Computational Geometry: Methods and Applications

Jianer Chen

Computer Science Department
Texas A&M University

February 19, 1996

Chapter 1

Introduction

Geometric objects such as points, lines, and polygons are the basis of a broad variety of important applications and give rise to an interesting set of problems and algorithms. The name geometry reminds us of its earliest use: for the measurement of land and materials. Today, computers are being used more and more to solve larger-scale geometric problems. Over the past two decades, a set of tools and techniques has been developed that takes advantage of the structure provided by geometry. This discipline is known as *Computational Geometry*.

The discipline was named and largely started around 1975 by Shamos, whose Ph.D. thesis attracted considerable attention. After a decade of development the field came into its own in 1985, when three components of any healthy discipline were realized: a textbook, a conference, and a journal. Preparata and Shamos's book *Computational Geometry: An Introduction* [23], the first textbook solely devoted to the topic, was published at about the same time as the first ACM Symposium on Computational Geometry was held, and just prior to the start of a new Springer-Verlag journal *Discrete and Computational Geometry*. The field is currently thriving. Since 1985, several texts, collections, and monographs have appeared [1, 10, 18, 20, 25, 26]. The annual symposium has attracted 100 papers and 200 attendees steadily. There is evidence that the field is broadening to touch geometric modeling and geometric theorem proving. Perhaps most importantly, the first students who obtained their Ph.D.s in computer science with theses in computational geometry have graduated, obtained positions, and are now training the next generation of researchers.

Computational geometry is of practical importance because Euclidean space of two and three dimensions forms the arena in which real physical objects are arranged. A large number of applications areas such as pattern recognition [28], computer graphics [19], image processing [22], operations research, statistics [4, 27], computer-aided design, robotics [25, 26], etc., have been the incubation bed of the discipline since they provide inherently geometric problems for which efficient algorithms have to be developed. A large number of manufacturing problems involve wire layout, facilities location, cutting-stock and related geometric optimization problems. Solving these efficiently on a high-speed computer requires the development of new geometrical tools, as well as the application of fast-algorithm techniques, and is not simply a matter of translating well-known theorems into computer programs. From a theoretical standpoint, the complexity of geometric algorithms is of interest because it sheds new light on the intrinsic difficulty of computation.

In this book, we concentrate on four major directions in computational geometry: the construction of convex hulls, proximity problems, searching problems and intersection problems.

Chapter 2

Algorithmic Foundations

For the past twenty years the analysis and design of computer algorithms has been one of the most thriving endeavors in computer science. The fundamental works of Knuth [14] and Aho-Hopcroft-Ullman [2] have brought order and systematization to a rich collection of isolated results, conceptualized the basic paradigms, and established a methodology that has become the standard of the field. It is beyond the scope of this book to review in detail the material of those excellent texts, with which the reader is assumed to be reasonably familiar. It is appropriate however, at least from the point of view of terminology, to briefly review the basic components of the language in which computational geometry will be described. These components are algorithms and data structures. Algorithms are programs to be executed on a suitable abstraction of actual "von Neumann" computers; data structures are ways to organize information, which, in conjunction with algorithms, permit the efficient and elegant solution of computational problems.

2.1 A Computational model

Many formal models of computation appear in the literature. There is no general consensus as to which of these is the best. In this book, we will adopt the most commonly-used model. More specifically, we will adopt random access machines (RAM) as our computational model.

Random access machine (RAM)

A random access machine (RAM) models a single-processor computer with a random access memory.

A RAM consists of a read-only input tape, a write-only output tape, a program and a (random access) memory. The memory consists of registers each capable of holding a real number of arbitrary precision. There is also no upper bound on the memory size. All computations take place in the processor. A RAM can access (read or write) any register in the memory in one time unit when it has the correct address of that register.

The following operations on real numbers can be done in unit time by a random access machine :

1) Arithmetic operations: $*$, $/$, $+$, $-$, log, exp, sin.
2) Comparisons
3) Indirect access

2.2 Complexity of algorithms and problems

The following notations have become standard:

- $O(f(n))$: the class C_1 of functions such that for any $g \in C_1$, there is a constant c_g such that $f(n) \geq c_g g(n)$ for all but a finite number of n's. Roughly speaking, $O(f(n))$ is the class of functions that are at most as large as $f(n)$.

- $o(f(n))$: the class C_2 of functions such that for any $g \in C_2$, $\lim_{n\to\infty} g(n)/f(n) = 0$. Roughly speaking, $o(f(n))$ is the class of functions that are less than $f(n)$.

- $\Omega(f(n))$: the class C_3 of functions such that for any $g \in C_3$, there is a constant c_g such that $f(n) \leq c_g g(n)$ for all but a finite number of n's. Roughly speaking, $\Omega(f(n))$ is the class of functions which are at least as large as $f(n)$.

- $\omega(f(n))$: the class C_4 of functions such that for any $g \in C_4$, $\lim_{n\to\infty} f(n)/g(n) = 0$. Roughly speaking, $\omega(f(n))$ is the class of functions that are larger than $f(n)$.

- $\Theta(f(n))$: the class C_5 of functions such that for any $g \in C_5$, $g(n) = O(f(n)$ and $g(n) = \Omega(f(n))$. Roughly speaking, $\Theta(f(n))$ is the class of functions which are of the same order as $f(n)$.

Complexity of algorithms

Let $\mathcal{A}$ be an algorithm implemented on a RAM. If for an input of size n, $\mathcal{A}$ halts after m steps, we say that the running time of the algorithm $\mathcal{A}$ is m on that input.

There are two types of analyses of algorithms: worst case and expected case. For the worst case analysis, we seek the maximum amount of time used by the algorithm for all possible inputs. For the expected case analysis we normally assume a certain probabilistic distribution on the input and study the performance of the algorithm for any input drawn from the distribution. Mostly, we are interested in the asymptotic analysis, i.e., the behavior of the algorithm as the input size approaches infinity. Since expected case analysis is usually harder to tackle, and moreover the probabilistic assumption sometimes is difficult to justify, emphasis will be placed on the worst case analysis. Unless otherwise specified, we shall consider only worst case analysis.

Definition Let $\mathcal{A}$ be an algorithm. The *time complexity* of $\mathcal{A}$ is $O(f(n))$ if there exists a constant c such that for every integer $n \geq 0$, the running time of $\mathcal{A}$ is at most $c \cdot f(n)$ for all inputs of size n.

Complexity of problems

While time complexity for an algorithm is fixed, this is not so for problems. For example, Sorting can be implemented by algorithms of different time complexity. The time complexity of a known algorithm for a problem gives us the information about *at most* how much time we need to solve the problem. We would also like to know the *minimum* amount of time we need to solve the problem.

Definition A function $u(n)$ is an *upper bound* on the time complexity of a problem $\mathcal{P}$ if there is an algorithm $\mathcal{A}$ solving $\mathcal{P}$ such that the running time of $\mathcal{A}$ is $u(n)$. A function $l(n)$ is a *lower bound* on the time complexity of a problem $\mathcal{P}$ if *any* algorithm solving $\mathcal{P}$ has time complexity at least $l(n)$.

2.3 A data structure supporting set operations

A *set* is a collection of *elements*. All elements of a set are different, which means no set can contain two copies of the same element.

When used as tools in computational geometry, elements of a set usually are normal geometric objects, such as points, straight lines, line segments, and planes in Euclidean spaces.

We shall sometimes assume that elements of a set are linearly ordered by a relation, usually denoted "$<$" and read "less than" or "precedes". For example, we can order a set of points in the 2-dimensional Euclidean space by their x-coordinates.

Let S be a set and let u be an arbitrary element of a universal set of which S is a subset. The fundamental operations occurring in set manipulation are:

- MEMBER(u, S): Is $u \in S$?
- INSERT(u, S): Add the element u to the set S.
- DELETE(u, S): Remove the element u from the set S.

When the universal set is linearly ordered, the following operations are very important:

- MINIMUM(S): Report the minimum element of the set S.
- SPLIT(u, S): Partition the set S into two sets S_1 and S_2, so that S_1 contains all the elements of S which are less than or equal to u, and S_2 contains all the elements of S which are larger than u.
- SPLICE(S, S_1, S_2): Assuming that all elements in the set S_1 are less than any element in the set S_2, form the ordered set $S = S_1 \cup S_2$.

We will introduce a special data structure: 2-3 trees, which represent sets of elements and support the above set operations efficiently.

Definition A 2-3 tree is a tree such that each non-leaf node has two or three children, and every path from the root to a leaf is of the same length.

The proof of the following theorem is straightforward and left to the reader.

Theorem 2.3.1 *A 2-3 tree of n leaves has height $O(\log n)$.*

A linearly ordered set of elements can be represented by a 2-3 tree by assigning the elements to the leaves of the tree in such a way that for any non-leaf node of the tree, all elements stored in its first child are less than any elements stored in its second child, and all elements stored in its second child are less than any elements stored in its third child (if it has a third child).

Each non-leaf node v of a 2-3 tree keeps three pieces of information for the corresponding subtree.

- $L(v)$: the largest element stored in the subtree rooted at its first child.
- $M(v)$: the largest element stored in the subtree rooted at its second child.
- $H(v)$: the largest element stored in the subtree rooted at its third child (if it has one).

2.3.1 Member

The algorithm for deciding the membership of an element in a 2-3 tree is given as follows, where T is a 2-3 tree, t is the root of T, and u is the element to be searched in the tree.

```
Algorithm  MEMBER(T, u)

  BEGIN
    IF  T is a leaf node then report properly
    ELSE IF  L(t) >= u then MEMBER(child1(T), u)
    ELSE IF  M(t) >= u then MEMBER(child2(T), u)
    ELSE IF  t has a third child
         THEN  MEMBER(child3(T), u)
         ELSE  report failure.
  END
```

Since the height of the tree is $O(\log n)$, and the algorithm simply follows a path in the tree from the root to a leaf, the time complexity of the algorithm MEMBER is $O(\log n)$.

2.3.2 Insert

To insert a new elment x into a 2-3 tree, we proceed at first as if we were testing membership of x in the set. However, at the level just above the leaves, we shall be at a node v that should be the parent of x. If v has only two children, we simply make x the third child of v, placing the children in the proper order. We then adjust the information contained by the node v to reflect the new situation.

Suppose, however, that x is the fourth child of the node v. We cannot have a node with four children in a 2-3 tree, so we split the node v into two nodes, which we call v and v'. The two smallest elements among the four children of v stay with v, while the two larger elements become children of node v'. Now, we must inset v' among the children of p, the parent of v. The problem now is solved recursively.

One special case occurs when we wind up splitting the root. In that case we create a new root, whose two children are the two nodes into which the old root was split. This is how the number of levels in a 2-3 tree increases.

The above discussion is implemented as the following algorithms, where T is a 2-3 tree and x is the element to be inserted.

```
Algorithm  INSERT(T, x)

  BEGIN
  1.  Find the proper node v in the tree T such that
      v is going to be the parent of x;
  2.  Create a leaf node d for the element x;
  3.  ADDSON(v, d)
  END
```

Where the procedure ADDSON is implemented by the following recursive algorithm, which adds a child d to a non-leaf node v in a 2-3 tree.

```
Algorithm  ADDSON(v, d)

  BEGIN
  1.  IF v is the root of the tree, add the node d properly.
      Otherwise, do the following.
  2.  IF v has two children, add d directly
  3.  ELSE
  3.1.  Suppose v has three children c1, c2, and c3.  Partition c1,
```

```
       c2, c3 and d properly into two groups (g1, g2) and (g3, g4).
       Let v be the parent of (g1, g2) and create a new node v' and
       let v' be the parent of (g3, g4).
  3.2. Recursively call ADDSON(father(v), v').
  END
```

Analysis: The algorithm INSERT can find the proper place in the tree for the element x in $O(\log n)$ time since all it needs to do is to follow a path from the root to a leaf. Step 2 in the algorithm INSERT can be done in constant time. The call to the procedure ADDSON in Step 3 can result in at most $O(\log n)$ recursive calls to the procedure ADDSON since each call will jump at least one level up in the 2-3 tree, and each recursive call takes constant time to perform Steps 1, 2, and 3.1 in the algorithm ADDSON. So Step 3 in the algorithm INSERT takes also $O(\log n)$ time. Therefore, the overall time complexity of the algorithm INSERT is $O(\log n)$.

2.3.3 Minimum

Given a 2-3 tree T we want to find out the minimum element stored in the tree. Recall that in a 2-3 tree the numbers are stored in leaf nodes in *ascending* order from left to right. Therefore the problem is reduced to going down the tree, always selecting the left most link, until a leaf node is reached. This leaf node should contain the minimum element stored in the tree. Evidently, the time complexity of this algorithm is $O(\log n)$ for a 2-3 tree with n leaves.

```
Algorithm  MINIMUM(T, min)

  BEGIN
     IF T is a leaf THEN
       min := T;
     ELSE  call MINIMUM(child1(T), min);
  END
```

2.3.4 Delete

When we delete a leaf from a 2-3 tree, we may leave its parent v with only one child. If v is the root, delete v and let its lone child be the new root. Otherwise, let p be the parent of v. If p has another child, adjacent to v on either the right or the left, and that child of p has three children, we can transfer the proper one of those three to v. Then v has two children, and we are done.

If the children of p adjacent to v have only two children, transfer the lone child of v to an adjacent sibling of v, and delete v. Should p now have only one child, repeat all the above, recursively, with p in place of v.

Summarizing these discussions together, we get the algorithm DELETE, as shown below. Where procedure DELETE() is merely a driver for sub-procedure DEL() in which the actual work is done.

The variables *done* and *1son* in DEL() are boolean flags used to indicate successful deletion and to detect the case when a node in the tree has only one child, respectively.

In the worst case we need to traverse a path in the tree from root to a leaf to locate the node to be deleted, then from that leaf node to the root, in case that every non-leaf node on the path has only two children in the original 2-3 tree T. Thus the time complexity of DELETE algorithm for a tree with n nodes is $O(\log n)$.

```
Algorithm  DELETE(T, x)

   BEGIN
      Call DEL(T, x, done, 1son);
      IF  done is true  THEN
        IF  1son is true  THEN  T := child1(T)
      ELSE  x was not found in T, handle properly
   END
```

```
Algorithm  DEL(T, x, done, 1son)

   BEGIN
1.    IF  children of T are leaves  THEN  process properly, i.e., if
      x is found, delete it; update the variables done and 1son;
2.    ELSE  IN CASE OF
             x <= L(T):   son := child1(T);
             L(T) < x <= M(T):   son := child2(T);
             M(T) < x <= H(T):  son := child3(T);
3.    Call DEL(son, x, done, 1son1);
4.    IF  1son1 is true  THEN
4.1.     IF  the node T has another child b that has three children,
         THEN  reorganize the grandchildren among the nodes son and
```

```
        b to make both have two children, and set  lson := false;
4.2.    ELSE  make the only child of the node son a child of a
        sibling of it, and delete the node son from T.  If T has
        only one child then set lson := true.
   END
```

2.3.5 Splice

Splicing two trees into one big tree is a special case of the more general operation of merging two trees. Splice assumes that all the keys in one of the trees are larger than all those in the other tree. This assumption effectively reduces the problem of merging the trees into "pasting" the smaller tree into a proper position in the larger tree. "Pasting" the smaller tree is actually no more than performing an ADDSON operation to a proper node in the larger tree.

To be more specific, let T_1 and T_2 be 2-3 trees which we wish to splice into the 2-3 tree T, where all keys in T_1 are smaller than those in T_2. Furthermore, assume that the height of T_1 is less than or equal to that of T_2 so that T_1 is "pasted" to T_2 as a left child of a leftmost node at the proper level in T_2. In the case where the heights are equal, both T_1 and T_2 are made children of the common root T; otherwise the proper level in T_2 is given by

$$height(T_2) - height(T_1) - 1$$

It is clear that the algorithm SPLICE runs in time $O(\log n)$. In fact, the running time is proportional to the height difference $height(T_2) - height(T_1) - 1$.

The implementation of the algorithm SPLICE is given below.

```
Algorithm SPLICE(T, T1, T2)

  { Suppose that all elements in T1 are less than any elements in T2,
    and that the height of T1 is at most that of T2.  Other cases can
    be dealt with similarly.}

   BEGIN
     IF  height(T1) = height(T2)
     THEN  make T a parent of T1 and T2.
     ELSE
       WHILE  height(T2)-1 > height(T1)  DO
```

```
        T2 := child1(T2)
     Call ADDSON(T2, T1).
   END
```

2.3.6 Split

By splitting a given 2-3 tree T into two 2-3 trees, T_1 and T_2, at a given element x, we mean to split the tree T in such a way that all elements in T that are less than or equal to x go to T_1 while the remaining elements in T go to T_2.

The idea is as follows: as the tree is searched for x, we store the subtrees to the left and right of the traversed path (split path). For this purpose two stacks are used, one for each side of the split path. As we go deeper into T, subtrees are pushed into the proper stack. Finally, the subtrees in each stack are spliced together to form the desired trees T_1 and T_2, respectively. The algorithm is given as follows.

```
Algorithm  SPLIT(T, x, T1, T2)

   {Split T into T1 and T2 such that all elements in T1 are less
    than or equal to x,  and all elements in T2 are greater than x.
    The stacks S1 and S2 are used to store the subtrees to the left
    and right of the path in the 2-3 tree T from the root to the
    leaf x, respectively.}

   BEGIN
   1.  WHILE  T is not leaf  DO
          IF  x <= L(T)  THEN
            S2 <-- child3(T), child2(T);
            T := child1(T);
          IF  L(T) < x <= M(T)  THEN
            S1 <-- child1(T);   S2 <-- child3(T);
            T := child2(T);
          IF  M(T) < x <= H(T)  THEN
            S1 <-- child1(T), child2(T);
            T := child3(T);
     {Reconstruct T1}
   2.  T1 <-- S1;
   3.  WHILE  S1  is not empty  DO
         t <-- S1;
         Call  SPLICE(T1, t, T1);
```

```
  {Reconstruct T2}
4.  T2 <-- S2;
5.  WHILE  S2  is not empty  DO
      t <-- S2;
      Call  SPLICE(T2, T2, t);
END
```

It is easy to see that the WHILE loop in Step 1 takes time $O(\log n)$. The analysis for the rest of the algorithm is a bit more complicated. Note that the use of the stacks S1 and S2 to store the subtrees guarantees that the height of a subtree closer to a stack top is less than or equal to the height of the subtree immediately deeper in the stack. A crucial observation is that since we splice shorter trees first (which are on the top part of the stacks), the difference between the heights of two trees to be spliced is always very small. In fact, the total time spent on splicing all these subtrees is bounded by $O(\log n)$. We give a formal proof as follows.

Assume that the subtrees stored in stack S1 are

$$t_1, t_2, \cdots, t_r \tag{2.1}$$

in the order from the stack top to stack bottom. Let $h(t)$ be the height of the 2-3 tree t. According to the algorithm SPLIT, we have

$$h(t_1) \leq h(t_2) \leq \cdots \leq h(t_r)$$

and no three consecutive subtrees in the stack have the same height. Thus, we can partition sequence (1) into "segments" which contains the subtrees of the same height in the sequence:

$$s_1, s_2, \cdots, s_q$$

Each s_i either is a single subtree or consists of two consecutive subtrees of the same height in sequence (1). Moreover, $q \leq \log n$. Let $h(s_i)$ be the height of the subtrees contained in the segment s_i. We have

$$h(s_1) < h(s_2) < \cdots < h(s_q)$$

The WHILE loop in Step 3 first splices the subtrees in segment s_1 into a single 2-3 tree $T_1^{(1)}$, then recursively splices the 2-3 tree $T_1^{(i-1)}$ and the subtrees in segment s_i into a 2-3 tree $T_1^{(i)}$, for $i = 2, \ldots, q$. We have the following lemma.

Lemma 2.3.2 *For all $i = 2, \ldots, q$, we have*

$$h(s_{i-1}) \leq h(T_1^{(i-1)}) \leq h(s_i) < h(s_{i+1}) < \cdots < h(s_q)$$

PROOF. That $h(s_1) \leq h(T_1^{(1)})$ is fairly clear since $T_1^{(1)}$ is obtained by splicing subtrees in the segment s_1. For $i > 2$, since $T_1^{(i-1)}$ is obtained by splicing the tree $T_1^{(i-2)}$ and the subtrees in s_{i-1}, and the subtrees in s_{i-1} have height $h(s_{i-1})$. Thus, the height of the 2-3 tree $T_1^{(i-1)}$ is at least $h(s_{i-1})$.

Now we prove the rest inequalities.

Since the 2-3 tree $T_1^{(1)}$ is obtained by splicing the subtrees in the segment s_1, and segment s_1 contains at most two subtrees, both of height $h(s_1)$. Thus, the height of the 2-3 tree $T_1^{(1)}$ is at most $h(s_1) + 1$, which is not larger than $h(s_2)$. Thus, the lemma is true for the case $i = 2$.

Now assume that the lemma is true for the case $i - 1$,

$$h(T_1^{(i-1)}) \leq h(s_i) < h(s_{i+1}) < \cdots < h(s_q)$$

We consider the height of the 2-3 tree $T_1^{(i)}$.

If the segment s_i is a single subtree t_i of height $h(s_i)$, then splicing the tree $T_1^{(i-1)}$ of height at most $h(s_i)$ and the tree t_i of height $h(s_i)$ results in a 2-3 tree $T_1^{(i)}$ of height at most $h(s_i) + 1$, which is not larger than $h(s_{i+1})$.

Now suppose that the segment s_i consists of two subtrees t_i' and t_i'' of height $h(s_i)$. Since the height of the tree $T_1^{(i-1)})$ is at most $h(s_i)$ by the inductive hypothesis, splicing the trees $T_1^{(i-1)})$ and t_i' results in a 2-3 tree T' of height at most $h(s_i) + 1$. Moreover, according the algorithm SPLICE, if the height of T' is $h(s_i) + 1$, then the root of T' has only two children. Now splice the trees T' and t_i'' into the 2-3 tree $T_1^{(i)}$: If the height of the tree T' is smaller than $h(s_i) + 1$, then splicing T' and the subtree t_i'' of height $h(s_i)$ results in a tree $T_1^{(i)}$ of height at most $h(s_i) + 1$, which is not larger than $h(s_{i+1})$. On the other hand, if the height of the tree T' is $h(s_i) + 1$, then the root of T' has only two children, thus splicing T' and t_i'' will not create a new root and the resulting tree $T_1^{(i)}$ has height $h(s_i) + 1$, again not larger than $h(s_{i+1})$. This concludes that we always have

$$h(T_1^{(i)}) \leq h(s_{i+1}) < h(s_{i+2}) < \cdots < h(s_q)$$

This completes the induction proof and shows the correctness of the lemma. $\square$

Now we are ready for the following theorem

Theorem 2.3.3 *The WHILE loop in Step 3 of the algorithm SPLIT takes time* $O(\log n)$.

PROOF. First we study the complexity of constructing the 2-3 tree $T_1^{(i)}$ from the 2-3 tree $T_1^{(i-1)}$ and the trees in the segment s_i. According to Lemma 2.3.2, we have

$$h(T_1^{(i-1)}) \leq h(s_i)$$

Thus, if s_i is a single subtree t_i, then according the analysis of the time complexity of the algorithm SPLICE, the time of splicing $T_1^{(i-1)}$ and t_i is bounded by a constant times

$$h(s_i) - h(T_1^{(i-1)})$$

On the other hand, if s_i consists of two subtrees t_i' and t_i'', then the time for splicing $T_1^{(i-1)}$ and t_i' is again bounded by a constant times $h(s_i) - h(T_1^{(i-1)})$. Moreover, note that the height of the resulting tree T' from splicing $T_1^{(i-1)}$ and t_i' is either $h(s_i)$ or $h(s_i) + 1$, and that the height of the subtree t_i'' is $h(s_i)$. Thus, splicing T' and t_i'' takes only constant time. Therefore, in this case, the total time to construct $T_1^{(i)}$ from $T_1^{(i-1)}$ and s_i is bounded by a constant times

$$h(s_i) - h(T_1^{(i-1)}) + 1$$

Therefore, to construct the final 2-3 tree $T_1^{(q)}$, the total time of the WHILE loop in Step 3 is bounded by a constant times

$$\sum_{i=2}^{q} (h(s_i) - h(T_1^{(i-1)}) + 1)$$

By Lemma 2.3.2, we have $h(s_{i-1}) \leq h(T_1^{(i-1)})$. Thus, the time complexity of the WHILE loop in Step 3 is bounded by a constant times

$$\sum_{i=2}^{q} (h(s_i) - h(s_{i-1}) + 1)$$

which is equal to $h(s_q) - h(s_1) + q$. Since all quantities $h(s_q)$, $h(s_1)$, and q are bounded by $\log n$, we conclude that the WHILE loop in Step 3 takes time $O(\log n)$. $\square$

The same proof shows that the WHILE loop in Step 5 also takes time $O(\log n)$. In conclusion, the algorithm SPLIT takes time $O(\log n)$.

2.4 Geometric graphs in the plane

A graph $G = (V, E)$ is *planar* if it can be embedded in the plane without edge crossings.

A *planar embedding* of a planar graph $G = (V, E)$ is a mapping of each vertex in V to a point in the plane and each edge in E to a simple curve between the two images of extreme vertices of the edge, so that no two images of edges intersect except at their endpoints. The image of the mapping is called a *geometric graph* in the plane.

If all edges of a geometric graph G are straight-line segments in the plane, G is called a *planar straight-line graph*, or PSLG. A PSLG G determines in general a subdivision of the plane. Each region R of the subdivision, together with the edges of G that are on the boundary of R, forms a polygon in the plane.

Euler's formula

Let v, e and f denote the number of vertices, edges and regions (including the unbounded region) of a PSLG, respectively. The famous Euler's formula relates these parameters by

$$v - e + f = 2$$

if we have an additional property that each vertex has degree at least 3 then we can prove the following relations:

$$v \leq \frac{2}{3}e$$

$$e \leq 3f - 6$$

$$f \leq \frac{2}{3}e$$

$$v \leq 2f - 4$$

$$e \leq 3v - 6$$

$$f \leq 2v - 4$$

That is, we have

$$\Theta(v) = \Theta(e) = \Theta(f)$$

Therefore, for a planar graph, the number of vertices, the number of edges, and the number of regions are all linearly related.

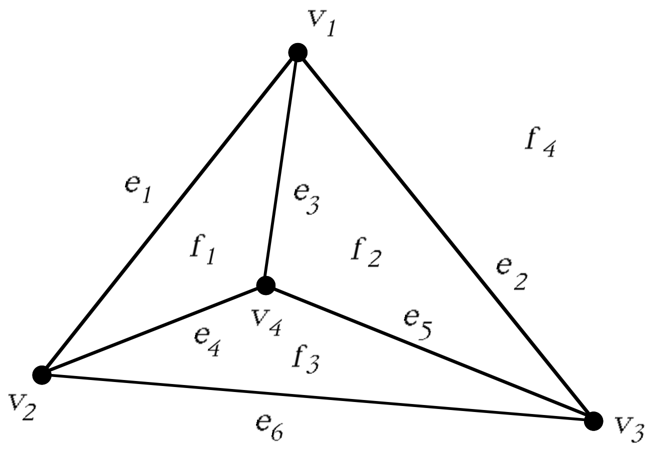

Figure 2.1: The planar imbedding of K_4

Doubly Connected Edge List (DCEL)

Given a planar imbedding I of the complete graph K_4, as depicted in Figure 2.1, what information should we keep for this imbedding? Of course, the set of vertices, and the set of edges of K_4 should be kept. Moreover, it is also necessary to keep the information about the regions of the imbedding I. To represent the information of the regions, we must know which edge will follow which edge when we travel around a vertex counterclockwise, i.e., we must know the cyclic ordering for the edges incident on each vertex of the imbedding I.

The *Doubly Connected Edge List (DCEL)* is an efficient data structure to represent a PSLG. The main component of DCEL for a PSLG G is the *edge node*. There is a one-to-one correspondence between the edges of G and edge nodes in the corresponding DCEL. An edge node consists of four information fields $V1, V2, F1$ and $F2$, and two pointer fields $P1$ and $P2$. The fields $V1$ and $V2$ contain the starting vertex and ending vertex of the edge, respectively. (So we give each edge of the PSLG G an orientation. This orientation can be defined arbitrarily.) The fields $F1$ and $F2$ contain the names of the regions which lie respectively to the left and right of the edge oriented from $V1$ to $V2$. The pointer $P1$ (or $P2$) points to the edge node containing the first edge encountered after the edge $(V1, V2)$ when one proceeds counterclockwise around $V1$ (or $V2$). Therefore, the edge $P1$ is the edge following the edge $(V1, V2)$ at the vertex $V1$, while the edge $P2$ is the edge following the edge $(V1, V2)$ at the vertex $V2$ in the imbedding $I(G)$.

The following is the DCEL for the PSLG, which is the complete graph K_4 in Figure 2.1.

	$V1$	$V2$	$F1$	$F2$	$P1$	$P2$
e_1	v_2	v_1	f_4	f_1	e_6	e_3
e_2	v_1	v_3	f_4	f_2	e_1	e_5
e_3	v_1	v_4	f_2	f_1	e_2	e_4
e_4	v_4	v_2	f_3	f_1	e_5	e_1
e_5	v_3	v_4	f_3	f_2	e_6	e_3
e_6	v_2	v_3	f_3	f_4	e_4	e_2

Note that the space used by a DCEL to represent a PSLG is linear to the number of edges of the PSLG.

Suppose that the set of vertices of a PSLG G is $\{v_1, \cdots, v_n\}$, and the set of regions of G is $\{f_1, \cdots, f_m\}$. We have another two arrays $HV[1..n]$ and $HF[1..m]$, where $HV[i]$ points to an edge node in the DCEL such that one edge end of the corresponding edge is v_i, for $i = 1, \cdots, n$, and $HF[j]$ points to an edge node on the DCEL such that the corresponding edge is on the boundary of the region f_j, for $j = 1, \cdots, m$.

Using DCEL of G we can travel the boundary of each region of G or the edges incident on a vertex of G. The following is an algorithm for traveling the boundary of a region when the DCEL of G is given. (The algorithm for traveling the edges incident on a vertex of G is given in [23].)

```
Algorithm  TRACE-REGION(i)
{ Trace the boundary edges of the region i. }

  BEGIN
  1.  a := HF[i];
  2.  a0 := a;
  3.  IF (DCEL[a][F1] = i) THEN
         a := DCEL[a][P1];
      ELSE  a := DCEL[a][P2];
  4.  WHILE  (a <> a0)  DO
        IF (DCEL[a][F1] = i) THEN
          a := DCEL[a][P1]
        ELSE  a := DCEL[a][P2];
  END.
```

For example, if we start with $HF[3] = 4$, and use the DCEL for the planar imbedding I of the complete graph K_4, then we will get the region f_3 as e_4, e_5, and e_6.

Note that if the rotation of edges incident on each vertex of the PSLG G is given in counterclockwise order in a DCEL, then the regions are traveled clockwise by the above algorithm. On the other hand, if the rotation of edges incident on each vertex of the PSLG G is given in clockwise order in a DCEL, then the regions are traveled counterclockwise by the above algorithm. Given a PSLG G, it is easy to see that a DCEL for G in which the rotation of edges incident on each vertex of G is given in counterclockwise order can be transformed in linear time into a DCEL for G in which the rotation of edges incident on each vertex of G is given in clockwise order, and vice versa. The detailed implementation of this transformation is straightforward and left to the reader as an exercise.

Chapter 3

Geometric Preliminaries

According to the nature of the geometric objects involved, we can identify basically five categories into which the entire collection of geometric problems can be conveniently classified, i.e., convexity, proximity, geometric searching, intersection, and optimization.

In this chapter, we will give the precise definitions of these problems and give an "intuitive" discussion on the mathematical background of them. Some of our statements and proofs are informal. This is because of the fact that some geometric theorems are "intuitively obvious" but no easy proofs are known though many great mathematicians have tried. An example is the following famous "Jordan Curve Theorem", which will actually serve as a fundamental basis for all of our discussions.

> **Jordan Curve Theorem** Let $\mathcal{C}$ be a simple closed curve in the plane, then the plane is subdivided into an interior region and an exterior region such that every curve connecting a point in the interior region and a point in the exterior region must intersect the curve $\mathcal{C}$.

The *k-dimensional Euclidean space* E^k is the space of all k-tuples $(x_1, \cdots, x_k)$ of real numbers x_i, $1 \leq i \leq k$. The *distance* between two points $p_1 = (x_1, \cdots, x_k)$ and $p_2 = (y_1, \cdots, y_k)$ in the k-dimensional space is defined by

$$d(p_1, p_2) = \left(\sum_{i=1}^{k} |y_i - x_i|^2\right)^{1/2}$$

The line passing through the points p_1 and p_2 can be parameterized by

$$\alpha p_1 + (1 - \alpha) p_2$$

where α ranges over the reals. If we restrict α to the interval $[0, 1]$, then we have a representation for a line segment, denoted $\overline{p_1 p_2}$, with the points p_1 and p_2 as its extreme points.

More generally, suppose $k+1$ independent points p_0, p_1, $\cdots$, p_k belong to a k-dimensional hyperplane. Then the hyperplane is parameterized by

$$\alpha_0 p_0 + \alpha_1 p_1 + \cdots + \alpha_k p_k$$

where $\sum_{i=0}^{k} \alpha_i = 1$. If we further restrict all $\alpha_i \geq 0$, then we have the representation for a simplex on $k+1$ points.

Given a triangle Δ with edges A, B and C, the angle θ between the two edges B and C can be obtained by the following formula:

$$\theta = \arccos \frac{|B|^2 + |C|^2 - |A|^2}{2 \cdot |B| \cdot |C|} \tag{3.1}$$

where $|A|$, $|B|$, and $|C|$ denote the lengths of the edges A, B, and C, respectively.

Suppose that Δ is a triangle in the plane E^2 with the vertices $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ and $p_3 = (x_3, y_3)$. Then the *signed area* of Δ is half of the determinant

$$D(p_1, p_2, p_3) = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \tag{3.2}$$

where the sign is positive if $(p_1 p_2 p_3)$ form a counterclockwise cycle, and negative if $(p_1 p_2 p_3)$ form a clockwise cycle. We say that the path from point p_1 through the line segment $\overline{p_1 p_2}$ to point p_2 then through the line segment $\overline{p_2 p_3}$ to point p_3 is a *left turn* if $D(p_1, p_2, p_3)$ is positive, otherwise, we say the path makes a *right turn*.

With the formulas (8.1) and (8.3), given three points p_1, p_2, and p_3 in the plane E^2, we can determine completely the value of the angle from the line segment $\overline{p_1 p_2}$ to the line segment $\overline{p_1 p_3}$ (denote this angle by $\angle p_2 p_1 p_3$).

A *line* L on the plane can be represented by a linear equation:

$$Ax + By + C = 0$$

such that a point $p = (x, y)$ is on the line if and only if the coordinates of p satisfy the equation. A *half plane* defined by the line L can be represented by either

$$Ax + By + C \geq 0$$

or

$$Ax + By + C \leq 0$$

3.1 Convex hulls

A subset $L \subseteq E^k$ is a *convex set* if for every pair p_1, p_2 of points in L, the line segment $\overline{p_1 p_2}$ is entirely in L.

Theorem 3.1.1 *The intersection of convex sets is convex.*

PROOF. Let S_i, $i = 1, 2, \cdots$, be convex sets. Denote by S the intersection of all these S_i's. We prove that S is again convex.

Let p_1 and p_2 be two points in S. Since S is the intersection of all S_i's, p_1 and p_2 are also points in each set S_i, $i = 1, 2, \cdots$. Since each S_i is convex, by definition, the entire line segment $\overline{p_1 p_2}$ is in S_i, for $i = 1, 2, \cdots$, thus in the intersection S of all these S_i's. $\square$

Definition Let $L \subseteq E^k$. The *convex hull CH(L)* of L is the smallest convex set containing L.

Given n points in the plane, we want to find their convex hull. This problem is as fundamental to computational geometry as sorting to general algorithms. It is also a vehicle for the solution of a number of apparently unrelated questions arising in computational geometry. The construction of the convex hull of a finite set of points has also found applications in many areas, such as in pattern recognition, in image processing, in Robotics, and in stock cutting and allocation.

Theorem 3.1.2 *Let $L \subseteq E^k$. The convex hull CH(L) of L equals the intersection of all convex sets containing L in E^k.*

PROOF. Let S be the intersection of all convex sets containing L in E^k. By theorem 3.1.1, S is convex, and obviously contains L. Now we prove that

S is the smallest such set. Let S' be an arbitrary convex set containing L. Then by the definition of S, S is the intersection of S' and other convex sets containing L, therefore, S is a subset of S'. That is, S is contained in every convex set containing L, so S is the smallest such set. $\square$

A *polygon* in E^k is a finite set of line segments satisfying the following two conditions:

1. every endpoint is shared by exactly 2 line segments; and

2. no proper subset has Property 1.

Now we study our problems in the plane E^2, i.e., the 2-dimensional Euclidean space.

Given a polygon P in the plane E^2, P is a *simple* polygon if there is no pair of nonconsecutive edges sharing a point. For any simple polygon in the plane, we can apply the Jordan Curve Theorem to divide the plane into the *interior* and the *exterior* of the simple polygon. For a simple polygon P, we will use P to refer to either the boundary of P, or the boundary plus the interior of P. The reader should not be confused from the contents. A polygon P is called a *convex polygon* if P is a simple polygon and the boundary plus the interior of P is a convex set in E^2.

Theorem 3.1.3 *The convex hull of a finite set S of points in E^2 is a simple polygon. Moreover, each hull vertex must be a point in the set S.*

PROOF. We give an informal, but intuitive proof here. A formal proof can be found in [17].

1. The convex hull *CH(S)* of the finite set S must be connected. Otherwise, let p_1 and p_2 be two points in two distinct connected components of *CH(S)*. Then the line segment $\overline{p_1 p_2}$ would not be entirely in *CH(S)*.

2. The convex hull *CH(S)* must be a bounded area. In fact, since S consists of finite number of points, we must be able to draw a circle C of a finite radius in the plane such that all points of S are inside C. The circle C is obviously convex. Now by definition, the convex hull *CH(S)* is contained in the circle C.

3. Let p_1 and p_2 be two points in S such that all points of S are on one side of the straight line through p_1 and p_2, then the line segment

$\overline{p_1p_2}$ is on the boundary of the convex hull *CH(S)*. First of all, the line segment $\overline{p_1p_2}$ must be contained in *CH(S)*. Moreover, since the half plane H_1 determined by the straight line through points p_1 and p_2 and containing all points of S is a convex set containing the set S, so the convex hull *CH(S)* is contained in H_1. Therefore, no point on the other side of the line segment $\overline{p_1p_2}$ can be in *CH(S)*. That is, $\overline{p_1p_2}$ is on the boundary of the convex hull *CH(S)*.

4. All points on the boundary of *CH(S)* must be on a line segment $\overline{p_1p_2}$, where p_1 and p_2 are points in the set S. Suppose that p is not such a point and p is on the boundary of *CH(S)*. If we "slightly" move the part of the boundary of *CH(S)* near the point p so that the resulting area is properly contained in *CH(S)*, is still convex, and contains all points in the set S, then we get a convex set that contains all points of the set S, and is "smaller" than *CH(S)*, contradicting the definition of convex hulls.

Therefore, the boundary of the convex hull *CH(S)* must consist of a finite set G of line segments of which the end-points are points in the set S. Suppose that a line segment $\overline{p_1p_2}$ is on the boundary of *CH(S)*. Without loss of generality, we can suppose that the points of the set S are on our left when we travel along the straight line L through p_1 and p_2 in the direction from p_1 to p_2. Now if we rotate the line L counterclockwise around the point p_2, the line L will eventually hit a first point p_3 of the set S. It is obvious to see that now the line segment $\overline{p_2p_3}$ is also on the boundary of *CH(S)*. Moreover, there is no other point p in S that can make the line segment $\overline{p_2p}$ on the boundary of *CH(S)* if we assume that no three points in the set S are co-linear (the proof can be modified properly for the general case), since the points p_1 and p_3 must lie on different sides of the straight line through $\overline{p_2p}$. Now based on the new line and the hull vertex p_3, we can find the next hull vertex, etc.. This process must be stopped eventually since there are only finite number of points in the set S. Therefore, we will eventually hit a point p in the set S that has been decided earlier to be a hull vertex. The point p must be the point p_1 since all other hull vertices found have already had the two line segments incident on them, which are on the boundary of *CH(S)*. Therefore, we have enclosed the points of the set S by a closed simple cycle, which is a simple polygon $P = \{p_1, p_2, \cdots, p_k\}$. No point p in the interior of P can be on the boundary of *CH(S)* since any straight line through the point p will intersect with a boundary edge of the polygon P, thus have points in the set S on both of its sides. Therefore, the simple polygon P is the convex hull *CH(S)*. □

3.2 Proximity problems

The examples of proximity problems include CLOSEST-PAIR, ALL-NEAREST-NEIGHBORS, EUCLIDEAN-MINIMUM-SPANNING-TREE, TRIANGULATION, and MAXIMUM-EMPTY-CIRCLE.

Proximity problems arise in many applications where physical or mathematical objects are represented as points in space. Examples include the following:

- *clustering:* a number of entities are grouped together if they are sufficiently close to one another;
- *classification:* a new pattern to be classified is assigned to the class of its closest (classified) neighbor; and
- *air-traffic control:* the two airplanes that are closest are the two most in danger.

We will restrict ourselves to 2-dimensions. The input to these problems is a set S of n points in the plane. The distance between points in S will be the Euclidean distance between the points.

- **CLOSEST-PAIR**

  Find a pair of points in the set S which are closest.

- **ALL-NEAREST-NEIGHBORS**

  For every point in the set S, find a point that is nearest to it.

- **EUCLIDEAN-MINIMUM-SPANNING-TREE**

  Find an interconnecting tree of minimum total length whose vertices are the points in the set S.

- **TRIANGULATION**

  Join the points in the set S by non-intersecting straight line segments so that every region interior to the convex hull of S is a triangle.

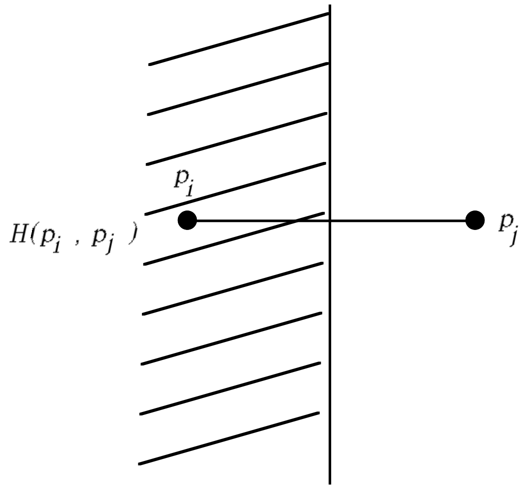

Figure 3.1: The points that are closer to p_i than to p_j

- **MAXIMUM-EMPTY-CIRCLE**

  Find a largest circle containing no points of the set S yet whose center is interior to the convex hull of S.

The problems posed above are related in the sense that they all deal with the respective distances among points in the plane. In the following, we will introduce a single geometric structure, called the *Voronoi diagram*, which contains all of the relevant proximity information in only linear space.

Let us get some motivation from the CLOSEST-PAIR problem. Let S be a set of n points in the plane. For any two points p_i and p_j in S, the set of points closer to p_i than to p_j is just the half-plane containing p_i that is defined by the perpendicular bisector of the segment $\overline{p_i p_j}$. See Figure 3.1. Denote this half-plane by $H(p_i, p_j)$ (note that $H(p_i, p_j) \neq H(p_j, p_i)$). Therefore, the set V_i of points in the plane that are closer to the point p_i than to any other points in the set S is the intersection of the sets $H(p_i, p_j)$ for all $p_j \in S - \{p_i\}$

$$V_i = \bigcap_{j \neq i} H(p_i, p_j)$$

Each $H(p_i, p_j)$ is a half-plane so it is convex. By Theorem 3.1.1, the set V_i, which is the intersection of these convex sets $H(p_i, p_j)$, is also convex. It is also easy to see that the set V_i is in fact a convex polygonal region. Observe that every point in the plane must belong to some region V_i. Moreover, no set V_i can be empty since all points in a small enough disc centered at the point p_i must be in V_i.

Thus these n convex polygonal regions V_1, V_2, $\cdots$, V_n partition the plane into a convex net. Motivated by this discussion, we introduce the following definition.

Definition A *Voronoi diagram* of a set $S = \{p_1, \cdots, p_n\}$ of n planar points is a partition of the plane into n regions V_1, V_2, $\cdots$, V_n such that any point in the region V_i is closer to the point p_i than to any other point in the set S.

The convex polygonal region V_i is called the *Voronoi polygon* of the point p_i in S. The vertices of the diagram are called *Voronoi vertices* and the line segments of the diagram are called *Voronoi edges.* The Voronoi diagram of a set S is denoted by *Vor(S)*. Note that Voronoi vertices are in general *not* the points in the set S.

3.3 Intersections

Intersection problems and their variations arise in many disciplines, such as architectural design, computer graphics, pattern recognition, etc. An architectural design cannot place two interpenetrable objects to share a common region. When displaying objects on a 2-dimensional display device, obscured portions (or intersecting portions) should be eliminated to enhance realism, a long standing problem known as hidden line/surface elimination problem [19]. In integrated circuit design two distinct components must be separated by a certain distance, and the detection of whether or not the separation rule is obeyed can be cast as an instance of intersection problems; since the task may involve thousands of objects, fast algorithms for detecting or reporting intersecting or overlapping objects are needed. Another motivation for studying the complexity of intersection algorithms is that light may be shed on the inherent complexity of fundamental geometric problems. For example, how difficult is it to decide if a given polygon with n vertices is simple or how much time is needed to determine if any two of n given objects in the plane, such as polygons, line segments, etc., intersect?

We list a few typical geometric intersection problems.

- **SEGMENT INTERSECTION**

Given n line segments in the plane, find all intersections.

- **HALF-PLANE INTERSECTION**

  Given n half-planes in the plane, compute their common intersection.

- **POLYGON INTERSECTION**

  Given two polygons P and Q with m and n vertices, respectively, computer their intersection.

3.4 Geometric searching

This geometric problem is well motivated by the following *Post Office Problem* proposed by Knuth [14]: Given a fixed map of n post offices, for an arbitrary query point, which is the nearest post office? The solution to this problem is simple: compare the distance between the query point and each post office and find the nearest one. The time complexity of this algorithm is obviously $O(n)$. It is also easy to see that to find the nearest post office, at least n comparisons are needed, since if the algorithm does not compute the distance between the query point and some post office, then we are always able to construct an input instance such that the query point is closest to the uncompared post office so that the algorithm outputs an incorrect answer. Therefore, for a single query point, the above simple algorithm is actually optimal.

On the other hand, suppose that we have, say, n query points and we are asking the nearest post office for each query point. If we again apply the above algorithm, then it takes time $O(n)$ to find the nearest post office for each query point, so totally we need time $O(n^2)$ to find the nearest post offices for all query points. Now it seems that the time $O(n^2)$ is not necessary. For example, after we have computed the distance between the first query point and each of the post office and found the nearest post office for the first query point, it seems that we can save some information about the post offices and use this information to speed up the computation of nearest post office for the latter query points. Even more cleverly, we can first organize the post offices into an easy-search structure such that searching the nearest post office for each query point can be done very efficiently on the organized structure.

One candidate of these smart structures is the Voronoi diagram, introduced in Section 2.2. Given n post offices, regarding them as a set S of n

points in the plane, we first construct the Voronoi diagram *Vor(S)* for the set S. Then finding the nearest post office for a query point is reduced to locating the query point in a Voronoi polygon of *Vor(S)*.

This is a typical geometric searching problem, called *point location problem*: Suppose that we have a subdivision G of the plane and we want to know in which region of G a given query point is located. In the simplest case, we have only one query point. Then we can search the point in each region of G directly to find the region containing the point. A one-time query of this type is called *single shot*. However, we may have many query points and want to find the containing region for each query point. Such queries are called *repetitive-mode queries*.

In the case of repetitive-mode queries, it may be worthwhile to arrange the subdivision G into a more organized structure to facilitate searching. Therefore, when we are considering the problem of repetitive-mode queries, we are interested in three computational resources: the *preprocessing time* that is used to convert the given subdivision G into an organized structure, the *storage* that is used to store the organized structure, and the *query time* that is needed to locate each query point.

Suppose that the input subdivision G has n vertices. In general, we cannot expect that the preprocessing time is less than $O(n)$ since even reading the input subdivision G takes time $\Omega(n)$. Similarly, we cannot expect that the storage used for the organized structure is less than $O(n)$ since even storing the unorganized structure, the subdivision G itself needs $\Omega(n)$ space. Finally, as pointed out by Knuth [14], any algorithm for searching an ordered table of length n by means of comparisons can be represented as a binary tree of n leaves, thus in the worst case, the searching time is at least $\Omega(\log n)$. While the point location problem is clearly a generalization of searching, we conclude that the query time of the point location problem is at least $\Omega(\log n)$.

Chapter 4

Geometric Sweeping

Geometric sweeping technique is a generalization of a technique called *plane sweeping*, that is primarily used for 2-dimensional problems. In most cases, we will illustrate the technique for 2-dimensional cases. The generalization to higher dimensions is straightforward. This technique is also known as the *scan-line method* in computer graphics, and is used for a variety of applications, such as shading, polygon filling, among others.

The technique is intuitively simple. Suppose that we have a line in the plane. To collect the geometric information we are interested in, we slide the line in some way so that the whole plane will be "scanned" by the line. While the line is sweeping the plane, we stop at some points and update our recording. We continue this process until all interesting objects are collected.

There are two basic structures associated with this technique. One is for the *sweeping line status*, which is an appropriate description of the relevant information of the geometric objects at the sweeping line, and the other is for the *event points*, which are the places we should stop and update our recording. Note that the structures may be implemented in different data structures under various situations. In general, the data structures should support efficient operations that are necessary for updating the structures while the line is sweeping the plane.

4.1 Intersection of line segments

The geometric sweeping technique can be best illustrated by the following example. Recall the SEGMENT INTERSECTION problem:

Given n line segments in the plane, find all intersections.

Suppose that we have a vertical line L. We sweep the plane from left to right. At every moment, the *sweeping line status* contains all segments intersecting the line L, sorted by the y-coordinates of their intersecting points with L. The sweeping line status is modified whenever one of the following three cases occurs:

1. The line L hits the left-end of a segment S. In this case, the segment S was not seen before and it may have intersections with other segments on the right side of the line L, so the segment S should be added to the sweeping line status;

2. The line L hits the right-end of a segment S. In this case, the segment S cannot have any intersections with other segments on the right side of the line L, so the segment S can be deleted from the sweeping line status;

3. The line L hits an intersection of two segments S_1 and S_2. In this case, the relative positions of the segments S_1 and S_2 in the sweeping line status should be swapped, since the segments in the sweeping line status are sorted by the y-coordinates of their intersection points with the line L.

It is easy to see that the sweeping line status of the line L will not be changed when it moves from left to right unless it hits either an endpoint of a segment or an intersection of two segments. Therefore, the set of *event points* consists of the endpoints of the given segments and the intersection points of the segments. We sort the event points by their x-coordinates.

We use two data structures $EVENT$ and $STATUS$ to store the event points and the sweeping line status, respectively, such that the set operations MINIMUM, INSERT, and DELETE can be performed efficiently (for example, they can be 2-3 trees). At very beginning, we suppose that the line L is far enough to the left so that no segments intersect L. At this moment, the sweeping line status is an empty set. We sort all endpoints of the segments by their x-coordinates and store them in $EVENT$. These are the event points at which the line L should stop and update the sweeping line status. However, the list is not complete since an intersection point of two segments should also be an event point. Unfortunately, these points are unknown to we at beginning. For this, we update the structure $EVENT$ in the following way. Whenever we find an intersection point of two segments while the line L is sweeping the plane, we add the intersection point to $EVENT$. But how do we find these intersection points? Note that if the next event point to be hit by the sweeping line L is an intersection point of two segments S_1 and S_2, then the segments S_1 and S_2 should be adjacent in the sweeping line status. Therefore, whenever we change the adjacency relation in *STATUS*, we check for intersection points for new adjacent segments. When the line L reaches the left-most endpoint of the segments, all possible intersection points are collected.

