Chapter 8. Computational Geometry

Computational geometry deals with the algorithmic aspects of geometrical problems. The typical objects of (plane) computational geometry are points, lines and line segments, polygons, planar subdivisions (= straight-line embeddings of planar graphs) and collections of these objects. Typical questions are e.g. intersection (line-polygon, polygon-polygon, ...), point location (point-point, point-polygon, point-planar subdivision) and decomposition problems (decomposition of a polygon into simpler polygons). The questions are motivated by widespread use of geometric objects in computer graphics and computer aided design. In particular, the computational problems arising in two- and three-dimensional computer graphics and in VLSI design shaped the field and generated a common interest in it.

We organize this chapter into six sections: convex polygons, convex hull, Voronoi Diagrams, the sweep paradigm, orthogonal objects, and geometric transformations. In each section we describe the computational tools and paradigms and give the algorithms for the basic problems. Most of the discussion is restricted to two dimensional geometry. However, we sometimes also discuss three- or higher-dimensional space or at least cite relevant references in the section containing bibliographic remarks.

Throughout the chapter we will use the following notation. Let p and q be points in $\mathbb{R}^2$. The **line segment** defined by p and q and denoted $L(p,q)$ is the set of points on the line passing through p and q and lying between p and q. Sometimes we will use $L(p,q)$ to denote the **line** through p and q or even the **oriented line** (in the direction from p to q) through p and q. The ambiguity in the notation will always be removed by the context.

Let L be an oriented line and let p be a point not on line L. Let $\vec{d}$ be the vector defining the orientation of line L and let q be an arbitrary point of line L. Then point p lies to **the right of** oriented line L if vector $\vec{p}-\vec{q}$ can be turned into vector $\vec{d}$ by a (counter-clockwise) rotation of less than 180 degrees, in other words, if the third component of the vector product of $\vec{p}-\vec{q}$ and $\vec{d}$ is positive, i.e., $\vec{p}-\vec{q}=(x,y)$, $\vec{d}=(d_x,d_y)$ and $xd_y - yd_x > 0$ (cf. Fig. 1).

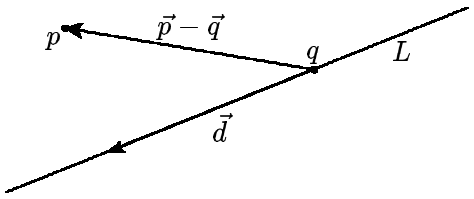


Figure 1. p lies to the right of L

The right (left) halfspace defined by oriented line L is the set of all points on or to the right (left) of line L. We always assume lines to be oriented. If no explicit orientation is defined then the default value is "upwards" (to the "right" in case of a horizontal line).

If L and L' are lines or line segments we use $\bigcap(L, L')$ to denote the intersection of L and L'.

A **polygon** is a sequence $v_0, v_1, \ldots, v_n$ of points. The v_i's are the **vertices** of the polygon. A polygon is **simple** if line segments $L(v_i, v_{i+1})$ and $L(v_j, v_{j+1})$, $0 \leq i < j \leq n$ (indices are taken mod$(n+1)$), intersect only if $j = i+1$ or $\{i, j\} = \{0, 1\}$ and then intersect in their common endpoint.

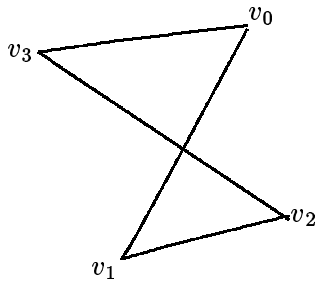


Figure 2. A non-simple polygon

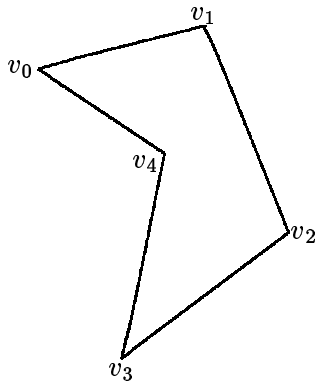


Figure 3. A simple polygon

Let $P = v_0, \ldots, v_n$ be a simple polygon. The removal of the polygonal chain of line segments $L(v_0, v_1), \ldots, L(v_{n-1}, v_n), L(v_n, v_0)$ from the plane divides the plane into two regions. The bounded region is called the **interior** of the polygon, the unbounded region is called the **exterior** of the polygon. We assume that the vertices of a simple polygon are ordered in a way that the interior is to the right as we traverse the sequence $v_0, v_1, \ldots, v_n, v_0$.

A region $R \subseteq \mathbb{R}^2$ is **convex** if for all points $p, q \in R$ the entire line segment $L(p, q)$ is contained in R. A simple polygon is convex if its interior is a convex region. Equivalently, a simple polygon is convex if no interior angle exceeds π. A vertex v of a simple polygon is a **cusp** if the interior angle at v exceeds π. A region is **polygonal** if its boundary is a simple polygon.

Let $S \subseteq \mathbb{R}^2$ be a set. The **convex hull** $CH(S)$ is the intersection of all convex sets containing S, i.e, $CH(S) = \bigcap\{R;\ S \subseteq R \text{ and } R \text{ convex }\}$. Since the

intersection of a family of convex sets is convex, the convex hull $CH(S)$ is a convex set. It is the smallest (with respect to set inclusion) convex set containing S. If S is a finite set then the convex hull of S is a convex polygonal region.

We end this introduction with an important application of binary search and more generally binary search trees. Let L, R be two vertical lines, let $p_1, \ldots, p_n$ be a sequence of points on L and let $q_1, \ldots, q_n$ be a sequence of points on R.

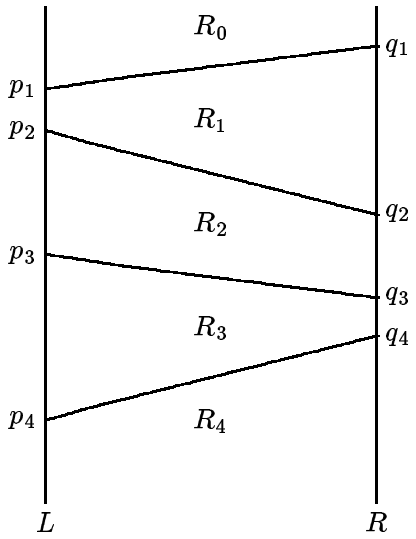


Figure 4. Binary search in a vertical strip

We assume that both sequences are ordered from top to bottom. Let $L_i = L(p_i, q_i)$ be the line segment connecting points p_i and q_i. Then the line segments L_i, $1 \leq i \leq n$, do not intersect (except maybe in points on line L or R) and divide the vertical strip between lines L and R into $n+1$ pairwise disjoint regions $R_0, R_1, \ldots, R_n$. The regions are ordered from top to bottom in a natural way and so are the line segments. We can use this ordering to find the region containing a query point p (lying in the strip between L and R) in time $O(\log n)$ by binary search. For Program 1 we assume that p lies below L_1 and above L_n.

```
(1)   top ← 1; bottom ← n; middle ← ⌊(n + 1)/2⌋;
(2)   while bottom > top + 1
(3)   do if p lies above L_middle
(4)         then bottom ← middle
(5)         else top ← middle
(6)      fi;
(7)      middle ← ⌊(top + bottom)/2⌋
(8)   od;
```

Program 1

On termination we know that p lies in the region between lines L_{top} and L_{bottom} and $bottom = top + 1$.

In line (3) we have to determine whether p lies above L_{middle}. If p_i (q_i, p) has cartesian coordinates (a, py_i) $((b, qy_i), (p_x, p_y))$ then the test in line (3) is equivalent to $p_y \geq py_i + (qy_i - py_i)(p_x - a)/(b - a)$ and hence takes time $O(1)$. Thus the entire search takes time $O(\log n)$.

A similar situation is depicted in Figure 5. Let $L_1, \ldots, L_n, L$ be lines and let $P_i := \bigcap(L_i, L)$, $1 \leq i \leq n$. Assume that the points p_i, $1 \leq i \leq n$, appear in the order $p_1, \ldots, p_n$ on line L.

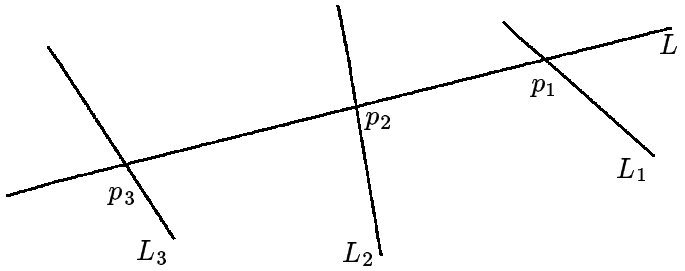


Figure 5. Binary Search on a line

Let L' be an arbitrary line and let $p := \bigcap(L', L)$. Then we can determine the position of point p relative to the points p_i by binary search in time $O(\log n)$, i.e., we can determine i so that p lies between p_i and p_{i+1} on line L. The only change required in the program above is to replace line (3) by

(3′) **do if** p lies above $\bigcap(L, L_{middle})$ on L

Note that line (3′) takes time $O(1)$ and that it is not necessary to precompute the p_i's. Instead $O(\log n)$ p_i's are computed in line (3′) during the execution of the algorithm.

We will use both applications of binary search and tree search frequently in the sequel. It is important to observe that all methods described in Sections 3.3 to 3.6 work for *all* ordered universes as long as a comparison takes time $O(1)$. In both examples above the line segments $L_1, L_2, \ldots, L_n$ form an oriented universe for the problem at hand.

Finally, if $v \in \mathbb{R}^2$ then we use $x(v)$ and $y(v)$ to denote the x- and y-coordinate of point v respectively.

8.1. Convex Polygons

Convex polygons are particularly easy to deal with computationally. They are also a preferable kind of polygons in many applications, e.g., in graphics and numerical analysis. Therefore we will study the problem of decomposing arbitrary polygons into convex parts, in particular into triangles, in Section 8.4.2. In this section we describe algorithms for basic questions about convex polygons: how to decide whether a point lies inside a polygon, to compute the intersection of a line and a polygon, to decide whether two polygons intersect and to compute the intersection of two polygons. We will see that convexity permits very efficient solutions, which is a general fact in computational geometry. The triangulation algorithm of Section 8.4.2 will be another example of the use of convexity for developing fast algorithms.

Throughout this section we assume that adjacent edges of a polygon are not collinear. In general, we assume that polygons are given by the sequence of their vertices in clockwise order. For convex polygons a (balanced) hierarchical representation is particularly useful.

Definition: A sequence $P_0, P_1, \ldots, P_k$ of polygons is a **balanced hierarchical representation** of convex polygon P if

a) P_0 has at most four vertices,

b) $P_k = P$, and

c) P_{i-1} can be derived from P_i by deleting some vertices.

More precisely, out of three consecutive vertices of P_i at least one is deleted and no four consecutive points are deleted. ∎

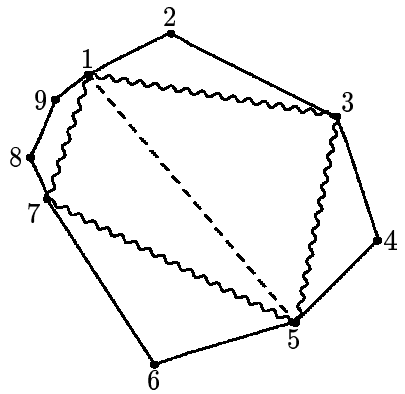


Figure 6. Balanced hierarchical representation

Figure 6 shows a balanced hierarchical representation of a convex polygon with 9 vertices. Polygon P_2 consists of all 9 vertices, P_1 consists of vertices 1, 3, 5 and 7 and P_0 consists of vertices 1 and 5. A balanced hierarchical representation of a

convex polygon is obtained naturally if we store the sequence of edges in the leaves of a balanced tree, say a $(2,4)$-tree (cf. Fig. 7).

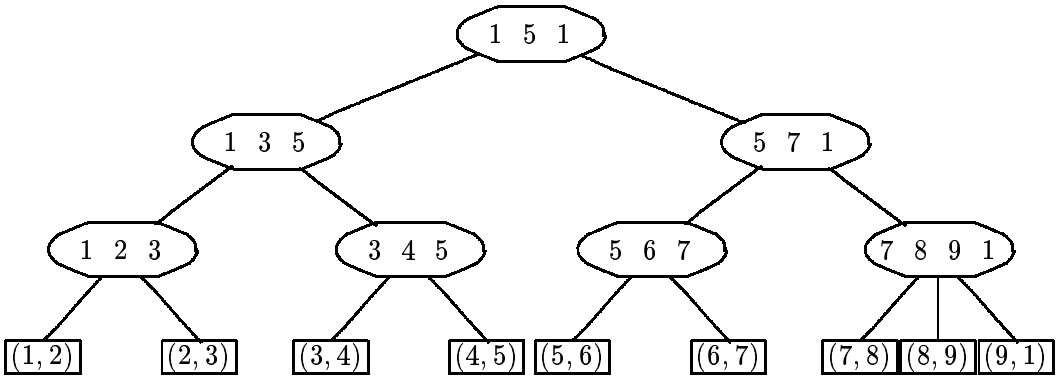


Figure 7. $(2,4)$-tree representing a convex polygon

Then every level of the tree corresponds to one of the polygons in the hierarchical representation, i.e., the root node is polygon P_0, the nodes of depth 1 represent polygon P_1, ... In our example polygon P_0 consists of points 1 and 5 only. When we pass to polygon P_1 we replace edge $(1,5)$ of P_0 by the chain $(1,3),(3,5)$, a fact which is reflected in the first son of the root, and we replace edge $(5,1)$ of P_0 by chain $(5,7),(7,1)$, a fact which is reflected in the second son of the root. Similarly, edge $(7,1)$ of polygon P_1 is replaced by chain $(7,8),(8,9),(9,1)$, a fact which is reflected in the right-most grandson of the root.

We can draw two simple, but important consequences from the fact that balanced hierarchical representations of polygons are obtained by storing the edges in a $(2,4)$-tree.

Lemma 1.

a) *A balanced hierarchical representation of convex polygon P can be computed in time $O(n)$ where n is the number of vertices of P.*

b) *If $P_0,\ldots,P_k$ is a balanced hierarchical representation of P then $k = O(\log n)$.*

Proof: obvious. ∎

Our first use of the balanced hierarchical representation of convex polygons is a simple algorithm for deciding whether a point lies inside a polygon.

Theorem 1. *Given a balanced hierarchical representation of a convex polygon P, a point p inside P and an arbitrary point x we can decide whether x lies in P in time $O(\log n)$ where n is the number of vertices of P.*

Proof: Consider a subdivision of the plane obtained by drawing n semi-infinite straight lines starting at point p and going through the vertices of P. This subdivision splits the plane into n segments which can be *ordered* in a natural way,

namely in clockwise (say) order around p. The idea is then to use binary search on the n segments to determine the segment which contains x and then to decide in an additional step whether x is inside or outside of P.

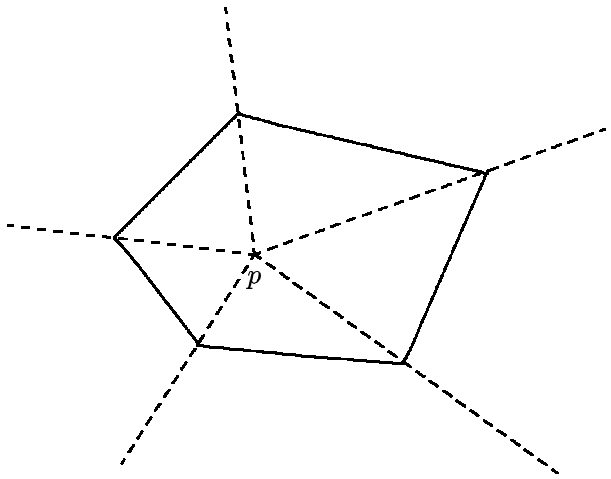


Figure 8. Rays from p through corners

This algorithm is easily implemented as a tree search. Let T be a $(2,4)$-tree defining the balanced hierarchical representation of P. The vertices stored in the root together with point p define a subdivision into at most 4 segments. The segment containing x can be found in $O(1)$ time. We proceed to the appropriate son of the root thereby replacing the segment containing x by at most 4 smaller segments. Again we can identify the segment containing x in $O(1)$ time. Iterating this process $O(\log n)$ times we finish the search. ∎

We will next describe an algorithm for intersecting a straight line and a convex polygon. The basic idea underlying this algorithm is very simple. We start with P_0 and determine the vertex v of P_0 closest to line L. Then we grow P_0 towards L, i.e., we replace the vincinity of vertex v by a part of P_1, and so on. Lemma 2 below states that letting a polygon grow towards a line is an efficient process.

Lemma 2. *Let $P_0, P_1, \ldots, P_k$ be a balanced hierarchical representation of convex polygon P and let d be a direction in the plane. Let $d(P_i)$, $0 \leq i \leq k$, be the set of vertices of P_i which are maximal in the direction d. Then*

a) $|d(P_i)| \leq 2$;

b) *if $p \in d(P_{i+1})$ then there is $q \in d(P_i)$ such that either $p = q$ or p and q are separated by at most two nodes on the vertex list of P_{i+1}.*

Proof: a) There are at most two vertices maximal in any direction since adjacent edges are not collinear.

b) Let $p \in d(P_{i+1}) - d(P_i)$. Draw a tangent to P_i which is perpendicular to direction d. It touches P_i in the vertices in $d(P_i)$ and it divides the plane into two halfspaces of which one completely contains P_i (cf. Fig. 9).

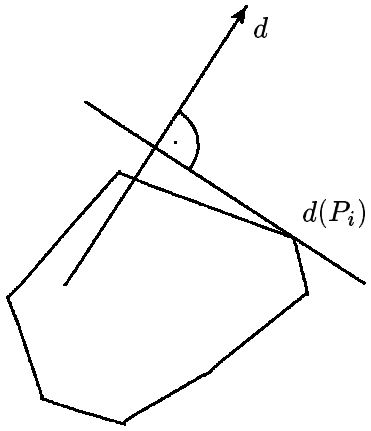


Figure 9. Tangent perpendicular to d

Consider edges of P_i and their replacements when passing to P_{i+1}. If such a replacement has a vertex in the other halfspace then the corresponding edge of P_i must be incident to a vertex of $d(P_i)$ by convexity. Finally observe that the replacement of an edge can introduce at most three new vertices. ∎

Lemma 2 leads immediately to a logarithmic time algorithm for intersecting straight lines and convex polygons.

Theorem 2. *Given a balanced hierarchical representation of a convex n-gon P and a straight line L the intersection of P and L can be determined in time $O(\log n)$.*

Proof: Assume that L does not intersect P_0, the reverse case is simpler. Let d be the direction perpendicular to line L. Since P_0 has at most four vertices we can determine $d(P_0)$ in time $O(1)$. Next we run through $P_1, P_2, \ldots, P_i, \ldots, P_k$ in turn and compute $d(P_i)$ until either $i = k$ or P_i intersects L. Note that $d(P_i)$ can be computed from $d(P_{i-1})$ in constant time by Lemma 2. Also assuming that P_{i-1} does not intersect L we can decide in constant time whether P_i intersects L. This follows from the observation that an intersection can only occur in a constant size neighborhood of $d(P_{i-1})$. We conclude that in time $O(\log n)$ we have either found that P_k does not intersect L or we have found the smallest i so that P_i intersects L. In the latter case we have also found the pair $e_1(i)$, $e_2(i)$ of edges of P_i which are intersected by L. We can now replace edges $e_1(i)$, $e_2(i)$ of P_i by the chain of (at most 3) edges of P_{i+1} which represent e_1, e_2 respectively and so compute edges $e_1(i+1)$, $e_2(i+1)$ of P_{i+1} intersected by L in constant time. Proceeding in this way we compute $P \cap L$ in time $O(\log n)$. ∎

Theorem 2 is readily extended to line segments and as a special case to points. It thus provides us with an alternative proof of Theorem 1.

Theorem 3. *Given a balanced hierarchical representation of convex n-gon P and*

a) *a line segment S we can compute $P \cap S$ in time $O(\log n)$;*

b) *a point p we can decide whether p is inside P in time $O(\log n)$.*

Proof: a) Let S be a segment of straight line L. By Theorem 3 we can compute $P \cap L$ in time $O(\log n)$. $P \cap L$ is a line segment. From $P \cap L$ we can compute $(P \cap L) \cap S = P \cap S$ in constant time.

b) Follows immediately from part a) and the observation that a point is a degenerated line segment. ∎

Our most complex use of balanced hierarchical representations of convex polygons lies in the decision whether convex polygons intersect. We will show that we can decide in logarithmic time whether two convex polygons intersect (Theorem 5). The actual computation of the intersection is much more difficult. Note that two convex n-gons may have $\Omega(n)$ points of intersection (cf. Fig. 10) and therefore no sublinear algorithm for the computation of intersections can exist. Theorem 6 describes an $O(n)$ algorithm for computing the intersection of two convex n-gons.

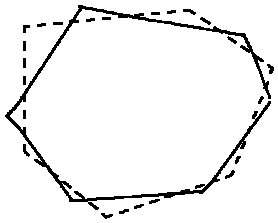


Figure 10. Two intersecting convex polygons

Theorem 4. *Given balanced hierarchical representations of convex n-gon P and convex m-gon Q we can decide in time $O(\log(m+n))$ whether P and Q intersect.*

Proof: The proof is based on a sequence of lemmas. In the first lemma we replace the problem by a simpler one, namely how to decide whether two monotone polygonal chains intersect, and in a second step we show how to solve the simplified problem in logarithmic time.

