THE UNIVERSITY OF CALGARY

Problems On a Set of Convex Objects

BY

Haihuai Chen

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

April, 1991

© Haihuai Chen 1991

National Library of Canada

Bibliothèque nationale du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-66846-6

Canada

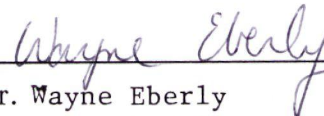
THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled *"Problems On a Set of Convex Objects"* submitted by Haihuai Chen in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor, Dr. Jon Rokne
Department of Computer Science

Dr. Bruce MacDonald
Department of Computer Science



Dr. Wayne Eberly
Department of Computer Science

Dr. Jun Gu
Department of Electrical Engineering

Date 1991-04-05

Abstract

This thesis studies several problems relating to sets of convex objects in the plane:

1. *Union Hull Construction*: Given a set S of k convex polygons, construct their union hull $U(S)$, i.e., the minimum convex hull that contains all the elements of S.

2. *Intersecting Convex Polygons*: Given a set S of k convex polygons, we study two problems: (1) Compute the intersection of these polygons, and (2) Detect whether the given polygons intersect or not.

3. *Separating Convex Objects*: Given a set S of k convex objects, decide whether a *separating line* exists, i.e., a straight line that separates S into two non-empty subsets and yet does not intersect any of the elements in S.

4. *Voronoi Diagram for Convex Objects*: This is an extended form of the ordinary Voronoi diagram construction problem. Instead of working on a set of given points, we construct the Voronoi diagram for a set of convex polygons.

The thesis studies these problems and some interesting results are achieved. Algorithmic lower bound analysis is given wherever possible, and some of algorithms presented here are optimal.

Acknowledgements

I wish to express my deep gratitude to my supervisor Dr. Jon Rokne for his continual encouragement and invaluable help ranging from topic selecting to the final thesis writing, without which the work presented here could never have been completed.

I also wish to thank Dr. Ian Witten who read and commented on my thesis proposal, and Dr. Wayne Eberly who rendered help when I was in need of it.

Lastly, but by no means the least, I thank my office mates Nick Malcolm, Zhao Zhang, my roommate Tom Fukushima, and other students of the department for creating a friendly, relaxed and stimulating environment in which I could concentrate on my research. My appreciation also goes to hardware and software support staff of the department who made it enjoyable to use the computing facilities.

Contents

Approval Page ii

Abstract iii

Acknowledgements iv

List of Figures vii

1 Introduction **1**

1.1 Preparation and Background 1

1.1.1 Computational geometry 1

1.1.2 Our computational model 3

1.1.3 Lower and upper bounds 3

1.1.4 Algorithm analysis 4

1.1.5 Algebraic decision tree 6

1.2 General definitions and notations 8

1.3 The thesis 9

2 Constructing the union hull **11**

2.1 Algorithms for union hull construction 12

2.1.1 The straight merge method 13

2.1.2 A divide-and-conquer solution 13

2.2 The Generalized Case 15

2.2.1 Divide-and-conquer vs. Huffman tree 16

2.2.2 A different solution 18

2.3 The lower bound 20

2.4 Another algorithm 22

2.5 Some comments 24

3 Intersecting Convex Polygons **26**

3.1 Preliminaries 27

3.1.1 Definitions and notations 27

3.1.2 Important results 28

3.2 Compute the Intersection of Convex Polygons 31

3.2.1 The lower bound 31

3.2.2 The algorithms 33

3.3 Detecting the Intersection of Convex Polygons 34

3.3.1 A linear algorithm 35
3.3.2 Reichling's algorithm 35
3.4 Using Newton Iteration 39
3.4.1 Intersecting a cup and a cap 39
3.4.2 The algorithm 43

4 Separating Convex Polygons **45**
4.1 A Comparison 46
4.1.1 Stabbing line 46
4.1.2 Separability vs. stabbing line 47
4.1.3 Some definitions 48
4.2 Separating line segments 49
4.2.1 Algorithm 1 49
4.2.2 A restricted problem 51
4.2.3 Algorithm 2 54
4.2.4 The potato peeling method 54
4.3 Lower Bound Analysis 57
4.4 Separating Convex Polygons 58
4.4.1 A straightforward solution 58
4.4.2 A solution based on convexity 58
4.4.3 An alternative solution 60

5 Voronoi Diagram for Convex Polygons **63**
5.1 Preliminaries 64
5.1.1 The Voronoi Diagram 64
5.1.2 Definitions and notations 66
5.2 Bisecting Two Line Segments 67
5.3 The Bisector of two convex polygons 69
5.4 Constructing the Voronoi Diagram 73
5.4.1 Inherent difficulties 74
5.4.2 Features of Voronoi regions 75
5.4.3 A direct solution 78
5.4.4 A divide-and-conquer solution 81
5.4.5 An interesting heuristic 84

6 Conclusion **87**

Bibliography **89**

List of Figures

2.1 The union hull of a convex m-gon and a convex n-gon can have up to $m+n$ vertices. 12
2.2 Different merging strategies makes difference. 17
2.3 Forming a large "interim hull". 19
2.4 Merging vertices on a parabola. 21
2.5 A preprocessing strategy. 24

3.1 The intersection of an infinite line with a convex polygon. 29
3.2 The intersection of a *cup* and a *cap*. 31
3.3 Setting the left bound b_l and the right bound b_r. 37
3.4 Tighten up the bounds – Reichling's algorithm. 38
3.5 Finding the intersection(s) of a cup and a cap. 41
3.6 The bounds b_l and b_r intersect CUP and CAP. 44

4.1 rays and wedges 49
4.2 separate via endpoints 51
4.3 Detect separability via plane coverage. 53
4.4 Separate via polygon vertices 59
4.5 Detect separability via plane coverage. 62

5.1 Voronoi polygon. 65
5.2 Voronoi diagram. 65
5.3 Bisecting a line segment and a point. 68
5.4 Bisecting two line segments that are (a): parallel, and (b): otherwise. 70
5.5 (a) The stripe of e for the convex polygon P. (b) Bisecting convex polygons P and Q. 72
5.6 (a) A Voronoi region is a connected area. (b) Adding another polygon into the picture. 77
5.7 (a) Any Voronoi region has at most $k-1$ edges. 79
5.8 S_1 and S_2 are separated by a single monotone chain. 83
5.9 An interesting correspondence. 86

Chapter 1

Introduction

This thesis investigates several problems relating to sets of convex objects in the plane.

The problems themselves are of theoretical and practical interest. By studying these problems more insight is obtained into the very important property of convexity. We believe, and prove by the work in the thesis, that algorithms handling convex objects can be made more efficient if more of the convexity of the input can be explored and utilized in the solution. We also show that algorithm design paradigms that are well known to be efficient in solving computational geometry problems, such as divide-and-conquer, geometrical transform, plane sweeping etc, can be applied to the problems.

1.1 Preparation and Background

We shall start with some background material and preliminaries.

1.1.1 Computational geometry

Computational Geometry is a new discipline that took shape over the past two decades. It is now established, as seen by the excellent advanced text by Preparata and Shamos [30], as well as the new journals that are appearing and the conferences (e.g. ACM Symposium on Computational Geometry) that are held annually.

The discipline is concerned with geometry and geometrical objects from a computational point of view, and the aim is to find efficient algorithms for these objects. The problems treated arise in many areas such as computer graphics, VLSI design, travel planning, and operation optimization etc, to name a few.

This discipline has seen a tremendous activity over the last two decades as witnessed by the bibliography [4] of materials relative to computational geometry. Initially the problems solved were fairly standard problems, such as the computation of the convex hull of a set of points in the plane. Many of these standard problems have now been solved in a satisfactory manner in that the complexities of the solutions are the complexities arrived at as best possible by complexity analysis. Thus, most of the standard problems now only admit refinement of detail in the computational process.

The research in the discipline also opened up new and exiting problem areas. The developments of new tools made it possible to consider problems that in the past were thought to be too difficult.

Problem solving in computational geometry consists of operation on some given set of geometrical objects, among which *convex objects* form the most prominent class. The reason that convex objects become a major focus in computational geometry research is that first, they characterize many real-world applications; second, associated with convex objects there is a very important property called *convexity*, which possesses some very valuable computational merits and enables efficient algorithms to be developed. In this thesis we shall study problems on planar convex objects.

1.1.2 Our computational model

The computational model we shall use in the thesis is an abstraction of an actual Von Neumann computer. In particular, we adopt a *random access machine* (RAM) similar to that described in [1], except that in our model each memory cell is able to hold a single real number. The following operations are primitive and cost unit time:

1. The arithmetic operations $(+,-,\times,/)$.
2. Comparisons between two real numbers.
3. Indirect addressing of memory (integer addresses only).

This model will be referred to as the *real RAM.* It closely reflects programs written in high-level languages such as PASCAL and ALGOL, in which real type variables are treated as having unlimited precision. This may cause implementational problems for applications where precise measurements are required, e.g., point positions, lengths, sizes, etc. In our discussion, however, we shall ignore questions such as how real numbers can be read or written in finite time.

1.1.3 Lower and upper bounds

When considering a problem under the real RAM model it is important to find out its inherent *computational complexity* (the number of operations performed). Given a problem P, its computational complexity may be assessed by establishing the lower bound or upper bound of the problem. The *lower bound* of a given problem is defined to be the minimum time required for running any algorithm that solves the problem

(usually measured by the worst-case running time). The *upper bound* of a problem, however, provides a ceiling for the convexity of the problem. It enables us to decide that the problem under study is "inherently not more difficult in term of computing time" than a given degree of complexity.

Lower bounds and upper bounds may be *tight*, in which case the measures of complexity they provide are accurate. They can also be *loose* , in which case they define a "range" as to how difficult the problem is, and the measures they provide are less precise.

An often used and proven effective method in establishing a lower bound (or upper bound) for a given problem is to find a relationship between the problem under study and another problem whose computation complexity is well known, such as *sorting*, *element uniqueness*, etc. This method is called *transformation of problems.* A detailed discussion of transformation of problems can be found in [30]. In many cases, however, tight lower and upper bounds may be extremely difficult to find.

1.1.4 Algorithm analysis

Solutions to problems are customarily presented in the form of a procedural description of their execution behavior. Such a description is called an *algorithm.* When an algorithm is given it should, wherever possible, be accompanied by an evaluation that predicts its performance in execution. *Algorithm performance evaluation* normally have two aspects: (a) the *expected space consumption*, and (b) more importantly, the *expected time requirement.*

The time used for the execution of an algorithm is the sum of the times of

the individual operations being executed. As the accurate running behavior of a certain algorithm may impossible to model, it is customary in the field of algorithm design and analysis to count only certain "key operations" that are executed. In our computational model, we consider only the following (algorithmic) operations:

1. Procedure calling.
2. Assignment.
3. Branching (comparing).
4. Looping.

Note that some operations are "omited", and the running time we come about will therefore account for only a part of the actual time requirement. This will cause no problem in lower bound analysis, for any unaccounted-for operations will only increase it. When dealing with upper bounds, however, we need to ensure that the selected operations accounts for a constant portion of all the operations that are executed. In our thesis, we shall use a notation device described in Knuth ([20]):

- $O(f(n))$ denotes the set of all functions $g(n)$ such that there exist positive constants C and N_0 with $|g(n)| \leq Cf(n)$ for all $n \geq N_0$.
- $\Omega(f(n))$ denotes the set of all functions $g(n)$ such that there exist positive constants C and N_0 with $g(n) \geq Cf(n)$ for all $n \geq N_0$.
- $\theta(f(n))$ denotes the set of all functions $g(n)$ such that there exist positive constants C_1, C_2 and N_0 with $C_1 f(n) \leq g(n) \leq C_2 f(n)$ for all $n \geq N_0$.

The *performance* of an algorithm can either be measured by its *worst-case complexity* or by its *average-case complexity*. Worst-case complexity is the maximum of

a measure of performance of a given algorithm over all problem instances of a given size. Average-case complexity, however, gives an estimate of the observed behavior of the algorithm. Normally worst-case complexity is easier to measure than its average-case counterpart. In this thesis we shall use both for performance evaluation purposes.

1.1.5 Algebraic decision tree

We now introduce a very important computational model, called the "algebraic decision tree", which will often be used in our complexity analysis.

An *algebraic decision tree* ([32], [10], [30]) on a set of variables $x_1, x_2, ..., x_n$ is a program with statements $L_1, L_2, ..., L_r$ of the form:

1. L_s: Compute $f(x_1, x_2, ..., x_n)$. If $f :: 0$ then go to L_i, else go to L_j ($::$ denotes any comparison relation).

2. L_x: Halt and return "Yes".

3. L_y: Halt and return "No".

In step 1, f is an *algebraic function* (a polynomial of a certain degree). The program is further assumed to be loop-free, i.e., it has the structure of a tree T, such that each nonleaf node v is described by

$$f_v(x_1, x_2, ..., x_n) :: 0.$$

where f_v is a polynomial in $x_1, x_2, ..., x_n$, and $::$ denotes a comparison relation. When f_v is a linear function we call the corresponding tree T a *linear decision tree.* The root of T represents the initial step of the computation and its leaves represent the

possible terminations and contain the possible answers. Without losing generality, we shall assume that the tree T is *binary*.

The major power of the algebraic decision tree model lies in its ability to solve the membership for decision problems. The decision problem $D(A)$ associated with a given problem A, which can either be a "computation" or a "subset selection" problem ([30]), is one that 1 (A being a computation problem): requests a Yes/No answer to a question of the type "Is $A \geq A_0$?" where A_0 is a constant and A is a parameter whose value is unknown; or 2 (A being a subset selection problem): requests a Yes/No answer to a question of the type "Does set S' satisfy property P?" where S' is a subset of a given set S and P is a certain property to be satisfied.

Let $D(A)$ be a decision problem and let $x_1, x_2, ..., x_n$ be the parameters associated with it. We can view each instance of the parameters as a point in the n-dimensional Euclidean space E^n. The decision problem then identifies a set of points $W \subseteq E^n$, or in other words it provides a Yes-answer if and only if $(x_1, x_2, ..., x_n) \in W$. If T is the decision tree of problem $D(A)$ we then say that T solves *the membership problem* for W.

Several results of significant theoretical value have been established regarding the above-mentioned membership problem ([10], [36], [5]). Following is a result due to Dobkin and Lipton ([10]):

Result 1 *Any linear decision tree algorithm that solves the membership problem in* $W \subseteq E^n$ *must have depth at least* $\log_2 \#(W)$*, where* $\#(W)$ *is the number of disjoint connected components of* W*.*

This result is very useful in establishing lower bounds for decision problems.

1.2 General definitions and notations

The objects contained in computational geometry are normally sets of points in Euclidean space. We shall assume a coordinate system of reference within which each point is represented as a vector of Cartesian coordinates of the appropriate dimension. Following are some general definitions and notations.

By E^d we denote the *d-dimensional Euclidean space*, i.e. the space of the d-tuples $(x_1, x_2, ..., x_d)$ of real numbers $x_i, i = 1, 2, ..., d$.

A *point* p in E^d is a d-tuple $(x_1, x_2, ..., x_d)$. A straight line l can be defined by any two different points on it, and a straight line segment s is defined by its two extreme points.

A domain D in E^d is *convex* if, for any two points q_1 and q_2 in D, the segment $\overline{q_1q_2}$ is entirely contained in D. Note that the intersection of two convex domains is still a convex domain.

The *convex hull* of a set of points S in E^d is the boundary of the smallest convex domain in E^d that contains S.

In E^2 a *polygon* is defined by a finite set of segments such that every segment extreme is shared by exactly two edges and no subset of edges has the same property. The segments are the *edges* and their extremes are the *vertices* of the polygon (Note that the number of edges and vertices are identical). An polygon with n vertices is called an n-gon.

A polygon P is *simple* if there is no pair of nonconsecutive edges sharing a point. A simple polygon partitions the plane into two disjoint regions, the *interior* (bounded) and the *exterior* (unbounded) separated by the polygon.

A polygon P is *convex* if its interior is a convex set.

1.3 The thesis

This thesis studies several basic problems on sets of convex objects, primarily convex polygons. It is organized as follows.

Chapter 2 discusses the construction of *union hulls*: given a set S of convex polygons, construct the convex hull of all vertices in S. A lower bound and several algorithms are presented.

Chapter 3 deals with the intersection problem. For a given set S of convex polygons we form two different types of intersection questions: (a) calculate the intersection of all convex polygons in S; and (b) detect whether the given convex polygons intersect or not. The chapter treats both problems in detail.

Chapter 4 studies the problem of separating a set of planar convex objects. Given a set S of convex objects, we are to find out whether there is a straight line l that separates S into two disjoint non-empty subsets and yet does not intersect any of the given polygons. Objects studied in this chapter are straight line segments and convex polygons.

Chapter 5 deals with an *extended* form of planar Voronoi Diagram. Ordinary Voronoi Diagrams are based on sets of planar points. In this chapter, however, we shall consider the problem of constructing the Voronoi diagram for a given set S of convex polygons. A solutions using the divide-and-conquer technique is given for a restricted version of the problem.

Finally chapter 6 gives a conclusion.

Throughout the thesis, we assume that polygons are represented in arrays with their vertices stored in counterclockwise order unless otherwise specified. This representation ensures that binary search can be performed when a vertex needs to be found (such cases arise when computing the intersection of two convex polygons).

Chapter 2

Constructing the union hull

In this chapter we study an extended form of convex hull construction which we shall call the *union hull*. By definition, the *convex hull* H for a set S of k planar points $p_1, p_2, ..., p_k$ is the minimum convex polygon that contains S ([15], [17], [30]). In a similar manner, the *union hull* for a set S of k convex polygons $P_1, P_2, ..., P_k$ is defined to be the minimum convex polygon that covers all the elements of S. We formalize the problem in the following:

Problem 1 *Given a set S of k planar convex n-gons $P_1, P_2, ...P_k$, construct the union hull U of S.*

The chapter is organized as follows: Section 2.1 studies the standard union hull construction problem where the input polygons all have the same number of vertices. A basic algorithm using the divide-and-conquer technique is given. Section 2.2 considers a generalized version of the problem where input polygons may differ in their numbers of vertices. An *Huffman tree model* is used to keep up with the changing input pattern thereby achieving a better worst-case as well as average-case performance. Section 2.3 establishes the lower bound for the union hull construction problem, and lastly in section 2.4 we introduce a general preprocessing method that is useful in solving computational geometry problems of a similar nature.

In the discussion below, by a "polygon" we mean a "convex polygon" unless otherwise specified.

