AN EFFICIENT ALGORITHM FOR PLANAR SUBDIVISION INTERSECTION PROBLEMS

by

Zhongmin Guo

B.Sc., Peking University, China, 1988

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© Zhongmin Guo 1994
SIMON FRASER UNIVERSITY
March 1994

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

APPROVAL

Name: Zhongmin Guo

Degree: Master of Science

Title of thesis: An Efficient Algorithm for Planar Subdivision Intersection Problems

Examining Committee: Dr. Tiko Kameda
Chair

Dr. Binay Bhattacharya, Senior Supervisor

Dr. Woshun Luk, Supervisor

Dr. Jia-Wei Han, External Examiner

Date Approved: March 17, 1994.

SIMON FRASER UNIVERSITY

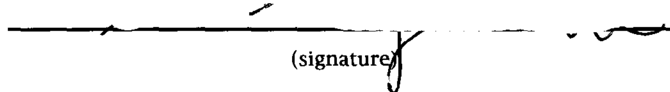
PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

An Efficient Algorithm for Planar Subdivision Intersection Problems.

Author:



(signature)

Zhongmin Guo

(name)

March 28, 1994

(date)

Abstract

Geometric intersection problem is a well developed topic in computational geometry which deals with pairwise intersections among a set of planar objects. A great number of algorithms for detecting whether two objects in the plane intersects have been proposed in the literature. While most of the objects involved in those algorithms are simple objects such as line segments, rectangles and circles, the intersecting properties among polygons are still relatively unknown. In this thesis, we will consider a special case of polygon intersection problems which is called planar subdivision intersection problem. Specifically, given two maps or planar subdivisions of simple polygons, we are required to report all the pairwise intersection of polygons when one is overlayed on top of the other. This problem arises in spatial databases applications and the popular way to handle this is by means of spatial indexing. In this thesis, we will apply the technique of computational geometry to solve the problem. An algorithm proposed by Mairson reports pairwise intersections between two sets of disjoint line segments in optimal time. However, this algorithm does not extend for the polygon case. We propose a generalization of Mairson's algorithm to solve the planar subdivision intersection problem. An implementation of the algorithm is presented and the empirical results are analyzed.

Acknowledgements

I would like to thank my Senior Supervisor, Dr. Binay Bhattacharya for his help, for his encouragement, and for his valuable advice on the drafts of my thesis. Without his support, this thesis would have not been possible. I am also grateful to Dr. Jia-Wei Han and Dr. Woshun Luk for their support during my stay in SFU. I would also like to thank Dr. T. Pattabhiraman, who spent his valuable time on reading and editing this draft. I would also like to thank my best friends around me in the past three years who made my life in Canada memorable.

My special thanks to Grace for putting up with me...

Contents

Abstract iii

Acknowledgements iv

1 Introduction 2

1.1 Background 2

1.2 Problem defined 3

1.2.1 Definition of the Problem 3

1.2.2 Previous Studies 4

1.2.3 General Idea of the Thesis 6

1.3 Thesis outline 7

2 Geometric Preliminaries 8

2.1 General definitions and notations 8

2.2 Representation of a Planar Subdivision 10

2.3 Constructing Convex Subdivisions 11

2.4 Constructing Monotone Subdivisions 13

2.5 Constructing Polygonal Subdivision 15

2.5.1 Generation from convex polygons 16

2.5.2 Generation from a set of straight lines 18

3 Monotone Subdivision Intersection **20**

3.1 Mairson's Line Segment Intersection Algorithm 21

3.2 A modification of Mairson's algorithm 26

3.2.1 Line Segments Properties 27

3.2.2 Removing the S-T cone by Linked List 29

3.3 Intersecting Monotone Subdivisions 31

3.3.1 Problem of the 4-way linked list: *hidden* intersection 31

3.3.2 Operations on 2-3 tree 32

3.3.3 Find the *hidden* intersections 33

3.3.4 Monotone Subdivision Intersection Algorithm 37

4 A Practical Algorithm **46**

4.1 General Ideas 47

4.2 The New Algorithm 48

4.2.1 When a polygon is leaving 49

4.2.2 When a new polygon is entering 51

4.3 Running Time of the Algorithm 52

5 The General Algorithm 55

5.1 Partitioning general planar subdivision to monotone planar subdivisions 56

5.1.1 Partitioning one single polygon into monotone pieces 56

5.1.2 Partition a planar subdivision 58

5.1.3 Partitioning and Reporting Intersections Concurrently 60

5.2 Reporting the Intersections Exactly Once 61

5.3 The Dynamic Data Structure of the Maps 63

5.3.1 Data Structure of a Monotone Piece 63

5.3.2 Data Structure of Simple Polygon 64

5.4 The Final Algorithm 65

6 Implementations and Conclusions 68

6.1 Implementation Environment 68

6.2 Special Case Policy 69

6.3 Empirical Results 70

6.4 Conclusion 72

Bibliography **72**

List of Tables

6.1 General planar subdivision intersections 71

6.2 Running time of general planar subdivision intersections 71

List of Figures

1.1 Example of redundant testing of polygon intersection 5

1.2 Unnecessary testing for polygon intersections 6

2.1 Monotone polygon with respect to horizontal line 9

2.2 Double connected edge list 11

2.3 Voronoi diagram 12

2.4 Example of K-D tree 13

2.5 Planar subdivision of monotone polygons 15

2.6 Example of hanging edges after deletion of the common edge 16

2.7 Planar subdivision after deletion of the common edge 17

2.8 Planar subdivisions generated by k-d tree 18

2.9 Planar subdivisions generated from a set of straight line 19

3.1 Edges in order 22

3.2 Finding all intersection when a segment is leaving 23

3.3 Blocking S-T intersection being detected 23

3.4 A S-T cone and the removal of it 24

3.5 Another S-T cone formed by t and s' after splitting s 25

3.6 Order of Monotone Polygons 28

3.7 S-T cones and related linked list 30

3.8 Example of hidden intersection 31

3.9 Picture of two subdivisions when p is encountered 34

3.10 Searching for the first polygon not been reported 36

3.11 A planar subdivision and its polygonal chains 38

3.12 Testing the bottom chain of s_1 with polygons t_2, t_3, t_4 39

3.13 A left end point sits in two cones 41

3.14 Four types of vertices 42

3.15 Proof of the algorithm 44

4.1 A polygon is leaving 49

4.2 S-T cone happens when a new polygon is coming 51

5.1 Sweeping line events 57

5.2 Partition a Planar Subdivision 59

5.3 A Picture of a partitioned subdivision 60

5.4 Reporting intersections when a polygon is leaving 62

Chapter 1

Introduction

1.1 Background

Computational geometry, as a field of study of computational complexity of finite geometric problems, emerged in the early seventies. Since Shamos [22] first gave the discipline its name in 1975, a large number of scientists have been attracted to this area. One of the major areas in computational geometry is known as the *geometric intersection problem.* This problem deals with the computation of intersection among sets of interesting planar objects such as line segments, rectangles and circles. The geometric intersection problem arises in many disciplines such as VLSI design (do any conductors cross?), architectural design (are two items being placed in the same spot), computer graphics (one object on the 2-D screen can be obscured by the other), etc. These problems are all due to the simple fact that two planar objects can not occupy the same region in the plane. As the need for industrial application grows, faster algorithms are needed for reporting intersecting or overlapping objects. A number of algorithms for computing the union or intersection of sets of geometric

objects and for counting and reporting all intersecting pairs in sets of such objects have been discussed by Shamos and Hoey [23] and later Bentley and Ottmann [4].

Since the motivation for studying the complexity of the intersection algorithm is so strong, the topic has been well-developed ever since the day the problems emerged. However, most of the effort was devoted to those simple objects such as line segment, rectangle, circle, etc., i.e., the objects whose descriptions take $O(1)$ space per object[5, 18]. However, not much is known about the intersection properties among polygons. More recently, as spatial databases have become a very active research topic, the problem of intersecting polygons has become increasingly important. The importance of the problem is enhanced by the fact that polygon is a frequently used object to represent spatial relationship in the plane. So far, most of the spatial database research has concentrated on the data modeling aspects, especially on the design of access methods to support spatial operations by indexing or the like. In this thesis we investigate this problem using techniques developed in computational geometry.

1.2 Problem defined

1.2.1 Definition of the Problem

A map or a planar subdivision can be viewed as a portion of the plane which is defined by a straight line planar embedding of a planar graph. Therefore, a polygonal

subdivision consists of a set of nonoverlapping polygonal regions.

This thesis will focus on a special case of the polygon intersection problem, which we shall call the **planar subdivision intersection** problem. Specifically, given two maps or planar subdivisions of simple polygons, we are required to report all the pairwise intersections of polygons when one is overlaid on top of the other. We will attempt to find a time and space efficient algorithm for this problem.

1.2.2 Previous Studies

The problem defined above arises in spatial database application. Spatial information is usually stored as maps. Maps are organized into layers such as streams, soils, world cities, crop productivity, and administrative boundaries such as land uses, time zones, trading areas, and political areas, etc. While each map is nothing but a set of polygons, queries such as "find out all the states in each time zone" will require us to find out all the pairwise intersections of polygons in the time zone map and political areas map. A naive method of solving this problem is to test all the possible intersection pairs even though the actual number may be small. When the number of polygons in the map is small, this solution is usually good enough. However, as the number of regions gets larger and larger, faster algorithms become essential.

In 1982, H. Mairson [19] proposed an algorithm which can report all pairwise intersections between two sets of disjoint line segments in $O(n \log n + I)$ where I is the

total number of intersections, n is the number of line segments in each set. This algorithm is optimal. Consequently, if all the pairwise edge intersections are computed, pair-wise polygon intersection can then be deduced. However, this solution has two shortcommings: First, we are not able to report the intersection if one polygon is totally enclosed inside the other. Second, there might be far more edge-wise intersections than polygon-wise intersections. See Figure 1.1.

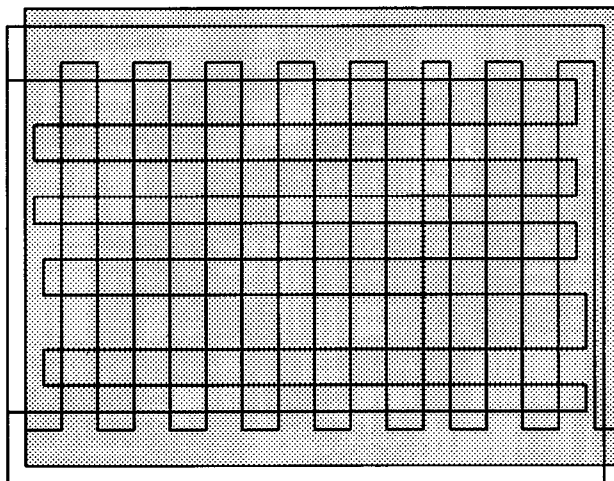


Figure 1.1: Example of redundant testing of polygon intersection

Hong Fan [11] in her M.S. thesis developed several practical ways to solve the problem. These methods are based on spatial indexing methods, where the underlying data structure usually is R-Tree [16] which represents hierarchical space. An R-tree is a structure that is based on a hierarchy of nested rectangles. It is derived from B-tree but can handle n-dimensional objects. To solve the problem, an R-tree for one of the maps is first built; then for each polygon in the other map, all its overlapping polygons are determined by spatial indexing through the use of the R-tree. While these indexing methods are ideally suited for use by the database applications

due to the database management , they also suffer serious performance penalties, as we could find many more rectangle intersections than the actual polygon intersection(figure 1.2).

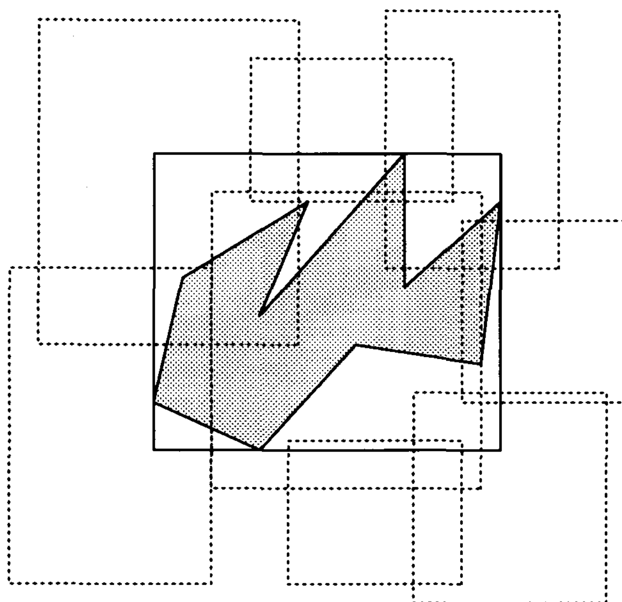


Figure 1.2: Unnecessary testing for polygon intersections

1.2.3 General Idea of the Thesis

Note that the line segment intersection problem is very similar to our problem; the only difference is that the objects dealt with are different. However, the algorithm of Mairson[19] does not extend to polygons. We first extend the algorithm[19] to report efficiently the pair-wise intersections of two planar subdivisions where each polygon in the subdivisions is monotone in one common direction. Later we show that the algorithm developed for monotone planar subdivision can be used to determine efficiently the pairwise intersection of two general planar subdivisions.

1.3 Thesis outline

The organization of the rest of the thesis is as follows: In chapter 2, we introduce the geometric terms used in the thesis. We also describe several methods to generate random monotone planar subdivisions. In chapter 3, we develop our algorithm for the monotone planar subdivision by modifying the line segment intersection algorithm of Mairson. In chapter 4, we further modify the algorithm of chapter 3 such that it has a better average running time. In chapter 5, we use the algorithm to report pair-wise intersection for general planar subdivision by partitioning a general polygon to monotone pieces. In chapter 6, we describe the experimental setup in terms of hardware and software, together with the empirical results and their analysis. The conclusion of this thesis and open problems are also presented in this chapter.

Chapter 2

Geometric Preliminaries

In this chapter, we first present the basic definitions and notations being used in this thesis. Then we present the data structure for representing a planar subdivision. Finally, we develop the algorithms for generating various types of planar subdivisions randomly.

2.1 General definitions and notations

The objects considered in this thesis are normally points, lines, and polygons in the 2-D plane. A **point** is represented as a vector of two dimensions. A **line segment** then is represented as two extreme points. A **polygon** is a set of line segments such that each end point of the segment is shared by exactly two segments and no subset of the segments has the same property. These line segments of a polygon are also called edges. The polygon which is of most interest in our thesis is **simple polygon**. A polygon is *simple* if it has no two edges intersecting with each other. Henceforth,

by **polygon**, we mean **simple polygon** unless otherwise specified.