A monotone polygonal chain is a sequence of vertices $v_0, v_1, \ldots, v_k$ with $y(v_i) > y(v_{i+1})$. A monotone polygonal chain defines an (infinite) convex region if we add two semi-infinite horizontal rays, one for $y = y(v_0)$ and one for $y = y(v_k)$. It can easily be derived that given a balanced hierarchical representation of convex n-gon P we can decompose P into two monotone polygonal chains P_L and P_R in time $O(\log n)$. In Figure 11, P_R (P_L) is closed to the right (left). The decomposition can be achieved by computing the vertices of P with maximal and minimal

y-coordinate which is easily done by the method outlined in Lemma 2 and Theorem 2. We can also compute balanced hierarchical representations for P_R and P_L in logarithmic time by performing a split operation of the $(2,4)$-tree representing P.

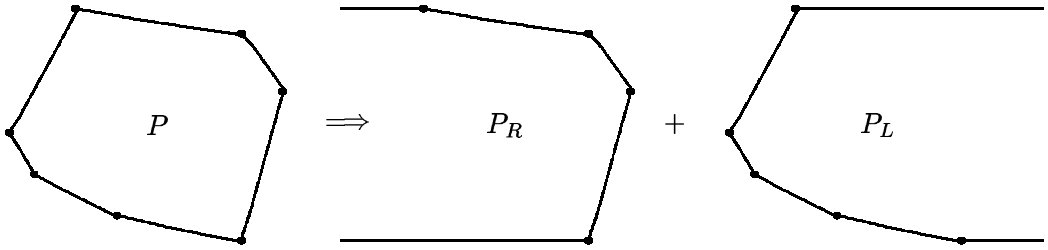


Figure 11. Replacing P by P_R and P_L

Lemma 3. *Let P and Q be convex polygons. Then $P \cap Q \neq \emptyset$ iff $P_L \cap Q_R \neq \emptyset$ and $P_R \cap Q_L \neq \emptyset$.*

Proof: "$\rightarrow$": Since $P = P_L \cap P_R$ and $Q = Q_L \cap Q_R$ we immediately have that $\emptyset \neq P \cap Q = P_L \cap P_R \cap Q_L \cap Q_R$ implies $P_L \cap Q_R \neq \emptyset$ and $P_R \cap Q_L \neq \emptyset$.
"$\leftarrow$": If $P \cap Q = \emptyset$ then there is a straight line L separating P and Q. If L is horizontal then clearly $P_L \cap Q_R = P_R \cap Q_L = \emptyset$. If L is not horizontal then assume w.l.o.g. that P is to the left of L. Then $P_R \cap Q_L = \emptyset$. ∎

Lemma 3 allows us to concentrate on a simpler problem: how to decide whether two monotone polygonal chains intersect. Let R (L) be a monotone polygonal chain which is closed to the right (left) and let $r_1, \ldots, r_m$ ($l_1, \ldots, l_n$) be the edges of R (L). Here r_1, r_m, l_1, l_n are infinite rays with r_1 (l_1) above r_m (l_n) and all the other edges are finite. We will use (a variant of) binary search to decide whether R and L intersect. Let $i = \lfloor (m+1)/2 \rfloor$ and $j = \lfloor (n+1)/2) \rfloor$ and let R_i (L_j) be the lines supporting line segments r_i (l_j). We assume that R_i and L_j intersect (otherwise, L_{j+1} will intersect R_i since adjacent edges are assumed not to be collinear). Then lines R_i and L_j divide the plane into four regions. R and L can each exist in two of these regions. Furthermore, they can coexist in one of the regions. Label the four regions LR, L, R and *empty* as shown in Figure 12, i.e., chain R can exist in regions R and LR.

Define four new monotone chains from R and L as follows. R_{top} consists of edges $r_1, \ldots, r_i$ and a horizontal ray ending in the lower endpoint of r_i. R_{bot} consists of edges $r_i, r_{i+1}, \ldots, r_m$ and a horizontal ray ending in the upper endpoint of r_i. L_{top} and L_{bot} are defined similarly. The algorithm is based on

Lemma 4. *If lines R_i and L_j intersect and segments r_i and l_j do not, and if region LR is above region empty (i.e., seeks $+\infty$ in the y-direction) then*

a) *if the lower endpoint of r_i does not lie in the LR region then $R \cap L \neq \emptyset$ iff $R_{top} \cap L \neq \emptyset$*

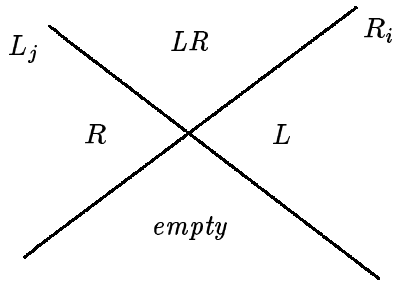


Figure 12. Subdivision induced by R_i and L_j

b) *if the lower endpoint of l_j does not lie in the LR region then $R \cap L \neq \emptyset$ iff $R \cap L_{top} \neq \emptyset$*

c) *if both endpoints of r_i and l_j lie in the LR region and the lower endpoint of r_i has no larger (no smaller) y-coordinate than the lower endpoint of l_j then $R \cap L \neq \emptyset$ iff $R_{top} \cap L \neq \emptyset$ ($R \cap L_{top} \neq \emptyset$).*

Proof: a) If the lower endpoint of r_i does not lie in the LR region then edges $r_{i+1}, \ldots, r_m$ of R are completely contained in region R and hence cannot intersect chain L. Also the new edge of R_{top} does not intersect L. Hence $R \cap L \neq \emptyset$ iff $R_{top} \cap L \neq \emptyset$.

b) similar to part a).

c) Suppose that both endpoints of r_i and l_j lie in the LR region and that the lower endpoint of r_i has no larger y-coordinate than the lower endpoint of l_j.

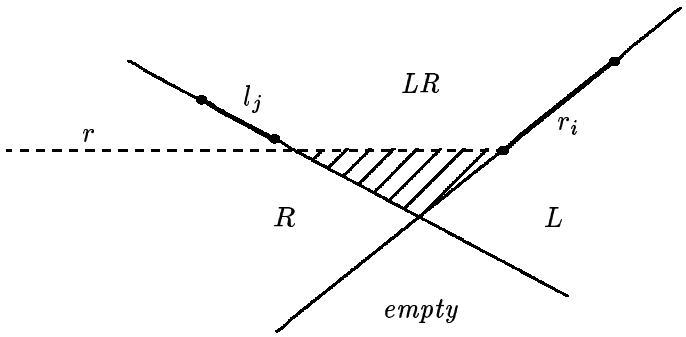


Figure 13. Case c) in Lemma 4

We claim that $R_{top} \cap L \neq \emptyset$ iff $R \cap L \neq \emptyset$. Assume first that $R_{top} \cap L = \emptyset$, but $R \cap L \neq \emptyset$. Then one of the edges $r_{i+1}, \ldots, r_m$ must intersect L. Hence chain L must have a point in the dashed region of Figure 13. Hence by convexity chain L must intersect R_{top}, contradiction. Hence $R_{top} \cap L = \emptyset$ implies $R \cap L = \emptyset$. Assume next that $R_{top} \cap L \neq \emptyset$, but $R \cap L = \emptyset$. Then only the lower horizontal ray of R_{top}, call it r, can intersect L. Ray r cannot intersect the top horizontal ray l_1 of chain L by the relative position of r_i and l_j. Hence if we follow chain L starting at the lower

horizontal ray l_n, we find $k > 1$, so that l_k intersects r. Continuing on chain L we move into the convex region defined by chain R. Since chain R seeks $-\infty$ in the x-direction and chain L seeks $+\infty$ in the x-direction there must be an intersection of R and L. Thus $R_{top} \cap L \neq \emptyset$ implies $R \cap L \neq \emptyset$. ∎

An analogous Lemma can be shown for the case when region LR is below region *empty*. Lemma 4 allows us to reduce in a constant number of steps the number of edges of one of the polygonal chains by half. More precisely, we consider the middle edges r_i and l_j and supporting lines R_i and L_j. If R_i and L_j are collinear and R_i is to the left of L_j then L and R do not intersect. If R_i and L_j are collinear and R_i is to the right of L_j then R_{i+1} (the line supporting edge r_{i+1}) and L_j will intersect. We assume for the rest of the discussion that R_i and L_j intersect, the reverse case is similar. If segments r_i and l_j intersect then we discovered a point of intersection. If segments r_i and l_j do not intersect then we can discard half of one of the chains by Lemma 4 and thus reduce the size of the problem by a constant fraction.

Thus in $O(\log(n+m))$ steps we will reduce one of the chains to a chain of a bounded number of edges, say at most 10. For each of these edges we can test for intersection with the other chain in logarithmic time by Theorem3a). Thus further $O(\log(n+m))$ steps will finish the test for intersection. ∎

Theorem 5. *Let P and Q be convex n-gons. Then $P \cap Q$ can be computed in time $O(n)$.*

Proof: Let $P = p_1, p_2, \ldots, p_n$ and let $Q = q_1, q_2, \ldots, q_m$, $m \leq n$, be the vertex lists for P and Q. Let p be a point inside P; e.g., we might take p as the center of gravity of vertices $p_1, p_2, \ldots, p_n$.

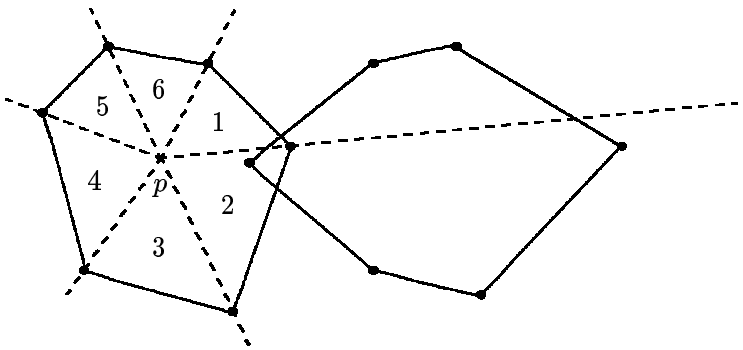


Figure 14. Subdivision with center p

Point p can certainly be computed in linear time from the vertex list of P. We proceed as in the proof of Theorem 1. Consider the subdivision of the plane defined by the rays starting at point p and going through the vertices of P. For vertex q_j of Q let $S(q_j)$ be the segment containing q_j.

We show next that we can compute all intersections of edges of Q with edges of P in linear time. Note first that we can certainly compute $S(q_1)$ in time $O(n)$. Next look at line segment $L(q_1, q_2)$. We can determine all intersections of edge $L(q_1, q_2)$ with edges of P in time $O(1 + s_1)$, where s_1 is the number of rays crossed by line segment $L(q_1, q_2)$. This follows from the fact that we only have to look at the rays bounding $S(q_1)$ and at the one edge of P going through that segment in order to decide whether $L(q_1, q_2)$ intersects that edge (and if it does, where it does intersect) and whether $L(q_1, q_2)$ leaves the segment. If it does not leave the segment we complete the operation in time $O(1)$, if it does leave the segment we can apply the same argument to the segment entered. Similarly, we can compute all intersections of edge $L(q_2, q_3)$ with the edges of P in time $O(1 + s_2)$ where s_2 is the number of rays intersecting $L(q_2, q_3)$. In this way, we compute all intersections of edges of Q and edges of P in time $O(m + \sum_i s_i)$. Finally, observe $\sum_i s_i \leq 2n$ since every ray can cut at most 2 edges of Q.

The algorithm as it is described above correctly computes $P \cap Q$ if there are some edges of P and Q which intersect or if Q is completely contained in P. Assume now that all vertices of Q are outside P and that there are no edge intersections; note that both facts will be reported by our algorithm. Then P and Q intersect iff P is contained in Q iff p is contained in Q. The latter fact is easily tested in time $O(m)$. To sum up, we have computed the intersection of P and Q in time $O(n + m)$. ∎

8.2. Convex Hulls

This section is devoted to convex hull problems. Let $S \subseteq \mathbb{R}^2$ be finite. The **convex hull** $CH(S)$ is the smallest convex set containing S. The convex hull of S is always a polygon whose vertices are points of S. This is intuitively obvious from the rubber band model. Take a rubber band and stretch it so that it encloses all points of S. If one lets it loose then the rubber band will form the convex hull of set S. It will clearly form a convex polygon whose vertices are points of S. A proof of this fact is left to the reader (Exercise 8).

We use $BCH(S)$ to denote the boundary of the convex hull of S. The **convex hull problem** is then defined as follows: Given $S \subseteq \mathbb{R}^2$ compute the boundary points $BCH(S)$ in clockwise order, i.e., compute the standard representation of convex polygon $BCH(S)$.

We show that $BCH(S)$ can be computed in time $O(n \log n)$, $n = |S|$, even if S is not given at once but is given point by point (Theorems 2 and 4). Moreover, this is optimal (Theorem 3). A linear time algorithm exists if S is sorted or, more generally, if a simple polygon with S as its vertex set is given (Theorem 1). Finally, convex hulls can be maintained under insertions, deletions in time $O((\log n)^2)$ per operation (Theorem 5).

Theorem 1. *Let $S \subseteq \mathbb{R}^2$ and let Q be a simple polygon whose vertices are the points of S. Then $BCH(S)$ can be computed in time $O(|S|)$.*

Proof: We will first show that it suffices to solve a somewhat simpler problem, the "upper" convex hull problem. Let v_{min} (v_{max}) be a point with minimal (maximal) x-coordinate in S. Then the chord $L(v_{min}, v_{max})$ divides $CH(S)$ into two convex regions which we call the upper and lower convex hull of S (with respect to chord $L(v_{min}, v_{max})$), cf. Figure 15. Apparently, it suffices to compute the upper and lower convex hull of S. We show how to compute the upper convex hull of S in linear time.

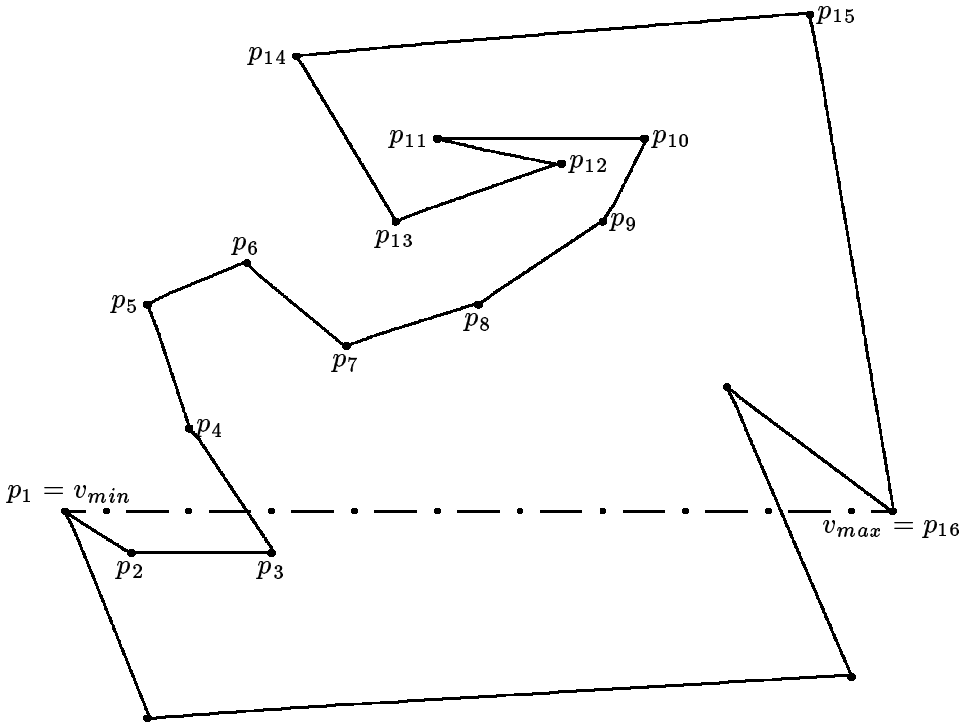


Figure 15. Splitting the convex hull problem into two subproblems

Let $p_1, \ldots, p_n$ with $p_1 = v_{min}$, $p_n = v_{max}$ be the upper path from v_{min} to v_{max} in polygon Q. For $1 \leq i \leq j \leq n$, let $P[v_i, v_j]$ denote the polygonial chain $v_i, v_{i+1}, \ldots, v_j$. Then $P = P[v_{min}, v_{max}]$ is the upper path from v_{min} to v_{max}. The upper convex hull of P is a subsequence $q_1, \ldots, q_m$ of $p_1, \ldots, p_n$ which certainly has the following two properties:

(1) $q_1 = v_{min}$, $q_m = v_{max}$ and (q_{i-1}, q_i, q_{i+1}) is a right turn for $2 \leq i \leq m-1$.

(2) No vertex of $P[q_i, q_{i+1}]$ lies left of $L(q_i, q_{i+1})$ for $1 \leq i \leq m-1$.

Figure 16 illustrates these properties. We call a subsequence $P[q_i, q_{i+1}]$ of the upper chain satisfying (2) a **pocket** and the finite closed region $R(q_i, q_{i+1})$ whose boundary

is $P[q_i, q_{i+1}] \circ \overline{q_{i+1}q_i}$ the pocket region associated with it. Our first observation is that properties (1) and (2) characterize the upper convex hull.

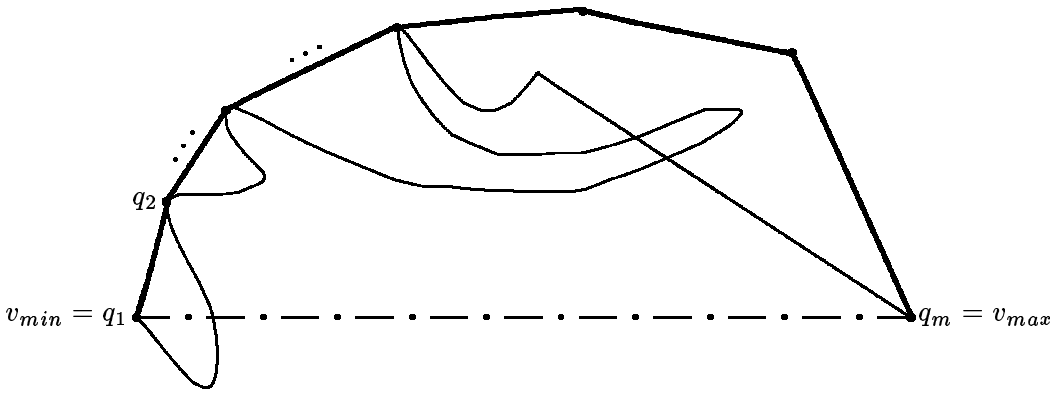


Figure 16. The upper convex hull

Lemma 1. *Let $q_1, \ldots, q_m$ be a subsequence of $p_1, \ldots, p_n$ satisfying (1) and (2). Then $q_1, \ldots, q_m$ is the upper convex hull.*

Proof: Since the chain $q_1, \ldots, q_m$ consists of right turns only by (1) it is convex and the upper convex hull cannot run below chain $q_1, \ldots, q_m$. Thus we only need to show that no point p of P lies above the chain $q_1, \ldots, q_m$. We do so by induction on $\mathit{depth}(p)$ where $\mathit{depth}(p)$ is the number of intersections between a vertical upward ray starting in p and the chain P, not counting the intersection at p. Note first that if p lies on $P[q_i, q_{i+1}]$ for some i and $x(p) < x(q_i)$ or $x(p) > x(q_{i+1})$ then p lies below either $P[q_1, q_i]$ or $P[q_{i+1}, q_m]$ and hence $\mathit{depth}(p) > 0$. So, if p lies on $P[q_i, q_{i+1}]$ and $\mathit{depth}(p) = 0$ then $x(q_i) \leq x(p) \leq x(q_{i+1})$. In this case p does not lie left of $L(q_i, q_{i+1})$ and hence not above the chain $q_1, \ldots, q_m$ by the definition of a pocket. This proves the claim for $\mathit{depth}(p) = 0$. If $\mathit{depth}(p) > 0$ then p lies strictly below an edge of P and hence the claim follows directly from the induction hypothesis. ∎

Our strategy is to compute the sequence $q_1, \ldots, q_m$ iteratively. Suppose that we have inspected vertices $p_1, \ldots, p_r$ of P. Then we will have constructed a sequence $q_1, \ldots, q_t$ such that

(1′) $q_1, \ldots, q_t$ is a subsequence of $p_1, \ldots, p_r$ with $q_1 = p_1$, $q_t = p_r$, $t \geq 2$, and $q_1, \ldots, q_t, v_{max}$ is a sequence of right turns.

(2′) $P[q_i, q_{i+1}]$ is a pocket for $1 \leq i < t$.

Figure 17 illustrates the invariants (1′) and (2′).

Clearly, if no vertex $p_{r+1}, \ldots, p_n$ lies above the chain $C = q_1, \ldots, q_t, v_{max}$ then $P[q_t, v_{max}]$ is also a pocket and we are done. So suppose that there is a successor v of q_t which lies above C. Figure 18 illustrates the two possible situations.