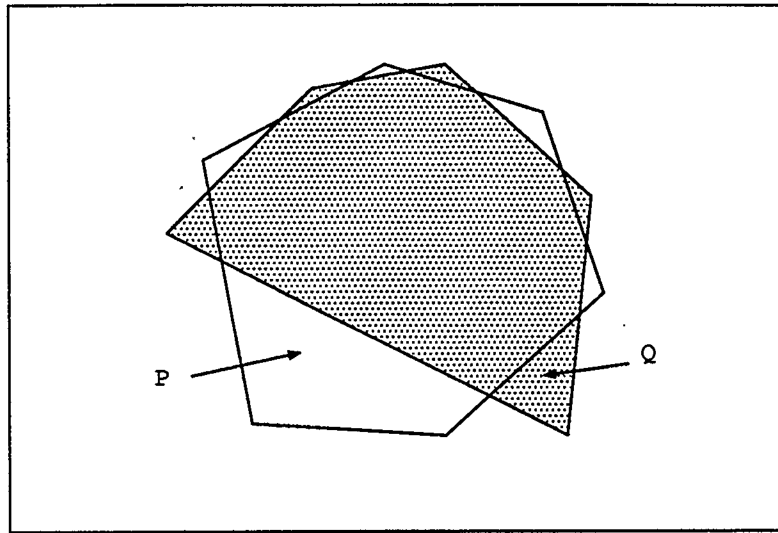


Figure 2.1: The union hull of a convex m-gon and a convex n-gon can have up to $m+n$ vertices.

2.1 Algorithms for union hull construction

In this chapter, we shall use *convex merge* (or *merge*), denoted by $\oplus$, to indicate the basic operation that computes the union hull of *two* convex polygons ([29], [30]). The following result on convex merge is of fundamental importance.

Result 2 *Finding the merge of a convex m-gon P and a convex n-gon Q requires* $O(m+n)$ *time, which is optimal in the worst case.*

Proof: There are several algorithms available that finds the merge of two convex polygons in linear time ([29], [30]), and we shall not give the details here. For optimality, it is obvious that the union hull of a convex m-gon and a convex n-gon may have up to $m+n$ vertices (figure 2.1), and reading them off takes $O(m+n)$ time. □

2.1.1 The straight merge method

Problem 1 can be solved by repetitively applying the convex merge operation. In fact, given k convex polygons $P_1, P_2, ..., P_k$, we can construct their union hull U by performing a sequence of $k-1$ merges. Let $U_1 = P_1$, the computation will proceed as follows:

$$U_1 \oplus P_2 = U_2; \quad (2.1)$$

......

$$U_{i-1} \oplus P_i = U_i; \quad (2.2)$$

......

$$U_{k-1} \oplus P_k = U; \quad (2.3)$$

we know that the union hull of a convex m-gon and a convex n-gon may have up to $m+n$ vertices, thus the ith $(1 \leq i \leq k-1)$ step of the above process (equation 2.2) will merge a (ni)-gon and a n-gon in the worst case. According to result 2 this step alone will take $O(ni)$ time. Apply this to all equations in the sequence for $i = 1, 2, ..., k$, we get the following total running time:

$$O(\sum_{i=1}^{k-1}(i \cdot n)) = O(nk^2)$$

Therefore the algorithm has an $O(nk^2)$ time complexity in the worst case.

2.1.2 A divide-and-conquer solution

The execution of the above algorithm can be characterized by a binary tree T which is constructed as follows: Initially the set S has k convex polygons $e_1, e_2, ..., e_k$, where each of which is assigned to be a "leaf node" of T. We then start to perform the

operation $\oplus$ on the elements of S. As the merging process proceeds, the contents and structure of S are accordingly changed.

When two elements of S, say e_i and e_j, are merged, we remove both from the set of S, form a new element $e = e_i \oplus e_j$ with arcs connected to both e_i and e_j. This new "parent node" e is then added to the now revised set of S. Every time a merge operation is performed, S loses two of its old nodes (e_i and e_j) and acquires a new node (e). After $(k-1)$ merges, the set of S should contain only one element e_0, which is the root node of the binary tree of T, and the construction process is completed.

We shall call a binary tree T obtained from the above process an *execution tree.* The following facts are easily observed about execution trees:

- A given set S of convex polygons may have many execution trees.
- Different merging strategies (for a given set S of convex polygons) individually define different execution trees, and vice versa. Thus an execution tree can fully determine the actual running time under its corresponding merging strategy.
- The running time $T(k)$ represented by an execution tree T can be calculated by summing up the depths of all its k leaf nodes, i.e.,

  $$T(k) = \sum_{i=1}^{k} d_i$$

  where the *depth* d_i of node e_i is defined to be the number of nodes on the path from the root node e_0 to e_i.

It is now clear that if the operation $\oplus$ is used as an atomic operation (we do not seem to have other choice), an optimal solution for problem 1 can be obtained

by finding the execution tree of the set of S whose leaf nodes have the minimum total depths. This can be done by invoking the *principle of balancing* ([1]), which immediately leads to the following divide-and-conquer algorithm:

1. Divide the original set S of k convex polygons into two subsets S_1 and S_2 of approximately equal sizes.

2. Compute the union hulls U_1 of S_1 and U_2 of S_2 recursively.

3. Merge U_1 and U_2 to obtain U.

Let $T(k)$ denote the time required to solve problem 1 using the divide-and-conquer algorithm, we have the following familiar divide-and-conquer formula:

$$T(k) = 2T(k/2) + t \tag{2.4}$$

where t is the time needed to merge the subsets S_1 and S_2. In the worst case both S_1 and S_2 can have $(nk/2)$ vertices, hence $t = O(nk)$ (result 2). Combine this and equation 2.4 we get the following:

$$T(k) = O(nk \log k). \tag{2.5}$$

which is the worst-case time complexity of the algorithm. We shall show later that this is also optimal for problem 1 (section 2.3).

2.2 The Generalized Case

We now consider a generalized version of problem 1:

Problem 2 *Given a set S of k planar convex polygons $P_1, P_2, \ldots, P_k$, with sizes $n_1, n_2, \ldots, n_k$ respectively, construct the union hull U of S.*

Problem 2 has been studied, either explicitly or implicitly, by a number of authors ([30], [28]). The major difference as compared with problem 1 is that here we allow for a more diversified input pattern for the given convex polygons. This difference, as we shall see later, creats a new situation which calls for a more careful study of the merging strategy, in order to keep up with the different input patterns.

2.2.1 Divide-and-conquer vs. Huffman tree

First consider the divide-and-conquer algorithm mentioned above: (1) Divide the initial set S of k convex polygons into two subsets S_1 and S_2 of approximately equal sizes. (2) Compute the unions U_1 of S_1 and U_2 of S_2 recursively. (3) Merge U_1 and U_2 to obtain U. Let $T(k)$ be the time required to solve problem 2 using this algorithm, then using a similar analysis as shown in section 2.1.2 we have the following worst-case time complexity of the algorithm:

$$T(k) = O((\sum_{i=1}^{k} n_i) \log k).$$

This algorithm is simple, but it fails to utilize important information contained in the input data. Indeed, the fact that the input polygons are different in sizes is not considered at all. This rigidness sometimes brings about redundancy into the computation process, and consequently the algorithm suffers from a loss of efficiency.

Consider, for example, figure 2.2 where we are to compute the union hull of three given convex polygons A, B, and C. Two merges operations have to be performed. Compare the following strategies of merging:

- First merge A and B, then take their union hull, say D, to merge with C.
- First merge A and C, then take their union hull, say E, to merge with B.

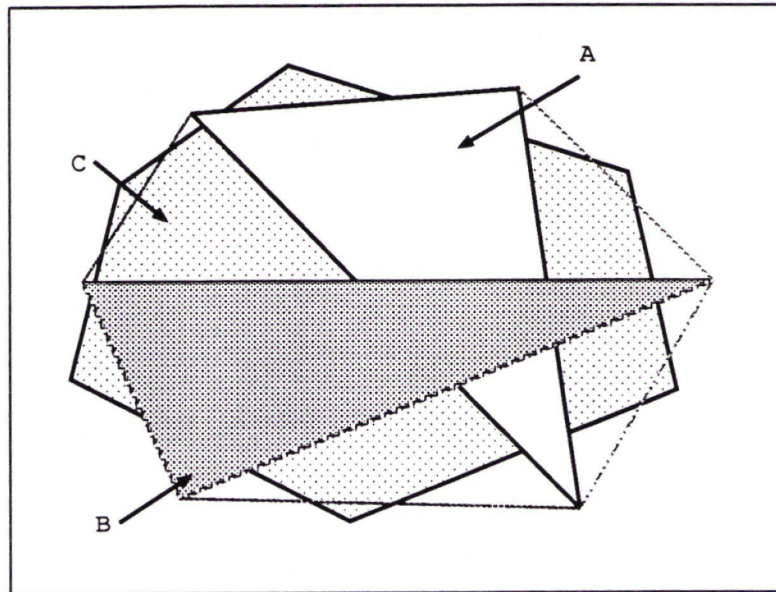


Figure 2.2: Different merging strategies makes difference.

The first strategy works better in the obvious way. It introduces the polygon that has the maximum number of vertices (which is C in this case) into the computation process at a later stage. We wish to design an algorithm such that when an input like this is given, our algorithm will be able to select a correct merging strategy. This observation leads to another solution that is based on the Huffman tree model.

By definition the *Huffman tree* $HT(S)$ for a set S of *weighted elements* $e_1, e_2, ..., e_k$ is a binary tree constructed as follows ([16]):

1. The given elements are assigned to be terminal nodes of $HT(S)$. Initially all nodes are "unmarked".

2. Find the two nodes with the smallest weights and mark them. Add a new node, with arcs to each of the nodes just marked. Set the weight of the new node to the sum of the weights of the it is connected to.

3. Repeat step 2 until all nodes are marked. The last node that is marked is the root node.

Under the Huffman tree model, the time required to compute the the union hull of a given set S of convex polygons is accurately reflected by the weight of the root node of $HT(S)$, plus the time used to construct the Huffman tree. When all the weights of the elements in S (in our case the number of vertices of the polygons in S) are known, it takes $O(k \log k)$ time to construct $HT(S)$ ([16]), which is negligible.

The divide-and-conquer algorithm always has a balanced execution tree. It provides an optimal solution to problem 1 where all the elements of S carry the same weight, but it fails to do the same for problem 2. The Huffman tree method, however, constructs the union hull in a more flexible way. It balances the "weights", rather than the "depths" of its subtrees ([18]). Although it has the same worst-case time complexity as its divide-and-conquer counterpart, heuristically its average-case performance should be remarkably better. In fact, just as Huffman code provides optimal binary coding (in terms of code length) when the occurrence frequencies of the characters are known, the Huffman tree method described here provides an optimal solution for each and every instance of problem 2.

2.2.2 A different solution

The next algorithm we shall introduce is one that uses some form of preprocessing. The algorithm is based on the idea of *identifying crucial vertices.* Given a set S of convex polygons and a polygonal vertex p of S, p is said to be *crucial* if it is a vertex on the final union hull of S, and it is said to be *potentially crucial* if it is not known to be *non-crucial.* Initially all the vertices of the input convex polygons are

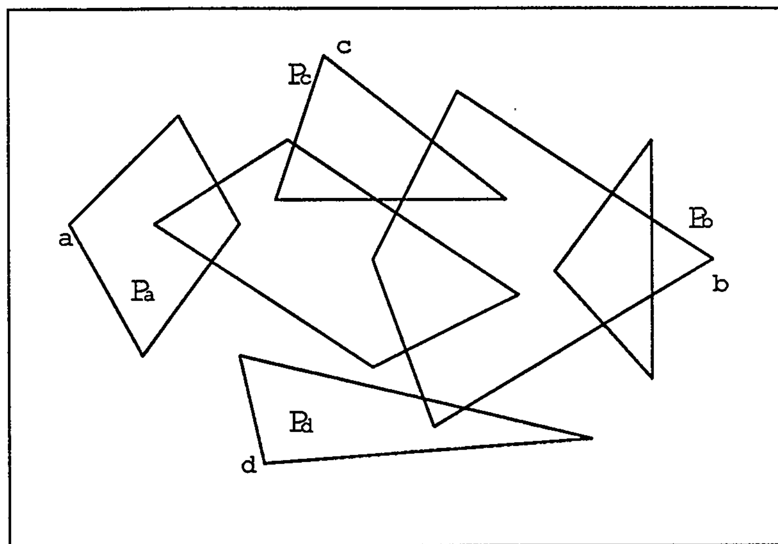


Figure 2.3: Forming a large "interim hull".

potentially crucial.

Our purpose is to eliminate non-crucial points as soon as possible, preferably when they first occur, thus to reduce unnecessary computation as much as possible. The algorithm starts with a single pass through the given polygons, and select those polygons that are holding "boundary" positions (see figure 2.3). By merging these "boundary" polygons, a somewhat "large" interim union hull is constructed. Then this one is used to merge with other convex polygons or interim hulls. We hope that when the "large interim hull" is carefully and reasonably constructed, there will be a good chance that non-crucial vertices be eliminated at their first occurrence.

Following is the algorithm:

1. Check through the input set S and select the convex polygons that contains the leftmost vertex a, the rightmost vertex b, the uppermost vertex c, and the

lowermost vertex d respectively. We call them P_a, P_b, P_c, and P_d.

2. Merge P_a and P_b into A, merge P_c and P_d into B.

3. Merge A and B into C.

4. If no more convex polygons are left unprocessed then stop, otherwise apply the Huffman tree algorithm to the now modified set of convex polygons.

Like the Huffman tree algorithm, this one has a worst-case time complexity of

$$O(N \log k).$$

where $N = \sum_{i=1}^{k} n_i$. This happens when all the polygonal vertices in S are crucial. For ordinary inputs, however, we claim that this algorithm should work considerably faster than the previous ones.

2.3 The lower bound

Result 2 shows that a convex m-gon A and a convex n-gon B can be merged within $O(m + n)$ time. We now wonder whether there exists a linear time algorithm for problem 2 as well. A closer study, however, shows that this cannot be true.

We demonstrate this by a simple contradiction: given a set of k points, its convex hull can be constructed by first divide the points into a number of triples, then compute the union hull of these triangles. As all triangles are convex polygons, if there is indeed a linear solution for problem 2, then there will also be a linear algorithm for the planar convex hull construction problem, which we know is not true.

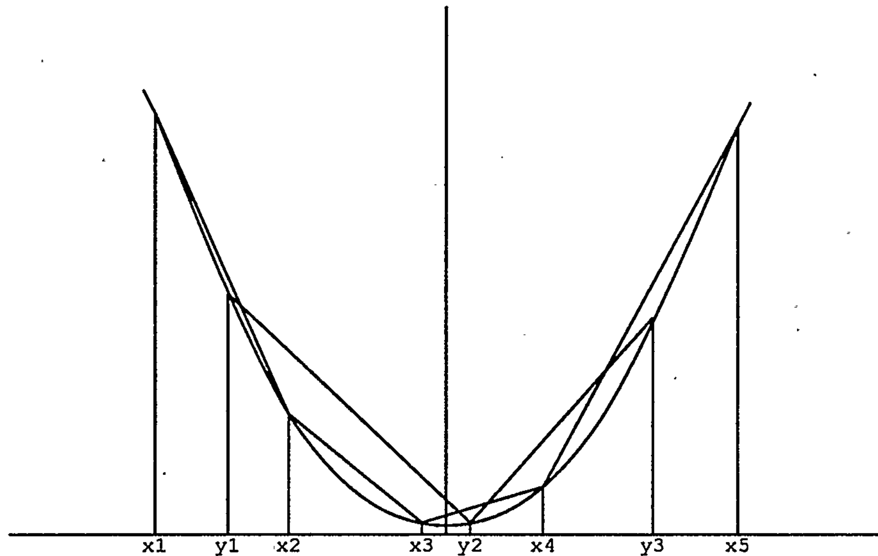


Figure 2.4: Merging vertices on a parabola.

In fact, any algorithm that solves problem 2, has a lower bound of $O((\sum_{i=1}^{k} n_i) \log k)$. We show this by a transform to the sorting problem (see figure 2.4).

Given n groups of real numbers, $(x_1^1, ...x_{n_1}^1), ...(x_1^k, ...x_{n_k}^k)$, all sorted. For each number x we construct the point (x, x^2), thereby to project all of them onto to the parabola $y = x^2$. For each of the original groups of numbers, which are sorted, we have on the parabola a corresponding convex hull. So on the parabola we have k convex hulls. To find their union, all the points have to be sorted and then the whole list be read in order. Obviously this can be done by merging the n sorted lists of numbers on the abscissa into one big sorted list. Therefore problem 2 can be transformed to

Problem 3 *Given k sorted lists $L_1, L_2, ..., L_k$ of real numbers, each of size $n_1, n_2, ..., n_k$ respectively. Merge them together into one sorted list L.*

Problem 3 clearly has a lower bound of $O(N \log k)$, where $N = \sum_{i=1}^{k} n_i$. In fact,

if problem 3 can be solved in a time complexity lower than this, then the ordinary n-sorting problem can be solved in less than $O(n \log n)$ time, which we know is not true.

2.4 Another algorithm

In this section we shall give an algorithm that uses a general preprocessing technique due to Golin and Sedgewick ([14]). The method was initially proposed to solve the convex hull construction problem for a set of points.

Given a set of k points in the plane, and assuming that the points in S are selected *uniformly and independently from a unit square*, the proposed technique works as follows: First construct several "guiding lines" that roughly mark the boundary of the set S, then use a parser to scan through the given set of points. By carefully selecting the guiding lines, virtually all of the "non-crucial" points (section 2.2.2) in the given set can be removed (provided the assumption on distribution holds). Finally, to the remaining point set, say S', apply a known convex hull algorithm. This method achieves linear time performance when the points in S are favorably distributed.

We now wish to apply this technique to our situation here. Consider problem 2. Assume that all the polygonal vertices of the given set S are chosen uniformly and independently from a convex r-gon. Also assume that each of the given convex polygons is represented by a linked list with the vertices stored in counter-clockwise order. Our union hull construction will proceed as follows:

1. Compute a rough "boundary" B for the given set S of convex polygons.

2. Match the polygonal vertices of S against the boundary obtained in step 1. If a vertex p is in the exterior of B then it is reserved, otherwise remove it from the set of S. When this step is completed almost all of the "non-crucial vertices" have been removed.

3. Apply one of the union hull construction algorithms described above to the remaining set S' of convex polygons.

In particular, the first step selects vertices that maximize the following functions: $(x+y), (x-y), (-x+y)$, and $(-x-y)$. This requires linear time. If there is more than one vertex maximizing the same function, say $(x+y)$, then select the two with the minimum and maximum x-coordinates. Then from the above we obtain at most eight points and they form a convex polygon B (anywhere from a quadrilateral to an octagon. See figure 2.5). Obviously any point that is in the interior of this polygon cannot be a vertex on the final union hull, and can therefore be eliminated.

The second step checks the original convex polygons one by one. For each vertex on a given polygon we check its inclusion in B. If the answer is "Yes" then the vertex is removed from its polygon, otherwise it remains in the list. To see if a point is in the interior of a given convex polygon or not takes constant time, so the second step can be done within linear cost. When all k convex polygons in S are examined the preprocessing stage is finished.