Two special kinds of polygons needed to be mentioned here are convex polygon and monotone polygon. A **Convex Polygon** is a polygon whose interior is a convex set. A convex set D is a set of points where for any two points $p, q \in D$, segment from p to q is also in D. A simple polygon P is said to be **monotone** with respect to a line L if P intersects any line normal to L in one contiguous part.

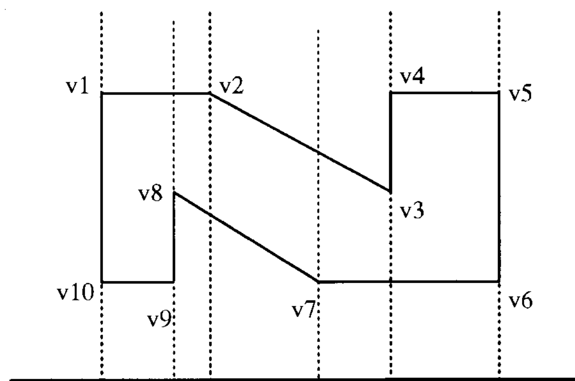


Figure 2.1: Monotone polygon with respect to horizontal line

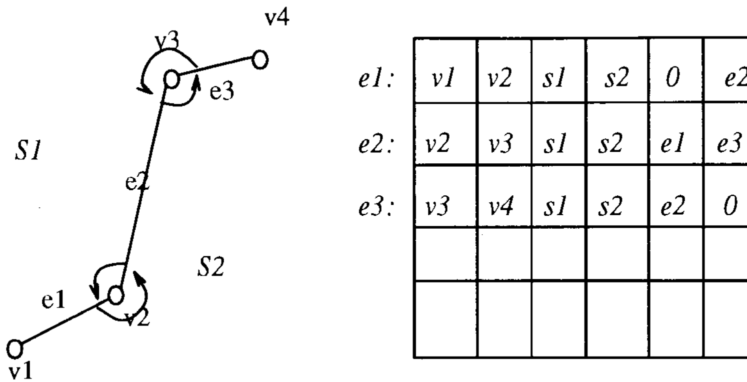
A polygon monotonic with respect to the horizontal line is shown in Figure 2.1. Note that not every polygon is monotone with respect to some line; and some polygons are monotone with respect to several lines.

Another term frequently used in the thesis is **Planar Subdivision** or **Map**. A planar subdivision or a map is a subdivision where all the regions are divided only by straight lines. It is also called polygonal subdivision since all the regions of it must be a polygon except the open region.

There are many different kinds of planar subdivisions. If all the regions in a planar subdivision are convex polygons, we call it a **Convex Subdivision**; if all the regions in the subdivision are monotone polygons with respect to a common line, we call it a **Monotone Subdivision**.

2.2 Representation of a Planar Subdivision

There are many different data structures that can be used to represent a planar subdivision. In our implementation, a widely used data structure called double-connected-edge-list(DCEL)[20] is used. Given a planar subdivision G $= (V, E)$, V $= (v_1 \ldots v_n)$ and E $= (e_1 \ldots e_m)$, the main component of the DCEL is the edge node, i.e., each edge is represented exactly once by an edge node. An edge node consists of four information fields, V_1, V_2, S_1 and S_2, and two pointer fields n and p. A planar subdivision can be implemented as an array of all the edge nodes. The meanings of the fields are as follows. The first two fields V_1 and V_2 contain the origin and terminus of the edge respectively. The fields P_1 and P_2 contain the names of the polygons which lie respectively on the left and on the right of the edge oriented from V_1 to V_2. The V and P fields can be taken as integers. The pointer n points to the edge node containing the first edge encountered after edge (V_1, V_2) when one proceeds counterclockwise around V_1. The pointer p points to the edge node containing the first edge encountered after edge (V_2, V_1) when one proceeds counterclockwise around V_2. As an example, a fragment of a graph and the corresponding fragment of the DCEL are shown in Figure 2.2.



e1:	v1	v2	s1	s2	0	e2
e2:	v2	v3	s1	s2	e1	e3
e3:	v3	v4	s1	s2	e2	0

Figure 2.2: Double connected edge list

Some features of the DCEL method are worth mentioning here. From the DCEL, we can get an array of all vertices of the graph G in which each vertex node keeps all the edges incident on it in time $O(N)$, where N is the number of vertices in G. We can also get the array of the polygons(regions) in time $O(N)$ from the DCEL. It is also easy to travel from edge to edge in the graph using DCEL. These features of DCEL influence us to use it for representing a planar subdivision in this thesis.

2.3 Constructing Convex Subdivisions

We will now start the discussion on random constructing planar subdivisions. In this section, we present the construction of planar subdivisions consisting of only convex polygons.

The construction of a convex planar subdivision turns out to be relatively easy

due to a well-known geometric object called the Voronoi diagram. A Voronoi diagram is an unbounded geometric object which reflects the proximity relations among a set of points in Euclidean space, see Figure 2.3. The points in the set are called sites. A Voronoi diagram divides the plane into Voronoi regions in which all the points inside the region are closer to its site than any other sites. The precise definition by Preparata and Shamos of Voronoi region regards it as the intersection of half-planes. We consider only the Voronoi diagram in a two-dimensional space. Since each voronoi region is a convex region, the Voronoi diagram is therefore a planar subdivision of convex polygons.

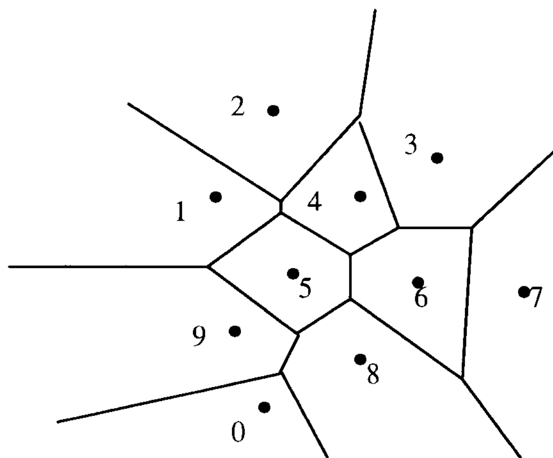


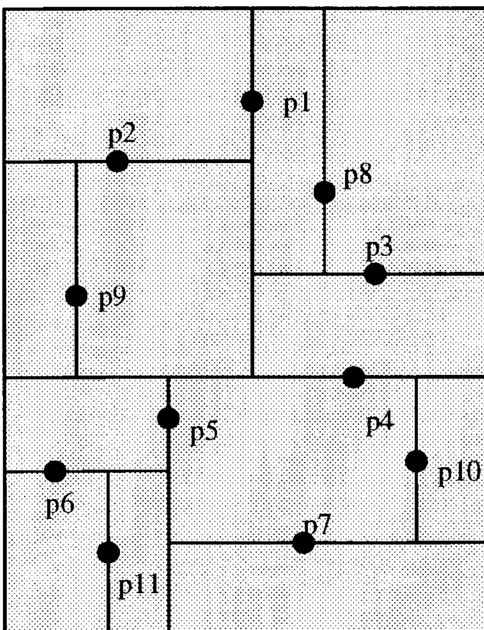
Figure 2.3: Voronoi diagram

Voronoi digram is useful in its own right, and much research has been devoted to the computation of these diagrams. Several practical algorithms have been proposed. We chose Fortune's algorithm[12], which takes $O(n \log n)$ time to generate the diagram. Our approach of generating convex planar subdivisions then consists of two stages. The first stage will generate a set of random points on the plane. The number of points determines the number of polygons we will get. The second stage will apply

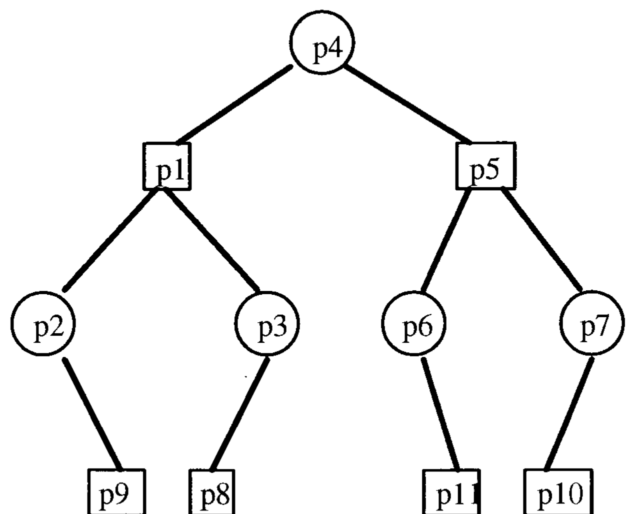
Fortune's algorithm to generate the Voronoi diagram of the given set.

2.4 Constructing Monotone Subdivisions

Monotone polygon plays a fundamental role in developing our algorithm. Therefore maps consisting of only monotone polygons are also constructed to test the algorithm. K-D tree data structure is used to generate random monotone planar subdivision. K-D tree was first introduced by Bentley[3] as a multidimensional binary search tree in 1975. We will give a brief description of K-D below.



(a)



(b)

Figure 2.4: Example of K-D tree

K-D tree is designed to solve the range search problem where there is a set of points on the plane. As in a standard binary search tree, the root of K-D tree contains one point of the set and the remaining points are assigned to the subtrees above or below according to whether they are above or below a horizontal line. The node chosen as the root is the one whose Y-coordinate separates the two subtrees. Unlike a standard tree, however, the node chosen as the root of the subtree is not the one whose Y-coordinate separates the two subtrees, but rather the one whose X-coordinate separates the two subtrees. The rest of the points in the subtree are assigned to the left and right tree according to whether they are to the left or to the right of the vertical line passing through the root node of its subtree. The same process is applied to all the subtrees recursively. The process stops when there is no point left in the subtree, and the corresponding node is a leaf of the tree. See the sample K-D tree in Figure 2.4.

Now, we examine the construction of a planar subdivision of monotone polygons. Without loss of generality, we assume that each polygon is monotone in the x-direction. The whole process consists of three steps (Figure 2.5 shows one map generated in this way):

(a): Generate a number of points on a two dimensional square and compute their K-D tree. The number of points is equal to the number of polygons in the map.

(b): Randomly put some points into each rectangle of the K-D tree. Sort the points inside each rectangle in increasing X-coordinate order. Then link the points to its adjacent points.

(c): Delete the horizontal edges of the tree. Note that each polygon in the map has at least two vertical boundary edges.

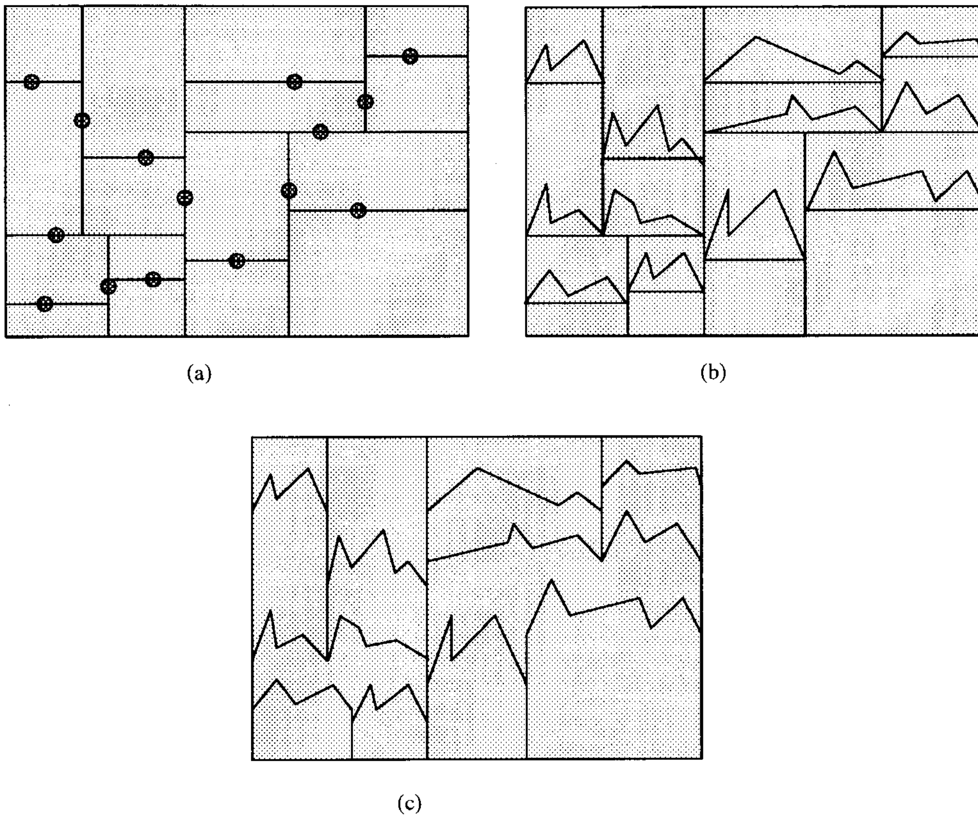


Figure 2.5: Planar subdivision of monotone polygons

2.5 Constructing Polygonal Subdivision

The random generation of polygonal subdivision is complex. We shall discuss several methods in this section that we have tried.

2.5.1 Generation from convex polygons

A. Random deletion and merging

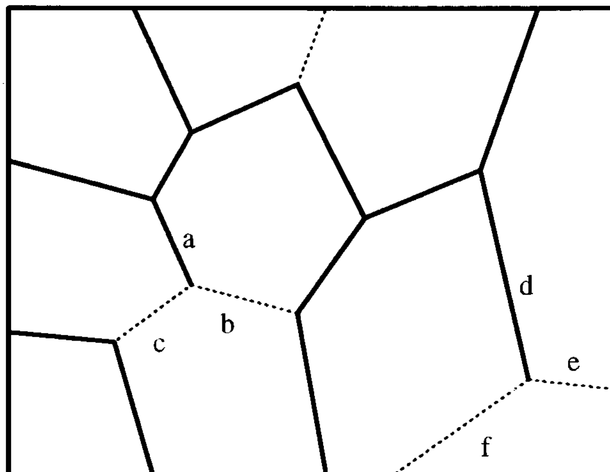


Figure 2.6: Example of hanging edges after deletion of the common edge

Two adjacent convex polygons can be merged into one simple polygon by removing their common edge. The same is true if we remove a common edge of two simple polygons except that there might be some dangling edges that should be removed later. In Figure 2.6, a and d are dangling because the removal of c, d and e, f. These dangling edges, however, can be detected and further removed. Therefore, starting from a planar subdivision of only convex polygons, we can obtain a planar subdivision consisting of simple polygons by removing edges from the graph randomly.

The removal of the hanging edges is accomplished by the "merge and find set" from Aho, Hopcroft & Ullman[2]. The basic idea is that each edge belongs to two different regions. Once an edge is deleted, its two regions are merged into one. Any edge belonging to the same region after merging should be deleted. Hence the algorithm

consists of two steps: First, generate a convex planar subdivision. Then, randomly delete some edges and remove those dangling edges. Figure 2.7 is a graph generated by this method.