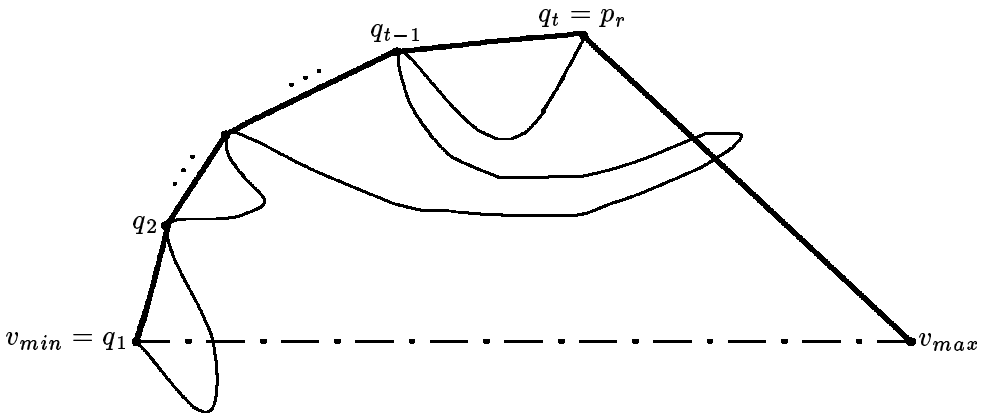


Figure 17. The invariant of the algorithm

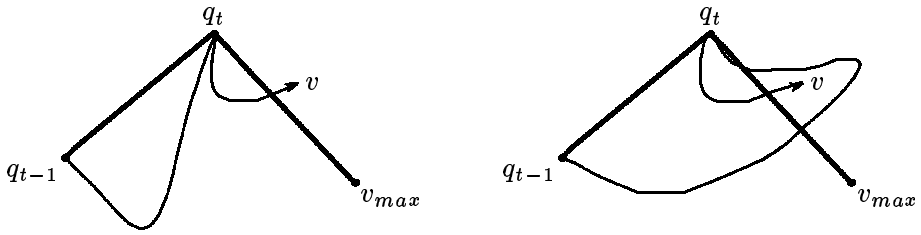


Figure 18. A successor v of q_t above the chain C

In the first situation, we can directly add v to the q-sequence. This will maintain the invariant. In the second situation, v lies in the pocket region $R(q_{t-1}, q_t)$. In this case P must finally cross the line segment $\overline{q_{t-1}q_t}$ and then continue to v_{max} above q_t. Thus we should proceed along P until we leave the pocket $R(q_{t-1}, q_t)$ and then update the q-sequence. In either situation we consider the closest successor p_s of q_t such that p_s lies above C and outside $R(q_{t-1}, q_t)$. The following characterization of p_s is useful.

Lemma 2. *Assume that p_s exists. If p_{r+1} lies left of $L(q_{t-1}, q_t)$ then $p_s = p_{r+1}$. If p_{r+1} lies in the pocket region $R(q_{t-1}, q_t)$ then p_s is the closest successor of q_t left of $L(q_{t-1}, q_t)$. If p_{r+1} does not lie left of $L(q_{t-1}, q_t)$ and not in $R(q_{t-1}, q_t)$ then p_s is the closest successor of q_t which lies left of $L(q_t, v_{max})$.*

Proof: The claim is obvious if p_{r+1} lies left of $L(q_{t-1}, q_t)$. If p_{r+1} lies in $R(q_{t-1}, q_t)$ then p_s is the closest successor of q_t outside $R(q_{t-1}, q_t)$ since this successor must also lie left of $L(q_{t-1}, q_t)$ by the simplicity of P. Finally, if p_{r+1} does not lie in $R(q_{t-1}, q_t)$ and not left of $L(q_{t-1}, q_t)$ then p_{r+1} lies either on $L(q_{t-1}, q_t) - \overline{q_{t-1}q_t}$ and hence $p_s = p_{r+1}$ or p_{r+1} lies right of $L(q_{t-1}, q_t)$. In the latter case, p_s is the closest successor of q_t left of $L(q_t, v_{max})$. ∎

How should we update the q-sequence? Let i, $1 \leq i \leq t$, be such that $L(q_i, p_s)$ is the upper tangent of p_s on C, cf. Figure 19, i.e., no q_j, $1 \leq j \leq t$, is left of $L(q_i, p_s)$.

Then i is clearly the maximal index $\leq t$ such that either (q_{i-1}, q_i, p_s) is a right turn or $i = 1$.

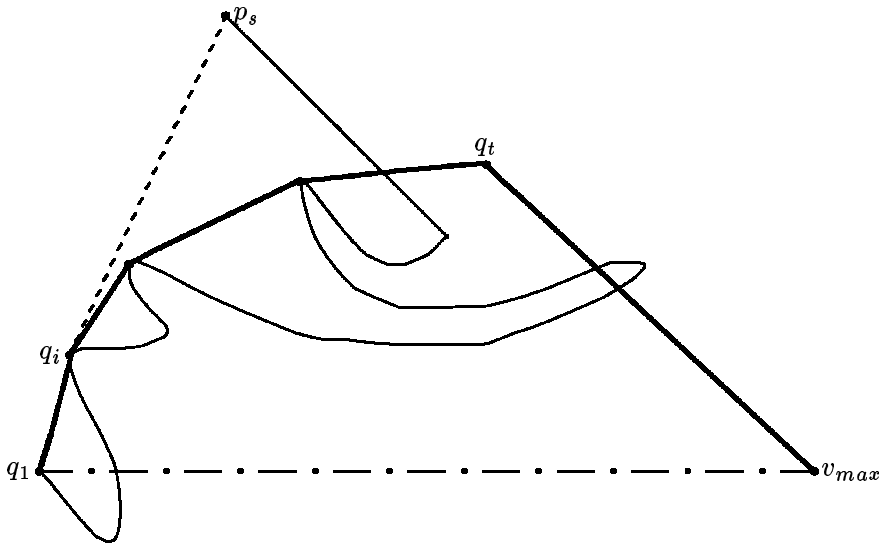


Figure 19. $L(p_s, q_i)$ is the upper tangent for p_s on C

With this choice of i the sequence $q_1, \ldots, q_i, p_s, v_{max}$ is a sequence of right turns, i.e., satisfies (1′). We claim that it also satisfies (2′).

Lemma 3. *$P[q_i, p_s]$ is a pocket.*

Proof: Let $v \in P[q_i, p_s]$. We show the stronger claim that v does not lie above the chain C' which consists of the line segments $\overline{q_1 q_2}, \ldots, \overline{q_{i-1} q_i}$ and the infinite ray starting in q_i and passing through p_s. In order to show this claim it is convenient to also assume an additional invariant:

(3) For $j \geq 2$, no vertex of $P[q_{j-1}, q_j]$ lies above C_j where C_j is the chain consisting of $\overline{q_1 q_2}, \ldots, \overline{q_{j-2} q_{j-1}}$ and the infinite ray starting in q_{j-1} and passing through q_j.

So let v be a vertex of $P[q_i, p_s]$. If $v \in P[q_{j-1}, q_j]$ for some $i < j \leq t$ then v does not lie above C_j by (3) and hence does not lie above C'. Assume next that $v \in P[q_t, p_s]$, $v \neq p_s$, $v \neq q_t$. If p_{r+1} lies in the pocket region $P[q_{t-1}, q_t]$, then v lies in that region and hence not above C_t. Also, p_s is the first vertex after q_t to lie above C_t and hence v does not lie above C'. If p_{r+1} does not lie in the pocket region $P[q_{t-1}, q_t]$ then either $r + 1 = s$ and we have nothing to show or p_{r+1} lies on or below $L(q_t, v_{max})$. In this case p_s is the closest successor of q_t left of $L(q_t, v_{max})$ and hence v is certainly not above C'. We have now shown that no vertex $v \in P[q_i, p_s]$ lies above C' and hence $P[q_i, p_s]$ is a pocket. It remains to verify that the sequence $q_1, \ldots, q_i, p_s$ again satisfies (3). This is obvious for $j \leq i$ and was shown for $P[q_i, p_s]$ in the preceding discussion. ∎

At this point we have arrived at the algorithm formulated by Program 2.

Initialization, i.e., determine a q-sequence satisfying $(1')$ and $(2')$;
while *not_done*
do co $(1')$ and $(2')$ hold at this point **oc**
  let $s > r$ be minimal such that p_s lies above the
  chain $q_1, \ldots, q_t, v_{max}$ and outside the pocket region $R(q_{t-1}, q_t)$;
  if p_s exists
  then let i be minimal such that either $i = 1$ or
    (q_{i-1}, q_i, p_s) is a right turn;
    remove $q_{i+1}, \ldots, q_t$ from the q-sequence and add p_s;
    $r \leftarrow s$
  else *not_done* $\leftarrow$ false
od.

Program 2

The q-sequence is best realized as a stack where an additional element $q_0 = v_{max}$ is added at the bottom. Since (v_{max}, v_{min}, p_s) is always a right turn the following program fragment suffices to update the q-sequence; here t is always the index of the top element of the stack:

while (q_{t-1}, q_t, p_s) is not a right turn
do pop the stack **od**;
push p_s;

Initialization is also easy. We only have to search for the first vertex which lies above $L(v_{max}, v_{min})$. This is achieved by the following program fragment.

$s \leftarrow 2$;
while (v_{max}, v_{min}, p_s) is not a right turn **do** $s \leftarrow s + 1$ **od**;
push v_{max}; push v_{min}; push p_s;

We still have to discuss how vertex p_s is found. We use the characterization given by Lemma 2. In order to apply it, we need to test whether p_{r+1} lies in the pocket region $R(q_{t-1}, q_t)$, cf. Figure 20.

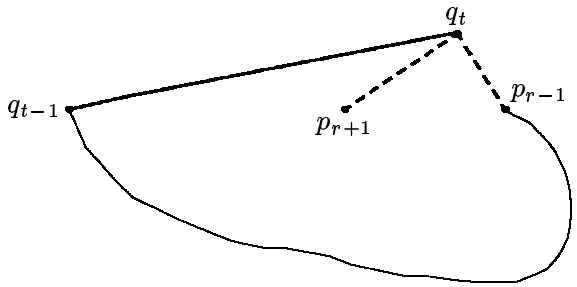


Figure 20. p_{r+1} lies in the pocket region

Lemma 4. *p_{r+1} lies in $R(q_{t-1}, q_t)$ iff either p_{r+1} lies on the line segment $\overline{q_{t-1}q_t}$ or (p_{r-1}, q_t, p_{r+1}) is a left turn and (q_{t-1}, q_t, p_{r+1}) is a right turn.*

Proof: If p_{r+1} lies on the line $L(q_{t-1}, q_t)$ the lemma certainly holds. If p_{r+1} does not lie on the line and p_{r+1} lies in $R(q_{t-1}, q_t)$ then (p_{r-1}, q_t, p_{r+1}) is a left turn and (q_{t-1}, q_t, p_{r+1}) is a right turn. Conversely, if (p_{r-1}, q_t, p_{r+1}) is a left turn and (q_{t-1}, q_t, p_{r+1}) is a right turn then $p_{r-1} \neq q_{t-1}$ and p_{r+1} lies in $R(q_{t-1}, q_t)$. ∎

Lemmas 1 and 4 imply that Program 3 correctly determines p_s.

$s \leftarrow r + 1$;
if (q_{t-1}, q_t, p_s) is not a left turn
then if p_s lies on $\overline{q_{t-1}q_t}$ or (p_{s-2}, q_t, p_s) is a left turn
  then co p_s in $R(q_{t-1}, q_t)$ **oc**
    while (q_{t-1}, q_t, p_s) is not a left turn
    do $s \leftarrow s + 1$ **od**
  else **while** (q_t, v_{max}, p_s) is not a left turn and $s \leq n$
    do $s \leftarrow s + 1$ **od**
  fi
fi

Program 3

Altogether we obtain Program 4 where for simplicity we eliminated variables r and *not_done*.

Program 4 clearly runs in linear time $O(n)$ since at most one vertex is pushed on the q-stack in each iteration. This completes the proof of Theorem 1. ∎

Figure 21 illustrates the state of algorithm on the example of Figure 15 after vertices $p_1, \ldots, p_8$ are processed. We have $v_{max}, v_{min}, p_5, p_6, p_8$ in the stack and enter the outer loop with $s = 8$ and $t = 4$. We determine $s = 9$, pop p_8 and add p_9. In the next iteration p_9 is popped and p_{10} is pushed. Then p_6 and p_{10} are popped and p_{11} is pushed. When we now search for p_s we find that p_{12} lies in the pocket $P[p_5, p_{11}]$ and hence $s = 14$. So p_{11} is popped and p_{14} is pushed, ...

Although the algorithm described above is fairly short the proof of correctness was quite complicated. If the simple polygon is known to be monotone in x-direction, i.e., $x(p_1) \leq x(p_2) \leq \cdots \leq x(p_n)$, then a much simpler algorithm can be used to construct the convex hull (Exercise 9); in essence, we can shrink the body of the major loop to the while-loop (lines (8) and (9)). A completely different convex hull algorithm for monotone simple polygons, i.e., sets ordered according to x-coordinate, can be derived from the divide and conquer paradigm (Exercise 12 and Lemma 5 below).

[Bemerkung an Kurt: stell das Kapitel um; beginne mit Satz 2 und 3; dann kommt 1, 4 und 5; der obige Absatz fällt dann weg]

```
s ← 2;
while ¬Rightturn(v_max, v_min, p_s) do s ← s + 1 od;
push v_max; push v_min; push p_s;
while s < n
do s ← r + 1;
    if (q_{t-1}, q_t, p_s) is not a left turn
    then if p_s lies on q_{t-1}q_t or (p_{s-2}, q_t, p_s) is a left turn
            then co p_s in R(q_{t-1}, q_t) oc
                    while (q_{t-1}, q_t, p_s) is not a left turn
                    do s ← s + 1 od
            else  while (q_t, v_max, p_s) is not a left turn and s ≤ n
                    do s ← s + 1 od
            fi
    fi;
    if s ≤ n
    then while (q_{t-1}, q_t, p_s) is not a right turn
            do pop the stack od;
            push p_s;
    fi;
od.
```

Program 4

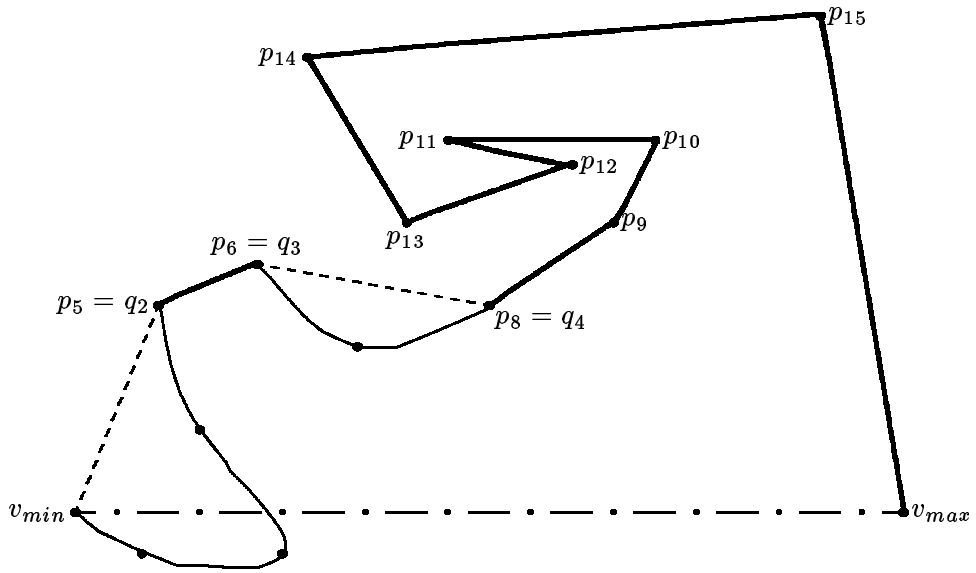


Figure 21. Situation with $s = 8$ and $t = 4$

Theorem 2. *Let $S \subseteq \mathbb{R}^2$, $|S| = n$. Then $BCH(S)$ can be computed in time $O(n \log n)$.*

Proof: Sort S lexicographically, i.e., $p < q$ if $x(p) < x(q)$ or $x(p) = x(q)$ and $y(p) < y(q)$. Let $p_1, \ldots, p_n$ be the sorted version of S. Then $p_1, \ldots, p_{n-1}, p_n, p_{n-1}, \ldots, p_1$ is a simple polygon through set S from which $BCH(S)$ can be computed in linear time. Thus the total cost of constructing the convex hull is $O(n \log n)$. ∎

Can we do better than time $O(n \log n)$? The following theorem shows that we cannot hope to do better.

Theorem 3. *In the rational decision tree model, it takes time $\Omega(n \log n)$ to compute $BCH(S)$ for a set S of n elements.*

Proof: We will reduce the sorting problem to the problem of constructing the convex hull. Let $x_1, x_2, \ldots, x_n$ be an arbitrary sequence of real numbers and let $S = \{(x_i, x_i^2) \mid 1 \leq i \leq n\}$. Then all points of S lie on the parabola $y = x^2$ and hence all points of S are vertices of the convex hull. Moreover, the order of the vertices of the convex hull is identical with the order of x-coordinates. Hence computing $BCH(S)$ is tantamount to sorting sequence $x_1, \ldots, x_n$ and the theorem follows from Theorem 10 of Section 2.1.6. ∎

In the proof of Theorem 3 our definition of the convex hull problem—compute the vertices of the boundary in *clock-wise order*—plays a crucial role. An apparently simpler problem is to compute the set of vertices in *some order*. An $\Omega(n \log n)$ lower bound for the simpler problem was shown in Theorem 15 of Section 2.1.6.

Let us turn to the convex hull problem for dynamic sets next, i.e., sets which grow and shrink by insertions and deletions. If we confine ourselves to insertions only, then the methods of Section 1 provide us with an efficient solution.

Theorem 4. *Let $S \subseteq \mathbb{R}^2$, $|S| = n$, and $p \in \mathbb{R}^2$. Given a balanced hierarchical representation of $BCH(S)$ a balanced hierarchical representation of $BCH(S \cup \{p\})$ can be computed in time $O(\log n)$.*

Proof: Determine first whether p lies in the interior of $BCH(S)$. This takes time $O(\log n)$ according to Theorem 1. If so, then we are done. If not, then we compute the tangents of p and $BCH(S)$ by binary search as follows. Suppose that we look for the "upper" tangent and let q be a vertex of $BCH(S)$, cf. Fig. 22.

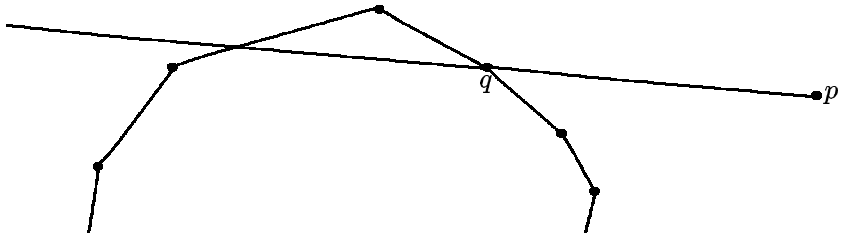


Figure 22. Drawing a tangent from p

Consider line $L(p,q)$ and how it intersects the polygon. If q is the only point of intersection then we are done. If not, then $L(p,q)$ either enters the polygon at q or leaves the polygon at q. The three cases can be easily distinguished in time $O(1)$ by comparing line $L(p,q)$ with the edges of $BCH(S)$ incident to q. Moreover, the case distinction determines whether to continue the search in clockwise or counterclockwise direction. Thus the tangents can be computed in time $O(\log n)$. Two split and one concatenate operation on the balanced hierarchical representations complete the construction. ∎

Intermixed insertions and deletions are harder to deal with. We show an $O((\log n)^2)$ bound by applying the theory of order decomposable problems.

Theorem 5. *There is a data structure for convex hulls so that insertions and deletions take time $O((\log n)^2)$ where n is the actual size of the set.*

Proof: By Theorem 10 of Section 7.1.3. it suffices to show that the convex hull problem is order decomposable with $C(n) = O(\log n)$. Let $\leq$ be the lexicographic ordering on $\mathbb{R}^2$. Then we have

Lemma 5. *Let $p_1 \leq p_2 \leq \cdots \leq p_n$, $p_i \in \mathbb{R}^2$, and $1 \leq m \leq n$. Given balanced hierarchical representations of $BCH(\{p_1,\ldots,p_m\})$ and $BCH(\{p_{m+1},\ldots,p_n\})$ we can compute the balanced hierarchical representation of $BCH(\{p_1,\ldots,p_n\})$ in time $O(\log n)$.*

Proof: Let $L = BCH(\{p_1,\ldots,p_m\})$ and $R = BCH(\{p_{m+1},\ldots,p_n\})$. Note that L and R are disjoint. Our main problem is to compute the two tangents of L and R. We show how to compute the upper tangent in time $O(\log n)$ by binary search on the upper path from the leftmost point of L (R) to the rightmost point of L (R).

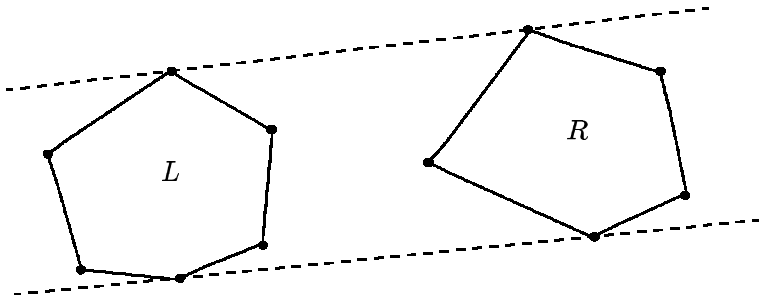


Figure 23. The upper and the lower tangent

Let $r_1,\ldots,r_t$ be that path in L and $q_1,\ldots,q_s$ be that path in R. Points r_1, r_t, q_1, q_t are easily determined in time $O(\log n)$. We have to find h, $1 \leq h \leq t$, and k, $1 \leq k \leq s$, so that line $L(r_h, q_k)$ intersects neither L nor R. Assume inductively, that we found *Llow*, *Lhigh*, *Rlow*, *Rhigh* such that $Llow \leq h \leq Lhigh$ and $Rlow \leq k \leq Rhigh$. Let $i \leftarrow \lfloor (Llow + Lhigh)/2 \rfloor$ and $j \leftarrow \lfloor (Rlow + Rhigh)/2 \rfloor$.