The next thing to do is applying one of the previous union hull construction algorithms to the remaining set S' of convex polygons to get the final result. According to our assumption, the polygonal points in the set of S are uniformly and independently distributed in a convex r-gon. Thus the union hull $UH(S)$ of S is expected

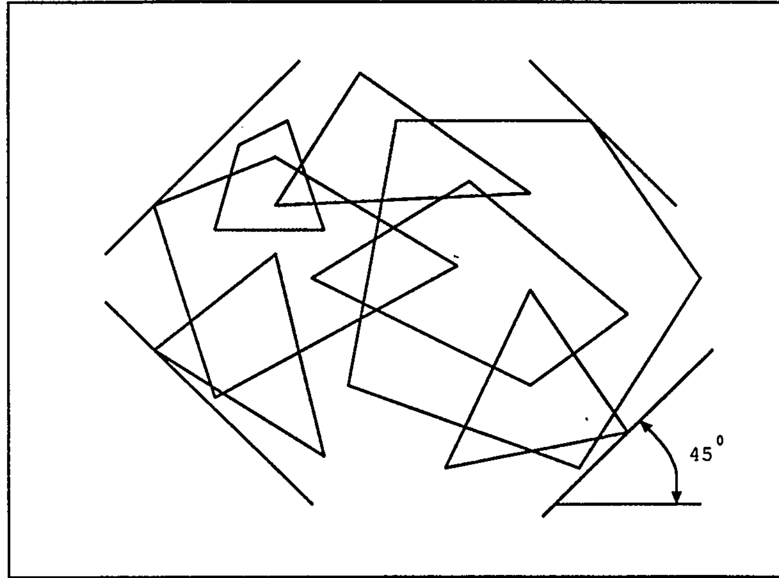


Figure 2.5: A preprocessing strategy.

to have $O(cr)$ hull vertices, where c is a constant ([33]). Following a similar analysis as in [14], we know that the set S' should contain, on the average basis, $O(c'r)$ of the original vertices in S, where c' is another constant. This means that the final step can be finished in an expected time of $O(r \log k)$ where r should be confined by $k/\log k$ (i.e., $r \leq k/\log k$)

This algorithm, when the assumed distribution is satisfied, has a linear time average-case time complexity ([14], [30]).

2.5 Some comments

There is an interesting comparison between the union hull problem (problem 2) and the ordinary convex hull problem. Considering their inputs, the ordinary convex hull problem takes a set of independent points, whereas the input for the union

hull construction problem is a set of convex polygons. When all the input convex polygons are reduced to having only one single vertex, the k hull union problem is degenerated into an ordinary convex hull construction problem. If, however, we group the input vertices of a convex hull construction problem into triples (triangles) then we are in fact facing an union hull problem. Therefore there is a linear time reduction between the two problems in both directions.

As is shown in our discussion, problem 2 can be solved in a *two-way-merge* manner, but other ways might exist as well. Indeed, a set S of k convex polygons can be viewed as something between a convex k-gon and a set of k independent points Hence we have reason to expect an in-between algorithm. Recall that when constructing the convex hull of a planar set of points, we either select those who *are* hull points, or eliminate those who *are not* hull points. This notion of convex hull construction, however, also applies to union hull construction, except that now a set of convex polygons is received as input. This leads to an interesting situation: we can still use either of the selecting or eliminating approach to construct the union hull, but normally we neither select nor eliminate a particular input polygon entirely. Rather, the process is conducted on two levels: the component polygon level and the vertex level. A careful coordination over the two levels is therefore required.

Chapter 3

Intersecting Convex Polygons

Intersection is an important set-theoretic operation on convex polygons. For example, CAD/CAM systems and graphics applications require efficient algorithms to find the intersection of two or more convex polygons. For a given set of convex polygons, we consider two (related) intersection problems:

- Calculate their intersection (computation).
- Find out if the polygons intersect or not (detection).

This chapter studies the two problems. The organization is as follows: Section 3.1 gives definitions and results that are important to our discussion and section 3.2 studies the intersection computation problem. A lower bound for this problem is given and an algorithms that achieves this lower bound is presented. Section 3.3 focuses on the intersection detection problem, in which we give an algorithm that improves the current solutions. Finally in section 3.4 we put these two problems on a common ground for some general comparisons.

In the following, we use "polygon" to denote both the boundary and its interior, and "polygon boundary" to indicate the boundary itself. In all circumstances "polygon" should stand for "convex polygon" if not otherwise specified. The term "intersection" may carry different meanings in our discussion: The *intersection of two polygons* means the polygonal area that is common to both, but the *intersection of two polygonal chains* (see section 3.1) will indicate their points of intersection.

Lastly all the angles of lines or lines segments are measured within the range of $[-\pi/2, \pi/2]$.

3.1 Preliminaries

In our discussion below, polygons are represented in arrays with their vertices given in clockwise order. Also, we assume that no three vertices of any of the polygons are collinear.

3.1.1 Definitions and notations

A *polygonal chain* is the part of a convex polygon that is in between two of its boundary points (not necessarily vertices). A polygon P can be decomposed into two polygonal chains by cutting at any two boundary points. By convexity, these two chains (excluding the case where one of them becomes a line segment) should have particular *orientations.*

When the two cutting points are chosen to be the polygon's leftmost vertex and rightmost vertex, we come up with two polygonal chains that are either facing upwards or downwards. For convenience, we shall hereafter call an up-oriented polygonal chain a *cup* and a down-oriented one a *cap.* Note that all cups and caps are strictly monotone (increasing or decreasing) in terms of the slopes of their edges.

A real function f defined on the integers $1, 2, ..., n$ is said to be *unimodal* if there is an integer $m (1 \leq m \leq n)$ such that f is strictly increasing (respectively, decreasing) on [1,m] and decreasing (respectively, increasing) on $[m+1, n]$. A real function g defined on integers $1, 2, ..., n$ is said to be *bimodal*,, if there is a r in [1,n] such that

$f(r), f(r+1), ...f(n), f(1), ...f(r-1)$ is unimodal [8].

In [18], Kiefer showed that *Fibonacci Search* is the optimal method to find the turning point of a unimodal function, requiring $O(\log n)$ probes. Chazelle and Dobkin [8] extended this result and showed that *the extrema of a bimodal function can also be computed in $O(\log n)$ time.* As this is a very important result on which much of our discussion in this chapter is based, we rephrase it as follows:

Result 3 *(Finding the extrema of a bimodal function). Given a bimodal function f defined on the integers $1, 2, ..., n$, its extrema (or turning points) can be found in time $O(\log n)$ by using Fibonacci Search.*

3.1.2 Important results

In the following we give several results that are of importance to our discussion.

Result 4 *The leftmost vertex and rightmost vertex(respectively, the uppermost vertex and the lowermost vertex) of a convex n-gon P can be found in time $O(\log n)$, which is optimal in the worst case.*

Proof: Draw a vertical (or, horizontal) line l across the plane (which may or may not intersect the polygon P). By convexity, the oriented distances $d_1, d_2, ..., d_n$ from P's vertices $p_1, p_2, ...p_n$ to l form a bimodal function. According to result 3 its extrema (the turning points) can be found in $O(\log n)$ time. □

Result 5 *The intersection of a infinite line l with a convex n-gon P can be computed in time $O(\log n)$, which is optimal in the worst case.*

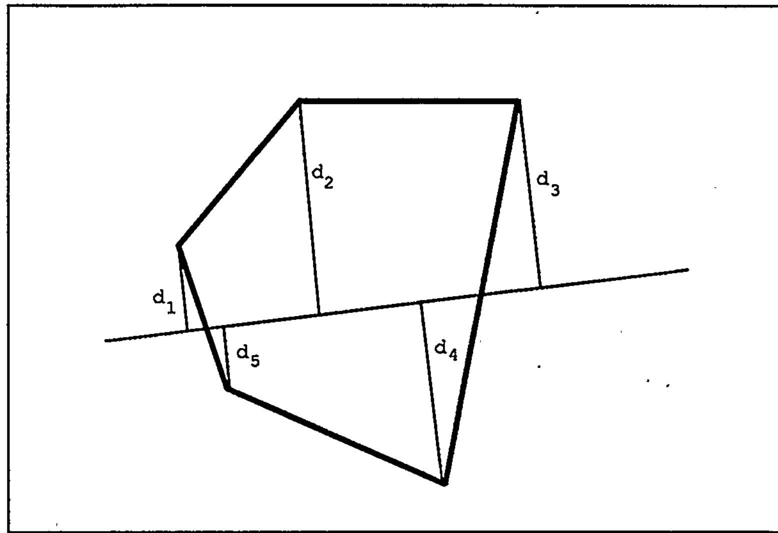


Figure 3.1: The intersection of an infinite line with a convex polygon.

Proof: By convexity, the oriented distances $d_1, d_2, ..., d_n$ from the polygon's vertices $p_1, p_2, ...p_n$ to l form a bimodal function. Using *Fibonacci Search* we can find its extrema in $O(\log n)$ time (result 3). If the two extrema are of the same sign (which means the two points are on the same side of l) then there is no intersection. Otherwise the polygon P intersects the line l. We can start from the two extreme vertices p_i, p_j and use binary search to find the intersections, which will require $O(\log n)$ time (see figure 3.1). □.

Result 6 *The inclusion of a point p in a convex n-gon P can be tested in time $O(\log n)$, which is optimal in the worst case.*

Proof: Draw a vertical line l across p and check the intersection(s) of polygon P and line l, which takes $O(\log n)$ time (result 5). If they do not intersect this means that p is not in the interior of P. Otherwise let q and q' be the two points where l meets the boundary of P (it is trivial if there is only one intersection), and we only need to

check whether p is in between the two points or not, which takes constant time. □

Result 7 *The intersection of a convex m-gon P and a convex n-gon Q can be computed in time $O(m+n)$, which is optimal in the worst case.*

Proof: There are two major approaches to compute the intersection of two convex polygons in linear time: one is based on "plane-sweeping" [35] and the other on the idea of "edge advancing" [27]. We shall not repeat the details of these algorithms. Note that the intersection of two convex polygons is still a convex polygon. □

Result 8 *The intersection of a convex m-gon P and a convex n-gon Q can be detected in time $O(\log(m+n))$, which is optimal in the worst case.*

Proof: Note that we *detect*, rather than *compute*, the intersection of two convex polygons. A detailed lower bound proof as well as an algorithm that achieves this lower bound can be found in [8]. An algorithm that solves the problem is expected to return "No" when the polygons do not intersect, and an intersecting point otherwise. □

Result 9 *The intersection of a* **cup** *C_1, which has m vertices, and a* **cap** *C_2, which has n vertices, can be detected in time $O(\log(m+n))$, which is optimal in the worst case.*

Proof: This is a direct consequence of result 8 (figure 3.2). Note that the problem is different from the general problem of detecting the intersection of *any* two given polygonal chains, which requires $O(m+n)$ time (where m and n are respectively the number of vertices on the chains, see [8]). □

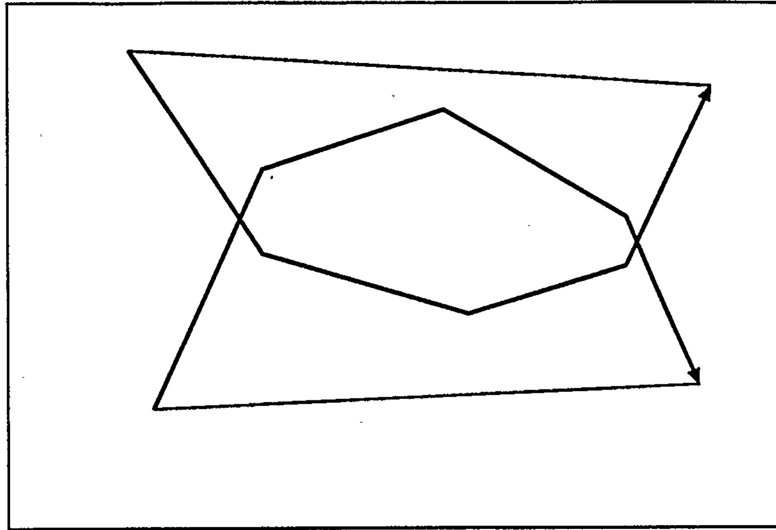


Figure 3.2: The intersection of a *cup* and a *cap*.

3.2 Compute the Intersection of Convex Polygons

We already know that to compute the intersection of a convex m-gon P and a convex n-gon Q takes $O(m+n)$ time (result 7). Now we wish to consider a more general version of this problem.

Problem 4 *Given a set S of k convex n-gons $P_1, P_2, ..., P_k$, calculate their intersection I.*

Before discussing a solution to problem 4, we shall first establish a lower bound.

3.2.1 The lower bound

Result 7 shows that linear time suffices to compute the intersection of two convex polygons. It is therefore natural to conjecture that problem 4 also has a linear solution. This conjecture, however, is not true.

Indeed, problem 4 and problem 1 have the same abstraction. Let $\cap$ denote the basic operation that computes the intersection of *two* convex polygons, then both problems fall into the following general problem: Given a set S of k objects $O_1, O_2, ..., O_k$, compute the parameter

$$PAR(S) = O_1(op)O_2(op)...(op)O_k.$$

with the following restrictions:

1. (op) is a 2-operand operation that can be either $\oplus$ or $\cap$.
2. All the O_i's $(i = 1, 2, ..., k)$ are objects of size n.
3. It takes $O(m+n)$ time to perform (op) on two objects of sizes m and n.
4. Each time only one operation (op) can be performed.
5. If O_1 has size m and O_2 has size n then $O_1(op)O_2$ has size $s \leq m+n$.
6. For any O_1 of size n and any integer m, there exists O_2 of size m such that $O_1(op)O_2$ has size exactly $n+m$.

Properties 1 through 4 attach to both problems in an obvious manner. Properties 5 and 6 holds for the union hull problem in that (1), the candidate vertices for the merge of a convex m-gon P and a convex n-gon Q are those vertices of P and Q alone, and (2), given P we can always find a convex n-gon R such that the merge of P and R has precisely $m+n$ vertices by first constructing a convex $(m+n)$-gon S which has P's m vertices as its own vertices, then read off the n vertices of S that do not belong to P. The same thing is also true for the intersection computation problem.

Under the above generalization, the two problems are identical *in the worst case*, i.e., the requested parameter $PAR(S)$ is of size kn. From section 2.3, we know that it takes $O(kn \log k)$ to solve problem 1 in the worst case, so the following result holds:

Result 10 *(Lower bound for intersection computation). The lower bound on the complexity of any algorithm that solves problem 5 is $O(kn \log k)$.*

3.2.2 The algorithms

Finding the intersection of a set of convex polygons is very similar to constructing the union hull (chapter 2). Indeed, with a little modification all the algorithms in section 2.1 and section 2.2 can be borrowed to serve our purpose here.

For example, a divide-and-conquer algorithm can be devised as follows:

1. Divide the original set S of k convex polygons into two subsets S_1 and S_2 of approximately equal sizes.

2. Compute the intersection I_1 of polygons in S_1 and the intersection I_2 of polygons in S_2, recursively.

3. Intersect I_1 and I_2 to get I, which is the intersection we want.

Let $T(k)$ be the time required to solve problem 4, we have the following formula:

$$T(k) = 2T(k/2) + t. \tag{3.1}$$

where t denotes the time needed to calculate the intersection I of I_1 and I_2. As I_1 and I_2 are intersections of $k/2$ convex n-gons, they each may have up to $nk/2$ vertices. Hence $t = O(kn)$ (result 7). Immediately we have:

$$T(k) = 2T(k/2) + O(kn) = O(kn \log k). \tag{3.2}$$

According to result 10, this divide-and-conquer algorithm is optimal in the worst case. However, we can still improve the average-case performance substantially by using methods introduced in the construction of union hulls (section 2.2, section 2.4). The only difference is that, when constructing the union hull for a set of convex polygons, we try to identify and *retain* those "boundary vertices" in our preprocessing. When computing the intersection, however, we should identify and *remove* those "boundary vertices" that are unlikely to be vertices of I. We shall not repeat this technique here.

3.3 Detecting the Intersection of Convex Polygons

Given a set S of k polygons $P_1, P_2, ..., P_k$, where each has at most n vertices, we are interested in the following problem:

Problem 5 *(Detecting the intersection of convex polygons) Given a set S of $k(k > 1)$ convex n-gons $P_1, P_2, ..., P_k$, find out whether they intersect or not.*

We observe that any algorithm that solves problem 4 should also work here: for a given set S of convex polygons we calculate their intersection I. If I is empty this means that the polygons share no common point. Otherwise any point in I is an intersection point, and we can pick, say the geometric center of I, and return it as an appropriate answer. According to result 10, this solution needs $O(nk \log k)$ time in the worst case.

This simple method, however, is by no means the best. Indeed, a detection task is in general inherently easier than its computation counterpart [7]. To be particular, a computation task normally has to give a detailed description of the object being

computed, which in the case of intersection computation, requires a full boundary description of the overlapped area. A detection task, however, does not usually have to deal with such detail. Taking the intersection detection problem for example, all we need to do is to find out whether an intersection exists or not. Because of this difference in nature between computation and detection tasks, we would like to find a solution to problem 5 that is more particular and more efficient.

3.3.1 A linear algorithm

The first solution we give is based on the observation that a convex polygon P is the common intersection of a set of halfplanes associated with its edges. Under this observation, the set S of k convex n-gons given in problem 5 can be considered as a set of nk halfplanes, where each corresponds to a linear inequality of the form:

$$a_i x + b_i y + c_i \leq 0, (i = 1, 2, ..., nk). \tag{3.3}$$

Our problem now is equivalent to finding a *feasible point* for a given set of linear inequalities. If no feasible point is found, this means that the given polygons do not intersect. Otherwise a calculated feasible point can be returned as an appropriate intersection. For this transformed problem, Dyer [11] and Megiddo [23] have given algorithms that operates in $O(nk)$ time.

3.3.2 Reichling's algorithm

An improvement over the above algorithm is made by Reichling [31]. Instead of taking into account all the edges of the polygons in order to find a common intersecting point (denoted by CIP hereafter), he establishes necessary conditions for CIPs to

exist. Attention is only given to a certain area (called *crucial area*) that will possibly contain a CIP. By carefully tightening the boundary of the crucial area the problem is solved in a more efficient manner. Following is a outline of this algorithm with some revision.

1. Establish a left bound b_l and a right bound b_r for possible CIP's. Both b_l and b_r are vertical lines that go through one or more of the kn vertices. If $b_l > b_r$, then the polygons should have no common intersection; otherwise any possible CIP for $P_1, P_2, ..., P_k$ must fall into the strip in between them (see figure 3.3).

2. Tighten up the bounds (i.e. define a new set of bounds). Make sure that every time this step is performed, a constant portion of the vertices is removed from consideration. After $O(\log n)$ applications of the process, one of the following must happen: (a) A contradiction did occur (no intersection); (b) A CIP has been found; (c) There are less than ck (c is a small constant, say 5) vertices left in the strip between b_l and b_r. For the first two cases the job is already done. For case (c) go to step 3.

3. There are now at most $2ck$ edges left to be considered, and we can do the remaining job by using the linear time algorithm mentioned above. This remaining computation will take $O(k)$ time.

