Chapter 6

Prune and Search

Prune and Search is a technique developed by Blum, Floyd, Pratt, Rivest, and Tarjan [5], originally used for finding the median of an unorganized set of elements. The technique, as applied to median finding, throws out in linear-time a constant fraction of the elements during each iteration of a loop. Solving the recurrence relation for such a procedure shows that the algorithm finds the medain of a set of n elements in time $O(n)$.

Let us have a more detailed review of the algorithm. We first generalize the problem a little bit by considering instead the problem of finding the k-th smallest element in a set S of n elements, for an arbitrary k that is also given in the input of the problem. To do this, we first divide, in time $O(n)$, the n elements in S into $n/5$ groups, with 5 elements per group, then find the median for each of the groups (note that finding the median in a group of 5 elements takes constant time). Let S' be the set of these $n/5$ group-medians. Recursively we find the median m of the set S'. Since m is not smaller than the group medians for half of the $n/5$ groups, while each group median is not smaller than 3 elements in its group, the value m is not smaller than at least $(1/2) \cdot (n/5) \cdot 3 = 3n/10$ elements in the original set S. Similarly, m is not larger than at least $3n/10$ elements in S. Therefore, using the element m, we can partition in time $O(n)$ the set S into two subsets S_1 and S_2 such that all elements in S_1 are less than or equal to m and all elements in S_2 are greater than or equal to m. Moreover, the size of each of these subsets S_1 and S_2 is at least $3n/10$. As a consequence, the size of each of the subsets S_1 and S_2 is bounded by $7n/10$. Now if the subset S_1 contains at least k elements, then recursively call the algorithm to find the k-th smallest element in the subset S_1. On the other hand, if the subset S_1 contains h elements such that $h < k$, then recursively call the algorithm to

find the $(k-h)$-th smallest element in the subset S_2. In either case, the recursive call works on a subset of size bounded by $7n/10$. A detailed and formal description 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 an input of n elements. Then to find the median of the set S' of the $n/5$ group-medians takes time $T(n/5)$. Since both subsets S_1 and S_2 are of size bounded by $7n/10$, to find the k-th smallest element in the subset S_1 or to find the $(h-k)$-th smallest element in the subset S_2 takes time at most $T(7n/10)$. It is also clear, from the above discussion, that the computation for the rest of the algorithm can be done in time bounded by bn, where b is a constant. Therefore, the function $T(n)$ satisfies the following recurrence relation:

$$T(n) \leq T(n/5) + T(7n/10) + bn.$$

Let g be an integer such that $g \geq 10b$ and $g \geq T(10)$ (for a set of no more than 10 elements, the algorithm will solve the problem in constant time using the brute-force method), then it is not difficult to prove by induction on the number n of elements in the input set that

$$T(n) \leq gn$$

That is, $T(n) = O(n)$. This gives a linear-time algorithm that finds the k-th smallest element in a set of n elements for an arbitrarily given k. The method used in the algorithm, that is, pruning a constant fraction of the input elements then searching in the set of the rest elements, is called *Prune-and-Search*, or *Decrease-and-Conquer* in the literature.

The technique can be used to solve other problems. A general form of a Prune-and-Search algorithm can be described, informally, as following.

```
Algorithm Prune-and-Search
Input: a problem instance S of size n
Output: a solution y to the instance S
1. If (the size n of S is small)
     solve S directly and Stop;
2. prune the instance S into h smaller instances S(1),...,S(h) of size
   c(1)*n,...,c(h)*n, respectively, such that c(1)+...+c(h) <= c < 1,
   where c is a fixed constant;
3. recursively solve the instances S(1),...,S(h);
4. use the results of step 3 to get a solution y to the instance S.
```

Let the time complexity of the algorithm `Prune-and-Search` be $T(n)$ on an input instance of size n. Suppose that steps 1, 2 and 4 of the algorithm take time $F(n)$. Then the function $T(n)$ can be represented by the following recurrence relation:

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

The time complexity $T(n)$ of the algorithm `Prune-and-Search` can be obtained by solving the above recurrence relation. In particular, if $F(n) = O(n)$, then it can be proved that the function $T(n)$ is also $O(n)$.

6.1 Output sensitive algorithms for convex hulls

Having seen a number of algorithms for constructing convex hulls for point sets in the plane, we present yet another few algorithms for the problem. These algorithms are *output sensitive*, in the sense that their complexity depends not only on the input size but also on the output size. An example of output sensitive algorithms is the `Jarvis-March` algorithm that constructs the convex hull of k vertices for a set of n points in the plane in time $O(kn)$.

In this section, we present two algorithms for constructing convex hulls of k vertices for a set of n points in the plane in time $O(n \log k)$. Since $k \leq n$, asymptotically, these algorithms are better than `Jarvis-March` whose complexity is $O(kn)$, and cannot be worse (in fact, can be much better when n is much larger than k) than `Graham-Scan` whose complexity is $O(n \log n)$. In fact, these algorithms have been called the "ultimate" algorithms for the convex hull problem, because it can be proved that any algorithm for convex hulls will take time at least $\Omega(n \log k)$.

6.1.1 Kirkpatrick-Seidel algorithm

The first algorithm is due to Kirkpatrick and Seidel [14] based on the prune and search technique.

Consider two sets S_L and S_R of points in the plane that are separated by a vertical line. That is, the x-coordinate of every point in the set S_L is smaller than the x-coordinate of any point in the set S_R. A segment $[p_l, p_r]$ is an *upper bridge* for (S_L, S_R) if $p_l \in S_L$ and $p_r \in S_R$ and no points in $S_L \cup S_R$ is above the line passing through the segment $[p_l, p_r]$. The upper bridge is obviously an edge of the convex hull for the set $S_L \cup S_R$. Moreover, if we assume that no three points are co-linear, then there are a unique point p_l in S_L and a unique point p_r in S_R such that the segment $[p_l, p_r]$ makes an upper bridge for (S_L, S_R).

Consider the following problem.

> Upper-Bridge
> Given two sets S_L and S_R of points in the plane that are separated by a vertical line, construct the upper bridge for (S_L, S_R).

In the algorithm `MergeHull` (see Chapter 5, section 1), we know that when the convex hulls $\mathrm{CH}(S_L)$ and $\mathrm{CH}(S_R)$ for the sets S_L and S_R are known, the upper bridge for (S_L, S_R) can be constructed in linear time. However, constructing the convex hulls for the sets S_L and S_R itself takes time $\Omega(n \log n)$, which is too much to us. What we expect is a linear-time algorithm that solves the UPPER-BRIDGE problem.

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

For a line L, we will denote by $slp(L)$ the slope of the line L, which is between $-\pi/2$ and $\pi/2$. Similarly, the slope $slp(p, q)$ of a segment $[p, q]$ is interpreted as the slope of the vector directed from the point p to the point q. For a segment $[p, q]$, we will always assume that the x-coordinate of the point p is not larger than that of the point q. As a consequence, the slope of a segment is always between $-\pi/2$ and $\pi/2$.

A line L is an *upper supporting line* of the point set $S_L \cup S_R$ if L contains points of $S_L \cup S_R$, and there are no points of $S_L \cup S_R$ above L. In particular, if an upper supporting line contains points in both S_L and S_R, then it contains the upper bridge for (S_L, S_R). Similarly, we can define *lower supporting lines* of the point set S. We have the following observations.

