high probability.

As we have already remarked, the crucial property of trapezoidal decompositions is that they have bounded degree property. But there is nothing special about trapezoidal decompositions. Any refinement of the arrangement $G(R)$ with bounded degree property is good enough. To emphasize this property, let us consider another way refining the arrangement $G(R)$. Assume that we are given a fixed triangle Γ in the plane. Let us restrict ourselves within Γ . Consider the restricted arrangement $G(R) \cap \Gamma$. If one wishes, one can let the vertices of Γ lie at infinity. So this is not a restriction really. Triangulate each face of $G(R) \cap \Gamma$ by connecting each of its vertices to its bottom vertex, i.e., the vertex with the minimum y -coordinate (Figure 5.4). (Ties can be resolved using lexicographic ordering.) If this bottom vertex is a vertex of Γ , it can lie at infinity. This causes no problems because a vertex at infinity is defined by a fixed direction. It only means that the segments joining this vertex are actually parallel half lines. The triangulation of $G(R) \cap \Gamma$ described above is called its *canonical triangulation*.

We claim that the conflict size of each triangle in $G(R) \cap \Gamma$ is also $O([n/r] \log r)$ with high probability. Let us see why. Let us define a new configuration space $\Sigma(N)$ as follows. Let us call a triangle σ in the plane *feasible* or *possible* over N if σ occurs in the canonical triangulation of $G(R) \cap \Gamma$, for some subset $R \subseteq N$. For a feasible triangle σ , we define the trigger set $D(\sigma)$ to be the set of segments in N that are adjacent to σ . We define the conflict set $L(\sigma)$ to be the set of segments in $N \setminus D(\sigma)$ intersecting σ . For

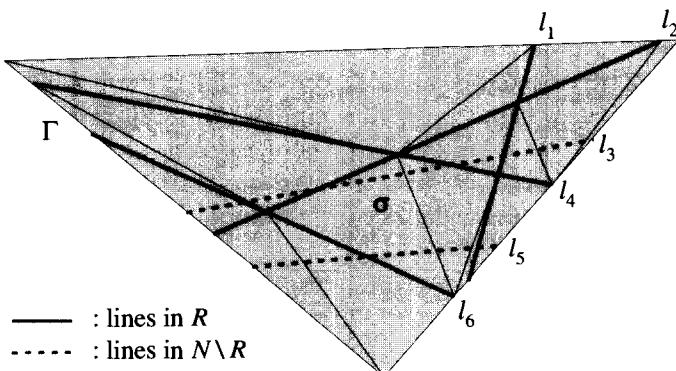


Figure 5.4: A canonical triangulation.

example, in Figure 5.4, the trigger set of the feasible triangle σ is $\{l_2, l_4, l_6\}$. The stopper set is $\{l_3, l_5\}$. In this fashion, we can associate an abstract configuration $(D(\sigma), L(\sigma))$ with every feasible triangle σ . It is easily seen that the size of $D(\sigma)$ is less than or equal to six, assuming that the segments in N are in general position. Thus, we get a configuration space $\Sigma(N)$ of all feasible triangles over N . It has bounded valence (why?). The claim in the beginning of this paragraph now follows by applying Theorem 5.1.2 to this configuration space.

In Theorem 5.1.2, we assumed that the random sample R was chosen by r random and independent draws without replacement. There are two other ways of choosing a random sample that are more convenient in practice. They are the following.

Sampling with replacement

We choose the first element randomly from N . We choose the second element independently and randomly from the whole set N again, rather than the set of remaining $n - 1$ elements. We repeat the process r times. The cardinality of the resulting set R need not be exactly r . This is because one element can be chosen more than once. But this will cause only a negligible difference as far as we are concerned.

Bernoulli sampling

Take a coin with probability of success $p = r/n$. A computer can simulate such a coin as follows. Choose a random number from the set $\{1, \dots, n\}$. If the number is less than or equal to r , we declare the “toss” as successful. For each element in N , flip the coin independently. Let R be the set of elements for which the toss was successful. The expected size of R is clearly r .

Analogues of Theorem 5.1.2 also hold for the preceding two methods of sampling. We leave this as a simple exercise for the reader.

Theorem 5.1.2 provides a simple method for randomized division. This leads to natural divide-and-conquer algorithms for several search problems. The search structures based random sampling can be built in either a top-down fashion or a bottom-up fashion. There are some significant differences between these two approaches. In Section 5.2, we shall describe the top-down approach. In Section 5.3, we shall describe the bottom-up approach.

Exercise

5.1.1 Prove analogues of Theorem 5.1.2 for random sampling with replacement and for Bernoulli sampling.

5.2 Top-down sampling

We describe in this section a top-down method of building search structures based on random sampling. First, we shall describe the method in intuitive, abstract terms.

The simplest search structure based on top-down sampling is a randomized binary tree (Section 1.3). Recall its definition based on the divide-and-conquer principle: Choose a random point p from the given set N of points. It divides N in two subsets, N_1 and N_2 , of roughly equal size. Label the root of the search tree with p . The children of this root are the recursively built trees for N_1 and N_2 .

Randomized binary trees can also be thought of as recording the history of quick-sort. In Chapter 3, we extended this view to several problems in computational geometry. Here we want to generalize the former view based on the randomized divide-and-conquer paradigm. A general search problem in computational geometry has the following abstract form (Chapter 2): Given a set N of objects in R^d , construct the induced complex (partition) $H(N)$ and a geometric search structure $\tilde{H}(N)$ that can be used to answer the queries over $H(N)$ quickly. A typical example of a query is a point location query.

We shall assume that the complex $H(N)$ has the bounded-degree property. This means that every face of $H(N)$, at least of the dimension that matters, is defined by a bounded number of objects in N . This assumption is needed to make the random sampling result of Section 5.1 applicable. If it is not satisfied by the partition under consideration, one has to work with its suitable refinement. We have already done such a thing before. For example, Voronoi regions need not have a bounded number of sides. Hence, we worked with radial triangulations in Section 3.3.

In abstract terms, a search structure based on top-down sampling is defined as follows.

1. Choose a random subset $R \subseteq N$ of a large enough constant size r .
2. Build $H(R)$ and a search structure for $H(R)$. Since the size of R is a constant, this search structure is typically trivial.
3. Build conflict lists of all faces of $H(R)$ of relevant dimensions. The notion of a conflict (Section 3.4) depends on the problem under consideration.
4. For each such face $\Delta \in H(R)$, recursively build a search structure for $N(\Delta)$, the set of objects in N in conflict with Δ .
5. Build an ascent structure, denoted by $\text{ascent}(N, R)$. It is used in queries in a manner described below.

The queries are answered as follows. The original query is over the set N . We first answer it over the smaller set R . This can be done using the trivial search structure associated with $H(R)$. Let us say that $\Delta \in H(R)$ is the answer to this smaller query. We recursively answer the query over the set $N(\Delta)$ of conflicting objects. After this, using the ascent structure $\text{ascent}(N, R)$, we somehow determine the answer over the set N .

In the next section, we give a simple application of top-down sampling to arrangements of lines. More applications will be provided later in the book.

5.2.1 Arrangements of lines

Consider the search problem associated with arrangements of lines in the plane (Section 2.2). Let N be a set of n given lines in the plane. Let $G(N)$ denote the arrangement formed by them. Our goal is to build a point location structure so that, given a query point p , one can locate quickly—say, in logarithmic time—the face of $G(N)$ containing p . It turns out to be more convenient to solve the following, slightly more general problem. We assume that we are given, in addition, a fixed, possibly unbounded, triangle Γ in the plane. The goal is to locate the face in the intersection $\Gamma \cap G(N)$ containing the query point p . At the root level, we can let Γ be a triangle whose vertices lie at infinity, that is, $\Gamma = R^d$.

The data structure is defined using the top-down recursive scheme described above. The root of the data structure is labeled with Γ . We store at the root the entire restricted arrangement $G(N) \cap \Gamma$. It can be constructed incrementally in $O(n^2)$ time and space (Section 2.8.2).

Next, we take a random sample $R \subseteq N$ of size r , where r is a large enough constant to be determined later. We construct $G(R) \cap \Gamma$, and further refine it, so that the refined partition has the bounded-degree property. Here we shall use the canonical triangulation of $G(R) \cap \Gamma$ as defined in the previous section (one could also use a trapezoidal decomposition). Let $H(R)$ denote the canonical triangulation of $G(R) \cap \Gamma$. It can be built in $O(1)$ time because R has constant size. For each triangle $\Delta \in H(R)$, let $N(\Delta)$ denote its conflict list, that is, the set of lines in $N \setminus R$ intersecting Δ . Since the size of R is constant, the conflict lists of all simplices in $H(R)$ can be trivially determined in $O(n)$ time: We just check for each pair of a triangle in $H(R)$ and a line in $N \setminus R$ if they conflict.

It follows (Example 5.1.3) that, with probability at least $1/2$, the conflict size of each triangle in $H(R)$ is less than $b(n/r) \log r$, for an appropriate constant $b > 1$, which can be calculated explicitly. In our algorithm, we check that our random sample R indeed satisfies this property. If that is not the case, we take a new random sample R . We keep on taking new samples until the condition is satisfied for every triangle in $H(R)$. We assume that

each trial is independent of the previous trials. The probability of success in each trial is at least $1/2$. Hence, the expected number of required trials is $O(1)$. This follows because the expected number of failures before success in a sequence of Bernoulli trials with a fair coin is $O(1)$ (Appendix A.1.2). As we have already seen, the work done in each trial is $O(n)$. Hence, the total expected work done in all these trials is $O(n)$.

Once a successful sample is obtained, we recur in the obvious fashion for each triangle $\Delta \in H(R)$. If the conflict size of Δ is less than a threshold (an appropriately chosen constant), we stop. Otherwise, we recur within Δ with respect to the set of lines $N(\Delta)$. We also associate with every face of the restricted arrangement $G(N(\Delta)) \cap \Delta$ a *parent pointer* to the face containing it in the arrangement $G(N) \cap \Gamma$ stored at the parent level. These parent pointers constitute the ascent structure that we alluded to in the abstract top-down scheme.

Point location is carried out as follows. Let p be the query point. We assume that it lies in the triangle Γ associated with the root level. (Γ can stand for the whole plane.) To locate p in $G(N) \cap \Gamma$, we first locate the triangle Δ in $H(R)$ containing p . This takes constant time because the size of R is constant. Then we recursively locate the face of $G(N(\Delta)) \cap \Delta$ containing p . The parent pointer associated with this face tells us the face of $G(N) \cap \Gamma$ containing p .

Analysis

From the preceding discussion, it is clear that the cost of point location satisfies the following recurrence relation:

$$q(n) = O(1),$$

if n is less than the threshold. Otherwise,

$$q(n) = O(1) + q\left(b \frac{n}{r} \log r\right).$$

If we choose the sample size r to be a large enough constant, it follows that $q(n) = O(\log n)$.

Let us turn to the cost of building a search structure for the given set N and the root triangle Γ . By convention, we let $N(\Gamma) = N$. Let $t(n) = t(|N(\Gamma)|)$ denote the expected time cost of building this search structure. It satisfies the following recurrence:

$$t(n) = O(1),$$

if n is less than the threshold. Otherwise, it is easily checked that

$$t(n) = t(|N(\Gamma)|) = O(n^2) + \sum_{\Delta \in H(R)} t(|N(\Delta)|) = O\left(n^2 + r^2 t\left(b \frac{n}{r} \log r\right)\right). \quad (5.6)$$

It easily follows by induction that $t(n) \leq n^2 c^{\log_r n}$, where c is a constant that is sufficiently larger than b and the constant within the Big-Oh notation. Intuitively, this should be obvious, because the depth of recursion is $O(\log_r n)$. We leave a routine check to the reader as a simple exercise. It then follows that, for every real number $\epsilon > 0$, we can choose r large enough, so that $t(n) = O(n^{2+\epsilon})$. Let $s(n)$ denote the size of our data structure (actual, not expected). It satisfies the same recurrence relations as $t(n)$ does. Hence, it follows that $s(n)$, too, can be made $O(n^{2+\epsilon})$, for every $\epsilon > 0$.

We have thus proved the following.

Theorem 5.2.1 For every arrangement of n lines in the plane and for every real number $\epsilon > 0$, one can construct a point location structure of $O(n^{2+\epsilon})$ size, guaranteeing $O(\log n)$ query time, in $O(n^{2+\epsilon})$ expected time.

There is a trade-off here between the cost of building the search structure and the query time: To achieve a smaller ϵ , one has to use a larger sample size. This, in turn, increases the constant factor within the query time bound.

5.3 Bottom-up sampling

The size of an arrangement of n lines is $O(n^2)$. Compare this with the $O(n^{2+\epsilon})$ bound in Theorem 5.2.1. Can we get rid of the extra n^ϵ term?

We shall see that this can be done by turning the sampling process upside down. In other words, we build the search structure bottom-up instead of top-down. The simplest search structure based on bottom-up sampling is the skip list (Section 1.4). Here we want to generalize the idea underlying skip lists. Since we are going to be using the scheme on several occasions, we shall state it in a general form first. For this, we will use the same formalism that we used at the beginning of the last section: Given a set N of objects, our goal is to build a search structure that can be used for answering queries over the geometric complex (partition) $H(N)$. We shall denote the search structure that we are going to associate with N by $\text{sample}(N)$. As usual, the size of N will be denoted by n . The reader should keep in mind that if N is a set of points on the line, $\text{sample}(N)$ is just going to be the skip list.

Starting with the set N , we obtain a sequence of sets

$$N = N_1 \supseteq N_2 \supseteq \cdots \supseteq N_{r-1} \supseteq N_r = \emptyset,$$

where N_{i+1} is obtained from N_i by flipping a fair coin² independently for each object in N_i and retaining only those objects for which the toss was a success (heads). The sequence of sets above is called a *gradation* of N . It

²We do not really need the coin to be fair. We only need that the probability of success is a fixed constant between 0 and 1.

follows just as in the case of skip lists (Section 1.4) that the length of this gradation is $\tilde{O}(\log n)$ and that the total size of the sets in this gradation is $\tilde{O}(n)$. Intuitively, this follows because the size of each N_i is roughly half the size of N_{i-1} . The state of $\text{sample}(N)$ is going to be determined by the gradation of N . $\text{Sample}(N)$ will consist of r levels, where r is the length of the above gradation. The set $N_i, 1 \leq i \leq r$, can be thought of as the set of objects stored in the i th level. In each level i , we store the geometric complex (partition) $H(N_i)$ formed by N_i . Each face of $H(N_i)$ is associated with the conflict list of the objects in $N_{i-1} \setminus N_i$ conflicting with it. Moreover, between every two successive levels i and $i + 1$, we keep a certain *descent structure*, which we shall denote by $\text{descent}(i+1, i)$. It is used in answering queries in a manner to be described soon.

In the preceding scheme, we assume that the complex $H(N)$ has the bounded-degree property so that the random sampling results in Section 5.1 are applicable. A crucial property of $\text{sample}(N)$ is the following: Since N_l is a random sample of N_{l-1} of roughly half the size, the conflict size of each $\Delta \in H(N_l)$ is logarithmic with high probability (Theorem 5.1.2 and Exercise 5.1.1).

The queries are answered as follows. Suppose we want to answer a query with respect to the set of objects $N = N_1$. We first answer it with respect to N_r . Since N_r is empty, this should be trivial. After this, we just descend through our data structure. Inductively, assume that we have the answer to our query with respect to the set $N_i, 1 \leq i \leq r$. This answer is a face Δ of $H(N_i)$. We find the answer with respect to the set N_{i-1} by using the descent structure $\text{descent}(i, i-1)$ and the conflict information associated with Δ . The exact nature of the descent depends on the problem under consideration. We only need that this descent from the i th level to $(i-1)$ th level can be done quickly, say, in polylogarithmic time. Descending in this fashion through all levels, we eventually obtain the required answer with respect to the set $N = N_1$.

The reader should verify that, when N is a set of points on the real line and $H(N)$ is the resulting partition of the line, $\text{sample}(N)$ coincides with the skip list, as defined in Section 1.4: The descent structure in this case consists of the *descent pointers* described in that section.

In the next section, we give a simple application of bottom-up sampling to arrangements of lines. More applications will be provided later.

In what follows, we shall denote the size of the set N_i by n_i .

5.3.1 Point location in arrangements

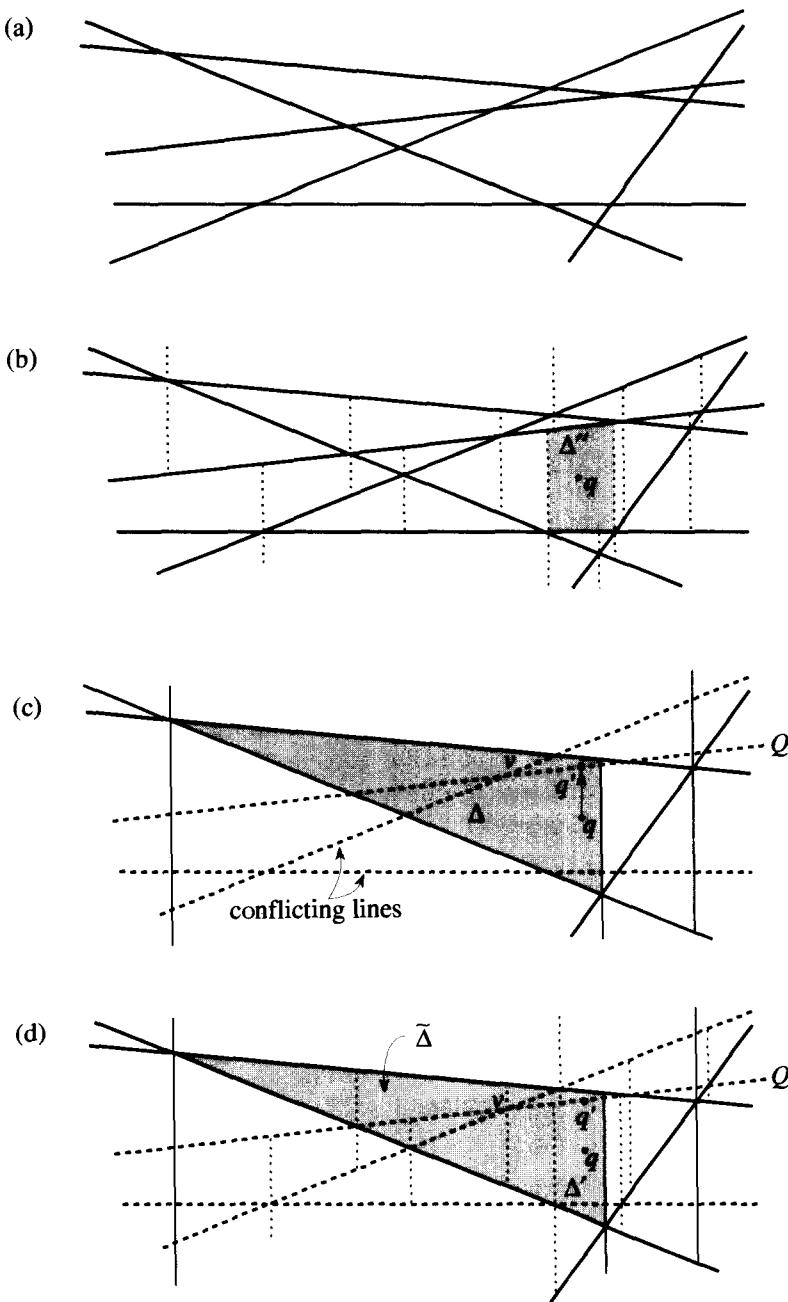
Let N be a set of n lines in the plane. Let $G(N)$ denote the resulting arrangement. The convex regions of $G(N)$ need not have a bounded number

of sides. To ensure the bounded-degree property, we refine $G(N)$ further. In Section 5.2, we used canonical triangulations. In this section, we shall use trapezoidal decompositions. Unlike in Section 5.2, the method of refinement does make a difference here. We shall see that using trapezoidal decompositions instead of canonical triangulations saves us a log factor in the search time. Let $H(N)$ denote the trapezoidal decomposition of $G(N)$. We shall now associate a point location structure with $H(N)$ using bottom-up sampling. We shall denote this search structure by $\text{sample}(N)$.

$\text{Sample}(N)$ is obtained by a straightforward application of the bottom-up sampling scheme. We grade N by repeatedly tossing a fair coin. Let $N = N_1 \supseteq N_2 \supseteq \dots \supseteq N_{r-1} \supseteq N_r = \emptyset$ be the resulting gradation. In each level l , $1 \leq l \leq r$, we store the trapezoidal decomposition $H(N_l)$. We also store with each trapezoid $\Delta \in H(N_l)$ a conflict list $L(\Delta)$ of the lines in $N_{l-1} \setminus N_l$ intersecting it. Finally, we define the descent structure $\text{descent}(l+1, l)$ as the partition $H(N_{l+1}) \oplus H(N_l)$ obtained by superimposing $H(N_l)$ and $H(N_{l+1})$ on each other (Figure 5.5). Intuitively, this will let us descend from $H(N_{l+1})$ to $H(N_l)$ by going through the intermediate partition $H(N_{l+1}) \oplus H(N_l)$. We also associate with each trapezoid in $\text{descent}(l+1, l)$ a pointer to the unique trapezoid in $H(N_l)$ containing it.

We shall show below that $\text{sample}(N)$ can be constructed in $\tilde{O}(n^2)$ time and space. But before that, let us see how $\text{sample}(N)$ is used for point location.

Let q be a fixed query point. Let r be the last level in our search structure. This means that N_r is empty. Locating q in $H(N_r)$ is trivial. Inductively, assume that we have located q in $H(N_{l+1})$, $1 \leq l < r$. We wish to descend from level $l+1$ to level l (Figure 5.5). Let $\Delta = \Delta_{l+1}$ be the trapezoid in $H(N_{l+1})$ containing the query point q . Let $\tilde{\Delta}$ denote the restriction of the superimposed partition $\text{descent}(l+1, l)$ within Δ . We assume that $\text{descent}(l+1, l)$ is stored in such a fashion that $\tilde{\Delta}$ is available to us. First, we shall determine the first line Q in N_l that intersects the vertical ray from q directed upwards (Figure 5.5(c)). (We can assume that Q exists by adding a dummy segment at infinity.) Obviously, Q is either the line bounding the upper side of Δ or it belongs to $L(\Delta)$. Hence, it can be trivially determined in $O(|L(\Delta)| + 1)$ time. Let q' be the point of intersection of the vertical ray with Q . Let v be the intersection of Q with a line in $L(\Delta)$ or the boundary of Δ , which is nearest to q' on its left side. It is easy to determine v in $O(|L(\Delta)| + 1)$ time. We can assume for this purpose that, for each such Q , we have available a list of its intersections with the other lines in $L(\Delta)$. This just requires augmenting the representation of $\tilde{\Delta}$. Once v is determined, we walk within $\text{descent}(l+1, l)$ from v to q' (Figure 5.5(d)). This locates q' and hence the trapezoid $\Delta' \in \text{descent}(l+1, l)$ containing q (Figure 5.5(d)). The cost of

Figure 5.5: (a) N_l . (b) $H(N_l)$. (c) $H(N_{l+1})$. (d) Descent($l+1, l$).

this walk is again $O(|L(\Delta)| + 1)$. The required trapezoid $\Delta'' = \Delta_l \in H(N_l)$ containing q is the unique one containing Δ' (Figure 5.5(b)).

Analysis

The preceding discussion shows that we can descend from level $l + 1$ to level l in $O(|L(\Delta_{l+1})| + 1)$ time. Since N_{l+1} is a random sample of N_l of roughly half the size, $|L(\Delta_l)| = \tilde{O}(\log n)$, for all l (Theorem 5.1.2, Exercise 5.1.1). We have already seen that the number of levels in our data structure is also $\tilde{O}(\log n)$. It follows that the time required to locate a fixed query point q is $\tilde{O}(\log^2 n)$. The following lemma shows that the query time is, in fact, $\tilde{O}(\log n)$.

Lemma 5.3.1 *For a fixed query point q , $\sum_{i \geq 1} |L(\Delta_i)|$ is $\tilde{O}(\log n)$, where Δ_i is the trapezoid containing q in $H(N_i)$.*

Proof. We extend the argument in Section 1.4, which showed that the query time for skip lists is $\tilde{O}(\log n)$. The reader may want to review that argument. Let $NB(s)$ denote the random variable that is equal to the number of tails (failures) obtained before obtaining s heads (successes) in a sequence of Bernoulli trials with a fair coin. It is the so-called negative binomial distribution (Appendix A.1.2). When $s = 1$, it is the geometric distribution.

Claim: For all i , $|L(\Delta_i)| \leq NB(a)$, for some fixed constant a .

Let us first see why the claim immediately implies the lemma. We use exactly the argument that we used for skip lists. The coin tosses used at any level of our data structure are independent from the coin tosses used in the preceding levels. Hence, for any fixed constant c , $\sum_{i \leq c \log n} |L(\Delta_i)| \leq NB(c \log n) = \tilde{O}(\log n)$. The last equality follows from the Chernoff bound for negative binomial distributions (Appendix A.1.2). Since the number of levels is $\tilde{O}(\log n)$, this proves the bound on the query time in the lemma.

Let us turn to the proof of the claim. Fix a level i . Also fix the set N_i of lines occurring in the i th level of the data structure. The set N_{i+1} is obtained by flipping a fair coin for each line in N_i and retaining those lines for which the toss was heads. We shall prove the following.

Lemma 5.3.2 *There is an imaginary, on-line ordering h_1, h_2, \dots of all lines in N_i such that the set of lines “defining” or intersecting the trapezoid Δ_{i+1} always occurs as an initial subsequence of h_1, h_2, \dots . By on-line ordering, we mean that h_{k+1} can be chosen on the basis of the known coin toss results for h_1, \dots, h_k . Note that Δ_{i+1} is not known to us a priori, because it depends on the results of coin tosses for the lines in N_i .*

Before we turn to the proof of this lemma, let us see how it implies the preceding claim. Let us examine the coin toss results for h_1, h_2, \dots in that order. For each line that is put in N_{i+1} , the toss is heads. Since the number

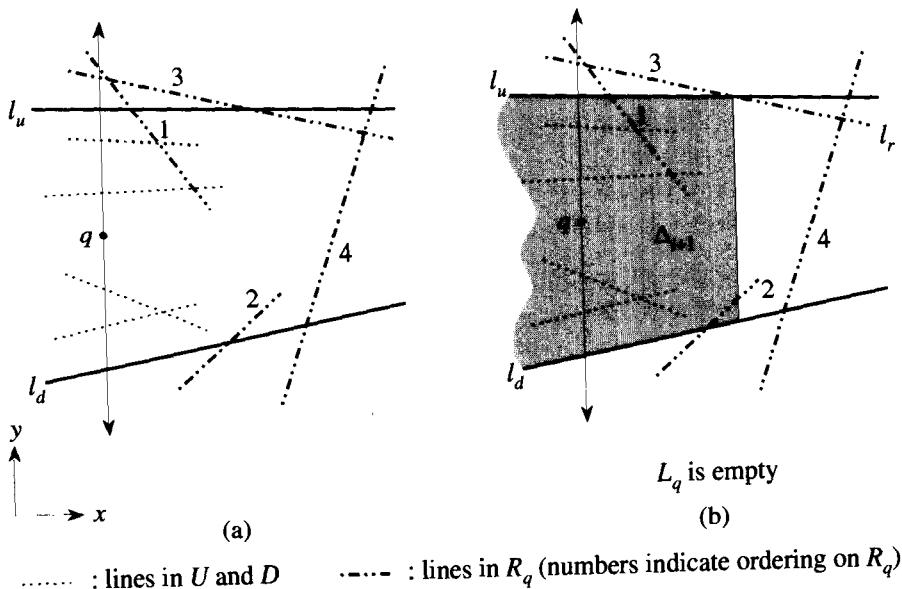


Figure 5.6: On-line ordering.

of lines defining any trapezoid is at most four, it follows from this lemma that $|L(\Delta_{i+1})| \leq NB(4)$, thereby establishing the claim.

Proof. Consider the set V_u of lines in N_i intersecting the vertical line extending upwards from the query point q (Figure 5.6). Order V_u in the increasing y direction. Initially, we shall toss coins for these lines in V_u in the increasing y direction away from q until we obtain heads, and then (temporarily) stop. Let l_u be the line for which we obtained heads. Let $U \subseteq V_u$ denote the set of lines before l_u for which we obtained tails. Clearly, $l_u \in N_{i+1}$, whereas no line in U belongs to N_{i+1} . Thus, l_u is obviously going to be bounding the top of the trapezoid Δ_{i+1} , which we do not know completely as yet. Moreover, all lines in U obviously conflict with Δ_{i+1} .

Now we resume our coin tossing, in a symmetric manner, for the lines in N_i intersecting the vertical line extending downwards from q until we obtain heads, and then we again stop temporarily. Let D be the set of lines for which we obtained tails and let l_d be the line for which we obtained heads. Obviously, l_d is going to be bounding the bottom of the trapezoid Δ_{i+1} , which we know partially by now.

Now discard (hypothetically) the lines in U and D and consider the intersections of the remaining lines with l_u and l_d . Let R_q be the set of remaining lines that intersect the region bounded by l_u , l_d , and the vertical line through q . We order R_q as follows. Given two lines l_1 and l_2 in R_q , we say that $l_1 \ll l_2$

iff the x -coordinate of either $l_1 \cap l_u$ or $l_1 \cap l_d$ is less than the x -coordinates of both $l_2 \cap l_u$ and $l_2 \cap l_d$. Figure 5.6 shows the ordering of R_q . Now we resume tossing coins for the lines in R_q in the increasing order until we obtain heads. Let l_r be the line for which we obtained heads. It is then clear that l_r defines the right side of Δ_{i+1} in the sense that the vertical right side of Δ_{i+1} will be adjacent to the intersection of l_r with either l_u or l_d (Figure 5.6(b)). Moreover, all lines for which we obtained tails will conflict with Δ_{i+1} .

Now discard (hypothetically) the lines in R_q as well. Let L_q be the set of remaining lines intersecting either l_u or l_d to the left of the vertical line through q . In Figure 5.6(b), L_q is empty. We order L_q in a symmetric fashion, just as we ordered R_q . Now we resume tossing coins for the lines in L_q in the increasing order (away from q) until we get heads. Then we temporarily stop. Let l_l be the line for which we obtained heads. For the same reasons as before, it defines the left side of the trapezoid Δ_{i+1} . All lines in L_q for which we obtained tails conflict with Δ_{i+1} .

At this point, the trapezoid Δ_{i+1} containing q in the $(i+1)$ th level has been completely determined. Indeed, l_u, l_d, l_r, l_l are the lines defining Δ_{i+1} and the lines for which we obtained tails so far are precisely the lines in conflict with Δ_{i+1} . (We did not take into account the exceptional cases such as when Δ_{i+1} is unbounded or when it is, in fact, a triangle. For example, in Figure 5.6(b) Δ_{i+1} is unbounded on the left because L_q is empty. A slight modification to the argument will cover these cases, too.)

We now toss coins for the remaining lines in any order whatsoever. It follows that the above on-line sequence of tosses has the desired property. \square

We showed above that the query time is $\tilde{O}(\log n)$ for a fixed query point. We shall now show that the maximum query cost is $\tilde{O}(\log n)$, regardless of where the query point is located. For this, we observe that the number of distinct combinatorial search paths in our data structure is bounded by a polynomial of fixed degree in n . Combinatorial distinctness means that we have different sequences of trapezoids in the search paths. More precisely, let $\bar{G}(N)$ be the refinement of $G(N)$ obtained by passing infinite vertical lines through all intersections among the lines in N . Then, for a fixed region Δ in $\bar{G}(N)$, it is easy to see that the search path in $\text{sample}(N)$ remains the same if the query point lies anywhere in Δ . (On the other hand, the search paths of two points lying in the same region of $G(N)$ need not be identical. This happens when they belong to different trapezoids in some $H(N_i)$.) It follows that the number of distinct search paths is bounded by the size of $\bar{G}(N)$, which is $O(n^2)$. Arguing as in the case of randomized binary trees (Observation 1.3.1), it now follows that the maximum cost of locating any point is also $\tilde{O}(\log n)$.

Now we shall show that $\text{sample}(N)$ can be built in $\tilde{O}(n^2)$ time. Each $H(N_l)$, $1 \leq l \leq r$, can be built incrementally in $O(n_l^2)$ time just as in Section 2.8.2. The only difference is that in Section 2.8.2 we dealt with arrangements instead of their trapezoidal decompositions. But this requires only minor extensions, which we shall leave to the reader. Next, let us turn to the construction of conflict lists. Fix a line $Q \in N_{l-1}$. By the Zone Theorem (Theorem 2.8.2), the number of trapezoids in $H(N_l)$ conflicting with Q is $O(n_l)$. All of them can be determined in $O(n_l)$ time as follows. Let $S \in N_l$ be any fixed line. We first locate the intersection $Q \cup S$ in $H(N_l)$ by a linear search along S in $H(N_l)$. After this, we travel along Q in $H(N_l)$ in both directions, starting at $Q \cap S$. At the end of this travel, we shall have determined the set of all trapezoids in $H(N_l)$ intersecting Q , linearly ordered along Q . We insert Q in the conflict lists of all these trapezoids. We also need to build the descent structure $H(N_l) \oplus H(N_{l-1})$. For this, we first build the superposition $H(N_l) \oplus G(N_{l-1})$, i.e., the superposition of the arrangement $G(N_{l-1})$ with the trapezoidal decomposition $H(N_l)$. It can then be refined to $H(N_l) \oplus H(N_{l-1})$ trivially. Note that $H(N_l) \oplus G(N_{l-1})$ is obtained by “drawing” the lines in $N_{l-1} \setminus N_l$ on top of $H(N_l)$ (Figure 5.5(c)). This too can be done incrementally very much as in Section 2.8.2. We add to $H(N_l)$ the lines in $N_{l-1} \setminus N_l$, one at a time, in any order. By a minor extension of the argument in Section 2.8.2, it follows that the addition of each line $Q \in N_{l-1} \setminus N_l$ takes $O(n_{l-1})$ time. (The only difference here is that Q also intersects the vertical attachments in $H(N_l)$. But we have already seen that the number of these intersections is $O(n_l)$.) It follows that the cost of building $\text{descent}(l, l-1)$ is $O(n_{l-1}^2)$.

Thus, the total cost of building $\text{sample}(N)$ is $O(\sum_l n_l^2) = \tilde{O}(n^2)$. The last equality follows because $\sum n_l$, the total size of the sets N_l , is $\tilde{O}(n)$, as we have already remarked.

It is also easy to ensure that the space requirement is $O(n^2)$ (worst case) without affecting the query time. For this, we ensure that r , the number of levels in the data structure, is such that $\sum_{i=1}^r n_i^2 \leq b n^2$, for some large enough constant b . The levels higher than the maximum permissible value of r are not maintained. During point location, we locate the query point in $H(N_r)$ trivially in $O(|N_r|)$ time (how?) and then descend through the data structure as before. If b is chosen large enough, with high probability, this “clipped” data structure coincides with the nonclipped data structure, as defined before. Hence, with high probability, the query time also remains unaffected. This clipping is mainly of theoretical interest. We shall ignore it in what follows.

Let us summarize the discussion so far.

Theorem 5.3.3 $\text{Sample}(N)$ can be built in $\tilde{O}(n^2)$ time and space. The point location cost is $\tilde{O}(\log n)$. Furthermore, the space requirement can be made $O(n^2)$ (worst case) without affecting the query time.

It is illuminating to compare this theorem with Theorem 5.2.1. The number of trapezoids that occur as “leaves” at the bottom of the top-down search structure is $O(n^{2+\epsilon})$. In contrast, this number is $O(n^2)$ for the bottom-up search structure. Bottom-up sampling controls the size of the data structure by keeping the number of “leaves” under control. Hence, it is generally more efficient than top-down sampling. On the other hand, the descent operation in bottom-up sampling is more sophisticated. This means it may not always be applicable in every case where top-down sampling is applicable. The converse is also true: there are some applications wherein bottom-up sampling is applicable, but top-down sampling is not.

5.4 Dynamic sampling

In Sections 5.2 and 5.3, we saw how random sampling can be used in geometric search. The search structures described in these sections were static. In this section, we shall give a technique, called *dynamic sampling*, for making such search structures dynamic. This will allow us to add or delete an object in the search structure efficiently. In this section, we shall only consider dynamization of the search structures based on bottom-up sampling. Dynamic sampling technique is applicable to the search structures based on top-down sampling as well. This will be done in the exercises and also later in the book (e.g., see Section 6.5.1). First, we shall describe our technique in an intuitive and abstract fashion.

The basic idea behind dynamic sampling is to simulate static sampling in a dynamic fashion. It is exactly the idea that we used in making skip lists dynamic (Section 1.4). Let us denote the set of current (undeleted) objects existing at any given time by M . We shall denote the size of M by m . The objects will depend on the problem under consideration. We maintain our search structure in such a way that at any given time, it looks as if it were constructed by applying the static bottom-up sampling, as described in Section 5.3, to the set M . In other words, we can imagine that our search structure, which we shall denote by $\text{sample}(M)$, is obtained as follows. We first obtain a gradation $M = M_1 \supseteq M_2 \supseteq \cdots \supseteq M_{r-1} \supseteq M_r = \emptyset$, where M_{i+1} is obtained from M_i by flipping a fair coin independently for each object in M_i and retaining only those objects for which the toss was a success (heads). The state of $\text{sample}(M)$ is completely determined by this gradation of M . It consists of $r = \tilde{O}(\log m)$ levels. Every level l is associated with a geometric complex (partition) $H(M_l)$. This complex depends upon the problem under

consideration. Between two successive levels $l + 1$ and l , there is a descent structure that is used during the answering of queries. We shall denote it by $\text{descent}(l + 1, l)$.

A key property of the dynamic sampling technique will be that the grading of M , and hence the state of $\text{sample}(M)$, will be independent of the sequence of updates that actually built $\text{sample}(M)$. This will imply that the bounds on the query time obtained in the static setting continue to hold in the dynamic setting as well.

Let us see how to add a new object S to $\text{sample}(M)$. Let $M' = M \cup \{S\}$. Our goal is to obtain $\text{sample}(M')$ from $\text{sample}(M)$. This is done as follows. We toss a fair coin repeatedly until we get failure (tails). Let j be the number of successes obtained before getting failure. The idea is simply to add S to levels 1 through $j + 1$. Let us elaborate what this means. For $1 \leq l \leq j + 1$, let M'_l denote $M_l \cup \{S\}$. Addition of S to the l th level means: $H(M_l)$ is updated to $H(M'_l)$ and $\text{descent}(l+1, l)$ is accordingly updated. This requires $\text{descent}(l + 1, l)$ to be dynamic as well. When S has been added to all levels from 1 to $j + 1$, we get the updated data structure $\text{sample}(M')$. It is completely determined by the gradation $M'_1 \supseteq M'_2 \supseteq \dots$ of M' . Here $M'_l = M_l$, for $l > j + 1$. Note that this gradation of M' is again independent of the actual sequence of the previous updates that built $\text{sample}(M')$.

Deletion is conceptually just the reverse of addition. Suppose we want to delete an object S from $\text{sample}(M)$. Let $M' = M \setminus \{S\}$. Our goal is to obtain $\text{sample}(M')$ from $\text{sample}(M)$. This is done by simply deleting S from all levels of $\text{sample}(M)$ containing it. Let us elaborate what this means. Suppose S is present in levels 1 through $j + 1$, for some $j \geq 0$. For $1 \leq l \leq j + 1$, let M'_l denote $M_l \setminus \{S\}$ for each such level l . Deletion of S from the l th level means: $H(M_l)$ is updated to $H(M'_l)$. Concurrently, we also update $\text{descent}(l+1, l)$. At the end of this procedure, we get $\text{sample}(M')$. The state of $\text{sample}(M')$ is determined by the gradation $M' = M'_1 \supseteq M'_2 \supseteq \dots$. Here $M'_l = M_l$, for $l > j + 1$. This gradation of M' is again independent of the actual sequence of the previous updates that built $\text{sample}(M')$.

In the next section, we shall apply this dynamization technique to the previously defined point location structure for arrangements of lines. In the later two sections, we shall give additional applications of dynamic sampling to point location in trapezoidal decompositions and Voronoi diagrams.

Notation. We shall denote the size of M by m and the size of the set M_l by m_l .

5.4.1 Point location in arrangements

We begin with dynamization of the search structure for arrangements of lines, described in Section 5.3.1. Let M denote the set of current (undeleted)

lines at any given moment. Let $G(M)$ denote the arrangement formed by M . Let $H(M)$ denote the trapezoidal decomposition of $G(M)$. We shall denote the search structure associated with M by $\text{sample}(M)$. It looks as if it were constructed by applying the static method in Section 5.3.1 to the set M . It has $r = \tilde{O}(\log m)$ levels. Each level l , $1 \leq l \leq r$, is associated with the trapezoidal decomposition $H(M_l)$. Each trapezoid $\Delta \in H(M_l)$ is also given a conflict list $L(\Delta)$ of the lines in $M_{l-1} \setminus M_l$ intersecting it. Finally, the descent structure, $\text{descent}(l+1, l)$, is the partition $H(M_{l+1}) \oplus H(M_l)$ obtained by superimposing $H(M_l)$ and $H(M_{l+1})$ on each other (Figure 5.5). Each trapezoid in $\text{descent}(l+1, l)$ is also given a pointer to the unique trapezoid in $H(M_l)$ containing it.

Let us turn to the updates. We shall only describe addition because deletion will be its exact reversal. Let us see how to add a new line S to $\text{sample}(M)$. We toss a fair coin repeatedly until we get failure. Let j be the number of successes obtained before getting failure. Our goal is to add S to levels 1 through $j+1$. For $1 \leq l \leq j+1$, let M'_l denote $M_l \cup \{S\}$. Addition of S to the l th level means: $H(M_l)$ is updated to $H(M'_l)$ and $\text{descent}(l+1, l)$ is accordingly updated. Addition of the line S to $H(M_l)$ takes $O(m_l)$ time: The addition procedure is just like the one in Section 2.8.2. The only difference is that in Section 2.8.2 we dealt with arrangements instead of their trapezoidal decompositions. But this only needs minor extensions.

Hence, we only need to worry about updating $\text{descent}(l+1, l)$, $1 \leq l \leq j+1$. Consider, first, the simpler case when $l = j+1$. In this case, the line S is to be added to M_l but not to M_{l+1} . Let $\Delta \in H(M_{l+1})$ be any trapezoid intersecting S . Let $\tilde{\Delta}$ be the restriction of $\text{descent}(l+1, l)$ to Δ . For every such trapezoid Δ , we need to update $\tilde{\Delta}$. This is done as follows (Figure 5.7(a)–(b)). First, we add S to the trapezoidal decomposition $\tilde{\Delta}$. This means that all vertical attachments in $\tilde{\Delta}$, such as the one through the intersection x in Figure 5.7(a), are contracted (see Figure 5.7(b)). We also add vertical attachments through new intersections on S , such as u and w in Figure 5.7(b). The cost of adding S to $\tilde{\Delta}$ is $O(|L(\Delta)|+1)$. This easily follows by applying the Zone Theorem (Theorem 2.8.2) to the restricted arrangement $G(M_l) \cap \Delta$.

Consider now the case when $l < j+1$ (Figure 5.7(c)–(d)). In this case, S is to be added to M_l as well as M_{l+1} . To take this into account, we only need to extend the above procedure of adding S to $\tilde{\Delta}$ as follows. First of all, if S intersects the lower (or upper) border of Δ , then the vertical attachment through this point of intersection w must extend to the opposite border of Δ (Figure 5.7(c)). This is because w belongs to the new trapezoidal decomposition associated with the $(l+1)$ th level. Hence, in $\text{descent}(l+1, l)$, the vertical attachment through w cuts through the lines in $L(\Delta)$ that do not

belong to the $(l + 1)$ th level. The cost of adding such an extended vertical attachment through w is clearly dominated by the size of its zone in the restricted arrangement $G(M'_l) \cap \Delta$ (shown shaded in Figure 5.7(c)), where $M'_l = M_l \cup \{S\}$. By the Zone Theorem, this cost is $O(|L(\Delta)| + 1)$. Finally, if S intersects the left or the right border of Δ then that border has to be contracted appropriately, as shown in Figure 5.7(d). This is because this border corresponds to the vertical attachment through an intersection v in

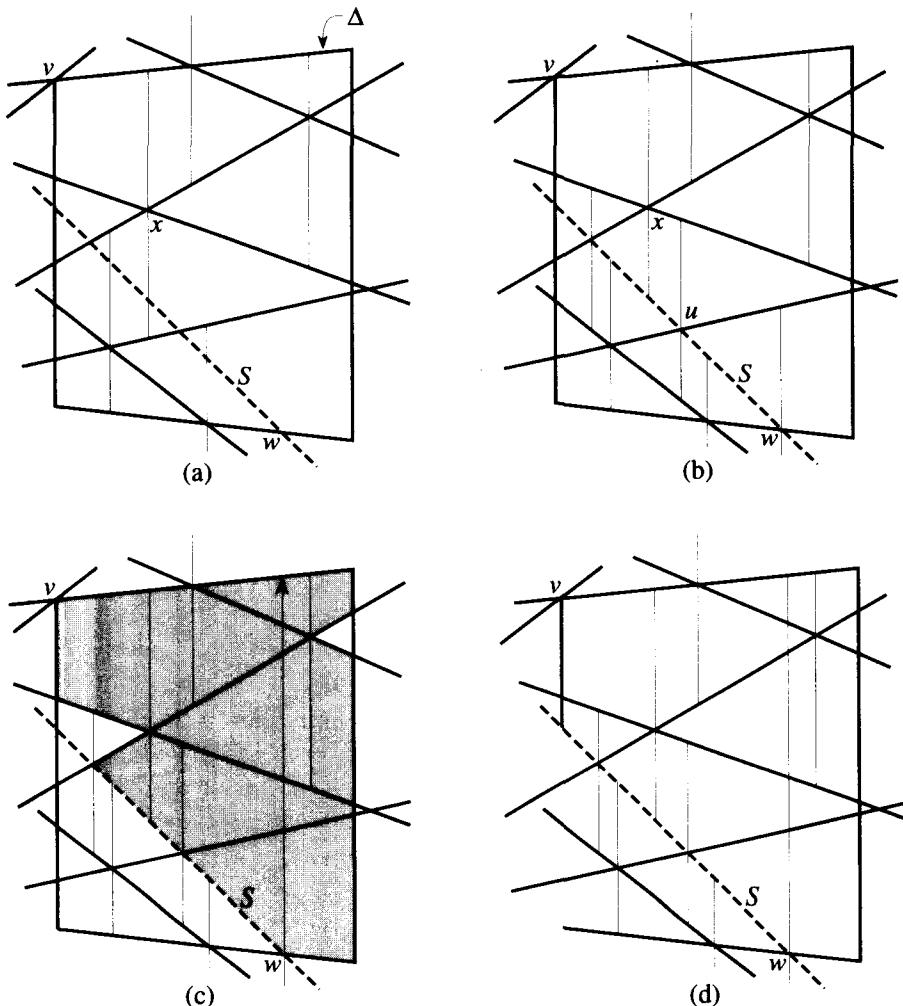


Figure 5.7: Addition of a new line S .

the old decomposition $H(M_{l+1})$. Hence, it gets split by the addition of S to the $(l+1)$ th level.

Analysis

It follows from the preceding discussion that the cost of updating the restriction of descent($l+1, l$) within any trapezoid $\Delta \in H(M_{l+1})$ is $O(|L(\Delta)| + 1)$. We have already seen that the conflict size $|L(\Delta)|$ is $\tilde{O}(\log m)$. Hence, the above cost is $\tilde{O}(\log m)$. By the Zone Theorem, the number of trapezoids in $H(M_{l+1})$ intersecting S is $O(m_{l+1})$. Hence, the total cost of updating descent($l+1, l$) is $\tilde{O}(m_{l+1} \log m)$.³ Summing over all levels, it follows that the total cost of updating the descent structures is $\tilde{O}(\log m \sum_l m_l) = \tilde{O}(m \log m)$. In fact, using a result to be proved in the next section, one can even establish a better $O(m)$ bound on the expected cost of update. As we have seen before, the cost of updating descent($l+1, l$) is proportional to $\sum_\Delta |L(\Delta)| + 1$, where Δ ranges over all trapezoids in $H(M_{l+1})$ intersecting S . The set M_{l+1} is a random sample of M_l of roughly half the size. Using a result in the next section (Exercise 5.5.1), it follows that the average conflict size of the trapezoids in $H(M_{l+1})$ intersecting S is $O(1)$. Here the conflict size is relative to M_l and the average is taken over all trapezoids in $H(M_{l+1})$ intersecting S . This means that the expected cost of updating descent($l+1, l$) is proportional to the number of trapezoids in $H(M_{l+1})$ intersecting S . By the Zone Theorem, this number is $O(m_{l+1})$. It follows that the expected cost of the update is $O(\sum_l m_l) = O(m)$.

To delete a line $S \in M$ from $\text{sample}(M)$, we simply delete S from the levels in $\text{sample}(M)$ containing S . The deletion of a line from a level is the exact reversal of the addition to a level. Hence, we do not elaborate it any further.

We summarize our main result as follows.

Theorem 5.4.1 *Let $G(M)$ be an arrangement of m lines in the plane. There exists a dynamic point location structure of $\tilde{O}(m^2)$ size allowing $\tilde{O}(\log m)$ query time that also allows insertion or deletion of a line in $\tilde{O}(m \log m)$ time. The expected cost of update is $O(m)$. Here the expectation is solely with respect to randomization in the data structure.*

³Here the probability of error, implied by the \tilde{O} notation, is inversely proportional to a large degree polynomial in m , not just m_{l+1} . This stronger property holds because Theorem 5.1.2 (or more precisely, Exercise 5.1.1 for Bernoulli sampling) implies that the size of the conflict list at each level is $\tilde{O}(\log m)$; put $s = n = m$ in that theorem. A similar property holds in what follows.

Exercise

5.4.1 Show how the space requirement of the data structure in this section can be made $O(m^2)$ (worst case) without affecting the query time. (Hint: Apply the clipping trick in Section 5.3.1.)

5.5 Average conflict size

We need to prove one general result about configuration spaces before giving further algorithmic applications of random sampling. For configuration spaces of bounded dimension, Theorem 5.1.2 gives a high-probability bound on the maximum conflict size of active configurations over a random sample. Can one get a better bound if one were only interested in the “average” conflict size? Yes. Here we shall show that the expected “average” conflict size is $O(n/r)$, roughly speaking. In fact, we shall show this for any configuration space. In other words, we do not need the configuration space to have bounded dimension.

Let $\Pi = \Pi(N)$ be any configuration space. Let R be a subset of N . Let $\Pi(R)$ denote the subspace induced by R . Let $\Pi^0(R)$ be the set of active configurations in $\Pi(R)$. For any configuration σ in $\Pi^0(R)$, let us denote the conflict size of σ relative to N by $l(\sigma)$. Let us define the total conflict size of $\Pi^0(R)$ as $\sum_{\sigma} l(\sigma)$, where σ ranges over all configurations in $\Pi^0(R)$. In general, for each integer $c \geq 0$, we define the c th order conflict size of $\Pi^0(R)$ as $\sum \binom{l(\sigma)}{c}$, where σ ranges over all configurations in $\Pi^0(R)$. Thus, the 0th order conflict size of $\Pi^0(R)$ is simply its size. We denote it, as usual, by $\pi^0(R)$. The first order conflict size is the ordinary conflict size, as defined before. Let the average conflict size of $\Pi^0(R)$ be defined as the total conflict size of $\Pi^0(R)$ divided by its size, $\pi^0(R)$.

Now assume that R is a random sample of N of size r . By Theorem 5.1.2, the maximum conflict size of every configuration in $\Pi^0(R)$ is $O([n/r] \log r)$, with high probability, if the configuration space has bounded dimension. Our goal is to prove that the expected value of the average conflict size is roughly $O(n/r)$. In other words, we want to prove that the expected total conflict size of $\Pi^0(R)$ is $O((n/r)E[\pi^0(R)])$, where $E[\pi^0(R)]$ denotes the expected size of $\Pi^0(R)$. Well, we cannot really prove this. The following lemma proves a somewhat weaker statement.

For a positive integer c , let $\pi^c(R)$ denote the size of $\Pi^c(R)$, the set of configurations in $\Pi(R)$ with level c . If c is a small constant, this is the set of configurations in $\Pi(R)$ that are “almost” active. Figure 5.8 shows $\Pi^1(R)$ when the objects under consideration are half-spaces (Example 3.4.2).

Lemma 5.5.1 *Let $\Pi(N)$ be any configuration space. If R is a random sample of N of size r , then the expected total conflict size of $\Pi^0(R)$ is bounded,*

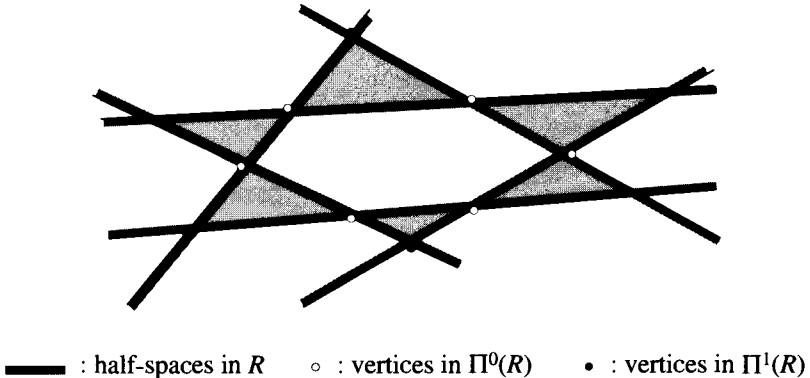


Figure 5.8: Vertices with level 1.

within a constant factor, by $(n/r)E[\pi^1(R)]$. In general, for any integer $c \geq 0$, the expected c th order conflict size of $\Pi^0(R)$ is bounded, within a constant factor, by $(n/r)^c E[\pi^c(R)]$.

The analogue for Bernoulli sampling is the following with $p = r/n$.

Lemma 5.5.2 Let $\Pi(N)$ be any configuration space. Fix a coin with probability of success $p > 0$, which we allow to be a function of n . Let $R \subseteq N$ be a random subset obtained by flipping this coin independently for each object in N and putting it in R iff the toss is successful. Then the expected total conflict size of $\Pi^0(R)$ is bounded, within a constant factor, by $(1/p)E[\pi^1(R)]$. In general, for any integer $c \geq 0$, the expected c th order conflict size of $\Pi^0(R)$ is bounded, within a constant factor, by $(1/p^c)E[\pi^c(R)]$.

The proofs of both lemmas are similar. We shall only prove Lemma 5.5.2 because that is the only one we shall need in the sequel. We shall leave the analogous proof of Lemma 5.5.1 to the reader.

For later reference, we shall actually prove a slightly stronger result than Lemma 5.5.2. For any integer $b \geq 0$, define the c th order conflict size of $\Pi^b(R)$ as

$$\sum_{\sigma \in \Pi^b(R)} \binom{l-b}{c}.$$

When $b = 0$, this coincides with the previous definition. As a convention, $\binom{l-b}{c} = 0$, if $l < b$. Then:

Lemma 5.5.3 For fixed integers $b, c \geq 0$, the expected c th order conflict size of $\Pi^b(R)$ is bounded, within a constant factor, by $(1/p^c)E[\pi^{b+c}(R)]$.

Proof. Fix a configuration $\sigma \in \Pi(N)$ with conflict size $l(\sigma)$ and degree $d(\sigma)$. It occurs in $\Pi^a(R)$, $a \geq 0$, iff R contains all $d(\sigma)$ triggers and exactly a of

the $l(\sigma)$ stoppers associated with σ . Hence,

$$\text{prob}\{\sigma \in \Pi^a(R)\} = \binom{l(\sigma)}{a} p^{a+d(\sigma)} (1-p)^{l(\sigma)-a}. \quad (5.7)$$

Since

$$E[\pi^{b+c}(R)] = \sum_{\sigma \in \Pi(N)} \text{prob}\{\sigma \in \Pi^{b+c}(R)\},$$

we get

$$E[\pi^{b+c}(R)] = \sum_{\sigma \in \Pi(N)} \binom{l(\sigma)}{b+c} p^{b+c+d(\sigma)} (1-p)^{l(\sigma)-b-c}. \quad (5.8)$$

The expected value of the c th order conflict size of $\Pi^b(R)$ is

$$\sum_{\sigma \in \Pi(N)} \binom{l(\sigma) - b}{c} \text{prob}\{\sigma \in \Pi^b(R)\}.$$

By (5.7), this is

$$\sum_{\sigma \in \Pi(N)} \binom{l(\sigma) - b}{c} \binom{l(\sigma)}{b} p^{b+d(\sigma)} (1-p)^{l(\sigma)-b}. \quad (5.9)$$

The lemma follows by comparing (5.8) and (5.9) and noting that

$$\binom{l(\sigma)}{b+c} \binom{b+c}{c} = \binom{l(\sigma)}{b} \binom{l(\sigma) - b}{c}. \quad \square$$

The bounds given in the preceding lemmas are not useful as they stand. They become useful when combined with the following lemma, which we already proved as Lemma 4.3.2. (In case the reader has not read Chapter 4, the proof of this lemma can be understood independently of the rest of that chapter.)

Lemma 5.5.4 *Let r be the size of R . For any fixed $a \geq 0$, $\pi^a(R) = O(E[\pi^0(Q)])$, where Q is a random sample of R of size $r/2$. The same result also holds if Q is chosen by flipping a fair coin for each object in R and putting it in Q iff the toss is successful.*

Now, let $\Pi = \Pi(N)$ be a fixed configuration space. Let $e(r) = e(\Pi, r)$ denote the expected size of $\Pi^0(R)$, where R is a random sample of N of size r . Let $\bar{e}(r) = \max\{e(i) \mid i \leq r\}$. Combining Lemmas 5.5.1 to 5.5.4, we get the following.

Theorem 5.5.5 Let $\Pi(N)$ be any configuration space.

1. Let R be a random sample of N of size r . Let $c > 0$ be a fixed integer. The expected value of the c th order conflict size of $\Pi^0(R)$, or more generally, $\Pi^b(R)$, for a constant b , is $O((n/r)^c e(r/2))$. It is also $O((n/r)^c \bar{e}(r))$.
2. If R is chosen by Bernoulli sampling using a coin with probability of success p , then the expected value of the c th order conflict size of $\Pi^0(R)$, or more generally $\Pi^b(R)$, for a constant b , is $O([1/p]^c E[\pi^0(Q)])$, where Q is obtained from R by Bernoulli sampling using a fair coin. (Thus, it is as if Q is obtained from N by Bernoulli sampling, using a coin of bias $p/2$.)

In this theorem, the Big-Oh notation contains a hidden constant factor 2^d , where d is the maximum trigger size of a configuration in $\Pi(N)$. By the definition of a configuration space, d is bounded. We shall reduce the constant term within the Big-Oh notation in the exercises.

Let us give a few applications of the theorem.

Example 5.5.6 (Segments in the plane)

Let N be a set of segments in the plane. Let $\Pi(N)$ be the configuration space of feasible trapezoids over N (Example 3.4.1). For any subset $R \subseteq N$, $\Pi^0(R)$ is just the set of trapezoids in the trapezoidal decomposition formed by R . Its size is proportional to the size of R plus the number of intersections among the segments in R . Hence, we might as well define $\pi^0(R)$ to be the size of R plus the number of intersections among the segments in R . In that case, it is clear that $e(i) \leq e(j)$, whenever $i \leq j \leq n$. Hence, $\bar{e}(r) = e(r)$. It follows from the above theorem that, for a random subset $R \subseteq N$ of size r , the expected conflict size of $\Pi^0(R)$ is $O([n/r]e(r)) = O([n/r] E[\pi^0(R)])$. In other words, the average conflict size of the trapezoids in the trapezoidal decomposition formed by R is $O(n/r)$. The same also holds for $\Pi^b(R)$, for a constant b .

Similarly, if R is chosen by Bernoulli sampling using a coin of bias p , then the average conflict size is $1/p$. An important special case occurs when p is a constant. In this case, it follows that the expected total conflict size of $\Pi^0(R)$ is $O(E[\pi^0(R)]) = O(\pi^0(N))$. The same also holds for the b th order conflict size, for any constant b .

Example 5.5.7 (Voronoi diagrams)

Let N be a set of sites in the plane. Let $\Pi(N)$ be the configuration space of feasible triangles over N (Example 3.4.3). For any subset $R \subseteq N$, $\Pi^0(R)$ is just the set of triangles in the radial triangulation of the Voronoi diagram formed by R . Its size is $O(r)$, because the size of a planar Voronoi diagram

is linear in the number of sites (Section 2.5). Hence, $\bar{e}(r) = O(r)$. It follows from the above theorem that, for a random subset $R \subseteq N$ of size r , the expected conflict size of $\Pi^0(R)$ is $O(n)$. In other words, the average conflict size of the triangles in the radial triangulation formed by R is $O(n/r)$. The same also holds for $\Pi^b(R)$, for a constant b . Similarly, if R is chosen by Bernoulli sampling using a coin of bias p , then the average conflict size is $1/p$.

Exercises

5.5.1 Let N be any set of lines in the plane. Let $\Pi(N)$ be the configuration space of feasible trapezoids over N . Fix any line $S \notin N$. Let $\Pi_S(N)$ be the subspace of $\Pi(N)$ consisting of the feasible trapezoids intersecting S . Thus, for any $R \subseteq N$, $\Pi_S^0(R)$ is just the set of trapezoids intersecting S in the trapezoidal decomposition formed by R . Now let R be a random sample of N of size r . Show that the average conflict size of the trapezoids in $\Pi^0(R)$ is $O(n/r)$. (Hint: Proceed as in Example 5.5.6. By the Zone Theorem for the arrangements of lines, $e(r) = O(r)$ in the present case.)

5.5.2 Apply Theorem 5.5.5 to the configuration space of feasible vertices of a convex polytope (Example 3.4.2). Conclude that the expected average conflict size over a random sample of size r is $O(n/r)$.

5.5.3 We did not give much attention to the constant factors in this section. Show that, for $c = 1$, the hidden constant factor in the second bound in Theorem 5.5.5.1 can be made linear in d , the maximum degree of a configuration in $\Pi(N)$. (Hint: Instead of taking a random sample of size $r/2$, as in Lemma 5.5.4, take a random sample of size $dr/(d+1)$. Do precise calculations using binomial coefficients and Stirling's approximation.)

5.6 More dynamic algorithms

We are now in position to give more algorithmic applications of the techniques in this chapter. In this section, we shall give dynamic algorithms for a few two-dimensional problems. With that in mind, we shall first formulate the dynamic sampling technique (Section 5.4) in a general setting of configuration spaces. Our formulation will be akin to the formulation we used in Section 3.4 for incremental algorithms. We shall also prove general results that will be useful later.

Let M denote the set of objects under consideration existing at any given time. Assume that one can associate with M a configuration space $\Pi(M)$. This will depend on the problem under consideration. The goal of an abstract dynamic algorithm is to maintain the set $\Pi^0(M)$ of active configurations in $\Pi(M)$. In addition, it also maintains a search structure $\text{sample}(M)$. We are only interested in the overall organization of $\text{sample}(M)$ and not its exact

nature. The state of $\text{sample}(M)$ is determined by its grading $M = M_1 \supseteq M_2 \supseteq \dots$. This grading is independent of the past. In what follows, we shall denote the size of M_l by m_l . Conceptually, M_l is obtained from M_{l-1} by flipping a fair coin. The set $\Pi^0(M_l)$ is stored in the l th level of $\text{sample}(M)$. We also associate with each active configuration $\sigma \in \Pi^0(M_l)$ its conflict list relative to M_{l-1} . In fact, we can store with σ any kind of information whose space requirement is within a bounded power of its conflict size (relative to M_{l-1}). We can afford to do so because M_l is a random sample of M_{l-1} of roughly half the size, and hence the average conflict size of the configurations in $\Pi^0(M_l)$ is $O(1)$ (Section 5.5). More formally:

Lemma 5.6.1 *Fix any constant integer $c \geq 0$. Consider the total c th order conflict size of all $\Pi^0(M_l)$. Its expected value has the same order as the expected total size of all $\Pi^0(M_l)$.*

More generally, for a constant $b \geq 0$, the expected total c th order conflict size of $\Pi^b(M_l)$, over all l , has the same order as the expected total size of all $\Pi^0(M_l)$.

Proof. M_l is obtained by flipping a fair coin independently for each object in M_{l-1} . Hence, for a fixed M_{l-1} , the expected c th order conflict size of $\Pi^0(M_l)$ (relative to M_{l-1}) has the same order as the expected size of $\Pi^0(Q)$, where Q is obtained from M_l by flipping a fair coin for each object in M_l (Theorem 5.5.5). In other words, Q can be just thought of as $M_{l+1}!$ The first part of the lemma follows by summing over all l .

The second part follows in exactly the same fashion by applying Theorem 5.5.5 to $\Pi^b(M_l)$. \square

Now let us turn to the updates on $\text{sample}(M)$. In the case of arrangements, we were able to get a good bound for every update sequence. In general, we might not be so lucky because an adversary can come up with pathological update sequences that can force unusually large structural change in $\Pi^0(M)$. For the same reasons as in Chapter 4, we shall find it illuminating in such cases to analyze the performance of our algorithm on a random sequence of updates. A random sequence of updates is defined as in the introduction of Chapter 4.

For example, consider the deletion of a random object S from M . Let $M' = M \setminus \{S\}$. What is the total expected change as we pass from $\text{sample}(M)$ to $\text{sample}(M')$? Let us elaborate what we mean by the expected change. Let us assume that the size of the information stored with each active configuration $\sigma \in \Pi^0(M_l)$ is bounded by $l(\sigma)^b$, for some constant b . We define the (b th order) *change* as $\sum_\sigma l(\sigma)^b$, where σ ranges over the destroyed configurations in $\text{sample}(M)$ and the newly created configurations in $\text{sample}(M')$. We say that a configuration $\sigma \in \Pi^0(M_l)$ is destroyed if it is no longer present in

$\Pi^0(M'_l)$. A newly created configuration in $\Pi^0(M'_l)$ is the one that was not present in $\Pi^0(M_l)$.

Theorem 5.6.2 For a fixed $b > 0$, the expected b th order change during the deletion of a random object S from $\text{sample}(M)$ is bounded, within a constant factor, by $1/m$ times the expected size of $\text{sample}(M)$.

Proof. Let ϕ be the sum of the (b th order) changes obtained by letting S range over all objects in M . We shall show that its expected value has the same order as the expected size of $\text{sample}(M)$. This expectation is solely with respect to randomization in the data structure. Since the deleted object is randomly chosen from M , the lemma will follow immediately.

A configuration $\sigma \in \Pi^0(M_l)$ contributes to the change iff it is destroyed, i.e., if the deleted object is one of the triggers associated with it. Let us sum the contributions of such destroyed configurations, by letting the deleted object S range over all objects in M . Each configuration in $\Pi^0(M_l)$ contributes $O(1)$ times, once for each of its triggers. Hence, this sum is proportional to the total b th order conflict size of all $\Pi^0(M_l)$. Its expected value has the same order as the expected size of $\text{sample}(M)$ (Lemma 5.6.1).

Now let us turn to the contribution of the newly created configurations. Consider any level l that is affected during the deletion. Let S denote the deleted object. Consider a newly created configuration σ in $\Pi^0(M'_l)$, where $M'_l = M_l \setminus \{S\}$. Clearly, S must be a stopper associated with σ ; otherwise σ will be present in $\Pi^0(M_l)$. This means σ can also be thought of as a configuration in $\Pi^1(M_l)$, with S being its only stopper in M_l . Thus, newly created configurations in $\Pi^0(M'_l)$ can be identified with the configurations in $\Pi^1(M_l)$ in conflict with S . (We have used a similar argument before. See Figure 4.4.) Let us sum the contributions of the newly created configurations, by letting S range over all of M . Each configuration in $\Pi^1(M_l)$ contributes precisely once, for its unique stopper in M_l . It follows that this sum is bounded by the b th order conflict size of all sets $\Pi^1(M_l)$. Its expected value has the same order as the expected size of $\text{sample}(M)$ (Lemma 5.6.1). \square

Next, let us turn to the addition of a new object S to $\text{sample}(M)$. Let $M'' = M \cup \{S\}$. Note that the change, as we pass from $\text{sample}(M)$ to $\text{sample}(M'')$, is the same as the change that occurs as we pass from $\text{sample}(M'')$ back to $\text{sample}(M)$ via an imaginary deletion of S from M'' . If the update sequence under consideration is random, then each object in M'' is equally likely to occur as S . This lets us estimate the expected cost of a random addition by applying the above lemma to this imaginary random deletion.

In the next few sections, we shall apply the dynamic sampling technique to a few more two-dimensional problems.

5.6.1 Point location in trapezoidal decompositions

In this section, we shall give yet another dynamic point location structure for trapezoidal decompositions. This will be based on the dynamic sampling technique (Section 5.4). It is a simple generalization of the point location structure for arrangements of lines that we described earlier (Section 5.4.1). The reader should compare this search structure with the ones described in Chapter 4. First, we shall describe our search structure in a static setting. We shall turn to its dynamization later.

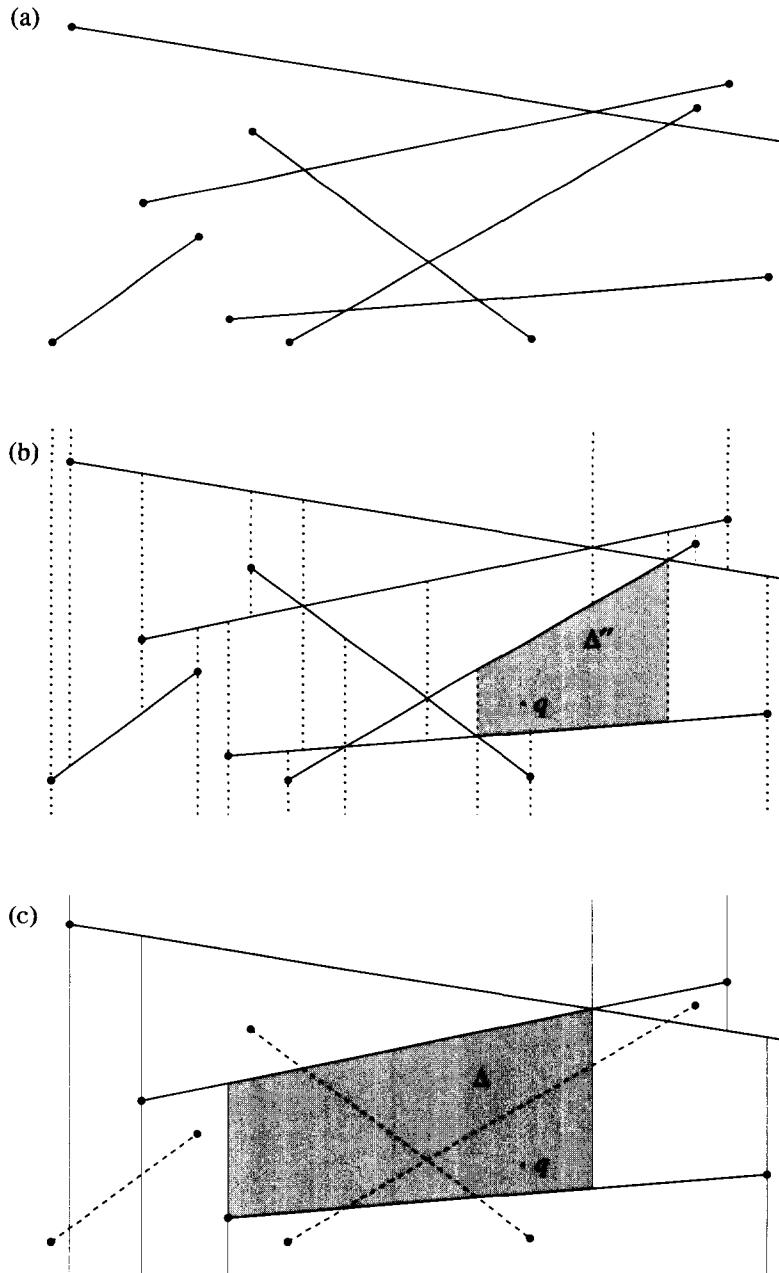
Static setting

Let $H(N)$ be a trapezoidal decomposition formed by a set N of segments in the plane. We can associate a point location structure with $H(N)$ using bottom-up sampling along the same lines as in Section 5.3.1. Let us denote this search structure by $\text{sample}(N)$. It is defined as follows. We first grade N by repeatedly tossing a fair coin. Let $N = N_1 \supseteq N_2 \supseteq \dots \supseteq N_{r-1} \supset N_r = \emptyset$ be the resulting gradation. With each level l , $1 \leq l \leq r$, we associate the trapezoidal decomposition $H(N_l)$. We also store with each trapezoid $\Delta \in H(N_l)$ a conflict list $L(\Delta)$ of the segments in $N_{l-1} \setminus N_l$ intersecting it. Finally, we define the descent structure $\text{descent}(l+1, l)$ as the partition $H(N_{l+1}) \oplus H(N_l)$ obtained by superimposing $H(N_l)$ and $H(N_{l+1})$ on each other (Figure 5.9(d)). We also associate with each trapezoid in $\text{descent}(l+1, l)$ a pointer to the unique trapezoid in $H(N_l)$ containing it.

$\text{Sample}(N)$ has the following important property. For each l , N_{l+1} is a random sample of N_l of roughly half the size. Hence, the conflict size of each trapezoid in $H(N_{l+1})$ relative to N_l is $\tilde{O}(\log n)$ (Theorem 5.1.2, Exercise 5.1.1). Moreover, the expected value of the average conflict size of a trapezoid in $H(N_{l+1})$ is $O(1)$ (Example 5.5.6).

First, let us see how $\text{sample}(N)$ can be used for point location (Figure 5.9). Let q be a fixed query point. Locating q in the last empty level is trivial. Inductively, assume that $\Delta = \Delta_{l+1}$ is the trapezoid in $H(N_{l+1})$ containing the query point q , where $1 \leq l < r$. Let $\tilde{\Delta}$ denote the restriction of the partition $\text{descent}(l+1, l)$ within Δ . We claim that the trapezoid $\Delta' \in \tilde{\Delta}$ containing q can be found in $O(|L(\Delta)| + 1)$ time. Once this is done, the trapezoid $\Delta'' = \Delta_l \in H(N_l)$ containing q is just the one containing Δ' .

Δ' can be located as follows. Figure 5.10 illustrates a different and slightly more complex situation than the one that arises in Figure 5.9. Let $\hat{\Delta}$ be the convex decomposition of Δ that is obtained by erasing all vertical attachments in $\tilde{\Delta}$ through segment intersections. We assume that $\text{descent}(l+1, l)$ is stored in such a fashion that $\hat{\Delta}$ is available to us. Let Q be the segment in N_l immediately above q . (We can assume that Q always exists by adding a dummy segment at infinity.) Obviously, Q is either the segment bound-

Figure 5.9: (a) N_l . (b) $H(N_l)$. (c) $H(N_{l+1})$. (continued)

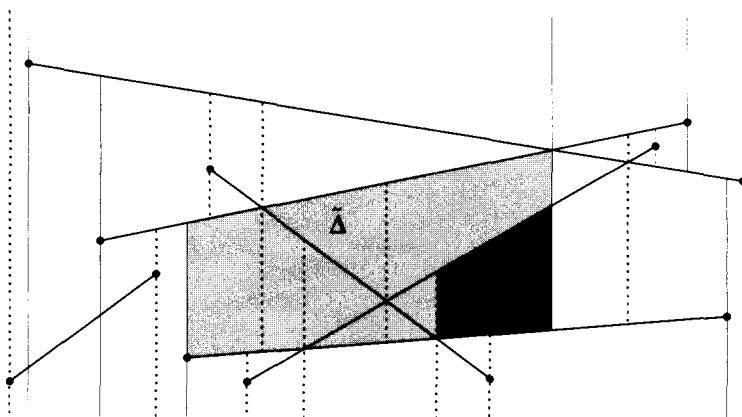


Figure 5.9: (d) Descent($l + 1, l$).

ing the upper side of Δ or it belongs to $L(\Delta)$. Hence, it can be found in $O(|L(\Delta)| + 1)$ time. Let q' be the intersection of the vertical ray through q with Q . We locate q' in $\tilde{\Delta}$ by a linear search on Q . This takes $O(|L(\Delta)| + 1)$ time. It also locates the convex region $\gamma \in \tilde{\Delta}$ containing q . The sought trapezoid Δ' is obtained by a linear search through the $O(|L(\Delta)| + 1)$ trapezoids of $\tilde{\Delta}$ contained in γ . Now the trapezoid $\Delta'' \in H(N_l)$ containing q is just the unique one containing Δ' .

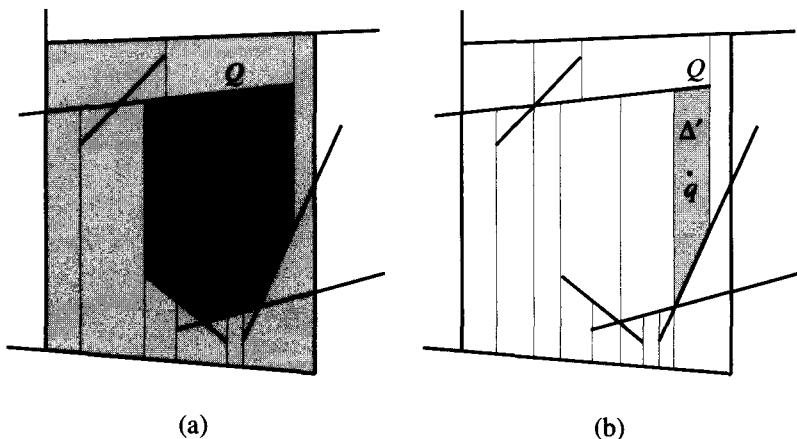


Figure 5.10: Point location: (a) $\hat{\Delta}$. (b) $\tilde{\Delta}$.

Analysis

The preceding discussion implies that we can descend from level $l+1$ to level l in $O(|L(\Delta_{l+1})| + 1)$ time. We have already noted that $|L(\Delta_l)| = \tilde{O}(\log n)$, for all l . Moreover, the number of levels in our data structure is $\tilde{O}(\log n)$. It follows that the time required to locate a fixed query point q is $\tilde{O}(\log^2 n)$. The following lemma shows that the query time is, in fact, $\tilde{O}(\log n)$.

Lemma 5.6.3 *For a fixed query point q , $\sum_{i \geq 1} |L(\Delta_i)|$ is $\tilde{O}(\log n)$, where Δ_i is the trapezoid containing q in $H(N_i)$.*

Proof. The proof is almost a verbatim translation of the proof of Lemma 5.3.1. Hence, we shall leave it to the reader. \square

We showed above that the query time is $\tilde{O}(\log n)$ for a fixed query point. We further note that the number of distinct combinatorial search paths in our data structure is bounded by a polynomial of a fixed degree in n . More precisely, let $\bar{H}(N)$ be the refinement of $H(N)$ obtained by passing infinite vertical lines through all intersections and endpoints of the segments in N . Then, for a fixed region R in $\bar{H}(N)$, it is easy to see that the search path in $\text{sample}(N)$ remains the same if the query point lies anywhere in R . This is because $\bar{H}(N)$ is a refinement of every $H(N_i)$. It follows that the maximum query cost is also $\tilde{O}(\log n)$ (Observation 1.3.1).

Let us bound the expected space requirement of $\text{sample}(N)$. At first, let us ignore the space requirement of the descent structures and the conflict lists. In that case, the size of $\text{sample}(N)$ is proportional to the total size of the partitions $H(N_i)$. Their total size is $\sum_i n_i + k_i$, where k_i is the number of intersections among the segments in N_i and n_i is the size of N_i . The expected value of n_i is clearly $n/2^i$. Let k be the number of intersections among the segments in N . Then the expected value of k_i is $O(k/4^i)$. This is because an intersection v between two segments in N occurs in the i th level iff both segments containing v occur in the i th level. This happens with probability $1/4^i$.

It follows that the expected total size of all $H(N_i)$ is proportional to

$$\sum_i \frac{n}{2^i} + \frac{k}{4^i} = O(n + k).$$

Next, let us turn to the space requirement of the descent structures and the conflict lists. What is the size of $\text{descent}(l+1, l)$? For any trapezoid $\Delta \in H(N_{l+1})$, the restriction of $\text{descent}(l+1, l)$ to Δ is a trapezoidal decomposition formed by the segments in N_l in conflict with Δ . Hence, its size is at most quadratic in $l(\Delta)$, the conflict size of Δ relative to N_l . In other words, the size of $\text{descent}(l+1, l)$ is bounded by the second-order conflict size of the trapezoids in $H(N_{l+1})$. Thus, the total size of $\text{sample}(N)$ is bounded by the

second-order conflict size of all $H(N_l)$. Its expected value has the same order as the expected total size of all $H(N_l)$ (Lemma 5.6.1).

We have already seen that the expected total size of all partitions $H(N_l)$ is $O(n + k) = O(|H(N)|)$. Here $|H(N)|$ denotes the size of $H(N)$. It follows that the expected size of $\text{sample}(N)$ is $O(n + k)$.

To summarize:

Theorem 5.6.4 *The expected space requirement of $\text{sample}(N)$ is $O(|H(N)|)$, that is, $O(n + k)$, where k is the number of intersections among the segments in N . The point location cost is $\tilde{O}(\log n)$.*

Dynamization

Now we shall dynamize the search structure described above. Let M denote the set of current (undeleted) segments at any given moment. Let $H(M)$ denote the resulting trapezoidal decomposition. We shall denote the search structure associated with M by $\text{sample}(M)$. It looks as if it were constructed by applying the previous static method to the set M .

We shall only describe the addition operation because deletion will be its exact reversal. Let us see how to add a segment S to $\text{sample}(M)$. We toss a fair coin repeatedly until we get failure. Let j be the number of successes obtained before getting failure. Our goal is to add S to levels 1 through $j + 1$. For $1 \leq l \leq j + 1$, let M'_l denote $M_l \cup \{S\}$. Addition of S to the l th level means: $H(M_l)$ is updated to $H(M'_l)$ and $\text{descent}(l + 1, l)$ is accordingly updated. This will be done as follows. Let a and b be the two endpoints of S . We first locate a using $\text{sample}(M)$. By the nature of our point location algorithm, this tells us, for every l , the trapezoid containing a in $H(M_l)$ as well as $\text{descent}(l + 1, l)$. After this, $H(M_l)$ can be updated to $H(M'_l)$ very much as in Section 3.1 by traveling along S from a to b in $H(M_l)$. This takes time proportional to $\sum \text{face-length}(\Delta)$, where Δ ranges over all trapezoids in $H(M_l)$ intersecting S . In a similar fashion, we can update the partition $\text{descent}(l + 1, l)$ by walking along S from a to b . This takes time proportional to $\sum \text{face-length}(\Gamma)$, where Γ ranges over all trapezoids in $\text{descent}(l + 1, l)$ intersecting S .

Analysis

It follows from the preceding discussion that the total cost of adding S to $\text{sample}(M)$ is proportional to

$$\sum_{\Delta} \text{face-length}(\Delta) + \sum_{\Gamma} \text{face-length}(\Gamma),$$

where Δ and Γ range, respectively, over the trapezoids intersecting S in $H(M_l)$ and $\text{descent}(l + 1, l)$, for $1 \leq l \leq j + 1$. We are going to analyze the

addition backwards (as usual). Hence, it will be more convenient to rewrite the above expression in terms of $\text{sample}(M')$ instead of $\text{sample}(M)$. After this rewriting, we get the following.

Lemma 5.6.5 *The total cost of adding S to $\text{sample}(M)$ is proportional to*

$$\sum_{\Delta'} \text{face-length}(\Delta') + \sum_{\Gamma'} \text{face-length}(\Gamma'),$$

where Δ' and Γ' range, respectively, over all trapezoids adjacent to S in $H(M'_l)$ and the new partition $\text{descent}(l+1, l)$. Here l ranges over all levels of $\text{sample}(M')$ containing S .

To delete a segment $S \in M$ from $\text{sample}(M)$, we simply delete S from the levels in $\text{sample}(M)$ containing S . Deletion from a level is the exact reversal of addition to a level. So we only state:

Lemma 5.6.6 *The total cost of deleting S from $\text{sample}(M)$ is proportional to*

$$\sum_{\Delta} \text{face-length}(\Delta) + \sum_{\Gamma} \text{face-length}(\Gamma),$$

where Δ and Γ range, respectively, over all trapezoids adjacent to S in $H(M_l)$ and $\text{descent}(l+1, l)$. Here l ranges over all levels of $\text{sample}(M)$ containing S .

We shall now estimate the expected cost of maintaining $\text{sample}(M)$ over a random update sequence. By a random update sequence, we mean a random (N, δ) -sequence as defined in the introduction of Chapter 4.

If one compares Lemmas 5.6.5 and 5.6.6, one notices that the cost of adding S to $\text{sample}(M)$ is the same as the cost of the imaginary deletion of S from $\text{sample}(M')$, where $M' = M \cup \{S\}$. If the update sequence is random, every segment in M' is equally likely to be involved in this addition. Thus the expected cost of a random addition to $\text{sample}(M)$ is the same as the expected cost of a random deletion from the search structure $\text{sample}(M')$ that results after this addition. Hence, in what follows, we shall only estimate the expected cost of a random deletion from $\text{sample}(M)$.

Let us sum the cost of deleting S from $\text{sample}(M)$ by letting S range over all segments in M . Let ϕ denote this sum. By Lemma 5.6.6, a trapezoid in $H(M_l)$ or $\text{descent}(l+1, l)$ contributes to this sum iff the segment S to be deleted is one of the bounded number of segments adjacent to it. Thus, ϕ is bounded, within a constant factor, by the total size of all $H(M_l)$ and $\text{descent}(l+1, l)$, or in other words, by the size of $\text{sample}(M)$. We have already seen that the expected size of $\text{sample}(M)$ is $O(|H(M)|)$. Hence, it follows that the expected value of ϕ is $O(|H(M)|)$.

If the segment $S \in M$ to be deleted is random, the expected cost of deletion is $E[\phi]/m = O(|H(M)|/m)$. As we have already noted, the expected

cost of a random addition is the same as the expected cost of an imaginary random deletion from the resulting data structure.

We summarize our main result as follows:

Theorem 5.6.7 *The expected cost of a random deletion from $\text{sample}(M)$ is proportional to $|H(M)|/m = O(1 + I(M)/m)$, where $I(M)$ denotes the number of intersections among the segments in M . The expected cost of a random addition to $\text{sample}(M)$ is proportional to $|H(M')|/m'$, where M' is the set of segments resulting after addition, and m' is the size of M' . This excludes the $\tilde{O}(\log m)$ cost of locating an endpoint of the segment being added.*

As a corollary, we get the following.