The left bound b_l and the right bound b_r in the algorithm can be computed as follows: First find the leftmost vertex l_i and the rightmost vertex r_i for polygon P_i ($i = 1, 2, ..., k$), which takes $O(k \log n)$ time according to result 4. Then compute the maximal x-coordinate x_l of the l_i's and the minimal x-coordinate x_r of the r_i's, and

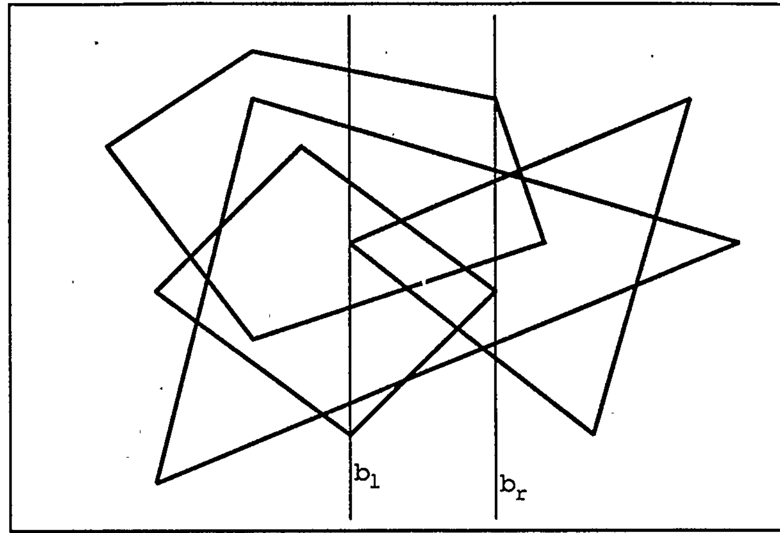


Figure 3.3: Setting the left bound b_l and the right bound b_r.

draw one vertical line across x_l which is b_l, another across x_r which is b_r. This part of the computation takes $O(k)$ time. Therefore b_l and b_r can be found in $O(k \log n)$ time. What remains is how to tighten up the bounds.

Imagine using a vertical line l between b_l and b_r to cut through the polygons. For each P_i we can calculate in $O(\log n)$ time the intersections m_i^u and m_i^l of l with P_i's upper and lower chain. So in $O(k \log n)$ time all of the k pairs of intersections can be determined. Let m^u be the m_i^u with the minimal y-coordinate and m^l be the m_i^l with the maximal y coordinate (m^u, m^l can be found in $O(k)$ time). If $m^u > m^l$ then any point on l between m^u and m^l is a CIP. Otherwise we calculate the slopes sl^u and sl^l of the edges defining m^u and m^l (if l passes through a vertex then both edges are considered). If $sl^u < sl^l$, then a CIP can only exist to the left-hand side of l, in which case we replace b_r with l. If $sl^u > sl^l$, then a CIP can only exist to the right-hand side of l, in which case we replace b_l with l. If $sl^u = sl^l$, then there can be no CIP in existence. (figure 3.4)

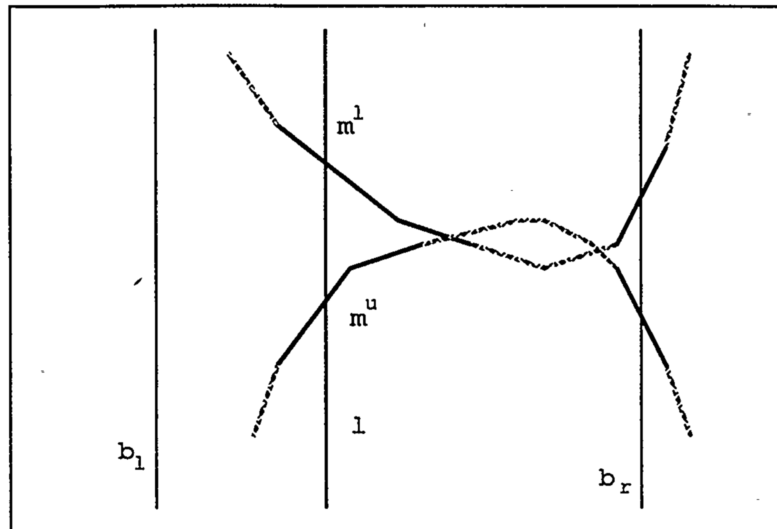


Figure 3.4: Tighten up the bounds – Reichling's algorithm.

Now we can use a vertical line l to tighten up the bounds, in $O(k \log n)$ time. It remains to decide how to select the line l in such a way that every time either it terminates the computation, i.e., a "Yes" or "No" answer is found, or a constant portion of the vertices between b_l and b_r are discarded. In Reichling's algorithm, such a vertical line l can be found in $O(k)$ time. To avoid lengthy description we shall not give the details here.

Summarizing the above, the algorithm has a time complexity of $O(k \log^2 n)$ in the worst case.

A variation of this algorithm is to set the line l in the middle of b_l and b_r, so that each time the stripe between b_l and b_r is reduced by half of its width. Serious problem may arise when the input vertices take up a weird distribution, but this can be avoided if a simple measure is taken to make sure that every time at least one point is removed. We believe that this variation should have a similar time complexity as its peer for ordinary input patterns, and can even be better for certain cases. The

major drawback is that its performance depends heavily on the distribution of the given vertices, and is thus unpredictable. Moreover, it has a time complexity of $O(nk)$ in the worst case (i.e., each time only one or two points are eliminated).

Reichling's algorithm has a very good worst-case performance, yet its average-case performance can still be considerably improved. In fact, due to its bound tightening-up strategy, a time of $O(k \log^2 n)$ is normally required when the given polygons do not intersect. In applications where k is big, it is likely that the given convex polygons share no common points. When this happens, we would like to return a negative answer much faster.

3.4 Using Newton Iteration

To further improve Reichling's result, the convexity of the input data has to be explored.

Remember that to cut the polygons $P_1, P_2, ..., P_k$ with a pair of bounds b_l and b_r, we end up with $2k$ polygonal chains of which k are cups and k are caps. By convexity, the upper boundary CUP formed by the k cups (respectively, the lower boundary CAP formed by the k caps) is also a cup (respectively, a cap). Obviously if the given convex polygons intersect so will CUP and CAP, and vice versa. Since both CUP and CAP are convex, the idea of Newton's Iteration can be used.

3.4.1 Intersecting a cup and a cap

Before giving the algorithm, we first introduce an important result:

Result 11 *Let A and B be respectively a cup and cap confined by a left bound b_l and a right bound b_r, each has no more than n vertices, then they have at most two intersection points, and their intersection(s) can be found in $O(\log n)$ time.*

Proof: We can determine whether A and B intersect or not in time $O(\log n)$ (see section 3.1), and it is easy to see that they have at most two intersections. In the following we shall assume that A and B intersect, and we will give an algorithm that finds the intersection(s) in time $O(\log n)$.

Let a and b be respectively the points where A and B meet b_l, and assume that their y-coordinates satisfy $a_y > b_y$ (otherwise we consider the intersections of A, B with b_r, the result will be the same). Without losing generality, we shall compute the intersection point p of A and B that is closer to b_l.

1. Let P_a be the convex polygon defined by A (join the two endpoints of A), and let P_b be the convex polygon defined by B. Then P_a and P_b should intersect, and a point q of their intersection can be found in time $O(\log n)$.

2. Draw a vertical line l through q, and assume that it intersects A and B respectively at a' and b'. It should hold that $a' < b'$. This again takes $O(\log n)$ time.

3. For either A or B we already know the number of vertices and their indices between the vertical lines b_l and l. Let i_1, i_2 be respectively the indices of a, a' on A and j_1, j_2 be respectively the indices of b, b' on B. The $\lfloor (i_1 + i_2)/2 \rfloor$-th vertex v_a of A (respectively, the $\lfloor (j_1 + j_2)/2 \rfloor$-th vertex v_b of B) is called the *current* vertex of A(respectively, B). The *current* edge e_a of A(respectively, e_b

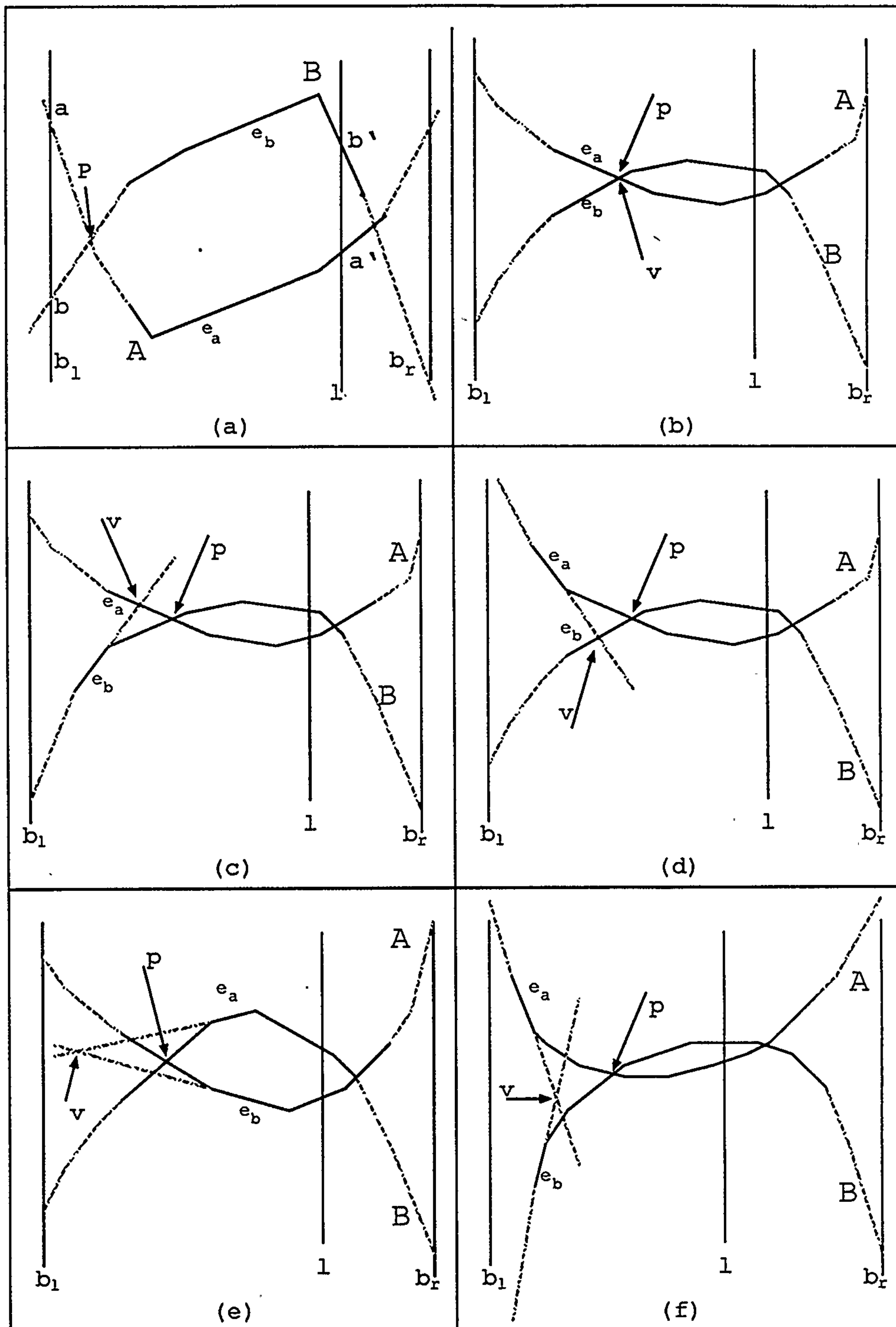


Figure 3.5: Finding the intersection(s) of a cup and a cap.

B) is defined to be the edge directly to the right of v_a(respectively, v_b). Now extend e_a and e_b at both directions and find their intersection v. There are the following cases:

(a) v does not exist(they are parallel). By complexity, this can only be the case shown in figure 3.5(a). We just let $i_2 = index(v_a)$ and $j_2 = index(v_b)$, and then continue the process.

(b) v falls into both e_a and e_b (figure 3.5(b)). When this happens v is the desired solution.

(c) v falls into e_a(figure 3.5(c)). For this case, if v is to the left-hand side of e_b then $j_2 = index(v_b)$, otherwise $j_1 = index(v_b)$.

(d) v falls into e_b(figure 3.5(d)). For this case, if v is to the left-hand side of e_a then $i_2 = index(v_a)$, otherwise $i_1 = index(v_a)$.

(e) v is to the left-hand side of e_a(figure 3.5(e)). For this case let $i_2 = index(v_a)$. if v is also to the left-hand side of e_b then $j_2 = index(v_b)$, otherwise $j_1 = index(v_b)$.

(f) v is to the right-hand side of e_a(figure 3.5(f)). For this case let $i_1 = index(v_a)$. If v is also to the right-hand side of e_b then $j_1 = index(v_b)$, otherwise $j_2 = index(v_b)$.

Keep on doing this until the intersection is found. As this is a typical binary search, the time needed for step 3 is also $O(\log n)$. □

3.4.2 The algorithm

We now give our algorithm that detects the intersection of a set S of k convex polygons $P_1, P_2, ..., P_k$.

1. Establish a left bound b_l and a right bound b_r, in the same way as described in Reichling's algorithm. This takes $O(k \log n)$ time. Now between b_l and b_r there are at most k cups and k caps.

2. Let CUP denote the upper boundary of the k cups and CAP the lower boundary of the k caps. Find the intersections u_1, v_1(respectively, u_2, v_2) of b_l(respectively, b_r) with CUP and CAP (figure 3.6). This takes time $O(k)$.

3. Let U_1, U_2, V_1, and V_2 be respectively the cups and caps that are associated with the points of u_1, u_2, v_1, and v_2. Compute the intersection(s) of U_1 and V_1 and name the one to the left as w_l. Compute the intersection(s) of U_2 and V_2 and name the one to the right as w_r. Now consider the following cases:

   (a) if w_l's x-coordinate is greater than that of w_r's, then no CIP could exist, so "No" is returned. Otherwise use the method in Reichling's algorithm to select a vertical dividing line l.

   (b) if l lies to the left of w_l, then draw a vertical line l' through w_l, compute the intersections of l' with all the k cups and k caps, and replace b_l with l'.

   (c) if l lies to the right of w_r, then draw a vertical line l' through w_r, compute the intersections of l' with all the k cups and k caps, and replace b_r with l'.

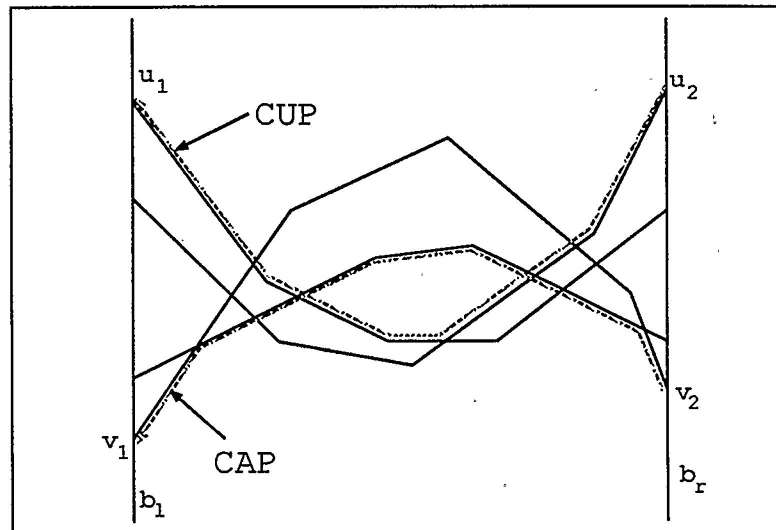


Figure 3.6: The bounds b_l and b_r intersect CUP and CAP.

(d) if l lies between w_l and w_r, then find the intersections u and v of l with CUP and CAP. If the y-coordinate of u is less than that of v, any point on l between u and v is a CIP. Otherwise use the strategy in Reichling's algorithm to decide whether b_l or b_r is to be replaced by l.

It is easy to see that for each and every case our algorithm has at least the same performance as Reichling's algorithm. For cases where the k polygons do not intersect, our algorithm is expected to return a negative answer significantly faster. This is due to the introduction of Newton iteration, which is a second order process by itself. The worst-case time complexity of this algorithm is also $O(k \log^2 n)$.

A tight lower bound to detect the intersection of k convex n-gon's is still unknown. If it could be shown that the most efficient way to decide whether a point p is an intersection point of k given polygons is to proceed by checking the inclusion of p in each of them, then it would follow (by result 7) that $O(k \log n)$ is a lower bound.

Chapter 4

Separating Convex Polygons

An interesting problem in computational geometry as well as computer graphics is *separability detection.* Given a set of (often convex) planar objects, we are interested in finding out whether the objects are "separable"" or not. By definition, a *separating line* for a given set S of k objects $O_1, O_2, ..., O_k$ in the plane is a straight line l that divides S into two nonempty subsets, and yet does not intersect any of its elements. The set S is said to be *separable* if a separating line exists, otherwise it is said to be *nonseparable.* Consider for example the problem of erecting a straight wall to separate a city into two parts.

This problem has, in the past decade, caught attention of both mathematicians and computer scientists ([13], [12]). There is no complete solution so far. In [12], the problem is included in a list of open problems that seem to bear a close relationship to the "stabbing line (or transversal) problem" (section 4.1.1).

This chapter investigates the separability detection problem, with the objects being line segments or convex polygons. Our discussion is arranged as follows: Section 4.1.1 compares our problem with the 'stabbing line problem", and show that the two are unlikely to be closely related. Section 4.2 studies the problem of separability detection for a given set of line segments. Several algorithms are given and a lower bound analysis is also presented. Finally in section 4.3 we discuss the more general problem of detecting separability for a set of convex polygons.

4.1 A Comparison

In this section we shall compare separability detection with the stabbing line problem. If, as is conjectured in [12], the two problems are indeed closely related, we would hope that the methods devised for the stabbing line problem could be utilized in solving the separability detection problem. If, however, such a relationship does not exist, or is not likely to exist, we then must find other ways to do separability detection.

4.1.1 Stabbing line

We begin with an introduction to the stabbing line problem. By definition, a *stabbing line* (or transversal) for a set S of k objects (often convex) $O_1, O_2, ..., O_k$ in d-dimensional Euclidian space E^d is a straight line l in E^d that intersects all of the k objects.

The planar stabbing line problem, i.e., detecting the existence of a transversal for a given set of planar convex objects, has already been thoroughly studied. In [12], Edelsbrunner et al. presents an algorithm that solves the stabbing line problem for a set of k line segments in $O(k \log k)$ time. The authors combined three important algorithm design paradigms: geometric transform, divide-and-conquer, and plane sweeping in the design of their algorithm, which provides not only transversal detection but also a complete description of all the transversals. Later in [3], Avis et al. proved that this $O(k \log k)$ time complexity is also a lower bound.

Little is done about the stabbing line problem, however, when it comes to three and higher dimensional spaces. Even the case for a set of convex polygons is less

well known.

4.1.2 Separability vs. stabbing line

Although it is suggested that there is likely a close relationship between separability detection and the stabbing line problem, we believe that such a relationship can not exist.