Lemma 6.1.1 *Let L be an upper supporting line of $S_L \cup S_R$ and let p, q be points in $S_L \cup S_R$. If L contains only points in S_L and $slp(p, q) \geq slp(L)$, then p is not on the upper bridge for (S_L, S_R). If L contains only points in S_R, and $slp(p, q) \leq slp(L)$, then q is not on the upper bridge for (S_L, S_R).*

PROOF. Let $[p_0, q_0]$ be the upper bridge for (S_L, S_R), with $p_0 \in S_L$, $q_0 \in S_R$.

If the upper supporting line L contains only points in S_L, then we must have $slp(L) > slp(p_0, q_0)$ (intuitively, this can be seen as follows: we can rotate the line L clockwise around the convex hull $\mathrm{CH}(S_L)$ of S_L until the line touches a point in S_R). Thus, if $slp(p, q) \geq slp(L)$, then $slp(p, q) > slp(p_0, q_0)$, and the point p on the upper bridge $[p_0, q_0]$ would make the point q to stay above the line containing the upper bridge $[p_0, q_0]$, contradicting the definition of the upper bridge $[p_0, q_0]$ since $q \in S_L \cup S_R$. This contradiction shows that the point p cannot be on the upper bridge in this case.

Similarly, if the upper supporting line L contains only points in S_R, then $slp(L) < slp(p_0, q_0)$. If $slp(p, q) \leq slp(L) < slp(p_0, q_0)$, then the point q on the upper bridge $[p_0, q_0]$ would imply that the point p is above the line containing the upper bridge $[p_0, q_0]$, deriving a contradiction and showing

that in this case the point q cannot be on the upper bridge $[p_0, q_0]$. □

Lemma 6.1.1 suggests the following algorithm, based on the Prune-and-Search method, to solve the UPPER-BRIDGE problem.

```
Algorithm UpperBridge(S, Lv)
Input: point set S and vertical line Lv that separates S into S_L and S_R
Output: the upper bridge for (S_L, S_R)
1. If (the set S contains no more than 4 points)
     construct the upper bridge by brute-force method; return;
2. arbitrarily pair up the points of S to make k = n/2 segments:
     [p_1, q_1], [p_2, q_2], ..., [p_k, q_k];
3. find a segment [p_h, q_h] with a median slope s_h over all segments;
4. construct an upper supporting line L with the slope s_h;
5. If (L contains a point p in S_L and a point q in S_R)
     return the upper bridge [p, q]; Stop;
6. If (L contains only points in S_L)
     For (each segment [p_i, q_i] with slope >= s_h)
        delete the point p_i from the set S;
7. Else  /* L contains only points in S_R */
     For (each segment [p_i, q_i] with slope <= s_h)
        delete the point q_i from the set S;
8. recursively call UpperBridge(S, Lv).
```

In the case of step 5, where the upper supporting line `L` contains a point `p` in `S_L` and a point `q` in `S_R`, the segment `[p,q]` obviously makes an upper bridge for (`S_L,S_R`). Thus, the algorithm successfully constructs an upper bridge for (`S_L,S_R`) and stops. Moreover, by Lemma 6.1.1, no point in the set `S` that are on the upper bridge for (`S_L,S_R`) is deleted by steps 6-7. In particular, the sets `S_L` and `S_R` remain nonempty in the recursive call in step 8. Thus, the upper bridge constructed in the recursive call in step 8 for the new set `S`, which is the original input point set with certain points deleted, should be the upper bridge of the original input point set `S`. This verifies the correctness of the algorithm `UpperBridge`.

Now consider the time complexity of the algorithm. Step 1 of the algorithm obviously takes constant time, and step 2 of the algorithm takes time $O(n)$. Note that the point pairing in step 2 is arbitrary. In particular, *it does not require to have one point in* `S_L` *and one point in* `S_R`. Step 3 of the algorithm can be done in time $O(n)$ using the Median-Finding algorithm [5], as described at the beginning of this chapter. To construct the upper supporting line `L` with the given slope `s_h` in step 4, we draw a line with slope `s_h` through each of the points in `S`, and take the one that intersects the y-axis at the highest point. This can be done in time $O(n)$. Steps 5-7 also take time $O(n)$. Thus, besides the recursive call in step 8, the algorithm `UpperBridge` takes time $O(n)$.

Now we study the size of the new set `S` in the recursive call in step 8. There are $n/2$ segments constructed in step 2 of the algorithm. Since `s_h` is the median slope for the segments, there are at least $n/4$ segments whose slope is not smaller than `s_h` and at least $n/4$ segments whose slope is not larger than `s_h`. Thus, no matter which case holds true in steps 6 and 7, at least $n/4$ points are deleted from the set `S`. This shows that the new set `S` in step 8 has at most $3n/4$ points. This gives the following recurrence relationship for the time complexity $T(n)$ of the algorithm `UpperBridge`:

$$T(n) = O(n) + T(3n/4), \quad \text{and} \quad T(c) = O(1) \text{ for } c \leq 4.$$

As we have seen at the beginning of this chapter, it gives $T(n) = O(n)$. Therefore, the algorithm `UpperBridge` runs in linear time.

With the algorithm `UpperBridge`, now we are able to develop an efficient algorithm for the UPPER-HULL problem. Recall (see section 5.1) that an instance of the UPPER HULL problem is of the form (S, p, q), where S is a point set, p and q are the points in S that have the smallest and the largest x-coordinates, respectively, and the line containing p and q is a lower supporting line of the set S. The objective of the instance (S, p, q) for UPPER-HULL is to construct the convex hull CH(S) for the set S. Note that the segment $[p, q]$ is obviously an edge of the convex hull CH(S). The algorithm for UPPER-HULL is given as follows.

```
Algorithm UpperHull(S, p_l, p_r)
Input: a point set S with a lower supporting line through p_l and p_r
Output: the upper hull for S
1. If (the upper hull of S has no more than 3 vertices)
     construct the upper hull by Jarvis-March; return;
2. construct a vertical line Lv that separates S into two subsets S_L
   and S_R of equal size;
3. [p_l', p_r'] = UpperBridge(S, Lv);
4. S_L' = the set of points in S_L that are above the line [p_l,p_l'];
   S_R' = the set of points in S_R that are above the line [p_r',p_r];
5. h_l = UpperHull(S_L', p_l, p_l');
   h_r = UpperHull(S_R', p_r', p_r);
6. return the concatenation of h_l, [p_l', p_r'], and h_r.
```

The correctness of the algorithm `UpperHull` is obvious: by the definition, the upper bridge `[p_1',p_r']` for (`S_L`, `S_R`) is an edge of the convex hull CH(`S`) of the point set `S`, and the concatenation of the upper hull `h_l` for the set `S_L'` of points between `p_l` and `p_l'`, the upper bridge `[p_1',p_r']` for (`S_L`, `S_R`), and the upper hull `h_r` for the set `S_R'` of points between `p_r'` and `p_r` certainly gives the upper hull for the set `S`.