Corollary 5.6.8 *The expected cost of executing a random (N, δ) -sequence is $O(n \log n + I)$, where I is the number of intersections among the segments in N .*

Proof. This follows directly from Theorem 5.6.7. The $O(n \log n)$ term accounts for the cost of point location. \square

Exercises

5.6.1 Show that in the static setting, $\text{sample}(N)$ can be built in $O(n \log n + k)$ time, where k is the number of intersections among the segments in N . (Hint: Of course, one could build $\text{sample}(N)$ by just adding the segments in N one at a time in random order, using the addition procedure in this section. What is expected is a simpler procedure that builds $\text{sample}(N)$ level by level, from the highest to the lowest level.)

5.6.2 Prove Lemma 5.6.3. (Hint: The only difference is encountered in proving the analogue of Lemma 5.3.2 in the present setting. Specify carefully the sets R_q and L_q in the present setting, along with the linear orders on them.)

5.6.2 Point location in Voronoi diagrams

We shall give in this section yet another dynamic point location structure for Voronoi diagrams. This will be based on the dynamic sampling technique (Section 5.4). The reader should compare this search structure with the ones described in Chapter 4. First, we shall describe our search structure in the static setting. We shall turn to its dynamization later.

Static setting

Let N be a set of sites in the plane. Let $G(N)$ be its Voronoi diagram. Let $H(N)$ be the radial triangulation of $G(N)$ (Section 3.3) obtained by connecting each site to the vertices of its Voronoi region. Our goal is to associate a point location structure with $H(N)$ using bottom-up sampling.

Once we locate the triangle in $H(N)$ containing the query point, we also know the nearest site to that query point.

Bottom-up sampling can be used as follows (Figure 5.11). Let

$$N = N_1 \supseteq \cdots \supseteq N_r = \emptyset$$

be the gradation of N obtained by tossing a fair coin. We store at the level l of our data structure the triangulation $H(N_l)$. With each triangle $\Delta \in H(N_l)$, we associate a list $L(\Delta)$ of the sites in $N_{l-1} \setminus N_l$ conflicting with it. A site conflicts with Δ if its imaginary addition to $H(N_l)$ will destroy Δ .

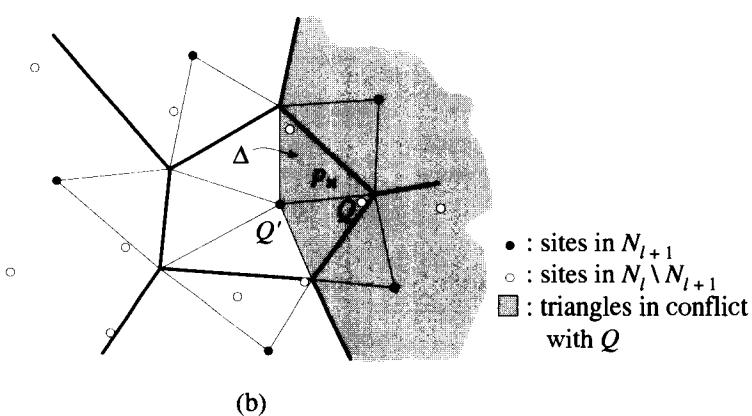
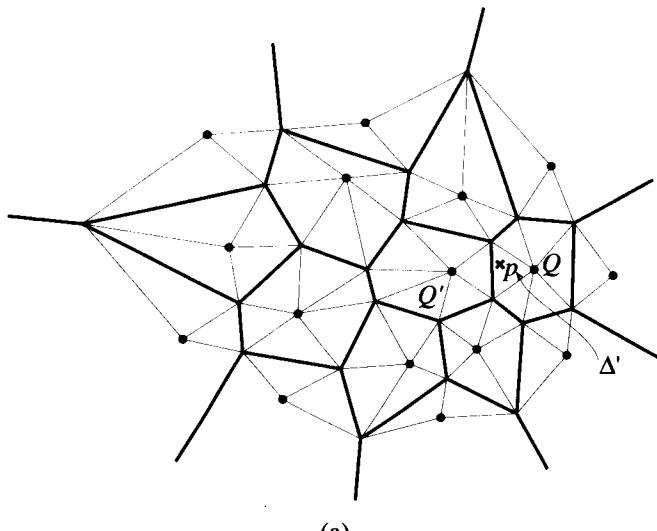


Figure 5.11: (a) $H(N_l)$. (b) $H(N_{l+1})$.

(This happens iff the site is nearer to some point in Δ than the site labeling the Voronoi region containing Δ .) Finally, we let the descent structure, $\text{descent}(l, l - 1)$, consist of a balanced binary tree, called the *radial tree*, for each Voronoi region of $G(N_{l-1})$. It is defined very much as in Section 3.3. Given a query point lying in a given Voronoi region of $G(N_{l-1})$, the associated radial tree should let us locate the triangle in $H(N_{l-1})$ containing the query point in logarithmic time.

Queries are answered as follows (Figure 5.11). Let q be any query point. As N_r is empty, locating q in the last level r is trivial. Inductively, assume that we have located the triangle $\Delta = \Delta_{l+1}$ in $H(N_{l+1})$ containing q . Let Q' be the site in N_{l+1} labeling the Voronoi region containing Δ . The site Q in N_l that is nearest to q is either Q' or it is contained in the conflict list $L(\Delta)$. Hence, it can be determined in $O(|L(\Delta)|) = \tilde{O}(\log n)$ time. At this stage, we know that q is contained in the Voronoi region in $G(N_l)$ labeled with Q . We can use the radial tree associated with this region to locate the triangle $\Delta' = \Delta_l$ in $H(N_l)$ containing q . This takes $O(\log n)$ time.

Analysis

The preceding discussion shows that the descent from level $l + 1$ to level l takes logarithmic time. The total number of levels is $\tilde{O}(\log n)$. Hence, the total cost of locating q in $H(N)$ is $\tilde{O}(\log^2 n)$.

The above query cost is for a fixed query point. But it can be checked that the number of distinct search paths in our search structure is bounded by a polynomial of a fixed degree in n . Hence, the depth of our search structure is $\tilde{O}(\log n)$ (Observation 1.3.1). It follows that the cost of locating any query point, not just a fixed one, is $\tilde{O}(\log^2 n)$.

Next, we shall show that the expected space requirement of $\text{sample}(N)$ is $O(n)$. We can ignore the cost of storing the conflict information. Formally, this follows from Lemma 5.6.1. Informally, this follows because each N_i is a random sample of N_{i-1} of roughly half the size. Hence, the average conflict size of a triangle in $H(N_i)$ relative to N_{i-1} is $O(1)$ (Example 5.5.7).

So we only need to estimate the expected total size of all partitions $H(N_i)$. The size of each $H(N_i)$ is $O(n_i)$, because the size of a planar Voronoi diagram is linear in the number of sites (Section 2.5). The expected value of n_i is $n/2^i$. It follows that the expected size of $\text{sample}(N)$ is proportional to $\sum_i n/2^i = O(n)$.

To summarize:

Theorem 5.6.9 *The expected size of $\text{sample}(N)$ is $O(n)$. The point location cost is $\tilde{O}(\log^2 n)$.*

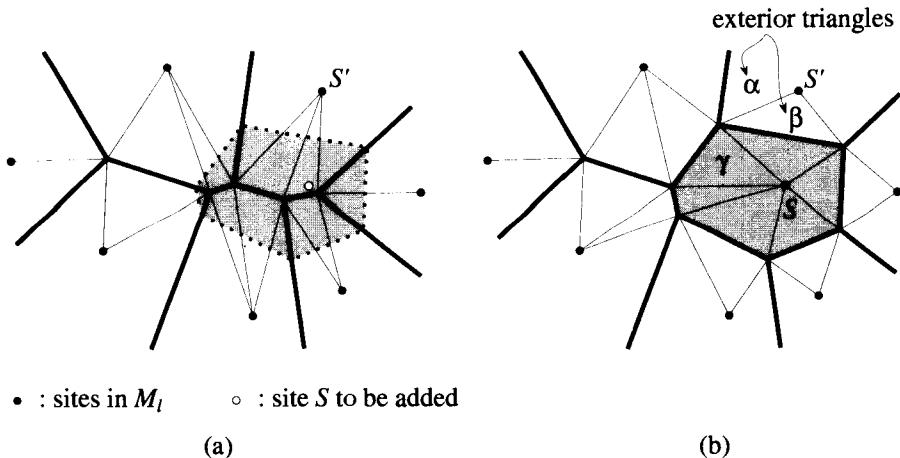


Figure 5.12: Addition of a new site: (a) $H(M_l)$. (b) $H(M'_l)$.

Dynamization

Next, we shall dynamize of the preceding search structure. Let M denote the set of current (undeleted) sites at any given moment. Let $G(M)$ denote the Voronoi diagram of M . Let $H(M)$ denote the radial triangulation of $G(M)$. We shall denote the search structure associated with M by $\text{sample}(M)$. It looks as if it were constructed by applying the previous static method to the set M .

Now let us see how to add a site S to $\text{sample}(M)$. We toss a fair coin repeatedly until we get failure. Let j be the number of successes obtained before getting failure. Our goal is to add S to levels 1 through $j + 1$. For $1 \leq l \leq j + 1$, let M'_l denote $M_l \cup \{S\}$. Addition of S to the l th level means: $H(M_l)$ is updated to $H(M'_l)$ and $\text{descent}(l+1, l)$ is accordingly updated. This is done as follows (Figure 5.12): We first locate the site S using $\text{sample}(M)$. This takes $\tilde{O}(\log^2 m)$ time and tells us the triangle in $H(M_l)$ containing S , for every l . Once we know the triangle in $H(M_l)$ containing S , we can update $H(M_l)$ to $H(M'_l)$, for $1 \leq l \leq j + 1$, in time proportional to the structural change in $H(M_l)$ (Section 3.3). Moreover, the radial trees associated with $H(M_l)$ can also be updated in the same order of time (Section 3.3). The conflict information in the l th level can be updated in time proportional to the total conflict change as we pass from $H(M_l)$ to $H(M'_l)$ (Section 3.3).

Deletion is just the reverse of addition. Suppose we want to delete a site S from $\text{sample}(M)$. We delete S from all the radial triangulations containing it. This can be done in time proportional to the structural change, ignoring a log factor (Section 4.2). The other steps are just the reverse of the analogous steps during addition. We shall not elaborate them any further.

Analysis

The preceding discussion can be summarized as follows.

Lemma 5.6.10 *The total cost of adding or deleting a site in $\text{sample}(M)$ is proportional to*

$$\sum_{\Delta} 1 + l(\Delta), \quad (5.10)$$

where Δ ranges over the newly created and the destroyed triangles in all the affected levels of $\text{sample}(M)$, and $l(\Delta)$ denotes the conflict size of Δ (relative to the previous level). We have excluded here the $\tilde{O}(\log^2 m)$ point location cost during addition. We have also excluded a logarithmic factor in the case of deletion.

We shall now estimate the expected total cost of maintaining $\text{sample}(M)$ over a random (N, δ) -sequence. As usual, we only need to worry about deletions, because additions can be analyzed backwards.

We have already shown that the expected size of $\text{sample}(M)$ is $O(m)$. Hence, it immediately follows from Theorem 5.6.2 that the expected value of the change, as defined in (5.10), is $O(1)$. Well, (5.10) ignores a log factor in the case of deletion. Hence, the expected cost of a random deletion is really $O(\log m)$. (It will be $O(1)$ if one can delete a site from $H(M_l)$ in time proportional to the structural change without incurring an extra log factor. There is a sophisticated deterministic way of doing that, but it is beyond the scope of this book.)

By analyzing an addition backwards, it follows that the expected cost of a random addition is also $O(1)$. This ignores the $\tilde{O}(\log^2 m)$ cost of point location during addition. This additional cost can be brought down to $O(\log m)$. This will be done in the exercises.

To summarize:

Theorem 5.6.11 *The expected cost of a random update is $O(\log m)$. This ignores the $\tilde{O}(\log^2 m)$ cost of point location during addition. (But this, too, can be brought down to $O(\log m)$.)*

Exercises

5.6.3 Show that in the static setting, $\text{sample}(N)$ can be built in expected $O(n \log n)$ time, level by level, from the highest to the lowest level.

5.6.4 Verify that the number of combinatorially distinct search paths in $\text{sample}(N)$ is indeed bounded by a polynomial in n of a fixed degree.

****5.6.5** In the addition of S to $\text{sample}(M)$, as described before, we located S in the radial triangulation $H(M_l)$, for every l . This takes $\tilde{O}(\log^2 m)$ time. An alternative to this point location procedure is the following. It suffices to locate one vertex of $H(M_l)$ in conflict with S . After this, S can be added to $H(M_l)$ as in Section 3.3.

Show how the data structure in this section can be used to locate a vertex of every $H(M_l)$ in conflict with S . The cost should be proportional to the number of triangles in all radial triangulations of $\text{sample}(M)$ in conflict with S . Show that the expected total cost of this conflict search during additions over a random (N, δ) -sequence is $O(n \log n)$.

5.6.6 Build an analogous search structure for point location in planar Delaunay triangulations with the same performance bounds as in this section. (Caution: Point location problems for Voronoi diagrams and Delaunay triangulations are not dual.)

5.6.7 Let N be a set of n half-spaces in R^3 . Let $H(N)$ denote the convex polytope formed by intersecting these half-spaces. Design a static search structure based on bottom-up sampling so that, given any linear function, its optimum on $H(N)$ can be determined in $\tilde{O}(\log^2 n)$ time. The expected space requirement should be $O(n)$. The expected time required to build the search structure should be $O(n \log n)$.

Show that the number of combinatorially distinct search paths in the data structure is bounded by a polynomial in n of fixed degree. Conclude that the maximum query time is $\tilde{O}(\log^2 n)$.

***5.6.8** How would you dynamize the search structure in the last exercise? (Hint: How would you update $H(M_l)$ during deletion?)

5.7 Range spaces and ϵ -nets

One of the observations that was crucial in the proof of Theorem 5.1.2 was the following: If $\Pi(N)$ is a configuration space of bounded valence, then for every subset $R \subseteq N$ of size r , the size of $\Pi(R)$ is $O(r^d)$. Here d is the maximum degree of a configuration in $\Pi(N)$. Recall that $\Pi(R)$ can be a multiset and the size of $\Pi(R)$ is calculated by counting the distinct configurations in $\Pi(R)$ with the same trigger and stopper sets separately. If we do not count the configurations with the same trigger and stopper sets separately, the number that we get is called the *reduced size* of $\Pi(R)$. In other words, the reduced size of $\Pi(R)$ is defined to be the number of possible different trigger and stopper set combinations in $\Pi(R)$. When the configuration space under consideration has a bounded valence, the reduced size and the size differ by only a constant factor. This is because the number of configurations with the same trigger set is bounded. In general, this need not be the case.

It turns out that Theorem 5.1.2 continues to hold if the configuration space $\Pi(N)$ has the following weaker property.

Definition 5.7.1 Let $\Pi(N)$ be a configuration space. For $r \leq n$, let $\tilde{\pi}(r)$ denote the maximum reduced size of $\Pi(R)$, for any subset $R \subseteq N$ of size r . The function $\tilde{\pi}(r)$ is called the *reduced size function* of $\Pi(N)$. We say that $\Pi(N)$ has a *bounded dimension*, if there is a constant d such that

$\tilde{\pi}(r) = O(r^d)$, for all $r \leq n$.⁴ In this case, we say that d is the dimension of $\Pi(N)$.

In this section, by size we shall always mean the reduced size.

If a configuration space $\Pi(N)$ has bounded valence, its dimension is also bounded; indeed, if d is the maximum degree of a configuration in $\Pi(N)$, then the dimension of $\Pi(N)$ is bounded by d (see the proof of Theorem 5.1.2). There are important classes of configuration spaces that do not have bounded valence, but whose dimension is bounded. An important special class is provided by the so-called range spaces. We say that a configuration space $\Pi(N)$ is a *range space*, if the trigger set of every configuration in $\Pi(N)$ is empty. In this case, the configurations in $\Pi(N)$ are also called *ranges*. A range space cannot have bounded valence because all ranges share the empty trigger set. However, a large class of interesting range spaces have bounded dimensions. We provide one example below. More examples will be provided later.

Example 5.7.2 (Half-space ranges)

We have already encountered half-space ranges in Section 2.1. Let N be a set of points in R^d , where the dimension d is a fixed constant. Given a half-space h , the *range* induced by h is defined to be the set of points in N contained in h (Figure 2.2). If two half-spaces contain exactly the same set of points in N , they induce the same range. Let $\Pi(N)$ be the set of all such distinct ranges induced by upper half-spaces. By upper half-spaces, we mean the half-spaces extending in the positive x_d direction. The space of ranges induced by lower half-spaces is similar. If we use the duality transformation D in Section 2.4.1, then we are led to the following dual configuration space. Let \hat{N} denote the set of hyperplanes obtained by transforming the points in N . Given a semi-infinite vertical line \hat{h} in R^d (Figure 2.16), let us define the range induced by \hat{h} to be the set of hyperplanes in \hat{N} intersecting \hat{h} . Let $\Pi(\hat{N})$ be the set of distinct ranges induced by semi-infinite lines extending in the negative x_d -direction. Such half-lines will be called negative half-lines. By Section 2.4.1, $\Pi(N)$ and $\Pi(\hat{N})$ are dual configuration spaces.

Let us show that the dimension of $\Pi(\hat{N})$ is d . Fix a d -cell in the arrangement formed by \hat{N} . The negative half-lines whose endpoints lie within Δ induce the same range. Conversely, negative half-lines whose endpoints are in different d -cells induce different ranges. It follows that the size of $\Pi(\hat{N})$ is equal to the number of d -cells in the arrangement formed by \hat{N} . The latter

⁴We are ignoring here exactly how the constant factor within the Big-Oh notation is allowed to depend on d . If d is a small constant, this does not matter anyway. Otherwise, the cleanest restriction is to stipulate that the constant factor does not depend on d at all. This works for all examples that we will be concerned with (cf. Section 5.7.1). Otherwise, we can allow, say, exponential dependence on d .

number is $O(n^d)$ (Exercise 2.2.3). Similarly, for any subset $R \subseteq N$ of size r , the reduced size $\tilde{\pi}(\hat{R}) = O(r^d)$. It follows that $\Pi(\hat{N})$ has dimension d . As $\Pi(N)$ is a dual configuration space, its dimension is also d .

The following theorem generalizes Theorem 5.1.2 to configuration spaces of bounded dimension. In Theorem 5.1.2, the random sample R was chosen by r independent random draws without replacement. Here we shall find it more convenient to choose R by r independent random draws from N with replacement. In this case, the size of R is not necessarily r . This is because the same object in N can be chosen more than once. This difference is negligible. We leave it to the reader to prove the analogous results for random sampling without replacement.

Theorem 5.7.3 *Let $\Pi(N)$ be any configuration space of bounded dimension d . Let n be the size of N . Let R be a random sample of N formed by $r \leq n$ random independent draws with replacement. With probability greater than $1/2$, for each active configuration $\sigma \in \Pi^0(R)$, the conflict size of σ relative to N is at most $c(n/r) \log r$, where c is a large enough constant. More generally, fix any $c > d$. For any $t \geq r$, with probability $1 - O(1/t^{c-d})$, the conflict size (relative to N) of each configuration in $\Pi^0(R)$ is less than $2c(n/r) \log_2 t$. (The constant factor within the Big-Oh notation depends exponentially on d .)*

This theorem follows from the following result by letting $\epsilon = (2c \log_2 t)/r$.

Theorem 5.7.4 *Let $\Pi(N)$ be a configuration space of bounded dimension d . Fix any $\epsilon \geq 0$ and $r \geq 8/\epsilon$. Let R be a subset of N obtained by r random independent draws from N with replacement. With probability at least $1 - 2\tilde{\pi}(2r)2^{-\epsilon r/2}$, for every configuration $\sigma \in \Pi^0(R)$, the conflict size of σ relative to N is at most ϵn .*

An interesting special case of this theorem arises when $\Pi(N)$ is a range space. In this case, the subset R with the property in the above theorem is called an ϵ -net. Formally, given $\epsilon > 0$, a subset $R \subseteq N$ is called an ϵ -net of the range space $\Pi(N)$, if the conflict size (relative to N) of every range in $\Pi^0(R)$ is at most ϵn . For range spaces, the theorem says that if r is large enough, then a random sample of size r is an ϵ -net with high probability. Roughly, r has to be larger than $(d/\epsilon) \log(1/\epsilon)$ by a sufficiently large constant factor. The exact bound will be derived in the exercises.

Before we turn to the proof of Theorem 5.7.4, let us prove a lemma.

Lemma 5.7.5 *Let ϵ and r be as in Theorem 5.7.4. Fix any configuration $\sigma \in \Pi(N)$ whose conflict size relative to N is greater than or equal to ϵn . Take r random independent draws from N with replacement. With probability greater than or equal to $1/2$, the objects chosen in at least $\epsilon r/2$ draws conflict with σ .*

Proof. Let $l(\sigma)$ denote the conflict size of σ relative to N . Each random draw picks an element in conflict with σ with probability $l(\sigma)/n \geq \epsilon$. For the purpose of bounding the probability, we might as well assume that $l(\sigma)/n = \epsilon$. This assumption can only lead to a more conservative bound. Now r random draws correspond to r Bernoulli trials with probability of success ϵ . The expected number of draws that choose elements in conflict with σ is ϵr . What the lemma is saying is that the actual number of such draws cannot deviate from the expected value by more than its half. This follows by applying Chebychev's inequality (Appendix A): The variance of the Bernoulli trials in the current setting is $\epsilon(1 - \epsilon)r$. Hence, the probability of picking fewer than $\epsilon r/2$ conflicting elements is bounded by

$$\frac{\epsilon(1 - \epsilon)r}{(\epsilon r/2)^2} \leq \frac{4}{\epsilon r} \leq \frac{1}{2},$$

because $r \geq 8/\epsilon$. □

Proof of Theorem 5.7.4: Fix an $\epsilon > 0$ and an integer $r > 8/\epsilon$. Take $2r$ random independent draws from N with replacement. Let S be the set of elements chosen in these draws. Permute the draws randomly. Let R be the set of elements chosen in the first r draws as per this permutation. Because of the random nature of the permutation, we can think that R is obtained by r independent draws from N . Let $p(r)$ denote the probability that there exists a configuration $\sigma \in \Pi(N)$ which is active over R and whose conflict size relative to N is at least ϵn . Saying that $\sigma \in \Pi(N)$ is active over R is the same as saying that σ , or more precisely its restriction, belongs to $\Pi^0(R)$. We shall show that

$$p(r) \leq 2\tilde{\pi}(2r)2^{-\epsilon r/2}. \quad (5.11)$$

This immediately implies the theorem.

Let $q(r)$ denote the probability that there exists a configuration $\sigma \in \Pi(N)$ such that

1. σ is active over R and the conflict size of σ relative to N is greater than or equal to ϵn , and
2. at least $\epsilon r/2$ of the later r draws in the permutation pick elements in conflict with σ .

Then $p(r)/2 \leq q(r)$. This is because a σ satisfying the first condition exists with probability $p(r)$ and, moreover, for any such σ , the probability that it satisfies the second condition is at least $1/2$ (Lemma 5.7.5).

We shall show that

$$q(r) \leq \tilde{\pi}(2r)2^{-\epsilon r/2}. \quad (5.12)$$

Since, $p(r) \leq 2q(r)$, (5.11) follows immediately.

Let S be the set of elements chosen in the above $2r$ draws. If $\sigma \in \Pi(N)$ satisfies the preceding two conditions, then its restriction $\bar{\sigma} = \sigma \downarrow S \in \Pi(S)$ (which is well defined because the configuration is active over $R \subseteq S$) trivially satisfies the following two conditions:

1. $\bar{\sigma}$ is active over R : In other words, none of the first r draws in the above random permutation picks an element in conflict with $\bar{\sigma}$, and all of its triggers are contained in R ;
2. at least $\epsilon r/2$ of the later r draws in the permutation pick elements in conflict with $\bar{\sigma}$.

Hence, $q(r)$ is bounded by the probability that there exists a configuration $\bar{\sigma} \in \Pi(S)$ satisfying these two conditions. Fix a configuration $\bar{\sigma} \in \Pi(S)$. If $l(\bar{\sigma})$ is less than $\epsilon r/2$, it cannot satisfy these two conditions. Otherwise, each of the $l = l(\bar{\sigma})$ conflicting elements in S has to be drawn in the later r draws. (Let us assume that S does not contain multiple copies of any of these elements; otherwise, the probability of this event will only get smaller.) As the permutation of the $2r$ draws is chosen randomly, this happens with probability roughly 2^{-l} , or more precisely, with probability

$$\binom{r}{l} / \binom{2r}{l} = \frac{r}{2r} \frac{r-1}{2r-1} \cdots \frac{r-l+1}{2r-l+1} \leq 2^{-l} \leq 2^{-\epsilon r/2}.$$

Thus, the probability that a fixed $\bar{\sigma} \in \Pi(S)$ satisfies the above two conditions is bounded by $2^{-\epsilon r/2}$. The number of configurations in $\Pi(S)$ with distinct trigger and stopper sets is bounded by $\tilde{\pi}(s)$, where s is the size of S . Clearly, $s \leq 2r$. Hence,

$$q(r) \leq \tilde{\pi}(s) 2^{-\epsilon r/2} \leq \tilde{\pi}(2r) 2^{-\epsilon r/2}.$$

This proves (5.12). □

The following observation is frequently useful in bounding the dimension of a range space.

Observation 5.7.6

1. Let $\Pi_1(N)$ and $\Pi_2(N)$ be two range spaces of bounded dimensions d_1 and d_2 , respectively. Then the range space $\Pi(N)$, which is obtained by taking pairwise intersections or unions of the ranges in $\Pi_1(N)$ and $\Pi_2(N)$, has dimension at most $d_1 + d_2$. Similarly, the range space $\Pi_1(N) \cup \Pi_2(N)$ has dimension at most $\max\{d_1, d_2\}$.
2. Let $\Pi(N)$ be a range space of bounded dimension d . Then the range space $\Pi'(N)$ obtained by taking the complements of the ranges in $\Pi(N)$ also has dimension d .

We give below some additional examples of range spaces with bounded dimension. As usual, the objects are assumed to be in general position.

Example 5.7.7 (Simplex ranges)

We have already encountered simplex ranges in Section 2.1. Let N be a set of points in R^d . Given any simplex Δ , possibly unbounded, define the range induced by Δ to be the set of points in N lying within Δ . Let $\Pi(N)$ be a configuration space of distinct simplex ranges over N . Each simplex is obtained by intersecting a bounded number of half-spaces. We have already seen in the beginning of this section that the configuration space of ranges induced by half-spaces has bounded dimension. It follows from Observation 5.7.6 that the configuration space of simplex ranges also has bounded dimension.

Example 5.7.8 (Linear ranges)

Let N be a set of hyperplanes in R^d . Given a linear segment l in R^d , possibly unbounded, define the range induced by l to be the set of hyperplanes in N intersecting l . For the sake of simplicity, we always assume that these intersection are proper. This means we do not count the hyperplanes containing l or intersecting l at its endpoints, if any. For example, the range induced by the segment (s_0, s_1) in Figure 5.13 is $\{h_5, h_4, h_6, h_2\}$.

Let $\Pi(N)$ be the configuration space of distinct linear ranges. We shall show that its dimension is bounded by $2d$. Let $R \subseteq N$ be any subset of size r . Fix any two d -cells Δ and Γ in the arrangement formed by R . Figure 5.13 gives an illustration for $R = N$. Note that all segments, with their endpoints in Δ and Γ , induce the same linear range; this range consists of the hyperplanes in N that separate the interiors of Δ and Γ . The number of d -cells in the arrangement formed by R is $O(r^d)$. This immediately implies that the reduced size of $\Pi(R)$ is $O(r^{2d})$. Hence, the dimension of $\Pi(N)$ is bounded by $2d$.

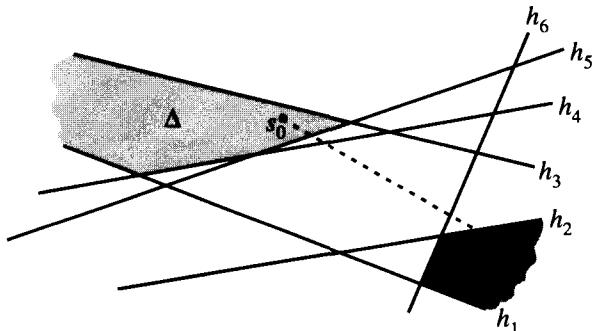


Figure 5.13: Linear ranges.

Exercises

5.7.1 What is the analogue of Observation 5.7.6 for general configuration spaces of bounded dimension?

5.7.2 Let N be a set of lines in the plane. Let R be a random sample of N of size r . Let $H(R)$ denote the trapezoidal decomposition formed by R . We have already seen that the conflict size of any trapezoid in $H(R)$ is $O([n/r] \log r)$, with high probability (Section 5.1). Show that this also follows from Theorem 5.7.3 because the configuration space of linear ranges over N has bounded dimension. (Hint: The linear ranges induced by the edges of any trapezoid σ in $H(R)$ are active over R . Any line conflicting with σ must intersect one of its bounding edges.)

5.7.3 Prove the analogue of Theorem 5.7.3 for Bernoulli sampling and for sampling without replacement.

5.7.1 VC dimension of a range space

In all previous examples of range spaces, we could bound their dimension by a direct argument. For several of the range spaces that arise in computational geometry, such direct arguments work. In other cases, the following criterion can also be used.

Let $\Pi(N)$ be any range space. Let us say that a subset $M \subseteq N$ is *shattered* by N if every subset of M occurs as a range in $\Pi(M)$. This is equivalent to saying that the reduced size of $\Pi(M)$ is 2^m . Here m is the size of M . We define the *VC dimension* (Vapnik–Chervonenkis dimension) of the range space $\Pi(N)$ as the maximum size d of a shattered subset of N . Since it requires 2^d distinct ranges in $\Pi(N)$ to shatter such a subset, clearly, 2^d is bounded by the reduced size of $\Pi(N)$. In other words, $d \leq \lfloor \log_2 \tilde{\pi}(N) \rfloor$.

Consider a simple example. Let N be a set of points on the line. Let $\Pi(N)$ be the space of ranges induced by half-lines (half-spaces) extending in the right direction. It is easy to see that the VC-dimension of $\Pi(N)$ is one: A singleton set can be easily shattered. A set $\{p, q\}$ of points, where $p < q$, cannot be shattered, because no half-space extending in the right direction can induce the singleton set $\{p\}$ as its range.

In general, let N be a set of points in R^d in general position. Let $\Pi(N)$ be the space of ranges induced by upper half-spaces (Example 5.7.2). We claim that the VC-dimension of $\Pi(N)$ is d . Let \hat{N} be the set of dual hyperplanes. Let $\Pi(\hat{N})$ be the dual configuration space as in Example 5.7.2. We have already seen (Example 5.7.2) that the size of $\Pi(\hat{N})$ is equal to the number of d -cells in the arrangement formed by \hat{N} . The number of these d -cells is at most $\phi_d(n)$ (Exercise 2.2.3), where $\phi_d(n)$ is a function defined as follows:

$$\phi_0(n) = 1, \phi_d(0) = 1, \text{ and } \phi_d(r) = \phi_d(r-1) + \phi_{d-1}(r-1), \text{ for } d, r \geq 1.$$

We leave it to the reader to check that

$$\phi_d(r) = \sum_{j=0}^d \binom{r}{j}, \text{ if } d < r. \text{ Otherwise, } \phi_d(r) = 2^r. \quad (5.13)$$

This means, if R is a subset of N of size d , then the reduced size of $\Pi(R)$ is $\phi_d(d) = 2^d$. Hence, R is shattered. If the size of R is greater than d , then the reduced size of $\Pi(R)$ is $\phi_d(r) < 2^r$. Hence, R cannot be shattered. It follows that the VC-dimension of $\Pi(N)$ is indeed d , as claimed.

The reduced size function $\tilde{\pi}(r)$ for the above range space is $\phi_d(r)$. In general, it turns out that $\phi_d(r)$ bounds the reduced size function of any range space of VC-dimension d :

Theorem 5.7.9 *Let $\Pi(N)$ be any range space of VC-dimension d . Let n be the size of N . For $d \leq r \leq n$, $\tilde{\pi}(r) \leq \phi_d(r)$. In particular, the dimension of $\Pi(N)$, as defined in the beginning of Section 5.7, is less than or equal to its VC-dimension.*

Proof. Without loss of generality, we can assume that $r = n$. This is because, for any subset $R \subseteq N$, the VC-dimension of the subspace $\Pi(R)$ is bounded by d .

We use double induction on d and n . The assertion is trivially true for $d = 0$ and $n = 0$. Assume that the assertion holds for any range space whose dimension is at most $d - 1$ and for any range space of dimension d with at most $n - 1$ elements in the object set.

Fix an object $S \in N$. Consider the subspace $\Pi(N \setminus \{S\})$. Clearly, its VC-dimension is at most d . By our inductive hypothesis, its reduced size is bounded by $\phi_d(n - 1)$.

Let $\Sigma(N \setminus \{S\})$ be the space of ranges $\sigma \in \Pi(N)$, not containing S , such that $\sigma \cup \{S\}$ is also a range in $\Pi(N)$. Its VC-dimension is at most $d - 1$: If σ is a shattered range in $\Sigma(N \setminus \{S\})$ then $\sigma \cup \{S\}$ is easily seen to be a shattered range in $\Pi(N)$. Since the VC-dimension of $\Pi(N)$ is d , the size of $\sigma \cup \{S\}$ is at most d . Hence, the size of σ is at most $d - 1$.

By the induction hypothesis, the reduced size of $\Sigma(N \setminus \{S\})$ is bounded by $\phi_{d-1}(n - 1)$.

The reduced size of $\Pi(N)$ is equal to the sum of the reduced sizes of $\Pi(N \setminus \{S\})$ and $\Sigma(N \setminus \{S\})$: Two distinct ranges $\sigma^1, \sigma^2 \in \Pi(N)$ can restrict to the same range in $\Pi(N \setminus \{S\})$ iff $\sigma^1 \cup \{S\} = \sigma^2$ (or the other way round), which means $\sigma^1 \in \Sigma(N \setminus \{S\})$. It follows that the reduced size of $\Pi(N)$ is bounded by $\phi_d(n - 1) + \phi_{d-1}(n - 1) = \phi_d(n)$. \square

Exercises

5.7.4 Let $\Pi(N)$ be a range space of VC-dimension d . Fix any $\epsilon > 0$ and $\delta < 1$. Let R be a subset of N obtained by r random independent draws with replacement. Assume that r is greater than $(4/\epsilon) \log_2(2/\delta)$ and $(8d/\epsilon) \log_2(8d/\epsilon)$. Show that R is an ϵ -net with probability at least $1 - \delta$. (Hint: Use Theorems 5.7.4 and 5.7.9.)

5.7.5 Let N be a set of points in R^d , where d is bounded. Show that the space of ranges induced by the spherical balls in R^d has a bounded VC-dimension. Calculate the VC-dimension explicitly in terms of d .

****5.7.6** Show that the space of ranges induced by half-spaces, whose boundaries are algebraic hypersurfaces of bounded degree, has a bounded VC-dimension.

5.7.7 Prove the analogue of Observation 5.7.6 for the VC-dimension.

5.8 Comparisons

We have developed several algorithmic methods in the first half of this book. Before proceeding to the higher-dimensional applications in the second half, we wish to pause in order to compare the various methods.

We have seen two forms of random sampling: top-down (Section 5.2) and bottom-up (Section 5.3). Bottom-up sampling is more efficient; for example, compare Theorem 5.3.3 and Theorem 5.2.1. On the other hand, it may not be applicable in every situation where top-down sampling is applicable. This could happen if the descent structures required to make the bottom-up sampling work are difficult to construct in a given problem. Range queries provide one such example (Section 6.5). There are also situations wherein bottom-up sampling is easier to apply than top-down sampling. Ray shooting in arrangements provides one such example (Section 6.4).

The basic idea behind dynamic sampling (Section 5.4) is simple: The search structure at any time should look as if it were constructed by applying the static random sampling method to the set of objects existing at that time. As such, this dynamization principle is equally applicable to the search structures based on top-down as well as bottom-up sampling; for dynamization of top-down sampling, see, for example, Section 6.5.1. It is illuminating to see how the dynamic sampling technique compares with the previous dynamization techniques based on history (Chapter 4). The search structures based on random sampling and the ones based on history look significantly different: Random sampling breaks down the given problem, in a *parallel* manner, into several smaller problems. As such, the search structures based on dynamic sampling can be maintained efficiently on parallel computers (we shall not deal with parallel computing in this book). In contrast, history is inherently *sequential* because it deals with one object at a time. Each link

structure in history records the changes that occur during the update involving a single object. In contrast, a descent structure required by dynamic sampling has to keep track of the global changes from one level to another, so it needs to be somewhat more sophisticated than a link structure. This means, in the semidynamic setting, where no deletions are involved, maintaining history could be easier. The situation changes considerably when deletions are involved. The deletion of an object from a history (Sections 4.5 and 4.6) is considerably more difficult than the deletion from a search structure based on dynamic sampling (Section 5.4). Of course, one could be lazy about deletions in history (Section 4.4), but laziness may not always work. For example, it does not work for three-dimensional convex polytopes (Exercise 4.2.2). Laziness has other disadvantages, too: It cannot guarantee good query costs during every update sequence (see the beginning of Section 4.6).

As a concrete example, consider the problem of dynamic planar point location. We have given several search structures for the more general point location problem for trapezoidal decompositions (in the planar point location problem, the segments do not intersect). For this problem, the semidynamic search structure based on history (Section 3.1.1) is a bit simpler than the one based on dynamic sampling (Section 5.6.1). The dynamic search structure based on lazy maintenance of history with periodic rebuildings (Sections 4.1 and 4.4) is comparable in complexity to the one based on dynamic sampling. The one based on history does not, however, guarantee good query costs during every update sequence. Thus, in the dynamic setting, the search structure based on dynamic sampling seems preferable. Later, we shall give yet another search structure based on randomized segment trees (Section 8.3). It has an additional advantage in that it guarantees a good bound on the cost of each update, whereas the search structure based on dynamic sampling guarantees good cost for only random updates. The search structure based on randomized segment trees can be made deterministic if one uses weight-balanced trees instead of skip lists in its construction. This, however, complicates the search structure considerably, especially in the dynamic setting. This search structure also assumes that the underlying planar graph remains connected throughout all updates. In addition, it is somewhat more complex, even in the randomized form, than the one based on dynamic sampling.

We said above that the planar point location structure based on dynamic sampling (Section 5.6.1) guarantees good costs for only random updates. What we had in mind here was a *provable* guarantee. It is very plausible that this search structure, in fact, works very well on almost all update sequences. This would require proving a high-probability bound in the model of randomness given in the beginning of Chapter 4. Once this is done, one could assume, for all practical purposes, that this search structure would

work as well as any search structure with guaranteed good performance on every update sequence. We have proven high-probability bounds on the query times of all search structures we have given so far. We have also proven high-probability bounds on the update costs in a few instances (Sections 5.4.1 and 3.5). But proving general high-probability bounds on the update costs in a fully dynamic setting remains one challenging open problem in this area.

Bibliographic notes

Two seminal works on the randomized divide-and-conquer paradigm are [66], where Clarkson proves the random sampling result (Theorem 5.1.2) for configuration spaces of bounded valence (our terminology), and [119], where Haussler and Welzl proved the random sampling result for ranges spaces of bounded VC dimension (Section 5.7). The framework of configurations spaces of bounded dimension that we used in this book allows us to unify these two approaches in a very simple way. For another unifying framework, see Chazelle and Friedman [48]. Haussler and Welzl [119] introduced the notion of ϵ -nets, inspired by the notion of ϵ -approximation and VC dimension due to Vapnik and Chervonenkis [220]. They proved the existence of an ϵ -net of at most $(8d/\epsilon) \log_2(8d/\epsilon)$ size (Theorem 5.7.4 and Exercise 5.7.4). Further improvements were carried out by Blumer et al. [24]. The logarithmic factor in the above bound is not a product of the proof technique: Komlós, Pach and Wöginger [131] show that the best bound is $\Omega([1/\epsilon] \log[1/\epsilon])$; for the further study of this question, see [143]. The range spaces of bounded dimension have found applications in other fields as well, such as learning theory [24] and discrepancy theory [145]. For more on ϵ -nets and ϵ -approximations, see Chapter 10, where they are studied in more detail. Exercise 5.7.5 is based on [223].

Theorem 5.0.1, the precursor of the random sampling result Theorem 5.1.2, was proved by Reischuck [194] in the context of fast parallel sorting. This work was later extended to several parallel algorithms in computation geometry by Reif and Sen [191]. For more on parallel computational geometry, see, for example, [4, 15, 69, 193]. For parallelization of the dynamic sampling technique, see Mulmuley and Sen [172].

The point location structure for arrangements based on top-down sampling (Section 5.2) is due to Clarkson [66]. The dynamic sampling technique (Section 5.4), along with the bottom-up sampling technique (Section 5.3), was proposed by Mulmuley and Sen [172], where they gave the point location structure for arrangements described in Sections 5.3 and 5.4. The dynamic sampling technique was extended further by Mulmuley [166, 167]; the dynamic point location structures for Voronoi diagrams and trapezoidal decompositions (Section 5.6) are from [166]. The results on average conflict size (Section 5.5) are due to Clarkson and Shor [71], with some extensions by Mulmuley [166]. Random sampling has several combinatorial applications besides the algorithmic applications studied in this book; for example, see [65].

For more applications and references, see the following chapters, where the techniques in this chapter are applied to specific problems.

Part II

Applications

Chapter 6

Arrangements of hyperplanes

We have studied arrangements of lines extensively in the previous chapters. In this chapter, we shall study higher-dimensional arrangements. These are formed by hyperplanes in R^d . Let N be a set of hyperplanes in R^d . Each hyperplane is defined as the solution set of a linear equation of the form $a_1 x_1 + \cdots + a_d x_d = a_0$, where each a_i is a real constant, and a_1, \dots, a_d are not all zero. The arrangement $G(N)$ formed by N is defined to be the natural partition of R^d into convex regions (faces) of varying dimensions along with the adjacencies among them. A formal definition is as follows. The hyperplanes in N divide R^d into several d -dimensional convex regions. These convex regions are called the *d-cells* or *d-faces* of $G(N)$. More generally, let P_1, P_2, \dots be any distinct hyperplanes in N . Their intersection $P_1 \cap P_2 \cap \dots$, if nonempty, is an affine space, which is similarly partitioned by the remaining hyperplanes in N . Let j be the dimension of this affine space. The j -dimensional regions of the partition within this space are called the *j-faces* (or *j-cells*) of $G(N)$. Thus, each j -face of $G(N)$ is a convex region (polytope) of dimension j . The 0-faces, 1-faces, $(d - 1)$ -faces, and d -faces of $G(N)$ are also called *vertices*, *edges*, *facets*, and *cells*, respectively.

For $i < j$, an i -face f is said to be *adjacent* to a j -face g if f is contained in the boundary of g . In this case, we also say that f is a *subface* of g . The total number of subfaces of g is called its *size*. It is denoted by $|g|$. Getting a good estimate on $|g|$, when the dimension of g is higher than three, is a nontrivial problem. It will be addressed in Chapter 7.

One natural way to represent an arrangement is in terms of its *facial lattice* (Section 2.2). This contains a node for each face of the arrangement. Two nodes are joined by an adjacency edge iff the corresponding faces are adjacent

and their dimensions differ by one. Figure 2.5 in Chapter 2 shows the facial lattice of an arrangement of lines. It is convenient to group the adjacency edges incident to every fixed node in the facial lattice in two classes: (1) those which correspond to adjacencies with the faces of higher dimensions, and (2) those which correspond to adjacencies with the faces of lower dimensions. We shall soon see that if the hyperplanes in N are in general position, then the cardinality of the first class is $O(1)$, for every fixed dimension d . This means, for any face f , we can determine the higher-dimensional faces adjacent to f in $O(1)$ time. The lower-dimensional faces adjacent to f can be determined in $O(|f|)$ time. In algorithmic applications, it is also convenient to associate with each node some auxiliary information: the list of hyperplanes containing the corresponding face, along with their equations; the coordinates, in case the node corresponds to a vertex; and so on.

For $j \leq d$, we define the j -skeleton of $G(N)$ as the collection of i -faces of $G(N)$, for all $i \leq j$, together with the adjacencies among them. Thus, the j -skeleton can be thought of as a sublattice of the facial lattice of $G(N)$. An important special case is the 1-skeleton of $G(N)$. It is also called the *edge skeleton*.

An arrangement $G(N)$ is called *simple* or *nondegenerate* if the intersection of any $j > 0$ hyperplanes in N is $(d - j)$ -dimensional; by convention, a negative-dimensional set is supposed to be empty. When $d = 2$, the above definition is equivalent to saying that no two lines in N are parallel and that no vertex in $G(N)$ is contained in more than two lines.

Proposition 6.0.1 *If $G(N)$ is simple, every vertex v of $G(N)$ is adjacent to exactly $2^i \binom{d}{i}$ i -faces, where $d \geq i \geq 0$.*

Proof. We distinguish between two cases:

(1) Full dimensional case ($i = d$): Because $G(N)$ is simple, exactly d hyperplanes contain the given vertex v . The d -cells of $G(N)$ adjacent to v are in

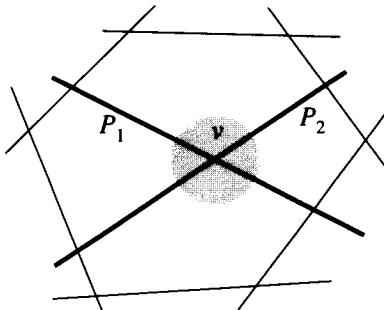


Figure 6.1: Cones.

one-to-one correspondence with the 2^d cones formed by these hyperplanes at v (Figure 6.1).

(2) $i < d$: Fix any set $\{P_1, \dots, P_{d-i}\}$ of $d-i$ hyperplanes through v . Consider the i -dimensional restricted arrangement $G(N) \cap P_1 \cap \dots \cap P_{d-i}$. Apply the above full dimensional case to this restricted arrangement. It follows that the number of i -cells of $G(N)$, which are adjacent to v and which are contained in the hyperplanes P_1, \dots, P_{d-i} , is 2^i . By simplicity of $G(N)$, every i -cell is contained in a unique set of $(d-i)$ hyperplanes in N . Hence, the total number of i -faces adjacent to f is $\binom{d}{d-i} 2^i = \binom{d}{i} 2^i$. \square

Let $|G(N)|$ denote the size of $G(N)$. By this, we mean the total number of its faces of all dimensions.

Corollary 6.0.2 For a fixed d , $|G(N)| = O(n^d)$, where n is the size of N .

Proof. We can assume, without loss of generality, that $G(N)$ is simple. This is because almost all infinitesimal perturbations make $G(N)$ simple, and such a perturbation does not increase the size (Exercise 6.0.3).

Assuming that $G(N)$ is simple, the number of vertices of $G(N)$ is $\binom{n}{d}$. This is because each vertex of $G(N)$ is determined by a subset of exactly d hyperplanes in N , and conversely each subset of d hyperplanes in N determines a vertex of $G(N)$. By Proposition 6.0.1, each vertex of $G(N)$ is adjacent to $2^i \binom{d}{i}$ i -faces. Hence, the number of i -faces of $G(N)$ is obviously bounded by $2^i \binom{d}{i} \binom{n}{d}$. This is not a precise bound because an i -face can be adjacent to more than one vertex of $G(N)$. In that case, it is counted more than once. We shall obtain a precise bound in the following exercise. \square

Exercises

6.0.1 If $G(N)$ is simple, show that every j -face f of $G(N)$ is adjacent to exactly $2^{i-j} \binom{d-j}{i-j}$ i -faces, for $d \geq i \geq j$. (Hint: Choose j hyperplanes $Q_1, \dots, Q_j \notin N$ such that $g = f \cap Q_1 \cap \dots \cap Q_j$ is a point, and examine the restricted arrangement $G(N) \cap Q_1 \cap \dots \cap Q_j$ at v .)

6.0.2 Assume that $G(N)$ is simple. Consider a fixed d -cell g in $G(N)$. Show that every j -subface f of g is adjacent to exactly $\binom{d-j}{i-j}$ i -subfaces, for $d \geq i \geq j$.

6.0.3 Let $G(N)$ be an arrangement of hyperplanes. Suppose $G(N)$ is not simple. Show that it becomes simple under almost all infinitesimal perturbations of the hyperplanes in N . (Hint: Exercise 2.2.1.) Show that the number of i -faces in $G(N)$, for any i , increases for almost all infinitesimal perturbations of the hyperplanes in N .

6.0.4 Let $n(i)$ denote the number of i -faces in an arrangement $G(N)$ formed by n hyperplanes. Prove the following precise bound on $n(i)$:

$$n(i) \leq \sum_{j=0}^i \binom{d-j}{i-j} \binom{n}{d-j},$$

with equality iff the arrangement is simple. (Hint: Use double induction on n and d .)

6.0.5 Show that the edge skeleton of a simple arrangement is a connected graph. (Hint: Use induction on the dimension.)

6.0.6 Given a j -skeleton of $G(N)$, for $j > 0$, show how one can determine the $(j + 1)$ -skeleton in time proportional to its size. Assume that each node in the j -skeleton is associated with the set of hyperplanes containing the corresponding face.

6.1 Incremental construction

Let N be a set of n hyperplanes. Let $G(N)$ be the arrangement formed by N . In this section, we describe a simple incremental construction of the facial lattice of $G(N)$. The cost of this construction is $O(n^d)$. This is optimal by Corollary 6.0.2. Here, as well as in the rest of this book, the dimension d is assumed to be fixed. The constant in the Big-Oh notation depends exponentially on d .

The algorithm is a simple generalization of the algorithm in Section 2.8.2 for constructing an arrangement of lines. We assume that the hyperplanes in N are in general position. The assumption is easy to remove, and this will be done in the exercises. We add the hyperplanes in N one at a time in any order, not necessarily random. As in Section 2.8.2, it turns out that an arrangement of hyperplanes is special enough so that any order works. Let N^i denote the set of the first i added hyperplanes. Let $G(N^i)$ denote the arrangement formed by them. Starting with the empty arrangement $G(N^0)$, we form a sequence of arrangements

$$G(N^0), G(N^1), \dots, G(N^{n-1}), G(N^n).$$

The final arrangement $G(N^n) = G(N)$ is the sought one. Addition of the $(i + 1)$ th hyperplane $S = S_{i+1}$ to $G(N^i)$ is achieved as follows.

Pick a line L in $G(N^i)$ formed by the intersection of any $d - 1$ hyperplanes in N^i . Let u be the intersection of S with L (Figure 2.23 illustrates a two-dimensional case). First, we locate the edge e of $G(N^i)$ containing u . This is done by linearly searching through all edges of $G(N^i)$ within L , from its one end to another. We go from one edge to the next using adjacency relationships. The number of edges of $G(N^i)$ within L is obviously bounded by i . Hence, this search takes $O(i)$ time.

At this stage, we know one edge of $G(N^i)$ intersecting S , namely the edge e containing u . Our next goal is to determine all j -faces of $G(N^i)$ intersecting S , for every $j \geq 1$. By our general position assumption, no vertex of $G(N^i)$

is contained in S . Let us first determine the edges and 2-faces of $G(N^i)$ intersecting S . This is accomplished by the following search algorithm.

Algorithm 6.1.1 (Search)

Initialize a set Φ by letting $\Phi = \{e\}$.

Until Φ is empty, do:

1. Remove any edge $c \in \Phi$. Mark c as well as all 2-faces of $G(N^i)$ adjacent to c .
2. For each 2-face f adjacent to c , examine all edges of f . If an edge b of f , other than c , intersects S , add b to Φ , if it is not already marked; see Figure 6.2.

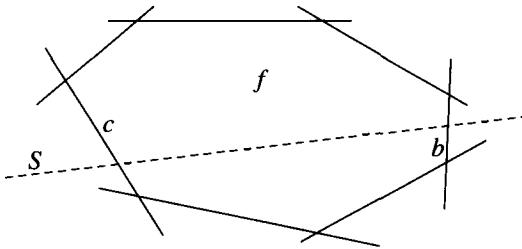


Figure 6.2: Face traversal.

We claim that this search algorithm eventually marks all edges and 2-faces of $G(N^i)$ intersecting S . The reason is simple. These edges and 2-faces are obviously in one-to-one correspondence with the vertices and edges of the arrangement $G(N^i) \cap S$; this latter arrangement is imaginary at this stage. Whenever the algorithm marks an edge c (or a 2-face f) in $G(N^i)$, we imagine marking the corresponding vertex $c \cap S$ (or the edge $f \cap S$) in $G(N^i) \cap S$. It is easily seen that the above search algorithm corresponds to traversing the edge skeleton of the imaginary arrangement $G(N^i) \cap S$. But this edge skeleton is connected (Exercise 6.0.5). Hence, all its vertices and edges will be eventually marked during this (imaginary) traversal on the edge skeleton.

The second step in the search algorithm examines $|f|$ edges of a 2-face f . Hence, it is easy to see that the total cost of the search is $O(\sum_f |f|)$, where f ranges over all 2-faces of $G(N^i)$ intersected by S .

We now know all edges, and also 2-faces, of $G(N^i)$ intersecting S . The j -faces of $G(N^i)$ intersecting S , for any $j > 2$, are simply the ones that are adjacent to these edges. Note that each edge is adjacent to $O(1)$ j -faces, for any $j > 1$ (Proposition 6.0.1).

Thus, we have determined all faces of $G(N^i)$ intersecting S . Now it is easy to update the facial lattice. For each j -face $f \in G(N^i)$ intersecting S ,

we replace the node for f in the facial lattice with two nodes for the j -faces f_1 and f_2 that f is split into. We also create a new node for the $(j-1)$ -face $g = f \cap S$. The adjacency relationships are also easy to update: Both f_1 and f_2 are adjacent to g . If h is a subface of f that is not split by S , then h is adjacent to either f_1 or f_2 —which case holds is easy to determine. If h is split by S , then each of its split parts is adjacent to either f_1 or f_2 —which case holds is again easy to determine. Moreover, $h \cap S$ is adjacent to $g = f \cap S$.

It is clear that the update of the facial lattice takes $O(\sum_f |f|)$ time, where f now ranges over all j -faces of $G(N^i)$ intersecting S .

Thus, $G(N^i)$ can be updated to $G(N^{i+1})$ in $O(i + \sum_f |f|)$ time, where f ranges over all j -faces of $G(N^i)$ intersecting S . The $O(i)$ term reflects the cost of locating the initial vertex u of $G(N^i) \cap S$. The Zone Theorem, to be proved in the next section, implies that the latter term $\sum_f |f|$ is $O(i^{d-1})$. This means that the cost of updating $G(N^i)$ to $G(N^{i+1})$ is $O(i^{d-1})$. It follows that the cost of constructing $G(N) = G(N^n)$ is $O(\sum_i i^{d-1}) = O(n^d)$. Thus:

Theorem 6.1.2 *An arrangement of n hyperplanes in R^d can be constructed in $O(n^d)$ time.*

It only remains to prove the required Zone Theorem.

6.2 Zone Theorem

In this section, we generalize the Zone Theorem for arrangements of lines (Theorem 2.8.2) to arbitrary dimension.

Let M be any set of m hyperplanes in R^d . For any hyperplane $S \notin M$, let $\text{zone}(M, S)$ denote the *zone* of S in $G(M)$. It is defined to be the set of d -faces in $G(M)$ intersecting S together with their subfaces. More precisely, it is the set of pairs of the form (f, c) , where c ranges over the d -cells of $G(M)$ intersecting S and, for any such c , f ranges over all its subfaces. A pair (f, c) is to be thought of as a *border* of the cell c that arises from f . Its dimension is defined to be the dimension of f . Its *codimension* is defined to be d minus its dimension. Each j -face of $G(M)$ gives rise to 2^{d-j} distinct borders. These correspond to the 2^{d-j} d -cells adjacent to it (Exercise 6.0.1). $\text{Zone}(M, S)$ can contain some, none, or all of these 2^{d-j} borders. For example, in Figure 6.3, the edge f gives rise to two borders, $\sigma_1 = (f, c)$ and $\sigma_2 = (f, d)$, only one of which, namely, σ_1 , belongs to the zone of S . The edge g gives rise to two borders τ_1 and τ_2 , both of which belong to the zone of S .

We define $\text{zone}_k(M, S) \subseteq \text{zone}(M, S)$ to be the subset of the borders with codimension k . We let $z_k(M, S)$ denote the size of $\text{zone}_k(M, S)$. We do not mention S when it is understood from the context. Thus, $\text{zone}_k(M, S)$ and $z_k(M, S)$ will be denoted by simply $\text{zone}_k(M)$ and $z_k(M)$, respectively.

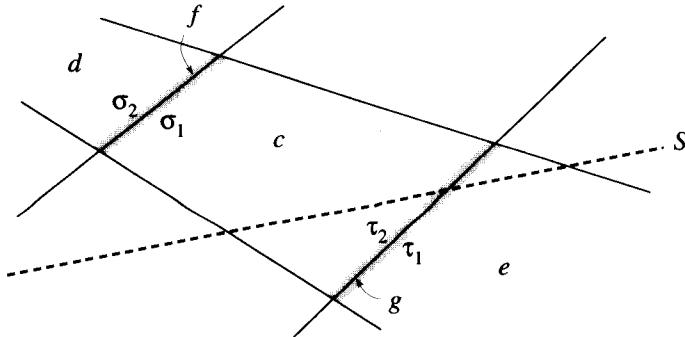


Figure 6.3: Zone of \$S\$.

Let $z_k(m, d)$ denote the maximum value of $z_k(M, S)$ over all possible choices of M and S in R^d , where the size of M is m . We can assume, without loss of generality, that the hyperplanes in $M \cup \{S\}$ are in general position, because the size of the zone increases for almost all infinitesimal perturbations (see Exercise 6.2.1). This assumption will be implicit in what follows.

Theorem 6.2.1 (Zone Theorem) For a fixed d and for each $0 \leq k \leq d$, $z_k(m, d) = O(m^{d-1})$.

Proof. In what follows, M will denote a fixed set of m hyperplanes in R^d , and $S \notin M$ will denote a fixed hyperplane, whose zone we are interested in.

Fix any hyperplane $Q \in M$. Let $\text{zone}(M_Q)$ be the zone of $S \cap Q$ in the $(d-1)$ -dimensional arrangement $G(M) \cap Q$. We define $\text{zone}_k(M_Q)$ similarly. Let $z_k(M_Q)$ be the size of $\text{zone}_k(M_Q)$. Finally, let $z_k(M \setminus \{Q\}, S)$ be the size of $\text{zone}_k(M \setminus \{Q\}, S)$; this is the zone of S with respect to the hyperplanes in M other than Q . We claim that, for $k < d$ and $d > 1$,

$$\sum_{\zeta \not\subseteq Q} 1 \leq z_k(M \setminus \{Q\}) + z_k(M_Q), \quad (6.1)$$

where ζ ranges over all borders in $\text{zone}_k(M)$ not contained in Q ; the left hand side is just the number of borders in $\text{zone}_k(M)$ not contained in Q . Every such border ζ coincides with or is contained in a unique border $\sigma \in \text{zone}_k(M \setminus \{Q\})$. Hence, our strategy is to consider separately each border $\sigma = (f, c)$ in $\text{zone}_k(M \setminus \{Q\})$ and see what happens to it when Q is added to $M \setminus \{Q\}$. We distinguish between two cases.

1. If σ does not intersect Q , the transition is trivial: In this case, σ may or may not become a border in $\text{zone}_k(M)$. For example, in Figure 6.4, σ_1 becomes a border in $\text{zone}_k(M)$, whereas σ_3 does not. But in either

case, the contribution of σ to the left hand side of (6.1) is ≤ 1 , whereas it contributes 1 to the term $z_k(M \setminus \{Q\})$ on the right hand side.

2. If σ does intersect Q , the transition is more interesting. In this case, σ is split into two borders. $\text{Zone}_k(M)$ can contain either one or both of these borders. In Figure 6.4, $\text{zone}_k(M)$ contains both borders split from σ_5 , but only one of the borders split from σ_4 . In general, $\text{zone}_k(M)$ contains both borders split from $\sigma = (f, c)$ iff S intersects the cell c on both sides of Q . When this happens, S must intersect $c \cap Q$ implying that $\sigma \cap Q \in \text{zone}_k(M_Q)$. In other words, when σ contributes two to the left hand side of (6.1), it also contributes one to each of the two terms on the right hand side.

This proves (6.1).

If we sum (6.1) over all hyperplanes $Q \in M$, we get

$$(m - k)z_k(M) \leq \sum_{Q \in M} z_k(M \setminus \{Q\}) + z_k(M_Q). \quad (6.2)$$

This is because every border in $\text{zone}_k(M)$ is counted $m - k$ times, once each for every hyperplane $Q \in M$ that does not contain it. It follows that

$$(m - k)z_k(m, d) \leq m[z_k(m - 1, d) + z_k(m - 1, d - 1)], \quad (6.3)$$

for $k < d, d > 1$, and $k < m$. Let us put $z_k(m, d) = \binom{m}{k} \bar{z}_k(m, d)$ in the above recurrence. This yields a simpler recurrence:

$$\bar{z}_k(m, d) \leq \bar{z}_k(m - 1, d) + \bar{z}_k(m - 1, d - 1), \quad (6.4)$$

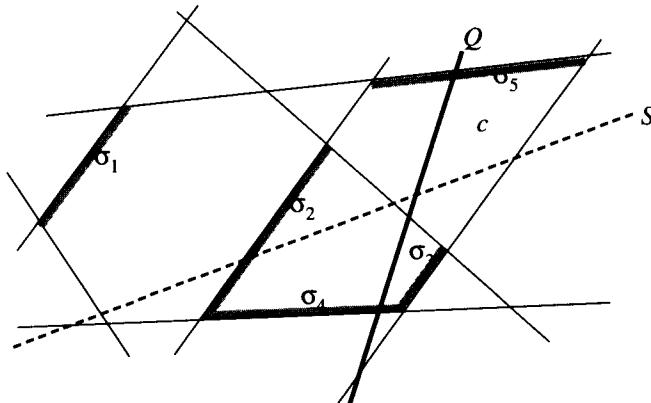


Figure 6.4: Changes to the zone of S during the addition of Q .

for $d > 1, k < d$, and $m > k$. We now apply this recurrence iteratively on the first summand. This yields

$$\bar{z}_k(m, d) \leq \bar{z}_k(k, d) + \sum_{\substack{k \leq n < m}} \bar{z}_k(n, d-1), \quad (6.5)$$

for $d > 1, k < d$, and $m > k$. The theorem now follows from simple induction on d . The base case $z_k(m, 2) = O(m)$ follows from the Zone Theorem for arrangements of lines (Theorem 2.8.2).

Now consider the case when $d > 2$. Assume inductively that $z_k(n, d-1) = O(n^{d-2})$, for $k < d$, where the Big-Oh notation hides a constant that depends on k and d . Then $\bar{z}_k(n, d-1) = O(n^{d-2-k})$. Hence, by (6.5),

$$\bar{z}_k(m, d) \leq \bar{z}_k(k, d) + \sum_{\substack{k \leq n < m}} O(n^{d-2-k}) = O(m^{d-1-k}), \quad (6.6)$$

for $k < d-1$. This implies that $z_k(m, d) = O(m^{d-1})$, provided $k < d-1$. When $k = d-1$, the above method would only yield a weaker $O(m^{d-1} \log m)$ bound. However, we can obtain the stronger $O(m^{d-1})$ bound in this case too. For this, we will need to make use of the Euler's relation for convex polytopes, which will be proved in Chapter 7 (Section 7.2.1). Fix any cell c in the arrangement $G(M)$. It is a convex polytope of dimension d . Let $r_k(c)$ denote the number of its faces of dimension k . Euler's relation states that the alternating sum $\sum_{k=0}^d (-1)^k r_k(c)$ is one if c is bounded. The alternating sum is 0 if c is unbounded. (Alternatively, we can assume that every c is bounded by adding, right in the beginning, $2d$ large hyperplanes bounding a huge box approaching infinity, and restricting ourselves within this box.) Let $\bar{r}_k(c)$ denote the number of subfaces of c of codimension k . Then Euler's relation can be rewritten as

$$\sum_{0 \leq k \leq d} (-1)^{d-k} \bar{r}_k(c) \geq 0.$$

Summing these relations over all d -cells in $G(M)$ intersecting S , we conclude that

$$\sum_{0 \leq k \leq d} (-1)^{d-k} z_k(M) \geq 0.$$

Because we have already proved that $z_k(M) = O(m^{d-1})$, for $k < d-1$, it follows that

$$z_{d-1}(M) - z_d(M) \leq \sum_{0 \leq k \leq d-2} (-1)^{d-k} z_k(M) = O(m^{d-1}). \quad (6.7)$$

On the other hand,

$$z_d(M) \leq \frac{2}{d} z_{d-1}(M). \quad (6.8)$$

This follows because we are assuming that the hyperplanes in M are in general position. Hence, each vertex of a d -cell $c \in G(M)$ is incident to exactly d edges of c (Exercise 6.0.2), and trivially each edge is incident to at most two vertices.

It follows from (6.7) and (6.8) that $z_k(M) = O(m^{d-1})$, for $k = d, d-1$ as well. This proves the Zone Theorem. \square

Exercises

6.2.1 Show that the size of the zone of S in $G(N)$ does not decrease under almost all infinitesimal perturbations of N .

6.2.2 In the incremental construction in this section, we maintained the full facial lattice of $G(N^i)$. It suffices to maintain the 2-skeleton of $G(N^i)$, because the facial lattice of $G(N)$ can be recovered from its 2-skeleton in time proportional to its size (Exercise 6.0.6). Show that the cost of this modified incremental construction is $O(n^d)$, using only the previous Zone Theorem for arrangements of lines.

6.2.3 Give the required modifications to the incremental algorithm when the hyperplanes are not in general position. Show that the running time remains $O(n^d)$. What happens in the presence of round-off errors?

****6.2.4** Show that

$$\sum_f |f|^2 = O(n^d \log^{\lfloor d/2 \rfloor - 1} n),$$

where f ranges over all d -cells in $G(N)$. (Hint: Unlike in the two-dimensional case (Exercise 2.8.9), this does not follow from the Zone Theorem. Rather, one needs to extend the proof technique. In addition, you will need to use the Dehn–Sommerville relations and the upper bound theorem for convex polytopes (Section 7.2). First, show that the above sum is of the order of $\sum_{(g,h)} 1$, where (g, h) ranges over all visible pairs of $\lceil d/2 \rceil$ -dimensional borders. By a visible pair, we mean that g and h border the same d -cell in $G(N)$. Bound the sum by extending the technique used for proving the Zone Theorem.)

6.2.5 Let $G(N)$ be an arrangement of n hyperplanes. Let P be any set of $m < n^d$ cells in $G(N)$. Show that the total size of all cells in P is $O(m^{1/2} n^{d/2} \log^{(\lfloor d/2 \rfloor - 1)/2} n)$. (Hint: Combine the result in Exercise 6.2.4 with Cauchy–Schwarz inequality very much as in Exercise 2.8.11.)

****6.2.6** (Zone of a hypersurface in an arrangement): Fix any bounded degree algebraic hypersurface in R^d . Show that $\sum_f |f| = O(n^{d-1/2} \log^{(\lfloor d/2 \rfloor - 1)/2} n)$, where f ranges over all cells in the arrangement $G(N)$ intersecting the hypersurface. (Hint: First, show that the hypersurface can intersect at most $O(n^{d-1})$ cells. Next, use Exercise 6.2.5.)

6.3 Canonical triangulations

In the previous section, we gave an optimal algorithm for constructing an arrangement of hyperplanes in arbitrary dimension. In the rest of this chapter,

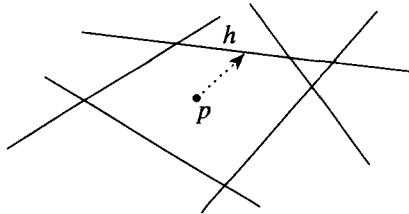


Figure 6.5: Ray shooting.

we shall investigate some associated search problems. There are two important search problems associated with arrangements: point location and ray shooting. In the point location problem, the goal is to locate the cell in the arrangement containing the query point quickly. In the ray shooting problem, we are given a ray originating from the query point. The goal is to locate the first hyperplane in the arrangement hit by this ray (Figure 6.5).

We shall give two search structures for point location. Both are based on random sampling techniques (Chapter 5). The first one is based on the bottom-up sampling technique (Section 5.3). Its advantage is that it can be used for ray shooting as well. The second one is based on the top-down sampling technique (Section 5.2). Its advantage is that it can be extended to deal with simplex range queries in a dual form. Both search structures can be dynamized using the dynamic sampling technique.

The random sampling techniques in Chapter 5 cannot be applied to arrangements directly. This is because an arrangement does not have the bounded-degree property: Its cells can have arbitrarily large sizes. (By the size of a cell, we mean the number of its subsfaces.) For this reason, one has to deal with a suitable refinement of the arrangement whose cells have bounded sizes. The refinement that we shall use is the so-called canonical triangulation. It is obtained by a straightforward generalization of the scheme that we used for arrangements of lines (Section 5.1).

Let N be a set of hyperplanes in R^d . Let $G(N)$ be the arrangement formed by them. Let Γ be any simplex in R^d . Consider the restricted arrangement $G(N) \cap \Gamma$. If we are interested in $G(N)$, we can let the vertices of Γ approach infinity. A canonical triangulation of $G(N) \cap \Gamma$ is defined as follows (see Figure 5.4 for a two-dimensional case). Fix the coordinate x_d . We shall triangulate the faces of $G(N) \cap \Gamma$ by induction on their dimension. For dimension one, we do not need to do anything. Consider now any face f of dimension $j > 1$. Let v be its bottom. By this, we mean the vertex with

the minimum x_d -coordinate.¹ By the induction hypothesis, all subsfaces of f have been triangulated. We triangulate f by extending all simplices on its border to cones with an apex at v . The cones are strictly speaking cylinders if the vertex v lies at infinity. We shall denote the canonical triangulation of $G(N) \cap \Gamma$ by just $H(N)$ if the simplex Γ is implicit. Its cells will be called d -simplices. They can be unbounded.

Every d -simplex of $H(N)$ is defined by a bounded number of hyperplanes in N adjacent to it. This makes the random sampling results in Chapter 5 immediately applicable. To see this, fix N and Γ . For any subset $R \subseteq N$, let $H(R)$ denote the canonical triangulation of $G(R) \cap \Gamma$. Call a simple Δ in R^d feasible if it occurs in $H(R)$, for some $R \subseteq N$. Define its conflict (stopper) set relative to N to be the set of hyperplanes in $N \setminus R$ intersecting Δ . Its trigger set is defined to be the set of hyperplanes in R adjacent to Δ . It is easily seen that, for a fixed Δ , these definitions do not depend on R . Thus we get a configuration space of feasible simplices over N just as in the case of arrangements of lines (Section 5.1). If R is a random sample of N of size r , it follows from the random sampling result (Theorem 5.1.2) that, with high probability, the conflict size of every simplex in $H(R)$ is at most $a(n/r) \log r$, for some constant a . This result also follows from the random sampling result for range spaces (Theorem 5.7.4). Indeed, it follows from Theorem 5.7.4 that, with high probability, R is a $O((\log r)/r)$ -net for the space $\Pi(N)$ of linear ranges induced by N (Example 5.7.8). So assume that R is indeed a $O((\log r)/r)$ -net. Now consider any simplex $\Delta \in H(R)$. Consider a simplex Δ' that is contained in the interior of Δ , but whose boundary is infinitesimally close to the boundary of Δ . Clearly, the linear ranges induced by the edges of Δ' are active over R , i.e., they do not intersect any hyperplane in R . Since R is a $O((\log r)/r)$ -net, this means that each edge of Δ' can intersect at most $O([n \log r]/r)$ hyperplanes in N . Fix a vertex of Δ' . Each hyperplane intersecting Δ must intersect one of the $(d+1)$ edges of Δ' adjacent to this fixed vertex. Hence, it again follows that any $\Delta \in H(R)$ can intersect at most $O((d+1)(n \log r)/r) = O((n \log r)/r)$ hyperplanes in N . A similar result holds in the setting of Bernoulli sampling, too (Exercise 5.1.1).

Exercises

6.3.1 Let $H(R)$ be defined as above. Show that $H(R)$ can be computed in $O(r^d)$ time. (Hint: Compute $G(R)$ in $O(r^d)$ time using the incremental procedure in Section 6.1. Then, canonically triangulate each cell in $G(R)$ by the procedure in this section. Note that the canonical triangulation of a d -cell in $G(R)$ does not increase its size by more than a constant factor.)

¹We assume that the x_d -coordinate is in general position. So the bottom is uniquely defined. Really speaking, uniqueness is not necessary. One could use a fixed lexicographic ordering to break ties.

6.3.2 Show that the total number of simplices in $H(R)$ intersected by any hyperplane $Q \in N \setminus R$ is $O(r^{d-1})$. Moreover, all of them can be determined in $O(r^{d-1})$ time. (Hint: Canonical triangulation of a d -cell in $G(R)$ does not increase its size by more than a constant factor. All simplices in $H(R)$ intersecting Q can be determined in time proportional to their number by the procedure for searching along a hyperplane in Section 6.1. It follows that the cost of this search is proportional to the total size of the d -cells in $G(R)$ intersected by Q . By the Zone Theorem, this is $O(r^{d-1})$.)

6.3.1 Cutting Lemma

The size of $H(R)$ that was constructed above is $O(r^d)$. It turns out that one can refine $H(R)$ further, keeping the *expected* space requirement within the $O(r^d)$ bound, so that following stronger property holds:

The conflict size of every simplex in the refined triangulation is bounded by n/r .

A triangulation with this property is called a $(1/r)$ -*cutting*. Let us see how to refine $H(R)$ to achieve this property. For each $\Delta \in H(R)$, let $L(\Delta)$ denote its conflict list. This is the set of hyperplanes in $N \setminus R$ intersecting Δ . Let $l(\Delta)$ be its size. If $l(\Delta) \leq n/r$, we are fine. Otherwise, let $s(\Delta) = l(\Delta)/(n/r)$ denote the excess ratio for Δ . Take a random sample of $L(\Delta)$ of size $s'(\Delta) = bs(\Delta) \log s(\Delta)$, for a large enough constant b . Let $\tilde{\Delta}$ denote the canonical triangulation of Δ induced by this random sample. With high probability, every simplex in the refinement $\tilde{\Delta}$ intersects $O([l(\Delta)/s'(\Delta)] \log s'(\Delta)) \leq n/r$ hyperplanes in $L(\Delta)$. This is assuming that the constant b is large enough. If this property is not satisfied for every simplex of $\tilde{\Delta}$, we keep on repeating the experiment until the property holds. The expected number of required repetitions is $O(1)$.

We refine each simplex in $H(R)$ in this fashion. Let $\tilde{H}(R)$ denote the resulting refinement of $H(R)$. Clearly, each simplex of $\tilde{H}(R)$ intersects at most n/r hyperplanes in $N \setminus R$. Hence, it is a $(1/r)$ -cutting. The size of $\tilde{H}(R)$ is proportional to $\sum_{\Delta} s'(\Delta)^d$, where Δ ranges over the simplices in $H(R)$. Since

$$s'(\Delta) = b s(\Delta) \log s(\Delta) = O(s^2(\Delta)),$$

this is bounded by

$$\sum_{\Delta} s^{2d}(\Delta) = \frac{1}{(n/r)^{2d}} \sum_{\Delta} l^{2d}(\Delta). \quad (6.9)$$

Since R is a random sample of N of size r , the expected value of $\sum_{\Delta} l^{2d}(\Delta)$ is $(n/r)^{2d} O(r^d)$ (Theorem 5.5.5). In other words, the “average” conflict size of $H(R)$ is n/r . It follows from (6.9) that the expected size of $\tilde{H}(R)$ is $O(r^d)$.

This implies the following.

Lemma 6.3.1 (Cutting Lemma) For any set N of hyperplanes, there exists a $(1/r)$ -cutting of $O(r^d)$ size.

The cost of constructing the cutting is analyzed in the following exercise.

Exercises

6.3.3 Show that the expected cost of finding a $(1/r)$ -cutting as in the Cutting Lemma is $O(nr^{d-1})$. (Hint: Implement and analyze the construction procedure carefully. For constructing the conflict lists, use Exercise 6.3.2.)

6.3.4 Assume that each hyperplane in N is assigned a positive integral weight. By $(1/r)$ -cutting, we now mean that the total weight of the hyperplanes intersecting any simplex in this cutting is at most $(1/r)$ th fraction of the total weight of N . Prove the analogue of the Cutting Lemma in this weighted setting. (Hint: Normalize the weight function (in linear time) so that the total weight of N is n . Next, construct a multiset N' from N by replacing each hyperplane $h \in N$ by $\lceil w(h) \rceil$ copies, where $w(h)$ is its normalized weight. The size of N' , taking into account the multiplicities, is clearly $2n$, because $\lceil w(h) \rceil \leq w(h) + 1$, and the sum of $w(h)$ is n . Now use the algorithm for the unweighted case on N' to find its $(1/2r)$ -cutting. Strictly speaking, the hyperplanes in N' are not in general position. But this can be overcome by perturbing them infinitesimally (in a symbolic fashion). Check that the resulting (unweighted) $(1/2r)$ -cutting for N' is a (weighted) $(1/r)$ -cutting for N .)

6.4 Point location and ray shooting

In this section, we give a dynamic search structure that can be used for point location as well as ray shooting. First, we shall state the search structure in the static setting and turn to its dynamization later.

6.4.1 Static setting

Our search structure will actually let us locate a certain *antenna* of the query point. This antenna location problem is a common generalization of the ray shooting and the point location problem. It can be stated as follows.

Let N denote the given set of n hyperplanes. Let $G(N)$ be their arrangement. Let p be any query point. Let us assume that the query also specifies, in addition, a certain linear function z . Let us define the *antenna* of the query point p with respect to z as follows (Figure 6.6). Consider the ray originating from p in the positive z -direction. By the positive z -direction, we mean the direction of the gradient of z at p . Let p_1 be the intersection of this ray with the first hyperplane $H_1 \in N$ that it hits. We shall say that p_1 is *above* p . Similarly, let p_2 be the intersection of the ray originating from p in the negative z -direction with the first hyperplane $H_2 \in N$ that it hits.

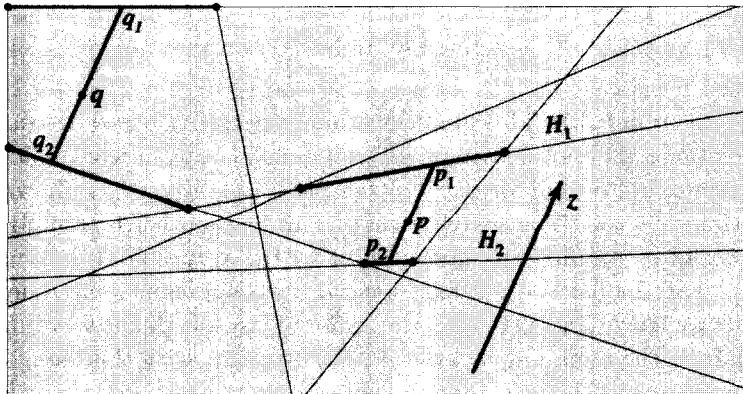


Figure 6.6: Antennas.

We shall say that p_2 is *below* p . We define $\text{antenna}(p)$ to be the union of the segment $[p_1, p_2]$ together with the recursively defined $\text{antenna}(p_1)$ and $\text{antenna}(p_2)$. Here $\text{antenna}(p_1)$ and $\text{antenna}(p_2)$ are defined with respect to z in the lower-dimensional arrangements $G(N) \cap H_1$ and $G(N) \cap H_2$, respectively. In the above definition, we have implicitly assumed that the antenna is bounded. This can be ensured by adding to N hyperplanes bounding a large cube approaching infinity. Once this is done, we restrict everything within this cube.² For example, refer to the antenna of q in Figure 6.6.

In this section, we shall give a search structure that will let us determine a complete specification of $\text{antenna}(p)$ quickly. If we are just interested in point location, we choose z arbitrarily. Once we know the terminals (endpoints) of $\text{antenna}(p)$, we also know the cell in $G(N)$ containing p : It is the one adjacent to all these terminals. If we are interested in ray shooting, we choose z so that the positive z -direction coincides with the direction of the given ray. The antenna will tell us the point p_1 above p and also the hyperplane H_1 containing this point. This is the hyperplane that is hit by the query ray.

In what follows, we shall denote the canonical triangulation of $G(N)$ by $H(N)$ (Section 6.3). The bottom vertices in this canonical triangulation are defined with respect to a fixed x_d -coordinate. The z -coordinate in the query is completely independent of this x_d -coordinate.

First, we shall describe our search structure in the static setting. We shall turn to its dynamization later. We have already described in Section 5.3.1 a point location structure for arrangements of lines based on bottom-up sampling. The data structure for higher-dimensional arrangements has the same

²The hyperplanes bounding this cube are to be defined symbolically as in the bounding box trick in Section 3.2.

organization. Let us denote the point location structure to be associated with $G(N)$ by $\tilde{G}(N)$. It is defined by induction on the dimension of $G(N)$.

For the basis case $d = 2$, we let $\tilde{G}(N)$ be the point location structure defined in Section 5.3.1. This point location structure is based on trapezoidal decompositions, and not on canonical triangulations, as in what follows. The use of trapezoidal decompositions in the basis case $d = 2$ is only for saving a logarithmic factor. The search structure in Section 5.3.1 actually locates the trapezoid containing the query point in the trapezoidal decomposition of $G(N)$. For the sake of antenna location, we augment it as follows. We maintain, for each line $Q \in N$, the ordered list of all intersections on it resulting from the remaining lines in N . We associate with this list a balanced search tree—a skip list or a randomized binary tree will do. Once we know the trapezoid containing the query point p , we also know the lines in N above and below p . Using the search trees associated with these lines, we can now easily determine the terminals of $\text{antenna}(p)$. In a dynamic setting, the above balanced trees need to be updated during the addition or deletion of a line in the data structure. We shall leave this routine update to the reader. One can also check that the bound on the update cost retains the same form as before.

Now assume that $d > 2$. The search structure $\tilde{G}(N)$ is defined by applying the bottom-up sampling scheme (Section 5.3). Starting with the set N , we obtain a gradation $N = N_1 \supseteq N_2 \supseteq \dots \supseteq N_{r-1} \supset N_r = \emptyset$, where N_{i+1} is obtained from N_i by flipping a fair coin³ independently for each hyperplane in N_i and retaining only those hyperplanes for which the toss was a success (heads). As we have seen, the length of this gradation is $\tilde{O}(\log n)$, where n is the size of N . The state of $\text{sample}(N)$ is going to be determined by the gradation of N . $\text{Sample}(N)$ will consist of r levels, where r is the length of the gradation. The set N_i , $1 \leq i \leq r$, can be thought of as the set of hyperplanes stored in the i th level. Each level i is associated with the canonical triangulation $H(N_i)$ of the arrangement $G(N_i)$. Each d -simplex of $H(N_i)$ is associated with the conflict list of the hyperplanes in $N_{i-1} \setminus N_i$ intersecting it. The size of this list is called the conflict size of the simplex. Between every two successive levels i and $i - 1$, we keep a certain *descent structure*, denoted by $\text{descent}(i, i - 1)$. It is used in answering queries in a manner to be described soon.

$\text{Descent}(i, i - 1)$ contains a recursively defined search structure for the $(d - 1)$ -dimensional arrangement $G(N_{i-1}) \cap Q$, for each hyperplane $Q \in N_{i-1}$. We shall denote this search structure by $\tilde{G}(N_{i-1}, Q)$.

³We do not really need the coin to be fair. We only need that the probability of success is a fixed constant between 0 and 1.

Important facts regarding our search structure: The number of levels in the search structure is $\tilde{O}(\log n)$, and the total size of the sets N_1, N_2, \dots is $\tilde{O}(n)$. This follows just as in the case of skip lists (Section 1.4). Intuitively, it follows because, for every $l > 1$, N_l is a random sample of N_{l-1} of roughly half the size. This latter fact also implies that with very high probability, the conflict size of every d -simplex $\Delta \in H(N_l)$ is $\tilde{O}(\log n)$, for every $l > 1$. Formally, this follows by applying the random sampling results (Theorem 5.1.2, Exercise 5.1.1) to the configuration spaces of feasible simplices over N_l (Section 6.3), for every l . In what follows, we shall denote the size of N_l by n_l .

Let us turn to the queries. Let p be a query point. Let z be the given linear function. It defines the antenna of p in $G(N)$. Our goal is to determine this antenna. By this, we mean determining the terminals of the antenna. In addition, we shall also determine, for each segment of the antenna, the hyperplanes in N containing it. Curiously enough, we shall not be able to locate the d -simplex in $H(N)$ containing p . This does not matter because the canonical triangulation is only a tool in our method. Even in the point location problem, one is only interested in locating the face of $G(N)$ containing the query point. If one could also locate the simplex in $H(N)$ containing the query point, that would have been an additional bonus.

To answer the query, we descend through our search structure, one level at a time, starting at the last empty level. Inductively, assume that we have determined the antenna of p in $G(N_i)$. Our goal is to determine the antenna in $G(N_{i-1})$ quickly. This will be done with the help of $\text{descent}(i, i - 1)$. Let Q_1 and Q_2 be the hyperplanes in N_i immediately above and below p in the z -direction (Figure 6.7). Consider the line through p parallel to the z -direction. Let p_1 and p_2 be the points of intersection of this line with Q_1 and Q_2 , respectively. Then $\text{antenna}(p)$ in $G(N_i)$ is the union of the segment $[p_1, p_2]$ together with the recursively defined $\text{antenna}(p_1)$ and $\text{antenna}(p_2)$. Let v_1, v_2, \dots, v_{2^d} be the terminal points of $\text{antenna}(p)$. We are assuming here that the antenna is bounded. As we have already seen, this can be ensured by restricting everything within a large symbolic box approaching infinity.

Lemma 6.4.1 *The hyperplane in N_{i-1} that is immediately above (or below) p in the z -direction is either Q_1 (resp. Q_2) or it must intersect a d -simplex in $H(N_i)$ adjacent to some v_i , $1 \leq i \leq 2^d$.*

Proof. We proceed by induction on d . For the sake of induction, we shall prove a somewhat stronger statement. We shall prove that any hyperplane $Q \in N_{i-1} \setminus N_i$ that intersects $\text{antenna}(p)$ must intersect a d -simplex in $H(N_i)$ adjacent to some v_i , $1 \leq i \leq 2^d$ (Figure 6.7).