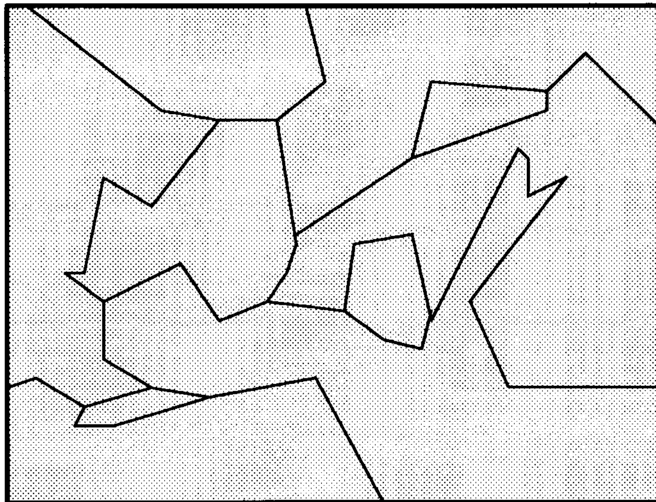


Figure 2.7: Planar subdivision after deletion of the common edge

B. Deleting edges by randomly generated K-D tree

Looking at the K-D tree in Figure 2.4, if we consider each rectangle as a simple polygon and the edge of the rectangle as a boundary chain of the polygon, we will see a more natural looking map of simple polygons. Thus if we put a K-D tree on top of a Voronoi diagram and delete those edges intersecting with the boundary of the tree as well as those edges which are not linked to any deleted edges, we can get a *natural* looking map. The method therefore includes three steps: the first step is to generate a random Voronoi diagram. The second step generates a random K-D tree. The last step considers each rectangle as the boundary of a simple polygon, puts the K-D tree on top of the Voronoi diagram, and deletes all the edges inside the polygon except

those boundary ones. In order to be able to make meaningful deletions, the ratio of total number of rectangles to the total number of sites in the Voronoi diagram should be very small. Figure 2.8 is a map generated this way.

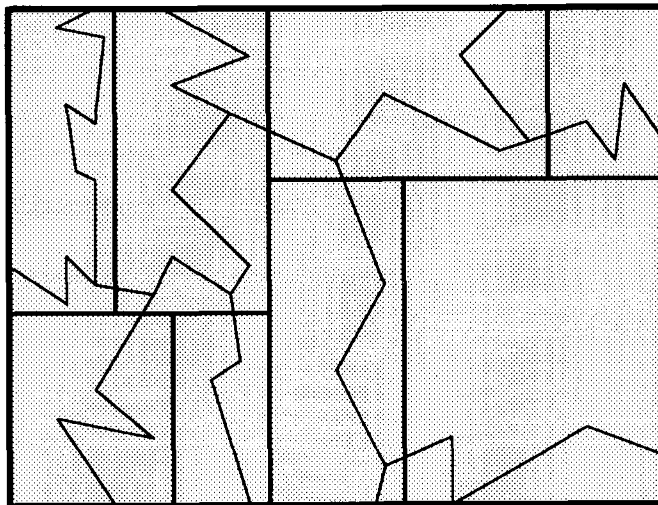


Figure 2.8: Planar subdivisions generated by k-d tree

2.5.2 Generation from a set of straight lines

In order to build up an experimental environment in which the main empirical results can be obtained, Hong Fan [11] described in her Master thesis how random polygonal maps are generated under different distributions. The basic steps are, first to generate a set of straight lines in a square on the plane with the endpoints of the lines lying on the square boundary, then to find out all the intersection points of those lines and replace the intersected line segments with an arbitrary number of edges. Several parameters including the ratio of the largest polygon to the smallest one are used to adjust the appearance of picture. This approach is straightforward and can be used

to generate very large maps. In fact, we used this method and generated several sets of data which are used in our experiment. Further information about this can be found in chapter 2 of Fan's thesis. Figure 2.9 shows one of the maps generated by this method.

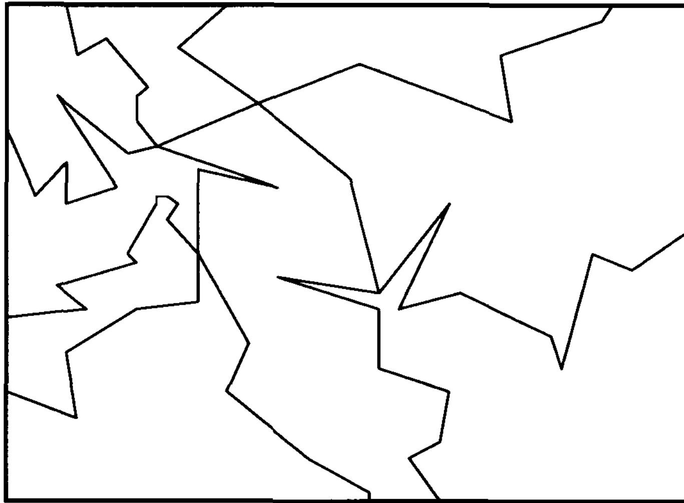


Figure 2.9: Planar subdivisions generated from a set of straight line

Chapter 3

Monotone Subdivision Intersection

In this chapter we present an algorithm which solves a special case of the planar subdivision intersection problem efficiently both in terms of time and storage space. The polygons in the subdivisions are all monotonic along a common direction. Without any loss of generality, we always assume that polygons in the subdivisions are monotonic along the X-direction. The proposed algorithm is an extension of Mairsion's line segment intersection algorithm[19]. The running time of our algorithm is $O(n \log n + I \log^2 n)$ where n is the total size of the planar subdivisions and I is the number of polygon intersections reported.

The organization of this chapter is as follows: Section 3.1 describes the algorithm of Mairson[19] and introduces the terminology used. Section 3.2 discusses a modification of Mairsion's algorithm which allows us to design an algorithm for monotone subdivision. The algorithm for monotone subdivisions is described in Section 3.3.

3.1 Mairson's Line Segment Intersection Algorithm

Given two sets of line segments S and T such that the segments in each set are pair-wise disjoint, Mairson[19] discovered an elegant optimal algorithm that can report all the intersecting pairs (s, t), $s \in S, t \in T$ in $O(n \log n + I)$ time where n is the total number of line segments, I is the total number of intersecting pairs between S and T.

Mairson's approach is based on a well-known technique in computational geometry called plane sweep technique[18]. The description of the algorithm in this section will be rather intuitive. A more formal description is presented in[19]. The primary step of the algorithm is to *sweep* a vertical line through the end points of the line segments in $S \cup T$. During the sweep we maintain two sets of line segments S_A and T_A, one for S and the other for T, in which we keep all the segments currently intersecting the sweeping line. We call $S_A(T_A)$ active edge list or sweep table of $S(T)$. Initially the sweep tables are empty. When a left end point of a line segment is encountered during the sweep, say from $S(T)$, we insert the segment into the corresponding sweep table $S_A(T_A)$. When a right end point of a segment, say from $S(T)$, is encountered, we deactivate it by removing it from the table for $S_A(T_A)$. Just before a segment from $S_A(T_A)$ is removed, we can detect and report all its intersection by comparing it with all the segments in the other set $T_A(S_A)$ in a brute force manner. The problem with this implementation is that it would take $O(n)$ time per segment in the worst case, independent of the number of intersections between S and T. This therefore results in an $O(n \log n + |S||T|)$ algorithm where $n = |S \cup T|$, $|S|$ indicates the size of S.

Two line segments A and B in the plane are said to be *comparable* if there exists a vertical line, say at x, that intersects both of them. When A and B are *comparable*, we can define a geometric relationship *above*. We say that A is *above* B, if the intersection point of A with the vertical line lies above the intersection point of B with that line. In Figure 3.1, segment A, B, C are *comparable*, C is *above* B, B is *above* A, C is also *above*A. D is only comparable with B, and it is *above* B.

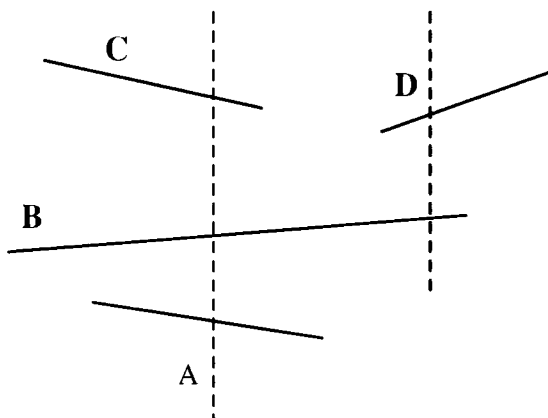


Figure 3.1: Edges in order

Note that the relation *above* is transitive and hence defines a total order. It is easy to locate the line segments immediately above and below the end point of any specific segment given the active edge list. When the right end point of a segment, say s from S, is processed, it would be nice if all the T segments intersected by s were consecutive elements in the active edge list T_A. In this case, we start from the two T_A segments right above or below the right end point, and test against the segment upward or downward along the list. We terminate the search in each direction at the first segment which does not intersect with s. In Figure 3.2, s_1 is from S, $t_1 \dots t_6$ are from T. When s_1 is leaving, we can detect all its intersections with segments in T_A

starting from T_1 and moving upwards.

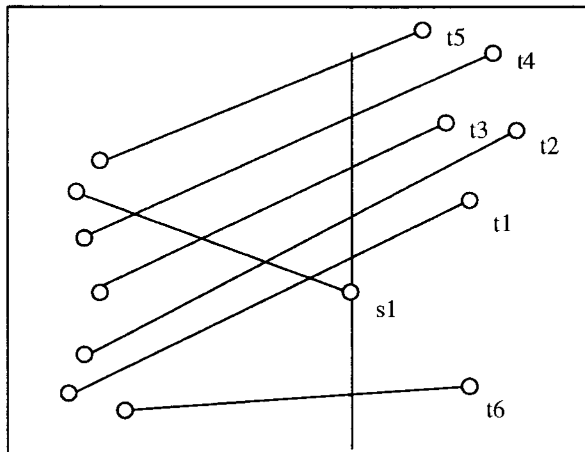


Figure 3.2: Finding all intersection when a segment is leaving

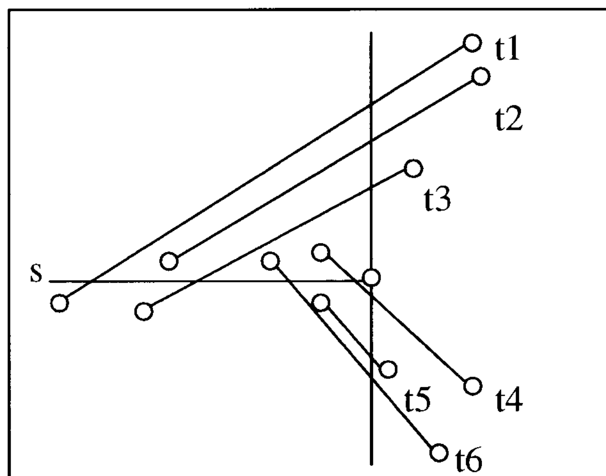


Figure 3.3: Blocking S-T intersection being detected

Unfortunately, there is a situation in which one segment can block other possible intersections to be reported. In Figure 3.3, s is from S_A, $t_1 \ldots t_6$ are from T_A. When the rightmost end point of s is encountered, t_2 will block segment t_1 and t_5 will block t_6 from being detected.

A segment r can block an S-T intersection from being detected by the linear search described above only if its left endpoint sits in a particular region which is bounded above and below by two intersecting segments s and t extending to the right of its intersection, this region is called a **S-T cone.** See the example in Figure 3.4.

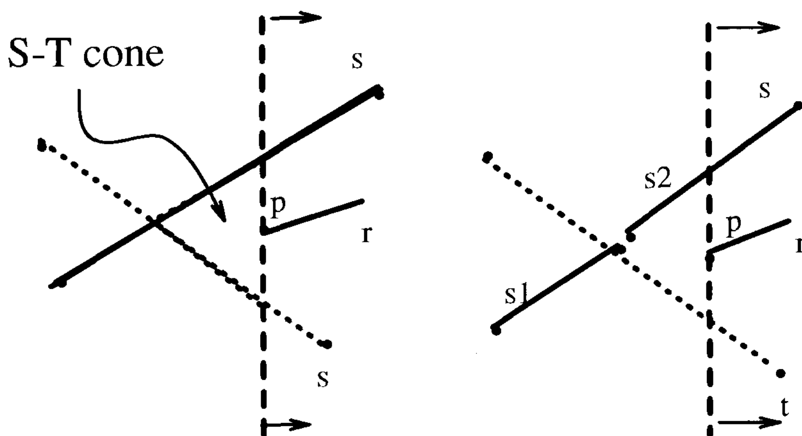


Figure 3.4: A S-T cone and the removal of it

The idea in [19] is to detect and destroy each S-T cone by splitting one of the two intersecting segments at some point to the right of the intersection as soon as r is encountered. After the splitting, r can no longer block the intersection of s and t. Therefore every time a new right end point is going to be processed, it is guaranteed that there are no segments in the active edge list being blocked from a possible intersection.

Fortunately, it is not difficult to test whether a new left end point p sits in a S-T cone or not. It is obvious that all the S-T cones containing p must consist of either an

upper side in T_A and lower side in S_A or the other way around, and those segments form a cone must be consecutive segments immediately above or below p.

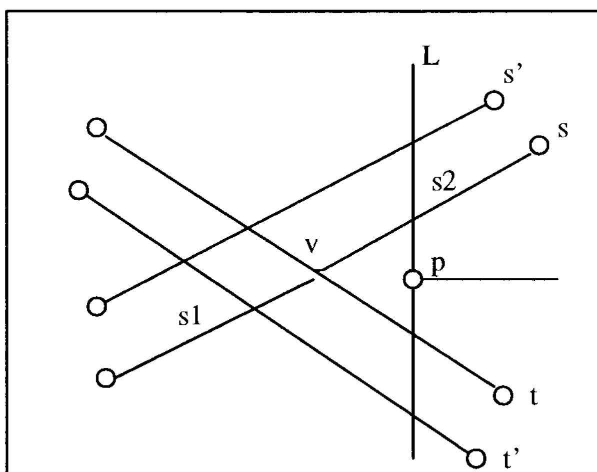


Figure 3.5: Another S-T cone formed by t and s' after splitting s

It does not matter which side of the cone we split; the algorithm in [19] is as follows: whenever a new left end point p of a line segment is encountered during the sweeping, locate the four segments immediately above and below it from both sets S_A and T_A. It is not important which set the new line segment is from. Test the upper segment in S_A against the lower segment in T_A and the lower segment in S_A against the upper segment in T_A. If any two intersect, split the upper side at a point just to the right of the intersection, as indicated in the Figure 3.4. Splitting of s produces two segments s_1 and s_2. Besides replacing s with s_2, we also report all the intersections of s_1 with the consecutive segments in T_A since this part s_1 of s will no longer remain active. Note that after splitting s, t can still intersect line segments in S_A and thereby forming another S-T cone(see Figure 3.5). We remove these S-T cones by splitting S_A edges until no more intersection is found. It takes $O(\log n + k)$ time to destroy

S-T cones for each left end point, where k is the number of intersections reported. $O(\log n)$ time is needed to determine the position of the entering line segment in S_A and T_A. Similarly it takes $O(\log n + k)$ to report k intersections when a right end point is encountered. So the total running time of algorithm is $O(n \log n + I)$ where I is the total number of intersecting pairs. Clearly, the total storage space requirement is $O(n + I)$.