To study the complexity of the algorithm `UpperHull` and its sensitivity to the output size, we use a two-parameter function $T(n, k)$ for the running

time of the algorithm `UpperHull` on an input set of n points whose convex hull has k vertices. Let `S` be such a point set, i.e., `S` contains n points and the convex hull CH(`S`) has k vertices. On the input (`S`, `p_l`, `p_r`), where `p_l` and `p_r` are the left-most and right-most points in `S`, respectively, that are on a lower supporting line for `S`, step 1 of the algorithm takes time $O(n)$, which is implemented by a procedure that applies `Jarvis-March`, starting from the point `p_r`, to construct the first two edges of the convex hull CH(`S`). Step 2 of the algorithm takes time $O(n)$ by the Median-Finding algorithm. Step 3 takes time $O(n)$, by our analysis of the algorithm `UpperBridge`. Step 4 and step 6 can be easily done in time $O(n)$. Thus, besides the recursive calls in step 5, the algorithm `UpperHull` takes time $O(n)$. Now suppose that the upper hull `h_l` for the set `S_L'` has k_l vertices and that the upper hull `h_r` for the set `S_R'` has k_r vertices. Then $k = k_l + k_r$, where k is the number of vertices of the upper hull for the set `S`. Moreover, let n_l be the number of points in the set `S_L'` and let n_r be the number of points in the set `S_R'`. Since `S_L'` is a subset of `S_L` and `S_R'` is a subset of `S_R` while each of the sets `S_L` and `S_R` contains $n/2$ points, we have $n_l \leq n/2$ and $n_r \leq n/2$. Summarizing all these together, we get

$$T(n,k) = T(n_l,k_l) + T(n_r,k_r) + O(n) \leq T(n/2,k_l) + T(n/2,k_r) + c_1 n,$$

with an initial condition $T(n,k) \leq c_2 n$ for $k \leq 3$, where both c_1 and c_2 are constants. We prove by induction on k that $T(n,k) = O(n \log k)$.

For $k \leq 3$, $T(n,k) = O(n \log k)$ holds true because of step 1 of the algorithm `UpperHull`, which takes time bounded by $c_2 n$.

Now assume inductively $T(n,k) \leq c_3 n \log k$, where $c_3 = \max\{c_1, c_2\}$. From the above recurrence relation and the inductive hypothesis, we have

$$\begin{aligned} T(n,k) &\leq T(n/2,k_l) + T(n/2,k_r) + c_1 n \\ &\leq c_3 n \log k_l/2 + c_3 n \log k_r/2 + c_1 n \\ &= (c_3 n/2)\log(k_l k_r) + c_1 n. \end{aligned}$$

Since $k_l + k_r = k$, the function $\log(k_l k_r)$ achieves its maximum value at $k_l = k_r = k/2$. Therefore, $T(n,k) \leq (c_3 n/2)\log(k^2/4) + c_1 n = c_3 n \log k$, and the induction goes through that shows $T(n) = O(n \log k)$.

The final `Kirkpatrick-Seidel` algorithm for constructing the convex hull of a general point set S is given as follows.

```
Algorithm Kirkpatrick-Seidel(S)
Input: a point set S of n points in the plane
Output: the convex hull for S
1. p_l = the point in S with the smallest x-coordinate;
```

```
      p_r = the point in S with the largest x-coordinate;
      L = the line through the points p_l and p_r;
   2. S_u = the set of points in S that are on or above the line L;
      S_l = the set of points in S that are below the line L;
   3. h_u = UpperHull(S_u, p_l, p_r);
      h_l = LowerHull(S_l, p_l, p_r);
   4. return the concatenation of h_u and h_l.
```

Steps 1, 2 and 4 of the algorithm `Kirkpatrick-Seidel` obviously take time $O(n)$. The call to the algorithm `UpperHull` in step 3 takes time $O(n \log k)$, as discussed above. The algorithm `LowerHull`, which also takes time $O(n \log k)$, is completely symmetric to the algorithm `UpperHull`, so we omit its detailed description. In summary, the algorithm `Kirkpatrick-Seidel`, which constructs the convex hull of a set S of n points with the convex hull $\mathrm{CH}(S)$ having k vertices, runs in time $O(n \log k)$.

We give some remarks on the algorithm `Kirkpatrick-Seidel`. In terms of asymptotic analysis, `Kirkpatrick-Seidel` of time complexity $O(n \log k)$ is always better than `Jarvis-March` that has time complexity $O(kn)$. Since $k \leq n$ for all cases, `Kirkpatrick-Seidel` will never be worse than `Graham-Scan` of time complexity $O(n \log n)$, and will become better than `Graham-Scan` when k is much smaller than n (this is a very common case in practice). On the other hand, since `Kirkpatrick-Seidel` uses the linear-time `Median-Finding` algorithm in several places, it has a fairly large constant hidden in the big-O in its complexity. Practically, `Graham-Scan` or even `Jarvis-March` work more efficiently in most cases.

Finally, we briefly discuss the difference and relationships in the algorithms `MergeHull`, `QuickHull`, and `Kirkpatrick-Seidel`. The algorithm `Kirkpatrick-Seidel` takes the advantage of both `MergeHull` and `QuickHull`. It divides the given input 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 have to be considered. In `QuickHull`, the point with a median x-coordinate in the given input point set S may unfortunately be not a hull vertex, therefore the algorithm would not work if we simply replace the furthest point in the algorithm by the median point.

6.1.2 Chan's algorithm

Another $O(n \log k)$-time algorithm for convex hulls is due to Chan [8], which can also be regarded as an application of the Prune-and-Search technique, but avoiding the linear-time but practically expensive `Median-Finding` algo-

rithm. The idea is to first partition the input points into groups of smaller size, then construct the convex hull for each of the groups (using `Graham Scan`), and finally construct the convex hull for the vertices of the convex hulls for the groups (using `Jarvis March`).

This approach has the following advantages: (1) practically, constructing the convex hulls for the groups can significantly reduce the number of points in the construction of the final convex hull for the original input point set, and (2) algorithmically, vertices of a convex hull are well organized so we can handle them more efficiently.

Formally, for a set S of points and a point p that is not in the interior of $\mathrm{CH}(S)$, we say that a point $p' \neq p$ in S is a *rightmost* point in S from p if when we traverse the line containing p and p' in the direction from p to p', no points of S are in the right side of the line. A rightmost point in S from a point p is obviously a vertex on the convex hull $\mathrm{CH}(S)$. In particular, `Jarvis March` is a process that repeatedly finds the rightmost point for the current hull vertex until the convex hull of the set is closed. Note that under the assumption that no three points in S are co-linear, the point p may have up to two rightmost points in S.

We discuss how to find rightmost vertices in a convex polygon P, which is given as a (circular) list $P[0\,..\,h-1]$, in the order when we traverse the polygon in the clockwise order (i.e., we only make right turns when we traverse the polygon. Note that this is consistent with the DCEL representation of the polygon P). We first consider how to decide if a chain $P[i\,..\,j]$ of the convex polygon P contains a rightmost vertex of P for a point p. The index operations given in the algorithm below are modulo h. Thus, $(h-1)+1 = 0$ and $0-1 = h-1$. We use $\mathrm{line}\langle p, q\rangle$ to denote the directed line containing points p and q, traversing in the direction from p to q. Note that to decide in which side of $\mathrm{line}\langle p, q\rangle$ a point p' is, we can compute the signed area of the triangle $\Delta(p, q, p')$ (see Chapter 3).