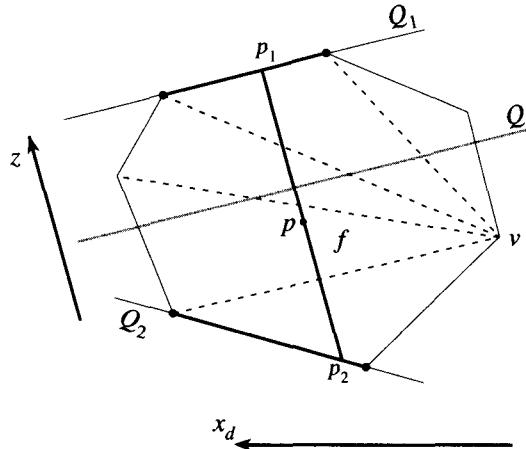


Figure 6.7: Descent.

Suppose Q intersects antenna(p_1). By the inductive hypothesis, it must intersect some $(d - 1)$ -simplex Δ which is contained in the restriction of $H(N_i)$ to Q_1 and which is adjacent to a terminal of antenna(p_1). So Q intersects a d -simplex in $H(N_i)$ adjacent to Δ . A similar thing happens if Q intersects antenna(p_2). In the remaining case, Q intersects $[p_1, p_2]$, but neither antenna(p_1) nor antenna(p_2) (Figure 6.7). This means antenna(p_1) lies completely above Q in the z -direction and antenna(p_2) lies completely below Q . Let f be the d -cell in $G(N_i)$ containing p . Let v be its bottom with respect to the x_d -coordinate. The canonical triangulation of f consists of simplices with an apex at v . Assume, without loss of generality, that v lies below Q , the other case being symmetric. Because antenna(p_1) lies completely above Q , it is clear that Q must intersect all d -simplices in f adjacent to the terminals of antenna(p_1). \square

Let us return to our descent from level i to $i - 1$. Lemma 6.4.1 implies that we only need to check through the conflict lists of all $O(1)$ simplices adjacent to the terminals of antenna(p) in $G(N_i)$. This lets us determine the hyperplanes Q'_1 and Q'_2 in N_{i-1} that are immediately above and below p . Since all conflict lists have $\tilde{O}(\log n)$ size, this takes $\tilde{O}(\log n)$ time. Let p'_1 be the point in Q'_1 immediately above p , and let p'_2 be the point in Q'_2 immediately below p . We can determine antenna(p'_1) and antenna(p'_2) recursively by using the search structures $\tilde{G}(N_{i-1}, Q'_1)$ and $\tilde{G}(N_{i-1}, Q'_2)$. This determines the antenna of p in $G(N_{i-1})$ completely.

Analysis

By induction on d , one can show that the query time is $\tilde{O}(\log^{d-1} n)$. We already know this for the basis case $d = 2$ (Section 5.3.1). So assume that $d > 2$. The dominant cost of the descent between two successive levels comes from the use of the recursively defined descent structure. By the induction hypothesis, this cost is $\tilde{O}(\log^{d-2} n)$. Our search structure has $\tilde{O}(\log n)$ levels. Hence, the query time is $\tilde{O}(\log^{d-1} n)$, for $d > 2$. This bound is for a *fixed* query point. But this holds for any query point (by Observation 1.3.1), because the number of distinct search paths in our data structure is bounded by a polynomial in n of fixed degree; we leave it to the reader to check the latter fact.

By induction on d , one can also show that the cost of building the search structure is $\tilde{O}(n^d)$. We already know that this holds for the basis case $d = 2$ (Section 5.3.1). So assume that $d > 2$. The size of each $H(N_i)$ is $O(n_i^d)$ and it can be built in the same order of time (Exercise 6.3.1). Now consider the cost of computing the conflict information. For each hyperplane $Q \in N_{i-1} \setminus N_i$, the simplices in $H(N_i)$ intersecting Q can be determined in $O(n_i^{d-1})$ time (Exercise 6.3.2). Hence, the total cost of computing the conflict information at the i th level is $O(n_i^d)$. It follows from the induction hypothesis that the descent structure between the i th and the $(i-1)$ th level, which consists of n_{i-1} recursively defined structures, can be computed in $O(n_{i-1}^d)$ time. Summing over all levels, it follows that the whole search structure can be computed in $O(\sum_i n_i^d) = \tilde{O}(n^d)$. The last equality follows because $\sum_i n_i$, the total size of the sets N_i , is $\tilde{O}(n)$.

To summarize:

Theorem 6.4.2 Antenna location (a generalization of point location and ray shooting) in an arrangement of n hyperplanes in R^d can be done in $\tilde{O}(\log^{d-1} n)$ time using a search structure that can be computed in $\tilde{O}(n^d)$ time and space.

Exercises

6.4.1 Check that the number of distinct search paths in the above data structure is bounded by a polynomial in n of fixed degree.

6.4.2 Show how $\tilde{O}(n^d)$ space bound can be converted to $O(n^d)$ bound without changing other bounds. (Hint: Use the clipping trick in Section 5.3.1.)

6.4.3 Describe a trivial $O(n)$ time procedure for locating $\text{antenna}(p)$ in $G(N)$. (Hint: Locate the points p_1, p_2 above and below p by computing the intersections of the ray with all hyperplanes in N . Proceed recursively.)

Give a trivial $O(n)$ time procedure for locating the query point in the canonical triangulation $H(N)$ of $G(N)$. (Hint: First, locate the cell f in $G(N)$ containing p by locating its antenna trivially as above. Let v be the bottom of this cell.

Consider the ray originating from v and passing through p . In $O(n)$ time, trivially determine the facet of f hit by this ray. Locate the hit point recursively in the canonical triangulation of this facet.)

6.4.2 Dynamization

The previous search structure can be easily dynamized using the dynamic sampling technique (Section 5.4). We shall maintain our search structure in such a way that, at any given time, its state will be independent of the actual sequence of updates that built it. Thus, if N were to denote the set of current (undeleted) hyperplanes, then the search structure $\tilde{G}(N)$ would be as if it were built by applying the previous static procedure to N . This will ensure that the static bound on the query cost holds in the dynamic setting as well.

We shall only describe addition because deletion will be its exact reversal. So let us see how to add a new hyperplane h to $\tilde{G}(N)$. We first toss an unbiased coin repeatedly until we get failure (tails). Let j be the number of successes before getting failure. We shall simply add h to levels 1 through $j + 1$. For $1 \leq l \leq j + 1$, let \bar{N}_l denote $N_l \cup \{h\}$. Addition of h to the l th level is carried out in three steps.

1. Update $H(N_l)$ to $H(\bar{N}_l)$.
2. Construct the conflict lists of the new d -simplices in $H(\bar{N}_l)$.
3. Update descent($l + 1, l$). This means, for each $Q \in N_l$, we update the lower-dimensional search structure $\tilde{G}(N_l, Q)$ recursively, and build $\tilde{G}(N_l, h)$ from scratch, using the static procedure.

It remains to elaborate the first two steps.

The d -cells in $G(N_l)$ intersecting h can be determined by a search along h in time proportional to their number (Section 6.1). Let f be any d -cell in $G(N_l)$ intersecting h . We remove all d -simplices in the old triangulation of f . We split f along h into two d -cells, f_1 and f_2 . We triangulate f_1 and f_2 from scratch. All this takes $O(|f|)$ time, where $|f|$ denotes the size of f , i.e., the total number of its subfaces.

We also need to construct the conflict lists of the new d -simplices within f_1 and f_2 . Let Q be any hyperplane in $N_{l-1} \setminus N_l$ that intersects f . The old conflict information tells us every such Q and also all edges of f intersecting Q . Starting at these boundary edges, we search along Q (Figure 6.8) in the new triangulations of f_1 and f_2 . (We have already described a similar search along a hyperplane in Section 6.1. Hence, there is no need to elaborate this search procedure further.) This lets us determine all d -simplices within f_1 and f_2 intersecting Q . It takes time proportional to their number. We repeat this procedure for every such Q . At the end, we obtain conflict lists of all

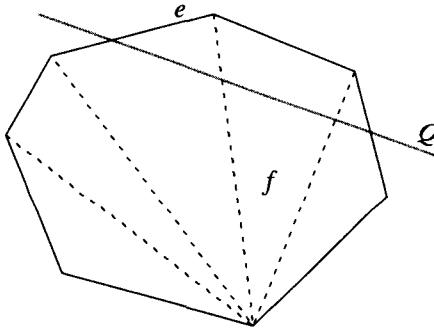


Figure 6.8: Conflict propagation.

new simplices in f_1 and f_2 in time proportional to their total conflict size, ignoring an additive $O(|f|)$ term.

Analysis

The cost of updating $H(N_l)$ to $H(\bar{N}_l)$ is $O(\sum_f |f|)$, where f ranges over all d -cells of $G(N_l)$ intersecting h . By the Zone Theorem (Section 6.2), this is $O(n_l^{d-1})$. The size of every conflict list, new or old, is $\tilde{O}(\log n)$. Hence, the total cost of updating the conflict lists is $\tilde{O}(\sum_f |f| \log n)$, where f again ranges over all d -cells of $G(N_l)$ intersecting h . By the Zone Theorem again, this cost is $\tilde{O}(n_l^{d-1} \log n)$.⁴ This also dominates the cost of rebuilding $\tilde{G}(N_l, h)$ from scratch.

Thus, the cost of updating the search structure is

$$\tilde{O}(\log n \sum_l n_l^{d-1}) = \tilde{O}(n^{d-1} \log n).$$

We did not take into account here the cost of the updating the recursively defined descent structures. The same bound continues to hold even when we take this cost into account. This can be proved by induction on d . The cost of update is $\tilde{O}(n \log n)$ for the basis case $d = 2$ (Section 5.4.1). Descent($l, l - 1$) consists of n_{l-1} recursively defined search structures. By the induction hypothesis, the cost of updating them is $\tilde{O}(n_{l-1} n_{l-1}^{d-2} \log n) = \tilde{O}(n_{l-1}^{d-1} \log n)$. This again sums to $\tilde{O}(n^{d-1} \log n)$ over all levels.

We have thus shown that the cost of inserting a new hyperplane in $\tilde{G}(N)$ is $\tilde{O}(n^{d-1} \log n)$. Deletion is the exact reversal of addition: The cost of

⁴Here the probability of error, implied by the \tilde{O} notation, is inversely proportional to a large degree polynomial in n , not just n_l . This stronger property holds because Theorem 5.1.2 (or more precisely, Exercise 5.1.1 for Bernoulli sampling) implies that the size of the conflict list at each level is $\tilde{O}(\log n)$; put $s = n$ in that theorem. A similar property holds in what follows.

deletion is of the same order as the cost of inserting the deleted hyperplane immediately afterwards. Hence, the cost of deleting any hyperplane from $\tilde{G}(N)$ is also $\tilde{O}(n^{d-1} \log n)$.

We have thus proved the following.

Theorem 6.4.3 *There exists a dynamic search structure for point location and ray shooting in an arrangement of hyperplanes in R^d with the following performance: $\tilde{O}(\log^{d-1} n)$ query time, $\tilde{O}(n^d)$ space requirement, and $\tilde{O}(n^{d-1} \log n)$ cost of adding or deleting a hyperplane.*

The expected cost of an update is $O(n^{d-1})$. This is shown in the following exercise.

Exercises

6.4.4 Show that the expected cost of deleting or adding a hyperplane to the search structure in this section is $O(n^{d-1})$. The expectation is solely with respect to randomization in the search structure. (Hint: The “average” conflict size of all affected simplices in a given level is $O(1)$. Show this by applying Theorem 5.5.5 to the configuration space of feasible simplices over N_l intersecting h .)

†Can you show that the cost of update is actually $\tilde{O}(n^{d-1})$?

†**6.4.5** Can you reduce the query cost to $\tilde{O}(\log n)$?

6.5 Point location and range queries

In this section, we give another point location structure for arrangements based on top-down sampling (Section 5.2). Dynamization of this search structure is a bit more difficult and the updates are not as efficient as in the previous section. On the other hand, this search structure can be easily extended to deal with half-space and simplex range queries in a dual setting, though not the ray shooting queries, like the earlier search structure.

We have already given a point location structure for arrangements of lines based on top-down sampling (Section 5.2.1). One can generalize this point location structure to arbitrary dimension by using the canonical triangulation in Section 6.3. The generalization can be achieved by a *verbatim* translation: Lines become hyperplanes and triangles become d -simplices. The root of this search structure is associated with a canonical triangulation $H(R)$, where R is a random sample of N of a large enough constant size (Figure 6.9). This canonical triangulation is restricted within a large enough symbolic simplex Γ (whose vertices can approach infinity if you wish). The children of this root correspond to the simplices of $H(R)$. The subtree rooted at every child is built recursively over the set of hyperplanes conflicting with (i.e., intersecting) the simplex labeling that child. The recursion stops when the number of conflicting hyperplanes at a node drops below a chosen constant.

It follows by a *verbatim* translation of the argument in Section 5.2.1 that the size of this search structure is $O(n^{d+\epsilon})$ and that the query time is $O(\log n)$. It can be built in $O(n^{d+\epsilon})$ expected time. This time, the exponent contains d because the size of an arrangement is proportional to the d th power of the number of hyperplanes in it (Corollary 6.0.2) and it can be built in the same order of time (Section 6.1). (Throughout this book, ϵ stands for a positive constant that can be made arbitrarily small by appropriately choosing other constants in the algorithm.)

Half-space range queries

One nice feature of the preceding search structure is that it can be easily augmented to deal with half-space range queries. Recall from Section 2.4.1 that half-space range queries assume the following form in a dual setting: Given a set N of hyperplanes and a query point p , report all hyperplanes in N that are above or below the query point p . The above and below relations are defined with respect to the x_d -coordinate. To handle the queries of this form, we augment the nodes of the preceding search structure as follows. Recall that each child of the root corresponds to a simplex $\Delta \in H(R)$. We store at this child the lists $\text{above}(\Delta)$ and $\text{below}(\Delta)$ of all hyperplanes in N above and below the interior of Δ . Thus, in Figure 6.9, $\text{above}(\Delta) = \{l_1, l_2\}$ and $\text{below}(\Delta) = \{l_3, l_4, l_5\}$. To facilitate the count-mode queries, we also store at this child the sizes of $\text{above}(\Delta)$ and $\text{below}(\Delta)$. We augment the other nodes of the search tree in a recursive fashion.

Let us turn to the queries. They are answered by a small change to the point location procedure. Say we want to know the set of hyperplanes in N above the query point p . This set is obtained by just joining the lists

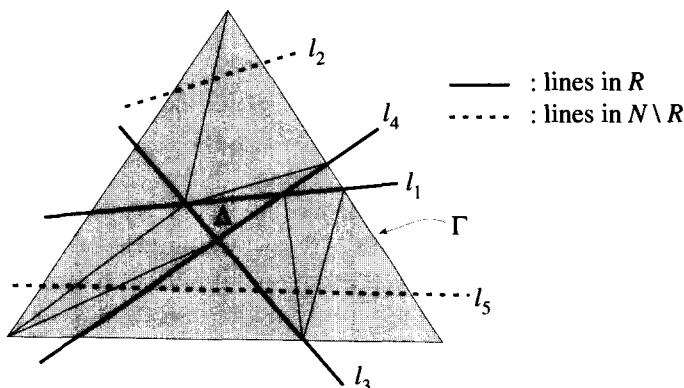


Figure 6.9: The canonical triangulation at the root.

$\text{above}(\Delta)$ for all nodes lying on the search path of p . In the count mode, we just add the sizes of these lists. Thus, the query time remains $O(\log n)$.

What about the space requirement or the expected preprocessing time? That still remains within the $O(n^{d+\epsilon})$ bound. Let us see why. Consider any node in the search tree. Let m denote the number of conflicting hyperplanes at this node. (For the root, $m = n$.) The node has $O(1)$ children, because our random samples have constant size. Hence, the auxiliary lists stored at all children of this node have $O(m)$ total space requirement. This is much less than the $O(m^{d+\epsilon})$ bound on the size of the subtree rooted at this node. Hence, it easily follows that the overall space requirement still respects a bound of the form $O(n^{d+\epsilon})$, but with a slightly larger ϵ . If you wish, you can also check this by writing down appropriate recurrence relations. Similarly, the bound for the expected preprocessing time also retains the same form.

Simplex range queries

We can extend the previous augmentation procedure even further to handle simplex range queries in a dual setting. A simplex is an intersection of at most $k \leq d + 1$ half-spaces. If $k < d + 1$, the simplex is unbounded. The simplex range queries assume the following form in the dual setting. We are given a list of $k \leq d + 1$ pairs $\{(p_i, \delta_i), i \leq k\}$, where p_i is a point in R^d and δ_i is a $+$ or $-$ symbol. Our goal is to report or count all hyperplanes in N such that, for each i , these hyperplanes are above p_i , if $\delta_i = +$, and below p_i , if $\delta_i = -$.

We build the search structure by induction on the query size k . If $k = 1$, the simplex becomes a half-space. We have already solved this problem. Otherwise, we proceed as follows. Our primary search structure will be a point location structure for the set N as described before. For each node in the search structure, we define the lists $\text{above}(\Delta)$ and $\text{below}(\Delta)$ as before. For $k = 1$, the auxiliary information at this node consisted of these lists together with their sizes. When $k > 1$, we store instead two recursively defined secondary search structures for the query size $k - 1$; one for the list of hyperplanes $\text{above}(\Delta)$ and the other for the list $\text{below}(\Delta)$.

The queries are answered as follows. We trace the search path of the point p_1 in the primary search structure. Say, $\delta_1 = +$, the other case being similar. At each node on this search path, we answer the query $\{(p_2, \delta_2), \dots, (p_k, \delta_k)\}$ with respect to the set $\text{above}(\Delta)$. This is done by using the secondary search structure. Finally, we combine the answers at all nodes on the search path of p_1 . It easily follows by induction on k that the query time is $O(\log^k n)$.

What about the space requirement? Here, too, we proceed by induction on k . For $k = 1$, we have already seen that the space requirement is $O(n^{d+\epsilon})$. Suppose this were true for query size $k - 1$. Then the total size of the secondary structures stored at the children of any node is $O(m^{d+\epsilon})$, where

m is the number of hyperplanes conflicting with that node. This bound is comparable to the $O(m^{d+\epsilon})$ bound on the total size of the primary subtree rooted at this node. Hence, it easily follows that the overall space requirement can be kept within $O(n^{d+\epsilon'})$, for some $\epsilon' > \epsilon$, by choosing an appropriate sample size. This can also be checked in a straightforward fashion by writing down recurrence relations in the spirit of (5.6). The constant ϵ' can be made arbitrarily small, by making ϵ correspondingly small. The preceding argument works only when k is bounded by a constant, because the constant ϵ increases in each step of the induction.

It similarly follows that the expected preprocessing time is also $O(n^{d+\epsilon})$.

The discussion in this section can be summarized as follows.

Theorem 6.5.1 Given a set of n hyperplanes in R^d , one can build a search structure of $O(n^{d+\epsilon})$ size in $O(n^{d+\epsilon})$ expected time and answer point location queries in $O(\log n)$ time. Similarly, given a set of n points in R^d , one can build a search structure of $O(n^{d+\epsilon})$ size in $O(n^{d+\epsilon})$ expected time and answer simplex range queries in $O(\log^{d+1} n)$ time.

As in the case of a two-dimensional point location structure (Section 5.2.1), there is a trade-off here between the cost of building a search structure and the query time. To achieve a smaller ϵ one has to use a larger sample size. This in turn increases the constant factor within the query time bound.

6.5.1 Dynamic maintenance

Next, we dynamize the preceding search structure based on top-down sampling. First, we shall consider dynamization of the point location structure. Dynamization of the augmented search structure for range queries is obtained by straightforward recursive considerations. The dynamization scheme in this section, which will also be called dynamic sampling, can be used for any search structure based on top-down sampling. The top-down search structures that we have encountered so far were based on random samples of constant size. If the sample size is not constant, the dynamization scheme becomes a bit more complex. This will be treated in the exercises.

Static setting

Let N denote the given set of hyperplanes as before. The top-down point location structure described before is difficult to dynamize as it is. Hence, we shall first describe another similar but more flexible static search structure. The flexibility of this search structure will make it easy to dynamize. We shall denote this search structure by $\text{sample}(N)$.

We assume that we are given a coin with some fixed bias $p < 1$. By bias, we mean the probability of success. We shall choose this bias appropriately later. Starting with the set N , we obtain a gradation: $N = N_1 \supseteq N_2 \supseteq$

$\cdots \supseteq N_r \supseteq N_{r+1} = \emptyset$. Here N_{i+1} is obtained from N_i by flipping the given coin independently for each hyperplane in N_i and retaining only those hyperplanes for which the toss was a success (heads). The length of this gradation is $\tilde{O}(\log_{1/p} n) = \tilde{O}(\log n)$. This follows by the Chernoff bound just as in the case of skip lists (Section 1.4).

The state of $\text{sample}(N)$ at any time is going to be completely determined by the gradation of N . This feature is also shared by the search structures based on bottom-up sampling. The difference now is that we proceed in a top-down fashion. At the root level r , we store $H(N_r)$, a canonical triangulation of the arrangement formed by N_r . We also store at the root the full arrangement $G(N)$ formed by N . We associate with each simplex $\Delta \in H(N_r)$ the list $N(\Delta)$ of the hyperplanes in $N \setminus N_r$ intersecting it. The gradation on N gives rise to a natural gradation on $N(\Delta)$ of smaller length:

$$N(\Delta) = N(\Delta)_1 \supseteq N(\Delta)_2 \subseteq \cdots \supseteq N(\Delta)_{r-1} \supseteq N(\Delta)_r = \emptyset,$$

where $N(\Delta)_i = N(\Delta) \cap N_{i+1}$. For each simplex Δ of $H(N_r)$, we recursively build $\text{sample}(N(\Delta))$. This construction is based on the inherited gradation of $N(\Delta)$. It is also restricted within Δ . We store the restricted arrangement $G(N(\Delta)) \cap \Delta$ at the node for each Δ . Each cell in this restricted arrangement is given a *parent pointer* to the unique cell containing it in $G(N)$, which is stored at the root level.

Queries are answered exactly as before. Given a query point q , we first locate the simplex Δ in $H(N_r)$ containing it. This can be trivially done in $O(n_r)$ time (Exercise 6.4.3). We recursively locate the cell in $G(N(\Delta)) \cap \Delta$ containing q . The parent pointer associated with this cell tells us the cell in $G(N)$ containing q .

Analysis

In what follows, we shall denote the size of N by n , the size of N_i by n_i , and the size of $N(\Delta)$ by $n(\Delta)$.

Lemma 6.5.2 *Sample(N) can be constructed in expected $O(n^{d+\epsilon})$ time and space, where $\epsilon > 0$ can be made arbitrarily small by choosing the bias p small enough. For a fixed query point q , the cost of locating q is $\tilde{O}(\log n)$.*

Remark. As usual, there is a trade-off here between the cost of building a search structure and the query time. To achieve a smaller ϵ , one has to use a smaller bias p . This in turn increases the constant factor within the query time bound. The constant factor is, in fact, $1/p$.

Proof. First, let us estimate the query cost. Let q be a fixed query point. Let $Q(n)$ denote the cost of locating q in $G(N)$. Then we have

$$Q(n) = O(n_r) + Q(n(\Delta)),$$

where Δ denotes the d -simplex in $H(N_r)$ containing q . Note that the coin flip for each hyperplane in N_r was a failure in the grading process. That is why the $(r + 1)$ th level is empty. This means that n_r is bounded by a geometric random variable with parameter p (Appendix A.1.2). This random variable is equal to the number of failures obtained before success in a sequence of Bernoulli trials with a coin of bias p . Its expected value is $1/p$. The coin tosses at any level of our data structure are independent of the coin tosses at the previous levels. Moreover, the number of levels in $\text{sample}(N)$ is $\tilde{O}(\log_{1/p} n)$. Hence, $Q(n)$ is bounded by a sum of $\tilde{O}(\log_{1/p} n)$ independent geometric variables. The $\tilde{O}(\log n)$ bound on the query cost follows from the Chernoff bound for this sum (Appendix A.1.2). Note that this bound is for a fixed query point q . The maximum query cost over every choice of the query point is analyzed in the exercises.

Let us turn to the expected cost of constructing $\text{sample}(N)$. For a fixed integer $s > 0$, let $T_s(n)$ denote the *clipped* cost of building $\text{sample}(N)$. This ignores the cost incurred at depths higher than s . In other words, we only take into account the cost incurred up to depth s in the recursive definition of $\text{sample}(N)$. As we already remarked, the depth of $\text{sample}(N)$ is $\tilde{O}(\log_{1/p} n)$. Hence the bound on the expected cost of building $\text{sample}(N)$ follows from the following lemma, wherein we let s be a large enough multiple of $\log_{1/p} n$ and choose p small enough, so that $1/p$ is much larger than b —in that case the factor b^s below can be made less than n^ϵ , for any $\epsilon > 0$. \square

Lemma 6.5.3 $E[T_s(n)]$, the expected value of $T_s(n)$, is $\leq n^d b^s$, for some constant $b > 0$ which depends only on the dimension d .

Proof. We shall proceed by induction on the depth s .

The canonical triangulation $H(N_r)$ stored at the root can be built in $O(n_r^d)$ time: First, we build the arrangement formed by N_r incrementally (Section 6.1) and then triangulate each cell in this arrangement in time proportional to its size. The conflict lists of all simplices in $H(N_r)$ can be built in $O(n_r^{d-1}n)$ time (Exercise 6.3.2). (Actually the trivial $O(n_r^d n)$ bound is also sufficient for our purposes, because the expected value of n_r is a constant. This latter bound follows because the conflict list of each simplex in $H(N_r)$ can be trivially built in $O(n)$ time.) We also spend $O(n^d)$ time in building the arrangement $G(N)$ stored at the root. What remains is the cost of building the recursively defined search structures $\text{sample}(N(\Delta))$, for every $\Delta \in H(N_r)$.

Thus, the total clipped cost $T_s(n)$ satisfies the following recurrence equation:

$$T_0(n) = O(1),$$

and, for $s > 0$,

$$T_s(n) = O(n^d) + \sum_{\Delta \in H(N_r)} T_{s-1}(n(\Delta)). \quad (6.10)$$

By the induction hypothesis, it follows that

$$E[T_s(n)] = O(n^d) + b^{s-1} E \left[\sum_{\Delta \in H(N_r)} n(\Delta)^d \right].$$

Here the expectation is based on the assumption that N_r is a random sample of N for any fixed choice of n_r . It is crucial that our induction works for every choice of n_r . Now notice that the “average” conflict size of the simplices in $H(N_r)$ is $O(n/n_r)$ (Section 5.5), and the size of $H(N_r)$ is $O(n_r^d)$, for any $N_r \subseteq N$. Hence

$$E \left[\sum_{\Delta \in H(N_r)} n(\Delta)^d \right] = O \left((n/n_r)^d n_r^d \right) = O(n^d).$$

More formally, this follows by applying Theorem 5.5.5 to the configuration space of feasible simplices over N described in Section 6.3.

If we choose b large enough, the lemma follows from the last two equations. □

It is quite illuminating to compare this static search structure with the top-down search structure defined previously. The overall organization of both data structures is the same. There is one crucial difference. The “outdegree” of each node in the previous data structure is bounded. In contrast, the outdegree of each node in the new data structure is a geometric random variable. This flexibility in the outdegree turns out to be essential in dynamization. The difference between two data structures is akin to the difference between a statically balanced binary tree and a dynamically balanced tree, such as a skip list or, say, an AVL tree (in case the reader is familiar with it). The outdegree of each node in a statically balanced binary tree is fixed. In contrast, the outdegree of a node in a skip list or an AVL tree is flexible: It is distributed according to the geometric distribution in the former case, and is either two or three in the latter case. Such flexibility makes updates easy.

Updates

Let us now turn to the updates.

Addition of a new hyperplane S to sample(N) is carried out as follows. We flip our given coin repeatedly until we get failure. Let j be the number of

successes before obtaining failure. We add S to the sets N_1, \dots, N_{j+1} . This gives us a new gradation of $N' = N \cup \{S\}$:

$$N' = N'_1 \supseteq N'_2 \supseteq \dots,$$

where $N'_i = N_i \cup \{S\}$, for $1 \leq i \leq j+1$, and $N'_i = N_i$, for $i > j+1$. Our goal is to update $\text{sample}(N)$ so as to obtain $\text{sample}(N')$ with respect to this gradation. This is done as follows. If $j+1 \geq r$, we construct $\text{sample}(N')$ with respect to the above gradation from scratch. This can be done using the static scheme given before (Lemma 6.5.2). Otherwise, we recursively add S to $\text{sample}(N(\Delta))$, for each $\Delta \in H(N_r)$ intersecting S .

Deletion of S from $\text{sample}(N)$ is the reverse operation. Let $N' = N \setminus \{S\}$. The grading of N' is naturally inherited from the grading of N . If $S \in N_r$, we build $\text{sample}(N')$ from scratch using the static scheme (Lemma 6.5.2). Otherwise, we recursively delete S from $\text{sample}(N(\Delta))$, for each $\Delta \in H(N_r)$ intersecting S .

Analysis

Let us now estimate the expected cost of an update. Here the expectation is meant to be solely with respect to randomization in the search structure. We shall only analyze deletions, because additions can be handled similarly. For a fixed integer $s > 0$, let $T_s(n)$ denote the *clipped* cost of updating $\text{sample}(N)$. This cost ignores the cost incurred at depths higher than s . In other words, we only take into account the cost incurred up to depth s .

The whole search structure is rebuilt from scratch if the deleted hyperplane belongs to the last level r . Because of the randomized nature of our construction, N_r is a random sample of N , and hence, this happens with probability n_r/n , for a fixed n_r . We have already remarked that n_r follows a geometric distribution. Hence, the probability of the above event is $O(1/n)$. If the event occurs, the expected cost of rebuilding is $O(n^{d+\epsilon'})$ (Lemma 6.5.2). The coin flips used in the gradation of N are independent. Hence, the expected value of $T_s(n)$ satisfies the following probabilistic recurrence:

$$T_0(n) = O(1),$$

and

$$E[T_s(n)] = O\left(\frac{1}{n} n^{d+\epsilon'}\right) + E\left[\sum_{\Delta} T_{s-1}(n(\Delta))\right],$$

where Δ ranges over the simplices in $H(N_r)$ intersecting S . Here the expectation is based on the assumption that N_r is a random sample of N . This equation can be solved by induction on the depth s just as we solved (6.10). We solved (6.10) by applying the average conflict size result (Theorem 5.5.5) to the configuration space of feasible simplices over N . Now the result is

applied to the subspace of feasible simplices intersecting S . In this application, we note that, by the Zone Theorem, the number of simplices in $H(N_r)$ intersecting S is $O(n_r^{d-1})$, for any $N_r \subseteq N$. Proceeding as in the case of (6.10), it then follows that

$$E[T_s(n)] \leq n^{d-1+\epsilon'} b^s,$$

for some constant $b > 0$ that depends only on the dimension d . The depth of our search structure is $\tilde{O}(\log_{1/p} n)$. By choosing p small enough, it follows that the expected cost of deletion can be made $O(n^{d-1+\epsilon})$, for some $\epsilon > \epsilon'$. We can make ϵ arbitrarily small, by making ϵ' correspondingly small.

Addition can be analyzed in exactly the same fashion. The preceding discussion can be summarized as follows.

Theorem 6.5.4 *One can maintain a point location structure with the following performance: $O(n^{d+\epsilon})$ expected space requirement, $\tilde{O}(\log n)$ query cost and $O(n^{d-1+\epsilon})$ expected cost of addition or deletion. The expectation is solely with respect to randomization in the search structure.*

The previous search structure can be extended to handle simplex range queries in a dual setting. This is similar to the extension of the previous static point location structure. It can be done by a *verbatim* translation. So we only state:

Theorem 6.5.5 *One can maintain a search structure for answering simplex range queries with the following performance: $O(n^{d+\epsilon})$ expected space requirement, $\tilde{O}(\log^{d+1} n)$ query cost and $O(n^{d-1+\epsilon})$ expected cost of addition or deletion. The expectation is solely with respect to randomization in the search structure.*

Exercises

6.5.1 Show that the maximum point location cost using $\text{sample}(N)$, over all $q \in R^d$, is $\tilde{O}(\log n \log \log n)$. (Hint: Show that the number of distinct search paths is $O(n^{O(\log \log n)})$.)

****6.5.2** (Variable bias in sampling) The top-down dynamic sampling scheme given in this section is based on a coin with a fixed bias $p > 0$. The set N_r associated with the root level of $\text{sample}(N)$ is a random sample of N of expected $O(1/p)$ size. The following variation of this dynamic sampling scheme is sometimes useful. Fix an appropriate function $f(n)$. Let the set N_r associated with the root level of $\text{sample}(N)$ be a random sample of N of expected size $O(f(n))$. It can be obtained from N by flipping a coin of variable bias $p = \lfloor \log_2(f(n)/n) \rfloor$. An analogous invariant, based on the function f , is recursively maintained at every node of the search structure.

Show how this scheme can be applied to bring down the expected space requirement of the search structures in this section to $O(n^d \text{polylog}(n))$. (Hint: Choose

$f(n) = n^\alpha$, for an appropriate $\alpha > 0$. Now you will also need a search structure for $H(N_r)$, because the expected size of N_r is not constant. Use the fixed bias search structure in this section for this purpose.)

Describe how to maintain such variable bias search structures in a dynamic fashion. If the bias associated with a node changes due to the change in the number of hyperplanes conflicting with that node, the whole search structure rooted at the node is built from scratch. Using this scheme, show how the expected update cost in this section can be brought down to $O(n^{d-1} \text{polylog}(n))$.

Now bring down the expected space requirement to $O(n^d)$ and the expected (amortized) cost of update to $O(n^{d-1})$. (Hint: Bootstrap.)

Formulate this variable-bias dynamic sampling scheme in a general setting of configuration spaces.

Bibliographic notes

Arrangements have been studied in combinatorial geometry for a long time; a good historical survey is provided by Grünbaum [116]. The incremental construction of arrangements (Section 6.1) is due to Edelsbrunner, O'Rourke, and Seidel [92]. The Zone Theorem for arrangements was proved by Edelsbrunner, Seidel and Sharir [94], which Section 6.2 closely follows. The near-optimal bound on the sum of the squares of the cell-sizes (Exercises 6.2.4 and 6.2.5) is due to Aronov, Matoušek, and Sharir [12]. A bound for the zone complexity of a hypersurface (Exercise 6.2.6) was proved by Aronov and Sharir [13, 12].

Canonical triangulations were proposed by Clarkson [66]. The Cutting Lemma (Section 6.3.1) is due to Chazelle and Friedman [48]. The static point location structure for arrangements (Section 6.5), based on top-down sampling, is due to Clarkson [66], and its extension to range search is due to Chazelle, Sharir, and Welzl [56]. Dynamization of this search structure (Section 6.5.1), using (top-down) dynamic sampling, was done by Mulmuley [167]. The dynamic search structure for point location and ray shooting (Section 6.4), based on (bottom-up) dynamic sampling, is due to Mulmuley and Sen [172]; this paper also gives a dynamic search structure with optimal $\tilde{O}(\log n)$ query time, $O(n^d)$ expected space, and $O(n^{d-1})$ expected (amortized) cost of update (Exercise 6.5.2). Agarwal and Matoušek [3] give a deterministic, dynamic search structure for ray shooting in arrangements, which requires somewhat more space and preprocessing time (by a factor of n^ϵ) and polylogarithmic query time. They also give space-time trade-offs. Our terminology “antenna” is borrowed from [47], where antennas were used earlier, differently, for static point location. Chazelle [40] gives a static, deterministic point location structure for arrangements with optimal $O(\log n)$ query time, and $O(n^d)$ space and preprocessing time.

Chapter 7

Convex polytopes

We have extensively studied three-dimensional convex polytopes in the earlier chapters. In this chapter, we want to study convex polytopes in arbitrary dimension. A convex polytope in R^d is a nonempty region that can be obtained by intersecting a finite set of closed half-spaces. Each half-space is defined as the solution set of a linear inequality of the form $a_1x_1 + \cdots + a_dx_d \geq a_0$, where each a_j is a constant, the x_j 's denote the coordinates in R^d , and a_1, \dots, a_d are not all zero. The boundary of this half-space is the hyperplane defined by $a_1x_1 + \cdots + a_dx_d = a_0$. We denote the bounding hyperplane of a half-space M by ∂M .

A convex polytope is indeed convex. This is not a tautology. What it formally means is that a segment joining any two points in the polytope completely lies within it. This follows because half-spaces are convex and the intersection of convex sets is convex.

Let $P = \cap_i M_i$ be any convex polytope in R^d , where each M_i is a half-space. It is possible that P lies completely within some hyperplane ∂M_i . This can happen, for example, when the set $\{M_i\}$ contains two half-spaces lying on the opposite sides of a common bounding hyperplane. In this case, P can be thought of as a convex polytope of lower dimension within the containing hyperplane. Otherwise, P is a full d -dimensional convex polytope in R^d . A d -dimensional convex polytope is also called a d -polytope. In what follows, we assume that P is a d -polytope in R^d .

A half-space M_i is called *redundant* if it can be thrown away without affecting P (Figure 7.1). This means the intersection of the remaining half-spaces is also P . Otherwise, the half-space is called *nonredundant*. The hyperplanes bounding nonredundant half-spaces are said to be the *bounding* hyperplanes of P .

A $(d - 1)$ -face or a *facet* of P is defined to be the intersection of P with one of its bounding hyperplanes. Each facet of P is a $(d - 1)$ -dimensional

convex polytope. We define the j -faces of P , where $j < d - 1$, inductively: A $(d - 2)$ -face of P is a $(d - 2)$ -face of a facet of P , and so on. Each j -face of P is a j -dimensional convex polytope. A 0-face is also called a vertex and a 1-face is also called an edge. As a convention, the d -face of P is P itself. We say that a j -face f is adjacent to an i -face g , where $j < i$, if f is contained in the boundary of g . In this case, we also say that f is a subface of g . The total number of subfaces of g is denoted by $|g|$. It is also called the *size* of g .

We have already encountered convex polytopes in Chapter 6, where they occurred as the faces of arrangements. In this chapter, we want to study a convex polytope as an object of independent interest.

As far as representing a convex polytope P is concerned, we can use the facial lattice representation, just as in the case of arrangements. The facial lattice of P contains a node for each face of P . Two nodes are joined by an adjacency edge iff the corresponding faces are adjacent and their dimensions differ by one. We group the adjacency edges incident to every fixed node in two classes: (1) the ones that correspond to adjacencies with the faces of higher dimensions, and (2) the ones that correspond to adjacencies with the faces of lower dimensions. The facial lattice allows us to determine the faces of higher or lower dimensions adjacent to any given face in time proportional to their number. In algorithmic applications, it is also convenient to associate with each node some auxiliary information, such as the list of hyperplanes containing the corresponding face; the coordinates, in case the node corresponds to a vertex; and so on.

For $j \leq d$, we define the j -skeleton of P to be the collection of its i -faces, for all $i \leq j$, together with the adjacencies among them. Thus, the j -skeleton can be thought of as a sublattice of the facial lattice of P . An important special case is the 1-skeleton of P . It is also called the *edge skeleton*.

The polytope P is said to be *simple* if every j -face of P is contained in exactly $d - j$ bounding hyperplanes (Figure 2.11). Every j -face of a simple d -polytope is adjacent to exactly $\binom{d-j}{i-j}$ i -faces (Exercise 6.0.2).

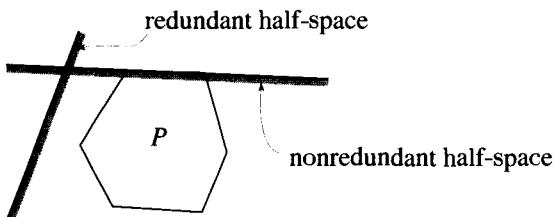


Figure 7.1: Redundant and nonredundant half-spaces.

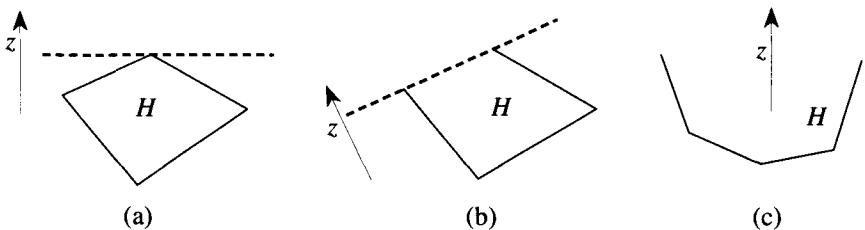


Figure 7.2: Objective function on a convex polytope: (a) Nondegenerate case. (b) Degenerate case. (c) Unbounded case.

Exercises

7.0.1 Suppose P is a nonsimple polytope. Show that almost all infinitesimal perturbations of its bounding hyperplanes yield a simple polytope. Show that an infinitesimal perturbation cannot decrease the number of j -faces, for any j . (Hint: Translate a hyperplane, infinitesimally, one at a time. Show that if the translation is sufficiently small, then the number of j -faces cannot decrease.)

7.0.2 Show that the edge skeleton of a bounded convex polytope is a connected graph. (Hint: Use induction on dimension.)

7.0.3 Given the j -skeleton of a bounded convex polytope, for $j > 0$, show that the $(j + 1)$ -skeleton can be constructed in time proportional to its size. Assume that each node in the j -skeleton is associated with the set of hyperplanes containing the corresponding face.

7.0.4 Let P be a d -polytope. Suppose P lies completely on one side of a hyperplane h . Show that $P \cap h$, if nonempty, is a face of P . In this case, we say that h supports the face $P \cap h$.

7.1 Linear programming

A natural and frequently studied problem concerning convex polytopes is the so-called linear programming problem. We are given a set N of half-spaces in \mathbb{R}^d . The half-spaces are called linear constraints in this context. In addition, we are given a linear function z . It is also called an objective function. Let $H(N)$ denote the convex polytope formed by intersecting the half-spaces in N . The goal is to determine the maximum value of the objective function on $H(N)$ and also the coordinates of a point where the maximum is achieved.

For example, the coordinates x_1, \dots, x_d could stand for the production quantities of various goods produced in a factory. Each linear constraint in N could stand for a production constraint. The objective function $z = b_1x_1 + \dots + b_dx_d$ could stand for the total profit, where b_i stands for the profit over one unit of the i th good. The goal is to determine the production quantities x_i so that the profit is maximized.

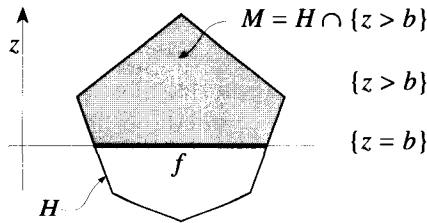


Figure 7.3: Uniqueness of a maximum value.

A linear function z need not always have a maximum on the convex polytope. This happens if it is unbounded in the z -direction (Figure 7.2(c)). In general,

Proposition 7.1.1 *A linear function z can attain at most one maximum value (local or global) over any polytope H . The set of points where this maximum (if any) is achieved forms a closed face of H .*

Proof. If H is unbounded in the z -direction, then it achieves no maximum on H (Figure 7.2(c)). If z is a constant function, then it achieves maximum on the whole of H . So in what follows, let us assume that z is a nonconstant linear function that achieves at least one local maximum value over H . Let b be any such value. We claim that $M = \{z > b\} \cap H$ is empty. If not, M (or more precisely, its closure) is a nonempty convex polytope. The set $\{z = b\} \cap H$ forms a face of this polytope. We can always move from any point on this face towards the interior of M (Figure 7.3), thereby increasing the value of z ; a contradiction.

Thus, $\{z > b\} \cap H$ is empty for any local maximum value b . It follows that there can be only one local maximum value (why?). Moreover, the polytope H is completely contained in the half-space $\{z \leq b\}$ for this maximum b . Hence, $\{z = b\} \cap H$, the set of points where this maximum is achieved, forms a face of H of dimension ≥ 0 (Exercise 7.0.4). \square

An important special case arises if the linear function z is *nondegenerate*. This means that z is not constant over any face of H of nonzero dimension (Figure 7.2). We leave it to the reader to check that almost all linear functions are nondegenerate. This means that if we change the direction of z infinitesimally, then almost all such perturbations make z nondegenerate (Exercise 7.1.3).

The previous proposition immediately implies the following.

Corollary 7.1.2 *If z is nondegenerate, then its (local or global) maximum on H , if any, is a unique vertex of H .*

Our goal in this section is to give an efficient algorithm for the following linear programming problem: Given a set N of n half-spaces in R^d and a linear function z , find the coordinates of a z -maximum (if any) on the polytope $H(N)$ formed by intersecting these half-spaces. Each half-space is specified by a linear inequality. We allow redundant half-spaces, because their detection is itself a nontrivial problem. We do not even assume that $H(N)$ is nonempty. Determining if $H(N)$ is empty or not is also nontrivial.

In the linear programming problem, the convex polytope $H(N)$ is not specified by the facial lattice, but only by the set of defining half-spaces. This is because we are only interested in computing the coordinates of a point where the maximum is achieved. Computing and storing all faces of $H(N)$ is computationally expensive and unnecessary.

We shall give a linear programming algorithm with $O(n)$ expected running time. The constant factor within the Big-Oh notation is proportional to $d!$. We are assuming here, as in most computational geometric applications, that the dimension d is a small fixed constant. This is not a reasonable assumption in all linear programming applications. Quite often the number of constraints and the dimension d are comparable. There are linear programming algorithms whose running times are polynomial in d , n , and the number of bits in the binary representation of the input. These methods are beyond the scope of this book because they employ very different techniques. In the exercises, we shall give alternative linear programming algorithms that achieve better dependence on d .

Our algorithm in this section will be randomized and incremental. We add the n half-spaces in the given input set, one at a time, in random order. Let N^i denote the set of the first i added half-spaces. Let $H(N^i)$ be their intersection. The algorithm proceeds as follows: We first compute the z -maximum v_d on $H(N^d)$. Then, for $d \leq i \leq n$, we successively compute the z -maximum v_i on the polytope $H(N^i)$. The maximum v_n on $H(N) = H(N^n)$ is the required answer.

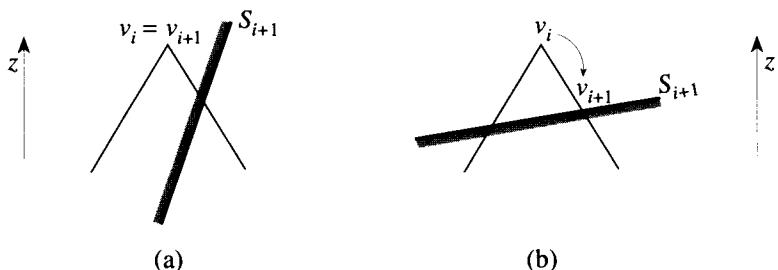


Figure 7.4: Change in the z -maximum during addition.

We are ignoring here the possibility that for a given i , v_i need not exist—this may happen if $H(N^i)$ is empty or unbounded in the z -direction—and even if v_i exists, it may not be unique, if z is degenerate over $H(N^i)$. For a moment, though, let us make a simplifying assumption that v_i always exists, that it is unique, and that it is contained in the bounding hyperplanes of exactly d half-spaces. We shall remove these assumptions later.

The addition of the $(i + 1)$ th half-space $S = S_{i+1}$ is achieved as follows (Figure 7.4). If v_i is contained in S , then obviously $v_{i+1} = v_i$, and nothing more needs to be done (Figure 7.4(a)). Otherwise, v_{i+1} must be contained in the hyperplane ∂S bounding S (Figure 7.4(b)). In this case, we compute the z -maximum on the lower-dimensional polytope formed by intersecting the set $\{\partial S \cap S_j \mid j \leq i\}$ of half-spaces within ∂S . This is done recursively by applying our procedure in dimension $d - 1$. The basis case $d = 1$ in the recursion is trivial: The half-spaces in this case are simply half lines and their intersection is a segment, which is trivial to compute in $O(n)$ time.

Analysis

We claim that the expected running time of the above algorithm is $O(d!n)$. The proof is by induction on d . The basis case $d = 1$ is trivial. For $d > 1$, it suffices to show that, for each i , the expected time required to compute v_{i+1} from v_i is $O(d!)$. If $v_i = v_{i+1}$, this is trivial. The other case occurs iff one of the d half-spaces whose bounding hyperplanes contain v_{i+1} was added last, i.e., if it was S_{i+1} . Because all orders of additions of the half-spaces are equally likely, this happens with probability $d/i + 1$. This is best seen by backward analysis: For any fixed N^{i+1} , every half-space in N^{i+1} is equally likely to be S_{i+1} . Moreover, exactly d half-spaces in N^{i+1} contain v_{i+1} . Hence, the claimed expression for the probability follows. By our inductive assumption, the expected cost of solving the resulting $(d - 1)$ -dimensional linear programming problem with i constraints is $O((d - 1)!i)$. Thus, the expected cost of obtaining v_{i+1} from v_i is $O((d - 1)!id/(i + 1)) = O(d!)$, as claimed.

It remains to relax the various simplifying assumptions made at the beginning of the algorithm. We can ensure that each $H(N^i)$ is bounded by using the bounding box trick in Section 3.2. This consists in adding to N^0 half-spaces bounding a huge symbolic cube approaching infinity. Emptiness of some $H(N^i)$ is not a problem: This is detected by the algorithm at the bottom level $d = 1$ of the recursion. If $H(N^i)$ is empty, $H(N)$ is also empty, so there is no need to proceed any further. The other simplifying assumptions are removed in the exercises.

Exercises

7.1.1 Give modifications required to handle the possibility that the optimum v_i on $H(N^i)$ may not be unique. (Hint: Enforce uniqueness by stipulating that v_i be the optimal vertex of $H(N^i)$ with lexicographically largest coordinate representation. Show that the previous analysis of the algorithm remains valid, because v_i is defined uniquely and depends only on $H(N^i)$. This means it does not depend on the actual order in which the half-spaces in N^i were added.)

7.1.2 Show that the assumption that each v_i is contained in the boundaries of exactly d half-spaces can be dropped altogether without any further change in the algorithm. (Hint: If v_{i+1} is contained in the boundaries of more than d half-spaces then show that the probability that v_i is different from v_{i+1} is less than the previously used bound $d/(i+1)$. This only decreases the expected running time of the algorithm.)

7.1.3 Let P be a fixed convex polytope. Let z be any linear function. Show that almost all infinitesimal perturbations of z produce a linear function that is nondegenerate over P .

7.1.4 Show how the linear programming algorithm in this section can be used to detect whether a given half-space in N is redundant in $O(n)$ expected time. Thus, all redundant half-spaces in a given set of n half-spaces can be detected in $O(n^2)$ expected time.

Can you do better for the latter problem? How much? (Currently, it is possible to achieve $O(n^{2-2/(1+\lfloor d/2 \rfloor)} + \epsilon)$ bound, for any $\epsilon > 0$; the constant factor depends on ϵ .)

****7.1.5** Give a linear programming algorithm for which the constant in the expected time bound depends subexponentially on d . (Hint: Modify the algorithm in this section as follows. Consider the addition of $S = S_{i+1}$. Let h_1, \dots, h_d be the half-spaces in N^i defining the current optimum v_i . If S does not contain v_i , we compute the optimum v_{i+1} on $H(N^{i+1})$ recursively. During this recursion, we first add the d half-spaces defining the optimum on $S \cap h_1 \cap \dots \cap h_d$. The reason behind this is that this optimum can be computed from v_i in $O(d^2)$ time. (How?) The expected running time of this modified algorithm is $O(nd \exp[4\sqrt{d \ln(n+1)}])$.)

***7.1.6** In this exercise, we give a different linear programming algorithm based on random sampling. Let N be a set of n half-spaces in R^d . Let z be the objective function. Enclose everything within a huge symbolic box as usual.

(a) Take a random sample $R_1 \subseteq N$ of size $cd\sqrt{n}$, for a large enough constant $c > 0$.

Let v be the z -optimum for the half-spaces in this random sample; it can be found recursively. Let V be the set of half-spaces in $N \setminus R_1$ conflicting with v . Show that the expected size of V is bounded by \sqrt{n} , if c is large enough. (Hint: Use the results in Section 5.5.) This, in conjunction with Markov's inequality (Appendix A), implies that the size of V is less than $2\sqrt{n}$ with probability at least $1/2$. If the size of V exceeds this bound, repeat the experiment. On the average, two trials are needed before the property under consideration is satisfied.

- (b) Show that V must contain at least one constraint defining the optimum for N . Now take a similar random sample R_2 from the remaining set $N \setminus R_1$. Repeat this phase d times. The resulting sample $R = R_1 \cup \dots \cup R_d$ contains all constraints defining the optimum on N . Thus, the optimum on R coincides with the optimum on N . The size of R is $O(d^2\sqrt{n})$. Find the optimum on R recursively.
- (c) As the basis case, use the algorithm in this section, when the number of constraints drops below ad^2 , for a suitable constant a .
- (d) Analyze the algorithm carefully. Show that its expected running time is $O(d^2n + (\log[n/d^2])^{\log d+2}d!d^2)$, where the constant factor does not depend on d . The second term arises from the use of the incremental algorithm in this section on $(\log[n/d^2])^{\log d+2}$ “small” problems with $O(d^2)$ constraints.
- (e) Alternatively, use the algorithm in Exercise 7.1.5 for the basis case. This brings down the $d!d^2$ factor in the second term to a factor that is exponential in $\sqrt{d}\log d$.

7.2 The number of faces

Let P be a d -polytope bounded by n hyperplanes. Let $f(j)$ denote the number of j -faces of P . The vector $[f(0), \dots, f(j), \dots, f(d)]$ is called the *face vector* or simply the f -vector of P . By convention, we let $f(d) = 1$. This is natural because P has only one d -face, namely, P itself. We are interested in obtaining an upper bound on $f(j)$ in terms of n and j . Without loss of generality, we can assume that P is bounded. Otherwise, we add $2d$ additional hyperplanes to N bounding a large box approaching infinity. We then consider instead the intersection of P with this box. This procedure can only increase $f(j)$, for any j . Similarly, we can also assume that P is simple. This is because almost all infinitesimal perturbations of the bounding hyperplanes transform P into a simple polytope. Moreover, $f(j)$, for any j , cannot decrease under such infinitesimal perturbation (Exercise 7.0.1).

So let us assume that P is simple and bounded. Each j -face of P is contained in a unique set of $d - j$ bounding hyperplanes. This implies that $f(j) \leq \binom{n}{d-j}$. This turns out to be a good upper bound when $j > \lfloor d/2 \rfloor$. Otherwise, it is far too unsatisfactory. For example, when $d = 3$, this implies $O(n^2)$ bound on $f(1)$, the number of edges. This is far worse than the $O(n)$ bound that we have already shown in this case (Section 2.4.1). The reader may recall that this followed from Euler’s relation for three-dimensional polytopes. In this section, we shall prove Euler’s relation and its generalization, the so-called Dehn–Sommerville relations, for convex polytopes in arbitrary dimension. The latter relations will immediately imply a tight asymptotic upper bound on $f(j)$, for every j .

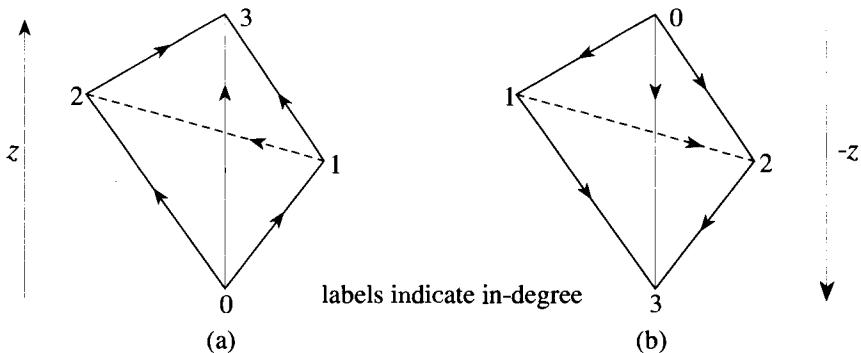


Figure 7.5: (a) In-degree with respect to a linear function z . (b) In-degree with respect to $-z$.

7.2.1 Dehn–Sommerville relations

Let P be a simple, bounded d -polytope. Let n be the number of hyperplanes bounding P . Choose any nondegenerate linear function z . This means z is not constant on any edge of P . Almost all choices of z have this property (Exercise 7.1.3). Orient all edges of P in the increasing z -direction. Let v be any vertex of P . The *in-degree* of v with respect to the linear function z is defined to be the number of edges of P oriented towards v (Figure 7.5). The *out-degree* of v is defined to be the number of edges of P oriented away from v . The out-degree of v with respect to z is equal to the in-degree of v with respect to $-z$ (Figure 7.5). Since P is simple, the sum of the in-degree and the out-degree of a vertex is always d .

The following proposition follows from Corollary 7.1.2.

Proposition 7.2.1 *Let P be a simple, bounded d -polytope. A vertex v of P has in-degree 0 (resp. d) iff it is the z -minimum (resp. z -maximum) of P . Such a vertex exists, because P is bounded, and is unique.*

Now, let $h_z(i)$, $0 \leq i \leq d$, denote the number of vertices of P with in-degree i with respect to z . If the coordinate z is implicit in the context, we shall denote this number by simply $h(i)$. By the above proposition, $h(0) = h(d) = 1$. The vector $[h(0), \dots, h(i), \dots, h(d)]$ is called the *h-vector* of P .

We have defined the *h*-vector above for a particular choice of z . But we shall soon see that it depends solely on the polytope P and not on the choice of z . For this, we first determine how the *f*-vector and the *h*-vector of P are related.

For each i -face of P , let us place one token marked i at the z -maximum on that face. This maximum exists because the i -face is bounded. It is unique because z is nondegenerate (Corollary 7.1.2).

Fix any vertex $v \in P$. Let $f_v(i)$ denote the number of tokens at v marked i . Note that

$$f(i) = \sum_v f_v(i), \quad (7.1)$$

where v ranges over all vertices of P . In other words, the total number of tokens marked i over all vertices of P is $f(i)$. This is because there is precisely one token per i -face. The f_v -vector $[f_v(0), \dots, f_v(d)]$ can be thought of as the contribution of v to the f -vector of P . Figure 7.6 gives an example. There, each vertex v is labeled with its f_v -vector.

Claim: If v is a vertex of P with in-degree j , then $f_v(i)$ is $\binom{j}{i}$, if $j \geq i$, and 0 otherwise.

This follows from the following observation. Let g be any i -face of P adjacent to v . If v is the z -maximum on g , then all i edges of g adjacent to v are oriented towards v ; in particular $i \leq j$. We can identify g with these i edges oriented towards v . Conversely, every set of i edges oriented towards v determines a unique i -face of P adjacent to v (Exercise 6.0.2). Moreover, v must be the z -maximum of this i -face because its in-degree relative to this face is i (Proposition 7.2.1). The claim follows because there are $\binom{j}{i}$ ways of choosing i edges oriented towards v .

It follows from the claim and (7.1) that

$$f(i) = \sum_{j=i}^d \binom{j}{i} h(j), \text{ for } 0 \leq i \leq d. \quad (7.2)$$

These relations can be stated in a succinct form as follows. Let us define the f -polynomial and the h -polynomial of P by

$$f(t) = \sum_{i=0}^d f(i)t^i, \text{ and } h(t) = \sum_{j=0}^d h(j)t^j, \quad (7.3)$$

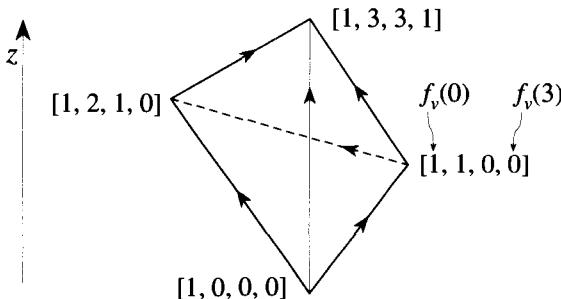


Figure 7.6: Contribution of a vertex to the face vector.

where t is a fixed variable. The relations in (7.2) are equivalent to the following single polynomial equation:

$$f(t) = h(t+1). \quad (7.4)$$

This can be seen from the binomial expansions of both sides. It follows that

$$h(t) = f(t-1). \quad (7.5)$$

If we expand both sides of this equation and equate the coefficients, we get

$$h(i) = \sum_{j=i}^d (-1)^{i+j} \binom{j}{i} f(j), \text{ for } 0 \leq i \leq d. \quad (7.6)$$

This means that the h -vector is completely determined by the f -vector via a linear transformation. The f -vector depends solely on the polytope P . Hence, the h -vector also depends only on P and not on the z -coordinate that was used in its definition. In particular,

$$h_z(i) = h_{-z}(i). \quad (7.7)$$

But notice that the in-degree of a vertex v with respect to $-z$ is the same as the out-degree of v with respect to z (Figure 7.5). Hence,

$$h_{-z}(i) = h_z(d-i).$$

Combining this equation with (7.7), we get

$$h_z(i) = h_z(d-i).$$

Since the h -vector does not depend on the coordinate z , we have proved the following.

Theorem 7.2.2 (Dehn–Sommerville relations)

For $0 \leq i \leq d$, $h(i) = h(d-i)$.

Using the relation between the h -vector and the f -vector, given by (7.6), we get an alternate formulation of the Dehn–Sommerville relations:

Theorem 7.2.3

$$\sum_{j=i}^d (-1)^{i+j} \binom{j}{i} f(j) = \sum_{j=d-i}^d (-1)^{d-i+j} \binom{j}{d-i} f(j), \text{ for } 0 \leq i \leq d.$$

Only $\lfloor d/2 \rfloor$ of these equations are linearly independent. This is obvious from the formulation in Theorem 7.2.2. The Dehn–Sommerville relation, for $i = 0$, is the so-called Euler’s relation:

Corollary 7.2.4 (Euler’s relation) $\sum_{j=0}^d (-1)^j f(j) = 1$.

Note that Euler’s relation is equivalent to the relation, $h(d) = h(0) = 1$, implied by Proposition 7.2.1.

7.2.2 Upper bound theorem: Asymptotic form

We have already observed in the beginning of this section that

$$f(i) \leq \binom{n}{d-i}, \quad (7.8)$$

for each i . This elementary bound turns out to be good for $i > [d/2]$. In Section 7.2.4, we shall give an example of a polytope for which this bound is actually attained. When $i \leq [d/2]$, the bound above is not good. A good bound is obtained by using Dehn–Sommerville relations as follows.

First, notice that $f(i)$ can be expressed as a positive linear combination of $h(j)$'s (see (7.2)). Hence, it suffices to bound $h(j)$, for every j .

Now notice that $h(i) \leq f(i)$. This is because every vertex of P with in-degree i is the z -maximum of a unique i -face of P . The elementary bound in (7.8) now yields

$$h(i) \leq f(i) \leq \binom{n}{d-i}. \quad (7.9)$$

Combining this with the Dehn–Sommerville relation, $h(i) = h(d-i)$, we get

$$h(i) \leq \min \left\{ \binom{n}{d-i}, \binom{n}{i} \right\}. \quad (7.10)$$

Substituting this bound in (7.2) we get

$$f(i) = O \left(\binom{n}{[d/2]} \right),$$

where the constant factor within the Big-Oh notation depends on d .

To summarize:

Theorem 7.2.5 (Upper Bound Theorem (asymptotic form))

$$f(i) = O \left(\min \left\{ \binom{n}{d-i}, \binom{n}{[d/2]} \right\} \right), \text{ for all } 0 \leq i \leq d.$$

7.2.3 Upper bound theorem: Exact form

Actually, $h(i)$ satisfies a tighter bound than the one in (7.10). For a constant d , the tighter bound is better by only a constant factor. Its virtue is that it can actually be attained. We shall show this in Section 7.2.4.

Theorem 7.2.6 (Upper bound theorem)

1. $h(i) \leq \binom{n-d+i-1}{i}$, for $i \leq \lfloor d/2 \rfloor$.
2. $h(i) \leq \binom{n-i-1}{d-i}$, for $i > \lfloor d/2 \rfloor$.

Because of the Dehn–Sommerville relation, $h(i) = h(d-i)$, the second bound in this theorem follows from the first by substituting $d-i$ for i .

Substituting the above bounds in (7.2), we get an alternative form of the upper bound theorem:

Theorem 7.2.7 For each $j \leq d$,

$$f(j) \leq \sum_{i=j}^{\lfloor d/2 \rfloor} \binom{i}{j} \binom{n-d+i-1}{i} + \sum_{i>\lfloor d/2 \rfloor} \binom{i}{j} \binom{n-i-1}{d-i}.$$

The proof of the upper bound theorem depends on the following two lemmas. Let F be any facet of P . It is a $(d-1)$ -dimensional polytope. Let h^F denote the h -vector for this polytope.

Lemma 7.2.8 $h^F(i) \leq h(i)$, for $i \leq d-1$.

Proof. Let z be any nondegenerate linear function. Fix any vertex $v \in F$. Let $\text{in-degree}(v) = \text{in-degree}(v, P)$ denote its usual in-degree with respect to z and P . Let $\text{in-degree}(v, F)$ denote its in-degree relative to F . This is the number of edges of F oriented towards v . Then $h^F(i)$ is the number of vertices of F with in-degree i relative to F . Similarly, $h(i)$ is the number of vertices of P with in-degree i relative to P . We know that both $h^F(i)$ and $h(i)$ do not depend on the linear function z . Hence, the lemma would follow immediately, if we exhibit a linear function z such that

$$\text{in-degree}(v, F) = \text{in-degree}(v, P), \quad \text{for every } v \in F. \quad (7.11)$$

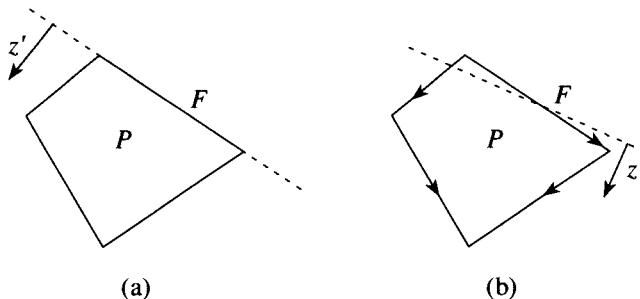


Figure 7.7: Choice of z .

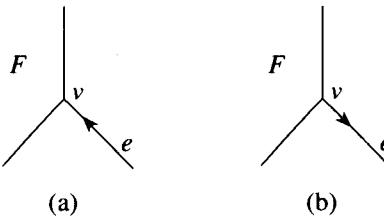


Figure 7.8: Pairing between an edge and a facet.

Let us turn to the choice of z . First, let us choose a linear function z' such that $\{z' = 0\}$ contains F and the half-space $z' \geq 0$ contains P (Figure 7.7(a)). This function is degenerate over P . Hence, let us perturb it infinitesimally so that the resulting linear function z is nondegenerate over P (Figure 7.7(b)).

The linear function z has the following important property: All the edges of P adjacent to F , but not lying within F , are oriented away from F . This immediately implies (7.11). \square

Lemma 7.2.9 For $0 \leq i \leq d - 1$,

$$\sum_F h^F(i) = (d - i)h(i) + (i + 1)h(i + 1),$$

where F ranges over all facets of the polytope P .

Proof. Fix any nondegenerate linear function z . Define $h^F(i)$ and $h(i)$ in terms of z as usual. We shall show that each vertex of P contributes equally to both sides of the above equation.

Fix a vertex $v \in P$. Since P is simple, exactly d facets of P are incident to v . For each such facet F , there is a unique edge e which is adjacent to v , but which is not contained in F (Figure 7.8). Conversely, for each edge e adjacent to v , there is a unique facet F which is adjacent to v but which does not contain e . The edge e could be oriented towards or away from v . We proceed by cases.

1. $\text{in-degree}(v)$ is less than i or greater than $i + 1$: In this case, it is easy to see that $\text{in-degree}(v, F)$ cannot be i , for any F adjacent to v . Thus, v contributes zero to both sides of the equation.
2. $\text{in-degree}(v) = i$: In this case $\text{in-degree}(v, F) = i$ iff the edge e is oriented away from v (Figure 7.8(b)). There are $d - i$ possible choices of e , because $\text{in-degree}(v) = i$. Correspondingly, there are $d - i$ possible choices of F . Thus, the contribution of v to the left-hand side of the equation is $d - i$. Its contribution to the right-hand side is the same. This is because v contributes 1 to $h(i)$ and 0 to $h(i + 1)$.

3. in-degree(v) = $i + 1$: In this case in-degree(v, F) = i iff the edge e is oriented towards v (Figure 7.8(a)). There are $i + 1$ possible ways of choosing e , and hence F . Thus, v contributes $i + 1$ to the left hand side of the equation. Its contribution to the right hand side is also the same. \square

Lemmas 7.2.8 and 7.2.9 imply that

$$(d - i)h(i) + (i + 1)h(i + 1) \leq nh(i).$$

This can be rewritten as

$$h(i + 1) \leq \frac{n - d + i}{i + 1} h(i).$$

The first bound in Theorem 7.2.6 follows by induction on i . For the basis case, we note that $h(0) = 1$. We have already noted that the second bound in the theorem follows from the first by Dehn–Sommerville relations.

7.2.4 Cyclic polytopes

The bounds given in the upper bound theorem are tight in the worst case. In this section, we shall describe a class of convex polytopes for which these bounds are actually attained. These polytopes are called *cyclic polytopes*. They are based on a certain *moment curve* in R^d whose points are parameterized as

$$t \longmapsto x(t) := (t, t^2, \dots, t^d).$$

Let t_1, \dots, t_n be any n distinct real numbers. Let $Q = Q(t_1, \dots, t_n)$ be the convex hull (Section 2.4.1) of the points $x(t_1), \dots, x(t_n)$. It is called a *cyclic polytope*. It depends on the choice of t_1, \dots, t_n . But we shall soon see that the number of its faces does not depend on that choice. We defined a cyclic polytope as a convex hull. If we take its dual (Section 2.4.1), we get a polytope that is defined as an intersection of half-spaces. We shall denote the dual of $Q = Q(t_1, \dots, t_n)$ by $P = P(t_1, \dots, t_n)$. It is called a *dual cyclic polytope*. The important properties of these polytopes are given in the following lemmas.

Lemma 7.2.10 Q is simplicial. Equivalently, P is simple.

Proof. To show that Q is simplicial, it suffices to show that any $d + 1$ of the points $x(t_1), \dots, x(t_n)$ are affinely independent. This means they are not contained in a common hyperplane. Fix any such set of $d + 1$ distinct points. Suppose to the contrary that they are contained in a common hyperplane $h_0 + h_1x_1 + \dots + h_dx_d = 0$. Since each point is of the form (t, \dots, t^d) , this would imply that the polynomial $h(t) = \sum_i h_i t^i$ has $d + 1$ distinct roots. This is a contradiction because the degree of $h(t)$ is d . \square

Lemma 7.2.11 *The number of $(j - 1)$ faces of Q , for $j \leq \lfloor d/2 \rfloor$, is $\binom{n}{j}$. By duality (Section 2.4.1), this is equivalent to saying that the number of i faces of P , for $i > \lfloor d/2 \rfloor$, is $\binom{n}{d-i}$.*

Proof. Fix any $j \leq \lfloor d/2 \rfloor$. It suffices to show that any j of the points $x(t_1), \dots, x(t_n)$ are vertices of a $(j-1)$ -face of Q . Fix any such j points. Without loss of generality, we can assume that these points are $x(t_1), \dots, x(t_j)$, reordering the points if necessary. Consider the hyperplane $p_0 + p_1x_1 + \dots + p_dx_d = 0$ whose coefficients coincide with the coefficients of the polynomial

$$p(t) = p_0 + p_1t + \dots + p_dt^d = \prod_{i=1}^j (t - t_i)^2.$$

For each $i \leq j$, this hyperplane contains the point $x(t_i)$, because $p(t_i) = 0$. The remaining points of Q lie strictly on one side of the hyperplane, because $p(t) > 0$, for every t other than t_1, \dots, t_j . Hence, the points $x(t_1), \dots, x(t_j)$ define a face of Q . This face must be a $(j-1)$ -face because Q is simplicial. \square

The lemma implies that for the dual cyclic polytope P :

$$f(i) = \binom{n}{d-i}, \text{ for } i > \lfloor d/2 \rfloor. \quad (7.12)$$

This is the maximum possible value for any simple polytope (cf. (7.8)). We shall show that the number of i -faces of P attains the maximum possible value for $i \leq \lfloor d/2 \rfloor$, too.

Let us substitute (7.12) in (7.6). Note that $h(i)$ depends only on $f(j)$, for $j \geq i$. The reader can check using binomial identities that

$$h(i) = \binom{n-i-1}{d-i}, \text{ for } i > \lfloor d/2 \rfloor. \quad (7.13)$$

By the Dehn–Sommerville relations, $h(i) = h(d-i)$. Hence,

$$h(i) = \binom{n-d+i-1}{i}, \text{ for } i \leq \lfloor d/2 \rfloor. \quad (7.14)$$

In other words, the bounds on $h(i)$ in the exact form of the upper bound theorem are actually attained for the dual cyclic polytope P . Using the relation between the h -vector and the f -vector, it follows that the bounds on $f(i)$ in Theorem 7.2.7 are also attained for the polytope P .

Another geometric proof of this fact is given in the following exercise.

Exercises

7.2.1 Prove (7.14) by showing that the equality holds in Lemma 7.2.8 if P is dual-cyclic. (Hint: Examine the proof of this lemma. Let z be the linear function used in its proof. If the equality did not hold, then there exists a vertex $v \notin F$ with $\text{in-degree}(v) = i$. Let G be the unique $(d - i)$ -face containing the $d - i$ outgoing edges at v . Then, G and F do not intersect. Let a_1, \dots, a_i be the bounding hyperplanes of P that contain G . Let a be the bounding hyperplane containing F . What this means is that $a \cap a_1 \cap \dots \cap a_i$ does not determine a face of P . This is a contradiction if $i + 1 \leq \lfloor d/2 \rfloor$: The proof of Lemma 7.2.11, in a dual setting, shows that the intersection of any $j \leq \lfloor d/2 \rfloor$ bounding hyperplanes determines a $(d - j)$ -face of P .)

Now, (7.13) follows from (7.14) by Dehn–Sommerville relations.

7.2.2 Let Q be a cyclic polytope as in this section. Assume that $t_1 < t_2 < \dots < t_n$. Let t_{i_1}, \dots, t_{i_d} be any d of these values. They divide the real line into several intervals. Call an interval proper if its endpoints are different from t_1 and t_n . It is said to be even, if it contains an even number of the remaining t_i 's in its interior. Show that the points $x(t_{i_1}), \dots, x(t_{i_d})$ form a facet of Q iff all proper intervals are even.

In general, show that the points $x(t_{i_1}), \dots, x(t_{i_j})$ form a $(j - 1)$ -face iff the number of odd proper intervals is at most $d - j$. Using this characterization, give another proof that the bounds in the upper bound theorem are actually attained for dual cyclic polytopes.

****7.2.3** Choose n points randomly from a uniform distribution on a unit ball in R^d . Show that the expected size of the convex hull of these points is $O(n)$, in fact, $O(n^{(d-1)/(d+1)})$. This shows that the bounds in the upper bound theorem are worst-case bounds. In reality, the size of a convex polytope can be much smaller.

7.3 Incremental construction

We have seen that a d -polytope can have at most $O(n^{\lfloor d/2 \rfloor})$ faces, where n is the number of bounding hyperplanes. We have also seen that this bound is tight in the worst case. In this section, we give a randomized incremental algorithm to construct the facial lattice of a d -polytope in $O(n^{\lfloor d/2 \rfloor})$ expected time. It is almost a *verbatim* translation of the algorithm in Section 3.2 for three-dimensional convex polytopes. Hence, we shall keep our description brief.

Let N be a set of n half-spaces in R^d . As usual, we assume that the half-spaces are in general position. Let $H(N)$ be the convex polytope formed by intersecting these half-spaces. It is simple because of our general position assumption (Exercise 7.0.1). Our goal is to construct the facial lattice of this polytope. It suffices to construct the 2-skeleton, because it can be easily extended to the full facial lattice (Exercise 7.0.3). We shall add the half spaces in N , one at a time, in random order. Let N^i be the set of the first i

half-spaces. We assume that $H(N^i)$ is always bounded. This can be ensured by the bounding box trick as in Section 3.2.

At the i -stage of the algorithm, we maintain the 2-skeleton of the polytope $H(N^i)$. Consider the addition of the $(i+1)$ th half-space $S = S_{i+1}$. We obtain $H(N^{i+1})$ from $H(N^i)$ by splitting off the cap, which is defined as $\text{cap}(S_{i+1}) = H(N^i) \cap \bar{S}$. Here \bar{S} denotes the complement of S . Details of this operation are as follows (Figure 7.9). Let us say that an edge of $H(N^i)$ conflicts with S if it intersects the complement half-space \bar{S} . A conflicting vertex or 2-face of $H(N^i)$ is defined similarly. Assume for a moment that we are given some vertex $p \in H(N^i)$ in conflict with S (if any). If there is no such vertex, then S must be redundant, because $H(N^i)$ is bounded. In that case, S can be thrown away. Otherwise, we visit all conflicting edges and vertices of $H(N^i)$ by a search on the edge skeleton of $H(N^i)$ starting at p . During this search, we take care of not crossing into the half-space S at any time. Figure 7.9 gives an illustration. This search works because the conflicting edges and vertices of $H(N^i)$ form a connected subgraph of the edge skeleton of $H(N^i)$. The latter fact holds because the convex polytope $H(N^i) \cap \bar{S}$ is bounded; see also Exercise 7.0.2.

At the end of the above search, we know all conflicting edges and vertices of $H(N^i)$. We also know its conflicting 2-faces because they are adjacent to the conflicting edges. We remove the conflicting vertices, edges, and 2-faces lying completely within \bar{S} . The remaining edges and 2-faces intersect S partially. They are split and only their intersections with S are retained. We also introduce new vertices and edges which correspond to their intersections with the bounding hyperplane ∂S .

It remains to determine the 2-faces in $H(N^i) \cap \partial S$. In the trivial three-dimensional case illustrated in Figure 7.9, there is a unique such 2-face. In general, there can be many. But we already know the edge skeleton of

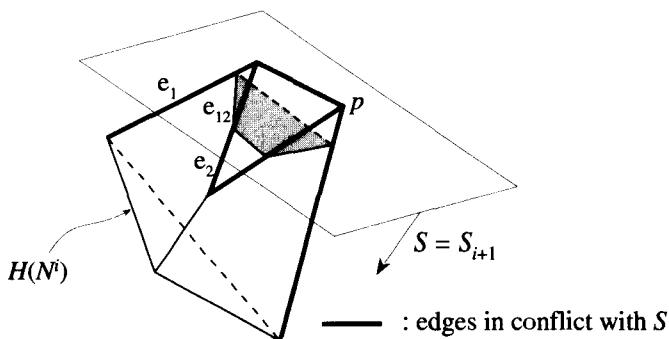


Figure 7.9: Addition of a new half-space S .

$H(N^i) \cap \partial S$ at this stage. So we can extend it to the 2-skeleton in linear time (Exercise 7.0.3).

Analysis

Assume that we are given a conflicting vertex in $H(N^i)$ for free. Then it follows from the preceding discussion that the addition of $S = S_{i+1}$ takes time proportional to the *structural change* during the addition. This is defined as the number of destroyed vertices in $H(N^i)$ plus the number of newly created vertices in $H(N^{i+1})$. Each newly created vertex can be destroyed only once. Hence, the amortized cost of this addition can be taken to be proportional to the number of newly created vertices in $H(N^{i+1})$. These are the vertices of $H(N^{i+1})$ contained in ∂S_{i+1} . Let $m(S, N^{i+1})$ denote their number. Now let us calculate the expected amortized cost of the $(i+1)$ th addition, conditional on a fixed N^{i+1} . Since each half-space in N^{i+1} is equally likely to occur as S_{i+1} , this is proportional to

$$\frac{1}{i+1} \sum_{S \in N^{i+1}} m(S, N^{i+1}).$$

Each vertex in $H(N^{i+1})$ is contained in d hyperplanes because the hyperplanes are in general position. Hence, the latter sum is proportional to the size of $H(N^{i+1})$. This is $O(i^{\lfloor d/2 \rfloor})$ by the Upper Bound Theorem. Thus, the expected amortized cost of the $(i+1)$ th addition, conditional on a fixed N^{i+1} , is $O(i^{\lfloor d/2 \rfloor - 1})$. This holds for arbitrary N^{i+1} . Hence, we have proved the following:

Lemma 7.3.1 *The expected amortized cost of the $(i+1)$ th addition is $O(i^{\lfloor d/2 \rfloor - 1})$, provided we know one vertex of $H(N^i)$ in conflict with S .*

But how do we find a conflicting vertex? We provide two solutions.

7.3.1 Conflicts and linear programming

The first solution is based on linear programming (Figure 7.10). Let z be a linear function such that $S = \{z \leq 0\}$. Because $H(N^i)$ is bounded and the half-spaces are in general position, there must be a unique z -maximum on $H(N^i)$. Let us denote it by p . It can be found in expected $O(i)$ time using the linear programming algorithm in Section 7.1. If S contains p , it must be redundant. In this case, it can be thrown away. Otherwise, p must conflict with S . The total cost of linear programming throughout the algorithm is $O(\sum i) = O(n^2)$. In conjunction with Lemma 7.3.1, this implies that the expected cost of the whole algorithm is $O(n^{\lfloor d/2 \rfloor})$.

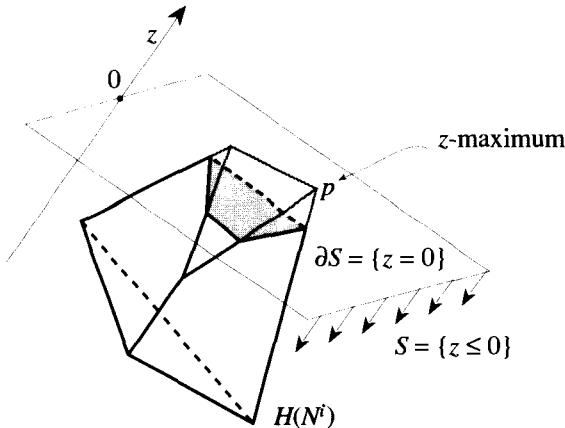


Figure 7.10: Detection of a conflicting vertex using linear programming.

7.3.2 Conflicts and history

The second solution is to maintain the conflict information just as in Section 3.2.1. At every stage i , we maintain, for every half-space $I \in N \setminus N^i$, a pointer to one vertex in $H(N^i)$ in conflict with it, if any. The conflict information provides us a required conflicting vertex (if any) during the addition of S_{i+1} . Now, during the addition of $S = S_{i+1}$, we also need to update the conflict information. This can be done by translating the conflict relocation procedure in Section 3.2.1 *verbatim*: We only need to substitute the word plane with the word hyperplane. It takes time proportional to the total conflict change during this addition. It also follows just as in Section 3.2.1 that the expected total conflict change over the whole algorithm is bounded by

$$\sum_{j=1}^n d^2 \frac{n-j}{j} \frac{e(j)}{j}. \quad (7.15)$$

Here $e(j)$ denotes the expected number of vertices in the convex polytope $H(N^j)$. The above equation is the same as (3.7). Our proof of (3.7) holds for arbitrary d , as the reader can verify. Alternatively, this also follows from Theorem 3.4.5. By the upper bound theorem, $e(j) = O(j^{\lfloor d/2 \rfloor})$. Substituting this bound in (7.15), it follows that the expected total cost of conflict relocation is $O(n^{\lfloor d/2 \rfloor})$, for $d > 3$, and $O(n \log n)$, for $d = 2, 3$.

Combining with Lemma 7.3.1, we have proved the following:

Theorem 7.3.2 *The convex polytope formed by intersecting n half-spaces in R^d can be constructed in expected $O(n^{\lfloor d/2 \rfloor})$ time, for $d > 3$, and $O(n \log n)$ time, for $d = 2, 3$.*

The above algorithm is not on-line, because it maintains conflicts of the unadded half-spaces at every stage. It can be made on-line exactly as in Section 3.2.2 by maintaining the *history* of the previous additions. The definition and the use of history are exactly as in Section 3.2.2. The reader only needs to reread Section 3.2.2, but without assuming that $d = 3$. We have already noted in Section 3.2.2 that the conversion into an on-line algorithm does not increase the running time by more than a constant factor. This is because history only delays the conflict relocation work.

Exercises

7.3.1 Reread the conflict relocation procedure in Section 3.2.1 and the use of history in Section 3.2.2 and convince yourself that the description given there is valid *verbatim* in arbitrary dimension.

7.3.2 Apply the results in section 3.5 to conclude that the running time of the on-line algorithm based on linear programming is $\tilde{O}(n^{\lfloor d/2 \rfloor} \log n)$.

† Can you prove the same bound for the algorithm based on history?

7.3.3 Translate the algorithm in this section to a dual setting directly, thereby obtaining a randomized incremental algorithm for constructing convex hulls.

7.3.4 Choose n points in R^d randomly from a uniform distribution on a unit ball as in Exercise 7.2.3. Let N be the set of dual hyperplanes. They implicitly stand for the half-spaces containing the origin. Show that the expected total structural change in the algorithm of this section is $O(n)$. Show that the expected total conflict change is $O(n \log n)$. (Hint: The expected value of $e(i)$ as in (7.15) or Theorem 3.4.5 is $O(i)$.)

This implies that the expected running time of the algorithm is $O(n \log n)$, if N is chosen in this fashion.

7.4 The expected structural and conflict change

The expected running times of the previous incremental algorithms are governed by the expected structural and conflict change over the whole algorithm. We have seen that its value is $O(n^{\lfloor d/2 \rfloor})$. This bound is tight in the worst case. For example, if $H(N)$ is a dual cyclic polytope, its size is $\Omega(n^{\lfloor d/2 \rfloor})$. So the algorithm must spend $\Omega(n^{\lfloor d/2 \rfloor})$ time in this case.

On the other hand, the $O(n^{\lfloor d/2 \rfloor})$ bound is rather pessimistic. In practice, the expected structural and conflict change can be rather small. For example, if N is chosen from a suitable uniform distribution, then the expected structural change can be as small as $O(n)$ (Exercise 7.3.4). We are not claiming that in practice the elements in N are usually drawn from a uniform distribution. That is certainly not the case. Hence, it is important to obtain bounds that do not depend on the input distribution. Theorem 3.4.5 gives

bounds on the expected structural and conflict change that do not depend on the input distribution. In this section, we shall give another complementary bounds. The bounds in Theorem 3.4.5 are based on random samples of N . The bounds in this section are stated in terms of N directly. This property will be helpful later (Chapter 10) when we study exactly how much randomness is necessary in randomized incremental algorithms. The bounds also reveal an intersecting property of randomized incremental algorithms: They are *quasi-output-sensitive* in a certain sense, as we shall see soon.