3.2 A modification of Mairson's algorithm

Mairson's algorithm[19] reports all the pairwise segment intersections in optimal time. Since a planar subdivision can be viewed as a set of disjoint line segments, where two segments sharing the same endpoint are not considered intersecting, we can solve the planar subdivision intersection problem by applying Mairson's algorithm to find out all the edge-wise intersections. We can then convert the edge-wise intersection result into polygon-wise intersection result. Unfortunately, this approach has a few drawbacks. One of them is that Mairson's algorithm[19] reports all pairwise intersections of line segments, but we only need to detect all the polygonal intersections(see Figure 1.1). In this case, the running time of the algorithm is determined by the number of edge intersections, not by the number of polygon intersections. This makes it less desirable. Moreover, the algorithm in [19] cannot report the containment relationship, i.e., if one polygon is totally enclosed by the other.

We now discuss our approach to extend Mairson's algorithm so that we can report

all pairwise polygon intersections in monotone subdivisions. Our objective is to design an algorithm whose worst case running time is a function of the number of pair-wise polygon intersections being reported.

3.2.1 Line Segments Properties

Let us first take a closer look at Mairson's algorithm. The correctness of Mairson's algorithm is dependent on three properties of line segments. The same algorithm might be used to solve intersection problems of other objects as long as those objects have the three properties as well.

Property P1 : Any vertical line through the object intersects the object exactly once, i.e., the intersections of a vertical line with the object is either empty or contiguous.

Property P2 : For any disjoint pair of objects intersecting the same vertical line, it is possible to determine the above-below relationship in constant time. This relationship does not change for any other vertical line.

Property P3 : The intersection of any two objects is either empty or connected.

Property P1 ensures that an order relation *above* will exist among the objects for any vertical line. Property P2 ensures that the relation can be computed. Property

P3 guarantees that the same intersection will be detected only once. Thus if a class of objects has properties P1-P3, all the intersection pairs among two sets of disjoint objects can be reported by modifying Mairson's algorithm easily.

As described earlier, we assume that all the polygons are monotone along the X-axis. We observe that a vertical line intersects a monotone polygon in one continuous part, thus satisfing Property 1. We also observe that an order relation can be defined on non-intersecting monotone polygons, not just on line segments. We say that two non-intersecting monotone polygons are comparable if they do not intersect and there is a vertical line intersecting both of the polygon. Given two comparable monotone polygons A and B, either A is above B or B is above A. And for any disjoint monotone polygons, the above relationship will not change while the vertical moving along the x-axis. See Figure 3.6, A is *above* B, B is *above* C, A is *above* C, and the order will not change.

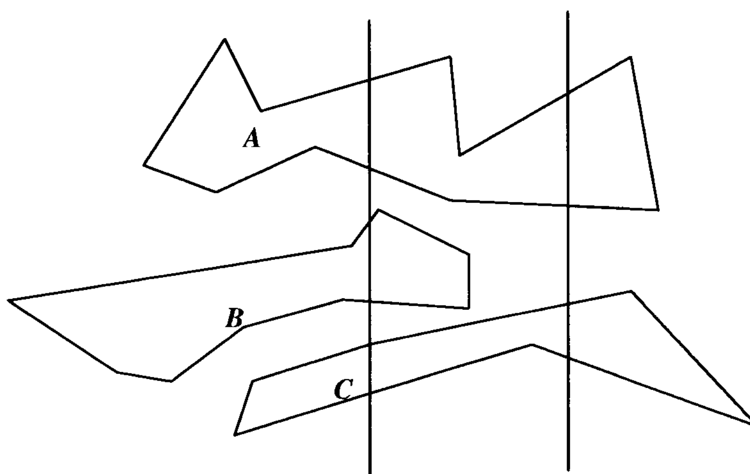


Figure 3.6: Order of Monotone Polygons

Property 3 requires that the intersection of two objects is connected or empty. But the intersection of two monotone polygons need not be connected. However this will only result in one intersecting pair being reported more than once. Given the above observations, we may thus employ Mairson's line segment algorithm to solve the monotone planar subdivision intersection problem.

In the following we propose an algorithm which modifies Mairson's algorithm[19]. The modified algorithm is then extended to determine pairwise intersection of monotone polygonal subdivisions.

3.2.2 Removing the S-T cone by Linked List

In Mairson's algorithm, an S-T cone is removed by cutting off one of the intersecting line segments right after the intersection point. While cutting a line segment is a trivial operation which takes constant time, cutting a polygon is not trivial. In this section we will propose another method to destroy each S-T cone by a 4-way connected linked list. The resulting algorithm is still optimal.

The 4-way connected linked list is to keep all the intersections being found. Each node of the list represents an intersection. It has six fields: s, t, n_s, l_s, n_t, l_t. s and t are the two edges forming the intersection. n_s is the next intersection node of s, l_s is the last intersection node of s. n_t, l_t have similar meaning as n_s, l_s. Each segment has a pointer pointing to the node representing the latest intersection which corresponds

to the last cut in Mairson's algorithm. Initially it is empty. Given the linked list, before testing whether any two line segments (say s and t) intersect, we need to check whether there is a node (s, t) in the list; if it is in, that means (s, t) formed a S-T cone and it had been detected, we can stop the linear searching. Testing whether any pair (s, t) is in the list can be done by checking both the head of s's linked list and the head of t's linked list. If neither of them is (s, t), the pair is not in the list. When a right end point of the line segment e is encountered, we delete all the nodes connected with e and report the intersections. The time needed is proportional to the number of intersections reported. Figure 3.7 is the picture of S-T cones and the related linked list.

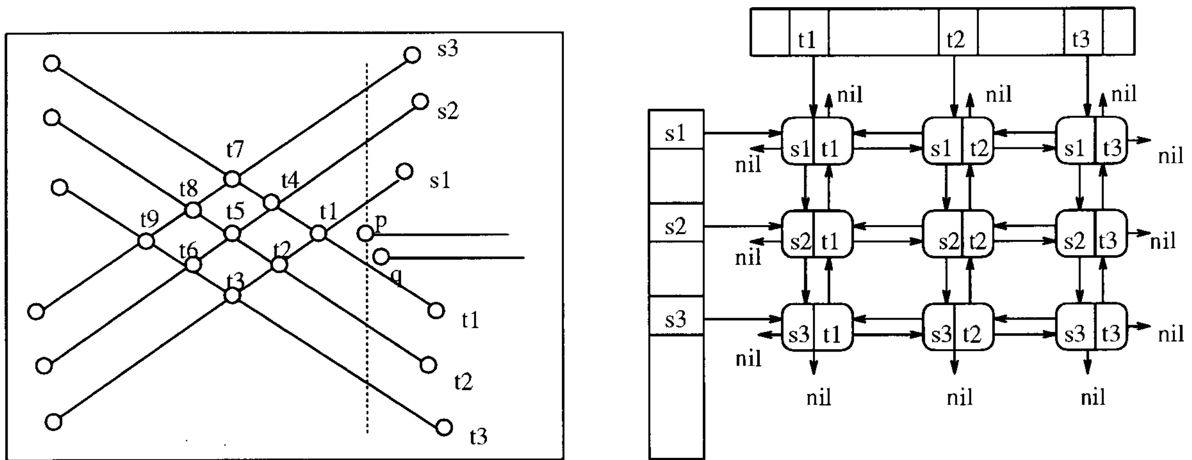


Figure 3.7: S-T cones and related linked list

The storage space needed for the linked list method is $O(n+I)$ where n is the total size of the planar subdivisions and I is the number of intersections reported. Thus the modified Mairson algorithm takes $O(n \log n + I)$ running time and $O(n + I)$ space.

3.3 Intersecting Monotone Subdivisions

3.3.1 Problem of the 4-way linked list: *hidden* intersection

The linked list method in the last section is designed for the line segment intersection problem. In that algorithm, when we are testing one line segment from one set with the active segments in the other set for intersections upward or downward, we stop the linear searching as soon as we hit the first intersection recorded in its intersection list. However this is not always true in case of monotone polygons. There might be a new intersection *hidden* behind the first intersection which has already been found before. See the example in Figure 3.8.

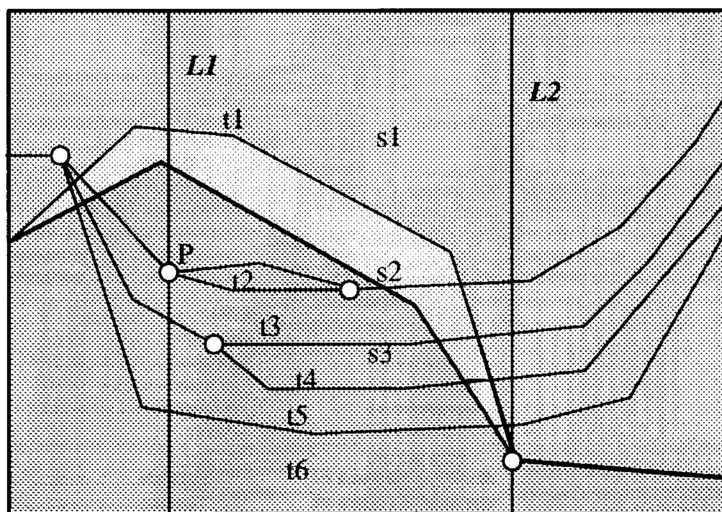


Figure 3.8: Example of hidden intersection

There are two sets of polygons in the picture: one is depicted by a different grey level, labelled as s_1, s_2 and s_3. The polygons in the other set are depicted by $t_1 \ldots t_6$. When the sweeping line moves to the position L_1, intersection pairs (s_2, t_3), (s_2, t_5)

and (s_2, t_6) are detected because p lies in their S-T cones. The intersection list of s_2 is: t_6, t_5, t_3. After several steps, the sweeping line moves to L_2. In this case, polygon s_2 is leaving. The intersection list of s_2 is still: t_6, t_5, t_3. We find the right end point of s_2 sitting in t_6 and start the linear searching from polygon t_6 both upward and downward. The first polygon in the upward search is t_5 and it is in the intersection list of s_2. In case of line segments, we stop the upward search for intersections because there will be no new intersections above t_5. However, there is a polygon p_4 above t_5 which intersects s_2 but has not been detected yet. We call (t_4, s_2) a *hidden* intersection. The underlying reason for the *hidden* intersection is that two line segments can intersect at no more than one place, but two monotone polygons can intersect at more than one place. In the example, t_4 is *hidden* from s_2 because t_5 intersects s_2 in two seperated parts. We now modify the 4-way linked list implementation of the intersection list to accommodate monotone polygons.

3.3.2 Operations on 2-3 tree

A 2-3 tree[1] is a tree in which a non-leaf node has 2 or 3 children, and every path from the root to a leaf is of the same length. Note that a single node is also a tree. 2-3 tree is used to store the active polygons during the sweep. The active polygons are stored in the leaf nodes. They are ordered from left to right by increasing Y-coordinate order. We will also use a 2-3 tree to store the intersection list. In this subsection, we present some interesting results about the 2-3 tree.

We already know that the insertion and deletion operations on a 2-3 tree with n

leaves can be executed in at most O($\log n$) time. We now define two operations **CONCATENATE** and **SPLIT** on 2-3 tree. The operation **CONCATENATE**(T_1, T_2) takes as input two 2-3 tree T_1 and T_2 such that every element in T_1 is less than every element in T_2. We then combine T_1 and T_2 into one single 2-3 tree T, and maintain the order of all the elements. The operation **SPLIT**(a, T) is to split a 2-3 tree T into two 2-3 trees T_1 and T_2 such that all leaves in T_1 are less than a, and all the elements in T_2 are greater than a; a can be in either tree depending on the specific situation. Previous studies[1] show that both **CONCATENATE** and **SPLIT** operations can also be executed in O($\log n$) time. We will not present the algorithm of the two operations nor prove them in this thesis, the result being presented here merely for our analysis.

Given any 2-3 tree, we can also get the number of its leaves by storing the number in its root node. This number can be correctly maintained during all the operations mentioned above without increasing the complexity of these operations. In each non-leaf node of a 2-3 tree, we always stores the smallest element of its second child node. This element therefore can be used to split the 2-3 tree into two subtrees with almost equal number of elements. Thus we can also split a 2-3 tree into two subtrees of comparable size in O($\log n$) time.

3.3.3 Find the *hidden* intersections

To detect the *hidden* intersections when we are testing for intersections of one polygon from one set with the active polygons from the other set, a naive implementation

is to test all the polygons in the active polygon list until a failure occurs, regardless whether the intersections have been reported or not. Obviously this causes lots of redundant tests. With reference to the example in Figure 3.8, when polygon s_2 is leaving, we start our searching upward along the active polygon list in the order of t_6, t_5, t_4, t_3, t_1. Among them, t_5 and t_3 have already been reported intersecting. If we can jump over to the first polygon which is not in the intersection list directly, it will save a lot of computations. This leads to the following discussions:

Let the active polygons of the subdivisions S and T be stored in the 2-3 trees A_S and A_T respectively. Let p be the leftmost vertex of a new active polygon. We now test for new S-T cones in which p lies. Let $A_S^U(p)$ and $A_S^B(p)$ be the set of active polygons of S which are above and below p repectively. $A_T^U(p)$ and $A_T^B(p)$ are defined similarly. Consider any polygon in $A_S^B(p)$, say P_S. We are interested in determining the first polygon in $A_T^U(p)$ whose intersection with P_S needs to be tested. See Figure 3.9. Intersections between $A_S^U(p)$ and $A_T^B(p)$ can similarly be determined.