Consider oriented line $L(r_i, q_j)$. This line either touches polygon L in r_i or enters it or leaves it and does so similarly for polygon R (cf. Fig. 24).

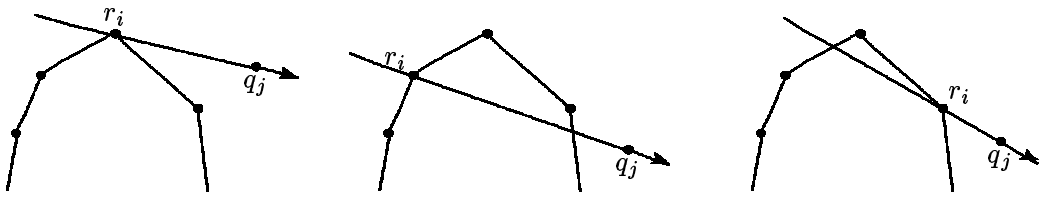


Figure 24. The 3 possibilities for $L(r_i, q_j)$

Thus we have to distinguish nine cases.

Case 1: $L(r_i, q_j)$ touches in r_i and q_j
Then we are done and have $h = i$, $k = j$.

Case 2: $L(r_i, q_j)$ touches in r_i and enters in q_j (cf. Fig. 25).

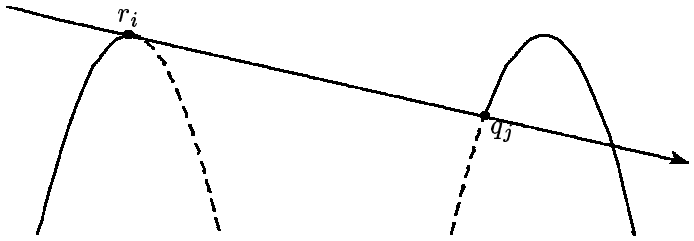


Figure 25. Case 2

Then r_h certainly does not follow r_i, i.e., $h \leq i$, and q_k does not precede q_j. Hence $Lhigh \leftarrow i$ and $Rlow \leftarrow j$ reduces the size of the problem and discards a fraction (shown dashed in Figure 25) of both polygonal chains.

Case 3: $L(r_i, q_j)$ touches in r_i and leaves in q_j (cf. Fig. 26).

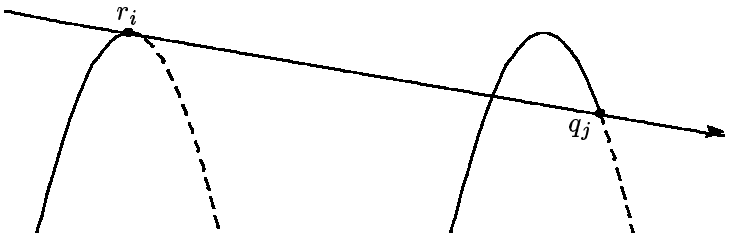


Figure 26. Case 3

Then r_h certainly does not follow r_i and q_k does not follow q_j. Hence $Lhigh \leftarrow i$ and $Rhigh \leftarrow j$ reduces the size of the problem.

Case 4: $L(r_i, q_j)$ leaves in r_i and touches in q_j.
This case is symmetric to case 2.

Case 5: $L(r_i, q_j)$ enters in r_i and touches in q_j.
This case is symmetric to case 3.

Case 6: $L(r_i, q_j)$ leaves in r_i and enters in q_j (cf. Fig. 27).

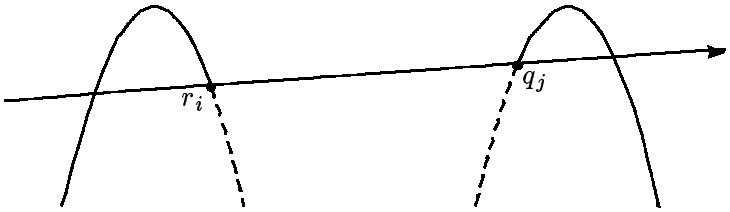


Figure 27. Case 6

Then r_h does not follow r_i and q_k does not precede q_j. Hence $Lhigh \leftarrow i$ and $Rlow \leftarrow j$ reduces the size of the problem.

Case 7: $L(r_i, q_j)$ leaves in r_i and leaves in q_j (cf. Fig. 28).

Figure 28. Case 7

Then certainly r_h does not follow r_i and hence $Lhigh \leftarrow i$ reduces the size of the problem.

Case 8: $L(r_i, q_j)$ enters in r_i and enters in q_j.
This case is symmetric to case 7.

Case 9: $L(r_i, q_j)$ enters in r_i and leaves in q_j (cf. Fig. 29).

Let m be a vertical line so that no point of L (R) is to the right (left) of m, and let t_L (t_R) be a tangent to L (R) in point r_i (q_j). Let p be the intersection of t_L and t_R. Assume that p lies to the left or on m, the reverse case is symmetric. Since all of R is to the right or on m and below or on t_R and hence below t_L we conclude that r_h cannot precede r_i. Hence $Llow \leftarrow i$ reduces the size of the problem.

In either case, we have shown how to eliminate in time $O(1)$ at least half of one of the paths. Hence after $\log s + \log t$ steps at least one of the paths is reduced to a

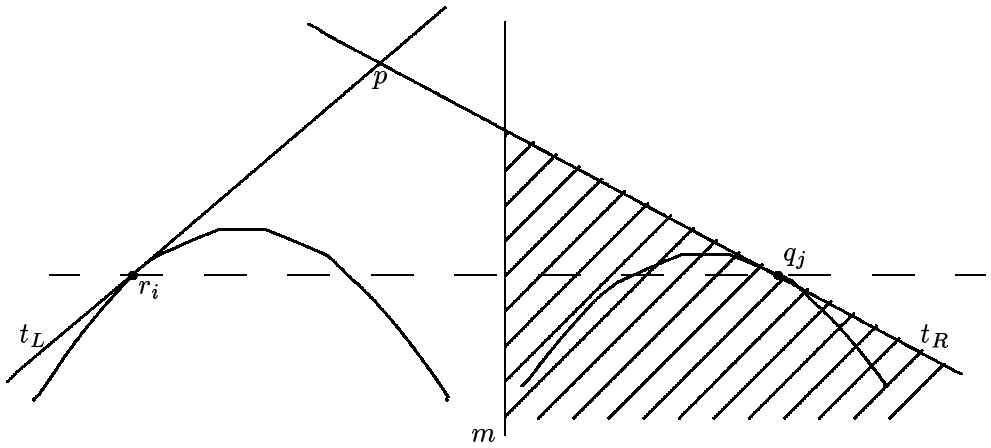


Figure 29. Case 9

single point. After that only cases 1 to 5 can occur and hence further $\log s + \log t$ steps will finish the computation.

We have thus shown how to compute the tangents of L and R in time $O(\log n)$. It is now easy to complete the construction. A few split and concatenate operations are sufficient. ∎

By Lemma 1 we can merge two (disjoint) convex hulls in time $O(\log n)$, thus the convex hull problem is order decomposable with $C(n) = O(\log n)$ and the theorem follows. ∎

8.3. Voronoi Diagrams and Searching Planar Subdivisions.

In this section we study closest point problems and related searching problems in the plane. A very versatile data structure for closest point problems are Voronoi Diagrams. A Voronoi Diagram for a point set S partitions the plane into $|S|$ polygonal regions, one for each point v of S. The (open) Voronoi region of point x consists of all points of $\mathbb{R}^2$ which are closer to x than to any other point of S. We will show that the Voronoi Diagram can be constructed in time $O(n \log n)$, where $n = |S|$. Moreover, Voronoi Diagrams can be searched efficiently in logarithmic time. More generally, we will show that any planar subdivision, i.e., a partition of the plane into polygonal regions allows for logarithmic search time. In addition, the data structure required for the search takes linear space and can be constructed in time $O(n \log n)$. We close the section with a discussion of several applications of Voronoi Diagrams.

[Bemerkung für Kurt: nenne die Elemente von S sites; diskutiere auch allgemeinere Metriken, d.h. abstrakte Diagramme; diskutiere auch Delauney Triangulierung]

8.3.1. Voronoi Diagrams

Let $S = \{x_1, \ldots, x_n\} \subseteq \mathbb{R}^2$. For i, $1 \leq i \leq n$, let $VR(x_i) = \{y \mid dist(x_i, y) \leq dist(x_j, y)$ for all $j\}$ be the Voronoi region of a point x_i. Figure 30 shows a Voronoi Diagram for a set of 5 points.

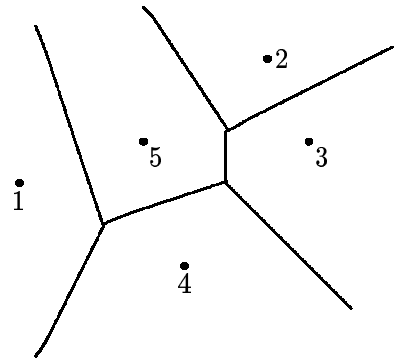


Figure 30. A point set and its Voronoi Diagram

For integers i, j, $1 \leq i, j \leq n$, let

$$H(i,j) = \{y \in \mathbb{R}^2 \mid dist(x_i, y) \leq dist(x_j, y)\}.$$

$H(i,j)$ is a half-space defined by the perpendicular bisector of line segment $L(x_i, x_j)$. Clearly, $VR(x_i) = \bigcup_{j \neq i} H(i,j)$.

Thus $VR(x_i)$ is a convex polygonal region. We can now define the **Voronoi Diagram** $VD(S)$ of S as the union of the set of edges and vertices of Voronoi regions $VR(x_i)$, $1 \leq i \leq n$. The regions of the Voronoi Diagram are the Voronoi regions. Clearly, for every vertex x of the Voronoi Diagram there are at least three points x_i, x_j, x_k of S such that $dist(x, x_i) = dist(x, x_j) = dist(x, x_k)$. Throughout this section we use *point* for elements of S and *vertex* for elements of Voronoi Diagram. Also, the edges of the Voronoi Diagram are (parts of) perpendicular bisectors of line segments $L(x_i, x_j)$, $i \neq j$. A Voronoi region is either bounded or unbounded. The unbounded regions are associated with boundary points of the convex hull.

Lemma 1. *$VR(x_i)$ is unbounded iff x_i belongs to $BCH(S)$.*

Proof: "$\Rightarrow$" (indirect). Assume that $VR(x_i)$ is unbounded but x_i does not belong to $BCH(S)$. Since $VR(x_i)$ is a convex polygonal region there is a semi-infinite ray, say L, starting in x_i and running within $VR(x_i)$. Since x_i does not belong to $BCH(S)$ ray L must intersect some edge, say $L(x_j, x_k)$, of the convex hull. Finally observe, that any point on ray L, which is far enough away from point x_i, is closer to x_j (and x_k) than to x_i, a contradiction.

"$\Leftarrow$": Suppose that x_i belongs to $BCH(S)$. Let x_j and x_k be the neighbors of x_i on $BCH(S)$, as shown in Figure 31.

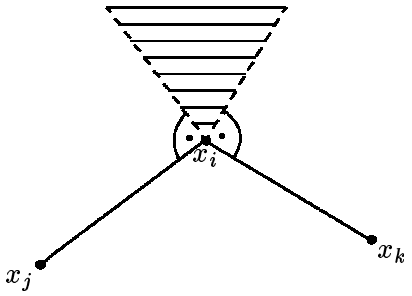


Figure 31. Shaded region is unbouded

Then every point in the shaded region, i.e., in the cone defined by the rays starting in x_i and being perpendicular to lines $L(x_i, x_k)$ and $L(x_i, x_j)$ respectively, is closer to x_i than to any other point of S. ∎

A Voronoi Diagram for point set S partitions the plane into n, $n = |S|$, convex polygonal regions. Thus it is essentially a planar graph. In view of Lemma 2 of Section 4.10, the following lemma is not surprising.

Lemma 2. *A Voronoi Diagram for a set of n points has at most $2n - 4$ vertices and $3n - 6$ edges.*

Proof: We consider a graph D which is dual to the Voronoi Diagram. The vertices of D are the regions of the Voronoi Diagram. Thus D has n vertices. Two vertices of D are connected by an edge if the corresponding regions share an edge (in Figure 32 dual edges are dashed). Thus D is a planar graph and has therefore at most $3n - 6$ edges (cf. Lemma 2 of Section 4.10). Since the edges of D are in one-to-one correspondence to edges of the Voronoi Diagram we infer that the number of edges of the Voronoi Diagram is at most $3n - 6$. Finally, since every vertex of the diagram has degree at least three, there are at most $2n - 4$ vertices. ∎

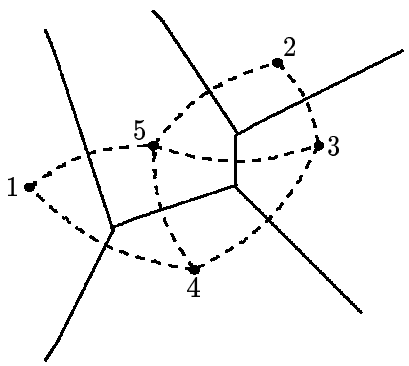


Figure 32. Voronoi Diagram and its dual

We are now ready for the main theorem of this section.

Theorem 1. *Let $S \subseteq \mathbb{R}^2$ be given, $n = |S|$. Then the Voronoi Diagram of S can be computed in time $O(n \log n)$.*

Proof: The algorithm is based on the divide-and-conquer paradigm. Since we aim for an $O(n \log n)$ algorithm we might as well assume that S is sorted lexicographically. Let S_L be the first half of sorted set S, $|S_L| = \lfloor n/2 \rfloor$, and let S_R be the second half of set S. Assume inductively, that we construct the Voronoi Diagrams $VD(S_L)$ and $VD(S_R)$ of sets S_L and S_R by applying out algorithm recursively. This will take time $T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil)$. Also, $T(1) = O(1)$, since the Voronoi Diagram of a singleton set is trivial. The goal is now to construct $VD(S)$ from $VD(S_L)$ and $VD(S_R)$ by "merging".

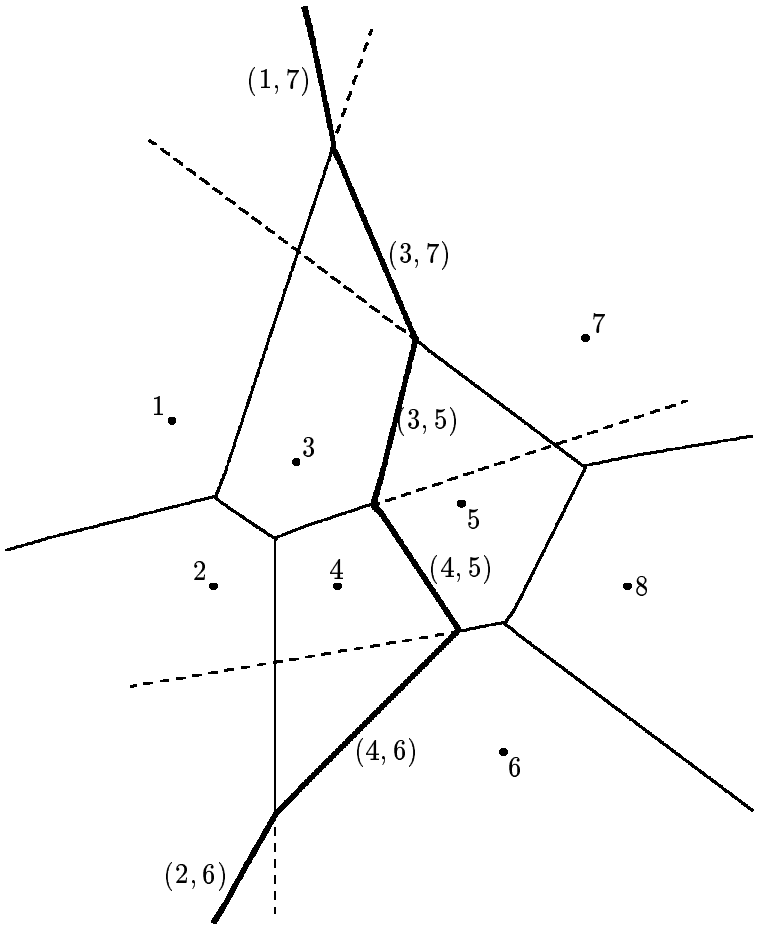


Figure 33. $VD(S_L)$, $VD(S_R)$ and merge line P

In the example of Figure 33 $VD(S)$ is shown solid, the parts of $VD(S_L)$ and $VD(S_R)$ which do not belong to $VD(S)$ are dashed, and line P which belongs to

$VD(S)$ but to neither $VD(S_L)$ nor $VD(S_R)$ is emphasized by a thick line. We have $S_L = \{1, 2, 3, 4\}$ and $S_R = \{5, 6, 7, 8\}$.

Line P is crucial for the merging process. We define it by

$$P := \{y \in \mathbb{R}_2;\ dist(y, S_L) = dist(y, S_R)\}$$

where $dist(y, T) = \min\{dist(x, y);\ x \in T\}$ for a point y and a set T. The following lemma shows that P consists of edges of $VD(S)$ and that P is monotone.

Lemma 3.

a) *$P = \{y;$ y lies on an edge of $VD(S)$ which is a perpendicular bisector of some $x_i \in S_L$ and some $x_j \in S_R\}$. In particular, P consists of two infinite rays and some number of straight line segments.*

b) *P is monotone, i.e., P can be directed such that no line segment runs downward.*

Proof: a) and b): Let P' be the set defined in part a) of the lemma. Then clearly $P' \subseteq P$. It remains to show the contrary. Let $y \in P$ be arbitrary. Then there are $x_i \in S_L$, $x_j \in S_R$ such that $dist(y, x_i) = dist(y, x_j) \leq dist(y, x)$ for all $x \in S$. Thus y lies on the edge which separates the Voronoi regions $VR(x_i)$ and $VR(x_j)$ and hence $y \in P'$. This proves $P = P'$.

We conclude, in particular that P consists of a set of line segments. Every line segment is the perpendicular bisector of some $x_i \in S_L$ and some $x_j \in S_R$. Direct the line segment such that x_i is to the left of the line segment. Then no line segment is directed downward, because this would imply that the x-coordinate of x_i is larger than the x-coordinate of x_j, a contradiction. Since S is lexicographically ordered, we may even conclude that there is at most one horizontal line segment (which then is directed right to left).

Finally, since the curve P is monotone, it cannot be a closed curve. Thus it consists of two infinite rays and some number of (finite) line segments. ∎

Lemma 3 characterizes line P. The significance of line P is given by

Lemma 4. *Let P be as defined above. Direct P in order of increasing y-values and let $L(P)$ be the region of the plane to the left of P. Similarly, let $R(P)$ be the region of the plane to the right of P. Then*

$$VD(S) = (VD(S_L) \cap L(P)) \cup P \cup (VD(S_R) \cap R(P)).$$

Proof: Let VD be the set defined by the expressions on the right hand side. We show $VD(S) \subseteq VD$ and $VD(S) \supseteq VD$.

"$\subseteq$": Let y be an element of $VD(S)$, i.e., y lies on an edge of $VD(S)$. Then there are i, j such that $dist(y, x_i) = dist(y, x_j) \leq dist(y, x)$ for all $x \in S$. If $x_i, x_j \in S_L$ then $y \in VD(S_L) \cap L(P)$, if $x_i \in S_L$, $x_j \in S_R$ or vice versa then $y \in P$ and if $x_i, x_j \in S_R$ then $y \in VD(S_R) \cap R(P)$.

"$\supseteq$": Let $y \in VD$. If $y \in P$ then $y \in VD(S)$ by Lemma 3. So let us assume that $y \in VD(S_L) \cap L(P)$. Since $y \in L(P)$ we have $dist(y, S_L) < dist(y, S_R)$ and since $y \in VD(S_L)$ there are $x_i, x_j \in S_L$ such that $dist(y, S_L) = dist(y, x_i) = dist(y, x_j)$. Thus $y \in VD(S)$. ∎

We infer from Lemma 4 that the construction of line P essentially solves the problem of merging diagrams $VD(S_L)$ and $VD(S_R)$. Lemma 3 characterizes line P. However, it does not give an efficient algorithm for constructing P. Our approach will be to construct P in order of increasing y-values. Thus the first goal must be to construct the (lower) infinite ray L of line P.

Consider the convex hull $BCH(S_L)$ and $BCH(S_R)$. We can obtain $BCH(S)$ from $BCH(S_L)$ and $BCH(S_R)$ by drawing two tangents T_L and T_U. Let T_L be the "lower" tangent. We have shown in Lemma 1 of Section 2 how to construct $BCH(S)$ and T_L in time $O(\log n)$ from $BCH(S_L)$ and $BCH(S_R)$. Moreover, we have:

Lemma 5. *Let the "lower" tangent T_L of $BCH(S_L)$ and $BCH(S_R)$ connect $x_i \in S_L$ and $x_j \in S_R$. Then L is the part of perpendicular bisector of line segment $L(x_i, x_j)$, i.e., to tangent T_L.*

Proof: By Lemma 3a), L is part of the perpendicular bisector of line segment $L(x_i, x_j)$ for some $x_i \in S_L$, $x_j \in S_R$. Moreover, L is an edge of $VD(S)$. Thus regions $VR(x_i)$ and $VD(x_j)$ are unbounded in $VD(S)$ and hence x_i and x_j belong to $BCH(S)$ by Lemma 1. Since $x_i \in S_L$, $x_j \in S_R$ we finally conclude that $L(x_i, x_j)$ is a tangent of $BCH(S_L)$ and $BCH(S_R)$. ∎

Lemma 5 allows us to start the construction of curve P. The idea is now to extend P line segment by line segment. More precisely, let $P = l_1, l_2, \ldots, l_m$ where l_1 and l_m are infinite rays and $l_2, \ldots, l_{m-1}$ are line segments. Assume inductively, that we constructed $l_1, \ldots, l_{h-1}$ for some $h \geq 1$ and that we have determined points $x_i \in S_L$, $x_j \in S_R$ such that l_h is part of the perpendicular bisector of x_i and x_j. Lemma 5 is the base of the induction.