// decide if a chain $P[i\,..\,j]$ contains a rightmost vertex from a point p not in P

Case 1. neither of $P[i-1]$ and $P[i+1]$ is on the right of $\mathrm{line}\langle p, P[i]\rangle$:
  return("yes") // $P[i]$ is a rightmost vertex;

Case 2. $P[i-1]$ is on the right of $\mathrm{line}\langle p, P[i]\rangle$
  If $P[j]$ is on the right of $\mathrm{line}\langle p, P[i]\rangle$ and $P[j+1]$ is not on the right of $\mathrm{line}\langle p, P[j]\rangle$
  Then return("yes") Else return("no");

Case 3. $P[i+1]$ is on the right side of $\mathrm{line}\langle p, P[i]\rangle$
  If $P[j]$ is on right side of $\mathrm{line}\langle p, P[i]\rangle$ and $P[j+1]$ is on right side of $\mathrm{line}\langle p, P[j]\rangle$
  Then return("no") Else return("yes");

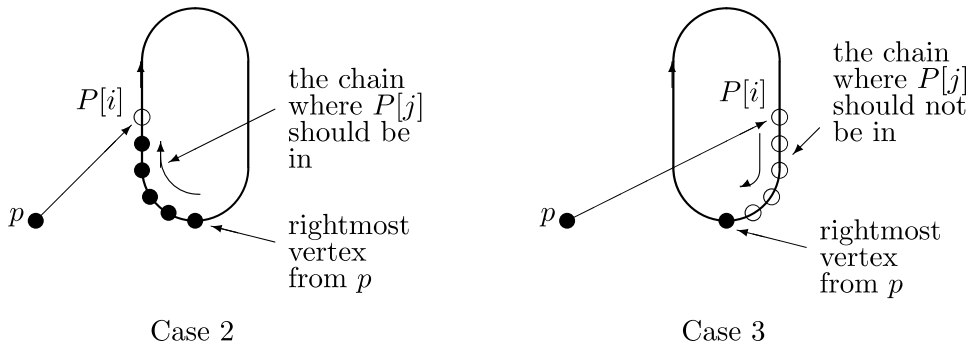

Figure 6.1: Deciding if a chain contains a rightmost vertex from a point p

The correctness of Case 1 in the algorithm is because of the convexity of the polygon P, and the correctness of steps 2-3 of the algorithm can be easily revified based on the figures given in Figure 6.1. As a result, we can decide in constant time if a given chain of vertices in the convex polygon P contains a rightmost vertex of P from the point p.

Now, we can use the technique of binary search to find a rightmost vertex on a convex polygon P from a point p not in the interior of P: we start with the chain $P[0\,..\,h-1]$ that contains all the vertices of the polygon P, which surely contains the rightmost vertices of P. In general, suppose that we are working on a chain $P[i\,..\,j]$ of the polygon P that contains rightmost vertices of P. We first find the "median" vertex $P[m]$ of the chain $P[i\,..\,j]$, where $m = \lfloor i + j \rfloor$. If $P[m]$ is a rightmost vertex from p, then we are done (as shown in Case 1 of the above algorithm, this can be decided in constant time). Otherwise, we look at the two subchains $P[i\,..\,m-1]$ and $P[m+1\,..\,j]$ and recursively work on the one that contains a rightmost vertex of P from the point p. This binary search algorithm obviously runs in time $O(\log h)$, where h is the size of the polygon P.

Now we are ready to present Chan's algorithm for the convex hull problem. First, we assume that the size of the convex hull is bounded by k. This constraint will be removed later. The algorithm is given as follows.

```
Algorithm Chan(S, k)
Input: a point set S and an upper bound k on the size of CH(S)
Output: the convex hull for S

1. divide S into (n/k) groups of k points: S_1, ..., S_{n/k};
2. for each 1 <= h <= n/k, construct the convex hull CH(S_h);
3. let p_0 be the lowest point in S, and let p_1 be the rightmost point
   in S from p_0;
5. h = 1;
6. while (p_h != p_0) do
6.1  for (each CH(S_i)) find the rightmost vertex of CH(S_i) from p_h;
```

```
6.2  let p_{h+1} be the rightmost point from p_h in those found in 6.1;
6.3  h = h+1.
```

The complexity analysis of the algorithm `Chan` is simple. Step 1 of the algorithm takes time $O(n)$. In step 2, we use `Graham Scan` to construct the convex hull $\mathrm{CH}(S_i)$ for each group S_i in time $O(k \log k)$. Thus, step 2 takes time $(n/k) \cdot O(k \log k) = O(n \log k)$. In step 3, to find the lowest point p_0 in S takes time $O(n)$, from which we can find the rightmost point p_1 in S from p_0 in time $O(n)$. The key difference of `Chan`'s algorithm from `Jarvis March` is step 6.1, where we find the rightmost vertex in the convex hull $\mathrm{CH}(S_i)$ for each i in time $O(\log k)$. Thus, step 6.1 in total takes time $(n/k) \cdot O(\log k)$. Since step 6.2 can obviously be done in time $O(n/k)$ and the while-loop of step 6 runs at most k times, the total running time of step 6 of the algorithm is bounded by $k \cdot (n/k) \cdot O(\log k) = O(n \log k)$. In conclusion, the algorithm `Chan` runs in time $O(n \log k)$.

Recall that we actually do not know in advance a good upper bound k for the size of the convex hull $\mathrm{CH}(S)$. To overcome this difficulty, we try all values of form $k = k_t = 2^{2^t}$, for $t = 1, 2, \ldots$, until the first t such that $k_t = 2^{2^t}$ is not smaller than the size of $\mathrm{CH}(S)$. Note that for values k_t that are smaller than the size of the convex hull $\mathrm{CH}(S)$, the algorithm will fail in closing the convex hull $\mathrm{CH}(S)$ in step 6 (more precisely, step 6 of the algorithm should be changed to "while $(h \leq k)$ and $(p_h \neq p_0)$" and when $h > k$ but $p_h \neq p_0$, the algorithm will report a failure. This repeated trial will give an algorithm that constructs the convex hull $\mathrm{CH}(S)$ of the given set S of points, without needing to know in advance an upper bound on the size of $\mathrm{CH}(S)$. Moreover, the running time of the algorithm is bounded by

$$O(n \log k_1) + O(n \log k_2) + \cdots + O(n \log k_t) = O(n \log(k_1 \cdot k_2 \cdot \cdots \cdot k_t),$$

where, if we let k be the size of the convex hull $\mathrm{CH}(S)$, then $k_t = 2^{2^t} \geq k$ and $k_{t-1} = 2^{2^{t-1}} < k$. Thus,

$$k_1 \cdot k_2 \cdot \cdots \cdot k_t = 2^{2^1} 2^{2^2} \cdots 2^{2^t} = 2^{2^1 + 2^2 + \cdots + 2^t} \leq 2^{2^{t+1}} = (2^{2^{t-1}})^4 \leq k^4.$$

In conclusion, `Chan`'s algorithm runs in time $O(n \log(k_1 \cdot k_2 \cdot \cdots \cdot k_t)) = O(n \log k^4) = O(n \log k)$.