Let us fix a set N of half-spaces in R^d in general position. Let \bar{u} be any random sequence of additions of these half-spaces. Let N^k denote the set of the first k half spaces in this sequence. Let us define the structural change during the k th addition to be the number of vertices (in the underlying convex polytope) that are newly created or destroyed during this addition. We are only considering vertices here because the number of newly created or destroyed faces of higher dimensions is of the same order. This is because each vertex of a d -polytope is adjacent to $O(1)$ faces if its bounding hyperplanes are in general position and d is constant. We define the total structural change during \bar{u} to be the sum of the structural changes over all additions.

We define the total conflict change in a similar fashion. Let v be any vertex existing at time k . This means it belongs to the polytope $H(N^k)$. The conflict level (size) of v is defined to be the number of half-spaces in $N \setminus N^k$ in conflict with v . We say that a half-space conflicts with v if its complement contains v . The conflict level of v will be denoted by $l(v)$. For every fixed integer $b \geq 0$, we define the b th order conflict change during the k th addition to be $\sum_v \binom{l(v)}{b}$, where v ranges over all newly created and destroyed vertices during the k th addition. The total b th order conflict change is defined to be the sum of these changes over all k . In the above definition of the change during the k th addition, we can let v range over only the newly created vertices. This is because each newly created vertex can be destroyed only once. Hence, the total change defined in this fashion differs from the previous definition by at most a factor of two. In what follows, we shall use this latter definition. The 0th order conflict change is the same as the structural change defined before. The first order conflict change is the usual conflict change.

Let $A(N)$ denote the arrangement in R^d formed by the hyperplanes bounding the half-spaces in N . Refer to Figure 7.11. For any fixed vertex v in this arrangement, define its level $l(v)$ to be the number of half-spaces in N that conflict with v , i.e., contain v in their complements. We say that v is created during \bar{u} if it occurs in the convex polytope $H(N^k)$, for some k . In this case, the definition of $l(v)$ coincides with the definition given before. If the convex polytope $H(N)$ is nonempty, the level can also be defined as follows (Figure 7.11). Fix the origin o anywhere within $H(N)$. The level of a

vertex v is the same as the number of hyperplanes that separate v from the origin. It increases as we move farther from the origin. All vertices in the convex polytope $H(N)$ have level 0.

Fix a vertex v in the arrangement $A(N)$. Let us define a 0–1 function $I(v)$ by: $I(v)=1$, if v is created during \bar{u} , and zero otherwise. It is clear that the total b th order conflict change during \bar{u} is

$$\sum_{v \in A(N)} \binom{l(v)}{b} I(v). \quad (7.16)$$

Let $L(v)$ denote the set of half-spaces in N conflicting with v . Let $D(v)$ denote the set of half-spaces defining v . By a defining half-space, we mean a half-space whose boundary contains v . The size of $D(v)$ is d , because the half-spaces are in general position. It is easy to see that v is created during \bar{u} iff all of the first d half-spaces to be added from the set $L(v) \cup D(v)$ belong to $D(v)$. If the sequence \bar{u} is random, this happens with probability $1/\binom{l(v)+d}{d}$. Hence, the expected value of $I(v)$ is $1/\binom{l(v)+d}{d}$. It follows from (7.16) that the expected b th order conflict change during a random sequence of additions is

$$\sum_{v \in A(N)} \frac{\binom{l}{b}}{\binom{l(v)+d}{d}} \approx \sum_{v \in A(N)} \frac{1}{(l(v) + d)^{d-b}}. \quad (7.17)$$

(Here \approx denotes asymptotic equality with constant factors ignored.) For notational convenience, let us denote the series on the right hand side of this equation by $\Theta(N, b)$, where it is assumed that $0 \leq b \leq d$. Note that the contribution of a vertex v in the arrangement $A(N)$ to this series is

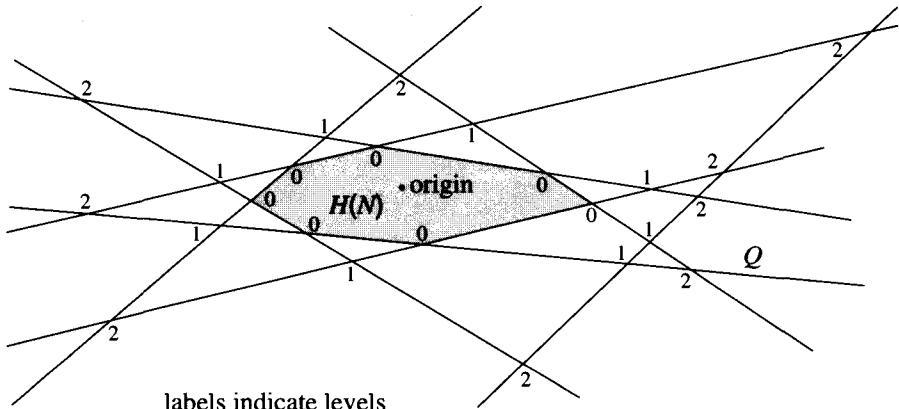


Figure 7.11: Levels of vertices in an arrangement.

inversely proportional to a power of its level. Observe that every vertex in the polytope $H(N)$ contributes a fixed constant to $\Theta(N, b)$. We can also rewrite the expression for $\Theta(N, b)$ in another illuminating way. Let $m_l(N)$ denote the number of vertices in $A(N)$ with level l . Thus, $m_0(N)$ is the number of vertices on the polytope $H(N)$. Then

$$\Theta(N, b) \approx \sum_l \frac{m_l(N)}{(l+d)^{d-b}}, \text{ for } 0 \leq b \leq d. \quad (7.18)$$

It follows from (7.17) that:

Proposition 7.4.1 *For $0 \leq b \leq d$, the expected b th order conflict change during a random N -sequence of additions is of the order of $\Theta(N, b)$, ignoring a constant factor.*

Let us turn to the randomized incremental algorithms once again. Any on-line algorithm for maintaining convex polytopes must spend time proportional to the structural change during the given sequence of additions. It follows from Proposition 7.4.1 that the expected running time of any on-line algorithm for maintaining convex polytopes is trivially $\Omega(\Theta(N, 0))$. Let us compare the running times of the previous randomized incremental algorithms with this trivial lower bound. It follows from the above proposition that the expected running time of the on-line algorithm based on linear programming (Section 7.3.1) is $O(\Theta(N, 0) + n^2)$. The first term accounts for the expected structural change. The second term accounts for the total cost of linear programming. The expected running time of the incremental algorithm based on conflicts (Section 7.3.2) is proportional to the expected conflict change. By the above proposition, this is $O(\Theta(N, 1))$. The expected running time of the on-line algorithm based on history is of the same order, because history only delays conflict relocation work (Section 3.2.2). The first term of $\Theta(N, 0)$, as well as $\Theta(N, 1)$, is $m_0(N)$, the size of $H(N)$ (see (7.18)). In practice, we should expect the series to converge to something “comparable” to this first term, though this is not a theoretical guarantee. In that sense, both algorithms are quasi-output-sensitive.

This leads us to yet another question. Can one design a genuinely output-sensitive algorithm for constructing $H(N)$? By this, we mean an algorithm whose running time is proportional to the output size of $H(N)$. Clearly, we cannot design an on-line algorithm with this property. In fact, we have already seen that the expected running time of any on-line algorithm on a random sequence of additions is trivially $\Omega(\Theta(N, 0))$. But what if we were only interested in the static setting, wherein all half spaces in N are given to us in advance? In that case, there does exist a deterministic algorithm whose running time is proportional to the output size, ignoring a certain quadratic

overhead. It is outside the scope of this book. Randomization does not seem to help much in the design of output-sensitive algorithms.

7.5 Dynamic maintenance

In this section, we shall generalize the on-line algorithm based on history (Section 7.3.2) to the fully dynamic setting. In the dynamic setting, the user is allowed to add or delete a half-space at any time. Our dynamic algorithm is a generalization and a *verbatim* translation of the previous algorithm for three-dimensional polytopes (Section 4.5.1). Hence, our treatment will be very brief.

Let M be the set of current (undeleted) half-spaces in R^d existing at any given time. Let m be the size of M . Our goal is to maintain the edge skeleton of the convex polytope $H(M)$ in a dynamic fashion. In addition, we shall also maintain a certain history at all times. It is defined as follows. Imagine adding the half-spaces in M in the same order in which they were added during the update sequence—but completely ignore the deleted half-spaces. Our history will correspond to this addition sequence as in Section 7.3.2. It looks as if the deleted half-spaces were never added in the first place. Addition of a new half-space is done exactly as before. It only remains to describe the deletion operation.

A half-space is deleted from the history via rotations just as in Section 4.5.1. This automatically updates $H(M)$ as well. The reader should reread Section 4.5.1 and be convinced that everything there translates *verbatim* into the present setting. The bound on the expected cost of rotations during a random deletion also remains the same as before (see (4.12)). It is of the order of

$$\frac{1}{m} \sum_{i=1}^m \frac{e(i, M)}{i}. \quad (7.19)$$

Here $e(i, M)$ denotes the expected size of the polytope $H(I)$, where I is a random sample of M of size i . The Upper Bound Theorem implies that $e(i, M) = O(i^{\lfloor d/2 \rfloor})$. Hence, the expected cost of rotations during a random deletion is $O(m^{\lfloor d/2 \rfloor - 1})$, for $d > 3$.¹ This is the dominant cost during deletion. During addition, there is also an additional cost of conflict search. Its expected value is similarly $O(m^{\lfloor d/2 \rfloor - 1})$, for $d > 3$. This follows from Theorem 4.3.5 in conjunction with the upper bound theorem again. It follows that:

¹For $d > 3$, there is no additional log factor in this bound, because there is really no need to maintain the priority queue, as in Section 4.5.1 (see the discussion on page 153).

Theorem 7.5.1 Suppose $d > 3$. Then the cost of adding or deleting a half-space in the set M is $O(m^{\lfloor d/2 \rfloor - 1})$, assuming that the updates are random. In other words, the expected cost of dynamic maintenance over a random (N, δ) -sequence is $O(n^{\lfloor d/2 \rfloor})$, for any N and δ .

Exercise

**7.5.1 For the dynamic algorithm in this section, show that the expected total cost of rotations over a random (N, δ) -sequence is $O(\Theta(N, 0) \log n)$, if the signature δ is weakly monotonic.

7.6 Voronoi diagrams

We have already seen that Voronoi diagrams in R^d are projections of certain upper convex polytopes in R^{d+1} (Section 2.5). Hence, the results in the previous section immediately imply the following results for Voronoi diagrams.

The upper bound theorem for convex polytopes (Theorem 7.2.5) implies the following.

Theorem 7.6.1 The size of a Voronoi diagram of n sites in R^d is $O(n^{\lceil d/2 \rceil})$.

Theorem 7.5.1 implies:

Theorem 7.6.2 Dynamic maintenance of Voronoi diagrams in R^d can be done so that the expected running time of the algorithm over a random (N, δ) -sequence is $O(n^{\lceil d/2 \rceil})$, for $d > 2$. Here N is any set of n sites and δ is any signature.

Exercises

*7.6.1 Demonstrate a set of sites so that the upper bound in Theorem 7.6.1 is actually attained.

7.6.2 Translate the algorithms for convex polytopes to the present setting of Voronoi diagrams directly. The algorithm should not transform the sites in R^d to hyperplanes in R^{d+1} .

7.6.3 Show that the discussion in Section 7.4 can be translated as follows in the setting of Voronoi diagrams. Let N be any set of n sites in R^d . Assume that the sites are in general position. Every subset Δ of $d+1$ sites from N defines a unique ball $B(\Delta)$ circumscribing those sites. Let $l(\Delta)$ denote the number of sites in N that are contained in the interior of $B(\Delta)$. Note that $l(\Delta)$ is zero iff Δ is a Delaunay sphere as defined in Exercise 2.5.4. For the given set N , let

$$\Theta(N, s) = \sum_{\Delta} \frac{1}{(l(\Delta) + d + 1)^{d+1-s}}, \quad \text{for } 0 \leq s \leq d + 1,$$

where Δ ranges over all subsets of $d+1$ sites in N . In other words, the contribution of a circumscribing sphere is inversely proportional to a power of its level. Now

translate the previous quasi-output-sensitive bounds to the dynamic and semidynamic algorithms for Voronoi diagrams.

- **7.6.4** Choose n sites from a uniform distribution on a unit ball in R^d . Show that the expected size of their Voronoi diagram is $O(n)$. Show that the expected running time of the randomized incremental algorithm for constructing this Voronoi diagram is $O(n \log n)$.

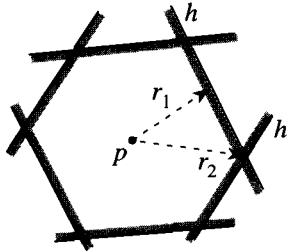


Figure 7.12: Ray shooting.

7.7 Search problems

There are two important search problems associated with convex polytopes. Let N be a set of half-spaces in R^d . Let $H(N)$ be the polytope formed by intersecting these half-spaces.

1. Polytope membership problem: Given a query point $p \in R^d$, decide if $p \in H(N)$.

2. Ray shooting problem: Given a ray shooting from a query point $p \in R^d$, locate the face of $H(N)$, if any, that is first hit by this ray. If the ray is in general position, such as the ray r_1 in Figure 7.12, it will hit a facet of $H(N)$. This facet is uniquely determined by the hyperplane containing it. To be specific, our goal is to determine this hyperplane.

It is possible that the ray hits a lower-dimensional face of $H(N)$. In this case, we should report all hyperplanes containing this face. For example, in Figure 7.12, h and h' should be reported if the query ray is r_2 . In what follows, we shall assume that the query ray is in general position. Only trivial modifications are required to handle the general case.

An important special case of the ray shooting problem is the following. Assume that all half-spaces in N contain a fixed base point o . Given a query ray originating from o , the goal is to locate the face of $H(N)$, if any, that is hit by this ray. This special case is important because the nearest neighbor problem is reducible to it (Section 2.5). Let us recall this reduction (Figure 7.13). We identify R^d with the hyperplane $x_{d+1} = 0$ in R^{d+1} . We transform the given sites in R^d to hyperplanes tangent to the unit paraboloid

in R^{d+1} . Let N be the set of these hyperplanes. They will implicitly stand for the upper half-spaces bounded by them. By an upper half space, we mean a half-space containing the point $(0, \dots, 0, \infty)$. The base point o will be this point at infinity. Let $p \in R^d$ be the query point. Consider the vertical ray originating at the point o at infinity and directed towards p . The nearest site to p corresponds to the first hyperplane h hit by this ray. In other words, the nearest neighbor problem is reducible to vertical ray shooting among a set of hyperplanes.

In what follows, we shall assume that the polytope $H(N)$ is an upper convex polytope. Thus, we can assume that the elements in N are hyperplanes that implicitly stand for the upper half-spaces bounded by them. The preceding assumption is only for the sake of simplicity. Our algorithms can be directly translated so that they can deal with general convex polytopes. This translation is straightforward, so we leave it to the reader.

The membership problem for upper convex polytopes is also reducible to vertical ray shooting. Let us see how. Let p be the query point. Suppose we wish to know whether p belongs to $H(N)$. Consider the vertical ray emanating from the point $(0, \dots, 0, \infty)$ at infinity and directed towards p . Let h be the first hyperplane hit by this ray. It is clear that $p \in H(N)$ iff p lies in the upper half-space bounded by h .

In the next section, we shall give an efficient search structure for vertical ray shooting among the hyperplanes in N . We shall see later (Section 8.7) that the same structure can also be used for general ray shooting. This will be done using a powerful technique called parametric search. For the moment, let us confine ourselves to vertical ray shooting. As we have already seen, the nearest neighbor problem is reducible to it.

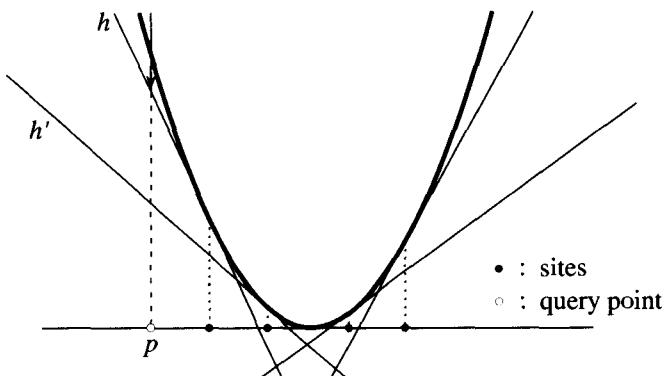


Figure 7.13: Connection between the nearest neighbor problem and vertical ray shooting.

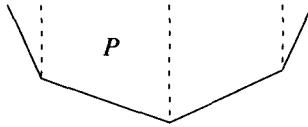


Figure 7.14: Vertical canonical triangulation.

7.7.1 Vertical ray shooting

Let N be a set of hyperplanes in R^d . Our goal is to build a search structure for vertical ray shooting among the hyperplanes in N . Given a vertical ray originating from the point $(0, \dots, 0, \infty)$ at infinity, the goal is to report the first hyperplane hit by this ray.

Our search structure will be based on the canonical triangulation described in Section 6.3. Let us recall it briefly. Let P be a given convex polytope. We triangulate P by induction on its dimension. When the dimension of P is one, nothing needs to be done. Otherwise, we triangulate its facets recursively. Next, we extend all simplices in the facets to cones with apex at the top of P . By the top of P , we mean its vertex with the maximum x_d -coordinate. In Section 6.3, we used the bottom instead of the top. This only amounts to reversing the direction of x_d . We shall refer to the cells in the triangulation of P as simplices. They can be cylinders, strictly speaking, when the top of P lies at infinity. In particular, if P is an upper convex polytope, the cells in the above triangulation are vertical cylinders extending in the positive x_d direction (Figure 7.14).

The vertical nature of our canonical triangulation will turn out to be important because the query rays are vertical. If R is a set of hyperplanes, we shall let $H(R)$ denote the vertical canonical triangulation of the upper convex polytope formed by R . The random sampling results can be applied to canonical triangulations just as in Section 6.3. Thus, if N is a set of hyperplanes and $R \subseteq N$ is a random sample of size of r , then with high probability, every simplex in $H(R)$ intersects $O([n/r] \log r)$ hyperplanes in $N \setminus R$.

Now let us turn to the search structure. It is a *verbatim* translation of the top-down search structure for arrangements (Section 6.5). Only a different interpretation of the notation $H(R)$ is required. In Section 6.5, $H(R)$ denoted the canonical triangulation of the arrangement formed by R . Here it will denote the canonical triangulation of the upper convex polytope formed by R . The root of our search structure is associated with $H(R)$, where R is a random sample of N of a large enough constant size. Every child of the root corresponds to a simplex $\Delta \in H(R)$. The subtree rooted at this child is built recursively over the set of hyperplanes in $N \setminus R$ intersecting

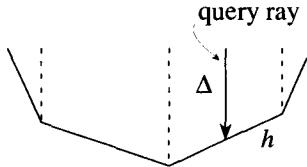


Figure 7.15: A vertical ray query.

Δ. The recursion stops when the number of intersecting hyperplanes drops below a chosen constant.

The queries are answered as follows (Figure 7.15). The query is a vertical ray originating at upper infinity. Let $h \in R$ be the first hyperplane hit by the query ray. Since R has a constant size, h can be found in constant time.

Let $\Delta \in H(R)$ be the cylinder whose facet is supported by h . If no hyperplane in $N \setminus R$ intersects Δ , the answer to our query is h . Otherwise, we locate the first hyperplane in $N \setminus R$ hit by the query ray. Let h' be this hyperplane. It is found recursively using the subtree for Δ . The answer to our query is h or h' , depending upon which one is hit first by the query ray.

Theorem 7.7.1 *The above search structure can be built in $O(n^{\lfloor d/2 \rfloor + \epsilon})$ space in expected $O(n^{\lfloor d/2 \rfloor + \epsilon})$ time. The query time is $O(\log n)$.*

Proof. This follows just as Theorem 6.5.1 did. There $H(R)$ denoted the canonical triangulation of the arrangement formed by R . It had $O(r^d)$ size, where r is the size of R . Now $H(R)$ denotes the canonical triangulation of the upper convex polytope formed by R . Its size is $O(r^{\lfloor d/2 \rfloor})$ by the Upper Bound Theorem. It can also be built in expected $O(r^{\lfloor d/2 \rfloor})$ time in a randomized incremental fashion (Section 7.3). So one only needs to substitute $\lfloor d/2 \rfloor$ for d in the statement of Theorem 6.5.1. \square

Exercise

7.7.1 In this section, we assumed that the base point o of the ray is located at infinity. Generalize the algorithm so as to allow the base point to be located anywhere in R^d .

7.7.2 Half-space range queries

The preceding search structure can also be used for answering half-space range queries in a dual form. Recall from Section 2.4.1 that half-space range queries assume the following form in the dual setting: Given a set N of hyperplanes and a query point p , report all hyperplanes in N that are above (or below) the query point p . The above and below relations are defined with respect to the x_d -coordinate.

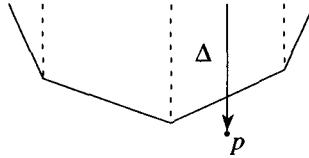


Figure 7.16: Half-space range queries in a dual setting.

We only need to augment our search structure slightly. We store the whole set N at its root. Recursively, we store at each node of the search structure the hyperplanes intersecting the labeling simplex.

The queries are answered as follows. Let p be the query point as before. Our goal is to determine all hyperplanes in N above p ; the hyperplanes below p can be determined using a similar structure. Let R denote the random sample at the root as before. We first determine the hyperplane in R that first meets the vertical ray directed towards p (Figure 7.16). Let Δ be the cylinder in $H(R)$ whose facet is bounded by this hyperplane. If p is within Δ , we recursively determine all hyperplanes intersecting Δ above p . If p is outside $H(R)$, we answer the query naively by going through the whole set N . This naive procedure takes $O(n)$ time. It suffices for the following reason: Let k be the number of hyperplanes in N above p . The set R is a random sample of N of constant size. The point p lies outside $H(R)$ iff R contains at least one of these k hyperplanes. The probability of this event is $O(k/n)$. Hence, the expected time spent at the root is $O([k/n]n) = O(k)$. The depth of our search structure is $O(\log n)$. Hence, it follows by a recursive argument that, for a fixed query point p , the expected query cost is $O(k \log n)$.

Another problem related to half-space ranges is the so-called *half-space emptiness* problem. Given a half-space, the goal is to determine quickly if the half-space is empty. In the present dual setting, the problem is to determine quickly if there is any hyperplane above p . This can be done by a modified search procedure as follows. Let R be the random sample associated with the root as before. If p lies outside $H(R)$, the answer is yes; in this case we can also provide the hyperplane in R above p as a witness. Otherwise, let $\Delta \in H(R)$ be the simplex containing p as before. We recur over the search structure for Δ . This tells us if there is any hyperplane above p among those intersecting Δ . Since the depth of our search structure is $O(\log n)$, the query cost is also $O(\log n)$.

We have thus proved the following.

Theorem 7.7.2 *The previous search structure can also be used for answering half-space range queries in a dual form. The expected cost of answering a fixed query is $O(k \log n)$, where k is the answer size. The cost of answering a half space emptiness query is $O(\log n)$.*

7.7.3 Nearest k -neighbor queries

The nearest k -neighbor problem is a generalization of the nearest neighbor problem. We are given a set of sites in R^d . The goal is to report the nearest k sites to the query point quickly. We have already seen that the nearest neighbor problem can be transformed to the vertical ray shooting problem (Figure 7.13). The same transformation reduces the nearest k -neighbor problem to the following: Given a set N of hyperplanes in R^{d+1} and a query vertical ray originating at the upper infinity, determine the first k hyperplanes in N that meet this vertical ray. Thus, in Figure 7.13, the first two such hyperplanes are h and h' .

The search structure in Section 7.7.1 can also be used for answering such queries. At the root level, we locate the hyperplane $h \in R$ meeting the vertical query ray as before (Figure 7.15). Let Δ be the cylinder whose facet is bounded by h . Recursively, we locate the first k hyperplanes, among those intersecting Δ , which meet the query ray above h . It may turn out that the number of such hyperplanes is less than k . In that case, we answer the query naively by going through the whole set N . It follows as in Section 7.7.2 that, for a fixed query ray, the expected query cost is $O(k \log n)$. In fact, the analysis in Section 7.7.2 is directly applicable if we let p in the analysis there be the point on the query ray just below the k hyperplanes in N . The point p is only used in the analysis. The algorithm need not know its whereabouts.

We have thus proved the following.

Theorem 7.7.3 *The previous search structure can also be used for answering the nearest k -neighbor queries in a transformed setting. The expected cost of answering a fixed query is $O(k \log n)$.*

Exercises

****7.7.2** Given a set N of n sites in R^d , show that one can determine the closest pair of sites in N in expected $O(n)$ time. (You are allowed to use the floor function.) Give a dynamic search structure for determining the closest pair with the expected $O(\log n)$ cost of update and the expected $O(n)$ space requirement.

****7.7.3** Given n sites in R^d , show that one can determine the nearest neighbors of all sites in $O(n \log n)$ time.

****7.7.4** Let N be a set of n sites in R^d . Fix a parameter $\epsilon > 0$. Consider the following approximate version of the nearest neighbor problem. Here, given a query point p , the goal is to find any site in N whose distance from p is at most $(1 + \epsilon)$ times the distance to the closest site from p . For a fixed ϵ , give a static, randomized search structure with $O(n)$ expected space requirement and $O(\log n)$ expected query time. The constant factor within the Big-Oh notation can depend on ϵ .

7.7.4 Dynamization

The search structure in Section 7.7.1 can be dynamized using the dynamic sampling scheme as in Section 6.5. First, we alter the static structure so as to make it more flexible. Let M denote the set of hyperplanes existing at any given time. We shall denote our search structure at this time by $\text{sample}(M)$. It is based on a gradation

$$M = M_1 \supseteq M_2 \supseteq \cdots \supseteq M_r \supset M_{r+1} = \emptyset.$$

Here M_{i+1} is obtained from M_i by flipping a coin with a suitable bias p . Let $H(M_r)$ denote the canonical triangulation of the upper convex polytope formed by M_r . We store $H(M_r)$ at the root of our search structure. The children of this root correspond to the simplices of $H(M_r)$. The subtree rooted at every child is built recursively over the set of hyperplanes in $M \setminus M_r$ conflicting with, i.e., intersecting the corresponding simplex. $\text{sample}(M)$ is akin to the dynamic point location structure for arrangements (Section 6.5): We are just replacing the word “arrangement” with the words “upper convex polytope.” $\text{sample}(M)$ can be built in $O(m^{\lfloor d/2 \rfloor + \epsilon})$ expected time and space and it guarantees $\tilde{O}(\log m)$ query cost. This follows just as in Section 6.5, but replacing d by $\lfloor d/2 \rfloor$ for the reason explained in the proof of Theorem 7.7.1. The constant ϵ can be made arbitrarily small by choosing the bias p small enough.

Additions and deletions are carried out exactly as in Section 6.5. To add a new hyperplane S to $\text{sample}(M)$, we flip our given coin repeatedly until we get failure. Let j be the number of successes before obtaining a failure. We add S to the sets M_1, \dots, M_{j+1} . This naturally gives rise to a gradation on the new set $M' = M \cup \{S\}$. Our goal is to update $\text{sample}(M)$ to $\text{sample}(M')$. This is done as follows. If $j+1 \geq r$, we construct $\text{sample}(M')$ with respect to the above gradation on M' from scratch, using the static scheme. Otherwise, we recursively add S to the subtree for each $\Delta \in H(M_r)$ intersecting S .

Deletion of S from $\text{sample}(M)$ is the reverse operation. Now let $M' = M \setminus \{S\}$. The gradation of M' is naturally inherited from the gradation of M . If $S \in M_r$, we build $\text{sample}(M')$ from scratch using the static scheme. Otherwise, we recursively delete S from the subtree for each $\Delta \in H(M_r)$ intersecting S .

Analysis

A crucial difference between the dynamic search structure for arrangements and the one here is the following. In Section 6.5.1, we could bound the expected cost of *every* update; expectation was solely with respect to randomization in the search structure. Now we shall only be able to bound the expected cost of a *random* update. The reason is, roughly speaking, the following. Let R be any set of r hyperplanes. In Section 6.5.1, $H(R)$ denoted

the canonical triangulation of the arrangement formed by R . In this case, the structural change in $H(R)$ during an addition or a deletion from R is always $O(r^{d-1})$. In the present case, $H(R)$ denotes the canonical triangulation of the upper convex polytope formed by R . Now one cannot satisfactorily bound the structural change in $H(R)$ during every such update. A satisfactory bound can be obtained only for random updates. For example, the expected structural change in $H(R)$ during the deletion of a random hyperplane in R is $O(r^{\lfloor d/2 \rfloor - 1})$ (refer to the proof of Lemma 7.3.1). This indicates that we should be able to replace d in the statement of Theorem 6.5.4 with $\lfloor d/2 \rfloor$, if we were only interested in random updates. That is indeed so.

Theorem 7.7.4 *The expected cost of maintaining the above dynamic search structure over a random (N, δ) -sequence is $O(n^{\lfloor d/2 \rfloor + \epsilon})$. Here both N and the signature δ are arbitrary and n denotes the size of N . (A random (N, δ) -sequence is defined as in the introduction of Chapter 4.)*

Proof. We shall only analyze random deletions because random additions can be analyzed backwards as usual. Let M denote the set of currently existing hyperplanes. Let m be its size. Let us sum the cost of updating $\text{sample}(M)$, by letting the deleted hyperplane range over all of M . Let ϕ be this sum. Let $E[\phi]$ be its expected value. We shall show that $E[\phi]$ can be kept within $O(m^{\lfloor d/2 \rfloor + \epsilon})$ bound by choosing the coin bias small enough. Since the deleted hyperplane is randomly chosen from M , the expected cost of a random deletion is $E[\phi]/m$, that is, $O(m^{\lfloor d/2 \rfloor - 1 + \epsilon})$.

Let z denote any node of $\text{sample}(M)$. Let Δ denote the simplex labeling z . Let t be the number of hyperplanes in M defining, i.e., adjacent to the children of Δ . Clearly, $t = O(1)$. Let $s = s(z)$ be the number of hyperplanes in M intersecting the interior of Δ . The subtree rooted at a child of z is rebuilt only if the deleted hyperplane is one of its defining hyperplanes. In this case, rebuilding is done by using the static scheme in expected $O(s^{\lfloor d/2 \rfloor + \epsilon'})$ time. The contribution to $E[\phi]$ of this rebuilding cost at the children of z , over all choices of the deleted hyperplane, is thus $O(ts^{\lfloor d/2 \rfloor + \epsilon'}) = O(s^{\lfloor d/2 \rfloor + \epsilon'})$, because $t = O(1)$. Now, $E[\phi]$ is obtained by summing such expressions for all nodes in $\text{sample}(M)$. The $O(s^{\lfloor d/2 \rfloor + \epsilon'})$ bound looks like a bound on the size of the subtree rooted at z . Hence, $E[\phi]$ can be kept within $O(m^{\lfloor d/2 \rfloor + \epsilon})$ bound by choosing the coin bias small enough. This can be proven by writing down a recurrence relation like (6.10). (The constant $\epsilon > \epsilon'$ here can be made arbitrarily small by choosing ϵ' correspondingly small.) We have already used a similar argument in Section 6.5.1, so we leave the verification to the reader. \square

Exercises

****7.7.5** Use the variable bias scheme in Exercise 6.5.2 to reduce the n^ϵ factor in Theorem 7.7.4 to $\text{polylog}(n)$.

† Can you get rid of the polylogarithmic factor too?

7.7.6 Prove an analogue of Cutting Lemma 6.3.1 for convex polytopes. More precisely, let N be a set of half-spaces in R^d . Let $H(N)$ denote the convex polytope formed by their intersection. Fix a positive parameter $r \leq n$. Construct a collection of $O(r^{\lfloor d/2 \rfloor})$ closed simplices, which cover $H(N)$, such that each simplex in the collection is intersected by the hyperplanes bounding at most n/r half-spaces in N . Do this by translating the proof of Lemma 6.3.1 *verbatim*. First, construct the canonical triangulation of $H(R)$, where $R \subseteq N$ is a random sample of size r . Then refine this triangulation suitably.

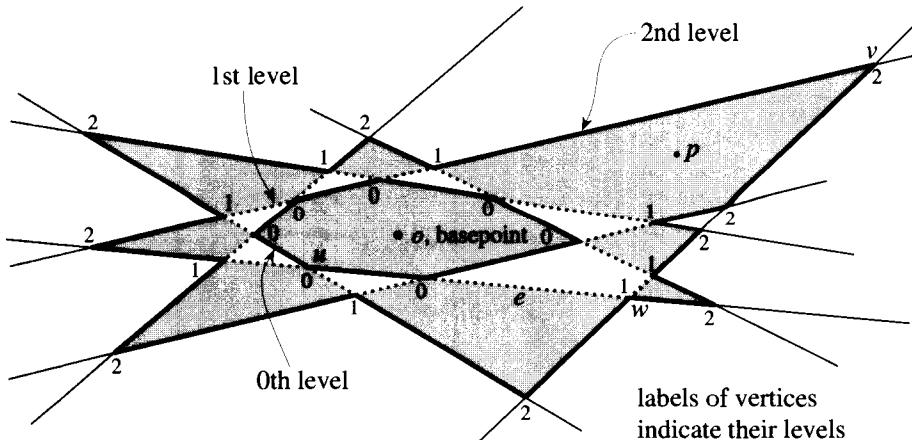
Note the following useful property of the resulting cutting (i.e., the collection of simplices): The complement of the union of the simplices in this cutting is covered by the union of the complements of the r half-spaces in R .

Also prove a general result in the weighted setting as in Exercise 6.3.4.

7.8 Levels and Voronoi diagrams of order k

In the previous sections, we studied convex polytopes and Voronoi diagrams in considerable detail. In this section, we shall briefly discuss their higher-order generalizations. These are called k -levels and k th order Voronoi diagrams, respectively. In this terminology, the zero-th level will turn out to be a convex polytope and the first order Voronoi diagram will turn out to be the usual Voronoi diagram.

Let us first define k -levels, for $k \geq 0$. Let N be a set of n hyperplanes in R^d in general position. Assume that we are given a base point (origin) in R^d , which we shall denote by o . The levels will be defined with respect to this base point. Each hyperplane in N will implicitly stand for the half-space bounded by it containing the base point. Let $A(N)$ denote the arrangement formed by the hyperplanes in N . Define the level of any point $p \in R^d$ to be the number of hyperplanes in N that conflict with p . We say that a hyperplane conflicts with p if it strictly separates p from the base point o . Thus, in Figure 7.17 the level of the point p or the vertex v is two. Define the level of any j -face $f \in A(N)$ to be the level of any point in its interior. It is easy to see that the choice of the interior point is immaterial. Let $C_k(N)$ denote the collection of the faces in $A(N)$ with level less than or equal to k . It is called the k -complex in $A(N)$. Note that $C_0(N)$ corresponds to the convex polytope surrounding the base point. Moreover, $C_n(N)$, where n is the size of N , is just the entire arrangement $A(N)$. In this way, a k -complex simultaneously generalizes the notion of a convex polytope and that of an arrangement. The boundary of $C_k(N)$ is called the k th level in $A(N)$. We

Figure 7.17: K -levels.

shall denote it by $L_k(N)$. Figure 7.17 gives an illustration. Note that $L_0(N)$ is just the boundary of the convex polytope surrounding the base point. In quite a few applications, the base point is located at $(0, \dots, 0, -\infty)$ or at $(0, \dots, 0, \infty)$. Figure 7.18 gives an illustration, for $d = 2$, when the base point is located at $(0, \infty)$. Note that the level of a point p with respect to $(0, \dots, 0, \infty)$ is just the number of hyperplanes in N that are above p in the x_d direction. Also note that the level of a point $p \in R^d$ with respect to $(0, \dots, 0, \infty)$ is n minus the level with respect to $(0, \dots, 0, -\infty)$, if p is not contained in any hyperplane in N .

A notion that is closely related to k -levels in a dual setting is that of k -sets. This notion turns out to be important in connection with half-space range queries. Let \hat{N} be a set of points in R^d . In the half-space range searching problem, the query is given in the form of a half-space. The goal is to report all points of \hat{N} contained within the query half-space. A half-space is called a k -space if it contains exactly k points of \hat{N} . For the sake of simplicity, we shall assume that the query half-space does not contain any points of \hat{N} on its boundary. Thus, the complement of a k -space is an $(n - k)$ -space. Let us call two half-spaces equivalent if they contain the same set of points in \hat{N} . The class of equivalent k -spaces corresponds to a subset of k points in \hat{N} . It is called a k -set. Not every subset of \hat{N} of size k is a k -set, because it need not correspond to a k -space. In fact, estimating the number of k -sets of \hat{N} is an interesting combinatorial question.

The notion of k -sets is intimately related to the notion of levels in arrangements. To see this, let us transform the points in \hat{N} to hyperplanes in R^d using the paraboloidal transform in Section 2.4.1. Let N be the resulting

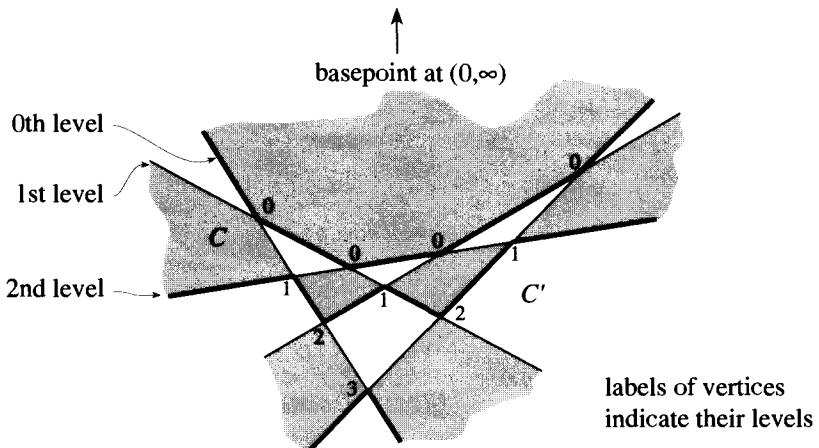


Figure 7.18: K -levels with the base point at $(0, \infty)$.

set of hyperplanes. Given a hyperplane h , let $T(h)$ denote the point that h is transformed into. The hyperplane h partitions \hat{N} into two sets of points lying on its two sides. This partition of \hat{N} corresponds to the partition of N in two sets of hyperplanes above and below $T(h)$ (Section 2.4.1). Let us call two hyperplanes h and h' in R^d equivalent if they induce the same partition of \hat{N} . Here we are assuming that neither h nor h' contains a point of \hat{N} . Observe that h and h' are equivalent iff the cells of $A(N)$ containing $T(h)$ and $T(h')$ are either identical or *projectively equivalent*. We say that two distinct cells $C, C' \in A(N)$ are projectively equivalent if the set of hyperplanes in N above (below) C is the same as the set of hyperplanes in N below (resp. above) C' (Figure 7.18). If C is bounded, then it cannot be projectively equivalent to any other cell. If C is unbounded, it is projectively equivalent to a unique cell at the “other end” of the arrangement. The preceding connection shows that the k -sets over \hat{N} correspond to the cells in $A(N)$ with level k or $n - k$, assuming that the projectively equivalent cells have been identified.

Finally, let us turn to the k th order Voronoi diagrams. They will turn out to be intimately linked to k -levels just as the usual Voronoi diagrams are linked to convex polytopes. Let \hat{N} be a set of points in R^d . Given a subset $\hat{B} \subseteq \hat{N}$ of size k , let us define the associated k th order Voronoi region, $\text{vor}(\hat{B})$, to be the set of points in R^d whose closest k -neighbors in N are the sites in \hat{B} . When $k = 1$, that is, when \hat{B} contains a single site, this definition coincides with the earlier definition of a Voronoi region. A crucial difference, when $k > 1$, is that $\text{vor}(\hat{B})$ can be empty. In fact, for a large k , it will turn out that most such higher order regions are empty. The nonempty k th order Voronoi regions form a partition of R^d . This partition is called the k th

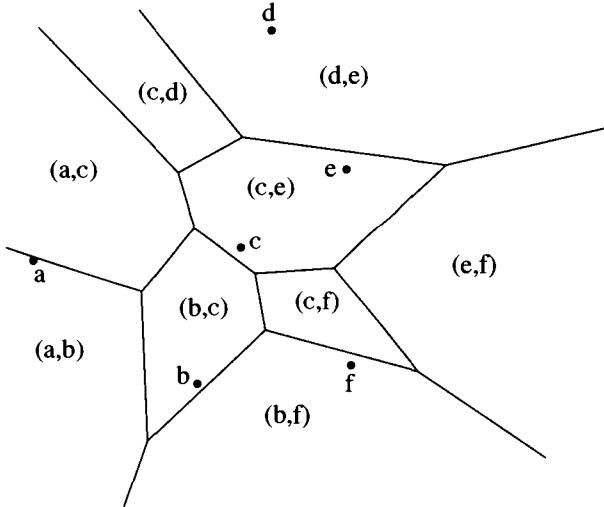


Figure 7.19: Second order Voronoi diagram in the plane.

order Voronoi diagram of \hat{N} . Figure 7.19 gives an example of a second order Voronoi diagram. Each region in this diagram is labeled with the associated pair of sites.

There is a simple connection between k -levels and k th order Voronoi diagrams. This is akin to the connection between convex polytopes and Voronoi diagrams that we described in Section 2.5. We saw there that a Voronoi diagram in R^d is the projection of the boundary of an upper convex polytope in R^{d+1} . This might lead one to believe that a k th order Voronoi diagram in R^d should be the projection of a $(k - 1)$ -level in R^{d+1} . Well, that is not true. But something very close to this holds.

First, let us recall the transformation (Figure 7.20) that we used in connecting Voronoi diagrams and convex polytopes (Section 2.5): We identify R^d with the hyperplane $x_{d+1} = 0$ in R^{d+1} . Then, we vertically project each point $p \in \hat{N}$ onto the unit paraboloid. Let \bar{p} denote this projection. Let $D(\bar{p})$ denote the tangent hyperplane to the unit paraboloid at \bar{p} . Let N be the set of these hyperplanes $D(\bar{p})$, $p \in \hat{N}$. We had already noted that this transformation has the following crucial property.

Distance transformation: Given a (query) point $q \in R^d$ and a site $p \in \hat{N}$, the square of the distance between q and p is the same as the vertical distance between \bar{q} and the hyperplane $D(\bar{p})$.

It follows that the k closest neighbors of q in \hat{N} correspond to the first k hyperplanes in N that are below \bar{q} (Figure 7.20). These are the first k hyperplanes that will be hit by the vertical ray emanating downwards from

\bar{q} . Let us examine the d -cell C in the arrangement $A(N)$ that is entered by this vertical ray after hitting these k hyperplanes. Figure 7.20 gives an illustration for $k = 2$. Notice that the level of this cell C with respect to $(0, \dots, 0, \infty)$ is k . Obviously, q is contained within the vertical projection of C onto $R^d = \{x_{d+1} = 0\}$. Conversely, every point contained within this vertical projection of C has the same set of k closest neighbors in \hat{N} —this is equivalent to saying that all points in the interior of C have the same set of k hyperplanes above them. It thus follows that the k th order Voronoi regions in R^d are just vertical projections of the d -cells in $A(N)$ with level k . Thus, in Figure 7.20, the second order Voronoi regions $\text{vor}(a, b)$, $\text{vor}(b, d)$, and $\text{vor}(d, c)$ are projections of the cells A , B , and C , respectively.

The preceding connection has an important algorithmic application: The construction of the k th order Voronoi diagram of \hat{N} is reducible to the construction of the d -cells in $A(N)$ with level k . In algorithmic applications, it also helps to note that the hyperplanes in N that arise in this connection are *nonredundant*. By this, we mean that they all support the convex polytope $C_0(N)$ in $A(N)$ surrounding the base point; in this scenario, the base point o is at $(0, \dots, 0, \infty)$. Nonredundance here has the following interesting im-

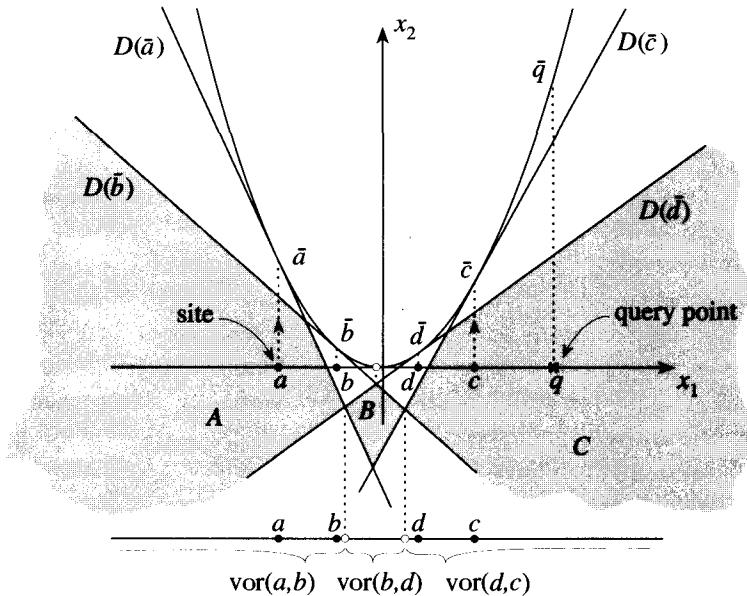


Figure 7.20: Second order Voronoi diagram via projection.

plication: The intersection of $C_k(N)$ with any hyperplane $P \in N$ gives rise to a k -complex in the $(d - 1)$ -dimensional arrangement $P \cap A(N)$. Let us elaborate what we mean by that. Let P be any hyperplane in N . Due to nonredundance, P actually supports the convex polytope $C_0(N)$ surrounding the basepoint o . Let o_P be any point in the interior of the facet of $C_0(N)$ that is supported by P . Let N_P be the set of intersections of the hyperplanes in $N \setminus \{P\}$ with P . Consider the $(d - 1)$ -dimensional arrangement $A(N_P) = A(N) \cap P$. We shall denote the k -complex of $A(N_P)$ with respect to the basepoint o_P by $C_k(N_P)$. The level $L_k(N_P)$ is defined similarly. Then,

Lemma 7.8.1 $C_k(N_P) = P \cap C_k(N)$ and $L_k(N_P) = P \cap L_k(N)$.

Proof. For the arrangements that arose above in connection with Voronoi diagrams, we assumed that the base point o was located at $(0, \dots, 0, \infty)$. However, the definition of $C_k(N)$ and $L_k(N)$ does not change if we let o be anywhere in the interior of the convex polytope $C_0(P)$. This is because the level of a point $p \in R^d$ with respect to o remains the same regardless of where o is actually located in the interior of $C_0(N)$. The assertion in the lemma becomes obvious if we take o to be arbitrarily close to o_P . \square

So far, we have described three closely related notions: levels in arrangements, higher order Voronoi diagrams, and k -sets. Now let us turn to their combinatorial complexity. Given a set of n points in R^d , what is the maximum number of k th order Voronoi regions (or k -sets) induced by these points? As we have already seen, this question can be reduced to a more general question of estimating the number of cells with level k in an arrangement of n hyperplanes. So we shall only concentrate on the latter question. It is, unfortunately, a rather formidable question, which has no satisfactory answer so far. In contrast, estimating the number of cells with level less than or equal to k , rather than exactly k , turns out to be much easier. We shall soon show that the size of any k -complex $C_k(N)$ in R^d is $O((k + 1)^{\lceil d/2 \rceil} n^{\lfloor d/2 \rfloor})$. When $k = 0$, this reduces to the asymptotic form of the upper bound theorem for convex polytopes (Section 7.2.2).

So let N be any set of n hyperplanes in R^d . Without loss of generality, we can assume that the hyperplanes are in general position. Otherwise, perturb the hyperplanes in N infinitesimally. This can only increase the size of $C_k(N)$ (this can be proved as in Exercise 7.0.1). It suffices to estimate the number of vertices in $C_k(N)$, because this bounds the total size of $C_k(N)$ within a constant factor, for a fixed d (Exercise 6.0.2). Let $w(l)$ denote the number of vertices in $A(N)$ with level l . The total number of vertices in $C_k(N)$ is thus $\sum_{l=0}^k w(l)$.

Theorem 7.8.2 $\sum_{l=0}^k w(l) = O((k + 1)^{\lceil d/2 \rceil} n^{\lfloor d/2 \rfloor})$, for $k \geq 0$.

Proof. For $k = 0$, this follows from the upper bound theorem for convex polytopes. So assume that $k > 0$. Randomly choose n/k hyperplanes from N . Let $\phi(k)$ be the expected number of vertices on the convex polytope (surrounding the base point) formed by these hyperplanes. By the upper bound theorem for convex polytopes, $\phi(k) = O((n/k)^{\lfloor d/2 \rfloor})$. A fixed vertex $v \in A(N)$ with level l occurs on this convex polytope iff all d hyperplanes containing it are chosen and the l hyperplanes separating it from the base point are not chosen. Let us denote the probability of this event by $p(l)$. Since each hyperplane in N is chosen with probability $1/k$, we have, for $l \leq k$,

$$p(l) \approx \left(\frac{1}{k}\right)^d \left(1 - \frac{1}{k}\right)^l \geq \left(\frac{1}{k}\right)^d \left(1 - \frac{1}{k}\right)^k \approx \left(\frac{1}{k}\right)^d \frac{1}{e}. \quad (7.20)$$

The first asymptotic equality here should be intuitively obvious. It can be easily proven by writing down an exact expression for $p(l)$ in terms of binomial coefficients and then using Stirling's approximation. We leave this calculation to the reader.

For $v \in A(N)$, let $pr(v)$ denote the probability that v occurs on the previous randomly formed convex polytope. By linearity of expectation, $\phi(k)$ is obtained by summing $pr(v)$ over all vertices in $A(N)$. Since the number of vertices with level l is $w(l)$, this is equivalent to saying that

$$\phi(k) = \sum_{l=0}^n p(l)w(l).$$

Hence,

$$\phi(k) \geq \sum_{l=0}^k p(l)w(l) = \Omega\left(\frac{1}{k^d} \sum_{l=0}^k w(l)\right), \quad (7.21)$$

using (7.20). The theorem follows because we have already seen that $\phi(k) = O((n/k)^{\lfloor d/2 \rfloor})$, for $k > 0$. \square

Exercises

7.8.1 Show that the edge skeleton of $C_k(N)$ is connected, assuming that the hyperplanes in N are in general position.

***7.8.2** Show that the bound in Theorem 7.8.2 becomes tight when the convex polytope $C_0(N)$ is dual-cyclic.

7.8.3 Given a j -skeleton of $C_k(N)$, where $j \geq 1$, show that its $(j+1)$ -skeleton can be obtained in time proportional to the size of the skeleton. Assume that each node in the j -skeleton is associated with the set of hyperplanes containing the corresponding face.

***7.8.4** Let $C_k(N)$ be a two-dimensional k -complex. Let P be any line not in N . Define the upper zone of $C_k(N)$, with respect to P , to be the set of edges in $C_k(N)$ (1) which are adjacent to the 2-faces in $C_k(N)$ intersecting P , and (2) which either lie above P or intersect P . (By above P , we mean the side of P not containing the base point o.) Show that the total size this upper zone is bounded within a constant factor by the number of edges in $C_k(N)$ intersecting P plus the number of edges in $L_k(N)$ above P . When $k = n$, this is equivalent to the Zone Theorem for line arrangements (in this case $L_n(N)$ is empty). (Hint: Generalize the proof of the Zone Theorem for line arrangements.)

****7.8.5** Consider the following series associated with the arrangement $A = A(N)$. For a real number $s \geq 0$, define

$$\theta(s, k, A) = \sum_{l=0}^k \frac{w(l)}{(l+d)^s},$$

where $w(l)$ is the number of vertices in $A(N)$ with level l (N.B.: This definition is slightly different from the one in Section 7.4 in that we are replacing s by $d - s$.) For $1 \leq k \leq n$, show the following:

- (a) $\theta(s, k) = O(n^{\lfloor d/2 \rfloor} k^{\lceil d/2 \rceil - s})$, for $s < \lceil d/2 \rceil$.
- (b) $\theta(\lceil d/2 \rceil, k) = O(n^{\lfloor d/2 \rfloor} \log k)$.
- (c) $\theta(s, k) = O(n^{\lfloor d/2 \rfloor})$, for $s > \lceil d/2 \rceil$.

A special case of (a), for $s = 0$, says that $\theta(0, k) = \sum_{l=0}^k w(l) = O(n^{\lfloor d/2 \rfloor} k^{\lceil d/2 \rceil})$. This is precisely Theorem 7.8.2.

More generally, for $1 \leq L \leq k \leq n$, and a real number $s \geq 0$, define

$$\theta_L(s, k, A) = \sum_{l=0}^L w(l) + \sum_{l>L}^k \left(\frac{L+d}{l+d} \right)^s w(l).$$

For any $1 \leq L \leq k \leq n$, show the following:

- (a) $\theta_L(s, k) = O(L^s k^{\lceil d/2 \rceil - s} n^{\lfloor d/2 \rfloor})$, if $s < \lceil d/2 \rceil$.
- (b) $\theta_L(\lceil d/2 \rceil, k) = O(L^{\lceil d/2 \rceil} n^{\lfloor d/2 \rfloor} \log (\min \{ k, n/L \}))$.
- (c) $\theta_L(s, k) = O(L^{\lceil d/2 \rceil} n^{\lfloor d/2 \rfloor})$, for $s > \lceil d/2 \rceil$.

†**7.8.6** The critical behavior observed in the preceding exercise in the neighborhood of $\lceil d/2 \rceil$ supports a plausible conjecture that $w(k)$ is $O(n^{\lfloor d/2 \rfloor} k^{\lceil d/2 \rceil - 1 + \epsilon})$, for every $\epsilon > 0$, where the constant factor is allowed to depend on ϵ . Prove or disprove this conjecture.

7.8.7 (Shallow cutting lemma) Prove the following generalization of Exercise 7.7.6. Let N be a set of n hyperplanes in R^d . Show that there exists a $(1/r)$ -cutting of $O(r^{\lfloor d/2 \rfloor} (kr/n + 1)^{\lceil d/2 \rceil})$ size covering the k -complex $C_k(N)$.

7.8.1 Incremental construction

We have seen that the size of $C_k(N)$ is $O((k+1)^{\lceil d/2 \rceil} n^{\lfloor d/2 \rfloor})$. In this section, we shall give a randomized incremental and on-line algorithm for constructing

$C_k(N)$ whose expected running time is optimal, i.e., $O((k+1)^{\lceil d/2 \rceil} n^{\lfloor d/2 \rfloor})$, with an extra $O(n \log n)$ additive term, for $d \leq 3$. As a corollary, we also get an algorithm for constructing Voronoi diagrams of order 1 to k of n sites in R^d in expected $O(k^{\lceil (d+1)/2 \rceil} n^{\lfloor (d+1)/2 \rfloor})$ time, with an extra $O(n \log n)$ additive term, for $d = 2$. This follows because we have already shown that the construction of Voronoi diagrams of order 1 in k in R^d is reducible to the construction of a k -complex in R^{d+1} . The hyperplanes in N that arise in this latter connection are nonredundant. Hence, for the sake of simplicity, we shall assume in this section that the hyperplanes in N are nonredundant. This means all of them support the convex polytope $C_0(N)$. In the exercises, we shall sketch how this assumption can be removed.

We shall only construct the 2-skeleton of $C_k(N)$, that is, the set of vertices, edges and 2-faces of $C_k(N)$ together with the adjacencies among them. This suffices because the full facial lattice of $C_k(N)$ can be recovered from its 2-skeleton in linear time (Exercise 7.8.3). In what follows, we shall denote the 2-skeleton of $C_k(N)$ by $D_k(N)$. We shall also find it convenient to assume, for the sake of simplicity, that $C_k(N)$ is always bounded. This can be ensured, as in the bounding box trick of Section 3.2, by initially adding a set of hyperplanes bounding a large symbolic cube approaching infinity, and subsequently, restricting everything within this cube. Alternatively, the following algorithm can be easily modified so that the boundedness assumption is never needed.

Our algorithm will be randomized and on-line. Initially, we choose a random subset $N^d = \{S_1, \dots, S_d\}$ of d hyperplanes from N , and build $D_k(N^d)$. Next, we add one hyperplane at a time in random order to get a sequence of complexes

$$D_k(N^d), D_k(N^{d+1}), \dots, D_k(N^i), D_k(N^{i+1}), \dots, D_k(N^n) = D_k(N),$$

where N^i denotes the set $\{S_1, \dots, S_i\}$ of the first i randomly chosen hyperplanes, and $D_k(N)$ is precisely the complex we wanted to build.

Before proceeding further, it is illuminating to consider the two extreme cases— $k = 0$ and $k = n$ —which we have already treated. In a sense, the algorithm in this section interpolates between these two extremes. The case $k = 0$ corresponds to the case of a convex polytope. We have already given a randomized on-line algorithm for this case (Section 7.3). In fact, the general algorithm will use this algorithm for $k = 0$ as a subroutine. When $k = n$, $C_k(N)$ is the full arrangement. In this case, our algorithm will reduce to the incremental algorithm for arrangements (Section 6.1) except for two differences: (1) For arrangements, hyperplanes could be added in any order; now they need to be added in random order. (2) In Section 6.1, we maintained the full facial lattice; now we shall only maintain the 2-skeleton.

Let us now describe the addition of $S = S_{i+1}$ to $D_k(N^i)$ in more detail. We shall assume that each face in $D_k(N^i)$ is associated with its current level. Our goal is to obtain $D_k(N^{i+1})$ from $D_k(N^i)$ quickly. By our nonredundancy assumption, S must intersect the convex polytope $C_0(N^i)$. Let us assume that we know an edge of this convex polytope intersecting S . How do we determine such an edge? Well, this is easy, because we are using the algorithm for the special case $k = 0$ as a subroutine. This algorithm identifies all edges of the convex polytope $C_0(N^i)$ that are affected during the addition of S . In particular, it can certainly provide us with one edge of $C_0(N^i)$ that intersects S . Let us denote this edge by a . After this, we proceed in the following steps.

Split $D_k(N^i)$

In this step, we determine all edges and 2-faces in $D_k(N^i)$ intersecting S and split them along S (Figure 7.21(a)). Let us see how. First, notice that $D_k(N^i) \cap S$ forms a connected graph. Indeed, by Lemma 7.8.1, $C_k(N^i) \cap S$ forms a k -complex within S and the above graph is just the edge skeleton of this k -complex. Hence, it is connected (Exercise 7.8.1). The edge a provided to us corresponds to the vertex $u = a \cap S$ of $D_k(N^i) \cap S$. Hence, we can construct $D_k(N^i) \cap S$, one edge at a time, by a search beginning at u . We have already described a similar geometric search in Section 6.1, hence, there is no need to describe it any further. The only crucial point is that this search works because the graph under consideration is connected. Note that any edge e in $D_k(N^i) \cap S$ corresponds to the 2-face $f \in D_k(N^i)$ such that $e = f \cap S$. Hence, traveling along e during the above search corresponds to splitting f along S . During this split, we shall be careful to visit only those vertices of f that lie above S , that is, on the side of S not containing the base point o . Why this care is needed will become clear in the analysis of the algorithm.

So far we have split the edges and the 2-faces of $D_k(N^i)$ intersecting S . We also need to form the new 2-faces that correspond to the intersection of S with the 3-faces of $C_k(N^i)$. Though we do not maintain the 3-faces of $C_k(N^i)$ explicitly, this step is easy: It just amounts to extending the edge skeleton of the k -complex $C_k(N^i) \cap S$ to its 2-skeleton. This can be done in time proportional to its size (Exercise 7.8.3).

Let us denote the complex that is obtained at the end of this phase by $\bar{D}_k(N^{i+1})$.

Peeling

In the next step, we remove all those faces of $\bar{D}_k(N^{i+1})$, whose levels have exceeded k (Figure 7.21). Let S^+ denote the half-space bounded by S not containing the base point. It is easy to see that the faces that need to be removed are adjacent to $L_k(N^i) \cap S^+$, assuming it is nonempty—otherwise, this step is redundant. We can visit all faces contained in $L_k(N^i) \cap S^+$

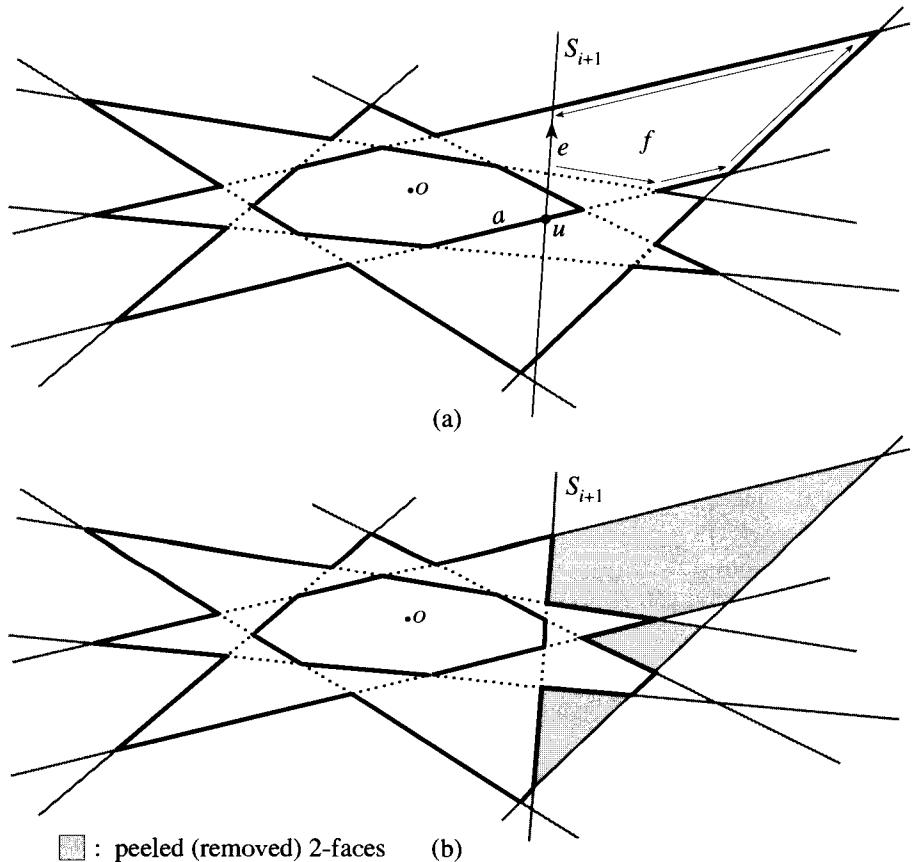


Figure 7.21: Addition of S_{i+1} to $D_k(N^i)$, for $k = 2$: (a) Splitting. (b) Peeling.

by a geometric search starting at the faces in $L_k(N^i) \cap S$ that we already know. This search works because $L_k(N^i) \cap S^+$ is connected and $L_k(N^i) \cap S$ is nonempty, if $L_k(N^i) \cap S^+$ is nonempty. (This is because we ensured that $C_k(N^i)$ is bounded, using the bounding box trick. What would you do if $C_k(N^i)$ were not bounded?)

Once the peeling operation is over, we get $D_k(N^{i+1})$, and we are ready to add the next hyperplane. This finishes the description of the algorithm.

Analysis

In what follows, we shall ignore the expected cost of the convex polytope subroutine, which we used for locating an edge of $C_0(N^i)$ intersecting $S = S_{i+1}$; this expected cost is $O(n^{\lfloor d/2 \rfloor})$, for $d > 3$, and $O(n \log n)$, for $d = 2, 3$.

Every face of $D_k(N^i)$ that is visited during the algorithm is either contained in S^+ or intersects S^+ . (It is crucial here that we do not visit the edges and vertices of $D_k(N^i)$ below S in the splitting step.) By a very pessimistic estimate, the cost of adding $S = S_{i+1}$ to $D_k(N^i)$ is dominated by the size of $S^+ \cap D_k(N^i)$. This size is proportional to the number of newly created vertices in $C_k(N^{i+1})$ plus the number of vertices in $C_k(N^i)$ conflicting with S . We can ignore the conflicting vertices in $C_k(N^i)$ that are removed during the addition of S , because a created vertex can be removed only once. Thus, the amortized cost of adding S is dominated by the number of vertices in $C_k(N^{i+1})$ which are either contained within S or which conflict with S . Let us sum this number over all $S \in N^{i+1}$ and let ϕ denote this sum. Note that the contribution of a fixed vertex $v \in C_k(N^{i+1})$ to ϕ is d plus its level. Thus, ϕ is certainly bounded by $k + d$ times the size of $C_k(N^{i+1})$. By Theorem 7.8.2, the size of $C_k(N^{i+1})$ is $O((k+1)^{\lceil d/2 \rceil} i^{\lfloor d/2 \rfloor})$. Hence $\phi = O((k+1)^{\lceil d/2 \rceil + 1} i^{\lfloor d/2 \rfloor})$.

Now observe that the expected cost of the $(i+1)$ th addition, conditional on a fixed N^{i+1} , is of the order of $\phi/(i+1)$. This is because each hyperplane in N^{i+1} is equally likely to occur as S_{i+1} ; we are using backward analysis here. Now, it follows that the expected cost of the $(i+1)$ th addition is $O((k+1)^{\lceil d/2 \rceil + 1} i^{\lfloor d/2 \rfloor - 1})$. Summing over all i , it follows that the expected running time of the algorithm is $O((k+1)^{\lceil d/2 \rceil + 1} n^{\lfloor d/2 \rfloor})$. This bound is still worse than the $O((k+1)^{\lceil d/2 \rceil} n^{\lfloor d/2 \rfloor})$ bound that we promised at the beginning of this section. The expected running time of the algorithm is, in fact, $O((k+1)^{\lceil d/2 \rceil} n^{\lfloor d/2 \rfloor})$. The proof of this fact is more nontrivial. It is sketched in the following exercise.

Exercises

***7.8.8** Show that the expected running time of the preceding algorithm is, in fact, $O(k^{\lceil d/2 \rceil} n^{\lfloor d/2 \rfloor})$, for $k > 0$, ignoring, as above, the cost of the convex polytope subroutine. Follow these steps:

- Show that the cost of adding S to $D_k(N^i)$ is proportional to the number of newly created vertices plus the number of vertices in the level $L_k(N^i)$ above S , i.e., conflicting with S . (Hint: Use the Zone Theorem for two-dimensional k -complexes (Exercise 7.8.4) to show that the cost of splitting the 2-faces in $D_k(N^i)$ is bounded by the total size of $D_k(N^i) \cap S$ and $L_k(N^i) \cap S^+$.)
- Show that the amortized cost of adding S can be taken to be proportional to the number of newly created vertices. (Hint: If a vertex $v \in L_k(N^i)$ is above S , then its level increases during the addition of S . Hence, v can be charged as in (a) at most d times before it is removed in the peeling operation.)
- Show that, for a fixed vertex $v \in A(N)$, the probability that v is created during the algorithm is $[k+d]_d/[l+d]_d$, where l is the level of v , and assuming $l \geq k$. We say that v is created during the algorithm if it occurs in some $C_k(N^i)$. Here $[x]_d$ denotes the falling factorial $x(x-1) \cdots (x-d+1)$. Conclude that the expected

running time of the algorithm is $O(\sum_{l=0}^k w(l) + \sum_{l>k} w(l)[k+d]_d/[l+d]_d)$, where $w(l)$ denotes the number of vertices in $A(N)$ with level l .

- (d) Using Exercise 7.8.5, show that the value of the above series is $O(k^{\lceil d/2 \rceil} n^{\lfloor d/2 \rfloor})$, for $k > 0$.

***7.8.9** Consider the following query problem. Given a query ray emanating from the base point, the goal is to determine the first k hyperplanes in the given set N intersecting the query ray. (As we have already seen, the nearest k neighbors problem is reducible to this problem.) For $d = 3$, extend the algorithm in this section so that one could answer queries of this form in $\tilde{O}(k \log n)$ time. (Hint: Use history. Specify the link structures carefully.)

****7.8.10** Make the algorithm in this section as well as the one in Exercise 7.8.9 fully dynamic using the dynamic shuffling technique. Show that the expected cost of executing a random (N, δ) -sequence is $O((k+1)^{\lceil d/2 \rceil+1} n^{\lfloor d/2 \rfloor})$ in the first case (for $d > 3$), and $O(k^2 n \log n)$ in the second, where n is the size of N ; the signature δ is arbitrary. Give a similar dynamic algorithm based on dynamic sampling.

***7.8.11** Remove the nonredundancy assumption made in this section as follows. The crucial point is to determine, during the addition of S_{i+1} , an edge of $D_k(N^i)$ intersecting S_{i+1} ; the rest of the algorithm will remain unaffected. Let $\bar{D}_k(N^i)$ denote the canonical triangulation of $D_k(N^i)$. The algorithm should now maintain $\bar{D}_k(N^i)$ at time i . It should also maintain, with each edge of $\bar{D}_k(N^i)$, the list of hyperplanes in $N \setminus N^i$ which intersect that edge. Conversely, it should maintain, with each hyperplane in $N \setminus N^i$, the list of edges in $\bar{D}_k(N^i)$ that it intersects. This automatically tells us all edges of $D_k(N^i)$ intersected by S_{i+1} . Show that the expected running time of this modified algorithm is $O((k+1)^{\lceil d/2 \rceil} n^{\lfloor d/2 \rfloor})$, for $d > 3$, $O(k^2 n \log[n/k])$, for $d = 3$, and $O(kn \log[n/k])$, for $d = 2$.

7.8.2 Single level

So far, we have mainly dealt with the k -complex $C_k(N)$. The reader must be mystified by our silence over the subject of a single level $L_k(N)$. The combinatorial and algorithmic questions regarding a single level $L_k(N)$ are certainly of great interest. Unfortunately, they seem rather formidable at present. Partly, this is due to the fact that the complexity of a single level $L_k(N)$ seems rather hard to estimate, though some progress in this direction has been made recently. It is also of interest to know whether the level $L_k(N)$ can be built in an output-sensitive fashion, that is, roughly in time proportional to its size. A satisfactory solution for this problem is known for $d = 3$. It is also known that $C_k(N)$ can be built in an output-sensitive fashion for arbitrary d . See the bibliographic notes for more on the issues raised here.

Exercises

****7.8.12** Show that the number of k -sets over a set of n points in the plane is $O(\sqrt{kn})$; compare this bound with the $O(kn)$ bound given by Theorem 7.8.2.

****7.8.13** Theorem 7.8.2 implies that the number of $(n/2)$ -sets over n points in R^3 is $O(n^3)$, which is trivially true. Prove a subcubic $O(n^{8/3} \text{polylog}(n))$ bound for this number.

****7.8.14** Let N be a set of planes in R^3 . Assume that all hyperplanes in N are nonredundant. In this case, show that $w(k)$, as defined in Theorem 7.8.2, is $O(k(n - k))$. Conclude that the number of k th order Voronoi regions induced by n sites in the plane is $O(k(n - k))$.

****7.8.15** Give a randomized algorithm to build the k th order Voronoi diagram of n sites in the plane in expected $O(kn^{1+\epsilon})$ time.

***7.8.16** In Section 7.7.2, we gave one search structure for half-space range reporting, with $O(n^{\lfloor d/2 \rfloor + \epsilon})$ space requirement, and $O(k \log n)$ expected query time, where k is the answer size. Improve the query time to $O(k + \log n)$, keeping the other bounds the same as before. (Hint: In the algorithm of Section 7.7.2, use levels of $O(\log r / \log \log r)$ th order instead of convex polytopes; here r is the sample size.)

***7.8.17** (The h -matrix for levels) Here, we shall generalize the h -vector for simple, bounded convex polytopes (Section 7.2.1) to the h -matrix for simple, bounded k -complexes. Let N be a set of hyperplanes in R^d . Assume that they form a simple arrangement. Let $C_k(N)$ be the k -complex defined with respect to a fixed basepoint. Assume that it is bounded. Let $f^l(j)$ denote the number of j -faces in $C_k(N)$ at level $l \leq k$. Define the f -polynomial of $C_k(N)$ as

$$f(x, y) = \sum_{l \leq k, 0 \leq i \leq d} f^l(i) x^l y^i.$$

Fix a nondegenerate linear function z . Orient all edges of $C_k(N)$ in the increasing z -direction. For any vertex $v \in C_k(N)$, define the in-degree of v to be the number of edges of the subcomplex $C_l(N)$ oriented towards v , where $l \leq k$ is the level of v . Define $h_z^l(j)$, where $l \leq k$, to be the number of vertices of $C_k(N)$ at level l with in-degree j with respect to z . When the linear function z is implicitly understood from the context, we shall denote this number by simply $h^l(j)$. The matrix $[h^l(j)]$, $0 \leq l \leq k$, $0 \leq j \leq d$, is called the h -matrix of $C_k(N)$. When $k = 0$, this reduces to the h -vector for the convex polytope $C_0(N)$. It is also convenient to define the associated h -polynomial of $C_k(N)$ by

$$h(x, y) = \sum_{0 \leq l \leq k, 0 \leq i \leq d} h^l(i) x^l y^i.$$

Note that the definition of the h -matrix (and the h -polynomial) depends upon the linear function z .

- (a) But show that the h -matrix is independent of the choice of z , assuming that $C_k(N)$ is bounded and simple. For this, show that

$$f(x, y) = \sum_{0 \leq i \leq d, l \leq k} h^l(i) x^l (1+y)^i (1+xy)^{d-i} \pmod{x^{k+1}}.$$

Here $\pmod{x^{k+1}}$ means that the terms that are divisible by x^{k+1} are ignored. It follows that the f -matrix $[f^l(i)]$, $l \leq k$, $i \leq d$, is related to the h -matrix

$[h^l(i)]$, $l \leq k$, $i \leq d$, by a linear transformation. Show that this transformation is invertible.

- (b) Show that: $h^l(i) = h^l(d - i)$, for all $l \leq k$ and $0 \leq i \leq d$. In addition, $h^l(0) = \binom{l+d-1}{d-1}$.
- (c) Conclude from (b) that, for every simple complex $C_l(N)$ (possibly unbounded), $h^l(0) \leq \binom{l+d-1}{d-1}$, for any non-degenerate linear function z . Show that this implies the Upper Bound Theorem for convex polytopes (the exact form in Section 7.2.3). (Hint: Use duality in linear programming.)

By (a), each $f^k(i)$ can be expressed as a positive linear combination of $h^l(j)$'s, $l \leq k$. Hence, good bounds on $h^l(j)$ can lead to good bounds on $f^k(i)$. We already know that this strategy works for convex polytopes (Section 7.2.2). Thus, one way of obtaining good bounds on $f^k(i)$ may be to obtain good bounds on $h^l(j)$, $j \leq d$, $l \leq k$. Show that

$$\sum_{l=0}^k h^l(j) = O(k^{\lceil d/2 \rceil} n^j), \quad j \leq \lfloor d/2 \rfloor.$$

It is plausible to conjecture that, for $l > 0$,

$$h^l(j) = O(l^{\lceil d/2 \rceil - 1 + \epsilon} n^j), \quad j \leq \lfloor d/2 \rfloor,$$

where the constant within the Big-Oh notation is allowed to depend on ϵ ; for $l = 0$, (b) already proves something stronger. In conjunction with the preceding relations, $h^l(j) = h^l(d - j)$, and the linear relation between the f -matrix and the h -matrix, this would imply that $f^k(i) = O(k^{\lceil d/2 \rceil - 1 + \epsilon} n^{\lfloor d/2 \rfloor})$, for all $k \leq n$ and $i \leq d$. The same bound would then also hold for the number of k -sets in the dual setting.

Bibliographic notes

Linear programming has been one of the most extensively studied subjects in computer science; see [198] for the latest survey of this area. There are polynomial time algorithms [124, 125] for linear programming whose running times depend polynomially on n , d , and the bit size of the input. An outstanding open problem is whether there exists a *strongly* polynomial algorithm, i.e., an algorithm whose running time is also polynomial in the real model of computation (in n and d only). For a fixed dimension d , Megiddo [154] gave a deterministic $O(n)$ time algorithm for linear programming, where n is the number of constraints, and the constant factor within the Big-Oh notation depends double-exponentially on d . Several people improved this dependence on d , e.g., see [67, 85]. Clarkson [67] gave a randomized algorithm with the best known quadratic dependence on d , for small d (Exercise 7.1.6). The randomized incremental algorithm (Section 7.1), with $d!$ factor in the bound, was given by Seidel [205]. An improved algorithm was given by Matoušek, Sharir, and Welzl [144], making the dependence on d subexponential (Exercise 7.1.5). Independently, a similar algorithm (in a dual setting) was given by Kalai [122]. For derandomization of Seidel's and Clarkson's linear programming algorithms, see [54]. A subquadratic bound on the cost of detecting all nonredundant half-spaces (Exercise 7.1.4) was proved by Matoušek and Schwarzkopf [142].

The combinatorial structure of convex polytopes has been extensively studied for a very long time; we shall only provide two survey references here [29, 115]. The proof of Dehn–Sommerville relations [78, 211] in Section 7.2.1 is due to McMullen and Walkup [152]. For a generalization of these relations to compact intersections of manifolds in general position, see Mulmuley [171]. The upper bound theorem for convex polytopes (Section 7.2.3) was proved by McMullen [151]; our proof follows Bronsted [29], who also gives a good historical survey of this result. Cyclic polytopes (Section 7.2.4) were discovered by Carathéodory [31], and rediscovered by Gale [109]; for more on cyclic polytopes, see [29, 115]. A sublinear bound on the size of a convex hull of points drawn from a uniform distribution on a ball (Exercise 7.2.3) was proved by Raynaud [190].

The construction of convex polytopes has been extensively studied in computational geometry since its earliest days, e.g., see [32, 184, 185, 202, 214]. Currently, there are deterministic algorithms that are worst case optimal (cf. Chazelle [40]) or output-sensitive (cf. Seidel [204]). The randomized incremental algorithm for convex polytopes in Section 7.3 is due to Clarkson and Shor [72]. The on-line version based on linear programming (Section 7.3.1) was given by Seidel [205]. The on-line version based on history (Section 7.3.2) is taken from Mulmuley [168]; a dual version can be found in Clarkson, Mehlhorn, and Seidel [70]. The analysis dealing with quasi-output-sensitivity (Section 7.4) is taken from Mulmuley [165]. We shall see in Chapter 10 that it easily carries over to the pseudo-random setting.

The dynamic maintenance of convex polytopes (Section 7.5) is taken from Mulmuley [168]. Independently, Schwarzkopf [200] proposed another dynamic algorithm, with similar running time, for dimensions ≥ 6 . Recently, Clarkson, Mehlhorn, and Seidel [70] have investigated dynamic maintenance of convex hulls further. They give an algorithm with essentially the same insertion procedure as here (in a dual setting), but the deletion procedure is different in the sense that the history is updated globally, instead of step by step through rotations, as done here. For an earlier deterministic algorithm for maintaining planar convex hulls (i.e., $d = 2$), see Overmars and Van Leeuwen [178].

Voronoi diagrams [222], Delaunay triangulations [79], and the related nearest neighbor problem have been a subject of extensive study in computation geometry since its earliest days. For a good historical survey of these notions and their applications, see Aurenhammer [16]. The early methods for constructing planar Voronoi diagrams (i.e., $d = 2$) were direct; see, for example, [103, 209]. We constructed Voronoi diagrams in R^d by constructing a related convex polytope in R^{d+1} (but such methods can also be directly translated to the setting of Voronoi diagrams without much difficulty (Exercise 7.6.2), so this is only a conceptual device). This approach originates from Brown [30]. The connection based on the paraboloidal transform is due to Edelsbrunner and Seidel [93]. That the worst-case bound provided by the Upper Bound Theorem is actually attained for Voronoi diagrams (Exercise 7.6.1) was proved by Seidel [203]. The linear bound on the expected size of a Voronoi diagram, when the sites are drawn from a uniform distribution on a unit ball (Exercise 7.6.4), was proved by Dwyer [84].

A randomized static search structure for the nearest neighbor problem with polylogarithmic query time was given by Clarkson [68], and this was generalized by Clarkson and Shor [72] to the nearest k -neighbors problem and half-space

range search problem (Sections 7.7.1 to 7.7.3, Exercise 7.8.16). Mulmuley [167] dynamized these search structures (actually, their simplified versions given in Section 7.7.4) using dynamic sampling, so that their expected performance on random updates is quasi-optimal. Agarwal and Matoušek [2] have recently given a deterministic dynamization scheme that is worst-case quasi-optimal. They also give space-time trade-offs. An $O(n \log n)$ algorithm for all nearest neighbors problem (Exercise 7.7.3) was given by Vaidya [216]. A randomized linear time algorithm for finding the closest pair of sites from a given set of sites was given by Rabin [188] in his seminal paper on randomized algorithms. A simpler randomized algorithm was given by Khuller and Matias [126], which was dynamized by Golin et al. [111] (Exercise 7.7.2). Recently, a randomized search structure with expected $O(n)$ space requirement and expected $O(\log n)$ query time (Exercise 7.7.4) has been given by Arya and Mount [14] for the *approximate* nearest neighbor problem.

The notion of higher order Voronoi diagrams was introduced by Shamos and Hoey [209]. Aurenhammer [16] gives a good survey of higher order Voronoi diagrams and related notions. The k -levels and the related k -sets have been investigated by several authors, e.g., [8, 55, 75, 96, 93, 101, 112, 224]. The bound on the size of a k -complex (Theorem 7.8.2, Exercise 7.8.2) was obtained by Clarkson and Shor [72], after the initial investigation by several authors [8, 55, 75, 96, 93, 101, 112, 224]. Its generalization in Exercise 7.8.5 is from Mulmuley [165]. The Zone Theorem for planar k -complex (Exercise 7.8.4) was proved by Mulmuley [165]. The connection between higher order Voronoi diagrams and levels that is used in Section 7.7.3 is due to Edelsbrunner and Seidel [93]. The shallow cutting lemma (Exercise 7.7.6 and Exercise 7.8.7) was proved by Matoušek [141].

A randomized incremental algorithm for constructing a k -complex in R^d (Section 7.8.1) was given by Mulmuley [165], and it was made fully dynamic (Exercise 7.8.10) in [168]. Another dynamization scheme that works for higher order Voronoi diagrams is given in [168]. For semidynamization, based on Delaunay trees, see [26]. For $d = 2$, a dynamic scheme with optimal space requirement, but with somewhat higher running time, is given by Aurenhammer and Schwarzkopf [17]. Mulmuley [162] gives a deterministic, output-sensitive algorithm for constructing higher order Voronoi diagrams of order 1 to k in R^d ; for $d = 2$, this essentially reduces to the earlier algorithm given by Lee [132]. The tight bound on the size of the k th order Voronoi diagram in the plane (Exercise 7.8.14) is also due to Lee [132].

The complexity issues regarding the k -level, rather than the k -complex, are far more difficult. A subcubic bound on the number of halving sets in R^3 was obtained by Bárány, Füredi, and Lovász [18], and improved further in [11] (Exercise 7.8.13). The $O(n\sqrt{k})$ bound on the number of k -sets in the plane (Exercise 7.8.12) is due to Erdős et al. [101]; some improvement in this bound is accomplished by Pach et al. [181]. For some recent bounds in arbitrary fixed dimension d , see [6]. Agarwal and Matoušek [2] give a quasi-optimal output-sensitive algorithm for constructing a k -level in R^3 . Clarkson [66] gives a quasi-optimal randomized algorithm for constructing the k th order Voronoi diagram in the plane (Exercise 7.8.15). The h -matrix for levels was studied by Mulmuley [170] (Exercise 7.8.17).

Chapter 8

Range search

A large number of problems in computational geometry can be formulated as *range searching problems* (Section 2.1): We are given a set of geometric objects in R^d . A query is given in the form of a connected region in R^d . The problem is to report or count the objects in N intersecting the range. We want to build a search structure that allows this to be done quickly. In this chapter, we shall study several instances of this problem.

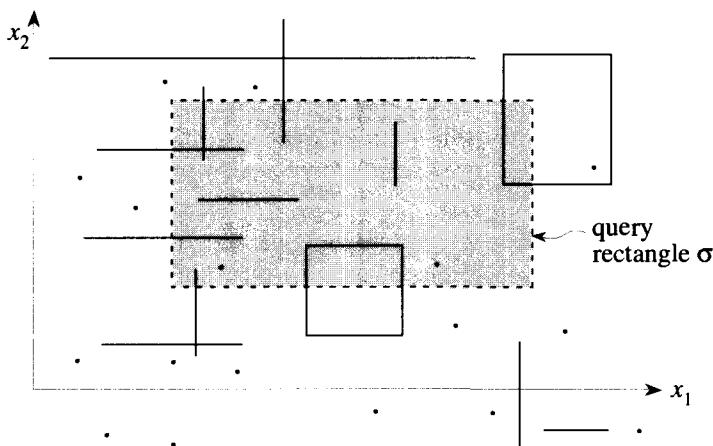


Figure 8.1: Orthogonal intersection search in the plane.

8.1 Orthogonal intersection search

Perhaps the simplest kind of range search is the *orthogonal intersection search*. Here the objects in N are *orthogonal*. By an orthogonal object in R^d , we mean a Cartesian product of d intervals. The intervals can be

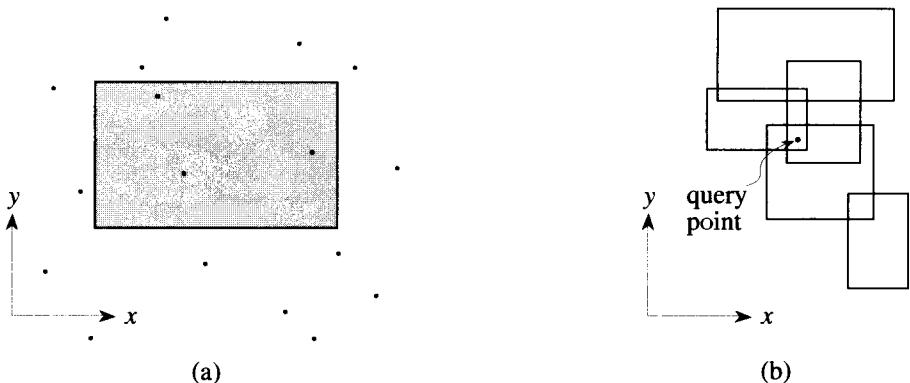


Figure 8.2: (a) Orthogonal range search in the plane. (b) Inverse orthogonal range search in the plane.

degenerate, i.e., points. The simplest orthogonal object is a point. An orthogonal object in R^2 is a point, a vertical or a horizontal segment, or an orthogonal rectangle. The query in this restricted problem is also an orthogonal object. Given a query, the goal is to find all objects in N intersecting the query (Figure 8.1). Though the orthogonal intersection search is a highly restricted problem, it is still interesting in practice. For example, the objects that arise in VLSI design are quite often orthogonal.