These ideas are summarized by the following algorithm.

```
Algorithm  SEGMENT-INTERSECTION

  Given:  n segments S1, S2, ... Sn
  Output: all intersections of these segments

  { Implicitly, we use a vertical line L to sweep the plane.  At any
    moment, the segments intersecting L are stored in STATUS, sorted
    by the y-coordinates of their intersection points with the line
    L.  The event points stored in EVENT are sorted by their x-coor-
    dinates }

  BEGIN
  1.  Sort the endpoints of the segments and put them in EVENT;
  2.  STATUS = {};
  3.  WHILE  EVENT is not empty  DO BEGIN
        p = MINIMUM(EVENT);
        DELETE p from EVENT;
        IF p is a right-end of some segment S
          Let Si and Sj be the two segments adjacent to S in STATUS;
          IF p is an intersection point of S with Si or Sj
            REPORT(p);
          DELETE S from STATUS;
          IF Si and Sj intersect at p1 and x(p1) >= x(p)
            INSERT p1 into EVENT
        ELSE IF p is a left-end of some segment S
          INSERT S into STATUS;
          Let Si and Sj be the adjacent segments of S in STATUS;
          IF p is an intersection point of S with Si or Sj
            REPORT(p);
          IF S intersects Si at p1, INSERT p1 into EVENT;
          IF S intersects Sj at p2, INSERT p2 into EVENT
        ELSE IF p is an intersection point of segments Si and Sj
        such that Si is on the left of Sj in STATUS
```

```
        REPORT(p);
        swap the positions of Si and Sj in STATUS;
        Let Sk be the segment left to Sj and let Sh be the segment
          right to Si in STATUS;
        IF Sk and Sj intersect at p1 and x(p1) > x(p)
          INSERT p1 into EVENT;
        IF Sh and Si intersect at p2 and x(p2) > x(p)
          INSERT p2 into EVENT;
    END; {WHILE}
END.
```

Let us analyze the algorithm. Step 1, sorting the $2n$ endpoints of the segments, can be done in time $O(n \log n)$, if we employ an efficient sorting algorithm, for example, the MergeSort. Step 2 takes constant time $O(1)$. To count the time spent by the WHILE loop, suppose there are m intersection points for these n segments. In the WHILE loop, each segment is inserted then deleted from the structure $STATUS$ exactly once, and each event point is inserted then deleted from the structure $EVENT$ exactly once. There are $n+m$ event points. If we suppose that the operations MINIMUM, INSERT, and DELETE can all be done in time $O(\log N)$ on a set of N elements, then processing each segment takes at most $O(\log n)$ time, and processing each event point takes at most $O(\log(n+m))$ time. Therefore, the algorithm runs in time

$$\begin{aligned} & O(n \log n) + O(1) + n \times O(\log n) + (n+m) \times O(\log(n+m)) \\ = \; & O((n+m)\log(n+m)) \end{aligned}$$

Observe that m is at most n^2, so $\log(n+m) = O(\log n)$. Thus we conclude that the algorithm SEGMENT-INTERSECTION runs in time $O((n+m)\log n)$.

We remark that the time complexity of the above algorithm depends on the number m of intersection points of the segment and the algorithm is not always efficient. For example, when the number m is of order $\Omega(n^2)$, then the algorithm runs in time $O(n^2 \log n)$, which is even worse than the straightforward method that picks every pair of segments and computes their intersection point. On the other hand, if the number m is of order $\Omega(n)$, then the algorithm runs efficiently in time $O(n \log n)$.

4.2 Constructing convex hulls

4.2.1 Jarvis's March

We start with a most straightforward method, Jarvis's March, which is also known as *gift wrapping method.*

The idea is based on the observation we gave in the proof of Theorem 3.1.3. Given a set S of n points in the plane, suppose we move a straight line L sweeping the plane until L hits a point p_1 of S. The point p_1 must be on the boundary of the convex hull *CH(S)* of S since at this moment, all points of S are in one side of the line L and the point p_1 is on the line L. Now we rotate the line L around the point p_1, say counterclockwise, until L hits another point p_2 of S. The segment $\overline{p_1p_2}$ is then on the boundary of the convex hull *CH(S)* since again all points of S are in one side of the line L and the segment $\overline{p_1p_2}$ is on the line L. Now we rotate the line L around p_2 counterclockwise until L hits a third point p_3 of S, then the line segment $\overline{p_2p_3}$ is the second boundary edge of *CH(S)*, ... Continue this process until we come back to the first point p_1. The convex hull *CH(S)* then is constructed.

This process can also be regarded as a "wrapping" process. Suppose we fix an end of a rope on a point p_1 that is known to be a hull vertex. Then we try to "wind" the points by the rope (or "wrap" the points by the rope). The rope obviously gives us the boundary of the convex hull when it comes back to the point p_1.

There are a few things we should mention in the above process. First of all, the sweeping manner is special: the line L is rotated around a point in the plane; secondly, the sweeping line status is very simple: it contains at any moment a single point that is the hull vertex most recently discovered; finally, the even points are the hull vertices.

Let us study the above process in detail. Suppose at some moment in the middle of the process, the consecutive hull vertices which have been found are $p_1, p_2, \cdots, p_i$. What point should be the next hull vertex? Obviously, the point p_{i+1} first touched by the rope should be it, when we rotate the rope around the point p_i. That is, the angle $\angle p_{i-1}p_ip_{i+1}$ should be the largest.

We implement this idea into the following algorithm.

```
Algorithm   JARVIS'S MARCH

   Given:    a set S of n points in the plane
```

```
Output:   the convex hull CH(S) of S

BEGIN
  Let p(1) be the point in the set S that has the smallest
     y-coordinate;
  Let p(2) be the point in the set S such that the slope of
     the line segment p(1)-p(2) is the smallest, with respect
     to the x-axis;
  PRINT(p(1), p(2));
  i := 2 ;
  WHILE  p(i) <> p(1)  DO
     Let p(i+1) be the point in the set S such that the angle
        <p(i-1)p(i)p(i+1) is the largest;
     i := i + 1 ;
     PRINT(p(i));
END.
```

Time complexity of Jarvis's March

Suppose there are k hull vertices in $CH(S)$. The points p_1 and p_2 are obviously hull vertices. Moreover, it is also clear that to find the points p_1 and p_2 takes time $O(n)$, assuming S has n points. To find each next hull vertex p_{i+1}, we check the angle $\angle p_{i-1}p_ip$ for each point p in the set S. Thus Step 4 spends time $O(n)$ on each hull vertex. Therefore, Jarvis's March runs in time $O(kn)$.

If k is small compared with n, for instance, if k is bounded by a constant, then Jarvis's March runs in linear time. However, if k is larger, such as $k = \Omega(n)$, then the time complexity of Jarvis's March is $\Omega(n^2)$.

4.2.2 Graham Scan

Look at Jarvis's algorithm. Each time based on the most recent hull vertex p and the most recent hull edge e, we find the next hull vertex by choosing the point p' which makes the angle between e and $\overline{pp'}$ largest. To find such a point p', we have to compute the angle between the segments e and $\overline{pq}$ for all points q in the set S. For each hull vertex, we have to perform this kind of computations. Therefore, in this process, even though we have found out that a point p is not qualified for the next hull vertex, we still cannot exclude the possibility that the point p is qualified for a later hull vertex. This is the reason that we have to consider the point again and again. A point can be considered up to n times in the worst case. A possible improvement is that we presort the set of points in some way so that once we find that a point is not qualified for the next hull vertex, then we can exclude the point forever. For example, let p_0, p_1 and p_2 be three distinct hull vertices of the convex hull *CH(S)* for the set S. Suppose that the line segment $\overline{p_1p_2}$ is known to be on the boundary of the convex hull *CH(S)*. Then the line segments $\overline{p_0p}$ for all points p of S that are between the angle $\angle p_1p_0p_2$ should be entirely in the triangle $\Delta p_0p_1p_2$. Therefore, if we start with the point p_1, scan the points of the set S, based on the point p_0, counterclockwise, and keep a record for the length of the line segment $\overline{p_0p}$ for each point of S we have visited, then once we reach the point p_2, we can eliminate all points we have visited between the points p_1 and p_2. This elimination is permanent, i.e., once a point is eliminated, it will be ignored forever.

This idea is implemented by the following well-known algorithm, known as Graham Scan algorithm.

```
Algorithm   GRAHAM SCAN

    Given:  a set S of n points in the plane
    Output: the convex hull CH(S) of S

    {St is a stack}

    BEGIN
    1.  Let p(0) be the point in S that has the smallest y-coordinate.
        { Without loss of generality, we can suppose that p(0) is the
          origin, otherwise, we make a coordinate transformation }
    2.  Sort the points in the set S - p(0) by their polar angles.
        Let the sorted list of the points be
           L' = { p(1), p(2), ..., p(n-1) }
          {in increasing polar angle ordering.}
    3.  Let
           L = { p(1), p(2), ..., p(n-1), p(n) }
          where  p(n) = p(0);
        q(1) = p(0);  q(2) = p(1);   PUSH(St, q(1));
        PUSH(St, q(2));   i = 2;   j = 2;
    4.  WHILE  i <=  n  DO
           IF  q(j-1)q(j)p(i)  is a left turn
           THEN   q(j+1) = p(i);
                  PUSH(St, q(j+1));
                  i++;
                  j++
```

```
        ELSE   POP(St);
               j--;
    END.
```

In Graham Scan, the sweeping line rotates around a fixed point p_0. All points in the set S are event points. Since the event points are presorted in Step 2, it takes only constant time to find the next event point in the sorted list L. This makes Graham Scan very efficient.

Let us consider the time complexity of the algorithm in detail. Step 1 can be done by comparing the y-coordinates of all points in the set S, thus it takes time $O(n)$; Step 2 can be done by any $O(n \log n)$ time sorting algorithm, for example, MergeSort; Step 3 obviously takes constant time. To discuss the time complexity of the loop in Step 4, observe that each point of the set S can be pushed into the stack St and then popped out of the stack at most once. Whenever a point is popped out from the stack, it will never be considered any more. Therefore, there are at most $2n$ stack pushes and pops. Now each execution of the loop in Step 4 either pushes a point into the stack (Step 4.2) or pops a point out the stack (Step 4.3). Thus the loop is executed at most $2n$ times. Since each execution of the loop obviously takes constant time, we conclude that the total time taken by Step 4 is bounded by $O(n)$.

Therefore, the time complexity of Graham Scan is $O(n \log n)$.

We remark that most of the time in Graham Scan algorithm is spent on Step 2's sorting. Besides sorting, Graham Scan runs in linear time.

The Step 2 in Graham Scan sorts the points in the given set S by their polar angles. This involves in trigonometric operations. Although we have assumed that our RAMs can perform trigonometric operations in constant time, trigonometric operations can be very time consuming in a real computer. We present a modified version of Graham Scan which avoids using trigonometric operations.

The idea is as follows. Suppose we are given a set S of n points in the plane. We add a new point p_0 to the set S such that p_0's y-coordinate is smaller than that of any point in the set S. Then we perform Graham Scan on this new set. Draw a line segment $\overline{p_0 p}$ for each point p in the set S. It can be easily seen that if the point p_0 moves toward the negative direction of the y-axis, these line segments are getting more and more parallel each other. Imagining that eventually p_0 reaches the infinite point along the negative direction of the y-axis, then all these line segments become vertical rays originating from the points of the set S. Now the ordering of the polar angles of the points of S around p_0 is identical with the ordering of the x-coordinates of these points. (In fact, p_0 does not have to be the infinite point, when p_0 is far enough from the set S, the above statement should be true.) Therefore, the convex hull of the new set can be constructed by first sorting the points in S by their x-coordinates instead of their polar angles. It is also easy to see that the convex hull of the new set consists of two vertical rays, originating from the two points $p_{\min}$ and $p_{\max}$ in the set S with smallest and largest x-coordinates, respectively, and the part UH of the convex hull of the original set S. This part UH of the convex hull *CH(S)* is in fact the *upper hull* of *CH(S)* in the sense that all points of the set S lie between the vertical lines $x = x_{\min}$ and $x = x_{max}$ and below the part UH. Similarly, the *lower hull* of the convex hull *CH(S)* can be constructed by the idea of adding an infinite point in the positive direction of the y-axis. The convex hull *CH(S)* is simply the circular catenation of the upper hull and the lower hull.

Now we give the formal algorithm as follows.

```
Algorithm   MODIFIED GRAHAM SCAN

    Given:   a set S of n points in the plane
    Output:  the convex hull CH(S) of S;

    BEGIN
       Sort the points of the set S in decreasing x-coordinate
           ordering;
       Let pmax and pmin be the points of S that have the
           largest and smallest x-coordinates, respectively.
       Suppose  pmax = (x, y), let  p(0) = (x, y-1),
           and  p(1) = pmax;
       Perform Graham Scan on the sorted list until the point
           pmin is included as a hull vertex;
       The ordered list of hull vertices found in this process
           minus the point p(0) is the upper hull;
       Construct the lower hull similarly;
       Catenate the upper hull and lower hull to form the convex
           hull  CH(S).
    END
```

The Modified Graham Scan obviously also takes time $O(n \log n)$.

4.3 The farthest pair problem

The problem we shall discuss in this section is formally defined as follows:

FARTHEST-PAIR

Find a pair of points in a given set which are farthest.

A brute force algorithm is to examine every pair of points to find the maximum distance thus determined. The brute force algorithm obviously runs in time $O(n^2)$.

To get a more efficient algorithm, let us first investigate what kind of properties a farthest pair of points in a set has. Let us suppose that S is a set of n points in the plane, and call a segment linking two farthest points in the set S a *diameter* of the set S.

Lemma 4.3.1 *Let $\overline{uv}$ be a diameter of the set S. Let l_u and l_v be two straight lines that are perpendicular to the segment $\overline{uv}$ such that l_u passes through u and l_v passes through v. Then all points of S are contained in the slab between l_u and l_v.*

PROOF. Without loss of generality, suppose that the segment $\overline{uv}$ is horizontal and the point u is on the left of the point v. Draw a circle C centered at u of radius $|\overline{uv}|$, then the line l_v is tangent to C because l_v is perpendicular to $\overline{uv}$. Thus the circle C is entirely on the left of the line l_v. Since v is the farthest point in the set S from the point u, all points of S are contained in the circle C. Consequently, all points of S are on the left of the line l_v. Similarly, we can prove that all points of S are on the right of the line l_u. Therefore, all points of the set S are between the lines l_u and l_v. $\square$

Corollary 4.3.2 *Let $\overline{uv}$ be a diameter of the set S, then the points u and v are hull vertices of CH(S).*

PROOF. As we discussed in Chapter 2, a point p in S is a hull vertex of *CH(S)* if and only if there is a line passing through p such that all points of S are on one side of the line. $\square$

Let u and v be two hull vertices of *CH(S)*. The vertices u and v are called an *antipodal pair* if we can draw two parallel supporting lines l_u and l_v of *CH(S)* such that l_u passes through u and l_v passes through v, and the convex hull *CH(S)* is entirely contained in the slab between the lines l_u and l_v.

Corollary 4.3.3 *Let $\overline{uv}$ be a diameter of the set S, then u and v are an antipodal pair.*

PROOF. By Corollary 4.3.2, u and v are hull vertices of *CH(S)*. By Lemma 4.3.1, we can draw two parallel lines l_u and l_v such that l_u passes through u, that l_v passes through v, and that all points of S are contained in the slab between l_u and l_v. The slab between l_u and l_v is clearly a convex set. Since the convex hull *CH(S)* of S is the smallest convex set containing all points of S, i.e., the convex hull *CH(S)* is contained in all convex sets containing all points of S, so the convex hull *CH(S)* is contained in the slab between the lines l_u and l_v. □

According to Lemma 4.3.1 and its corollaries, to find a farthest pair of a set S of n points in the plane, we only need to find a farthest pair of the hull vertices of the convex hull *CH(S)*. Moreover, we only need to consider the antipodal pairs on the convex hull *CH(S)*. This greatly simplifies our problem. We now consider the following problem: given a vertex u of a convex polygon P, what vertices of P can constitute an antipodal pair with the vertex u? To answer this question, we suppose that the vertices of the convex polygon P are given in counterclockwise ordering: $\{u_1, u_2, \cdots, u_m\}$. For simplicity, we say that a vertex u_i of P is the *farthest* from an edge $\overline{u_{k-1}u_k}$ of P if u_i is the farthest vertex in P from the straight line on which $\overline{u_{k-1}u_k}$ lies.

Lemma 4.3.4 *Let $\overline{u_{k-1}u_k}$ be an edge of P. We scan the vertices of P in counterclockwise order, starting with the vertex u_k. Let u_i be the first farthest vertex from the edge $\overline{u_{k-1}u_k}$. Then no vertex between u_k and u_i can constitute an antipodal pair with u_k.*

PROOF. Without loss of generality, suppose that the edge $\overline{u_{k-1}u_k}$ is horizontal and the vertex u_k is on the right of the vertex u_{k-1}. First note that for any vertex u_i of P, the angle between the edge $\overline{u_iu_{i+1}}$ and the x-axis is between 0 and 2π. Let α be the angle between the edge $\overline{u_ku_{k+1}}$ and the x-axis. Suppose that α_1 (α_2) is the angle between the edge $\overline{u_{i-1}u_i}$ ($\overline{u_iu_{i+1}}$) and the x-axis. Since P is convex, $\alpha_1 \leq \alpha_2$. See Figure 4.1 for illustration. It is easy to see that the vertex u_i constitutes an antipodal pair with the vertex u_k if and only if the angle region $[\alpha_1, \alpha_2]$ contains an angle between π and $\pi + \alpha$. Let u_j be a vertex between u_k and u_i, ($u_j \neq u_k, u_i$). Then u_j is not farthest from the edge $\overline{u_{k-1}u_k}$. Thus the angle between the edge $\overline{u_ju_{j+1}}$

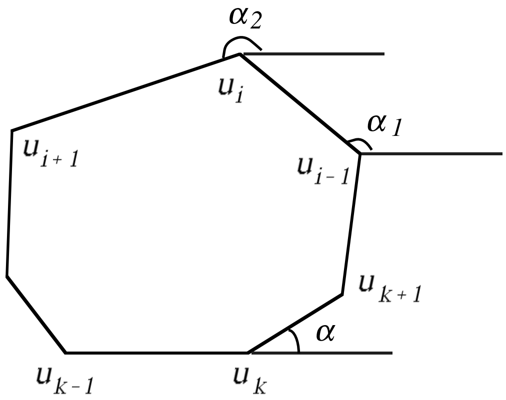

Figure 4.1: The convex polygon P

and the x-axis, and the angle between the edge $\overline{u_{j-1}u_j}$ and the x-axis are all strictly less than π. That is, the vertex u_j does not constitute an antipodal pair with u_k. □

Lemma 4.3.5 *Let $\overline{u_{k-1}u_k}$ be an edge of P. We scan the vertices of P in counterclockwise order, starting with the vertex u_k. Let u_r be the last farthest vertex from the edge $\overline{u_{k-1}u_k}$. Then no vertex between u_r and u_{k-1} (in counterclockwise ordering on the boundary of P) can constitute an antipodal pair with u_{k-1}.*

PROOF. Completely similar to the proof of Lemma 4.3.4. □

Now it is clear how we find all antipodal pairs on the convex polygon P: starting with an edge $\overline{u_{k-1}u_k}$, we scan the vertices of P counterclockwise until we hit the first farthest vertex u_i from the edge $\overline{u_{k-1}u_k}$. By Lemma 4.3.4, u_i is the first vertex of P that constitutes an antipodal pair with the vertex u_k. Now we continue scanning the vertices until we hit a vertex u_r that is the last farthest vertex to the edge $\overline{u_k u_{k+1}}$. By Lemma 4.3.5, u_r is the last vertex that constitutes an antipodal pair with the vertex u_k. Now a vertex constitutes an antipodal pair with u_k if and only if it is between u_i and u_r. Moreover, since we suppose that no three vertices of P are co-linear, there are at most two farthest vertices from an edge on P. The algorithm of finding all antipodal pairs of a convex polygon P is given in detail as follows.

```
Algorithm   ANTIPODAL-PAIRS

    Given:   a convex polygon  P = { u(1), ...., u(m) } in
             counterclockwise ordering
    Output:  all antipodal pairs of P

    BEGIN
    1.    Starting with the edge  {u(0), u(1)}, where we let
             u(0) be the vertex u(m).  Set k = 1 and i = 2.
    2.    WHILE  u(i) is not a farthest vertex from the edge
             {u(k-1), u(k)}
             i = i + 1;
    3.    { At this point u(i) is a farthest vertex from the
             edge {u(k-1), u(k)}. }
          WHILE  u(i) is not a farthest vertex from the edge
             {u(k), u(k+1)}
             OUTPUT  [u(k), u(i)] as an antipodal pair;
             i = i + 1;
    4.    { At this point u(i) is the first farthest vertex
             from the edge {u(k), u(k+1)}.  We check if u(i)
             is the last farthest vertex from the edge
             {u(k), u(k+1)}.  }
          IF  u(i+1)  is also a farthest vertex from the edge
             {u(k), u(k+1)}
             OUTPUT  [u(k), u(i)], [u(k+1), u(i)]  as
               antipodal pairs;
             i = i + 1;
    5.    { Now u(i) must be the last vertex that can consti-
            tute an antipodal pair with u(k).  }
          OUTPUT   [u(k), u(i)] as an antipodal pair;
    6.    IF  k < m, THEN
            k = k + 1;
            GOTO  Step 3;
    END.
```

The addition $i = i + 1$ in the algorithm should be "(mod m)", that is, if $i = m$, then $i + 1 = 1$. Note that the distance from a vertex u_i to the line on which the edge $\overline{u_{k-1}u_k}$ lies is proportional to the area of the triangle $\triangle(u_i u_{k-1} u_k)$, therefore the vertex u_i is the farthest from the edge $\overline{u_{k-1}u_k}$ if and only if the area of the triangle $\triangle(u_i u_{k-1} u_k)$ is less than neither the area of the triangle $\triangle(u_{i-1} u_{k-1} u_k)$ nor the area of the triangle $\triangle(u_{i+1} u_{k-1} u_k)$.

An intuitive description of the above algorithm is that we use two parallel lines to sandwich the convex polygon P, then rotate the lines along the boundary of P, keeping the lines in parallel. We report all pairs of vertices of P that are at some moment on the two parallel lines at the same time, respectively, when we rotate the lines.

The analysis of the algorithm is straightforward. We keep two pointers k and i. In constant time, at least one pointer is advanced. Since the pointer k is from 1 to m and the pointer i marches the convex polygon P at most twice (the pointer i stops at the last farthest vertex from the edge $\overline{u_m u_1}$), we conclude that the time complexity of the algorithm is bounded by $O(m)$.

A further improvement can be made in the above algorithm if we observe that when the pointer i reaches the vertex u_m, then all antipodal pairs have actually been found. In fact, if the pointer i is advanced from the vertex u_m to the vertex u_1, then we are considering the vertex u_1 as a candidate that constitutes an antipodal pair with some other vertex of P. However, all vertices that constitute antipodal pairs with u_1 have been found when the pointer k is advanced from the vertex u_1 to the vertex u_2. Since this improvement does not change the asymptotical order of the time complexity of the algorithm, we will not discuss it in detail.

Now we give the algorithm for the FARTHEST-PAIR problem.

```
Algorithm    FARTHEST-PAIR

    Given:   a set S of n points in the plane
    Output:  the farthest pair

    BEGIN
    1.  Construct the convex hull CH(S) of S;
    2.  Call ANTIPODAL-PAIRS on CH(S);
    3.  Scan the result of Step 2 and select the pair
        with the longest distance.
    END.
```

By the discussions given in this section, the above algorithm finds the farthest pair for a given set S correctly. Moreover, the algorithm runs in time $O(n \log n)$ since it is dominated by the first step.

4.4 Triangulations

TRIANGULATING a set S of n points in the plane is to joint the points in the set S by non-intersecting straight line segments so that every region interior to the convex hull of S is a triangle. In this section we shall discuss a more general version of TRIANGULATION: given a set S of n points in the plane and a set E of non-intersecting straight line segments whose endpoints are the points in S, construct a triangulation $T(S)$ of S such that all the segments in the set E appear in the triangulation $T(S)$.

Recall that a *planar straight line graph (PSLG)* $G = (S, E)$ is a finite set S of points in the plane plus a set E of non-intersecting straight line segments whose endpoints are the points in the set S. We always suppose that a PSLG G is represented by a doubly-connected edge list (DCEL).

The problem we shall discuss is called *Constrained Triangulation.*

CONSTRAINED TRIANGULATION

Given a PSLG $G = (S, E)$, construct a triangulation $T(S)$ of S such that all segments of E are edges of $T(S)$.

4.4.1 Triangulating a monotone polygon

We first discuss the problem for a special class of PSLG's, called *monotone polygon.*

A *chain* $C = (v_1, v_2, \cdots, v_r)$ is a PSLG with a point set $S = \{v_1, v_2, \cdots, v_r\}$ and a segment set $E = \{(v_i, v_{i+1}) \mid 1 \leq i \leq n-1\}$. A chain C is *monotone* with respect to a straight line l if any straight line orthogonal to l intersects the chain C at at most one point.

Definition A polygon P is said to be *monotone* with respect to a straight line l if P is a simple polygon and the boundary of P can be decomposed into two chains monotone with respect to the straight line l.

If a polygon P is monotone with respect to the y-axis, we simply say that the polygon P is *monotone.*

We first solve the following problem: given a monotone polygon P, triangulate the interior of P. That is, we add edges to the polygon P so that each region in the interior of P is a triangle.

A vertex u of a polygon P is *visible* from a vertex v if we can draw a straight line segment s connecting u and v such that the interior of the segment s is entirely in the interior of the polygon P. In particular, a vertex is not visible from any of its adjacent neighbors. Moreover, note that a vertex v is visible from a vertex u if and only if the vertex u is visible from the vertex v.

The method we are going to use is a "greedy" method. Standing at each vertex v of the polygon P, we look through the interior of the polygon P and see which vertex of the polygon P is visible. Whenever we find that a vertex u of the polygon P is visible from the vertex v, we add an edge between the vertices v and u. Keeping doing this until no vertex of P is visible from the vertex v, then we move to another vertex v' of P and add edges to those vertices that are visible from v', an so on. Note that once there is no vertex visible from a vertex v of P, then no vertex can become a visible vertex from v later, since the only operation we are performing is adding edges to the interior of the polygon P. Therefore, once we add edges to a vertex v of P so that there is no vertex of P visible from v, we do not have to come back and check the vertex v again. Moreover, if the interior of the polygon P is not triangulated, then there must be a pair of vertices v and u between which we can add a new edge e without edge-crossing. But this implies that the vertex u is still visible from the vertex v before we add the new edge e. Thus, if we process all vertices of P such that from any vertex v of P there is no visible vertex, then we must have triangulated the interior of the polygon P.

The above method is principally valid for triangulating any PSLG. However, to find all visible vertices from a vertex of a general PSLG may be time-consuming. On the other hand, if the PSLG is a monotone polygon, then the process above can be done very efficiently.

The following is the algorithm of triangulating a monotone polygon P. We process the vertices, in the way described above, in the ordering of decreasing y-coordinate. A stack *STACK* is used to store those vertices of P that have been processed such that no processed vertices are still visible from a vertex in the *STACK* and each vertex in the *STACK* is still visible from some unprocessed vertices of P.

```
Algorithm    TRIANGULATING-MONOTONE-POLYGON

    Given:   a monotone polygon P
    Output:  a triangulation of P
```

```
BEGIN
1.   Sort the vertices of P in decreasing y-coordinate,
     Let the sorted list be
             L = { v(1), v(2), ...., v(n) }
2.   Push the vertices v(1) and v(2) into the stack
     STACK.  Let i = 3.
3.   Suppose that the vertices in the STACK are
            STACK = { u(1), u(2), ...., u(s) }
     where u(s) is the top and u(1) is the bottom.
4.   IF  v(i) is adjacent to u(1) but not to u(s)
     { we will prove later that in this case, stack
       vertices u(2), u(3), ...., u(s) are all visible
       from v(i). }  THEN
       add edges {v(i), u(2)}, {v(i), u(3)}, ....,
         {v(i), u(s)}, pop all STACK vertices, then
         push u(s) and v(i) into the STACK;
       i++;
       GOTO  Step 7;
5.   IF  v(i) is adjacent to u(s) but not to u(1)
     { in this case, u(s) is not visible from v(i), we
       check if any other STACK vertices are visible
       from v(i). }  THEN
        WHILE  the second top vertex of the STACK
               (call it u') is visible from v(i)  DO
          add an edge {v(i), u'};
          pop the top vertex from STACK;
        PUSH  v(i) into STACK;
        i++;
        GOTO Step 7;
6.   IF  v(i) is adjacent to both u(s) and u(1)
     { in this case, v(i) is the last vertex in the
       list L, and all STACK vertices except u(s) and
       u(1) are visible from v(i).  }   THEN
       add edges  {v(i), u(2)}, {v(i), u(3)}, ......,
          {v(i), u(s-1)};
       POP all STACK vertices and STOP.
7.    IF   i <= n, go back to Step 3.
END.
```

We first discuss the correctness of the algorithm. Each execution of the loop Step 3 - Step 6 results in a PSLG. Let the PSLG after processing the vertex v_i be G_i. (So $G_0 = P$ and G_n should be a triangulation of P.)

We prove that the following properties are always maintained for all G_i's. Suppose the *STACK* content is $\{u_1, u_2, \cdots, u_s\}$.

Properties of G_i

1. The *STACK* contains at least two vertices for G_0, G_1, $\cdots$, G_{n-1}.

2. The *STACK* vertices $\{u_1, \cdots, u_s\}$ is a monotone chain on the boundary of some region P_i of G_i that is a monotone polygon.

3. The processed vertices that are not in the *STACK* are not visible from any vertex of G_i.

4. No *STACK* vertex is visible from any other *STACK* vertex in G_i.

If for each G_i, the above properties are maintained, then since for G_n, all vertices of P are processed and the *STACK* is empty, by Property 3, no vertex of G_n is visible from any other vertex of G_n. As we discussed earlier in this section, the PSLG G_n must be a triangulation.

We prove by induction that the above four properties are always maintained by every PSLG G_i. For G_0, the properties are trivially maintained because of Step 1 and Step 2. Now suppose that the properties are also maintained for the PSLG G_{i-1}. To obtain G_i, we execute Step 3 - Step 6 based on G_{i-1}.

Property 1 is obviously maintained, since if $i < n$ then either Step 4 or Step 5 is executed. But both of them leave at least two vertices in the *STACK*.

To maintain Property 2, note that by inductive hypothesis, all processed vertices that are not in *STACK* for G_{i-1} are not visible from any vertex of G_{i-1}, that is, all the regions incident to those vertices must be triangles. Thus the edges to be added in Step 4 or Step 5 must be within the monotone polygon P_i. Moreover, the vertex v_i is the only new vertex added to the *STACK* and the y-coordinate of v_i is less than that of any *STACK* vertex. Finally, the vertex v_i is always connected to the top vertex in *STACK* before v_i is pushed into *STACK*. These observations make sure that Property 2 is also maintained for G_i.

Now let us consider Property 3. For those processed vertices that are not in *STACK* for G_{i-1}, they are not visible from any vertices of G_{i-1}, thus they are also not visible from any vertices of G_i since G_i is obtained by adding edges to G_{i-1}. Suppose that u_r is a vertex that is in the *STACK* for G_{i-1} but popped out by Step 4, by Step 5, or by Step 6.

If u_r is popped by Step 4, then $r < s$. The vertex u_r is visible from neither a vertex in *STACK* nor a processed vertex that is not in *STACK*, by the inductive hypothesis. Moreover, the edge $\overline{v_i u_{r+1}}$ blocks u_r from being visible from any unprocessed vertex. The same proof applies to the case that u_r is popped by Step 6.

If u_r is popped by Step 5, then at some moment in the "While" loop of Step 5, u_r is the top vertex of *STACK*. Let u' be the second top vertex of the *STACK*. The vertex u_r is popped because the edge $\overline{v_i u'}$ is added. Since v_i has a smaller y-coordinate than u_r, the edge $\overline{v_i u'}$ blocks u_r from being visible from any unprocessed vertex.

Therefore, if u_r is popped from the *STACK* for G_{i-1} when we are constructing G_i, then u_r is no longer visible from any vertex of G_i.

Finally, consider Property 4. If Step 4 is executed, the *STACK* contains two adjacent vertices u_s and v_i, so Property 4 is obviously maintained. If Step 5 is executed, then we add an edge between the second top vertex u' of *STACK* and v_i when u' is visible from v_i. We keep doing this until the second top vertex u' of *STACK* is no longer visible from v_i. At this point, no other *STACK* vertex could be visible from v_i since otherwise, let u'' be the first vertex in *STACK* that is visible from v_i, then it is easy to see that u'' should also be visible from the first top vertex of *STACK*, contradicting our inductive hypothesis. This proves that Property 4 can always be maintained.

By the above discussion, it can also be realized that if Step 4 is the case, then all *STACK* vertices u_2, u_3, $\cdots$, u_s are visible from the vertex v_i. In fact, if u_2 is not visible from v_i, then the edge $\overline{v_i u_2}$ must intersects some edge of G_{i-1}. Since vertices v_i, u_1, u_2, $\cdots$, u_s are consecutive vertices on the boundary of P_{i-1} (remember that in this case we suppose that v_i is adjacent to u_1), if $\overline{v_i u_2}$ intersects some edges of G_{i-1}, then $\overline{v_i u_2}$ must also intersect the chain $C = \{u_1, u_2, \cdots, u_s\}$ on the boundary of P_{i-1}. But this implies that some vertex on the chain C is visible from the vertex u_1, contradicting our inductive hypothesis. Similarly, we can prove that after adding edges $\overline{v_i u_2}$, $\overline{v_i u_3}$, $\cdots$, $\overline{v_i u_{r-1}}$, the vertex u_r is still visible from the vertex v_i, for $r = 3, \cdots, s$.

This completes the discussion of the correctness of the algorithm.

The analysis of the algorithm is easier. Since the polygon P is monotone, there are two vertices v_0 and v_r of P with the largest and the smallest y-coordinates, respectively. Moreover, the boundary of the polygon P can be decomposed into two monotone chains

$$C = (u_0, u_1, \cdots, u_k) \quad \text{and} \quad C' = (u'_0, u'_1, \cdots, u'_h)$$

where $u_0 = u'_0 = v_0$ and $u_k = u'_h = v_r$ and the vertices in both chains C and C' are in decreasing y-coordinate ordering. We can merge the two chains C and C' in linear time to obtain the list L of vertices of the polygon P sorted by decreasing y-coordinate. Therefore, Step 1 of the algorithm takes linear time.

Within the loop of Step 4 - Step 7, we add each new edge in constant time. Since the final triangulation G_n is a planar graph that has at most $O(n)$ edges, so the total time for adding new edges is bounded by $O(n)$. Finally, since each vertex of P is pushed into then popped out the stack *STACK* exactly once, the total time is again bounded by $O(n)$.

We close this subsection with the conclusion that the problem of triangulating a monotone polygon can be solved in linear time.

4.4.2 Triangulating a general PSLG

Now we consider the problem of triangulating a general PSLG. Given a general PSLG G of n points, let

$$F = \{P_1, P_2, \cdots, P_r\}$$

be the set of regions of G. If each region of G is a monotone polygon, we can use the following method to triangulate G: use the TRACE-REGION algorithm in Section 1.4 to find all regions

$$P_1, P_2, \cdots, P_r$$

Let $\#P_i$ be the number of edges of the polygon P_i, which is also the number of vertices of P_i. Then the region P_i can be constructed in time $O(\#P_i)$. Therefore, to find all regions of G takes time

$$O(\#P_1) + O(\#P_2) + \cdots + O(\#P_r) = O(\#P_1 + \#P_2 + \cdots + \#P_r)$$

Since each edge of G is used by exactly two regions of G in their boundary, $(\#P_1 + \#P_2 + \cdots + \#P_r)$ is twice the number of edges of G, which is bounded by $O(n)$ since G is a planar graph. That is, the regions of G can be constructed in linear time. Now we triangulate each region P_i of G using the algorithm *TRIANGULATING-MONOTONE-POLYGON* given in the last subsection. The time for triangulating the monotone polygon P_i is bounded by $O(\#P_i)$. Therefore, triangulating all regions of G takes linear time. It is easy to see that putting all these triangulated regions together to get a triangulation of G can also be done in linear time. We conclude that if all regions of a PSLG G are monotone polygons then the triangulation of G can be done in linear time.

Therefore, the problem of triangulating a general PSLG G is reduced to the problem of converting the PSLG G into a PSLG G' such that all regions of G' are monotone polygons. Without loss of generality, we suppose that our PSLG G has no two points with the same y-coordinate (otherwise we can achieve this by rotating the coordinate system slightly). Let us first introduce some definitions.

Let G be a PSLG and let v be a point of G. An edge $\{u, v\}$ is an *upper edge* of v if the y-coordinate of u is larger than that of v, and an edge $\{w, v\}$ is a *lower edge* of v if the y-coordinate of w is smaller than that of v. A vertex v of G is *regular* if either v is the vertex of G with maximum or minimum y-coordinate or v has both upper edges and lower edges.

Definition Let G be a PSLG. G is a *regular* PSLG if every vertex of G is a regular vertex.

Note that if G is a regular PSLG, then G must be connected. In fact, suppose that G is not connected, let v_0 and v_0' be the vertices of maximum y-coordinate of two different connected components of G, respectively. Then both v_0 and v_0' have no upper edges, so one of them must be an unregular vertex of G.

Lemma 4.4.1 *If G is a regular PSLG, then all regions of G are monotone polygons.*

PROOF. Suppose that G is a regular PSLG but a region P of G is not a monotone polygon. Let v_0 be the vertex of P that has the largest y-coordinate. Since P is a simple polygon and no vertex of P has the same y-coordinate as v_0, when a horizontal straight line l is close enough to the vertex v_0, l intersects P at exactly two points. Because P is not monotone, there must be some horizontal lines intersecting P at more than two points. Let

$$r_0 = \sup\{r \mid \text{the line } y = r \text{ intersects } P \text{ at more than two points.}\}$$

Let l_0 be the horizontal straight line $y = r_0$. There are two possible cases.

The line l_0 intersects P at two points. Then since a slight moving down of the line l_0 would make the line intersect more than two points, there must be a vertex v of P on the line l_0 such that the vertex v has two lower edges.

Since l_0 intersects P at two points, v is not v_0. However, v has no upper edges since each vertex of P is incident to exactly two edges of P, so v is not a regular vertex and G is not a regular PSLG.

On the other hand, suppose that l_0 intersects P at more than two points, then a slight moving up of the line l_0 would make the line intersect exactly two points. Thus one of those intersecting points of l_0 and P must be a vertex of P without upper edges. But this again contradicts the assumption that G is regular.

Therefore, the region P must be a monotone polygon. $\square$

Therefore, the TRIANGULATION problem for regular PSLGs can be done in linear time. In the next subsection, we will show that given a general PSLG G, in time $O(n \log n)$ we can convert G into a regular PSLG by adding edges to G. Consequently, the problem CONSTRAINED-TRIANGULATION can be solved in time $O(n \log n)$.

Remark:

Chazelle [8] has recently proven that triangulating a simple polygon (not necessarily a monotone polygon) can be done in linear time. Since for a connected PSLG G, the regions of G can be constructed in linear time, and each region is a simple polygon, we use Chazelle's linear time algorithm to triangulate each region of G then put them together. This gives us a linear time algorithm for triangulating a connected PSLG.

4.4.3 Regularization of PSLGs

We thereby have the following problem.

REGULARIZATION-PSLG

Given a general PSLG G, add edges to G so that the resulting PSLG is regular.

Intuitively, to regularize a PSLG, we add an upper edge to a vertex if it does not have an upper edge, and add a lower edge to a vertex if it does not have a lower edge. The problem is, how do we add the edges so that edge-crossing is avoided. Therefore, when we are working on a vertex of a PSLG G, we should have enough information about the local environment of the vertex. But how do we maintain and update the information about the local environment efficiently when we move from one vertex to another vertex?

Again, the plane sweeping technique helps. Let $V = \{v_1, v_2, \cdots, v_n\}$ be the vertex set of a PSLG G. Without loss of generality, suppose that no two vertices in V have the same y-coordinate.[1] We first sort the vertices in V by their y-coordinate. Then we sweep the plane by a horizontal line from bottom up. The sweeping stops at each vertex of G and check if the vertex has an upper edge. If the vertex does not have an upper edge we add one for it. Then we sweep the plane once again from top down to add lower edges for those vertices without lower edges. After these two sweepings, every vertex has at least one upper edge (except for the vertex with the maximum y-coordinate) and at least one lower edge (except for the vertex with the minimum y-coordinate). Thus the PSLG becomes regular. We discuss the bottom-up sweeping in detail. The top-down sweeping can be handled similarly.

Without loss of generality, suppose that the list $\{v_1, v_2, \cdots, v_n\}$ is the sorted list of vertices of the PSLG G in ascending y-coordinate. Consider the sweeping line l at vertex v_i, where $i < n$. The sweeping line l partitions the PSLG G into three parts G_1, G_2 and G_3. G_1 is the "past history" containing those vertices of G that are below the line l and have at least one upper edge each, and those edges of G that are entirely below the line l. G_2 is the "current status" containing the vertices of G that are either on the line l or below the line l and have no upper edges, and those edges of G that intersect the line l. G_3 is the "unknown future" containing the vertices and edges of G that are entirely above the line l. The elements in G_1 are "nice" elements that we have seen and we know that they do not make trouble for us. The elements in G_2 are "current" elements that we should process. The elements in G_3 are unknown elements to us since we have not seen them during the bottom-up sweeping. Therefore, the process of the plane sweeping is a process of updating the current status G_2 when we pass through a vertex v of the PSLG G. This is easy to see that during the sweeping between two consecutive vertices in the list $\{v_1, v_2, \cdots, v_n\}$, the current status G_2 is invariant. The current status G_2 only changes when we pass through a vertex of the PSLG G. This is the reason why our sweeping is discrete (i.e., the sweeping only stops at the vertices of G and updates the current status).

We require that between two intersecting edges of G in G_2 that are

[1] In fact, with a minor modification, our algorithms will also work for the general case.

consecutive on the line l, there is at most one "hanged vertex", i.e., a vertex that is below the line l and has no upper edges. This condition can be easily maintained since when a second hanged vertex is added between the edges, we can connect it to the first hanged vertex by a new edge thus give the first hanged vertex an upper edge and unhang it.

The current status G_2 can be maintained in the following way when we are passing through a vertex v_i: we first search the nearest left edge e_1 and the nearest right edge and e_r of v_i in G_2. Let e_2, $\cdots$, e_{r-1} be the lower edges incident on v_i in counterclockwise ordering. We then check if there is a hanged vertex v_h between a pair (e_j, e_{j+1}) of edges, for $j = 1, \cdots, r-1$. If there is one then we add a new edge between v_i and v_h and unhang the vertex v_h. Then we delete the lower edges of v_i from G_2 and add the upper edges of v_i to G_2. If v_i has no upper edges, then we hang v_i between the two nearest edges e_1 and e_r in G_2. Sweeping all vertices from v_1 to v_n and updating G_2 and G dynamically, we will finally finish adding upper edges to the vertices of G. It is easy to see that after this process, each vertex of G, except v_n, has at least one upper edge.

Therefore, the following operations should be done efficiently on G_2 by our algorithm: finding the edges e_1, e_2, $\cdots$, e_r in G_2 such that e_1 and e_r are the nearest left and the nearest right edges of the vertex v_i in G_2, respectively, and e_2, $\cdots$, e_{r-1} are the lower edges incident on v_i; deleting an edge from G_2; and adding an edge to G_2.

Note that if we lower the sweeping line l a little bit, the intersecting points of the line l and the edges e_1, e_2, $\cdots$, e_r are consecutive on l. Therefore, if we put the edges in G_2 in a list in the ordering of their intersections with the line l, then the edges e_1, e_2, $\cdots$, e_r correspond to a consecutive sublist of the list.

A proper data structure for efficiently implementing the above operations is a 2-3 tree T. The edges in G_2 are ordered from left to right according to the ordering of their intersections with the line l. Hanged vertices in G_2 are hanged between consecutive leaves in the tree T.

The following algorithm is based on the above discussion.

```
Algorithm    ADD-UPPER-EDGES

    Given:   a PSLG G of n vertices, represented by a DCEL
    Output:  a PSLG G', obtained by adding edges to G such
               that each vertex of G' (except the highest
               one) has at least one upper edge.
```

```
BEGIN
1.  Sort the vertices of G in increasing y-coordinates,
    let { v(1), ...., v(n) } be the sorted vertex list;
2.  Create an empty 2-3 tree T; insert the upper edges
    of v(1) into T if they exist , otherwise hang v(1);
3.  FOR  i = 2 up to n  DO
3a.     Using the x-coordinate of the vertex v(i) to
        find two edges e(1) and e(r) in T that are the
        nearest left and the nearest right edges of
        v(i) in T.  All the edges e(2), ...., e(r-1)
        that are between e(1) and e(r) in the tree T
        are lower edges of v(i).
3b.     For j = 1 to r-1
           IF   there is a hanged vertex v(h) between
                e(j) and e(j+1)  THEN
              add a new edge {v(h), v(i)};
              unhang v(h);
3c.     Delete the lower edges e(2), ...., e(r-1) of
        v(i) from T if they exist;
3d.     IF  v(i) has upper edges  THEN
           insert the upper edges of v(i) into T
        ELSE
           hang v(i) between the nearest left and
           right edges e(1) and e(r) if i <> n.
END.
```

We give the analysis of the algorithm. Step 1 can be done in time $(n \log n)$ by any optimal sorting algorithm. Since each leaf of T corresponds to an edge in G and G is a planar graph, T contains at most $O(n)$ leaves. Consequently, the depth of the tree T is at most $O(\log n)$. Thus Step 3a can be done in time $O(\log n)$ for each vertex of G. Each vertex of G can be hanged and unhanged at most once so the total time used to hang and unhang vertices of G is bounded by $O(n)$. Finally, each edge of G is inserted exactly once (at its lower endpoint) then deleted exactly once (at its upper endpoint) in the tree T, thus the time spent on inserting and deleting a single edge of G is bounded by $O(\log n)$. Summarizing all these discussions, we conclude that the algorithm ADD-UPPER-EDGES has time complexity $O(n \log n)$.

Chapter 5

Divide and Conquer

Divide and Conquer is a classical problem solving technique and has proven its value for geometric problems as well. This technique normally involves partitioning of the original problem into several subproblems, recursively solving each subproblem, and then combining the solutions to the subproblems to obtain the solution to the original problem. A general form of a divide and conquer algorithm is as follows:

```
Algorithm    DIVIDE AND CONQUER

    Given:    A problem P of size n
    Output:   A solution to P

    BEGIN
    0.  IF  n = 1  THEN
           Solve the problem P directly and STOP;
    1.  Divide the problem P into k subproblems of size n/k;
    2.  Recursively solve each subproblem;
    3.  Combine the solutions to the subproblems to obtain
        a solution to the problem P;
    END.
```

The "size" of the problem P is a reasonable measure of the quantity of input data. For example, if the problem is to construct the convex hull of a set S of points in the plane, then the size of the problem can be the number of points in the given set S.

To make the algorithm efficient, we in general expect that Step 1 of dividing into subproblems and Step 3 of combining subsolutions can be done in linear time.

Now we analyze the algorithm. Suppose that the time complexity of the algorithm is $T(n)$ on problems of size n. We assume reasonably that when the problem has size 1, the problem can be solved in constant time, i.e., $T(1) = b$, where b is a constant. By our assumption, Step 1 and Step 3 can be done in time cn, where c is again a constant. Recursively, each subproblem of size n/k can be solved in time $T(n/k)$. So to solve all subproblems, Step 2 takes time $kT(n/k)$. Therefore, the time complexity of the algorithm DIVIDE AND CONQUER can be expressed by the following recurrence

$$\begin{aligned} T(1) &= b \\ T(n) &= kT(n/k) + cn \end{aligned}$$

It is an easy exercise to obtain the closed form for the function $T(n)$, as stated by the following theorem.

Theorem 5.0.2 *If Step 1 and Step 2 can be done in linear time, then the algorithm DIVIDE AND CONQUER runs in time*

$$T(n) = O(n \log n)$$

5.1 Convex hulls again

In this section, we present two divide and conquer algorithms for constructing convex hulls for sets of points in the plane, *MERGEHULL* and *QUICKHULL*, which are the analogues of the famous sorting algorithms *MERGESORT* and *QUICKSORT*, respectively.

The idea of *MERGEHULL* is exactly like that of *MERGESORT*. Given a set S of n points in the plane, we first split S into two subsets S_1 and S_2 of roughly equal size, then we separately construct the convex hulls $CH(S_1)$ and $CH(S_2)$ for each of the sets S_1 and S_2. Finally, we merge the two hulls into a larger hull for the original set S.

Some details are worth to discuss. To make our merge process easier, we would like to let the two hulls $CH(S_1)$ and $CH(S_2)$ disjoint. This can be done by letting the subset S_1 be the half of S with smaller x-coordinates, while letting the subset S_2 be the half with larger x-coordinates. There are two different ways to split the set into such two sets. One is to use a linear time algorithm, developed by Blum, Floyd, Pratt, Rivest, and Tarjan [5], to find the point p with the median x-coordinate, then spit S into S_1 and S_2 according to p. Another way is to presort the set S by x-coordinates, then for a sorted list, the median can always be found in linear time. We will adopt the second approach.

The following is the detailed *MERGEHULL* algorithm.

```
Algorithm    MERGEHULL

    Given:   a set S of n points in the plane
    Output:  the convex hull of S

    BEGIN
    1.   Sort S by x-coordinates;
    2.   Call MHULL(S)
    END.
```

The subroutine *MHULL(S)* is as follows.

```
Algorithm    MHULL(S)

    Given:    a set S of n points in the plane, sorted by
              x-coordinate
    Output:   the convex hull of S

    BEGIN
    1.   IF S contains less than four points, construct the
         convex hull CH(S) directly.  Otherwise, do the
         following.
    2.   Split S into two subsets S_1 and S_2 of roughly
         equal size, such that the x-coordinate of any point
         in S_1 is less than the x-coordinate of any point
         in S_2;
    3.   Recursively call MHULL(S_1) and MHULL(S_2) to
         construct the convex hulls CH(S_1) and CH(S_2);
    4.   MERGE(CH(S_1), CH(S_2)) to obtain CH(S).
    END.
```

All that is left to specify is how to perform the subroutine *MERGE(CH(S_1), CH(S_2))*. For this, we must find two lines: one that is tangent to the top of both $CH(S_1)$ and $CH(S_2)$ (the *upper bridge*) and one that is tangent to the bottom of both hulls (the *lower bridge*). Let $u(S_1)$ and $l(S_1)$ be the vertices in set S_1 that are on the upper and lower bridges, respectively (similarly define $u(S_2)$ and $l(S_2)$). Then all vertices in $CH(S_1)$ proceeding clockwise from $u(S_1)$ to $l(S_1)$ can be discarded. Similarly, all vertices in $CH(S_2)$ proceeding counterclockwise from $u(S_2)$ to $l(S_2)$ can be discarded. All the remaining vertices form the convex hull $CH(S)$.

Now we find the upper bridge (lower bridge is a symmetric operation). Let us assume that the convex hulls of $CH(S_1)$ and $CH(S_2)$ are each stored as a doubly-linked list. In constant time, we can add a point , delete a point, or find the clockwise or counterclockwise neighbor of a point. Suppose we had a guess for the endpoints of the upper bridge. How can we verify the guess? Suppose we guess that some line l through $p \in CH(S_1)$ is tangent to the hull $CH(S_1)$ at point p. Let p' and p'' be the two neighbors of the point p in the hull $CH(S_1)$. The line l is tangent to the top of $CH(S_1)$ at the point p if and only if both points p' and p'' are on or below the line l.

Therefore, to construct the upper bridge, we can pick any hull vertex p from $CH(S_1)$ and any hull vertex q from $CH(S_2)$ and let l be the line through p and q. Now we try to "lift" the line l as much as possible with the condition that l intersects both hulls $CH(S_1)$ and $CH(S_2)$. Once we cannot lift the line l anymore, the line l must be tangent to the top of both $CH(S_1)$ and $CH(S_2)$, i.e., l is the upper bridge of $CH(S_1)$ and $CH(S_2)$. Note that if the two neighbors p' and p'' of the point p are on the two sides of the line l, we can always use "signed triangle area" to decide which neighbor is above the line l.

We give the detailed algorithm as follows.

```
Algorithm    UpperBridge(CH(S_1), CH(S_2))

     Given:    two convex hulls CH(S_1) and CH(S_2) that
               are separated by a vertical line such that
               CH(S_1) is on the left of the line
     Output:   the upper bridge of CH(S_1) and CH(S_2)

     BEGIN
     1.    Let p be the point in CH(S_1) with the smallest
           x-coordinate, and let q be the point in CH(S_2)
```

```
      with the largest x-coordinate.  Let L be the
      line through p and q;
2.    WHILE  L is not the upper bridge  DO
2.1.     WHILE  there is a neighbor p' of p in CH(S_1)
         above the line L, replace the point p by the
         point p' and construct the new line L;
2.2.     WHILE  there is a neighbor q' of q in CH(S_2)
         above the line L, replace the point q by the
         point q' and construct the new line L;
END.
```

The lower bridge of $CH(S_1)$ and $CH(S_2)$ can be found by an algorithm which is identical with the algorithm *UpperBridge* except that the word "above" is replaced by the word "below".

In the worst case, the line l in the algorithm *UpperBridge* passes through every hull vertices of $CH(S_1)$ and every hull vertices of $CH(S_2)$. Then the algorithm must stop. Therefore, the running time of the algorithm *UpperBridge* is at most linear to the number of hull vertices of the two hulls, which is bounded by the number of points in the set $S_1 \cup S_2$. To merge two hulls which are separated by a vertical line, it suffices to find the upper and lower bridges, delete a partial hull from each of the hulls, and catenate the remaining parts of the hulls with the upper and lower bridges properly. All these can be obviously done in time $O(n)$ if the set $S_1 \cup S_2$ contains n points and the convex hulls are stored as doubly-linked lists. We conclude that the time complexity of the subroutine *MERGE* is $O(n)$.