We note that `Chan`'s algorithm can be extended to the convex hull problem in 3-dimensional space without principle difficulties.

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 a large class of 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 that contains the point. A one-time query of this type is called a *single shot*. On the other hand, 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 more 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 be 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 be less than $O(n)$ since even storing the unorganized structure, the PSLG G itself needs $\Omega(n)$ space. Finally, it has been well-known [15] that any algorithm for searching in a set S of n elements by means of comparisons, no matter how the set S is organized, can be represented as a binary tree of n leaves. Thus, in the worst case, the searching time of the algorithm is at least $\Omega(\log n)$. Since 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 start with a simple example for the point location problem, where the PSLG is a convex polygon P whose vertices are given in counterclockwise ordering $\{v_1, v_2, \cdots, v_n\}$. We first organize P by the following algorithm, where the slop of a ray is the angle value between 0 and 2π.

```
Algorithm PreProcessing(P)
Input: a convex polygon P
```

```
Output: an organized structure L for P
1. find an interior point p_0 in P;
2. For (each vertex v of P)
      let r be the ray starting from p_0 and containing v;
      attach the vertex v to the ray r;
3. sort the rays in step 2 by their slopes: L = {r_1, r_2, ..., r_n};
4. return the list L and the point p_0.
```

With the list `L`, we can locate query points by the following algorithm.

```
Algorithm Query(q)
/* assume the list L and the point p_0 above for P are given */
Input: a query point q
Output: determine if the point q is inside the convex polygon P
1. compute the slope of the ray r starting at p_0 and containing q;
2. use binary search on the list L to find the ray r_i such that
   the wedge bounded by r_i and r_(i+1) contains the point q;
3. the point q is inside the convex polygon P if and only if q and
   p_0 are on the same side of the line through [v_i, v_(i+1)].
```

The correctness of the algorithm `Query(q)` is obvious. If the point `q` is in the convex polygon `P`, it must be contained in one of the triangles formed by three points `p_0`, `r_i`, and `r_(i+1)` for some `i`, which is equivalent to that the point `q` is contained in the wedge bounded by `r_i` and `r_(i+1)`, and that the points `q` and `p_0` are on the same side of the line through the points `r_i`, and `r_(i+1)`, as checked by the algorithm.

Now we analyze the complexity of the algorithms.

Preprocessing time. The preprocessing is implemented by the algorithm `PreProcessing`. The interior point `p_0` in the convex polygon `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 for each vertex `v` of `P`, in constant time the equation of the ray starting at `p_0` and passing through `v` can be constructed and the slope of the ray can be computed. To consider step 3, note that because the polygon `P` is convex, the rays constructed in step 2 are cyclically sorted by their slopes if we construct the rays following the order of traversing the boundary of the polygon `P`. Thus, it is easy to construct the sorted list `L` in step 3 in time $O(n)$. In summary, the preprocessing phase for the problem takes time $O(n)$.

Storage. We need to store the list `L` and the point `p_0`. Each element in the list `L` consists of a vertex `v_i` of the polygon `P` and the slope of the ray that starts from the point `p_0` and passes through the vertex `v_i`, which obviously takes space $O(1)$. Thus, the storage requirement for the algorithm is $O(n)$.

Query time. Locating a query point is implemented by the algorithm `Query(q)`, which takes time $O(\log n)$ because of the binary search in step 2.

The following theorem summarizes the above discussion.

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. Each region between two consecutive segments in a slab uniquely belongs to a region in the original PSLG G, which will be called a "slab region." Thus, if we construct a list for the edge segments in each slab, ordered from left to right, then for a point q in the slab, we can use binary search to easily find the slab region that contains the point q. This gives the algorithm shown below, where `L` is a list of the slabs, sorted by y-coordinates (that is, any point in slab `L[i]` has a larger y-coordinate than that of a point in slab `L[k]` for all `i > k`). Each element `L[i]` in the list `L` is also a list that is for the edge segments in the corresponding slab, sorted from left to right. For each slab region in `L[i]`, we attach the name of the corresponding region in the original PSLG.

```
Algorithm Locating(q)
Input: a query point q
Output: the region of the PSLG that contains the point q
/* the PSLG is given by a list L of slabs, and each slab L[i] is a  */
/* list of the edge segments in the slab ordered from left to right */
1. use the y-coordinate of the point q and perform binary search in
   the list L to find the slab L[i] that contains the point q;
2. use the x-coordinate of the point q and perform binary search in
   the list L[i] to find the slab region that contains the point q;
3. return the name of the region in the original PSLG.
```

Suppose that the PSLG G has n vertices. The binary search in step 1 of the algorithm `Locating(q)` takes time $O(\log n)$. 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, and the binary search in step 2 of the algorithm can also be done in time $O(\log n)$. Since we have attached the name of the corresponding region in the original PSLG G to each slab region, once we find the slab region that contains the point

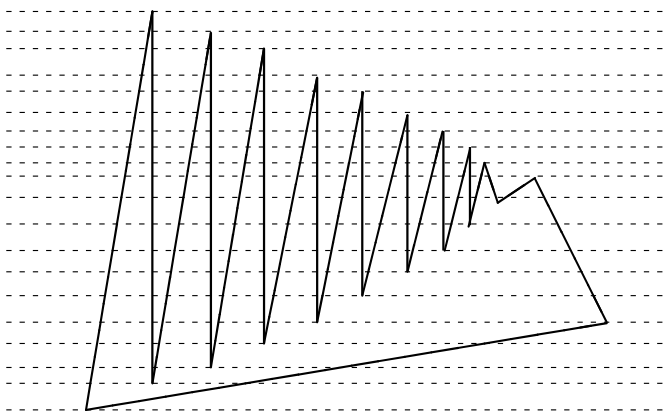

Figure 6.2: A PSLG containing $\Omega(n^2)$ edge segments

`q`, we can directly get the name of the region in G that contains the point `q`. We conclude that the query time of this slab method is $O(\log n)$.

Now we consider 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 the $n+1$ lists `L[i]` is bounded by $O(n^2)$. This storage cannot be improved since some PSLGs do have the structure such that there are $\Omega(n^2)$ edge segments in the slabs. Figure 6.2 gives an example of such a PSLG.