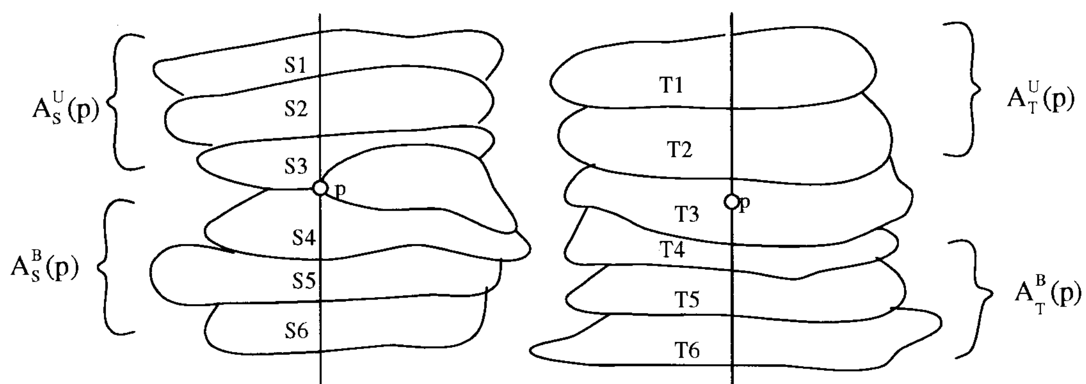


Figure 3.9: Picture of two subdivisions when p is encountered

Let $A'^U_T(p)$ be the polygons in $A_T^U(p)$ whose intersections with P_S are present in

the intersection list of P_S. The intersection list of any active polygon is also stored in a 2-3 tree. The polygons in $A_T^U(p)$ and $A'^U_T(p)$ are ordered along the positive direction of the Y-axis. Once p is known, the lists $A_T^U(p)$ and $A'^U_T(p)$ can be determined in $O(\log n)$ time.

We know that bisection is crucial to the development of any optimal searching algorithm. It is natural to try to generalize bisection for our problem. What bisection does is to split a set of objects into two equal size parts. Next, one of the two parts is chosen and split again. The splitting process repeats until it cannot be split any more. In our case, however, we will split the two sets $A'^U_T(p)$, $A_T^U(p)$ simultaneously. The process is described by the following procedure:

We first split $A'^U_T(p)$ into two subtrees of comparable size. Let the two subtrees be A'_1 and A'_2. The polygons in A'_1 are below the polygons in A'_2. Let the last polygon in A'_1 be P'. We then split $A_T^U(p)$ into two subtrees A_1 and A_2 by P'. The polygons in A_1 are always below the polygons in A_2. If $|A'_1| = |A_1|$, all the active polygons in A_1 are in the intersection list. The first missing polygon must be in A_2. We then choose A_2 and A'_2 for further splitting. If $|A'_1| \neq |A_1|$, there must be a polygon in A_1 which is not in A'_1. We choose A_1 and A'_1 for further splittings. The splitting procedure stops when the each subtree has only one polygon.

An example of searching by splitting the two 2-3 trees is illustrated in Figure 3.10. There are 17 polygons $1 \ldots 17$ in the active list $A_T^U(p)$ and 9 of them are in the intersecting list $A'^U_T(p)$. Ps is a polygon from A_S. It is below polygon 1 at point p.

Its intersection list is $A'^U_T(p)$. Our goal is to find the first polygon above p which is not in $A'^U_T(p)$. In the picture, we denote a 2-3 tree with a triangle. The numbers beside the arrows indicate the sequence of splitting. We first split A'^U_T into two subtrees. The polygon used for the splitting is polygon 7. Then we also use polygon 7 to split A^U_T into two subtrees. After comparing the number of polygons in each subtree, we choose to split their left subtrees. Polygon 5 is chosen to split the subtrees and so on. In the end, we find that polygon 1 is the first missing polygon above p.

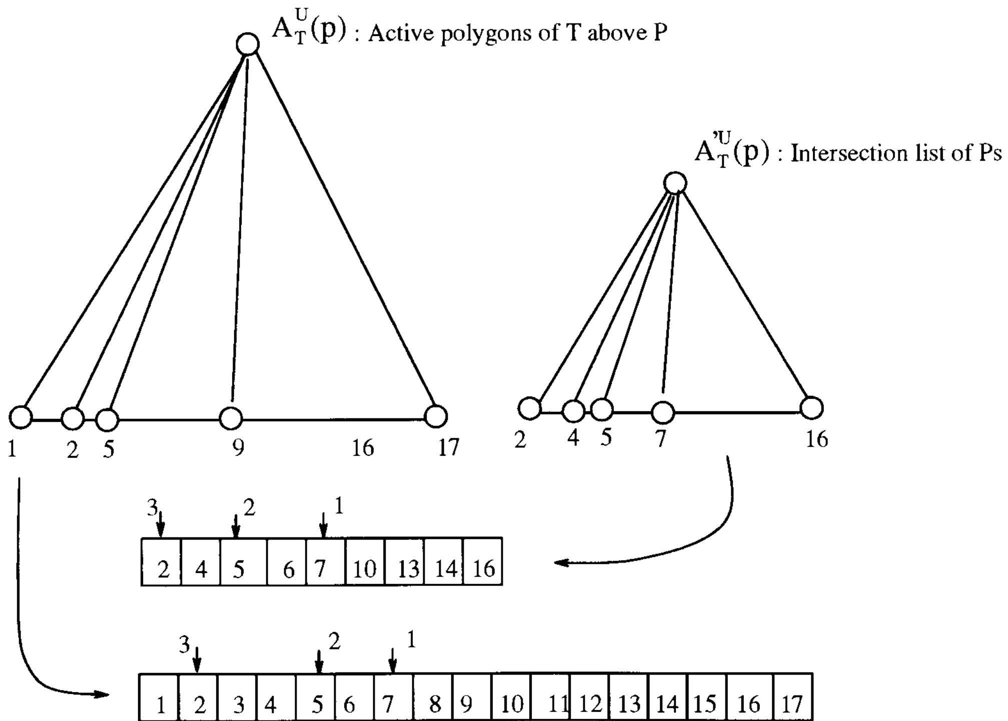


Figure 3.10: Searching for the first polygon not been reported

Each round of the splitting in the above procedure takes at most $O(\log n)$ time where n is the number of polygons in the subdivisions. Since the number of times of splitting A'^U_T is at most $O(\log n)$, the loop will be executed at most $O(\log n)$ time.

Therefore the execution time of the above approach to find the first missing polygon is $O(\log^2 n)$.

The above procedure finds the first missing polygon by splitting the 2-3 trees into two subtrees of comparable size. After finding the first missing polygon, we should concatenate the smaller trees together into the original tree again. This can be done by concatenating those smaller trees in the reverse order of the splitting process. Since both splitting and concatenating of 2-3 tree run in time logarithm to the size of the tree, the complexity of restoring the original tree is the same as that of splitting the trees. So the total time needed to find the first missing polygon for a given p is at most $O(\log^2 n)$ where n is number of polygons in the subdivisions.

3.3.4 Monotone Subdivision Intersection Algorithm

The only question that remains is: how to determine whether two selected monotone polygons P and Q intersect or not. This can be done in $O(|P| + |Q|)$ time. So the worst case complexity to determine I intersections this way would be $O(I \times n)$. We now show that this can be reduced to $O(n \log n + I \log n)$ where n is the total number of edges in the two subdivisions. We first state an interesting result from Chazelle and Guibas [8]:

Lemma 1. We can preprocess a polygon Q in $O(|Q| \log |Q|)$ time requiring $O(|Q|)$ space so that it is possible to determine in $O(\log |Q|)$ time whether an arbitrary given

line segment intersects Q.

A monotone subdivision can be viewed as a set of disjoint polygons, see the example in Figure 3.11. We decompose each monotone polygon into two chains, the upper chain and the lower chain. This is done by splitting it at its leftmost and rightmost vertices.

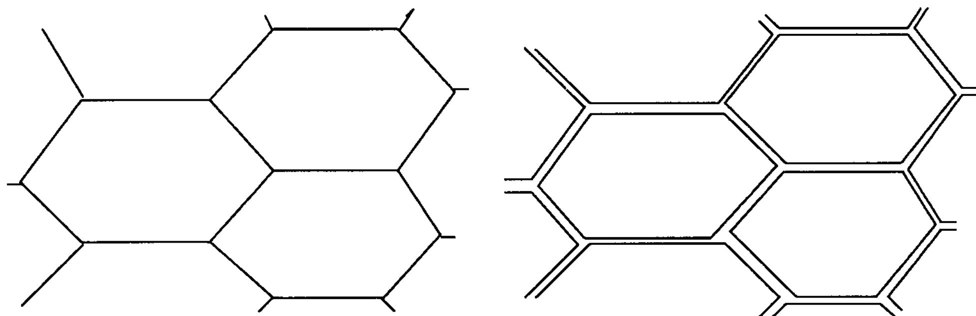


Figure 3.11: A planar subdivision and its polygonal chains

Viewing the monotone subdivision as a set of disjoint monotone polygons, the problem becomes of one testing one polygon with a set of polygons. Let us refer to the example in Figure 3.12. Polygons t_1, t_2, t_3, t_4 are from map T, and the shadowed polygon s_1 is from another map S. We use s_1^u, s_1^l to denote the upper chain and lower chain of s_1. t_i^u and t_i^l are defined similarly, $i = 1 \ldots 4$. We assume that no S-T cone has occurred so far. This means that the intersection list of s_1 is empty. When s_1 is leaving, first we find its right end point v lying inside polygon t_1, and report (s_1, t_1) as an intersection pair immediately. Next we start searching intersections along the active polygon list of T both upward and downward until we find a polygon not intersecting with s_1. Obviously, the top chain s_1^u should be used during the upward testing, and the bottom chain s_1^l should be used during the downward testing. We

only consider the downward searching. The upward searching is conducted similarly. Since the intersecion list of s_1 is empty, the first missing polygon below t_1 is obviously t_2. The next missing one is t_3, and so on. We start the searching by comparing s_1^l with t_2. Here we use the algorithm of Chazelle and Guibas [8] to determine the intersection of s_1^l with t_2. It is as follows:

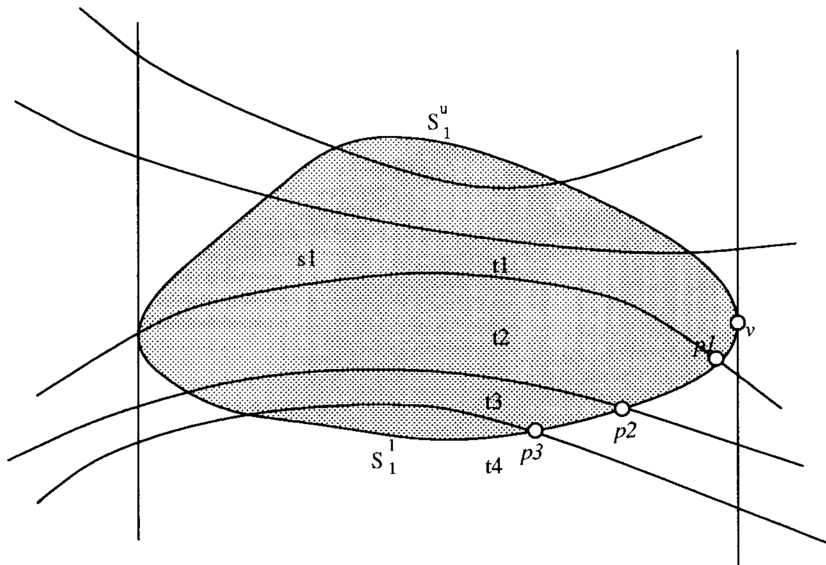


Figure 3.12: Testing the bottom chain of s_1 with polygons t_2, t_3, t_4

First of all, we represent each polygonal chain as a linked list of its edges from right to left. This edge list can be easily built while we are sweeping all the vertices from left to right. To detect the intersection of chain s_1^b with polygon t_2, we test all the edges of s_1^l from right to left against t_2, one at a time. According Lemma 1, given $O(|t_2| \log |t_2|)$ time to compute a data structure of t_2, we can determine the intersection of each edge with t_2 in $O(\log |t_2|)$ time. Then we continue the search by comparing s_1 with t_3. Let p_1 be the intersection point of s_1^l with t_2. If no intersection is found, we stop the downward searching. Since the part of s_1^l from p_1 to v

is above t_3, we can continue our testing of s_1^l with t_3 from p_1 instead of v. Finding the intersection point p_2 of s_1 with t_3, again we continue the testing from p_2 with t_4 until we encounter a polygon which does not intersect with s_1^l. As a result, we are guaranteed that when s_1 is leaving, each segment in s_1 is visited only once except that this segment intersects a polygon in the other set.

However, when an S-T cone is formed, we also test the intersection of a chain with the polygons in the other set edge by edge. Thus an edge may actually be visited many times before it leaves. This situation can be resolved by adding an field to each chain to record its rightmost point which has been visited before. In a later testing event, we stop the test once we hit that point. Thus we are guaranteed that each edge is visited only once to test the intersections unless it is found intersecting with a polygon in the other set.

The time we need to compute the data structures of all the polygons in the subdivision according the algorithm of Chazelle and Guibas [8] is $O(n \log n)$ where n is the total number of edges in the subdivisions. The total number of times we call the algorithm of Chazelle and Guibas[8] to determine the intersection of an edge with a polygon is $n + I$ where I is the number of intersections reported. So the total time we spend on testing all intersections is $(n \log n + k \log n)$.

During the implementation stage of the algorithm, we found yet another difference between line segment intersection and polygon intersection. In the line segment intersection algorithm, to test if a new left end point sits in an S-T cone, we first find

out the edges immediately above and below the point in the two active edge lists. Then we test the above edge S_a against the below edge T_b. If they intersect, an S-T cone is formed. Otherwise, we test the below edge S_b against the above line segment T_a. In this case, if one pair of the segments intersect, the other pair cannot intersect. But in case of polygons, it is possible that both pairs of polygons intersect. See the example in Figure 3.13. When the sweeping line hits the leftmost end point of s_3, two intersection pairs (s_1, t_1) and (s_2, t_2) are found. So we need to test both pairs no matter the first pair intersect or not.

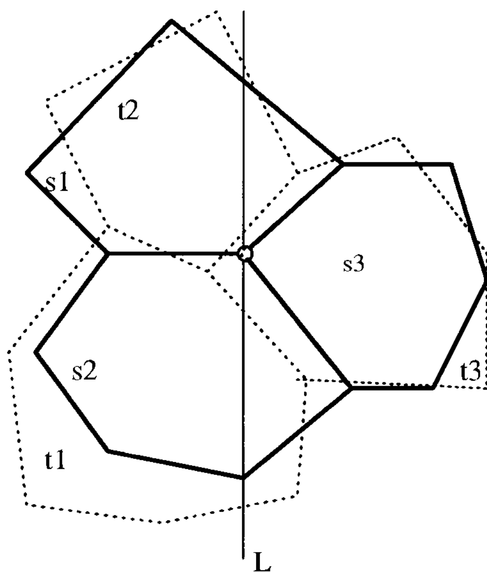


Figure 3.13: A left end point sits in two cones

We now detail the algorithm. The inputs of the algorithm are the DCEL edges of two monotone subdivisions. To simplify the processing, we still store all the active edges in a 2-3 tree although the objects which we are dealing with now are monotone polygons. This still complies with our algorithm since we can easily locate the polygon from an edge in O(1) time. Without of loss of generality, we also assume

that no two vertices can lie on the same vertical line. Unlike the algorithm of line segments where there are only two types of end points, there are four types of event vertices, illustrated in Figure 3.14. Suppose that v lies between edges c and d, and $n \geq 2, m \geq 2$, L is the sweeping line.