Taking a set S of k planar line segments for example. If we consider each individual segment of S as *a door between two semi-infinite straight walls*, then the stabbing line problem (hereinafter denoted by SL) and the separability detection problem (hereinafter denoted by SD) can respectively be interpreted as follows:

- SL: Determine whether there is a light that shines through all of the k doors.
- SD: Determine whether a mouse can dig through k walls (there are traps at the doors), following a straight line and has at least one door on either side of the line.

Under this model we have some important observations. Consider the SL problem. Suppose the light has just passed i ($i < k$) doors, then for the next door only two things may happen: either it passes through the door, or it stops. Therefore there are two possibilities in front of each door, one "Yes" and another "No'. The SD problem, however, is different. When the mouse comes to each door three things may happen: it either digs through the wall on the left, or it digs through the wall on the right, or can be trapped at the door. So the possibilities are two "Yes"s and one "No". Not only this, suppose the mouse is lucky enough to successfully dig through

k walls (the maximum it can do), the job still continues — it has to look back and see whether there is at least one door on either side of its route.

The fact that the SD problem has two "Yes" options in front of each door implies that it has a much larger search space as compared with its SL counterpart, and this difficulty is added to by the constraint that there must be at least one door on either side of the route. In fact, the algorithm given in [12] for the SL problem is based on the prerequisite that there can be only one "Yes" option for each door. Hence their method can not be utilized for the SD problem, and other solutions have to be found.

4.1.3 Some definitions

The following definitions are to be used in our discussion:

A *ray* is a semi-infinite straight line with one endpoint. Given a ray r, its *angle* is defined to be the angle measured clockwise from r to the positive x-axis. Rays have inverses, and the *inverse* $\overline{r}$ of a given ray r is the ray that has the same endpoint as r, yet extends in the opposite direction.

Two rays r_1 and r_2 having a common endpoint define two fan-shaped areas, and such fan-shaped areas we shall call *wedges*; Given a wedge W, its *angle* is defined by the angle of the one of its two rays which we hit first when travelling clockwise from an interior point. Wedges also have inverses. The *inverse* $\overline{W}$ of a given wedge W is the wedge constructed from the inverses of W's two rays, so that $W \cap \overline{W}$ consists of a single point (figure 4.1).

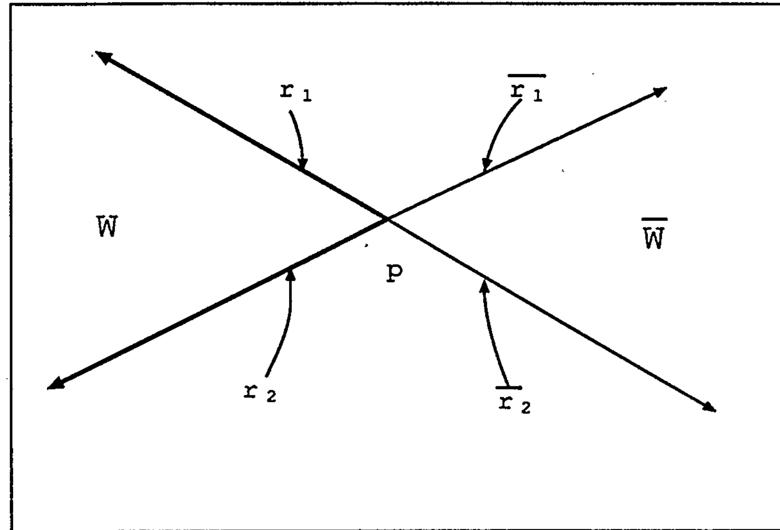


Figure 4.1: rays and wedges

4.2 Separating line segments

Before working with convex polygons, we first consider separating a given set of line segments:

Problem 6 *Given a set S of k line segments $l_1, l_2, ..., l_k$ on the plane, find out whether there is a separating line for S or not.*

We assume that all segments are represented by their endpoints. If not otherwise specified, p_i and q_i should denote the endpoints of the line segment l_i $(1 \leq i \leq k)$. For convenience of discussion, all line segments are considered to be *open*, i.e., l_i is the equivalent of (p_i, q_i), for $i = 1, 2, ..., k$.

4.2.1 Algorithm 1

Our first attempt for a solution is suggested by Jarvis's March [17]: compute the convex hull of S (which have $2k$ points), start up at a hull point p and walk coun-

terclockwise. If then somewhere during the trip one can "look through" the set of segments this means S is separable. If p is visited again and no such "looking through" point has been found then a negative answer is returned.

This method will not work, however, due to the fact that in Jarvis's March, we only need to walk through the points in the given set. Our case is different: not only do the convex hull vertices need to be checked, but also all the points on the hull edges, because a "looking through" point may occur anywhere on the convex hull. This is computationally impossible. We wonder, therefore, if it is possible to check only a limited number of points.

Figure 4.2 reveals two simple facts: first, if S has a separating line l of slope sl, it must have another separating line l' (may be the same as l) that is parallel to l and goes through at least one of the endpoints of S. Second, if an above-mentioned separating line l' is found, then we can further find another separating line l_1 (may be the same as l') that passes through not only the endpoint p but also another endpoint q at the same time. Hence we have:

Result 12 *If a set S of k line segments $l_1, l_2, ..., l_k$ is separable, then there must be a separating line of S that goes through at least two of the given endpoints.*

With result 12, we immediately have the following algorithm:

1. Select an endpoint p_i of l_i and an endpoint p_j of l_j.

2. Check to see whether the straight line l defined by p_i and p_j is a separating line or not.

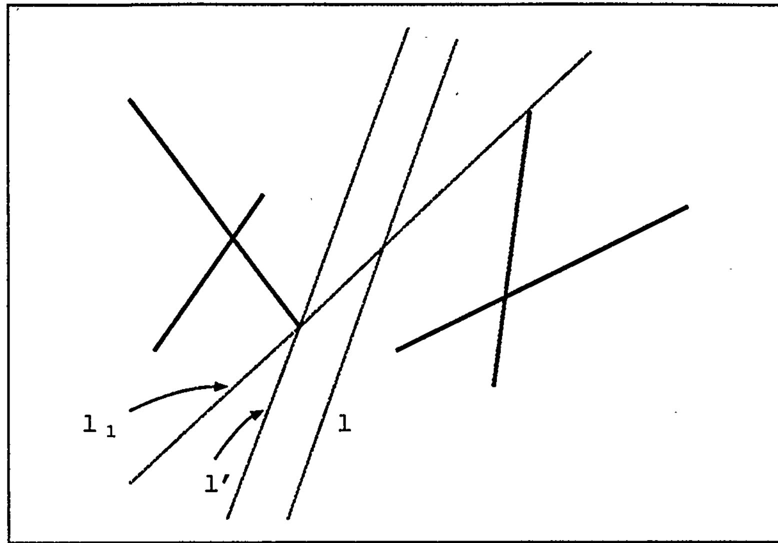


Figure 4.2: separate via endpoints

3. Keep on doing this for different pairs of endpoints until either a separating line is found or all of the pairwise endpoints have been checked.

The k segments l_1 through l_k has altogether $2k$ endpoints which pairwise define $O(k^2)$ different straight lines. To test whether a given line is a separating line or not takes $O(k)$ time, hence the algorithm runs in $O(k^3)$ time and $O(k)$ space.

This algorithm is not completely satisfactory. In fact an algorithm that runs in time $O(k^2 \log k))$ can be obtained by applying a general method due to Edelsbrunner et al. [13]. The method is based on the basic concept of *view change*. In the following, however, we shall consider a solution to problem 6 from a different perspective.

4.2.2 A restricted problem

We shall first consider the following problem:

Problem 7 *Given a set S of k line segments $l_1, l_2, ..., l_k$ and a point p, find out whether there is a separating line of S that goes through p.*

Clearly this problem is closely related to problem 6. Indeed, result 12 shows that to solve problem 6, we only need to check the straight lines that pass through the given endpoints of S, and to see whether there are separating lines or not. If we find a solution to problem 7, then by substituting each of the endpoints of S into the position of p, problem 6 can be solved. Following is an algorithm for problem 7:

1. Construct the convex hull H of the given $2k$ endpoints. If p is on or outside of H then go to step 6, otherwise continue.

2. Shoot two rays r_i^1 and r_i^2 from p to each of l_i's two end points p_i and q_i $(i = 1, 2, ..., k)$. The k pairs of rays $(r_1^1, r_1^2), (r_2^1, r_2^2), ..., (r_k^1, r_k^2)$ define k wedges, namely $W_1, W_2, ..., W_k$, where each of which contains one of the given line segments.

3. Construct the inverses $\overline{W_1}, \overline{W_2}, ..., \overline{W_k}$ of these k wedges.

4. Sort the $2k$ wedges $(W_1, W_2, ..., W_k, \overline{W_1}, \overline{W_2}, ..., \overline{W_k})$ according to their angles into an ordered list $W'_1, W'_2, ..., W'_{2k}$.

5. Walk counterclockwise around p and check the sequence of $W'_1, W'_2, ..., W'_{2k}$ to see whether or not they cover the entire plane (figure 4.3). If they do not then this implies that any straight line l that passes through p and is not covered by the wedges is a separating line of S, otherwise RETURN a negative answer.

6. Sort the k wedges $(W_1, W_2, ...W_k)$ according to their angles into an ordered list $W'_1, W'_2, ..., W'_k$. Draw two semi-infinite supporting lines l_1 and l_2 of H from

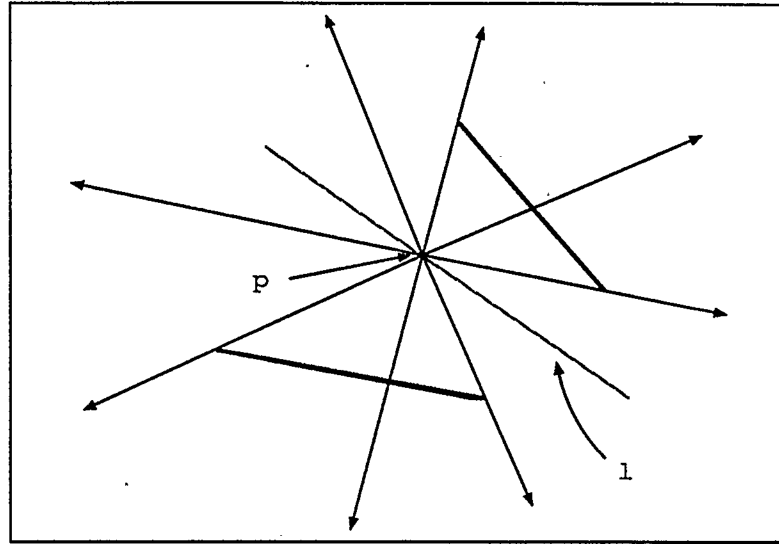


Figure 4.3: Detect separability via plane coverage.

p (a supporting line of a polygon P is a straight line that touches, yet not penetrate P), then these two lines l_1 and l_2 define a wedge W.

7. Walk counterclockwise around p and check the sequence of $W'_1, W'_2, ..., W'_k$ to see whether they cover the wedge W or not. If they do not then a separating line must exist, otherwise return a "No".

There are $2k$ endpoints. So steps 2, 3, 5 and 7 can be done in linear time. Step 1, 3 and 6 all perform sorting (on either k or $2k$ wedges), therefore require $O(k \log k)$ time. Thus the algorithm has a time complexity of $O(k \log k)$. We rephrase it in the following result:

Result 13 *Problem 7 can be solved in $O(k \log k)$ time, which is also optimal.*

Proof: The algorithm ensures an $O(k \log k)$ solution. We shall show in section 4.3 that this result is also optimal. □

4.2.3 Algorithm 2

Now that problem 7 is solved, combining result 12 and result 13 we have the following algorithm for problem 6:

1. Select an endpoint p_1 of S and run the above algorithm by substituting p_1 into the position of point p in the algorithm. If a separating line is found then stop, otherwise go to step 2.

2. Do the same thing for other endpoints of S, one at a time, until either (a) a separating line is found, or (b) all the $2k$ endpoints in the set of S have been checked, in which case a negative answer is returned.

As there are up to $2k$ endpoints needs to be checked, this algorithm has a worst-case time complexity of $O(k^2 \log k)$. As far as we know at this point, there is no algorithm that solves problem 6 faster. In fact, we conjecture that this is a lower bound for algorithms solving problem 6 (section 4.3). However, as we shall see immediately, a simple method can substantially improve the average-case performance.

4.2.4 The potato peeling method

We now introduce a solution to problem 6 that is based on *potato peeling*, an idea that deals with the "convex layers" and the "depths" of a given set of points.

Let S' be a set of k points in the plane (to distinguish from the set S of line segments). The *convex layers* of S' is the set of convex polygons obtained by iterating on the following process: compute the convex hull of S' and remove its vertices from S'. The *depth* of a test point within the convex layers of S' is defined to be the number of layers that enclose the test point, and the *depth* of S' is defined to be

the number of convex layers of S'. In [6], Chazelle gives an optimal algorithm that computes the layers of S', as well as the depths of all the given points in S', in $O(k \log k)$ time.

Suppose the set S' has d convex layers, with the outermost layer denoted by L_1 and the innermost layer denoted by L_d, the following fact is easily observed:

Result 14 *Let $p_i^1, p_i^2, ..., p_i^s$ (given by clockwise order) be the vertices on the convex layer L_i of S' $(1 \leq i \leq r)$, and let p be any point enclosed in L_i, the sequence $p_i^1, p_i^2, ..., p_i^s$ form a sorted list according to their polar angles with respect to p.*

Result 14 reveals that when using the previous algorithm to solve problem 6 (section 4.2.3), we may not need to repeat the sorting process in section 4.2.2 every time we check an endpoint for possible separating lines. In fact, let S' denote the set of the endpoints that are contained in the set S of line segments, the information of convex layers and depths obtained from "potato peeling" S' can be utilized in the process of separability detection. Following is the algorithm:

1. Use the algorithm given by Chazelle ([6]) to compute the convex layers of S' and the depths of all the points in S'. If S' has a depth d, $(d \geq k/c)$ (where c is a small constant, say 8), then proceed with the algorithm described in section 4.2.3. Otherwise go to step 2.

2. Start with an endpoint p_0 on the innermost convex layer L_d of S' and check for possible separating lines. By result 14, all the vertices on an enclosing convex layer L_i are already sorted by their polar angles with respect to p_0. Therefore a complete list of all the points in S', sorted by their polar angles with respect

to p, can be constructed by "merging" the d sublists that are already sorted. This takes $O(k \log d)$ time.

3. Do the above for all the endpoints that are of the same depth as p, then proceed to the next enclosing layer that is outside of it. Here the situation is slightly different. Consider a point p on layer L_i $(1 \leq i \leq d)$. We know that vertices on an enclosing layer L_j $(j < i)$ are already sorted by their polar angle with respect to p, but this is not the case for a layer L_s that is inside of L_i. The polar angles of vertices on L_s with respect to p form a *bimodal* function F (section 3.1), rather than being a sorted list itself. We need to find the two extrema of F thereby to break the vertices on L_s into two ordered lists. This requires $O(\log n_s)$ time in each occurance where a Fibonacci search is performed.

4. Keep on doing step 3 until either (a) a separating line is found, or (b) all the points on L_1, the outermost layer, have been processed.

There are up to $2k$ endpoints to be processed, which will require a running time of $O(k^2 \log d + kd \log k)$. Thus the worst-case time complexity of this algorithm is also $O(k^2 \log k)$ (there are $O(k)$ convex layers). On an average-case basis, however, this time complexity is less then that of the previous algorithm. Particularly, when S' has a constant number of convex layers (e.g., $k = 3$), we come up with an $O(k^2)$ performance.

4.3 Lower Bound Analysis

We now establish a lower bound on the time complexity of algorithms solving problem 7 by using the algebraic decision-tree model (see section 1.1.5). Consider problem 7 with the following restrictions:

- all l_i ($1 \leq i \leq k$) are on x-axis and are of unit length.
- the leftmost and rightmost endpoint of S are at $(0,0)$ and $(k,0)$ respectively.
- the observing point p is selected at the positive infinite of y-axis.

With the above restrictions we are able to represent each segment l_i by its left endpoint p_i, thus any set S of k line segments satisfying the above restrictions can be viewed as a point $(x_1, x_2, ..., x_k)$ in E^k, where $(x_1, x_2, ..., x_k)$ is a permutation of $(p_1, p_2, ..., p_k)$. Let M be the "non-separable" membership set (i.e., M contains all the points in E^k whose corresponding line segment set in E^2 are non-separable). We claim that M contains $k!$ disjoint connected components, with each component being a single point. Indeed, every point $m \subset M$ in E^k matches an arrangement of $(l_1, l_2, ..., l_k)$ where no overlapping is allowed except at their endpoints. Thus any permutation π of $1, 2, ..., k$ corresponds to a point $m_\pi \subset M$ in E^k.

It is clear that

$$M = \cup_{all \pi} m_\pi$$

where the m_π's are disjoint and connected, and there are altogether $k!$ m_π's. According to result 1 we have

Result 15 *In the algebraic decision tree model any algorithm that solves problem 7 requires $O(k \log k)$ tests.*

The lower bound for problem 6, however, is more difficult to work out.

4.4 Separating Convex Polygons

Now consider the separability detection problem for a set of convex polygons.

Problem 8 *Given a set S of k convex n-gons $P_1, P_2, ..., P_k$ in the plane, find out whether there is a separating line for S.*

As polygons can be viewed as collections of line segments, any algorithm designed for problem 6 can be directly used for problem 8.

4.4.1 A straightforward solution

If we consider the set S of k convex n-gons as a set S' of nk line segments, then by using the algorithms described above, we immediately have a solution that runs in $O(n^2k^2(\log n + \log k))$ time.

However, this solution can be applied to any arbitrary collection of planar line segments, and the convexity associated with our input polygons is not utilized. We need to explore algorithms that are more specific and more efficient.

4.4.2 A solution based on convexity

The following result, similar to result 12, is easily observed:

Result 16 *If a set S of k convex n-gons $P_1, P_2, ..., P_k$ is separable, then there must be a separating line l', which is a common tangent of two of the given polygons, say P_i and P_j $(1 \leq i < j \leq k)$.*

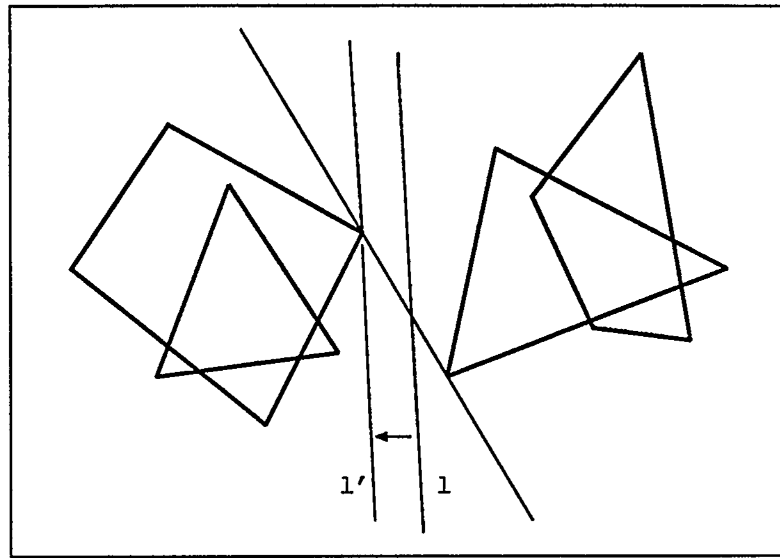


Figure 4.4: Separate via polygon vertices

Proof: Let l be a separating line of S (figure 4.4). If l does not go through any of the polygon vertices we just translate it sideways until it hits a vertex that is closest in l's direction of moving. By doing this we obtain a separating line l' that passes through a certain polygon vertex, say p. Let P be the polygon that defines p. We then "roll" the line l along the boundary of P in a clockwise direction, until it hits another vertex, say q, of another polygon Q. At this time, we come up with a separating line that is a common tangent of polygons P and Q. □

Based on result 16, we have the following algorithm that solves problem 8:

1. Construct a common tangent l defined by two polygons of S.

2. Test whether l is a separating line or not. If "Yes" then stop, otherwise go to step 3.

3. Keep on constructing new common tangents and do the above test until either (*a*) a separating line is found, or (*b*) all the pairwise common tangents defined by polygons in S are tested, in which case S is non-separable.