The simplest instance of the orthogonal intersection search is the so-called *orthogonal range search*. In this problem, the objects in N are points (Figure 8.2(a)). In the *inverse orthogonal range searching* problem, the range is a point (Figure 8.2(b)).

In this section, we give a randomized search structure for the general orthogonal intersection search problem with $\tilde{O}(n \text{ polylog}(n))$ space requirement and $O(k + \text{polylog}(n))$ query time. Here k denotes the answer size. Count-mode queries are discussed in the exercises.

8.1.1 Randomized segment tree

First, let us consider the simplest case, when $d = 1$. In this case, the objects in N are points or intervals (segments). The query is either a point or a segment (Figure 8.3).

Let M be the set consisting of the points in N and the endpoints of the segments in N . We will show how the skip list on M can be augmented to handle the required queries. This augmented skip list will also be called a randomized segment tree. A skip list is not really a tree, though it can

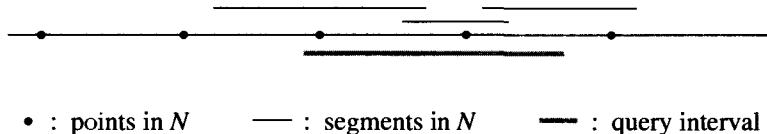


Figure 8.3: Orthogonal intersection search on the line.

be thought of as one (Section 1.4). The terminology is purely for historical reasons.

We build a skip list on M as usual. We also maintain adjacencies among the points in M and the segments in N . Let $M = M_1 \supseteq M_2 \supseteq \dots$ be the underlying gradation of M . Let $H(M_i)$ denote the partition of the real line formed by M_i .

The basis of our method is the following canonical decomposition scheme. Let $\sigma = [x, \bar{x}]$ be any interval on the real line (Figure 8.4). Call an interval $I \in H(M_i)$ a *canonical subinterval* of σ if $I \subseteq \sigma$ and there is no interval $J \in H(M_j)$, $j > i$, such that $I \subseteq J \subseteq \sigma$. We also say that σ covers I canonically. It is possible that the same interval I occurs in several levels of the skip list. In this case, I is covered canonically only in the highest level containing it.

The canonical subintervals cover the whole of σ , excepting the two uncovered end-intervals adjacent to x and \bar{x} (Figure 8.4). These uncovered end-intervals are empty, i.e., they do not contain any point in N . For this reason, they can be ignored.

Observation 8.1.1 An interval σ covers $J \in H(M_i)$ canonically iff σ covers J and the parent interval of J contains an endpoint of σ .

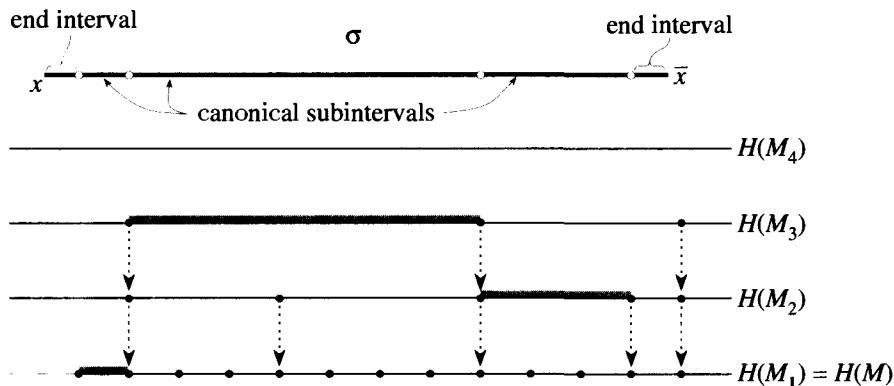


Figure 8.4: Canonical subintervals.

This observation enables us to determine the canonical subintervals of σ quickly. We just examine the children of the intervals that lie on the search paths of its endpoints. By the results in Section 1.4, this takes $\tilde{O}(\log n)$ time, where n is the size of N . It also shows that σ has $\tilde{O}(\log n)$ canonical subintervals.

We are now in position to augment the skip list on M to handle the required queries. With each interval $I \in H(M_i)$, we associate a *canonical covering list* $\text{cover}(I)$ of the segments in N , which cover I canonically. The space requirement of this augmented skip list is obviously $\tilde{O}(n + s \log n)$, where s is the number of segments in N . This follows because each segment in N has $\tilde{O}(\log n)$ canonical subintervals.

Queries are answered as follows. First, consider the case when the query is in the form of a point p . We want to know all segments in N containing p . The point p is contained in a segment $S \in N$ iff it is also contained in some canonical subinterval of S . Hence, we report the segments in the canonical covering lists of the intervals that lie on the search path of p . This takes $\tilde{O}(k + \log n)$ time, where k is the answer size.

Next, consider the case when the query is in the form of an interval $\sigma = [p, \bar{p}]$. The intervals in N containing p or \bar{p} can be reported as before. We also need to report the segments or the points of N that lie within σ . This can be done by locating p in $H(M)$, using our skip list, and then traveling from p to \bar{p} in $H(M)$. Each point in M that is encountered in this travel is either a point in N —in which case, it is reported—or it is an endpoint of a segment in N —in which case, the segment is reported. In this fashion, we report all points of N that lie between p and \bar{p} , and also all segments in N whose endpoints lie between p and \bar{p} . The query time is again $\tilde{O}(k + \log n)$, where k is the answer size.

Each segment in N canonically covers $\tilde{O}(\log n)$ intervals. Hence, the space requirement is $\tilde{O}(n + s \log n)$.

Updates

Let us turn to the updates. We shall only consider additions because deletions are their exact reversals.

First, consider the addition of a new point p to M . Let $M' = M \cup \{p\}$. We add p to the skip list on M as usual (Section 1.4). We also need to build the canonical covering list of each newly created interval J . By Observation 8.1.1, the parent interval of J contains an endpoint of every segment canonically covering J .

With this in mind, let us define the conflict list of any interval I stored in the skip list as the set of points in M contained within I . We shall denote it by $\text{conflict}(I)$. Its size is called the *conflict size* of I . If I is the parent interval of a newly created interval J , then the newly added point

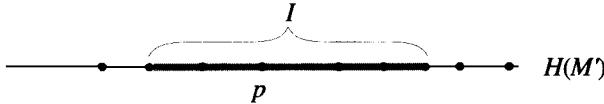


Figure 8.5: Conflict list of an interval.

p is either contained in I or it is adjacent to I . Hence, $\text{conflict}(I)$ can be determined by traveling within the partition $H(M')$ from p to the endpoints of I (Figure 8.5). This takes time proportional to the conflict size of I , ignoring an additive $O(1)$ cost. After this, the canonical covering list of I can be constructed by examining each segment in N that is adjacent to a point in $\text{conflict}(I)$ (Observation 8.1.1).

Now consider the addition of a new segment $S = [x, \bar{x}]$ to N . This is done in two steps. We first add its endpoints to N by the procedure described earlier. Then we determine all intervals in the new skip list that are canonically covered by S . We have already seen that the number of canonical intervals within any segment on the real line is $\tilde{O}(\log n)$ and that they can be determined in the same order of time. We insert S in the canonical covering list of each such interval.

Analysis of the updates

First, consider the addition of a new point p to M . Addition of p to the skip list on M takes logarithmic time with high probability. We only need to estimate the cost of building the canonical covering lists of the newly created intervals. By the preceding discussion, this is clearly proportional to the total conflict size of their parent intervals, ignoring $\tilde{O}(\log n)$ additive term. It follows from the following proposition that the expected value of this total conflict size is $O(\log n)$.

Proposition 8.1.2 *The expected total conflict size of the intervals that are destroyed during the addition of a new point and the intervals that are parents of the destroyed intervals is $O(\log n)$. Similarly, the expected total conflict size of the intervals that are destroyed during the deletion of a point in N is $O(\log n)$.*

Proof. We only consider addition, because deletion can be treated by simply reversing the argument.

Consider the addition of a new point p to N . Let M be, as usual, the set of points in N and the endpoints of the segments in N . Fix a level i . We claim that the expected conflict size of the interval containing p in the i th level is $O(2^i)$. This should be intuitively clear, because the size of N_i is roughly $n/2^i$. A rigorous justification is as follows.

Let $J(i)$ denote the interval in the i th level containing p . Let $r(i)$ be the number of points in M that are contained in the interior of $J(i)$ to the right of p . Similarly, we define $l(i)$ to be the number of points in M that are contained in the interior of $J(i)$ to the left of p . The conflict size of $J(i)$ is thus $r(i) + l(i)$. By symmetry, it suffices to show that the expected value of $r(i)$ is 2^i .

Notice that $r(i) = k$ iff none of the k nearest points in M to the right of p occurs in the i th level. For each point in N , the probability that it occurs in the i th level is $1/2^i$. The probabilities are independent for all points in M . Hence, $r(i)$ is distributed according to the geometric distribution with parameter $1/2^i$ (Appendix A.1.2). This becomes clear if we imagine tossing a pseudo-coin with bias $1/2^i$ for the points in M . Examine the results of the coin tosses for the points on the right hand side of p in the increasing order of their distance from p . Then $r(i)$ is just the number of failures obtained before the first success. It follows that the expected value of $r(i)$ is 2^i , as claimed (Appendix A.1.2).

Thus, for each i , the expected conflict size of the interval in the i th level containing p is $O(2^i)$. This interval, or its child, is destroyed during the addition of p only if p is added to the first $i - 1$ levels. The probability of this event is $1/2^{i-1}$. The coin tosses for p are independent of the coin tosses for the other points in M . Hence, the total conflict size of the intervals that are destroyed during the addition of p , and their parents is

$$\sum_i \frac{1}{2^{i-1}} O(2^i) = \sum_i O(1),$$

where i ranges over all levels. The number of levels is $\tilde{O}(\log n)$. Hence, the proposition follows. \square

It follows that the expected cost of adding a new point to N is $O(\log n)$. Deletion of a point is just the reverse. Its expected cost is also $O(\log n)$. The cost of adding a segment S is dominated by the cost of adding its endpoints, ignoring the $\tilde{O}(\log n)$ cost of determining the intervals canonically covered by S . So the expected cost of adding a segment is also $O(\log n)$. The expected cost of deleting a segment is also $O(\log n)$, because deletion is just the reverse of addition.

The following theorem summarizes the performance of our search structure.

Theorem 8.1.3 *Let N be a set of points and intervals on the real line. A search structure for N can be built in $\tilde{O}(n+s \log n)$ space and $\tilde{O}(n \log n)$ time, where s denotes the number of segments in N . Given a query point or a query interval, all intersecting elements in N can be reported in $\tilde{O}(k + \log n)$ time,*

where k denotes the answer size. The expected cost of adding or deleting an element in N is $O(\log n)$.

Exercises

8.1.1 Give modifications required to handle degeneracies that occur when two points in N are allowed to have the same x -coordinate. The performance bounds should remain the same as before.

8.1.2 The goal in this exercise is to design an alternative randomized segment tree based on randomized binary trees (Section 1.3) instead of skip lists. Let M be the set of points as in this section. Let S_1, S_2, \dots be the points in M in the increasing order of priorities. Let M^j be the set of the first j points S_1, \dots, S_j .

- (a) Recall that each node in a randomized binary tree corresponds to an interval in $H(M^j)$, for some j . Label each node with the corresponding interval. Define its conflict size as the number of points in $M \setminus M^j$ contained in the interval. This is essentially the number of leaves of the subtree rooted at this node.
- (b) Given a query interval σ , define its canonical subintervals with respect to the existing randomized binary tree on M . Each canonical subinterval must label a node of the tree. Show that the number of canonical subintervals of σ is $\tilde{O}(\log n)$.
- (c) Prove the analogue of Proposition 8.1.2.
- (d) Now prove that the performance bounds proved in this section also hold for this randomized segment tree.

8.1.3 Assume that the objects in N are all points. Modify the data structures in this section and Exercise 8.1.2 to handle count-mode queries. Given a query interval σ , it should be possible to count all points of N lying within σ in $\tilde{O}(\log n)$ time. (Hint: Associate with each interval in the skip list its conflict size. Break the query interval into canonical subintervals.)

Describe how this data structure is updated during the addition or deletion of a point in N . Show that an update can be carried out in $\tilde{O}(\log n)$ time.

***8.1.4** Give a deterministic segment tree with a performance comparable to the randomized segment tree in this section. (Hint: Use weight-balanced trees.)

8.1.2 Arbitrary dimension

In this section, we address orthogonal search in arbitrary dimension.

Let N denote the given set of n orthogonal objects in R^d . Throughout this section, we make the following general position assumption. For each i , we assume that the projections of any two orthogonal objects on the x_i -axis do not share a common endpoint. Note that these projections could be intervals or points (degenerate intervals). The general position assumption is removed in the exercises.

We first define our search structure in a static setting and address its dynamization later. The search structure is defined by induction on d . The basis case $d = 1$ has already been treated.

So assume that $d > 1$. Let x_1, \dots, x_d denote the coordinates in R^d . Project each object in N orthogonally onto the x_1 -axis. Let \bar{N} denote the set of these projections. These are either points or intervals on the x_1 -axis.

We build an augmented skip list, as in Section 8.1.1, on \bar{N} . This will be called the *primary* skip list. It is based on a gradation $\bar{N} = \bar{N}_1 \supseteq \bar{N}_2 \supseteq \dots$. Here \bar{N}_i denotes the set of points in its i th level. Let $H(\bar{N}_i)$ denote the partition of the x_1 -axis formed by \bar{N}_i .

Fix an interval $I \in H(\bar{N}_i)$. Fix an object $\gamma \in N$. The projection of γ onto the x_1 -axis is an interval $[p, p']$. This interval can be degenerate, i.e., it can be a point. In that case, p coincides with p' . We say that γ conflicts with I iff I contains p or p' . We define $\text{conflict}(I)$ as the set of objects in N in conflict with I (Figures 8.1 and 8.6). The size of I is called the *conflict size* of I . Similarly, we define the canonical covering list, $\text{cover}(I)$, as the set of those objects in N whose projections onto the x_1 -axis cover I canonically (Figures 8.1 and 8.7). The definition of $\text{cover}(I)$ is dependent on the existing skip list. Not all objects whose projections cover I belong to $\text{cover}(I)$.

We associate with I two recursively defined $(d-1)$ -dimensional *secondary* search structures. One will be for the projections of the objects in $\text{conflict}(I)$ onto the coordinate hyperplane $x_1 = 0$ (Figure 8.6(b)). The other will be for the similar projections of the objects in $\text{cover}(I)$ onto the coordinate hyperplane $x_1 = 0$ (Figure 8.7(b)).

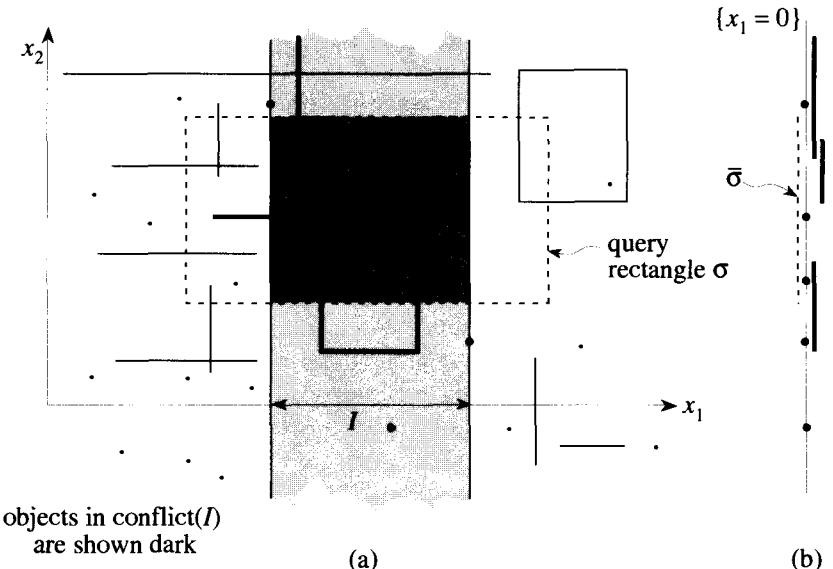


Figure 8.6: Conflict list of an interval.

Let us turn to the query. Let σ denote the orthogonal query object. We want to report all objects in N intersecting σ . Let γ be any orthogonal object in N . Denote the projections of σ and γ onto the hyperplane $x_1 = 0$ by $\bar{\sigma}$ and $\bar{\gamma}$, respectively. Denote their projections onto the x_1 -axis by $[p, p']$ and $[q, q']$, respectively. Here p can coincide with p' and similarly q can coincide with q' . The query object σ intersects γ iff $\bar{\sigma}$ intersects $\bar{\gamma}$ and $[p, p']$ intersects $[q, q']$. We make the following trivial observation:

Observation 8.1.4 *The segment $[p, p']$ intersects $[q, q']$ iff one or both of the following conditions hold:*

1. *p or p' is contained in $[q, q']$, and, hence, in some (closed) canonical subinterval of $[q, q']$.*
2. *q or q' is contained in $[p, p']$, and, hence, in some (closed) canonical subinterval of $[p, p']$.*

This observation leads to the following procedure for handling a query. We determine the intervals in the primary skip list containing p or p' . These are the intervals that lie on the search path of p or p' . Their number is $\tilde{O}(\log n)$. For each such interval I , we report all objects in $\text{cover}(I)$ whose projections on the coordinate hyperplane $x_1 = 0$ intersect $\bar{\sigma}$ (Figure 8.7). This is done with the help of the secondary search structure associated with $\text{cover}(I)$.

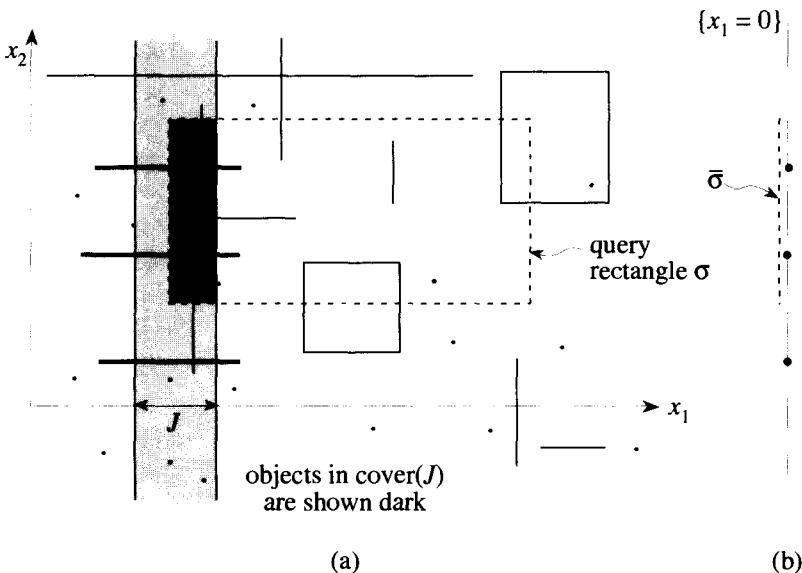


Figure 8.7: Canonical covering list of an interval.

Similarly, we determine all intervals in the primary skip list that are canonically covered by $[p, p']$. The number of such intervals is $\tilde{O}(\log n)$. For each such interval J , we report all objects in $\text{conflict}(J)$ whose projections onto the coordinate hyperplane $x_1 = 0$ intersect $\bar{\sigma}$ (Figure 8.6). This is done with the help of the secondary search structure associated with $\text{conflict}(J)$.

Analysis

Let us turn to the query time. Note that more than one condition in Observation 8.1.4 can be satisfied simultaneously. Hence, an object in N intersecting the query object may be reported by more than one, but obviously, by only a bounded number of $(d - 1)$ -dimensional searches. As d is fixed, each intersecting object is reported $O(1)$ times. Ignoring the cost of reporting, the remaining cost of a query is $\tilde{O}(\log^d n)$. This follows because each d -dimensional query is broken into $\tilde{O}(\log n)$ $(d - 1)$ -dimensional queries.

What is the space requirement of our data structure? Our data structure is a hierarchy of skip lists with depth d . For a fixed object $\gamma \in N$, let us count the number of skip lists involving that object, or, more precisely, its iterated projection. When $d = 1$, γ is obviously involved in only one skip list. When $d > 1$, $\tilde{O}(\log n)$ intervals in the primary skip list contain γ in their conflict lists or canonical covering lists. Hence, γ is involved in $\tilde{O}(\log n)$ $(d - 1)$ -dimensional secondary search structures. By induction, γ is involved in $\tilde{O}(\log^{d-1} n)$ skip lists. The expected size of each such skip list can be bounded by Theorem 8.1.3. It is $O(n' + s' \log n')$, where n' is the number of objects involved in this skip list and s' is the number of involved objects that are not points.¹ Let s denote the number of objects in N that are not points. Since each object in N is involved in $\tilde{O}(\log^{d-1} n)$ skip lists, it follows, by summation, that the expected total space requirement of our data structure is $O(n \log^{d-1} n + s \log^d n)$. In fact, the space requirement is $\tilde{O}(n \log^{d-1} n + s \log^d n)$. This is proved in the following exercises.

It also follows that the total cost of building our search structure is $\tilde{O}(n \log^d n)$. This is because, with high probability, each skip list can be built in time proportional to the number of involved objects, within $O(\log n)$ factor. As each object in N is involved in $\tilde{O}(\log^{d-1} n)$ skip lists, the claimed bound follows.

Dynamization

Let us now dynamize our data structure. The basic idea is the same as in dynamic sampling (Section 5.4): If N denotes the set of existing (undeleted) objects at any given time, then our search structure at that time will look as

¹Actually, s' here can be the number of involved objects whose relevant projections are segments; but this number is obviously less than the number of involved objects that are not points.

if it were constructed by applying the previous static procedure to the set N . This implies that the bound on the query cost proved in the static setting carries over to the dynamic setting.

Consider the addition of a new object σ to N . Let the projection of σ on the x_1 -axis be $[p, p']$, where p can coincide with p' . We add the points p and p' to the primary skip list in the usual fashion. The crucial issue is the update of the secondary search structures. As in Section 8.1.1, we can determine the conflict list of each newly created interval in the primary skip list in time proportional to its conflict size, ignoring an additive $O(1)$ term. By Observation 8.1.1, the canonical covering list of a newly created interval can be obtained by examining each object in the conflict list of its parent interval. For this reason, we also compute conflict lists of the parents of the newly created intervals. By Proposition 8.1.2, it follows that the expected total size of the conflict lists and canonical covering lists of all newly created intervals is $O(\log n)$.

We compute the $(d - 1)$ -dimensional secondary search structures associated with each newly created interval J from scratch. This is done using the static procedure already described. Ignoring $\tilde{O}(\log^{d-1} n)$ factor, this takes time proportional to the total size of $\text{conflict}(J)$ and $\text{cover}(J)$. We have already seen that the expected total size of these lists for all newly created intervals is $O(\log n)$.

Hence, the expected cost of adding a new object is $O(\log^d n)$. Deletion is exactly the reverse. Its expected cost is also $O(\log^d n)$.

We have thus proved the following.

Theorem 8.1.5 *Let N be a given set of n orthogonal objects in R^d . Let s denote the number of objects in N that are not points. A dynamic search structure of $\tilde{O}(n \log^{d-1} n + s \log^d n)$ size can be maintained so that the cost of a report-mode orthogonal intersection query is $\tilde{O}(k + \log^d n)$, where k denotes the answer size. The expected cost of adding or deleting an orthogonal object in N is $O(\log^d n)$.*

In a reasonable model of computation, it can be shown that the bounds in this theorem are optimal in the dynamic setting (see the bibliographic notes). We shall not prove this lower bound here.

Exercises

8.1.5 Remove the general position assumption made at the beginning of this section.

8.1.6 Verify that the space requirement of the search structure in this section is, in fact, $\tilde{O}(n \log^{d-1} n + s \log^d n)$, i.e., the bound holds with high probability. (Hint: For each object $\gamma \in N$ and a skip list ϕ (primary or secondary) in the search structure, let $X(\phi, \gamma)$ denote the number of levels in ϕ that (the projection of

γ is involved in. We know that $X(\phi, \gamma)$ is distributed according to a geometric distribution. The total space requirement of our search structure is proportional to $\sum X(\phi, \gamma)$. Observe that each object can be involved in only $\tilde{O}(\log^{d-1} n)$ skip lists and then apply the Chernoff bound for the sum of independent, geometric distributions (Appendix A.1.2.)

8.1.7 Assume that the objects in N are points. Modify the data structure in this section to handle count-mode queries in $\tilde{O}(\log^d n)$ time. (Hint: See Exercise 8.1.3.) What difficulties arise if the objects are not points?

8.1.8 Give an alternative, based on randomized binary trees, to the data structure in this section. Prove the same performance bounds. (Hint: See Exercise 8.1.2.)

8.1.9 Give a deterministic alternative to the data structure in this section, using the deterministic segment tree in Exercise 8.1.4. Prove similar performance bounds.

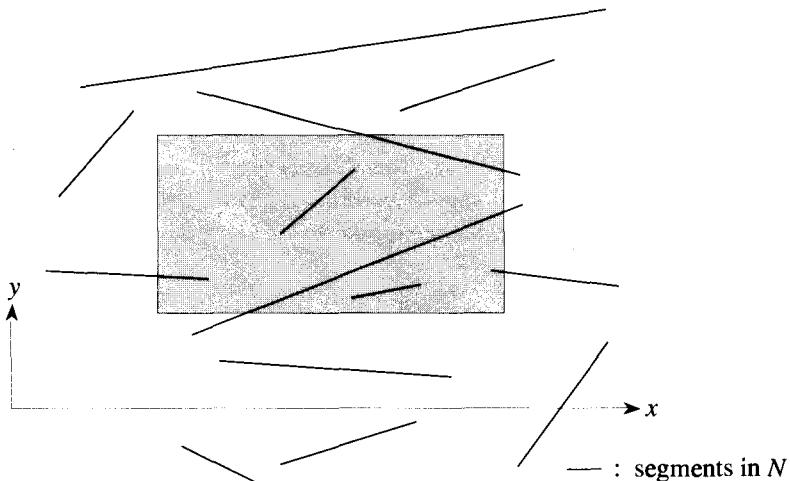


Figure 8.8: Orthogonal search over nonintersecting segments.

8.2 Nonintersecting segments in the plane

In the previous section, we gave an efficient solution to the range searching problem when the objects are orthogonal. Unfortunately, the objects that arise in practice are not always orthogonal. In this section, we consider the simplest nonorthogonal range searching problem. We are given a set N of any nonintersecting segments in the plane. But the query range is orthogonal, i.e., it is either a vertical or a horizontal segment or an orthogonal rectangle. The problem is to report all segments in N that intersect the range. This problem occurs quite frequently in computer graphics. There the query is an orthogonal view window. The problem is to figure out which segments are visible within the view window (Figure 8.8).

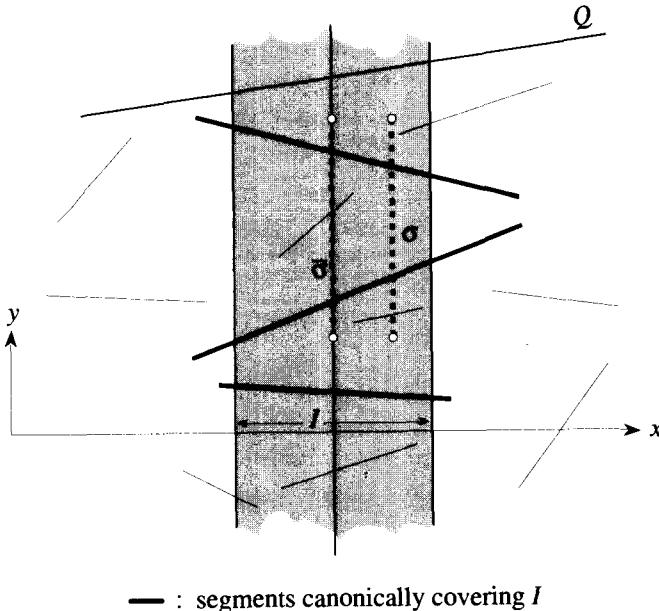


Figure 8.9: A canonical covering list of an interval.

One could obviously generalize this problem to arbitrary dimension, by letting the objects in N be polyhedra in arbitrary dimension. This general problem is, however, far more difficult.

In this section, we give an efficient randomized search structure for orthogonal queries over a set of nonintersecting segments. For the sake of simplicity, we shall assume that no two endpoints of the segments in N share a common x or y -coordinate. This nondegeneracy assumption is removed in the exercises.

First, we give a data structure that can only handle vertical segments as queries. The data structure for handling horizontal queries is analogous. We shall address rectangular queries later.

Project the endpoints of all segments in N onto the x -axis. Let \bar{N} be the set of projections on the x -axis. Build a randomized segment tree, i.e., an augmented skip list, on \bar{N} (Section 8.1.1). With each interval I in this primary skip list, we associate a list of the segments in N whose projections cover I canonically. This list is denoted by $\text{cover}(I)$. Figure 8.9 gives an illustration. Note that $\text{cover}(I)$ does not contain every segment whose projection covers I —such as Q in Figure 8.9—but only those which cover I canonically. Thus, $\text{cover}(I)$ depends on the existing skip list. The segments in $\text{cover}(I)$ can be ordered linearly in the increasing y -direction in the obvious fashion.

With each I , we associate a *secondary* skip list that corresponds to this linear order on $\text{cover}(I)$. To be absolutely precise, the secondary skip list is defined over the intersections of the segments in $\text{cover}(I)$ with the vertical line through a fixed point of I , say, its midpoint.

Queries

Queries are answered as follows. Given a vertical query interval σ with x -coordinate a , we follow the search path for a in the primary skip list. If a segment in N intersects σ , a is contained in its projection, and hence in some canonical subinterval of the projection. So we only need to figure out, for each interval I lying on the search path of a , the segments in $\text{cover}(I)$ intersecting σ . This is equivalent to the following one-dimensional problem. Denote the projection of σ onto the vertical line passing through the midpoint of I by $\bar{\sigma}$. Our problem is equivalent to figuring out which segments in $\text{cover}(I)$ intersect $\bar{\sigma}$. This can be done with the help of the secondary skip list associated with I . The cost of this step is proportional to the number of such intersecting segments, ignoring an $\tilde{O}(\log n)$ additive overhead.

The number of intervals on the search path of a is $\tilde{O}(\log n)$. Each segment Q intersecting σ is reported just once, when the unique canonical interval, which is covered by Q and contains a , is encountered. It follows that the total query cost is $\tilde{O}(k + \log^2 n)$, where k is the answer size. The space requirement of the primary skip list, including that of the canonical covering lists, is $\tilde{O}(n \log n)$ (Theorem 8.1.3). The expected size of a secondary skip list is proportional to the size of the corresponding canonical covering list.

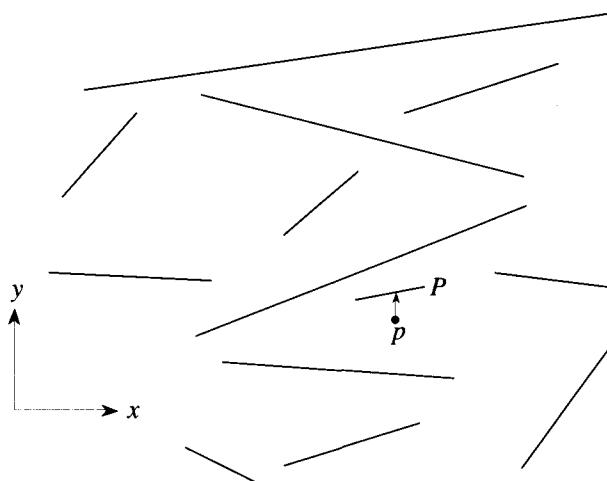


Figure 8.10: Vertical ray shooting.

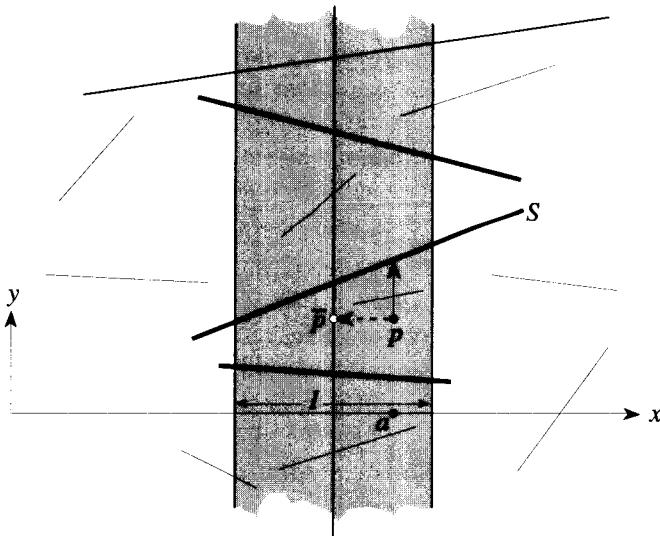


Figure 8.11: Answering a vertical ray query.

Hence, the expected space requirement of our search structure is $O(n \log n)$. In fact, the $O(n \log n)$ bound holds with high probability. This follows just as in Exercise 8.1.6.

Using the above search structure, we can also answer a different kind of query that will be useful to us later on. Given a query point p in the plane, the problem is to find the segment in N that is immediately above p —i.e., the first segment, if any, that is hit by the vertical ray emanating from p (Figure 8.10). This can be done as follows (Figure 8.11).

Let a be the projection of p onto the x -axis. For each interval I lying on the search path of a , we determine the segment in $\text{cover}(I)$ that is immediately above p . This is done as follows. Let \bar{p} denote the projection of p onto the vertical line through the midpoint of I . Our problem is equivalent to locating \bar{p} in the partition of this line formed by the segments in $\text{cover}(I)$. This is done in logarithmic time, with high probability, using the secondary skip list associated with I . In this fashion, we locate the segment in $\text{cover}(I)$ above p , for each I on the search path of the projection a . Finally, we retain the segment that is nearest to p in the vertical direction. The whole query cost is $\tilde{O}(\log^2 n)$.

Updates: We shall only consider additions because deletions are similar. Consider the addition of a new segment S to N . This is done in two steps. We first add the projections of the endpoints of S to the primary skip list. This can be done in expected $O(\log n)$ time (Section 8.1.1). We also determine

the canonical covering lists of all newly created intervals in the primary skip list and build the corresponding secondary skip lists from scratch. It follows, by a verbatim translation of the argument in Section 8.1.2, that this takes $O(\log^2 n)$ expected time. Finally, we determine all $\tilde{O}(\log n)$ intervals in the new primary skip list that are canonically covered by S . We insert S in the secondary skip list associated with each such interval. The total cost of adding S in all secondary skip lists is $\tilde{O}(\log^2 n)$.

Rectangular queries: Let us now turn to a general query given in the form of an orthogonal rectangle. If a segment in N intersects this rectangle, then either it intersects one of its sides or, if that is not the case, its endpoints lie within the rectangle. This suggests the following procedure for answering a rectangular query. The segments intersecting the vertical sides of the query rectangle can be determined by using the search structure we have already described. The segments intersecting its horizontal sides can be determined using a similar search structure defined with respect to the projections on the y -axis. We also build a search structure as in Section 8.1.2 for orthogonal range queries over the set of endpoints of the segments in N . We are using here only a special case of Theorem 8.1.5, wherein the objects are points and the queries are rectangles. We can determine the segments in N whose endpoints lie within the query rectangle by using this orthogonal range tree. It follows that the cost of a rectangular query is also $\tilde{O}(\log^2 n)$.

The following theorem now immediately follows from Theorems 8.1.5 and 8.1.3, and the discussion in this section.

Theorem 8.2.1 *Let N be a set of nonintersecting segments in the plane. A randomized search structure for N can be built in $\tilde{O}(n \log n)$ space and $\tilde{O}(n \log^2 n)$ time. Given an orthogonal query σ , all segments in N intersecting σ can be reported in $\tilde{O}(k + \log^2 n)$ time, where k denotes the answer size. Similarly, given any query point p in the plane, the segment in N above p can be reported in $\tilde{O}(\log^2 n)$ time. The expected cost of adding or deleting a segment to the search structure is $O(\log^2 n)$.*

Exercises

8.2.1 Remove the nondegeneracy assumption made in the beginning of this section. In particular, make sure that the algorithm works with the same performance bounds when the segments in N are allowed to share endpoints. (Hint: Sort the segments adjacent to the same point by their angles. Maintain a secondary search structure for this sorted order.)

8.2.2 Give modifications required to handle count-mode queries. Make sure that no segment is counted (or reported) more than once. Prove $\tilde{O}(\log^2 n)$ bound on the count-mode query time. The other performance bounds should remain the same.

8.2.3 Describe an alternative to the search structure in section that is based on randomized binary trees instead of skip lists. Prove analogous bounds. (Hint: See Exercise 8.1.2.) Also give a deterministic analogue. (Hint: See Exercise 8.1.4.)

***8.2.4** Extend the algorithm in this section to cover cases where the edges of the planar graph are not straight but algebraic of bounded degree. Prove that the performance bounds remain the same, within a constant factor that depends quadratically on the degree bound. (Hint: Break the algebraic edges into monotonic pieces, as in Exercise 3.1.4.)

****8.2.5** Generalize the results in this section to dynamic orthogonal queries over possibly intersecting segments.

†**8.2.6** Design search structures, with analogous performance bounds, for orthogonal queries over nonintersecting polygons in R^3 or more generally nonintersecting polyhedra in R^d .

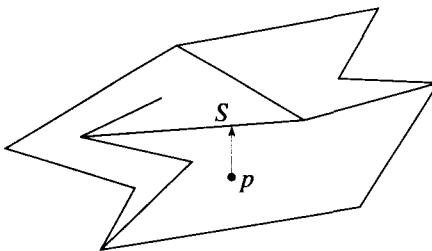


Figure 8.12: Planar point location.

8.3 Dynamic point location

Let N be a set of nonintersecting segments in the plane. We allow the segments to share endpoints. Consider the planar graph $G(N)$ formed by N . Given a query point p in the plane, the planar point location problem is concerned with finding the region in this planar graph containing p .

The search structure in Section 8.2 (see also Exercise 8.2.1) can be readily used for this problem if the planar graph is connected. We represent the planar graph by storing adjacencies among the endpoints and segments. Each region in the planar graph is represented by the cyclically ordered list of the segments on its boundary. We also associate pointers from each segment to the two adjacent regions, or, more precisely, their labels.

Given a query point p , we determine the segment $S \in N$ above p by using the search structure of Section 8.2 (Figure 8.12). If there is no such segment, p belongs to the unique unbounded region of the planar graph. If S exists, the adjacency information tells us the region below S , which contains p .

The updates consist of additions or deletions of segments. We assume that during each update connectivity of the underlying planar graph is preserved. The search structure can be updated as in Section 8.2.

A subtle problem is encountered during the update of the underlying planar graph. So far, we have been assuming that each segment is given pointers to the labels of the regions adjacent to it. When a region is split, this would require us to update the pointers associated with all segments adjacent to that region. This is far too costly. For this reason, we associate a *boundary tree* with each region. It is just a search tree for the cyclically ordered list of the adjacent edges. It can be based on skip lists or randomized binary trees. The root of a boundary tree is given the label of the corresponding region. Each segment S now occurs in exactly two boundary trees. The label of a region adjacent to S can be accessed by traveling in the appropriate boundary tree from the node labeled with S to its root. This takes logarithmic time, with high probability.

We also need to update the boundary trees during the addition or deletion of a segment. This can be easily done in logarithmic time, with high probability, with the help of the split and concatenate operations (Exercise 1.4.5). We leave the details to the reader as a simple exercise.

We have thus proved the following.

Theorem 8.3.1 One can construct a dynamic point location structure of $\tilde{O}(n \log n)$ size for connected planar graphs with $\tilde{O}(\log^2 n)$ query time. The expected cost of update is $O(\log^2 n)$.

Exercises

8.3.1 Describe in detail the update of the boundary trees during the addition or deletion of a segment.

8.3.2 Extend the algorithm in this section to handle the following operations:

- (a) Split an edge $e \in N$ at a given point p in expected $O(\log^2 n)$ time.
- (b) The reverse of the preceding operation in expected $O(\log^2 n)$ time.
- (c) Move a vertex of the planar graph to a new location in expected $O(\log^2 n)$ time. Assume that the topology of the planar graph is not changed.

***8.3.3** Extend the algorithm to planar graphs whose edges are not linear but algebraic of bounded degree. (Hint: See Exercise 3.1.4.)

8.3.4 Give a deterministic analogue to the data structure in this section. (Hint: See Exercise 8.1.4.)

8.4 Simplex range search

In Section 6.5, we gave a search structure for answering simplex range queries in R^d with $O(k + \text{polylog}(n))$ query time, where k is the answer size, and n is the number of points in the object set. It took $O(n^{d+\epsilon})$ space, and could be built in $O(n^{d+\epsilon})$ expected time. Now suppose one could not afford to

allocate $O(n^{d+\epsilon})$ space for the search structure. Is it then possible to build a smaller structure at the cost of a bigger query time? This is the question that we wish to tackle in this section.

Consider the following extreme. Suppose one could only afford to allocate $O(n)$ space for the search structure. This is clearly the minimum, because we need $\Omega(n)$ space just to store all points in the object set. What query time can one guarantee with just $O(n)$ space requirement? Guaranteeing $O(n)$ query time is trivial: One can just examine each object point individually to see if it lies in the query range. Can one do better? Surprisingly, the answer is yes. One can build a search structure of $O(n)$ size and answer queries in $O(n^{1-1/d} \text{polylog}(n))$ time.

More generally, given any parameter m in the range $[n, n^d]$, one can build a search structure of $O(m^{1+\epsilon})$ size guaranteeing $O((n/m^{1/d})\text{polylog}(n))$ query time.² This query time is close to optimal: One can show in a reasonable model of computation that any search structure of size m must take $\Omega(n/(m^{1/d} \log n))$ query time. We shall not prove this lower bound here; see bibliographic notes for more information.

First, we shall describe the linear size search structure. Combining this approach with the one in Section 6.5, one can then easily achieve the desired space-time trade-off. Our goal now is the following: Given a set of n points in R^d , build a search structure of $O(n)$ size guaranteeing $O(n^{1-1/d} \text{polylog}(n))$ query time.

Our search structure will be a certain *partition tree*. It is based on the following partition scheme (Figure 8.13): Given a set of n points in R^d , somehow divide the space into $O(r)$ regions, for some $r \leq n$, each of which contains at least n/r points. The partition should have the property that any given hyperplane in R^d can intersect only a “few” of these regions. For the partition scheme to be described in this section, “few” will mean $O(r^{1-1/d})$. Since a simplex can be thought of as the intersection of $O(1)$ half-spaces, the boundary of any query simplex will also intersect at most $O(r^{1-1/d})$ regions. The regions that do not intersect this boundary are easy to handle: Either all points contained in that region are within the query simplex (e.g., see Δ_1 in Figure 8.13) or all of them are outside (e.g., see Δ_3 in Figure 8.13). It remains to handle the points in the regions intersected by the boundary of the range (e.g., Δ_2 in Figure 8.13). Towards this end, the partition scheme is applied recursively for the subsets in each of the regions of the above partition until we reach trivial small subsets of constant size. A data structure recording this recursive partition of the input point set will be our partition tree. In

²Here $\epsilon > 0$ is a constant that can be made arbitrarily small and the constant within the second Big-Oh notation depends on ϵ .

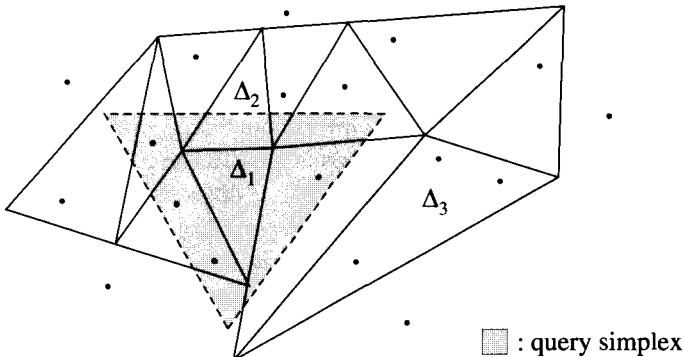


Figure 8.13: A partition.

the query answering process, the regions intersected by the boundary of the range will be handled recursively using this partition tree.

Let us now formalize the preceding partitioning scheme. The regions in our partition will be simplices. However, they will not be disjoint, as we assumed above. The disjointness condition can be relaxed as follows without affecting the algorithm. Let N be the given set of n points in R^d . A *simplicial partition* for N is defined to be a collection

$$\mathcal{P} = \{(N_1, \Delta_1), \dots, (N_m, \Delta_m)\},$$

where N is a disjoint union of the N_i 's, and each Δ_i is a simplex containing N_i (Figure 8.14). Each pair (N_i, Δ_i) is called a class of \mathcal{P} . It is possible that Δ_i contains other points of N_j , for some $j \neq i$. But within Δ_i , we shall only be interested in the point set N_i . Since the point sets N_i are disjoint, this partition will turn out to be as good as a disjoint partition of R^d . In the above definition, the dimension of each Δ_i need not be d , though it will generally be the case.

If h is a hyperplane and Δ is a simplex, we say that h crosses Δ if it intersects Δ properly. By properly, we mean that Δ is not contained within h . We define the crossing number of h relative to \mathcal{P} to be the number of simplices among the Δ_i 's that are crossed by h . We say that the partition \mathcal{P} is *fine* if the size of each N_i is less than or equal to $2n/m$, where m is the size of \mathcal{P} . (There is nothing special about the constant 2. It is just big enough.) By the size of \mathcal{P} , we mean the number of simplices in it.

A basic result regarding partitions is the following.

Theorem 8.4.1 (Partition theorem) *Let N be a set of n points in R^d . Let $r \leq n$ be a parameter. There exists a fine simplicial partition \mathcal{P} of size $O(r^{1-1/d})$. If r is a constant, \mathcal{P} can be computed in $O(n)$ expected time.*

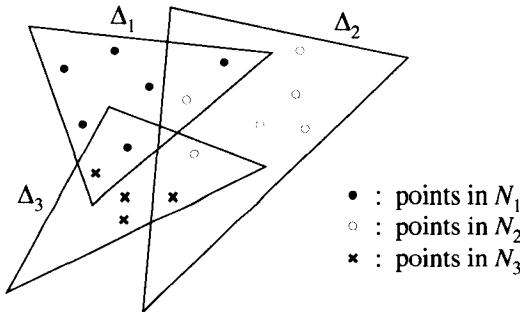


Figure 8.14: A simplicial partition.

We shall prove this theorem in Section 8.4.1. Let us first see why it implies the existence of a suitable partition tree.

Theorem 8.4.2 *Let N be a set of n points in R^d . There exists a partition tree of $O(n)$ size that can be used for answering any simplex range query in $O(n^{1-1/d} \text{polylog}(n))$ time. In the report mode, the query time is $O(k + n^{1-1/d} \text{polylog}(n))$, where k is the answer size.*

Proof. The tree is built recursively using the partition theorem. Let z denote the root of this tree. It will correspond to a partition \mathcal{P}_z of N as guaranteed by Theorem 8.4.1 with $r = n^{1-1/d}$. We store the size of N at this root. We also store at z the descriptions of the simplices Δ_i in \mathcal{P}_z . But we do not store the whole set N at z —that would be too expensive. The children of z correspond to the point sets N_i of the partition \mathcal{P}_z . The subtrees rooted at these children are built recursively. We stop the recursion when the size of the point set associated with a node drops below a small enough constant. Thus, the leaves of our partition tree correspond to the point sets of constant size. Because of their constant size, we can afford to store these point sets at the corresponding leaves. These point sets for the leaves form a partition of N into constant size subsets.

The queries are answered as follows. Given a query simplex, we start at the root z . We examine the simplices of \mathcal{P}_z one by one. If a simplex $\Delta = \Delta_i$ of \mathcal{P}_z is contained within the query simplex, we note the size of N_i ; this size is stored at the child labeled with Δ . In the report mode, we traverse the subtree rooted at this child and report the points stored at its leaves. If a simplex Δ of \mathcal{P}_z is outside the query simplex, we just ignore it. If Δ intersects the boundary of the query simplex, then we handle Δ recursively using the corresponding subtree. By the partition theorem, the number of such simplices is $O(r^{1-1/d})$. In the count mode, we get the

following recurrence for the query time:

$$Q(n) = O(1),$$

for n bounded by a suitable constant, and

$$Q(n) = O(r) + O(r^{1-1/d})Q\left(\frac{2n}{r}\right), \quad (8.1)$$

where $r = n^{1-1/d}$. The depth of our tree, with this choice of r , is $O(\log \log n)$. Hence, the solution of the above recurrence relation is

$$Q(n) = O(n^{1-1/d} 2^{O(\log \log n)}) = O(n^{1-1/d} \text{polylog}(n)). \quad (8.2)$$

The space requirement of our data structure satisfies the recurrence

$$s(n) = O(1),$$

if n is bounded by a suitable constant, and

$$s(n) = O(r) + O(r)s\left(\frac{2n}{r}\right), \quad (8.3)$$

where $r = n^{1-1/d}$. The solution of this recurrence is $O(n)$. Thus, the size of our partition tree is linear in n . Finally, let us turn to the query time in the report mode. Recall that in the report mode, we traverse a subtree rooted at a child of z if the simplex associated with this child is contained in the query simplex. For the same reasons as before, the size of this subtree is linear in the number of points stored in its leaves; these points are all reported. Hence, it follows that we only spend $O(k)$ additional time in the report mode, where k is the answer size. \square

It can be shown that the partition tree described above can be built in $O(n \log n)$ expected time. We shall turn to the preprocessing issue in Section 8.4.2.

8.4.1 Partition theorem

First, we shall prove Theorem 8.4.1. Our goal is to construct a fine partition that has a small crossing number with respect to every possible hyperplane. Towards this end, we shall first construct a certain set of “test” hyperplanes with the following property:

If any given partition \mathcal{P} with certain required properties has a small crossing number with respect to every test hyperplane, then it has a small crossing number with respect to every hyperplane.

This means we will only need to worry about a small number of test hyperplanes instead of the infinitely many possible hyperplanes. A test set of hyperplanes with the preceding property can be constructed as follows. Let \hat{N} be the set of hyperplanes that are dual to the points in the given set N . We can use any duality transform in Section 2.4.1 for this purpose. Let \hat{R} be a random sample of \hat{N} of size $r^{1/d} \log r$. Let $G(\hat{R})$ be the arrangement formed by these hyperplanes.

Let $H(\hat{R})$ be the canonical triangulation of $G(\hat{R})$. By the random sampling results (Sections 5.1 and 6.3), there exists an explicit constant $b > 0$ such that, with a high probability, every simplex of $H(\hat{R})$ intersects at most

$$\frac{b n}{r^{1/d} \log r} \log r = \frac{b n}{r^{1/d}}$$

hyperplanes in $\hat{N} \setminus \hat{R}$. If this property is not satisfied, we keep on taking a new random sample until the property is satisfied. Now, let the test hyperplanes be the duals of the vertices in $G(\hat{R})$. Their number is clearly bounded by the size of $G(\hat{R})$. This is $O(r \log^d r)$, because the size of \hat{R} is $r^{1/d} \log r$. We claim that these test hyperplanes have the property mentioned above.

Lemma 8.4.3 (Test set lemma) Consider any fine simplicial partition

$$\mathcal{P} = \{(N_1, \Delta_1), \dots, (N_m, \Delta_m)\}$$

satisfying, in addition, $|N_i| \geq \lfloor n/r \rfloor$ for all i . Let l be the maximum crossing number of any test hyperplane with respect to \mathcal{P} . Then the crossing number of any hyperplane relative to \mathcal{P} is bounded by $(d+1)l + O(r^{1-1/d})$.

Proof. Let h be any hyperplane. Let \hat{h} be the dual point. Let Σ be the simplex in $H(\hat{R})$ containing \hat{h} . Let h_0, \dots, h_d be the test hyperplanes dual to the vertices of Σ .³ These hyperplanes together intersect at most $(d+1)l$ simplices in the partition \mathcal{P} . It remains to estimate the number of simplices in \mathcal{P} that are intersected by h but not by h_0, \dots, h_d (Figure 8.15). All these simplices are obviously contained in the zone of h within the arrangement formed by h_0, \dots, h_d . From the properties of the duality transform in Section 2.4.1, it easily follows that any point of N lying in this zone dualizes to a hyperplane intersecting the simplex Σ in $H(\hat{R})$.

By the construction of $H(\hat{R})$, the number of such hyperplanes is $O(n/r^{1/d})$. Hence, the zone of h may contain at most so many points of N . By assumption, each N_i has at least $\lfloor n/r \rfloor$ size. Hence, the zone of h can contain at most $O(\lfloor n/r^{1/d} \rfloor / \lfloor n/r \rfloor) = O(r^{1-1/d})$ such N_i 's. In other words, the number of simplices of \mathcal{P} that are crossed by h but not by h_0, \dots, h_d is $O(r^{1-1/d})$. \square

³The number of such hyperplanes is actually less than $d+1$, if the simplex is unbounded. But that will only lead to a better bound.

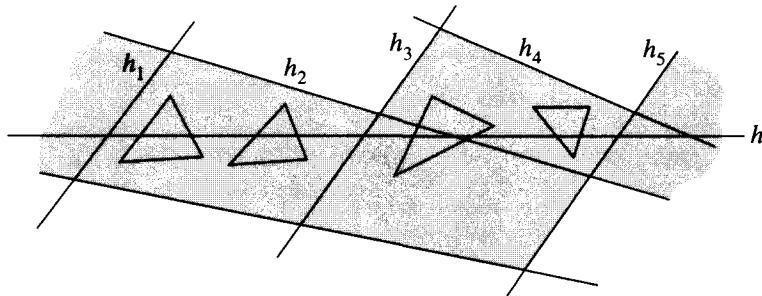


Figure 8.15: The simplices in \mathcal{P} intersecting h but not h_0, \dots, h_d (strictly speaking, $d = 2$ for the plane).

The partition theorem will now follow from the following result.

Lemma 8.4.4 *Let N be any set of n points in R^d . Fix any parameter $r \leq n$. There exists a subset $\bar{N} \subseteq N$ of at least $n/2$ points and a simplicial partition $\bar{\mathcal{P}} = \{(N_1, \Delta_1), \dots, (N_m, \Delta_m)\}$ of \bar{N} with $|N_i| = \lfloor n/r \rfloor$, for all i (which also means $m \leq r$), and with crossing number $O(r^{1-1/d})$ for each test hyperplane.*

Let us first see why this lemma implies the partition theorem. We first apply the lemma to the original set $N^0 = N$ with parameter $r_0 = r$. This yields a simplicial partition \mathcal{P}^0 for at least half of the points in N^0 . We let N^1 be the set of the remaining points in N^0 and choose $r_1 \leq r/2$ so that $\lfloor |N^1|/r_1 \rfloor = \lfloor n/r \rfloor$. We again apply the lemma to N^1 , with parameter r_1 , to obtain a simplicial partition \mathcal{P}^1 of at least half the points in N^1 . We iterate this process $i = O(\log r)$ times until N^i has less than n/r points. At this point, we stop and take the union of the simplicial partitions formed so far. We also add one large simplex containing the remaining points. The resulting simplicial partition of N has size at most $r + r/2 + r/4 + \dots \leq 2r$. Each of its classes contains at most $\lfloor n/r \rfloor$ points. Hence, the partition is fine. It also satisfies the assumption in the test set lemma (ignoring the last class). The crossing number of each test hyperplane relative to this partition is within the order of

$$r^{1-1/d} + \left(\frac{r}{2}\right)^{1-1/d} + \left(\frac{r}{4}\right)^{1-1/d} + \dots = O(r^{1-1/d}).$$

By Lemma 8.4.3, the crossing number of any hyperplane also has the same order.

Proof of Lemma 8.4.4. We construct the classes (N_i, Δ_i) , one at a time, by induction on i . The construction finishes when the total size of N_i 's exceeds $n/2$. In other words, we stop as soon as i becomes $m = \lceil (n/2)/\lfloor n/r \rfloor \rceil$. Let Q be the set of test hyperplanes as in Lemma 8.4.3. Throughout the

construction, we maintain a certain positive integral weight with each test hyperplane. A high-level description of the construction is as follows.

Algorithm 8.4.5

Initialization: Let the initial weight of each hyperplane in the test set Q be one.

For $i = 1$ to m , do:

- Let $N'_i = N \setminus \{N_1, \dots, N_{i-1}\}$ be the set of the points that are unchosen so far.
- Choose a simplex Δ_i , in a way to be described soon, so that:
 1. The total weight of all hyperplanes in Q intersecting Δ_i is at most a $(1/t)$ th fraction of the total weight of Q , for some $t = \Omega(r^{1/d})$, and
 2. The simplex Δ_i contains exactly $\lfloor N/r \rfloor$ points of N'_i . Let N_i be the set of these points.
- Double the weight of each hyperplane in Q intersecting Δ_i .

First, let us see why the partition $\{(N_1, \Delta_1), \dots, (N_m, \Delta_m)\}$ output by this algorithm will have the required property. We shall address the construction of Δ_i with required properties later. Let $w_i(Q)$ denote the total weight of the hyperplanes in Q at time i . Notice that during the i th step the weights of the hyperplanes intersecting Δ_i are doubled. The weights of the remaining hyperplanes remain unchanged. The total weight of the hyperplanes intersecting Δ_i is bounded by $w_i(Q)/t = O(w_i(Q)/r^{1/d})$. Hence, it follows that

$$w_{i+1}(Q) \leq w_i(Q) + O\left(\frac{w_i(Q)}{r^{1/d}}\right) = w_i(Q) \left[1 + O\left(\frac{1}{r^{1/d}}\right)\right].$$

Moreover, the initial weight $w_0(Q)$ is just the size of Q . It follows that

$$w_m(Q) \leq w_0(Q) \left[1 + O\left(\frac{1}{r^d}\right)\right]^m = |Q| \left[1 + O\left(\frac{1}{r^d}\right)\right]^m. \quad (8.4)$$

Now, fix any hyperplane $h \in Q$. Suppose its crossing number is l . Then, its weight $w_m(h)$ is 2^l . This is because the weight of h is doubled precisely during the addition of the Δ_i 's that it crosses. Hence, from (8.4),

$$l \leq \log_2 w_m(h) \leq \log_2 w_m(Q) \leq \log_2 |Q| + m \log_2 [1 + O(1/r^d)] = O(r^{1-1/d}), \quad (8.5)$$

because $m \leq r$, $\ln(1+x) \leq x$ and $|Q| = O(r \log^d r)$ (for the last equality, refer to the construction of the test set).

It remains to describe the choice of Δ_i in Algorithm 8.4.5 at time i . This is done as follows. We choose a $(1/t)$ -cutting of the weighted set Q . For this, we use the weighted version of the Cutting Lemma 6.3.1 (Exercise 6.3.4). The size of this cutting is $O(t^d)$. If $t = cr^{1/d}$ for a suitably small constant $c > 0$, then we can assume that the total number of simplices in this cutting (of all dimensions) is at most $r/2$. Since the size of N'_i is greater than $n/2$ (otherwise the algorithm terminates), some simplex in this cutting contains at least $(n/2)/(r/2) = n/r$ points. Uniformly shrink this simplex so that it contains exactly $\lfloor n/r \rfloor$ points. Let Δ_i be this simplex. It has the required properties. \square

8.4.2 Preprocessing time

It remains to estimate the cost of building a partition with the properties as in Theorem 8.4.1. Fix a parameter $r \leq n^{1-\delta}$, for an arbitrarily small but fixed $\delta > 0$. It can be shown that a partition with the properties in Theorem 8.4.1 can be built in $O(n \log r)$ expected time. The constant within the Big-Oh notation depends upon δ . This is sufficient for the construction of the partition tree described in the proof of Theorem 8.4.2. It then easily follows that the expected cost of constructing this partition tree is $O(n \log n)$. To show this one only needs to write down a recurrence equation like (8.3).

Unfortunately, constructing a partition in $O(n \log r)$ expected time is quite difficult. It basically requires several bootstrappings. This means one starts with a not so efficient algorithm and then obtains a more efficient algorithm by using the less efficient algorithm as a subroutine. The process can be repeated, if necessary.

Of course, one has to start with some algorithm initially. For a start, one can use an algorithm for the special case when r is bounded by a constant. If one examines the construction in Section 8.4.1, one sees that it requires $O(n)$ expected time. This is because all random samples used in the construction are of constant size, and hence, Algorithm 8.4.5 works in $O(n)$ expected time. We shall defer the bootstrapping procedure to the exercises. Here we shall only examine what happens if one lets r to be a large enough constant in the construction of the partition tree as given in the proof of Theorem 8.4.2. We only need to reexamine the recurrence equations (8.1) and (8.2), letting r be a large enough constant this time. The depth of our partition tree is now $O(\log_r n)$. Hence, the query time becomes $O(n^{1-1/d} 2^{O(\log_r n)})$. This can be made $O(n^{1-1/d+\epsilon})$, for any $\epsilon > 0$, if we choose r large enough. If we let r be a constant in (8.3), the space requirement is seen to be $O(n \log n)$. Similarly, it is easy to check that the expected preprocessing time is also $O(n \log n)$.

The preceding discussion can be summarized as follows.

Theorem 8.4.6 Given a set of n points in R^d , a partition tree of $O(n \log n)$ size can be built in $O(n \log n)$ expected time and simplex range queries can be answered in $O(n^{1-1/d+\epsilon})$ time.

The query time and the space requirement are improved further in the exercises.

Exercises

8.4.1 Verify that the construction of the partition in Section 8.4.1 takes $O(n)$ expected time, if $r = O(1)$. (Hint: The construction of a $(1/r)$ -cutting takes $O(n)$ expected time (Exercise 6.3.3). So Algorithm 8.4.5 also takes $O(n)$ expected time.)

8.4.2 Combine the approach in this section and the one in Section 6.5 to achieve the following space-time trade-off. Given a parameter m in the range $[n, n^d]$, construct a search structure of $O(m^{1+\epsilon})$ size, with $O([n/m]^{1/d} \text{polylog}(n))$ query time. (Hint: Build a partition tree as in the proof of Theorem 8.4.2. But terminate the construction when the number of points associated with a node drops below $(m/n)^{1/(d-1)}$.)

8.4.3 Fix any positive constant δ . Show how a partition as in Theorem 8.4.1 can be built in $O(n \log r)$ expected time, for any $r \leq n^{1-\delta}$, where the constant factor within the Big-Oh notation is allowed to depend on δ . This bound is better than $O(n \log n)$ when r is very small. Begin with the $O(n)$ expected time procedure given before for a constant r , and get a better procedure via bootstrappings. Follow these steps:

- (a) Given a set N and a parameter $r \leq n$, show that a fine simplicial partition of $O(r)$ size with crossing number $O(r^{1-1/d+\delta})$, for an arbitrary fixed $\delta > 0$, can be constructed in expected $O(n \log r)$ time. (Hint: Start with a partition with $r = r_0$, a suitable constant. Recursively refine each class in this partition.)
- (b) Suppose you are given a procedure for constructing a partition as in Theorem 8.4.1, with $r \leq n^\alpha$, for a fixed $\alpha < 1$, and with $O(n \log r)$ expected running time. Show how this procedure can be used to get a similar procedure for $r \leq n^{1-\delta}$, for any fixed $\delta > 0$. (Hint: Use the same composition trick as in (a).)
- **(c) Show how the procedure in (a) could be used to obtain an $O(n \log r)$ expected time procedure for $r \leq n^\alpha$, with a sufficiently small $\alpha < 1$. (Hint: Examine the proof of the partition theorem carefully. The critical part is Algorithm 8.4.5. To select Δ_i one needs to compute the number of points within every simplex in the $(1/t)$ -cutting of Q . One can use the simplex-range search structure in (a) for this purpose. A subtle point is that this search structure has to be dynamic—it must allow deletions, because the points in \bar{N}^i are continuously deleted in the procedure. You will need to use the dynamization idea from Section 8.6. Also remember that these queries do not have to be answered exactly.)
- (d) Conclude that a partition as in Theorem 8.4.1 can be built in $O(n \log r)$ expected time, for any $r \leq n^{1-\delta}$.

8.4.4 (Weighted setting)

- (a) Extend the algorithms in this section to the following weighted setting. Each point in the object set is given an integer weight. In the count mode, the goal is to calculate the total weight of the points lying in the query simplex.
- *(b) In general, let the weights be drawn from a commutative semigroup with abstract addition operation $+$. Modify the algorithm accordingly. (The notion of a semigroup lets us unify the counting and reporting versions. In the counting version, the weight of each point is 1, and $+$ is the ordinary addition. In the reporting version, the weight of each point is a singleton set consisting of its symbolic name, and $+$ denotes the set union operation.)

8.5 Half-space range queries

In this section, we consider the half-space range reporting problem (Section 2.1). Let N be any set of n points in R^d . The goal is to build a search structure so that, given any half-space, all points of N lying within the half-space can be reported quickly. In Section 7.7.2, we gave one search structure for half-space range reporting with $O(n^{\lfloor d/2 \rfloor + \epsilon})$ space requirement and $O(k \log n)$ expected query time, where k is the answer size. (The query time can actually be improved to $O(k + \log n)$; see Exercise 7.8.16.) In this section, we shall deal with the same question as in the last section: Can one build a smaller search structure at the cost of a bigger query time?

For example, consider what happens if one is allowed only linear or almost linear space. In this case, one can use the search structure in the previous section, because a half-space can be thought of as an unbounded simplex. This gives us an $O(n)$ size search structure with $O(k + n^{1-1/d} \text{polylog}(n))$ query time (Theorem 8.4.2). Can one do better? Yes. We shall give in this section an $O(n \log \log n)$ size search structure guaranteeing $O(k + n^{1-1/\lfloor d/2 \rfloor} \text{polylog}(n))$ query time. If one combines the approach in this section with the one in Section 7.7.2, one can achieve a space-time trade-off: Given any parameter m in the range $[n, n^{\lfloor d/2 \rfloor}]$, one can build a search structure of $O(m^{1+\epsilon})$ size and answer queries in $O(k + [n/m]^{1/\lfloor d/2 \rfloor} \text{polylog}(n))$ time. One could conjecture that this query time is quasi-optimal. But no rigorous proof for this conjecture is known as yet.

So, let us turn to the search structure with almost linear size. Our goal is the following. Given a set of n points in R^d , we wish to build a search structure of $O(n \log \log n)$ size guaranteeing $O(k + n^{1-1/\lfloor d/2 \rfloor} \text{polylog}(n))$ query time.

Our search structure will be a partition tree, similar to the partition tree for simplex range queries described before. It will also be based on a partition theorem akin to Theorem 8.4.1. The difference is going to be the following. Theorem 8.4.1 constructs a partition of $O(r)$ size with $O(r^{1-1/d})$ crossing

number for every hyperplane. The new partition will guarantee even lower crossing number, but only for a special class of hyperplanes called *shallow* hyperplanes. We say that a hyperplane h is k -shallow, with respect to the set N , if one of the two half-spaces bounded by h contains k or fewer points of N . The new partition theorem will guarantee $O(r^{1-1/\lfloor d/2 \rfloor})$ crossing number for (n/r) -shallow hyperplanes.

Theorem 8.5.1 (Partition theorem for shallow hyperplanes)

Let N be any set of n points in R^d . Let $r \leq n$ be a parameter. Then there exists a fine simplicial partition $\mathcal{P} = \{(N_1, \Delta_1), \dots, (N_m, \Delta_m)\}$ of N of $O(r)$ size such that the crossing number of any (n/r) -shallow hyperplane relative to \mathcal{P} is $O(r^{1-1/\lfloor d/2 \rfloor})$, for $d \geq 4$. If $d = 2, 3$, the crossing number is $O(\log^2 r)$. (By a fine partition, we mean that $|N_i| \leq 2n/r$, for each i .) If r is a constant, \mathcal{P} can be built in $O(n)$ expected time.

We shall prove this theorem in Section 8.5.1. It guarantees a low crossing number for only (n/r) -shallow hyperplanes. Why is that sufficient? For example, consider the simplest case when r is a constant. If the hyperplane bounding the query half space is not (n/r) -shallow, then the query half-space contains at least n/r points. In other words, the answer size k in this case is $\Omega(n/r) = \Omega(n)$. So, when the query hyperplane is not (n/r) -shallow, we can afford to answer the query trivially by examining all points in N .

Now consider the case when r is of the order of n^α , for some $0 < \alpha < 1$. Then the answer size k would be $\Omega(n^{1-\alpha})$, if the query hyperplane were not (n/r) -shallow. If $\alpha < 1/d$, one could use the search structure in the previous section to handle such queries. This would require $O(n^{1-1/d} + k) = O(k)$ time, which is just fine.

The preceding discussion leads us to the following strategy for answering queries. If we have to report k points in the answer, we must in any case spend $\Omega(k)$ time in the reporting. Hence, we can spend $O(k)$ time on computing the answer without affecting the asymptotic efficiency. In other words, for a large k , we may use a less efficient procedure than for a small k . This strategy is called a *filtering search* strategy. It allows us to solve the half-space range reporting problem faster than the simplex range reporting problem. It would not work if one were only required to count the number of points in the query half-space. It is still open whether the half-space counting problem can be solved more efficiently than the simplex counting problem.

Now we are ready to give our search structure formally.

Theorem 8.5.2 *Let N be a set of n points in R^d . There exists a partition tree of $O(n \log \log n)$ size that can be used for answering half-space reporting queries in $O(k + n^{1-1/\lfloor d/2 \rfloor} \text{polylog}(n))$ time.*

Remark. For $d = 2, 3$, the size can be made $O(n)$ and the query time can be made $O(k + \log n)$ (Exercise 8.5.7).

Proof. The tree is built recursively, using the partition theorem for shallow hyperplanes. Let z denote the root of this tree. It will correspond to a partition \mathcal{P}_z of N as guaranteed by Theorem 8.5.1, with $r = n^\alpha$, for some positive $\alpha < 1/d$. We store at the root the whole set N as well as the partition \mathcal{P}_z . We also store at the root an auxiliary search structure that will be used in case the hyperplane bounding the query half-space is not (n/r) -shallow. If r were a constant, there would be no need for an auxiliary search structure: As we have already noted, one could just examine all points in N trivially when the query hyperplane is not shallow. When $r \leq n^\alpha$, we can use the simplex range search structure in Theorem 8.4.2 for this purpose.

The children of z correspond to the point sets N_i of the partition \mathcal{P}_z . The subtrees rooted at these children are built recursively. We stop the recursion when the size of a point set associated with a node drops down to a small constant. The point sets associated with the nodes at any fixed level of the tree constitute a partition of N . Thus, our partition tree takes $O(n)$ size per level. Since the depth of our tree is $O(\log \log n)$, the total space requirement is $O(n \log \log n)$. If the parameter r were a constant, the depth would be $O(\log n)$. This would increase the space requirement to $O(n \log n)$.

The queries are answered in an obvious fashion. We begin at the root z . By Theorem 8.5.1, the maximum crossing number of any (n/r) -shallow hyperplane with respect to \mathcal{P}_z is bounded by $ar^{1-1/\lfloor d/2 \rfloor}$, for some explicitly computable constant a . If the hyperplane bounding the query half-space intersects more than $ar^{1-1/\lfloor d/2 \rfloor}$ simplices of \mathcal{P}_z , we know that it is not (n/r) -shallow. In this case, we have already seen that all points in the query half-space can be reported in $O(k)$ time using the auxiliary search structure at z .

If the hyperplane under consideration intersects at most $ar^{1-1/\lfloor d/2 \rfloor}$ simplices of \mathcal{P}_z , we proceed recursively down the subtrees rooted at the corresponding children.

If we ignore the cost of using the auxiliary structures during reporting, the rest of the query time satisfies the following recurrence relation:

$$Q(n) = O(1),$$

if n is bounded by some constant. Otherwise,

$$Q(n) = O(r) + O(r^{1-1/\lfloor d/2 \rfloor})Q\left(\frac{2n}{r}\right),$$

where $r = n^\alpha$. The solution of this recurrence is

$$Q(n) = O(n^{1-1/\lfloor d/2 \rfloor} \text{polylog}(n)),$$

because the depth of our tree is $O(\log \log n)$. If the parameter r were a constant, the solution would be $Q(n) = O(n^{1-1/\lfloor d/2 \rfloor + \epsilon})$, for any fixed $\epsilon > 0$. This is assuming that r is chosen sufficiently large for a given ϵ . \square

What about the preprocessing time? It can be shown that the search structure in Theorem 8.5.2 can be constructed in expected $O(n \log n)$ time. This procedure requires several bootstrappings; it is deferred to the exercises. Here we shall only examine what happens when the parameter r is a constant. In this case, we have already noted that the query time becomes $O(k + n^{1-1/\lfloor d/2 \rfloor + \epsilon})$ and the space requirement becomes $O(n \log n)$. The expected preprocessing time is $O(n \log n)$ because the partition in Theorem 8.5.1 can be constructed in expected linear time.

To summarize:

Theorem 8.5.3 *Let N be a set of n points in R^d . A partition tree can be built in $O(n \log n)$ space and expected time, and half-space reporting queries can be answered in $O(k + n^{1-1/\lfloor d/2 \rfloor + \epsilon})$ time.*

8.5.1 Partition theorem

The proof of Theorem 8.5.1 follows the proof of Theorem 8.4.1 very closely. The following is an analogue of Lemma 8.4.3.

Lemma 8.5.4 (Test set lemma) *Let N be the given set of n points in R^d . There exists a set of $O(r^d \log^d r)$ test hyperplanes that are $(2n/r)$ -shallow and have the following property. For any simplicial partition*

$$\mathcal{P} = \{(N_1, \Delta_1), \dots, (N_m, \Delta_m)\},$$

with $|N_i| \geq \lfloor n/r \rfloor$, for every i , if l is the maximum crossing number of any test hyperplane relative to P , then the crossing number of any (n/r) -shallow hyperplane relative to P is bounded by $(d+1)l + 1$.

Proof. We translate the proof of Lemma 8.4.3. Let \hat{N} be the set of hyperplanes dual to the points in N . To be specific, let us assume that the duality transform under consideration is the paraboloidal transform in Section 2.4.1.

Choose a random sample $\hat{R} \subseteq \hat{N}$ of size $\bar{r} = ar \log r$, for a large enough constant a . Let $G(\hat{R})$ be the arrangement formed by \hat{R} . Let $H(\hat{R})$ be the canonical triangulation of $G(\hat{R})$. The random sampling result (Sections 5.1 and 6.3) implies that, with high probability, every simplex of $H(\hat{R})$ intersects at most $O((n/\bar{r}) \log \bar{r}) \leq n/r$ hyperplanes in $\hat{N} \setminus \hat{R}$. This is assuming that the constant a is chosen large enough. If this property is not satisfied for the chosen sample, we keep on taking a new random sample until the property is satisfied.

For any vertex $v \in H(\hat{R})$, define its level with respect to \hat{N} to be the number of hyperplanes in \hat{N} above v in the positive x_d direction. We let the test hyperplanes be the duals of the vertices in $H(\hat{R})$ whose levels are *not* in the range $(2n/r, n - 2n/r)$; thus all test hyperplanes are $(2n/r)$ -shallow. (In Lemma 8.4.3, the duals of all vertices in $H(\hat{R})$ were included in the test set.)

We claim that these test hyperplanes have the required property. Let h be any (n/r) -shallow hyperplane. Let \hat{h} be the dual point. Let Q be the simplex of $H(\hat{R})$ containing \hat{h} . Since h is (n/r) -shallow, one of the two half-spaces bounded by h contains at most n/r points of N . By duality, either the number of hyperplanes in \hat{N} above \hat{h} or the number of hyperplanes below \hat{h} is bounded by n/r . Since Q is intersected by at most n/r hyperplanes of \hat{N} , it follows that the level of every vertex of Q is outside the range $(2n/r, n - 2n/r)$. Thus, the duals of all vertices of Q are test hyperplanes. Let us denote them by h_0, \dots, h_d .⁴ They intersect at most $(d + 1)l$ simplices in the partition \mathcal{P} .

It remains to estimate the number of simplices in \mathcal{P} that are intersected by h but not by h_0, \dots, h_d . All these simplices are contained in the zone of h within the arrangement formed by h_0, \dots, h_d (Figure 8.15). Any point of N lying in this zone dualizes to a hyperplane intersecting Q . The number of hyperplanes intersecting Q is bounded by n/r . Since the size of each N_i is at least $\lfloor n/r \rfloor$, the zone of h contains at most one simplex of \mathcal{P} . In other words, h can intersect at most one simplex of \mathcal{P} that is not crossed by h_0, \dots, h_d . \square

Theorem 8.5.1 now follows from the following lemma just as Theorem 8.4.1 follows from Lemma 8.4.4. We leave the verification to the reader.

Lemma 8.5.5 *Let N be any set of n points in R^d . Then there exists a subset $\bar{N} \subseteq N$ of at least $n/2$ points and a simplicial partition*

$$\bar{\mathcal{P}} = \{(N_1, \Delta_1), \dots, (N_m, \Delta_m)\}$$

of \bar{N} with $|N_i| = \lfloor n/r \rfloor$, for all i , and with crossing number $O(r^{1-1/\lfloor d/2 \rfloor})$ for each test hyperplane, assuming $d \geq 4$. The crossing number is $O(\log r)$, for $d = 2, 3$.

Proof. Let Q denote the set of test hyperplanes. The overall construction scheme is exactly the one we used in the proof of Lemma 8.4.4. But this time, we let $t = \Omega(r^{1/\lfloor d/2 \rfloor})$ in Algorithm 8.4.5. Since $|Q|$ is bounded by a polynomial in r , it would then follow that the crossing number of any test hyperplane is $O(r^{1-1/\lfloor d/2 \rfloor})$. You only need to substitute d with $\lfloor d/2 \rfloor$ throughout the proof of Lemma 8.4.4 and, in particular, in (8.5). For $d = 2, 3$, it follows from (8.5) that the crossing number is $O(\log r)$, because $|Q|$ is bounded by a polynomial in r .

⁴The number of hyperplanes is actually less than $d + 1$, if the simplex Q is unbounded. But this will only lead to a better bound.

The only thing that remains to be specified is the choice of the class (N_i, Δ_i) at time i . This is done as follows. Consider the upper convex polytope formed by the hyperplanes in Q . Choose a $(1/t)$ -cutting of $O(t^{\lfloor d/2 \rfloor})$ size covering this polytope of the weighted set Q . This can be done using the cutting lemma for convex polytopes in the weighted setting (Exercise 7.7.6). In contrast, the reader should recall that in the case of simplex range queries, or more precisely, in the proof of Lemma 8.4.4, we used the cutting for arrangements, which had a larger $O(t^d)$ size. In a nutshell, this is where we gain efficiency over the procedure for answering simplex range queries.

Assuming that $t = cr^{1/\lfloor d/2 \rfloor}$ for a small enough positive constant c , we can assume that the size of the above cutting is bounded by $r/3$. Let \bar{N}_i be the set of the points in N covered by the simplices of this cutting. Consider the union of the simplices in this cutting. Its complement is contained in the union of at most t half-spaces bounded by the hyperplanes in Q (Exercise 7.7.6). Since the hyperplanes in Q are (n/r) -shallow, it follows that

$$|\bar{N}_i| \geq n - \frac{n}{r} t = n - \frac{n}{r} O(r^{1/\lfloor d/2 \rfloor}) \geq \frac{5}{6}n,$$

if r is large enough.

Assume that the set $N_1 \cup \dots \cup N_{i-1}$ has less than $n/2$ points; otherwise the algorithm terminates. Let $N'_i = \bar{N}_i \setminus (N_1 \cup \dots \cup N_{i-1})$. This set has more than $5n/6 - n/2 = n/3$ points. So there is a simplex in the cutting (of some dimension) that contains at least $(n/3)/(r/3) = n/r$ points. Uniformly shrink it so that it contains exactly $\lfloor n/r \rfloor$ points. Let Δ_i be this shrunk simplex. Let N_i be the set of points in N'_i contained within Δ_i .

It is easy to see that Δ_i has the properties required in Algorithm 8.4.5, with $t = \Omega(r^{1/\lfloor d/2 \rfloor})$. \square

8.5.2 Half-space emptiness

As we have already mentioned, the search structure in this section is suitable only for report-mode queries. One cannot use it in the count mode. However, one can easily handle the following special case of the counting problem: Given a half-space, decide if it is empty. In other words, decide whether the number of points in the query half-space is zero. This problem is called the half-space emptiness problem. It is the most important case of the counting problem.

Half-space emptiness queries can be handled by just modifying our previous algorithm for answering queries. The previous answering scheme for the half-space reporting problem has the following form. At every node of the partition tree visited during the search: (1) Determine whether the hyperplane bounding the query half-space is shallow enough. (2) If not, report all

points lying in the half-space using an auxiliary procedure. (3) Otherwise, proceed recursively.

To handle a half-space emptiness query, we just omit the second reporting step: If the query half-space is not shallow, it is certainly not empty. So we can terminate our search in that case. Thus, the cost of a half-space emptiness query is $O(n^{1-1/\lfloor d/2 \rfloor} \text{polylog}(n))$ for the partition tree described in Theorem 8.5.2. It is $O(n^{1-1/\lfloor d/2 \rfloor + \epsilon})$ for the partition tree in Theorem 8.5.3.

The above procedure for handling half-space emptiness has a curious property. If the half-space is not empty, it cannot provide a witness, i.e., some point lying in the half-space. We shall rectify this shortcoming in Section 8.7.

Exercises

8.5.1 Show that the dual of the half-space emptiness problem is the following: Given a set of hyperplanes in R^d and a query point p , verify whether p lies in the upper (or lower) convex polytope formed by these hyperplanes.

8.5.2 Extend the search structure in this section to the weighted setting as in Exercise 8.4.4.

8.5.3 Obtain a space–time trade-off along the lines of Exercise 8.4.2, with $\lfloor d/2 \rfloor$ replacing d in that exercise. (Hint: Combine the approach here with the one in Section 7.7.2.)

***8.5.4** Show that a partition tree as in Theorem 8.5.2 can be built in $O(n \log n)$ expected time by repeated bootstrappings. (Hint: Similar to Exercise 8.4.3.)

†8.5.5 Can one design a search structure with the space and preprocessing cost as in this section and $O(n^{1-1/\lfloor d/2 \rfloor} \text{polylog}(n))$ cost for counting the number of points in the query half-space range?

***8.5.6** Show that the crossing number in Theorem 8.5.1 is, in fact, $O(\log r)$, for $d = 2, 3$.

***8.5.7** Give a search structure for half-plane queries (i.e., for $d = 2$) with the following performance: $O(k + \log n)$ query time, $O(n)$ space requirement, and $O(n \log n)$ expected preprocessing time.

Give a search structure with the same performance for $d = 3$. (Hint: You might wish to allow almost linear space requirement at first.)

***8.5.8** Give a search structure for half-space emptiness problem with the following performance: $O(n)$ space, $O(n^{1-\lfloor d/2 \rfloor} 2^{O(\log_* n)})$ query time, and $O(n^{1+\epsilon})$ expected processing time. Here $\log_* n$ denotes the iterated log, i.e., the number of log operations needed to end up with a number less than 1, starting with n . (Hint: Show that the search structure in Theorem 8.5.2 can be improved if one is only interested in half-space emptiness.)

8.5.9 Let N be a set of points in R^d . Suppose the interior of the query half-space is empty, but its boundary does contain some points in N . If the points in N are in general position, the boundary can contain at most $d+1$ points in N . Modify the algorithm for the half-space emptiness problem so that all points on the boundary can be reported in time proportional to their number.

The dual of the above half-space emptiness problem is the following (Exercise 8.5.1): Given a set of hyperplanes in R^d and a query point p , determine whether p belongs to the upper convex polytope formed by these hyperplanes; if p lies on the boundary of this polytope, report all hyperplanes containing p .

8.6 Decomposable search problems

The search structures described in Sections 8.4 and 8.5 are static. We shall dynamize them in this section. There is a very simple method for converting static structures for range searching into semidynamic ones. In the semidynamic setting, only additions are allowed. Deletions are generally more nontrivial. We shall address them later.

The conversion of a static range search structure into a semidynamic one is based on the following simple observation. If N_1 and N_2 are two disjoint object sets, then, for a given range query, answers with respect to N_1 and N_2 can be combined into an answer for $N_1 \cup N_2$ in constant time: In the count mode, we just need to add the two counts. In the report mode, we only need to join the two lists of answers. The search problems with the above property are called *decomposable*. The scheme that we are about to describe works for any decomposable search problem.

Let N denote the set of objects in a given problem. Assume that we know how to build a static search structure for N . Let us denote this static structure by $\tilde{H}(N)$. Let $T(n)$ denote a bound on the cost of building $\tilde{H}(N)$. Let $Q(n)$ denote a bound on the maximum query cost for this structure. Let $S(n)$ denote a bound on the space requirement. We assume that $T(n)$, $Q(n)$, and $S(n)$ are all nondecreasing functions of n .

Now let us turn to the semidynamic setting. Let M denote the set of objects existing at any given time. Let m be the size of M . Our goal is to maintain a semidynamic search structure associated with M . This will be done as follows. We shall always keep M decomposed into $\log_2 m$ subsets M_0, M_1, \dots . The size of M_i will be $m_i = b_i 2^i$, where b_i corresponds to the 0–1 bit in the binary representation of m . The use of binary representation here is purely arbitrary. One could as well use p -adic representation, for any constant p .

Our search structure for M will consist of the static search structures $\tilde{H}(M_i)$, for each i . Since the problem is decomposable, the answer to a query over M can be obtained by combining the answers for the M_i 's. Let $Q(m_i)$ denote the maximum query cost over M_i . Then the total query cost is $O(\log_2 m) + \sum_i Q(m_i)$.

The addition of a new object to M is akin to the addition of 1 to m in the binary system. Let S denote the new object. Let $M' = M \cup \{S\}$.

Let $m' = m + 1$ be the size of M' . Let b'_i denote the i th bit in the binary representation of m' . How do we get the binary representation of m' from that of m ? Well, let j be the maximum integer such that the bits b_0, \dots, b_j are all ones. Then,

$$b'_i = b_i, \text{ for } i > j + 1; \quad b'_{j+1} = 1; \quad \text{and} \quad b'_i = 0, \text{ for } i \leq j.$$

Accordingly, we let

$$M'_i = M_i, \text{ for } i > j + 1; \quad M'_{j+1} = M_0 \cup \dots \cup M_j \cup \{S\}; \quad M_i = \emptyset, \text{ for } i \leq j.$$

We build the static search structure $\tilde{H}(M'_{j+1})$ from scratch. Let $T(m'_{j+1})$ denote the cost of this building, where m'_{j+1} is the size of M'_{j+1} .