Line segment l_h starts at the terminal point of l_{h-1} ($-\infty$ for $h = 1$). It is part of the perpendicular bisector, call it L, of x_i and x_j for some $x_i \in S_L$, $x_j \in S_R$. Conceptually travel along ray L (in order of increasing y) starting in the terminal point of l_{h-1}. At the beginning of the journey we are within Voronoi regions $VR_L(x_i)$ of x_i with respect to S_L and $VR_R(x_j)$ of x_j with respect to S_R. Ray L will intersect either an edge of $VD(S_L)$ before an edge of $VD(S_R)$ or vice versa, or it intersects neither an edge of $VD(S_L)$ nor an edge of $VD(S_R)$. In the latter case, $L = l_m$ and we are done.

In the former case assume w.l.o.g. that L intersects an edge, say e, of $VD(S_L)$ before (or simultaneously with) an edge of $VD(S_R)$. Call the point of intersection z. Point z lies on the boundary of the Voronoi region of x_i with respect to S_L. Thus there is $x_i' \in S_L$ such that $dist(x_i, z) = dist(x_i', z) = dist(x_j, z)$. In other words, z is a vertex of the Voronoi Diagram of set S as shown in the diagram of Figure 34.

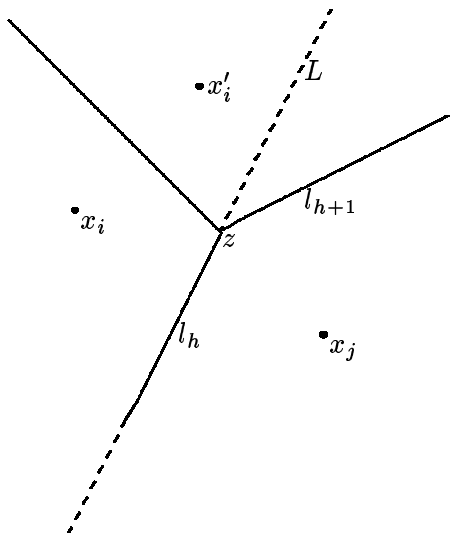


Figure 34. A bend of P at z

In vertex z three edges of $VD(S)$ meet, namely l_h, the perpendicular bisector of x_i and x_j, e, the perpendicular bisector of x_i and x_i', and l_{h+1}. Thus l_{h+1} is the perpendicular bisector of x_i' and x_j. We summarize the discussion in the Program 5.

Before we can analyze Program 5 we need to be more specific about the representation of the Voronoi Diagram. We postulate the following representation:

a) each face of the Voronoi Diagram, i.e., each Voronoi region is given by the doubly-linked list of its boundary edges. Also the point of S to which the region belongs and the representation of the region are linked and the two occurrences of every edge are linked.

b) the boundary of the convex hull of S is given by its hierarchical representation.

Part b) of the representation is only needed for the algorithm above, part a) is the genuine representation of the diagram. We have seen in Section 8.2 that the lower tangent T_L, points x and y in line (1) and the hierarchical representation of $BCH(S)$ can be constructed in time $O(\log n)$ from the hierarchical representation of $BCH(S_L)$ and $BCH(S_R)$. Thus line (1) takes time $O(\log n)$.

Let us consider **while**-loop (2)–(10) next. From the considerations above it is clear that at most n iterations of the loop can occur. The test in line (2) is carried out as follows.

When line (2) is executed we "move" within the Voronoi regions $VR_L(x)$ of point $x \in S_L$ with respect to S_L and $VR_R(y)$ of point $y \in S_R$ with respect to S_R. Region $VR_L(x)$ (and $VR_R(y)$) is represented by a circular list. Find the lowest (smallest y-coordinate) point on that list. This is done at most once for every point and hence takes total time $O(n)$ by Lemma 2. Next associate two pointers with the list, one with the left part and one with the right part (cf. Figure 35).

We use these pointers to implement a scan line which scans $VR_L(x)$ and $VR_R(y)$ simultaneously from bottom to top and finds the first (= lowest, since

```
(1)   h := 1;
      Let T_L be the "lower" tangent of BCH(S_L) and BCH(S_R),
      T_L connects x ∈ S_L and y ∈ S_R, say;
      Let L be an infinite ray starting at −∞ and being
      the perpendicular bisector of x and y.
      Furthermore, let line segment l_1 on L start in −∞.
(2)   while L intersects either VD(S_L) or VD(S_R)
(3)   do if L intersects an edge, say e of VD(S_L) not after an edge of VD(S_R)
(4)      then let z be the point of intersection;
(5)           terminate l_h in z;
(6)           let e be the perpendicular bisector of x ∈ S_L and x' ∈ S_L;
(7)           x := x'; h := h + 1;
(8)           let L be the infinite ray, starting in z and being part of the

              perpendicular bisector of x and y, and extending towards +∞;
(9)           let l_h be a line segment on L starting in z
         else
              ⋮
              symmetric to then-case
              ⋮
         fi
(10)  od.
```

Program 5

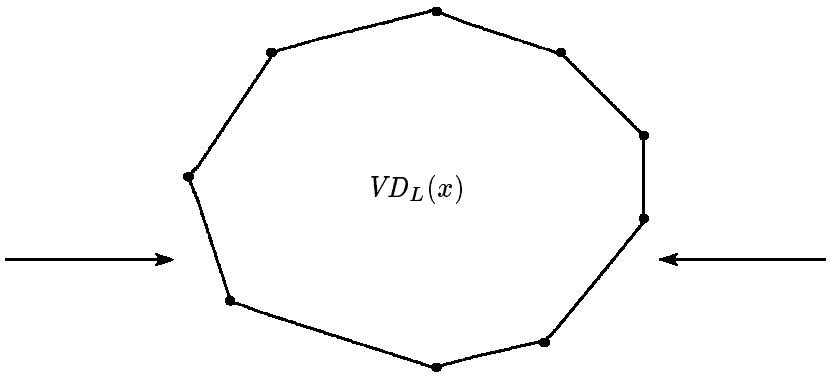


Figure 35. Searching upwards a region boundary in parallel

P is monotonic) intersection of L with any edge of $VR_L(x)$ and $VR_R(y)$. This process is very similar to merging four sorted lists. The time required to find an intersection is clearly proportional to the number of edges discarded. Since P is monotone and hence no back-tracking is ever needed every edge is discarded only once. Once we have determined the edge of intersection we only have to follow

the pointer to the other copy of the edge in order to find point x' in line (6). The process is then continued with x' instead of x in the next iteration of the loop. We summarize in

Lemma 6. *Voronoi Diagram $VD(S)$ can be constructed from $VD(S_L)$ and $VD(S_R)$ in time $O(n)$.*

Proof: We argued above that the cost of one execution of the loop body is proportional to the number of edges discarded. Also every edge is discarded at most once and the number of iterations of the loop is at most n. Thus line P can be constructed in time $O(n)$ from $VD(S_L)$ and $VD(S_R)$. Once line P is found it is easy to construct $VD(S)$ from $VD(S_L)$ and $VD(S_R)$ in linear time. In fact, the construction is easily incorporated into Program 5. The only change required is to update $VR(x)$ and $VR(y)$ in the loop body by throwing away some of their edges and to add l_h as a new edge. We leave the simple details to the reader. ∎

It is now easy to complete the proof of Theorem 1. By Lemma 6 we have the following recurrence for the time $T(n)$ required to construct the Voronoi Diagram for a set of n points.

$$T(1) = O(1)$$
$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n)$$

Thus $T(n) = O(n \log n)$ as claimed. ∎

How good is our algorithm for constructing Voronoi Diagrams? Is it even optimal? The following argument shows that constructing Voronoi Diagrams is at least as hard as sorting and hence that the algorithm above is optimal with respect to a wide range of computational models. Consider $S \subseteq \mathbb{R}^2$ where S consists of $n+1$ points, the origin and n points on the unit circle. Then the Voronoi Diagram for S is an n-gon containing the origin and n rays emanating from the vertices of the n-gon. The rays sort the n points on the unit circle in an obvious way. Thus sorting n points
?? on the unit circle by angle is no easier than constructing the Voronoi Diagram.

We close this section with a brief discussion on updating Voronoi Diagrams. Suppose that we have computed the Voronoi Diagram $VD(S)$ and we either want to delete $x \in S$ or we want to add a point y to S. In the latter case we also assume that $x \in S$ with $y \in VR(x)$ is given. We will discuss methods for finding y in the next section.

A worst case bound for the complexity of insertions and deletions is given by the theory of order decomposable problems. Let $\leq$ be the lexicographical order on $\mathbb{R}^2$. Then the problem of constructing the Voronoi Diagram is order-decomposable with respect to ordering $\leq$ with merging time $C(n) = O(n)$. Thus insertions and deletions take time $O(n)$ in the worst case by the results of Section 7.1.3. Since the Voronoi Diagram may be changed drastically by an insertion or deletion this bound cannot be improved.

However, on the average one can do much better. Exercises 14 and 15 discuss algorithms for updating Voronoi Diagrams whose running time is bounded by $O(s)$ and $O(s \log s)$ respectively, where s is the size of the change of the diagram.

8.3.2. Searching Planar Subdivisions

A Voronoi Diagram is a partition of the plane into polygonal regions some of which are unbounded. In this section we show how to search Voronoi Diagrams in logarithmic time, i.e., we describe two data structures which given $y \in \mathbb{R}^2$ allow to find $x \in S$ with $dist(y, x)$ minimal in logarithmic time. Moreover, the data structures can be constructed in linear time and use linear space.

In searching Voronoi Diagrams, the Voronoi regions are the regions of "constant answer", i.e., $dist(y, x) = \min\{dist(y, x);\ x \in S\}$ if and only if $y \in VR(x)$. In other words, x is the nearest point to y among all points in S iff $y \in VR(x)$. Thus in some sense the Voronoi Diagram is a method of tabulating the answers to all nearest neighbor searches. Of course, we still have to describe how to search the table efficiently. That is the purpose of this section. More generally, the methods to be described can be used in the following situation. Suppose that $f : \mathbb{R}^2 \to T$ for some set T assumes only finitely many different values and that for each $t \in T$, $f^{-1}(t)$ is a polygonal region $R_t \subseteq \mathbb{R}^2$. Then $\{R_t;\ t \in T\}$ is a subdivision of $\mathbb{R}^2$. Assume further that the total number of edges of all polygonal regions R_t, $t \in T$, is m. Then the data structures to be described allow us to compute f in time $O(\log m)$. Moreover, the data structures can be constructed in time $O(m \log m)$ and require space $O(m)$. In the Voronoi Diagram searching problem we have $T = S$ and $f(y) = x$ where $dist(x, y) = dist(y, S)$. Other examples are $T = S$ and $f(y) = x$ where $dist(x, y) \geq dist(z, y)$ for all $z \in S$ (furthest neighbor) or $T = S \times S$ and $f(y) = (x, x')$ where $dist(x, y) \leq dist(x', y) \leq dist(z, y)$ for all $z \in S$, $z \neq x, x'$ (two nearest neighbors). In both examples, it is clear that the regions of constant answer are convex polygonal regions since these regions can be written as intersection of half-spaces. It is not clear however whether these regions can be computed efficiently. We refer the reader to the exercises.

A **planar subdivision** is a straight line embedding $\hat{G}$ of a planar graph G. An embedding is straight line if all edges of G are embedded as straight line segments (cf. Figure 36).

In this section we describe three solutions to the planar subdivision searching problem; several other solutions are discussed in the exercises. The first two solutions have logarithmic search time but can deal only with static subdivisions, the third solution has search time $O((\log n)^2)$ but can deal with dynamic subdivisions. We present three solutions because each solution illustrates an important algorithmic paradigm. In the first solution the planar subdivision is successively simplified by the removal of a large independent set of vertices of small degree. The existence of such a set is guaranteed in every planar graph (Lemma 8 below). The method of removing large independent sets of vertices can also be successfully used

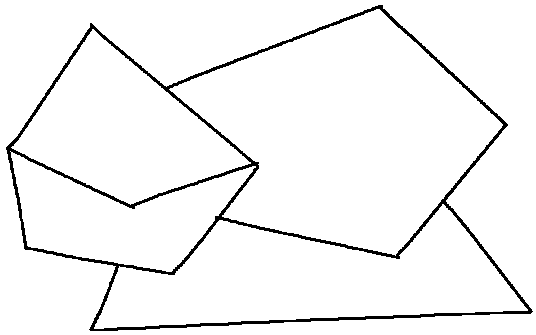


Figure 36. A planar subdivision

for obtaining hierarchical representations of (convex) polyhedra, cf. Exercise 2 and Section 8.4.3. The second solution is based on path decompositions of planar subdivisions. It is computationally superior to the first solution in the sense that the constants involved in the O-expressions are much smaller. Path decomposition will be used again in Section 8.5.1.4 on skeleton structures. Finally, the third solution combines the ideas of path decomposition and weight-balanced trees in order to obtain a dynamic solution.

We close this introduction with a short discussion of a more general and of a more restricted searching problem. A planar subdivision is **simple** if all (including the infinite) faces of $\hat{G}$ are triangles. A **generalized** planar subdivision is a planar subdivision together with a set of pairwise non-intersecting rays which start at the nodes on the boundary of the infinite face.

Figure 37 shows a generalized planar subdivision. If the dashed edges are added then we obtain a generalized subdivision with a simple core. The following lemma shows that it suffices to solve simple searching problems.

Lemma 7. *If the searching problem for simple planar subdivisions with n edges can be solved with search time $O(\log n)$, preprocessing time $O(n)$ and space $O(n)$ then the searching problem for generalized subdivisions with n edges can be solved with search time $O(\log n)$, preprocessing time $O(n \log n)$ and space $O(n)$. If all faces of the generalized subdivision are convex then preprocessing time $O(n)$ suffices.*

Proof: Let $\hat{G}$ be a generalized planar subdivision with n edges and $m \leq n$ vertices. We enclose the finite part of $\hat{G}$ in two large component triangles T_1 and T_2 (cf. Figure 37) such that T_2 contains T_1. Triangles T_2 and T_1 together with the part of $\hat{G}$ which lies in the interior of T_1 defines a planar subdivision. We turn this subdivision into a simple subdivision $\hat{G}'$ by triangulating all its faces. Note that every infinite ray defines an edge of $\hat{G}'$ which has one vertex on triangle T_1. However, no part of the infinite rays outside T_1 belongs to $\hat{G}'$. Thus triangulation actually produces a subdivision in which the outer face is also a triangle, i.e., $\hat{G}'$ is simple. The simple subdivision $\hat{G}'$ has clearly $O(n)$ edges and can be obtained in time $O(n \log n)$ using the methods of Section 8.4.2. If all faces of $\hat{G}$ are convex then $\hat{G}'$ can be obtained in time $O(n)$.

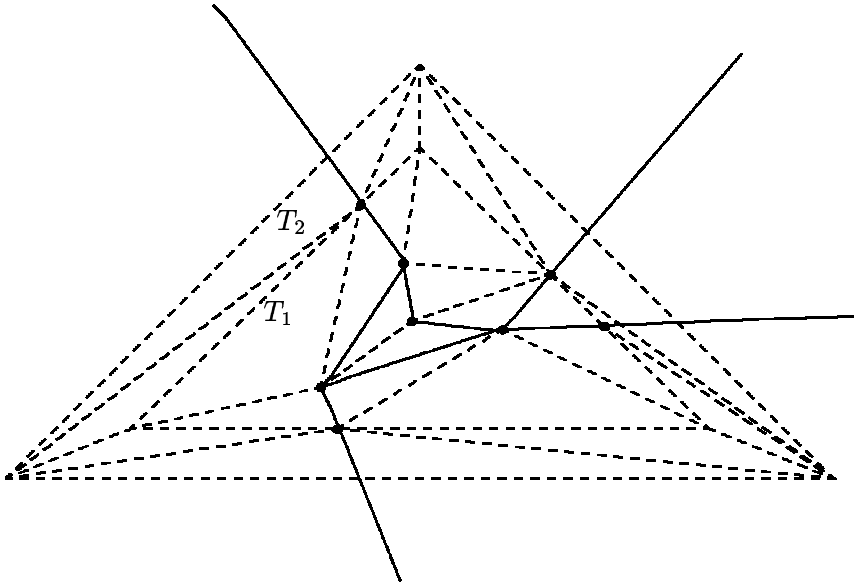


Figure 37. A generalized planar subdivision

We can now search for point y in $\hat{G}$ as follows. In time $O(1)$ we decide whether y lies inside or outside T_1. If y lies inside T_1 then we use the efficient solution for simple subdivision $\hat{G}'$ which exists by assumption. If y lies outside T_1 then we locate y in logarithmic time by binary search on the infinite rays as follows. We divide the set of rays into two disjoint sets, the right-going and the left-going rays. (A ray is right-going if it extends towards infinity in the direction of the positive x-axis, cf. Fig. 38).

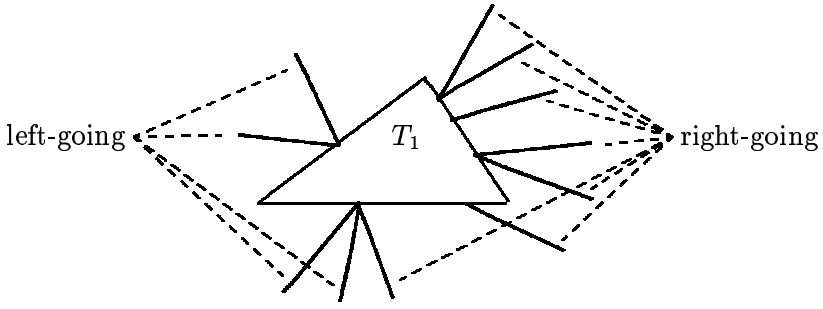


Figure 38. left- and right-going edges

For the rays in each class, a simple comparison decides in time $O(1)$ whether y lies above or below the ray and hence binary search locates point y in time $O(\log n)$.

Thus in either case (y inside or outside T_1) we can locate y in time $O(\log n)$. Furthermore, the space requirement is clearly $O(n+$ space of solution for $\hat{G}')$. Also the preprocessing time is $O(n \log n+$ preprocessing time for $\hat{G}')$. ∎

8.3.2.1.. Removal of Large Independent Sets

Our first solution uses divide-and-conquer by the removal of large independent sets. The same technique is applicable to a number of other problems in computational geometry, most notably to problems concerning convex polyhedra, cf. Exercise 2 and Section 4.3. In these applications we uses the techniques of removing large independent sets of nodes in order to obtain "simpler" versions ov convex polyhedra.

The idea is as follows. Let $\hat{G}$ be a simple planar subdivision with n vertices and hence $3n-6$ edges. Since $\hat{G}$ is a planar graph it has a large (at least size $c \cdot n$ for some constant $c > 0$) subset I of vertices which are pairwise independent and have small degree, say at most 9. Two vertices are independent if they are not connected by an edge. Removal of the vertices in I from $\hat{G}$ yields a subdivision $\hat{G}'$ with at most $(1-c)n$ vertices. The faces of $\hat{G}'$ are m-gons with $m \leq 9$ and different vertices in I lie in different faces in $\hat{G}'$. We turn $\hat{G}'$ into a simple subdivision $\tilde{G}$ by triangulating all faces of $\hat{G}'$. In the following example (see next page), nodes x, y, and z form an independent set. Removal of $\{x, y, z\}$ yields $\hat{G}'$; triangulating the three non-triangular faces yields $\tilde{G}$. The newly added edges are drawn dashed. The faces of $\tilde{G}$ which correspond to the same node in $I = \{x, y, z\}$ are labelled by the same character.

Let $\tilde{D}$ be a search structure for $\tilde{G}$ which we assume to exist inductively. We can construct $\tilde{D}$ by applying the techniquer recursively or by some other means. In our example $\tilde{D}$ has 10 leaves corresponding to the 10 faces of $\tilde{G}$. We obtain a search structure $\hat{D}'$ for $\hat{G}'$ from $\tilde{D}$ by simply combining all the leaves corresponding to the subfaces of a face of $\hat{G}'$ to a single leaf. In our example, we combine faces A_1, A_2 and A_3 to face A. The search structure $\hat{D}'$ is now easily turned into a search structure $\hat{D}$ for $\hat{G}$. Consider a face, say F, of $\hat{G}'$. If there is no vertex in I which lies in F, then F is also a face of $\hat{G}$ and there is nothing to do. In our example, this case arises for the infinite face. If there is a vertex, say x, in I which lies in F, then we observe first that this vertex is unique since I is an independent set of vertices. Vertex x and the edges, say $e_1, \ldots, e_k$ $(k \leq 9)$, incident to x subdivide F into faces $F_1, \ldots, F_k$ of $\tilde{G}$. Hence we obtain $\hat{D}$ from $\hat{D}'$ by replacing the leaf corresponding to F by a program which locates a query point with respect to the "star" defined by vertex x and the edges incident to x. Since x has degree at most 9 this decision takes time $O(1)$ and hence the cost of locating a point in $\hat{G}$ is only a constant more than the cost of locating a point in $\tilde{G}$. Since $\tilde{G}$ has at most $(1-c)n$ vertices we should obtain logarithmic search time in this way. In our example, face A of $\hat{D}'$ has subfaces a, b, c, d of $\hat{G}$.