The given k convex polygons will define four common tangents pairwise. So S defines a total number of $O(k^2)$ common tangents. To construct one of them takes $O(\log n)$ time ([30], recall that S's elements are n-gons), hence computing the $O(k^2)$ common tangents needs time $O(k^2 \log n)$. Also for each of these tangents it takes $O(k \log n)$ to check whether it is a separating line or not (see result 5), thus this algorithm will run in $O(k^3 \log n)$ time and $O(k^2)$ space.

This is a better result than the one mentioned above. However, in applications where k is big (there is a great number of polygons) and n is relatively small (each polygon has a limited number of vertices), this algorithm will lose its advantage. We therefore need to find some alternative.

4.4.3 An alternative solution

We need to find a solution to problem 8 that works better in cases where the above algorithm's performance degrades. The "plane coverage" method used in solving the line segments separation problem provides some inspiration.

Recall that when trying to detect a separating line for a set S of k line segments (problem 6), we started with solving its restricted version, problem 7. Then by applying the "plane coverage" algorithm to all the points in S, we were able to find whether S has a separating line or not. In a similar manner we now construct a restricted version for problem 8

Problem 9 *Given a set S of k convex n-gons $P_1, P_2, ..., P_k$ and a point p, decide whether there is a separating line of S that goes through p.*

Following is a plane coverage algorithm that solves problem 9

1. Draw $2k$ supporting lines from p to the k polygons $P_1, P_2, ..., P_k$. Let l_i^1 and l_i^2 be the two supporting lines from p to polygon P_i, we then have k pairs of supporting lines $(l_1^1, l_1^2), (l_2^1, l_2^2), ..., (l_k^1, l_k^2)$ which define k wedges, namely $W_1, W_2, ..., W_k$. We also create the inverses of these wedges.

2. Sort the $2k$ wedges ($W_1, W_2, ..., W_k$ and their inverses) by their angles and form an ordered list of $2k$ wedges $W'_1, W'_2, ..., W'_{2k}$.

3. Walk counterclockwise around p and check the sequence of $W'_1, W'_2, ..., W'_{2k}$ to see whether or not they cover the entire plane (figure 4.5). If they do then return "yes", otherwise return "no".

Computing the supporting line from a given point to a convex n-gon requires $O(\log n)$ time, so step 1 can be done in time $O(kn \log n)$. Step 2 sorts $2k$ elements and takes $O(k \log k)$ time. Step 3 is a simple walk-through that can be done in linear time. Thus the algorithm runs in a total time of $O(k(\log k + \log n))$. This is summarized in the following result:

Result 17 $O(k(\log k + \log n))$ *time is sufficient to solve problem 9.*

Combining results 16 and 17 we have the following algorithm for problem 8:

1. Select a vertex p_0 form the set S of k convex n-gons, and run the above algorithm by substituting p_0 into the position of point p. If a separating line is found then stop, otherwise go to step 2.

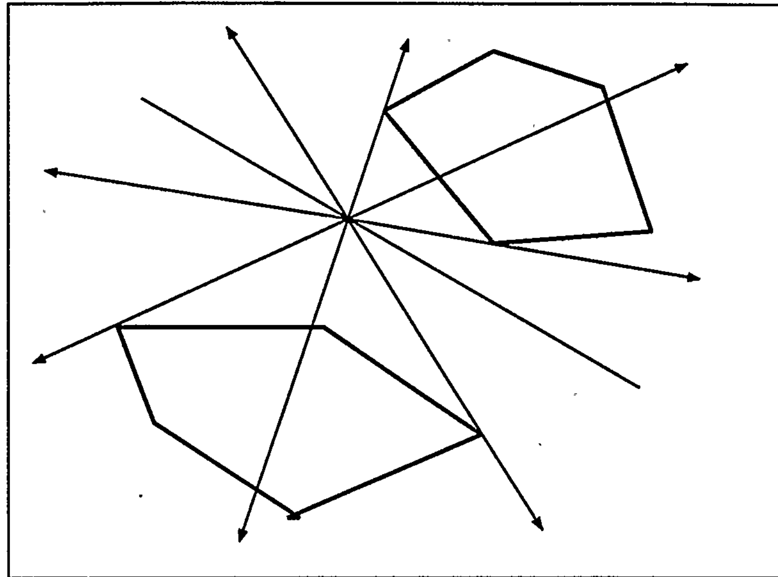


Figure 4.5: Detect separability via plane coverage.

2. Do the same thing for other vertices of S, one by one, until either (a) a separating line is found, or (b) all of the nk vertices of S are processed.

As there are altogether kn vertices in S, this algorithm requires, in the worst case, $O(k^2n(\log n + \log k))$ time and $O(nk)$ space.

The algorithm by itself is not necessarily better then the previous one. However, it is able to solve the problem faster in cases where the previous algorithm loses performance. Thus by combining the two algorithms together we are able to handle extreme cases of either k being too big (compared to n) or the other way around. The potato peeling method we used to detect separability for a set of line segments does not apply here.

Chapter 5

Voronoi Diagram for Convex Polygons

The computation of Voronoi Diagrams has been an active research topic in computational geometry in the past decade [2]. Much of the earlier work, and some of the current work, concerns the Voronoi Diagram of a discrete set of points. This has been extended by Kirkpatrik [19], Lee and Drysdale [22], and others to cover a collection of two-dimensional objects such as line segments, circular arcs and polygons.

Some applications in two-dimensional geometric modeling require the computation of Voronoi Diagrams on a set of convex polygons. We shall study this problem in this chapter.

Our discussion is organized as follows: Section 5.1 gives preliminaries including Voronoi diagrams of planar point sets, definitions and notations. Section 5.2 discusses how to bisect two given straight line segments. Section 5.3 concerns bisecting two convex polygons, which is fundamental to construct the Voronoi diagram for a set of convex polygons. At last in section 5.4, we first consider the nature of the Voronoi diagram construction problem for convex polygons, followed by an analysis of several inherent obstacles that make it a particularly difficult task to find a solution. Having demonstrated this, we turn to solve a restricted version of the problem: the input set being non-intersecting translates of a given convex polygon.

5.1 Preliminaries

5.1.1 The Voronoi Diagram

The Voronoi Diagram solves the following proximity problem.

Problem 10 *(Loci of Proximity). Given a set S of n points in the plane, for each point p_i in S find the locus of points (x,y) in the plane that are closer to p_i than to any other points of S.*

Intuitively, the solution of the above problem is a partition of the plane into regions associated with the given points. For two points p_i and p_j, the set of points closer to p_i than to p_j is the halfplane containing p_i that is defined by the perpendicular bisector of $\overline{p_i p_j}$. Let us denote this halfplane by $H(p_i, p_j)$. The locus of points closer to p_i than to any other point, which we denote by $V(i)$, is the intersection of $n-1$ halfplanes, and it is a convex polygonal region [34] having no more than $n-1$ sides, that is,

$$V(i) = \bigcap_{i \neq j} H(p_i, p_j). \tag{5.1}$$

$V(i)$ is called the *Voronoi polygon associated with p_i*. Figure 5.1 is a Voronoi polygon [34].

There is a region for each of the n points, and the n regions form a planar diagram which we shall call the *Voronoi diagram*, denoted by $Vor(s)$ [30]. An example is shown in figure 5.2. The vertices of the diagram are *Voronoi vertices*, and the edges of the diagram are *Voronoi edges.*

Regarding Voronoi diagram construction, there is the following very important result:

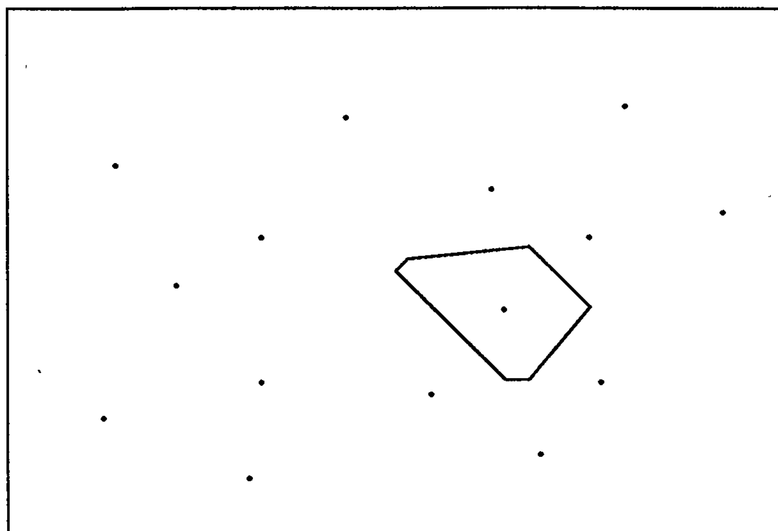


Figure 5.1: Voronoi polygon.

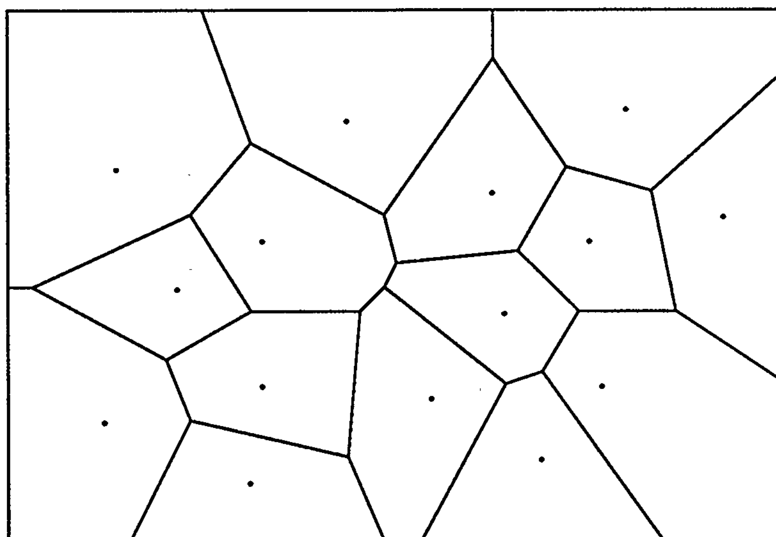


Figure 5.2: Voronoi diagram.

Result 18 *(Lower bound) The Voronoi diagram $V(S)$ for a set S of n points can be constructed in $O(n \log n)$ time, which is optimal.*

Many algorithms have been developed to achieve this optimal result. It is interesting to note that although problem 10 seems to be "more difficult" than the *closest pair problem* ([30]) in terms of degree of difficulty, they nonetheless have the same level of complexity.

5.1.2 Definitions and notations

The Voronoi diagram of a set of elements is defined in terms of halfplanes where a measure of distance is used. Following are notations and definitions used in our discussion. Note that when we mention a line segment l, we are talking about a *closed* line segment.

A *closed line segment* $\overline{ab}$ is the union of two endpoints a and b and the open line segment (a, b). Points and open line segments are called *elements.*

The *distance* between a point p and a point q is denoted $d(p, q)$. The *distance* between a point p and a nonempty set X, denoted $d(p, X)$, is $d(p, X) = min\{d(p, q) : q \in X\}$. In particular, the distance $d(p, \overline{ab})$ between a point p and a closed segment $\overline{ab}$ is the distance between the point p and its projection (defined in a normal sense) onto $\overline{ab}$ if the projection belongs to $\overline{ab}$. Otherwise it is defined to be the minimum of $d(p, a)$ and $d(p, b)$.

The *bisector* $B(e_i, e_j)$ of two elements e_i and e_j is the locus of points equidistant form e_i and e_j. The bisector $B(X, Y)$ of two sets of elements X and Y is the locus of points equidistant from X and Y.

The *halfplane* $H(e_i, e_j)$ is the set of points closer to element e_i than to element e_j. Note that the boundary of a halfplane $H(e_i, e_j)$ is the bisector $B(e_i, e_j)$.

5.2 Bisecting Two Line Segments

A polygon can be considered as a collection of edges. Therefore before constructing the Voronoi Diagram for a set S of convex polygons, we give an essential algorithm that solves the following problem.

Problem 11 *(Bisecting two straight line segments) Given two non-intersecting line segments s_1 and s_2, construct their bisector $B(s_1, s_2)$.*

We shall start with bisecting a line segment and a point (see also [24], [25]). The following result is important:

Result 19 *The bisector $B(p, \overline{ab})$ of a point p and a line segments $\overline{ab}$ can be found in constant time.*

Proof: We show this by a simple algorithm (figure 5.3)

Let l be the straight line that contains segment $\overline{ab}$. Let l_a (l_b) be the straight line that is perpendicular to l and passes through a (b). Construct the parabola $B(p, l)$ (requires constant time), and assume that $B(p, l)$ intersects l_a and l_b at p_a and p_b respectively. We obtain the bisector $B(p, \overline{ab})$ which consists of three parts: (a) the semi-infinite ray B_1 which is a part of $B(p, a)$. (b) the parabola sector B_2 of parabola $B(p, l)$ that is in between l_a and l_b. (c) the semi-infinite ray B_3 which is a part of $B(p, b)$. Clearly all B_1, B_2, B_3 can be computed in constant time, so result 19 is proved. □

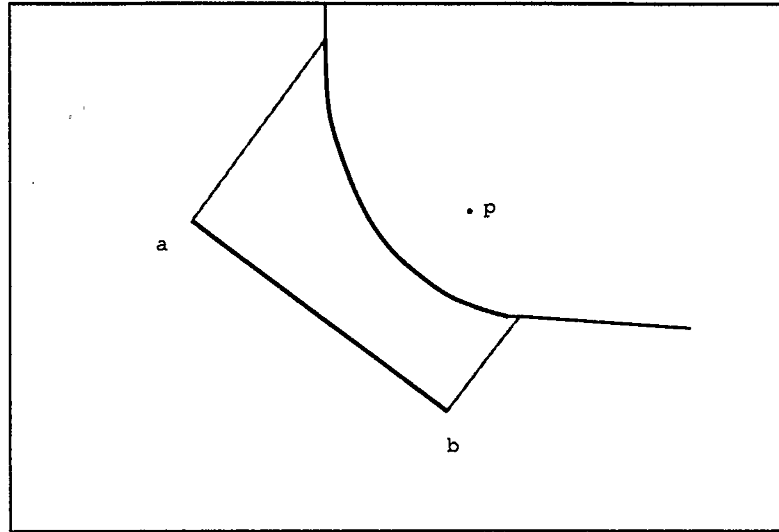


Figure 5.3: Bisecting a line segment and a point.

The following is a simple extension of result 19:

Result 20 *The bisector $B(\overline{ab}, \overline{cd})$ of two line segments $\overline{ab}$ and $\overline{cd}$ can be found in constant time.*

Proof: We present an algorithm that achieves the above result.

Let l and l' be the straight lines that contain $\overline{ab}$ and $\overline{cd}$ respectively. Let l_a (l_b) be the straight line that is perpendicular to l and passes through a (b), let l'_c (l'_d) be the straight line that is perpendicular to l' and passes through c (d).

If l and l' are parallel then result 20 is straightforward (see figure 5.4(a)). If they are not parallel then find their intersection p. Draw the bisector ll of $\angle apc$, and let ll intersect l_a, l_b, l'_c, l'_d at p_a, p_b, p_c, p_d respectively (figure 5.4(b)), then we have the following cases (without loss of generality, assume that all theses points are different): (a) $\overline{p_a p_b}$ and $\overline{p_c p_d}$ do not intersect. (b) $\overline{p_a p_b}$ and $\overline{p_c p_d}$ "bite" into each other (i.e., they overlap but no one is contained in the other). (c) One of the two

segments is contained in the other.

Figure 5.4(b) shows that case (b) can be computed in constant time, and this is true for the other two cases as well. Therefore the computation of bisector $B(\overline{ab}, \overline{cd})$ needs constant time. □

5.3 The Bisector of two convex polygons

In this section we shall consider the problem of constructing the bisector of two disjoint convex polygons, which is a basic operation in constructing the Voronoi diagram for a set of convex polygons.

Problem 12 *(Computing the bisector of two disjoint convex polygons). Given a convex m-gon P and a convex n-gon Q, compute their bisector $B(P, Q)$.*

Intuitively the bisector of two convex polygons will partition the plane into two halfplanes. However, unlike bisecting two points, the boundary that divides the two halfplanes is not a straight line. Indeed it is a sequence of interweaved line segments and parabolic sections. This can easily be seen by looking at figure 5.4(b).

How quickly we can solve problem 12 is of fundamental importance to the construction of the Vonoroi diagram for a set of convex polygons. In fact, we have the following result:

Result 21 *It requires $O(m + n)$ to solve problem 12, which is optimal in the worst case.*

Proof: The following algorithm solves problem 12 in $O(m + n)$ time.