Let us analyze the total building cost over a sequence of n additions. Fix any $i \leq \log_2 n$. How many times is the static structure for the i th class built from scratch? Note that two such building operations are separated by $2^{i+1} - 1$ additions. This follows by examining the changes in the binary representation of m . Hence, the total building cost is $\sum_i T(2^i)(n/2^{i+1})$. Let us amortize (distribute) this cost uniformly over all additions. Then the amortized cost of each addition is of the order of $\sum_{i=0}^{\log_2 n} T(2^i)/2^{i+1}$. To summarize:

Lemma 8.6.1 *Let m denote the number of existing objects at the given time. Let $m_i = b_i 2^i$, where b_i corresponds to the 0–1 bit in the binary representation of m . The query time in the semidynamic setting is $O(\log_2 m + \sum_{i=0}^{\log_2 m} Q(m_i))$. The space requirement is $\sum_{i=0}^{\log_2 m} S(m_i)$. The total cost of maintenance over a sequence of n additions is bounded, within a constant factor, by $n \sum_{i=0}^{\log_2 n} T(2^i)/2^i$. In other words, the amortized cost of addition is bounded, within a constant factor, by $\sum_{i=0}^{\log_2 n} T(2^i)/2^i$.*

As an example, let us apply the preceding scheme to the search structure for simplex range queries described in Section 8.4. In this case, it can be verified easily that the query time and the space requirement in the semi-dynamic setting remains the same as in the static setting. The total cost of maintenance over a sequence of n additions is higher than the static cost $T(n)$ by only a logarithmic factor. The same holds for the search structure for half-space range queries (Section 8.5).

As the reader must have noticed, the previous scheme does not help us in deletions. Let us see why. Consider the deletion of an object S in M . Suppose S belongs to the highest class M_i , where $i = \log_2 m$. It would be too costly to rebuild the new search structure for $M_i \setminus \{S\}$ from scratch. This is because the highest class contains roughly half of the objects in M . Just imagine what would happen if we were to delete all these objects one by one. One would end up rebuilding the large search structures from scratch too

many times. The problem will not go away even if the update sequence were random—even a randomly deleted object belongs to the highest class with probability $1/2$.

But let us assume that we are given an efficient procedure for deleting an object from any search structure $\tilde{H}(M_i)$. Then, a simple extension of the previous scheme will yield a search structure that supports additions as well. In other words, given a search structure that supports deletions, we can obtain a search structure that supports additions as well as deletions. Let us see how.

Let M denote the set of (undeleted) objects existing at any give time. Let m be its size. Let $\sum b_i 2^i$ be the binary representation of m . As before, we shall keep M partitioned into logarithmic number of classes M_0, M_1, \dots . We shall ensure that the number of objects in the i th class is always bounded by $b_i 2^i$. But this number need not be exactly $b_i 2^i$, as was the case in the semidynamic setting. We shall also ensure that the number of classes at any given time is at most $\lceil \log_2 m \rceil$.

To delete an object $S \in M$, we locate the class M_i containing S and update $\tilde{H}(M_i)$ to $\tilde{H}(M_i \setminus \{S\})$ using the deletion algorithm available to us. Let $D(m_i)$ denote a (nondecreasing) bound on the cost of this deletion. We also need to ensure our invariant on the number of classes. Let $m' = m - 1$ be the number of objects after deletion. Let $M' = M \setminus \{S\}$ be the set of objects after deletion. If $\lceil \log_2 m' \rceil = \lceil \log_2 m \rceil - 1$, we just rebuild our entire search structure from scratch, and let

$$\begin{aligned} M'_i &= M', \quad \text{for } i = \lceil \log_2 m' \rceil, \text{ and} \\ &= \emptyset, \quad \text{otherwise.} \end{aligned}$$

The construction of $\tilde{H}(M'_i) = \tilde{H}(M')$ is done using the static procedure. Its cost is bounded by $T(m)$.

Note that two rebuilding operations of this kind are separated by at least $m/2$ deletions. Hence, the cost of such rebuildings can be amortized by charging $O(T(m)/m)$ additional cost to each deletion.

The addition of a new object S to M is done as before. Let j be the maximum integer such that M_i , for each $i \leq j$, contains $b_i 2^i$ objects. We let

$$M'_i = M_i, \quad \text{for } i > j + 1; \quad M'_i = \emptyset, \quad \text{for } i \leq j; \quad M_{j+1}' = M_1 \cup \dots \cup M_j \cup \{S\}.$$

We build $\tilde{H}(M'_{j+1})$ from scratch using the static procedure. For a fixed i , the rebuildings of $\tilde{H}(M_i)$ during addition are separated by at least $2^{i+1} - 1$ additions. Hence, the total cost of rebuildings during additions can be bounded just as in the semidynamic setting.

We thus get the following generalization of Lemma 8.6.1.

Lemma 8.6.2 *The amortized cost of deletion is $O(D(m) + T(m)/m)$, where m denotes the number of existing objects. The other bounds are just as in Lemma 8.6.1.*

In the above scheme, we assumed that we were already given an efficient algorithm for deleting an object from the search structures $\tilde{H}(M_i)$. That is generally the nontrivial part. In the next section, we shall describe this deletion operation for the search structures in Sections 8.4 and 8.5.

Exercises

8.6.1 The dynamization scheme in this section was based on the binary representation of m . In general, consider a p -adic representation of m , for any fixed constant p . Extend the scheme accordingly. Examine the various trade-offs that occur when one varies the size of p .

***8.6.2** The dynamization scheme in this section has the following problem. Whenever an update triggers a massive rebuilding operation, the response time for that update is affected badly. One solution is to carry out the rebuilding in the background, using the old search structure in the meanwhile. We have already used this idea in Exercise 4.4.3. Formalize the idea. Apply it to the semidynamic search structures for simplex and half-space range queries.

8.6.1 Dynamic range queries

Let us now apply the preceding dynamization scheme to the search structure for simplex range queries given in Theorem 8.4.6. The search structure for half-space range queries (Theorem 8.5.3) can be dynamized similarly. By Lemma 8.6.2, it suffices to deal with only deletions. Our basic scheme for deletion is quite simple. To delete a point p from the partition tree, we simply mark p as “dead” or “deleted” everywhere in the partition tree. At each node z , we now keep the count of the dead and the active, i.e., undeleted, points in the subtree rooted at z . The queries are answered just as before with minor modifications. In the report mode, we do not report the points that are marked dead. In the count mode, we only add the counts of the active points.

Now a new problem crops up. If the partition tree becomes too dirty, i.e., if it contains too many dead points, then the query time can deteriorate. Hence, we shall ensure that the number of dead points within a subtree rooted at any node z is always less than the number of active points in that subtree. This implies that the query time does not deteriorate by more than a constant factor. (Why?) Whenever this invariant is violated, we build a new subtree rooted at z . This subtree is constructed from scratch by applying the static procedure (Theorem 8.4.6) to the set of active points. Let s be the number of active points in the subtree rooted at z . Then this procedure takes $O(s \log s)$

expected time. We can amortize this cost by charging a logarithmic cost to each of the s dead points that are removed from the subtree. Any given point occurs in only one node in each level of the partition tree. The partition tree has a logarithmic number of levels. Hence, a given dead point can be charged as above in only a logarithmic number of constructions. It follows that the amortized cost of deletion is $O(\log^2 m)$, where m is the number of current points.

Combining with Lemma 8.6.2 and Theorem 8.4.6, we get the following.

Theorem 8.6.3 *There exists a search structure for simplex range queries with the following performance: $O(n \log n)$ space requirement, $O(n \log n)$ expected preprocessing time, $O(n^{1-1/d+\epsilon})$ query time in the count mode, $O(n^{1-1/d+\epsilon} + k)$ query time in the report mode, and $O(\log^2 n)$ expected amortized cost of update. (The expectation is solely with respect to randomization in the algorithm.)*

Remark. When we say that the amortized update time is $f(n)$, we mean that starting with the empty search structure, the cost of maintenance over a sequence of n updates is $O(nf(n))$.

Exercises

8.6.3 Given a parameter m in the range $[n, n^d]$, show that there exists a search structure for answering simplex range queries with the following performance: $O(m^{1+\epsilon})$ preprocessing time, $O([n/m^{1/d}] \text{polylog } n)$ query time in the count mode, $O([n/m^{1/d}] \text{polylog } n + k)$ query time in the report mode, $O(m^{1+\epsilon}/n)$ amortized update time. All bounds are expected bounds, where the expectation is solely with respect to randomization in the algorithm. (Hint: Combine the techniques in Theorems 8.6.3 and 6.5.5. Refer to Exercise 8.4.2.)

† Can you get rid of the m^ϵ factor in these costs?

8.6.4 Dynamize the search structure for half-space range queries (Theorem 8.5.3) in a similar fashion. Combine this search structure with the one in Section 7.7.4. Obtain space-time trade-off as in Exercise 8.6.3, with d replaced by $\lfloor d/2 \rfloor$.

8.7 Parametric search

The dual of the half-space emptiness problem (Section 8.5.2) is the half-line emptiness problem (Figure 8.16(a)): Given a set N of hyperplanes in R^d and a query point p , verify if the vertical half line originating from p is empty. By an empty half line, we mean that it does not intersect any hyperplane in N .

In other words, the emptiness problem is a *decision problem*. It is equivalent to the following:

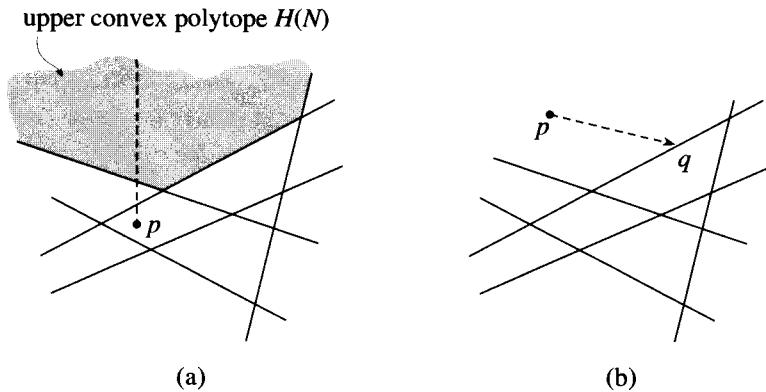


Figure 8.16: (a) A decision problem. (b) A search problem.

Polytope membership problem: Given a point p , does it lie within the upper convex polytope of N ?

Let us denote the upper convex polytope of N by $H(N)$. We are restricting ourselves to upper convex polytopes only for the sake of convenience. We shall deal with arbitrary convex polytopes in the exercises.

Related to the above decision problem is the following *search* problem (Figure 8.16(b)):

Ray shooting problem: Given a point p in $H(N)$ and a ray emanating from p , determine the first hyperplane h hit by this ray, if any. (One can verify whether p is indeed within $H(N)$ by using an algorithm for the previous membership problem.)

Well, what is the connection between the search problem and the decision problem? One connection is clear. If some oracle tells us a solution to the search problem, then using the algorithm for the decision problem, one can verify whether the solution is correct or not. Let us see how. Let us say the oracle tells us that h is the first hyperplane hit by the ray. Let q be the intersection of h with the given ray emanating from p . We only need to verify the q also lies within $H(N)$. If so, the solution provided by the oracle is correct.

There is, in fact, a deeper connection. Under reasonable assumptions, a procedure for the decision problem can also be used for finding the answer for the the search problem, not just for verifying the answer. This will imply, for example, that a data structure for polytope membership can also be used for answering ray shooting queries.

In this section, we shall develop this connection. It is based on a technique called *parametric search*. It is a very powerful technique, which is applicable

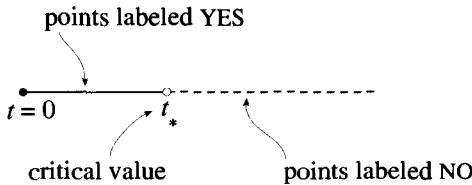


Figure 8.17: Critical parameter value.

to a large number of problems. We shall first describe it in a general form and then apply it to the above ray shooting problem.

Let us reexamine the ray shooting problem a bit more before we move to the abstract setting. Conceptually speaking, it can be paraphrased as follows. (Refer to Figure 8.16(b).) Fix the query point p . Fix the ray emanating from p . Let t be a linear parameter so that the points on the ray are of the form $q(t)$, for $t \in [0, \infty]$, where $q(t)$ is a linear function and $q(0) = p$.

Let us label a point $t \in [0, \infty]$ on the real line “YES” if $q(t)$ lies within $H(N)$ and “NO” otherwise (Figure 8.17). Note that the intersection q of the ray with the first hyperplane h (Figure 8.16) corresponds to the right endpoint of the interval consisting of the points labeled YES. The value of the parameter at this point is said to be critical. Our problem is to find this critical value of the parameter. We shall denote it by t_* .

With the above example in mind, we shall develop the parametric search technique in the following abstract setting. Let N denote the set of objects involved in the given decision problem. We assume that the search problem is stated in terms of a real nonnegative parameter t . This parameter is assumed to have the following properties:

1. Each value of the parameter corresponds to a decision problem with respect to N . We assume that we are given a procedure for resolving the resulting decision problem. Given a parameter value for t , the decision procedure answers either YES or NO.
2. The points in the half line $[0, \infty]$, for which the answer is YES, form a connected interval, possibly unbounded. We assume that one endpoint of this interval is 0. The search problem is finding the other endpoint of this interval, if any. This point, if it exists, is called the critical point. It separates the points with YES and NO answers.

In what follows, we shall denote the critical parameter value by t_* . We assume that the decision procedure can detect when its input parameter value is critical. In this case, it should output “CRITICAL.” It should also be able to furnish the required auxiliary information regarding this critical value. For example, refer to Figure 8.16(b); here the decision procedure should tell us the hyperplane h containing the point $q = q(t_*)$. This additional information

can usually be provided by a simple modification to the decision procedure; for example, see Exercise 8.5.9 for the half-line emptiness problem.

Let us denote the decision procedure given to us by P . Let T_P denote the maximum running time of P on any parameter value. In what follows, we shall let t_* denote the critical parameter value.

Our goal is to determine t_* . Even if we do not know t_* , we can always compare any given parameter value a with t_* . All that we need to do is to invoke P on the parameter value a . Depending upon whether the output is “YES,” “NO,” or “CRITICAL,” we can conclude that $a < t_*$, $a > t_*$, or $a = t_*$, respectively. The cost of this comparison is T_P . In general, we can compare any two rational functions of t_* of bounded degree in $O(T_P)$ time, without knowing t_* . By a rational function, we mean a ratio of two polynomial functions. It is said to be of a bounded degree, if both the numerator and the denominator are polynomials of bounded degree. It is easy to see that the comparison of two rational functions of t_* can be reduced to testing the sign of a single suitable polynomial function $f(t)$ at the critical value t_* . Let a_0, a_1, \dots be the roots of $f(t)$. If $f(t)$ has a bounded degree, all roots of $f(t)$ can be found in $O(1)$ time⁵: This can be done by a numerical method such as Newton’s approximation. We shall not go into these numerical methods here because in our applications $f(t)$ will be linear; hence, this issue would never arise. To test the sign of $f(t)$ at $t = t_*$, it suffices to compare the roots a_0, a_1, \dots with t_* . We have already seen that each such comparison can be done in $O(T(P))$ time. Thus, we are led to the following crucial observation.

Observation 8.7.1 Two rational functions in t_* of bounded degree can be compared in $O(T_P)$ time, even without knowing the value of t_* .

Let us now turn to our main search problem, namely, finding the value of t_* . We shall denote our procedure for this search problem by Q . Surprisingly enough, it is going to be just the decision procedure P , but with one crucial difference. The input to the decision procedure P is always a specific parameter value. The input to Q is going to be the critical value t_* , whose value we do not even know! In fact, our problem is to find the value of t_* . So what does it mean to run Q on t_* ? Let us elaborate this a bit. The procedure Q is going to simulate P , treating t_* as a variable. The quantities that arise in this simulation are going to be rational functions in t_* . The simulation of the arithmetic operations $+, -, *,$ and $/$ is easy. We just need to interpret these operations over the domain of rational functions.

But how does Q simulate a comparison operation? This requires Q to compare two rational functions in t_* . If these rational functions have

⁵We are using here the real-RAM model of computation. In practice, the running time also depends on the required precision.

a bounded degree, this comparison can be done in $O(T_P)$ time (Observation 8.7.1). When the dimension d of the problem is fixed, this bounded-degree assumption is usually satisfied. In fact, in our ray shooting application, the quantities will turn out to be linear in the parameter value t_* . (This is because we will only use comparisons of the form: Given a point and a hyperplane, does the point lie on the hyperplane? If not, which side of the hyperplane contains the point? This is a linear test.)

We conclude that Q can correctly simulate P on t_* without even knowing the actual value of t_* . Eventually, Q would output “CRITICAL” just as P would if it were given the actual value of t_* .

On the other hand, when Q outputs “CRITICAL,” it must have accumulated enough information, via the comparison operations, to be able to do so. Since the critical value is unique (if it exists), the constraints that correspond to the executed comparison operations must have a unique solution (if any). This solution must be t_* . In other words, Q knows the actual value of t_* by the time it is done. If t_* does not exist, Q should come to know that, too.

With the above discussion in mind, let us assume that Q maintains throughout its execution a certain interval I that is supposed to contain the critical value t_* . Initially, $I = [0, \infty]$. When Q executes a comparison operation, the interval I is shrunk suitably. We have already seen that each comparison operation can be reduced to $O(1)$ comparisons between t_* and the roots a_0, a_1, \dots of a certain polynomial. Without loss of generality, assume that these roots are in the increasing order. If in the comparison, it is discovered that t_* is equal to some root, Q outputs this value of t_* , announces “CRITICAL,” and stops. If the comparison indicates that t_* lies between the roots a_j and a_{j+1} , we shrink the current interval I by intersecting it with (a_j, a_{j+1}) . If this intersection is empty, we conclude that t_* does not exist, and hence stop.

Otherwise, Q continues the simulation of P . Since this simulation is correct, Q must eventually announce “CRITICAL” if t_* actually exists. At the same time, it will also output the unique value of t_* . If t_* does not exist, the interval I will become empty at some point during the execution. Hence, Q can determine that, too.

What is the cost of this simulation? Q simulates at most T_P instructions of P . If an instruction involves a comparison, its simulation costs $O(T_P)$ time. Hence, the total cost of simulation is $O(T_P^2)$. If T_P is small, this cost might be acceptable.

If T_P is not small, it is desirable to reduce the cost of simulation further. We shall now give one scheme for this reduction. It assumes that there is a fast parallel version of the decision procedure P . By a parallel algorithm, we mean an algorithm that can execute in one step several *independent* instruc-

tions in parallel. We shall only be interested in conceptual parallelism. So we do not need to worry about the processor allocation or the coordination among the processors and so on. Let us denote the parallel version of P by \bar{P} . Now Q will simulate the parallel version \bar{P} instead of P . Let $T_{\bar{P}}$ denote the maximum number of parallel steps taken by \bar{P} on any input. Let $W_{\bar{P}}$ denote the maximum number of independent instructions that are executed in one parallel step. Thus, one parallel step involves at most $W_{\bar{P}}$ independent comparisons.

How do we simulate these comparisons? We have seen that each comparison can be simulated in $O(T_P)$ time; we continue to use the serial version P during the simulation of a comparison. If we simulated the $W_{\bar{P}}$ comparisons of one parallel step separately, this would take $O(W_{\bar{P}}T_P)$ time. However, we can reduce the simulation cost of one parallel step to $O(W_{\bar{P}} \log W_{\bar{P}} + T_{\bar{P}} \log W_{\bar{P}})$ time as follows.

We have already seen that each comparison can be reduced to testing the sign of a single polynomial at $t = t_*$. For this, it suffices to compare the roots of this polynomial with t_* . However, this comparison with t_* involves a call to P . Hence, our goal is to reduce the number of such calls. This can be done as follows. We first sort the roots of all polynomials involved in the independent sign tests that arise in a given parallel step. The number of such roots is $O(W_{\bar{P}})$, because each polynomial has a bounded degree. Hence, this sorting takes $O(W_{\bar{P}} \log W_{\bar{P}})$ time. Now we can locate t_* among these roots by binary search. This involves $O(\log W_{\bar{P}})$ comparisons. The cost of each comparison is $O(T_P)$. It follows that the simulation of one parallel step costs $O(W_{\bar{P}} \log W_{\bar{P}} + T_{\bar{P}} \log W_{\bar{P}})$ time. In all applications that we will be interested in, $W_{\bar{P}}$ and $T_{\bar{P}}$ will be bounded by T_P . With that assumption, this cost is $O(T_P \log W_{\bar{P}})$. It follows that the total cost of simulating the parallel version \bar{P} is $O(T_{\bar{P}} T_P \log W_{\bar{P}})$.

Let us summarize our discussion so far.

Theorem 8.7.2 *The cost of parametric search is $O(T_P^2)$, when one is only given the serial version P of the decision procedure. Here T_P denotes the maximum cost of P on any parameter value. The cost of parametric search is $O(T_{\bar{P}} T_P \log W_{\bar{P}})$, when one is also given, in addition, a parallel version \bar{P} of the decision procedure. Here $T_{\bar{P}}$ denotes the maximum number of parallel steps in \bar{P} , and $W_{\bar{P}}$ denotes the maximum number of instructions executed in one parallel step.*

Let us now apply the parametric search technique to the ray shooting problem mentioned in the beginning of this section. The decision problem in this context is the polytope membership problem. In Section 7.7, we gave an $O(n^{\lfloor d/2 \rfloor} + \epsilon)$ size data structure for answering membership queries in $O(\log n)$ time. Let P denote the procedure for answering membership queries using

this data structure. Its running time T_P is $O(\log n)$. From Theorem 8.7.2, it follows that one can answer ray shooting queries in $O(T_P^2) = O(\log^2 n)$ time using the same data structure—only the query answering procedure is different.

In Section 8.5.2, we gave an almost linear size partition tree for answering half-space emptiness queries. This partition tree can be used for answering the membership question for upper convex polytopes in a dual setting (Exercise 8.5.1). The query time is $O(n^{1-1/\lfloor d/2 \rfloor} \text{polylog}(n))$. The procedure for answering the queries can be parallelized easily. The idea is to parallelize the procedure for descending from one level of the partition tree to the next. In other words, having visited the required nodes in the partition tree at any fixed level, we determine in parallel which children of these nodes, if any, need to be visited. Accordingly, we descend to these children in parallel, and so on. The partition tree of Theorem 8.5.2 has $O(\log \log n)$ levels. Hence, the query procedure based on this partition tree can be carried out in $O(\log \log n)$ parallel steps. The number of comparisons involved in any step is $O(n^{1-1/\lfloor d/2 \rfloor} \text{polylog}(n))$, because this even bounds the query cost in the sequential setting. (Remember that the decision problem corresponds, dually, to the half-space emptiness problem. So we use the bound in Section 8.5.2.) It follows from Theorem 8.7.2 that ray shooting queries can be answered with the help of this data structure in $O(n^{1-1/\lfloor d/2 \rfloor} \text{polylog}(n))$ time. If one were to use the partition tree in Theorem 8.5.3 instead, the query time would become $O(n^{1-1/\lfloor d/2 \rfloor + \epsilon})$.

More generally, the procedures described in the preceding two paragraphs can be suitably combined to achieve a space–time trade-off. This is done in the exercises.

Exercises

8.7.1 Apply the parametric search technique to the data structure in Exercise 8.5.3 for answering ray shooting queries. Show that, given any parameter m in the range $[n, n^{\lfloor d/2 \rfloor}]$, there exists a search structure of $O(m^{1+\epsilon})$ size that can be used to answer ray shooting queries in $O([n/m^{1/\lfloor d/2 \rfloor}] \text{polylog}(n))$ time.

8.7.2 In the ray shooting problem considered in this section, we always assumed that the origin of the ray was within the polytope $H(N)$. Remove this restriction by appropriately modifying the parametric search technique.

8.7.3 We assumed in this section that the polytope involved in the ray shooting problem was an upper convex polytope. Extend the procedure directly to any convex polytope.

Bibliographic notes

Orthogonal range search and intersection search have been investigated by several authors, e.g., Lueker [135], Lueker and Willard [136], Willard [227], Mehlhorn and Näher [157], Smid [210], Chazelle [34, 37], Chazelle and Guibas [49, 50], Edelsbrunner [86], Vaishnavi and Wood [218], McCreight [150], Van Kreveld and Overmars [219], and Scholten and Overmars [197]; a good survey is given in [63]. Edelsbrunner and Maurer [90] give search trees for orthogonal intersection queries unifying the ideas in several of the works mentioned above. Section 8.1 describes randomized analogues of these earlier deterministic and dynamic search structures. The deterministic counterparts are covered in the exercises. The randomized search structure seems especially simpler in the dynamic setting. For dynamic orthogonal range searching problem, Fredman [105] proves, in a reasonable model of computation, an $\Omega(n \log^{\omega} n)$ lower bound on the cost of executing a sequence of total n updates and queries. This matches the upper bound in Theorem 8.1.5 for the special case of orthogonal range search. Lower bounds for static range queries have been investigated in [35, 217, 228].

Segment trees (deterministic) were introduced by Bentley [20]. Section 8.1.1 describes a randomized segment tree based on skip lists (instead of deterministic search trees), and Exercise 8.1.2 gives a randomized segment tree based on randomized binary trees. The deterministic version is covered in Exercise 8.1.4. Overmars [176] shows how segment trees can be used for vertical ray shooting among nonintersecting segments in the plane and for dynamic planar point location; Sections 8.2 and 8.3 are based on his work, with the use of a deterministic segment tree substituted by the use of its simpler, randomized analogue. The deterministic version is covered in the exercises. This dynamic planar point location algorithm is off the optimal solution by (poly)logarithmic factors. Currently, almost optimal, deterministic algorithms are known for dynamic planar point location [58, 62, 106, 113, 186]; [63] provides a good survey of the static and dynamic planar point location problem. We have also seen other randomized dynamic planar point location structures earlier in this book—the one based on dynamic sampling (Section 5.6.1) and the ones based on dynamic shuffling and lazy balancing (Chapter 4). These search structures were described for the more general problem concerning trapezoidal decompositions. In the special case of planar point location, the segments do not intersect, but can share endpoints, and thus give rise to a planar graph.

Solutions for simplex and (or) half-space range search based on various kinds of partitioning schemes have been extensively investigated in computation geometry by several authors, e.g., Willard [226], Edelsbrunner and Welzl [97], Yao and Yao [230], Haussler and Welzl [119], Chazelle [34], Chazelle, Guibas, and Lee [51], Chazelle and Welzl [57], Matoušek [140, 141, 147], and Paterson and Yao [182]. The use of randomization in the construction of range search trees was initiated by Haussler and Welzl [119], and pursued further by several authors, e.g., Chazelle, Sharir, and Welzl [56], Clarkson and Shor [72], Matoušek [147, 141, 140], and Mulmuley [167]. The partition trees for simplex range search (Section 8.4), half-space range search (Section 8.5) and half-space emptiness (Exercise 8.5.8) given here are due to Matoušek [147, 141]. Actually, Matoušek even shows how to derandomize these trees—this topic will be taken up in Chapter 10. For simplex range search,

the performance of these partition trees almost matches Chazelle's lower bound [38, 39], especially if one takes into account the improvements accomplished in [140]. Even for half-space range search this is expected to be the case, but a tight lower bound for half-space range search has not yet been established; see Brönnimann and Chazelle [28] for a recent progress. For dimension less than four, the partition trees in Section 8.5 are not optimal like the earlier solutions (Exercise 8.5.7) given by Chazelle, Guibas, and Lee [51], for $d = 2$, and Chazelle and Preparata [55], with an improvement by Aggarwal et al. [5], for $d = 3$. The filtering search strategy mentioned in Section 8.5 was proposed by Chazelle [34]. For space-time trade-offs (Exercise 8.4.2, Exercise 8.6.3), see, for example, [3, 56, 147, 141].

The study of dynamization of decomposable search problems (Section 8.6) was initiated by Bentley and Saxe [22] and pursued further by several authors, e.g., Overmars [175], Overmars and Van Leeuwen [179], Maurer and Ottmann [148], Mehlhorn [155], Agarwal and Matoušek [2]; Overmars' thesis [175] and a more recent [63] provide good surveys of this method. Dynamization of the simplex and half-space range search methods (Section 8.6.1) is due to Matoušek [147, 141] and Agarwal and Matoušek [2], respectively.

The parametric search method (Section 8.7) is due to Megiddo [153]. It has found extensive applications in computation geometry. We will mention just a few: Agarwal and Matoušek [3], Matoušek and Schwarzkopf [142], Chazelle, Edelsbrunner, Guibas and Sharir [45], Cole [74], and Alon and Megiddo [9]. The ray shooting application in Section 8.7 is due to Agarwal and Matoušek [3]. For more on ray shooting within convex polytopes, see Schwarzkopf [201]. Quite often, the use of parametric search can be replaced by a simpler randomized algorithm; for example, see [82, 139] for the special slope selection problem.

Chapter 9

Computer graphics

In this chapter, we shall apply the randomized methods developed earlier in this book to some basic problems in computer graphics. The basic problem in computer graphics is roughly the following. One is given some representation of a three-dimensional world. Given a viewpoint, the goal is to render the view visible from that viewpoint. The problem depends on several things: the attributes of the objects in the given world, such as their color and texture, positions of the light sources, and so on. It gets even more challenging when the viewpoint or the objects are moving. We shall begin by stating various simplified versions of this general problem.

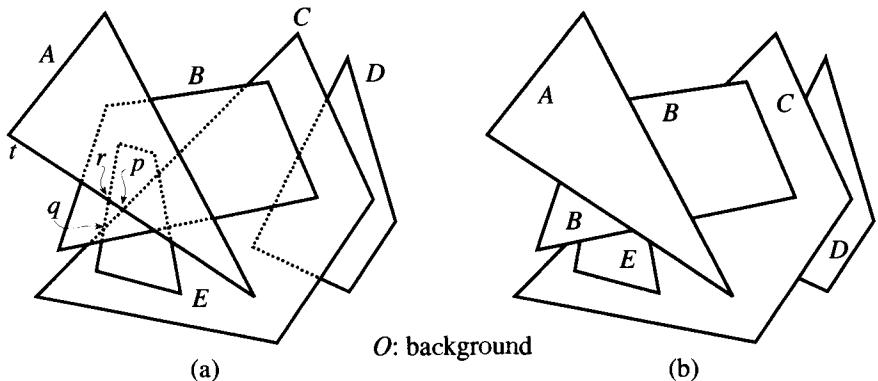


Figure 9.1: Hidden surface removal: (a) A set N of polygons. (b) View map.

One simple version is the following *hidden surface removal problem* in the so-called *object space* setting. Assume that all objects in the scene are opaque polygons in R^3 . Let N denote this set of polygons. For the sake of simplicity, assume that the polygons are non-intersecting; but they can share

edges. Assume that we are given a fixed viewpoint $v \in R^3$ and a view plane V close to v . The goal is to construct the view visible from v . What this means is the following. Project the visible parts of all polygon edges onto the view plane V (Figure 9.1). (A point $p \in R^3$ is projected to the intersection of the view plane V with the line passing through p and v .) Each region in the resulting partition of the view plane is labeled with the polygon in N that is visible in that region. This labeled partition of V is called the *view map (graph)*.¹ Given a set N of polygons, a viewpoint v , and a view plane V , the problem is to construct the resulting view map. Once this is done, each region in the view map can be colored in accordance with the attributes of the polygon visible in that region. Note that this version of the hidden surface removal problem is highly simplified. In reality, one also needs to take into consideration several additional features of the problem, such as shadows, positions of the light sources, reflectance properties of the objects, and so on.

A hidden surface removal algorithm that explicitly constructs the view map is called an *object space* algorithm. There is another way of approaching the hidden surface removal problem that is based on the way the real visual devices work. Let us assume that somehow the polygons in N can be linearly ordered with respect to the given viewpoint v so that the resulting linear order has the following property: If a polygon $P \in N$ overlaps a polygon $Q \in N$, when seen from v , then P must precede Q in the linear order. Such a linear order, if it exists, is called a *priority order*. If one knew a priority order among the polygons in N , then one could “paint” the polygons in N onto the device screen in the decreasing priority order. This means if a polygon P overlaps Q then P will be painted after Q . In the process, the overlapped part of Q becomes invisible. What does “painting” on a device screen mean? Well, the actual device screen is divided in a grid-like fashion into several minute pixels. Each pixel is allocated a location in a special fast memory. This location holds attributes of that pixel, such as the color and brightness. Painting a polygon Q simply means setting attribute values of the pixels that lie within the projection of Q onto the view window. This is done in accordance with the attributes of Q . If one paints an overlapping polygon P after Q , the overlapped pixels automatically get set according to the attributes of P .

We were assuming above that a priority order indeed exists among the given set of polygons. A little thought tells us that this need not always be the case. For example, some polygons in N can overlap in a cyclic fashion (Figure 9.2(a)). In this case, one can break the polygons in N so as to

¹The term view map is preferable, because the term view graph has also other meanings in computational geometry.

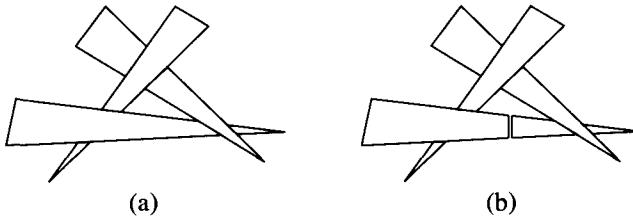


Figure 9.2: (a) A cyclic overlap. (b) After removing the cyclic overlap.

remove the cyclic overlaps (Figure 9.2(b)). There is an obvious problem here, namely, how can one decompose the given set of polygons so that the resulting fragments do not admit overlapping cycles with respect to any viewpoint? Here, one would want the number of fragments to be as small as possible. In addition, one must also be able to generate a priority order among these fragments quickly for any given viewpoint. In this chapter, we shall study one method that addresses these problems. It is based on the so-called Binary Space Partitions (BSP).

So far, we have been assuming that the viewpoint is fixed. What should one do if the viewpoint itself is moving? One simple solution would be to compute the view with respect to every position of the viewpoint from scratch. This might turn out to be too expensive, especially in a situation, as in flight simulation, where one is expected to generate a large number of views per second. In this connection, it becomes desirable to preprocess the given set N and build an appropriate data structure so that, given any viewpoint v , the view visible from v can be constructed quickly.

Finally, we should also mention that all of the problems discussed above admit dynamic variants. In a dynamic setting, the user is allowed to add or delete a polygon in N and the goal is to update the relevant data structures quickly.

In the following sections, we shall attack the preceding problems, one by one.

Exercise

9.0.1 Assume that the viewpoint is fixed. Show that in the hidden surface removal problem one can assume, without loss of generality, that the viewpoint is located at $(0, 0, -\infty)$ and that the view plane coincides with the plane $z = 0$. (Hint: Push the viewpoint to $(0, 0, -\infty)$ using a perspective transformation. Transform the polygons in N accordingly.)

9.1 Hidden surface removal

In this section, we shall consider the object space version of the hidden surface removal problem, assuming that the viewpoint is fixed. Let N be a given set of polygons in R^3 . We shall assume that the polygons have bounded face lengths and that they are non-intersecting. Intersecting polygons will be treated in the exercises. We can also assume, without loss of generality, that the viewpoint is located at $(0, 0, -\infty)$ and that the view plane coincides with the plane $\{z = 0\}$ (Exercise 9.0.1). Project the visible parts of all polygon edges onto the view plane $\{z = 0\}$. Our goal is to construct the resulting view map. Each region of the view map is to be labeled with the polygon visible in that region. In this section, we shall give a randomized incremental algorithm for this problem. We shall see later that this algorithm can also be made fully dynamic through the use of history.

What we shall actually compute is the *view partition*, which is defined as the trapezoidal decomposition of the view map (Figure 9.3(a)). This is desirable in practice because the regions in the view map can have complicated shapes, and this can make the “painting” of these regions onto the device screen difficult. For the sake of space efficiency, we shall extend vertical attachments through the T -junctions in the view map, such as u in Figure 9.3(a), only on one side. We shall denote the view partition by $H(N)$. Every trapezoid in the view partition will be labeled with the unique polygon in N that is visible in it. If no polygon in N is visible in the trapezoid, it will be labeled with the background face. Our goal is to construct $H(N)$, given N .

We shall construct $H(N)$ incrementally by adding the polygons in N , one at a time, in random order. For $1 \leq i \leq n$, let N^i denote the set of the first i polygons in this addition sequence. At the i th stage of the algorithm, we shall maintain the partition $H(N^i)$ (Figure 9.3). In addition, we shall also maintain, with each trapezoid $f \in H(N^i)$, a list $L(f)$ of the polygons in $N \setminus N^i$ *conflicting* with f (Figure 9.3(b)). We say that a polygon $S \in N \setminus N^i$ conflicts with f , if a part or the whole of S is visible in f , i.e., if the polygons in N^i do not hide S completely within f . Conversely, with each polygon in $N \setminus N^i$, we shall maintain a list of the trapezoids in $H(N^i)$ in conflict with it. The motivation behind maintaining the conflict information is the same as in the previous randomized incremental algorithms: It aids the addition of a new object (polygon) to $H(N^i)$.

Let us also specify in more detail the representation of $H(N_i)$ that we shall be using. One possibility is the obvious planar graph representation. But we shall find it more convenient here to use an alternative representation, wherein a trapezoid and a vertex in $H(N^i)$ are considered adjacent iff that vertex is a corner of the trapezoid. For example, in Figure 9.3(b) the

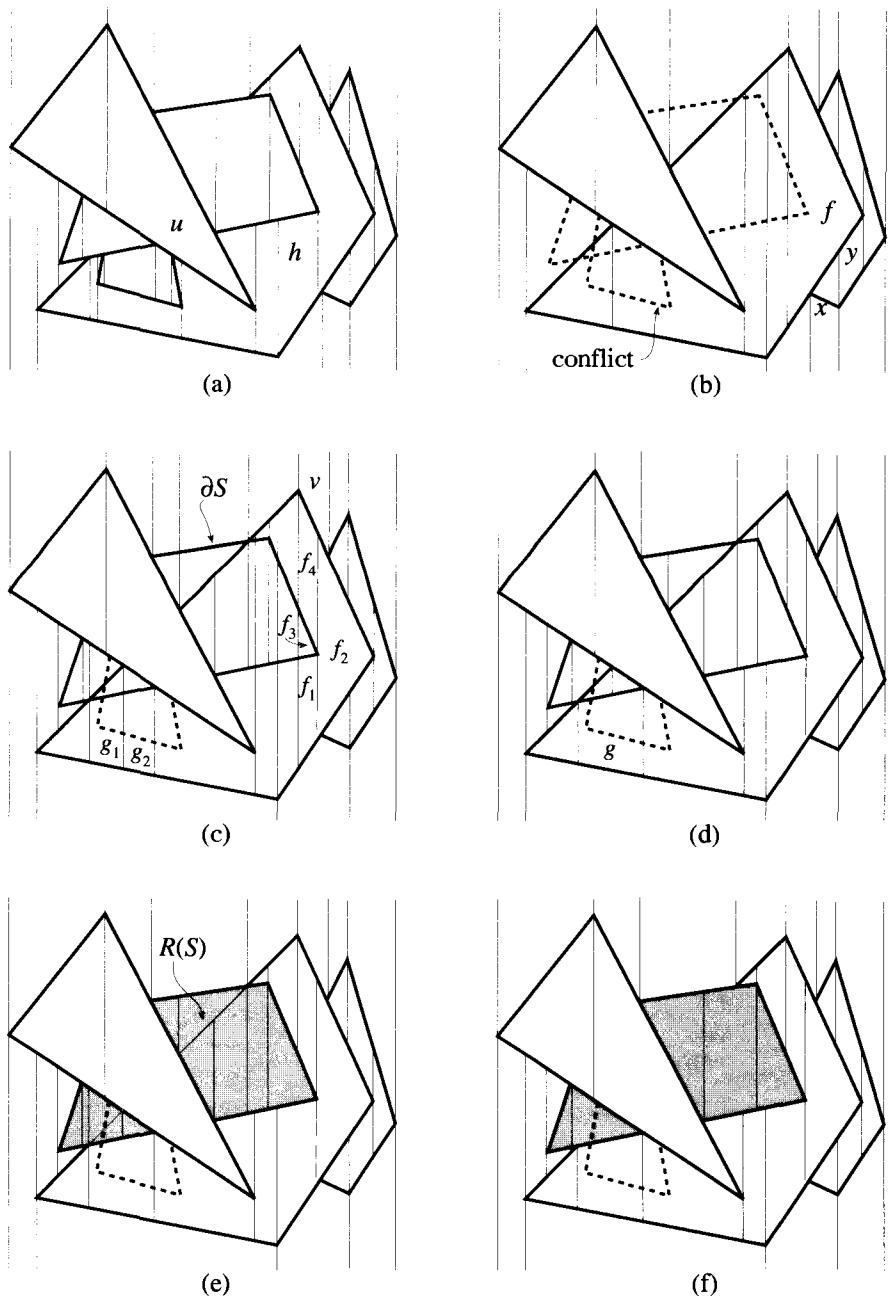


Figure 9.3: Hidden surface removal: (a) $H(N)$. (b) $H(N^3)$. (c) $H_1(N^4)$. (d) $H_2(N^4)$. (e) Region $R(S)$ before reconfiguration, (f) and after reconfiguration.

vertices x and y are not considered adjacent to f . In this representation, each trapezoid is represented by a circular list of its corner vertices. We also maintain adjacency relationships as specified above among the trapezoids and the vertices in $H(N_i)$.

Addition of the $(i+1)$ th polygon $S = S_{i+1}$ to $H(N^i)$ is achieved as follows (Figure 9.3). The conflict information tells us all trapezoids in $H(N^i)$ where S would be visible (Figure 9.3(b)). Let f be any such trapezoid. There are two cases to consider.

1. The boundary ∂S is visible in f (Figure 9.4(a)): In this case, we split f along the projection of ∂S , and further refine this partition of f into trapezoids (Figure 9.4(b)). Because the face-length of S is bounded, the cost of splitting f is $O(1)$. Note that a large number of vertices, besides the corners of f , can possibly lie on the upper and lower sides of f (Figure 9.4). The crucial point here is that these vertices are not involved in the splitting operation, because they are not considered adjacent to f in our representation. We label each of the split trapezoids with either S or the previous polygon labeling f , depending upon which one is the new polygon visible in that trapezoid. We also build conflict lists of the split trapezoids from the conflict list of f .
2. The boundary ∂S is not visible in f (Figure 9.4(c)): In this case, S is going to be in front of all other polygons in N^{i+1} , as seen within f . So we label f with S . We also update the conflict list of f by removing all polygons in $N \setminus N^{i+1}$ that are behind S , as seen within f .

The above procedure is applied to all trapezoids of $H(N^i)$ in conflict with S . Let us denote the resulting partition by $H_1(N^{i+1})$ (Figure 9.3(c)). In the next step, we contract all vertical attachments in $H(N^i)$ intersecting the pro-

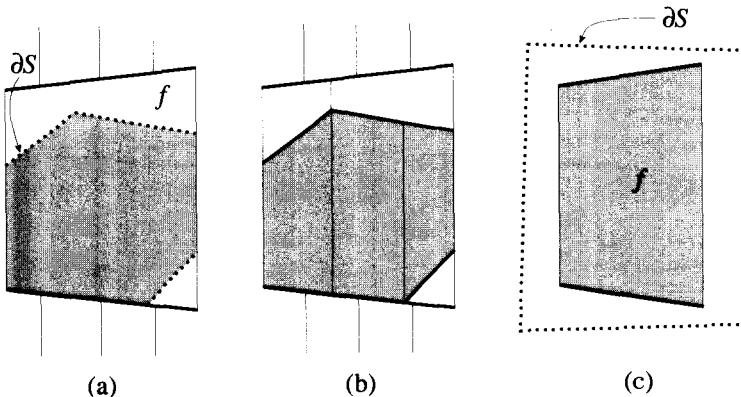


Figure 9.4: Splitting of a trapezoid.

jected visible boundary of S . More precisely, if a vertical attachment through a junction v intersects the projected visible boundary of S , we retain only that part of the vertical attachment which is adjacent to v (Figure 9.3(d)). These contractions cause several trapezoids in $H_1(N^i)$ to merge. For example, consider the case when the trapezoids f_1, f_2, \dots in $H_1(N^i)$ are merged into a single trapezoid f (Figure 9.5). The labeling polygon of f_i must be the same for all i . We label f with this unique polygon. It is also necessary to merge the conflict lists of f_1, f_2, \dots into a single conflict list. If the same polygon in $N \setminus N^{i+1}$ occurs in several of these conflict lists, only one occurrence of that polygon is to be retained. Merging of the conflict lists can be done in time proportional to the sum of the conflict sizes of f_1, f_2, \dots .

Let us call the partition obtained at the end of the above contraction procedure by $H_2(N^{i+1})$ (Figure 9.3(d)). Our discussion so far shows the following:

Lemma 9.1.1 $H_2(N^{i+1})$ can be obtained from $H(N^i)$ in $O(\sum_f 1 + |L(f)|)$ time, where f ranges over all trapezoids in $H(N^i)$ in conflict with S .

Next, consider the region $R = R(S)$ formed by the union of all trapezoids in $H_2(N^{i+1})$ labeled with S . In Figure 9.3(e), this region is shown shaded. In general, R can have several components and these components can have complicated shapes. In addition, R can contain several edges of the old view map that became invisible during the addition of S (Figure 9.6). These segments must be removed. This can be done as follows. We redecompose R into trapezoids by applying the trapezoidal decomposition algorithm (Section 3.1) to the set of segments on the boundary of R (Figure 9.6(a)–(b)). The expected time taken by this procedure is proportional to the number of segments on the boundary of R , ignoring a logarithmic factor. As a by-product, we also get a point location structure for the trapezoidal decomposition of R (Section 3.1.1). This will be useful when we build conflict lists of the new trapezoids within R . We address this problem next.

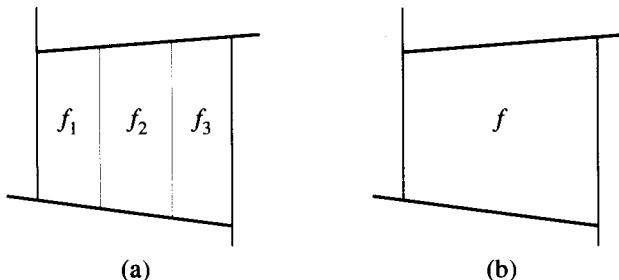


Figure 9.5: Merging of trapezoids.

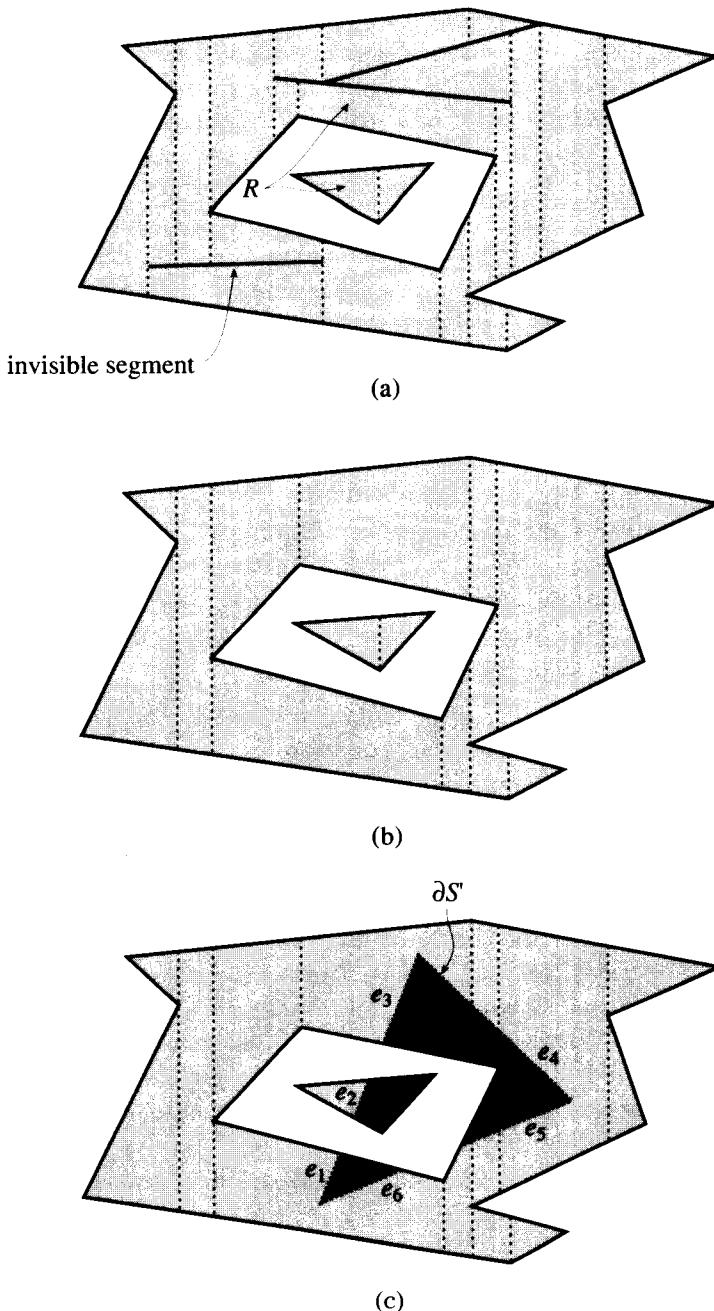


Figure 9.6: Reconfiguration: (a) Region $R = R(S)$ (shown shaded) before, and (b) after reconfiguration. (c) Conflict relocation within $R(S)$.

Consider the set of polygons in $N \setminus N^{i+1}$ that conflict with the trapezoids within R . This set is the same for the new as well as the old decomposition of R . Hence, we know it already. Fix any polygon S' in this set. Let e_1, e_2, \dots be the visible segments within R of the projected boundary $\partial S'$ (Figure 9.6(c)). From the conflict information associated with the trapezoids in the old decomposition of R , we can easily determine the endpoints of all segments e_1, e_2, \dots . Let v_1, v_2, \dots denote these endpoints. Using the point location structure associated with the new trapezoidal decomposition of R , we can locate each such endpoint in logarithmic time (with high probability, strictly speaking). Now, we search within the new trapezoidal decomposition of R , starting at these points and also the junctions on the boundary of R where S' is visible (which we can determine from the old conflict information), so as to access all new trapezoids within R conflicting with S' . This takes time proportional to the number of such trapezoids. In Figure 9.6(c), these are the trapezoids that intersect the darkly shaded region.

We repeat the same procedure for all polygons in $N \setminus N^{i+1}$ that are in conflict with the trapezoids within R . At the end of this procedure, we have the sought partition $H(N^{i+1})$, with the updated conflict information (Figure 9.3(f)). This finishes the description of the algorithm.

Our discussion so far proves the following.

Lemma 9.1.2 $H(N^{i+1})$ can be obtained from the intermediate partition $H_2(N^{i+1})$ in expected time proportional to $\sum_g 1 + |L(g)|$, ignoring a logarithmic factor, where g ranges over the old and the newly created trapezoids within R .

Remark. The expectation here is solely with respect to randomization in the trapezoidal decomposition algorithm (Section 3.1), which is used for re-configuration. We could also have used for this purpose the deterministic procedure (Sections 2.8.3 and 2.8.5). In that case, the above bound will be worst case, and that is what we shall pretend in what follows. Finally, we are only going to be interested in the expected running time of the whole algorithm. So, it will be irrelevant whether the reconfiguration is done in a deterministic or a randomized fashion. But the randomized reconfiguration procedure is definitely simpler.

Combining the above lemma with Lemma 9.1.1, we get the following estimate.

Lemma 9.1.3 The view partition $H(N^i)$ can be updated to $H(N^{i+1})$ in time proportional to $\sum_f 1 + |L(f)|$, ignoring a logarithmic factor, where f ranges over the destroyed trapezoids in $H(N^i)$ and also the newly created trapezoids in $H(N^{i+1})$.

Exercises

9.1.1 The algorithm in this section is not on-line because it maintains the conflict information. Make the algorithm on-line by making use of history (Chapter 4). Follow these steps:

- (a) Specify the link structures in detail. $\text{Link}(S_{i+1})$ should be such that, given a set Γ of killed trapezoids in $H(N^i)$ in conflict with a polygon $S' \notin N^{i+1}$, one should be able to determine quickly the set Γ' of the newly created trapezoids in $H(N^{i+1})$ in conflict with S' . This procedure should take $O(|\Gamma| + |\Gamma'|)$ time, ignoring a log factor. (Hint: Examine the proof of Lemma 9.1.3 carefully. You can use suitable planar point location structures.)
- (b) Give a procedure for conflict search through the history using the method in Chapter 4 (for example, see Section 4.2). Given a polygon $S \notin N^i$, the goal is to determine all trapezoids in $H(N^i)$ in conflict with S . Show that this can be done in time proportional to the number of nodes in the history in conflict with S , ignoring a logarithmic factor.
- (c) Conclude that the conversion into the on-line form does not change the running time of the algorithm by more than a logarithmic factor.

9.1.1 Analysis

Next, we shall estimate the expected running time of the previous hidden surface removal algorithm, assuming that the polygons in N were added in random order. Before we do so, let us pause for a while, and ask ourselves what kind of performance we will be expecting from this algorithm. We have already noted that the algorithm can be made on-line without affecting its running time by more than a logarithmic factor (Exercise 9.1.1). Hence, we shall first obtain a trivial lower bound on the expected running time of any on-line hidden surface removal algorithm on a random N -sequence of additions. Once this is done, we shall obtain an upper bound on the expected running time of our hidden surface removal algorithm and compare it with the trivial lower bound.

Clearly, the running time of any on-line hidden surface removal algorithm over a given update sequence is bounded below by the total structural change in the underlying view map. This follows trivially because the algorithm is required to maintain the view map throughout. (However, we cannot assume that the algorithm will necessarily maintain a trapezoidal decomposition of the view map, as we have done here.) We shall next calculate the expected value of this structural change. It will trivially bound from below the expected running time of any on-line hidden surface removal algorithm.

Let us imagine projecting all edges of the polygons in N orthogonally onto the view plane $\{z = 0\}$. We assume, as before, that the viewpoint is located at $(0, 0, -\infty)$. The projection gives rise to several junctions in the view plane (Figure 9.1), which are either crossings among the projected

edges, such as the crossings p, q , and r in Figure 9.1, or they are projections of the vertices of the polygons in N , such as the junction t in Figure 9.1. Let $\Pi = \Pi(N)$ denote the set of all such junctions in the view plane. Observe that the total structural change in the underlying view map during a given addition sequence \bar{u} is proportional to the total number junctions in Π that are created during \bar{u} . We say that a junction $q \in \Pi$ is *created* during \bar{u} , if it occurs as a vertex of the view map at some time during the execution of \bar{u} . Here we are not counting the destroyed junctions, because a junction, once created, can be destroyed only once. Not every junction in Π is created during the given addition sequence. Typically, only a few will be. For this reason, we shall also refer to the junctions in Π as *feasible* junctions.

Now consider a random N -sequence of additions. Denote it by \bar{u} . Our goal is to calculate the total expected structural change in the underlying view map during \bar{u} . This is clearly of the same order as the expected number of junctions in Π that are created during \bar{u} . Fix a junction q in Π . Define a 0–1 indicator function $I(q)$ as: $I(q) = 1$ if q is created during \bar{u} , and zero otherwise. Clearly, the total number of junctions in Π that are created during \bar{u} is $\sum_q I(q)$, where q ranges over all junctions in Π . Hence, the expected number of created junctions is $E[\sum_q I(q)]$. But $E[I(q)]$ is just the probability that q is created during \bar{u} . What is this probability? For this, we shall examine q a bit more closely. Let e_1 and e_2 be the two polygon edges whose projections intersect at q . Let f_1 and f_2 be the polygons in N containing e_1 and e_2 . (When q is the projection of a vertex of a polygon in N , $f_1 = f_2$.)

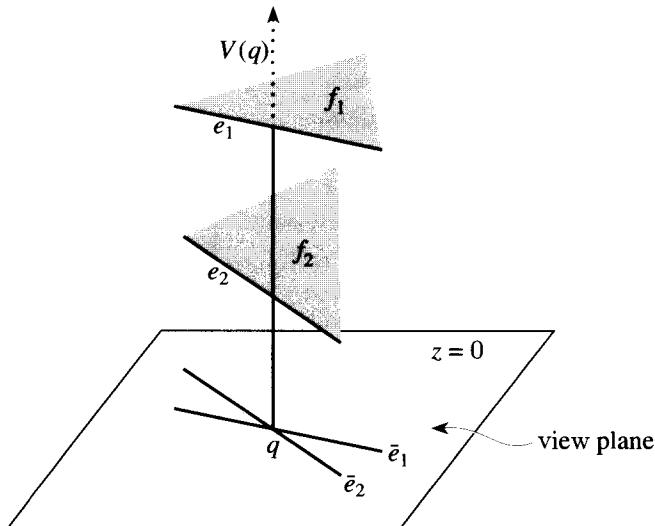


Figure 9.7: Level of a junction.

We say that f_1 and f_2 *define* q . Let $D(q)$ be the set of these polygons defining q . In Figure 9.1, the defining set of the junction q is $\{C, E\}$. The defining set of the junction t is $\{A\}$. We define the degree $d(q)$ of a junction q as the size of its defining set $D(q)$. It is two when q is a crossing. It is one when q is the projection of a vertex of a polygon in N . Let $V(q)$ be the vertical line through q parallel to the z-axis (Figure 9.7). Let $L(q)$ be the set of polygons in N that intersect $V(q)$ before e_1 or e_2 , i.e., within the dark portion of $V(q)$ in Figure 9.7. The polygons in $L(q)$ are precisely the ones that obstruct the junction q , i.e., the ones that make q invisible to the viewer at $(0, 0, -\infty)$. Let $l(q)$, the *level* (or the conflict size) of q , be defined as the size of $L(q)$. In Figure 9.1, $L(q) = \{B\}$ and $L(p) = \{B, E\}$. Note that the junctions with level 0 are precisely the ones that occur in the view map for the entire set N .

Observe that a junction $q \in \Pi$ is created during \bar{u} iff the polygons in $D(q)$ are added before the polygons in $L(q)$, i.e., iff the first $d(q)$ polygons to be added from the set $D(q) \cup L(q)$ all belong to $D(q)$. Since the addition sequence is random, this happens with probability

$$\frac{1}{\binom{l(q)+d(q)}{d(q)}}. \quad (9.1)$$

It follows that the expected number of junctions that are created during a random N -sequence of additions is

$$\sum_{q \in \Pi(N)} \frac{1}{\binom{l(q)+d(q)}{d(q)}}. \quad (9.2)$$

One more series of this form will turn out to be useful to us. So let us make a simple definition for N :

$$\Theta(N, s) = \sum_{q \in \Pi(N)} \frac{1}{[l(q) + d(q)]^{d(q)-s}}, \text{ for } s = 0, \text{ or } 1. \quad (9.3)$$

Notice that the contribution of a junction to the above series is inversely proportional to a power of its level.

It follows from (9.2) and (9.3) that the expected structural change in the view map during a random N -sequence of additions is of the order of $\Theta(N, 0)$. Thus, the expected running time of any on-line hidden surface removal algorithm over a random N -sequence of additions is trivially $\Omega(\Theta(N, 0))$. The expected running time of our on-line hidden surface removal algorithm over a random N -sequence of additions will turn out to be $O(\Theta(N, 1) \log^2 n)$. This does not match the trivial lower bound, but it is reasonably close.

So let us turn to estimating the expected running time of the randomized incremental algorithm. Our bound would also hold for the on-line form

of the algorithm, ignoring a log factor (Exercise 9.1.1). For the purpose of analysis, we shall define a certain configuration space $\Sigma = \Sigma(N)$ of all feasible labeled trapezoids over N . By a *feasible* labeled trapezoid Δ , we mean a trapezoid that occurs in the view partition $H(R)$, for some $R \subseteq N$. Its label, $\text{label}(\Delta)$, is the polygon in R that would be visible there, if the scene were to consist only of the polygons in R . The trigger set associated with this labeled trapezoid consists of the polygon labeling Δ and the polygons in R whose projected edges define (i.e., are adjacent to) Δ . Its stopper set consists of the polygons in $N \setminus R$ which conflict with Δ , i.e., which hide $\text{label}(\Delta)$ completely or partially within Δ . It is easy to see that these definitions of the trigger and stopper sets do not depend on R . Also observe that a labeled trapezoid occurs in $H(R)$ iff it is active over R as a configuration.

By Lemma 9.1.3, the total running time of the algorithm, ignoring a logarithmic factor, is proportional to the total conflict change over the configuration space Σ during the whole addition sequence. By Theorem 3.4.5, the expected value of this conflict change is of the order of

$$\sum_{i=1}^n \frac{n-i}{i} \frac{e(i)}{i}, \quad (9.4)$$

where $e(i)$ denotes the expected number of active configurations in Σ over a random sample $I \subseteq N$ of size i . In other words, $e(i)$ is the expected number of trapezoids in $H(I)$, where $I \subseteq N$ is a random sample of size i . Let Π be the set of feasible junctions as defined in the beginning of this section. Note that Π can be thought of as a configuration space, with the trigger set $D(q)$ and the stopper set $L(q)$ of each feasible junction q defined as before. Also observe that the size of $H(I)$, $I \subseteq N$, is proportional to the number of junctions in Π that are active over I ; a junction is active over I iff it occurs in $H(I)$. It follows, ignoring a constant factor, that the quantity in (9.4) can also be interpreted as the total expected conflict change over the configuration space Π (Theorem 3.4.5, Exercise 3.4.1). But the expected structural change over Π is easy to calculate directly. We have already seen that each junction $q \in \Pi$ gets created in the algorithm with probability $1/\binom{l(q)+d(q)}{d(q)}$ (cf. (9.1)). Hence, the expected conflict change over Π is of the order of

$$\sum_{q \in \Pi} \frac{l(q)}{\binom{l(q)+d(q)}{d(q)}}.$$

By (9.3), this latter quantity is of the order of $\Theta(N, 1)$. Thus, we have proved the following.

Theorem 9.1.4 *The expected running time of the randomized incremental algorithm for hidden surface removal (and also its on-line version) is proportional to $\Theta(N, 1)$, ignoring a (poly)logarithmic factor. In contrast, the*

expected running time of any on-line hidden surface removal algorithm over a random N -sequence of additions is trivially $\Omega(\Theta(N, 0))$.

Closing the gap between the upper and lower bounds remains an open problem. Note that the expected running time of our algorithm is not close to optimal in the static setting. The trivial lower bound on the running time of any static hidden surface removal algorithm is the size of the output view map. An algorithm whose running time is actually proportional to the size of the output view map is called *output-sensitive*. Randomization does not seem to provide any help in achieving complete output sensitivity. However, the randomized algorithm given before is *quasi-output-sensitive* in the following sense. For this, we reformulate the definition in (9.3) in another way. Let $m_l(N)$ denote the number of crossings in $\Pi = \Pi(N)$ with level l . Thus, $m_0(N)$ is just the number of crossings in the view map of N . Reformulating (9.3), we deduce that

$$\Theta(N, 1) = O(n) + \sum_l \frac{m_l(N)}{l+2}. \quad (9.5)$$

The first term here arises due to the contribution of the junctions in $\Pi(N)$ with degree 1; these are projections of the vertices of the polygons in N and their number is $O(n)$.

The expected running time of our hidden surface removal algorithm is proportional to $\Theta(N, 1)$. In practice, $\Theta(N, 1)$ should be “comparable” to the first term $m_0(N)$ in the above series, which is just the output size, ignoring an additive $O(n)$ term. In that sense, the preceding algorithm can be considered to be quasi-output-sensitive.

Heuristics: In practice, it may be expensive to maintain the conflict information in a manner discussed in this section. A less expensive but theoretically unjustified way is the following. We arbitrarily distinguish one vertex of each polygon in N . With each trapezoid in $H(N^i)$, we maintain a “conflict” list of the distinguished vertices whose projections lie within that trapezoid. During the addition of $S = S_{i+1}$, this information tells us the trapezoid in $H(N^i)$ containing the projection of the distinguished vertex of S (which may be invisible at this stage). We search within $H(N^i)$, starting at this trapezoid, so as to access all trapezoids that intersect the projection of S . In several of these trapezoids, S might not be visible. We simply ignore such trapezoids. The remaining trapezoids are precisely the ones that conflict with S , in the sense of this section. After this, $H(N^i)$ can be updated to $H(N^{i+1})$ as before. Update of the conflict information is now easier, and we leave it to the reader.

Exercises

***9.1.2** Generalize the algorithm in this section so that it can handle intersecting polygons. Prove the bound in Theorem 9.1.4 for this algorithm, with appropriate new interpretation.

****9.1.3** (Dynamic hidden surface removal)

- (a) Make the on-line hidden surface removal algorithm in Exercise 9.1.1 fully dynamic by allowing deletions of polygons. (Hint: Use dynamic shuffling.)
- (b) Show that the expected structural change in the visibility map over a random (N, δ) -sequence is $O(\Theta(N, 0) \log n)$, if the signature δ is weakly monotonic.
- (c) Similarly, show that the expected conflict change over a random (N, δ) -sequence is $O(\Theta(N, 1) \log n)$, if the signature δ is weakly monotonic.
- (d) Show that the expected running time of the dynamic hidden surface removal algorithm over a random (N, δ) -sequence is $O(\Theta(N, 1)\text{polylog}n)$, if the signature δ is weakly monotonic.

9.2 Binary Space Partitions

In this section, we consider the problem of priority generation that arises in connection with the image space rendering of the visible view. Recall that the problem is the following. We are given a set N of n polygons in R^3 . The goal is to break the polygons in N into pieces so that the resulting pieces do not admit overlapping cycles with respect to any viewpoint. Let $\Gamma(N)$ denote the resulting set of pieces. We want the size of $\Gamma(N)$ to be as small as possible. We also wish to have a data structure so that, given any viewpoint, a priority order over $\Gamma(N)$ with respect to this viewpoint can be generated quickly—say, in $O(|\Gamma(N)|)$ time. In this section, we shall present one method for decomposing the polygons in N and generating a priority order with respect to a given viewpoint. It is based on the so-called *Binary Space Partitions* (BSP).

9.2.1 Dimension two

We shall first illustrate the basic ideas underlying Binary Space Partitions in the simplest setting by considering the analogous problem in two dimensions. In two dimensions, the set N consists of segments in the plane rather than polygons in R^3 , and the view from any viewpoint is one-dimensional rather than two-dimensional. Observe that in two dimensions, the problem of overlapping cycles cannot arise for any viewpoint. Then why is there any need to break the segments in N ? Well, the problem is that, in general, it is difficult to generate the priority order on N with respect to the given viewpoint quickly. For this reason, we shall break the segments in N into a set $\Gamma(N)$ of fragments, so that the priority order over $\Gamma(N)$ can be generated quickly. Our

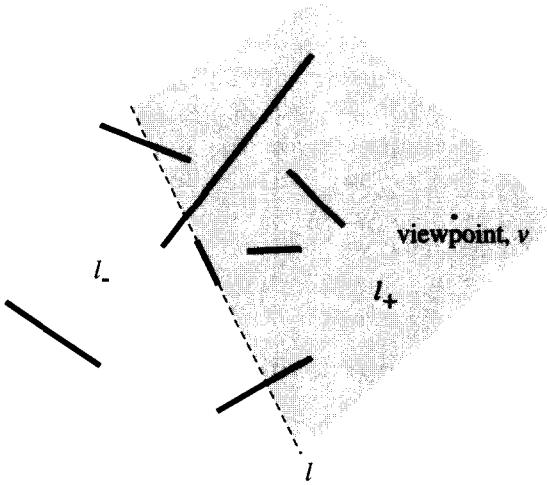


Figure 9.8: Divide and conquer.

method of decomposition will be randomized. The expected size of $\Gamma(N)$ will turn out to be $O(n \log n)$. Concurrently, we shall also build a data structure, called a *randomized BSP tree*, which will be used for generating a priority order over $\Gamma(N)$ with respect to the given viewpoint. The nodes of the BSP tree will be in one-to-one correspondence with the fragments in $\Gamma(N)$.

So let N be the given set of n segments. The basic idea underlying a BSP tree is based on the following divide-and-conquer paradigm (Figure 9.8). Fix any line l in the plane. Let l_+ and l_- denote the two half-spaces bounded by l . As a convention, we shall let l_+ denote the upper half-space bounded by l and let l_- denote the lower half-space bounded by l ; if l is vertical, we label the half-spaces arbitrarily. We allow l to contain segments in N . If the segments in N are in general position—as we shall assume in what follows—then l can contain at most one segment in N . Cut the segments in N along l . Let \bar{N} denote the set of resulting fragments (pieces). Partition \bar{N} into three sets: \bar{N}_0 , the set of pieces contained in l (possibly empty); \bar{N}_+ , the set of pieces contained in l_+ ; and \bar{N}_- , the set of pieces contained in l_- . Observe that, for any viewpoint $v \in l_+$, a priority order for \bar{N} can be obtained by concatenating the priority orders for \bar{N}_+ , \bar{N}_0 , and \bar{N}_- , in that order. This follows because no segment in \bar{N}_- can overlap a segment in $\bar{N}_0 \cup \bar{N}_+$ as seen from v , and similarly, a segment in \bar{N}_0 cannot overlap a segment in \bar{N}_+ (Figure 9.8). If the viewpoint is contained in l_- , a priority order for \bar{N} can be obtained by concatenating the priority orders for \bar{N}_- , \bar{N}_0 , and \bar{N}_+ , in that order.

This leads us to the following preprocessing algorithm. Given a set N of segments, the algorithm cuts these segments into pieces (fragments), and concurrently builds a so-called BSP (Binary Space Partition) tree. We shall denote this tree by $\text{BSP}(N)$. The nodes of $\text{BSP}(N)$ will be in one-to-one correspondence with the generated fragments. The reader should compare the resulting randomized BSP tree with a randomized binary tree (Section 1.3): a randomized binary tree can be thought of as a randomized BSP tree in one dimension. In the beginning, we shall sort the segments in N in a random order. Once this order on N is fixed, $\text{BSP}(N)$ is completely determined as follows.

Algorithm 9.2.1 (Randomized BSP tree)

1. If N is empty, the BSP tree is NIL (empty).
2. Otherwise, randomly choose a segment $S \in N$. This is equivalent to choosing the first segment in N according to the initial (randomly chosen) order. Label the root of $\text{BSP}(N)$ with S .
3. Let l denote the line through S . Cut the segments in N , other than S , along l . Let \bar{N}_+ and \bar{N}_- be the resulting sets of fragments contained in the half-spaces l_+ and l_- , respectively. The orderings on \bar{N}_+ and \bar{N}_- are derived from the ordering on N in a natural way.
4. Let the left (negative) and the right (positive) subtrees of $\text{BSP}(N)$ be the recursively computed trees $\text{BSP}(\bar{N}_-)$ and $\text{BSP}(\bar{N}_+)$.

Each node σ in $\text{BSP}(N)$ can be naturally identified with a string of + and – symbols that corresponds to the path from the root to σ . This string will be called the signature of σ . Figure 9.9 gives an example of a BSP tree. The nodes of the BSP tree in Figure 9.9(b) are labeled with their signatures. The fragments in Figure 9.9(a) are labeled with the signatures of the corresponding nodes in the BSP tree. We shall denote the fragment labeling a node σ in $\text{BSP}(N)$ by $S(\sigma)$. The set of fragments labeling the nodes of the subtree rooted at σ will be denoted by $N(\sigma)$.

Observe that each node σ of $\text{BSP}(N)$ corresponds in a natural way to a convex region $R(\sigma) \subseteq R^2$:

1. The root of $\text{BSP}(N)$ corresponds to the whole of R^2 .
2. Inductively, if $l = l(\sigma)$ is the line passing through the segment $S(\sigma)$ labeling σ , then the negative child of σ can be identified with the region $R(\sigma) \cap l_-$ and the positive child can be identified with the region $R(\sigma) \cap l_+$.

The regions in the partition shown in Figure 9.9(a) correspond to the leaves of the BSP tree shown in Figure 9.9(b). The two half-spaces in Figure 9.8 correspond to the nodes labeled + and –.

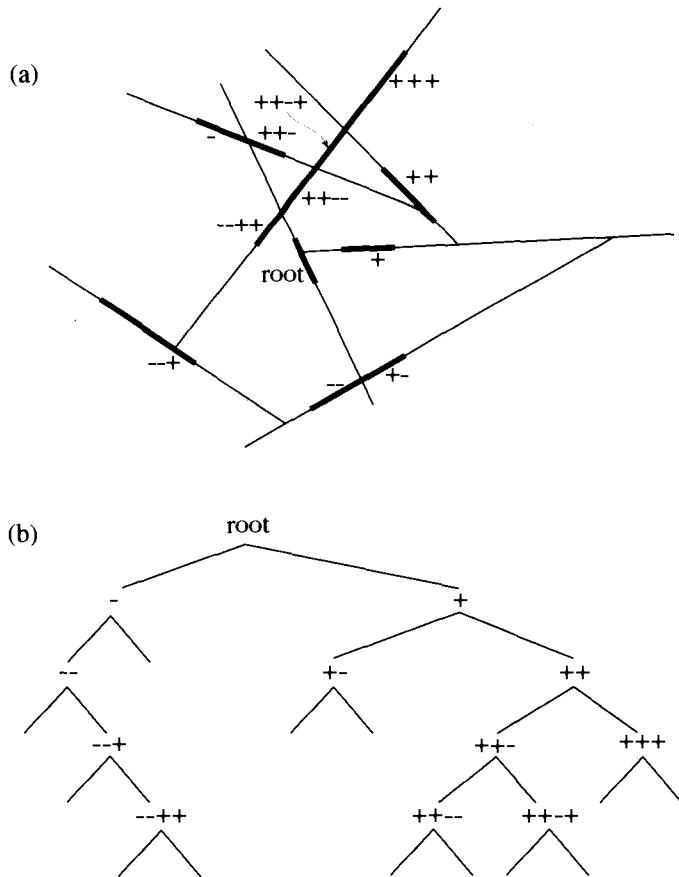


Figure 9.9: (a) A Binary Space Partition. (b) Corresponding BSP tree.

The association of the region $R(\sigma)$ with a node σ in $\text{BSP}(N)$ is purely conceptual. We do not actually associate with σ the description of $R(\sigma)$. This would be costly as well as unnecessary. The convex regions associated with the leaves of $\text{BSP}(N)$ constitute a partition of the plane (Figure 9.9(a)). The key property of this partition is that no segment in N intersects the interior of any of its regions.

Let $\Gamma(N)$ denote the set of fragments labeling the nodes of $\text{BSP}(N)$. Given a view point v , a priority order on $\Gamma(N)$ can be generated by the in-order traversal of the BSP tree. The following algorithm outputs the fragments in $\Gamma(N)$ in the increasing priority order. It is initially called with σ pointing to the root of $\text{BSP}(N)$.

Algorithm 9.2.2 (BSP tree traversal)

1. If $v \in l(\sigma)_+$:
 - (a) Recur on the positive subtree of σ (if it is nonempty).
 - (b) Output $S(\sigma)$.
 - (c) Recur on the negative subtree of σ (if it is nonempty).
2. If $v \in l(\sigma)_-$, do the above three steps in reverse order.

The previous definition of a BSP tree was based on the divide and conquer paradigm. We can also give an equivalent definition that is based on the paradigm of randomized incrementation. We have already seen how a randomized binary tree can be thought of as the history of a randomized incremental algorithm (Section 1.3). One can do exactly the same for $\text{BSP}(N)$ as well. For this, we imagine adding the segments in N , one at a time, in random order. Let N^k denote the set of the first k added segments. The initial $\text{BSP}(N^0)$ consists of just one node that corresponds to the whole plane. In general, a leaf σ of $\text{BSP}(N^k)$ corresponds to a convex region $R(\sigma)$ in the plane. Then, $N(\sigma)$ can be defined as the set of intersections of the segments in $N \setminus N^k$ with $R(\sigma)$. The convex regions that correspond to the leaves of $\text{BSP}(N^k)$ constitute a partition of the plane. The addition of the $(k+1)$ th segment S^{k+1} to $\text{BSP}(N^k)$ (Figure 9.10) is done by splitting the regions intersecting S^{k+1} . The splitting is done along the line $l(S^{k+1})$ passing through S^{k+1} . Accordingly, the node for each region $R(\sigma)$ intersecting S^{k+1} gets two children. The fragments in $N(\sigma)$ are also cut along $l(S^{k+1})$.

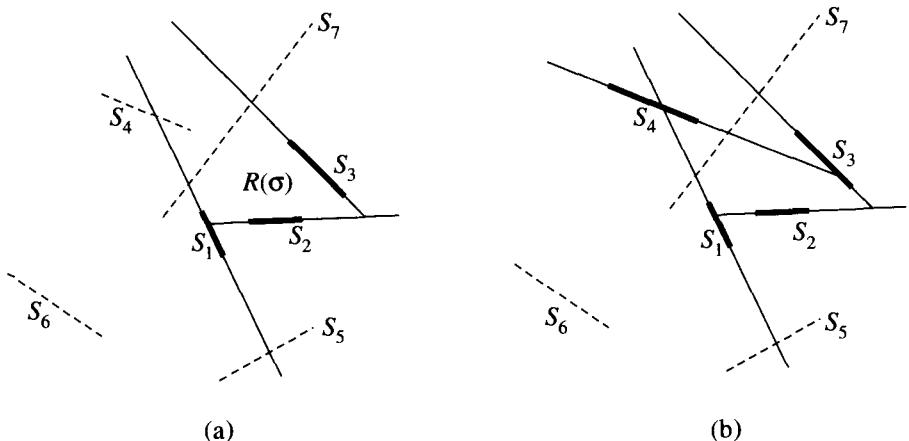
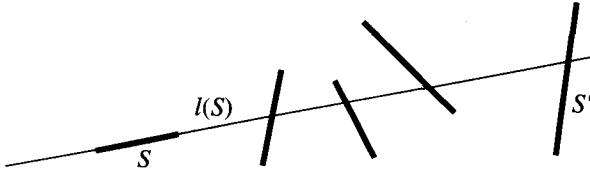


Figure 9.10: Addition of a segment to a BSP tree: (a) $\text{BSP}(N^3)$. (b) $\text{BSP}(N^4)$.



$$i(S, S') = 3$$

Figure 9.11: Index of a pair of segments.

Theorem 9.2.3 Assume that the segments in N are non-intersecting. Then the expected size of $\text{BSP}(N)$ generated by our algorithm is $O(n \log n)$. The expected cost of building $\text{BSP}(N)$ is $O(n^2 \log n)$.

Proof. It suffices to bound the expected size of $\text{BSP}(N)$, because the time spent by the algorithm at each node of $\text{BSP}(N)$ is clearly $O(n)$.

The size of $\text{BSP}(N)$ is equal to the size of $\Gamma(N)$, the set of generated fragments labeling the nodes of $\text{BSP}(N)$. This is n plus the number of “cuts” that take place during the algorithm. Hence, it suffices to estimate the expected number of cuts.

Fix any two segments $S, S' \in N$. Let $l(S)$ denote the line passing through S . Let the index $i(S, S')$ denote the number of segments in N intersecting $l(S)$ between S and S' (Figure 9.11). By convention, $i(S, S') = \infty$, if $l(S)$ does not intersect S' .

We interpret $\text{BSP}(N)$ as the history of a random N -sequence of additions. Let us say that S cuts S' during this sequence if S' , or more precisely its fragment, gets cut along the line $l(S)$ during the addition of S . This can happen only if S is added before S' and the $i(S, S')$ segments between S and S' . (The converse is not true.) The probability of the latter event is $1/[2 + i(S, S')]$. Hence, the expected number of cuts is less than or equal to

$$\sum_S \sum_{S'} \frac{1}{i(S, S') + 2},$$

where S and S' range over all segments in N . The inner sum can be broken in two parts, by letting S' range over the segments intersecting $l(S)$ to the right or to the left of S . Each part is clearly bounded by $\sum_{i=1}^n 1/i = O(\log n)$. It follows that the expected number of cuts, and hence the expected size of $\text{BSP}(N)$, is $O(n \log n)$. \square

We shall end this section with one heuristic that should improve the preceding randomized construction of BSP trees. In the present scheme, the segment $S(\sigma)$ labeling a node $\sigma \in \text{BSP}(N)$ is chosen from the set $N(\sigma)$

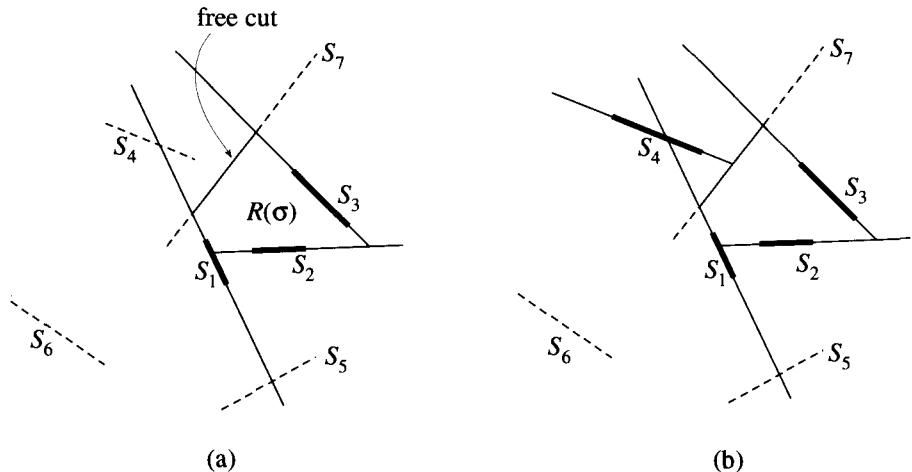


Figure 9.12: Effect of a free cut.

randomly. One can do better if $N(\sigma)$ contains a fragment \bar{S} that spans $R(\sigma)$. By this, we mean that \bar{S} cuts $R(\sigma)$ in two pieces; e.g., see S_7 in Figure 9.10(a). Choosing this fragment as $S(\sigma)$ provides a *free cut* in the sense that the remaining segments in $N(\sigma)$ are not cut by this choice. We only need to partition $N(\sigma) \setminus \{\bar{S}\}$ into two subsets of fragments—one for each half-space bounded by the line through \bar{S} . For example, in Figure 9.10(a) the region $R(\sigma)$ can be cut for free by the fragment of S_7 within it. Hence, the partition in Figure 9.10(a) can be refined further by a free cut, as shown in Figure 9.12(a). The new partition that results after the addition of four segments is shown in Figure 9.12(b). Compare it with the partition in Figure 9.10(b). Notice how the middle fragment of S_7 is not cut further in Figure 9.12(b). Note that this heuristic does not change the order of addition of segments (in the randomized incremental version of the algorithm). The only change is that we exploit free cuts, whenever possible. When a segment is added, we ignore its fragments that have been used in the earlier free cuts.

There is one subtle issue here. How do we know which fragments in $N(\sigma)$ span $R(\sigma)$? This is a valid question because we do not carry the description of $R(\sigma)$. Fortunately, the problem under construction is easy to handle. A fragment $\bar{S} \in N(\sigma)$ spans $R(\sigma)$ iff none of its endpoints coincides with an endpoint of the original segment in N containing \bar{S} . If several fragments in $N(\sigma)$ span $R(\sigma)$, we just choose the one with the lowest order; recall that the order on $N(\sigma)$ is inherited from the initial random order on N . If there is no spanning segment in $N(\sigma)$, we proceed as before.

Let us now summarize our refined construction of a BSP tree. We shall only give a randomized incremental version, leaving the divide-and-conquer version to the reader.

Initially, we put a random order on the set N . We shall add the segments in N in this fixed order. Let S_i denote the i th segment in this order, and let N^i denote the set of the first i segments in the order. The initial $\text{BSP}(N^0)$ consists of one node τ that corresponds to the whole plane. We let $N(\tau) = N$. The addition of $S = S_{i+1}$ to $\text{BSP}(N^i)$ is achieved as follows.

Algorithm 9.2.4 (Addition that exploits free cuts)

1. For each leaf σ of $\text{BSP}(N^i)$ such that $N(\sigma)$ contains a fragment of S , cut $N(\sigma)$ along this fragment. By this, we mean: Cut the fragments in $N(\sigma) \setminus \{S\}$ so as to get two sets $N(\sigma)_+$ and $N(\sigma)_-$. Give σ two children σ_+ and σ_- . Let $N(\sigma_+) = N(\sigma)_+$ and $N(\sigma_-) = N(\sigma)_-$
2. While there is a leaf β , such that $N(\beta)$ contains a spanning fragment:
 - (a) Choose the spanning fragment in $N(\beta)$ with the least order.
 - (b) Cut $N(\beta)$ freely along this fragment.

We leave it to the reader to verify that free cuts can only decrease the size of the BSP tree.

Exercises

9.2.1 Rigorously verify that the free-cut heuristic does not increase the size of $\text{BSP}(N)$.

†9.2.2 Prove or disprove: For any set of n nonintersecting segments in the plane, there exists a BSP tree of $O(n)$ size.

9.2.3 How do the algorithms in this section perform when the segments in N are allowed to intersect?

9.2.2 Dimension three

Let us now turn to the real three-dimensional world of computer graphics. Here N is not a set of segments, but rather a set of polygons in R^3 . All two-dimensional algorithms given before (Algorithms 9.2.1, 9.2.2, and 9.2.4) can be translated to the three-dimensional setting *verbatim*: Segments become polygons and dividing lines become dividing planes. We leave this straightforward translation to the reader. It only remains to see how the resulting algorithms perform in three dimensions. It is easy to see that, for any set N of n polygons, the BSP trees generated by these algorithms have $O(n^3)$ size. Can one say something better about the expected size, if the polygons in N are nonintersecting? For the algorithms given before, this seems rather difficult. But we shall see that a slight variation of the algorithm based on free

cuts (Algorithm 9.2.4) generates a BSP tree of expected $O(n^2)$ size. This bound is worst-case optimal: There exists a set N of polygons such that every BSP tree over N must have $\Omega(n^2)$ size (Exercise 9.2.6).

We shall now describe this three-dimensional variant of Algorithm 9.2.4. Let N be a set of n non-intersecting polygons in R^3 . We assume that each polygon has a constant number of edges. Initially, we put a random order on the set N . We shall add the polygons in N in this order. Let S_i be the i th polygon in this order and let N^i be the set of the first i polygons in this order. We construct $\text{BSP}(N^i)$ by induction on i . Each leaf β of $\text{BSP}(N^i)$ will correspond in a natural way to a convex region $R(\beta) \subseteq R^3$. This correspondence is only conceptual. We do not maintain the description of $R(\beta)$ with β . The convex regions for all leaves of $\text{BSP}(N^i)$ will constitute a convex partition of R^3 . We shall denote this partition by $H(N^i)$. Each node of $\text{BSP}(N^i)$ will be labeled with a fragment of a polygon in N^i . Initially, $H(N^0)$ consists of just one region, namely, the whole of R^3 , and $\text{BSP}(N^0)$ consists of one node τ that corresponds to this region. We let $N(\tau) = N$.