A straightforward method to produce these lists is to first sort the vertices of G by y-coordinates to get the sorted list `L` of the slabs, then for each slab, sort the edge segments in the slab to get the sorted list `L[i]`. By this approach, the time complexity to obtain the sorted list `L` is $O(n \log n)$, and the time complexity to obtain the $n+1$ sorted lists `L[i]` for `i` $= 1, 2, \ldots, n+1$ 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 store all the edge segments of a slab in a 2-3 tree, sorted by the order from left toright in the slab. 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 then represents exactly the list of the edge segments, ordered from left to right, of the next slab. We print the leaves of the 2-3 tree for each slab, from left to right, and obtain the list `L[i]` for `i` $= 1, 2, \ldots, n+1$. 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 PreProcessing(G)
Input: a PSLG G of n vertices, given by a DCEL
Output: the lists L and L[i] for i = 1, 2, ..., n-1
1. sort the vertices of G by increasing y-coordinates:
               L = {v_1, v_2, ..., v_n};
   /* slab L[i] is given by lines y = v_i.y and y = v_(i+1).y  */
2. T = an empty 2-3 tree;
3. For (i = 1; i < n; i++)
3.1   delete all lower edges of vertex v_i from T;
3.2   insert all upper edges of vertex v_i into T;
3.3   L[i] = the leaves of T from left to right;
3.4   output(L[i]);
4. output(L)
```

It is easy to see that the algorithm `PreProcessing` is correct. We analyze the complexity of the algorithm. Step 1 takes time $O(n \log n)$ by using any optimal sorting algorithm. Consider the loop of step 3. Since each slab contains at most $O(n)$ edge segments, the 2-3 tree `T` has $O(n)$ leaves. As a result, the depth of the 2-3 tree `T` is bounded by $O(\log n)$. Therefore, each of the edge insertions and edge deletions in steps 3.1-3.2 takes time $O(\log n)$. Each edge of the PSLG G is inserted by step 3.1 exactly once into the 2-3 tree `T` then is deleted by step 3.2 exactly once from the 2-3 tree `T`. Moreover, given the vertex `v_i`, all the edges of G incident to `v_i` can be found by an algorithm called `Trace-Vertex`, which is similar to the algorithm `Trace-Region` in section 2.4, in time proportional to the number of these edges (note that the PSLG G is given by a DCEL). Thus, each of the lower edges and upper edges of `v_i` in step 3 can be found in constant time. To summarize these together, we conclude that the time of insertion and deletion of an edge of the PSLG G is bounded by $O(\log n)$ for the entire algorithm. Consequently, the total time of the algorithm taken by steps 3.1-3.2 is bounded by $O(n \log n)$ since G contains $O(n)$ edges. Now to read the leaves of the 2-3 tree `T` from left to right in step 3.3, we can use, say, Depth First Search on the tree `T`. (For a discussion on Depth First Search, see [2].) The time to search the tree `T` and then to produce the list `L[i]` in step 3.3 thus is bounded by a constant times the number of nodes in the tree `T`, which is bounded by $O(n)$. Therefore, the total time taken by steps 3.3-3.4 of the algorithm is bounded by $O(n^2)$. Note that this bound cannot be further improved, as we have seen that some PSLG contain $\Omega(n^2)$ many edge segments. As a result, the total time spent on step 3 is $O(n \log n + n^2) = O(n^2)$. This concludes that the running time of the algorithm `PreProcessing` on a PSLG of n vertices is bounded by $O(n^2)$.

Theorem 6.2.2 *The slab method solves the point location problem, with*

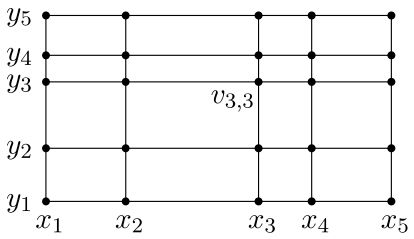

Figure 6.3: A 5×5 rectangle with the center vertex $v_{3,3}$

preprocessing time $O(n^2)$*, storage* $O(n^2)$*, and query time* $O(\log n)$*.*

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 $v_{i,j} = (x_i, y_j)$, $i, j = 1, 2, \ldots, m$. 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 this PSLG an $m \times m$ *rectangle* with the index sets X and Y, in which each region is a rectangle. Figure 6.3 pictures a 5×5 rectangle.

Clearly, the point location problem on a PSLG R_m that is an $m \times m$ rectangle can be simply done by doing two binary searches, one on the list X and the other on the list Y. Alternatively, we can locate a query point $q = (x', y')$ in the PSLG R_m as follows. Define the *center vertex* of the $m \times m$ rectangle R_m by $v_{m/2,m/2} = (x_{m/2,m/2}, y_{m/2,m/2})$. By comparing the x-coordinate x' of the point q with the x-coordinate of the center vertex $v_{m/2,m/2}$, and the y-coordinate y' of the point q with the y-coordinate of the center vertex $v_{m/2,m/2}$, we can uniquely find a $(m/2) \times (m/2)$ rectangle $R_{m/2}$ in R_m that contains the point q. Then, we recursively work on the $(m/2) \times (m/2)$ rectangle, until the rectangle becomes a single region in G.

Formally, let R_m be an $m \times m$ rectangle with the index sets $X =$

$(x_{s+1}, \cdots, x_{s+m})$ and $Y = (y_{t+1}, \cdots, y_{t+m})$. A vertex $v_{i,j} = (x_i, y_j)$ is the *center vertex* of R_m if $i = s + \lceil m/2 \rceil$ and $j = t + \lceil m/2 \rceil$. We organize the $m \times m$ rectangle R_m as a tree T_m, as follows. The root N_m of T_m is labeled with the center vertex of R_m, and has four children, corresponding to the four $(m/2) \times (m/2)$ rectangles of R_m obtained by dividing R_m by the horizontal and vertical lines passing through the center vertex. Each child of N_m is recursively constructed using the corresponding $(m/2) \times (m/2)$ rectangle. The algorithm of constructing this tree is formally presented as follows:

```
Algorithm Quad-tree(G)
Input: an m by m rectangle R_m
Output: the quadtree T_m for R_m
1. If (R_m is a single region, i.e., a 2 by 2 rectangle)
     return a node for R_m labeled with the region name;
2. create the root N_m for T_m, labeled with the center vertex of R_m;
   the center vertex divides R_m into four (m/2) by (m/2) rectangles:
   R_(m/2)-1, R_(m/2)-2, R_(m/2)-3, R_(m/2)-4,
3. r_1 = Quad-tree(R_(m/2)-1);  r_2 = Quad-tree(R_(m/2)-2);
   r_3 = Quad-tree(R_(m/2)-3);  r_4 = Quad-tree(R_(m/2)-4),
4. let the four children of N_m be r_1, r_2, r_3, r_4;
5. return(N_m)
```

The tree `T_m` constructed by the algorithm `Quad-tree` is called a *quadtree* in the literature [26] because each of its node either is a leaf or has exactly four children. Each leaf in the tree `T_m` corresponds uniquely to a region of the rectangle `R_m`, and each internal node of `T_m` is labeled with a vertex in `R_m` that is the center vertex for a $k \times k$ rectangle in `R_m` for some integer k. Since there are $n = m^2$ vertices and $O(n)$ regions in the $m \times m$ 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 $O(n)$.

The algorithm `Quad-tree` gives the preprocessing for the point location problem on $m \times m$ rectangles, which takes time $O(n)$ and space $O(n)$.

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 $O(\log n)$. For a given query point q, using the quadtree `T_m`, we can easily locate the point q in a region in the corresponding $m \times m$ rectangle `R_m`, as shown by the following algorithm.