Now we analyze the time complexity of the algorithm *MERGEHULL*. If an $O(n \log n)$ time sorting algorithm is used, then the time of the algorithm *MERGEHULL* equals $O(n \log n)$ plus the time of the algorithm *MHULL*.

Since the set S is presorted by x-coordinates, Step 2 of the algorithm *MHULL(S)*, i.e., splitting S into S_1 and S_2 can be done in time $O(n)$. By the analysis above, Step 4 of the algorithm can also be done in time $O(n)$. Moreover, since the set S is presorted by x-coordinates, when we pass the sets S_1 and S_2 to the recursive calls *MHULL*(S_1) and *MHULL*(S_2), the sets S_1 and S_2 are also presorted by x-coordinates. Thus the subroutine *MHULL* directly applies. Now by the discussion at the beginning of this chapter, we conclude that the time complexity of the algorithm *MHULL* is $O(n \log n)$.

As in *MERGESORT*, *MERGEHULL* splits the given input S carefully (into two equal size subsets), then merges carefully the two hulls which are obtained by recursive calls. The algorithm is in some sense very "stable".

That is, the time complexity is almost invariant for all inputs. On the other hand, *QUICKSORT* randomly splits the given list of numbers, recursively calls the subroutine, then simply catenates the two sorted sublists. Therefore, much less work is done besides the two recursive calls. The worst case time complexity of *QUICKSORT* is bad. However, for most inputs (or many inputs), *QUICKSORT* runs even fast than *MERGESORT*.

This discussion motivates the derivation of the following *QUICKHULL* algorithm, which is analogous to *QUICKSORT*, and has a bad worst case time complexity and, in general a good "average time complexity".

```
Algorithm    QUICKHULL(S)

      Given:   a set S of n points in the plane
      Output:  the convex hull of S

      BEGIN
      1.    Find the points p_min and p_max in S, with
            the smallest and largest x-coordinates,
            respectively;
      2.    Let S' be the subset of points in S that are
            above the line L through p_min and p_max,
            and let S'' be the set of points in S that
            are below the line L;
      3.    Call UpperHULL(S', p_min, p_max) and
                 LowerHULL(S'', p_min, p_max);
      4.    Catenate the upper and lower hulls.
      END.
```

Where the subroutines *UpperHULL* and *LowerHULL* are similar. We only give the *UpperHULL* as follows.

```
Algorithm    UpperHULL(S, l, r)

      Given:    a set S of points in the plane such that
                all points in S are above the line L
                through the points l and r.
      Output:   Find the the convex hull for S + {l, r}

      BEGIN
      1.   Find a point p in S that is furthest to the
```

```
     line L;
2.   Let S_1 be the subset of S that contains all
     the points above the line through l and p,
     and let S_2 be the subset of S that contains
     all the points above the line through p and r;
3.   Recursively call  UpperHULL(S_1, l, p) and
     UpperHULL(S_2, p, r);
4.   Catenate the two parts obtained in Step 3;
END.
```

Similar to *QUICKSORT*, it can be proved that in the worst case, the time complexity of the algorithm *QUICKHULL* is $\Omega(n^2)$. While with a reasonable assumption on the probability distribution of the points in the set S, the running time of the algorithm *QUICKHULL* is $O(n \log n)$. We leave the discussions to the interested reader.

5.2 The Voronoi diagram

We first recall the definition of a Voronoi diagram.

Definition A *Voronoi diagram* of a set $S = \{p_1, \cdots, p_n\}$ of n points in the plane is a partition of the plane into n regions V_1, V_2, $\cdots$, V_n such that any point in the region V_i is closer to the point p_i than to any other point in the set S.

The convex polygonal region V_i is called the *Voronoi polygon* of the point p_i in S. The vertices of the diagram are called *Voronoi vertices* and the line segments of the diagram are called *Voronoi edges*. The Voronoi diagram of a set S is denoted by *Vor(S)*. Note that Voronoi vertices are in general *not* the points in the set S.

We first prove some interesting and important properties about Voronoi diagrams. Throughout our proofs, we need to make a crucial assumption. (This assumption can be eliminated but some properties will no longer hold and the proofs will become much harder.)

ASSUMPTION

No four points in the set S are co-circular.

With this assumption, the Voronoi diagram has a simple structure.

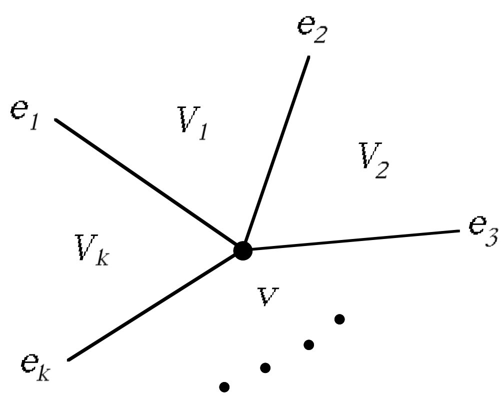

Figure 5.1: A Voronoi vertex and its incident Voronoi edges

Lemma 5.2.1 *Every Voronoi vertex has degree exactly three.*

PROOF. Any Voronoi vertex is the intersection of a set of Voronoi edges. Let $e_1, e_2, \cdots, e_k$ be a sequence of Voronoi edges incident on a Voronoi vertex v, such that the edge e_i is common to the Voronoi polygons V_{i-1} and V_i for $i = 2, 3, \cdots, k$, and edge e_1 is common to the Voronoi polygons V_k and V_1. Without loss of generality, we suppose that V_i is the Voronoi polygon of the point p_i in the set S, for $i = 1, \cdots, k$. See Figure 5.1.

Since the Voronoi vertex v is on e_1, v is equidistant from the points p_k and p_1. Similarly, since the Voronoi vertex v is on e_i, for $i = 2, \cdots, k$, v is equidistant from the points p_{i-1} and p_i. Therefore, v is equidistant from all points p_i, for $i = 1, \cdots, k$. This implies that all these points p_i, $i = 1, \cdots, k$, are on the circle whose center is v with the radius $|\overline{vp_1}|$. Since no four points in the set S can be co-circular, we conclude $k \leq 3$.

If $k = 2$, then both e_1 and e_2 are common to the Voronoi polygons V_1 and V_2. Hence they both belong to the perpendicular bisector of the segment $\overline{p_1p_2}$. Therefore, the vertex v is in fact an interior point of some Voronoi edge, so it is not a Voronoi vertex, a contradiction.

Finally, if $k = 1$. Then both sides of the edge e_1 are in the same Voronoi polygon V_1, so the Voronoi polygon is not convex, again a contradiction.

This proves that k must be exactly 3. $\square$

Suppose that we have constructed the Voronoi diagram for the set S of n points in the plane. The following lemma tells us that for any point p_i in S,

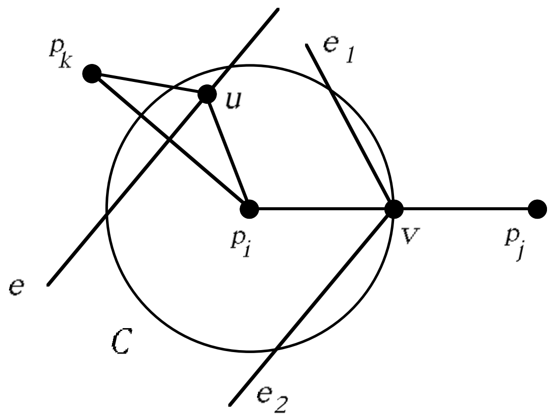

Figure 5.2: The nearest points defines a Voronoi edge

the nearest point in S can be found by "locally" looking at the corresponding Voronoi polygon V_i.

Lemma 5.2.2 *The nearest neighbor of every given point p_i in S has a Voronoi edge that bounds the Voronoi polygon V_i of the point p_i.*

PROOF. Let p_i and p_j be two points in the given set S, and suppose that p_j is the nearest neighbor of p_i. Let v be the midpoint of the segment $\overline{p_i p_j}$. Draw a circle C of radius $|\overline{p_i v}|$ whose center is p_i. We first show that the circle C is completely contained in the Voronoi polygon V_i. Suppose otherwise that e were a Voronoi edge of V_i that contains a point u that is in the interior of C. (See Figure 5.2.) Then e must lie on the perpendicular bisector of a segment $\overline{p_i p_k}$, where p_k is a point in S and $p_k \neq p_j$ (since the perpendicular bisector of $\overline{p_i p_j}$ is tangent to the circle C). Therefore, we must have

$$|\overline{p_i p_k}| \leq 2|\overline{p_i u}| < 2|\overline{p_i v}| = |\overline{p_i p_j}|$$

So p_i is closer to p_k than to p_j, contradicting the assumption that p_j is the nearest neighbor of p_i.

Therefore, the circle C is completely contained in the Voronoi polygon V_i. Since any point in the segment $\overline{v p_j}$ is closer to p_j than to p_i and the point v is on the circle C, the point v must be on the boundary of the Voronoi polygon V_i. Now we show that v is an interior point of some Voronoi edge on the boundary of V_i. Suppose otherwise, v is a Voronoi vertex. Let e_1 and e_2

be the Voronoi edges on the boundary of V_i such that e_1 and e_2 intersect at v. Since V_i is convex, the angle $\angle e_1 v e_2$ must be less than π. See Figure 5.2. But then at least one of the edges e_1 and e_2 intersects the interior of the circle C. This is impossible by our discussion above. Therefore, there is exactly one Voronoi edge e_1 of V_i that passes through v. The edge e_1 must be tangent to the circle C otherwise e_1 intersects the interior of the circle C. Thus, the edge e_1 is on the perpendicular bisector of the segment $\overline{p_i p_j}$, i.e., the edge e_1 is defined by the point p_j. $\square$

For each Voronoi vertex v, by Lemma 5.2.1, there are exactly three Voronoi polygons V_i, V_j, and V_k incident on v. Let p_i, p_j, and p_k be the three corresponding points in the set S. By the proof of Lemma 5.2.1, the point v is equidistant from these three points p_i, p_j, and p_k. Denote by $C(v)$ the unique circle defined by p_i, p_j, and p_k. The circle $C(v)$ is centered at v and has radius $|\overline{vp_i}|$.

Lemma 5.2.3 *For any Voronoi vertex v, the circle $C(v)$ contains no points of the set S in its interior.*

PROOF. Let p_i, p_j, and p_k be the three points in the set S which define the circle $C(v)$. Then by the definition of $C(v)$, v is on the boundary of the Voronoi polygons V_i, V_j, and V_k, which correspond to the points p_i, p_j, and p_k of the set S, respectively. Now by the definition of Voronoi polygons, in the set S, the points p_i, p_j, and p_k are the three closest points of the point v. If there is another point p_h in S that is in the interior of $C(v)$, then v would be closer to the point p_h than to any of the three points p_i, p_j, and p_k. This is a contradiction. $\square$

Now we discuss the relationship between CONVEX HULL and the Voronoi diagram.

Let r be a semi-infinite ray originating from a finite point p_0. For any point p on the ray r, denote by r_p the ray obtained by cutting away the segment $\overline{p_0 p}$ (excluding the point p) from the ray r.

Lemma 5.2.4 *Two points in the set S are consecutive hull vertices of the convex hull CH(S) if and only if the two corresponding Voronoi polygons share a Voronoi edge that is a semi-infinite ray.*

PROOF. Let p_1 and p_2 be two points in the set S, and let V_1 and V_2 be the two corresponding Voronoi polygons in *Vor(S)*, respectively.

Suppose that the points p_1 and p_2 are two consecutive hull vertices of the convex hull *CH(S)*. Then the segment $\overline{p_1p_2}$ is an edge on the boundary of the convex hull *CH(S)*. Let l be the straight line that passes through the segment $\overline{p_1p_2}$, then one side of l contains no points of the set S. Let r be a semi-infinite ray in the side of l that contains no points of S such that the ray r is on the perpendicular bisector of the segment $\overline{p_1p_2}$, and originates from the middle point of the segment $\overline{p_1p_2}$. Let p be a point on the ray r, draw a circle C_p centered at the point p with the radius $|\overline{pp_1}|$. Imagine that the point p travels along the ray r toward infinity. Then the radius of the circle C_p is getting larger and larger and the circle C_p is getting closer and closer to the straight line l (more precisely, for any fixed point p_l on the line l, the circle C_p can be arbitrarily close to p_l when the radius of the circle C_p is large enough). Since no points of the set S are in the same side of the line l as the point p, and the set S contains only finitely many points, there must be a point p_0 on the ray r such that for any point p on the ray r that is beyond the point p_0, no points of the set S, except the points p_1 and p_2, is contained in the interior or on the boundary of the circle C_p. That is, all points on the ray r that are beyond the point p_0 are closer to the points p_1 and p_2 than to any other points in the set S. By the definition of Voronoi edges, therefore, the entire semi-infinite ray r_{p_0} must be contained in the boundary of the Voronoi polygons V_1 and V_2. That is, the Voronoi polygons V_1 and V_2 share a Voronoi edge that is a semi-infinite ray.

Conversely, if the Voronoi polygons V_1 and V_2 share a Voronoi edge that is a semi-infinite ray r. Then every point on the ray r is equidistant from the points p_1 and p_2, by the definition of a Voronoi polygon, thus the ray r is on the perpendicular bisector of the segment $\overline{p_1p_2}$. Draw a circle C_p centered at a point p on the ray r such that C_p passes through the points p_1 and p_2. Then for any point p on the ray r, the circle C_p contains no other points of the set S in its interior. Let the point p travel along the ray r toward infinity, then the circle C_p is getting closer and closer to the straight line l that passes through the segment $\overline{p_1p_2}$, and C_p never contains any points of the set S in its interior. Since the set S is finite, we conclude that one side of the line l contains no points of the set S. This implies that the segment $\overline{p_1p_2}$ is an edge on the boundary of the convex hull *CH(S)*. Therefore, the points p_1 and p_2 are consecutive hull vertices of the convex hull *CH(S)*[1] □

Every hull vertex p of the convex hull *CH(S)* has a neighbor hull vertex p'. By Lemma 5.2.4, the corresponding Voronoi polygons V and V' share a semi-

[1]For simplicity, we suppose that no three points of the set S are co-linear.

infinite ray, therefore, the Voronoi polygon V corresponding to the point p must be unbounded. Conversely, if a Voronoi polygon V is unbounded, then V must share a semi-infinite ray with another unbounded Voronoi polygon V'. Again by Lemma 5.2.4, the two corresponding points p and p' of the set S are consecutive hull vertices of the convex hull *CH(S)*, therefore, the point p corresponding to the Voronoi polygon V must be a hull vertex. This proves the following corollary.

Corollary 5.2.5 *A Voronoi polygon V of Vor(S) is unbounded if and only if the corresponding point of the set S is a hull vertex of CH(S).*

Finally, we need a lemma to consider how much space is needed to represent a Voronoi diagram of a set of n points in the plane.

Lemma 5.2.6 *The Voronoi diagram contains at most $2n-3$ vertices, $3n-5$ edges and n regions.*

PROOF. Since the regions of a Voronoi diagram *Vor(S)* are Voronoi polygons that one-to-one correspond to the points of the set S, thus the Voronoi diagram *Vor(S)* of the set S has exactly n regions.

Every semi-infinite ray of the Voronoi diagram *Vor(S)* can be written in the form (v, α), where v is the Voronoi vertex from which the ray originates and α is the polar angle of the ray. Introduce a new vertex w. Replace each ray (v, α) of the Voronoi diagram *Vor(S)* by a finite edge (v, w), which may be a curve, not necessarily a straight line. The resulting picture is a planar imbedding I of a finite graph G that has the same number of regions and the same number of edges as the Voronoi diagram *Vor(S)*. The number of vertices of I is one more than that of the Voronoi diagram *Vor(S)*. Let V, E, and F be the number of vertices, the number of edges and the number of regions of the imbedding I. By Euler's formula

$$V - E + F = 2$$

By the above discussion, $F = n$. Moreover, we have $3V \leq 2E$ since each vertex of the graph G has degree at least 3 (note that there are at least three semi-infinite rays in the Voronoi diagram *Vor(S)*, since the convex hull *CH(S)* has at least three hull vertices, so by Lemma 5.2.4, *Vor(S)* has at least three unbounded Voronoi polygons). Combining these two relations, we get

$$V \leq 2n - 4$$

Remember that the number of vertices of the graph G is one more than that of the Voronoi diagram *Vor(S)*, we conclude that the number of vertices of the Voronoi diagram *Vor(S)* is at most $2n-3$.

Now apply Euler's formula again, we obtain

$$E \leq 3n - 5$$

□

Therefore, the number of vertices, the number of edges, and the number of regions of a Voronoi diagram are all of order $O(n)$.

We can use the Doubly-Connected Edge List (DCEL), as introduced in Section 1.4 to represent in computers a Voronoi diagram of a set of points in the plane. For this we need a slight generalization. For each unbounded Voronoi polygon V in a Voronoi diagram, we call the semi-infinite ray r of V the *first ray* of V if when we travel from infinity along the ray r toward the Voronoi vertex from which r originates, the region V is on our right. The other semi-infinite ray of V is called the *last ray* of V. Now given a semi-infinite ray r of a Voronoi diagram, suppose that r is the last ray of a Voronoi polygon V_i. Then in the edge node corresponding to the ray r, the pointer $P2$ will point to the semi-infinite ray that is the first ray of the Voronoi polygon V_i. Moreover, each region V, which is a Voronoi polygon of the Voronoi diagram, can be named by its corresponding point in the set S.

5.3 Constructing the Voronoi diagram

In this section, we present an algorithm that constructs the Voronoi diagram given a set S of n planar points. The algorithm runs in time $O(n \log n)$.

The algorithm is the standard divide-and-conquer method. We first give a rough sketch of the algorithm as follows.

```
Algorithm    VORONOI DIAGRAM

    Given:    a set S of n points in the plane
    Output:   the Voronoi diagram Vor(S) of S

    BEGIN
    1.   Presort the points in the set S by x-coordinate;
```

```
    2.   Call the subroutine Voronoi(S)
    END.
```

Where the subroutine *Voronoi(S)* is given as follows.

```
Algorithm     Voronoi(S)

     Given:     a set S of n points in the plane, sorted
                by x-coordinates
     Output:    the Voronoi diagram Vor(S) of S

     BEGIN
     1.   Split the set S into two approximately equal
          size subsets S_L and S_R by a vertical line
          L such that all points in S_L are in the left
          side of L and all points in S_R are in the
          right side of L;
     2.   Recursively call Voronoi(S_L) and Voronoi(S_R);
     3.   Merge Vor(S_L) and Vor(S_R) to construct Vor(S).
     END.
```

Step 1 in the algorithm *Voronoi(S)* can be done in linear time, since the given set S is sorted by x-coordinate. If the merge part (Step 3) in the algorithm *Voronoi(S)* can also be done in linear time, then by the standard technique in Algorithm Analysis, the algorithm *Voronoi(S)* runs in time $O(n \log n)$. Consequently, the algorithm *VORONOI DIAGRAM* runs in time $O(n \log n)$.

Therefore, the problem of constructing the Voronoi diagram of the set S in time $O(n \log n)$ is reduced to the problem of merging in linear time the two Voronoi diagrams *Vor(S_L)* and *Vor(S_R)* into the Voronoi diagram *Vor(S)*, where S_L and S_R are two sets separated by a vertical line l and $S_L \cup S_R = S$.

Consider the Voronoi diagrams *Vor(S)*, *Vor(S_L)*, and *Vor(S_R)*. We first discuss what of *Vor(S)* can be missing in *Vor(S_L)* and *Vor(S_R)*. Let e be a Voronoi edge of *Vor(S)* defined by two points p_i and p_j of S, that is, e is a Voronoi edge on the boundary between the Voronoi polygons V_i and V_j of the points p_i and p_j, respectively. By the definition of Voronoi polygons, the points p_i and p_j are the closest points in the set S to the points on the edge e. If both p_i and p_j are in the set S_L, then the points p_i and p_j must be the closest points in the set S_L to the points on the edge e since the set S_L is a subset of the set S. Therefore, the edge e must be also present in the Voronoi diagram *Vor*(S_L), either as a Voronoi edge or as part of a Voronoi edge of *Vor*(S_L). Similarly, if both p_i and p_j are in the set S_R, then the edge e must be also present in the Voronoi diagram *Vor*(S_R), either as a Voronoi edge or as part of a Voronoi edge of *Vor*(S_R). Therefore, a Voronoi edge e of *Vor(S)* that is missing in both *Vor*(S_L) and *Vor*(S_R) must be defined by two points such that one is in the set S_L and the other is in the set S_R.

Let σ be the subgraph of *Vor(S)* that consists of the Voronoi edges of *Vor(S)* that are defined by the pairs (p_i, p_j) of points in S such that $p_i \in S_L$ and $p_j \in S_R$. We do not presume that σ is a connected graph. We first discuss what σ looks like.

Lemma 5.3.1 *Each vertex of σ has degree exactly 2.*

PROOF. Since each vertex v of σ is also a Voronoi vertex of *Vor(S)*, by Lemma 5.2.1, the degree of v is at most 3 in σ. Suppose that e_1, e_2, and e_3 are the three Voronoi edges incident at v in the Voronoi diagram *Vor(S)*, and that V_1, V_2, and V_3 are the Voronoi polygons incident at v such that e_1 is between V_1 and V_2, e_2 is between V_2 and V_3, and e_3 is between V_3 and V_1. Let p_1, p_2, and p_3 be the three points in the set S that correspond to the Voronoi polygons V_1, V_2, and V_3, respectively.

If the vertex v has degree 3 in σ, then all Voronoi edges e_1, e_2, and e_3 are in σ. Since e_1 is in σ, by the definition of σ, without loss of generality, we can suppose that the point p_1 is in the set S_L and the point p_2 is in the set S_R. Then because e_2 is between V_2 and V_3 and e_2 is in σ, the point p_3 must be in the set S_L. Finally, because e_3 is between V_3 and V_1 and e_3 is in σ, we must also have that p_1 is in S_R. This gives us a contradiction that the point p_1 is in both sets S_L and S_R. Therefore, the vertex v cannot have degree 3 in σ.

If the vertex v has degree 1 in σ. Then suppose that the unique Voronoi edge that is incident on v and in σ is e_1. Thus we can suppose, without loss of generality, that the point p_1 is in the set S_L and the point p_2 is in the set S_R. However, now if the point p_3 is in the set S_L then the edge e_2 should be in σ, while if the point p_3 is in the set S_R, then the edge e_3 should be in σ, either case contradicts the assumption that the vertex v has degree 1 in σ.

This proves that each vertex of σ has degree 2 in σ. □

Therefore, each connected component of σ is either a closed simple cycle, or a simple chain whose both ends are semi-infinite rays.

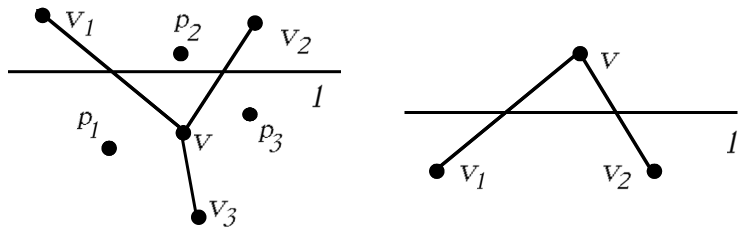

Figure 5.3: A horizontal line separating v from v_1 and v_2.

Recall that a chain C is said to be *monotone* if any horizontal line intersects the chain C in exactly one point.

Lemma 5.3.2 *Every connected component of σ is monotone. In other words, every horizontal line cuts a connected component of σ at exactly one point.*

PROOF. First we prove that no edge in σ can be horizontal. Suppose that an edge e in σ is horizontal. Let p_L and p_R be the two points in the set S that define the edge e, $p_L \in S_L$ and $p_R \in S_R$. Then the segment $\overline{p_L p_R}$ is vertical, contradicting the fact that the sets S_L and S_R are separated by a vertical line.

Now suppose that a connected component C of σ is not monotone. Since each vertex in σ has degree exactly 2 (Lemma 5.3.1), we must be able to find a vertex v on C whose two adjacent vertices are v_1 and v_2 such that a horizontal line l separates v from v_1 and v_2. See Figure 5.3.

Without loss of generality, suppose that v is below the line l, and that v_1 and v_2 are above the line l. The vertex v is a Voronoi vertex in the Voronoi diagram *Vor(S)*, and v_1 and v_2 are two adjacent Voronoi vertices in *Vor(S)*. Suppose that the third Voronoi vertex adjacent to v is v_3. The vertex v_3 must be below the horizontal line l since each Voronoi polygon has to be convex and each Voronoi vertex has degree exactly 3, by Lemma 5.2.1. Let p_1, p_2, and p_3 be the three points in the set S, such that edge $\{v, v_1\}$ is defined by p_1 and p_2, the edge $\{v, v_2\}$ is defined by p_2 and p_3, and the edge $\{v, v_3\}$ is defined by p_3 and p_1. See Figure 5.3. Without loss of generality, suppose that the point p_2 is in the set S_L, then both points p_1 and p_3 are in the set S_R. However, it is easy to see that we can draw two vertical lines l_1 and l_2 such that p_1 is on the left side of l_1 and p_2 is on the right side of

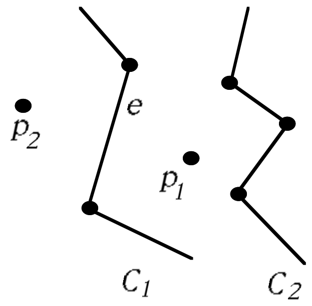

Figure 5.4: Two separated chains in σ

l_1, while p_2 is on the left side of l_2 and p_3 is on the right side of l_2. But this contradicts the assumption that the sets S_L and S_R are separated by a vertical line.

This proves that the connected component C of σ must be monotone. □

So no connected component of σ can be a cycle. Finally, we investigate how many connected components σ can have.

Lemma 5.3.3 *The graph σ has exactly one connected component.*

PROOF. First at all, σ must have at least one connected component since the Voronoi diagram is connected, so there is at least one Voronoi edge that bounds two Voronoi polygons corresponding to a pair of points that are from the sets S_L and S_R, respectively.

Now suppose that there are more than one connected components in σ. By Lemma 5.3.1 and Lemma 5.3.2, all these connected components are monotone chains, and no two of them intersect. Let C_1 and C_2 be the two adjacent chains in σ, i.e., there is no other chain in σ that is between the slice bounded by C_1 and C_2. Suppose also that C_1 is on the left of C_2. See Figure 5.4. Then all Voronoi polygons of *Vor(S)* that are between C_1 and C_2 correspond to points in a single set of S_L and S_R. Suppose all of them correspond to points in set S_L. Now look at any edge e on C_1. The edge e must be defined by a point p_1 that is between the slice of C_1 and C_2 and thereby in the set S_L and a point p_2 that is on the left side of C_1. See Figure 5.4. By the definition of σ, the point p_2 is in the set S_R. It is easy to see that there is a vertical line such that the point p_2 is on its left while the point p_1 is on its right. However, this contradicts the fact that all points in S_R should be on the right of all points in S_L. On the other hand, if all Voronoi polygons between C_1 and C_2 correspond to points in the set S_R, then we can similarly derive a contradiction by considering an edge on the chain C_2.

This proves that σ consists of a single monotone chain. $\Box$

Since σ is a single monotone chain, and two end edges of σ must be semi-infinite rays, we can talk about the "left side" and the "right side" of the chain σ. By the discussion above, we know that only the edges in σ could be missing in the Voronoi diagrams *Vor*(S_L) and *Vor*(S_R). Thus, we need to add the chain σ to the graph *Vor*(S_L) $\cup$ *Vor*(S_R) to construct the Voronoi diagram *Vor*(S).

Now we discuss what should be deleted from *Vor*(S_L) and *Vor*(S_R) in order to construct *Vor*(S).

Lemma 5.3.4 *Let e be a Voronoi edge or part of a Voronoi edge of Vor(S_L). The edge e disappears in Vor(S) if and only if e entirely lies on the right side of σ. Similarly, if e' is a Voronoi edge or part of a Voronoi edge of Vor(S_R), then e' disappears in Vor(S) if and only if e entirely lies on the left side of σ.*

PROOF. First of all, no point in S_L can be on the right side of σ, otherwise, we would be able to find a point p in S_L such that the Voronoi polygon of p has a boundary edge e on σ. This would give a point in the set S_R that is on the left of the point p, contradicting the definition of the sets S_L and S_R.

Let e be a Voronoi edge or part of a Voronoi edge of *Vor*(S_L) that entirely lies on the right side of σ. Let e be defined by two points p_1 and p_2 in the set S_L. If e is present in the Voronoi diagram *Vor*(S), then the closest points in S to a point p on the edge e would be p_1 and p_2. That is, the point p is in (the boundary of) the Voronoi polygon V_1 in *Vor*(S) that corresponds to the point p_1. Since V_1 must be convex, the segment $\overline{p_1p}$ must be in V_1. Moreover, since the point p_1 is in the interior of V_1, the segment $\overline{p_1p}$ in fact does not intersect any Voronoi edges in *Vor*(S) except the edge e. However, since the point p_1 is on the left side of σ and the point p is on the right side of σ, and σ partitions the plane into two separated parts, the segment $\overline{p_1p}$ must intersect σ at some point. That is, the segment $\overline{p_1p}$ must intersect some Voronoi edge of *Vor*(S) that is not e, since e is defined by two vertices in S_L while each edge on σ is defined by a point in S_L and a point in S_R. This is a contradiction. Therefore, the edge e of *Vor*(S_L) must disappear in *Vor(S)*.

This actually proves that for any point p on the right side of σ, the closest point in the set S must be a point in the set S_R.

Similarly, a Voronoi edge or part of a Voronoi edge of *Vor*(S_R) that entirely lies on the left side of σ disappears in *Vor(S)*.

On the other hand, let e be a Voronoi edge or part of a Voronoi edge of *Vor*(S_L) that lies entirely on the left side of σ. Suppose that e is defined by two points p_1 and p_2 in the set S_L. By the discussion above, the closest points in S to the points of e are still the points in the set S_L. Therefore, the two closest points in the set S to the points in e are still the points p_1 and p_2. That is, e is still on the boundary of the two Voronoi polygons V_1 and V_2 in *Vor(S)* corresponding to the points p_1 and p_2, respectively, i.e., e is still present in the Voronoi diagram *Vor(S)*.

This completes the proof. $\square$

By Lemmas 5.3.1, 5.3.2, 5.3.3, and 5.3.4, we can use the following algorithm to construct the Voronoi diagram *Vor(S)* from the Voronoi diagrams *Vor*(S_L) and *Vor*(S_R).

```
Algorithm     MERGE(Vor(S_L), Vor(S_R))

     Given:    the Voronoi diagrams Vor(S_L) and Vor(S_R)
     Output:   the Voronoi diagram Vor(S)

     BEGIN
     1.   Construct the separating chain SIGMA;
     2.   Delete all edges and partial edges of Vor(S_L) that are
          entirely on the right side of SIGMA;
     3.   Delete all edges and partial edges of Vor(S_R) that are
          entirely on the left side of SIGMA;
     END.
```

None of the steps can be obviously done in linear time. In the remaining of this section, we will discuss how to construct the separating chain σ. At the meantime, we find all intersections of σ with *Vor*(S_L) and *Vor*(S_R), and delete the proper edges and partial edges from *Vor*(S_L) and *Vor*(S_R).

First we consider how to construct the two semi-infinite rays of the chain σ. Let the two semi-infinite rays of the chain σ be l_1 and l_2. Suppose that

l_1 is the Voronoi edge of *Vor(S)* that is shared by two unbounded Voronoi polygons V_1 and V_2 of two points p_1 and p_2 in the set S, respectively. By Lemma 5.2.4, the points p_1 and p_2 are two consecutive hull vertices of the convex hull *CH(S)*, and the ray l_1 is on the perpendicular bisector of the segment $\overline{p_1p_2}$. Since l_1 is in σ, we can suppose that the point p_1 is in the set S_L and the point p_2 is in the set S_R. Therefore, the segment $\overline{p_1p_2}$ is in fact a supporting bridge of the two convex hulls $CH(S_L)$ and $CH(S_R)$ (see Section 4.1 and note that the two sets S_L and S_R are separated by a vertical line). Similarly, the ray l_2 is on the perpendicular bisector of the other supporting bridge of the two convex hulls $CH(S_L)$ and $CH(S_R)$. Therefore, if the two convex hulls $CH(S_L)$ and $CH(S_R)$ are known, then we can find the two bridges of $CH(S_L)$ and $CH(S_R)$ in linear time (see Section 4.1). With these two bridges, the two semi-infinite rays of σ can be found in constant time. Note that at meantime, we have also constructed in linear time the convex hull *CH(S)* of the set S as a by-product, which can be used for the later induction steps. Therefore, the algorithm of constructing the chain σ looks as follows.

```
Algorithm    CONSTRUCTING-SIGMA

     Given:    the Voronoi diagrams Vor(S_L) and Vor(S_R)
               and the convex hulls CH(S_L) and CH(S_R)
     Output:   the separating chain SIGMA and the convex
               hull CH(S)

     BEGIN
     1.   Find the upper bridge b_u and the lower bridge
          b_l of the two convex hulls CH(S_L), CH(S_R);
     2.   Construct the perpendicular bisectors l_u and
          l_l of the bridges b_u and b_l, respectively;
     3.   With the bridges b_u and b_l, construct the
          convex hull CH(S);
     4.   traverse the chain SIGMA in the direction of
          decreasing y, starting from the infinite end
          of the upper ray l_u of SIGMA, construct SIGMA
          edge by edge, until the lower ray l_l is
          reached;
     END.
```

Step 1 and Step 3 can be done in linear time, by the discussion of Section 4.1. Step 2 can be easily done in constant time. We must discuss how Step 4 is done in linear time. In the meantime, we also have to discuss how we find the intersections of σ with the Voronoi diagrams *Vor*(S_L) and *Vor*(S_R), and delete proper edges and partial edges from *Vor*(S_L) and *Vor*(S_R) and construct the Voronoi diagram *Vor*(S).

Remember that we can use Doubly-Connected-Edge-List (DCEL) to represent a Voronoi diagram. We suppose that the Voronoi diagrams *Vor*(S_L) and *Vor*(S_R) are represented by two DCELs. Moreover, we suppose that the rotation of edges incident on each vertex of *Vor*(S_L) is given in counterclockwise order in the corresponding DCEL, while the rotation of edges incident on each vertex of *Vor*(S_R) is given in clockwise order. Therefore, the regions of *Vor*(S_L) will be traced clockwise, while the regions of *Vor*(S_R) will be traced counterclockwise, by the algorithm *TRACE-REGION* given in Section 1.4.

Now suppose inductively that we are traversing the chain σ in the direction of decreasing y, and we are in the intersection area of the Voronoi polygon V_L of *Vor*(S_L) of some point $p_L \in S_L$ and the Voronoi polygon V_R of *Vor*(S_R) of some point $p_R \in S_R$. Since in this area, the closest point of S_L is p_L and the closest point of S_R is p_R, we must follow the perpendicular bisector of the segment $\overline{p_L p_R}$, in the direction of decreasing y. Suppose along this direction, we are traversing an edge e_0 in σ. We keep going along this direction until we hit an Voronoi edge e of *Vor*(S_L) or of *Vor*(S_R). Without loss of generality, suppose that e is a Voronoi edge of *Vor*(S_R). The edge e is on the boundary of the Voronoi polygon V_R, so e must be defined by the point p_R and another point $p'_R \in S_R$. Let the Voronoi polygon of the point p'_R in *Vor*(S_R) be V'_R. If we keep going the same direction, we will cross the edge e and enter the Voronoi polygon V'_R of *Vor*(S_R). Now the closest point in the set S_R is the point p'_R. The closest point in the set S_L is still the point p_L. Therefore, to continue traversing the chain σ, we should go along the perpendicular bisector of the segment $\overline{p_L p'_R}$, in the direction of decreasing y. To make this change, at the intersection of the chain σ and the edge e, we simply switch our direction from the perpendicular bisector of $\overline{p_L p_R}$ to the perpendicular bisector of $\overline{p_L p'_R}$, both in the direction of decreasing y. Now we are on the next edge of the chain σ. We inductively work in this way to find the next edge of the chain σ, and so on, until we hit the low ray l_l of σ.

Note that we have no difficulty to initialize this process. We can start at a point p on the upper ray l_u that is "far enough" from the upper bridge $b_u = (p_L, p_R)$, where $p_L \in S_L$ and $p_R \in S_R$. Then we must be in the intersection area of the Voronoi polygon of p_L in *Vor*(S_L) and the Voronoi polygon of p_R in $Vor(S_L)$.

Summarizing this discussion, we get the following algorithm.

```
Algorithm     CONSTRUCTING-SIGMA

     BEGIN
     1.   Let p_0 be a point on the upper ray l_u that
          is far enough from the upper bridge
          b_u = (p_L, p_R), where p_L is in S_L, and
          p_R is in S_R.  Let l_0 be the semi-infinite
          ray originating from the point p_0 that has
          the opposite direction of the ray l_u, and let
          V_L and V_R be the Voronoi polygons of the
          points p_L and p_R in the Voronoi diagrams
          Vor(S_L) and Vor(S_R), respectively;
     2.   IF l_0 is not identical with the lower ray l_l
          THEN
     2.1. Compute the point q_L that is the intersection
          of l_0 with the boundary of V_L, and compute
          the point q_R that is the intersection of l_0
          with the boundary of V_R;
     2.2  IF p_0 is closer to q_L than to q_R, THEN
          suppose that the point q_L is on a Voronoi edge
          e_L of Vor(S_L) that is defined by the point
          p_L and another point p_L' in S_L, then let
          p_0 = q_L, and let l_0 be the semi-infinite ray
          originating from q_L that is on the perpendicular
          bisector of the segment {p_L', p_R} in the direction
          of decreasing y.  Finally, let the current Voronoi
          polygon V_L of Vor(S_L) be the Voronoi polygon of
          the point p_L';
     2.3. IF p_0 is closer to q_R than to q_L  THEN
          update the parameters p_0, l_0, and V_R similarly;
     3.   Go back to Step 2.
     END.
```

As we mentioned before, Step 1 can be done in constant time when we know the upper and lower bridges of the convex hulls $CH(S_L)$ and $CH(S_R)$. The loop of Step 2 - Step 6 can be executed at most $O(n)$ times since each execution of the loop finds one more edge on the chain σ and as a subgraph of $Vor(S)$, the chain σ contains at most $O(n)$ edges. Within each execution of the loop, Step 4 and Step 5 take at most constant time since we only need some local modifications.

The remaining question is how much time is needed to find the intersecting points q_L and q_R in each execution of the loop of Step 2 - Step 6.

Knowing p_0, l_0, V_L and V_R, we can trace the boundary edges of the polygon V_L to find a boundary edge of V_L that contains the point p_L. Similarly we can find the point p_R. However, since the chain σ can contain up to $\Omega(n)$ edges and the polygons V_L and V_R can have up to $\Omega(n)$ boundary edges, this straightforward algorithm would run in time $\Omega(n^2)$ to find the chain σ. Therefore, in order to construct σ in linear time, we must not trace each Voronoi polygon of the Voronoi diagrams *Vor*(S_L) and *Vor*(S_R) too many times during the entire process of constructing the chain σ.

Lemma 5.3.5 *Suppose that the chain σ is traversed in the direction of decreasing y. Let V_L be a Voronoi polygon of the Voronoi diagram Vor(S_L). If the chain σ makes a turn at an interior point of V_L, then the turn must be a right turn. Similarly, if the chain σ makes a turn at an interior point of some Voronoi polygon of Vor(S_R), then the turn must be a left turn.*

PROOF. Suppose that the point in S_L corresponding to the Voronoi polygon V_L in *Vor*(S_L) is p_L. Let $v_1v_2v_3$ be a turn of the chain σ in the direction of decreasing y, where v_2 is an interior point of V_L. Then vertices v_1 and v_3 are also in V_L, since at an exit of V_L, the chain must make another turn. (However, v_1 and v_3 may be on the boundary of V_L.) Since V_L is convex, the segments $\overline{v_1v_2}$ and $\overline{v_2v_3}$ are entirely contained in V_L. Therefore, the closest point in the set S_L to the points on $\overline{v_1v_2}$ and $\overline{v_2v_3}$ is still p_L. Let V be the Voronoi diagram of the point p_L in the Voronoi diagram *Vor*(S). By the definition of the chain σ, the segments $\overline{v_1v_2}$ and $\overline{v_2v_3}$ are two consecutive boundary edges of V. When we traverse from v_1 to v_2 then to v_3, the point p_L must be on our right, because $\overline{v_1v_2}$ and $\overline{v_2v_3}$ are on the chain σ and all points of S_L are on our right when we traverse σ in the direction of decreasing y. Since V is a convex polygon, the turn we make at the point v_2 must be a right turn. $\square$

By this lemma, we can find the point q_L and q_R in the algorithm *CONSTRUCTING-SIGMA* as follows. Suppose that we entered the Voronoi polygon V_L at the point p_0, which is on a boundary edge e_0 of V_L. Starting from the edge e_0, trace the region V_L clockwise, using the algorithm *TRACE-REGION* in Section 1.4, until we find the boundary edge e_L that intersects

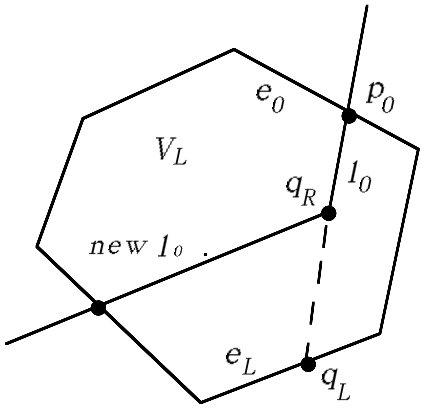

Figure 5.5: σ makes only right turn in V_L

the ray l_0 at point q_L. (Note there is only one such a boundary edge of V_L.) Similarly find the point q_R. If the point p_0 is closer to the point q_R than to the point q_L, then the chain σ makes a turn at the point q_R. Since the point q_R is in the interior of V_L, by Lemma 5.3.5, the turn of σ at q_R must be a right turn. We modify the parameters p_0, l_0, and V_R properly. Now we have to find the intersection of the new l_0 with V_L again. However, since the turn of the chain σ at the point q_R is a right turn, the new l_0 cannot intersect any edges between the edges e_0 and e_L we have already traced. See Figure 5.5. Therefore, to find the intersection of V_L and the new l_0, we can trace the region V_L starting from the edge e_L. If the chain σ eventually exits V_L, then we must come to an exit edge e_E of V_L for σ before we trace back to the edge e_0. Therefore, to traverse the partial chain of σ in the Voronoi polygon V_L from the entering edge e_0 to the exit edge e_E, we only have to trace the boundary edges of V_L between the edge e_0 and the edge e_E clockwise.

This is still not the end, however. Although traversing a continuous partial chain of σ in the Voronoi polygon V_L can be done efficiently, there may be more than one continuous partial chain of σ that are contained in the Voronoi polygon V_L. We must prove that traversing all these continuous partial chains of σ in V_L can also be done efficiently. Let P_1 and P_2 be two continuous partial chains of σ such that P_1 enters V_L at an edge e_0 and exits V_L at an edge e_E, while P_2 enters V_L at an edge e'_0 and exits V_L at an edge e'_E. As we discuss above, to traverse P_1, we need to trace the boundary edges of V_L between the edge e_0 and e_E clockwise. As we explained in the proof of Lemma 5.3.5, the partial chains P_1 and P_2 are all on the boundary of the Voronoi polygon V of the point p_L in the Voronoi diagram *Vor(S)*. Since all turns on P_1 are right turn, the area in V_L between P_1 and the partial boundary of V_L we have traced is excluded from the Voronoi polygon V of the point p_L in the Voronoi diagram *Vor(S)*. Now the partial chain P_2 is also on the boundary of the Voronoi polygon V, so P_2 cannot enter or exit V_L from an edge that is between e_0 and e_E. Therefore, the edges e'_0 and e'_E must be among those untraced boundary edges of V_L (including the edges e_0 and e_E). In other words, the sequence of the boundary edges of V_L we trace for P_1 and the sequence of the boundary edges of V_L we trace for P_2 are internally disjoint. This conclusion is easily generalized to more than two continuous partial chains of σ in the Voronoi polygon V_L.

Therefore, for a boundary edge of V_L at which no partial chain of σ enters or exits, our algorithm traces it at most once. On the other hand, for a boundary edge of V_L at which some partial chains of σ enter and/or exit, each visit of the edge produces a new edge on the chain σ. Therefore, the total time of the traversing of the chain σ is bounded by $O(n_\sigma + m_L)$, where n_σ is the number of edges on the chain σ, and m_L is the sum of the region sizes over all regions of *Vor(S_L)*. Since n_σ is bounded by n, the number of points in the set S, and m_L equals two times the number of edges of *Vor(S_L)*, which is bounded by $3n$, by Lemma 5.2.6, the total time to construct the chain σ is bounded by $O(n)$.

The traversing of the chain σ in a Voronoi polygon V_R of the Voronoi diagram *Vor(S_R)* can be done symmetrically. Here since the rotation of edges incident on each vertex of *Vor(S_R)* is clockwise in the DCEL, the regions of *Vor(S_R)* are traced counterclockwise. Completely similar as we did in Lemma 5.3.5, we can prove that if σ makes a turn at an interior point of V_R, then the turn must be a left turn. Therefore, the chain σ can also be traversed efficiently in the Voronoi polygons of *Vor(S_R)*, and the total time is also bounded by $O(n)$.

Finally we explain how to delete the edges and partial edges of *Vor(S_L)* that are on the right side of σ and the edges and partial edges of *Vor(S_R)* that are on the left side of σ. Note that when we traverse the chain σ in the direction of decreasing y, we can find all intersections of σ with the Voronoi diagrams *Vor(S_L)* and *Vor(S_R)*. Therefore, it is easy for us to decide which part of the Voronoi diagrams should be thrown away.

Therefore, we conclude that the running time of the algorithm *MERGE(Vor(S_L), Vor(S_R))* is $O(n)$. Consequently, the running time of the algorithm *VORONOI DIAGRAM* is $O(n \log n)$.

Theorem 5.3.6 *Given a set S of n points in the plane, the Voronoi diagram of S can be constructed in time $O(n \log n)$.*

Chapter 6

Prune and Search

Prune and Search is a technique originally used for finding medians developed by Blum, Floyd, Pratt, Rivest, and Tarjan [5]. The technique, as applied to median finding, throws out a constant fraction of the numbers during each iteration of a loop. Solving the recurrence gives us an $O(n)$ time algorithm for finding a median.

Let us have a more detailed review of the above algorithm. To find the median of a list, we first generalize the problem a little bit. We consider the problem of finding the kth smallest number of a list L of n numbers, for an arbitrary k. We first divide the n numbers into $n/5$ groups, each of 5 numbers, then find the median for each of the groups. Let L' be the list of these $n/5$ medians. Recursively find the median m of the list L'. It can be proved that m is greater than or equal to at least one fourth of the numbers in the original list L, and also less than or equal to at least one fourth of the numbers in the original list L. Therefore, the number m partitions the list L into two sublists L_1 and L_2 such that all numbers in L_1 are less than or equal to m and all numbers in L_2 are greater than or equal to m. Moreover, the size of each of these two sublists L_1 and L_2 is at least one fourth of the original list L. Now if the sublist L_1 contains at least k numbers, then recursively call the algorithm to find the kth smallest number in the list L_1. On the other hand, if the sublist L_1 contains h numbers such that $h < k$, then recursively call the algorithm to find the $(k-h)$th smallest number in the sublist L_2. In any case, the size of the sublist we are going to work on is at most three fourth of the size of the original list L. The detailed discussion of this algorithm can be found in [2], Section 3.5.

Let us analyze the above Median Finding algorithm. Suppose that the time complexity of the algorithm is $T(n)$ on inputs of size n. Then to find the median of the list L' of the $n/5$ medians takes time $T(n/5)$. Since both lists L_1 and L_2 are of size at most $3n/4$, to find the kth smallest number in the list L_1 or to find the $(h-k)$th smallest number in the list L_2 takes time at most $T(3n/4)$. It is also clear that the computation for the rest of the algorithm can be done in time bn, where b is a constant. Therefore, the function $T(n)$ satisfies the following recurrence.

$$T(n) = T(n/5) + T(3n/4) + bn$$

Let g be an integer such that $g \geq 20b$ and $g \geq T(1)$, then it is not difficult to prove, by induction, that

$$T(n) \leq gn$$

That is, $T(n) = O(n)$.

A general form of a prune and search algorithm can be described, informally, as following.

```
Algorithm     PRUNE AND SEARCH

     Given:    a problem P of size n
     Output:   a solution S of the problem

     BEGIN
     0.   IF the size n of P is small
            Solve P directly and STOP;
     1.   'Prune' the problem P into k smaller problems
          P1, P2, ..., Pk, of size (c_1)n, (c_2)n, ...,
          (c_k)n, respectively, such that
              (c_1) + (c_2) + ... + (c_k) <= c < 1
          where c is a fixed constant;
     2.   Recursively solve the problems P1, P2, ..., Pk;
     3.   Use the results of Step 2 to derive a solution
          for the problem P;
     END.
```

Suppose that the time complexity of the algorithm PRUNE AND SEARCH is $T(n)$, and suppose that Step 1 and Step 3 of the algorithm take time $F(n)$. Then the function $T(n)$ can be represented by the following recurrence:

$$T(n) = T(c_1 n) + T(c_2) + \cdots + T(c_k) + F(n)$$

The time complexity $T(n)$ of the algorithm PRUNE AND SEARCH can be obtained by solving the above recurrence. In particular, if the function $F(n)$ is $O(n)$, then it can be proved that the function $T(n)$ is also $O(n)$.

6.1 Kirkpatrick-Seidel's algorithm for convex hulls

We present a prune and search algorithm for constructing convex hulls, which is due to Kirkpatrick and Seidel [13].

Let us first consider the following problem.

Problem:

given two sets S_L and S_R of points in the plane, such that there is a vertical line l such that S_L is on the left of l and S_R is on the right of l, how do we find the upper bridge of S_L and S_R, i.e., the line passing through a point in S_L and a point in S_R such that all points in S_L and S_R are on or below the line?

In the algorithm *MERGEHULL*, we know that when the convex hulls of both sets S_L and S_R are known, the upper bridge can be constructed in linear time by lifting a line segment between S_L and S_R until the segment cannot be lifted anymore. However, constructing the convex hulls for S_L and S_R itself takes $\Omega(n \log n)$ time, which is too much to us. What we expect is a linear time algorithm solving this problem.

The prune and search technique is used to solve the above problem. The main idea involves finding a "suitable" line in $O(n)$ time, a line that allows us to throw away a constant fraction of the points as candidates for the bridge. We then recurse on the remaining points.

Definition An *upper supporting line* of a set S of points in the plane contains at least one point of S, and all points of S lie below or on the line.

Now let L_p be an upper supporting line of the set $S_L \cup S_R$ passing through a point p of S_L, and suppose that L_p is not an upper bridge of S_L and S_R, also let $\overline{p'q'}$ be a line segment where $p', q' \in S_L \cup S_R$. If the slope of $\overline{p'q'}$ is not less than the slope of L_p, then it is easy to see that the line segment $\overline{p'q'}$ cannot be contained in the upper bridge of S_L and S_R. In particular, the point p' cannot be on the upper bridge. Similarly, if L_q is an upper supporting line of the set $S_L \cup S_R$ passing through a point q of S_R, and suppose that L_q is not an upper bridge of S_L and S_R, and $\overline{p''q''}$ is a line segment where $p'', q'' \in S_L \cup S_R$. If the slope of $\overline{p''q''}$ is not larger than the slope of L_q, then the line segment $\overline{p''q''}$ cannot be in the upper bridge of S_L and S_R. In particular, the point q'' cannot be on the upper bridge.

This crucial observation gives us the following algorithm to solve the above problem.

```
Algorithm     UpperBridge(S, l)

     Given:   a set S of n points in the plane and a vertical
              line l separating S into a left subset S_L and
              a right subset S_R
     Output:  the upper bridge of the sets S_L and S_R

     BEGIN
     1.   Arbitrarily pair up the points of S:
          (p_1, q_1), (p_2, q_2), ..., (p_{n/2}, q_{n/2});
     2.   Let the slope of the segment [p_i, q_i] be s_i,
          i = 1, ..., n.  Using the Median Finding
          algorithm to find a pair (p_l, q_l) such that
          the slope s_l of it is the median in
          s_1, s_2, ..., s_{n/2};
     3.   Construct an upper supporting line L with the
          slope s_l.  To do this, draw a line with the
          slope s_l through each point in S.  Then take
          the line that has the highest intersection with
          the y-axis;
     4.   If L passes through points in both S_L and S_R,
          then L is the upper bridge we want, so we stop
          and return;  Otherwise, we do the following steps;
     5.   If L passes through only points in S_L, then scan
          the list of pairs (p_i, q_i) we made in Step 1.
          If the slope of a segment [p_i, q_i] is not less
          than the slope of the supporting line L, then
          throw away the point p_i;
     6.   If L passes through only points in S_R, then scan
          the list of pairs (p_i, q_i) we made in Step 1.
          If the slope of a segment [p_i, q_i] is not larger
          than the slope of the supporting line L, then
          throw away the point q_i;
     7.   Let S' be the set of the remaining points of S,
          recursively call  UpperBridge(S', l).
     END.
```

The correctness of the algorithm *UpperBridge* can be proved using the discussion preceding the algorithm: we never delete the points on the upper bridge. Now let us consider the time complexity of the algorithm. Step 1, Step 3 and Step 4 can be obviously done in time $O(n)$. Step 2 can be done in linear time using the Median Finding algorithm described before. Now let us consider how many points are left for the recursive call of the algorithm in Step 7. Since the slope s_l of L is the median of the slopes of the segments $\overline{p_i q_i}$, for $i = 1, \cdots, n/2$, if Step 5 is executed, at least half of the segments $\overline{p_i q_i}$ have a slope not less than s_l. So the corresponding points p_i are thrown away. Therefore, at least one fourth of the points in S are thrown away. Similarly, if Step 6 is executed, also at least one fourth points in S are thrown away. Therefore, at most three fourth points in S are left for the recursive call in Step 7. Let $T(n)$ be the time complexity of the algorithm *UpperBridge*, then we have the following recurrence relation:

$$T(n) = O(n) + T(\frac{3n}{4})$$

It is easy to obtain that $T(n) = O(n)$. Therefore, the algorithm *UpperBridge* runs in linear time.