Now consider the addition of $S = S_{i+1}$ to N^i . In the three-dimensional analogue of Algorithm 9.2.4, only the regions of $H(N^i)$ that intersect $S = S_{i+1}$ would be cut. In the following modified algorithm, a region of $H(N^i)$ can be cut by the plane through S_{i+1} even if it does not intersect S_{i+1} . The modified addition of $S = S_{i+1}$ to $\text{BSP}(N^i)$ works as follows.

Algorithm 9.2.5 (Addition)

1. For each leaf σ of $\text{BSP}(N^i)$, do:

If some fragment in $N(\sigma)$ intersects the plane $p(S)$ through S , then label σ with S , cut $N(\sigma)$ along the plane $p(S)$, and give two children to σ .
2. While there is a newly created leaf β such that $N(\beta)$ contains a spanning fragment, do:
 - (a) Choose the spanning fragment in $N(\beta)$ with the least order. Label β with it.
 - (b) Cut $N(\beta)$ along this fragment freely and give two children to β .

By a spanning fragment in $N(\beta)$, we mean a fragment that cuts the convex region $R(\beta)$ into two pieces. This happens iff the fragment is completely contained in the interior of the containing polygon in N . The cut via a spanning fragment is *free* in the sense that no polygon in $N(\beta)$ is actually cut by this choice.

Analysis

What is the expected size of the BSP tree generated by the above algorithm? First, we have to define the size of a BSP tree formally. We shall define it

as the total number of nodes in the tree. We could also have defined the size as the total face-length of the fragments labeling the nodes of $\text{BSP}(N)$. Fortunately, the two definitions differ by at most a constant factor. To see this, fix a polygon $S \in N$. The fragments contained within S constitute a partition of S (Figure 9.13), which gives rise to a planar graph whose vertices have degree ≤ 3 . It immediately follows from the Euler's relation for this planar graph (Exercise 2.3.4) that the total face-length of the fragments within S is proportional to their number. This implies that the preceding two definitions of size differ only by a constant factor. In what follows, by the size of $\text{BSP}(N)$, we shall mean the total number of its nodes.

Let us now bound the expected size of the BSP tree produced by our algorithm, assuming that the polygons in N are non-intersecting. For this, it suffices to bound, for each i , the number of fragments that are cut by $p(S_{i+1})$ in the first step of Algorithm 9.2.5. In other words, for each polygon $Q \in N \setminus N^{i+1}$, we count the increase in the number of its fragments. We do not need to count the free cuts, because a generated fragment can be used for a free cut only once.

So fix a polygon $Q \in N \setminus N^{i+1}$. For any fixed N^{i+1} , we shall estimate the expected number of newly created fragments within Q during the $(i+1)$ th addition, assuming that each polygon in N^{i+1} is equally likely to be S_{i+1} . Notice that the fragments of Q that occur in the lists $N(\beta)$'s associated with the leaves of $\text{BSP}(N^i)$ must be *exterior*, i.e., they must be adjacent to the boundary of Q . This is because an interior fragment of Q would have been immediately used up in a free cut. Figure 9.14(a) illustrates this. It shows the arrangement formed within Q by the intersection of the planes $p(S_j)$, $1 \leq j \leq i$, with Q . The exterior fragments of Q are shown shaded. We are interested in the number of exterior fragments of Q that are cut by S_{i+1} . This number is obviously bounded by the number of the newly created exterior fragments within Q . Consider the set of exterior fragments within Q existing after $i+1$ additions. These correspond to the exterior regions in the arrangement that is formed within Q by the intersections of the planes $p(S_j)$, $1 \leq j \leq i+1$, with Q (Figure 9.14(b)). The newly created exterior regions are precisely the ones adjacent to $p(S_{i+1})$, or, more precisely,

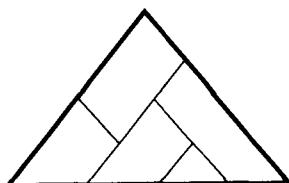


Figure 9.13: Partition of a polygon by its fragments.

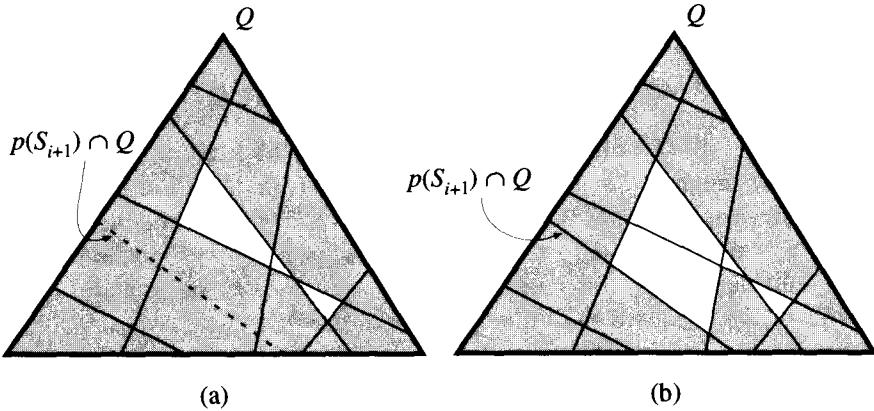


Figure 9.14: Exterior regions within Q (shown shaded): (a) After i additions.
(b) After $i + 1$ additions.

to $p(S_{i+1}) \cap Q$. Let us denote their number by $m(Q, S_{i+1})$. Now we shall analyze the $(i + 1)$ th addition backwards. For a fixed N^{i+1} (not N^i), what is the expected number of exterior fragments in Q that are newly created during the $(i + 1)$ th addition? Because of the random nature of additions, each polygon in N^{i+1} is equally likely to occur as S_{i+1} . Hence, the expected number of newly created exterior regions within Q is

$$\frac{1}{i+1} \sum_{S_{i+1} \in N^{i+1}} p(Q, S_{i+1}). \quad (9.6)$$

The latter sum is just the total face-length of all exterior regions in the arrangement formed within Q by $p(P) \cap Q$, $P \in N^{i+1}$. Each such exterior region belongs to the zone of some line bounding Q . Applying the Zone Theorem for line arrangements to each of the $O(1)$ lines bounding Q , it follows that this total face-length is $O(i + 1)$. Thus, the expression in (9.6) is $O(1)$. In other words, for a fixed $Q \notin N^{i+1}$, the expected number of newly created fragments within Q during the $(i + 1)$ th addition is $O(1)$.

For the $(i + 1)$ th addition, it thus follows that the expected number of newly created fragments within all polygons in $N \setminus N^{i+1}$ is $O(n - i)$. Thus, the expected size of the BSP tree is $O(\sum_i n - i) = O(n^2)$.

What is the expected cost of generating the above BSP tree? First of all, what is the cost of detecting every leaf β of $\text{BSP}(N^i)$, such that a fragment in $N(\beta)$ intersects $p(S_{i+1})$? (Refer to the first step in Algorithm 9.2.5.) We have already seen that, for a fixed $Q \in N \setminus N^i$, the fragments of Q that occur in the lists $N(\beta)$'s are the exterior fragments of Q and their number is $O(i)$ (Figure 9.14(a)). Hence, the total number of fragments that occur in the

lists $N(\beta)$'s associated with the leaves of $\text{BSP}(N^i)$ is $O(i(n - i))$. For a fixed fragment, testing whether it intersects $p(S_{i+1})$ can take time proportional to the number of its vertices if the test is done by brute force. However, we can always maintain a balanced tree with the circular list of vertices of every fragment, so that this test takes only logarithmic time (Exercise 2.8.4). Thus, the total cost of detecting all leaves β 's of $\text{BSP}(N^i)$ such that $p(S_{i+1})$ intersects a fragment in $N(\beta)$ is $O(i(n - i) \log i)$. Over the whole algorithm, this adds up to $O(n^3 \log n)$.

As far as the cost of the rest of the algorithm is concerned, we only need to bound, for each node σ of $\text{BSP}(N)$, the cost of cutting $N(\sigma)$. This is obviously bounded by the total number of edges of all fragments in $N(\sigma)$. But an edge generated by a fixed cut in the first step of Algorithm 9.2.5 can bound a fragment in $N(\sigma)$ for at most n choices of σ . This is because the depth of $\text{BSP}(N)$ is at most n . We have already seen that the expected number of cuts is $O(n^2)$. Hence, the expected total cost incurred in this part of the algorithm is $O(n^3)$.

It thus follows that the expected cost of generating the BSP tree is $O(n^3 \log n)$. To summarize:

Theorem 9.2.6 The expected size of the BSP tree generated by the algorithm is $O(n^2)$, if the polygons in N are non-intersecting. The expected cost of its generation is $O(n^3 \log n)$.

Exercises

***9.2.4** Bring down the expected cost of BSP tree generation to $O(n^3)$.

9.2.5 Generalize the results in this section to arbitrary fixed dimension d . Let N be a set of $(d - 1)$ -dimensional polytopes in R^d . Assume that each polytope has a bounded size. Show that the d -dimensional variant of the algorithm in this section generates a BSP tree of $O(n^{d-1})$ expected size.

9.2.6 Construct a set N of n polygons in R^3 , so that any BSP tree over N must have $\Omega(n^2)$ size.

9.3 Moving viewpoint

In this section, we consider the problem of hidden surface removal with respect to a moving viewpoint. One way of approaching this problem would be to compute the view with respect to every viewpoint from scratch. For example, one can traverse the BSP tree all over for every viewpoint. This will create a priority order among the given polygons, or, more precisely, their fragments, with respect to the given viewpoint. After this, the polygons can be painted onto the device screen in the decreasing priority order. One disadvantage of this method is that all fragments of the polygons in N , even the

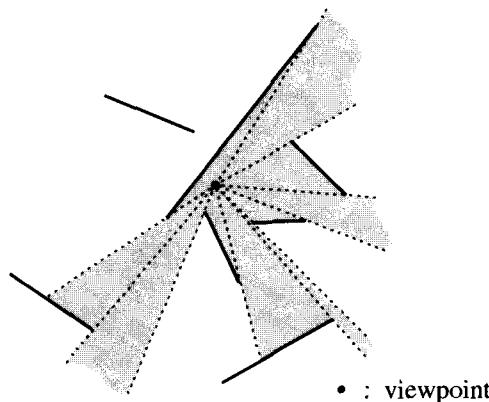


Figure 9.15: A view zone.

ones that are completely invisible, need to be painted for every viewpoint. When the response time is critical, this may become too expensive.

We shall now indicate another method for hidden surface removal, which does not have this drawback. For this purpose, it will be convenient to consider the following alternative formulation of the hidden surface removal problem. Imagine, for each point in the view window, a ray from the viewpoint that passes through this point and extends until it hits the first polygon in the scene; if there is no such polygon, the ray extends to infinity. The collection of these rays defines the *view zone* with respect to the given viewpoint (Figure 9.15). In other words, the view zone is just the set of all visible points in R^3 . In Figure 9.15, we are assuming, for the sake of simplicity, that the viewer can look in all directions. Hidden surface removal is essentially equivalent to the computation of the view zone. We shall now give a method for preprocessing the set N and building a data structure so that, given any viewpoint v , the view zone with respect to v can be computed quickly.

Before we tackle the real three-dimensional problem, let us give the analogue of the method in the toy world of two-dimensional computer graphics. Here N becomes a set of segments in the plane. In the preprocessing phase, we shall build a trapezoidal decomposition $H(N)$ formed by N , and also a point location structure for $H(N)$. This can be done using the randomized incremental algorithm in Section 3.1. In what follows, the vertical attachments in $H(N)$ are considered transparent. We shall also refer to them as transparent walls. In contrast, the segments in N are considered opaque. Given any view point v , the view zone with respect to v can be computed as follows. We first locate v in $H(N)$ in logarithmic time, with high probability, using the point location structure associated with $H(N)$. Then, we begin with the initial approximation to the view zone. This is defined to be

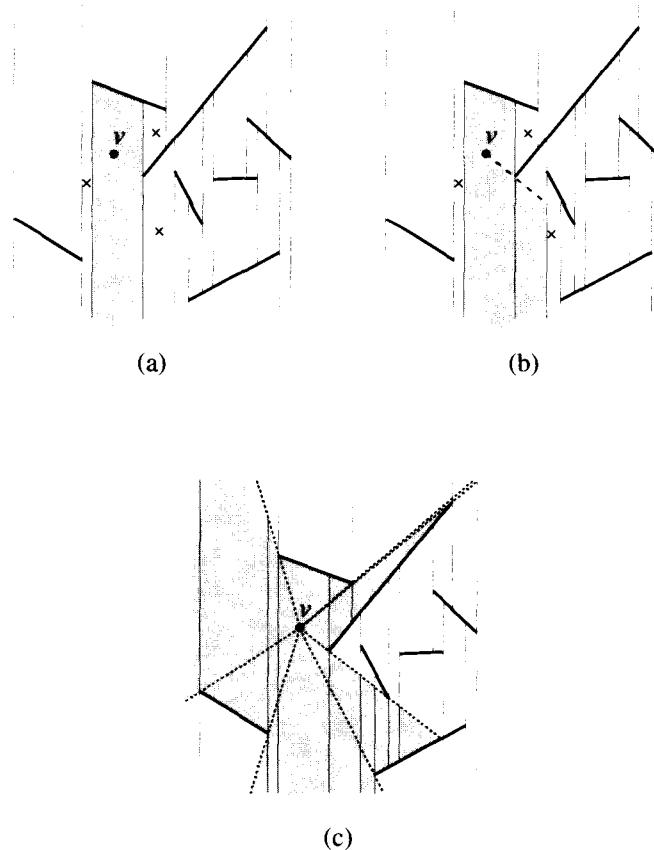


Figure 9.16: View expansion: (a) Initial approximation to the view zone. (b) The zone Z after one expansion step. (c) Final view zone.

the restriction of the view zone to the cell (trapezoid) in $H(N)$ containing the viewpoint (Figure 9.16(a)). After this, we shall “expand” this approximate view zone step by step until it coincides with the complete view zone (Figure 9.16). Let us denote this expanding zone by Z . We shall maintain throughout the algorithm an appropriate representation of the boundary of Z . We shall also maintain a list Φ of the cells in $H(N)$ whose transparent walls meet the boundary of Z from outside. In Figure 9.16 we are assuming, for the sake of simplicity, that the viewer can look in all directions. The zone Z is shown shaded. The cells in Φ at each step are shown marked with crosses. The view zone expansion algorithm, in essence, is the following.

Algorithm 9.3.1 (View zone expansion)

While Φ is non-empty, remove any cell R from Φ , and do:

1. Imagine “expanding” the zone Z into R (Figure 9.16); that is, imagine extending into R the rays from the viewpoint that pass through the points that are shared by the boundary of Z and the transparent walls of R . Update the boundary of Z accordingly.
2. Add to Φ those cells adjacent to R whose transparent walls meet the new boundary of Z from outside.

Clearly Z coincides with the complete view zone, when the algorithm terminates. It is easy to see that the total running time of the algorithm is proportional to the total size of the partition obtained by intersecting $H(N)$ with the view zone with respect to v . In practice, one may expect this size to be “proportional” to the size of the view zone, although no such theoretical guarantee can be given.

To generalize the preceding hidden surface removal method to three dimensions, one needs a suitable three-dimensional generalization of a trapezoidal decomposition. We shall now present one such generalization called a *cylindrical decomposition*. A cylindrical decomposition can be used for the computation of a view zone just as we used a trapezoidal decomposition in the two-dimensional setting. We should remark that cylindrical decompositions have other uses too. For example, we can use them for answering range queries over a set of polygons in R^3 , just as we can use trapezoidal decompositions for answering range queries over a set of segments in the plane (Section 2.3).

So let us get back to the real world of 3D graphics. Now the set N consists of polygons in R^3 . For the sake of simplicity, assume that the polygons are nonintersecting. Intersecting polygons are treated in the exercises. For the sake of simplicity, we shall also assume that the polygons have bounded face-lengths. Our goal is to preprocess N and build a certain cylindrical decomposition $H(N)$, which can then be used for computing the view zone with respect to any viewpoint quickly.

The decomposition $H(N)$ is defined as follows. When the elements in N were segments, the trapezoidal decomposition was formed by passing vertical attachments through their endpoints. Analogously, we shall now raise certain vertical walls, i.e., walls parallel to the z -axis, through the edges of the polygons in N . The choice of the z -axis here is purely arbitrary. Fix an edge e of a polygon in N . Consider a vertical segment through any point t on e that extends upwards (and downwards) in the z -direction until it hits the first polygon; if no such polygon exists, then it extends to infinity. The union of such vertical segments through all points on e defines a vertical wall, which we shall denote by $\text{wall}(e)$. Such a vertical wall is called a *primary*

wall. In Figure 9.17(a), we have shown a top view (along the z -axis) of a set N of polygons. Figure 9.17(b) shows the intersections of the polygons in N with the imaginary vertical plane $P(b)$ through the edge b parallel to the z -axis. The vertical wall through b is shown shaded.

When vertical walls are passed through all polygon edges, we get a certain decomposition $H'(N)$ of R^3 . Each region of this decomposition is a “cylinder” parallel to the z -axis. To see this, consider any fixed polygon $B \in N$. Consider the restriction of $H'(N)$ on a fixed side of B . Figure 9.17(c)–(d) gives an example. It shows the intersections of the primary walls in $H'(N)$ with the upper and lower sides of the polygon B . Note that the intersections of the primary walls with the lower side of B are not visible on its upper side and vice versa. This is because the polygons in N are treated as opaque. The intersections of the primary walls with the upper and lower sides of B decompose these sides into several regions. Each such region bounds a unique cylinder in $H'(N)$, either as its top or as its bottom.

Note that the top and bottom of a cylinder in $H'(N)$ are generally not parallel. In fact, they can even touch each other if sharing of edges is allowed. Notice also that the cylinders in $H'(N)$ can have complicated shapes. Hence, we shall subdivide them further as follows. First, we decompose the top (or, equivalently the bottom) of each cylinder into trapezoids by passing attachments through its corners parallel to the yz -plane (Figure 9.17(e)–(f)). This decomposition of the top or bottom of a cylinder can be vertically extended so as to divide the whole cylinder into “parallelepipeds” (i.e., cylinders whose tops and bottoms are trapezoids or triangles; we are misusing the terminology a bit, because the tops and bottoms of these cylinders are usually not parallel). The vertical walls that subdivide the cylinders in $H'(N)$ will be called *secondary* walls.

The partition $H(N)$ that we get after decomposing all 3-cells in $H'(N)$ is the cylindrical decomposition that we sought. It partitions R^3 into several vertical cylinders whose tops and bottoms are trapezoids or triangles. These cylinders will also be called the *cells* of $H(N)$. In what follows, the vertical walls of the cylinders in $H(N)$ will be considered transparent, whereas the polygons in N will be considered opaque. The restriction of $H(N)$ to the upper or lower side of any polygon $B \in N$ is a trapezoidal decomposition. These trapezoidal decompositions will be denoted by $H^u(N, B)$ and $H^l(N, B)$, respectively (Figure 9.17(e)–(f)).

A compact way to represent $H(N)$ in computer memory is as follows. For each polygon $B \in N$, we store the trapezoidal decompositions $H^u(N, B)$ and $H^l(N, B)$. Each trapezoid Δ in these decompositions is paired with a unique other trapezoid that bounds the same cylinder in $H(N)$ as Δ . This *pairing information* implicitly represents all cylinders in $H(N)$. It is not necessary

to explicitly represent the faces of $H(N)$ that are contained in the primary or secondary walls. The adjacencies in $H(N)$ are defined in the obvious fashion. One only needs to remember that all polygons N are considered

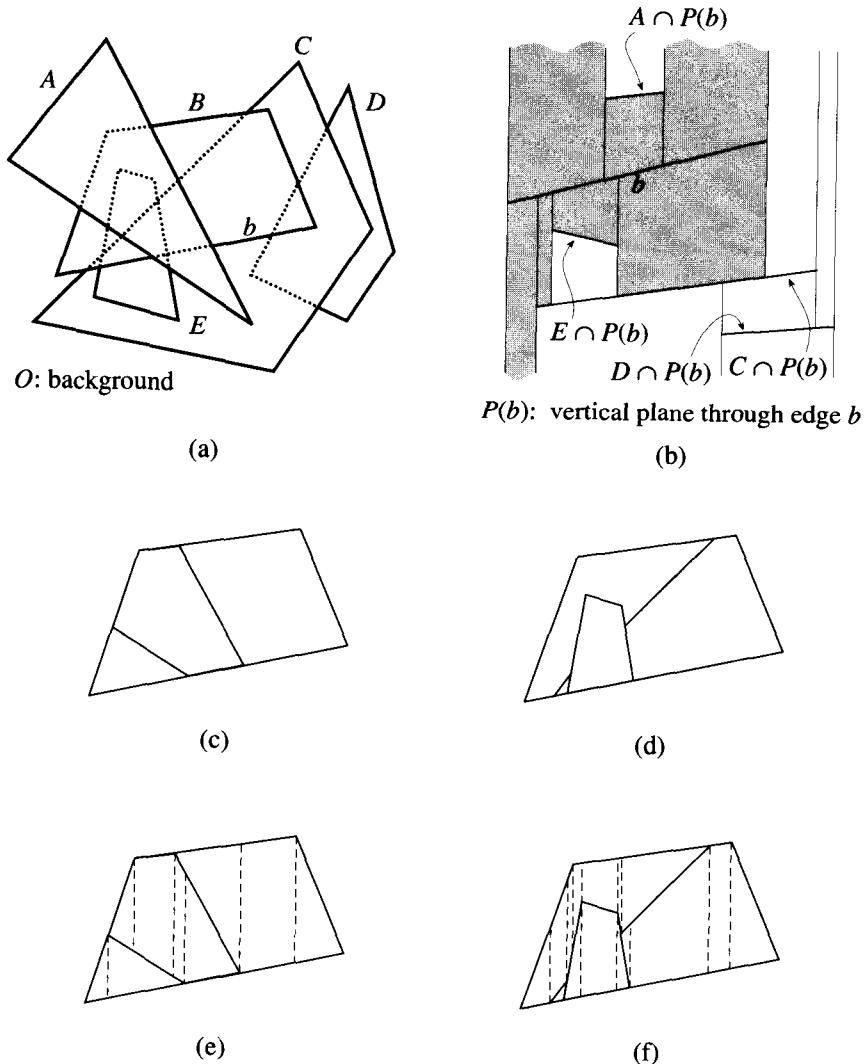


Figure 9.17: (a) A set N of polygons. (b) A primary wall through edge b (shown shaded). (c) Intersections of the primary walls with the upper side of B , and (d) the lower side of B . (e) $H^u(N, B)$: the restriction of $H(N)$ to the upper side of B . (f) $H^l(N, B)$: the restriction of $H(N)$ to the lower side of B .

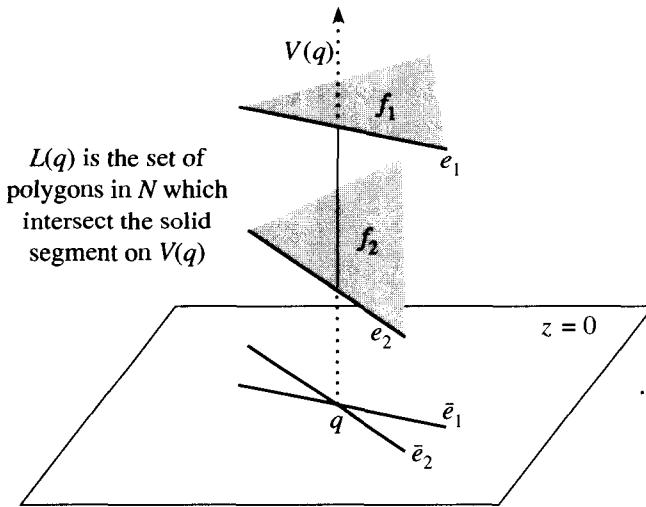


Figure 9.18: Level of a junction.

opaque for the purpose of this definition. For example, the cylinder in $H(N)$ that is adjacent to a trapezoid on the upper side of a polygon $B \in N$ is not considered adjacent to any trapezoids that occur on the lower side of B . In this representation, $H(N)$ is basically stored as a collection of several trapezoidal decompositions, with some extra pairing information. This makes the representation very simple conceptually.

The size of $H(N)$ is $O(n^2)$. To see this, it suffices to bound the size of the preliminary partition $H'(N)$, because the secondary walls can increase the size by at most a constant factor. But the size of $H'(N)$ is clearly $O(n^2)$, because the size of each primary wall is $O(n)$.

The quadratic bound on the size of $H(N)$ is overly pessimistic. To see this, let us try to bound the size of $H(N)$ in a different way. Without loss of generality, we can assume that all polygons lie above the $z = 0$ plane, changing the z -coordinate if necessary. In what follows, the z -direction will be assigned a special role because the cells of $H(N)$ are parallel to the z -axis. Project the edges of all polygons in N orthogonally onto the plane $\{z = 0\}$. This gives rise to several junctions among the projected edges. Let $q := \bar{e}_1 \cap \bar{e}_2$ be any such junction, where \bar{e}_1 and \bar{e}_2 denote projections of the polygon edges e_1 and e_2 , respectively (Figure 9.18). Let f_1 and f_2 be the polygons in N containing e_1 and e_2 . (When q is the projection of a vertex of a polygon in N , $f_1 = f_2$.) We say that f_1 and f_2 define q . Let $D(q)$ be the set of these polygons defining q . We define the degree $d(q)$ of q to be the size of $D(q)$. It is two, when q is a crossing. It is one, when q is the projection of

a vertex of a polygon in N . Let $V(q)$ be the vertical line through q parallel to the z -axis. Let $L(q)$ be the set of polygons in N that intersect $V(q)$ between e_1 and e_2 , i.e., in the part of $V(q)$ shown solid in Figure 9.18. We say that these polygons *obstruct* or *conflict* with q . Let $\text{level}(q)$, the level (or the conflict size) of q , be defined as the size of $L(q)$. As a convention, we define the level of a junction q to be zero if it corresponds to the projection of a vertex of a polygon in N . The reader should compare the definition of level in this section with that in Section 9.1 (compare Figures 9.18 and 9.7): In Section 9.1, $L(q)$ also contained the polygons that intersect $V(q)$ below e_1 and e_2 .

Observe that the size of the preliminary partition $H'(N)$ is essentially the number of junctions with level 0, ignoring a constant factor. This is because the size of the primary wall through any polygon edge e is proportional to the number of junctions with level 0 lying on the projection \bar{e} of e . (Why?) It follows that the size of $H(N)$ is also proportional to the number of junctions with level 0. In practice, the number of such junctions is subquadratic and sometimes almost linear in n .

Cylindrical decompositions can be used for the computation of a view zone, just as we used trapezoidal decompositions in the two-dimensional setting. Indeed, the basic view expansion idea in Algorithm 9.3.1 can be extended to three dimensions *verbatim*. The resulting algorithm computes the view zone with respect to a given viewpoint in time proportional to the size of the partition obtained by intersecting the cylindrical decomposition $H(N)$ with the view zone. We leave the verification of this fact to the reader. One can also use the cylindrical decomposition for answering range queries, where the range is a connected region in R^3 and one is asked to report the polygons in N intersecting the range. This can be done by first locating a point of the range in $H(N)$ using the associated point location structure and then traveling along the range in $H(N)$. This is very similar to the way one can use trapezoidal decompositions for answering range queries in the plane (Section 2.3). So we do not elaborate it any further.

Heuristics: One can augment the method for hidden surface removal based on view expansion with several heuristics in practice. For example, instead of using the cylindrical decomposition of the whole of R^3 , one can suitably divide R^3 into buckets, in a grid-like fashion, and then decompose each bucket cylindrically.

The hidden surface removal algorithm given in this section is an object space method. The principal objection against object space methods is usually that they are numerically imprecise and unstable. Fortunately, this is not a sensitive issue for the view expansion method, because it is very simple, and because the correctness of the output view zone is not all that crucial. If

the numerical precision is high enough in comparison with the resolution of the visual device, the view computed by this method should look very similar to the absolutely correct view. The stability becomes a crucial issue when one is required to guarantee that the output be consistent in some sense. For example, this is very crucial when one computes the decomposition $H(N)$. In that case, one is required to guarantee that the computed adjacency information is consistent. Otherwise, the view expansion algorithm, which works on the basis of this information, can err. On the other hand, the computation of $H(N)$ is only a preprocessing step. Since this step is infrequent, one can hope to guarantee such consistency through more time and extra care.

Exercise

9.3.1 Fill in the details of the three-dimensional view expansion algorithm. Can one use Binary Space Partition for view expansion? If so, give an algorithm based on BSP, and compare it with the one based on cylindrical decompositions. (Hint: The regions in a BSP need not have bounded descriptions. The worst-case $O(n^3)$ size of a BSP is higher than the worst-case size of a cylindrical decomposition, which is proportional to the number of junctions with level 0. In practice too, the size of a BSP is likely to be higher than the size of a cylindrical decomposition. Why?)

9.3.1 Construction of a cylindrical partition

Given a set N of non-intersecting polygons, how fast can one construct $H(N)$?

One trivial method to construct $H(N)$ is the following. For each edge b of a polygon in N , we compute the primary wall through b directly. For this, we intersect the polygons in N with the vertical plane through b and retain only those intersections that intersect the vertical strip through b (Figure 9.17(b)). We can compute the trapezoidal decomposition formed by these intersections in expected $O(n \log n)$ time using the randomized incremental algorithm in Section 3.1. Once this is done, we only retain those trapezoids in the decomposition that are adjacent to b . Their union is precisely the primary wall through b . We compute all primary walls in this fashion in $O(n^2 \log n)$ expected time overall.

Now consider any fixed polygon $B \in N$ and its fixed side—say, the upper side. At this stage, we know the intersections of all primary walls with this side of B . We compute the trapezoidal decomposition of B formed by these intersections, once again using the randomized incremental algorithm. This takes expected time proportional to the size of the trapezoidal decomposition, ignoring a logarithmic factor. The resulting trapezoidal decomposition corresponds to $H^u(N, B)$, the restriction of $H(N)$ to the upper side of B (Figure 9.17(e)). In this fashion, we compute the restrictions of $H(N)$ to

both sides of all polygons in N . This takes time (expected) proportional to their total size, or, in other words, the size of $H(N)$, ignoring a logarithmic factor. Once we know the restrictions of $H(N)$ to the upper and lower sides of all polygons in N , and all primary walls, we basically know the whole decomposition $H(N)$. The rest is a routine affair, and we do not elaborate it any further.

To summarize:

Proposition 9.3.2 $H(N)$ can be built in expected $O(n^2 \log n)$ time.

This result still leaves a lot to be desired, because the actual size of $H(N)$ can be subquadratic. The bottleneck in the above procedure is the computation of the primary walls. Once this is done, the remaining phase takes (expected) time proportional to the output size, up to a log factor. With suitable heuristics, it is actually possible to modify the computation of the primary walls, so that, in practice, the algorithm takes much less than $O(n^2 \log n)$ expected time.

Nevertheless, it is desirable to have an alternative method with a better performance guarantee for several reasons. First, the method given above is purely static. We want a simple method that can also be dynamized. This is essential when the set N can be changed by the user in an on-line fashion by adding or deleting a polygon. Second, the preceding method does not yield a point location structure for $H(N)$. Such a point location structure is crucial because the method for view zone computation begins by locating the viewpoint in $H(N)$.

9.3.2 Randomized incremental construction

With this in mind, we now give another randomized incremental method for constructing $H(N)$. This method can be easily dynamized through the use of history. Moreover, the history can also be used for fast point location.

The randomized incremental method performs better than the previous method, in a sense that it is quasi-output-sensitive like the algorithm in Section 9.1. What we mean by this is the following. Project the edges of all polygons in N onto the coordinate plane $z = 0$. Each arising junction q is assigned a level as before (Figure 9.18). Let

$$\Theta(N, s) = \sum_q \frac{1}{[\text{level}(q) + d(q)]^{d(q)-s}}, \text{ for } s = 0, 1,$$

where q ranges over all junctions in the coordinate plane. In other words, the contribution of a junction is inversely proportional to a power of its level. We can also reformulate the preceding definition in another illuminating way.

Let $m_l(N)$ denote the number of crossings at level l . Then

$$\Theta(N, s) = O(n) + \sum_l \frac{m_l(N)}{(l+2)^{2-s}}. \quad (9.7)$$

The first term here arises due to the contribution of the junctions with degree 1; these are projections of the vertices of the polygons in N and their number is $O(n)$. Observe that $m_l(0)$, the number of crossings with level 0, is essentially the size of $H(N)$, ignoring an additive $O(n)$ term. The expected running time of the randomized incremental algorithm in this section will turn out to be $O([\Theta(N, 0) + n] \log n)$. (In Section 9.1, the bound involved analogous $\Theta(N, 1)$. In that sense the algorithm here is better.) Since the first term in the above expansion of $\Theta(N, 0)$ is $m_0(N)$, the algorithm can be considered quasi-output-sensitive in the static setting. The on-line version of the algorithm can be considered close to optimal. Indeed, by a verbatim translation of the argument in Section 9.1.1, it follows that the expected structural change in the underlying cylindrical decomposition over a random N -sequence of additions is trivially $\Omega(\Theta(N, 0))$. Hence, the expected running time of any on-line algorithm for maintaining cylindrical decompositions is also $\Omega(\Theta(N, 0))$.

Let us now turn to the randomized incremental algorithm for constructing $H(N)$. It will be a natural generalization of the trapezoidal decomposition algorithm in Section 3.1. This is only to be expected, because cylindrical decompositions generalize trapezoidal decompositions to three dimensions. We shall add the polygons in N , one at a time, in random order. Let N^i denote the set of the first i added polygons. At the i -stage, we shall maintain the partition $H(N^i)$. Moreover, for each cell $\Delta \in H(N^i)$, we shall maintain a conflict list of the input vertices that lie within Δ . By input vertices, we mean the vertices of the polygons in N . These conflict lists are analogous to the conflict lists of the trapezoids in the trapezoidal decomposition algorithm (Section 3.1). Strictly speaking, it is sufficient to maintain conflict information for just one arbitrarily distinguished vertex of every polygon in N .

Addition of the $(i+1)$ th polygon $S = S_{i+1}$ to $H(N^i)$ is achieved as follows. The conflict information tells us the cell $\Delta \in H(N^i)$ containing the distinguished vertex of S . Starting at Δ , we can determine all cells in $H(N^i)$ intersecting S by a simple search:

Algorithm 9.3.3 (Search)

Initialization: Mark Δ as “visited”. Let $\Phi = \{\Delta\}$.

While Φ is non-empty, remove any cell, say $\bar{\Delta}$, in Φ , and do:

- Determine all cells in $H(N^i)$ that are adjacent to the vertical walls of Δ ; their number is clearly $O(1)$. Determine which ones of these adjacent

cells intersect S . Mark the intersecting cells that are not already marked and insert them in Φ . (Note that S cannot intersect the cells that are adjacent to the top or the bottom of $\bar{\Delta}$, but not to its vertical walls. This is because the polygons in N are non-intersecting.)

Because all cells of $H(N^i)$ intersecting S lie contiguously, it is clear that this search will determine all of them in time proportional to their number. All these marked cells will eventually be removed.

Now, consider the set of all polygons in N that are adjacent to the marked cells. Fix a polygon P in this set. Let us say that the upper side of P is adjacent to the marked cells; the other possibility can be handled in a symmetric fashion. Consider the restriction $H^u(N^i, P)$ on the upper side of P . Imagine a viewer looking in the positive z -direction from below P , with the view window coinciding with P . In that case, $H^u(N^i, P)$ is precisely the view seen by this imaginary viewer, when the scene consists of the polygons in N^i above P . Similarly, $H^u(N^{i+1}, P)$ is the view seen by this imaginary viewer after the addition of S . We already know the trapezoids in the imaginary view $H^u(N^i, P)$ that are in conflict with S —these are precisely the trapezoids in $H^u(N^i, P)$ that are adjacent to the marked cells. Hence, proceeding as in the hidden surface removal algorithm in Section 9.1, we can update $H^u(N^i, P)$ to $H^u(N^{i+1}, P)$ in time proportional to the structural change, ignoring a logarithmic factor.

In this fashion, we update $H^u(N^i, P)$, or $H^l(N^i, P)$ as the case may be, for each polygon that is adjacent to the marked cells in $H(N^i)$.

We also need to form the trapezoidal decompositions $H^u(N^{i+1}, S)$ and $H^l(N^{i+1}, S)$ on the upper and lower sides of the newly added polygon S . For example, let us see how to construct the trapezoidal decomposition on the lower side of S , the other case being symmetric. We vertically project the newly formed trapezoids in $H^u(N^{i+1}, P)$ onto the lower side of S . Here P ranges over all polygons in N^i that were adjacent to the bottoms of the marked cells in $H(N^i)$. We ignore the new trapezoids that project outside S . Then, $H^l(N^{i+1}, S)$ is precisely the decomposition that is formed by the projected trapezoids on the lower side of S . The adjacencies among these projected trapezoids are easy to figure out, as one can readily check.

Finally, we also need to pair each newly created trapezoid Δ with another trapezoid that bounds the same cylinder in $H(N^{i+1})$ as Δ . This procedure is rather routine, and we shall not elaborate it.

The preceding discussion shows that $H(N^i)$ can be updated to $H(N^{i+1})$ in time proportional to the structural change, ignoring a logarithmic factor. It remains to update the conflict lists. Let us say that an input vertex of a polygon in $N \setminus N^i$ conflicts with a trapezoid in $H^u(N^i, P)$ (or $H^l(N^i, P)$), if it conflicts with the cell in $H(N^i)$ adjacent to that trapezoid. In this way, the

conflict list of a cylinder adjacent to a trapezoid can also be thought of as the conflict list of that trapezoid. When we update $H^u(N^i, P)$ to $H^u(N^{i+1}, P)$, we also update these conflicts within the trapezoids just as we updated the vertex–trapezoid conflicts in Section 9.1; especially, see the discussion before Lemma 9.1.2. Thus, it follows that the conflict lists for the trapezoids in $H^u(N^i, P)$ (or $H^l(N^i, P)$) can be updated in time proportional to the total conflict change, ignoring a logarithmic factor. By the total conflict change, we mean the total conflict size of the destroyed trapezoids in $H^u(N^i, P)$ plus the total conflict size of the newly created trapezoids in $H^u(N^{i+1}, P)$. We apply this procedure to all polygons in N^i that are adjacent to the bottoms or tops of the marked cells in $H(N^i)$. It follows that the total cost of conflict update is proportional to the total conflict change, ignoring a logarithmic factor.

Analysis

The preceding discussion shows that the cost of adding S to $H(N^i)$ is proportional, ignoring a logarithmic factor, to $\sum_{\Delta} 1 + l(\Delta)$, where Δ ranges over all destroyed and newly created cells, and $l(\Delta)$ denotes its conflict size. Since a created cell can be destroyed only once, the amortized cost of adding S to $H(N^i)$ can be taken to be proportional to

$$\sum_{\Delta} 1 + l(\Delta), \quad (9.8)$$

where Δ now ranges over only the newly created cells in $H(N^{i+1})$.

We shall now estimate the expected running time of the whole algorithm using backward analysis. For this, we shall estimate the expected cost of the $(i+1)$ th addition, conditional on a fixed N^{i+1} . With this in mind, let us sum the quantity in (9.8) by letting S range over all polygons in N^{i+1} . The resulting sum ϕ is proportional to $n - i$ plus the size of $H(N^{i+1})$. This is because the total conflict size of all cells in $H(N^{i+1})$ is $O(n - i - 1)$, and each cell in $H(N^{i+1})$ contributes to ϕ iff S is one of the $O(1)$ polygons in N^{i+1} defining it. Since each polygon in N is equally likely to occur as $S = S_{i+1}$, it follows that the expected cost of the $(i+1)$ th addition, conditional on a fixed N^{i+1} , is proportional to

$$\frac{n - i}{i} + \frac{|H(N^{i+1})|}{i + 1}, \quad (9.9)$$

where $|H(N^{i+1})|$ denotes the size of $H(N^{i+1})$. The first term here sums to $O(n \log n)$, over all i . So we only need to worry about the second term. Let $\Pi = \Pi(N)$ be the set of junctions obtained by projecting all edges of the polygons in N onto the plane $z = 0$ (Figure 9.18). With each such junction

$q = \bar{e}_1 \cap \bar{e}_2$, we associate, as before, a stopper set $L(q)$ of the polygons that obstruct q , and the trigger set $D(q)$ consisting of the polygon(s) in N containing the edges e_1 and e_2 . In this fashion, Π can be thought of as a configuration space. The size of the subspace $\Pi(N^{i+1})$ is proportional to the number of junctions in Π active over N^{i+1} . These active junctions are the ones whose triggers are contained in N^{i+1} but none of whose stoppers is contained in N^{i+1} . Since N^{i+1} is itself a random sample of N of size $i + 1$, it follows from (9.9) that the expected running time of the algorithm is proportional to

$$\sum_i \frac{n-i}{i} + \frac{e(i)}{i} \approx O(n \log n) + \sum_i \frac{e(i)}{i},$$

where $e(i)$ is the expected number of junctions in Π that are active over a random sample of N of size i . The second term in the above expression can also be interpreted as the expected structural change over the configuration space Π during a random N -sequence of additions (Theorem 3.4.5, Exercise 3.4.1). But this expected structural change is easy to calculate directly. Indeed, a junction $q \in \Pi$ becomes active after i additions with probability $1/\binom{l(q)+d(q)}{d(q)}$; we have already reasoned in a similar fashion before (see (9.1)). Thus, it follows that the expected structural change under consideration is proportional to

$$\sum_{q \in \Pi} \frac{1}{\binom{l(q)+d(q)}{d(q)}} \approx \Theta(N, 0).$$

To summarize:

Theorem 9.3.4 *The expected running time of the randomized incremental algorithm is $O(n \log n + \Theta(N, 0) \log n)$. In contrast, the expected running time of any on-line algorithm for maintaining cylindrical decompositions is trivially $\Omega(\Theta(N, 0) + n)$ over a random N -sequence of additions.*

Exercises

9.3.2 Make the algorithm in this section on-line making use of history. Specify the link structures carefully. Show how the resulting history can also be used for point location. The cost of point location should be polylogarithmic with high probability. Show that the expected running time of the on-line version over a random N -sequence of additions is $O(n \text{polylog } n + \Theta(N, 0) \log n)$.

****9.3.3** Make the preceding on-line algorithm fully dynamic using dynamic shuffling. Show that the expected running time of this algorithm over a random (N, δ) -sequence is $O(n \text{polylog } n + \Theta(N, 0) \log^2 n)$, if the signature δ is weakly monotonic.

Give a dynamic algorithm for the same purpose based on dynamic sampling.

***9.3.4** Generalize the results in this section to sets of intersecting polygons. Generalize the randomized incremental algorithm to such sets and show that Theorem 9.3.4 holds for the generalized algorithm, with appropriate new interpretation.

Bibliographic notes

For an excellent survey of computer graphics, see [102]. Here we shall confine ourselves to randomized methods.

A randomized incremental algorithm for hidden surface removal (Section 9.1) was given by Mulmuley [161], and it was made fully dynamic (Exercises 9.1.1 and 9.1.3) in [168]; intersecting polygons (Exercise 9.1.2) and algebraic patches are handled in [161] (journal version). An important question in the static setting is the design of an output-sensitive hidden surface removal algorithm. Randomization does not seem to help here. However, significant progress has been made using deterministic methods by several authors, e.g., Bern [23], Reif and Sen [192], Overmars and Sharir [177], de Berg and Overmars [77], de Berg et al. [76], and Agarwal and Matoušek [3].

The use of Binary Space Partitions in hidden surface removal has been investigated by several people, e.g., Fuchs, Kedem, and Naylor [108], and Fuchs, Abram, and Grant [107]. The randomized BSP trees (Section 9.2.1 and 9.2.2) are due to Paterson and Yao [183]. A worst-case (unpublished) example requiring a BSP tree of $\Omega(n^2)$ size (Exercise 9.2.6) has been given by Eppstein; see [183].

Three-dimensional cylindrical decompositions were used by Clarkson et al. [65] in the context of incidence counting. Mulmuley [164] investigated its use in computer graphics (Section 9.3), and gave randomized incremental (Section 9.3.2) and dynamic algorithms (Exercises 9.3.2 and 9.3.3) for its construction. For a generalization of cylindrical decomposition to higher dimensions and its use in robotics, see Chazelle et al. [44].

Chapter 10

How crucial is randomness?

So far, we have been assuming that we are given a random source that can provide us with any number of completely independent random bits. In practice, the situation is not so ideal. Mainly, this has got to do with the way random number generators work in practice. Typically, they take a few numbers, called *seeds*, as input. These seeds are supposed to be, or rather hoped to be, completely random. After this, they output a long sequence of numbers by performing a succession of arithmetic operations on the seeds. One example is provided by the so-called *linear congruential generator*. It takes two random seeds a and b in the range 0 to $n - 1$ as input; here n is a large enough prime number. A sequence of “random” numbers is then generated by evaluating the expression $ax + b$, modulo n , where x ranges over all integers from 0 to $n - 1$. As one would expect, the generated numbers are not truly random—we shall see later that they are only pairwise independent. For this reason, it is more appropriate to call them *pseudo-random* numbers. The linear congruential generator is just one example of a pseudo-random generator. The pseudo-random generators that are used in practice vary a lot.

It would naturally be of great interest to know how the randomized algorithms given in this book perform when the “random” bits or numbers that are used by them are only pseudo-random. We shall show that most of the randomized incremental algorithms that we have described perform quite well in a pseudo-random setting. Roughly, all they need is that the “random” numbers provided to them be t -wise independent, where t is a large enough constant depending on the problem under consideration. Such pseudo-random sequences can be generated, in fact, using only $O(\log n)$ truly random bits, where n is the input size. In other words, the amount of randomness that is required to guarantee good performance by these algorithms is really quite small.

That leads us to another question: Can one get rid of randomness altogether? We shall see that this is quite often possible for the algorithms based on top-down random sampling. (For randomized incremental algorithms too, this is quite often possible, but it is a bit more difficult.) The technique that is used for this purpose is called *derandomization*. It basically consists in simulating a randomized algorithm in a deterministic fashion. One must point out that this simulation can be much more complex than the original algorithm. In that regard, the situation here is somewhat different from the pseudo-random setting, wherein the original algorithm is essentially kept as it is, and only the source of randomness is changed. We are certainly not claiming that derandomized algorithms are practical. But still they have important theoretical implications. First, existence of a deterministic algorithm for a given problem is always an important theoretical concern. Second, derandomization clarifies the trade-off between randomness and determinism—randomness lets you achieve simplicity, but in the process, you lose deterministic behavior. Third, there is always a hope that a derandomized, deterministic algorithm for a given problem may be followed by a direct and simpler deterministic algorithm.

10.1 Pseudo-random sources

In this section, we shall show that the expected running times of most of the randomized incremental and on-line algorithms discussed in this book do not change by more than a constant factor when the random bits that are used by these algorithms are not truly random, but only “pseudo-random.” This does not entail any change in the algorithm. Only the source of randomness is different.

Recall that randomized incremental algorithms work by adding the objects in the set N under consideration, one at a time, in random order (Chapter 3). At each stage, the algorithm maintains various data structures, such as the facial lattice of the geometric complex being constructed, history, and so forth. In a pseudo-random version of the algorithm, the N -sequence of additions will not be truly random, but rather “pseudo-random”. It takes $\Omega(n \log n)$ random bits to generate a truly random N -sequence of additions, where n is the size of N . In contrast, a pseudo-random N -sequence will be generated using only $O(\log n)$ truly random bits. The rest of the algorithm remains the same as before.

To generate a pseudo-random sequence of additions, we shall need a sequence X_0, X_1, \dots, X_{n-1} of random variables that is *t-wise independent*, for some constant t . By this, we mean that every t -tuple of distinct variables in this sequence is independent, but the knowledge of any t variables in the

sequence may determine all the others. In our study, each random variable X_i will take a value from the set $[n] = \{0, \dots, n - 1\}$. We assume in what follows that n is prime. Otherwise, we only need to pad N with dummy objects so that its size equals a prime number not much larger than n .

For every fixed constant t , we shall now show how to generate a t -wise independent sequence of random variables X_i using only a few random bits. It will turn out that $t \lceil \log n \rceil$ random bits suffice.¹ One, but not the only, way of generating such a sequence is based upon a simple generalization of the linear congruential generator. The idea is to generate a random polynomial $f(x)$ of a fixed degree by choosing its coefficients at random from the set $[n]$. A sequence of random variables is then generated by letting x take all values from 0 to $n - 1$ and evaluating $f(x)$ modulo n . This pseudo-random generator will be called a Polynomial Congruential Generator (PCG). When the degree of the generated polynomial is one, it reduces to the linear congruential generator described before.

Let us now describe the preceding pseudo-random generator in more detail. We choose t numbers, also called *seeds*, a_0, \dots, a_{t-1} independently from a uniform random distribution on $[n]$. This requires $\lceil \log_2 n \rceil$ random bits per seed. Assume, as before, that n is prime. For each $i \in [n]$, let

$$X_i = a_0 + a_1 i + a_2 i^2 + \cdots + a_{t-1} i^{t-1} \pmod{n}.$$

Proposition 10.1.1 For each i , X_i is uniformly distributed on $[n]$ and the sequence X_0, \dots, X_{n-1} is t -wise independent. Fix any $z \in [n]$. Conditional on $X_i = z$, each X_j , $j \neq i$, is still uniformly distributed on $[n]$ and the sequence X_0, \dots, X_{n-1} is $(t - 1)$ -wise independent.

Proof. Fix any t distinct indices $i_1, \dots, i_t \in [n]$. It suffices to show that, for every such choice, the t -dimensional random variable $Z = (X_{i_1}, \dots, X_{i_t})$ is uniformly distributed. Since the t seeds are chosen from the uniform distribution on $[n]$, it suffices to show that each t -tuple value from $[n]^t$ is assumed by Z for a unique choice of the seeds a_0, \dots, a_{t-1} . Now notice that the t -tuple Z is related to the t -tuple $A = (a_0, \dots, a_{t-1})$ by the linear relation $Z = VA \pmod{n}$, where V is a Vandermonde matrix. The j th row of V is $(1, i_j, \dots, i_j^{t-1})$. Since n is prime, the set $[n]$ is a field. The claim follows because a Vandermonde matrix is invertible over every field. \square

Assume that the set N of objects is given to us in the form of an array of length n , where each entry contains the description of an object. For $0 \leq i \leq n - 1$, we let $N[i]$ denote the object contained in the i th entry of this array. We shall now give an algorithm to generate a pseudo-random N -sequence of additions, i.e., a sequence of additions of the objects in N .

¹All logarithms, in what follows, are to base two.

First, we construct a t -wise independent sequence X_0, \dots, X_{n-1} of numbers using the preceding scheme. This needs $t\lceil \log n \rceil$ random bits. The constant t will depend on the problem under consideration. After this, we proceed as follows:

Algorithm 10.1.2 (Generation of a pseudo-random N -sequence)

Main phase

For $i = 0$ to $n - 1$, do:

Add $N(X_i)$, if it is not already added.

Clean-up phase

It is possible that at the end of the main phase of additions, some objects in N are still not added. This is because the n variables X_i need not take distinct values. In the clean-up phase, we add the remaining objects in N in *any* order.

In what follows, by a pseudo-random sequence, we shall always mean a sequence generated by Algorithm 10.1.2, using a large enough (constant) number of seeds. The exact number of seeds will depend on the problem under consideration.

It turns out that the expected running times of the most of the randomized incremental and on-line algorithms in this book do not change by more than a constant factor when one uses a pseudo-random N -sequence instead of a truly random N -sequence of additions. As we have seen on several occasions, the running times of randomized incremental algorithms can be estimated in terms of the expected structural and conflict change over the underlying configuration spaces. We shall now examine this expected change in more detail. We will show that its value has the same order, ignoring a constant factor, in the random as well as pseudo-random setting.

So let $\Pi = \Pi(N)$ be any configuration space. Let n be the size of N . Let \bar{u} be any N -sequence of additions. Let N^i denote the first i added objects in this sequence. Let σ be a fixed configuration in Π . We say that σ is active at time i during \bar{u} if it is active over the set N^i . This means N^i contains all triggers but no stopper associated with σ . We say that σ is *created* during \bar{u} , if it is active at some time $i \leq n$. If i is the least such integer, then we say that σ is created during \bar{u} at time i . We say that σ is killed (or destroyed) at time j , if σ is active over N^{j-1} but not over N^j . A created configuration during \bar{u} can be destroyed only once, because \bar{u} contains only additions.

Proposition 10.1.3 *If the sequence \bar{u} of additions is a truly random permutation of N , then the probability that σ is created during \bar{u} is $1/\binom{\ell(\sigma)+d(\sigma)}{d(\sigma)}$.*

Proof. Observe that σ is created during \bar{u} iff the first $d(\sigma)$ elements that are added from the set $L(\sigma) \cup D(\sigma)$ are all in $D(\sigma)$. Because \bar{u} is truly random,

each subset of $d(\sigma)$ elements in $L(\sigma) \cup D(\sigma)$ is equally likely to be added during \bar{u} before the remaining elements in $L(\sigma) \cup D(\sigma)$. \square

What happens if \bar{u} is a pseudo-random sequence of additions, as generated by Algorithm 10.1.2? It turns out that the following analogue of the above proposition still holds.

Lemma 10.1.4 (Main Lemma) *Let \bar{u} be a pseudo-random sequence of additions generated by Algorithm 10.1.2 with t seeds. If $t \geq 3d(\sigma) + 2$, the probability that σ is created during \bar{u} is $O(1/(l(\sigma)+d(\sigma)))$. The constant factor within the Big-Oh notation depends on $d(\sigma)$.*

We defer the proof of this lemma to the end of this section. We remark that the actual nature of pseudo-random generation is important only in the proof of this lemma. In turn, the lemma uses only the property in Proposition 10.1.1. If the pseudo-random generator that one wishes to use is different from PCG, then one needs to prove the Main Lemma or Proposition 10.1.1 for that pseudo-random generator. The rest of the treatment in this section does not depend on the nature of the pseudo-random generator. Hence, it would remain valid as it is.

Once the Main Lemma is proved, the rest is a straightforward affair. Let $\Pi = \Pi(N)$ be any configuration space. Let \bar{u} be any N -sequence of additions. Recall that the total *structural change* during \bar{u} is defined to be the number of configurations in Π that are created during \bar{u} . More precisely, for each configuration $\sigma \in \Pi$, let us define a 0–1 indicator function as follows: $i(\sigma, \bar{u})=1$, if σ is created during \bar{u} ; it is zero otherwise. Then the total structural change during \bar{u} is

$$\sum_{\sigma \in \Pi} i(\sigma, \bar{u}). \quad (10.1)$$

This also bounds the number of configurations that are destroyed during \bar{u} , because a created configuration can be destroyed only once.

More generally, given $a \geq 0$, we define the a th order conflict change during \bar{u} as

$$\sum_{\sigma \in \Pi} [d(\sigma) + l(\sigma)]^a i(\sigma, \bar{u}). \quad (10.2)$$

This also bounds the sum of the a th powers of the conflict sizes of the destroyed configurations, because each created configuration can be destroyed only once. The 0th order conflict change is simply the structural change as defined before. The first order conflict change is the ordinary conflict change.

Let $d = d_{\max}(\Pi) = \max\{d(\sigma) \mid \sigma \in \Pi\}$. We assume in what follows that d is bounded by a constant. Let \bar{u} be either a truly random sequence or a pseudo-random sequence of additions that is generated by Algorithm 10.1.2

using at least $3d + 2$ seeds. Then it follows from (10.2), Proposition 10.1.3, and the Main Lemma that the expected a th order conflict change during \bar{u} is bounded, within a constant factor, by

$$\sum_{\sigma \in \Pi} \frac{[d(\sigma) + l(\sigma)]^a}{\binom{l(\sigma) + d(\sigma)}{d(\sigma)}} \approx \sum_{\sigma \in \Pi} \frac{1}{[l(\sigma) + d(\sigma)]^{d(\sigma)-a}}. \quad (10.3)$$

(Here, as well as in what follows, \approx denotes asymptotic equality with constant factors ignored.) By (10.2) and Proposition 10.1.3, this is also a lower bound if the sequence \bar{u} is truly random. For notational convenience, let us denote the series on the right hand side of (10.3) by $\Theta(\Pi, a)$, where it is assumed that $0 \leq a \leq d_{\min}(\Pi) = \min\{d(\sigma) | \sigma \in \Pi\}$. (We have already encountered similar quantities in our investigation of convex polytopes (Section 7.4), hidden surface removal (Section 9.1.1) and cylindrical decompositions (Section 9.3.2).)

Our discussion so far can be summarized as follows: Let $\Pi = \Pi(N)$ be any configuration space. Assume that $d = d_{\max}(\Pi)$ is bounded by a constant. Let \bar{u} be a truly random sequence of additions or a pseudo-random sequence of additions for which the Main Lemma holds. In particular, this holds if one uses Algorithm 10.1.2 using at least $3d + 2$ seeds. Then, for each fixed $a \geq 0$, the expected value of the a th order conflict change over Π is $O(\Theta(\Pi, a))$. If \bar{u} is a truly random sequence of additions, then this expected value is also $\Omega(\Theta(\Pi, a))$.

In particular, it follows that:

Theorem 10.1.5 *For any such configuration space Π and any fixed $a \geq 0$, the expected value of the a th order conflict change over a pseudo-random sequence generated as above is bounded, within a constant factor, by the expected value over a truly random sequence.*

Let us now apply this theorem to a few randomized incremental algorithms. The remaining randomized incremental algorithms in the book will be treated in the exercises. In each example, we assume that the number of seeds used is a large enough constant. It has to be greater than or equal to $3d_{\max}(\Pi) + 2$, where Π is the configuration space under consideration.

Example 10.1.6 (Trivial configuration space)

Suppose $\Pi(N)$ is a trivial configuration space (Example 3.4.6). This means the number of active configurations over any $R \subseteq N$ is $O(1)$. In that case, we have already seen that the expected structural change over a truly random N -sequence of additions is $O(\log n)$ (Example 3.4.6). It follows from Theorem 10.1.5 that the expected structural change remains $O(\log n)$ if the N -sequence is generated in a pseudo-random fashion using enough seeds. The reader may recall that trivial configurations spaces arise while estimating the length of a search path through history (Example 3.4.6). It follows from the

above result that, in such situations, the expected length of the search path remains logarithmic even in the pseudo-random setting. As a very simple application, let N be a set of n points on the line. It is now easy to see that the expected running time of the pseudo-random version of quick-sort, or more generally, its online version (Section 1.3), which builds a randomized binary tree as the history, is $O(n \log n)$. This follows because, for each fixed point in N , the expected length of the search path in this tree is $O(\log n)$. Hence, the expected size of the tree is $O(n \log n)$. We have already seen that the running time of quick-sort is of the same order as the size of its history (Section 1.3).

Example 10.1.7 (Trapezoidal decompositions)

Let N be a set of segments in the plane. In Section 3.1, we gave a randomized incremental algorithm for constructing the trapezoidal decomposition formed by N . Its expected running time was $O(k + n \log n)$, where k is the number of intersections among the given segments. We shall show that the expected running time remains of the same order, even if the sequence of additions is pseudo-random. This is assuming that the number of seeds is large enough.

We have already seen that the total running time of this algorithm, ignoring conflict maintenance, is of the same order as the total number racquets (Figure 3.10) that are destroyed or created in the algorithm (Example 3.4.4). Let Π be the configuration space of feasible racquets (Example 3.4.4). Applying Theorem 10.1.5 to Π , it follows that the expected number of racquets that are created or destroyed during a pseudo-random sequence is bounded, within a constant factor, by the expected value over a truly random sequence. In other words, the expected running time of the trapezoidal decomposition algorithm, ignoring conflict maintenance, respects the same Big-Oh bound in the pseudo-random setting as in the truly random setting. This is assuming that the number of seeds is large enough. Since $d_{\max}(\Pi)$ is six, twenty seeds would suffice.

It still remains to take into account the cost of maintaining the conflict lists. Recall that the conflict list associated with any trapezoid in the decomposition formed by N^i consists of the unadded endpoints that lie within the trapezoid; by the unadded endpoints, we mean the endpoints of the segments in $N \setminus N^i$. We saw that the cost of conflict update during an addition is proportional to the total conflict change during that addition. We can amortize this cost by charging $O(1)$ cost to each endpoint that occurs in the conflict list of a destroyed trapezoid. Thus, for a fixed endpoint v of a segment in N , the total cost charged to v in the whole algorithm is proportional to the total number of trapezoids containing v that got destroyed in the course of the algorithm. Let Σ_v be the trivial configuration space of feasible trapezoids over N containing v (Example 3.4.6). By the above ob-

servation, the total cost charged to v is proportional to the total structural change over Σ_v . Its expected value remains $O(\log n)$ in the pseudo-random setting (Example 10.1.6). Thus, it follows that the expected running time of the pseudo-random version of the trapezoidal decomposition algorithm is also $O(k + n \log n)$.

Recall that the incremental trapezoidal decomposition algorithm can be converted into an on-line form by making use of history (Section 3.1.1). It follows just as before that the expected performance of this on-line algorithm too does not change by more than a constant factor in the pseudo-random setting. The only additional result that needs to be proved is that, for a fixed query point, the expected length of the search path in the history remains logarithmic. But we have already seen that the length of the search path for a query point p is proportional to the structural change over the trivial configuration space Σ_p (Example 3.4.6). Hence, its expected value remains logarithmic (Example 10.1.6).

Example 10.1.8 (Convex polytopes)

Let N be a set of half-spaces in R^d . Let $H(N)$ be the convex polytope formed by intersecting these half-spaces. Let $A(N)$ be the arrangement formed by the hyperplanes bounding the half-spaces in N . Let $\Pi(N)$ be the configuration space of the vertices in this arrangement (Example 3.4.2).

We have already seen that the running times of the incremental and on-line algorithms for constructing the convex polytope $H(N)$ are governed by the structural and (or) conflict change over the configuration space $\Pi(N)$ (Section 7.4). Applying Theorem 10.1.5 to $\Pi(N)$, it follows that the expected structural and conflict changes have the same order of magnitude in the random as well as the pseudo-random setting. Hence, the bounds on the expected running times of the convex polytope algorithms, which we proved in the truly random setting, remain valid in the pseudo-random setting. This bound was $O(n^{\lfloor d/2 \rfloor})$, for $n > 3$, and $O(n \log n)$, for $d = 2, 3$. We also proved tighter bounds in terms of the Θ -series associated with the arrangement $A(N)$. These bounds remain valid, too.

Example 10.1.9 (Hidden surface removal)

Let N be a set of polygons in R^3 . Let $\Pi(N)$ be the configuration space of feasible, labeled trapezoids over N . By a feasible, labeled trapezoid, we mean a labeled trapezoid that occurs in the visibility partition over some subset $R \subseteq N$. We saw that the running time of the incremental (and also on-line) hidden surface removal algorithm was governed by the total conflict change over $\Pi(N)$ (Section 9.1.1). Applying Theorem 10.1.5 to $\Pi(N)$, it follows that the bounds for the expected running time proved in Section 9.1.1 remain valid in the pseudo-random setting as well.

Proof of the Main Lemma

Let $\Pi(N)$ be the given configuration space. Let σ be a fixed configuration in $\Pi(N)$. Let $D = D(\sigma)$ and $L = L(\sigma)$ denote the disjoint trigger and stopper sets of σ . Let $d = |D|$ be the degree of σ and let $l = |L|$ be the conflict size of σ . We assume that d is bounded by a constant. We also assume, in what follows, that $l > 0$; otherwise, the main lemma follows trivially.

Let Z_1, \dots, Z_d be the elements of D in any order. Let X_0, X_1, \dots, X_{n-1} be the t -wise independent sequence of random variables generated by our pseudo-random generator PCG. Here each X_i is uniformly distributed over $[n]$.

Lemma 10.1.10 *Assume that the sequence X_0, \dots, X_{n-1} is t -wise independent, where $t > d$. Then, for any $0 < i_1 < i_2 \dots < i_d \leq i \leq n$, the probability that*

$$N[X_j] \notin L, \text{ for each } j \leq i,$$

conditional on

$$N[X_{i_1}] = Z_1, N[X_{i_2}] = Z_2, \dots, N[X_{i_d}] = Z_d,$$

$$\text{is } O\left(\left[\frac{n}{l(i-d)}\right]^{\lfloor \frac{t-d}{2} \rfloor}\right).$$

Proof. The idea is to construct a suitable sequence of random variables, and then obtain a tail estimate for their sum by Chebychev's technique (Appendix A.2).

For each $j \notin \{i_1, i_2, \dots, i_d\}$, let us define a conditional random variable Y_j as follows: Y_j is equal to X_j , conditional on

$$N[X_{i_1}] = Z_1, N[X_{i_2}] = Z_2, \dots, N[X_{i_d}] = Z_d.$$

By Proposition 10.1.1, the sequence of random variables Y_j is $(t - d)$ -wise independent. Moreover, each Y_j is uniformly distributed on $[n]$.

For each $j \notin \{i_1, i_2, \dots, i_d\}$, let us define the indicator function I_j by

$$\begin{aligned} I_j &= 1, && \text{if } N(Y_j) \in L, \\ &= 0, && \text{otherwise.} \end{aligned}$$

It follows that the sequence of random variables I_j is also $(t - d)$ -wise independent. Because Y_j is uniformly distributed on $[n]$, it follows that

$$p = \text{prob}\{I_j = 1\} = l/n, \text{ for each } j.$$

Let $I = \sum_{j \leq i} I_j$, where $j \notin \{i_1, i_2, \dots, i_d\}$. It follows that the expected value of I is $\mu = (i - d)l/n$.

Observe that $N[X_j] \notin L$, for any j , iff I is zero. Hence, the proof can be completed by applying Theorem A.2.1 in the appendix. \square

Corollary 10.1.11 *If X_0, \dots, X_{n-1} is a t -wise independent sequence, then, for any subset $L \subseteq N$, with $l = |L| > 0$, the probability that*

$$N[X_j] \notin L, \text{ for each } j \leq n,$$

is $O(1/l^{\lfloor t/2 \rfloor})$.

Proof. Put $i = n$ in Lemma 10.1.10, and let D be the empty set. \square

Now fix L and D as before, where $d = |D|$ is assumed to be a constant.

Corollary 10.1.12 *Under the same conditions as in Lemma 10.1.10, the probability that*

$$N[X_j] \notin L, \text{ for each } j \leq i, \text{ and } N[X_{i_1}] = Z_1, N[X_{i_2}] = Z_2, \dots, N[X_{i_d}] = Z_d$$

is

$$O\left(\frac{1}{n^d} \left[\frac{n}{l(i-d)}\right]^{\lfloor \frac{t-d}{2} \rfloor}\right), \text{ if } i-d > n/l, \text{ and } O\left(\frac{1}{n^d}\right), \text{ otherwise.}$$

Proof. The probability that

$$N[X_{i_1}] = Z_1, N[X_{i_2}] = Z_2, \dots, N[X_{i_d}] = Z_d \quad (10.4)$$

is $1/n^d$ because the random variables X_j are uniformly distributed and t -wise independent, with $t > d$.

By Lemma 10.1.10, the probability that $N[X_j] \notin L$, for each $j \leq i$, under the condition in (10.4), is

$$O\left(\left[\frac{n}{l(i-d)}\right]^{\lfloor \frac{t-d}{2} \rfloor}\right), \text{ if } i-d > n/l, \text{ and } \leq 1, \text{ otherwise (trivially).}$$

The corollary follows. \square

Now we turn to the proof of the Main Lemma. Let σ be the configuration as in that lemma. Let D and L be its trigger and stopper sets, with $l = |L|$ and $d = |D|$. Let \bar{u} be a pseudo-random sequence of additions generated by Algorithm 10.1.2, using at least $3d + 2$ seeds. We only need to estimate the probability that σ gets activated during the main phase of additions in \bar{u} . This is because σ can become active in the clean-up phase only if no stopper

in L is added during the main phase of additions. By Corollary 10.1.11, with $t = 3d + 2$, the probability of the latter event is $O(1/l^{\lfloor t/2 \rfloor}) = O(1/l^d)$.

So we only need to worry about the main phase of additions. For $l = 0$ or $d = 0$, Main Lemma holds trivially. For $d, l > 0$, it follows from the following result, because the maximum degree d is a constant.

Lemma 10.1.13 *Assume that the sequence X_0, \dots, X_{n-1} of random variables is $(3d+2)$ -wise independent. Then, the probability that the first d elements chosen from the set $L \cup D$ all belong to D is $O(1/l^d)$, for $d, l > 0$.*

Proof. Label the elements in D as Z_1, \dots, Z_d in any fashion. The number of different ways of labeling is clearly $d! = O(1)$, because d is a constant. We shall show that, for each fixed labeling, the probability that, for each $k \leq d$, the k th element chosen from $L \cup D$ is Z_k , is $O(1/l^d)$.

Fix the labeling Z_1, \dots, Z_d of the elements in D . For any $0 < i_1 < i_2 \dots < i_d = i$, let $\text{prob}(i_1, \dots, i_d)$ denote the probability that

$$N[X_j] \notin L, \text{ for each } j \leq i, \text{ and } N[X_{i_1}] = Z_1, \dots, N[X_{i_d}] = N[X_i] = Z_d.$$

Applying Corollary 10.1.12, with $t = 3d+2$, we conclude that $\text{prob}(i_1, \dots, i_d)$ is

$$O\left(\frac{1}{n^d} \left[\frac{n}{l(i-d)}\right]^{d+1}\right), \text{ if } i > n/l + d, \text{ and } O\left(\frac{1}{n^d}\right), \text{ otherwise.}$$

The probability that the first d elements chosen from $L \cup D$ are Z_1, \dots, Z_d in that order is

$$\sum_{0 < i_1 < i_2 < \dots < i_d} \text{prob}(i_1, \dots, i_d),$$

which is, therefore, bounded, within a constant factor, by

$$\sum_{\substack{0 < i_1 < i_2 < \dots < i_d = i : \\ i \leq n/l+d}} \frac{1}{n^d} + \sum_{\substack{0 < i_1 < i_2 < \dots < i_d = i : \\ i > n/l+d}} \frac{n}{l^{d+1}(i-d)^{d+1}}.$$

The first sum contains $\binom{n/l+d}{d}$ terms, and hence, is bounded by

$$\binom{\frac{n}{l} + d}{d} \frac{1}{n^d} = O\left(\frac{1}{l^d}\right).$$

The second term is also $O(1/l^d)$, because, after changing the variable i to $i+d$, it is seen to be bounded by

$$\sum_{n/l < i} \binom{i+d-1}{d-1} \frac{n}{l^{d+1} i^{d+1}} = \frac{n}{l^{d+1}} O\left(\sum_{n/l < i} \frac{1}{i^2}\right) = \frac{n}{l^{d+1}} O\left(\frac{l}{n}\right) = O\left(\frac{1}{l^d}\right).$$

This uses that $\binom{i+d-1}{d-1} = O(i^{d-1})$ and that $\sum_{i>x} 1/i^2 = O(1/x)$. □

Exercises

10.1.1 (Linear programming)

- (a) Show that the expected running time of the randomized incremental algorithm for linear programming (Section 7.1) is $O(d^{O(d)} n \log^{d-1} n)$, when it is pseudo-randomized using PCG with $O(d^2)$ seeds. (Use a different set of $O(d)$ seeds for each level of recursion.) Can you decrease the number of seeds further?
 †(b) Can you show that the expected running time remains linear in n , even in the pseudo-random setting?

10.1.2 Show that the bound on the expected running time of the cylindrical decomposition algorithm given in Section 9.3.2 remains valid in the pseudo-random setting if the number of seeds is a large enough constant. Calculate a precise lower bound on the number of required seeds.

***10.1.3** Let $\Gamma = \Gamma(N)$ be any configuration space. Let $m(\Gamma, r)$ denote the expected number of configurations in Γ that are active over a truly random subset of N of size r . Let $d_{\min}(\Gamma)$ denote the minimum degree of a configuration in Γ . If $0 \leq s < d_{\min}(\Gamma)$, show that

$$\sum_{r=s+1}^n m(\Gamma, r) \frac{[n]_s}{[r]_{s+1}} \approx \Theta(\Gamma, s).$$

Here $[x]_d$ denotes the generalized factorial $x(x-1)\cdots(x-d+1)$. If $s = d_{\min}(\Gamma)$, show that

$$\sum_{r=s+1}^n m(\Gamma, r) \frac{[n]_s}{[r]_{s+1}} \approx \Theta(\Gamma', s) + |\Gamma''| \log n.$$

Here $\Gamma'' \subseteq \Gamma$ is the subspace of configurations of degree $d_{\min}(\Gamma)$ and $|\Gamma''|$ denotes its cardinality. And $\Gamma' \subseteq \Gamma$ is the subspace of configurations of degree greater than $d_{\min}(\Gamma)$. (Hint: Handle each configuration in Γ separately, and then sum over all configurations. You will need to use the Vandermonde identity for binomial coefficients.)

10.1.4 Let Π and Σ be any two configuration spaces over the same set of objects N . For any subset $R \subseteq N$, define $n(\Sigma, R)$ to be the number of configurations in Σ that are active over R . We define $n(\Pi, R)$ similarly. We define the *index* $\Sigma : \Pi$ as

$$\max\{n(\Sigma, R)/n(\Pi, R) | R \subseteq N\}.$$

Suppose the index $\Sigma : \Pi$ is bounded by a constant. Let n be the number of elements in N . For any fixed s , show the following:

- (a) $\Theta(\Sigma, s) = O(\Theta(\Pi, s))$, if s is less than $d_{\min}(\Sigma)$ and $d_{\min}(\Pi)$.
- (b) $\Theta(\Sigma, s) = O(\Theta(\Pi', s) + |\Pi''| \log n)$, if $s = d_{\min}(\Pi) < d_{\min}(\Sigma)$.

Here $\Pi' \subseteq \Pi$ is the subspace of configurations of degree greater than s , $\Pi'' \subseteq \Pi$ is the subspace of configurations of degree equal to s , and $|\Pi''|$ denotes the cardinality of Π'' . (Hint: Apply Exercise 10.1.3 to both Σ and Π .)

10.1.5 The result in Exercise 10.1.4 can be used to analyze randomized incremental algorithms in an alternative fashion. For example, consider the trapezoidal decomposition algorithm. Let $\Pi(N)$ be the configuration space of feasible racquets, as in Example 10.1.7. We saw there that the expected running time of the trapezoidal decomposition algorithm, ignoring conflict maintenance, is proportional to the expected structural change over Π . As in the proof of Theorem 10.1.5, this is $O(\Theta(\Pi, 0))$. Let $\Sigma = \Sigma(N)$ be the configuration space that consists of all endpoints and intersections of the segments in N . If v is an endpoint of a segment $S \in N$, we let the trigger set $D(v) = \{S\}$. If v is the intersection of two segments $S, S' \in N$, we let the trigger set $D(v) = \{S, S'\}$. The stopper set $L(v)$ is defined to be empty, for every v . It is easy to see that the index $\Pi : \Sigma$ is $O(1)$. This follows because the total number of racquets in the trapezoidal decomposition $H(R)$, $R \subseteq N$, is proportional to the size of R plus the total number of intersections among the segments in R . Let k denote the number of intersections among the segments in N . Since $d_{\min}(\Pi) = 1$, Exercise 10.1.4 implies that

$$\begin{aligned}\Theta(\Pi, a) &= O(\Theta(\Sigma, a)) = O(k + n), && \text{if } a < 1, \\ &= O(k + n \log n), && \text{if } a = 1.\end{aligned}\tag{10.5}$$

In particular, $\Theta(\Pi, 0)$ is $O(k + n)$. Hence, the expected running time of the trapezoidal decomposition algorithm, ignoring conflict maintenance, is $O(k + n)$. Including conflict maintenance, this is $O(k + n \log n)$ as before.

Alternatively, prove the same $O(k + n \log n)$ bound by applying Exercise 10.1.3 to Π directly.

In a similar fashion, analyze other randomized incremental algorithms in the book.

10.1.6 Choose your favorite pseudo-random generator other than PCG. See if you can prove the Main Lemma for this generator. As we have already noticed, the other results would then carry forth unchanged.

10.2 Derandomization

In this section, we shall give a certain derandomization technique that can be used to convert several algorithms based on top-down random sampling into deterministic ones. All uses of random sampling in this book eventually use the random sampling result for configuration spaces of bounded dimension (Theorem 5.7.3). In most cases, the special case of this result for range spaces of bounded dimension (Theorem 5.7.4) suffices. This latter result implies that any range space $\Pi(N)$ of bounded dimension has a $(1/r)$ -net of $O(r \log r)$ size. Moreover, such a net can be found in a randomized fashion trivially: For a large enough constant a , a random sample $R \subseteq N$ of $ar \log r$ size is a $(1/r)$ -net with high probability. Here we shall give a deterministic algorithm for the same problem. It is reasonably fast, if r is small enough:

Theorem 10.2.1 Suppose $\Pi(N)$ is a range space of bounded dimension $d \geq 1$. Let n be the size of N . Fix a parameter $r > 1$. Then, a $(1/r)$ -net for $\Pi(N)$ of $O(r \log r)$ size can be found deterministically in $O(n[r^2 \log r]^d)$ time, assuming that $\Pi(N)$ satisfies a certain enumerability assumption.

The enumerability assumption is that, for any subset $S \subseteq N$, we can enumerate all ranges in $\Pi(S)$ in $O(s \pi(S))$ time; here s is the size of S and $\pi(S)$ is the size of $\Pi(S)$. The multiplicative factor s takes into account the fact that the stopper set of any range in $\Pi(S)$ has at most s objects. Thus, $O(s \pi(S))$ is a trivial upper bound on the description size of all ranges in $\Pi(S)$. The enumerability assumption is generally easy to satisfy in the geometric applications that we shall be interested in. For example, consider the case when $\Pi(N)$ is the space of linear ranges induced by a set N of hyperplanes in R^d (Example 5.7.8). Given any subset $S \subseteq N$, we can enumerate all ranges in $\Pi(S)$ as follows. First, we build the arrangement formed by S . This can be done incrementally in $O(s^d)$ time (Section 6.1). After this, for every pair (Δ, Σ) of (open) d -cells in the arrangement, we output the linear range formed by the hyperplanes in S separating Δ and Σ , i.e., the hyperplanes intersecting an arbitrarily fixed linear segment (s_0, s_1) , with $s_0 \in \Delta$ and $s_1 \in \Sigma$ (Figure 5.13). Since the number of possible pairs of d -cells is $O(s^{2d})$, this whole procedure takes $O(s^{2d+1}) = O(s \pi(S))$ time.

Before turning to the proof of Theorem 10.2.1, let us first illustrate its use in derandomization. We provide one prototype example below. More examples will be provided in the exercises.

Example 10.2.2 (Arrangements of hyperplanes)

Consider the construction of a point location structure, based on top-down random sampling, for arrangements of hyperplanes (Section 6.5). Randomization is used in this construction only in the recursive division step. Given a set N of hyperplanes, this step consists in choosing a random sample $R \subseteq N$ of size r , for a large enough constant r , and then dividing the problem suitably. It follows from the random sampling results that, with high probability, each simplex in the canonical triangulation formed by R intersects only $b(n/r) \log r$ hyperplanes, for some explicit constant $b > 1$. We verify that this is indeed the case. If not, we repeat the experiment until the property under consideration is satisfied. This randomized division needs $O(1)$ trials on the average, and hence, $O(n)$ expected time, since r is a constant. We can derandomize the division step as follows. Let $\Pi(N)$ be the space of linear ranges induced by N . Using Theorem 10.2.1, we obtain a $(\log r)/r$ -net for $\Pi(N)$ of $O(r)$ size in $O(n)$ time. Each simplex Δ in the canonical triangulation formed by R intersects at most $(d+1)(n \log r)/r = O([n \log r]/r)$ hyperplanes in N . This can be seen as follows. First, notice that the linear ranges induced by the edges of Δ are active over R , i.e., no hyperplane

in $N \setminus R$ intersects any of these edges. Since R is a $(\log r)/r$ -net, at most $n(\log r)/r$ hyperplanes in $N \setminus R$ can intersect any such edge. Moreover, each hyperplane in $N \setminus R$ intersecting Δ must intersect one of the $d + 1$ edges incident to its fixed vertex. This proves our claim.

In this fashion, the division step can be accomplished deterministically in $O(n)$ time, assuming that r is constant. In comparison, the randomized division step took *expected* $O(n)$ time. Substituting the randomized divisions by the deterministic ones, we are led to the following deterministic version of Theorem 6.5.1: Given any set of n hyperplanes in R^d , one can deterministically build a point location structure for the resulting arrangement, guaranteeing $O(\log n)$ query time, in $O(n^{d+\epsilon})$ time and space. The n^ϵ factor can be completely removed; see the exercises at the end of this section.

We have crucially used above the fact that the random sample size r is a constant. The constant factor within the Big-Oh notation depends rather badly on r . When the sample size is not constant, one can still get around the difficulty sometimes by using the properties that are special to the problem under consideration. Arrangements of hyperplanes provide a good example in this regard; see the exercises at the end of this section.

In this section, we shall confine ourselves to derandomization of only the static random sampling method. Derandomization of the dynamic sampling method (Section 5.4) is much more difficult, in the top-down setting (as in, say, Section 6.5.1) and also in the bottom-up setting.

10.2.1 ϵ -approximations

Now let $\Pi(N)$ be an arbitrary range space. In what follows, we adopt the convention of identifying every range $\sigma \in \Pi(N)$ with its stopper set, i.e., the set of objects in N in conflict with σ . This makes sense, because the trigger set of every range is empty. Thus, every range can be thought of as just a subset of N .

To prove Theorem 10.2.1, we shall need a notion that is stronger than ϵ -nets:

Definition 10.2.3 A subset $R \subseteq N$ is called an ϵ -approximation for a range space $\Pi(N)$, if, for every range $\sigma \in \Pi(N)$,

$$\left| \frac{|\sigma \cap R|}{r} - \frac{|\sigma|}{n} \right| \leq \epsilon, \quad (10.6)$$

where r and n denote the sizes of R and N , respectively.

What (10.6) says is that the relative number of points of R in every range approximates the relative size of that range with accuracy ϵ .

It is easy to see that every ϵ -approximation is an ϵ -net. But what makes ϵ -approximations so central in derandomization is that they are well behaved under union and also composition (in contrast, ϵ -nets are not well behaved under composition):

Observation 10.2.4

1. Suppose N_1, N_2, \dots, N_i are disjoint subsets of N of equal cardinality and suppose, for each $j \leq i$, R_j is an ϵ -approximation for $\Pi(N_j)$ of the same cardinality. Then $R_1 \cup \dots \cup R_i$ is an ϵ -approximation for $\Pi(N_1 \cup \dots \cup N_i)$.
2. Suppose R is an ϵ -approximation of a range space $\Pi(N)$ and S is a δ -approximation of the subspace $\Pi(R)$. Then S is an $(\epsilon + \delta)$ -approximation for $\Pi(N)$.

This observation will make the divide-and-conquer principle eminently applicable in the construction of ϵ -approximations. The following observation will also be useful to us:

Observation 10.2.5 Suppose R is an ϵ -approximation of a range space $\Pi(N)$ and S is a δ -net of the subspace $\Pi(R)$. Then S is an $(\epsilon + \delta)$ -net of $\Pi(N)$.

The analogue of Theorem 10.2.1 for ϵ -approximations is the following.

Theorem 10.2.6 Let $\Pi(N)$ be a range space of bounded dimension d . Let n be the size of N . Fix a parameter $r \geq 1$. Then there exists a $(1/r)$ -approximation for $\Pi(N)$ of $O(r^2 \log r)$ size. Moreover, one can deterministically compute such an approximation in $O(n[r^2 \log r]^d)$ time, assuming that $\Pi(N)$ satisfies the enumerability assumption.

10.2.2 The method of conditional probabilities

We shall begin by proving Theorems 10.2.6 and 10.2.1 for the special case, when the range space $\Pi(N)$ is *small* with respect to the given parameter r . By this, we mean that $\pi(N)$, the size of $\Pi(N)$, is bounded by a fixed degree polynomial in r . For sufficiently small range spaces, Theorem 10.2.6 follows from the following result. Since $\Pi(N)$ satisfies the enumerability assumption, we can assume that we are given a list of all ranges in $\Pi(N)$, wherein each range is specified as a subset of N .

Theorem 10.2.7 Assume that we are given a list of all ranges in $\Pi(N)$. Fix any $s \leq n$ such that $s \geq cr^2 \log \pi(N)$, for a large enough constant c . Then there exists a $(1/r)$ -approximation for $\Pi(N)$ of size exactly s . Moreover, it can be found deterministically in $O(n \pi(N) + n^2)$ time.

Remark. Note that $\log \pi(N) = O(\log r)$ for small range spaces.

Proof. First, we shall prove the existence of a required approximation in a probabilistic fashion, quite in the spirit of the earlier random sampling results (Chapter 5). Choose a random subset $S \subseteq N$ by independently tossing a coin with probability of success $p = s/n$, for each object in N . An object is put into S iff the toss for that object is successful. We shall show that, with positive probability, for every range $\sigma \in \Pi(N)$,

$$|\sigma \cap S| - p |\sigma| \leq \frac{s}{2r}. \quad (10.7)$$

In other words, the size of $\sigma \cap S$ differs from its expected size $p |\sigma|$ by at most $s/2r$. This means S is a $(1/2r)$ -approximation for $\Pi(N)$. We assume here, as well as in what follows, that $\Pi(N)$ contains the special range $\sigma = N$; if not, we just add this range to $\Pi(N)$. If we put $\sigma = N$ in (10.7), it follows that the size of every set S satisfying this equation differs from s by at most $s/2r$. Hence, we can expand S to a set containing exactly s elements by adding or deleting at most $s/2r$ elements. The resulting set is obviously a $(1/r)$ -approximation for $\Pi(N)$.

For each $\sigma \in \Pi(N)$, let q_σ denote the probability that (10.7) is violated for σ , if S is chosen in a random fashion, as described before. That (10.7) holds for all ranges with positive probability follows from the following stronger result:

$$\sum_{\sigma \in \Pi(N)} q_\sigma < 1. \quad (10.8)$$

We shall prove (10.8) by estimating each q_σ separately using the Chernoff technique. More precisely, we apply Corollary A.1.4 in the appendix to the set $M = \sigma$. This implies that, for each $\sigma \in \Pi(N)$,

$$q_\sigma \leq \exp\left(-\frac{s}{64r^2} + 32\right) \leq \exp\left(-\frac{c}{64} \log \pi(N) + 32\right) < \frac{1}{\pi(N)},$$

if c is large enough (here $\exp(z)$ denotes e^z). This implies (10.8).