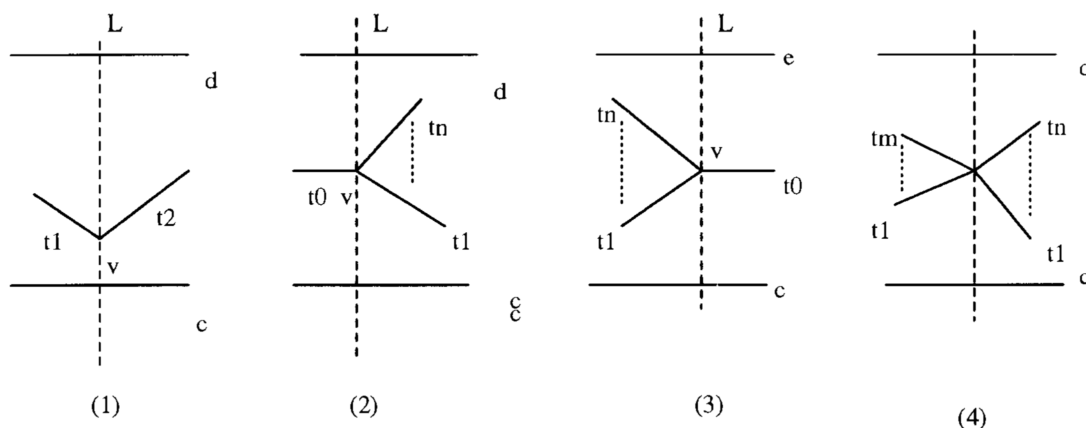


Figure 3.14: Four types of vertices

Algorithm 1 . Reporting all the pairwise intersections of polygons in two monotone planar subdivisions.

Input: two planar subdivisions of monotone polygons in DCEL form.

Output: all the pairs of polygon intersection.

Step 1. sort all the vertices in the planar subdivisions from left to right.

Step 2. initialize the active edge lists with two edges $-\infty$ and $+\infty$.

Step 3. **FOR** each vertex v **DO** (see Figure 3.14 for different cases)

- **CASE 1:** v has only two edges t_1, t_2 incident on it. t_1 is to the left of it, t_2 is to the right of it.

  - Remove t_1 from the active edge list.
  - Insert t_2 into the active edge list.

- **CASE 2:** v is a vertex with one edge to the left of it and m edges to the right of it.
  - Delete the edge to the left of it from the active edge list.
  - Insert the new edges into the active edge list.
  - Find the intersection as new polygons come in.

- **CASE 3:** v is a vertex with one edge to the right of it and m edges to the left of it.
  - Delete all the edges to the left of v from the active edge list.
  - Insert the new edge into the active edge list.
  - Find the insertions as some of the polygon are leaving.

- **CASE 4:** v is a vertex with m edges to the left of it, n edges to the right of it. (This is a general case of 2 and 3)
  - Delete all the edge to the left of v from the active edge list.
  - Insert all the edges to the right of v into the active edge list.
  - Find all the intersections as some polygons are leaving.
  - Find all the intersection as some new polygons coming in,i.e., an S-T cone is formed.

- **END CASE**

Given two maps with n edges, Step 1 of the algorithm can be executed in $O(n \log n)$, Step 2 can be executed in constant time. Step 3 consists of two parts: testing polygon

intersections runs in $O(n \log n + k \log n)$ time where k is the total number of intersection pairs; and the total time to find which polygons should be tested for intersections is $O(k \log^2 n)$. So the total running time of Algorithm 1 is $O(n \log n + k \log^2 n)$. Clearly the space needed is $O(n + k)$.

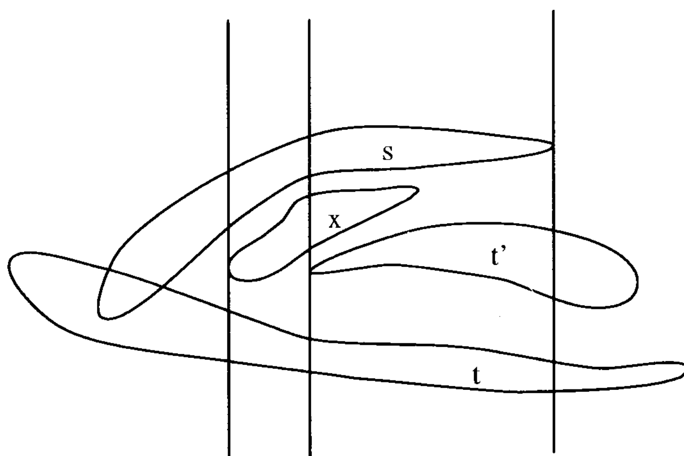


Figure 3.15: Proof of the algorithm

The algorithm detects and reports the intersections when a polygon is leaving or an S-T cone is formed. It can report all the intersecting pairs. If not, there must be an intersecting pair $(s \in S, t \in T)$ not reported. Let s leave before t. Since they intersect, when s leaves, t must be in the active edge list. The right end point of s is either above t or below t; it cannot be in t since the intersection is not reported. Suppose it is above t. There must be a polygon t' which is above t not intersecting s, otherwise t would be found intersecting s when we are testing s with the active T edges downward. We can also prove that the left end point of t' is to the right of both the left end points of s and t. See Figure 3.15, since all the polygons are monotone with respect to the X-axis, there is no way for s to intersect t if the left end point of t' is to the left of either the left end point of s and t. And the intersection of s and

t must be to the left of t'. Now back to the moment when the left end point of t' is hit by the sweeping line. We find all the S-T intersections which are blocked by t'. If (s, t) is not reported this time, there must be another polygon x which is to the left of t' which is blocking (s, t) being reported. Since the number of polygons in each set is limited, we will evently come down to a point where the intersection of (s, t) should be found if they actually intersect.

We state the above result as a theorem:

Theorem 1. Algorithm 1 finds out all the pairwise polygon intersection of two planar monotone subdivisions. The total running time is: $O(n \log n + I \log^2 n)$, and the total storage space needed is $O(n + I)$, where n is the total number of vertices in the two subdivisions, I is the total number of intersection reported.

Chapter 4

A Practical Algorithm

The algorithm proposed in the last chapter solves the monotone subdivision intersection problem in $O(n \log n + I \log^2 n)$ time where n is the total number of edges in the subdivisions and I is the number of intersecting pairs reported. The algorithm is complex and the overhead of manipulating the data structures is high. In this chapter, we modify the algorithm so that it is simple and has a low overhead cost. We will show that the algorithm has $O(n \log n + I)$ running time under the assumption that each polygon can intersect no more than a constant number of other polygons.

This chapter is organized as follows: in section 4.1, we describe the general idea of the algorithm. In section 4.2, we describe the details of the algorithm. In section 4.3, we analyze the running time of the modified algorithm under the condition mentioned above.

4.1 General Ideas

The idea of the algorithm remains the same as that of the algorithm given in the last chapter. Given two monotone subdivisions S and T, we sweep through all the vertices in both subdivisions from left to right. During the sweep, we maintain two separate lists of all the active polygons for each subdivision. The active polygons of S and T are stored in two 2-3 trees A_S and A_T respectively. The active polygons in the 2-3 trees are ordered by the increasing Y-coordinate. For each active polygon p, we also maintain a list I_p. I_p records all the polygons which are found intersecting p when an S-T cone happens. We stop the sweep at each vertex to update the data structures and to detect the intersections. Stopping at a vertex is called an event.

There are four types of vertex events, as shown in Figure 3.14 of the last chapter. In the first case, we need only to replace the leaving edge with the incoming edge. In the second case, a new polygon is becoming active. We need to find whether the new vertex is lying in some S-T cones or not. If it is in some S-T cone, we will destroy the S-T cone by detecting all the intersections blocked by it. The new polygon is also inserted into the corresponding active polygon list. In the third case, a polygon is leaving, we detect all its intersections with the polygons in the active list and report them. And we also delete it from the active polygon list. The last case is simply the combination of the second and third cases.

There are two improvements we will make in this chapter. One is how to determine whether two polygons intersect. In the last chapter, we use the algorithm of Chazelle

and Guibas [8] to compute the intersection in logarithmic time. The preprocessing cost and the execution time of that algorithm are very high. The other is how to locate the next possible intersecting polygon while we are searching along the active polygon list. This is done in the last chapter by splitting and concatenating 2-3 trees. Obviously the overhead of manipulating the trees is also very high. In this chapter, we will find alternatives for them.

4.2 The New Algorithm

Before the sweeping starts, we first initialize the active polygon lists and the intersection list of each polygon to empty. Whenever we find a new intersecting pair (s, t) where $s \in S$ and $t \in T$ during the sweeping, we insert s into I_t and t into I_s respectively. When a polygon p is leaving, we report (q, p) as intersections for every q in I_p. We also delete all the ps from every I_q. I_p is implemented as a linked list of polygons.

A monotone polygon consists of two monotone polygonal chains: the upper chain and the lower chain. Let p^u and p^l be the upper chain and the lower chain of any polygon p. We can compute the intersection of any two polygon p_1 and p_2 by comparing p_1^l against p_2^u or p_1^u against p_2^l. If we apply this method to test the intersection between polygons, an edge may be tested many times. In our algorithm, we will not test any particular edge more than once.

Intersections between polygons are detected when a polygon is leaving during the

sweeping or when a polygon is entering during the sweeping. We first consider the case when a polygon is leaving.

4.2.1 When a polygon is leaving

In Figure 4.1, the dashed polygon s_1, s_2, s_3 are from S, polygons $t_1 \dots t_5$ are from T. v is the right most vertex of s_1. When the sweeping line moves to v, we need to report the intersections between s_1 and the active polygons in A_T. Let $A_S^U(v)$ and $A_S^B(v)$ be the polygons in A_S which is above and below the point v respectively. $A_T^U(v)$ and $A_T^B(v)$ are defined similarly. We can detect all the intersections of s_1 with polygons in A_T by testing s_1^u with the polygons in $A_T^U(v)$ and s_1^l with the polygons in $A_T^B(v)$. Here we consider testing the lower chain s_1^l with the polygons in $A_T^B(v)$. The intersections between s_1^u and $A_T^U(v)$ can be computed similarly.

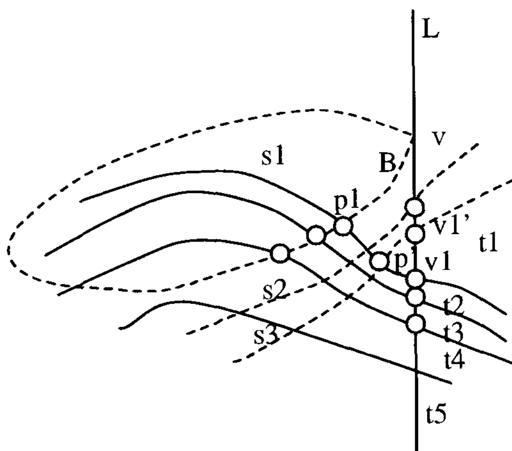


Figure 4.1: A polygon is leaving

We first find v lying inside the polygon t_1. We then detect the intersections by

testing s_1 with polygons in $A_T^B(v)$ from t_1 downward until we find a polygon not intersecting with s_1. However, some intersections may have already been detected because of S-T cones. We need to check I_{s_1} before we actually test their polygonal chains. If the pair has been reported intersecting, we move downward to the next polygon in $A_T^B(v)$. If it has not been reported intersecting, we will test their polygonal chains for intersection. Without loss of generality, we assume that the intersection list of s_1 is empty. We report (s_1, t_1) as an intersecting pair immediately.

Each polygonal chain is represented as a linked list in a way that we are able to visit all its edges from right to left. We compute the intersection between s_1^l and t_2^u by testing the their edges starting from right to left. Let the first intersection point we detected be p_1. We report (s_1, t_2) as an intersecting pair. If there were no other S polygons between s_1^l and t_2^u, the part of t_2^u which has just been visited from v_1 to p_1 could be removed since it can not intersect any other polygons. Since the starting point of the chain and the intersection point are known, the removal of partial chain takes only O(1) time. In this way, each edge of the chain will be visited only once except when it is cut.

However, there are two other S polygons between s_1^l and t_2^u. If we remove the visited part of t_2^u, the intersections of t_2 with s_2 and s_3 will not be detected. To resolve this problem, before testing t_2 with s_1, we first test the S polygons immediately above t_2. In our case, s_3 is the first polygon above t_2. So we test s_3^l against t_2^u. Let their intersection be p'. The visited chain from p' to v_1 now can be removed safely. The visited chain from p' to v'_1 can also be removed. We continue testing t_2 upward

with the polygons in $A_S^B(v_1)$ until we hit s_1. During the testing, once we find an S polygon which does not intersect t_2, we stop the searching since s_1 can not intersect t_2. After we find s_1 intersecting t_2, the next polygon we need to test against s_1 is t_3. Again, before testing the intersection between s_1 and t_3, we first test t_3 with those S polygons which is above t_3, and so on.

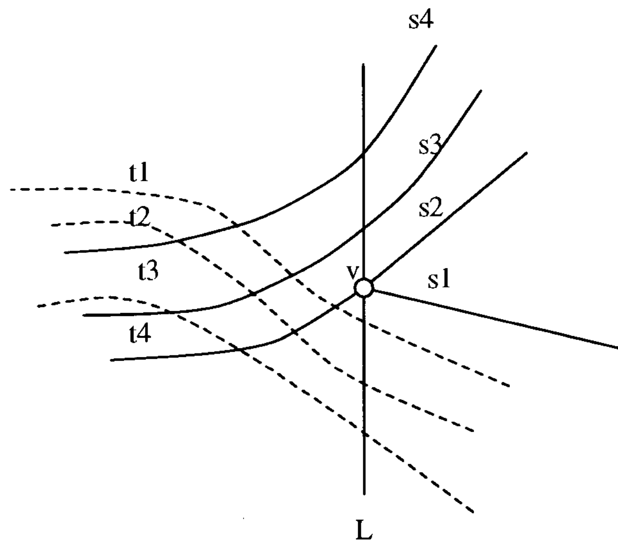


Figure 4.2: S-T cone happens when a new polygon is coming

4.2.2 When a new polygon is entering

The above case is when a polygon is leaving. The same technique also applies to the case where a new polygon is coming in, i.e., when an S-T cone is detected. Let us refer to Figure 4.2. In the picture, there are four S polygons $s_1 \ldots s_4$, depicted by solid lines. There are four T polygons $t_1 \ldots t_4$ depicted by dashed lines. When sweeping line L moves to v, a new polygon s_1 is coming in. We detect all the S-T cones by testing polygons in $A_S^B(v)$ against polygons in $A_T^U(v)$ and polygons in $A_S^U(v)$ against

polygons in $A_T^B(v)$. Here we consider testing the polygons in $A_S^U(v)$ against polygons in $A_T^B(v)$ only. The intersection between $A_S^B(v)$ and $A_T^U(v)$ can be computed similarly.