With this preparation, now we are able to present Kirkpatrick-Seidel algorithm as follows.

```
Algorithm     KIRKPATRICK-SEIDEL(S)

    Given:   a set S of n points in the plane
    Output:  the convex hull of S

    BEGIN
    1.     Let p_min and p_max be the points in S with
           the smallest and the largest x-coordinates,
           respectively, let the line through p_min and
           p_max be L;
    2.     Split the set S into two subsets S' and S'',
           such that S' is the set of points of S above
           the line L, and S'' is the set of points of
           S below the line L;
    3.     Call UpperHull(S', p_min, p_max);
    4.     Call LowerHull(S'', p_min, p_max);
    END.
```

Step 1 and Step 2 of the algorithm KIRKPATRICK-SEIDEL can be done in linear time. The subroutines *UpperHull* and *LowerHull* are similar. We only discuss the subroutine *UpperHull* as follows.

```
Algorithm    UpperHull(S, p_min, p_max)

     Given:    a set S of n points in the plane that
               are all above the line through the
               points p_min and p_max, which are also
               points in S
     Output:   the upper hull of the set S

     BEGIN
     1.   Using the Median Finding algorithm to find a
          vertical line L_d which divides the set S
          into two equal size subsets S_L and S_R;
     2.   Call UpperBridge(S, L_d) to construct the
          upper bridge [p_l p_r] of S_L and S_R, where
          p_l is in S_L and p_r is in S_R;
     3.   Let S' be the set of points in S that are
          above the line through p_min and p_l, and let
          S'' be the set of points in S that are above
          the line through p_r and p_max;
     4.   Recursively call UpperHull(S', p_min, p_l) and
          UpperHull(S'', p_r, p_max);
     5.   Merge the results of Step 4 with the upper
          bridge [p_l p_r] properly;
     END.
```

Now let us consider the time complexity of the algorithm *UpperHull*. Suppose that there are k points of the set S on the convex hull *CH(S)*. Let $T(n,k)$ be the time complexity of the algorithm. Step 1 takes time $O(n)$ by the Median Finding algorithm. Step 2 takes time $O(n)$ by our analysis of the algorithm *UpperBridge*. Step 3 and Step 5 can be obviously done in time $O(n)$. Now suppose that there are k' hull vertices of *CH(S)* contained in the set S', and k'' hull vertices of *CH(S)* contained in the set S''. Then the recursive calls in Step 4 takes time at most $T(n/2,k') + T(n/2,k'')$, where $k' + k'' = k$, since it is easy to see that $S' \subseteq S_L$ and $S'' \subseteq S_R$. Therefore, we have the following recurrence relation.

$$T(n,k) = T(n/2,k') + T(n/2,k'') + O(n)$$

We can prove by induction on k that $T(n, k) = O(n \log k)$. The detailed proof is left to the reader.

Thus KIRKPATRICK-SEIDEL algorithm runs in time $O(n \log k)$. When k is small, KIRKPATRICK-SEIDEL algorithm is not worse than Jarvis' March that has the time complexity $O(kn)$ (even better), and when k is large, it is still not worse than Graham Scan, since k is always less than or equal to n. However, KIRKPATRICK-SEIDEL algorithm has very nasty constants because the algorithm to find the median is hard to program. So, in the real world, people use Graham Scan.

Finally, we briefly discuss the difference between *MERGEHULL*, *QUICKHULL* and KIRKPATRICK-SEIDEL algorithm. KIRKPATRICK-SEIDEL algorithm has the advantages in both *MERGEHULL* and *QUICKHULL*. It divides the given set evenly, like *MERGEHULL*, and merges partial hulls efficiently, like *QUICKHULL*. The time complexity of *MERGEHULL* has a factor $\log n$ instead of $\log k$ because in the two recursive calls, many points that are in the convex hulls of the two subsets but not in the convex hull of the original set are introduced. In *QUICKHULL*, the median point of the given set S may unfortunately be not a hull vertex, therefore algorithm would not work if we simply replace the furthest point in the algorithm by the median point.

6.2 Point location problems

In the remaining of this chapter, we discuss the point location problems. We first present a simple algorithm, the slab method, which runs in $O(n^2)$ preprocessing time, $O(n^2)$ storage, and $O(\log n)$ query time, where the geometric sweeping technique is used in the preprocessing. Then we give an optimal algorithm for the point location problem, Kirkpatrick's algorithm, which runs in $O(n)$ preprocessing time, $O(n)$ storage, and $O(\log n)$ query time for connected PSLGs, where the refinement method, which is a variety of prune and search technique, is used.

6.2.1 Complexity measures and a simple example

Suppose that we have a PSLG G, and we want to know in which region of G a given query point is located. In the simplest case, we have only one query point. Then we can search the point in each region of G directly to find the region containing the point. A one-time query of this type is called *single shot*. However, we may have many query points and want to find the containing region for each query point. Such queries are called *repetitive-mode queries*.

In the case of repetitive-mode queries, it may be worthwhile to arrange the PSLG G into a more organized structure to facilitate searching. Therefore, when we are considering the problem of repetitive-mode queries, we are interested in three computational resources: the *preprocessing time* that is used to convert the given PSLG into an organized structure, the *storage* that is used to store the organized structure, and the *query time* that is needed to locate each query point.

Suppose that the input PSLG G has n vertices. In general, we cannot expect that the preprocessing time is less than $O(n)$ since even reading the input PSLG G takes time $\Omega(n)$. Similarly, we cannot expect that the storage used for the organized structure is less than $O(n)$ since even storing the unorganized structure, the PSLG G itself needs $\Omega(n)$ space. Finally, as pointed out by Knuth [14], any algorithm for searching an ordered table of length n by means of comparisons can be represented as a binary tree of n leaves, thus in the worst case, the searching time is at least $\Omega(\log n)$. While the point location problem is clearly a generalization of searching, we conclude that the query time of the point location problem is at least $\Omega(\log n)$.

Let us consider a simple example. Suppose that the PSLG G is a convex polygon P of n vertices. So the vertices of P are given in, say, counterclockwise ordering $\{v_1, v_2, \cdots, v_n\}$. We first organize P by the following algorithm:

```
Algorithm    PREPROCESSING (P)
      Given:    a convex polygon P
      Output:   an organized structure L for P

      BEGIN
      1.   Find an internal point p_0 of P;
      2.   For each edge {v_i, v_(i+1)} of P, i = 1, ..., n,
           (where we let v_(n+1) = v_1) construct the wedge
           W_i formed by the ray started at the point p_0
           and passing through v_i (call it the starting ray
           of the wedge W_i) and the ray started at p_0 and
           passing through v_(i+1) (call it the ending ray
           of the wedge W_i);
      3.   Sort the wedges { W_i | 1 <= i <= n } by the slopes
           of their starting ray.  Let the sorted list be L;
      4.   Attach the edge {v_i, v_(i+1)} to the element of L
```

```
          corresponding to the wedge W_i,  for i = 1, ..., n;
     END.
```

With the list L, we can locate each query point by the following algorithm.

```
Algorithm     QUERY (q)

     Given:    a query point q and the organized structure
               L of P
     Output:   an answer to "the point q is inside P?"

     BEGIN
     1.   Compute the slope of the ray started at p_0 and
          passing through q;
     2.   Using binary search on the list L to locate the
          point q in a wedge W_i;
     3.   The point q is inside the convex polygon P if and
          only if the point q is inside the triangle formed
          by the wedge W_i and the edge {v_i, v_(i+1)};
     END.
```

Now we analyze the above algorithms.

Preprocessing time

The preprocessing is implemented by the algorithm *PREPROCESSING*. The internal point p_0 of P can be found by, for example, computing the centroid of the triangle determined by any three vertices of the convex polygon P. Thus Step 1 takes constant time. Step 2 takes time $O(n)$ because given two points, the equation of the ray passing through them can be constructed in constant time. To consider Step 3, we suppose, without loss of generality, that the slope of the starting ray of the wedge W_1 is 0 (otherwise, we rotate the system to make this). Then the wedges, sorted by their starting rays, are exactly in the order W_1, W_2, $\cdots$, W_n. Since we can read the edges of P in counterclockwise ordering, the wedges can be read exactly in the sorted ordering. Therefore, Step 3 to construct the list L, together with Step 4 to attach edges to the list L, takes time $O(n)$. We conclude that the total preprocessing time is $O(n)$

Storage

Since the equation of each ray is a linear equation of two variables, which can be represented in constant space, each element of the list L takes constant space. Consequently, the list L takes $O(n)$ space.

Query time

Locating each query point is implemented by the algorithm *QUERY (q)*. It is easy to see that Step 1 in the algorithm takes constant time, while Step 2 in the algorithm takes $O(\log n)$ time. Finally, knowing the two rays forming the wedge W_i and the edge $\{v_i, v_{i+1}\}$, we can determine in constant time if the query point is inside the triangle formed by the wedge and the edge.

We conclude with the following theorem.

Theorem 6.2.1 *Point location problem on convex polygons can be solved with $O(n)$ preprocessing time, $O(n)$ storage, and $O(\log n)$ query time.*

6.2.2 Slab method

Now we consider the point location problem on general PSLGs. Let G be a PSLG with n vertices. Through each vertex of G, we draw a horizontal line. The plane is subdivided by these horizontal lines into "slabs". Since G is a PSLG, there is no edge-crossing in G and since we have drawn a horizontal line through each vertex of G, in the interior of each slab, there is neither edge intersection nor vertex of G. Therefore, the edge segments of G contained in a slab can be ordered from left to right. If we construct a list of edge segments, ordered from left to right, for each slab, then the algorithm for locating a query point will look as follows, where L is a list of slabs, sorted by y-coordinate (that is, any point in slab $L[j]$ has a larger y-coordinate than a point in slab $L[i]$ for $i < j$). Each element $L[i]$ in the list L also has a pointer to a list l_i of edge segments in the corresponding slab, ordered from left to right.

```
Algorithm    LOCATING (p_0)

      Given:   a query point p_0, and a PSLG G represented
               by a list L of slabs.  Each slab L[i] is
               associated with a list l_i of edge segments
               in the slab ordered from left to right
      Output:  a region of G that contains the point p_0
```

```
BEGIN
1.   Using the y-coordinate y_0 of the point p_0,
     we perform binary search in the list L to find
     a slab L[i] that contains the point p_0;
2.   Using the x-coordinate x_0 of the point p_0,
     we perform binary search in the list l_i to
     find a pair of edge segments e_1 and e_2 such
     that the point p_0 is between these two edge
     segments;
END.
```

There are exactly n vertices of G, therefore, the binary search in Step 1 of the algorithm can be done in time $O(\log n)$. Moreover, since G is a planar graph it has $O(n)$ edges. Each edge of G can contribute at most one edge segment to a slab. Thus each slab contains $O(n)$ edge segments. Therefore, the binary search in Step 2 of the algorithm can also be done in time $O(\log n)$. Two consecutive edge segments in a slab correspond to a unique region of the PSLG G. So if we attach the region name to each pair of consecutive edge segments in each slab, then after Step 2 of the above algorithm, we can read directly the name of the region that contains the point p_0. We conclude that the query time of this slab method is $O(\log n)$.

Now we discuss how we produce and store the sorted list L and the sorted lists l_i. As the analysis given above, each list l_i contains at most $O(n)$ edge segments, thus the space we used to store the lists L and l_i's is bounded by $O(n^2)$. This storage cannot be improved since some PSLG does have the structure such that there are $\Omega(n^2)$ edge segments in the slabs. Figure 6.1 gives an example of such a PSLG.

A straightforward method to produce these lists is to sort the vertices of G by y-coordinate first to get the sorted list L of the slabs, then for each slab $L[i]$, sort the edge segments in the slab to get the sorted list l_i. Then the time complexity to obtain the list L is $O(n \log n)$, and the time complexity to obtain all the lists l_i, $i = 1, 2, \cdots, n+1$ will be $O((n+1)n \log n) = O(n^2 \log n)$. Can we do better?

Again we exploit the idea of geometric sweeping. We maintain the edge segments of a slab in a 2-3 tree and let the edge segments be ordered from left to right in the tree. When we move up from one slab to another slab, we look at those vertices on the boundary of the two slabs. We delete the lower edges and insert the upper edges for these vertices. The resulting 2-3 tree

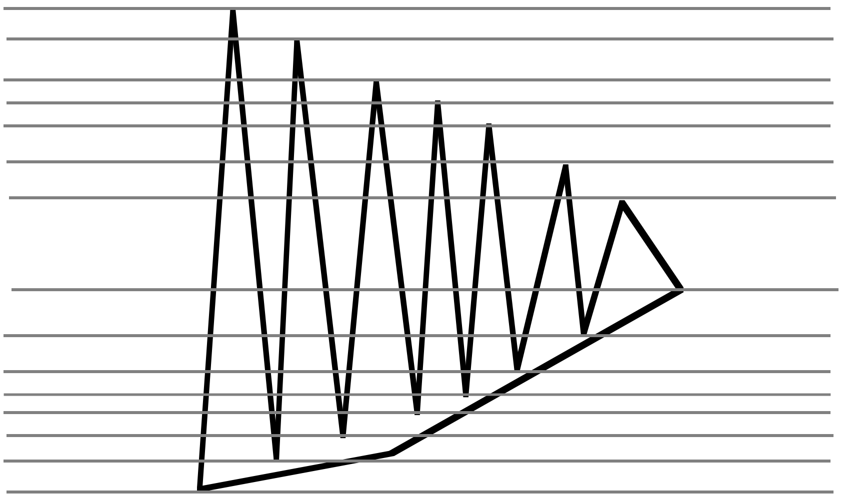

Figure 6.1: A PSLG containing $\Omega(n^2)$ edge segments

then represents exactly the list of the edge segments, ordered from left to right, of the next slab. We print the leaves of each 2-3 tree, from left to right, and obtain the lists l_i for $i = 1, \cdots, n+1$. The following is the algorithm of the preprocessing of the slab method. For simplicity, we assume that no two vertices of the PSLG G have the same y-coordinate. If this condition is not satisfied, we either rotate the coordinate system slightly, or make a straightforward modification on the algorithm.

```
Algorithm     PREPROCESS (G)

      Given:    a PSLG G, represented by a DCEL
      Output:   the lists L and l_i for i = 1, ..., n+1

      BEGIN
      1.   Sort the vertices of G by increasing y-coordinate.
           Let the sorted list of the vertices of G be
                     { v_1, v_2, ..., v_n }
           Then construct the list L;
           (each slab L[i] of L, i = 1, ..., n+1, is
```

```
     associated with two vertices v_{i-1} and v_i
     of G, one is on the lower boundary and the other
     is on the upper boundary of the slab, where v_0
     has a very large negative y-coordinate while v_(n+1)
     has a very large positive y-coordinate.)
2.   For slab L[1], construct an empty 2-3 tree T_1.
     The list l_1 for the slab L[1] is also empty;
     Set k = 2;
3.   Look at the vertex v_(k-1), delete all lower edges
     of the vertex v_(k-1) from the tree T_(k-1) and
     insert all upper edges of the vertex v_(k-1) into
     the tree T_(k-1).  The resulting tree T_k is the
     2-3 tree for the slab L[k].
4.   Read the leaves of the 2-3 tree T_k, from left
     to right, and produce the list l_k;
5.   If k <= n then k = k + 1 and go back to Step 3;
END.
```

It is obvious that the above algorithm is correct. Now we analyze the algorithm. Step 1 takes time $O(n \log n)$ by using any optimal sorting algorithm. Consider the loop of Step 3 - Step 5. Since each slab contains at most $O(n)$ edge segments, all 2-3 trees T_k, $k = 1, \cdots, n+1$, have size $O(n)$. Consequently, the depth of each 2-3 tree T_k is bounded by $O(\log n)$. Therefore, each edge insertion and edge deletion can be done in time $O(\log n)$. Each edge of G is inserted exactly once into some 2-3 tree T_k then deleted exactly once from some other 2-3 tree $T_{k'}$. Moreover, given the vertex v_{k-1}, all the edges of G incident to v_{k-1} can be found by an algorithm called TRACE-VERTEX, which is similar to the algorithm TRACE-REGION in Section 1.4, in time proportional to the number of these edges (we suppose that the PSLG G is represented by a DCEL). Thus each of the lower edges and upper edges of v_{k-1} in Step 3 can be found in constant time. Therefore, the time of insertion and deletion of an edge of the PSLG G is bounded by $O(\log n)$ for the whole algorithm. Consequently, the total time of the algorithm taken by Step 3 is bounded by $O(n \log n)$ since G contains $O(n)$ edges. Now to read the leaves of the 2-3 tree T_k from left to right, we can use, say, depth first search on the tree T_k. (For the discussion of depth first search of a graph, see [2].) The time to read the tree T_k and then to produce the list l_k thus is bounded by some constant times the number of nodes in the tree T_k, which is bounded by $O(n)$. Therefore, the total time of the algorithm taken by Step 4 is bounded by $O(n^2)$. This cannot be improved as we have seen, some PSLG contain $\Omega(n^2)$ many edge segments.

Thus the time complexity of the algorithm *PREPROCESS* is bounded by

$$O(n \log n) + O(n \log n) + O(n^2) = O(n^2)$$

We conclude with the following theorem.

Theorem 6.2.2 *Using the slab method solving the point location problem, the preprocessing time is $O(n^2)$, the storage is $O(n^2)$, and the query time is $O(\log n)$.*

6.2.3 Refinement method I: on rectangles

The refinement method for point location problem is a variety of the prune and search technique. To motivate the refinement method for the point location problem, we first consider a class of simple PSLGs.

Let $X = (x_1, x_2, \cdots, x_m)$ and $Y = (y_1, y_2, \cdots, y_m)$ be two lists of m real numbers sorted in increasing order. Define a PSLG G as follows: G has $n = m^2$ vertices $v_{i,j} = (x_i, y_j)$, $i, j = 1, 2, \cdots, m$. For $2 \leq i, j \leq m - 1$, the vertex $v_{i,j}$ is adjacent to exactly four vertices $v_{i,j-1}$, $v_{i,j+1}$, $v_{i-1,j}$ and $v_{i+1,j}$. The vertex $v_{1,j}$ (resp. $v_{n,j}$) for $2 \leq j \leq m-1$ is adjacent to the vertices $v_{1,j-1}$, $v_{1,j+1}$ and $v_{2,j}$ (resp. $v_{n,j-1}$, $v_{n,j+1}$, and $v_{n-1,j}$). The vertex $v_{i,1}$ (resp. $v_{i,n}$) for $2 \leq i \leq m-1$ is adjacent to the vertices $v_{i-1,1}$, $v_{i+1,1}$ and $v_{i,2}$ (resp. $v_{i-1,n}$, $v_{i+1,n}$, and $v_{i,n-1}$). Finally, the vertex $v_{1,1}$ is adjacent to $v_{1,2}$ and $v_{2,1}$, the vertex $v_{1,n}$ is adjacent to $v_{1,n-1}$ and $v_{2,n}$, the vertex $v_{n,1}$ is adjacent to $v_{n,2}$ and $v_{n-1,1}$, and the vertex $v_{n,n}$ is adjacent to $v_{n-1,n}$ and $v_{n,n-1}$. Call the whole PSLG an $m \times m$ *rectangle* with the index sets X and Y. Figure 6.2 pictures a 5×5 rectangle.

Clearly, the point location problem on this kind of PSLGs can be simply done by doing two binary searchings, one on the list X and the other on the list Y. Alternatively, we can also locate a query point $p_0 = (x_0, y_0)$ in the following way: compare the value x_0 with the middle number $x_{m/2}$ in the list X and determine that the point p_0 is in the left rectangle R_l bounded by the vertices $v_{1,1}$, $v_{m/2,1}$, $v_{m/2,m}$ and $v_{1,m}$ or in the right rectangle R_r bounded by the vertices $v_{m/2,1}$, $v_{m,1}$, $v_{m,m}$ and $v_{m/2,m}$. Suppose that p_0 is in the left rectangle R_l. Now we compare the value y_0 with the middle number $y_{m/2}$ in the list Y to determine that the point p_0 is in the upper rectangle $R_{l,u}$ bounded by the vertices $v_{1,m/2}$, $v_{m/2,m/2}$, $v_{m/2,m}$ and $v_{1,m}$ or in the lower rectangle $R_{l,l}$ bounded by the vertices $v_{1,1}$, $v_{m/2,1}$, $v_{m/2,m/2}$ and

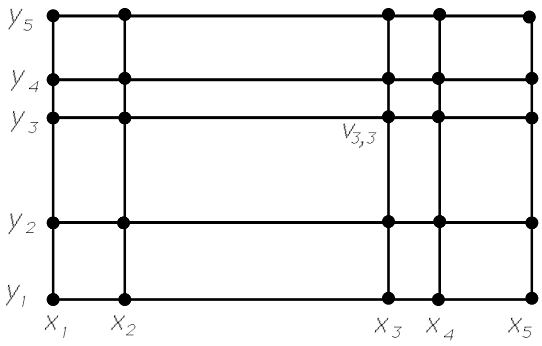

Figure 6.2: A 5×5 rectangle with the center vertex $v_{3,3}$

$v_{1,m/2}$. Thus two comparisons restrict the point p_0 to an $(m/2) \times (m/2)$ subrectangle. Now we recursively work on the $(m/2) \times (m/2)$ rectangle.

Let R_m be an $m \times m$ rectangle with index sets

$$X = (x_1, \cdots, x_m) \qquad \text{and} \qquad Y = (y_1, \cdots, y_m)$$

A vertex $v_{i,j}$ is the *center vertex* of R_m if $i = j = m_0 = \lceil m/2 \rceil$. Note that to determine which $(m/2) \times (m/2)$ subrectangle a query point p_0 is in, we only need two values from the index sets: the middle number x_{m_0} in the list X and the middle number y_{m_0} in the list Y. But (x_{m_0}, y_{m_0}) is just the coordinates of the center vertex v_{m_0,m_0} of the rectangle R_m. Therefore, the $m \times m$ rectangle R_m can be organized in the following way: construct a tree T_m whose root N_0 is attached with the center vertex v_{m_0,m_0} of T_m. There are four children of the root N_0, corresponding to the four $(m/2) \times (m/2)$ subrectangles of R_m obtained by dividing R_m by a horizontal line and a vertical line passing through the center vertex v_{m_0,m_0}. The algorithm of constructing this tree is presented as follows:

```
Algorithm    CONSTRUCTING-TREE(R_m)

     Given:   a PSLG R_m that is an m by m rectangle
     Output:  a hierarchy tree T_m

     BEGIN
     1.   If R_m is a 2 by 2 rectangle, then R_m is a
```

```
     single region.  Create a tree node for R_m
     and attach the name of the region to the
     node;  STOP.
2.   { R_m is not a single region. }
     Create a node N_m for R_m, attach the center
     vertex v_(m_0,m_0) of R_m to N_m.  Draw a
     horizontal line and a vertical line passing
     through the center vertex v_(m_0,m_0) that divides
     the rectangle R_m into four (m/2) by (m/2)
     subrectangles;
3.   Recursively call the algorithm CONSTRUCTING-TREE
     on the four (m/2) by (m/2) subrectangles.  Let
     the resulting four trees be T_1, T_2, T_3, and
     T_4;
4.   Let T_1, T_2, T_3 and T_4 be the children of the
     node N_m;
END.
```

Each leaf in the tree T_m corresponds uniquely to a region of the rectangle R_m, and each internal node of the tree T_m corresponds to a vertex of R_m. Since there are $n = m^2$ vertices and $O(n)$ regions in the rectangle R_m, we conclude that the number of nodes of the tree T_m is bounded by $O(n)$. Moreover, since we spend constant time to create a node in the tree T_m, the total time of constructing the tree T_m is bounded by $O(n)$.

Since the tree T_m is very balanced: each internal node of T_m has exactly four children, and since the tree T_m has $O(n)$ nodes, we conclude that the depth of the tree T_m is bounded by $O(\log n)$.

This is the preprocessing for the point location problem on rectangles.

Given a query point p_0, it is easy to locate p_0 in an $m \times m$ rectangle R_m with the help of the tree T_m, as shown by the following algorithm.

```
Algorithm      LOCATING (p_0)

      Given:    a query point p_0 and the hierarchy
                tree T_m
      Output:   the region that contains the point p_0

      BEGIN
      1.   First use the four corner vertices v_(1,1),
           v_(m,1), v_(m,m) and v_(1,m) to determine
```

```
     if p_0 is contained in the rectangle R_m.
     If p_0 is out R_m, report so and STOP.
2.   { p_0 is inside R_m. }
     Starting by the root N_0 of the tree T_m,
     compare p_0 with the center point of R_m
     to find a child of N_0 that corresponds to
     an (m/2) by (m/2) rectangle R_(m/2) containing
     the point p_0;
3.   Recursively search p_0 in the rectangle R_(m/2);
END.
```

It is clear that the algorithm LOCATING runs in time $O(\log n)$ for each query point p_0.

Therefore, the point location problem on rectangles can be solved by $O(n)$ preprocessing time, $O(n)$ storage, and $O(\log n)$ query time.

Let us summarize the above idea: We first locate the query point into a large $m \times m$ rectangle R_m, then we refine the rectangle R_m into four smaller $(m/2) \times (m/2)$ rectangles by dividing the rectangle R_m by a horizontal line and a vertical line passing through the center vertex of R_m, then we recursively locate the point p_0 in one of these smaller rectangles.

Two properties we have used heavily in this method:

- A father and its children have the same geometric shape (here are rectangles), so the recursive call is effective.
- Each father has only constant many children so that in constant time we can move one level down in the search tree T_m.

6.2.4 Refinement method II: on general PSLGs

Now we try to extend the idea in the last section to solve the point location problem on general PSLGs. The algorithm discussed in this section is due to Kirkpatrick [12].

All the geometric objects in the refinement method on rectangles are simple rectangles. Moreover, it is easy to refine a rectangle into four smaller rectangles by a horizontal line and a vertical line. However, in a general PSLG, a region can be an arbitrary simple polygon, and it is not guaranteed that a simple polygon can be refined into smaller polygons of the same shape. Therefore, we must first fix a geometric shape we are going to use. It is natural to consider the simplest geometric shape, the triangles. However, not every PSLG can be obtained by refining a triangle. Extra care should be taken to make our idea work.

A PSLG G is *completely triangulated* if G is connected and the boundary of every region of G (including the unbounded region) is a triangle. We first discuss how to convert a general PSLG into a completely triangulated PSLG.

Given a general PSLG G which is not completely triangulated. We first add a big triangle $\triangle$ that encloses the whole G. This can be done by first scanning the vertices of G to find the minimum x_0 of the x-coordinates of the vertices of G, the minimum y_0 of the y-coordinates of the vertices of G, and the maximum z_0 of the values $x + y$ where (x, y) is a vertex of G. Now the triangle formed by the horizontal line $l_h : y = y_0 - 1$, the vertical line $l_v : x = x_0 - 1$, and the line $l : x + y = z_0 + 1$ will enclose the whole PSLG G. Let the PSLG consisting of G and $\triangle$ be G'. Now triangulating G' gives us a completely triangulated PSLG G_0.

Delete an internal vertex v from G_0 and let the resulting PSLG be G_0'. If the vertex v has degree k in the PSLG G_0, then G_0' has all its regions being triangles except one region that is a k-gon P_k. To make G_0' have the same geometric property as G_0, we retriangulate the k-gon P_k of G_0'. Of course, we can perform the above operation on other vertices of G_0 as well provided that the vertices we delete are not adjacent to each other in G_0. Let G_1 be the new completely triangulated PSLG obtained by this kind of deleting-vertex-then-retriangulating operation on a set of non-adjacent vertices of G_0. All regions of G_0 are regions of G_1 except those that disappear when we delete the vertices of G_0 (call these regions *old triangles*). All regions of G_1 are regions of G_0 except those that are created when we retriangulate the non-triangle regions resulting from deleting vertices in G_0 (call these regions *new triangles*). We set a pointer from a new triangle to an old triangle if their intersection is not empty. Note that the new PSLG G_1 has less vertices than the old PSLG G_0. The old PSLG G_0 thus can be regarded as a refinement of the new PSLG G_1.

This solves our first problem: the inverse of the deleting-vertex-then-retriangulating operation refines a completely triangulated PSLG G_1 into a larger completely triangulated PSLG G_0 (here “larger” means containing more vertices and more regions. In this sense, the regions of G_0 are “smaller” than that of G_1).

The query algorithm now goes as follows: suppose that we have located a query point p_0 in a new triangle $\triangle$, then we look at all old triangles that intersect the new triangle $\triangle$ and determine which old triangle contains the query point p_0.

However, how many old triangles intersect the new triangle $\triangle$? And how many completely triangulated PSLGs should we go through in order to locate the query point p_0 in a triangle of the original PSLG? In order to achieve an $O(\log n)$ query time, we must move from one completely triangulated PSLG to another completely triangulated PSLG in constant time, and go through at most $O(\log n)$ completely triangulated PSLGs to reach the original completely triangulated PSLG. For this purpose, we require that the vertices to be deleted from one completely triangulated PSLG in order to construct the next PSLG satisfy the following conditions:

1. All these vertices should be internal vertices, that is, they are not the three hull vertices of the completely triangulated PSLG.
2. No two of these vertices are adjacent.
3. The degree of these vertices is small.
4. There are enough vertices of the current completely triangulated PSLG to be deleted.

The first condition makes all our PSLGs completely triangulated. The second condition ensures that the relationship between new triangles and old triangles simple, that is, an old triangle incident to a deleted vertex v can only intersect those new triangles that are obtained by retriangulating the simple polygon resulting from deleting the vertex v from G_0. The second and the third conditions together ensure that each old triangle intersects very few new triangles, and each new triangle intersects very few old triangles. Finally, the fourth condition ensures that the rate of the size-shrinking of the completely triangulated PSLGs is fast so that a query point goes through very few completely triangulated PSLGs to reach the original PSLG.

The existence of a set of vertices of a completely triangulated PSLG that satisfies all conditions above is proved by a pure combinatorial counting technique.

Let G be a completely triangulated PSLG. Suppose that the set of vertices, the set of edges, and the set of regions of G are V, E, and F, respectively. Since G is a planar imbedding, by Euler's formula:

$$|V| - |E| + |F| = 2$$

Since G is a completely triangulated PSLG, each region of G has exactly 3 boundary edges. On the other hand, each edge of G is a boundary edge for exactly two regions. This gives us

$$3|F| = 2|E|$$

Replacing $|F|$ in Euler's formula by $\frac{2}{3}|E|$, we obtain

$$|E| = 3|V| - 6 < 3|V|$$

Let $deg(v)$ be the degree of the vertex v, then each vertex v of G incident to exactly $deg(v)$ edge-ends. On the other hand, each edge has exactly two edge-ends, thus we have

$$\sum_{v \text{ is a vertex of } G} deg(v) = 2|E| < 6|V|$$

Therefore, at least half of the vertices of G have degree less than 12. If we exclude the three hull vertices of G, then there are at least $|V|/2-3$ vertices of G that have degree less than 12. For each vertex of degree less than 12, there are at most 11 adjacent vertices, thus there are at least $(|V|/2-3)/12$ vertices of degree less than 12 in G such that no two of them are adjacent. When $|V| \geq 48$, we have $(|V|/2-3)/12 \geq |V|/48$. Therefore, for an arbitrary completely triangulated PSLG G with n vertices, with $n \geq 48$, we can find at least $n/48$ internal non-adjacent vertices of G of degree less than 12.

This analysis gives us the following algorithm to construct a searching hierarchy T_G.

```
Algorithm     CONSTRUCT-HIERARCHY(G)

      Given:     a general PSLG G
      Output:    a searching hierarchy T_G for the
                 point location problem on G

      BEGIN
      1.   Add an enclosing triangle that contains the whole
           G, then triangulate the resulting PSLG.  Let the
           completely triangulated PSLG be G_0;
      2.   Using the TRACE-REGION algorithm in Section 1.4 to
           find all triangles of G_0.  For each triangle of
           G_0, create a node in level 0 in the hierarchy T_G;
      3.   Set k = 0;
      4.   Suppose that the PSLG G_k contains n_k vertices.
           Find at least (n_k)/48 internal non-adjacent
```

```
     vertices of G_k that have degree less than 12;
5.   For each vertex v found in Step 4, delete v from
     G_k, and retriangulate the simple polygon resulting
     from this deletion.  For each new triangle obtained
     from this retriangulation, create a node in level
     k+1 of the hierarchy T_G and set a pointer from
     this node in the hierarchy T_G to a node corres-
     ponding to an old triangle incident to the vertex
     v in G_k if the intersection of the old triangle
     and the new triangle is not empty.
6.   Let the resulting completely triangulated PSLG be
     G_(k+1), then set k = k + 1.  If the PSLG has
     more than 48 vertices, go back to Step 4.
END.
```

We analyze the algorithm of constructing the hierarchy. Suppose that the number of vertices of G_0 is n, which is three more than that of the original PSLG G, and that each completely triangulated PSLG G_k is represented by a doubly-connected edge list (DCEL). As we discussed before, It takes $O(n)$ time to construct an enclosing triangle. Then the triangulation takes time $O(n \log n)$ if the PSLG is a general PSLG, or takes time $O(n)$ if the PSLG is connected (triangulating a connected PSLG in linear time is a recent breakthrough due to Chazelle [8]). Therefore, Step 1 of the algorithm takes time $O(n \log n)$ for a general PSLG G and takes time $O(n)$ for a connected PSLG G.

The TRACE-REGION algorithm takes time $O(n)$ to find all regions, therefore, Step 2 of the algorithm takes time $O(n)$.

By the analysis given above, each PSLG G_k contains at least $n_k/48$ internal non-adjacent vertices of degree less than 12. To find these vertices of G_k, we simply scan the DCEL for G_k (using a TRACE-VERTEX algorithm that is similar to the algorithm TRACE-REGION), whenever we find a vertex v of degree less than 12, we take v and mark all vertices adjacent to v "unusable". We scan the list of vertices of G_k and ignore those "unusable" vertices. In this way, by the analysis we gave above, we can find at least $n_k/48$ internal non-adjacent vertices of degree less than 12. In this process, we scan each vertex of G_k at most once and scan each edge of G_k at most twice. Therefore, Step 4 of the algorithm takes time $O(n_k)$ for the PSLG G_k.

For each vertex v found in Step 4, since the degree of v is less than 12, there are at most 11 triangles incident to v. Moreover, deleting v results in a simple polygon of at most 11 vertices since v has degree less than 12, so at most 9 new triangles are created when we retriangulate the simple polygon. Consequently, each new triangle intersects at most 11 old triangles and each old triangle intersects at most 9 new triangles. Therefore, each node of a new triangle has at most 11 pointers to nodes of old triangles, and a node for a new triangle together with its pointers to the old triangles can be created in constant time. So to produce the level $k+1$ in the hierarchy T_G takes time proportional to the number of regions in the PSLG G_{k+1}, which is bounded by $O(n_{k+1})$ where n_{k+1} is the number of vertices of G_{k+1}. It is also easy to see that constructing the DCEL for the PSLG G_{k+1} from the DCEL for the PSLG G_k also takes time $O(n_{k+1})$.

Therefore, the total time used in Step 4 - Step 6 to build up the hierarchy T_G is bounded by

$$O(n_0) + O(n_1) + \cdots + O(n_h) = O(n_0 + n_1 + \cdots + n_h)$$

if the hierarchy T_G has $h+1$ levels.

$n_0 = n$. Since G_1 is obtained from G_0 by deleting at least $n_0/48$ vertices, so we have $n_1 \le (47/48)n$. A simple induction proves that $n_k \le (47/48)^k n$ for all $k \le 1$. Therefore,

$$\begin{array}{ll} & O(n_0 + n_1 + \cdots + n_h) \\ \le & O(n + (47/48)n + \cdots + (47/48)^h n) \\ < & O(n + (47/48)n + \cdots + (47/48)^h n + \cdots) \\ = & O(\frac{n}{1-(47/48)}) \\ = & O(n) \end{array}$$

That is, the total time to build up the hierarchy T_G for the completely triangulated PSLG G_0 is bounded by $O(n)$. Consequently, the hierarchy T_G contains $O(n)$ nodes thus can be stored in space $O(n)$.

Now the searching algorithm for a query point in the hierarchy T_G is straightforward.

```
Algorithm     LOCATING (p_0)

      Given:    a query point p_0 and the hierarchy structure
                T_G for a PSLG G
      Output:   the region of G that contains the point p_0

      BEGIN
```

```
1.    In the highest level of the hierarchy T_G, locate
      the point p_0 into one of the triangles;
2.    Suppose p_0 is in a node N_0 of the hierarchy T_G.
      Check each triangle whose node in the hierarchy
      T_G is pointed by a pointer from N_0 to find a
      node N' whose triangle contains the point p_0;
3.    IF N' is at level 0, then we have located the
      point p_0 into a triangle in the original PSLG.
      ELSE let N_0 = N' and go back to Step 2;
END.
```

Since a point p_0 is contained in a new triangle after the retriangulation if and only if it is contained in some old triangle before the vertex deletion, the point p_0 is contained in the triangle corresponding to the node N_0 if and only if it is contained in a triangle whose corresponding node in the hierarchy T_G is pointed by a pointer from N_0. Therefore, the above algorithm *LOCATING*(p_0) correctly finds a triangle in the original PSLG that contains the point p_0. Since each pointer in the hierarchy T_G is always from a higher level to a lower level and each node in the hierarchy has at most 11 pointers, the searching time of the algorithm *LOCATING* is proportional to the depth of the hierarchy T_G. Let n_k be the number of vertices of the PSLG G_k, for $k = 0, 1, \cdots$, then as analyzed above, we have $n_k \leq (47/47)^k n$, and we stop producing more levels when we reach $n_k \leq 48$. This gives us immediately

$$\text{The number of levels in the hierarchy } T_G = O(\log n)$$

We summarize the above results in the following theorems.

Theorem 6.2.3 *For a general PSLG G, the point location problem can be solved with $O(n \log n)$ preprocessing time, $O(n)$ space, and $O(\log n)$ query time.*

Theorem 6.2.4 *For a connected PSLG G, the point location problem can be solved with $O(n)$ preprocessing time, $O(n)$ space, and $O(\log n)$ query time.*

6.3 Exercises

1. Based on the idea described in the text, design a linear time algorithm that finds the median given a set of numbers.

2. Design an algorithm to solve the following problem: given a set S of N points in the plane, with preprocessing, decide for a query point if the point is in a triangle whose three vertices are points of S. If it is, output the three vertices of the triangle (if there are more than one such triangles, pick any one of them). Analyze your algorithm for query time, preprocessing time, and space.

3. Solve the Point Location Problem for the set of *PSLG*s whose faces are of size at most 5. What are the query time, preprocessing time and space of your algorithm?

4. Given a PSLG G such that the number of intersection points of any vertical line and G is bounded by 50. Moreover, a sorted list of the vertices of the PSLG G is also given. Discuss the preprocessing time, space, and query time of the point location problem on G.

5. A *k-monotone polygon with respect to a line l* is a simple polygon which can be decomposed inot k chains monotone with respect to the line l. Let k be a fixed constant. Design an algorithm to solve Point Location Problem for k-monotone polygons, i.e., given a k-monotone polygon P, with preprocessing, determine if a query point is internal to P. Analyze your algorithm for query time, preprocessing time and space.

6. Given two sets of points $S_p = \{p_1, \cdots, p_n\}$ and $S_q = \{q_1, \cdots, q_m\}$. For each point in S_q, find the closest point in S_p. Solve this problem for the case

   (1). m is much larger than n, say $m = 2^n$;

   (2). m is much smaller than n, say $m = \log\log n$.

   Do you use the same algorithm to solve the problem for both cases or you use different algorithms for the two cases? Give a detailed analysis for yours algorithm(s).

7. A point p is said to be *dominated* by a point q if both x- and y-coordinates of p are no greater than those of q, respectively. Solve the following problem: given a set S of n points in the plane, with preprocessing allowed, for each query point q, find the number of points in S dominated by q. That are the preprocessing time, storage, and query time of your algorithm?

8. Suppose that we can construct the kth order Voronoi diagram in time $O(k^2 N \log N)$. Analyze the query time, preprocessing time, and the storage for the k-Nearest Points Problem.

9. Let $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ be two points in the plane. We say that point p_1 *dominates* point p_2 if $x_1 \geq x_2$ and $y_1 \geq y_2$.

   Let S be a set of points in the plane. A point $p \in S$ is a *maximal element* if p is not dominated by any other point in S.

   Solve the following problem:

   Given a set of n points in the plane, let k denote the number of maximal elements in this set. Design a divide-and-conquer algorithm of time $O(n \log k)$ for finding these maximal elements. Prove the correctness of your algorithm.

Chapter 7

Reductions

Let P and P' be two problems. We say that the problem P can be *reduced* to the problem P' in time $O(t(n))$, express it as

$$P \propto_{t(n)} P'$$

if there is an algorithm $\mathcal{T}$ solving the problem P in the following way.

1. For any input x of size n to the problem P, convert x in time $O(t(n))$ into an input x' to the problem P';

2. Call a subroutine to solve the problem P' on input x';

3. Convert in time $O(t(n))$ the solution to the problem P' on input x' into a solution to the problem P on input x.

Note that the subroutine in Step 2 that solves the problem P' is unspecified. If the problem P' can be solved efficiently, then the problem P can also be solved efficiently, as explained by the following theorem.

Lemma 7.0.1 *Suppose that a problem P is reduced to a problem P' in time $O(t(n))$*

$$P \propto_{t(n)} P'$$

and that the problem P' can be solved in time $O(T(n))$. Then the problem P can be solved in time $O(t(n) + T(O(t(n))))$.

PROOF. Suppose that the algorithm $\mathcal{T}$ gives a $O(t(n))$-time reduction from the problem P to the problem P', and suppose that an algorithm $\mathcal{A}'$

solves the problem P' in time $O(T(n))$. The problem P can be solved by the algorithm $\mathcal{T}$, in whose Step 2, calling a subroutine to solve the problem P' on input x', we use the algorithm $\mathcal{A}'$.

To analyze the algorithm $\mathcal{T}$, note that Step 1 and Step 3 of the algorithm $\mathcal{T}$ take time $O(t(n))$, as we have assumed. Since Step 1 takes time $O(t(n))$, the size of x' is also bounded by $O(t(n))$. Therefore, in Step 2 of the algorithm $\mathcal{T}$, the algorithm $\mathcal{A}'$ of time complexity $O(T(n))$ on inputs of size n takes time $O(T(O(t(n))))$ on input x', which is of size $O(t(n))$. This concludes that the running time of the algorithm $\mathcal{T}$ is bounded by

$$O(t(n)) + O(T(O(t(n)))) = O(t(n) + T(O(t(n))))$$

$\square$

The reduction technique plays an important role in the study of complexity of geometric problems, both for deriving lower bounds and for designing efficient algorithms. In this chapter, we will study how to use this technique to design efficient geometric algorithms, and in the next chapter, we will explain how we use this technique to derive lower bounds for geometric problems.

We close this introductory section by the following corollary, which will be heavily used in our discussion.

Corollary 7.0.2 *Suppose that a problem P is reduced to a problem P' in linear time*

$$P \propto_n P'$$

If the problem P' can be solved by an algorithm in time $O(T(n))$, with $T(n) = \Omega(n)$ and $T(O(n)) = O(T(n))$, then the problem P can also be solved in time $O(T(n))$.

PROOF. As shown in Lemma 7.0.1, the problem P can be solved by the algorithm $\mathcal{T}$ in time $O(n + T(O(n)))$. By our assumption, $T(O(n)) = O(T(n))$. Moreover, $T(n) = \Omega(n)$. Therefore, the time complexity of the algorithm $\mathcal{T}$ in this special case is bounded by

$$O(n + O(T(n))) = O(T(n))$$

$\square$

Notice that most of the complexity functions $T(n)$ we use in this book, such as n, $n \log n$, n^k, and $n^k \log^h n$ satisfy the conditions $T(n) = \Omega(n)$ and $T(O(n)) = O(T(n))$.

7.1 Convex hull and sorting

Consider the algorithm of Graham Scan for constructing convex hulls of points in the plane. If a given set S of n points in the plane is sorted by x-coordinates, then the Graham Scan algorithm needs only linear time to construct the convex hull for S. In fact, it is not hard to see that

$$\text{CONVEX HULL} \propto_n \text{SORTING}$$

by the following argument. Given an instance of CONVEX HULL, which is a set S of n points in the plane, we can simply regard S as an instance of SORTING if we let the x-coordinate of a point p in S be the "key" of the point p. Therefore, we can simply translates instances of the problem CONVEX HULL to instances of the problem SORTING. Now the solution of SORTING on input S is a list of the points in S which is sorted by the x-coordinates. The generalized Graham Scan algorithm shows that with this solution to the SORTING, the convex hull *CH(S)* of the set S, which is the solution of CONVEX HULL on the input S, can be constructed in time $O(n)$.

It is interesting that we can prove that the problem SORTING can also be reduced to the problem CONVEX HULL in linear time.

Theorem 7.1.1

$$\textit{SORTING} \propto_n \textit{CONVEX HULL}$$

PROOF. Given a list L of n real numbers $x_1, x_2, \cdots, x_n$, which is an instance to the problem SORTING, we can suppose that all of them are non-negative since otherwise, we first scan the list and find the "most negative" number x, then add $-x$ to each given number to make all non-negative.

We first scan the list to find the largest number $x_{\max}$. Now for each number x_i in the given list L, we convert x_i into a point p_i in the plane such that the polar angle of p_i is $2\pi\frac{x_i}{x_{\max}}$ and the distance between p_i and the origin O is 1 (so the point p_i is on the unit circle). Let S be the set of all these n points $p_1, p_2, \cdots, p_n$ in the plane. The set S is an instance of the problem CONVEX HULL. Note that the set S can be obtained from the list L in time $O(n)$, since all we do is to scan the list L at most twice, then for each number x_i, we spend constant time to obtain the corresponding point p_i.

Since the unit circle itself is convex, the n points in the set S must be all on the convex hull $CH(S)$. Therefore, when the solution $CH(S)$ is returned back for the problem CONVEX HULL on the input S, we must get the hull vertices of S in counterclockwise ordering. If we suppose that we start with the point with the smallest polar angle, then the hull vertices must be given in increasing ordering of their polar angles. But since the polar angle of a point in the set S is proportional to the value of the corresponding number in the list L, the list of the polar angles of the points times $\frac{x_{\max}}{2\pi}$ given in counterclockwise ordering on *CH(S)* gives the sorted list of the numbers of the list L. Therefore, given the solution for the problem CONVEX HULL on the input S, we can obtain the solution for the problem SORTING on the input L, that is a sorted list of the n numbers of L, by first finding the point with the smallest polar angle, then scanning the convex hull *CH(S)* in counterclockwise ordering and multiplying the polar angle of each point by $\frac{x_{\max}}{2\pi}$. This can obviously done in linear time. □

Let P and P' be two problems, and let $t(n)$ be a function of n. If we have both

$$P \propto_{t(n)} P' \qquad \text{and} \qquad P' \propto_{t(n)} P$$

then we say that the problems P and P' are *equivalently complex* up to a $t(n)$-time reduction, and express as

$$P \equiv_{t(n)} P'$$

When two problems are equivalently complex up to a linear time reduction, then if one of them can be solved in time $O(T(n))$, where $T(n) = \Omega(t(n))$ and $T(O(n)) = O(T(n)$, then by Corollary 7.0.2, the other can also be solved in time $O(T(n))$.

By the above discussions, we have already shown

$$\text{SORTING} \equiv_n \text{CONVEX HULL}$$

In fact, construction of convex hulls for sets of points in the plane is a generalization of sorting. In sorting n numbers, we are asked to find the ordering of a set of points in the real line, while in constructing a convex hull, we are asked to find the ordering of polar angles, relative to an interior point of the convex hull, of the "extreme points". The difference is that in sorting, every given number will appear in the final sorted list, while in constructing a convex hull, we also have to make the decision whether a given point is a non-extreme point, and if yes, exclude it from the final output list. On the other hand, as we have discussed in the section, sorting is not easier at all than constructing convex hulls for points in the plane.

7.2 Closest pair and all nearest neighbor

According to the definition of the Voronoi diagram (a partition of the plane into regions such that each region is the locus of points closer to a point of the set S than to any other point of S), it is not surprising that the problems CLOSEST-PAIR and ALL-NEAREST-NEIGHBOR can be solved efficiently through the Voronoi diagram. Recall that the problem CLOSEST-PAIR is to find the closest pair in a set of n points in the plane, while the problem ALL-NEAREST-NEIGHBOR is that given a set S of n points in the plane, for each point of S, find the nearest neighbor in S. Finally, let VORONOI-DIAGRAM denote the problem of constructing the Voronoi diagram for a given set of n points in the plane.

Theorem 7.2.1

$$\textit{ALL-NEAREST-NEIGHBOR} \propto_n \textit{VORONOI-DIAGRAM}$$

PROOF. Suppose that a set S of n points in the plane is an input to the problem ALL-NEAREST-NEIGHBOR, we pass the input S to VORONOI-DIAGRAM directly. The solution of VORONOI-DIAGRAM on the input S will be the Voronoi diagram *Vor(S)* of the set S. By Lemma 5.2.2, for each point p_i in the set S, the nearest neighbor of p_i in S defines a non-degenerate Voronoi edge for the Voronoi polygon V_i of p_i. Therefore, by tracing the boundary of the Voronoi polygon of each point in the set S, we can find the nearest point for each point in the set S. This is the solution of the problem ALL-NEAREST-NEIGHBOR. Given the Voronoi diagram *Vor(S)* of the set S, each Voronoi polygon can be traced by the algorithm TRACE-REGION given in Section 1.4 in time proportional to the number of edges on the boundary of the polygon. Since each Voronoi edge is on the boundary of exactly two Voronoi polygons, the sum of boundary edges of all Voronoi polygons in *Vor(S)* equals twice the number of edges in the Voronoi diagram *Vor(S)*. We conclude that tracing all Voronoi polygons of the Voronoi diagram *Vor(S)*, thus finding the nearest neighbor for each point in the set S when the Voronoi diagram *Vor(S)* is given, takes time proportional to the number of edges of *Vor(S)*, that is of order $O(n)$ since the Voronoi diagram is a planar graph. □

Since the Voronoi diagram of a set of n points can be constructed in time $O(n \log n)$ (Theorem 5.3.6), by Corollary 7.0.2, we obtain

Corollary 7.2.2 *The problem ALL-NEAREST-NEIGHBOR can be solved in time* $O(n \log n)$*.*

It is easy to see that given a set S of n points in the plane, the solution of the problem CLOSEST-PAIR can be obtained from the solution of the problem ALL-NEAREST-NEIGHBORS in linear time, that is,

$$\textit{CLOSEST-PAIR} \propto_n \textit{ALL-NEAREST-NEIGHBOR}$$

by simply computing the distance between each point and its nearest neighbor, then taking the point that has the shortest distance to its nearest neighbor. By Corollary 7.2.2, the problem ALL-NEAREST-NEIGHBOR can be solved in time $O(n \log n)$. Therefore by Corollary 7.0.2, we have

Corollary 7.2.3 *The problem CLOSEST-PAIR can be solved in time* $O(n \log n)$*.*

7.3 Triangulation

Given a Voronoi diagram *Vor(S)* for the set S of n points in the plane. We draw a segment $\overline{p_i p_j}$ for each pair of points p_i and p_j that define a Voronoi edge in *Vor(S)*. Let $D(S)$ be the collection of these segments, which is called the *straight-line dual* of the Voronoi diagram *Vor(S)*.

We prove that the straight-line dual $D(S)$ of the Voronoi diagram *Vor(S)* is a triangulation of the set S. For this, we must show that the straight-line dual $D(S)$ partitions the convex hull *CH(S)* of the set S into triangles such that 1) no two triangles overlap in the interior, and 2) every point in the convex hull *CH(S)* (more precisely, every point in the area bounded by the convex hull *CH(S)*) must be contained in at least one such triangles.