Once we have determined that a query point y lies in A a simple comparison of y with the edges incident to x also determines the face of $\hat{G}$ which contains y.

It remains to fill in the details. We first show that there are always large independent sets of nodes of small degree.

Lemma 8. *Let $G = (V, E)$ be a planar graph with $n = |V|$ nodes, minimum degree 3, and let $V' \subseteq V$. Then there is an independent set $I \subseteq V - V'$ of nodes of*

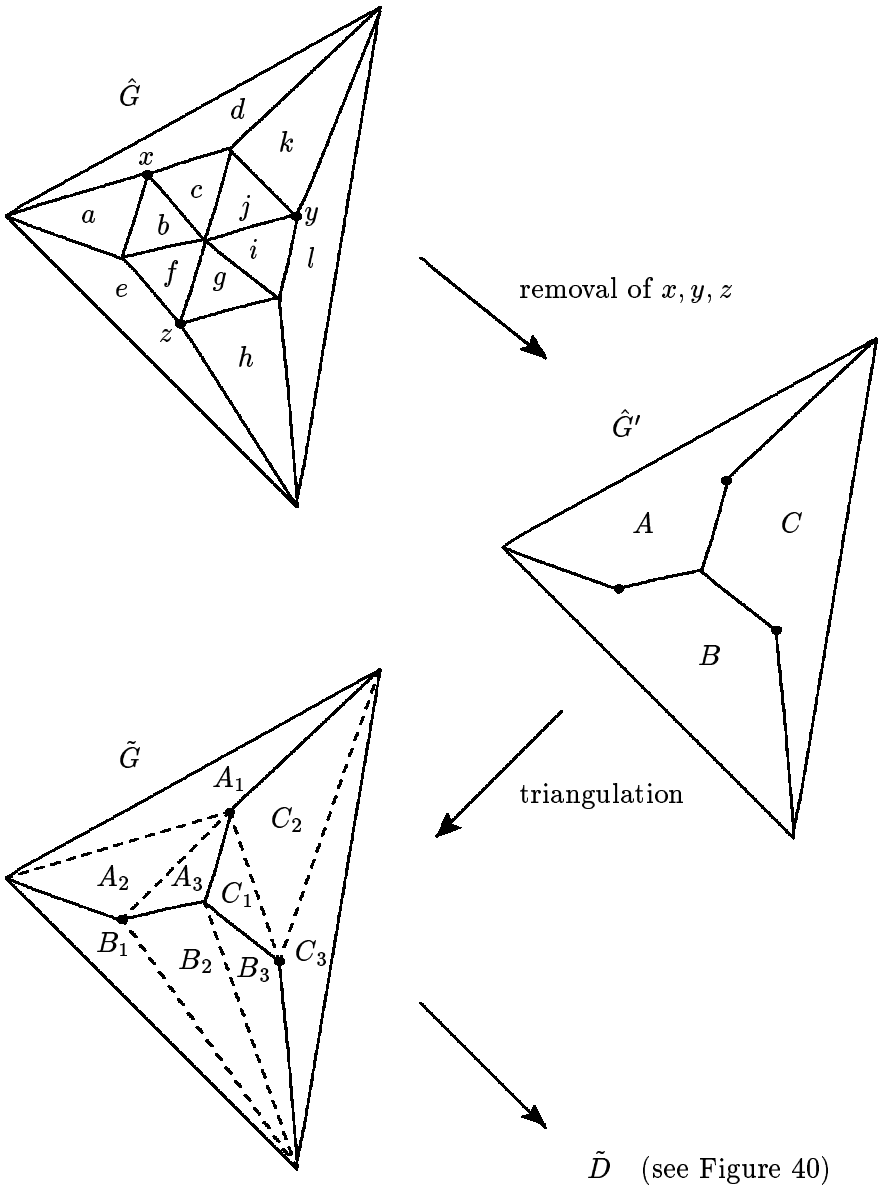


Figure 39.

degree at most 9 which has size at least $(4 \cdot n + 12 - 7 \cdot |V|)/70$. *$I$ is independent if* $v, w \in I$ *implies* $(v, w) \notin E$. *Moreover, I can be found in time* $O(n)$.

Proof: Let V'' be the set of nodes of G which have degree at most 9, and let $x = |V''|$. Since G is planar the number of edges of G is at most $3 \cdot n - 6$. Also there are exactly $n - x$ nodes of degree 10 or more and every other node has degree at least three. Thus $3 \cdot x + 10 \cdot (n - x)/4) \leq 3 \cdot n - 6$ or $x \geq (4 \cdot n + 12)/7$.

Consider the subgraph induced by $V'' - V'$. It has $x - |V'|$ nodes and every

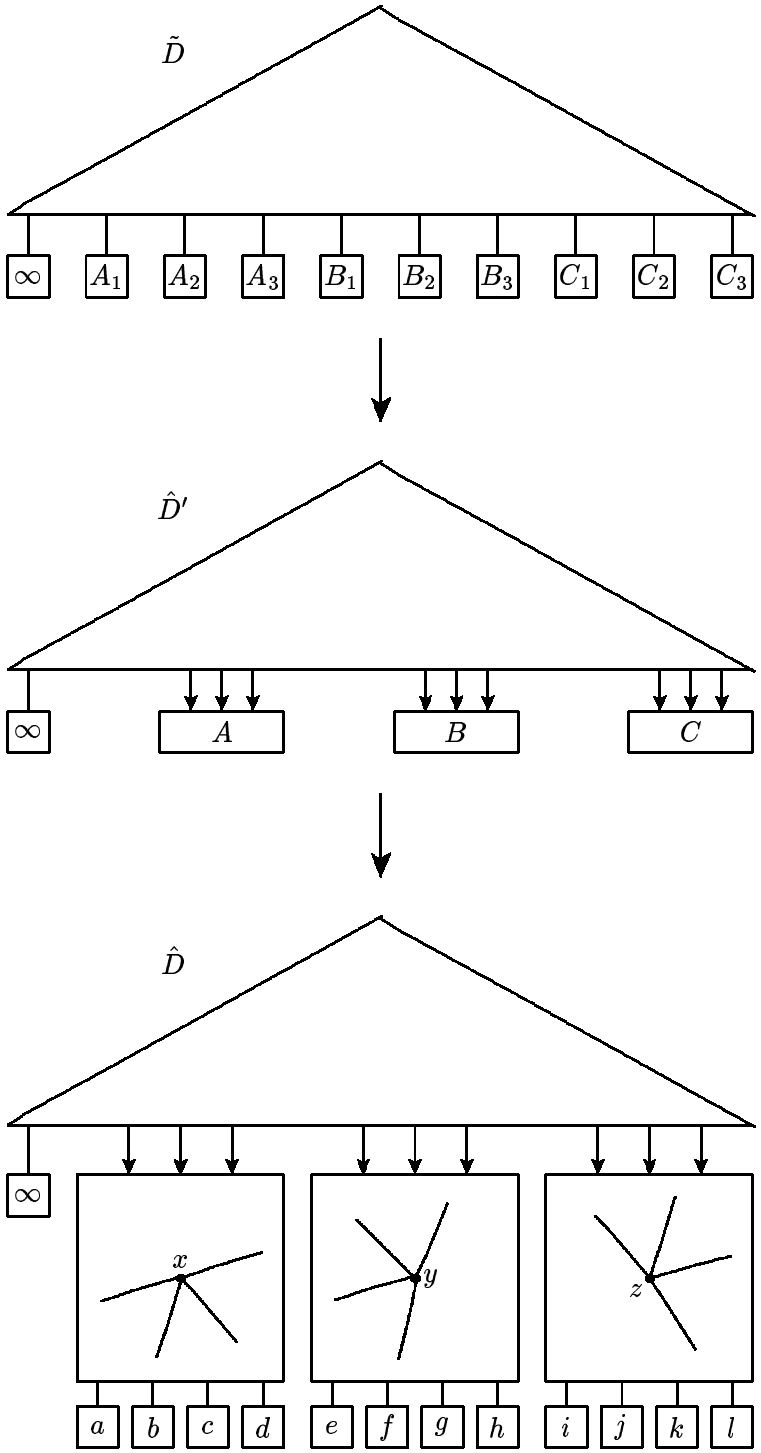

$\tilde{D}$
∞
A_1
A_2
A_3
B_1
B_2
B_3
C_1
C_2
C_3
$\hat{D}'$
∞
A
B
C
$\hat{D}$
∞
x
y
z
a
b
c
d
e
f
g
h
i
j
k
l

Figure 40.

node has degree at most 9. Thus it can be colored using at most 10 colors such that adjacent nodes are colored differently. Moreover, such a coloring can be found in time $O(n)$ as follows. Assume that we use colors $1, \ldots, 10$. Consider the nodes in some order. When node v is considered, color it with the lowest numbered color not yet used on a neighbor of v. This algorithm clearly runs in time $O(n)$ and uses at most 10 colors. Finally observe, that there must be one color class containing at least $(x - |V'|)/10 \geq (4 \cdot n + 12 - 7 \cdot |V'|)/70$ nodes. ∎

We are now ready to prove

Theorem 2. *Let $\hat{G}$ be a simple planar subdivision with n vertices. Then the searching problem with respect to $\hat{G}$ can be solved with search time $O(\log n)$, space requirement $O(n)$, and preprocessing time $O(n)$.*

Proof: We use induction on the number n of vertices in $\hat{G}$. For $n \leq 100$ the claims are certainly true by appropriate choice of the constants in the bounds.

Assume now that $n \geq 100$. Let I be an independent set of nodes none of which has degree 10 or more and none of which lies on the boundary of the infinite face. By Lemma 8 (let V' be the vertices on the boundary of the infinite face which is triangle and hence $|V'| = 3$) we can find such a set I with $|I| \geq (4 \cdot n - 9)/70$ in time $O(n)$.

Removal of set I and the edges incident to vertices in I leaves us with a planar subdivision $\hat{G}'$ with at most $66 \cdot n/70 + 1 < n$ vertices. Note that the faces of $\hat{G}'$ are m-gons with $3 \leq m \leq 9$. Every face of $\hat{G}'$ can be triangulated in time $O(1)$ using $m - 3$ edges. Thus we can turn $\hat{G}'$ into a simple planar subdivision $\tilde{G}$ with at most $66 \cdot n/70 + 1$ vertices in time $O(n)$.

Applying the method recursively to $\tilde{G}$, we obtain a search structure $\tilde{D}$ for $\tilde{G}$ which we then turn into a search structure $\hat{D}$ for $\hat{G}$ as described above. We can clearly obtain $\hat{D}$ from $\tilde{D}$ in time $O(n)$ using additional space $O(n)$ and increasing the depth by $O(1)$. Thus we obtain the following recurrences for the depth $d(n)$ of the search structure, the construction time $T(n)$, and the space requirement $S(n)$.

$$d(n) = O(1) \qquad \text{for } n \leq 100,$$
$$d(n) = O(1) + \max\{d(n');\ n' \leq 66 \cdot n/70 + 1\} \text{for } n > 100,$$

and

$$T(n) = O(1) \qquad \text{for } n \leq 100,$$
$$T(n) = \max\{O(n) + T(n');\ n' \leq 66 \cdot n/70 + 1\} \text{for } n > 100,$$

and similarly for $S(n)$. Thus $d(n) = O(\log n)$, $T(n) = O(n)$ and $S(n) = O(n)$ as a simple inductive argument shows. ∎

Bemerkung an Kurt: [Benutze 5-Färbung um Konstante zu verbessern; mache Bemerkung, daß die folgenden Algorithmen schneller sind; füge noch die persistency-Lösung hinzu]

8.3.2.2. Path Decomposition

We will now present a second solution to the planar subdivision searching problem. It is based on path decomposition of planar subdivisions. We will use path decomposition again in Section 5.1.4 where we will show how to greatly extend the power of plane sweep algorithms. The present section is organized as follows. We first introduce path decompositions and derive a suboptimal solution with $O((\log n)^2)$ search time and $O(n^2)$ storage space from it. We will then show how to reduce the space requirement to $O(n)$ by removing redundance. Finally, we reduce search time to $O(\log n)$ by a clever combination of binary search in x- and y-direction.

Let $\hat{G}$ be a simple planar subdivision. We assume w.l.o.g. that no edge of $\hat{G}$ is horizontal. Let $s(t)$ be the vertex of $\hat{G}$ with maximal (minimal) y-coordinate. an **s-t path** is a sequence of $v_0, \ldots, v_m$ of vertices such that $v_0 = s$, $v_m = t$, and (v_i, v_{i+1}) is an edge of $\hat{G}$ for $0 \leq i < m$. A path $v_0, \ldots, v_m$ is **y-monotone** if $y(v_i) \geq y(v_{i+1})$ for $0 \leq i < m$, where $y(v_i)$ is the y-coordinate of vertex v_i. Throughout this section we will use path to mean y-monotone (s,t)-path.

The significance of y-monotone paths stems from the following simple observation. A y-monotone path divides the strip between the horizontal lines through s and t into two parts. Given a point q in the strip, one can clearly locate the part containing q in time $O(\log m)$ by binary search for $y(p)$ in the ordered set $y(v_0), \ldots, y(v_m)$ followed by a comparison of p with a single edge of the path.

A sequence $P_1, \ldots, P_k$ of paths is a path decomposition of simple planar subdivision $\hat{G}$ if

1) every P_i is a y-monotone (s,t)-path
2) every edge of $\hat{G}$ belongs to at least one path
3) if $i < j$ then every horizontal line L intersects P_i to the left or at the same point as P_j.

For the sequel, we will always assume that $k = 2^{d+k} - 1$ for some integer d. This can always be achieved by duplicating path P_k a few times. Figure 42 shows a simple planar subdivision and a path decomposition of it.

A path decomposition $P_1, \ldots, P_k$ of $\hat{G}$ gives rise to an $O((\log k) \cdot (\log n))$ search algorithm immediately; we show below that $k \leq 4 \cdot n$ is always possible. We arrange the paths in a complete binary tree, which we call super-tree (see Figure), 41of depth d with $k = 2^{d+1} - 1$ nodes. We number the nodes inorder such that path P_i is associated with node i. In each node of the super-tree we use the binary search algorithm described above to determine the position of the query point with respect to the path associated with that node. More precisely, we determine also the edge of the path which is intersected by the horizontal line (call it L) through q. In this

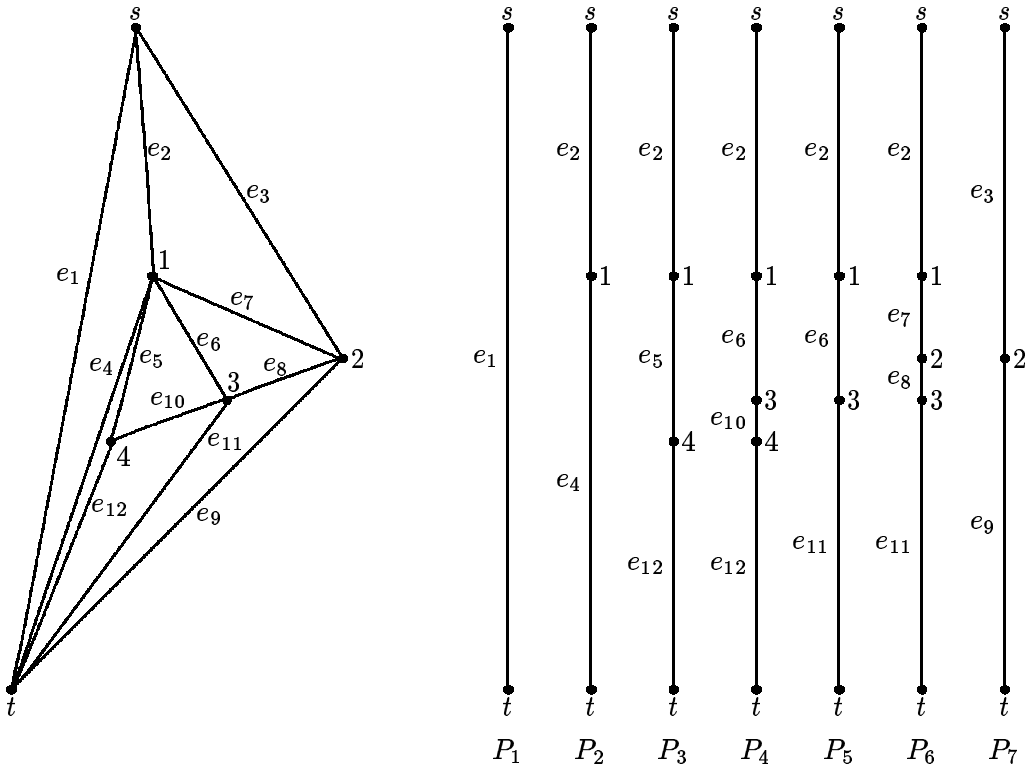


Figure 41. Super-tree

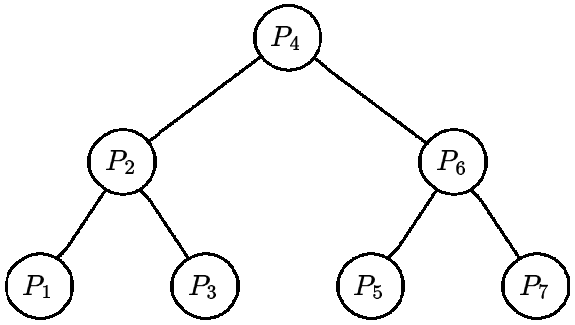


Figure 42. planar subdivision and path decomposition

way, we determine in time $O((\log n)^2)$ and index i and edges e of P_i and e' of P_{i+1} such that q lies between edge e of P_i and e' of P_{i+1}. This pair of edges determines a unique face of $\hat{G}$.

It is interesting to observe that the solution to subdivision searching based on path decomposition is nothing but twofold application of binary search. Note that a path decomposition decomposes the planar subdivision into regions (between adjacent paths) which are ordered in x-direction in a natural way. We can therefore use binary search in x-direction to locate a query point with respect to these regions.

Each step in this binary search process requires us to determine the location of the query point with respect to a path. Since paths are y-monotone we can use binary search in y-direction for that purpose.

The basic algorithm described above has several shortcommings; it does not achieve search time $O(\log n)$ and it uses space $O(n^2)$ since it requires to store $O(n)$ paths of length $O(n)$ each. We show how to reduce the storage requirement first.

Let e be an edge of a simple planar subdivision $\hat{G}$ and let $P_1, \ldots P_k$ be a path decomposition. Then, if e belongs to P_i and P_j, $i < j$, then e also belongs to P_l for all l, $i \leq l < j$. This follows immediately from property (3) of path decompositions. We can therefore describe a path decomposition in linear space by listing for each edge e of $\hat{G}$ a pair $(L(e), R(e))$ of integers such that e belongs to P_j iff $L(e) \leq j \leq R(e)$. In our example, the values of $L(e)$ and $R(e)$ are given by the table in Figure 43. The significance of entry $Pos(e)$ is explained below. We call this representation of a path decomposition its implicit representation.

	L	R	Pos
e_1	1	1	1
e_2	2	6	4
e_3	7	7	7
e_4	2	2	2
e_5	3	3	3
e_6	4	5	4
e_7	6	6	6
e_8	6	6	6
e_9	7	7	7
e_{10}	4	4	4
e_{11}	5	6	6
e_{12}	3	4	4

Figure 43.

Lemma 9. *The implicit representation of a path decomposition can be constructed in linear time.*

Proof: We construct the paths from left to right. Suppose that we constructed (the implicit representation of) path P and also all positions on P where we can "move" the path to the right. We call these positions the **candidates** of path P. Let $P = v_0, \ldots, v_m$. An edge (v_i, v_{i+1}) is a candidate of P and edges (v_i, p), (p, v_{i+1}) are its **substitutes** if v_i, p, v_{i+1} is a face of $\hat{G}$, p is to the right of P and $y(v_i) > y(p) > y(v_{i+1})$. A pair $(v_i, v_{i+1}), (v_{i+1}, v_{i+2})$ of consecutive edges of P is a candidate and edge (v_i, v_{i+2}) is its substitute if v_i, v_{i+1}, v_{i+2} is a face of $\hat{G}$ which lies to the right of P. Figure 44 illustrates both notations.

We are now ready for the algorithm.