The polygons in $A_S^U(v)$ are always above the polygons in $A_T^B(v)$. Therefore we will use the lower chain of a S polygon and the upper chain of a T polygon to test for intersection. We start the testing from s_2 against $t_2 \ldots t_4$. If s_2 intersects one of them, we continue our testing with s_3 against $t_2 \ldots t_4$, and so on. The intersections of any S polygon with the T polygons can be detected by the same way we test them when the S polygon is leaving.

4.3 Running Time of the Algorithm

We now analyze the performance of the algorithm assuming that any polygon can intersect at most C other polygons where C is a constant number. We will show that under this condition the running time of the algorithm is $O(n \log n + I)$ where n is the total size of the two maps and I is the intersecting pairs reported.

The algorithm consists of two parts. The first part is to sort all the vertices, which takes $O(n \log n)$ time where n is the total size of the two subdivisions. The second part is to sweep through all the vertices and report the intersections. There are two types of vertices: extreme point(left or right) of a polygon or just a connection point. If the vertex is a connection point, we only need to update the polygonal chain, which takes $O(1)$ time. However, it is more complex if the vertex is an extreme point. We

discuss it in the following:

1. The vertex v is the right extreme point of a polygon s, $s \in S$. The polygon s is leaving. According to the algorithm, we first take $O(\log n)$ time to locate the polygon t of A_T where v lies in. We then start searching for intersections from t both downward and upward along A_T. Before testing s with any polygon in A_T, we check the intersection list I_s to see if the pair has been reported. In the worst case, all the polygons which intersects s can be in I_s. Since s can intersect no more than C polygons, I_s has a size of C. As a result, it takes constant time to go through all the intersected polygons. If there is a polygon t' intersecting s but it is not in I_s, we should test the polygonal chains of them. Before testing the intersection between s and t', we locate the first S polygon above(below) t' and check if they intersect. Under our assumption, s can only intersect C number of T polygons. If there are more than C number of S polygons bet ween s and t', s and t' can not intersect. To find the first S polygon above(below) t', we start searching along A_S from s downward(upward) until we hit t'. If we visited C number S polygons and did not hit t', we stop right here because s can not intersect t' under our assumption. As a result, this process takes only constant time. So besides the time spending on testing the edges for intersections, the total time spending at a left extreme point is $O(\log n + I_v)$ where n is the total size of the subdivisions and I_v is the number of intersections found.

2. The vertex v is the left extreme point of a polygon s, $s \in S$. In this case, we need to find the S-T cone and destroy it by detecting all the intersections blocked by it. We spend $O(\log n)$ time to insert the new polygon into the active polygon list.

In the worst case, we also spend $O(C)$ time to go through all the intersections which have been reported. The rest process is similar to that of the case when a polygon is leaving. The resulting running time at this type of vertex is also $O(\log n + I_v)$ besides the time spending on testing edge by edge for intersections.

The time we spend on testing intersections edge by edge is linear to the number of edges we visited. As we mentioned in the last section, each edge is visited only once unless it is found intersecting with other polygon. So the total time spending on testing for intersections is $O(n + I)$. As a result, the total running time of the modified algorithm is $O(n \log n + I)$ under the condition that any polygon can intersect no more than C number other polygons where C is a constant, n is the size of the two subdivisions, I is the number of intersecting pairs reported.

Chapter 5

The General Algorithm

In the last two chapters, we have successfully solved the monotone subdivision intersection problem. In this chapter, we will extend the algorithms to general subdivisions by partitioning the simple polygon into monotone pieces and applying last two chapter's algorithms to the monotone pieces. Given two polygonal subdivisions S and T of size n, we first transform them to two monotone subdivisions S' and T' of size $O(n)$ in $O(n \log n)$ time. Then we apply the last two chapters' algorithms to report the intersections between S' and T' in $O(n \log n + I' \log^2 n)$ time or $O(n \log n + I')$ time under certain condition. Here I' is the number of pairwise intersections of monotone pieces. We show that the result of monotone piece wise intersection then can be converted to polygon wise intersection in $O(I')$ time.

This chapter is organized as follows: in section 5.1, we present the partitioning of a polygonal subdivision into a monotone subdivision. Section 5.2 describes how to report each intersection only once. In section 5.3, we present the data structure used in the algorithm. Finally section 5.4 contains the complete algorithm.

5.1 Partitioning general planar subdivision to monotone planar subdivisions

A planar subdivision consists of a set of simple polygons. To partition the polygonal subdivision to a monotone subdivision, we first consider the partitioning of one single polygon into monotone pieces.

5.1.1 Partitioning one single polygon into monotone pieces

The partitioning of a simple polygon is itself a considerably interesting topic, and a lot of algorithms[9, 10, 13, 17] have been proposed over the past years. One approach which is of particularly interest to us is based on the plane sweep technique too. The algorithm works as follows: we sweep a vertical line L over the polygon from left to right, stopping at each vertex. During the sweep, we keep all the edges intersecting with L in some type of data structures. These edges currently intersecting with the sweeping line L are called active edges; the data structure which keeps all the active edges is called a sweep table. The discrete vertices of the polygon are called events. Both the polygon partitioning and the sweep table updating occur at each event vertex.

We now detail the update at each event. There are three possible types of events,

illustrated in Figure 5.1. a, b, c, d are edges of the polygon. L is the sweeping line. Suppose that v lies between edges a and b, and v is shared by c and d.

1. c is to the left of L and d is to the right. Then delete c and insert d, nothing else needs to be done.

2. Both c and d are to the right of L. Then insert both of them into the sweep table; split a into a_1 and a_2, add a new edge e into the polygon. Now we have two more monotone pieces to the right of the sweep line.

3. Both c and d are to the left of L. Then delete both of them from the sweep table; split edge b into b_1 and b_2, add a new edge e into the polygon. Now we have two less monotone pieces.

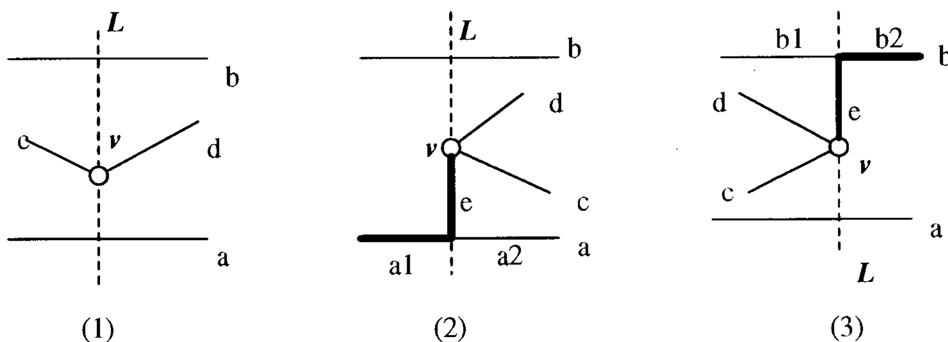


Figure 5.1: Sweeping line events

In order to sweep m vertices from left to right, we sort the vertices by the X-coordinate, since the polygon is general, this requires $O(m \log m)$ time. The sweep table can be implemented as a height-balanced tree or a 2-3 tree(this is possible due to the same reason we mentioned in last chapter, that the active edges can be totally

ordered). Therefore at each vertex, we spend $O(\log m)$ accessing and(or) updating of the sweep table. So the total running time is $O(m \log m)$. It is obvious that the storage space used is linear to the size of the polygon. We state this fact as a theorem for our future referencing.

Theorem 4.1 An m vertices polygon can be partitioned to monotone pieces in $O(m \log m)$ time and $O(m)$ space.

5.1.2 Partition a planar subdivision

The above algorithm successfully partitions one single polygon of size m to several monotone pieces in $O(m \log m)$ time. It is obviously not efficient to partition a planar subdivision by partitioning each single polygon separately. However, partitioning a set of polygons simultaneously is not difficult. The algorithm is almost identical to the single polygon algorithm except that there are six types of events now. See Figure 5.2, where $x, y \geq 2$.

1. c is to the left of L and d is to the right. Then update the sweeping table by deleting c and inserting d; nothing else needs to be done.
2. There are m edges to the right of L. Insert all of them into the sweep table; split a into a_1 and a_2, and add a new edge e. There are m more monotone pieces in the picture.

3. There are m edges to the left of L. Delete all of them from the sweep table; split b into b_1 and b_2, and add a new edge e to the polygon. There are m less monotone pieces now in the picture.

4. This case is similar to case 1 except that there are more than one edge to the right of L. So besides updating the sweep table, we should also report the new monotone pieces.

5. This case is also similar to case 1 except that there are more than one edge to the left of L. Besides updating the sweep table, we should also report some monotone pieces leaving.

6. This case is actually a generalized case of the last two. We need only to update the sweep table, and report some new monotone pieces and some leaving monotone pieces.

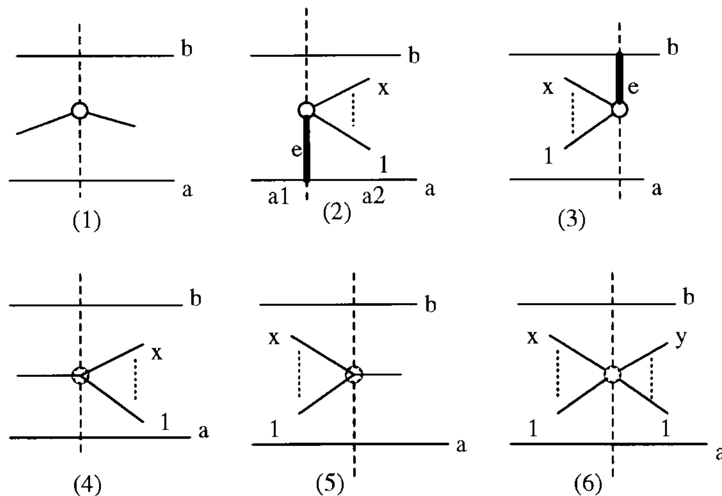


Figure 5.2: Partition a Planar Subdivision

Figure 5.3 is an example of the partition of a planar subdivision. To move the sweep line from left to right, we should sort all the vertices along the X-axis. For a

subdivision of size n, it takes $O(n \log n)$ time. With a little effort, the processing at each vertex can be done in $O(\log n)$ time, so the total running time is $O(n \log n)$. The total space needed is obviously $O(n)$. This leads to the following theorem:

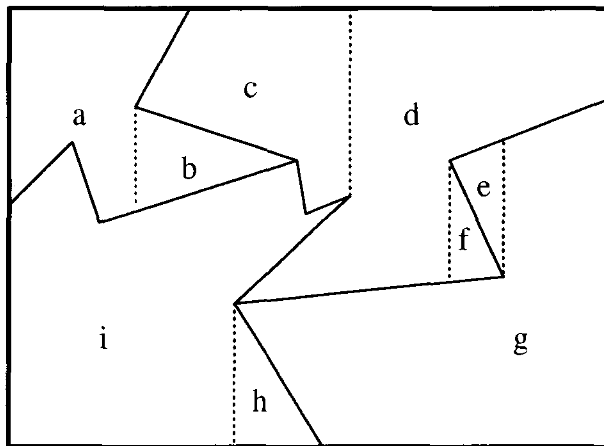


Figure 5.3: A Picture of a partitioned subdivision

Theorem 3.2. An n vertices general planar subdivision can be partitioned into a $O(n)$ vertices monotone planar subdivision in $O(n \log n)$ time.

5.1.3 Partitioning and Reporting Intersections Concurrently

The polygon partitioning and the intersection detecting can be easily processed simultaneously during the same sweeping process, i.e., it is not necessary to partition the map into monotone pieces first, then find the intersection pairs by applying the special case algorithm of last chapter. This is possible due to the fact that both partitioning and intersection detecting process maintain the same active edge lists. During the intersections finding process, we only consider the polygons to the left of

the sweep line. Therefore, it requires that only the portion of the planar subdivision which is to the left of the sweep line be monotone. This condition is guaranteed by the above mentioned partitioning algorithm. Now it remains to be shown in the following sections that it is possible to maintain a dynamic data structure of the map during the sweep, and that we can report all the intersections exactly once.

5.2 Reporting the Intersections Exactly Once

The edges of two polygons can intersect in more than one places. If we report every edge intersection once it is found, we will probably report the same polygon intersection pair many times. In the case of monotone polygons, we take care of the duplicates by the linked list, i.e. the intersections are kept in a list and only reported when the sweep passes the monotone pieces. However, a simple polygon can be split into many monotone pieces and two intersecting pieces can have exactly the same intersection polygons. One way to identify these duplications is to store all the intersection pairs in a two-dimensional array, and report them after all the intersections have been detected. Although efficient, it becomes less desirable when the number of polygons in the map gets larger, since the storage space requirement is quadratic. In this section, we propose an algorithm which requires space only linear to the total number of intersection pairs $O(I)$, at the cost of an slightly increased running time.

To achieve the above goal, the intersections we found during the sweep are kept by each monotone piece temporarily. However, they are not reported at the time the

monotone piece is leaving, but they are reported at the time all the monotone pieces of a polygon become inactive, i.e., the sweep line passes over the entire polygon, and no part of it will enter any more. Moreover, when reporting the intersections, we will not report all the intersections kept by the monotone pieces; we only report the intersections between the active polygons. By active, we mean that the sweep line still intersects at least one monotone piece of the polygon. This again prevents the same intersection from being reported twice by the polygon it intersects.