Each Voronoi vertex v is incident to exactly three Voronoi edges e_1, e_2, and e_3, and exactly three Voronoi polygons V_1, V_2, and V_3 of three points p_1, p_2, and p_3 in the set S. Each of the edges e_1, e_2, and e_3 is defined by a pair of the points p_1, p_2, and p_3. Therefore, the segments $\overline{p_1 p_2}$, $\overline{p_2 p_3}$, and $\overline{p_3 p_1}$ are all in the straight-line dual $D(S)$ of *Vor(S)*. Thus, each Voronoi vertex v corresponds to a triangle $\triangle p_1 p_2 p_3$ in the straight-line dual $D(S)$. Denote by $\triangle(v)$ the triangle $\triangle p_1 p_2 p_3$. On the other hand, since a Voronoi

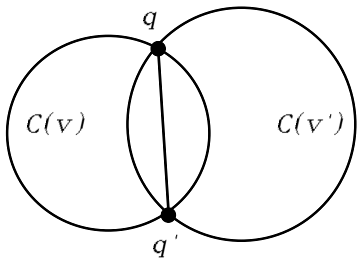

Figure 7.1: Two circumcircles intersect at q and q'

edge is incident on two Voronoi vertices, each segment in the straight-line dual $D(S)$ is a boundary edge of two such triangles $\triangle(v)$ and $\triangle(v')$, where v and v' are two Voronoi vertices in *Vor(S)*.

Lemma 7.3.1 *No two triangles $\triangle(v)$ and $\triangle(v')$ overlap in the interior, where v and v' are two different Voronoi vertices in Vor(S).*

PROOF. Let $\triangle(v)$ and $\triangle(v')$ be two arbitrary triangles in the straight-line dual $D(S)$ of the Voronoi diagram *Vor(S)*. Let $C(v)$ and $C(v')$ be the circumcircles of the triangles $\triangle(v)$ and $\triangle(v')$, respectively. If the circumcircles $C(v)$ and $C(v')$ do not overlap in the interior, then of course the triangles $\triangle(v)$ and $\triangle(v')$ do not overlap in the interior. So we suppose that $C(v)$ and $C(v')$ do overlap in the interior. Note that each of the circumcircles $C(v)$ and $C(v')$ contains exactly three points in the set S on its boundary, and by Lemma 5.2.3, no point of S is contained in the interior of $C(v)$ or $C(v')$. Moreover, $C(v)$ and $C(v')$ cannot be coincide otherwise at least four points in the set S would be co-circular. Moreover, no one of the circles $C(v)$ and $C(v')$ can be entirely contained in the other, since otherwise some point of the set S would be contained in the interior of $C(v)$ or $C(v')$, contradicting to Lemma 5.2.3. So the boundaries of the circumcircles $C(v)$ and $C(v')$ must intersect at exactly two points q and q'. See Figure 7.1.

The two points q and q' partition the circle $C(v)$ into two disjoint curves, one is entirely contained in the circle $C(v')$ and the other is completely outside the circle $C(v')$. No vertex of the triangle $\triangle(v)$ can be on the curve of $C(v)$ that is entirely contained in the circle $C(v')$ otherwise that vertex, which is a point in the set S, would be in the interior of the circle $C(v')$, contradicting Lemma 5.2.3. Thus the three vertices of $\triangle(v)$ must be on the curve of $C(v)$ that is outside $C(v')$. Similarly, the three vertices of $\triangle(v')$ are

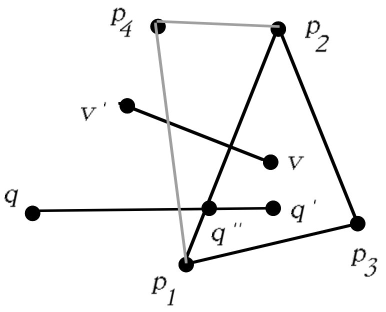

Figure 7.2: A point q outside all triangles

on the curve of $C(v')$ that is outside $C(v)$. Therefore, the three vertices of the triangle $\triangle(v)$ and the three vertices of the triangle $\triangle(v')$ must be separated by the segment $\overline{qq'}$, so the triangles $\triangle(v)$ and $\triangle(v')$ do not overlap in the interior. $\square$

Lemma 7.3.2 *Every point in the convex hull CH(S) is contained in some triangle $\triangle(v)$ for some Voronoi vertex v of Vor(S).*

PROOF. Suppose that the lemma is not true and that some point q in *CH(S)* is not contained in any such triangles. Then we can find a triangle $\triangle(v)$, where v is a Voronoi vertex of *Vor(S)*, and an interior point q' in the triangle $\triangle(v)$ such that the segment $\overline{qq'}$ intersects no triangles in $D(S)$ except the triangle $\triangle(v)$. Moreover, we can suppose that the segment $\overline{qq'}$ intersects $\triangle(v)$ at a unique point that is not a vertex of $\triangle(v)$. This condition can be always satisfied since we can move the point q' slightly in the triangle $\triangle(v)$.

Therefore, we can suppose that the triangle $\triangle(v)$ has three vertices p_1, p_2, and p_3, that the segment $\overline{qq'}$ intersects the edge $\overline{p_1p_2}$ of $\triangle(v)$ at an internal point q'', and that no points on the segment $\overline{qq''}$ (excluding the point q'') are contained in any triangle $\triangle(u)$ for some Voronoi vertex u. See Figure 7.2. Then the point p_3 and the point q are on different sides of the segment $\overline{p_1p_2}$. Since both points q and p_3 are contained in the convex hull *CH(S)*, the segment $\overline{p_1p_2}$ cannot be a boundary edge of *CH(S)*. Let $e = \{v, v'\}$ be the Voronoi edge defined by p_1 and p_2 (note that the vertex v must be an end-point of e), then by Lemma 5.2.4, e is a finite edge since the points p_1 and p_2 are not consecutive hull vertices on *CH(S)*. So the Voronoi vertex v' is not the infinite point, and v' must correspond to a triangle $\triangle(v')$ in the straight-line dual $D(S)$. By the definition of $\triangle(v')$, two vertices of $\triangle(v')$ must be the points p_1 and p_2, and the other vertex p_4 of $\triangle(v')$ must be different from the point p_3 since $v \neq v'$. Since the two triangles $\triangle(v)$ and $\triangle(v')$ do not overlap in the interior, by Lemma 7.3.1, the two points p_3 and p_4 must be on different sides of the segment $\overline{p_1p_2}$. Consequently, however, some points on the segment $\overline{qq''}$ which are very close to the point q'' would be contained in the interior of the triangle $\triangle(v')$. This contradicts our assumption that no points on $\overline{qq''}$ (excluding q'') is contained in any such triangles. This contradiction shows that q must belong to a triangle $\triangle(w)$ for some Voronoi vertex w in *Vor(S)*. $\square$

By Lemma 7.3.1 and Lemma 7.3.2, we obtain immediately that the straight-line dual $D(S)$ of the Voronoi diagram *Vor(S)* is a triangulation of the set S. This triangulation of S is called the *Delaunay Triangulation* of the set S.

Theorem 7.3.3 *TRIANGULATION* $\propto_n$ *VORONOI-DIAGRAM.*

PROOF. Given a set S of n points in the plane, which is an input to the problem TRIANGULATION, we simply pass S to the problem VORONOI-DIAGRAM. The solution to VORONOI-DIAGRAM on input S is the Voronoi diagram *Vor(S)* of S. Then from the Voronoi diagram *Vor(S)*, we construct the Delaunay Triangulation $D(S)$ of S by tracing all the Voronoi edges of *Vor(S)*. If the Voronoi diagram *Vor(S)* is given by a DCEL, then it is easy to see that the Delaunay Triangulation $D(S)$ of S can be constructed from *Vor(S)* in linear time. $\square$

Since the Voronoi diagram of a set of n points in the plane can be constructed in time $O(n \log n)$, by Corollary 7.0.2, we have

Corollary 7.3.4 *The problem TRIANGULATION can be solved in time $O(n \log n)$. In particular, the Delaunay triangulation $D(S)$ of a set S of n points in the plane can be constructed in time $O(n \log n)$.*

7.4 Euclidean minimum spanning tree

Consider the following problem: given a set S of n points in the plane, interconnect all the points by straight line segments so that the total length of the segments is minimum. This problem has an obvious application in computer networking where we want to interconnect all the computers at minimum cost.

It is easy to see that the resulting connected PSLG after the above interconnection must be a tree. In fact, if the resulting PSLG were not a tree, then we would be able to find a cycle, delete an edge from the cycle, and still keep the PSLG connected. But this would contradict the assumption that the resulting connected PSLG has the minimum total length of its edges. This tree is called a *Euclidean minimum spanning tree (EMST)* of the set S. In general, the Euclidean minimum spanning tree of a set is not unique.

The problem of finding Euclidean minimum spanning tree for a set of points in the plane is closely related to the following problem of finding the *minimum weight spanning tree*: given a weighted graph G, find a spanning tree of G with the minimum total weight. In fact, the problem of finding Euclidean minimum spanning tree can be reduced to the problem of finding minimum weight spanning tree, as we illustrate as follows.

Let S be a set of n points in the plane. To construct a Euclidean minimum spanning tree of S, we can regard S as a weighted complete graph G_S of n vertices, such that the weight of an edge $e = \{p, p'\}$ in G_S, where p and p' are two points in S, is the Euclidean distance between p and p'. Therefore, a Euclidean minimum spanning tree of the set S is a minimum weight spanning tree of the graph G_S, and vice versa. There are a few efficient algorithms constructing the minimum weight spanning tree for weighted graphs. For example, Kruskal's algorithm [15] constructs the minimum weight spanning tree for a weighted graph with m edges in time $O(m \log m)$. However, the complete graph G_S has $\Omega(n^2)$ edges. Therefore, a direct application of Kruskal's algorithm to the complete graph G_S would result in an $O(n^2 \log n)$ time algorithm for constructing a Euclidean minimum spanning tree for the set S.

Interesting enough, with the help of the Voronoi diagram and the Delaunay Triangulation of the set S, a single preprocessing can eliminate most of the edges of the complete graph G_S from our consideration.

Lemma 7.4.1 *Partition the set S into two non-empty disjoint subsets S_1 and S_2. If $\overline{p_1p_2}$ is the shortest line segment such that $p_1 \in S_1$ and $p_2 \in S_2$, then the line segment $\overline{p_1p_2}$ is an edge in the Delaunay Triangulation $D(S)$.*

PROOF. Suppose that the segment $\overline{p_1p_2}$ is not an edge in $D(S)$. Then the perpendicular bisector of $\overline{p_1p_2}$ contains no Voronoi edge of *Vor(S)*. Let V_1

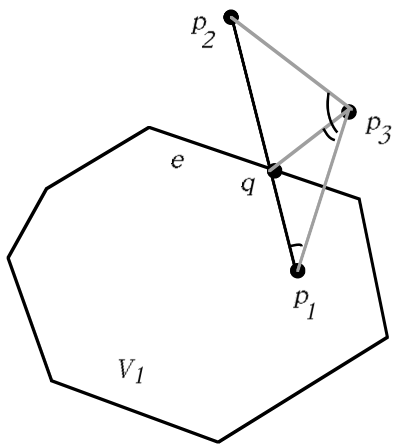

Figure 7.3: $\overline{p_1p_2}$ intersects V_1 at q

be the Voronoi polygon of the point p_1 in the Voronoi diagram *Vor(S)*, and suppose that the segment $\overline{p_1p_2}$ intersects the Voronoi polygon V_1 at a point q that is on the Voronoi edge e of V_1 in *Vor(S)*. (The point p_2 cannot be contained in V_1 (including the boundary of V_1) since V_1 is the locus of points closer to p_1 than to any other points in S.) Suppose that the Voronoi edge e is defined by the point p_1 and another point p_3 in S. See Figure 7.3. By the definition, the points p_1 and p_3 are the closest points in S to the points on the edge e. Therefore,

$$|\overline{p_1p_2}| = |\overline{p_1q}| + |\overline{qp_2}| > |\overline{p_1q}| + |\overline{qp_3}| \geq |\overline{p_1p_3}|$$

Moreover, since we have $\angle qp_3p_1 = \angle p_3p_1q$, and the point q is an internal point of the segment $\overline{p_1p_2}$, we must have

$$\angle p_2p_3p_1 > \angle qp_3p_1 = \angle p_3p_1q = \angle p_3p_1p_2$$

Therefore, we have

$$|\overline{p_1p_2}| > |\overline{p_2p_3}|$$

Now we obtain a contradiction, since both segments $\overline{p_2p_3}$ and $\overline{p_1p_3}$ are shorter than the segment $\overline{p_1p_2}$. Now if $p_3 \in S_1$ we pick $\overline{p_2p_3}$, and if $p_3 \in S_2$ we pick $\overline{p_1p_3}$. No matter what set the point p_3 is in, we are always able to find a segment with one end in S_1 and the other end in S_2 such that the segment is shorter than $\overline{p_1p_2}$. This contradicts the assumption that $\overline{p_1p_2}$ is the shortest such segment.

This contradiction proves that the segment $\overline{p_1p_2}$ must be an edge in the Delaunay Triangulation $D(S)$ of the set S. $\square$

Lemma 7.4.2 *Let p_1 and p_2 be two points in the set S. The segment $\overline{p_1p_2}$ is an edge of some Euclidean minimum spanning tree if and only if there is a partition of the set S into two non-empty sets S_1 and S_2 such that $\overline{p_1p_2}$ is the shortest segment with one end in S_1 and the other end in S_2.*

PROOF. Suppose that $\overline{p_1p_2}$ is an edge of a Euclidean minimum spanning tree T. Then deleting the edge $\overline{p_1p_2}$ from T results in two disjoint subtrees T_1 and T_2. Let S_1 and S_2 be the sets of points in S that are the vertices of the trees T_1 and T_2, respectively. S_1 and S_2 obviously form a partition of the set S and each of the sets S_1 and S_2 contains exactly one of the points p_1 and p_2. We claim that the segment $\overline{p_1p_2}$ is the shortest segment with one end in S_1 and the other end in S_2. In fact, if $\overline{pp'}$ is a shorter segment with one end in S_1 and the other end in S_2, then in the tree T, replacing the segment $\overline{p_1p_2}$ by the segment $\overline{pp'}$ would give us a Euclidean spanning tree T' of S such that the sum of the edge lengths of T' is less than the sum of the edge lengths of T. This contradicts the fact that T is a Euclidean minimum spanning tree.

Conversely, suppose that there is a partition of S into two non-empty subsets S_1 and S_2 such that $\overline{p_1p_2}$ is the shortest segment with one end in S_1 and the other end in S_2. Let T be a Euclidean minimum spanning tree of S. If T contains $\overline{p_1p_2}$, then we are done. Otherwise, adding the segment $\overline{p_1p_2}$ to T results in a unique simple cycle C. Since the segment $\overline{p_1p_2}$ is on the cycle C and p_1 and p_2 are in different sets of S_1 and S_2, there must be another segment $\overline{pp'}$ on the cycle such that the points p and p' are in different sets of S_1 and S_2. Since $\overline{p_1p_2}$ is the shortest segment with two ends in different sets of S_1 and S_2, the segment $\overline{pp'}$ is at least as long as the segment $\overline{p_1p_2}$. Replacing the segment $\overline{pp'}$ in T by the segment $\overline{p_1p_2}$ gives us a new Euclidean spanning tree T' of S such that the sum of the edge lengths of T' is not larger than the sum of the edge lengths of T. Since T is a Euclidean minimum spanning tree of S, the sum of the edge lengths of T' must be the same as that of T. Therefore, T' is also a Euclidean minimum spanning tree and T' contains the segment $\overline{p_1p_2}$. $\square$

Corollary 7.4.3 *If a segment $\overline{p_1p_2}$ is an edge of some Euclidean minimum* *spanning tree of the set S, then $\overline{p_1 p_2}$ is an edge in the Delaunay Triangulation $D(S)$ of the set S.*

PROOF. The proof follows from Lemma 7.4.1 and Lemma 7.4.2 directly. □

Therefore, the Delaunay Triangulation $D(S)$ contains all segments that are in Euclidean minimum spanning trees of the set S. Now if we regard $D(S)$ as a weighted graph $G_{D(S)}$ in which the weight of a segment $\overline{p_1 p_2}$ in $D(S)$ is the Euclidean distance between the two points p_1 and p_2, then a Euclidean minimum spanning tree of the set S is a minimum weighted spanning tree of the graph $G_{D(S)}$. This suggests the following algorithm.

```
Algorithm    EMST(S)

      Given:    a set of n points in the plane
      Output:   a Euclidean minimum spanning tree of S

      BEGIN
      1.   Construct the Delaunay triangulation D(S);
      2.   Construct a weighted graph G_D(S) that is
           isomorphic to D(S) such that the weight of an
           edge {p_i, p_j} in G_D(S) is the length of the
           corresponding edge in D(S);
      3.   Apply Kruskal's algorithm to find a minimum
           weight spanning tree T for G_D(S).  This tree
           T is a Euclidean minimum spanning tree for S;
      END.
```

The analysis of the algorithm EMST is straightforward. By Corollary 7.3.4, Step 1 for constructing the Delaunay triangulation $D(S)$ can be done in time $O(n \log n)$. To construct the graph $G_{D(S)}$, we simply compute the length of each edge in $D(S)$. Since $D(S)$ is a planar graph of n points, the number of edges of $G_{D(S)}$ is bounded by $O(n)$ (see Section 1.4). So Step 2 can also be done in time $O(n)$. Kruskal's algorithm runs in time $O(m \log m)$ on weighted graphs with m edges. Since the graph $G_{D(S)}$ has only $O(n)$ many edges, the application of Kruskal's algorithm on $C_{D(S)}$ takes time $O(n \log n)$. This gives the following theorem.

Theorem 7.4.4 *Given a set S of n points in the plane, the Euclidean minimum spanning tree of S can be constructed in time $O(n \log n)$.*

For the reason of completeness, we give a description of Kruskal's algorithm. Since the algorithm has been well studied in the course of Algorithm Analysis, we give only a brief outline of the algorithm and omit most of the details. The interested reader is referred to [2].

Kruskal's algorithm finds the minimum weight spanning tree for a weighted graph G by simply adding edges one at a time, at each step using the lightest edge that does not form a cycle. This algorithm gradually builds up the tree one edge at a time from disconnected components. The correctness of the algorithm follows from a theorem for weighted graphs that is similar to our Lemma 7.4.2.

To implement Kruskal's algorithm, suppose that the number of vertices of the graph G is n, and the number of edges of the graph G is m. We first presort all edges of G by their weight, then try to add the edges in order. The presorting of edges of G takes time $O(m \log m)$. We then maintain a forest F, which is a list of disjoint subtrees in the graph G. Each tree T in the forest F is represented by a UNION-FIND tree whose leaf-nodes contain the vertices of the tree T[1] (to distinguish the trees in the forest F, which are the trees in the weighted graph G, from the UNION-FIND trees that represent the trees in F, we call the vertices of the trees in F *vertices*, while call the vertices of the UNION-FIND trees *nodes*). Initially, the forest F is a list of n trivial trees, each is a single vertex of G. Pick the next edge $e = \{v, u\}$ from the sorted list of edges of G, and check if v and u are in the same UNION-FIND tree in the forest F. This can be done by two FIND operations followed by checking if the roots of the two UNION-FIND trees are identical. If v and u are in the same UNION-FIND tree in the forest F, then adding e would result in a cycle in the forest F. So we should throw the edge e. On the other hand, if v and u are in different UNION-FIND trees in the forest F, then the edge e does not form a cycle in the forest F, so we should add the edge e to the forest F. This is equivalent to merging the two UNION-FIND trees containing the vertices v and u in F. This can be done by a single UNION operation. We keep adding edges until the forest F contains a single tree, which is the minimum weight spanning tree of the weighted graph G. Since for each edge in the graph, at most three UNION-FIND operations are performed, to construct the final minimum weight spanning tree, we need at most $3m$ UNION-FIND operations. This can be done in time $O(m\alpha(m))$, where $\alpha(m) = o(\log(m))$ (see [2], Section 4.7

[1]For detailed discussion of UNION-FIND problem, the reader is referred to [2], Section 4.7.

for detailed discussion). Now since $m\alpha(m) = o(m \log m)$, we conclude that the running time of the Kurskal's algorithm is

$$O(m \log m) + O(m\alpha(m)) = O(m \log m)$$

7.5 Maximum empty circle

Given a set S of n points in the plane, the problem MAXIMUM-EMPTY-CIRCLE is to find a largest circle that contains no points of the set S and whose center is internal to the convex hull of the set S. We will call such a circle the *maximum empty circle* of the set S. The maximum empty circle of a set S can be specified by its center and its radius.

We first discuss where the center of the maximum empty circle can be located.

Lemma 7.5.1 *The center of the maximum empty circle of the set S must be either a Voronoi vertex of Vor(S), or the intersection of a Voronoi edge of Vor(S) and a boundary edge of the convex hull CH(S).*

PROOF. Suppose that C is a maximum empty circle of the set S such that C is centered at a point c.

Since C is the maximum empty circle, the boundary of the circle C must contain at least one point of the set S, otherwise, we can increase the radius of C (without moving the center c of C) to get a larger empty circle.

If the boundary of C contains only one point p in the set S, then we can move the center c of C away from the point p and increase the radius of C. This contradicts our assumption that C is the maximum empty circle.

Consequently, the center c of the circle C cannot be in the interior of any Voronoi polygon V of a point p of the set S, since otherwise, the point p is the only closest point in S to the center c, so the boundary of the circle C cannot contain any other points of S except p.

Therefore, the point c must be on a Voronoi edge of *Vor(S)* and the boundary of the circle C contains at least two points of the set S. Now suppose that c is not a Voronoi vertex of *Vor(S)*, then there are exactly two points p and p' of the set S on the boundary of the circle C. If c is not already on the convex hull, we can move it along the perpendicular bisector of p and p' away from both p and p' (without getting out of the convex hull *CH(S)*), and increase the radius of the circle C. This again contradicts the assumption that C is the maximum empty circle of S.

Therefore, the center c of the maximum empty circle must be either a Voronoi vertex in *Vor(S)*, or an intersection of a Voronoi edge and a boundary edge of the convex hull of S. $\square$

Let c be a Voronoi vertex of *Vor(S)* or an intersection of a Voronoi edge and a boundary edge of *CH(S)*. The radius of the largest empty circle centered at c can be computed easily. In fact, if c is a Voronoi vertex of *Vor(S)*, then c is equidistant from three points in the set S and no points of S is in the interior of the circle defined by these three points (Lemma 5.2.3). Therefore, the circle defined by these three points must be the largest empty circle centered at c. On the other hand, if c is an intersection of a Voronoi edge and a boundary edge of *CH(S)*, then exactly two points p and p' in S are closest to c, so the largest empty circle centered at c must have radius $|\overline{cp}| = |\overline{cp'}|$.

If the Voronoi diagram is given by a DCEL, then in constant time, we can compute the radius of the largest empty circle centered at a Voronoi vertex v, by an algorithm TRACE-VERTEX, which is similar to the algorithm TRACE-REGION in Section 1.4, to trace all incident Voronoi edges and all incident Voronoi polygons of the vertex v. (Note that a Voronoi vertex has degree exactly 3.) Since the Voronoi diagram *Vor(S)* has only $O(n)$ Voronoi vertices (Lemma 5.2.6), in linear time we can construct all largest empty circles that are centered at the Voronoi vertices of *Vor(S)*. Note that not all these circles are candidates of the maximum empty circle of S: those largest empty circles that are centered at a Voronoi vertex that is outside the convex hull *CH(S)* are disqualified. We will discuss later how to find these Voronoi vertices that are outside the convex hull *CH(S)*.

Now let us discuss the points that are intersections of Voronoi edges and the boundary edges of *CH(S)*. The first question is: how many such intersections can we have?

Lemma 7.5.2 *There are at most $O(n)$ intersections of Voronoi edges and the boundary edges of CH(S).*

PROOF. Since the convex hull *CH(S)* is convex, a Voronoi edge, which is a single straight line segment or a single straight semi-infinite ray, can intersect *CH(S)* at at most two points. Moreover, by Lemma 5.2.6, the Voronoi diagram *Vor(S)* has at most $O(n)$ Voronoi edges. $\square$

The following observation is also important.

Lemma 7.5.3 *Each boundary edge of the convex hull CH(S) intersects at least one Voronoi edge of Vor(S).*

PROOF. If a boundary edge $e = \{v, v'\}$ of *CH(S)* does not intersect any Voronoi edge, then the whole segment $\overline{vv'}$ is contained in a single Voronoi polygon of *Vor(S)*. But this is impossible, since the points on $\overline{vv'}$ that are very close to the point v should be contained in the Voronoi polygon of v, while the points on $\overline{vv'}$ that are very close to the point v' should be contained in the Voronoi polygon of v'. □

For simplicity, call the intersections of the Voronoi edges of *Vor(S)* and the boundary edges of *CH(S)* that are not a Voronoi vertex, the *intersecting points*. An intersecting point p_2 is the *successor* of an intersecting point p_1 if the partial chain on the boundary of the convex hull *CH(S)* from p_1 to p_2, in clockwise ordering, contains no other intersecting points.

Lemma 7.5.4 *If we trace the boundary of a Voronoi polygon clockwise, starting from an intersecting point p and leaving the convex hull, then we must encounter at least another intersecting point. The first intersecting point after p we encounter must be the successor of p.*

PROOF. Let the Voronoi polygon we are going to travel be V. Since the point p is on the boundary of V and is an intersecting point, the Voronoi polygon V must have at least one vertex inside the convex hull *CH(S)* and at least one vertex outside the convex hull *CH(S)*. Now since we are traveling the boundary of V and leaving the convex hull *CH(S)*, we must eventually come back and enter the convex hull *CH(S)* in order to reach the vertices of V that are inside *CH(S)*. Therefore, the boundary of the polygon V must intersect *CH(S)* at at least another point. Let p' be the first intersecting point after p we encounter. Since both the partial chain of V between p and p', and the partial chain of *CH(S)* between p and p' make only right turns, and because both V and *CH(S)* are convex, the partial chain of *CH(S)* between p and p' must be entirely contained in the Voronoi polygon V. That implies that no intersecting points are between the points p and p' on the partial chain of *CH(S)*. Therefore, the intersecting point p' is the successor of the intersecting point p. □

Now it is quite clear how we find all intersecting points. We start with an intersecting point p, travel the Voronoi polygon in the direction of leaving the convex hull *CH(S)*. We will encounter another intersecting point p', which is the successor of the intersecting point p. At the point p', we reverse the traveling direction and start traveling the adjacent Voronoi polygon from the point p', again in clockwise order and in the direction of leaving the convex hull *CH(S)*. We will hit the successor of p', etc.. We keep doing this until we come back to the first intersecting point.

We summarize this in the following algorithm.

```
Algorithm     FIND-ALL-INTERSECTIONS

      Given:    the Voronoi diagram Vor(S) and the convex
                hull CH(S) of a set S of n points
      Output:   all the intersecting points of Vor(S) and
                CH(S)

      BEGIN
      1.   Find an intersecting point p_0;
      2.   Let p = p_0;
      3.   Travel a Voronoi polygon clockwise in the direction
           of leaving the convex hull CH(S), starting from the
           point p to find the successor p' of p;
      4.   If p' <> p_0 then replace p by p' and go back to
           Step 3;
      END.
```

We analyze the algorithm. Suppose that the Voronoi diagram *Vor(S)* is given by a DCEL and the convex hull *CH(S)* is given by a circular doubly-linked list.

To find the first intersecting point p_0, we pick any boundary edge e of the convex hull *CH(S)*. Then we scan the DCEL representing the Voronoi diagram *Vor(S)* edge by edge and check which intersects e. By Lemma 7.5.3, e intersects at least one Voronoi edge in *Vor(S)*. So in linear time, we will find a Voronoi edge that intersects e and obtain the first intersecting point p_0. So Step 1 of the algorithm can be done in linear time.

Starting from an intersecting point p, we travel the part of the Voronoi polygon that is outside the convex hull *CH(S)*. By Lemma 7.5.4, we will encounter the successor of p. For this, we have to check, for each Voronoi edge e we are traveling, if e intersects the convex hull *CH(S)*. This seems to need $\Omega(n)$ time to check all boundary edges of the convex hull *CH(S)* for each Voronoi edge e. Fortunately, since each boundary edge of *CH(S)* contains at least one intersecting point (Lemma 7.5.3), the successor of p must be either on the boundary edge e of *CH(S)* where p is located, or on the boundary edge of *CH(S)* that is next to e. Therefore, for each Voronoi edge e we are traveling, we only have to check two boundary edges on *CH(S)*. So each Voronoi edge can be processed in constant time. Moreover, each Voronoi edge that is outside the convex hull *CH(S)* is traveled at most twice since each Voronoi edge is on the boundary of exactly two Voronoi polygons. Therefore, the total time spent on Step 3 and Step 4 in the algorithm FIND-ALL-INTERSECTIONS is bounded by the number of Voronoi edges that are outside the convex hull *CH(S)*, which is in turn bounded by the number of Voronoi edges of the Voronoi diagram *Vor(S)*, which is, by Lemma 5.2.6, bounded by $O(n)$.

Thererfore, the time complexity of the algorithm FIND-ALL-INTERSECTIONS is bounded by $O(n)$.

Finally, we discuss how to determine if a Voronoi vertex v is inside or outside the convex hull *CH(S)*. In the algorithm FIND-ALL-INTERSECTIONS, all the Voronoi vertices we encounter are outside the convex hull *CH(S)*. So we can simply mark them and not use them as potential candidates of the center of the maximum empty circle. The question is, can there be any Voronoi vertex that is outside the convex hull *CH(S)* and not encountered by our algorithm FIND-ALL-INTERSECTIONS? The answer is NO, as explained by the following paragraph.

Suppose that v is a Voronoi vertex of *Vor(S)* and that v is outside of the convex hull *CH(S)*. Let v be on the boundary of some Voronoi polygon V. The Voronoi polygon V cannot be completely outside the convex hull *Vor(S)*, since otherwise the corresponding point of the set S would be outside the convex hull *Vor(S)*. So the polygon V intersects *CH(S)* at at least two points. Let p and p' be two intersecting points of the polygon V and the convex hull *CH(S)* such that the vertex v is contained in the partial chain on the boundary of V from p to p' in clockwise ordering, and that no other intersecting points are on this partial chain. Then the algorithm FIND-ALL-INTERSECTIONS will eventually encounter the intersecting point p and trace this partial chain from p to p'. Now the vertex v must be encountered.

Summarizing the above discussions gives us the following algorithm for solving the problem MAXIMUM-EMPTY-CIRCLE.

```
Algorithm    MAXIMUM-EMPTY-CIRCLE
```

```
Given:   a set S of n points in the plane
Output:  the maximum empty circle of S

BEGIN
1.   Construct the Voronoi diagram Vor(S) and the
     convex hull CH(S);
2.   Call the subroutine FIND-ALL-INTERSECTIONS to
     find all intersecting points of Vor(S) and CH(S),
     and mark all Voronoi vertices that are outside
     the convex hull CH(S);
3.   For each q of such intersecting points, compute
     the largest empty circle centered at q;
4.   For each unmarked Voronoi vertex v, compute the
     largest empty circle centered at v;
5.   The largest among the largest empty circles
     constructed in Step 3 and Step 4 is the maximum
     empty circle of S;
END.
```

Step 1 takes time $O(n \log n)$, by Theorem 5.3.6 and by, say, the Graham Scan algorithm. Step 2 takes linear time, as we have discussed above. The other steps in the algorithm trivially take only linear time, by Lemma 5.2.6 and Lemma 7.5.2. Therefore, we obtain the following theorem.

Theorem 7.5.5 *The problem MAXIMUM-EMPTY-CIRCLE can be solved in time* $O(n \log n)$.

7.6 All-farthest vertex

The "inverse" of the problem ALL-NEAREST-NEIGHBOR is the problem ALL-FARTHEST-NEIGHBOR, in which we are asked to find the farthest neighbor for each point of a given set. The ALL-FARTHEST-NEIGHBOR problem can be solved through the Farthest Neighbor Voronoi Diagram. It can be shown that given a set S of n points in the plane, the Farthest Neighbor Voronoi Diagram of S can be constructed in time $O(n \log n)$. Moreover, with the Farthest Neighbor Voronoi Diagram, the problem ALL-FARTHEST-NEIGHBOR can be solved in another $O(n \log n)$ time, using the techniques of point location, as we discussed in Chapter 5. Therefore, the ALL-FARTHEST-NEIGHBOR problem can be solved in time $O(n \log n)$.

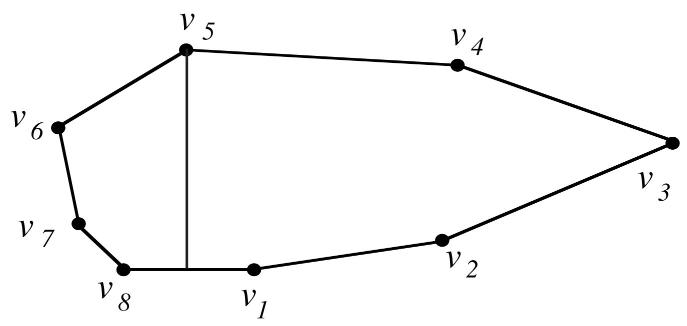

Figure 7.4: The vertices v_1 and v_3 are not an antipodal pair

In this section, we will discuss a restricted version of the problem ALL-FARTHEST-NEIGHBOR, the all-farthest-vertex problem for the set of vertices of a convex polygon. The goal is for each vertex of the convex polygon find the farthest vertex. Since the problem is "simpler" than the general problem, we expect a better algorithm, say, a linear time algorithm for solving this problem.

Let us first formally define the ALL-FARTHEST-VERTEX problem.

ALL-FARTHEST-VERTEX

For each vertex v of a convex polygon P, find a vertex of P that is farthest from v.

It would seem a simple generalization of the algorithm for finding the diameter of a convex polygon, as we showed in Section 3.3. That is, the farthest vertex of a vertex v must be a vertex that constitutes an antipodal pair with v. We first show that this intuition is incorrect.

Look at the Figure 7.4. The vertex v_3 is obviously the farthest vertex from the vertex v_1. However, since the vertex v_5 is the first farthest vertex from the edge $\overline{v_8v_1}$, by Lemma 4.3.4, the vertices v_1 and v_3 are not an antipodal pair.

To solve the problem ALL-FARTHEST-VERTEX, we first make an assumption that for each vertex v of P, the distances from v to any two vertices u and w of P are different. This assumption loses no generality since we can define the distance from v to a vertex u to be a triple $D(v,u) = (d,x,y)$,

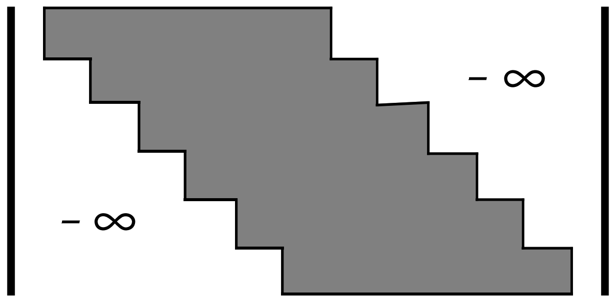

Figure 7.5: The matrix M_P

where d is the Euclidean distance between v and u, while x and y are the x- and y-coordinates of the vertex u, respectively. The distance $D(v, u)$ is ordered lexicographically.[2] With this assumption, each vertex of P has a unique farthest neighbor.

7.6.1 A monotone matrix

Let the vertices of a convex polygon P be given in counterclockwise order $(v_1, v_2, \cdots, v_n)$. It is convenient to describe the problem ALL-FARTHEST-VERTEX in terms of an $n \times (2n-1)$ matrix M_P pictured in Figure 7.5. In the ith row of M_P, the cell $(i, i+k)$ holds the distance $D(v_i, v_{k'})$ (where $k' = ((i+k-1) \pmod{n}) + 1$), for $1 \leq i \leq n$ and $0 \leq k \leq n-1$. All other cells of M_P hold $-\infty$. Solving the problem ALL-FARTHEST-VERTEX is equivalent to finding the maximal element in each row of the matrix M_P.

Note that we are not actually constructing the matrix M_P in the implementation of our algorithm. There are $\Omega(n^2)$ elements in the matrix M_P, so even writing the matrix M_P out takes time $\Omega(n^2)$. Instead, we keep a list for the indices of the rows and a list for the indices of the columns of the matrix M_P. Given a pair of indices (i, j), the element with the index (i, j) in the matrix M_P can be computed in constant time.

The matrix M_P has a very nice property, called monotone property.

[2]Note that the distance $D(v, u)$ is not symmetric, that is, in general, $D(v, u) \neq D(u, v)$. However, if $D(v, u)$ is the largest then the vertex u must be one of the vertex of P that has the farthest Euclidean distance from the vertex v.

Definition An $n \times m$ matrix $M = (a_{i,j})$ is *monotone* if for any two pairs (i_1, i_2) and (j_1, j_2) of indices, where $1 \leq i_1 < i_2 \leq n$ and $1 \leq j_1 < j_2 \leq m$, the 2×2 submatrix of M

$$\begin{pmatrix} a_{i_1 j_1} & a_{i_1 j_2} \\ a_{i_2 j_1} & a_{i_2 j_2} \end{pmatrix}$$

has the property that it is not simultaneously possible that $a_{i_1 j_1} < a_{i_1 j_2}$ and $a_{i_2 j_1} > a_{i_2 j_2}$.

Lemma 7.6.1 *The matrix M_P is monotone.*

PROOF. Given a 2×2 submatrix of M_P

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

taken from the i_1th and i_2th rows and j_1th and j_2th columns of the matrix M_P, where $i_1 < i_2$ and $j_1 < j_2$. Suppose by contradiction that we have $a < b$ and $c > d$. Then b and c cannot be $-\infty$.

The number a cannot be $-\infty$. Otherwise since $b \neq -\infty$ and a is on the left of b in the matrix M_P and $a = -\infty$, so c must be $-\infty$ since c is in the same column as a and c is below a. But by our assumption, c is not $-\infty$.

Similarly, the number d cannot be $-\infty$.

So none of a, b, c, and d can be $-\infty$. Now let us consider the relations among the indices i_1, i_2, j_1, and j_2.

Since the element c has index (i_2, j_1) and $c \neq -\infty$, so we must have $i_2 \leq j_1$. If $i_2 = j_1$ then c is the distance between the vertex v_{i_1} to itself in P, thus $c = 0$. But this is impossible since $c > d$ and $d \neq -\infty$. Thus we must have $i_2 < j_1$. Therefore, we can write explicitly

$$i_1 < i_2 < j_1 < j_2$$

Therefore, the vertices v_{i_1}, v_{i_2}, v_{j_1} and v_{j_2} must appear on the convex polygon P in exactly this order (the indices j_1 and j_2 actually take values $(\mathrm{mod}\ n) + 1$). See Figure 7.6. However, now the conditions $a < b$ and $c > d$ implies that $c + b > d + a$. Since the polygon P is convex, $c + b > d + a$ says that the sum of the lengths of opposite sides of a convex quadrilateral is

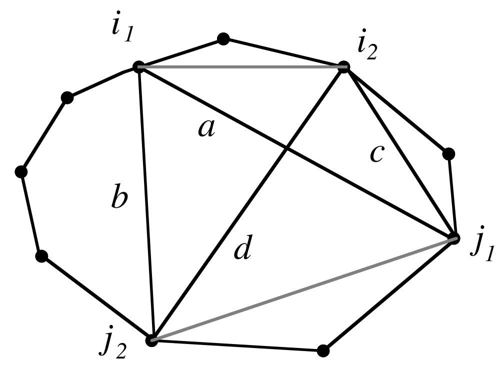

Figure 7.6: The convex polygon P

greater than the sum of the lengths of the diagonals of the same quadrilateral. However, this contradicts a fundamental theorem in elementary geometry. (Remark: this is the only place we use the convexity of the polygon P.)

Therefore, if we suppose that $a < b$ and $c > d$, then we are always able to derive a contradiction. This proves that $a < b$ and $c > d$ cannot be simultaneously possible. That is, the matrix M_P is monotone. □

Corollary 7.6.2 *Every submatrix of the matrix M_P is monotone.*

PROOF. This is because that each 2×2 submatrix of a submatrix of M_P is also a submatrix of M_P. □

Lemma 7.6.1 and Corollary 7.6.2 are crucial for the algorithms we are going to give.

7.6.2 Squaring a monotone matrix

The matrix M_P is a rectangle matrix that contains more columns than rows. Since we are only interested in finding the maximal element in each row of the matrix, at most n columns are really useful to us. In the following, we will discuss how to square a rectangle matrix without deleting the maximal element in each row. We will actually consider a little bit more general case, that is, how do we square a rectangle submatrix of M_P that has more columns than rows such that the maximal element in each row is kept in the resulting square matrix.

Let

$$M = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,h} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,h} \\ & \cdots & \cdots & \\ a_{r,1} & a_{r,2} & \cdots & a_{r,h} \end{pmatrix}$$

be an $r \times h$ submatrix of the matrix M_P. By Corollary 7.6.2, the matrix M is monotone.

Now let us look at the first row, compare the elements $a_{1,1}$ and $a_{1,2}$. If $a_{1,1} < a_{1,2}$, then no maximal element in any row can be in the first column. In fact, $a_{1,1}$ is not the maximal element in the first row. Suppose that the maximal element of the ith row of M is in the first column, $i > 1$, then we have $a_{i,1} > a_{i,2}$. This together with $a_{1,1} < a_{1,2}$ contradicts the fact that M is a monotone matrix. Therefore, the first column of M can be deleted in this case. After deleting the first column of M, we compare the elements $a_{1,2}$ and $a_{1,3}$. Similarly, if $a_{1,2} < a_{1,3}$ then we can delete the second column of M and compare the elements $a_{1,3}$ and $a_{1,4}$ and so on. We keep doing this until we find an index i_1 such that $a_{1,i_1} > a_{1,i_1+1}$. Now we save the column i_1 and move to the second row of M.[3]

We look at the second row and compare the elements a_{2,i_1+1} and a_{2,i_1+2}. If $a_{2,i_1+1} \geq a_{2,i_1+2}$ then we save the (i_1+1)st column of M and move to the third row. On the other hand, if $a_{2,i_1+1} < a_{2,i_1+2}$, then none of the 2nd, 3rd, $\cdots$, rth rows of M can have their maximal element in the (i_1+1)st column of M because M is monotone. Moreover, since we know that $a_{1,i_1} > a_{1,i_1+1}$, the first row of M does not have its maximal element in the (i_1+1)st column either. Therefore, the (i_1+1)st column of M contains no maximal elements for any row, thus can be deleted. Now since we know no relation between the elements a_{1,i_1} and a_{1,i_1+2}, so after deleting the (i_1+1)st column, we move back to the first row and compare a_{1,i_1} and a_{1,i_1+2}.

Now we discuss the general case. Inductively, suppose that we have moved to the kth row of M with $2 \leq k \leq r-1$, and we have saved the i_1th, i_2th, $\cdots$, and i_{k-1}th columns of M such that

$$\begin{aligned} a_{1,i_1} &> a_{1,i_2} \\ a_{2,i_2} &> a_{2,i_3} \end{aligned}$$

[3]It seems that we have ignored the equality case, i.e., the case when $a_{1,i_1} = a_{1,i_1+1}$. However, the equality case can never happen. It is because that by our definition of the distance $D(v,u)$, vertex v has a unique distance to a vertex u. So case $a_{1,i_1} = a_{1,i_1+1}$ happens if and only if both a_{1,i_1} and a_{1,i_1+1} are $-\infty$. But by our selection of i_1, a_{1,i_1} can never be $-\infty$.

$$\begin{array}{rcl} & \cdots & \\ a_{k-2,i_{k-2}} & > & a_{k-2,i_{k-1}} \\ a_{k-1,i_{k-1}} & > & a_{k-1,i_{k-1}+1} \end{array}$$

and none of the deleted columns contain maximal elements of any row of M. Then we compare the elements $a_{k,i_{k-1}+1}$ and $a_{k,i_{k-1}+2}$ in the kth row of M. If $a_{k,i_{k-1}+1} > a_{k,i_{k-1}+2}$ then we save the $(i_{k-1}+1)$st column and move to the $(k+1)$st row of M. On the hand, if $a_{k,i_{k-1}+1} < a_{k,i_{k-1}+2}$, then for any $j > k$ the jth row in the matrix M cannot have its maximal element in the $(i_{k-1}+1)$st column since M is monotone. Moreover, by our inductive hypothesis, $a_{k-1,i_{k-1}} > a_{k-1,i_{k-1}+1}$, so the $(k-1)$st row does not have its maximal element in the $(i_{k-1}+1)$st column. If for some j such that $j < k-1$ such that the jth row has its maximal element in the $(i_{k-1}+1)$st column, then $a_{j,i_{k-1}} < a_{j,i_{k-1}+1}$. But this together with $a_{k-1,i_{k-1}} > a_{k-1,i_{k-1}+1}$, contradicts the fact that the matrix M is monotone. Summarizing these discussions, we conclude that the $(i_{k-1}+1)$st column of the matrix M contains no maximal element for any row, thus can be deleted.[4] Now since we have no idea about the relation between the elements $a_{k-1,i_{k-1}}$ and $a_{k-1,i_{k-1}+2}$, we move back to the $(k-1)$st row of M and compare the two elements.

The case for the last row (the rth row) should be treated specially. Suppose that we have moved to the rth row of M, and we have saved the i_1th, i_2th, $\cdots$, and i_{r-1}th columns of M such that

$$\begin{array}{rcl} a_{1,i_1} & > & a_{1,i_2} \\ a_{2,i_2} & > & a_{2,i_3} \\ & \cdots & \\ a_{r-2,i_{r-2}} & > & a_{r-2,i_{r-1}} \\ a_{r-1,i_{r-1}} & > & a_{r-1,i_{r-1}+1} \end{array}$$

and none of the deleted columns contain maximal elements of any row of M. We compare the elements $a_{r,i_{r-1}+1}$ and $a_{r,i_{r-1}+2}$ in the rth row of M. Again there are two cases.

If $a_{r,i_{r-1}+1} < a_{r,i_{r-1}+2}$, then exactly as we have discussed for the case $2 \leq k \leq r-1$, the $(i_{r-1}+1)$st column of the matrix M contains no row maximal elements, thus can be deleted. Moreover, since we have no idea

[4] Again, since we can easily prove that $a_{k,i_{k-1}+1}$ can never be $-\infty$, we ignore the case when $a_{k,i_{k-1}+1} = a_{k,i_{k-1}+2}$.

about the relation between the elements $a_{r-1,i_{r-1}}$ and $a_{r-1,i_{r-1}+2}$, we move back to the $(r-1)$st row and compare the two elements.

On the other hand, if $a_{r,i_{r-1}+1} > a_{r,i_{r-1}+2}$, then $a_{r,i_{r-1}+2}$ cannot be the row maximal element for the rth row. Moreover, no other elements in the $(i_{r-1}+2)$nd column can be row maximal elements, since otherwise we would find an index $j < r$ such that

$$a_{j,i_{r-1}+1} < a_{j,i_{r-1}+2} \qquad \text{and} \qquad a_{r,i_{r-1}+1} > a_{r,i_{r-1}+2}$$

contradicting the fact that the matrix M is monotone. Therefore, the $(i_{r-1}+2)$nd column can be deleted. Now we compare the elements $a_{r,i_{r-1}+1}$ and $a_{r,i_{r-1}+3}$, and so on.

Keeping doing the above process, we will get a square matrix at some moment. This can be shown as follows. Suppose that the number of columns is greater than the number of rows. If we are at the kth row with $2 \leq k \leq r-1$, then either we will delete a column then move one row up or we will move one row down without deleting any columns. If we are at the first row, then either we delete a column and remain in the first row or we move to the second row without deleting any columns. If we are at the last row, then either we delete a column and move back to the $(r-1)$st row or we delete a column and remain in the rth row. Therefore, only when we are moving down we do not delete columns. However, we cannot move down forever since eventually we will reach the last row, in which we have to delete columns. By our inductive proof above, all maximal elements of the rows of M are contained in the resulting square matrix.

We implement the above idea into the following algorithm SQUARE. Suppose that the submatrix M of the matrix M_P contains the elements in the k_1th, k_2th, $\cdots$, k_rth rows and the j_1th, j_2th, $\cdots$, j_hth columns such that

$$k_1 < k_2 < \cdots < k_r \quad \text{and} \quad j_1 < j_2 < \cdots < j_h$$

and $r < h$. The indices $k_1, k_2, \cdots, k_r$ are stored in a doubly-linked list L_{row}, and the indices $j_1, j_2, \cdots, j_h$ are stored in another doubly-linked list L_{col}. The algorithm SQUARE takes the two doubly-linked lists L_{row} and L_{col} as its input, and outputs a doubly-linked list L_c that contains the indices of the columns of M that are saved in the process. The list L_c contains r indices. Since the lists are doubly-linked, for each element in a list, we can always access in constant time the previous element in the list through a pointer "*last*", and the next element in the list through a pointer "*next*".

```
Algorithm    SQUARE(L_row, L_col)

      Given:   a rectangle submatrix M of M_P, represented
               by a list of row indices L_row and a list of
               column indices L_col
      Output:  a square matrix M' obtained from M by deleting
               some columns of M such that all row maximal
               elements in M are kept in M'

      BEGIN
      1.   Let L_c = L_col, let j and k be the first elements
           in the list L_c and L_row, respectively;
      2.   WHILE  the matrix is not square DO
      2.1  CASE 1:  k is the first element in the list L_row
             IF a_{k, j} < a_{k, next(j)}  THEN
               let j = next(j) and delete the first element in
               the list L_c
             ELSE (* so a_{k, j} > a_{k, next(j)} *)
               let j = next(j), and let k be the second element
               in the list L_row,
      2.2  CASE 2: k is neither the first nor the last in L_row
             IF a_{k, j} < a_{k, next(j)}   THEN
               let  j = last(j) and delete the old j from the
               list L_c, and let k = last(k)
             ELSE (* so a_{k, j} > a_{k, next(j)} *)
               let k = next(k) and j = next(j);
      2.3  CASE 3: k is the last element in the list L-row
             IF a_{k, j} < a_{k, next(j)}  THEN
               let  j = last(j) and delete the old j from the
               list L_c, and let k = last(k);
             ELSE (* so a_{k, j} > a_{k, next(j)} *)
               let j = next(j), and delete the old j from the
               list L_c
           END of WHILE;
      3.   Output the list L_c;
      END.
```

We analyze the algorithm. The algorithm is obviously dominated by the WHILE loop (Step 2). First note that the value of an element $a_{k,j}$ in the matrix M can be computed in constant time if we are given the convex polygon P and the indices k and j. Therefore, each execution of the WHILE loop takes constant time. Now let us discuss how many times the WHILE loop in the algorithm can be executed. Note that whenever we move one row up, we delete one column. To make the matrix square, we delete exactly $h-r$ columns from the matrix M. So we can move up at most $h - r$ rows. This also implies that we can move down at most $r+(h-r) = h$ rows. Thus, there are at most $2h - r$ executions of the WHILE loop that move on row up or down. If an execution of the WHILE loop does not move a row up or down, then we must be at the first row or the last row, but then we must delete a column. Thus, there are at most $h - r$ executions of the WHILE loop that does not move on row up or down. Summarizing this together, we conclude that the WHILE loop is executed at most $2h - r + h - r = O(h)$ times. Consequently, the time complexity of the algorithm SQUARE is bounded by $O(h)$.

7.6.3 The main algorithm

Before we give the main algorithm for our problem, we consider the following problem: let M be a monotone matrix, and suppose that the maximal element of the ith row of M is in the j_1th column, while the maximal element of the $(i+2)$nd row of M is in the j_2th column, then what column of M can the maximal element of the $(i+1)$st row be in? Since the matrix M is monotone, we must have $j_1 \leq j_2$. Moreover, since M is monotone, the maximal element of the $(i+1)$st row must be in a column that is between the j_1th column and j_2th column (including the j_1th column and j_2th column themselves). Therefore, instead of scanning the whole $(i+1)$st row to find the maximal element, we only have to scan the elements in the $(i+1)$st row that are between the j_1th column and j_2th column.