```
Algorithm Locating(q, r_m)
Input: query point q and the root r_m of the quadtree T_m for R_m
Output: the region of R_m that contains q
1. If (r_m is a leaf node)
     return the region name labeling the leaf r_m;
2. compare q with the center vertex labeling r_m to find the
   child r_(m/2) of r_m that contains q;
3. return(Locating(q, r_(m/2)).
```

By the discussion above, the quadtree `T_m` for an $m \times m$ rectangle of $n = m^2$ vertices has depth $O(\log n)$. Since the algorithm `Locating` spends constant time on each node in its traversing from the root to a leaf of the quadtre tree `T_m`, the algorithm `Locating` runs in time $O(\log n)$ for each query point q. This concludes the following theorem.

Theorem 6.2.3 *The point location problem on $m \times m$ rectangles of $n = m^2$ vertices can be solved with preprocessing time $O(n)$, space $O(n)$, and query time $O(\log n)$.*

Let us summarize the above idea. Our search process can be depicted as a search tree $\mathcal{T}$ in which each node is associated with a geometric structure where we try to locate a given query point q. The search process starts with a large given rectangle R that contains the query point q. It divide the rectangle R into four smaller rectangles using the center vertex of R, and "refines" the search process by recursively working on one R' of the smaller rectangles that contains the query point q. Thus, the smaller rectangle R' becomes a child of the larger rectangle R in the search tree $\mathcal{T}$. Three properties of the search tree $\mathcal{T}$ that we have heavily used are:

- a parent node and its child nodes in the search process have the same geometric structure (here are rectangles), so the recursion is effective;
- each parent node has only a small number of child nodes so that from a parent node, the search process can quickly move to the "correct" child node of a more refined structure;
- the refinement speed from a parent node to a child node is sufficiently fast so that the search process only needs to examine a few (i.e., $O(\log n)$) nodes before it reaches a desired node.

6.2.4 Refinement method II: on general PSLGs

Now we apply the idea in the last subsection and develop algorithms for the point location problem on general PSLGs. The algorithm presented in this subsection is due to Kirkpatrick [13].

All 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

type. Therefore, we must first fix a geometric structure that we are going to use. It is natural to consider the simplest geometric shape, i.e., triangles. However, not every PSLG can be obtained by refining a triangle. Extra care should be taken to make the 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 $l_h : y = y_0$, the vertical line $l_v : x = x_0$, and the line $l : x + y = z_0$ will enclose the entire PSLG G. Let the PSLG consisting of G and the big triangle be G'. Now triangulating G' gives us a completely triangulated PSLG G_0. We can simply attach the name of a region of G to the triangles of G_0 that are contained in the region. Thus, once we locate the query point in a triangle in the PSLG G_0, we can easily decide the region of G that contains the query point.

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 structure as G_0 that is a completed triangulated PSLG, 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 disappeared when we deleted the vertices of G_0 (call these regions *old triangles*). All regions of G_1 are regions of G_0 except those that were created when we retriangulated 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. Therefore, the old PSLG G_0 can be regarded as a refinement of the new PSLG G_1. Now if the completely triangulated PSLG G_1 is still not simple enough to handle, then we apply the above process on G_1 to obtain another completely triangulated PSLG G_2 such that G_1 is a refinement of G_2. We repeat (recursively) this process to produce a sequence of completely triangulated PSLGs $(G_0, G_1, \ldots, G_k)$, where for each $0 \leq h \leq k - 1$, G_h is a

refinement of G_{h+1}, until we obtain a simple enough PSLG G_k so that we can directly locate the query point q in one of the triangles in G_k.

This solves our first problem: the inverse of the deleting-vertex-then-retriangulating operation "refines" a completely triangulated PSLG G_{h+1} into another completely triangulated PSLG G_h, for each h.

The query algorithm now goes as follows: we start with the simple PSLG G_k in which we can directly locate the given query point q in one of its triangles. Recursively, for each $h \geq 0$, suppose that we have located the query point q in a triangle T in the PSLG G_{h+1}, then we look at all triangles in the PSLG G_h that intersect the triangle T and determine which triangle of G_h contains the query point q. The search process repeats until it locates a triangle for the query point q in the PSLG G_0, which then directly gives the region of the original PSLG G that contains the query point q.

However, in the above process for each h, how many triangles in G_h can intersect the triangle T in G_{h+1}, and how many completely triangulated PSLG's should we go through in order to locate the query point q in a triangle of the PSLG G_0? In order to achieve an $O(\log n)$ query time, we must move from one completely triangulated PSLG G_{h+1} to another completely triangulated PSLG G_h in constant time, and go through at most $O(\log n)$ completely triangulated PSLG's to reach the original completely triangulated PSLG G_0. For this purpose, we require that the vertices to be deleted from the completely triangulated PSLG G_{h+1} in the construction of the completely triangulated PSLG G_h satisfy the following conditions:

1. all these vertices should be interior vertices of the completely triangulated G_h, i.e., they should not be the three hull vertices of G_h;
2. no two of these vertices can be adjacent in G_h;
3. the degree of these vertices should be small; and
4. there should be sufficiently many this kind of vertices in G_h.

The first condition makes it easy to make the PSLG G_{h+1} completely triangulated. The second condition keeps a simple relationship between the new triangles in G_{h+1} and the old triangles in G_h. That is, an old triangle incident to a deleted vertex v in G_h can only intersect those new triangles in G_{h+1} that are obtained by retriangulating the simple polygon resulting from deleting the vertex v from G_h. 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 refinement speed of the completely triangulated

PSLGs is fast enough so that a query point goes through very few completely triangulated PSLGs to reach the original PSLG G_0.

We need to show the existence of such a set of vertices in the completely triangulated PSLG G_{h+1} that satisfies all four conditions above. This is done by a pure combinatorial counting technique.

Let $G = (V, E)$ be a completely triangulated PSLG and let F be the set of regions of G. Since G is a planar embedding of a graph, by Euler's formula (see subsection 2.4.1), we have:

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

Since G is completely triangulated, 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 $2|E|/3$, we obtain

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

For a vertex v of G, let $deg(v)$ be the degree of 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

$$\sum_{v \in V} deg(v) = 2|E| < 6|V|.$$

Therefore, at least a half of the vertices of G have degree less than 12. If we exclude the three hull vertices of G, there are at least $|V|/2 - 3$ interior 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. For $|V| \geq 12$, we have $(|V|/2 - 3)/12 \geq |V|/48$. Therefore, for an arbitrary completely triangulated PSLG G with n vertices, with $n \geq 12$, we can find at least $(n/2 - 3)/12 \geq n/48$ interior vertices of G of which no two are adjacent and all are of degree less than 12. This gives the set of vertices in the PSLG G that satisfies all four conditions listed above.

This analysis suggests the following algorithm to construct a searching hierarchy `H_G` for the point location problem on a general PSLG `G`.