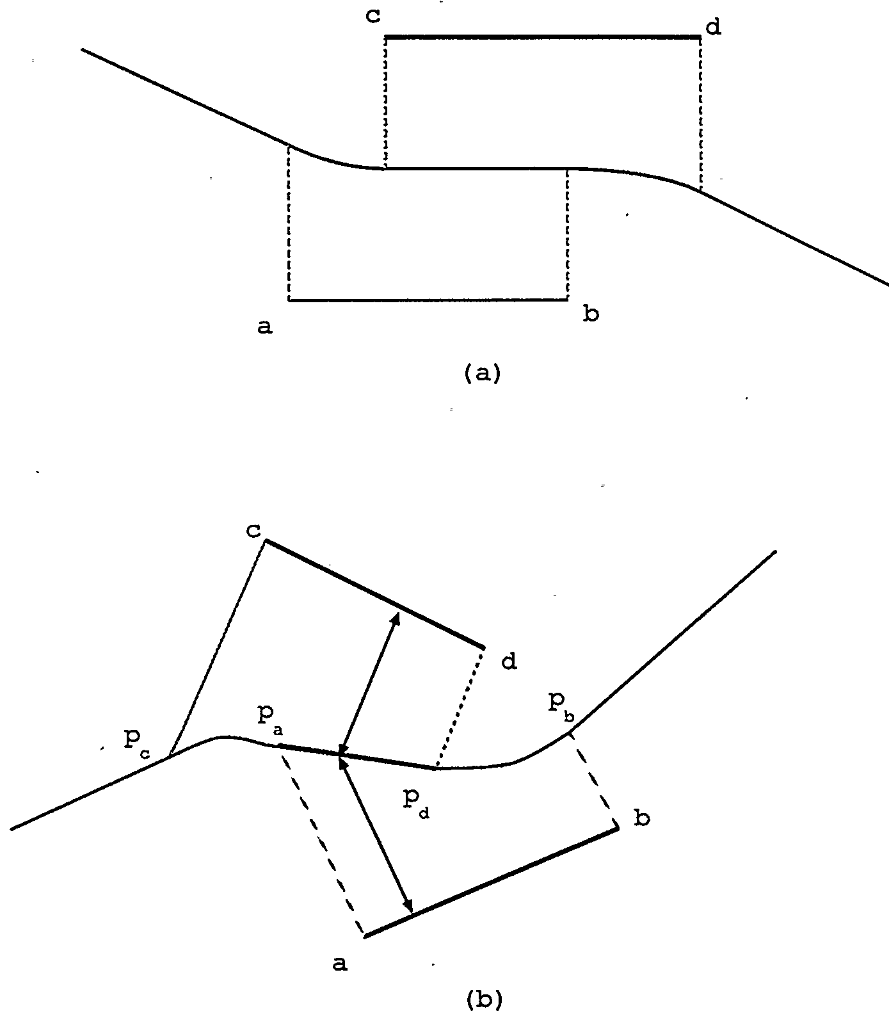


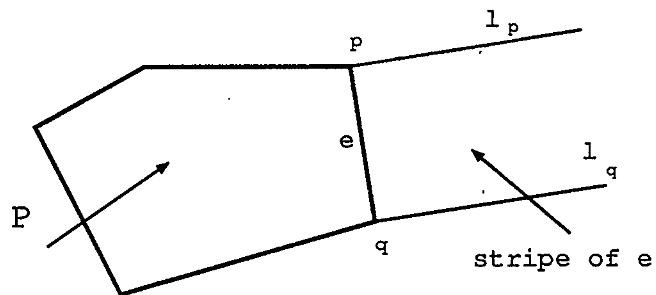
Figure 5.4: Bisecting two line segments that are (a): parallel, and (b): otherwise.

We begin with a definition. Given a convex n-gon P and an edge e of P, let p and q be e's two endpoints. Draw two straight lines that are perpendicular to e and grows outwardly (with respect to P): l_p from p and l_q from q (figure 5.5 (a)). The open area that is defined by e, l_p and l_q is called the *stripe of edge e for the convex polygon P*, or sometimes the *stripe of e* when there is no confusion. By definition, a convex n-gon P should have n stripes.

The first step of the algorithm is to find the supporting segments t_1, t_2 of P and Q (by a supporting segment t of a polygon P we mean that t touches to, but not penetrates through P), which needs $O(m+n)$ time. Obviously $B(P,Q)$ will contain the semi-infinite perpendicular bisector r_1 of t_1 and r_2 of t_2. We shall construct $B(P,Q)$ by walking from r_1 to r_2.

We observe that t_1 and t_2 cut each of P and Q into two polygonal chains. Let C_1 (consisting of the vertex list $(p_1, p_2, ..., p_r)$) be the polygonal chain of P that is convex to Q, and similarly let C_2 (consisting of the vertex list $(q_1, q_2, ..., q_s)$) be the polygonal chain of Q that is convex to P. Our algorithm is based on the chains C_1 and C_2 (figure 5.5 (b)).

The construction process starts with the semi-infinite bisector r_1. It is useful to refer to the example of figure 5.5 (b), where, for simplicity, the edge e_i is shown by its index i. Imagine a point z on r_1, moving down from infinity. Initially z lies on the bisector of points a and b. When it travels downward, it will enter the stripes s_1 of 1 and s_4 of 4 eventually. When it enters either of them a new situation will come up and a new track has to be taken. There are the following cases: (a) It hits s_1 first at point p. This means that z is now closer to edge 1 than it is to point a (see section 5.1 for the definitions of different types of distances), so it must take a new



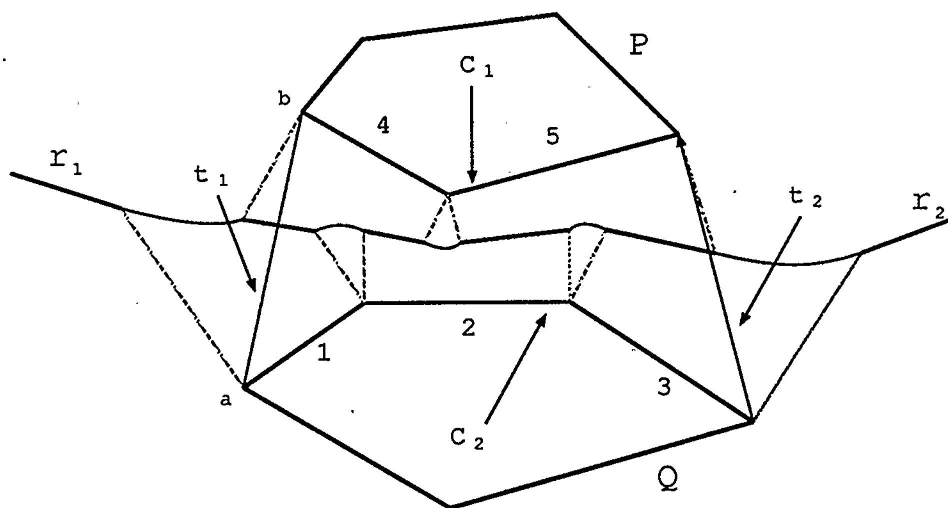


Figure 5.5: (*a*) The stripe of e for the convex polygon P. (*b*) Bisecting convex polygons P and Q.

route along the $b-1$ bisector. (b) It hits s_4 first at point q. This means that z is now closer to edge 4 than it is to point b, so it must be moved to the $a-4$ bisector. (c) It hits s_1 and s_4 together at the same point. This means that z is now closer to edges 1 and 4 than to points a and b, so the $1-4$ bisector needs to be assumed.

We repeat this process until all the edges in both polygonal chains C_1 and C_2 are covered. Since the edges on C_1, C_2 are already given in order, this process can be finished in linear time. □

5.4 Constructing the Voronoi Diagram

We now consider problem 10 with the input being a set of convex polygons.

Problem 13 *(Loci of proximity – 2). Given a set S of k non-intersecting convex n-gons $P_1, P_2, ..., P_k$, for each polygon P_i what is the locus of points (x, y) in the plane that are closer to P_i than to any other polygon of S?*

To solve this problem we need to construct the *extended* Voronoi diagram (Problem 14). The Voronoi diagram for a set of convex polygons is defined in a similar way as with a set of points. The plane is partitioned into k regions, called *Voronoi regions*, each of which is associated with a particular polygon. The regions together define a planar diagram which we shall refer to as the *Voronoi diagram* of the set S of convex polygons. Two neighboring Voronoi regions define a chain of interwaved line segments and parabolic sections between them. This chain we shall call a Voronoi edge. The intersection points of Voronoi edges are called Voronoi vertices.

Problem 14 *(Voronoi diagram construction) Given a set S of k non-intersecting convex n-gons $P_1, P_2, ..., P_k$, construct their Voronoi diagram $Vor(S)$.*

Just as the perpendicular bisectors are fundamental in constructing the Voronoi diagram for a set of points, so are the bisectors of two convex polygons in constructing the Voronoi diagram for a set of convex polygons. The difference is that the former are always straight lines whereas the latter are not. This, as we shall see, presents some special difficulty to the task of Voronoi diagram construction. The bisectors of two convex polygons will hereafter for simplicity be referred to as the *dividing chains* between their defining polygons.

5.4.1 Inherent difficulties

The Voronoi diagram of a set of k non-intersecting convex polygons is similar to that of a planar point set. Given two convex polygons P_i and P_j, the set of points closer to P_i than to P_j is the halfplane containing P_i that is defined by the dividing chain of P_i and P_j. Let us denote this halfplane by $H'(i,j)$. By definition, the Voronoi region $V(i)$ associated with P_i is the intersection of $k-1$ halfplanes:

$$V(i) = \bigcap_{i \neq j} H'(P_i, P_j). \tag{5.2}$$

However, since the dividing chains are not straight lines, the following problems arise:

- The Voronoi regions may be non-convex.
- The dividing chains consist of straight line segments and parabolic sections. This brings problems in representation and storage.
- The dividing chains may not be monotonic, which precludes the use of binary search in finding their intersections, thus making it hard to determine the Voronoi vertices.

- Two dividing chains may intersect at several points. This makes the construction of the Voronoi diagram much harder.

- Although there are k convex polygons, a Voronoi region associated with a particular convex polygon may have more than $k-1$ Voronoi edges (see discussions below).

With the difficulties listed above, we further look at what may happen when constructing the Voronoi diagram.

5.4.2 Features of Voronoi regions

We give some important results concerning the Voronoi diagram for a set of convex polygons.

Result 22 *(Length of dividing chain). Given two non-intersecting convex n-gons, the length of their dividing chain (i.e. the total number of line segments and parabolic sections contained in the chain) is $O(n)$.*

Proof: This is illustrated in the dividing chain construction process in section 5.3. □

Result 23 *(Continuity of Voronoi region) The Voronoi region associated with a given convex polygon is a connected area in the plane.*

Proof: For a contradiction, suppose that the Voronoi region $V(i)$ associated with convex polygon P_i has two disjoint areas A and A' (see figure 5.6 (a)).

By definition, $V(i)$ must contain P_i. Suppose P_i is contained in A. As A and A' share no common point, we can draw a line c (not necessarily straight) that separates

the two areas. Let p and a be the point on P_i and A' respectively that are closest to each other, and let b be the point where $\overline{pa}$ intersects c. Clearly if a is closer to P_i than it is to any other polygons, so is c. Hence we have a contradiction which shows that $V(i)$ has to be a connected area. □

Result 24 *(Number of edges for a Voronoi region). Any given Voronoi region can have at most $2k-3$ edges (where $k>1$ is the number of convex polygons), and two adjacent Voronoi regions may have up to $k-1$ common edges.*

Proof: As k convex polygons define k Voronoi regions, each one of them can have at most $k-1$ neighboring regions. Considering a particular region R, we prove the result by induction: when $k=2$, there are two regions and each of them has one edge, the conclusion holds. Assume that for the case of $k-1$ regions the conclusion holds, then when there are k Voronoi regions we have the following possibilities: (1) R shares at most one common edge with each of its neighboring regions, in which case we are done. (2) if R shares two edges e and e' with another Voronoi region R', then R's neighboring regions (aside from R') are divided into two groups: those that are in between e and e', and those that are not (this group may be empty). From our assumption we know that R can have at most $2(k-2)-1+2=2k-3$ edges (figure 5.6 (b)). The case where R and R' share more than two edges can be proved in a similar manner. Therefore the first half of the result is proven.

The second half of the result is obvious in that if the regions R and R' have k common edges, then there must be at least another $k-1$ regions (aside from R and R') to separate these k edges, which means that there are at least $k+1$ Voronoi regions in all, a contradiction. □

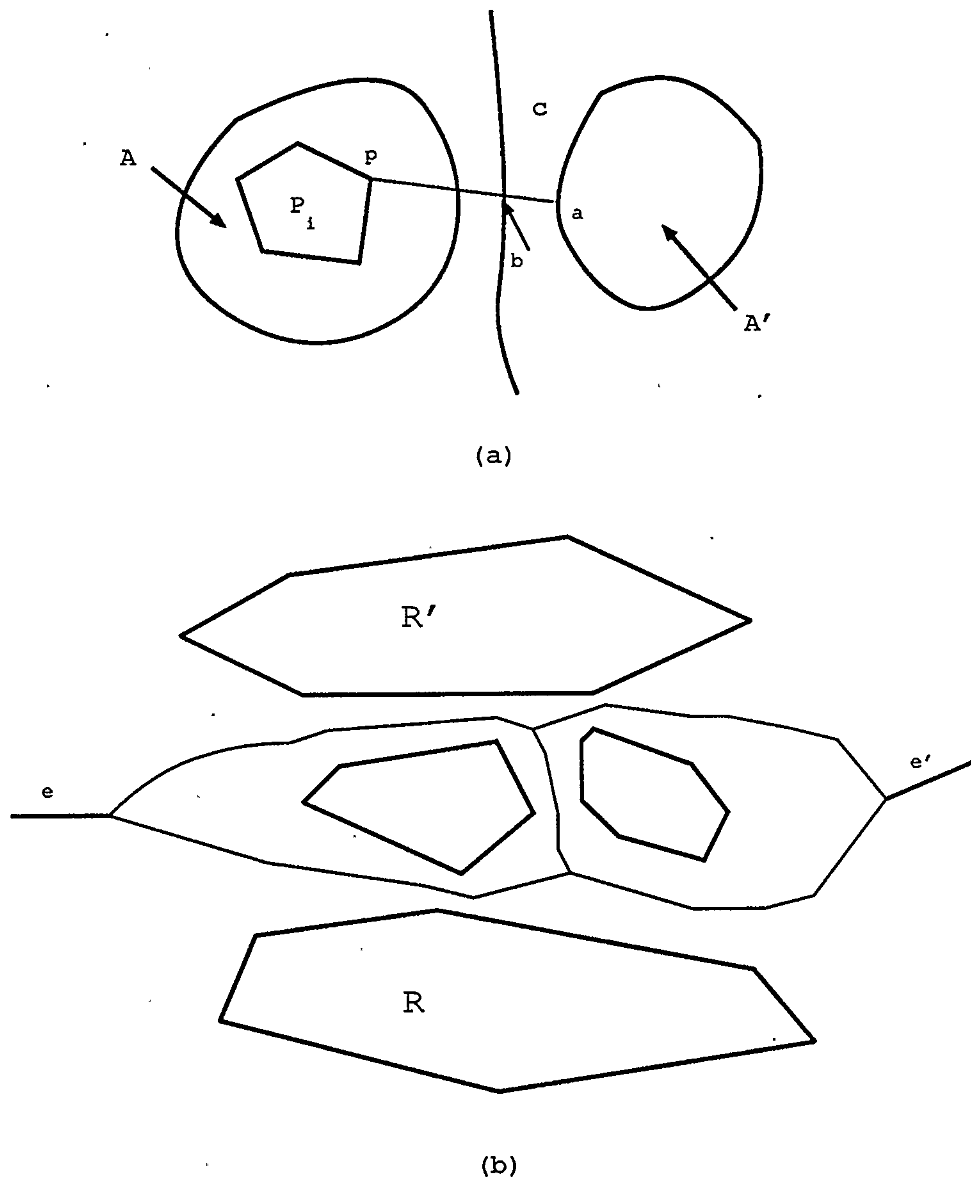


Figure 5.6: (*a*) A Voronoi region is a connected area. (*b*) Adding another polygon into the picture.

The difficulties listed in section 5.4.1 already make it hard to find a solution for problem 14, and this difficulty is increased by result 24. As dividing chains are not necessary monotonic, binary search cannot be used in calculating their intersections. Also as two dividing chains may have more than one common point, this leads to a quadratic worst-case time requirement to find all the intersections of two dividing chains. Still considering the fact that we need to compute the intersection of $k-1$ dividing chains in order to construct a Voronoi region (result 22), and that two adjacent Voronoi regions may have up to $k-1$ common edges, we believe, at this stage, that the construction of a Voronoi diagram can become formidably expensive.

Realizing this difficulty, we shall in the following consider a restricted version of the Voronoi diagram construction problem.

Problem 15 *Restricted Voronoi diagram construction Given a set S of k non-intersecting translates $P_1, P_2, ..., P_k$ of a convex n-gon P, construct their Voronoi diagram $Vor(S)$.*

5.4.3 A direct solution

A naive (almost brutal) approach to solve problem 15 is to construct its individual regions one at a time. The difficulties outlined in section 5.4.1 still exist, yet result 24 in section 5.4.2 will no longer apply. In fact, any Voronoi region constructed for problem 15 has *at most $k-1$ edges.*

This can be shown by a contradiction. Consider two of P's translates P_i and P_j. Without loss of generality, suppose that their Voronoi regions $V(i)$ and $V(j)$ have two common edges e_1 and e_2 (figure 5.7). Since e_1 and e_2 are non-consecutive, there must be an edge e_r in between that separates $V(i)$ and another translate P_r's

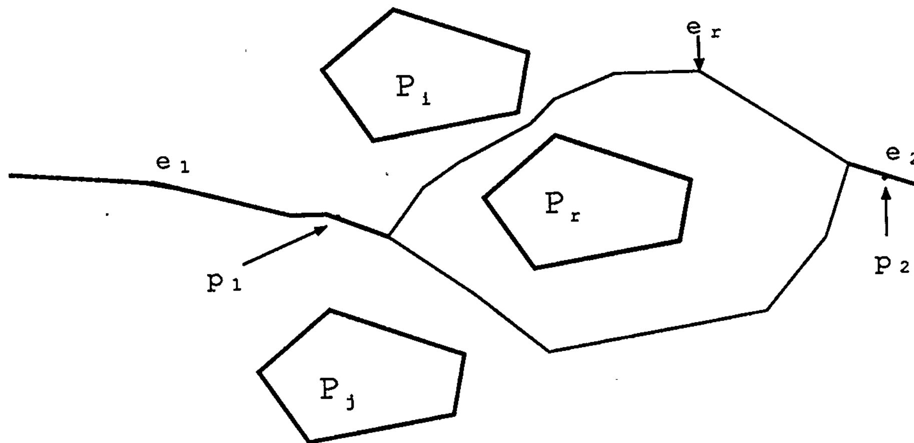


Figure 5.7: (a) Any Voronoi region has at most $k-1$ edges.

Voronoi region $V(r)$. Let p_1 be a point on e_1 and p_2 be a point on e_2, then by definition the distance from p_1 to P_i (and P_j) is shorter than that from p_1 to P_r, and the same thing holds for p_2. However, we find that this cannot be true in the obvious way: considering the case of P_i, P_j and P_r being single points, then we are creating a situation that the perimeter of a rhombus is less than twice the total distance from a inner point to two of the rhombus's vertices. This contradiction means that two adjacent Voronoi regions cannot have more than one common edge, hence each region will have at most $k-1$ edges. □

We also observe the following fact regarding the computation of Voronoi regions: A Voronoi region $V(i)$ is the intersection of $k-1$ halfplanes defined by the dividing chains $B(P_i, P_j)$ $(i \neq j)$. For simplicity we introduce two definitions here (1) Given a dividing chain C, the two convex polygons that define C are called the *defining polygons* of C. (2) Two dividing chains are said to be *coaxial* if they share a common defining polygon. In the case where the input set consists of translates of a given convex polygon, it is obvious that any two coaxial dividing chains can have *at most*

one intersection point.