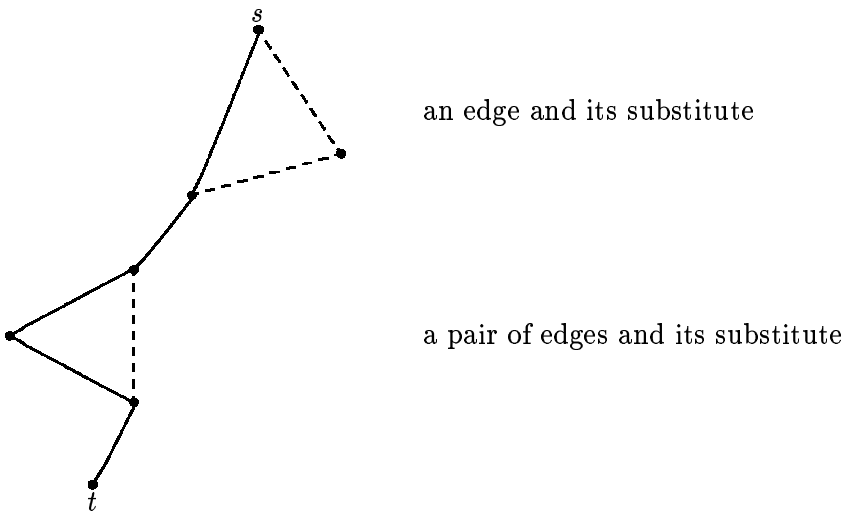


Figure 44.

```
(1)   P ← leftmost (s,t)-path; count ← 1;
(2)   for all edges e of P do L(e) ← 1 od;
(3)   while P has a non-empty set of candidates
(4)   do for all edges e in the set of candidates
(5)      do R(e) ← count;
(6)         replace e by its substitute(s) e' (and e'');
(7)         L(e) ← (L(e'') ←)count + 1
(8)      od;
(9)      count ← count + 1;
(10)  od
(11)  for all edges e of P do R(e) ← count od.
```

Program 6

For the correctness of this algorithm it suffices to show that every path P different from the rightmost path has a non-empty set of candidates. Assume otherwise. Then every face to the right of P has at most one edge in common with P. Let $P = v_0, \ldots, v_m$ and let i be minimal such that $y(p) > y(v_{i+1})$ where p is the third vertex of the face which has edge (v_i, v_{i+1}) on its boundary and is to the right of P. The existence of index i can be seen as follows. Let $k > 0$ be minimal such that v_k lies on the rightmost path and v_{k-1} does not. Then edge (v_{k-1}, v_k) has the property stated above since the rightmost path is y-monotone. We claim that $y(v_i) > y(p)$ and hence edge (v_i, v_{i+1}) is a candidate. Assume otherwise, i.e., $y(v_i) < y(p)$. Then $i > 0$ since $s = v_0$ is the vertex with maximal y-coordinate. Also, we must have $y(v_i) < y(p')$ where p' is the third vertex of the face (to the right of P) determined by edge (v_{i-1}, v_i). This contradicts the choice of i and hence establishes correctness.

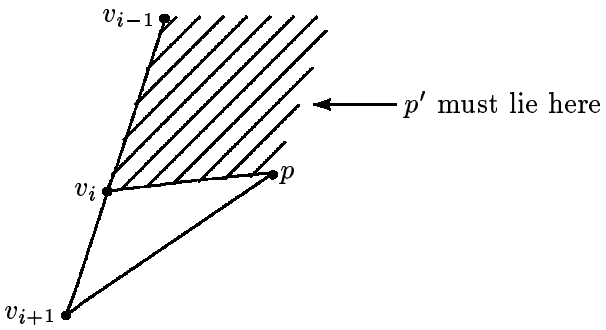


Figure 45.

It remains to estimate running time. Path P is represented as a doubly linked list. Also, the set of candidates of P is represented as a linked list. It is then easy to execute lines (5)–(8) in time $O(1)$ and also to check whether one of the substitutes is a candidate for the new path. Note that every candidate of the new path must contain an edge added in line (6). Thus one execution of lines (4)–(10) takes time proportional to the number of edges for which $R(e)$ is defined during that execution. Since $R(e)$ is defined only once for each edge the time bound follows.

We have now established that the implicit representation of some path decomposition $P_1, \ldots, P_k$, $k = 2^{d+1} - 1 \leq 4 \cdot n$ (recall that we replicate P_k in order to bring k into that special form) of a simple planar subdivision can be constructed in linear time. We will next use this fact to reduce the storage requirement of the search structure to $O(n)$ by storing every edge of $\hat{G}$ only once. More precisely, we store edge e of $\hat{G}$ only in the highest node of the super-tree whose associated path contains e. Using functions $L(e)$ and $R(e)$ it is easy to define that node, call it $Pos(e)$. Node $Pos(e)$ is the lowest common ancestor of nodes $L(e)$ and $R(e)$ of the super-tree. Then path $P_{Pos(e)}$ contains e (since $L(e) \leq Pos(e) \leq R(e)$) and no path associated with an ancestor of $Pos(e)$ contains e. Another way of characterizing $Pos(e)$ is as follows. If $L(e) = R(e)$ then clearly $Pos(e) = L(e)$. If $L(e) < R(e)$ then let $\ldots \alpha_2\alpha_1\alpha_0$ ($\ldots \beta_2\beta_1\beta_0$) be the binary representation of $L(e)$ ($R(e)$) and let j be maximal such that $\alpha_j \neq \beta_j$. Then $\alpha_j = 0$, $\beta_j = 1$ and $\ldots \alpha_{j+2}\alpha_{j+1}\gamma 0 \ldots 0$ is the binary representation of $Pos(e)$ where $\gamma = 0$ if $\alpha_j = \alpha_{j-1} = \cdots = \alpha_0 = 0$ and $\gamma = 1$ otherwise.

We are now ready for the definition of the **reduced search structure**. In the reduced search structure node i contains a balanced search tree T_i for the y-coordinates of the endpoints of all edges e with $Pos(e) = i$. Also, the nodes of T_i corresponding to the lower endpoint of edge e contains a pointer to edge e. In our example we have

Lemma 10.

a) *The search time in the reduced search structure is* $O((\log n)^2)$

b) *The reduced search structure requires storage space* $O(n)$ *and can be constructed in time* $O(n)$.

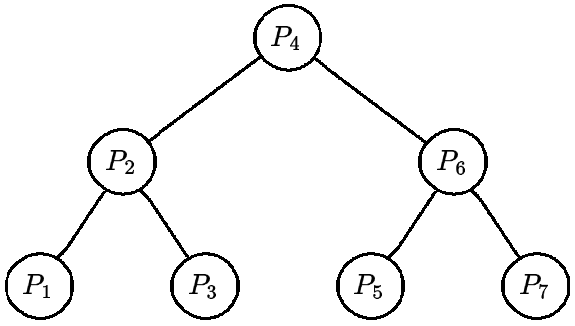


Figure 46.

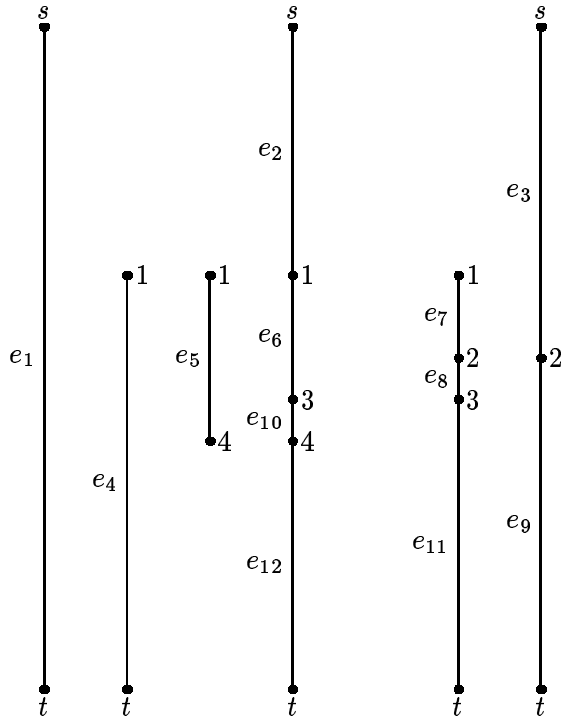


Figure 47.

Proof: a) For the following algorithm we assume $P_0 = P_1$, $P_{k+1} = P_k$ and $k = 2^{d+1} - 1$ for some d. Furthermore, we assume that query point q lies between P_0 and P_{k+1}, a fact which is easily checked in time $O(1)$. Recall that P_0 and P_{k+1} are the boundary of the infinite face which is a triangle. Finally, L is a horizontal line through q.

The correctness of this algorithm is almost immediate. If either edge e_l or edge e_r belongs to P_m then the step in the binary search is trivial. If neither e_l nor e_r belongs to P_m then let e be the edge of P which is intersected by L. We must have $Pos(e) = m$ since P_m runs between P_l and P_r, $e \neq e_l$, $e \neq e_r$. Thus the correct

```
(1)   l ← 0; r ← k + 1;
(2)   e_l (e_r) ← edge of P_l (P_r) intersected by L;
(3)   while r > l + 1
(4)   do co point q lies between P_l and P_r and line L
           intersects edge e_l (e_r) of P_l (P_r); oc
(5)      m ← ⌈(l + r)/2⌉;
(6)      if R(e_l) ≥ m
(7)      then l ← m
(8)      else if L(e_r) ≤ m
(9)           then r ← m
(10)          else find the edge of P_m intersected by L by binary search
                   and redefine l, r, e_l and e_r appropriately
(11)          fi
(12)     fi
(13)  od.
```

Program 7

decision is made in line (10). This proves part a).

b) The space bound is obvious because every edge of $\hat{G}$ is stored exactly once. It remains to argue the $O(n)$ bound on preprocessing time. We show an $O(n \log n)$ bound first (which is probably good enough for practical purposes) and then sketch the linear time solution.

Observe first that $Pos(e)$ can be computed in time $O(\log n)$ by the simple algorithm of Program 8, for every edge e.

```
if L(e) = R(e)
then Pos(e) ← L(e)
else count ← −1; l ← L(e); r ← R(e); flag ← true;
     while l ≠ r
     do count ← count + 1;
        if l is odd then flag ← false fi;
        l ← ⌊l/2⌋; r ← ⌊r/2⌋
     od;
     if flag then Pos(e) ← L(e) else Pos(e) ← (l · 2 + 1)^count fi
fi
```

Program 8

We conclude that array $Pos(e)$ $e \in E$, can be computed in time $O(n \log n)$. Using bucket sort ($Pos(e)$ is the bucket to which edge e is sent; cf. Section 2.2.1) we can compute the set of edges associated with any node of the super-tree. However, we want this set sorted according to y-coordinate. The $O(n \log n)$ cost of

sorting can be overcome as follows. Turn $\hat{G}$ into directed graph by directing all edges downwards, i.e., direct edge (v,w) from v to w iff $y(v) > y(w)$. This takes time $O(n)$. Then sort $\hat{G}$ topologically (cf. Section 4.2) in time $O(n)$; let $Num(e)$ be the number of edge e. Associate pair $(Pos(e), Num(e))$ with edge e and sort these pairs lexicographically by bucket sort in time $O(n)$. In this way, we obtain the edges associated with any node of the super-tree in sorted order according to the y-coordinates of their endpoints. Finally, we have to build a balanced search tree for each node of the super-tree. This clearly takes linear time. In summary, we have shown how to construct the reduced search structure in time $O(n \log n)$.

In order to prove an $O(n)$ bound on the preprocessing time it suffices to compute the array $Pos(e)$, $e \in E$, in time $O(n)$. A more abstract view of the problem is as follows. Given the complete binary tree T with $k = 2^{d+1} - 1$ nodes and pairs (x_i, y_i), $1 \leq i \leq 4 \cdot n$, of nodes compute for each pair its lowest common ancestor $a_i = Lca(x_i, y_i)$. We have $x_i = L(e)$, $y_i = R(e)$ and $a_i = Pos(e)$ for some edge e in the problem of computing array Pos.

We solve the lowest common ancestor problem as follows. In a first step we compute several auxiliary functions on tree T by an ordinary tree traversal (cf. Section 1.5), namely

$$Rthread(v) = \begin{cases} \text{rightmost leaf, which is a descendant of } v & \text{if } v \text{ is not a leaf;} \\ \text{successor of } v \text{ in the inorder traversal of } T & \text{if } v \text{ is a leaf;} \end{cases}$$

$$Rmost(v) = \textbf{if } v \text{ is a leaf } \textbf{then } v \textbf{ else } Rthread(v)$$

and

$$Lra(v) = Rthread(Rmost(v)).$$

Function Lra yields for every node v the lowest ancestor which follows v in the inorder traversal. The symmetric functions $Lthread$, $Lmost$ and Lla are defined similarly. We leave it to the reader to show that all these functions can be computed in time $O(n)$. Using functions $Rmost$ and $Lmost$ one can decide in time $O(1)$ whether node x is an ancestor of node y; namely x is an ancestor of y iff $Lmost(x) \leq Lmost(y) \leq Rmost(y) \leq Rmost(x)$. Thus in time $O(n)$ we can compute a_i for all pairs (x_i, y_i) where x_i is an ancestor of y_i or vice versa. If neither x_i is an ancestor of y_i nor y_i is an ancestor of x_i then $Lca(x_i, y_i) = Lca(Rmost(x_i), Rmost(y_i))$. We may therefore assume w.l.o.g. that the x_i's and y_i's are leaves of T.

Let $d_i = y_i - x_i + 1$ and $ld_i = \lfloor \log(y_i - x_i + 1) \rfloor$. We can compute ld_i, $1 \leq i \leq 4 \cdot n$ in time $O(n)$ by first tabulating function $m \mapsto \lceil \log m \rceil$, $1 \leq m \leq k$, and then using table-lookup. Let C_h be the nodes of height $h-1$ of T and let $Q_h = \{(x_i, y_i);\ ld_i = h\}$ for $h \geq 1$. We can compute sets Q_h ordered in increasing order of the x_i's by creating triples $ld_i, x_i, i)$ and sorting them into lexicographic order by bucket sort. In a next step we compute app_i (approximate lowest common ancestor) where $app_i \in C_{ld_i}$ and app_i is an ancestor of x_i for all i. We can compute app_i for all $(x_i, y_i) \in Q_h$ by "merging" Q_h with C_h in time $O(|Q_h| + |C_h|)$. Since

C_h, $h \geq 0$, is a partition of the node set of T and since Q_h, $h \geq 0$, is a partition of the set $\{(x_i, y_i), 1 \leq i \leq 4 \cdot n\}$ the total time needed to compute app_i for all i is $O(n)$.

We claim that $a_i \in \{Lra(app_i), Lra(Lra(app_i))\}$. Let $w = Lra(Lra(app_i))$. Since x_i is descendant of app_i and hence of $Lra(app_i)$ and w, it suffices to show that y_i is not a descendant of app_i, but y_i is a descendant of w. Let $h = ld_i$.

(1) y_i is not a descendant of app_i
Node app_i is a node of height $h-1$ and hence has $2^h - 1$ descendants. leaf x_i is a descendant of app_i an $d_i \geq 2^h - 1$. hence y_i is not a descendant of app_i.

(2) y_i is a descendant of $w = Lra(Lra(app_i))$.
Let z be the right son of w. Then node z has height at least h. If y_i is not a descendant of w then all descendants of z lie between x_i and y_i. Hence $y_i - x_i - \geq 2^{h+1} - 1$ or $ld_i \geq h + 1$, a contradiction.

We have thus shown that array $Pos(e)$, $e \in E$, can be computed in time $O(n)$. Hence the time to construct the reduced search structure is $O(n)$. This proves part b). ∎

For the remainder of this section we show how to improve the search time to $O(\log n)$.∎
The search time in the reduced structure is $O((\log n)^2)$ because we spend time $O(\log n)$ in each node on the search path in the super-tree. Time $O(\log n)$ per super-node is required because we need to do a binary search (in y-direction) on a path of length $O(n)$. No attempt is made to use the information gained about the y-coordinate of query point q in order to speed up later searches in y-direction. One possible usage were a locality principle, i.e., having determined the position of query point with respect to path P_i we can restrict attention to a small subsegment of the paths stored in the sons of node i. Realization of a locality principle requires that we "align" the paths stored in adjacent nodes of the super-tree so as to provide for a smooth transition between them. How can we align the path stored in node v with the paths stored in the sons, say x and y, of node v? One possible way of achieving alignment is to include a suitable subset of the vertices of the paths associated with nodes x and y into the path stored in node v. The subset has to be dense within the paths in nodes x and y in order to enforce locality and it should be not too dense in order to keep storage space linear. A reasonable compromise is to include every other node. The details are as follows.

Let P_i^{red} be the sequence (ordered according to y-coordinate) of vertices stored in node i of the reduced search structure. We use Q_i to denote the sequence (ordered according to y-coordinate) stored in node i of the improved search structure. We define

$$Q_i = \begin{cases} P_i^{red} \cup \{s\} & \text{if } i \text{ is a leaf of the super-tree;} \\ P_i^{red} \cup Half(Q_{lson(i)}) \cup Half(Q_{rson(i)}) & \text{if } i \text{ is not a leaf,} \end{cases}$$

where operator *Half* extracts the odd elements of a sequence, i.e., $Half(v_1, v_2, v_3, v_4, \ldots) =$∎
$v_1, v_3, v_5, \ldots$. In our example we obtain the table of Figure 48.

Super-Node	P_i^{red}	Q_i
1	s, t	s, t
2	$1, t$	$s, 1, 4, t$
3	$1, 4$	$s, 1, 4$
4	$s, 1, 3, 4, t$	$s, 1, 2, 3, 4, t$
5	$\emptyset$	s
6	$1, 2, 3, t$	$s, 1, 2, 3, t$
7	$s, 2, t$	$s, 2, t$

Figure 48.

The sequences Q_i are stored as ordered linear lists except for Q_{root} which is organized as a balanced search tree. The balanced search tree is organized according to the y-coordinates of the elements of Q_{root}. With every interval between adjacent elements of Q_i we associate a pointer (the edge-pointer) to an edge of $\hat{Q}$. The pointer is stored in the upper endpoint of the interval and points to edge e if e is stored in P^{red} and if the interval is contained in the interval (of y-coordinates) covered by edge e. If there is no such edge then the pointer is undefined. In out example, sequences Q_4 and Q_6 have the structure shown in Figure 49.

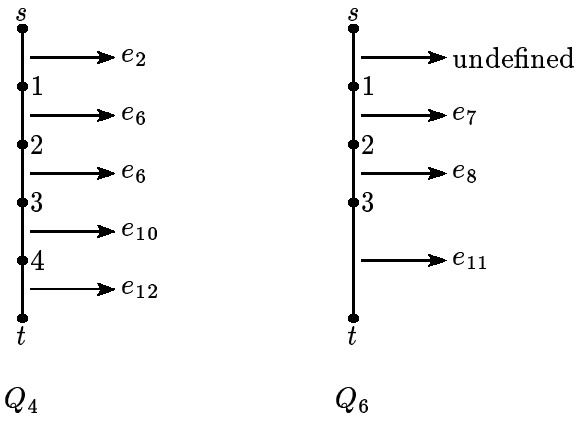


Figure 49.

Note that in Q_4 there are two intervals pointing to edge e_6 and that in Q_6 the edge pointer of interval $(s, 1)$ is undefined.

Finally, we have to make the alignment between the sequences explicit. We do so by associating two pointers (*Ralign* and *Lalign*)) with every element of Q_i; if v is an element of Q_i and i is not a leaf of the super-tree then $Ralign(v)$ points to node w on $Q_{rson(i)}$ where w is such that $y(w) \geq y(v) > y(suc(w))$ and $suc(w)$ is the successor of w in $Q_{rson(i)}$. If i is a leaf of the super-tree then $Ralign(v)$ points to v itself. Pointer $Lalign(v)$ is defined symmetrically. In our example, we obtain Figure 50. There the Q_i's are drawn vertically, the alignment pointers are

drawn horizontally, and the values of the edge-pointers are shown directly on the sequences.

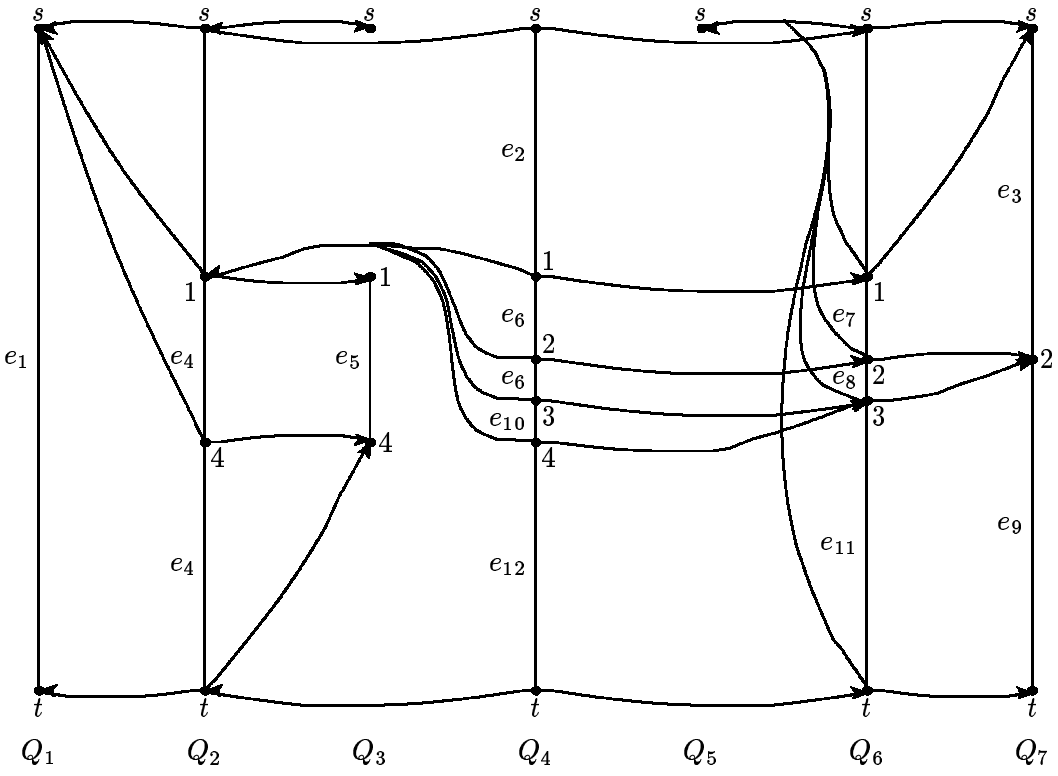


Figure 50.

Lemma 11.

a) *The search time in the improved search structure is* $O(\log n)$.

b) *The improved search structure requires storage space* $O(n)$ *and can be constructed in time* $O(n)$ *from the reduced search structure.*

Proof: a) We will modify the search algorithm described in Lemma 10a) such that $O(\log n)$ time is spent in the root node of the super-tree and $O(1)$ time in every other super-node on the search path. Clearly, we obtain $O(\log n)$ search time in this way.