We shall now convert the probabilistic existence proof given above into a deterministic construction algorithm, using a certain method based on conditional probabilities. This will yield us a set S satisfying (10.7), for all $\sigma \in \Pi(N)$.

Let a_1, \dots, a_n be any enumeration of the objects in N . For a range $\sigma \in \Pi(N)$, and any arguments $\delta_1, \dots, \delta_i \in \{0, 1\}$, let $q_\sigma(\delta_1, \dots, \delta_i)$ denote the probability that (10.7) is violated for σ , if S in that equation is chosen as follows: For $1 \leq j \leq i$, S contains a_j iff $\delta_j = 1$. For each $j > i$, we independently toss a coin with success probability p as before, and include a_j in S iff the toss is successful.

In particular, q_σ without any arguments, i.e., for $i = 0$, has the same meaning as in (10.8). Similarly, $q_\sigma(\delta_1, \dots, \delta_n)$ is 1 or 0 depending upon the validity of (10.7), for the range σ and the set $S \subseteq N$ with characteristic vector $(\delta_1, \dots, \delta_n)$, i.e., the set S such that, for all $j \leq n$, $a_j \in S$ iff $\delta_j = 1$. Let us now define a quantity

$$Q_i(\delta_1, \dots, \delta_i) = \sum_{\sigma \in \Pi(N)} q_\sigma(\delta_1, \dots, \delta_i). \quad (10.9)$$

We shall construct the set S by choosing the components of its characteristic vector $(\delta_1, \dots, \delta_n)$, one by one. Suppose the components $\delta_1, \dots, \delta_{i-1}$ have already been chosen. Next, we evaluate the terms $Q_i(\delta_1, \dots, \delta_{i-1}, 0)$ and $Q_i(\delta_1, \dots, \delta_{i-1}, 1)$ somehow. We then select δ_i so that $Q_i(\delta_1, \dots, \delta_i)$ is the smaller of these two terms. In this fashion, S is completely specified in n steps.

Why is the method correct? By (10.8), we know that $Q_0 < 1$. By the law of conditional probability, $Q_i(\delta_1, \dots, \delta_i)$ is a weighted average of $Q_i(\delta_1, \dots, \delta_{i-1}, 0)$ and $Q_i(\delta_1, \dots, \delta_{i-1}, 1)$. Hence, by construction,

$$Q_i(\delta_1, \dots, \delta_i) \leq Q_{i-1}(\delta_1, \dots, \delta_{i-1}). \quad (10.10)$$

Thus, we have

$$Q_n(\delta_1, \dots, \delta_n) \leq \dots \leq Q_1(\delta_1) \leq Q_0 < 1. \quad (10.11)$$

Since $Q_n(\delta_1, \dots, \delta_n)$ is either 0 or 1, this implies that it must be 0. This is the same as saying that S with the characteristic vector $(\delta_1, \dots, \delta_n)$ satisfies (10.7), and we are done.

It remains to specify how to evaluate the terms $Q_i(\delta_1, \dots, \delta_i)$ at each stage. Practically, this is the most expensive part of the method. The idea is to calculate each $q_\sigma(\delta_1, \dots, \delta_i)$ by brute force, using the fact that it only depends on the cardinalities of the sets $X_\sigma = \sigma \setminus \{a_1, \dots, a_i\}$ and $Y_\sigma = \sigma \cap \{a_j \mid j \leq i, \delta_j = 1\}$. Let us denote the cardinalities of these two sets by x_σ and y_σ , respectively. Then, clearly, $q_\sigma(\delta_1, \dots, \delta_i)$ is the sum of two probabilities: (1) the probability that the number of randomly chosen elements from X_σ is less than or equal to $\lceil p|\sigma| - y_\sigma - s/2r \rceil$, and (2) the probability that the number of randomly chosen elements from X_σ is greater than $\lfloor p|\sigma| - y_\sigma + s/2r \rfloor - 1$. Let $f(u, k)$ denote the probability of obtaining $\leq k$ successes in a sequence of u independent tosses of our coin with bias p . Clearly,

$$f(u, k) = \sum_{j=0}^k \binom{u}{j} p^j (1-p)^{u-j}.$$

By the preceding observation, $q_\sigma(\delta_1, \dots, \delta_i)$ is

$$[f(x_\sigma, \lceil p|\sigma| - y_\sigma - s/2r \rceil)] + [1 - f(x_\sigma, \lfloor p|\sigma| - y_\sigma + s/2r \rfloor - 1)].$$

We shall tabulate in $O(n^2)$ time the numbers $f(u, k)$ in advance, for all $1 \leq k \leq u \leq n$, and maintain the numbers x_σ and y_σ , for each range σ , throughout the construction of S . Then each desired sum $Q_i(\delta_1, \dots, \delta_i)$ can be evaluated in $O(\pi(N))$ time.² Thus, the total running time of the algorithm is $O(n\pi(N) + n^2)$. \square

Similarly, for sufficiently small range spaces, Theorem 10.2.1 follows from the following result.

Theorem 10.2.8 *Assume that we are given a list of all ranges in $\Pi(N)$. A $(1/r)$ -net of size $O(r \log \pi(N))$ can be computed for $\Pi(N)$ in $O(n\pi(N) + n^2)$ time, where n and $\pi(N)$ denote the sizes of N and $\Pi(N)$, respectively.*

Proof. The existence of such a net follows Theorem 5.7.4. In fact, this theorem implies existence of a $(1/r)$ -net of even better $O(r \log r)$ size. The inequality (10.8) will also hold in the present setting, if we redefine the various quantities appropriately. The parameter s is now set to be larger than $cr \log \pi(N)$, for a large enough constant c . The set S is randomly chosen as before by independently tossing a coin of bias $p = s/n$ for each object in N . The quantity q_σ in (10.8) is now redefined as follows. If the size of σ exceeds n/r , we let q_σ denote the probability that $\sigma \cap S$ is empty. Otherwise, we define q_σ to be zero. For the special range $\sigma = N$, we let q_σ denote the probability that the size of S exceeds $br \log \pi(N)$, for some large enough constant b . Thus, any S satisfying (10.8) with this new interpretation is a $(1/r)$ -net for $\Pi(N)$.

That (10.8) holds in the above setting is implicit in the proofs of the earlier random sampling results; e.g., see the proof of Theorem 5.1.2, or rather its analogue in the setting of Bernoulli sampling (Exercise 5.1.1). In fact, the proof of (10.8) is much simpler in the present setting. We do not even need the power of the Chernoff technique.

Once we know that (10.8) holds with this new interpretation, a candidate set S satisfying this equation can be constructed deterministically by the method of conditional probabilities, just as in the proof of Theorem 10.2.7. In fact, the construction is simpler this time, because the quantities $q_\sigma(\delta_1, \dots, \delta_i)$ are easier to compute; we leave this verification to the reader. The constructed set S is a $(1/r)$ -net for $\Pi(N)$ of $O(r \log \pi(N))$ size. \square

²In the real (RAM) model of computation. In practice, we need to evaluate each sum with only a limited $1/2n$ precision, if we ensure that Q_0 is, in fact, less than $1/2$. In this case, we can afford to lose a quantity of size $1/2n$ in each step, and $Q_n(\delta_1, \dots, \delta_n)$ would still be less than one.

10.2.3 Divide and conquer

We now proceed to prove Theorems 10.2.1 and 10.2.6 for arbitrary range spaces of bounded dimension d . It suffices to prove Theorem 10.2.6, because it readily implies Theorem 10.2.1, in conjunction with Theorem 10.2.8. Let us see how. Suppose we want to construct a $(1/r)$ -net for $\Pi(N)$. First, we construct a $(1/2r)$ -approximation R for $\Pi(N)$ of $O(r^2 \log r)$ size. This can be done using Theorem 10.2.6. Next, we compute a $(1/2r)$ -net for the subspace $\Pi(R)$ of $O(r \log r)$ size. This can be done using Theorem 10.2.8, because the subspace $\Pi(R)$ is evidently small—its size is $O([r^2 \log r]^d)$. What results is a $(1/r)$ -net for $\Pi(N)$ of $O(r \log r)$ size (Observation 10.2.5). The overall cost of this construction is dominated by the $O(n(r^2 \log r)^d)$ cost of the first step.

So let us turn to the proof of Theorem 10.2.6 for an arbitrary range space $\Pi(N)$ of bounded dimension d . The basic idea is to reduce our problem to constructions of ϵ -approximations for small subspaces of $\Pi(N)$. This will be done by using the divide-and-conquer principle provided by Observation 10.2.4.

Since we are going to be using Theorem 10.2.7 repeatedly, let us fix an integer function

$$s(r) = \lfloor ar^2 \log r \rfloor, \quad (10.12)$$

where the constant a will be chosen to be large enough. Thus, Theorem 10.2.7 enables us to find a $(1/r)$ -approximation of size exactly $s(r)$, for any r . We assume here that r is bounded below by some large enough constant.

We begin by choosing a constant $b = 4^{d+1} + 1$ and an additional parameter k in a manner to be specified soon. Partition N arbitrarily into b^k groups of size exactly $\lfloor n/b^k \rfloor$ and a subset N' of at most b^k remaining objects. Let us ignore the set N' for a while. Let \bar{N} denote the union of the b^k groups. We shall choose k such that the size of each of these groups is $O(r^2 \log r)$. To be precise, let us choose k so that

$$s(r_0) \leq \lfloor n/b^k \rfloor < bs(r_0), \text{ where } r_0 = 8r.$$

Note that the size of the remaining set $N' = N \setminus \bar{N}$ is at most $b^k \leq n/s(r_0) \leq n/2r$, assuming that the constant $a > 2$. Hence, it suffices to compute a $(1/2r)$ -approximation for \bar{N} , because it would trivially be a $1/r$ -approximation for the whole set N . So in what follows, we shall completely ignore the set N' . Our goal is to compute a $(1/2r)$ -approximation for \bar{N} .

Consider the complete b -ary tree with height k , whose leaves are in one-to-one correspondence with the preceding b^k groups within \bar{N} . Each interior node τ of this tree naturally corresponds to the union of the groups associated with the leaves below τ . (This correspondence is only conceptual.) By the leaves below τ , we mean the leaves of the subtree rooted at τ . Observe that

the set that corresponds to the root of the tree is \bar{N} . We shall compute certain approximations of the sets associated with the nodes of this tree in a bottom-up fashion. In the initial 0th step, we shall compute, for the set associated with each leaf of the tree, a $(1/r_0)$ -approximation, where $r_0 = 8r$ as before. Using Theorem 10.2.7, we can find such an approximation of size exactly $s_0 = s(r_0)$. Inductively, let us consider the i th step, where $1 \leq i \leq k$. By this time, we would have computed appropriate approximations for the sets associated with all nodes at height $< i$. For each node at height i , we shall apply Theorem 10.2.7 as follows. We form the union U of the approximations associated with its b children, and then find a $(1/r_i)$ -approximation of size $s_i = s(r_i)$ for this union, for a certain parameter r_i , to be chosen soon. We have to make sure that Theorem 10.2.7 is applicable in this computation. With that in mind, we shall ensure that

$$r_{i-1} \leq 2r_i, \text{ for all } i. \quad (10.13)$$

In that case, the size of the above union U is $bs(r_{i-1}) \leq bs(2r_i)$. Since the range space has bounded dimension d , the size of the range subspace $\Pi(U)$ is $O([bs(2r_i)]^d)$; in particular, $\Pi(U)$ is small. For Theorem 10.2.7 to be applicable, we only need that

$$s(r_i) \geq cr_i^2 \log \pi(U).$$

This is clearly satisfied if the constant a in (10.12) is large enough.

At the end of the k th step, we would have computed a certain approximation of size $s(r_k)$ for the set \bar{N} that corresponds to the root. By the repeated use of Observation 10.2.4, it follows that this is an ϵ -approximation, where

$$\epsilon \leq 1/r_0 + 1/r_1 + \cdots + 1/r_{k+1}. \quad (10.14)$$

We shall ensure that

$$1/r_0 + 1/r_1 + \cdots + 1/r_{k+1} \leq 1/2r, \text{ and } r_k \text{ is a constant multiple of } r. \quad (10.15)$$

This would imply that the approximation that we have at the end is, in fact, a $(1/2r)$ -approximation for \bar{N} of size $s(r_k) = O(r^2 \log r)$.

Let us specify the parameters r_i , for $i > 0$, so that (10.13) and (10.15) are satisfied. It is easy to see that the following choice would do:

$$\begin{aligned} r_i &= 2^{i+3}r, & \text{for } i = 1, \dots, \lfloor (k+1)/2 \rfloor, \text{ and} \\ r_i &= 2^{k+4-i}r, & \text{for } i = \lfloor (k+1)/2 \rfloor + 1, \dots, k+1. \end{aligned}$$

It only remains to analyze the running time of the algorithm. The computation at a node with height $j+1$, $j \geq 0$, consists in finding an approximation for a set of size $bs_j = O(s_j)$. The range space induced by this

set has size $O(s_j^d)$. By Theorem 10.2.7, the cost of this step is $O(s_j^{d+1}) = O((r_j^2 \log r_j)^{d+1})$. The number of nodes at height $j+1$ is $b^{k-j} = (1+4^{d+1})^{k-j}$. Hence, the total cost of the $(j+1)$ th step is bounded, within a constant factor, by

$$(1 + 4^{d+1})^{k-j} (r_j^2 \log r_j)^{d+1}.$$

If one substitutes the preceding definitions of r_j in this bound, it is easily seen to be decreasing exponentially with j . Hence, the total cost is dominated by the cost of the initial step. Since the number of leaves is $b^k \leq n/s_0$, the cost of the initial step is $O([n/s_0]s_0^{d+1}) = O(n[r^2 \log r]^d)$.

This finishes the proof of Theorem 10.2.6.

Exercises

10.2.1 Let N be a set of n hyperplanes in R^d . Let $\Pi(N)$ be the space of linear ranges induced by N . Fix a simplex Δ in R^d . Let $\nu(\Delta)$ denote the number of vertices of the arrangement formed by N within the (relative) interior of Δ . Similarly, for any $S \subseteq N$, let $\rho(\Delta)$ denote the number of vertices of the arrangement formed by S within the (relative) interior of Δ . If S were a random sample of N , then the expected value of $\rho(\Delta)$ would be $O([s/n]^d \nu(\Delta))$, where s denotes the size of S (why?). With this in mind, let us call S a *strong* $(1/r)$ -net for (N, Δ) , if it is a $(1/r)$ -net for $\Pi(N)$ in the usual sense, and moreover, $\rho(\Delta)$ is bounded by, say, $4(s/n)^d \nu(\Delta)$. Show that a strong $(1/r)$ -net for (N, Δ) of size $O(r \log n)$ can be deterministically computed in time polynomial in n . (Hint: First, give a probabilistic existence proof and then convert it into a deterministic construction algorithm using the method of conditional probabilities.)

10.2.2 (Deterministic version of the Cutting Lemma) Let N be a set of n hyperplanes in R^d . In Section 6.3.1, we demonstrated the existence of a $(1/r)$ -cutting for N of $O(r^d)$ size. We also gave a randomized algorithm to construct such a cutting in $O(nr^{d-1})$ expected time. The goal now is to give a deterministic algorithm with $O(nr^{d-1})$ running time. Proceed in the following steps.

- (a) Let $\Pi(N)$ be the space of linear ranges induced by the set N , and let S be an ϵ -approximation for $\Pi(N)$. Let R be a δ -cutting for S . Then show that R is a $d(\epsilon + \delta)$ -cutting for N .
- (b) Suppose r is a constant. In that case, show that a $(1/r)$ -cutting of $O(r^d) = O(1)$ size can be computed deterministically in $O(n)$ time. (Hint: First, compute a $(1/2dr)$ -approximation of constant size. Then compute a $(1/2dr)$ -cutting of this constant-size approximation in constant time. By (a), what results is a $(1/r)$ -cutting.)
- **(c) Now, consider the general case, when $r \leq n$ is arbitrary. Fix r_0 to be a large enough constant. For $k = 1, \dots, \lceil \log_{r_0} r \rceil$, we shall compute a $(1/r_0^k)$ -cutting by successive refinement. The last cutting would be the required $(1/r)$ -cutting. For $k = 1$, we simply use the algorithm in the previous step for constant r_0 . At the k th step, we shall refine each simplex Δ in the cutting that exists at the end of the $(k-1)$ th step. This is done as follows. Let $N(\Delta)$ be the set of hyperplanes in N intersecting Δ . If the size of $N(\Delta)$ is less than n/r_0^k , we leave Δ as it is.

Otherwise, with $q = r_0^k |N(\Delta)|/n$, we first compute a $(1/2dq)$ -approximation S for $N(\Delta)$, using Theorem 10.2.6, and a strong $(1/2dq)$ -net \tilde{S} for (S, Δ) , using Exercise 10.2.1. We compute the restriction of the arrangement formed by \tilde{S} within Δ , and triangulate it in a canonical fashion.

Show that the cutting that results at the end of the k th stage is a $(1/r)$ -cutting for N . Show that its size is $O(r^d)$ and that its construction takes $O(nr^{d-1})$ time.

- (d) The hierarchy of cuttings computed above, for $r = n$, can be easily converted into a point location structure for the arrangement formed by N . The idea is to trace the query point through the succession of cuttings until one lands in a simplex in the last k th cutting; here $k = O(\log n)$. Since $r = n$, this last simplex is entirely contained within the cell of the arrangement over N containing the query point. Elaborate this idea fully. Show that the resulting point location structure has $O(n^d)$ size and that the preprocessing time is $O(n^d)$.

***10.2.3** Derandomize the data structures for simplex range search and half-space range search in Chapter 8. You will need to use the deterministic version of the Cutting Lemma for constant r .

10.2.4 Explicitly calculate how the constant factors within the Big-Oh notations in this section depend on the dimension d of the range space.

****10.2.5** Derandomize the linear programming algorithm in Exercise 7.1.6 so as to obtain a deterministic algorithm with $O(d^{O(d)}n)$ running time, where d denotes the dimension, and n denotes the number of constraints.

†Can you design a deterministic algorithm with subexponential dependence on d ?

Bibliographic notes

Reducing the number of random bits used by randomized algorithms has been an extensively investigated topic; see, e.g., Alon et al. [7], Luby [134], and Naor and Naor [173]. That randomized incremental algorithms in computational geometry perform well in a pseudo-random setting (Section 10.1) was proved by Mulmuley [169]. Prior to this, Karloff and Raghavan [123] had proved that quick-sort performs well in the same pseudo-random setting with five seeds; this is a one-dimensional analogue of the above result. The limited independence property of the pseudo-random generator used here (Proposition 10.1.1) was proved by Chor and Goldreich [64]. For derandomization of the randomized incremental algorithms for convex hulls and linear programming, see [40] and [54], respectively.

The existence of an ϵ -approximation with $O([1/\epsilon^2] \log[1/\epsilon])$ size was proved by Vapnik and Chervonenkis [220, 221]. Its deterministic construction (Section 10.2) was given by Matoušek [146], culminating a series of earlier works [1, 48, 137, 138]; also see [54] for some improvements. The application (Exercise 10.2.3) to range search is also due to Matoušek [147, 141]. The derandomization technique was applied to Clarkson's linear programming algorithm (Exercise 10.2.5) by Chazelle and Matoušek [54]. The method of conditional probabilities used in Section 10.2.2 is due to Raghavan [189] and Spencer [212]. The deterministic construction of

ϵ -nets for small range spaces (Theorem 10.2.8) is due to Chazelle and Friedman [48], where it is done using a greedy method. The deterministic construction of a $(1/r)$ -cutting (Exercise 10.2.2) is due to Matoušek [146], for $r = O(n^{1-\epsilon})$, and due to Chazelle [40], for general $r \leq n$. The notion of strong nets, its deterministic construction (Exercise 10.2.1), and the point location structure in Exercise 10.2.2 are also due to Chazelle [40].

Appendix A

Tail estimates

We give here a self-contained treatment of some tail estimates from probability theory. These are used on a few occasions in the book, mainly in proving high-probability performance bounds. A reader with less theoretical inclination can either ignore such performance bounds, or simply take them on trust. In that case, this appendix can be ignored. Even a theoretically inclined reader may wish to refer to this appendix only on the second reading.

Let X be a (discrete) random variable. Let $E[X]$ denote its expected value. We are interested in bounding the probability that X deviates from $E[X]$ by a given amount. The goal is to prove as tight a bound as possible. The nature of the bound will depend on the random variable X .

The simplest bound that holds for any nonnegative random variable X is the so-called *Markov's inequality*: For any $\alpha > 0$,

$$\text{prob}\{X \geq \alpha\} \leq \frac{E(X)}{\alpha}. \quad (\text{A.1})$$

This follows directly from the definition of $E[X]$. Another important inequality is the so-called *Chebychev's inequality*. Let μ denote the mean, i.e., the expected value of X . The *variance* of X is defined to be the expected value of the random variable $(X - \mu)^2$. Its square root is called the *standard deviation* of X . Let us denote it by σ . Chebychev's inequality is the following:

$$\text{prob}\{|X - \mu| \geq t\sigma\} \leq 1/t^2. \quad (\text{A.2})$$

This follows by applying Markov's inequality to the nonnegative random variable $(X - \mu)^2$.

Notation. In what follows, we shall sometimes denote the function e^x by $\exp(x)$.

A.1 Chernoff's technique

Markov's and Chebychev's inequalities hold for any (nonnegative) random variable. Quite often, it is possible to obtain a much better bound if the random variable X in question can be expressed as a sum $\sum_{i=1}^n X_i$ of independent random variables X_i . This can be done using the so-called *Chernoff technique*. The basic idea of this technique is the following. Consider the random variables $e^{\lambda X_i}$, where λ is a real parameter. Since the X_i 's are independent, the random variables $e^{\lambda X_i}$ are also independent. Hence,

$$E[e^{\lambda X}] = E[e^{\lambda(X_1 + \dots + X_n)}] = E\left[\prod_{i=1}^n e^{\lambda X_i}\right] = \prod_{i=1}^n E[e^{\lambda X_i}]. \quad (\text{A.3})$$

The method consists in applying this equality in conjunction with Markov's inequality for the random variable $e^{\lambda X}$; the parameter λ has to be chosen judiciously. More precisely, we note that, for $\lambda > 0$ and any real number x ,

$$\text{prob}\{X > x\} = \text{prob}\{e^{\lambda X} > e^{\lambda x}\} \leq e^{-\lambda x} E[e^{\lambda X}],$$

where the last inequality follows by applying Markov's inequality to the non-negative random variable $e^{\lambda X}$. Substituting (A.3) in this inequality, we get

$$\text{prob}\{X > x\} \leq e^{-\lambda x} \prod_{i=1}^n E[e^{\lambda X_i}], \text{ for } \lambda > 0. \quad (\text{A.4})$$

Using $\lambda < 0$, we similarly get

$$\text{prob}\{X < x\} = \text{prob}\{e^{\lambda X} > e^{\lambda x}\} \leq e^{-\lambda x} \prod_{i=1}^n E[e^{\lambda X_i}], \text{ for } \lambda < 0. \quad (\text{A.5})$$

We shall now illustrate this method with the help of several examples. Before proceeding further, let us note that we do not really need full independence among all X_i 's for (A.3) to hold. It suffices if, say, each X_i is independent of all X_j , $j > i$; this is actually the condition which is satisfied in all applications in this book. In what follows, this limited independence condition would also suffice.

We begin with the simplest example. Let X_1, \dots, X_n be i.i.d. (independent, identically distributed) random variables taking values 1 and -1 with equal probability. Let $X = X_1 + \dots + X_n$. Then $E(X) = 0$. The Chernoff technique provides a good bound on the probability that X takes values at a distance ϵn away from its expected value 0:

Theorem A.1.1 $\text{Prob}\{X > \epsilon n\} \leq e^{-\epsilon^2 n/2}$, for any $\epsilon > 0$.

Proof. For any λ ,

$$E(e^{\lambda X_i}) = (e^\lambda + e^{-\lambda})/2 = \cosh(\lambda) \leq e^{\lambda^2/2},$$

where the last inequality follows from the power series expansion for $\cosh(\lambda)$. Substituting the above bound in (A.4) with $x = \epsilon n$, we get

$$\text{prob}\{X > \epsilon n\} \leq e^{-\lambda \epsilon n + \lambda^2 n/2}, \text{ for } \lambda > 0.$$

Choosing $\lambda = \epsilon$ gives the required bound. \square

The previous example illustrates the Chernoff technique very well. The remaining bounds below are proved in a similar, but technically more complex fashion. Since these technical modifications are not very illuminating, the reader can skip the proofs on the first reading.

A.1.1 Binomial distribution

Now consider the case when X is distributed according to a *binomial distribution*. This is defined as follows. Assume that we are given a coin with bias (probability of success) $p > 0$. Let X be the random variable that is defined to be the number of successes obtained during n independent coin tosses. It is said to be distributed according to the *binomial distribution* with parameter p . Let X_i denote the random variable that is 0 with probability $1 - p$ and 1 with probability p . Clearly, $X = \sum_{i=1}^n X_i$, where the X_i 's are independent. This makes the Chernoff technique immediately applicable. First, consider the simplest case, when $p = 1/2$. In this case, we can define a new set of random variables $Z_i = 2X_i - 1$. The variable Z_i takes values 1 and -1 with equal probability. We can now apply Theorem A.1.1 to the sum $Z = \sum_{i=1}^n Z_i$. This immediately implies:

Corollary A.1.2 $\text{Prob}\{X - E[X] > \epsilon E[X]\} \leq e^{-\epsilon^2 n/2}$, for $\epsilon > 0$. Here the expected value $E[X]$ is $n/2$.

A refined tail estimate*

In Chapter 10, we shall require the following refined tail estimate for binomial distributions. The reader not interested in derandomization can skip this section.

Let $X = X_1 + \dots + X_m$ be a random variable that is distributed according to the binomial distribution with parameter p . Here X_1, \dots, X_m are i.i.d. random variables taking values 0 with probability $1 - p$ and 1 with probability p . Then $E(X) = mp$. The Chernoff technique helps us prove that the probability that X takes values at a distance a away from its expected value mp decreases fast:

Theorem A.1.3 Assume that $mp > 2$. For an arbitrary $0 < a \leq 2mp$,

1. $\text{prob}\{X - mp > a\} \leq \exp(-\frac{a^2}{4mp} + \frac{a^3}{2m^3p^3})$.
2. $\text{prob}\{X - mp < -a\} \leq \exp(-\frac{a^2}{2mp})$.

Proof. For $\lambda > 0$,

$$E(e^{\lambda X_i}) = (1-p) + pe^\lambda = 1 + p(e^\lambda - 1) \leq \exp[p(e^\lambda - 1)],$$

where the last inequality follows because $1+x \leq \exp(x)$, for every x . Substituting this bound for the expected value in (A.4), with $x = mp+a$ and $n = m$, we get

$$\text{prob}\{X - mp > a\} \leq \exp[-\lambda(mp+a) + mp(e^\lambda - 1)], \quad (\text{A.6})$$

for $\lambda > 0$. Pick

$$\lambda = \frac{a}{2mp} - \frac{a^2}{2m^3p^3}.$$

Since $mp > 2$, and $0 < a \leq 2mp$, we have $0 < \lambda < 1$. For λ in this range, $e^\lambda - 1 \leq \lambda + \lambda^2$. Hence the exponent within the square brackets in (A.6) is bounded by

$$-\lambda(mp+a) + mp(\lambda + \lambda^2) = -a\lambda + mp\lambda^2 \leq -\frac{a^2}{4mp} + \frac{a^3}{2m^3p^3}.$$

This proves the first estimate. For the other estimate, we will need $\lambda < 0$. So we use (A.5), with $x = mp-a$ and $n = m$, to get

$$\text{prob}\{X - mp < -a\} \leq \exp[-\lambda(mp-a) + mp(e^\lambda - 1)].$$

Set $\lambda = -a/(mp)$. Since $e^\lambda - 1 \leq \lambda + \lambda^2/2$, for all negative λ , the exponent within the square brackets on the right hand side is bounded by

$$-\lambda(mp-a) + mp \left(\lambda + \frac{\lambda^2}{2} \right) = a\lambda + mp \frac{\lambda^2}{2} \leq -\frac{a^2}{2mp}.$$

This proves the second estimate. □

Now let M be any set of cardinality m . For each element in M , toss a coin with bias (the probability of success) p independently. The element in M is said to be chosen iff the toss for that element is successful. Let us say that the bias p is of the form $p = s/n$, where s and n are some positive integers, with $n \geq m$.

Corollary A.1.4 *The probability that the number of chosen elements deviates from its expected value $pm = sm/n$ by more than $s/2r$, for a parameter $r > 1$, is bounded by $\exp(-s/(64r^2) + 32)$.*

Proof. In what follows, we shall assume that $s \geq 64r^2$. Otherwise, the bound in the corollary holds trivially because it gets larger than 1.

For each element in M , define a random variable which is 1 if that element is chosen—this happens with probability p —and zero otherwise. The number of chosen elements in M is thus the sum of such m independent random variables. Its expected value is pm . The deviation from the expected value pm can be bounded using Theorem A.1.3. We shall let $a = s/2r$ in that theorem. We have

$$pm \leq s, \text{ and hence, } a^2/pm \geq s/4r^2.$$

If m is large enough, say, $m \geq m_0 = n/4r$, then $a \leq 2mp$, and $a^3/(2p^3m^3) \leq 4$. Since $s \geq 64r^2$ and $r > 1$, we also have $mp > 2$, for $m \geq m_0$. Applying Theorem A.1.3, we now conclude that the deviation probability is bounded by

$$\exp\left(-\frac{s}{16r^2} + 4\right), \text{ if } m \geq m_0 = \frac{n}{4r}. \quad (\text{A.7})$$

For $m < m_0$, the first estimate in Theorem A.1.3 is not satisfactory because of its cubic term. We can get around this difficulty as follows. When $m < m_0$, we have $pm - s/2r < 0$. Hence, we only need to bound the probability that the number of chosen elements exceeds pm by more than $s/2r$. This is trivially bounded by the probability that the number of chosen elements exceeds $s/2r$. The latter probability obviously increases with m . When $m = m_0 = n/4r$, it equals the probability that the number of chosen elements deviates from the expected value $pm_0 = s/4r$ by more than $s/4r$. Hence, it can be estimated by applying Theorem A.1.3 as before, but with $a = s/4r$ this time. This yields the bound

$$\exp\left(-\frac{s}{64r^2} + 32\right), \text{ for } m \leq m_0 = \frac{n}{4r}.$$

Comparing with (A.7), it follows that the above bound holds for all m . □

A.1.2 Geometric distribution

In this section, we shall apply the Chernoff technique to the sum of independent *geometric distributions*. A geometric distribution is defined as follows. Suppose we are given a coin with probability of success $p > 0$. Let us toss the coin repeatedly and independently until the toss is successful. Let T be

the number of failures obtained before obtaining success;¹ as a convention, we shall assume that T also counts the last successful toss. We say that T is distributed according to the *geometric distribution* with parameter p . More generally, let T_n be the number of tosses required to obtain n successes; thus, $T = T_1$. We say that T_n is distributed according to the *negative binomial distribution* with parameter p . We can express T_n as a sum of n independent geometric distributions as follows. Let X_i be the random variable standing for the number of tosses between the i th and the $(i+1)$ th successes, including the toss for the $(i+1)$ th success. Then $T_n = X_1 + \dots + X_n$ and the X_i 's follow independent geometric distributions with parameter p . This makes the Chernoff technique immediately applicable.

Notice that X_i takes a positive integer value t with probability $p(1-p)^{t-1}$. Hence,

$$E(X_i) = \sum_{t=1}^{\infty} tp(1-p)^{t-1} = \frac{1}{p} \text{ and } E(T_n) = \frac{n}{p}.$$

Now we shall derive a tail estimate for T_n using the Chernoff technique. For the sake of simplicity, we shall assume that $p = 1/2$. This is sufficient in the book. We shall leave the general case to the reader.

When $p = 1/2$, $E[T_n] = 2n$. The Chernoff technique provides a sharp estimate on the probability that T_n exceeds its expected value $2n$ by a given fraction:

Theorem A.1.5 *Let $X = T_n = X_1 + \dots + X_n$. Then*

$$\text{prob}\{X > (2 + \epsilon)n\} \leq \exp\left(\frac{-\epsilon n}{4}\right), \text{ for } \epsilon \geq 3.$$

Proof. For $0 < \lambda < \ln 2$,

$$E(e^{\lambda X_i}) = \sum_{t=1}^{\infty} \frac{e^{\lambda t}}{2^t} = \frac{e^{\lambda}}{2 - e^{\lambda}}.$$

Applying (A.4) with $x = (2 + \epsilon)n$, we get

$$\text{prob}\{X > (2 + \epsilon)n\} \leq e^{-\lambda(2+\epsilon)n} \left(\frac{e^{\lambda}}{2 - e^{\lambda}}\right)^n = \left[\frac{e^{-\lambda(1+\epsilon)}}{(2 - e^{\lambda})}\right]^n. \quad (\text{A.8})$$

Set $e^{\lambda} = 1 + \epsilon/(2 + \epsilon)$. This satisfies our requirement that $\lambda < \ln 2$. Since

$$2 - e^{\lambda} = \frac{1}{1 + \epsilon/2}, \text{ and } e^{-\lambda} = 1 - \frac{\epsilon}{2 + 2\epsilon},$$

¹or the number of successes obtained before obtaining failure, depending upon whether you are an optimist or a pessimist.

the expression inside the square brackets in (A.8) becomes

$$\left(1 - \frac{\epsilon}{2+2\epsilon}\right)^{1+\epsilon} \left(1 + \frac{\epsilon}{2}\right). \quad (\text{A.9})$$

Since $1 - x \leq e^{-x}$,

$$\left(1 - \frac{\epsilon}{2+2\epsilon}\right)^{1+\epsilon} \leq \exp\left(-\frac{\epsilon}{2+2\epsilon}[1+\epsilon]\right) = \exp\left(-\frac{\epsilon}{2}\right).$$

In addition, for $\epsilon \geq 3$, $1+\epsilon/2 \leq e^{\epsilon/4}$. Hence, the expression within the square brackets in (A.9) is at most $e^{-\epsilon/4}$, and the theorem follows. \square

Corollary A.1.6 *Let $n = a \ln m$, for any constant $a > 0$. Let $X = T_n$. Then*

$$\text{prob}\{X > a \ln m\} \leq m^{-a(c-2)/4}, \text{ for large enough } c.$$

Proof. Let $n = a \ln m$ in Theorem A.1.5. \square

A.1.3 Harmonic distribution

Consider the random variables X_1, \dots, X_n , independently distributed, where $X_i = \mu(i)$, with probability $1/i$, and $X_i = 0$, with probability $1 - 1/i$. Let $X = X_1 + \dots + X_n$. When $\mu(i) = 1$, for all i , we say that X is distributed according to the *harmonic distribution*. Otherwise, we say it is distributed according to the *generalized harmonic distribution*. We shall be interested in two special cases:

1. Each $\mu(i)$ is bounded by a constant μ : In this case, $E[X]$ is bounded by μH_n , where $H_n = \sum_i 1/i$ is the n th harmonic number.
2. The quantity $\mu(i)/i$ is a nondecreasing function of i : In this case, $E[X]$ is bounded by $n(\mu(n)/n) = \mu(n)$.

For the preceding two special cases, we shall be able to show that X deviates from the corresponding bound on its expected value with a very low probability.

Theorem A.1.7 *Let $c > 1$. Then*

1. $\text{prob}\{X \geq c\mu(n)\} \leq (1/e)(e/c)^c$, if $\mu(i)/i$ is a nondecreasing function of i .
2. $\text{prob}\{X \geq c\mu H_n\} \leq e^{-H_n[1+c \ln(c/e)]}$, if $\mu(i)$ is bounded by a constant μ , for all i .

Proof. For a parameter λ ,

$$E(e^{\lambda X_i}) = (1-1/i)+(1/i)e^{\lambda\mu(i)} = 1+(1/i)(e^{\lambda\mu(i)}-1) \leq \exp[(1/i)(e^{\lambda\mu(i)}-1)], \quad (\text{A.10})$$

where the last inequality follows because $1 + x \leq \exp(x)$.

(1) If $\mu(i)/i$ is nondecreasing, then

$$(1/i)(e^{\lambda\mu(i)} - 1) \leq (1/n)(e^{\lambda\mu(n)} - 1), \text{ for each } i.$$

This follows because the function

$$f(\lambda) = (1/n)(e^{\lambda\mu(n)} - 1) - (1/i)(e^{\lambda\mu(i)} - 1)$$

is 0 at $\lambda = 0$, and $f'(\lambda) > 0$, for all positive λ . It follows that

$$E(e^{\lambda X_i}) \leq \exp[(1/n)(e^{\lambda\mu(n)} - 1)].$$

Using (A.4) with $x = c\mu(n)$, we get

$$\text{prob}\{X \geq c\mu(n)\} \leq \exp[e^{\lambda\mu(n)} - 1 - \lambda c\mu(n)].$$

Setting $\lambda = \ln(c)/\mu(n)$ proves the first claim.

(2) When each $\mu(i)$ is bounded by μ , (A.10) can be simplified as follows:

$$E(e^{\lambda X_i}) \leq \exp[(1/i)(e^{\lambda\mu} - 1)]. \quad (\text{A.11})$$

Using (A.4), with $x = c\mu H_n$, we now get

$$\text{prob}\{X \geq c\mu H_n\} \leq \exp[H_n(e^{\lambda\mu} - 1) - \lambda c\mu H_n].$$

Choosing $\lambda = (\ln c)/\mu$ proves the second claim. \square

A.2 Chebychev's technique

To apply Chernoff's technique to a sum $I = \sum_{i=1}^m I_i$, one needs (almost) full independence among the random variables I_i . When the random variables under consideration are only mildly independent, the following Chebychev's technique can be used to derive weaker, but still useful tail estimates.

A sequence I_1, \dots, I_m of random variables is called t -wise independent, if every t -tuple of distinct variables in the sequence is independent; but the knowledge of any t of these variables may determine all the others. In what follows, we shall demonstrate Chebychev's technique when each I_j is an identical, Bernoulli random variable. This means each I_j is a 0–1 random variable such that $I_j = 1$, with probability p , and $I_j = 0$, with probability $1 - p$. Here p is a parameter in the range $(0, 1)$.

Theorem A.2.1 Let $I = \sum_{j=1}^m I_j$ be the sum of $2k$ -wise independent, identical Bernoulli random variables. By a Bernoulli random variable, we mean a variable that is one with probability $p > 0$ and zero otherwise. Let $\mu := E(I) = mp$. Then, for a fixed k , $\text{prob}\{|I - \mu| \geq \mu\} = O(1/\mu^k)$. In particular, $\text{prob}\{I = 0\} = O(1/\mu^k)$.

Proof. Consider the following generalized Chebychev's inequality, which can be proved just like Markov's inequality (A.1):

$$\text{prob}\{|J| \geq t\} \leq \frac{E[\phi(J)]}{\phi(t)},$$

where ϕ is any nonnegative function, and J is any random variable. Letting $\phi(t) = t^{2k}$, $J = I - \mu$, and setting $t = \mu$, we get

$$\text{prob}\{|I - \mu| \geq \mu\} \leq \frac{E[(I - \mu)^{2k}]}{\mu^{2k}}. \quad (\text{A.12})$$

It suffices to show that the numerator is $O(\mu^k)$. Since $\mu = E[I] = \sum_{j=1}^m E[I_j]$, we can write $(I - \mu)^{2k} = (\sum_{j=1}^m I_j - E[I_j])^{2k}$. By linearity of expectation, it suffices to estimate the expectation of each individual term in the multinomial expansion of the right-hand side. For any j , all terms divisible by $I_j - E[I_j]$, but not by its higher power, have zero expectation. This follows from the $2k$ -wise independence property. Each remaining term has $c \leq k$ distinct nonunit product terms of the form $(I_j - E[I_j])^i$, such that $i > 1$ and $\sum i = 2k$. By the $2k$ -wise independence property, the expectation of such term is the product of $E[(I_j - E[I_j])^i]$. Because $E[I_j] = p$, we have

$$\begin{aligned} E[(I_j - E[I_j])^i] &= (1-p)(-p)^i + p(1-p)^{i-1} \\ &\leq p[-(1-p)(-p)^{i-1} + (1-p)^i] \leq p[1+1] = 2p. \end{aligned}$$

Hence, the expectation of a term that has c distinct nonunit product terms is $O(p^c)$. There are $\binom{m}{c} \binom{2k-1}{c-1} = O(m^c)$ such terms, assuming that k and c are constants. Their total contribution is $O(m^c p^c) = O(\mu^c)$. We have already seen that the maximum value of c is k . Hence, the numerator of the right side of (A.12) is $O(k \mu^k) = O(\mu^k)$, as claimed. Since the denominator is μ^{2k} , the theorem follows. \square

Bibliographic notes

Chernoff's technique originates in [60]. Theorems A.1.1 and A.1.3 are from Spencer [212]. Corollary A.1.4 is from Matoušek [146]. Theorem A.1.7 is from Clarkson, Mehlhorn, and Seidel [70]. Theorem A.2.1 is from Reif and Sen [191].

Bibliography

The following abbreviations are used in this bibliography:

FOCS	<i>Annual IEEE Symposium on the Foundations Of Computer Science</i> (proceedings)
SCG	<i>Annual ACM Symposium on Computational Geometry</i> (proceedings)
SODA	<i>Annual ACM-SIAM Symposium On Discrete Algorithms</i> (proceedings)
STOC	<i>Annual ACM Symposium on the Theory Of Computing</i> (proceedings)
DCG	<i>Discrete & Computational Geometry</i>

- [1] P. K. Agarwal. Partitioning arrangements of lines: I. An efficient deterministic algorithm. *DCG*, 5:449–483, 1990.
- [2] P. K. Agarwal and J. Matoušek. Dynamic half-space range reporting and its applications. Technical Report CS-1991-43, Department of Computer Science, Duke University, Durham, NC, 1991.
- [3] P. K. Agarwal and J. Matoušek. Ray shooting and parametric search. In *24th STOC*, pages 517–526, 1992.
- [4] A. Aggarwal, B. Chazelle, L. J. Guibas, C. Ó'Dúnlaing, and C. K. Yap. Parallel computational geometry. *Algorithmica*, 3:293–327, 1988.
- [5] A. Aggarwal, M. Hansen, and T. Leighton. Solving query-retrieval problems by compacting Voronoi diagrams. In *21st STOC*, pages 331–340, 1990.
- [6] N. Alon, I. Bárány, Z. Füredi, and D. Kleitman. Point selections and weak ε -nets for convex hulls. Manuscript, 1991.
- [7] N. Alon, O. Goldreich, J. Hastad, and R. Peralta. Simple constructions of almost k -wise independent random variables. In *31st FOCS*, pages 544–553, 1990.
- [8] N. Alon and E. Gyori. The number of small semispaces of a finite set of points in the plane. *J. Combin. Theory, Ser. A*, 41:154–157, 1986.
- [9] N. Alon and N. Megiddo. Parallel linear programming in fixed dimension almost surely in constant time. In *31st FOCS*, pages 574–582, 1990.
- [10] C. Aragon and R. Seidel. Randomized search trees. In *30th FOCS*, pages 540–545, 1989.
- [11] B. Aronov, B. Chazelle, H. Edelsbrunner, L. J. Guibas, M. Sharir, and R. Wenger. Points and triangles in the plane and halving planes in space. *DCG*, 6:435–442, 1991.
- [12] B. Aronov, J. Matoušek, and M. Sharir. On the sum of squares of cell complexities in hyperplane arrangements. In *7th SCG*, pages 307–313, 1991.

- [13] B. Aronov and M. Sharir. On the zone of a surface in a hyperplane arrangement. In *Proceedings of the 2nd Workshop on Algorithms and Data Structures*, volume 519 of *Lecture Notes in Computer Science*, pages 13–19. Springer-Verlag, 1991.
- [14] S. Arya and D. Mount. Approximate nearest neighbor queries in fixed dimensions. To appear in 4th SODA, 1993.
- [15] M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading divide-and-conquer: A technique for designing parallel algorithms. *SIAM J. Comput.*, 18:499–532, 1989.
- [16] F. Aurenhammer. Power diagrams: Properties, algorithms and applications. *SIAM J. Comput.*, 16:78–96, 1987.
- [17] F. Aurenhammer and O. Schwarzkopf. A simple on-line randomized incremental algorithm for computing higher order Voronoi diagrams. In *7th SCG*, pages 142–151, 1991.
- [18] I. Bárány, Z. Füredi, and L. Lovász. On the number of halving planes. In *5th SCG*, pages 140–144, 1989.
- [19] M. Ben-Or. Lower bounds for algebraic computation trees. In *15th STOC*, pages 80–86, 1983.
- [20] J. L. Bentley. Algorithms for Klee’s rectangle problems. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, 1977.
- [21] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28:643–647, 1979.
- [22] J. L. Bentley and J. Saxe. Decomposable searching problems: I Static-to-dynamic transformations. *J. Algorithms*, 1:301–358, 1980.
- [23] M. Bern, D. Dobkin, and D. Eppstein. Hidden surface removal for rectangles. *J. Comput. Syst. Sci.*, 40:49–59, 1989.
- [24] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth. Classifying learnable geometric concepts with the Vapnik-Chervonenkis dimension. *J. ACM*, 36:929–965, 1989.
- [25] J.-D. Boissonnat, O. Devillers, R. Schott, M. Teillaud, and M. Yvinec. Applications of random sampling to on-line algorithms in computational geometry. *DCG*, 8:51–71, 1992.
- [26] J.-D. Boissonnat, O. Devillers, and M. Teillaud. A randomized incremental algorithm for constructing higher order Voronoi diagrams. To appear in Algorithmica. An abstract is available in *Second Canadian Conference on Computational Geometry, 1990*.
- [27] J.-D. Boissonnat and M. Teillaud. On the randomized construction of the Delaunay tree. Technical Report 1140, INRIA (France), 1989.
- [28] H. Brönnimann and B. Chazelle. How hard is halfspace range searching? In *8th SCG*, pages 271–275, 1992.
- [29] A. Bronsted. *An Introduction to Convex Polytopes*. Springer-Verlag, New York, 1983.
- [30] K. Q. Brown. Voronoi diagrams from convex hulls. *Inform. Process. Lett.*, 9:223–228, 1979.
- [31] C. Carathéodory. Über den variabilitätsbereich der koeffizienten von potenzreihen, die gegebene werte nicht annehmen. *Math. Ann.*, 64:95–115, 1907.
- [32] D. R. Chand and S. S. Kapur. An algorithm for convex polytopes. *J. ACM*, 17:78–86, 1970.

- [33] B. Chazelle. A theorem on polygon cutting with applications. In *23rd FOCS*, pages 339–349, 1982.
- [34] B. Chazelle. Filtering search: A new approach to query-answering. *SIAM J. Comput.*, 15:703–724, 1986.
- [35] B. Chazelle. Lower bounds on the complexity of multidimensional searching. In *27th FOCS*, pages 87–96, 1986.
- [36] B. Chazelle. Reporting and counting segment intersections. *J. Comput. Syst. Sci.*, 32:156–182, 1986.
- [37] B. Chazelle. A functional approach to data structures and its use in multi-dimensional searching. *SIAM J. Comput.*, 17:427–462, 1988.
- [38] B. Chazelle. Lower bounds for orthogonal range searching, I: the reporting case. *J. ACM*, 37:200–212, 1990.
- [39] B. Chazelle. Lower bounds for orthogonal range searching, II: the arithmetic model. *J. ACM*, 37:439–463, 1990.
- [40] B. Chazelle. An optimal convex hull algorithm and new results on cuttings. In *32nd FOCS*, pages 29–38, 1991.
- [41] B. Chazelle. Triangulating a simple polygon in linear time. *DCG*, 6:485–524, 1991.
- [42] B. Chazelle. An optimal algorithm for intersecting three-dimensional convex polytopes. *SIAM J. Comput.*, 21(4):671–696, 1992.
- [43] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *J. ACM*, 39:1–54, 1992.
- [44] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir. A singly-exponential stratification scheme for real semi-algebraic varieties and its applications. *Theoret. Comput. Sci.*, 84:77–105, 1991.
- [45] B. Chazelle, H. Edelsbrunner, L. J. Guibas, and M. Sharir. Diameter, width, closest line pair, and parametric searching. In *8th SCG*, pages 120–129, 1992.
- [46] B. Chazelle, H. Edelsbrunner, L. J. Guibas, M. Sharir, and J. Snoeyink. Computing a face in an arrangement of line segments. In *2nd SODA*, pages 441–448, 1991.
- [47] B. Chazelle and J. Friedman. Point location among hyperplanes and vertical ray shooting. *Comput. Geom. Theory Appl.* To appear.
- [48] B. Chazelle and J. Friedman. A deterministic view of random sampling and its use in geometry. *Combinatorica*, 10(3):229–249, 1990.
- [49] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.
- [50] B. Chazelle and L. J. Guibas. Fractional cascading: II. Applications. *Algorithmica*, 1:163–191, 1986.
- [51] B. Chazelle, L. J. Guibas, and D. T. Lee. The power of geometric duality. *BIT*, 25:76–90, 1985.
- [52] B. Chazelle and J. Icerpi. Triangulation and shape-complexity. *ACM Trans. Graph.*, 3(2):135–152, 1984.
- [53] B. Chazelle and J. Matoušek. Derandomizing an output-sensitive convex hull algorithm in three dimensions. Kam series in discrete mathematics 91-209, Charles University, Prague, 1991.
- [54] B. Chazelle and J. Matoušek. On linear-time deterministic algorithms for optimization problems in fixed dimension. To appear in 4th SODA, 1993.

- [55] B. Chazelle and F. P. Preparata. Halfspace range searching: an algorithmic application of k -sets. *DCG*, 1:83–93, 1986.
- [56] B. Chazelle, M. Sharir, and E. Welzl. Quasi-optimal upper bounds for simplex range searching and new zone theorems. *Algorithmica*, 8:407–429, 1992.
- [57] B. Chazelle and E. Welzl. Quasi-optimal range searching in spaces of finite VC-dimension. *DCG*, 4:467–489, 1989.
- [58] S. W. Cheng and R. Janardan. New results on dynamic planar point location. In *31st FOCS*, pages 96–105, 1990.
- [59] D. Cheriton and R. E. Tarjan. Finding minimum spanning trees. *SIAM J. Comput.*, 5:724–742, 1976.
- [60] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Math. Statistics*, 23:493–507, 1952.
- [61] L. P. Chew. Building Voronoi diagrams for convex polygons in linear expected time. Report, Department of Mathematical and Computer Science, Dartmouth College, Hanover, NH, 1985.
- [62] Y.-J. Chiang and R. Tamassia. Dynamization of the trapezoid method for planar point location. In *7th SCG*, pages 61–70, 1991.
- [63] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proc. IEEE*, 80(9), 1992.
- [64] B. Chor and O. Goldreich. The power of two-point sampling. To appear in *J. Complexity*.
- [65] K. Clarkson, H. Edelsbrunner, L. Guibas, M. Sharir, and E. Welzl. Combinatorial complexity bounds for arrangements of curves and spheres. *DCG*, 5:99–160, 1990.
- [66] K. L. Clarkson. New applications of random sampling in computational geometry. *DCG*, 2:195–222, 1987.
- [67] K. L. Clarkson. A Las Vegas algorithm for linear programming when the dimension is small. In *29th FOCS*, pages 452–456, 1988.
- [68] K. L. Clarkson. A randomized algorithm for closest-point queries. *SIAM J. Comput.*, 17:830–847, 1988.
- [69] K. L. Clarkson, R. Cole, and R. E. Tarjan. Randomized parallel algorithms for trapezoidal diagrams. In *7th SCG*, pages 152–161, 1991.
- [70] K. L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. In *Proceedings of the 9th Symposium on Theoretical Aspects of Computer Science*, volume 577 of *Lecture Notes in Computer Science*, pages 463–474. Springer-Verlag, 1992.
- [71] K. L. Clarkson and P. W. Shor. Algorithms for diametral pairs and convex hulls that are optimal, randomized, and incremental. In *4th SCG*, pages 12–17, 1988.
- [72] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *DCG*, 4:387–421, 1989.
- [73] K. L. Clarkson, R. E. Tarjan, and C. J. Van Wyk. A fast Las Vegas algorithm for triangulating a simple polygon. *DCG*, 4:423–432, 1989.
- [74] R. Cole. Slowing down sorting networks to obtain faster sorting algorithms. *J. ACM*, 34:200–208, 1987.
- [75] R. Cole, M. Sharir, and C. K. Yap. On k -hulls and related problems. *SIAM J. Comput.*, 16:61–77, 1987.

- [76] M. de Berg, D. Halperin, M. H. Overmars, J. Snoeyink, and M. van Kreveld. Efficient ray shooting and hidden surface removal. In *7th SCG*, pages 21–30, 1991.
- [77] M. de Berg and M. Overmars. Hidden surface removal for c -oriented polyhedra. *Comput. Geom. Theory Appl.*, 1(5):247–268, 1992.
- [78] M. Dehn. Die Eulersche Formel in Zusammenhang mit dem Inhalt in der nicht-Euklidischen Geometrie. *Math. Ann.*, 61:561–586, 1905.
- [79] B. Delaunay. Sur la sphère vide. *Bull. Acad. Sci. USSR VII: Class. Sci. Mat. Nat.*, 7:793–800, 1934.
- [80] O. Devillers. Randomization yields simple $O(n \log^* n)$ algorithms for difficult $\Omega(n)$ problems. *Internat. J. Comput. Geom. Appl.*, 2(1):621–635, 1992.
- [81] O. Devillers, S. Meiser, and M. Teillaud. Fully dynamic Delaunay triangulation in logarithmic expected time per operation. In *Proceedings of the 2nd Workshop on Algorithms and Data Structures*, volume 519 of *Lecture Notes in Computer Science*, pages 42–53. Springer-Verlag, 1991.
- [82] M. Dillencourt, D. Mount, and N. Netanyahu. A randomized algorithm for slope selection. In *Proc. 3rd Canad. Conf. on Comput. Geom*, pages 135–140, 1991.
- [83] D. P. Dobkin and D. Silver. Applied computational geometry: Towards robust solutions of basic problems. *J. Comput. Syst. Sci.*, 40:70–87, 1989.
- [84] R. A. Dwyer. Higher-dimensional Voronoi diagrams in linear expected time. *DCG*, 6:343–367, 1991.
- [85] M. E. Dyer and A. M. Frieze. A randomized algorithm for fixed-dimension linear programming. Unpublished manuscript, 1987.
- [86] H. Edelsbrunner. A new approach to rectangle intersections, Part I. *Internat. J. Comput. Math.*, 13:209–219, 1983.
- [87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, Heidelberg, 1987.
- [88] H. Edelsbrunner, L. J. Guibas, and M. Sharir. The complexity of many cells in arrangements of planes and related problems. *DCG*, 5:197–216, 1990.
- [89] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Computing*, 15:317–340, 1986.
- [90] H. Edelsbrunner and H. A. Maurer. On the intersection of orthogonal objects. *Inform. Process. Lett.*, 13:177–181, 1981.
- [91] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9:66–104, 1990.
- [92] H. Edelsbrunner, J. O'Rourke, and R. Seidel. Constructing arrangements of lines and hyperplanes with applications. *SIAM J. Computing*, 15:341–363, 1986.
- [93] H. Edelsbrunner and R. Seidel. Voronoi diagrams and arrangements. *DCG*, 1:25–44, 1986.
- [94] H. Edelsbrunner, R. Seidel, and M. Sharir. On the zone theorem for hyperplane arrangements. In H. Maurer, editor, *New Results and New Trends in Computer Science*, volume 555 of *Lecture Notes in Computer Science*, pages 108–123. Springer-Verlag, 1991.
- [95] H. Edelsbrunner and W. Shi. An $O(n \log^2 h)$ time algorithm for the three-dimensional convex hull problem. *SIAM J. Comput.*, 20:259–277, 1991.

- [96] H. Edelsbrunner and E. Welzl. On the number of line separations of a finite set in the plane. *J. Combin. Theory, Ser. A*, 38:15–29, 1985.
- [97] H. Edelsbrunner and E. Welzl. Halfplanar range search in linear space and $O(n^{0.695})$ query time. *Inform. Process. Lett.*, 23(6):289–293, 1986.
- [98] H. Edelsbrunner and E. Welzl. On the maximal number of edges of many faces in an arrangement. *J. Combin. Theory, Ser. A*, 41:159–166, 1986.
- [99] I. Emiris and J. Canny. An efficient approach to removing geometric degeneracies. In *8th SCG*, pages 74–82, 1992.
- [100] D. Eppstein. Dynamic three-dimensional linear programming. In *32nd FOCS*, pages 488–494, 1991.
- [101] P. Erdős, L. Lovász, A. Simmons, and E. Straus. Dissection graphs of planar point sets. In J. N. Srivastava, editor, *A Survey of Combinatorial Theory*, pages 139–154. North-Holland, Amsterdam, 1973.
- [102] J. D. Foley and A. Van Dam. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, Reading, MA, 1982.
- [103] S. J. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.
- [104] S. J. Fortune. Numerical stability of algorithms for 2-d Delaunay triangulations and Voronoi diagrams. In *8th SCG*, pages 83–92, 1992.
- [105] M. L. Fredman. A lower bound on the complexity of orthogonal range queries. *J. ACM*, 28:696–705, 1981.
- [106] O. Fries, K. Mehlhorn, and S. Näher. Dynamization of geometric data structures. In *1st SCG*, pages 168–176, 1985.
- [107] H. Fuchs, G. Abram, and E. Grant. Near real-time shaded display of rigid objects. *Comput. Graph.*, 17(3), 1983.
- [108] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Comput. Graph.*, 14(3):124–133, 1980.
- [109] D. Gale. On convex polyhedra. Abstract 794. *Bull. Amer. Math. Soc.*, 61:556, 1955.
- [110] M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan. Triangulating a simple polygon. *Inform. Process. Lett.*, 7:175–179, 1978.
- [111] M. Golin, R. Raman, C. Schwarz, and M. Smid. Randomized data structures for the dynamic closet-pair problem. In *3rd SODA*, 1992.
- [112] I. Goodman and R. Pollack. On the number of k -subsets of a set of n points in the plane. *J. Combin. Theory, Ser. A*, 36:101–104, 1984.
- [113] M. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. In *23rd STOC*, pages 523–533, 1991.
- [114] R. L. Graham. An efficient algorithm for constructing the convex hull of a finite planar set. *Inform. Process. Lett.*, 1:132–133, 1972.
- [115] B. Grünbaum. *Convex Polytopes*. John Wiley & Sons, New York, 1967.
- [116] B. Grünbaum. Arrangements of hyperplanes. *Congr. Numer.*, 3:41–106, 1971.
- [117] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–413, 1992.
- [118] L. J. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: building robust algorithms from imprecise computations. In *5th SCG*, pages 208–217, 1989.
- [119] D. Haussler and E. Welzl. Epsilon-nets and simplex range queries. *DCG*, 2:127–151, 1987.

- [120] C. A. R. Hoare. Quicksort. *Comput. J.*, 5:13–28, 1962.
- [121] C. M. Hoffmann, J. E. Hopcroft, and M. S. Karasick. Towards implementing robust geometric computations. In *4th SCG*, pages 106–117, 1988.
- [122] G. Kalai. A subexponential randomized simplex algorithm. In *24th STOC*, pages 475–482, 1992.
- [123] H. J. Karloff and P. Raghavan. Randomized algorithms and pseudorandom numbers. In *20th STOC*, pages 310–321, 1988.
- [124] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.
- [125] L. G. Khachiyan. Polynomial algorithms in linear programming. *U.S.S.R. Comput. Math. and Math. Phys.*, 20:53–72, 1980.
- [126] S. Khuller and Y. Matias. A simple randomized sieve algorithm for the closest-pair problem. In *Proceedings of the 3rd Canadian Conference on Computational Geometry*, pages 130–134, 1991.
- [127] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12:28–35, 1983.
- [128] D. G. Kirkpatrick, M. M. Klawe, and R. E. Tarjan. Polygon triangulation in $O(n \log \log n)$ time with simple data structures. In *6th SCG*, pages 34–43, 1990.
- [129] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM J. Comput.*, 15:287–299, 1986.
- [130] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 1968.
- [131] J. Komlós, J. Pach, and G. Wöginger. Almost tight bounds for ϵ -nets. *DCG*, 7:163–173, 1992.
- [132] D. T. Lee. On k -nearest neighbour Voronoi diagrams in the plane. *IEEE Trans. Comput.*, C-31:478–487, 1982.
- [133] R. J. Lipton and R. E. Tarjan. Applications of a planar separator theorem. *SIAM J. Comput.*, 9:615–627, 1980.
- [134] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15:1036–1053, 1986.
- [135] G. S. Lueker. A data structure for orthogonal range queries. *Inform. Process. Lett.*, 15(5):209–213, 1982.
- [136] G. S. Lueker and D. E. Willard. A data structure for dynamic range searching. *Inform. Process. Lett.*, 15:209–213, 1982.
- [137] J. Matoušek. Construction of ϵ -nets. *DCG*, 5:427–448, 1990.
- [138] J. Matoušek. Cutting hyperplane arrangements. *DCG*, 6:385–406, 1991.
- [139] J. Matoušek. Randomized optimal algorithm for slope selection. *Inform. Process. Lett.*, 39:183–187, 1991.
- [140] J. Matoušek. Range searching with efficient hierarchical cuttings. In *8th SCG*, pages 276–285, 1992.
- [141] J. Matoušek. Reporting points in halfspaces. *Computational Geometry: Theory and Applications*, 2(3):169–186, 1992.
- [142] J. Matoušek and O. Schwarzkopf. Linear optimization queries. In *8th SCG*, pages 16–25, 1992.
- [143] J. Matoušek, R. Seidel, and E. Welzl. How to net a lot with little: Small ϵ -nets for disks and halfspaces. In *6th SCG*, pages 16–22, 1990.

- [144] J. Matoušek, M. Sharir, and E. Welzl. A subexponential bound for linear programming. In *8th SCG*, pages 1–8, 1992.
- [145] J. Matoušek, E. Welzl, and L. Wernisch. Discrepancy and ϵ -approximations for bounded VC-dimension. In *32nd FOCS*, pages 424–430, 1991.
- [146] J. Matoušek. Approximations and optimal geometric divide-and-conquer. In *23rd STOC*, pages 505–511, 1991.
- [147] J. Matoušek. Efficient partition trees. *DCG*, 8:315–334, 1992.
- [148] H. A. Maurer and T. A. Ottmann. Dynamic solutions of decomposable searching problems. In U. Pape, editor, *Discrete Structures and Algorithms*, pages 17–24. Carl Hanser Verlag, Munich, 1979.
- [149] D. McCallum and D. Avis. A linear algorithm for finding the convex hull of a simple polygon. *Inform. Process. Lett.*, 9:201–206, 1979.
- [150] E. M. McCreight. Priority search trees. *SIAM J. Comput.*, 14:257–276, 1985.
- [151] P. McMullen. The maximum number of faces of a convex polytope. *Mathematika*, 17:179–184, 1970.
- [152] P. McMullen and D. Walkup. A generalized lower-bound conjecture for simplicial polytopes. *Mathematika*, 18:246–273, 1971.
- [153] N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *J. ACM*, 30:852–865, 1983.
- [154] N. Megiddo. Linear programming in linear time when the dimension is fixed. *J. ACM*, 31:114–127, 1984.
- [155] K. Mehlhorn. *Multi-dimensional Searching and Computational Geometry*, volume 3 of *Data Structures and Algorithms*. Springer-Verlag, New York, 1985.
- [156] K. Mehlhorn, S. Meiser, and C. Ó'Dúnlaing. On the construction of abstract Voronoi diagrams. *DCG*, 6:211–224, 1991.
- [157] K. Mehlhorn and S. Näher. Bounded ordered dictionaries in $O(\log \log n)$ time and $O(n)$ space. *Inform. Process. Lett.*, 35:183–189, 1990.
- [158] K. Mehlhorn, M. Sharir, and E. Welzl. Tail estimates for the space complexity of randomized incremental algorithms. In *3rd SODA*, pages 89–93, 1992.
- [159] V. Milenkovic. Double precision geometry: A general technique for calculating line and segment intersections using rounded arithmetic. In *30th FOCS*, pages 500–505, 1989.
- [160] K. Mulmuley. A fast planar partition algorithm, I. In *29th FOCS*, pages 580–589, 1988.
- [161] K. Mulmuley. An efficient algorithm for hidden surface removal. *Comput. Graph. (Proc. ACM SIGGRAPH '89)*, 23(3):379–388, 1989. Full version to appear in the special issue of *J. of Algorithms* on the 30th FOCS.
- [162] K. Mulmuley. Output sensitive construction of levels and Voronoi diagrams in R^d of order 1 to k . In *22nd STOC*, pages 322–330, 1990.
- [163] K. Mulmuley. A fast planar partition algorithm, II. *J. ACM*, 38:74–103, 1991.
- [164] K. Mulmuley. Hidden surface removal with respect to a moving point. In *23rd STOC*, pages 512–522, 1991.
- [165] K. Mulmuley. On levels in arrangements and Voronoi diagrams. *DCG*, 6:307–338, 1991.
- [166] K. Mulmuley. Randomized multidimensional search trees: Dynamic sampling. In *7th SCG*, pages 121–131, 1991.

- [167] K. Mulmuley. Randomized multidimensional search trees: Further results in dynamic sampling. In *32nd FOCS*, pages 216–227, 1991.
- [168] K. Mulmuley. Randomized multidimensional search trees: Lazy balancing and dynamic shuffling. In *32nd FOCS*, pages 180–196, 1991.
- [169] K. Mulmuley. Randomized geometric algorithms and pseudo-random generators. In *33rd FOCS*, pages 90–100, 1992.
- [170] K. Mulmuley. Dehn-Sommerville relations, upper bound theorem, and levels in arrangements. To appear in *SCG*, 1993.
- [171] K. Mulmuley. A generalization of Dehn-Sommerville relations to simple stratified spaces. *DCG*, 9:47–55, 1993.
- [172] K. Mulmuley and S. Sen. Dynamic point location in arrangements of hyperplanes. *DCG*, 8:335–360, 1992.
- [173] J. Naor and M. Naor. Small-bias probability spaces: Efficient constructions and applications. In *22nd STOC*, pages 213–223, 1990.
- [174] J. O'Rourke. *Computational geometry in C*. Cambridge Univ. Press, 1993.
- [175] M. H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer-Verlag, 1983.
- [176] M. H. Overmars. Range searching in a set of line segments. In *1st SCG*, pages 177–185, 1985.
- [177] M. H. Overmars and M. Sharir. Output-sensitive hidden surface removal. In *30th FOCS*, pages 598–603, 1989.
- [178] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.*, 23:166–204, 1981.
- [179] M. H. Overmars and J. van Leeuwen. Two general methods for dynamizing decomposable searching problems. *Computing*, 26:155–166, 1981.
- [180] J. Pach and P. Agarwal. Combinatorial geometry. Report 91-51, DIMACS, Rutgers University, New Brunswick, 1991.
- [181] J. Pach, W. Steiger, and E. Szemerédi. An upper bound on the number of planar k -sets. *DCG*, 7:109–123, 1992.
- [182] M. S. Paterson and F. F. Yao. Point retrieval for polygons. *J. Algorithms*, 7:441–447, 1986.
- [183] M. S. Paterson and F. F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *DCG*, 5:485–503, 1990.
- [184] F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Commun. ACM*, 20:87–93, 1977.
- [185] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.
- [186] F. P. Preparata and R. Tamassia. Fully dynamic point location in a monotone subdivision. *SIAM J. Comput.*, 18:811–830, 1989.
- [187] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.
- [188] M. O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity*, pages 21–30. Academic Press, New York, 1976.
- [189] P. Raghavan. Probabilistic construction of deterministic algorithms: Approximating packing integer programs. *J. Comput. System Sci.*, 37:130–143, 1988.
- [190] H. Raynaud. Sur l'enveloppe convexe des nuages des points aléatoires dans R^n . *J. Appl. Probab.*, 7:35–48, 1970.

- [191] J. H. Reif and S. Sen. Optimal randomized parallel algorithms for computational geometry. In *Proceedings of the 16th International Conference on Parallel Processing*, 1987. Full version to appear in *Algorithmica*.
- [192] J. H. Reif and S. Sen. An efficient output-sensitive hidden-surface removal algorithm and its parallelization. In *4th SCG*, pages 193–200, 1988.
- [193] J. H. Reif and S. Sen. Polling: A new randomized sampling technique for computational geometry. In *21st STOC*, pages 394–404, 1989.
- [194] R. Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM J. Comput.*, 14:396–409, 1985.
- [195] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM J. Comput.*, 6:563–581, 1977.
- [196] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29:669–679, 1986.
- [197] H. W. Scholten and M. H. Overmars. General methods for adding range restrictions to decomposable searching problems. *J. Symbolic Comput.*, 7:1–10, 1989.
- [198] A. Schrijver. *Theory of Linear Programming*. John Wiley & Sons, New York, 1986.
- [199] J. T. Schwartz and M. Sharir. Algorithmic motion planning in robotics. In J. van Leeuwen, editor, *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*, pages 391–430. Elsevier, Amsterdam, 1990.
- [200] O. Schwarzkopf. Dynamic maintenance of geometric structures made easy. In *32nd FOCS*, pages 197–206, 1991.
- [201] O. Schwarzkopf. Ray shooting in convex polytopes. In *8th SCG*, pages 286–295, 1992.
- [202] R. Seidel. A convex hull algorithm optimal for point sets in even dimensions. Report 81/14, Department of Computer Science, University of British Columbia, Vancouver, 1981.
- [203] R. Seidel. The complexity of Voronoi diagrams in higher dimensions. In *Proceedings of the 20th Allerton Conf. Commun. Control Comput.*, pages 94–95, 1982.
- [204] R. Seidel. Constructing higher-dimensional convex hulls at logarithmic cost per face. In *18th STOC*, pages 404–413, 1986.
- [205] R. Seidel. Low dimensional linear programming and convex hulls made easy. *DCG*, 6:423–434, 1991.
- [206] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom. Theory Appl.*, 1:51–64, 1991.
- [207] R. Seidel. Backwards analysis of randomized geometric algorithms. Report TR-92-014, ICSI, Berkeley, 1992.
- [208] M. I. Shamos. *Computational Geometry*. PhD thesis, Yale University, 1978.
- [209] M. I. Shamos and D. Hoey. Closest-point problems. In *16th FOCS*, pages 151–162, 1975.
- [210] M. Smid. Range trees with slack parameter. *Algorithms Review*, 2:77–87, 1991.

- [211] D. M. Y. Sommerville. The relations connecting the angle-sums and volume of a polytope in space of n dimensions. *Proc. Roy. Soc. Lond.*, Ser. A, 115:103–119, 1927.
- [212] J. Spencer. Ten lectures on probabilistic method. In *CBMS-NSF, SIAM*, 1987.
- [213] J. M. Steele and A. C. Yao. Lower bounds for algebraic decision trees. *J. Algorithms*, 3:1–8, 1982.
- [214] G. F. Swart. Finding the convex hull facet by facet. *J. Algorithms*, 6:17–48, 1985.
- [215] R. E. Tarjan and C. J. Van Wyk. An $O(n \log \log n)$ -time algorithm for triangulating a simple polygon. *SIAM J. Comput.*, 17:143–178, 1988. Erratum in 17:106.
- [216] P. M. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbors problem. *DCG*, 4:101–115, 1989.
- [217] P. M. Vaidya. Space-time tradeoffs for orthogonal range queries. *SIAM J. Comput.*, 18:748–758, 1989.
- [218] V. K. Vaishnavi and D. Wood. Rectilinear line segment intersection, layered segment trees and dynamization. *J. Algorithms*, 3:160–176, 1982.
- [219] M. van Kreveld and M. H. Overmars. Concatenable structures for decomposable problems. Ruu-cs-89-16, Department of Computer Science, University Utrecht, Netherlands, 1989.
- [220] V. N. Vapnik and A. Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory Probab. Appl.*, 16:264–280, 1971.
- [221] V. N. Vapnik and A. Y. Chervonenkis. *The Theory of Pattern Recognition*. Nauka, Moscow, 1974.
- [222] M. G. Voronoi. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. *J. Reine Angew. Math.*, 134:198–287, 1908.
- [223] R. Wanocur and R. Dudley. Some special Vapnik-Chervonenkis classes. *Discrete Math.*, 33:313–318, 1981.
- [224] E. Welzl. More on k -sets of finite sets in the plane. *DCG*, 1:95–100, 1986.
- [225] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and New Trends in Computer Science*, volume 577 of *Lecture Notes in Computer Science*, pages 359–370. Springer-Verlag, 1991.
- [226] D. E. Willard. Polygon retrieval. *SIAM J. Comput.*, 11:149–165, 1982.
- [227] D. E. Willard. New data structures for orthogonal range queries. *SIAM J. Comput.*, 14:232–253, 1985.
- [228] A. C. Yao. On the complexity of maintaining partial sums. *SIAM J. Comput.*, 14:277–288, 1985.
- [229] A. C. Yao. Lower bounds for algebraic computation trees with integer inputs. In *30th FOCS*, pages 308–313, 1989.
- [230] A. C. Yao and F. F. Yao. A general approach to D -dimensional geometric queries. In *17th STOC*, pages 163–168, 1985.
- [231] F. F. Yao. Computational geometry. In J. van Leeuwen, editor, *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*, pages 343–390. Elsevier, Amsterdam, 1990.
- [232] C. K. Yap. A geometric consistency theorem for a symbolic perturbation scheme. *J. Comput. Syst. Sci.*, 40:2–18, 1989.

Index

- N -sequence, 83, 121
 (N, δ) -sequence, 127
 ϵ -approximation, 412
 ϵ -net, 215, 217
 d -cell, 27, 229
 h -matrix, 307
 h -vector, 268
 j -face, 27
 k -complex, 294
 k -level, 294
 k -set, 295
 k th order Voronoi diagram, 294
- activation of a configuration, 152
active configuration, 112
 over a subset, 114
adjacent face, 27
adjacent object to a configuration,
 111
amortization, 98
antecedent, 152
antenna, 242
arrangements
 bottom-up sampling, 185, 242
 dynamic (bottom-up) sampling,
 248
 dynamic (top-down) sampling,
 253
 dynamic point location in, 248,
 253
 incremental construction, 232, 276
 point location in, 242, 250
 top-down sampling, 250
arrangements of lines, 29
 incremental construction, 58
 point location in, 182
 top-down sampling, 182
- average conflict size, 197
- backwards analysis, 7
Bernoulli sampling, 180
bias of a coin, 18
binary space partition, 372
binomial distribution, 424
bottom-up sampling, 184, 185, 242
bounded valence, 178
bounded-degree property, 93, 181
bounding box trick, 97
bounding half-space, 260
bounding hyperplane, 38
- canonical covering list, 314
canonical subinterval, 313
canonical triangulation, 179, 238
cap, 158
cell (d -cell), 27
 of an arrangement, 29, 30, 229
 of a Voronoi diagram, 47
change, 202
Chebychev's inequality, 422
Chebychev's technique, 429
Chernoff technique, 423
child
 of an interval, 19
 of a trapezoid, 91, 133
 of a triangle, 76, 110
commutativity condition, 154
commute, 14, 154
conditional probabilities, method of,
 413
configuration, 111
 defining objects, 111
 destroyed, killed, deactivated, 141
 newly created, 141
configuration space, 111

- bounded dimension, 215
 - bounded valence, 176
 - of feasible racquets, 117
 - of feasible trapezoids, 112
 - of feasible vertices, 115
 - of feasible Voronoi vertices, 116
 - history, 140
 - conflict, 111
 - conflict change, 119
 - cth order, 120
 - total, 119
 - conflict list (set), 111
 - conflict search
 - expected cost, 142
 - in convex polytopes, 279
 - through linear programming, 278
 - in Voronoi diagrams, 137
 - conflict set (list), 111
 - conflict size, 111
 - average, 197
 - of a configuration, 114
 - absolute, 111
 - cth order, 197
 - relative to a set, 114
 - of an interval, 174
 - of a junction, 369
 - of a vertex, 101
 - conflicting nodes of history, 142
 - convex hull, 40, 56
 - convex polytope, 36, 37, 260
 - dynamic maintenance, 158, 284
 - history, 103, 279
 - randomized incremental
 - construction, 96
 - search problems, 286
 - convex set, 36
 - created (newly) configuration, 141
 - creator, 152
 - critical point on an algebraic curve, 36
 - critical time value, 62
 - cutting, 241
 - Cutting Lemma, 241
 - cyclic polytopes, 274
 - cylindrical partition
 - construction, 391
 - randomized incremental
 - construction, 392
- deactivated configuration, 141
 - dead object, 149
 - decomposable search problems, 345
 - defining objects of a configuration, 111
 - defining site, 108
 - degeneracies, 52, 53
 - degenerate arrangement, 30
 - degenerate configuration, 53
 - degenerate linear function, 39
 - degree of a configuration, 111
 - Dehn–Sommerville relation, 268
 - Delaunay sphere, 50
 - Delaunay triangulation, 49
 - deletion in history, 151
 - derandomization, 410
 - descendant, 152
 - descent pointer, in skip-list, 19
 - destroyed configuration, 118, 141
 - dimension of a configuration space, 215
 - duality, 40
 - dynamic maintenance
 - of convex polytopes, 158, 284
 - of trapezoidal decompositions, 129, 162
 - of Voronoi diagrams, 135
 - dynamic planar point location, 327
 - dynamic point location in
 - arrangements, 248, 253
 - dynamic problems, 27
 - dynamic range queries and
 - arrangements, 253, 348
 - dynamic ray shooting in
 - arrangements, 248
 - dynamic sampling, 192
 - for half-space range queries, 292
 - for nearest neighbor queries, 292
 - for point location in
 - arrangements, 193, 248
 - for point location in Voronoi
 - diagrams, 210, 253
 - for ray shooting, 292

- for ray shooting in arrangements, 248
- for trapezoidal decompositions, 204
- dynamic shuffling, 167, 168
- dynamization of decomposable search problems, 345
- edge (1-face), 27
- edge skeleton
 - of an arrangement, 230
 - of a convex polytope, 261
- end addition, 169
- end deletion, 153, 155
- enumerability assumption, 411
- event schedule, 63
- expected change, 202
- expected performance, 6, 79
- expected running time, 6
- expected search cost, 8
- exterior child, 110
- extremal point, 41
- face (j -face), 27
 - of an arrangement, 30, 229
 - of a convex polytope, 38
 - of a Voronoi diagram, 47
- face-length, 59
- face vector, 267
- facet of an arrangement, 229
- facet of a convex polytope, 38, 260
- facial lattice of an arrangement, 30, 229
- feasible junction, 368
- feasible labeled trapezoid, 370
- feasible trapezoid, 112
- feasible triangle, 116, 179
- feasible vertex edge, 116
- filtered update sequence, 150
- filtering search, 339
- fine partition, 330
- free cut, 378
- frontier list, 63
- general position assumption, 53
- geometric complex, 26
- geometric distribution, 426
- geometric partition, 26
- gradation, 19, 184
- half-space, 36, 290
 - emptiness, 290, 343
 - non-redundant, 38, 290
- half-space range, 216
- half-space range queries, 251, 289, 338
 - and dynamic sampling, 292
- half-space range search, 29
- handle of a racquet, 117
- harmonic distribution, 428
- hidden surface removal, 51
 - object space, 358
 - randomized incremental, 361
 - with respect to a moving viewpoint, 383
- higher order levels, construction, 301
- higher order Voronoi diagrams, 294
 - incremental construction, 301
- history
 - of configuration spaces, 140
 - conflicting nodes in, 142
 - of convex polytopes, 103, 279, 280
 - deletions in, 151
 - of quick-sort, 8
 - rebuilding, 149
 - of trapezoidal decomposition, 90
 - of Voronoi diagrams, 106
- hyperplane, 30
- incremental algorithm, 81
- in-degree of a vertex, 268
- independence (t -wise), 399
- independent vertices, 75
- interior child of triangle, 110
- interval in a skip list, 19
- killed configuration, 141
- killed node of history, 141
- killed trapezoid, 91, 133
- killed vertex
 - of a convex polytope, 103
 - of a Voronoi diagram, 138
- killer, 152
- level of a configuration, 111, 114

- level of a junction, 369
levels, 294
linear congruential generator, 398
linear programming, 38, 262
 in conflict search, 278
link structure, 141
lower bridge, 69
lower chain, 56
lower edge, 71
- Markov's inequality, 422
method of conditional probabilities, 413
model of randomness, 78
monotone chain, 94
moving viewpoint, 383
- nearest (k)-neighbor queries, 291
 dynamic sampling, 292
negative binomial distribution, 427
newly created configuration, 118
newly created node of history, 141
nondegenerate
 arrangement, 30, 230
 configuration, 53
 linear function, 263
nonextremal point, 41
nonredundant half-space, 38, 260
nonredundant hyperplane, 38
numerical precision, 52
- objective function, 39
objects defining a configuration, 111
on-line linear programming, 104
on-line problems, 27
opaque representation, 93
orthogonal intersection search, 311
 dynamic, 320
orthogonal object, 311
orthogonal range queries over
 segments in the plane, 322
orthogonal range search, 29, 311, 312
 dynamic, 320
output-sensitivity, 371
- parametric search, 349
parent interval, 19
- parent pointer, 183, 254
partially active configuration, 112
partition theorem, 332, 341
partition tree, 329
persistent search tree, 77
planar graph
 trapezoidal decomposition of, 94
 triangulation of, 94
planar point location, 34, 74, 132
point location
 in arrangements, 185, 193, 242, 250
 in trapezoidal decompositions, 162, 204
 in Voronoi diagrams, 210
polygons defining a junction, 369, 389
polytope (d -polytope), 37
possible trapezoid, 112
possible triangle, 179
primary skip list, 318
primary wall, 386
priority, 169
priority order, 359
 randomized binary trees, 11
pseudo-random numbers, 398
pseudo-random source, 399
- quasi-output-sensitive, 280, 281, 371
quick-sort, 4
 as randomized
 divide-and-conquer, 4
 as a randomized incremental
 algorithm, 5
- racquet, feasible, 117
radial tree, 110, 212
radial triangulation, 108
random N -sequence (of additions), 83
random (N, δ) -sequence, 127
random sampling, 119
 with replacement, 180
 without replacement, 173
random update, 129
randomized binary tree, 7
 dynamic, 11

- randomized, incremental algorithm, 81
- range, 28, 216
- range queries, 250
- range searching, 28, 311
 - and arrangements, 250
 - dynamic, 253
- range space, 215, 216
 - VC dimension, 221
- rank of a signature, 127
- ray shooting
 - in arrangements, 242
 - and dynamic sampling, 242, 292
 - vertical, 288
- real (RAM) model of computation, 52
- rebuilding history, 149
- reduced size, 112, 215
- redundant half-space, 38, 260
- relative conflict size of a configuration, 114
- restriction of a configuration, 113
- rotation, 12, 13, 153
- rounding errors, 54
- sampling
 - bottom-up, 184
 - dynamic, 192
 - top-down, 181
- scanning, 61
- secondary search structure, 318
- secondary wall, 387
- seed, 398, 400
- segment tree, randomized, 312
- semidynamic problems, 27
- shallow (k -shallow) hyperplane, 339, 341
- shattered set, 221
- shuffle, 11
- shuffling, dynamic, 167
- sibling of a trapezoid, 162
- signature, 126
- simple arrangement, 30, 230
- simple configurations, 53
- simple convex polytope, 38, 261
- simple polygon, 58
 - trapezoidal decomposition of, 94
- triangulation of, 94
- simple trapezoidal decomposition, 36
- simple Voronoi diagram, 50
- simplex range queries, 29, 252, 328, 348
 - dynamization, 348
- simplicial convex hull, 41
- simplicial Delaunay triangulation, 50
- simplicial partition, 330
- size
 - of an arrangement, 231
 - of a configuration space, 112
 - of a convex polytope, 38
 - of a face, 27, 229, 261
 - of a geometric complex or partition, 27
- skeleton (j -skeleton), 230
- skip list, 18
- standard deviation, 422
- static problems, 27
- stoppers of a configuration, 111
- structural change, 82, 119, 141
 - during a rotation, 155
 - total, 119
- subface, 27, 229
- subspace of a configuration space, 113
- tail estimates, 120, 422
- top-down sampling, 181
 - for arrangements, 250
 - for arrangements of lines, 182
- trapezoidal decomposition, 32, 33
 - dynamic maintenance, 129, 162
 - history, 90
 - point location in, 132, 162, 204
 - randomized incremental construction, 84
 - scanning, 61
- triangulation of a planar graph, 35
- triggers, 111
 - of a racquet, 118
 - of a trapezoid, 112
 - of a triangle, 116
 - of a vertex, 115
- trivial configuration space, 119

Upper Bound Theorem
asymptotic form, 271
exact form, 271
upper bridge, 69
upper chain, 56
upper convex hull, 41
upper convex polytope, 37
upper edge, 71

valence, bounded, 178
valence of configuration spaces, 176
variance, 422
VC dimension, 221
vertex, 27
 of an arrangement, 29, 30, 229
 of a convex polytope, 38
view map (graph), 359
view partition, 361
view zone, 384
viewpoint, moving, 383
visibility map (graph), 52
visibility polygon, 67
Voronoi diagram, 46, 285
 divide-and-conquer construction,
 67
 dynamic maintenance of, 135, 285
 randomized incremental
 construction, 106, 285
 upper bound theorem, 285
Voronoi region, 46

weakly monotonic signature, 127

zone, 59, 234
Zone Theorem, 59, 234

Computational Geometry

An Introduction Through Randomized Algorithms

Ketan Mulmuley

This textbook gives a concise and up-to-date introduction to computational geometry with emphasis on randomized algorithms. It begins with simple two-dimensional problems and their deterministic solutions and shifts to randomized solutions as the problems become more complex. Since the simplest solutions to a large number of problems in computational geometry are randomized, this approach should enable readers to get a wide perspective of the field quickly.

Another feature of randomized methods, which makes them ideal for the purpose of introduction, is that they are all based on a few basic principles which can be applied systematically to a large number of apparently dissimilar problems. This book strives to give an account that focuses on simplicity and unity of randomized algorithms as well as their depth.

The book is organized into two parts. In the first part, the basic principles are formulated and illustrated with the help of simple problems in planar geometry. Higher dimensional applications are given in the second part. Thus, this book is useful to beginning graduate students as well as to specialists.

PRENTICE HALL
Englewood Cliffs, NJ 07632

ISBN 0-13-336363-5



9 780133363630