```
Algorithm Search-Hierarchy(G)
Input: a general PSLG G
Output: a searching hierarchy H_G for point location problem on G
```

```
1. add an enclosing triangle that contains G; triangulate the resulting
   PSLG to construct a completely triangulated PSLG G_0;
2. For (each triangle T in G_0)
      create a node N for T at level 0 in the hierarchy H_G;
3. k = 0;
4. While (G_k has at least 12 vertices)
4.1   mark all interior vertices of degree less than 12 in G_k;
4.2   While (there are marked vertices in G_k)
4.2.1   pick a marked vertex v in G_k and unmark all its neighbors;
4.2.2   delete v from G_k, and triangulate the resulting polygon;
4.2.3   For (each new triangle T constructed in step 4.2.2)
           create a node N for T at level k+1 in the hierarchy H_G;
           For (each node N' at level k whose triangle T' intersects T)
              set a pointer from N to N' in H_G;
4.3   For (each node N' for a triangle T' at level k in H_G)
        If (T' does not contain a vertex deleted in step 4.2.2)
           create a node N for T' at level k+1 in H_G;
           set a pointer from N to N' in H_G;
4.4   k = k + 1;
      G_k = the PSLG consisting of the triangles at level k in H_G;
```

We analyze the algorithm `Search-Hierarchy(G)` for constructing the searching hierarchy `H_G`. Suppose that the number of vertices of the input PSLG `G` is n, and that each completely triangulated PSLG `G_k` is represented by a doubly-connected edge list (DCEL) and has n_k vertices, with $n_0 = n+3$. As we have described, it takes time $O(n)$ to construct the enclosing triangle. The triangulation takes time $O(n \log n)$ if the PSLG `G` is a general PSLG, or takes time $O(n)$ if all regions of the PSLG `G` are simple polygons (by the linear-time triangulation algorithm due to Chazelle [9]). Therefore, step 1 of the algorithm takes time $O(n \log n)$ for a general PSLG `G` and takes time $O(n)$ for a PSLG `G` whose regions are all simple polygons.

Step 2 of the algorithm can be implemented by calling the `Trace-Region` algorithm in subsection 2.4.2 on each triangle of `G_0`. As we have seen, the algorithm will generate all triangles in the PSLG `G_0` in time $O(n_0) = O(n)$.

To find all interior vertices of degree less than 12 in the PSLG `G_k` in step 4.1, we call the algorithm `Trace-Vertex` on each vertex of `G_k`. The `Trace-Vertex`(v) algorithm is similar to the algorithm `Trace-Region` but traverses all edges incident to the vertex v in time $O(d(v))$, where $d(v)$ is the degree of the vertex v. Thus, in total time $O(n_k)$ we can count the degree for each vertex in `G_k` and mark those whose degree is less than 12.

Now consider step 4.2. Obviously, the **While**-loop of step 4.2 is executed no more than n_k times. Since each vertex `v` picked in step 4.2.1 has degree less than 12, it is incident to at most 11 triangles at level k in `H_G`, and the number of new triangles at level $k+1$ constructed in step 4.2.2 is bounded by $11 - 2 = 9$. Therefore, there are at most $11 \times 9 = 99$ pairs of triangles that are examined in step 4.2.3, i.e., each execution of step 4.2.3 takes time

$O(1)$. In conclusion, the running time of steps 4.2 is bounded by $O(n_k)$.

Step 4.3 takes time $O(n_k)$ by calling the `Trace-Region` algorithm on each region of `G_k`, and step 4.4 obviously takes time $O(n_k)$. In conclusion, for the PSLG `G_k`, steps 4.1-4.4 of the algorithm takes time $O(n_k)$.

By the discussion above and assuming $n_k \geq 12$, the PSLG `G_k` has at least $(n_k/2) - 3$ interior vertices of degree less than 12, which are marked in step 4.1. Since step 4.2.1 unmarks at most 11 marked vertices and step 4.2.2 deletes the marked vertex `v`, each execution of the steps 4.2.1-4.2.3 "consumes" at most 12 marked vertices in `G_k`. As a resykt, in total at least $(n_k/2 - 3)/12 \geq n_k/48$ vertices of `G_k` are deleted by step 4.2.2 in the execution of the **While**-loop of step 4.2. Therefore, the number n_{k+1} of vertices in the next PSLG `G_(k+1)` at level $k+1$ in the hierarchy `H_G` is bounded by $47n_k/48$. Combining this with the complexity of steps 4.1-4.4 as we discussed above, we conclude that the total running time of step 4 is bounded by

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

if the searching hierarchy T_G has $h+1$ levels, where $n_0 = n + 3$, $n_h < 12$, and $n_{k+1} \leq 47n_k/48$, which implies $n_k \leq (47/48)^k n_0$ for all $0 \leq k \leq h$.

Since

$$\begin{aligned} & n_0 + n_1 + \cdots + n_h \\ \leq\ & n_0 + (47/48)n_0 + \cdots + (47/48)^h n_0 \\ <\ & n_0 + (47/48)n_0 + \cdots + (47/48)^h n_0 + \cdots \\ =\ & \frac{n_0}{1 - (47/48)} \\ =\ & O(n), \end{aligned} \tag{6.1}$$

the total running time for step 4 is bounded by $O(n)$. In conclusion, the algorithm `Search-Hierarchy` runs in time $O(n \log n)$ on a general PSLG of n vertices, while runs in time $O(n)$ on a PSLG of n vertices in which each region is a simple polygon. Since the PSLG `G_k` has n_k vertices, for each k, level k for the searching hierarchy `H_G` consists of $O(n_k)$ triangles. Thus, the bound in (6.1) above also shows that the hierarchy `H_G` has $O(n)$ nodes thus can be stored in space $O(n)$.

For the depth h of the searching hierarchy `H_G`, since n_h is the first index h such that $n_h < 12$, we have $n_{h-1} \geq 12$. Combining this with $(47/48)^{h-1} n_0 \geq n_{h-1}$, we get $h = O(\log n)$. Thus, the depth of the searching hierarchy `H_G` is $O(\log n)$.

The searching algorithm for a query point based on the searching hierarchy H_G is now straightforward.

```
Algorithm Locating(q)
Input: a query point q and the searching hierarchy H_G for PSLG G
Output: the region of G that contains q
1. at the top level of H_G, find a node N whose triangle T contains q;
2. While (N is not at level 0)
      following the pointers from N to find a node N' in the level below
       whose triangle contains q;
      N = N';
3. /* the node N is at level 0 in H_G */
   return the region label on the triangle for the node N.
```

Since the top level of the searching hierarchy H_G has less than 12 vertices, we can decide in constant time in step 1 of the algorithm Locating the node at the top level of H_G whose triangle contains the query point q. Moreover, as we suggested, we label each triangle T in the completely triangulated PSLG G_0 with the name of the region in the PSLG G that contains the triangle T. Thus, once we reach a triangle at level 0 that contains the query point q in step 3 of the algorithm, we can return in constant time the region name of the PSLG G that contains the query point q. Finally, we consider step 2 of the algorithm Locating. As discussed above, a triangle at level k intersects at most 11 triangles at level $k-1$. Therefore, for a node N at level k with $k > 0$, there are at most 11 pointers to level $k-1$. As a result, in constant time, we can decide a node N' at level $k-1$ whose triangle contains the query point q and move to level $k-1$. Since the depth of the searching hierarchy H_G is $O(\log n)$, we conclude that the algorithm Locating(q) finds the region of the PSLG G that contains the query point q in time $O(\log n)$.

We summarize the above results in the following theorems.

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

Theorem 6.2.5 *For a PSLG G of n vertices in which all regions are simple polygons, the point location problem can be solved with $O(n)$ preprocessing time, $O(n)$ space, and $O(\log n)$ query time.*