A direct consequence of the above result is that binary search can be used to compute the intersection of two coaxial dividing chains. $O(\log n)$ comparisons are sufficient for the computation, and for each comparison we need to find whether a certain point is to the "left" or "right" side of a given dividing chain, which requires $O(\log n)$ time as well. So we have:

Result 25 *Finding the intersection of two coaxial dividing chains takes* $O(\log^2 n)$ *time (provided the inputs is a set of translates of a given convex polygon).*

Now we shall solve problem 15 by constructing the Voronoi regions one at a time. The concern here is to find the intersection of $k-1$ halfplanes defined by $k-1$ coaxial dividing chains. For convenience of discussion, we shall hereinafter call a planar region defined by i coaxial dividing chains a *i-chain region*, and call an edge of such a region a *chain side*.

We use the divide-and-conquer approach. The input are $k-1$ halfplanes defined by coaxial dividing chains. The output will be their intersection, a planar region that has at most $k-1$ chain sides. The algorithm is as follows.

1. Partition the half-plane into two sets of approximately equal sizes.

2. Recursively form the intersection of the half-planes in each subset.

3. Merge the subproblem solutions by intersecting the two resulting regions.

Let $T(k)$ denote the time used to form the intersection of k halfplanes by this algorithm, we have:

$$T(k) = 2T(k/2) + t. \tag{5.3}$$

where t denotes the time needed for the merging step.

The time t can be found using an approach due to O'Rourke et. al [27]. The original idea was based on the technique of "edge advancing", and was proposed for computing the intersection of two convex polygons. However, it also applies here. Given two k-chain regions, it will take $O(k)$ edge advancing steps to compute their overlapped area, where each step needs to check the intersection of two chain sides. By result 25 we know that $O(\log^2 k)$ time is required to find the intersection of two dividing chains, so to intersect $k-1$ k-chain regions takes $O(k\log^2 n)$ time where n is the number of vertices on an input polygon. Also constructing the $k-1$ dividing chains will take $O(kn)$ time, hence $t = O(kn + k\log^2 n)$. substitute this into equation 5.3 we have:

$$T(k) = 2T(k/2) + O(kn + k\log^2 n) = O(k\log k(n + \log^2 n)). \quad (5.4)$$

Equation 5.4 gives the time needed to compute one Voronoi region. To solve problem 15 we need to construct k Voronoi regions. So the algorithm will run in $O(kT(k))$ time which is summarized in the following:

Result 26 *Problem 15 can be solved in $O(k^2\log k(n + \log^2 n))$ time, which is equivalent to $O(k^2 n\log k)$.*

5.4.4 A divide-and-conquer solution

A better solution for problem 13 can be obtained by using the divide-and-conquer approach.

The method we shall use is similar to the Voronoi diagram construction algorithm devised by Preparata and Shamos [30]. Some modifications are made so that their

technique can be applied to the convex polygon situation. The method is based on the following crucial result:

Result 27 *Given a set S of k points $p_1, p_2, ..., p_k$ on the plane. Let S_1, S_2 be a partition that is linearly separated (which means if more than one point belongs to the separating line, all of these are assigned to the same set of the partition) of S, and let $\sigma(S_1, S_2)$ denote the set of Voronoi edges that are shared by pairs of Voronoi regions $V(i)$ and $V(j)$ of $Vor(S)$, for $p_i \in S_1$ and $p_j \in S_2$. It holds that $\sigma(S_1, S_2)$ consists of a single monotone chain.*

The above result is a prerequisite for the correct operation of their algorithm. Therefore, before applying their method we need to ensure that the theorem holds for our situation, i.e. a set S of k translates $P_1, P_2, ..., P_k$ of a given convex polygon P. The following points we should note:

- For a set of convex polygons, *linearly separated partition* means that if more than one polygon intersects the separating line, all of these are assigned to the same set of the partition.

- For a given set S of convex polygons, we can rotate both the x-axis and the y-axis simultaneously by a same angle (a rotation of the coordinate system), and this will not affect the Voronoi diagram of S.

According to the above points, we can first rotate both coordinate axes by a certain degree, thereby to "scatter" the set S of convex polygons horizontally (with respect to the x-axis). Then we scan through the polygons to find a vertical line that linearly separates S into two subsets of approximately equal sizes. The scan will take linear

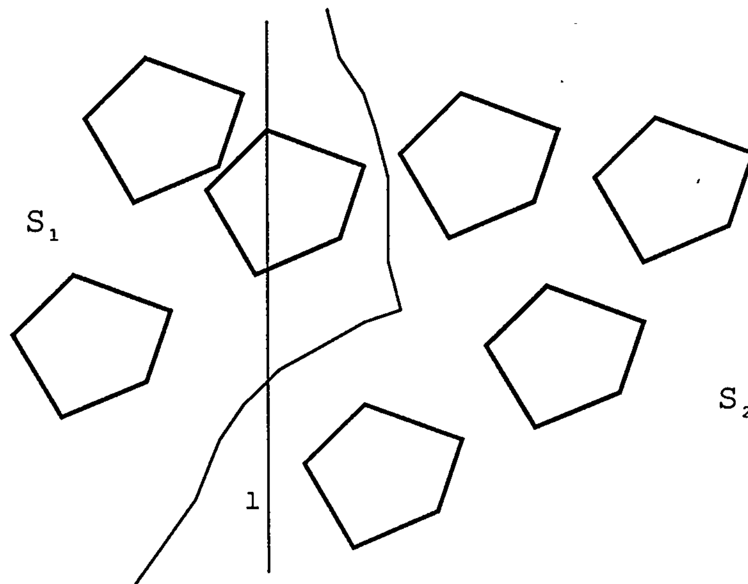


Figure 5.8: S_1 and S_2 are separated by a single monotone chain.

time. The correctness of result 27 for this situation can be proved using a similar argument as in [30]. An illustration is shown in figure 5.4.4. The algorithm goes as follows:

1. Use a vertical line l to linearly separate the set S into two subsets S_1 and S_2 of approximately equal sizes.

2. Construct the Voronoi diagrams $Vor(S_1)$ and $Vor(S_2)$ of S_1 and S_2 recursively.

3. Merge $Vor(S_1)$ and $Vor(S_2)$ to get $Vor(S)$.

Step 1 can be done by first sorting $P_1, P_2, \ldots P_k$ according the x-coordinates of their rightmost vertices, then scanning through the sorted list to determine the separating line. This requires $O(k \log k)$ time. What remains is to find an efficient way to merge the Voronoi diagrams obtained from the two subsets. An elegant merge algorithm

is provided in [30] which, as indicated above, can be applied to serve our purpose. For the sake of space we shall not repeat its lengthy detail here. The basic idea is to construct the chain $\sigma(S_1, S_2)$ (which is monotone) that separates S_1 and S_2. By definition, $\sigma(S_1, S_2)$ is the collection of Voronoi edges that are shared by Voronoi regions of polygons in S_1 and S_2. Let π_l be the halfplane that is to the left of $\sigma(S_1, S_2)$ and π_r be the one to the right, then the Voronoi diagram $Vor(S)$ is the union of $Vor(S_1) \cap \pi_l$ and $Vor(S_2) \cap \pi_r$. This observation ensures that when $\sigma(S_1, S_2)$ is available, $Vor(S)$ can be obtained by discarding all edges of $Vor(S_1)$ that lie to its right and all edges of $Vor(S_2)$ that lie to its left.

$\sigma(S_1, S_2)$ has two semi-infinite rays at its two ends, called the *upper ray* and the *lower ray* respectively. The algorithm starts from a point on one of them, say the upper ray, and takes $O(k)$ "moves" or "steps" to reach the lower ray, thereby to construct the separating chain. As each move involves computing a dividing chain and making a constant number of checks for intersections between dividing chains, the total time required to construct $\sigma(S_1, S_2)$ will be $O(kn + k\log^2 n)$. Let $T(k)$ be the time used to solve problem 15 by this algorithm, we have:

$$T(k) = 2T(k/2) + O(kn + k\log^2 n) = O(k\log k(n + \log^2 n)). \tag{5.5}$$

This is an $O(k)$ improvement over the previous solution. However, we are not sure whether it is optimal for problem 15.

5.4.5 An interesting heuristic

Before ending our discussion for problem 15, we present an interesting idea that may be of help to future research in this area. The idea is based on the following

observations about Voronoi diagram construction:

1. There are k different convex polygons in S, which define $k(k-1)$ different polygon pairs.

2. Each of these polygon pairs determines an individual dividing chain, hence there are $k(k-1)$ dividing chains.

3. Every Voronoi polygon is *decided* by $k-1$ dividing chains, but this does not mean that each of these dividing chains will *appear on the edges of the Voronoi polygon*. In other words, not all of the dividing chains will contribute to the final Vonoroi diagram.

4. If we can identify those dividing chains that actually contribute to the Voronoi edges, then we should be able to design a very efficient algorithm.

As the elements of S are *translates* of a given convex polygon, we can possibly construct the Voronoi diagram in a particular way. For example, the geometrical centers $c_1, c_2, ..., c_k$ of the given translates $P_1, P_2, ..., P_k$ may provide very important proximity information about the Voronoi diagram construction. Let S_1 be the set of geometrical centers of $c_1, c_2, ..., c_k$. We wonder whether the Voronoi diagrams $Vor(S)$ of S and $Vor(S_1)$ of S_1 have a one-to-one edge and vertex correspondence. If this is the case, then the construction of $Vor(S)$ can be done by first constructing the Voronoi diagram $Vor(S_1)$ of S_1. If two points (say c_i and c_j) define a Voronoi edge e_1 in $Vor(S_1)$, then their corresponding polygons P_i and P_j will also define a Voronoi edge e in $Vor(S)$ (see figure 5.9). This means that the Voronoi diagram construction

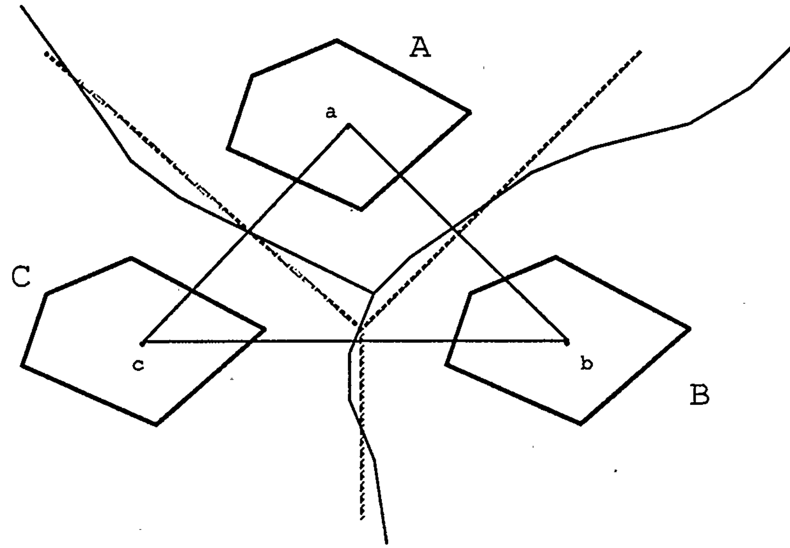


Figure 5.9: An interesting correspondence.

process can be *directed* by the information stored in $Vor(S_1)$, therefore only useful computation is performed.

Unfortunately, there *is not* a one-to-one correspondence between $Vor(S)$ and $Vor(S_1)$. Although in most of the cases our assumption is true, counter examples can be found when the translates in S are long and narrow slates. There are two possible ways to fix this problem: (1) use another model that better captures the proximity information of a given set S of convex polygon translates, and (2) still use the geometrical centers of the translates, but something has to be done to check out whether an edge in $Vor(S)$ indeed has a corresponding edge in $Vor(S_1)$. We shall not further discuss this in this thesis.

Chapter 6

Conclusion

This thesis investigated several problems relating to sets of planar convex objects.

First the problem of *Union Hull Construction* is studied. For an input set S of k convex n-gons, we presented an optimal algorithm that constructs the union hull $U(S)$ in $O(nk \log k)$ time. We also discussed the generalized union hull construction problem, and an new algorithmic lower bound was established.

Next the *Convex Polygon Intersection* problems are investigated. We discussed in detail both the *Intersection Computation* problem for which an algorithmic lower bound as well as an optimal algorithm was given, and the *Intersection Detection* problem for which we presented an algorithm that substantially outperformed the current algorithm on an average-case basis. The lower bound for the latter, however, is still unknown.

Then we proceeded to consider the problem of *Separability Detection* for sets of convex objects such as line segments and convex polygons. Finding the optimal solution for separability detection was an open problem and it still is. However, an algorithm is presented here which we claim will work faster than the best solution so far.

Finally the problem of *Voronoi Diagram Construction* for a set of convex polygons was discussed. We studied a restricted version of the problem where the inputs are *translates* of a given convex polygon and we gave a new algorithm for this restricted problem. More work remains to be done on constructing Voronoi diagrams for any

given set of convex polygons, and we believe that the work presented here can be valuable to future efforts in solving this problem.

Much of the work done in this thesis is based on previous research. The *union hull* problem is a generalization of the ordinary convex hull problem, although as far as we know it has never been formally brought up and studied. Both of the *convex polygon intersection* and the *convex polygon separation* were studied before and interesting results were achieved. Nontheless, this thesis provides algorithms that considerably improve the current results. The *Voronoi diagram* construction problem for a set of planar objects (other than a set of points, which has received adequate treatment already) is now catching more and more attention from researchers. However, as far as we know, there is still not much work done on constructing the Voronoi diagram for a set of convex polygons. We believe that the work presented here is a valuable attempt, and could be useful to future efforts in solving this problem.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, Mass., 1974.

[2] F. Aurenhammer, *Voronoi diagrams – A survey of a fundamental geometric data structure*, Research report $B90 - -09$, Institute for computing science, Department of Mathematics, Freie University, Berlin. Nov. 1990.

[3] D. Avis, J. M. Robert, *Lower bound for line stabbing*, Info. Proc. Lett. 33 (1989), p59-62.

[4] 1990 distribution, *Computational Geometry Bibliography*, Sept, 1990.

[5] M. Ben-Or, *Lower bounds for algebraic computation trees*, Proc. 15th ACM Annaul Symp. on Theory of Computing., 1983, p80-86.

[6] B. Chazelle, *Optimal algorithm for computing depths and layers*, Proc. 21st Annual Allerton Conf. on Comm., Control and Compt. (1983), p427-436.

[7] B. Chazelle, D. P. Dobkin, *Detection is easier than computation*, Proc. 12th Ann. ACM symp. on Theory of computing (1980), p146-153.

[8] B. Chazelle, D. P. Dobkin, *Intersection of convex objects in two and three dimensions*, Journal of the A.C.M, V-34(1), Jan.,1987, p1-27.

[9] D. P. Dobkin, D. G. Kirkpatrik, *Fast detection of polyhedral intersection*, Theoretical Computer Science 27 (1983), p241-253.

[10] D. Dobkin, R. Lipton, *On the complexity of computations under varying set of primitives*, Journal of Computer and Systems Sciences 18 (1979), p86-91.

[11] M. E. Dyer, *On a multidimensional search technique and its application to the Euclidean one-center problem*, SIAM J. Comput. 15 (1986), p725-738.

[12] H. Edelsbrunner, H. A. Maurer, F. P. Preparata, A. L. Rosenberg, E. Welzi, D. Wood, *Stabbing line segments*, BIT 22 (1982), p274-281.

[13] H. Edelsbrunner, M. H. Overmars, D. Wood, *Graphics in flatland: A case study*, Advances in Computing Research, V-1 (1983), F. P. Preparata Ed., JAI Press, p35-59.

[14] M. Golin, R. Sedgewick, *Analysis of a simple yet efficient convex hull algorithm*, Proc. 4th Symposium on Computational Geometry (1988), p153-163.

[15] R. L. Graham, *An efficient algorithm for determining the convex hull of a finit planar set*, Info. Proc. Lett. 1 (1972), p132-133.

[16] E. Horowitz, S. Sahni, *Fundamentals of data structures*, Computer Science Press: Woodland Hills, Calif., 1977.

[17] R. A. Jarvis, *On the identification of the convex hull of a finite set of points in the plane*, Info. Proc. Lett. 2 (1973), p18-21.

[18] J. Kiefer, *Sequential minimax search for a maximum*, Proc. American Math. Soc. 4 (1953), p502-506.

[19] D. G. Kirkpatrik, *Efficient computation of continuous skeletons*, IEEE 20th Annual Symposium on Foundations of Computer Science (1979), p18-27.

[20] D. E. Knuth, *The art of computer programming, Volume 1: Fundamental algorithms*, Addison-Wesley, Reading, MA, 1968.

[21] D. E. Knuth, *The art of computer programming, Volume 3: Sorting and searching*, Addison-Wesley, Reading, MA, 1973.

[22] D. T. Lee, R. L. Drysdale, *Generalization of Voronoi Diagrams in the plane*, SIAM J. Computing V-10(1), Feb., 1981, p73-87.

[23] N. Megiddo, *Linear-time algorithms for linear programming in R^3 and related problems*, SIAM J. Comput. 12 (1983), p759-776.

[24] S. N. Meshkat, C. M. Sakkas, *Voronoi diagram for multiply-connected polygonal domains 1: Algoirthms*, IBM J. Res. Developments 31(3), May, 1987, p361-372.

[25] S. N. Meshkat, C. M. Sakkas, *Voronoi diagram for multiply-connected polygonal domains 2: Implementation and application*, IBM J. Res. Developments 31(3), May, 1987, p373-381.

[26] D. E. Muller, F. P. Preparata, *Finding the intersection of two convex polyhedra*, Theoretical Computer Science 7 (1978), p217-236.

[27] J. O'Rourke, C. B. Chien, T. Olson, D. Naddor, *A new linear algorithm for intersecting convex polygons*, Computer Graphics and Image Processing 19 (1982),p384-391.

[28] F. P. Preparata, *An optimal real time algorithm for planar convex hulls*, Comm. ACM 22 (1979), p402-405.

[29] F. P. Preparata, S. J. Hong, *Convex hull of finite sets of points in two and three dimensions*, Comm. ACM. V-20(2), Feb.,1977, p87-93.

[30] F. P. Preparata, M. I. Shamos, *Computational geometry, an introduction*, Spring-Verlag, New York (1985).

[31] M. Reichling, *On the detection of a common intersection of k convex objects in the plane*, Info. Proc. Lett. 29 (1988), p25-29.

[32] E. M. Reingold, *On the optimality of some set algorithms*, Journal of the ACM. 19 (1972), p649-659.

[33] A. Renyi, R. Shulanke, *Ueber die konvexe Hulle von n zufallig gewahlten Punkten*, I,Z. Wahrschein, 2 (1963), p75-84.

[34] C. A. Rogers, *Packing and covering,* Cambridge University Press, Cambridge, England, 1964.

[35] M. I. Shamos, D. Hoey, *Geometric intersection problems*, Seventeenth Annual *IEEE* Symposium on Foundations of Computer Science (1976), p208-215.

[36] J. M. Steele, A. C. Yao, *Lower bounds for algebraic decision trees*, Journal of Algorithms 3 (1982), p1-8.