Now we are ready for the main algorithm for the problem ALL-FARTHEST-VERTEX. Suppose we are given a convex polygon P of n vertices v_1, v_2, $\cdots$, and v_n. Define the matrix M_P as above. The value of each element $a_{i,j}$ of the matrix M_P can be computed in constant time if we know the indices i and j. We use the following algorithm to find the farthest vertex for each vertex of the convex polygon P. The subroutine ROW-MAXIMAL takes an $r \times r$ monotone submatrix M of M_P as input and returns back a list L of r indices such that for $1 \leq i \leq r$, if the ith element in L is k_i, then the element a_{i,k_i} is the maximal element in the ith row of the submatrix M.

```
Algorithm     ALL-FARTHEST-VERTEX(P)
```

```
Given:   a convex polygon P
Output:  for each vertex of P, find the farthest vertex

BEGIN
1.   Construct a doubly-linked list L_row containing
     the indices 1, 2, ..., n, and a doubly-linked
     list L_col containing the indices 1, 2, ...,
     2n-1;
2.   Call the subroutine SQUARE(L_row, L_col) to
     obtain a list L_c of column indices of M_P
     such that these columns constitute a square
     submatrix of M_P that contains the maximal
     element for each row of M_P;
3.   Call the subroutine ROW-MAXIMAL(L_row, L_c);
4.   Suppose that the subroutine ROW-MAXIMAL(L_row, L_c)
     returns a list L, then for 1 <= i <= n,
     if the ith element of L is k_i, then the vertex
     v_{k_i'} is the farthest vertex from the vertex
     v_i in the convex polygon P, where
               k_i' = (k_i - 1)mod(n) + 1;
END.
```

The subroutine ROW-MAXIMAL is given as follows. Here we suppose that M is a $r \times r$ submatrix of the matrix M_P, and the row indices and column indices of M are given by two lists L_r and L_c, respectively.

```
Algorithm   ROW-MAXIMAL(L_r, L_c)

      Given:    two doubly-linked lists L_r and L_c containing
                the indices of rows and the columns of an r by
                r submatrix M of the matrix M_P, respectively
      Output:   a list L of column indices such that the ith
                element of L is the column index of the maximal
                element in the ith row of the matrix M

      BEGIN
      1.   IF L_c contains one element, return L_c directly;
      2.   Delete every other element from the list L_r.  Let
           the resulting list be L_r';
           { This is equivalent to deleting all rows with even
             index from the matrix M.  Let the resulting matrix
             be M_1.  M_1 consists of the rows of M that have
```

```
          odd index.  The matrix M_1 is an r/2 by r matrix. }
     3.   Call the subroutine  SQUARE(L_r', L_c);
          { The algorithm SQUARE returns a list L_c' of size
            r/2, which corresponds to a list of column indices
            such that these columns constitute an r/2 by r/2
            square matrix that contains all maximal elements
            in the odd rows of the matrix M. }
     4.   Recursively call the subroutine ROW-MAXIMAL(L_r', L_c');
          { This recursive call will return a list L that contains
            the column indices with which the maximal elements in
            the odd rows are located. }
     5.   With the help of the list L, determine the column indices
          for the maximal elements in the even rows of M.  For this,
          suppose in the (2 i - 1)st row of M, the maximal element
          is in the j_1th column, and in the (2 i + 1)st row of M,
          the maximal element is in the j_2th column, then scan the
          elements in the (2i)th row only from column j_1 to column
          j_2, the maximal element among these elements must be the
          maximal element of the (2i)th row;
     END.
```

Let us first look at the time complexity of the algorithm ROW-MAXIMAL. Step 1 and Step 2 can obviously done in time $O(r)$, if the input matrix M is an $r \times r$ matrix. By the analysis of the algorithm SQUARE, Step 3 can done in time $O(r)$. Now look at Step 5. Suppose that the list L contains $(r/2)$ indices j_1, j_2, $\cdots$, and $j_{r/2}$, which are the column indices of maximal elements of odd rows of M, then since the submatrix M is monotone

$$j_1 \leq j_2 \leq \cdots \leq j_{r/2}$$

As we discussed before, to find the maximal element in the $(2i)$th row, we only have to scan the elements in the $(2i)$th row from the column j_i to the column j_{i+1}. Therefore, to find maximal elements for all even rows, we will scan at most

$$\begin{aligned} & (j_2 - j_1 + 1) + (j_3 - j_2 + 1) + \cdots + (j_{r/2} - j_{r/2-1} + 1) + (r - j_{r/2} + 1) \\ = \; & r/2 + r - j_1 \\ = \; & O(r) \end{aligned}$$

elements. Therefore, the time for executing Step 5 is also bounded by $O(r)$.

Let the time complexity of the algorithm ROW-MAXIMAL be $T(r)$ when the input is an $r \times r$ matrix, then Step 4 in the algorithm takes time $T(r/2)$, and all other steps take time $O(r)$, so we have

$$T(r) \leq T(r/2) + cn$$

where c is a constant. It is easy to see that $T(n) = O(n)$. That is, the algorithm ROW-MAXIMAL takes linear time.

Now we analyze the algorithm ALL-FARTHEST-VERTEX. Step 1 and Step 4 obviously take time $O(n)$. Since the algorithm SQUARE takes time $O(n)$, and by the analysis above, Step 3, the algorithm ROW-MAXIMAL also takes time $O(n)$, we conclude that the time complexity of the algorithm ALL-FARTHEST-VERTEX runs in time $O(n)$.

Theorem 7.6.3 *The problem ALL-FARTHEST-VERTEX for convex polygons can be solved in linear time.*

7.7 Exercises

1. Give examples to show that a problem P' may have very high complexity (e.g. *NP*-complete) even a linear time solvable problem P is linear time reducible to P'.

2. A *star-shaped polygon* $P = \{p_1, \cdots, p_n\}$ is a simple polygon containing at least one point q such that the segment $\overline{qp_i}$ lies entirely within P for all $1 \leq i \leq n$. The problem *STAR-POLYGON* is to find a star-shaped polygon whose vertex set is the given set of points in the plane. Show that the problem *CONVEX HULL* is linear time reducible to the problem *STAR-POLYGON*.

3. Given a star-shaped polygon P, find two vertices of P that are the farthest apart.

4. Give a detailed proof that the problem *CONVEX HULL* is linear time reducible to the problem *VORONOI-DIAGRAM*.

5. Consider the following problem in Robotics: Let S be a set of obstacles on the plane. These obstacles are discs of the same radii. You have a mobile "Robot" R which has shape of disc with a radius of 1. We want an algorithm such that for any obstacle set S, and for any two points p and q, the algorithm will find a path for the robot R from position p to position q, avoiding the obstacles. If no such path exists, the algorithm

reports accordingly. Design and analyze an algorithm for this problem. (Hint: construct the Voronoi diagram for the centers of the obstacles).

6. Given a set of n points in the plane, prove that the Delaunay triangulation contains at most $2n-5$ vertices and at most $3n-6$ edges.

7. A *monotone polygon* is a simple polygon whose boundary can be decomposed into two monotone chains (a chain is monoton if every vertical line intersects it at at most 1 point). The problem *MONOTON-POLYGON* is to find a monoton polygon whose vertex set is the given set of points in the plane. Show that the problem *CONVEX HULL* is linear time reducible to the problem *MONOTON-POLYGON*.

8. Show that the problem *CONVEX HULL* is linear time reducible to the following problem.

   INTERSECTION-OF-HALF-PLANE

   given a system of N linear inequalities of the form

   $$a_i x + b_i y + c_i \geq 0 \qquad i = 1, 2, \ldots, N.$$

   find the region of the solutions of it.

9. Show that the problem *CONVEX HULL* is linear time reducible to the problem of constructing the convex hull of points in 3-dimension space even if the points are given sorted with respect to the x-coordinates. (Recall that the convex hull computation requires the reporting of vertices, edges, and faces that lie on the convex hull and their adjacency relations with respect to one another.)

10. Suppose that a problem P is reducible to a problem P' in $O(n \log n)$ time and that the problem P' is solvable in time $O(n \log n)$. Is the problem P necessarily solvable in time $O(n \log n)$? Justfy your answer.

11. Given two sets A and B, with m and n planar points, respectively. Find two points, one from each set, that are closest. (*Hint*: You should consider the following three different cases: (1) m is much larger than n; (2) n is much larger than m; (3) m and n are of the same order.)

12. The problem *All Nearest Neighbors* is stated as follows: given a set S of n points in the plane, find a nearest neighbor of each. Show that

this problem can be reduced in linear time to the problme *Voronoi-Diagram.*

13. It has been recently shown that triangulating a simple polygon can be done in linear time. Use this result to show that triangulating a connected PSLG in which each face is a simple polygon can be done in linear time.

14. Consider the following problem of *SECOND CLOSEST PAIR*: Given a set S of n points in the plane, find a pair of points p_1 and p_2 in S such that the distance between p_1 and p_2 is the second shortest among all pairs of points of S. (Of course, if there are two distinct closest pairs, then either of them can be regarded as the second closest pair).

    Show that the problem *SECOND CLOSEST PAIR* can be reduced to the problem *VORONOI DIAGRAM* in linear time. Thus, it can be solved in $O(n \log n)$ time.

15. Design an efficient algorithm that computes the area of an n-vertex simple, but not necessarily convex polygon.

16. Design an efficient algorithm that finds the second farthest pair from among n points in the plane.

17. Design a linear time algorithm for the following problem: given $Vor(S)$, where S is a set of n points in the plane, find a σ-chain (i.e., a path in $Vor(S)$ with both ends extended to infinity) such that each side of the σ-chain contains half of the points in S.

18. The *Euclidean Traveling Saleman* problem (*ETS*) is to find a shortest closed path through n given points in the plane. Show that an approximate ETS tour whose length is less than twice the length of a shortest tour can be constructed in time $O(n \log n)$. (*Hint:* reduce the problem to the problem of *Euclidean Minimum Spanning Tree* problem.)

Chapter 8

Lower Bound Techniques

We have discussed quite a few algorithms for geometric problems, including constructing convex hulls of finite sets of points in the plane, solving proximity problems, finding the intersection of geometric objects, and searching in PSLGs. Most of these problems can be solved by brute force methods in time $O(n^2)$ or more. Our techniques (geometric sweeping, divide and conquer, prune and search, and reduction) gives faster algorithms for solving these problems. Most of our algorithms run in linear time or in time $O(n \log n)$. For those linear time algorithms, we know that we have obtained asymptotically optimal solutions because even just reading the input for the problems takes linear time. For those $O(n \log n)$ time algorithms, however, a very natural question is whether we can further improve them, or, equivalently, are these algorithms the best possible.

This question brings us to an important, deep, and in general difficult branch in theoretical computer science, the study of lower bounds of problems. Here instead of *designing* a *single* efficient algorithm for a given problem, we want to *prove* that *any* algorithm solving the problem takes at least certain amount of time.

Let us look at the problem of constructing convex hulls. We have discussed the relationship between constructing convex hulls and sorting (see Section 6.1), we may have realized that an algorithm faster than $O(n \log n)$ for convex hull is impossible, since as we have seen in Algorithm Analysis that sorting n numbers requires at least $\Omega(n \log n)$ comparisons (see, for example, [2]), and since

$$\text{SORTING} \propto_n \text{CONVEX HULL}$$

so the problem CONVEX HULL is at least as hard as SORTING. However, we are not completely satisfied with this result because the computational model used is too restricted: it cannot even do multiplication! On the other hand, just computing the standard Euclidean distance metric requires quadratic polynomials.

In this chapter, we will introduce a general technique for deriving lower bounds for geometric problems. We first look closely at the computational model that can do only comparison, the linear decision tree, then extend the result on linear decision tree to a more powerful computational model, the algebraic decision tree. Lower bounds then are obtained on this model for most of the geometric problems we have discussed in the previous chapters. Combining these lower bounds and the algorithms we have derived, we conclude that most of those algorithms developed in the previous chapters are in fact optimal.

8.1 Preliminaries

Let us first have a brief review of geometry. Let S be a subset of the n-dimensional Euclidean space E^n. S is *connected* if for any pair of points p and q of S, there is a curve C adjoining them such that C is entirely contained in S. By the definition, a convex set in E^n is connected. Now suppose that W is a subset of E^n that is not necessarily connected, then a *connected component* of W is a maximal connected subset of W. We will use $\#W$ to denote the number of connected components of the set W.

A function $f(x_1, \cdots, x_n)$ is a *polynomial* if f is a sum of terms of the form $cx_1^{i_1}x_2^{i_2}\cdots x_n^{i_n}$, where c is a constant, and all i_j's are non-negative integers. The *degree* of the term $cx_1^{i_1}x_2^{i_2}\cdots x_n^{i_n}$ is defined to be the number $i_1+i_2+\cdots+i_n$. The *degree* of a polynomial is the maximum of the degrees of its terms. The function f is a *linear polynomial* if in each term of the above form, we have $i_j \leq 1$, for all $1 \leq j \leq n$. An equation $f(x_1, \cdots, x_n) = 0$ with f being a linear polynomial defines a hyperplane in the n-dimensional Euclidean space E^n. An open inequality $f(x_1, \cdots, x_n) > 0$ (or $f(x_1, \cdots, x_n) < 0$) defines an *open halfspace* in E^n, with the hyperplane $f(x_1, \cdots, x_n) = 0$ being its boundary. Similarly, a closed inequality $f(x_1, \cdots, x_n) \geq 0$ (or $f(x_1, \cdots, x_n) \leq 0$) defines a *closed halfspace* in E^n, with the hyperplane $f(x_1, \cdots, x_n) = 0$ being its boundary. It is easy to see that hyperplanes, open halfspaces, and closed halfspaces are all convex sets in E^n.

Let S be the set of points $(x_1, \cdots, x_n)$ satisfying a sequence of relations:

$$f_i(x_1, \cdots, x_n) = 0 \qquad i = 1, \cdots, m_1$$

$$g_j(x_1, \cdots, x_n) > 0 \qquad j = 1, \cdots, m_2$$
$$h_k(x_1, \cdots, x_n) \geq 0 \qquad k = 1, \cdots, m_3$$

where all functions f_i, g_j, and h_k, where $i = 1, \cdots, m_1$, $j = 1, \cdots, m_2$, and $k = 1, \cdots, m_3$ are linear polynomials. Then S is the intersection of the hyperplanes $f_i = 0$, $1 \leq i \leq m_1$, the open halfspaces $g_j > 0$, $1 \leq j \leq m_2$, and the closed halfspaces $h_k \geq 0$, $1 \leq k \leq m_3$. Since all hyperplanes, open halfspaces, closed halfspaces are convex, by Theorem 3.1.1, the set S is also convex.

A problem is a *decision problem* if it has only two possible solutions, either the answer YES or the answer NO. Abstractly, a decision problem consists simply of a set of *instances* that contains a subset called the set of *YES-instances*. As we have studied in Algorithm Analysis, decision problems play a very important role in the analysis of *NP*-completeness. In practice, many general problems can be reduced to decision problems such that a general problem and the corresponding decision problem have the same complexity.

There are certain problems where it is realistic to consider the number of branching instructions executed as the primary measure of complexity. In the case of sorting, for example, the outputs are identical to the inputs except for order. It thus becomes reasonable to consider a model in which all steps are two-way branches based on a "decision" that we should make when computation reaches that point.

The usual representation for a program of branches is a binary tree called a *decision tree*. Each non-leaf vertex represents a decision. The test represented by the root is made first, and "control" then passes to one of its sons, depending on the outcome of the decision. In general, control continues to pass from a vertex to one of its sons, the choice in each case depending on the outcome of the decision at the vertex, until a leaf is reached. The desired output is available at the leaf reached. If the decision at each non-leaf vertex of a decision tree is a comparison of a polynomial of the input variables with the number 0, then the decision tree is called an *algebraic decision tree*.

It should be pointed out that although the algebraic decision tree model seems much weaker than a real computer, in fact this intuitive feeling is not very correct. First of all, given a computer program, we can always represent it by a decision tree by "unwinding" loops in the program. Secondly, the operations a real computer can perform are essentially additions and branchings. All other operations are in fact done by microprograms that consists of those elementary operations. For example, the value of $\sin(x)$ for a number

x is actually obtained by an approximation of the Taylor's extension of the function $\sin(x)$. Finally, we simply ignore the computation instructions and concentrate on only branching instructions because we are working on lower bound of algorithms. If we can prove that for some problem, at least N branchings should be made, then of course, the number of total instructions, including computation instructions and branching instructions, is at least N.

Let us now give a less informal definition. We will concentrate on decision tree models for decision problems.

Definition An *algebraic decision tree* on a set of n variables $(x_1, \cdots, x_n)$ is a binary tree such that each vertex of it is labeled with a statement satisfying the following conditions.

1. Every non-leaf statement L is of the form

   if $f(x_1, \cdots, x_n) \bowtie 0$ **then** goto L_i **else** goto L_j

   where $f(x_1, \cdots, x_n)$ is a polynomial of x_1, $\cdots$, x_n, and $\bowtie$ is any comparison relation from the set $\{=, >, <, \leq, \geq\}$. The statements L_i and L_j are the children of the statement L;

2. Every leaf statement is either a YES or a NO answer to the decision problem.

If all polynomials at non-leaf vertices of an algebraic decision tree are linear polynomials, then we call it a *linear decision tree*.

Let P be a decision problem with inputs of n real numbers. Then P corresponds to a subset W of the n-dimensional space E^n such that a point $(x_1, \cdots, x_n) \in E^n$ is in W if and only if the answer of the problem P to the input $(x_1, \cdots, x_n)$ is YES. Let T be an algebraic decision tree that "solves" the problem P in the following way: for any point $p = (x_1, \cdots, x_n) \in E^n$, the answer of P to the input p is YES if and only if when we feed the root of the algebraic decision tree T with the input p, then eventually we are led to a YES leaf v in the tree T by following the decisions made on the non-leaf vertices on the path from the root to the leaf v in the tree T. In this case, we also say that the algebraic decision tree T *accepts* the subset W in E^n.

8.2 Algebraic decision trees

The *depth* of a tree is the length of the longest path from the root to a leaf in the tree. It is easy to see that the depth of an algebraic decision tree corresponds to the worst case time complexity of the tree. Therefore, to derive a lower bound on the worst case time complexity of a problem P, it suffices to derive a lower bound on the depth of the algebraic decision trees that solve the problem P. In this section, we show a lower bound on the depth of an algebraic decision tree, assuming that we know the number of connected components of the corresponding subset in E^n the tree accepts.

We first observe the following simple lemma.

Lemma 8.2.1 *The depth of a binary tree with m leaves is at least* $\lceil \log m \rceil$.

Now suppose that P is a decision problem, and let W be the subset of E^n that corresponds to the YES-instances of the problem P in E^n. That is, a point $p = (x_1, \cdots, x_n) \in E^n$ is in the set W if and only if the solution of the problem P to the input p is YES. Let T be an algebraic decision tree that solves P, or equivalently that accepts the subset W.

Suppose in some way that the number $\#W$ of the connected components of the set W is known. What can we say about the depth of the algebraic decision trees that accept W? We answer this question first for the linear decision tree model, then we extend the result to the algebraic decision tree model.

Theorem 8.2.2 *Let W be a subset of E^n, and let T be a linear decision tree of n variables that accepts the set W. Then the depth of T is at least* $\lceil \log(\#W) \rceil$.

PROOF. Every path from the root to a leaf l in T corresponds to a sequence of conditions:

$$\begin{aligned} f_i(x_1, \cdots, x_n) &= 0 \qquad i = 1, \cdots, m_1 \\ g_j(x_1, \cdots, x_n) &> 0 \qquad j = 1, \cdots, m_2 \\ h_k(x_1, \cdots, x_n) &\geq 0 \qquad k = 1, \cdots, m_3 \end{aligned} \tag{8.1}$$

which are the testings occurring on the path. Each of these functions is a linear polynomial since we assume that the tree T is a linear decision tree. If we feed the root with a point $p = (x_1, \cdots, x_n)$ in E^n, then the point p eventually goes to the leaf l if and only if the coordinates $(x_1, \cdots, x_n)$ of p

satisfies all the conditions in (8.1). Therefore, the leaf l corresponds to a set S_l of points in E^n that satisfy all the conditions in (8.1). Thus the set S_l is the intersection of the hyperplanes, the open halfspaces, and the closed halfspaces represented by these conditions. By the discussion we gave in the last section, we conclude that the set S_l is convex. Consequently, S_l is connected.

Now let l be a YES leaf, then the corresponding set S_l is a subset of the set W. Since S_l is connected, by the definition of a connected component that a connected component of W is a maximal connected subset of W, S_l must be entirely contained in a single connected component of W. Therefore, each YES leaf of the linear decision tree T only accepts points in a single connected component of W. Since W has $\#W$ connected components, and each point of W should be accepted by some YES leaf of T, we conclude that the tree T contains at least $\#W$ YES leaves. Consequently, the number of leaves of T is at least $\#W$. Now by Lemma 8.2.1, the depth of the linear decision tree T is at least $\lceil \log(\#W) \rceil$. □

The linear decision tree model seems too restricted (people would never be happy if you tell them that their computers cannot do multiplication). It is desired to extend the result above for the linear decision tree model to the algebraic decision tree model. Let us see what is the obstacle to such an extension. Suppose that an algebraic decision tree T accepts a subset W of E^n. Each YES leaf l of T accepts a subset S_l of the set W. The subset S_l is again the intersection of the subsets presented by the conditions appearing on the path from the root to the leaf l in the tree T. However, since the polynomials at the non-leaves of T are not necessarily linear polynomials, the set S_l may be *not* connected.[1] Therefore, each leaf now can accept points from many different connected components of W. Suppose that each leaf can accept points from at most c connected components, then the only thing we can conclude is that there are at least $\#W/c$ YES-leaves. Therefore, by Lemma 8.2.1 again, we conclude that the depth of T is at least $\lceil \log(\#W/c) \rceil$. However, if the number c is of the same order as $\#W$, then we will obtain a trivial constant lower bound on the depth of the algebraic decision tree T.

However, if the number c above is bounded by some constant, then $\lceil \log(\#W/c) \rceil$ will have the same order as $\lceil \log(\#W) \rceil$, thus again we ob-

[1]For example, in the space E^2, even a single condition with a non-linear polynomial

$$x^2 - y^2 \leq 1$$

defines a non-connected area.

tain a nontrivial lower bound on the depth of the algebraic decision tree T. Therefore, we would like to know under what conditions the number c, i.e., the maximum number of connected components whose points can be accepted by a single leaf of an algebraic decision tree, can be bounded. Here is a condition:

Theorem 8.2.3 (Milnor-Thom) *Let S be the set of points in the n-dimensional Euclidean space E^n defined by the conditions*

$$f_i(x_1, \cdots, x_n) = 0 \qquad i = 1, \cdots, h \tag{8.2}$$

where all f_i, $1 \leq i \leq h$, are polynomials of degree at most d. Then the number $\#S$ of connected components of the set S is bounded by $d(2d-1)^{n-1}$, a number that is independent of the number of the conditions in (8.2).

The above theorem is a deep result in algebraic geometry. However, the idea of the theorem is fairly intuitive: a polynomial of small degree defines a subset of "simple shape" in a Euclidean space, and the intersection of "simple-shape" subsets in a Euclidean space cannot have a very complicated shape, that is, it cannot have many pieces of connected components.

Unfortunately, Milnor-Thom Theorem cannot be used directly to our algebraic decision trees: it only covers the case of equalities, while our algebraic decision trees also have inequalities. Thus it is necessary to extend Milnor-Thom Theorem to cover inequalities.

Lemma 8.2.4 *Let S be the set of points in the n-dimensional Euclidean space E^n defined by the following conditions*

$$\begin{aligned} f_i(x_1, \cdots, x_n) &= 0 \qquad & i &= 1, \cdots, m_1 \\ g_j(x_1, \cdots, x_n) &> 0 \qquad & j &= 1, \cdots, m_2 \\ h_k(x_1, \cdots, x_n) &\geq 0 \qquad & k &= 1, \cdots, m_3 \end{aligned} \tag{8.3}$$

where all f_i, g_j, and h_k, $1 \leq i \leq m_1$, $1 \leq j \leq m_2$, and $1 \leq k \leq m_3$, are polynomials of degree at most d. Then the number of connected components of the set S is bounded by $d(2d-1)^{n+m_2+m_3-1}$.

PROOF. Suppose that S has r distinct connected components C_i, $1 \leq i \leq r$, arbitrarily pick a point $p_i = (x_1^{(i)}, \cdots, x_n^{(i)})$ from the connected component C_i, $1 \leq i \leq r$. Now consider the rm_2 real numbers

$$g_j(x_1^{(i)}, \cdots, x_n^{(i)}) \qquad 1 \leq j \leq m_2, \;\; 1 \leq i \leq r$$

Note that all these rm_2 real numbers are positive since all these points $p_i = (x_1^{(i)}, \cdots, x_n^{(i)})$, $1 \le i \le r$, are in S. Let ϵ be the smallest real number in these rm_2 real numbers. Note that $\epsilon > 0$.

Consider the set S' in E^n defined by the following conditions

$$\begin{aligned} f_i(x_1, \cdots, x_n) &= 0 \qquad i = 1, \cdots, m_1 \\ g_j(x_1, \cdots, x_n) - \epsilon &\ge 0 \qquad j = 1, \cdots, m_2 \\ h_k(x_1, \cdots, x_n) &\ge 0 \qquad k = 1, \cdots, m_3 \end{aligned} \tag{8.4}$$

We claim that the number of connected components of the set S' is at least as large as the number of connected components of the set S. In fact, the set S' is a subset of the set S since a point satisfying the conditions in (8.4) obviously satisfies the conditions in (8.3). Therefore, no two connected components of the set S can be "merged" into a single connected components of the set S'. Moreover, no connected components of S completely disappear in S', since for each connected component C_i of S, at least the chosen point p_i satisfies all conditions in (8.4), by the definition of the number ϵ. Therefore, instead of bounding the number of connected components of the set S, which is defined by the equalities, the open inequalities, and the closed inequalities of (8.3), we can work on a bound of the number of connected components of the set S', which is defined by the equalities and the closed inequalities of (8.4).

The technique of converting a closed inequality into an equality is well-known in linear programming. For the set S' defined by the conditions in (8.4), we introduce $m_2 + m_3$ new variables y_j and z_k, $1 \le j \le m_2$, $1 \le k \le m_3$, and construct the following $m_1 + m_2 + m_3$ conditions with $n + m_2 + m_3$ variables x_i, y_j and z_k, $1 \le i \le n$, $1 \le j \le m_2$, and $1 \le k \le m_3$:

$$\begin{aligned} f_i(x_1, \cdots, x_n) &= 0 \qquad i = 1, \cdots, m_1 \\ g_j(x_1, \cdots, x_n) - \epsilon - y_j^2 &= 0 \qquad j = 1, \cdots, m_2 \\ h_k(x_1, \cdots, x_n) - z_k^2 &= 0 \qquad k = 1, \cdots, m_3 \end{aligned} \tag{8.5}$$

Let S'' be the subset of $E^{n+m_2+m_3}$ that is defined by the conditions in (8.5). It is easy to see that the number $\#S''$ of connected components of S'' is the same as the number $\#S'$ of connected components of S', which is at least as large as the number $\#S$ of connected components of the set S. By Milnor-Thom Theorem, the number $\#S''$ is bounded by $d(2d-1)^{n+m_2+m_3-1}$. Therefore, the number $\#S$ is also bounded by $d(2d-1)^{n+m_2+m_3-1}$. This completes the proof. $\square$

Now similar as the proof for the case of linear decision trees, we can prove a lower bound on the depth of general algebraic decision trees.

Definition An algebraic decision tree is of *order d* if all polynomials occurring in the non-leaves of the tree have degree at most d.

Theorem 8.2.5 (Ben-or's Theorem) *Let W be a subset of E^n and let d be a fixed integer. Then any order d algebraic decision tree T that accepts W has depth at least $\Omega(\log \#W - n)$.*

PROOF. Suppose that T is an order d algebraic decision tree that accepts the set W. Let l be a YES leaf of the tree T that is associated with the following conditions:

$$\begin{aligned} f_i(x_1, \cdots, x_n) &= 0 \qquad i = 1, \cdots, m_1 \\ g_j(x_1, \cdots, x_n) &> 0 \qquad j = 1, \cdots, m_2 \\ h_k(x_1, \cdots, x_n) &\geq 0 \qquad k = 1, \cdots, m_3 \end{aligned} \tag{8.6}$$

where all f_i, g_j, and h_k, $1 \leq i \leq m_1$, $1 \leq j \leq m_2$, and $1 \leq k \leq m_3$, are polynomials of degree at most d.

Let S_l be the set accepted by the leaf l, that is, S_l is the set in E^n defined by the conditions in (8.6). Since $m_1 + m_2 + m_3$ is the length of the path from the root of T to the leaf, $m_1 + m_2 + m_3$ is bounded by the depth h of the algebraic decision tree T.

By Lemma 8.2.4, the number of connected components of the set S_l is bounded by $d(2d-1)^{n+m_2+m_3-1}$, which is bounded by $d(2d-1)^{n+h-1}$. Now since the set W has $\#W$ connected components, and each point of W must be accepted by some leaf of T, we conclude that the algebraic decision tree T has at least $\#W/(d(2d-1)^{n+h-1}$ leaves. By Lemma 8.2.1, the depth h of the tree T is at least

$$\log(\frac{\#W}{d(2d-1)^{n+h-1}})$$

From this, we get

$$h \geq \log(\#W) - \log d - (n + h - 1)\log(2d - 1)$$

That is

$$h \geq \frac{1}{1 + \log(2d-1)}(\log(\#W) - n\log(2d-1) + \log((2d-1)/d))$$

When the number d is a fixed constant, we get $h = \Omega(\log(\#W) - n)$. $\square$

Therefore, to derive the lower bound of a problem, we may consider the corresponding set W in the space E^n for all n, then compute the number of connected components of the set W. We will use this technique to derive non-trivial lower bounds for several problems.

8.3 Proving lower bounds directly

With the lower bound on the depth of algebraic decision trees obtained in the last section, now we are ready to derive a few lower bounds for problems, including the EXTREME-POINTS, ELEMENT-UNIQUENESS, UNIFORM-GAP, and SET-DISJOINTNESS.

The basic idea is as follows: given a decision problem $\mathcal{P}$, we try to formulate the YES-instances of $\mathcal{P}$ with n parameters into a subset W of the n-dimensional Euclidean space E^n. Then we derive a lower bound B on the number of connected components of the subset W. Now by Ben-or's theorem, the logarithm of B gives us a lower bound on the depth of algebraic decision trees that solve the problem $\mathcal{P}$, that is in consequence a lower bound on the computational time of the algebraic decision trees solving the problem $\mathcal{P}$.

8.3.1 Element uniqueness

We start with a simplest example, the problem of ELEMENT-UNIQUENESS. The problem is formally defined as follows.

ELEMENT-UNIQUENESS

Input: A set S of n real numbers.

Question: Are there two numbers in S equal?

We derive a lower bound for the problem ELEMENT-UNIQUENESS by using Ben-or's theorem (Theorem 8.2.5) directly.

Theorem 8.3.1 *Any bounded order algebraic decision tree that solves the problem ELEMENT-UNIQUENESS runs in time at least* $\Omega(n \log n)$.

PROOF. Adopting the standard technique, we first consider the number of connected components of the following set in the n-dimensional Euclidean space.

$$W = \{(x_1, \cdots, x_n) \mid \text{all } x_i\text{'s are distinct}\}$$

A point $(x_1, \cdots, x_n)$ in n-dimensional Euclidean space is a YES-instance of the problem ELEMENT-UNIQUENESS if and only if the point belongs to the set W.

Fix a point $(x_1, \cdots, x_n)$ in n-dimensional Euclidean space such that all x_i's are distinct. Consider the $n!$ points in the n-dimensional Euclidean space obtained by permuting $(x_1, \cdots, x_n)$.

$$P_\sigma = (x_{\sigma(1)}, \cdots, x_{\sigma(n)}) \qquad \sigma \text{ is a permutation of } (1, \cdots, n)$$

Clearly, all these $n!$ points are in the set W. We claim that no two of these $n!$ points share the same connected component of W. In fact, suppose that σ and σ' are two different permutations of $(1, \cdots, n)$ and that the points P_σ and $P_{\sigma'}$ are in the same connected component of W, then there is a continuous curve C in W connecting P_σ and $P_{\sigma'}$. That is, we can find n continuous functions $f_i(x)$, $1 \leq i \leq n$, such that

$$f_i(0) = x_{\sigma(i)} \quad \text{and} \quad f_i(1) = x_{\sigma'(i)} \quad \text{for } 1 \leq i \leq n$$

Since σ and σ' are different permutations of $(1, \cdots, n)$, we can find an index k such that $x_{\sigma(k)}$ is the smallest number such that $x_{\sigma(k)} \neq x_{\sigma'(k)}$. Suppose that $x_{\sigma(k)} = x_{\sigma'(h)}$ for some index $h \neq k$, then we also have $x_{\sigma(h)} \neq x_{\sigma'(h)}$. So

$$x_{\sigma(k)} < x_{\sigma(h)} \quad \text{and} \quad x_{\sigma'(k)} > x_{\sigma'(h)}$$

by the definition of the index k.

Since $f_k(x)$ and $f_h(x)$ are continuous functions and

$$f_k(0) = x_{\sigma(k)} \quad f_k(1) = x_{\sigma'(k)} \quad f_h(0) = x_{\sigma(h)} \quad f_h(1) = x_{\sigma'(h)}$$

Thus

$$f_k(0) < f_h(0) \quad \text{and} \quad f_k(1) > f_h(1)$$

there must be a real number r in the interval $(0, 1)$ such that $f_k(r) = f_h(r)$. However, by our assumption, the point $(f_1(r), f_2(r), \cdots, f_n(r))$ on the curve C is in the set W, so all numbers $f_i(r)$ are distinct. In particular, the numbers $f_k(r)$ and $f_h(r)$ are distinct. This contradiction proves that each point P_σ is in a different connected component of the set W.

Thus the set W has at least $n!$ connected components. So $\#W \geq n!$. Now since

$$n! = 1 \cdot 2 \cdot \cdots \cdot n > \frac{n}{2} \cdot (\frac{n}{2}+1) \cdot \cdots \cdot n \geq (\frac{n}{2})^{\frac{n}{2}}$$

So we have

$$\log(\#W) \geq \log(n!) \geq \log{(\frac{n}{2})}^{\frac{n}{2}} = \frac{n}{2}\log(\frac{n}{2}) = \Omega(n \log n)$$

By Ben-or's theorem (Theorem 8.2.5), any bounded order algebraic decision tree that solves the problem ELEMENT-UNIQUENESS runs in time at least

$$\Omega(\log(\#W) - n) = \Omega(n \log n)$$

□

8.3.2 Uniform gap

Given a set S of real numbers, we say that numbers x and y are *consecutive* if y is the smallest number in $S - \{x\}$ that is not less than x.

The UNIFORM ϵ-GAP problem is stated as follows, where ϵ is a fixed real number.

UNIFORM ϵ-GAP

Input: A set S of n real numbers.

Question: Are the distances between consecutive numbers in S uniformly equal to ϵ?

Theorem 8.3.2 *Any bounded order algebraic decision tree that solves the problem UNIFORM ϵ-GAP runs in time at least $\Omega(n \log n)$.*

PROOF. The proof is quite similar to the proof of Theorem 8.3.1.

Consider the following set in the n-dimensional Euclidean space

$$W = \{(x_1, \cdots, x_n) \mid (x_1, \cdots, x_n) \text{ is a YES-instance of UNIFORM } \epsilon\text{-GAP}\}$$

Thus a point $(x_1, \cdots, x_n)$ in n-dimensional Euclidean space is in the set W if and only if there is a permutation σ of $(1, \cdots, n)$ such that $x_{\sigma(i)} + \epsilon = x_{\sigma(i+1)}$, for $1 \leq i \leq n-1$.

Fix a point $(x_1, \cdots, x_n)$ in n-dimensional Euclidean space such that $x_i + \epsilon = x_{i+1}$, for all $1 \le i \le n-1$. Consider the $n!$ points in the n-dimensional Euclidean space obtained by permuting $(x_1, \cdots, x_n)$

$$P_\sigma = (x_{\sigma(1)}, \cdots, x_{\sigma(n)}) \qquad \sigma \text{ is a permutation of } (1, \cdots, n)$$

Clearly, all these $n!$ points are in the set W. We claim that no two of these $n!$ points share the same connected component of W. In fact, suppose that σ and σ' are two different permutations of $(1, \cdots, n)$ and that the points P_σ and $P_{\sigma'}$ are in the same connected component of W, then there is a continuous curve C in W connecting P_σ and $P_{\sigma'}$. That is, we can find n continuous functions $f_i(x)$, $1 \le i \le n$, such that

$$f_i(0) = x_{\sigma(i)} \quad \text{and} \quad f_i(1) = x_{\sigma'(i)} \quad \text{for } 1 \le i \le n$$

Exactly the same as in the proof of Theorem 8.3.1, we can find two indices k and h such that

$$f_k(0) < f_h(0) \quad \text{and} \quad f_k(1) > f_h(1)$$

So there exists a real number r in the interval $(0, 1)$ such that $f_k(r) = f_h(r)$. But then the point $(f_1(r), f_2(r), \cdots, f_n(r))$ on the curve C cannot be in the set W since the distance between the numbers $f_k(r)$ and $f_h(r)$ is less than ϵ. This contradiction proves that the set W has at least $n!$ connected components. By Ben-or's theorem (Theorem 8.2.5), any bounded order algebraic decision tree that solves the problem UNIFORM ϵ-GAP runs in time at least

$$\Omega(\log(\#W) - n) = \Omega(n \log n)$$

□

8.3.3 Set disjointness

The third problem we study is the following problem.

SET-DISJOINTNESS

Given two sets $X = \{x_1, \cdots, x_n\}$ and $Y = \{y_1, \cdots, y_n\}$ of real numbers, do they have an empty intersection?

For each instance (X, Y) of the problem SET-DISJOINTNESS, where $X = \{x_1, x_2, \cdots, x_n\}$ and $Y = \{y_1, y_2, \cdots, y_n\}$, we associate it with a point in the $2n$-dimensional Euclidean space E^{2n}:

$$(x_1, y_1, x_2, y_2, \cdots, x_n, y_n)$$

This mapping gives us a one-to-one correspondence between the points in E^{2n} and the instance of size n of the problem SET-DISJOINTNESS if we suppose that the sets X and Y are "ordered sets". (We call (X, Y) an instance of size n if both the sets X and Y contain n real numbers.) Let W be the subset of E^{2n} that corresponds to the YES-instances of size n of the problem SET-DISJOINTNESS. We first prove that W has at least $n!$ connected components.

Fix two sets $X = (x_1, x_2, \cdots, x_n)$ and $Y = (y_1, y_2, \cdots, y_n)$ of real numbers such that

$$x_1 > y_1 > x_2 > y_2 > \cdots > x_n > y_n$$

Then (X, Y) is a YES-instance of the problem SET-DISJOINTNESS which corresponds to a point

$$p = (x_1, y_1, x_2, y_2, \cdots, x_n, y_n)$$

in the $2n$-dimensional Euclidean space E^{2n}. Thus the point p is in the set W. Consider the $n!$ points in E^{2n} that are obtained by permuting the n components in p with even indices. That is, consider the $n!$ points

$$p_\sigma = (x_1, y_{\sigma(1)}, x_2, y_{\sigma(2)}, \cdots, x_n, y_{\sigma(n)})$$

where σ is a permutation of $(1, 2, \cdots, n)$.

Clearly, all these $n!$ points p_σ are in the set W. We claim that no two of these $n!$ points share the same connected component of W. In fact, suppose that σ and σ' are two different permutations of $(1, \cdots, n)$ and that the points p_σ and $p_{\sigma'}$ are in the same connected component of W, then there is a continuous curve C in W connecting p_σ and $p_{\sigma'}$. That is, we can find $2n$ continuous functions $h_i(t)$, $f_i(t)$, $1 \leq i \leq n$, such that

$$h_i(0) = h_i(1) = x_i \qquad \text{for } 1 \leq i \leq n$$

$$f_i(0) = y_{\sigma(i)} \quad \text{and} \quad f_i(1) = y_{\sigma'(i)} \quad \text{for } 1 \leq i \leq n$$

Since σ and σ' are different permutations of $(1, \cdots, n)$, we can find an index k such that $y_{\sigma(k)}$ is the smallest number in $(y_1, \cdots, y_n)$ such that $y_{\sigma(k)} \neq y_{\sigma'(k)}$. Since $y_{\sigma(k)}$ is the smallest in $(y_1, \cdots, y_n)$, we have $y_{\sigma(k)} < y_{\sigma'(k)}$. By the definition of our point p, there must be an x_l such that

$$y_{\sigma(k)} < x_l < y_{\sigma'(k)}$$

Now consider the function $F(t) = h_l(t) - f_k(t)$, we have

$$F(0) = h_l(0) - f_k(0) = x_l - y_{\sigma(k)} > 0$$

and

$$F(1) = h_l(1) - f_k(1) = x_l - y_{\sigma'(k)} < 0$$

The function $F(t)$ is continuous because $h_l(t)$ and $f_k(t)$ are. Therefore, there is a real number α such that $0 < \alpha < 1$ and

$$F(\alpha) = h_l(\alpha) - f_k(\alpha) = 0$$

That is, $h_l(\alpha) = f_k(\alpha)$. However, by our assumption, the point

$$p' = (h_1(\alpha), f_1(\alpha), h_2(\alpha), f_2(\alpha), \cdots, h_n(\alpha), f_n(\alpha))$$

on the curve C is in the set W, so every component $h_i(\alpha)$ is distinct from any component $f_j(\alpha)$ in the point p'. In particular, the numbers $h_l(\alpha)$ and $f_k(\alpha)$ are distinct. This contradiction proves that each point P_σ is in a different connected component of the set W.

Thus the set W has at least $n!$ connected components. So $\#W \geq n!$. By Ben-or's theorem (Theorem 8.2.5), any bounded order algebraic decision tree that solves the problem SET-DISJOINTNESS runs in time at least

$$\Omega(\log(\#W) - n) = \Omega(n \log n)$$

The above discussion gives the following theorem.

Theorem 8.3.3 *Any bounded order algebraic decision tree that solves the problem SET-DISJOINTNESS runs in time at least* $\Omega(n \log n)$.

8.3.4 Extreme points

The above three problems are combinatorial problems. In this subsection, we derive a lower bound for a geometric problem that is called EXTREME-POINTS problem, which is closed related to the problem CONVEX-HULL. The proof is again similar to those given above, though slightly more complicated.

Definition Let S be a set of points in the plane E^2. A point $p \in S$ is an *extreme point* of S if p is on the boundary of the convex hull *CH(S)*, and p

is not an interior point of any boundary edge of *CH(S)*.

The CONVEX-HULL problem is to find all extreme points of a given set S in the counterclockwise order with respect to some interior point of *CH(S)*. The following decision problem has an obvious relationship with the CONVEX-HULL problem.

EXTREME-POINTS

Input: A list S of n points in the plane E^2.
Output: Are all points in S extreme points of S?

The EXTREME-POINTS problem seems "simpler" than the CONVEX-HULL problem since it is not required to check the counterclockwise order of the extreme points on the boundary of the convex hull *CH(S)*. We will see, however, that it takes the same amount of time to solve the EXTREME-POINTS problem as to solve the CONVEX-HULL problem.

A point p in the plane E^2 can be uniquely represented by a tuple of two real numbers $p = (x, y)$, where x and y are the x- and y- coordinates of p, respectively. Similarly, an ordered list of $2n$ points $(p_1, \cdots, p_{2n})$ in the plane E^2 can be uniquely represented by a tuple of $4n$ real numbers $(p_1, \cdots, p_{2n}) = (x_1, \cdots, x_{4n})$, where $p_i = (x_{2i-1}, x_{2i})$, for $1 \le i \le n$. Therefore, each $2n$-point instance $(p_1, \cdots, p_{2n})$ for the EXTREME-POINTS problem uniquely corresponds to a point in the $4n$-dimensional space E^{4n}. Conversely, any point $(x_1, \cdots, x_{4n})$ in the space E^{4n} can be regarded uniquely as a $2n$-point instance for the EXTREME-POINTS problem, if we let $p_i = (x_{2i-1}, x_{2i})$, for $1 \le i \le 2n$. Therefore, the set of $2n$-point YES-instances of the EXTREME-POINTS problem is a subset of the $4n$-dimensional space E^{4n}. Note that a set of $2n$ points $S = \{p_1, \cdots, p_{2n}\}$ in the plane E^2 can correspond to up to $(2n)!$ different ordered lists, thus $(2n)!$ different points in the space E^{4n}, if we consider all permutations of these $2n$ points. Thus any set of $2n$ points in the plane makes $(2n)!$ different instances for the EXTREME-POINTS problem.

Lemma 8.3.4 *Let W be the subset of the space E^{4n} that corresponds to the set of $2n$-point YES-instances for the EXTREME-POINTS problem. Then W has at least $n!$ connected components.*

PROOF. We construct $n!$ points in the set W and prove that no two of these points are contained in the same connected component of the set W.

Let $I = (p_1, q_1, p_2, q_2, \cdots, p_n, q_n)$ be a counterclockwise sequence of $2n$ distinct extreme points of a convex polygon. Then $I \in E^{4n}$ is a point in the set W. Consider the following $n!$ different sequences of $2n$-point instances of the problem EXTREME-POINTS:

$$I_i = (p_1, q_1^{(i)}, p_2, q_2^{(i)}, \cdots, p_n, q_n^{(i)}) \qquad i = 1, \cdots, n! \tag{8.7}$$

where each sequence $(q_1^{(i)}, q_2^{(i)}, \cdots, q_n^{(i)})$ is a permutation of the sequence $(q_1, q_2, \cdots, q_n)$. Each instance I_i corresponds to a point in the space E^{4n}. Since each instance I_i shares the same set of points $\{p_1, q_1, \cdots, p_n, q_n\}$ in the plane E^2 with the instance I and the I is a YES-instance of the problem EXTREME-POINTS, the instance I_i should also be a YES-instance of the problem EXTREME-POINTS (i.e., every point in I_i is an extreme point). That is, the instance I_i, for $1 \le i \le n!$, corresponds to a point in the set W.

Now we prove that any pair of instances (8.7) are in two different connected components of the set W. Suppose otherwise, there are two instances I_i and I_j in (8.7) that are two points in E^{4n} and in the same connected component of the set W. Then there is a continuous curve $\mathcal{C}$ in E^{4n} that adjoins the two points I_i and I_j. More precisely, there are $2n$ continuous functions on the interval [0, 1]

$$S_1(t), T_1(t), S_2(t), T_2(t), \cdots, S_n(t), T_n(t)$$

such that $S_h(0) = S_h(1) = p_h$, and $T_h(0) = q_h^{(i)}$ and $T_h(1) = q_h^{(j)}$, for $h = 1, 2, \cdots, n$, and for all $t \in [0, 1]$, the point

$$(S_1(t), T_1(t), S_2(t), T_2(t), \cdots, S_n(t), T_n(t))$$

is in the set W.

Since I_i and I_j are two different instances in (8.7), there must be an index k such that $q_k^{(i)}$ and $q_k^{(j)}$ are different points in the set $\{q_1, q_2, \cdots, q_n\}$. Without loss of generality, suppose the $q_k^{(i)} = q_1$ then $q_k^{(j)} \neq q_1$. Then the signed triangle area $\triangle(p_1 q_k^{(i)} p_2)$ is positive while the signed triangle area $\triangle(p_1 q_k^{(j)} p_2)$ is negative, because the only point q in $\{q_1, q_2, \cdots, q_n\}$ that makes $p_1 q p_2$ a left turn is the point q_1. Since $S_1(t)$, $T_k(t)$, and $S_2(t)$ are continuous functions of t, the signed triangle area $\triangle(S_1(t)T_k(t)S_2(t))$ is also a continuous function of the variable t. Moreover, since we have $\triangle(S_1(0)T_k(0)S_2(0)) = \triangle(p_1 q_k^{(i)} p_2) > 0$ and $\triangle(S_1(1)T_k(1)S_2(1)) = \triangle(p_1 q_k^{(j)} p_2) < 0$, there must be a number $t_0 \in (0,1)$ such that $\triangle(S_1(t_0)T_k(t_0)S_2(t_0)) = 0$. That is, on the curve $\mathcal{C}$ in E^{4n}, there is a point

$$I_0 = (S_1(t_0), T_1(t_0), S_2(t_0), T_2(t_0), \cdots, S_n(t_0), T_n(t_0))$$

such that $S_1(t_0)$, $T_k(t_0)$, and $S_2(t_0)$ are three co-linear points in the plane E^2. Therefore, at least one point in the set $\{S_1(t_0), T_k(t_0), S_2(t_0)\}$ is not an extreme point of the set of $2n$ points

$$\{S_1(t_0), T_1(t_0), S_2(t_0), T_2(t_0), \cdots, S_n(t_0), T_n(t_0)\}$$

Thus the instance

$$I_0 = (S_1(t_0), T_1(t_0), S_2(t_0), T_2(t_0), \cdots, S_n(t_0), T_n(t_0))$$

should be a NO-instance of the problem EXTREME-POINTS, so $I_0 \notin W$. This contradicts the assumption that the entire curve $\mathcal{C}$ is contained in the set W. The contradiction proves that no two points in (8.7) can be in the same connected component of the set W. Since there are $n!$ different points in (8.7), we conclude that the set W has at least $n!$ connected components. $\square$

We say that an algebraic decision has a *bounded order* if the order of the tree is bounded by a constant that is independent of the input size of the tree. Now combining the lemma above with Theorem 8.2.5, we easily obtained a lower bound for the problem EXTREME-POINTS.

Theorem 8.3.5 *Any bounded order algebraic decision tree that solves the problem EXTREME-POINTS runs in time at least $\Omega(n \log n)$ on an input of n points in the plane.*

PROOF. Remember that we are working on worst case time complexity. Therefore, we only have to show that for some integer n, the theorem is true.

Let T be an order d algebraic decision tree that solves the problem EXTREME-POINTS with inputs of $n = 2m$ points in the plane. Thus the number of input variables of the tree T is $2n = 4m$. Let W be the set of points in the space E^{4m} that are the YES-instances of the problem EXTREME-POINTS. Then the algebraic decision tree T accepts the set W. By Lemma 8.3.4, W has at least $m!$ connected components. Now by Theorem 8.2.5, the depth of the tree T is at least $\Omega(\log(\#W) - 4m)$. Since we have

$$m! = 1 \cdot 2 \cdots m > \frac{m}{2} \cdot (\frac{m}{2} + 1)(\frac{m}{2} + 2) \cdots m > (\frac{m}{2})^{\frac{m}{2}}$$

Therefore,

$$\log(\#W) \geq \log(m!) \geq \log((\frac{m}{2})^{\frac{m}{2}}) = \frac{m}{2}\log(\frac{m}{2})$$

Now the depth of the algebraic decision tree T is $\Omega(\log(\#W) - 4m) = \Omega(m \log m) = \Omega(n \log n)$. $\square$

8.4 Deriving lower bounds by reductions

The techniques used in the last section for deriving lower bounds on problems seem impressive. Such elegant techniques were developed and such deep mathematics results were used in deriving the lower bounds. It is not clear how these techniques can be generalized to deriving lower bounds for general geometric problems. Fortunately, we do not have to do this very often. For some geometric problems, the lower bounds can be derived by "reducing" the problems to some other problems for which the lower bounds are known.

Let us first review the concept of problem reductions. We say that a problem P can be *reduced* to a problem P' in time $O(t(n))$, express it as $P \propto_{t(n)} P'$, if there is an algorithm $\mathcal{T}$ solving the problem P in the following way.

1. For any input x of size n to the problem P, convert x in time $O(t(n))$ into an input x' to the problem P';
2. Call a subroutine to solve the problem P' on input x';
3. Convert in time $O(t(n))$ the solution to the problem P' on input x' into a solution to the problem P on input x.

We have seen in Chapter 6 that the technique of reduction is very useful in designing efficient algorithms for geometric problems. In this section, we will study how to use this technique to derive lower bounds for geometric problems. The following theorem plays an important role in our discussion.

Theorem 8.4.1 *Suppose that a problem P is reduced to a problem P' in linear time*

$$P \propto_n P'$$

If it is known that solving the problem P takes at least $\Omega(T(n))$ time, then solving the problem P' also takes at least $\Omega(T(n))$ time. In other words, a *lower bound of the time complexity of the problem P is also a lower bound of the time complexity of the problem P'.*

PROOF. Suppose otherwise, the problem P' can be solved in time $T_1(n)$, with $T_1(n) = o(T(n))$. Then by Lemma **??**, the problem P can also be solved in time $O(T_1(n))$. But this would imply that the problem P can be solved in time $o(T(n))$, contradicting our assumption that $T(n)$ is a lower bound on the time complexity of the problem P. $\square$

We first use Theorem 8.4.1 to derive a lower bounds for the problem CONVEX-HULL.

Theorem 8.4.2 *Any bounded order algebraic decision tree that constructs the convex hull for a set of points in the plane runs in time at least $\Omega(n \log n)$ on an input of n points in the plane.*

PROOF. By Theorem 8.3.5, any bounded order algebraic decision tree that solves the problem EXTREME-POINTS runs in time at least $\Omega(n \log n)$. According to Theorem 8.4.1, it will suffice to prove the theorem by showing that

$$\textit{EXTREME-POINTS} \propto_n \textit{CONVEX-HULL}$$

We give this reduction by the following algorithm.

```
Algorithm     REDUCTION I
     { Reduce the problem EXTREME-POINTS to the problem
       CONVEX-HULL. }

      BEGIN
      1.   Given an input S of the problem EXTREME-POINTS,
           where S is a set of n points in the plane, pass
           the set S directly to the problem CONVEX-HULL;
      2.   The solution of CONVEX-HULL to the set S is the
           convex hull CH(S) of the set S.  Pass CH(S)
           back to the problem EXTREME-POINTS;
      3.   If the convex hull CH(S) has n hull vertices,
           and no hull vertex is at the middle of the
           straight line segment passing through its two
           neighbors, then the answer of the problem
           EXTREME-POINTS to the input S is YES;
           Otherwise, the answer should be NO.
```

```
END.
```

Since both Step 1 and Step 3 take at most time $O(n)$, the above algorithm is a linear time reduction of the problem EXTREME-POINTS to the problem CONVEX-HULL. □

Thus constructing convex hulls of sets of points in the plane takes time at least $\Omega(n \log n)$. This result implies that many algorithms we discussed before for construction of convex hulls, including Graham Scan, MergeHull, Kirkpatrick-Seidel algorithm, are optimal.

As we have discussed in the last chapter, the problem CONVEX-HULL can be reduced to the problem SORTING in time $O(n)$. By Theorem 8.4.1 and Theorem 8.4.2, we also obtain

Theorem 8.4.3 *Any bounded order algebraic decision tree sorting n real numbers runs in time at least $\Omega(n \log n)$.*

This theorem is stronger than the one we got in Algorithm Analysis. In Algorithm Analysis, it is proved that a linear decision tree model that sorts runs in time $\Omega(n \log n)$. On the other hand, Theorem 8.4.3 claims that even the computation model is allowed to do multiplication, it still needs at least $\Omega(n \log n)$ time to sort.