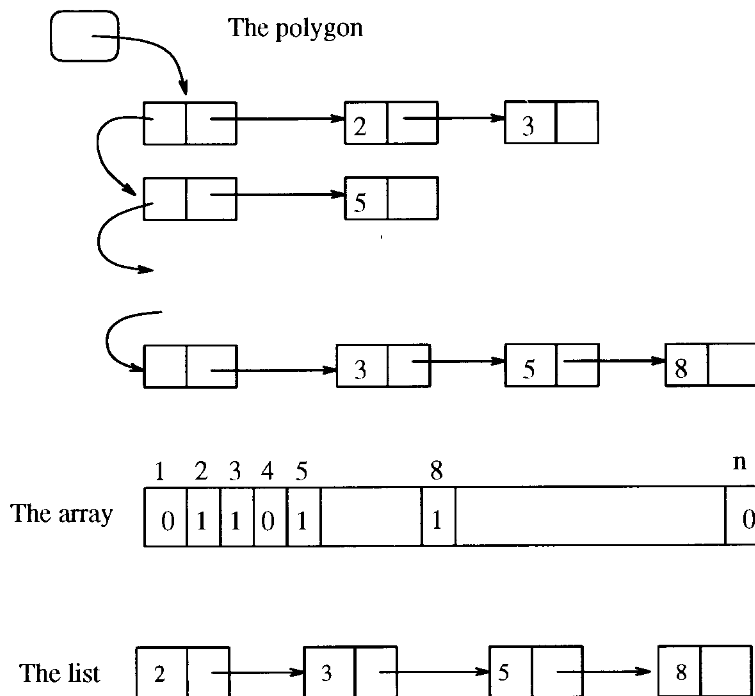


Figure 5.4: Reporting intersections when a polygon is leaving

To eliminate the duplicate intersections generated by different monotone pieces when we are reporting the intersections of a polygon, a temporary one dimensional array is used. The size of the array is the number of the polygons in the subdivisions. And the value of array can only be zeros and ones, we call the array a filter (See

Figure 5.4).

When the program starts, the filter is initialized with zeros. When a polygon is leaving, we go through all its intersecting polygons and set the corresponding cell of the filter to 1. After we have gone through all the intersections, we report them according to the value in the filter then reset the filter to zero for the next polygon to use. To report all the intersection stored in the array, we build up a link list for all the 1s in the array. We insert a node into the list at the same time we change the value of a cell from 0 to 1; we delete a node of the list when we report an intersection and change the value from 1 back to 0. As a result, we are guaranteed that each intersection pair be reported only once, and the total cost of reporting(not including detecting) is only proportional to the total number of intersections. By Theorem 3.2, we are also guaranteed that all the intersection can be found. The only part remains is to show how to keep the map in a dynamic data structure during the sweep, which is the content of the following section.

5.3 The Dynamic Data Structure of the Maps

5.3.1 Data Structure of a Monotone Piece

The input of our algorithm does not contain any explicit information about monotone pieces. Therefore all the monotone pieces must be dynamically allocated during the

partitioning process. In our algorithm, given an edge, we need to locate the corresponding polygons right away. From the DCEL, all we know is which polygon the edge belongs to, but what we really need to know is which monotone pieces it belongs to. This is done by attaching the monotone pieces to all it boundary edges when they become active. The data structure of a monotone piece is as follows:

```
MonotonePiece: {
    Chain*   UpperChain;      //pointer to its upper chain(link list of edge)
    Chain*   BottomChain;     //pointer to its bottom chain
    int      Active;          //flag to indicate if it is still active
    Tree*    intersectTree;   //the intersection list
}
```

5.3.2 Data Structure of Simple Polygon

We create a polygon object when the sweeping line encounters a new polygon, and the object is deleted when the sweeping line passes over it. The polygon object is used to keep track of all its monotone pieces, it has two objects: one intersection list, one counter which keep the number of the current active pieces. During the sweep process, when a new monotone piece is created, the counter is increased by one; when a monotone piece is deactivated, the counter is decreased by one, and all the monotone piece's intersection list is added to the polygon object. If the counter becomes zero, which means the sweeping line passes the entire polygon, we report the intersections and delete the polygon. The data structure of a polygon object is as follows:

```
Polygon {
```

```
    List*        intersectList //the list of all its intersections
    int          activeCount   //number of active pieces
}
```

5.4 The Final Algorithm

Now we shall summarize our approach of reporting the pairwise intersection of two planar subdivisions as **Algorithm 2.**

Algorithm 2 . Reporting all the pairwise polygon intersections of two planar subdivisions.

Input: two planar subdivisions of monotone polygons.

Output: all the pairs of polygon intersection.

Step 1. sort all the vertices in the planar subdivisions in increasing order by the X-axis.

Step 2. initialize the sweep table with two edges $-\infty$ and $+\infty$.

Step 3. FOR each vertex v **DO** (see Figure 5.2 for different cases)

- **CASE 1:** v is simply a vertex of c and d.
  - Remove c from the sweep table.
  - Insert d into the sweep table.
- **CASE 2:** v is the left end point of all its edges.

  - Split the polygon to make it monotonic.
  - Insert all the new edges into the sweep table.
  - Find all the intersections as a left end point of a polygon coming in, i.e. an S-T cone is formed.

- **CASE 3:** v is the right end point of all its edges.
  - Split the polygon to make it monotonic.
  - Remove those leaving edges from the Sweep Table.
  - Find all the intersecting polygons of the leaving monotone pieces. If the leaving of any monotone pieces causes the leaving of the whole polygon, report the intersection of this polygon.

- **CASE 4:** v is a connection vertex with one edge to its left and arbitrary number of edges to its right.
  - Insert all the new edges into the Sweep Table.
  - Find the intersections as new polygons coming in, i.e. an S-T cone is formed.

- **CASE 5:** v is a connection vertex with one edge to its right and arbitrary number of edges to its left.
  - Remove those leaving edges, insert the new coming edge.
  - Find the intersection of those leaving monotone pieces. If any of those polygons becomes in-active, report its intersections.

- **CASE 6:** v is a connection vertex with arbitrary number of edge to both of its sides.
  - Remove the leaving edges and insert the new edges.

  - Find all the intersections as an S-T cone is formed and some of the monotone pieces leaving. If any polygon becomes in-active, report its intersections.

- **END CASE**

The correctness of the algorithm has already been shown. However, since our algorithm employed partitioning simple polygons into monotone pieces, in which one polygon can have as many as $O(n)$ pieces or as few as one piece, the total running time of the algorithm is unknown in terms of its output size.

Chapter 6

Implementations and Conclusions

This chapter presents our implementation of the planar subdivision intersection algorithm given in chapter 4. It contains a brief introduction to the implementation environment in which the main empirical results are obtained, the special case policy, the empirical results, and finally, the conclusions.

6.1 Implementation Environment

We ran our program and performance test on Sun SparcStations IPX running SunOs4.1. The program is written in C++. The code is compiled by SUN C++ 2.0. Performance was measured by the CPU running time. The testing data is stored in the Unix file system.

Six different sets of data are tested in the experiment, with two maps in each

set of data. The number of polygons in each set ranging from 200 to 21,000. These data set are generated randomly using the straight line method from Hong Fan. The algorithm, however, is designed flexibly for maps with arbitrary number of polygons.

6.2 Special Case Policy

Most geometry algorithms need to deal with special cases, so ours is no exception. As you may have already noticed, the sweep line technique employs a vertical line sweeping through the plane from left to right; so what if there are two points on the same vertical line? For dealing with special cases, four major method may be distinguished:

1. Rule them out: require that the input data be restricted such that the special case never happen.
2. Write extra code to handle the special cases properly.
3. Find a different geometric system in which the special cases disappear.
4. Find another representation of the same idea to avoid the special case.

Since we are not to apply our algorithm to a real world application, to simplify our implementation, we adopt the ruling out (1) policy. This does not mean that our algorithm cannot deal with the special cases. With a little care, these cases can be solved without increasing the complexity of the algorithm.

6.3 Empirical Results

The test results are listed in Table 6.1 and Table 6.2. Table 6.1 shows the data size and the outcome of the testing. Each set of data has two maps. The first two columns shows the number of edges and number of polygons in each map. In our algorithm, we need to partition the polygon into monotone pieces. The third column records the number of the monotone pieces we generated from the original map. This number of monotone pieces is almost eight times the number of polygons. The rest of the columns shows the number of intersection reported. The fourth column shows the number of monotone pieces intersections. It is almost five times that of polygon intersections. Which means that there is a fair amount of redundant testing in our algorithm.

Table 6.2 shows the running time of our algorithm comparing with Mairson's line segment intersection algorithm. Both algorithms read in the same data files. The data file stores the array of DCEL edges in each planar subdivision. We do not include the I/O time here. The running time is the sum of *user* time and *system* time.

The table lists the running time of each stage of the algorithms. Each algorithm has been divided into three stages: preprocessing, sorting and sweeping. The preprocessing of Mairson's algorithm involves initilization of the results space. The preprocessing of our algorithm involves extracting the vertex information from the input file. From Table 6.2, we can see that the preprocessing time of Mairson's algorithm is a little more than our algorithm. The reason is that the storage space needed to

Data Set	Input Size			Intersection Report		
	Edge	Polygon	MonoPieces	Pieces	Edge	Polygon
1	328, 812	19, 43	145, 339	623	132	139
2	2104,2018	109, 162	801, 1260	2731	578	642
3	5826, 6700	304, 350	2504, 2914	7473	1699	1714
4	11076,11526	540,571	4856, 4927	13095	2816	2836
5	16746,16284	830, 803	7416, 6765	19577	4463	4414
6	20986,22168	1056,1105	8807,9629	25007	5756	5575

Table 6.1: General planar subdivision intersections

Data Set	Mairson's(sec.)			Ours(sec.)		
	Preprocess	Sort	Sweep	Preprocess	Sort	Sweep
1	0.04	0.2	0.3	0.03	0.05	0.18
2	0.2	0.7	1.4	0.2	0.2	0.78
3	0.5	1.9	3.8	0.48	0.56	2.2
4	1.1	3.4	7.4	0.83	1.03	3.93
5	2.0	4.9	11.7	1.3	1.5	6.1
6	3.1	7.5	16.0	1.7	2.2	8.1

Table 6.2: Running time of general planar subdivision intersections

be initilized is $O(n^2)$. The sorting time of Mairson's algorithm is also more than that of our algorithm. The reason for this is that each vertex is considered only once during the sorting in our algorithm, but it is considered v number of times in Mairson's algorithm where v is the number of edges incident on that vertex. Our algorithm is faster in sweeping too. Also, Mairson's algorithm only reports edge intersections. Extra effort is needed to convert the edge intersection results to polygon intersections.

From the result, we can also see that the running time of the program does not

increase very fast as the size of the data sets increases. This is a very import feature we always want from our algorithm.

6.4 Conclusion

In this thesis, we have proposed an algorithm to solve the planar subdivision problem. We have also shown several ways to construct planar subdivisions randomly. In case of monotone planar subdivision with n edges and I intersection pairs, we have proposed an algorithm that runs in $O(n \log n + I \log^2 n)$ time, and take $O(n + I)$ storage space. We have also proposed a practical algorithm which has a $O(n \log n + I)$ worst case running time under the condition that any monotone polygon can not intersect more than constant number of other polygons. We use this algorithm to solve the general planar subdivision intersection algorithm by partitioning the simple polygons into monotone pieces. But the output sensitive running time of the general planar subdivision case is unknown. The experimental results indicates that the algorithm has a satisfactory performance.

A few open problems emerge as the results of this thesis. One is that whether it is possible to find an $O(n \log n + I)$ running time algorithm to report the intersections between two monotone subdivisions under no conditions. The other is to analyze the running time of the algorithm given in chapter 5. Finally, what is the best running time to report the intersections between two general subdivisions.

Bibliography

[1] A.V. Aho, J.E.Hopcroft, and J.D. Ullman, The Design and Analysis of Computer Algorithms, Reading, MA: Addison-Wesley, 1974.

[2] A.V. Aho, J.E.Hopcroft, and J.D. Ullman, Data Structures and Algorithms, Reading, MA: Addison-Wesley, 1983.

[3] J.L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching", Communications ACM, vol 18, No. 9, pp.509-517, 1975.

[4] J.L.Bentley and T. Ottmann, "Algorithms for reporting and counting geometric intersections", IEEE Transaction on Computers., vol C-28, pp.643-647, Sep, 1979.

[5] J.L. Bentley and D. Wood, "An optimal worst case algorithm for reporting intersections of rectangles", IEEE Transaction on Computers, vol C-29, No.7, pp. 571-576, Jul, 1980.

[6] T. Brinkhoff, H. P. Kriegel and B. Seeger "Efficient Processing of Spatial Joins Using R-trees", Proceedings of 1993 ACM-SIGMOD Conference Management of Data, pp. 237-246, May 1993.

[7] B. Chazelle, "A theorem on polygon cutting with applications", Proceedings of 23rd Annual FOCS Symposium., pp. 339-349, 1982.

[8] B. Chazelle and L. J. Guibas, "Visibility and Intersection Problems in Plane Geometry", Computational geometry. Proceedings of the symposium. ACM. pp. 135-146, 1985.

[9] E. Edelsbrunner, L.J. Guibas and J.Stolfi, "Optimal point location in a monotone subdivision", DEC/SRC Technical Report No.2, 1984.

[10] E. Edelsbrunner, L.J. Guibas and J.Stolfi, "Optimal point location in a monotone subdivision", SIAM Journal of Computer, 15(2), pp. 317-340, 1986.

[11] H. Fan, Spatial Join: A Study of Complex Spatial Operation and Its Underlying Spatial Indexing Methods, Msc. Thesis, Simon Fraser University, pp. 18-29, 1992.

[12] S. Fortune, "A Sweepline Algorithm for Voronoi Diagrams", Proceedings of 2nd Annual Symposiym on Computational Geometry", ACM, pp. 313-322, 1986.

[13] S. Fournier and D.Y. Montuno "Triangulating simple polygons and equivalent problems", ACM Transaction on Graphics 3, 153-174, 1984.

[14] L. Guibas, L. Ranshaw, and J. Stolfi, "A kinetic framework for computational geometry", Proceeding of 24th Annual FOCS Symposium, pp. 100-111, 1983.

[15] O. Günther, "Efficient Computation of Spatial Joins", Proceedings 9th International Conference on Data Engineering, pp. 50-60, April 1993.

[16] A.Guttman, "R-Trees: A dynamic index structure for spatial searching", Proc. ACM SIGMOD Conference on Management of Data, 1984.

[17] D.T.Lee and F.P. Preparata "Location of a point in a planar subdivision and its applications," SIAM Journal of Computer, vol. 6, no. 3, pp. 594-606, Sept. 1977.

[18] D.T.Lee and F.P.Preparata "Computational Geometry — A Survey", IEEE Transaction on Computers, vol C-33, No. 12, pp.1072-1101, Dec, 1984.

[19] H.G. Mairson and J. Stolfi, "Reporting and counting line segment intersections," Department of Computer Science, Stanford Univ., Stanford, CA, Extended Abstract, 1984.

[20] F. Preparata and M. Shamos, "Computational Geometry : an introduction", Springer-Verlag, 1985.

[21] D. Rotem, "Spatial Join Indices", Proceedings of 7th International Conference on Data Engineering, Kobe, Japan, pp. 500-509, April 1991.

[22] M. Shamos, "Geometric Complexity", Seventh ACM Annual Symposium on Theory of Computing, pp. 224-233, May 1975.

[23] M.I. Shamos and D.J. Hoey, "Geometric intersection problems" in Proceeding of 17th Annual IEEE Symposium on Foundations of Computer Science, pp 208-215, 1976.