We have seen that many proximity problems can be solved in time $O(n \log n)$. Now we prove that our algorithms for these problems are in fact optimal.

We start with the two whose lower bound is easily obtained from the problem SORTING: EUCLIDEAN-MINIMUM-SPANNING-TREE (EMST) and TRIANGULATION. For this, we first prove a simple lemma.

Lemma 8.4.4 *Let S be a set of n real numbers x_1, x_2, $\cdots$, x_n. If S is given in such a way that for each $1 \leq k \leq n$, the number x_k is companied by an index I_k such that x_{I_k} is the smallest number in S that is larger than x_k. Then the set S can be sorted in linear time.*

PROOF. To sort the set S, we first scan the set S to find the minimum number x_{k_1} in S. Since x_{k_1} is companied by an index $k_2 = I_{k_1}$ such that x_{k_2} is the smallest number in S that is larger than x_{k_1}, x_{k_2} must be the second smallest number in S. Moreover, since we know the index k_2, we can get x_{k_2} and put it immediately after x_{k_1} in constant time. In general, suppose we have obtained x_{k_i} that is the ith smallest number in S. Then since x_{k_i} is companied by an index $k_{i+1} = I_{k_i}$ such that $x_{k_{i+1}}$ is the smallest number in S that is larger than x_{k_i}, $x_{k_{i+1}}$ is the $(i+1)$st smallest number in S, and we can get $x_{k_{i+1}}$ and put it immediately after x_{k_i} in constant time. It is clear that after $n-1$ such iterations, we reach the largest number in S and obtain a sorted list of the numbers in S. Since each iteration takes only constant time, we conclude that the set S is sorted in linear time. $\Box$

We first consider the problem EMST.

Lemma 8.4.5 *SORTING can be reduced to EMST in linear time.*

PROOF. Given a set S of n real numbers $x_1, x_2, \cdots, x_n$, which is an instance of SORTING, we construct an instance S' of EMST which is the set of n points

$$(x_1, 0), (x_2, 0), \cdots, (x_n, 0)$$

in the plane. Moreover, for each $1 \le i \le n$, we attach an index i to the point $(x_i, 0)$. It is easy to see that the solution to EMST on the input S' is a chain A' of $n-1$ segments in the plane, such that a segment $\overline{(x_i, 0)(x_j, 0)}$ is in A' if and only if the number x_j is the smallest number in S that is larger than x_i.

Now pass the chain A' back to SORTING. For each segment $\overline{(x_i, 0)(x_j, 0)}$ in A', we construct a pair (x_i, j) (remember that the index j is attached to the point $(x_j, 0)$). Using these pairs, we can construct the sorted list of S in linear time, by Lemma 8.4.4. This proves

$$\text{SORTING} \propto_n \text{EMST}$$

$\Box$

This Lemma, together with Theorem 8.4.3 and Theorem 8.4.1 gives us the following theorem.

Theorem 8.4.6 *Any bounded order algebraic decision tree that constructs the Euclidean minimum spanning tree for a set of n points in the plane runs in time at least $\Omega(n \log n)$.*

Therefore, the algorithm presented in Section 6.4 that constructs the Euclidean minimum spanning tree for sets of points in the plane is optimal.

Now we consider the problem TRIANGULATION.

Lemma 8.4.7 *SORTING can be reduced to TRIANGULATION in linear time.*

PROOF. The proof is very similar to the proof of Lemma 8.4.5. Given a set S of n real numbers $x_1, x_2, \cdots, x_n$, we construct a set S' of $n+1$ points in the plane

$$q = (x_1, 2), p_1 = (x_1, 0), p_2 = (x_2, 0), \cdots, p_n = (x_n, 0)$$

It is easy to see that the set S' has a unique triangulation that consists of the n segments $\overline{qp_i}$ for $1 \leq i \leq n$, and the $n-1$ segments $\overline{p_i p_j}$ where the number x_j is the smallest number in S that is larger than x_i.

Now using the similar argument as the one we used in the proof of Lemma 8.4.5, we conclude that we can construct the sorted list of S from the triangulation of S' in linear time. □

Theorem 8.4.8 *Any bounded order algebraic decision tree that constructs the triangulation for a set of n points in the plane runs in time at least $\Omega(n \log n)$.*

Thus the problem TRIANGULATION also has an optimal algorithm, which was presented in Section 6.3.

A simple generalization of the problem TRIANGULATION is the problem CONSTRAINED-TRIANGULATION, as introduced in Section 3.4. A lower bound for the CONSTRAINED-TRIANGULATION can be easily obtained from the lower bound of TRIANGULATION.

Theorem 8.4.9 *Any bounded order algebraic decision tree solving the problem CONSTRAINED TRIANGULATION runs in time at least $\Omega(n \log n)$.*

PROOF. It is easy to prove that

$$\text{TRIANGULATION} \propto_n \text{CONSTRAINED TRIANGULATION}$$

In fact, every instance of the problem TRIANGULATION, which is a set S of n points in the plane, is an instance $G = (S, \phi)$ of the problem CONSTRAINED TRIANGULATION, in which the set of segments is empty.

Since the problem TRIANGULATION has a lower bound $\Omega(n \log n)$, by Theorem 8.4.1, the problem CONSTRAINED TRIANGULATION has a lower bound $\Omega(n \log n)$. □

To derive lower bounds for the problems CLOSEST-PAIR and ALL-NEAREST-NEIGHBORS, we use the lower bound for the problem ELEMENT-UNIQUENESS, derived in the last section.

Theorem 8.4.10 *Any bounded order algebraic decision tree finding the closest pair for a set of n points in the plane runs in time at least $\Omega(n \log n)$.*

PROOF. We prove that

$$\text{ELEMENT-UNIQUENESS} \propto_n \text{CLOSEST-PAIR}$$

Given a set S of n real numbers $x_1, \cdots, x_n$, we construct an instance for CLOSEST-PAIR:

$$(x_1, 0), (x_2, 0), \cdots, (x_n, 0)$$

which is a set S' of n points in the plane. Clearly, all elements of S are distinct if and only if the closest pair in S' does not consist of two identical points. So the problem ELEMENT-UNIQUENESS is reducible to the problem CLOSEST-PAIR in linear time. Now the theorem follows from Theorem 8.3.1 and Theorem 8.4.1. $\square$

Since it is straightforward that

$$\text{CLOSEST-PAIR} \propto_n \text{ALL-NEAREST-NEIGHBORS}$$

by Theorem 8.4.10 and Theorem 8.4.1, we also obtain the following theorem.

Theorem 8.4.11 *Any bounded order algebraic decision tree finding the nearest neighbor for each point of a set of n points in the plane runs in time at least $\Omega(n \log n)$.*

Thus the algorithms we derived in Section 6.2 for the problems CLOSEST-PAIR and ALL-NEAREST-NEIGHBORS are also optimal.

To discuss the lower bound on the time complexity of the problem MAXIMUM-EMPTY-CIRCLE, we use the $\Omega(n \log n)$ lower bound for the problem UNIFORM-GAP, derived in the last section.

Theorem 8.4.12 *Any bounded order algebraic decision tree that constructs a maximum empty circle for a set of n planar points runs in time at least $\Omega(n \log n)$.*

PROOF. We show that

$$\text{UNIFORM-GAP} \propto_n \text{MAXIMUM-EMPTY-CIRCLE}$$

Given a set S of n real numbers $x_1, \cdots, x_n$ and another real number ϵ, which is an instance of the problem UNIFORM-GAP, we construct a set S' of n planar points

$$(x_1, 0), (x_2, 0), \cdots, (x_n, 0)$$

which is an instance of the problem MAXIMUM-EMPTY-CIRCLE.

Note that the diameter d of the maximum empty circle of S', which is part of the solution of MAXIMUM-EMPTY-CIRCLE on the input S', is the maximum distance of two consecutive numbers in the set S. Therefore, if $d \neq \epsilon$, the S is not a YES-instance of UNIFORM-GAP. However, $d = \epsilon$ does not imply that S is a YES-instance of UNIFORM-GAP since some consecutive numbers could have distance less than ϵ. To make sure that every pair of consecutive numbers has distance exactly ϵ, we scan the set S to find the maximum number $x_{\max}$ and the minimum number $x_{\min}$ in S. Now note that S is a YES-instance of UNIFORM-GAP if and only if

$$d = \epsilon \qquad \text{and} \qquad x_{\max} - x_{min} = (n-1)\epsilon$$

Therefore, given the diameter d of the maximum empty circle of S', a correct solution to UNIFORM-GAP on the input S can be obtained in linear time. This proves that UNIFORM-GAP is reducible to MAXIMUM-EMPTY-CIRCLE in linear time.

By Theorem 8.3.2 and Theorem 8.4.1, a lower bound on the time complexity of the problem MAXIMUM-EMPTY-CIRCLE is $\Omega(n \log n)$. □

Therefore, our algorithm in Section 6.5 for finding the maximum empty circle given a set of points in the plane is also optimal.

We have presented an $O(n \log n)$ time algorithm for the FARTHEST-PAIR problem in Section 3.3. We now prove that this algorithm is optimal by showing a lower bound on the time complexity of the problem. For this, we make use of the $O(n \log n)$ lower bound for the problem SET-DISJOINTNESS.

Theorem 8.4.13 *Any bounded order algebraic decision tree that solves the problem FARTHEST-PAIR runs in time at least* $\Omega(n \log n)$.

PROOF. We prove

$$\textit{SET-DISJOINTNESS} \propto_n \textit{FARTHEST-PAIR}$$

Given an instance $I = (X, Y)$ of the problem SET-DISJOINTNESS, we transform I into an instance of FARTHEST-PAIR as follows. Without loss of generality, suppose that all numbers in X and Y are positive. (Otherwise, we scan the sets X and Y to find the smallest number z in $X \cup Y$, then add the number $z + 1$ to each number in X and in Y.) Now find the largest number $z_{\max}$ in $X \cup Y$. Convert each number x_i in the set X into a point on the unit circle in the plane which has a polar angle $\frac{x_i}{z_{\max}}\pi$, and convert each number y_j in the set Y into a point on the unit circle in the plane which has a polar angle $\frac{y_j}{z_{\max}}\pi + \pi$. Intuitively, we transform all numbers in the set X into points in the first and second quadrants of the unit circle in the plane, while transform all numbers in the set Y into points in the third and fourth quadrants of the unit circle. Such a transformation gives us a set S of $2n$ planar points. It is easy to see that the diameter of S is 2 if and only if the intersection of X and Y is not empty. This proves that the problem SET-DISJOINTNESS can be reduced to the problem FARTHEST-PAIR in linear time.

By Theorem 8.3.3, the problem SET-DISJOINTNESS has a lower bound $\Omega(n \log n)$. Now by Theorem 8.4.1, the problem FARTHEST-PAIR also has a lower bound $\Omega(n \log n)$ on its time complexity. □

8.5 A remark on our model

Here is an interesting story that surprised many researchers in Algorithm Analysis.

Consider the following problem.

MAXIMUM-GAP

Input: A set S of n real numbers.
Output: The maximum distance between two consecutive numbers in S.

It is easily seen from the proof of Theorem 8.4.12 that

$$\text{UNIFORM-GAP} \propto_n \text{MAXIMUM-GAP}$$

Therefore, any bounded order algebraic decision tree that solves the problem MAXIMUM-GAP runs in time at least $\Omega(n \log n)$.

However, it is surprising that if the "floor function" $\lfloor \ \rfloor$ is allowed in our computational model, then the problem MAXIMUM-GAP can be solved in linear time! We describe the algorithm as follows.

Given a set S of n real numbers $x_1, x_2, \cdots, x_n$, we begin by finding the maximum number $x_{\max}$ and the minimum number $x_{\min}$ in S. This can be done in linear time by scanning the set S. Next, we divide the interval $[x_{\min}, x_{\max}]$ into $(n-1)$ "buckets"

$$[x_{\min},\ x_{\min}+\delta),\ [x_{\min}+\delta,\ x_{\min}+2\delta),\ [x_{\min}+2\delta,\ x_{\min}+3\delta),\ \cdots,$$

$$[x_{\min}+(n-3)\delta,\ x_{\min}+(n-2)\delta),\ [x_{\min}+(n-2)\delta,\ x_{\max}]$$

where $\delta = (x_{\max} - x_{\min})/(n-1)$. Call the bucket $[x_{\min}+(n-2)\delta,\ x_{\max}]$ the $(n-1)$*st bucket* B_{n-1}, and call the bucket $[x_{\min}+(k-1)\delta, x_{\min}+k\delta)$ the *kth bucket* B_k, for $1 \le k \le n-2$. Now for each x_i of the $n-2$ numbers in $S - \{x_{\min}, x_{\max}\}$, determine which bucket the number x_i should belong to. The number x_i belongs to the kth bucket B_k if and only if $\lfloor (x_i - x_{\min})/\delta \rfloor = k-1$. Therefore, each number in $S - \{x_{\min}, x_{\max}\}$ can be distributed to the proper bucket in constant time, and consequently, the $n-2$ numbers in $S - \{x_{\min}, x_{\max}\}$ can be distributed to proper buckets in linear time if the buckets are implemented by linked lists. Now for each bucket B_k, compute the minimum number $x^{(k)}_{\min}$ and the maximum number $x^{(k)}_{\max}$ in B_k. If a bucket B_k contains one number, return the unique number as both $x^{(k)}_{\min}$, $x^{(k)}_{\max}$, and if a bucket is empty, return nothing. Since for each bucket B_k, the numbers $x^{(k)}_{\min}$ and $x^{(k)}_{\max}$ can be computed in the time proportional to the size of the bucket B_k, all these $x^{(k)}_{\min}$ and $x^{(k)}_{\max}$, $1 \le k \le n-1$ can be computed in time linear to n. Now construct a list L

$$L: \quad x^{(1)}_{\min},\ x^{(1)}_{\max},\ x^{(2)}_{\min},\ x^{(2)}_{\max},\ \cdots,\ x^{(n-1)}_{\min},\ x^{(n-1)}_{\max}$$

(Note that some numbers above may not appear in the list L if the corresponding bucket is empty.) The list L can be easily constructed in linear time from the $n-1$ buckets.

Since there are $n-1$ buckets and only $n-2$ numbers in $S - \{x_{\min}, x_{\max}\}$, at least one bucket is empty. Therefore, the maximum distance between a pair of consecutive numbers in S is at least the length of a bucket. This implies that no two consecutive numbers contained in the same bucket can make the maximum distance. Thus the maximum distance must be made by a pair of the numbers (x_i, x_j) that are either $x_i = x_{\max}^{(k)}$ and $x_j = x_{\min}^{(h)}$ for some k and h (where all buckets $B_{k+1}, \cdots, B_{h-1}$ are empty), or $x_i = x_{\min}$ and $x_j = x_{\min}^{(k)}$ (where all buckets $B_1, \cdots, B_{k-1}$ are empty), or $x_i = x_{\max}^{(k)}$ and $x_j = x_{\max}$ (where all buckets $B_{(k+1)}, \cdots, B_{(n-1)}$ are empty). Moreover, all these pairs can be found in linear time by scanning the list L. Therefore, the maximum distance between pairs of consecutive numbers in S can be computed in linear time.

In the following, we give an even simpler linear time algorithm to solve the problem UNIFORM ϵ-GAP[2]. In this algorithm, we even do not require floor function. The only non-algebraic operation we need is a test if a given real number is an integer. Note that with the floor function, the test "*Is r an integer*" can be easily done in constant time.

```
Algorithm    MAGIC

      Given:      A set S = { x_1, x_2, ..., x_n } of real numbers.
      Question:   Is the distance between any two consecutive
                  numbers of S uniformly equal to epsilon?

      {In the following algorithm, A is an array of size n, which
       is initialized to empty.}

      BEGIN
      1.   Find the minimum number x_min and the maximum number
           x_max in S;
      2.   Let epsilon = (x_max - x_min)/(n-1);
      3.   For i = 1 to n do BEGIN
      3.1     Let  k = (x_i - x_min)/epsilon + 1;
      3.2     IF  k is not an integer OR A[k] is not empty  THEN
                STOP with an answer NO
      3.3     ELSE
                put x_i in the array element A[k];
           END;
       4.  STOP with an answer YES;
      END.
```

The above algorithm obviously runs in linear time. To see the correctness, suppose that the algorithm stops at Step 4. Then if a number x is in

[2]The author was informed of this algorithm by Roger B. Dubbs III.

$A[k]$, then the value of x must be $x_{\min}+\epsilon(k-1)$. Moreover, no array element of A holds more than one number. Consequently, every array element of A holds exactly one number from the set S, and these numbers are $x_{\min}+i\cdot\epsilon$, for $i=0,1,\cdots,n-1$. Therefore, the set S should be a YES-instance of the problem UNIFORM ϵ- GAP.

On the other hand, if the algorithm stops at Step 3.2, then either S is not uniformly distributed (otherwise all values $(x-x_{\min})/\epsilon+1$ should be integral) or the set S contains two identical numbers. In the latter case, the set S again cannot be a YES-instance of the problem UNIFORM ϵ-GAP.

The examples bring up an interesting point: there are certain very common operations not included in the algebraic decision tree model that allow us to do things that are not possible in the algebraic decision tree model. The floor function and the integral testing are examples of this kind of operations. Note that these examples imply that the floor operation and the integral testing cannot be performed in constant time in the algebraic decision tree model.

8.6 Exercises

1. Let P be an arbitrary non-trivial problem (i.e., it has YES-instances as well as NO-instances). Show that the problem *MAX-ELEMENT* (given a set of numbers, find the maximum) is linear time reducible to P.

2. Use Ben-or's technique directly to prove that the following problem has a lower bound $\Omega(n\log n)$ on the time complexity.

   SET-DISJOINTNESS

   Given two sets $X=\{x_1,\cdots,x_n\}$ and $Y=\{y_1,\cdots,y_n\}$ of real numbers, are they disjoint, i.e., $X\cap Y=\phi$?

3. Prove that the problems *STAR-POLYGON*, *INTERSECTION-OF-HALF-PLANE*, and *MONOTON-POLYGON* take $\Omega(n\log n)$ time in the algebraic decision tree model.

4. Prove that the problem *VORONOI-DIAGRAM* takes $\Omega(n\log n)$ time in the algebraic decision tree model.

5. Design an optimal algorithm that constructs convex hulls for sets of points in 3-dimensional Euclidean space.

6. Show that the problem *SECOND CLOSEST PAIR* takes $\Omega(n \log n)$ time in the algebraic decision tree model.

7. Given two sets A and B of points in the plane, each containing N elements, find the two closest points, one in A and the other in B. Show that this problem requires $\Omega(N \log N)$ opertations (*Hint:* what problem can we reduce to this problem?).

8. Give an optimal algorithm that, given a set of $2N$ points, half with positive x-coordinates, half with negative x-coordinates, finds the closest pair with one member of the pair in each half.

9. Prove that the following problem has an $\Omega(N \log N)$ lower bound:

   Given N points in the plane, construct a regular *PSLG* whose vertices are these N points.

10. Given a PSLG G, design an algorithm regularizing G in time $O(n \log n)$. Provide sufficient details for the implementation of your algorithm. (This does not mean you give a PASCAL or C program. Instead, you should provide sufficient detail for the data structure you use to suppose your operations.)

11. Prove that your algorithm for the last question is optimal.

12. Prove that the following problem has a lower bound $\Omega(n \log n)$:

    Given a PSLG G, add edges to G so that the resulting graph is a PSLG G' such that each region of G' is a simple polygon.

    (*Hint*: You can suppose Chazelle's result.)

13. Prove that the following problem requires $\Omega(n \log n)$ time in algebraic decision tree models: given n points and n lines in the plane, determine whether any point lies on any line.

14. Given a set of n points in the plane, let h denote the number of vertices that lie on its convex hull. Show that any algorithm for computing the convex hull must require $\Omega(n \log h)$ time in the algebraic decision tree model.

15. Given a convex n-gon, show that determining whether a query point lies inside or outside this n-gon takes $\Omega(\log n)$ time in the algebraic decision tree model.

16. Given a set S of n points in the plane, show that the problem of finding the minimum area rectangle that contains these points requires $\Omega(n \log n)$ time in the algebraic decision tree model.

17. Can you construct another example that requires $\Omega(n \log n)$ time in the algebraic decision tree model but is solvable in linear time?

Chapter 9

Geometric Transformations

In this chapter, we will discuss an important technique in computational geometry: The geometric transformations. We will introduce the method by showing how this method is applied to solve geometric intersection problems, such as half plane intersection and convex polygon intersection. We will also apply the method to find the smallest area triangles. We will see that the geometric transformation techniques enable us to convert these geometric problems into more familiar problems we have discussed.

Geometric transformations have their roots in the mathematics of the early nineteenth century [6]. Their applications to problems of computing dates back to the concept of primal and dual problems in the study of linear programming (see, for example, [21]).

Brown [7] gives a systematic treatment of transformations and their applications to problems of computational geometry. Since his dissertation, these methods have found vast application.

Typically, transformations change geometric objets into other geometric objects (for example, take points into lines) while preserving relations which held between the original objects (for example, order or whether they intersected). A number of geometric problems are best solved through the use of transformations. The standard scheme is to transform the objects under consideration, solve a simpler problem on the transformed objects, and then use that solution to solve the original problem. No single transformation applies in all cases; a number of different transformations have been used effectively. Here, we describe two commonly used transformations and demonstrate their applications.

9.1 Mathematical background

Let l be a straight line on the Euclidean plane. If l is a vertical line, then l can be characterized by an equation

$$x = a$$

if the line l intersects the x-axis at the point $(a, 0)$. On the other hand, if l is not a vertical line, let θ be the angle from the positive direction to the line l,[1] then l can be characterized by the equation

$$y = ax + b$$

If $a = \tan\theta$ and the line l intersects the y-axis at the point $(0, b)$. We will call θ the *direction* of the line l, and call the value $a = \tan\theta$ the *slope* of the line l. The slope of a straight line l is denoted by *slope(l)*.

The domain of all of our two-dimensional transformations will be the *projective plane*, which is an enhanced version of the Euclidean plane in which each pair of lines intersects. The projective plane contains all points of the Euclidean plane (call them the *proper points*). We introduce a set of *improper points* with one point P_a associated with every slope a in the plane. Two parallel lines, then, intersect at that improper point indicated by the slope of the parallel lines (this can be thought of as a point at infinity). All improper points are considered to lie on the same line: the *improper line* or the *line at infinity*. Thus, any two lines in the projective plane intersect at exactly one point: two nonparallel proper lines intersect at a proper point (i.e., one of the Euclidean plane); two parallel proper lines intersect at the improper point bearing the same slope; and a proper line intersects the improper line at the improper point defining the slope of the proper line. Likewise, between every two points passes exactly one line: There is a proper line passing through every pair of proper points; a proper line passes through a given improper point and a given proper point; and the improper line passes through any two improper points.

In general, the actual algorithms used to solve problems rely solely on the Euclidean geometry. Therefore, although all the transformations will map the projective plane onto itself, we will wish to choose a transformation which maps the objects under consideration to "proper" objects. Thus, the

[1] In this case, we always suppose that $-\pi/2 < \theta \leq \pi/2$. That is, we always suppose that the direction of the straight line l goes to the infinity either in the first quadrant or in the fourth quadrant.

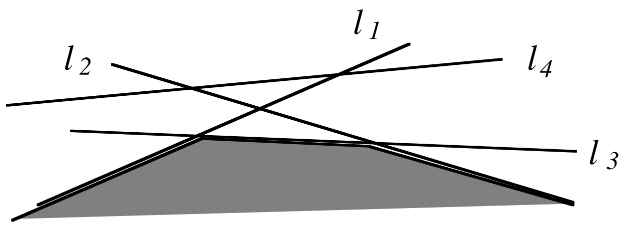

Figure 9.1: The half plane H_4 is redundant

parameters of the original problem will dictate which transformations are appropriate. Throughout this chapter, we will use the following notation: The images of a geometric object G, which can be a point p, a line l, or a polygon P, etc., under a transformation B will be denoted by $B(G)$.

9.2 Half plane intersections

We introduce the first geometric transformation through the following example.

Given a set of n lower half planes

$$H_i: \quad y \leq a_i x + b_i \qquad i = 1, \cdots, n$$

Let P be the intersection of these n lower half planes. It is easy to see that P is an unbounded convex area with an upper boundary

$$(e_{i_1} e_{i_2} \cdots e_{i_r})$$

which is a polygonal chain from left to right, where e_{i_1} and e_{i_r} are semi-infinite rays, and e_{i_j} for $2 \leq j \leq r-1$ are straight line segments, such that if traveling along the chain from e_{i_1} to e_{i_r}, we always make right turn.

Not every lower half plane is useful for the intersection P. For example, in Figure 9.1 the lower half plane H_4 defined by the line l_4 is not useful for the intersection P since the intersection P is entirely contained in the lower half plane H_4.

We say that a lower half plane $H_k: \; y \leq a_k x + b_k$ is *redundant* to the intersection P if the intersection P is entirely contained in the half plane

H_k. This simply implies

$$\bigcap_{1 \le i \le n} H_i = \bigcap_{i \ne k} H_i$$

Now given n lower half planes H_i, $i = 1, \cdots, n$, and let P be their intersection. How do we find all redundant half planes to the intersection? For this, we first discuss the property of a redundant half plane.

Suppose that $H : \quad y \le ax + b \quad$ is a lower half plane. We call the line $l : \quad y = ax + b \quad$ the *boundary line* of the lower half plane H. We also say that the lower half plane H is *defined* by the straight line $l : \quad y = ax + b$.

Lemma 9.2.1 *Given a set S of n lower half planes H_i defined by the straight lines l_i, $i = 1, \cdots, n$. A lower half plane H_k is redundant if and only if there are two lower half planes H_c and H_d in S such that*

$$slope(l_c) \le slope(l_k) \le slope(l_d)$$

and the intersecting point of l_c and l_d is below the line l_k.

PROOF. Suppose that the lower half plane H_k is redundant, then the intersection P of the n lower half planes in S is entirely below the line l_k. Let

$$\sigma = (e_{i_1} e_{i_2} \cdots e_{i_r})$$

be the boundary polygonal chain of P from left to right, where e_{i_1} and e_{i_r} are semi-infinite rays, and e_{i_j} for $2 \le j \le r - 1$ are straight line segments, such that if traveling along the chain from e_{i_1} to e_{i_r}, we always make right turn. Suppose that the edge e_{i_h} is on the line l_{i_h} for $h = 1, \cdots, r$. We claim

$$slope(l_{i_1}) \ge slope(l_k) \ge slope(l_{i_r})$$

In fact, if $slope(l_{i_1}) < slope(l_k)$ then since the starting point of the ray e_{i_1} is below the line l_k, the ray e_{i_1} must cross the line l_k at some point, contradicting the assumption that H_k is redundant. Similarly we can prove that $slope(l_k) \ge slope(l_{i_r})$. Since the chain σ makes only right turn when we travel from e_{i_1} to e_{i_r}, the slopes of the sequence of lines

$$l_{i_1}, l_{i_2}, \cdots, l_{i_r}$$

are strictly decreasing. Thus there must be two consecutive lines l_{i_h} and $l_{i_{h+1}}$ such that

$$slope(l_{i_h}) \ge slope(l_k) \ge slope(l_{i_{h+1}})$$

Moreover, the intersecting point of the lines l_{i_h} and $l_{i_{h+1}}$ is the point on the chain σ which is incident to the edges e_{i_h} and $e_{i_{h+1}}$, thus must be below the line l_k.

Conversely, if there are two lines l_c and l_d such that

$$slope(l_c) \leq slope(l_k) \leq slope(l_d)$$

and the intersection point of l_c and l_d is below the line l_k. Then clearly, the intersection $H_c \cap H_d$ of the two lower half planes H_c and H_d is entirely contained in the lower half plane H_k. Therefore,

$$P = \bigcap_{1 \leq i \leq n} H_i \subseteq H_c \cap H_d \subseteq H_k$$

That is, the lower half plane H_k is redundant to the intersection P. $\square$

By this lemma, a naive method of deciding the redundancy of a given half plane H_k entails comparing its boundary line against all other pairs of boundary lines in time $O(n^2)$. Finding all redundant lower half planes thus takes time $O(n^3)$.

We use the technique of geometric transformations to design a more efficient algorithm to find redundant lower half planes.

Let us first see what kinds of geometric properties are used for redundant lower half planes. To show that a lower half plane H defined by a line $l: \quad y = ax + b$ is redundant, we must show the existence of two lower half planes H_c and H_d defined by the lines $l_c: \quad y = a_c x + b_c$ and $l_d: \quad y = a_d x + b_d$ such that

$$slope(l_c) \leq slope(l) \leq slope(l_d)$$

and the intersecting point of l_c and l_d is below the line l. Therefore, if T is a geometric transformation, then given a line l, we would like that the parameter $slope(l)$ is mapped to a parameter of the geometric object $T(l)$ such that the ordering of the slopes of lines is preserved. Moreover, let p be a point and l be a line, then we want the relations "above" and "below" between p and l are also preserved for the geometric objects $T(p)$ and $T(l)$.

Now consider the following geometric transformation T_1. Given a point $p: \ (a, b)$ in the Euclidean plane, the image $T_1(p)$ under the transformation T_1 is a straight line

$$T_1(p): \quad y = ax + b$$

Now let $l: \quad y = \alpha x + \beta$ be a line. Since each point l is mapped to a line by the transformation T_1, $T_1(l)$ is a collection of lines. However, all these lines have a common intersecting point. In fact, let $q_0 = (x_0, y_0)$ be a point on the line l, then we have

$$y_0 = \alpha x_0 + \beta$$

Thus the image of q_0 under T_1 is the line

$$T_1(q_0): \qquad y = x_0 x + y_0 = x_0 x + (\alpha x_0 + \beta)$$

It is easy to see that the line $T_1(q_0)$ passes through the point $(-\alpha, \beta)$. Therefore, instead of regarding that the $T_1(l)$ as a collection of lines, we regard $T_1(l)$ as a single point $(-\alpha, \beta)$, and say that the transformation T_1 maps a line into a point.

Note that the above process of deriving the image $T_1(l)$ of the line l from the images of the points on the line l can be reversed. That is, since we have defined the image of a point $p = (a, b)$ under T_1 to be the line $T_1(p): \ y = ax + b$, we can derive that the image of a line $l: \ y = \alpha x + \beta$ under T_1 is a point $T_1(l) = (-\alpha, \beta)$. Alternatively, if we start by defining that the image of a line $l: \quad y = \alpha x + \beta$ is the point $T_1(l) = (-\alpha, \beta)$, then given a point $p = (a, b)$, we regard p as a collection of all lines passing through the point $p = (a, b)$. Any line l_0 in this collection can be represented by an equation

$$l_0: \ \ y = \delta x + (b - \delta a)$$

where δ can be any real number. Thus the image of l_0 under T_1 is a point $(-\delta, b - \delta a)$, which is a point on the line $y = ax + b$. Thus the image of the point $p = (a, b)$ under T_1 is the line

$$T_1(p): \ \ y = ax + b$$

This discussion shows that the intersection relation between a point and a line is preserved under the transformation. A more precise description is given in the following observation.

Observation 1.

Two points p_1 and p_2 are on the same line l if and only if the two lines $T_1(p_1)$ and $T_1(p_2)$ intersect at the point $T_1(l)$.

Observation 2.

Two lines l_1 and l_2 intersect at a point p if and only if the two points $T_1(l_1)$ and $T_1(l_2)$ are on the same line $T_1(p)$.

The definition of the transformation T_1 has only been given for the "normal points", which are the points in the Euclidean plane, and for the "normal lines", which are of the form $y = ax + b$ that is not a vertical line. We need to extend the definition to the improper points in the projective plane and to vertical lines. Using the idea of regarding a vertical line l as the collection of points on the line, and regarding an improper point P_α as the collection of the parallel lines of slope α, we can easily see that the image of a vertical line $x = a$ is the improper point P_a and the image of an improper point P_α is the vertical line $x = -\alpha$. We leave the detail derivation of these to the reader.

We now show that the ordering of the slope of lines, as well as the relations "above" and "below" of points and lines, are preserved under the transformation T_1. Let

$$l_1 : \quad y = a_1x + b_1 \qquad \text{and} \qquad l_2 : \quad y = a_2x + b_2$$

be two lines with the slopes a_1 and a_2, respectively. Then after the transformation T_1, the line l_1 becomes a point $(-a_1, b_1)$ while the line l_2 becomes a point $(-a_2, b_2)$. Therefore, if we denote by $x(p)$ the x-coordinate of the point p, then we have

Observation 3.

The slope of line l_1 is greater than the slope of line l_2 if and only if the x-coordinate of the point $T_1(l_1)$ is less than the x-coordinate of the point $T_1(l_2)$. That is

$$slope(l_1) > slope(l_2) \quad \text{iff} \quad x(T_1(l_1)) < x(T_1(l_2))$$

Now let $p = (a, b)$ be a point and let $l : \quad y = \alpha x + \beta$ be a line such that p is below the line l. Thus $b < \alpha a + \beta$. After the transformation T_1, the point p becomes a line $T_1(p) : \quad y = ax + b$ while the line l becomes a point $T_1(l) = (-\alpha, \beta)$. Since $\beta > a(-\alpha) + b$, the point $T_1(l)$ is above the line $T_1(p)$. This gives us the third observation.

Observation 4.

A point p is below a line l if and only if the line $T_1(p)$ is below the point $T_1(l)$.

Now we return back to the problem of deciding redundant lower half planes. Given a set A of n points in the plane, by the *lower hull* of A we denote the partial chain on the convex hull *CH(A)* which is from the point of the minimum x-coordinate in A to the point of maximum x-coordinate in A and bounds the convex hull *CH(A)* from below. Using the modified Graham scan algorithm, we know that the lower hull of the set A can be constructed in time $O(n \log n)$ (see Section 2.2.)

Now given a set S of n lower half planes H_i, where the lower half plane H_i is defined by a straight lines l_i, for $i = 1, \cdots, n$. Let P be the intersection of these n lower half planes. We first transform each line l_i in S by the transformation T_1 into a point $T_1(l_i)$. Let $T_1(S)$ be the set of images of the lines in S under T_1. $T_1(S)$ is a set of n planar points.

Theorem 9.2.2 *A lower half plane H_k in S is redundant to P if and only if the point $T_1(l_k)$ is not on the lower hull of CH($T_1(S)$).*

PROOF. If the lower half plane H_k in S is redundant, By Lemma 9.2.1, there are two lower half planes H_c and H_d in S such that

$$slope(l_c) \leq slope(l_k) \leq slope(l_d)$$

and the intersecting point p of l_c and l_d is below the line l_k. By Observation 3, we have

$$x(T_1(l_c)) \geq x(T_1(l_k)) \geq x(T_1(l_d))$$

Moreover, the point $T_1(l_k)$ is above the line $T_1(p)$, by Observation 4. Finally, by Observation 2, the two points $T_1(l_c)$ and $T_1(l_d)$ are on the line $T_1(p)$, thus the point $T(l_k)$ is above the line segment $\overline{T_1(l_c)T_1(l_d)}$. That is, the point $T_1(l_k)$ cannot be on the lower hull of $T_1(S)$.

Conversely, suppose that the point $T_1(l_k)$ is not on the lower hull of $T_1(S)$, then there are two points $T_1(l_c)$ and $T_1(l_d)$ in the set $T_1(S)$ such that the point $T_1(l_k)$ is above the line segment $\overline{T_1(l_c)T_1(l_d)}$. So we have

$$x(T_1(l_c)) \geq x(T_1(l_k)) \geq x(T_1(l_d))$$

By Observation 3, we have

$$slope(l_c) \leq slope(l_k) \leq slope(l_d)$$

Moreover, let l be the line on which the line segment $\overline{T_1(l_c)T_1(l_d)}$ lies, then by Observation 2, the line l is the image of the intersecting point p of the lines l_c and l_d under T_1. By Observation 4, the intersecting point p is below the line l_k. Now by Lemma 9.2.1, the line l_k is redundant to the intersection P. $\square$

Now it is straightforward to derive an algorithm finding the redundant lower half planes given a set of lower half planes.

Algorithm *FIND-REDUNDANCY (S)*

{ *Given a set S of n lower half planes H_i, where H_i is defined by a line l_i, for $i = 1, \cdots, n$, find all redundant half planes.* }

begin

1. *Using the transformation T_1 to transform each line l_i into a point $T_1(l_i)$; Let the set of the images of lines in S under T_1 be $T_1(S)$;*
2. *Construct the lower hull $LH(T_1(S))$ of $T_1(S)$;*
3. *For $k = 1, \cdots, n$, a lower half plane H_k is redundant if and only if the point $T_1(l_k)$ is not on the lower hull $LH(T_1(S))$.*

end

The algorithm is correct according to Theorem 9.2.2. Step 2 in the algorithm takes time $O(n \log n)$ by our discussion in Section 2.2. All other steps trivially take linear time. So the time complexity of the algorithm is $O(n \log n)$.

With the algorithm FIND-REDUNDANCY, it is easy to design an algorithm computing the intersection of half planes.

HALF-PLANE-INTERSECTION

Given $n + m$ half planes

$$y \leq a_i x + b_i \qquad i = 1, \cdots, n$$

$$y \geq c_j x + d_j \qquad j = 1, \cdots, m$$

in the plane, compute the intersection of them.

The problem can be split into two problems as follows: We consider the intersection P_1 of the n lower half planes:

$$y \leq a_i x + b_i \qquad\qquad i = 1, \cdots, n$$

and the intersection P_2 of the m upper half planes:

$$y \geq c_j x + d_j \qquad\qquad j = 1, \cdots, m$$

Then the intersection of the $n + m$ half planes is the intersection of P_1 and P_2.

The two areas P_1 and P_2 are unbounded polygonal areas, and both of them are convex. It is a simple exercise to show that the intersection of the two polygonal areas P_1 and P_2 can be computed in time $O(n_1 + n_2)$, where n_i, $i = 1, 2$, is the number of boundary edges of the polygonal area P_i.

Therefore, the problem HALF-PLANE-INTERSECTION can be reduced to computing the intersection of lower half planes and computing the intersection of upper half planes. Since the two problems are symmetric, we will concentrate on the problem of intersection of lower half planes.

LOWER-HALF-PLANE-INTERSECTION

Given n lower half planes

$$H_i : \quad y \leq a_i x + b_i \qquad\qquad i = 1, \cdots, n$$

in the plane, compute the intersection P_1 of them.

A half plane that is not redundant is called a *useful half plane* of the intersection P_1. Clearly, a half plane $H_k : \; y \leq a_k x + b_k$ is useful to the intersection P_1 if the straight line $y = a_k x + b_k$ contributes a boundary edge to the polygonal area P_1. Let

$$H_{i_k} : \quad y \leq a_{i_k} x + b_{i_k} \qquad\qquad k = 1, \cdots, r$$

be the set of useful half planes to the intersection P_1 such that the boundary of the polygonal area P_1 is formed by a polygonal chain

$$(e_{i_1} e_{i_2} \cdots e_{i_r})$$

from left to right, where the edge e_{i_k} on the chain is contributed by the straight line

$$l_{i_k} : \quad y = a_{i_k} x + b_{i_k}$$

for $k = 1, \cdots, r$. Since P_1 is a convex polygonal area and is below its boundary, the slope of the lines l_{i_k} must be strictly decreasing. This observation gives us immediately an algorithm to compute the intersection P_1.

Algorithm *LOWER-PLANE-INTERSECTION*
{ *Given the set S of n lower half planes, compute their intersection.* }
begin

1. *Eliminate all redundant lower half planes,*
2. *Sort the boundary lines of the useful half planes by their slopes in decreasing ordering. Let the sorted list of the lines be*
$$\{l_{i_1}, l_{i_2}, \cdots, l_{i_r}\}$$
3. *For $k = 1$ to $r - 1$, compute the intersecting point p_k of the lines l_{i_k} and $l_{i_{k+1}}$.*
4. *The polygonal chain*
$$\{l_{i_1} p_1 p_2 \cdots p_{r-1} l_{i_r}\}$$
is the boundary chain of the intersection P_1.

end

Step 1 takes time $O(n \log n)$ using the algorithm FIND-REDUNDANCY. Step 2 takes also time $O(n \log n)$ by any optimal sorting algorithm. The remaining of the algorithm takes linear time. Therefore, the above algorithm runs in time $O(n \log n)$. By our comments before, the intersection of n half planes can also be computed in time $O(n \log n)$. It is easy to see that this is also a lower bound for the problem, since the problem SORTING can be easily reduced to this problem. Thus we have

Theorem 9.2.3 *The problem HALF-PLANE-INTERSECTION can be solved by an optimal algorithm in $O(n \log n)$ time.*

Finally we remark that our algorithm for using a geometric transformation to solve the problem HALF-PLANE-INTERSECTION consisted of three parts: We first identified the geometric techniques we might use (here is eliminating redundant half planes). Next, we identified the invariants required by a transformation (here are the "above"/"below" relation and the ordering of slopes). Finally, we found an appropriate transformation and solved the problem. This is a classic example of how geometric transformations are used.

9.3 The smallest area triangle

We give another example of the applications of the geometric transformation T_1. Consider the following problem.

THE-SMALLEST-TRIANGLE

Given a set S of n points in the plane, find the smallest area triangle whose three vertices are points in S.

A brute force way to solve this problem is to compute, for every three points in S, the area of the triangle formed by these three points, and then pick the one with the smallest area. This algorithm takes time proportional to $\binom{n}{3} = O(n^3)$.

A variation of the above algorithm is that given a pair of points p_i and p_j in S, compute the distance $d(k,i,j)$ from a point p_k to the line $l_{i,j}$ passing through the points p_i and p_j, where p_k is any point picked from $S - \{p_i, p_j\}$. Since the area of the triangle formed by p_i, p_j, and p_k is half of the product $d(k,i,j) \cdot |\overline{p_i p_j}|$, this will give us the areas of all triangles one of whose edge is the segment $\overline{p_i p_j}$. If we do this for every pair of points in S, we will obtain the areas of all triangles formed by points in S, thus pick the one with smallest area. The time complexity of this variation is $O(\binom{n}{2}(n-1))$, which is again $O(n^3)$.

Definition Let p be a point and l be a line. The *vertical distance*, denoted $d_v(p,l)$, from p to l is the distance from the point p to the intersecting point of the line l and the vertical line passing through the point p.

Note that the vertical distance from a point p to a line l is in general different from the *distance* from the point to the line, which is the distance from the point p to the intersecting point of the line l and the line which passes through p and is perpendicular to the line l.

Let p_k, p_i, and p_j be three points in S. We denote by $d_v(k,i,j)$ the vertical distance from the point p_k to the line $l_{i,j}$ passing through the points p_i and p_j. For simplicity, sometime we also call $d_v(k,i,j)$ the vertical distance from the point p_k to the segment $\overline{p_i p_j}$.

Lemma 9.3.1 *Fix a pair of points p_i and p_j in the set S. Let $l_{i,j}$ be the line passing through the points p_i and p_j. Then a point p_k in $S - \{p_i, p_j\}$*

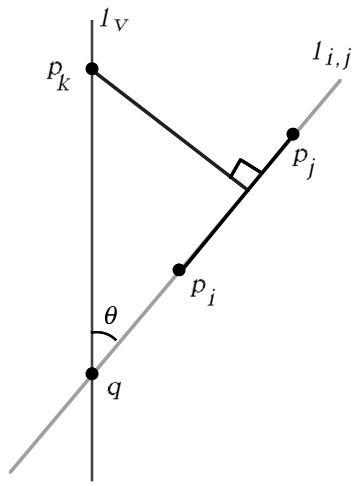

Figure 9.2: The vertical distance from a point to a line

has the smallest distance $d(k,i,j)$ from the line $l_{i,j}$ if and only if p_k has the smallest vertical distance $d_v(k,i,j)$ from the line $l_{i,j}$.

PROOF. Let l_v be the vertical line passing through the point p_k which intersects the line $l_{i,j}$ at a point q. Let θ be the angle between the lines $l_{i,j}$ and l_v. By the definition, the vertical distance $d_v(k,i,j)$ from p_k to $l_{i,j}$ is the length of the line segment $\overline{p_k q}$. Moreover, it is easy to see that the distance $d(k,i,j)$ from p_k to $l_{i,j}$ is equal to $|\overline{p_k q}| \cdot \sin\theta$. See Figure 9.2. Thus, the vertical distance from p_k to $l_{i,j}$ is proportional to the distance from p_k to $l_{i,j}$. The lemma follows immediately. $\square$

Therefore, to find the smallest area triangle, for each pair of points p_i and p_j in S, we only need to consider such a point p_k in $S - \{p_i, p_j\}$ such that the vertical distance $d_v(k,i,j)$ is the shortest. But how this observation helps us?

We first apply the transformation T_1 on the set S of planar points. We know that a point p_k in S is mapped under T_1 to a line $T_1(p_k)$ while a line $l_{i,j}$ passing through two points p_i and p_j in S is mapped under T_1 to the intersecting point $T_1(l_{i,j})$ of the lines $T_1(p_i)$ and $T_1(p_j)$. A nice property of the transformation is that the vertical distance is preserved under the transformation, as shown by the following lemma.[2]

Lemma 9.3.2 *The vertical distance $d_v(p_k, l_{i,j})$ from the point p_k to the line $l_{i,j}$ is equal to the vertical distance $d_v(T_1(l_{i,j}), T_1(p_k))$ from the point $T_1(l_{i,j})$ to the line $T_1(p_k)$*

$$d_v(p_k, l_{i,j}) = d_v(T_1(l_{i,j}), T_1(p_k))$$

PROOF. The proof is straightforward through the calculations using basic formulas in analytical geometry. Suppose that the coordinates of p_i, p_j, and p_k are

$$p_i = (a_i, b_i) \qquad p_j = (a_j, b_j) \qquad p_k = (a_k, b_k)$$

Then the line $l_{i,j}$ has the equation

$$l_{i,j}: \quad y = \frac{b_i - b_j}{a_i - a_j}x + \frac{a_i b_j - a_j b_i}{a_i - a_j}$$

Under the transformation T_1, they are mapped to the lines $T_1(p_i)$, $T_1(p_j)$ and $T_1(p_k)$:

$$T_1(p_i): \; y = a_i x + b_i \qquad T_1(p_j): \; y = a_j x + b_j \qquad T_1(p_k): \; y = a_k x + b_k$$

and the point

$$T_1(l_{i,j}) = (-\frac{b_i - b_j}{a_i - a_j}, \frac{a_i b_j - a_j b_i}{a_i - a_j})$$

The intersecting point of the line $l_{i,j}$ and the vertical line passing through the point p_k is

$$(a_k, \frac{b_i - b_j}{a_i - a_j}a_k + \frac{a_i b_j - a_j b_i}{a_i - a_j})$$

Thus the vertical distance from the point p_k to the line $l_{i,j}$ is the absolute value of the following number

$$\frac{b_i - b_j}{a_i - a_j}a_k + \frac{a_i b_j - a_j b_i}{a_i - a_j} - b_k = \frac{(a_i b_j + a_j b_k + a_k b_i) - (a_i b_k + a_j b_i + a_k b_j)}{a_i - a_j}$$

Similarly, the intersecting point of the line $T_1(p_k)$ and the vertical line passing through the point $T_1(l_{i,j})$ is

$$(-\frac{b_i - b_j}{a_i - a_j}, -a_k \frac{b_i - b_j}{a_i - a_j} + b_k)$$

[2]Without loss of generality, we suppose that no two points in the set S have the same x-coordinate. If this condition is not satisfied, we slightly rotate the coordinate system.

Thus the vertical distance from the point $T_1(l_{i,j})$ to the line $T_1(p_k)$ is the absolute value of the following number

$$\frac{a_ib_j - a_jb_i}{a_i - a_j} - (-a_k\frac{b_i - b_j}{a_i - a_j} + b_k) = \frac{(a_ib_j + a_jb_k + a_kb_i) - (a_ib_k + a_jb_i + a_kb_j)}{a_i - a_j}$$

This proves the lemma. $\square$

But why this lemma helps? Let us first transform each point p_i in the set S under T_1 to a line $T_1(p_i)$ for $i = 1, \cdots, n$. Then we will obtain a set $T_1(S)$ of n straight lines

$$T_1(S) = \{T_1(p_1), T_1(p_2), \cdots, T_1(p_n)\}$$

The set $T_1(S)$ of these n lines $T_1(p_i)$, $i = 1, \cdots, n$, forms a PSLG, if we regard each intersecting point of a pair of lines in $T_1(S)$ as a vertex. Let $l_{i,j}$ be the line passing through the points p_i and p_j in S. Note that to find a point in S which has the smallest vertical distance to the line $l_{i,j}$, we have to check each vertex in $S - \{p_i, p_j\}$. However, to find the line $T_1(p_k)$ in $T_1(S)$ such that the point $T_1(l_{i,j})$ has the smallest vertical distance to $T_1(p_k)$, we only need to check in $T_1(S)$ the lines immediately above and immediately below the point $T_1(l_{i,j})$. Therefore, if we well organize the PSLG $T_1(S)$, we can find efficiently the line $T_1(p_k)$ in $T_1(S)$ such that the vertical distance from the point $T_1(l_{i,j})$ to the line $T_1(p_k)$ is the smallest over all lines in $T_1(S) - \{T_1(p_i), T_1(p_j)\}$. By Lemma 9.3.1 and Lemma 9.3.2, the distance from the point p_k to the line $l_{i,j}$ is the smallest over all points in $S - \{p_i, p_j\}$, which implies that the triangle $\triangle(p_kp_ip_j)$ has the smallest area over all triangles one of whose edge is the segment $\overline{p_ip_j}$. For each pair of points p_i and p_j in the set S, perform the above process, we obtain the smallest area triangle.

Now we discuss how to find the lines closest to the point $T_1(l_{i,j})$ in $T_1(S)$. We perform a topological sweeping on the PSLG $T_1(S)$ from left to right by a vertical line L. The lines of $T_1(S)$ are stored in a 2-3 tree A in the ordering of their intersecting points with the line L on L. Since $T_1(S)$ contains exactly n lines $T_1(p_i)$, $i = 1, \cdots, n$, the number of leaves of the 2-3 tree A is n, thus the depth of A is bounded by $O(\log n)$. Suppose that at some moment, the line L moves to a vertex $T_1(l_{i,j})$ of $T_1(S)$ from left to right, then it is easy to see that except for the two lines $T_1(p_i)$ and $T_1(p_j)$ that intersect at $T_1(l_{i,j})$, all other lines in $T_1(S)$ maintain their relative position with respect to each other in the 2-3 tree A. On the other hand, the lines $T_1(p_i)$ and $T_1(p_j)$ should exchange their positions in the 2-3 tree A. In other words, if the line $T_1(p_i)$ is above the line $T_1(p_j)$ on the left of the point $T_1(l_{i,j})$, then the line $T_1(p_i)$ should be below the line $T_1(p_j)$ on the right of the point $T_1(l_{i,j})$, and vice versa. Let the two lines that are immediately above and below the vertex $T_1(l_{i,j})$ in the PSLG $T_1(S)$ be $T_1(p_k)$ and $T_1(p_h)$, respectively. Then the lines $T_1(p_k)$ and $T_1(p_h)$ can be also accessed in time $O(\log n)$ from the 2-3 tree A. To find the relative position of a vertex $T_1(l_{i,j})$ in the 2-3 tree A, we do a search in the 2-3 tree A by the values of y-coordinate while fixing the x-coordinate of each line in A to the value of the x-coordinate of the vertex $T_1(l_{i,j})$.

The following is the implementation of the above discussion, which finds the smallest area triangle given a set S of points in the plane.

Algorithm *SMALLEST-TRIANGLE (S)*

{ *Given a set S of n planar points, find the smallest area triangle whose three vertices are points in S.* }

begin

1. *For each point p_i in S, $i = 1, \cdots, n$, construct the line $T_1(p_i)$.*

2. *For each pair of lines $T_1(p_i)$ and $T_1(p_j)$ constructed in Step 1, $i, j = 1, \cdots, n$, compute the intersecting vertex $T_1(l_{i,j})$.*

3. *Sort all intersecting vertices $T_1(l_{i,j})$, $i, j = 1, \cdots, n$ in increasing x-coordinate ordering. Let the sorted list be*

$$\{v_1, v_2, \cdots, v_m\}$$

   where $m = \binom{n}{2}$, and $v_i = (x_i, y_i)$, for $i = 1, \cdots, m$.

4. *Construct a 2-3 tree A whose leaves are the lines $T_1(p_i)$, $i = 1, \cdots, n$, ordered by the y-coordinates of their intersecting points with the vertical line $x = x_1 - 1$.*

5. *For $r = 1, \cdots, m$ do the following*

   Suppose that the vertex v_r is the intersecting vertex of the lines $T_1(p_i)$ and $T_1(p_j)$ and that the lines immediately above and below the vertex v_r are $T_1(p_k)$ and $T_1(p_h)$, respectively. Compute the areas of the triangles $\triangle(p_k p_i p_j)$ and $\triangle(p_h p_i p_j)$. Exchange the positions of $T_1(p_i)$ and $T_1(p_j)$ in the 2-3 tree A.

6. *The triangle that is constructed in Step 5 and has the smallest area is the the smallest area triangle.*

end

One case we have ignored in the above algorithm is the case when there are three lines $T_1(p_i)$, $T_1(p_j)$, and $T_1(p_k)$ of $T_1(S)$ intersecting at a common point. However, this means that the intersecting point $T_1(l_{i,j})$ of the lines $T_1(p_i)$ and $T_1(p_j)$ has a zero vertical distance from the line $T_1(p_k)$. By Lemma 9.3.2, the point p_k has a zero vertical distance from the line $l_{i,j}$. Consequently, the three points p_i, p_j, and p_k are co-linear and the smallest area triangle of the set S has area zero. Therefore, whenever we find that three lines $T_1(p_i)$, $T_1(p_j)$, and $T_1(p_k)$ in $T_1(S)$ are co-linear, we stop immediately and return the triple (p_i, p_j, p_k) as the smallest area triangle.

We analyze the algorithm. Step 1 takes time $O(n)$. Step 2 takes time $O(n^2)$ and produces $m = \binom{n}{2} = O(n^2)$ intersecting vertices in $T_1(S)$. Thus Step 3 takes time $O(n^2 \log n)$ to sort the intersecting vertices constructed in Step 2. Step 4 takes time $O(n \log n)$. For each vertex v_r, Step 5 spends $O(\log n)$ time to locate the position of the vertex v_r in the 2-3 tree A, to update the 2-3 tree A and to compute the areas of the two triangles, so totally Step 5 takes time $O(m \log n) = O(n^2 \log n)$. Since for each vertex v_r, we construct at most two triangles, the number of triangles constructed in Step 5 is bounded by $O(m) = O(n^2)$. Consequently, Step 6 takes time $O(n^2)$. Therefore, the time complexity of the above algorithm is bounded by $O(n^2 \log n)$.

Since each line in the PSLG $T_1(S)$ corresponds to a point in S, the 2-3 tree A has exactly n leaves. However, the space used to store the vertices v_r, $r = 1, \cdots, m = \binom{n}{2}$ is $\Omega(n^2)$. So the space used by the algorithm is $O(n^2)$.

Now we discuss how we can reduce the amount of space used by the algorithm. As pointed out above, the $O(n^2)$ space is used to store the m intersecting vertices of the lines $T_1(p_i)$, for $i = 1, \cdots, n$. However, we do not really need the whole sorted list of these intersecting vertices. What we are really interested in is that at each stage which vertex is the next to the current vertex v_r. This next vertex must be the one that is on the right of the current vertex v_r and the closest to the vertical line passing through the current vertex v_r. Note that such a vertex must be the intersecting vertex of two lines in $T_1(S)$ that are consecutive leaves in the current 2-3 tree A. Therefore, if we keep a list B of the records for the intersecting vertices of the consecutive leaves in the current 2-3 tree A that are on the right of the current vertex v_r (there are at most $n-1$ such intersecting vertices), then the one in the list B that is the closest to the vertical line passing through the current vertex v_r must be the next vertex to be processed in Step 5 of the above algorithm.

Therefore, instead of producing the whole list

$$v_1, v_2, \cdots, v_m$$

of intersecting vertices of the lines $T_1(p_i)$, $i = 1, \cdots, n$, we use a 2-3 tree B to store the intersecting vertices of consecutive lines in the 2-3 tree A that are on the right of the current vertex v_r, sorted by their x-coordinates. The number of leaves of the tree B is bounded by $n-1$. Suppose that the current vertex is v_r. To find the next vertex, we simply find the vertex v_{r+1} in the 2-3 tree B that has the smallest x-coordinate. Then the vertex v_r is deleted from the tree B. Note that after processing the vertex v_r, adjacency relations among only four lines in A can be changed. That is, suppose that the vertex v_r is the intersecting vertex of the lines $T_1(p_i)$ and $T_1(p_j)$, that the lines in $T_1(S)$ immediately above and below the vertex v_r are $T_1(p_k)$ and $T_1(p_h)$, respectively, and that before processing the vertex v_r, the line $T_1(p_i)$ is above the line $T_1(p_j)$. Then after processing the vertex v_r, the line $T_1(p_j)$ is above the line $T_1(p_i)$. Therefore, before processing the vertex v_r, these lines are in the ordering

$$T_1(p_k), T_1(p_i), T_1(p_j), T_1(p_h)$$

in the 2-3 tree A, while after processing the vertex v_r, the line ordering becomes

$$T_1(p_k), T_1(p_j), T_1(p_i), T_1(p_h)$$

Accordingly, the 2-3 tree B can be updated by deleting the intersecting vertices of $T_1(p_k)$ and $T_1(p_i)$, and of $T_1(p_j)$ and $T_1(p_h)$, and inserting the intersecting vertices of $T_1(p_k)$ and $T_1(p_j)$, and of $T_1(p_i)$ and $T_1(p_h)$, if they are on the right of the vertex v_r. The intersecting vertices of $T_1(p_i)$ and $T_1(p_j)$ is the vertex v_r. Since the number of leaves of the 2-3 tree B is bounded by $n-1$, each of the above operations can be done in time $O(\log n)$. Therefore, processing a vertex v_r in Step 5 of the algorithm takes time $O(\log n)$. And the space now used by the algorithm, which is the sum of the 2-3 tree A and the 2-3 tree B, is bounded by $O(n)$.

This completes our description of an $O(n^2 \log n)$ time and $O(n)$ space algorithm that solves the problem THE-SMALLEST-TRIANGLE.

Final Remark:

If the area of the smallest area triangle is zero, then the three points forming this triangle are co-linear. Consequently, the above algorithm can be used to check if there exist three points that are co-linear in a given set of n planar points. The algorithm we presented in this section is not the best algorithm. The best algorithm we know for the problem THE-SMALLEST-TRIANGLE is due to Edelsbrunner, O'Rourke, and Seidel, which runs in time $O(n^2)$ and space $O(n)$ [11]. Whereas, the only known lower bound is $\Omega(n \log n)$. In fact, even for checking whether there exist three co-linear points, the only bounds that we know are $O(n^2)$ and $\Omega(n \log n)$. Improving the upper or lower bounds for either of these problems remains an extremely tantalizing open problem in computational geometry.

9.4 Convex polygon intersections

Now we introduce the second geometric transformation T_2. Given a point $p = (a, b)$ in the plane, we define the image of p under T_2 to be the line

$$T_2(p): \qquad ax + by + 1 = 0$$

In a similar way as we did for the transformation T_1, we discuss what is the image of a line $l: \quad \alpha x + \beta y + 1 = 0$. A point $q_0 = (a_0, b_0)$ on the line l satisfies

$$\alpha x_0 + \beta y_0 + 1 = 0$$

Thus $y_0 = (-\alpha/\beta)x_0 - 1/\beta$, and the point q_0 is mapped to the line

$$T_2(q_0): \; x_0 x + y_0 y + 1 = 0 \qquad \text{or} \qquad x_0 x + ((-\alpha/\beta)x_0 - 1/\beta)y + 1 = 0$$

It is easy to check that the line $T_2(q_0)$ passes through the point (α, β). Thus every point on the line $l: \; \alpha x + \beta y + 1$ is mapped to a line passing through the point (α, β). Thus we simply regard the image of the line l to be the point (α, β).

Again, the transformation T_2 preserves the relation of intersection of a point and a line. That is, a point p is on a line l if and only if the line $T_2(p)$ contains the point $T_2(l)$. More precisely, we have

Observation 1

Two points p and q are on the line l if and only if the two lines $T_2(p)$ and $T_2(q)$ intersect at the point $T_2(l)$.

Observation 2

Two lines l_1 and l_2 intersect at a point p if and only if the two points $T_2(l_1)$ and $T_2(l_2)$ are on the same line $T_2(p)$.

Another nice property of the transformation T_2 is that the distance of an object from the origin is preserved. In fact, by analytical geometry, we know that the distance of a point (a, b) from the origin is $\sqrt{a^2 + b^2}$ and the distance of a line $l: \ ax + by + 1 = 0$ from the origin is $1/\sqrt{a^2 + b^2}$. Thus, points or lines further from the origin are mapped to lines or points closer to the origin. That the transformation T_2 is also its own inverse (i.e., $G = T_2(T_2(G))$ where G is either a point or a line) also contributes to its usefulness. Moreover, let $p = (a, b)$ and $q = (c, d)$ be two points, which have distance $\sqrt{a^2 + b^2}$ and $\sqrt{c^2 + d^2}$ from the origin, respectively. The transformation T_2 maps them to two lines

$$T_2(p): \ ax + by + 1 = 0 \quad \text{and} \quad T_2(q): \ cx + dy + 1 = 0$$

which have distance $1/\sqrt{a^2 + b^2}$ and $1/\sqrt{c^2 + d^2}$ from the origin, respectively. Therefore,

Observation 3

If a point p is closer than a point q to the origin, then the line $T_2(p)$ is further than the line $T_2(q)$ from the origin. Similarly, if a line l_1 is closer than a line l_2 to the origin, then the point $T_2(l_1)$ is further than the point $T_2(l_2)$ from the origin.

Finally, it is also easy to check the following observation.

Observation 4

If two points p and q are on the same ray starting from the origin, then the lines $T_2(p)$ and $T_2(q)$ are in parallel.

We note that each improper point P_α is mapped to the line through the origin having the slope α and vice versa. Similarly, the origin and the improper line are duals. Consequently, the transformation T_2 should not be applied to lines or to segments of lines which pass through the origin. Nonetheless, the mere fact that the domain of a problem contains a line through the origin should not make us abandon T_2. By translating the axes in one direction or another, we may be able to insure that T_2 will map every object in the domain of our problem to another proper object.

Let $P = \{v_1, v_2, \cdots, v_n\}$ be a convex polygon that contains the origin O. For a vertex $v_i = (a_i, b_i)$ of P, the image of v_i under T_2 is the line

$$T_2(v_i): \quad a_i x + b_i y + 1 = 0$$

which does not pass through the origin. Call the half plane H_i with the boundary line $T_2(v_i)$ and containing the origin the *half plane defined by* $T_2(v_i)$. Let l_i be the line on which the boundary edge $\overline{v_i v_{i+1}}$ of P lies. Then the image $T_2(l_i)$ of l_i is the intersecting point of the lines $T_2(v_i)$ and $T_2(v_{i+1})$ and the origin is contained in the intersection of the half planes H_i and H_{i+1} defined by $T_2(v_i)$ and $T_2(v_{i+1})$, respectively. Since the vertex v_i is connected to the vertex v_{i+1} by the boundary edge $\overline{v_i v_{i+1}}$ of P which is on the line l_i, for $i = 1, \cdots, n$ (here $v_{n+1} = v_1$), the line $T_2(v_i)$ intersects the line $T_2(v_{i+1})$ at the point $T_2(l_i)$. Thus the intersection of the half planes H_i, defined by $T_2(v_i)$, for $i = 1, \cdots, n$ is a bounded area, which is a convex polygon containing the origin such that the sequence of boundary vertices of the convex polygon is $T_2(l_1), T_2(l_2), \cdots, T_2(l_n)$. Therefore, we can regard the image of the convex polygon P containing the origin under T_2 to be again a convex polygon $T_2(P)$ containing the origin with the boundary vertices $T_2(l_1), T_2(l_2), \cdots, T_2(l_n)$.

It is easy to see that given a convex polygon P that contains the origin, the image $T_2(P)$ of P under T_2 can be constructed in time proportional to the number of vertices of P.

Now we apply the transformation T_2 to the following problem.

CONVEX-POLYGON-INTERSECTION

Given a set of convex polygons that contain the origin, compute the intersection of them.

Suppose that S is a set of convex polygons $P_1, \cdots, P_n$ that contain the origin. We first construct the image $T_2(P_i)$ for each convex polygon P_i in S. Let S' be the set of vertices of all these convex polygons $T_2(P_1), T_2(P_2), \cdots, T_2(P_n)$. We will show that the intersection of the convex polygons of S corresponds to the convex hull of the set S'. To prove this, we need a few lemmas.

Let l be a line that does not pass through the origin. Draw a ray r starting from the origin and intersecting the line l at p. Let q be an arbitrary point

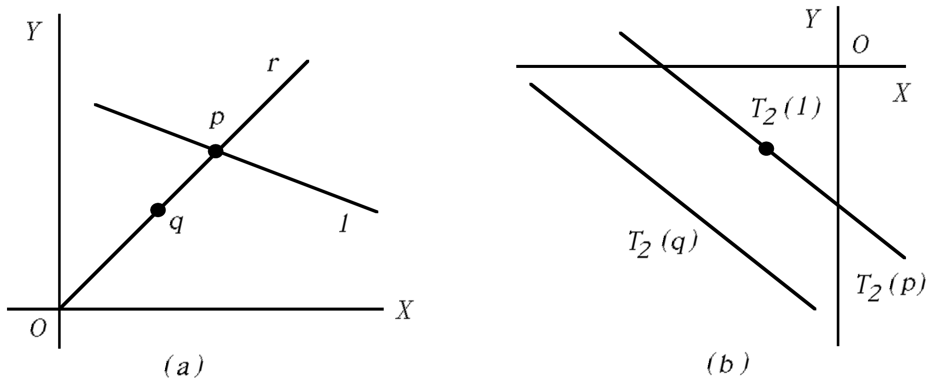

Figure 9.3: $\overline{Oq}$ does not intersects l

on the ray r.

Lemma 9.4.1 *The segment $\overline{Oq}$ intersects the line l if and only if the segment $\overline{OT_2(l)}$ intersects the line $T_2(q)$.*

PROOF. Suppose that the segment $\overline{Oq}$ does not intersect the line l, as shown in Figure 9.3(a). Then the point q is closer than the point p to the origin. Thus the line $T_2(q)$ is further than the line $T_2(p)$, by Observation 3. Moreover, the lines $T_2(q)$ and $T_2(p)$ are in parallel, by Observation 4, and the point $T_2(l)$ is on the line $T_2(p)$. Consequently, the segment $\overline{OT_2(l)}$ does not intersect the line $T_2(q)$, see Figure 9.3(b). The inverse can be proved in a very similar way, thus we omit it here. □

Let the intersection of the convex polygons in S be I. Suppose that l is a line on which an edge of some polygon in S lies. Then we know that $T_2(l)$ is a point in the set S'. We say that the line l contributes a boundary edge to the intersection I if part of l is on the boundary of the intersection I.

Lemma 9.4.2 *The line l contributes a boundary edge to the intersection I if and only if the point $T_2(l)$ is on the convex hull of the set S'.*

PROOF. Suppose that the line l contributes an edge to the intersection I but $T_2(l)$ is not a hull vertex of S'. Let r be the ray starting from the

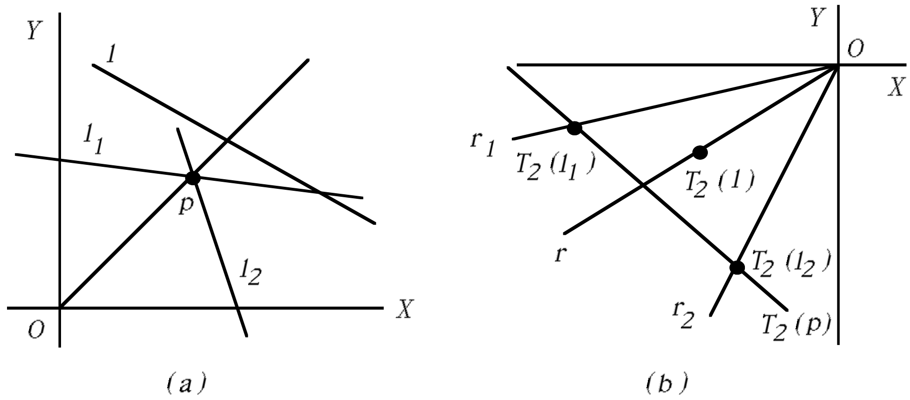

Figure 9.4: $T_2(l)$ is not a hull vertex

origin and passing through the point $T_2(l)$. Then we must be able to find two points $T_2(l_1)$ and $T_2(l_2)$ in the set S' such that if we let r_1 and r_2 be the rays staring from the origin and passing through the points $T_2(l_1)$ and $T_2(l_2)$, respectively, then the ray r is between the two rays r_1 and r_2, and that the segment $\overline{OT_2(l)}$ does not intersect the line $T_2(p)$ passing through the points $T_2(l_1)$ and $T_2(l_2)$, where p is the intersecting point of the lines l_1 and l_2, see Figure 9.4(b). By lemma 9.4.1, the segment $\overline{Op}$ does not intersect the line l. Moreover, since the ray r is between the two rays r_1 and r_2, $slope(l)$ is between $slope(l_1)$ and $slope(l_2)$. Therefore, if we let H, H_1, and H_2 be the half planes defined by the lines l, l_1, and l_2, respectively, then the area $H_1 \cap H_2$ is entirely contained in the half plane H, see Figure 9.4(a). But the intersection I is entirely contained in $H_1 \cap H_2$ thus is entirely contained in the half plane H. But this contradicts the assumption that the line l contributes an edge to I. This contradiction shows that the point $T_2(l)$ must be a hull vertex of the set S'.

The inverse that if $T_2(l)$ is a hull vertex of S' then the line l contributes an edge to the intersection I can be proved similarly and is left as an exercise to the reader. $\square$

Lemma 9.4.2 immediately suggests the following algorithm to solve the problem CONVEX-POLYGON-INTERSECTION.

Algorithm *CONVEX-POLYGON-INTERSECTION (S)*

{ *Given a set S of convex polygons that contain the origin, compute their intersection.* }
begin

1. *For each convex polygon P_i and for each edge e of the polygon P_i, if the edge e lies on a line l, construct the point $T_2(l)$.*

2. *Let S' be the set of points produced in Step 1, construct the convex hull $CH(S')$ of S'.*

3. *Let S'' be the set of lines that are preimages of the hull vertices in $CH(S')$. Sort S'' by slopes, let the sorted list be*

$$l_1, l_2, \cdots, l_r$$

4. *For $i = 1, \cdots, r$ compute the intersecting point p_i of l_i and l_{i+1} (here $l_{r+1} = l_1$), then sequence*

$$p_1, p_2, \cdots, p_r$$

is a convex polygon that is the intersection of convex polygons in S.

end

The algorithm correctly finds the intersection of the polygons in the set S, as we have discussed above. Moreover, if the sum of the number of edges of the polygons in S is N, then the above algorithm trivially runs in time $O(N \log N)$.

Chapter 10

Geometric Problems in Higher Dimensions

In this chapter, we introduce techniques for solving geometric problems in more than two dimensions. Section 1 introduces the preliminaries of higher dimensional geometry and representation of geometric objects in higher dimensions in a computer. Section 2 describes a divide-and-conquer algorithm for constructing the convex hull of a set of points in 3-dimensional Euclidean space. Section 3 gives an optimal algorithm for constructing the intersection of a set of half-spaces in 3-dimensional Euclidean space. Section 4 demonstrates an interesting relationship between a convex hull of a set of points in the n-dimensional Euclidean space and the Voronoi diagram of a set of projected points in the $(n+1)$-dimensional Euclidean space. Section 3 and Section 4 actually gives an optimal algorithm for constructing the Voronoi diagram for a set of points in the plane using reduction techniques.

10.1 Preliminaries

10.2 Convex hulls in three dimension

From Preperata and Shamos.

10.3 Intersection of half-spaces

From Preperata and Shamos.

10.4 Convex hull and Voronoi diagram

From MIT Lecture Notes by Agarwal.

Chapter 11

Dynamization Techniques

The techniques are developed for problems whose database is changing over (discrete) time. The idea is to make use of good data structures for a static (fixed) database and add to them certain dynamization mechanisms so that insertions or deletions of elements in the database can be accommodated efficiently.

11.1 On-line construction of convex hulls

Each of the convex hull algorithms we have examined thus far requires all of the data points to be present before any processing begins. In many geometric applications, particularly those that run in real-time, this condition cannot be met and some computation must be done as the points are being received. In general, an algorithm that cannot look ahead at its input is referred to as *on-line*, while one that operates on all the data collectively is termed *off-line*. Obviously, given a problem, an on-line algorithm cannot be more efficient than the best off-line algorithm.

Let us formally describe the problem.

ON-LINE HULL

Given a sequence of n points $p_1, p_2, \cdots, p_n$ in the plane, find their convex hull in such a way that after p_i is processed we have the convex hull for the set of points $\{p_1, p_2, \cdots, p_i\}$.

Let CH_i denote the convex hull of the i points p_1, p_2, $\cdots$, p_i. The ON-LINE HULL problem is obviously reduced to the following problem: for

$i = 1, \cdots, n-1$, suppose that we have the convex hull CH_i, we "insert" the point p_{i+1} properly into CH_i to obtain the convex hull CH_{i+1}. Thus, an algorithm for ON-LINE HULL should look pretty much like the algorithm HEAPSORT, as we studied in Algorithm Analysis, where we always keep a sorted list for the first i numbers, and insert the $(i+1)$st number to the list to form a sorted list of the first $(i+1)$ numbers. In fact, we will use a technique pretty similar to HEAPSORT to solve the ON-LINE HULL problem.

An on-line algorithm must spend at least time $\Omega(n \log n)$, when the last point p_n has been processed, since we have shown that $\Omega(n \log n)$ is a lower bound for off-line algorithms of construction of convex hulls. Therefore, the best we can expect is to insert p_{i+1} into the hull CH_i in time $O(\log n)$. In other words, if an algorithm inserts each point p_{i+1} into the convex hull CH_i in time $O(\log n)$, for $i = 1, 2, \cdots, n-1$, then the algorithm is optimal.

Now let us see how we insert the point p_{i+1} into the convex hull CH_i. There are two possible cases, either the point p_{i+1} is internal to the convex hull CH_i, then $CH_i = CH_{i+1}$ and we do nothing; or the point p_{i+1} is external to the convex hull CH_i, then we have to construct the two bridges from the point p_{i+1} to the convex hull CH_i and form the convex hull CH_i. Therefore, the algorithm should look like the following:

Algorithm *INSERTVERTEX*
begin

1. **if** p_{i+1} *is internal to* CH_i*, then do nothing.*

2. **else** *find the two bridges* B_1 *and* B_2 *from* p_{i+1} *to* CH_i*. Let* q_1 *and* q_2 *be the points in* CH_i *that are on the bridges* B_1 *and* B_2*, respectively, replace a chain on* CH_i *that is between* q_1 *and* q_2 *by two line segments* $\overline{q_2 p_{i+1}}$ *and* $\overline{p_{i+1} q_1}$.

end

Step 2 in the above algorithm involves SEARCHING the points q_1 and q_2, DELETING a chain on CH_i between q_1 and q_2, and INSERTING the point p_{i+1}. To make our algorithm optimal, all these operations should be done in time $O(\log n)$. Thus the 2-3 trees introduced in Chapter 1 seems a proper data structure for this purpose.

Let us store the convex hull CH_i in a 2-3 tree T in the following way. The hull vertices of CH_i are stored in the leaves of T from left to right in the counterclockwise ordering on the hull CH_i. Each non-leaf vertex v

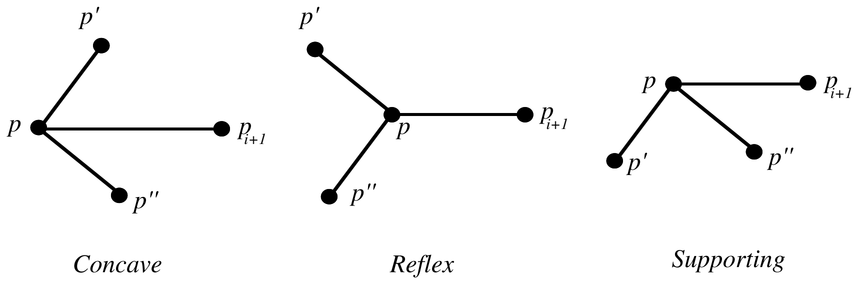

Figure 11.1: Concave, reflex, and supporting points

of T keeps three pieces of information, $L(v)$, $M(v)$, and $R(v)$, where $L(v)$ contains the right most hull vertex v_l stored in the subtree rooted at the left son of v, together with the two neighbors of v_l on the hull CH_i. Similarly, $M(v)$ and $R(v)$ contains the right most hull vertex v_m and v_r stored in the subtrees rooted at the middle son and right son of v, respectively, together with their two neighbors on the hull CH_i.

Let p be a hull vertex of CH_i and let p' and p'' be the two neighbors of p on CH_i. Draw a line segment $\overline{p_{i+1}p}$ between the point p and p_{i+1}. Let α be the angle that is less than π and formed by the line segments $\overline{pp'}$ and $\overline{pp''}$, and let l be the straight line passing through the two points p and p_{i+1}. We say that the vertex p is *concave with respect to* p_{i+1} if the points p' and p'' are in two different sides of the line l and the point p_{i+1} is within the wedge of the angle α. The point p is *reflex with respect to* p_{i+1} if the points p' and p'' are in two different sides of the line l and the point p_{i+1} is outside the wedge of the angle α. The point p_{i+1} is *supporting with respect to* p_{i+1} if the two points p' and p'' are in the same side of the line l. Figure 11.1 depicts these three different cases.

Note that given the points p, p', p'', and p_{i+1}, we can decide in constant time if the point p is concave, reflex, or supporting with respect to the point p_{i+1}.

It is easy to see that if p_{i+1} is internal to CH_i, then all hull vertices of CH_i are concave with respect to p_{i+1}, and if p_{i+1} is external to CH_i, then exactly two hull vertices of CH_i are supporting with respect to p_{i+1}, which are the two points q_1 and q_2 in Step 2 in the algorithm INSERTVERTEX.

Without loss of generality, we suppose that q_1 is the "right supporting point" with respect to p_{i+1} such that the whole convex hull CH_i lies on the left side of $\overline{p_{i+1}q_1}$ (we say that a point q is on the left (right) side of a directed line segment $\overline{p'p''}$ if q is on our left (right) side when we travel the straight line passing through the points p' and p'' in the direction from p' to p''). Similarly, we call the point q_2 the "left supporting point" with respect to p_{i+1}.

Suppose that the point p_{i+1} is external to the convex hull CH_i, we discuss how to find the right and left supporting points q_1 and q_2 with respect to p_{i+1} in the 2-3 tree representing the convex hull CH_i. Since the algorithms are similar for finding the right supporting point and left supporting point, we only discuss the algorithm for finding the right supporting point q_1.

Recursively, suppose that we know that the point q_1 is stored in a subtree $T(v)$ rooted at a non-leaf vertex v. The root v contains three pieces of information $L(v)$, $M(v)$, and $R(v)$ for its left son *LSON(v)*, middle son *MSON(v)*, and right son *RSON(v)*, respectively. We first use the information stored in $M(v)$ and $R(v)$ to decide if q_1 is stored in the right son *RSON(v)* of v. If q_1 is not stored in the right son *RSON(v)*, then we use the information stored in $L(v)$ and $M(v)$ to decide that q_1 is stored in the left son *LSON(v)* or in the middle son *MSON(v)*. Since the methods for these two decisions are similar, we only describe how we use the information $L(v)$ and $M(v)$ to decide in which of the left and middle sons the right supporting point q_1 is stored. Since $L(v)$ contains the right most point v_l in the subtree *LSON(v)* together with its two neighbors on the convex hull CH_i, we can decide in constant time that the point v_l is concave, reflex, or supporting with respect to p_{i+1}. If v_l is the right supporting point with respect to p_{i+1}, then we are done. Therefore, we only have to consider the cases that v_l is concave, reflex, or left supporting, with respect to p_{i+1}. Again, the processes for these three cases are quite similar, we only discuss the case that v_l is reflex with respect to p_{i+1}.

So we suppose that v_l is reflex with respect to p_{i+1}. Let the right most point in the subtree *MSON(v)* be r_m (the point r_m and its neighbors on CH_i are contained in the information $M(v)$ of v. There are six different positions, relative to the position of v_l, for v_m to locate:

1. The point v_m is concave with respect to p_{i+1}, and v_m is on the right of $\overline{p_{i+1}v_l}$.

2. The point v_m is concave with respect to p_{i+1}, and v_m is on the left of $\overline{p_{i+1}v_l}$.

3. The point v_m is reflex with respect to p_{i+1}, and v_m is on the right of

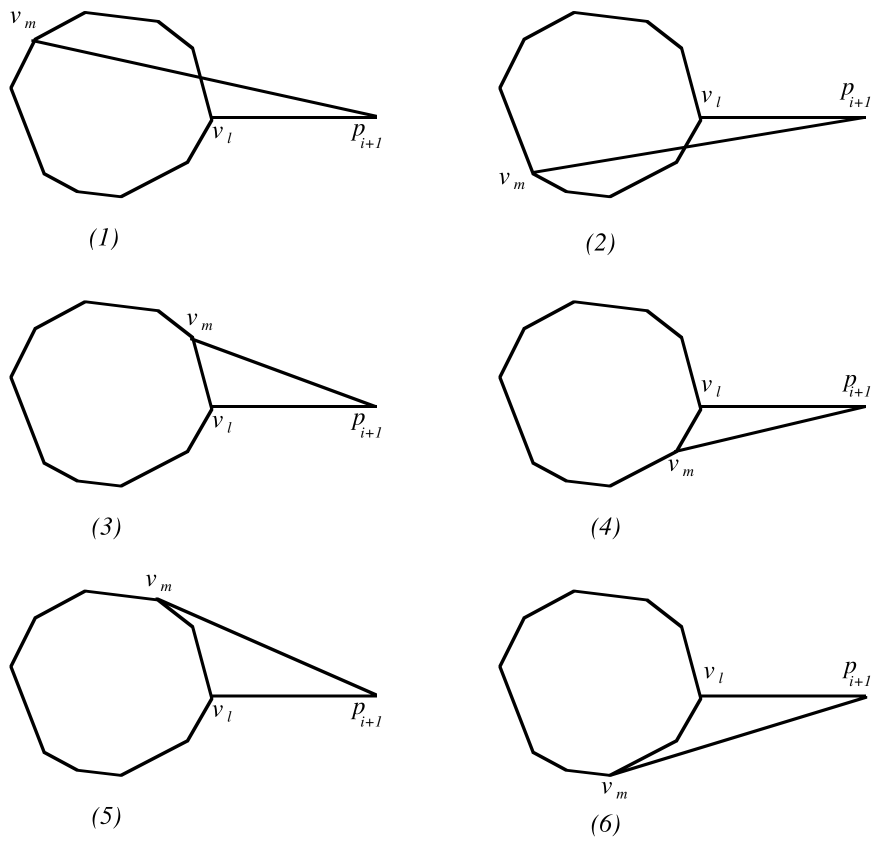

Figure 11.2: Six positions for v_m when v_l is reflex

$\overline{p_{i+1}v_l}$.

4. The point v_m is reflex with respect to p_{i+1}, and v_m is on the left of $\overline{p_{i+1}v_l}$.

5. The point v_m is right supporting with respect to p_{i+1}.

6. The point v_m is left supporting with respect to p_{i+1}.

Figure 11.2 illustrates all these six cases.

From Figure 11.2, it is easy to decide in which subtree the right supporting point q_1 with respect to p_{i+1} is stored. We discuss this case by case.

Remember that the hull vertices of CH_i are stored in the 2-3 tree from left to right in the counterclockwise ordering, therefore, the points stored in the subtree *MSON(v)*, together with the point v_l, correspond to the chain on the convex hull CH_i starting from the point v_l, making travel in counterclockwise order, and ending at the point v_m. Call this chain a *MSON-chain.*

CASE 1 In this case, v_l is reflex and v_m is concave. If we travel the MSON-chain from v_l to v_m, the points on the chain change from reflex to concave with respect to p_{i+1}, and all points are on the right of $p_{i+1}v_l$. Thus we must pass the right supporting point q_1. Therefore, in this case, the right supporting point q_1 is stored in the middle son *MSON(v).*

CASE 2 The analysis is similar to Case 1, the right supporting point q_1 is stored in the middle son *MSON(v).*

CASE 3 In this case, both v_l and v_m are reflex, and v_m is on the right side of $\overline{p_{i+1}v_l}$. Therefore, if we travel the MSON-chain from v_l to v_m, the right supporting point q_1 must not be passed. Therefore, in this case the middle son MSON(v) does not contain the right supporting point q_1. The point q_1 must be stored in the left subtree *LSON(v).*

CASE 4 Similar to Case 2, the right supporting point q_1 is stored in the middle son *MSON(v).*

CASE 5 This is the most lucky case, since the right supporting point $q_1 = v_m$.

CASE 6 Similar to Case 2, the right supporting point q_1 is stored in the middle son *MSON(v).*

Therefore, for each of these six cases, we can decide in constant time which subtree we should further search. We summarize these discussions in the following algorithm.

Algorithm *RIGHTPOINT(v)*

{ *Search the right supporting point q_1 in the subtree rooted at the non-leaf vertex v. The points v_l and v_m are the right most points in the subtrees LSON(v) and MSON(v), respectively.* }

begin

1. **if** p_{i+1} *is external to* CH_i

1.1 **if** q_1 *is stored in RSON(v), call RIGHTPOINT(RSON(v));*

1.2 **else**

1.2.1 **if** v_l *is reflex*

1.2.1.1 **if** v_m *is right supporting, then done;*

1.2.1.2 **else if** v_m *is reflex and on the right of* $\overline{p_{i+1}v_l}$

1.2.1.3 *Call RIGHTPOINT(LSON(v));*
1.2.1.4 **else** *Call RIGHTPOINT(MSON(v));*
1.2.2 **else if** v_l *is concave* $\cdots$
1.2.3 **else if** v_l *is supporting* $\cdots$

2. **else if** p_{i+1} *is internal to* CH_i
2.1 $\cdots$

end

We give a few remarks on the above algorithm.

1. To decide if the point q_1 is stored in *RSON(v)* in Step 1.1, we use the method similar to those in Steps 1.2.1 - 1.2.3. The only exception is that the information used is *MSON(v)* and *RSON(v)*, instead of *LSON(v)* and *MSON(v)*.

2. We actually do not need Step 2 to check if p_{i+1} is internal to CH_i. In fact, if p_{i+1} is internal to CH_i, the recursive calls of Step 1 eventually locate a single point q_1 on the convex hull CH_i, and this point q_1 is still concave with respect to p_{i+1}. Since if the point p_{i+1} is external to CH_i, then the final point q_1 must be the right supporting point, so if we find out that the final point q_1 is still concave with respect to p_{i+1}, then we conclude that the point p_{i+1} is internal to CH_i.

3. The left supporting point q_2 is found by a similar subroutind *LEFT-POINT(v)*.

4. With the above discussions and the similarities, the reader should have no trouble to fill up the omitted part in the algorithm.

Therefore, to find the right and left supporting points q_1 and q_2 in the convex hull CH_i, which is represented by a 2-3 tree T rooted at v, we simply call

RIGHTPOINT(v); *LEFTPOINT(v)*

By the discussions above, these two supporting points can be found in time $O(\log n)$.

Since the subroutines also tell us if p_{i+1} is internal to CH_i, so if we are told that p_{i+1} is internal to CH_i, then $CH_i = CH_{i+1}$ and we are done. Otherwise, the right and left supporting points q_1 and q_2 are returned. Let C be the chain between q_1 and q_2 in the tree T. Pick any point q in the chain C. If the point q is reflex, then all hull vertices in the chain C should be deleted and all other hull vertices should be kept. On the other hand, if the point q is concave, then all hull vertices in the chain C should be kept and all other hull vertices should be deleted. Therefore, we first split the the tree T into three trees T_1, T_2, and T_3 such that the leaves of the tree T_2 are those points that are in the chain C. In the case that q is reflex, we splice the two trees T_1 and T_3 into a new tree T', and in the case that q is concave, we let T_2 be the new tree T'. It is clear to see that the new tree T' corresponds to the partial chain in CH_i that should be kept in the convex hull CH_{i+1}. Moreover, since the data structure we are using is a 2-3 tree, these split and splice operations can be done in time $O(\log n)$. Finally, we insert in time $O(\log n)$ the point p_{i+1} into the tree T' to form the 2-3 tree representing the convex hull CH_{i+1}.

Summarize the above discussions, we conclude that constructing the convex hull CH_{i+1} from the convex hull CH_i can be done in time $O(\log n)$. This consequently gives us the following theorem.

Theorem 11.1.1 *The ON-LINE HULL problem can be solved by an optimal algorithm.*

Chapter 12

Randomized Methods

This chapter may contain the following materials: expected time for constructing convex hulls in 2-dimensional space (Preperata and Shamos, see also Overmars Lecture Notes), expected time for constructing intersection of half-spaces in 3-dimensional space. The papers by Clarson should be read to find more examples.

Chapter 13

Parallel Constructions

Parallel random access machine (PRAM)

The computational model we are based on in this chapter is called *parallel random access machine (PRAM)*. This kind of machine model is also known as the Shared-Memory Single Instruction Multiple Data computer. Here, many processors share a common (random access) memory that they use in the same way a group of people may use a bulletin board. Each processor also has its own local memory in which the processor can save its own intermediate computational results. When two processors wish to communicate, they do so through the shared memory. Say processor P_i wishes to pass a number to processor P_j. This is done in two steps. First, processor P_i writes the number in the shared memory at a given register which is known to processor P_j. Then, processor P_j reads the number from that register.

The number of processors of a PRAM, the size of the shared memory, and the size of the local memory for each processor are all assumed to be unbounded.

Depending on the way of simultaneous access of a register in the shared memory, the class of PRAM can further be subdivided into four subclasses: EREW PRAM, CREW PRAM, ERCW PRAM, and CRCW PRAM. We are not going to discuss the details in this book.

13.1 Parallel construction of convex hulls

Our last discussion on the construction of convex hulls is a description of a parallel algorithm constructing a convex hull given a set of n points in the plane.

How do we evaluate a parallel algorithm? First of all, the computation time, here we call *parallel time* of a parallel algorithm is important. Moreover, a second important criterion in evaluating a parallel algorithm is the *number of processors* the algorithm requires during its computation. Therefore, a good parallel algorithm should not only run in least time, but also use least number of processors.

To describe the parallel algorithm constructing convex hulls for planar points, we need to be able to solve some elementary problems efficiently in parallel. We list below the parallel complexity for these problems, and explain briefly the basic idea of the parallel algorithms solving these problems. For more detailed discussions on efficient parallel algorithms, the reader is referred to [3].

MAXIMUM

Given n numbers, find the maximum number.

Theorem 13.1.1 *MAXIMUM can be solved in $O(\log n)$ parallel time using $O(n)$ processors.*

PROOF. To find the maximum number in a list of n numbers, we first use $n/2$ processors, each picks a pair of numbers and compares them. Then algorithm is recursively applied on the $n/2$ winners. □

LISTRANK

Given a linked list of n elements, compute the rank for each element. That is, for the ith element in the list, we compute the number $n - i$.

Theorem 13.1.2 *LISTRANK can be solved in $O(\log n)$ parallel time using $O(n)$ processors.*

PROOF. Since the idea of the algorithm PARALLELRANK is so basic and will be used later, we describe it here in a little more detail.

We assume that the linked list is represented by a contents array $c[1 \dots n]$ and a successor array $s[1 \dots n]$. Here for all i, $c[i]$ is initialized to 1 except that for the last element, $c[n] = 0$, and $s[i]$ is initialized to point to the next element in the linked list except that for the last element, $s[n]$ points to the nth element itself. In general, $c[i]$ is the distance between the element i

and the element pointed by $s[i]$. The following simple algorithm solves the LISTRANK problem.

Algorithm *PARALLELRANKING*
begin
1. **for** $\log n$ *iteration repeat*
2. *In parallel,* **for** $i = 1, \cdots, n$ **do**
3. $c[i] = c[i] + c[s[i]]; \quad s[i] = s[s[i]].$
end

The operation used in this algorithm of replacing each pointer $s[i]$ by the pointer's pointer $s[s[i]]$ is called *pointer jumping*, and is a fundamental technique in parallel algorithm design. The correctness and time complexity of the algorithm can be obtained by inductively proving the following two claims: for all $i = 1, \cdots, n$, (1) at the start of each iteration, $c[i]$ is the distance between the element i and the element pointed by $s[i]$; and (2) after $\log(n-i)$ iterations, the point $s[i]$ is pointing to the last element in the linked list.

We can use one processor for each element. Then in each iteration, Step 2 and Step 3 can be executed in constant time by the processor for the element. We conclude that in parallel time $O(\log n)$ and using $O(n)$ processors, the LISTRANK problem can be solved. □

ARRAY-COMPRESSION

Let A be an array containing $m = n + n'$ elements, n of them are red and n' of them are blue. Delete all blue elements and compress all red elements into an array A' of size n.

Theorem 13.1.3 *ARRAY-COMPRESSION can be solved in parallel time $O(\log m)$ using $O(m)$ processors.*

PROOF. Initially, make each element of the array A a linked list of a single element. Then for each pair of linked lists which correspond to two consecutive elements in the array A, combine them into a single linked list. In this process, if both linked lists are red elements, then simply connect the tail of the first to the head of the second and make a linked list of two red elements; if exactly one linked list is a red element, then ignore the linked list of a blue element; finally, if both are linked list of blue elements, then let the new linked list be a linked list of a single blue element. Recursively combine these new linked lists. It is easy to see that after at most $\log m$ iterations, all red elements are stored in a linked list, and all blue elements are thrown away. Now use the PARALLELRANKING algorithm to compute the rank for each red element in the final linked list. With the rank for each red element in the final linked list, a processor can copy the red element directly to the array A' in constant time. □

Theorem 13.1.4 *SORTING can be solved in time $O(\log n)$ using $O(n)$ processors.*

PROOF. See [9]. □

Now we are ready for describing the parallel algorithm for constructing convex hulls for planar points. The algorithm looks as follows.

Algorithm *PARALLELHULL*

{ *Given a set S of n planar points stored in an array A, find the convex hull CH(S) of S.* }

begin

1. *Find the pair of points $p_{\min}$ and $p_{\max}$ with the maximum and minimum x-coordinates;*
2. *Partition the set S into two sets S_1 and S_2 such that S_1 is the set of points in S that are above the segment $\overline{p_{\min}p_{\max}}$, and S_2 is the set of points in S that are below the segment $\overline{p_{\min}p_{\max}}$;*
3. *Sort S_1 by x-coordinate, and sort S_2 by x-coordinate;*
4. *construct the upper hull UH for the set S_1, and construct the lower hull LH for the set S_2;*
5. *Merge UH and LH to get the convex hull CH(S).*

end

Step 1 in the algorithm PARALLELHULL can be done in $O(\log n)$ parallel time using $O(n)$ processors by Theorem 13.1.1. Step 2 can be done in $O(\log n)$ parallel time using $O(n)$ processors in the following way: we use two new arrays A_1 and A_2. A single processor is used for each i, $i = 1, \cdots, n$, to decide if the ith point p_i is above or below $\overline{p_{\min} p_{\max}}$. If p_i is above $\overline{p_{\min} p_{\max}}$, put p_i in the ith position in the array A_1, otherwise, put p_i in the ith position in the array A_2. By Theorem 13.1.3, the arrays A_1 and A_2 can be compressed in $O(\log n)$ parallel time using $O(n)$ processors. Step 3 can be done in $O(\log n)$ parallel time using $O(n)$ processors by Theorem 13.1.4. Step 5 can obviously be done in $O(\log n)$ parallel time using $O(n)$ processors.

Therefore, if Step 4 of the algorithm can be done in $O(\log n)$ parallel time using $O(n)$ processors, then the algorithm PARALLELHULL takes $O(\log n)$ parallel time and $O(n)$ processors.

Since constructions of the upper hull UH and the lower hull LH are similar, we only discuss the algorithm for constructing the upper hull.

Algorithm *UPPER-HULL*

{ *Given the set S_1 of n planar points sorted by x-coordinates in an array A_1, construct the upper hull UH of S_1 and put it in an array B_1.* }

begin

1. *Partition the array A_1 into $\sqrt{n}$ subarrays each containing $\sqrt{n}$ consecutive elements in A_1;*
2. *Recursively construct the upper hull for the points in each subarray (in parallel) (call them upper subhulls);*
3. *Merge these $\sqrt{n}$ upper subhulls into the upper hull UH of S_1.*

end

First let us assume that Step 3 in the algorithm UPPER-HULL can be done in parallel time $O(\log n)$ using $O(n)$ processors. Then since the $O(n)$ processors are "reusable" in Step 2, we conclude that the algorithm *UPPER-HULL* uses $O(n)$ processors. Moreover, since Step 1 can obviously be done in $O(\log n)$ parallel time using $O(n)$ processors, if we suppose that the parallel running time of the algorithm is $T(n)$, then we have the recurrence relation

$$T(n) \leq c \log n + T(\sqrt{n})$$

It is not hard to see that $T(n) = O(\log n)$.

Therefore, the problem is finally reduced to merging those $\sqrt{n}$ upper subhulls into the upper hull of S_1 in $O(\log n)$ parallel time using $O(n)$ processors. This is the most non-trivial part of our algorithm.

Consider two upper subhulls H and H'. Since we divide S_1 by x-coordinate, this ensures that the x-coordinates of any two upper subhulls do not overlap. By recursive assumption, the upper subhulls H and H' are stored in arrays in sorted order by x-coordinate. A single processor can compute in $O(\log n)$ time the unique line which is tangent to both upper subhulls, together with the two points of tangency. This can be done by a binary search that is similar to the searching procedure we discussed in the last section for on-line convex hull construction. For each pair of upper subhulls, construct the corresponding tangent. Since there are totally $\binom{\sqrt{n}}{2} < n$ such pairs of upper subhulls, we can use n processors to construct all these tangents in parallel time $O(\log n)$.

Now we have n' vertices, $n' \leq n$, which are the vertices on the $\sqrt{n}$ upper subhulls. Moreover, we have m edges, $m \leq 2n$, which are the edges on the $\sqrt{n}$ upper subhulls and the tangents we constructed above for pairs of upper subhulls. Note that all edges on the final upper hull UH are within these $2m$ edges. For each edge with two endpoints v_1 and v_2, we make two directed edges, one is (v_1, v_2) and the other is (v_2, v_1). Now for these $2m$ directed edges, we sort them by the first component. What we will obtain is an array in which all edges incident on a vertex are consecutive. By Theorem 13.1.4, this can be done in parallel time $O(\log n)$ using $O(n)$ processors. Then for each consecutive subarray corresponding to the set of edges incident on the same vertex v, we find the two edges l_v and r_v of the smallest and largest slope with respect to the vertex v (the angle of a slope is measured from $-3\pi/2$ to $\pi/2$). This can also be done in parallel time $O(\log n)$ using $O(n)$ processors, by Theorem 13.1.1.

The edges l_v and r_v form a "roof" at the vertex v. By the construction of all these $2m$ edges, it is easy to see that for any vertex v that is neither $p_{\min}$ nor $p_{\max}$, the two edges l_v and r_v must exist and must be in two different sides of the vertical line through v. Therefore, we will call l_v and r_v the *left roof* and the *right roof*, respectively. The vertex $p_{\min}$ has only right roof, and the vertex $p_{\max}$ has only left roof.

If the angle formed by l_v and r_v is greater than π (measured counter-clockwise from the left roof l_v to the right roof r_v), then clearly the vertex v cannot be on the final upper hull *UH*, so we mark the vertex v by 0, meaning "not on *UH*". The two corresponding roof edges l_v and r_v are also ignored. Moreover, we ignore all edges that are not a roof edge for any vertex.

For each v of those vertices that have not been marked 0, we try to travel from it "from left to right" through roof edges that have not been ignored. Note that if a vertex v is on the upper hull UH, then the two roof edges of

v must be on the upper hull UH, and the other endpoint of the right roof edge of v must be the next vertex on the upper hull UH, whose roof edges are again edges on UH. Therefore, if we start with a vertex v on the upper hull UH, the trip will be a partial chain on the upper hull UH between the vertex v and the vertex $p_{\max}$, and eventually lead us to the vertex $p_{\max}$. On the other hand, if v is not on the upper hull UH, the trip from v will lead us either to a dead vertex (i.e., a vertex that has no right roof edge or a vertex that has been marked 0) that is not the vertex $p_{\max}$, or to a vertex w such that the edge leading us to w is not the left roof edge of w. Therefore, starting at a vertex v, we can decide if v is on the upper hull UH by this kind of traveling. This kind of traveling is very similar to the traveling we discussed for PARALLELRANKING. In fact, for each vertex v that is not marked 0, a processor can first check if v is a "direct dead" vertex (i.e., a vertex w that is not $p_{\max}$ and either has no right roof edge, or the right roof edge (w, u) is not the left roof edge of the other endpoint u). If v is a direct dead vertex, make the successor $s[v]$ of v point to v itself, and let $c[v] = 0$ (like the last element in the linked list in our algorithm PARALLELRANKING). For all other vertex v, set $c[v] = 1$ and let $s[v]$ point to the other endpoint of the left roof edge of v. Now exactly like in the algorithm PARALLELRANKING, after at most $O(\log n)$ iterations of pointer jumping, the travels from all vertices are finished. If the successor of a vertex v now is the vertex $p_{\max}$, then the vertex v is on the upper hull UH, otherwise, the successor of v is a direct dead vertex and the vertex v is not on the upper hull UH. Mark all vertices on the upper hull UH by 1, and mark all vertices not on the upper hull UH by 0. Now (perhaps after another sorting by x-coordinate) delete the vertices marked 0 using ARRAY-COMPRESSION, and we eventually obtain the upper hull UH of S_1, which is stored in an array. Notice the the upper hull UH now is also ready for the further recursive calls.

Since PARALLELRANKING and ARRAY-COMPRESSION both can be done in $O(\log n)$ parallel time using $O(n)$ processors, we conclude that the merge part (Step 3) in algorithm UPPER-HULL can be done in $O(\log n)$ parallel time using $O(n)$ processors.

This completes our description of the $O(\log n)$ parallel time, $O(n)$ processor parallel algorithm for constructing convex hulls for planar points.

Bibliography

[1] A. Aggarwal and J. Wein, *Computational Geometry*, MIT Technical Report, MIT/LCS/RSS 3, (1988).

[2] A. V. Aho, J. E. Hopcropt, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., (1974).

[3] S. G. Akl, *The Design and Analysis of Parallel Algorithms*, Prentice Hall, Engiewood, N.J., (1989).

[4] J. L. Bentley and M. I. Shamos, A problem in multivariate statistics: Algorithms, data structure, and applications, *Proc. 15th Allerton Conf. Commun., Contr., and Comput.*, (1977), pp.193-201.

[5] M. Blum, W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan, Time bounds for selection, *J. Computer and System Sciences 7*, (1972), pp.448-461.

[6] C. B. Boyer, *A History of Mathematics*, New York: Wiley, (1968).

[7] K. Q. Brown, Geometric transforms for fast geometric algorithms, *Tech. Report CMU-CS-80-101*, Carnegie-Mellon, (1979).

[8] B. Chazelle, Triangulating a simple polygon in linear time, *Discrete and Computational Geometry*, (1991), pp. 485-524.

[9] R. Cole, Parallel merge sort, *SIAM J. Computing 17*, (1988), pp.770-785.

[10] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Berlin, (1987).

[11] H. Edelsbrunner, J. O'Rourke, and R. Seidel, Constructing arrangements of lines and hyperplanes with applications, *SIAM J. Computing 15*, (1986), pp.341-363.

[12] D. G. Kirkpatrick, Optimal search in planar subdivisions, *SIAM J. Computing 12*, (1983), pp.28-35.

[13] D. G. Kirkpatrick and R. Seidel, The ultimate planar convex hull algorithm? *SIAM J. Computing 15*, (1986), pp.287-299.

[14] D. E. Knuth, *The Art of Computer Programming. Volume III: Sorting and Searching*, Addison-Wesley, Reading, Mass., (1973).

[15] J. B. Kruskal, On the shortest spanning subtree of a graph and the traveling salesman problem, *Proc. AMS 7*, (1956), pp.48-50.

[16] D. T. Lee and F. P. Preparata, Computational Geometry - A Survey, *IEEE Transitions on Computers C-33, No. 12*, (1984), pp.1072-1101.

[17] P. McMullen and G. C. Shephard, *Convex Polytopes and the Upper Bound Conjecture*, Cambridge University Press, Cambridge, England, (1971).

[18] K. Mehlhorn, *Multidimensional Searching and Computational Geometry*, Springer-Verlag, Berlin, (1984).

[19] W. M. Newman and R. F. Sproull, *Principles of Interactive Computer Graphics*, McGraw-Hill, New York, (1979).

[20] J. O'Rourke, *Art Gallery Theorems and Algorithms*, Oxford, New York, (1987).

[21] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Englewood Cliffs, NJ: Prentice Hall, (1982).

[22] T. Pavlidis, *Algorithms for Graphics and Image Processing*, Springer-Verlag, Berlin, (1982).

[23] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, (1985).

[24] R. C. Prim, Shortest connection networks and some generalizations, *Bell Sys. Tech, J. 36*, (1957), pp.1389-1401.

[25] J. Schwartz, M. Sharir, and J. Hopcroft, *Planning, Geometry, and Complexity of Robot Motion*, Ablex Publishing Co., Norwood, New Jersey, (1987).

[26] J. Schwartz and C. Yap, *Algorithmic and Geometric Aspects of Robotics*, Vol. 1, Erlbaum, Hillsdale, New Jersey, (1987).

[27] M. I. Shamos, Geometry and statistics: Problems at the interface, in *Algorithms and Complexity*, J. F. Traub, Ed., Academic, New York, (1976), pp.251-280.

[28] G. T. Toussaint, Pattern recognition and geometrical complexity, *Proc. 5th Int. Conf. Pattern Recog.*, (1980), pp.1324-1347.

Contents

List of Figures