As in Lemma 10 we assume $P_0 = P_1$, $P_{k+1} = P_k$ and $k = 2^{d+1} - 1$ for some d, that query point q lies between P_0 and P_{k+1} and that L is a horizontal line through q. We assume furthermore that each element of a sequence Q_i is a record with fields y (the y-coordinate of the element), *suc* (a pointer to the successor on Q_i), *Edgepointer*, *Ralign* and *Lalign*. Fields *Edgepointer*, *Ralign* and *Lalign* are as defined above.

The running time of this algorithm is clearly $O(\log n)$ since we need time $O(\log n)$ to find node v in line (3) by binary search on the balanced tree Q_{root}

```
(1)   l ← 0; r ← k + 1; m ← ⌈(l + r)/2⌉;
(2)   e_l (e_r) ← edge on P_l (P_r) intersected by L;
(3)   let v be a pointer to the node on Q_m such that v↑.y ≥ y(q) > v↑.suc↑.y
(4)   while r > l + 1
(5)   do co v is a pointer to a node on Q_m, m = ⌈(l + r)/2⌉, with
            v↑.y ≥ y(q) > v↑.suc↑.y. Also, point q lies between P_l and P_r
            and line L intersects edge e_l (e_r) of P_l (P_r) oc
(6)      if L(e_r) ≤ m or L(e_r) > m and q lies to the left of
         edge v↑.Edgepointer
(7)      then m ← r; v ← v↑.Lalign; redefine e_r
(8)      else  l ← m; v ← v↑.Ralign; redefine e_l
(9)      fi;
(10)     m ← ⌈(l + r)/2⌉;
(11)     if q(y) ≤ v↑.suc↑.y
(12)     then v ← v↑.suc
(13)     fi
(14)  od.
```

Program 9

and because each execution of the loop body takes time $O(1)$. It remains to argue correctness. Note first that the position of query point q with respect to path P_m is decided correctly in lines (6)–(9); point q lies to the left of P_m iff either edge e_r belongs to P_m or e_r does not belong to P_m and q lies to the left of the edge associated with interval $v\uparrow.y$ to $v\uparrow.suc\uparrow.y$.

Let u be the node of Q_m which is pointed to by v and let w be the node on the new Q_m which is aligned with u, i.e., v points to w before execution of line (11). By definition of pointers *Ralign* and *Lalign* we have $y(w) \geq y(u)$. Also since every other element of the new Q_m is also an element of the old Q_m and hence either $suc(w)$ or $suc(suc(w))$ (successor with respect to the new Q_m) is an element of the old Q_m we have $y(suc(u)) \geq y(suc(suc(w)))$ where $suc(u)$ is taken with respect to the old Q_m. Thus we correctly establish invariant $v\uparrow.y \geq y(q) \geq v\uparrow.suc\uparrow.y$ in lines (11)–(13). This proves correctness.

b) In order to prove the linear bound on the storage requirement it suffices to prove that the total length of the sequences Q_i is $O(n)$. We have

$$|Q_i| = \begin{cases} 1 + |P_i^{red}| & \text{if } i \text{ is a leaf;} \\ |P_i^{red}| + (|Q_{lson(i)}| + |Q_{rson(i)}|)/2 & \text{if } i \text{ is not a leaf} \end{cases}$$

and hence

$$\begin{aligned}\sum_{i=1}^{k}|Q_i| &\leq \sum_{i=1}^{k}(1+|P_i^{red}|)\cdot(1+1/2+1/4+1/8+\cdots)\\ &= 2\cdot\left(n+\sum_{i=1}^{k}P_i^{red}\right)\\ &= O(n)\end{aligned}$$

The first inequality follows from the observation that all nodes of P_i^{red} contribute to Q_i, one half of them contributes to $Q_{father(i)}$, one fourth of them to $Q_{father(father(i))}, \ldots$. ∎

It remains to show that we can obtain the improved search structure in time $O(n)$ from the reduced search structure. We can clearly construct Q_i from P_i^{red}, $Q_{lson(i)}$ and $Q_{rson(i)}$ by merging in time $O(|P_i^{red}|+|Q_{lson(i)}|+|Q_{rson(i)}|)$. Furthermore, it is easy to set-up the pointers *Edgepointer*, *Lalign* and *Ralign* during the construction without increasing the cost by more than a constant factor. Hence the improved search structure can be constructed in time $O(\sum_i |Q_i|) = O(n)$. ∎

We summarize the discussion of this section in

Theorem 3. *Based on path decomposition the searching problem for simple planar subdivision can be solved with search time $O(\log n)$, storage space $O(n)$, and preprocessing time $O(n)$.*

We have travelled quite a distance in this section. We started out with the basic concept of path decomposition and obtained very quickly a data structure with $O((\log n)^2)$ search time and $O(n^2)$ storage requirement. We then refined the data structure and first reduced storage space and preprocessing time to $O(n)$ and then search time to $O(\log n)$. We will see path decomposition again in Section 5.1.4.

8.3.2.3. Searching Dynamic Planar Subdivision

In the preceding section we described to optimal solutins to the searching problem for planar subdivisions. Several other solutions are discussed in the exercises. All these solutions have a common weakness. They only apply to static subdivisions and do not seem to adopt easily to changes in the underlaying planar subdivision.

Consider the following szenario which captures a simple from of dynamic behavior; a solution to more genereal forms of dynamic behavior is not within sight at the day of this writing. At any point of time we can either query the planar subdivision with respect to a query point $q \in \mathbb{R}^2$ or subdivide one of the finite faces of the subdivision by a straight line thus adding one additional edge and up to two additional vertices. The initial subdivision is a triangle. Note that all vertices constructed by adding new edges have degree at least three. Our goal for this section is to prove the following

Theorem 4. *There is a solution to the dynamic planar subdivision searching problem with query and update time $O((\log n)^2)$. The bound on query time is worst case and the bound on update time is amortized.*

Proof: Our approach is similar to the one used in Section 3.2.2, i.e., we will againn use binary search on a path decomposition. However, there are somemajor differences. First, we cannot assume that our planar subdivision is triangulated and hence we cannot insist on a path decomposition into monotone paths. Rather, we have to use arbitrary (s,t)-paths in the decomposition. Second, the super-tree is not static anymore but needs to be adapted as the subdivision changes. In order to keep the cost of updating the super-tree low we organize the super-tree as a weight-balanced tree.

Let $\hat{G}$ be the (current) planar subdivision and let s (t) be the vertex of $\hat{G}$ with maximal (minimal) y-coordinate. Note that s and t do not depend on time since only finite faces can be refined. As before, an (s,t)-path is a sequence $v_0, v_1, \ldots, v_m$ of vertices with $v_0 = s$, $v_m = t$, (v_i, v_{i+1}) an edge of G for $0 \leq i < m$, and $(v_i, v_{i+1}) \neq (v_j, v_{j+1})$ for $i \neq j$. An (s,t)-path P divides $\hat{G}$ into two parts which we call the left and right part of $\hat{G}$ with respect to P.

A sequence $P_1, \ldots, P_N$ of paths is a complete path decomposition of $\hat{G}$ if

1) every P_i, $_{\leq} i \leq N$, is an (s,t)-path and every edge of $\hat{G}$ belongs to at least one P_i;

2) P_1 is the leftmost (s,t)-path, P_N is the rightmost (s,t)-path and P_{i+1} is to the right if P_i, $1 \leq i < N$, i.e., no horizontal line intersects P_i to the right of P_{i+1};

3) for every i, $0 \leq i \leq N-1$, there is exactly one face of $\hat{G}$, say F, such that all edges belonging to either P_i or P_{i+1} but not to both border the samme face of $\hat{G}$. We will say that pair $8P_i, P_{i+1})$ moves across face F in this case.

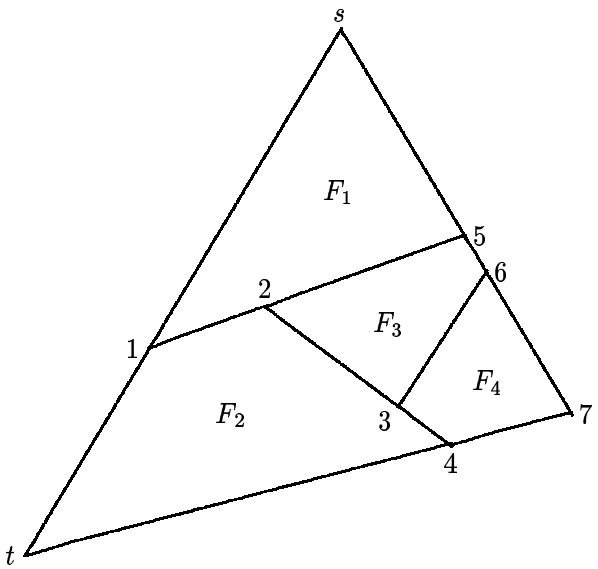


Figure 51.

In our example, there are four vaces and hence a complete path decomposition consists of five paths P_1, P_2, P_3, P_4 and P_5. We might have $P_1 = s, 1, t$, $P_2 = s, 1, 2, 3, 4, t$, $P_3 = s, 1, 2, 5, 6, 3, 4, t$, $P_4 = s, 5, 6, 3, 4, t$ and $P_5 = s, 5, 6, 7, 4, t$. Then pair (P_1, P_2) moves across face F_2 and pair (P_2, P_3) moves across face F_3.

We can now describe the search structure for subdivision $\hat{G}$ in more detail. Let $P_1, \ldots, P_N$ be a complete path decomposition of $\hat{G}$. Then the super-tree is a BB$[\alpha]$-tree, say $\alpha = 0.28$, with N nodes; cf. Section 3.5.1 for a discussion of BB$[\alpha]$-trees. Node i corresponds to path P_i. Node i contains a data structure which allows us to decide the position of query point q with respect to path P_i in time $O(\log n)$. Since the super-tree has depth $O(\log n)$ a total query cost of $O((\log n)^2)$ results.

As before, this basic solution suffers from its huge storage requirement which now also implies huge update cost. We proceed as in the preceding section, and store in node v of the super-tree only those edges of path P_v which do not belong to a path P_w for w an ancestor of v. We denote this set of edges as P_v^{red}. In our example, we might use the super-tree of Figure 52 in BB[1/4]. Then $P_3^{red} = \{(2,5),(5,6)\}$ and $P_5^{red} = \{(6,7),(7,4)\}$.

For edge e, let $L[e]$ ($R[e]$) be the minimal (maximal) i such that e belongs to path P_i. We will describe below how integers $L(e)$ and $R(e)$ are maintained such that computation of $L(e)$ and $R(e)$ take time $O(\log n)$ each. For the description of the query algorithm we assume the existence of algorithms for computing $L(e)$ and $R(e)$. Also we assume that the data structure associated with node i has the following property (called property $(*)$):

Given point $q \in \mathbb{R}^2$ and a horizontal line L through find edges e and e' in P_i^{red} (if they exist) such that L intersects e and e', q lies between e and e' and no other edge in P_i^{red} intersects L between the intersections with e and e'.

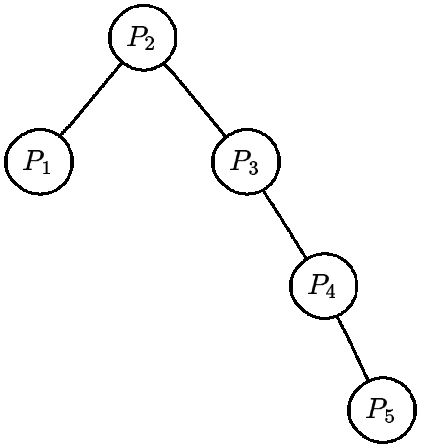


Figure 52.

Lemma 12. *Let $m = |P_i^{red}|$. Then a search structure for P_i^{red} with property $(*)$ and query time $O(\log m)$ can be constructed in time $O(m \cdot \log m)$.*

Proof: The set P_i^{red} is a set of pairwise non-intersecting edges (except at common endpoints). Draw horizontal lines through all endpoints of edges in P_i^{red} and extend them to the closest edge in P_i^{red}.

In our example, we have $P_3^{red} = \{(2,5),(5,6)\}$ and hence we obtain the following planar subdivision by this proces.

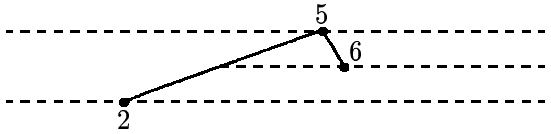


Figure 53.

In general, we obtain a planar subdivision with $O(m)$ vertices all of whose faces are convex. Moreover, we can obtain this subdivision in time $O(m \log m)$ by plane sweep. We can now use either one of the methods of the preceding sections to obtain a search structure with property $(*)$ of depth $O(\log m)$ in time $O(m \log m)$. ∎

The bound on the query time is easily derived at this point.

Lemma 13. *Given query point $q \in \mathbb{R}^2$ one can determine the face of $\hat{G}$ containing q in time $O((\log n)^2)$ where n is the number of vertices of $\hat{G}$.*

Proof: We use a tree search on the super-tree. In each node of the super-tree we spend time $O(\log n)$. Let L be a horizontal line through q. Assume inductively that the search has reached node i of the super-tree and that we determined paths P_j and P_k and edges e_L and e_R on P_k such that q lies between e_L and e_R, i.e., L

intersects e_L and e_R. Also, nodes j and k are ancestors of node i. Note that either j or k or both may not exist. Figure 54 illustrates the situation. In this diagram paths P_j and P_k are shown dashed and path P_i is shown solid. Furthermore, edges e_L, e_R, e and e' are indicated.

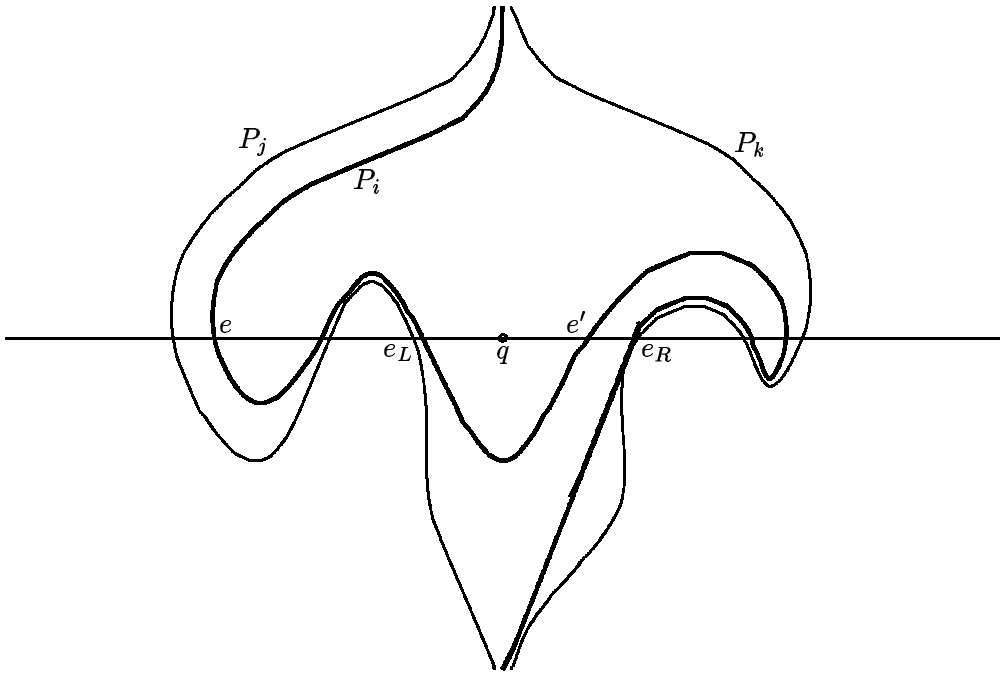


Figure 54.

We can now use the data structure (with property $(*)$) for P_i^{red} to determine a pair of edges e and e' of P_i^{red} such that q lies between e and e'. Also, we can determine whether edges e_L and e_R belong to P_i by looking up their L- and R-values. Using this knowledge it is then easy to decide on which side of path P_i the query point q lies. Thus time $O(\log n)$ suffices to determine the position of point q with respect to P_i and hence total search time is $O((\log n)^2)$.

We turn to the insertion of new edges next. An additional edge splits a face, say F, into F_1 and F_2. If $P_i, P_{i+1})$ is the pair of paths which moves across F then $P_1, \ldots, P_i, P, P_{i+1}, \ldots, P_N$ is a complete path decomposition of the new planar subdivision where path P runs betwen paths P_i and P_{i+1} and uses edge e. More precisely, if $e = (x, y)$, then P consists of the initial segment of P_i (P_{i+1}) from w to x if x lies on the left (right) boundary of F, followed by edge e, followed by a terminal segment of P_i (P_{i+1}) from Y to t if y lies on the left (right) boundary of F.

In our example, we can split face F_4 by adding an edge from vertex 6 to vertex 4. Then $P_1, P_2, P_3, P_4, P, P_5$ is a complete path decomposition of the new planar subdivision where $P = s, 5, 6, 4, t$. We obtain a super-tree for the new path decomposition by adding a node P between P_4 and P_5. The new super-tree does

not belong to class BB[0.28] since the balance of node P_3 is $1/4 \notin [0.28, 0.72]$. Therefore, we need to rebalance the super tree at node P_3.

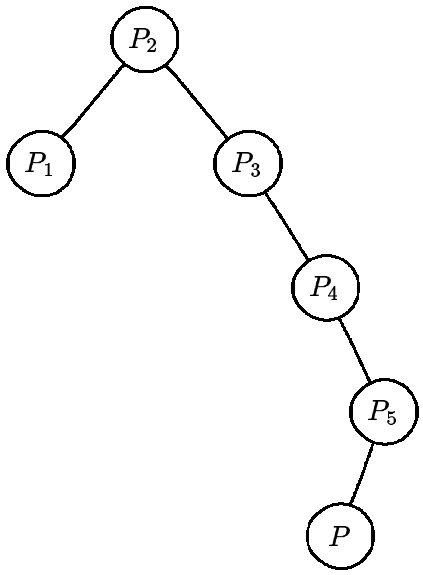


Figure 55.

In general, we add a node for P as either a right son of node i (if i is a descendant of $i+1$) or as a left son of node $i+1$ (if node $i+1$ is a descendant of node i).

Also, P^{red} consists exactly of the new edge e since all other edges of P also belong to ancestors of the new node. It is therefore trivial to construct a data structure (with property $(*)$) for P^{red}.

Of course, the super-tree can go out of balance, i.e., leave class BB[α], by adding the new node corresponding to path P and hence rebalancing is necessary. Before we can describe the rebalancing algorithm we need to discuss in detail how arrays L and R are maintained. We store them implicitely as follows. Note first that we can partition the edges on the boundary of every face into two blocks in a natural way. Let F be a face and let pair $(P_i, P1i+1)$ be the pair of paths which moves across F. Then the left boundary $LB(F)$ of F consists of all edges of P_i which do not belong to P_{i+1}. The right boundary $RB(F)$ is defined symmetrically. Next observe, that if edge e belongs to $LB(F)$ then $R(e) = i$ and if edge e' belongs to $RB(F)$ then $L(e) = i+1$ where pair (P_i, P_{i+1}) moves across face F. Finally, observe that every edge belongs to the boundary of exactly two faces. More precisely, it belongs to the left boundary of some and to the right boundary of some other face.

These observations suggest the following method for storing arrays L and R implicitely. For every face F we store the edges in $LB(F)$ $(RB(F))$ in a balanced tree. The edges are ordered clockwise. In the root of the tree for $LB(F)$ $(RB(F))$ we store a pointer to node i $(i+1)$ of the super-tree where i $(i+1)$ is the common value of $R[e]$ $(L[e])$ for the edges in $LB(F)$ $(RB(F))$. With every edge e we associate

two pointers. Pointer $LP(e)$ ($RP(e)$) points to the copy of edge e in a left (right) boundary list $LB(F)$ ($RB(F)$) for some face F. Using these data structures we can compute $R[e]$ as follows; the computation of $L[e]$ is symmetric. We follow pointer $LB(e)$ to the (unique) copy of edge e on a left boundary list, say $LB(F)$, then we inspect the root of the tree representation of $LB(F)$ and obtain a pointer to node $R[e]$ of the super-tree. Note that a knowledge of a pointer to node $R(e)$ suffices for the query algorithm; the numericalvalue $R[e]$ is not required. The representation described above clearly allows us to compute $L[e]$ and $R[e]$ in time $O(\log n)$.

We now turn to the discussion of the insertion algorithm. Adding a new edge e splits a face, say F, into two faces, say F_1 and F_2. Also a new path P is added to the path decomposition. The tree representation of $LB(F_1)$, $LB(F_2)$, $RB(F_1)$, $RB(F_2)$ can be obtained from $LB(F)$ and $RB(F)$ in time $O(\log n)$ by a few split and concatenate operations (cf. Section 3.5.3.1). This shows that we can add a new edge in time $O(\log n)$ excluding the time required for rebalancing the super-tree.

Let i be the highest node of the super-tree which is out of balance after adding edge e. Let $I = [l\,..\,r]$ be the set of descendants of i (including i) and let $E = \bigcup_{j\in I} P_j^{red}$ be the set of edges stored in the descendants of i. Finally, let $m = |E|$ and $k = |I|$. We rebalance the tree by replacing the subtree rooted at i by a perfectly balanced tree with node set I. This takes time $O(k)$. We then go through all edges in E, compute their L- and R-values as described above and decide in what node of the new subtree each eedge has to be stored. This takes time $O(m \log n)$ since each L- and R-value can be computed in time $O(\log n)$. In this way we compute for each node j, $l \le j \le r$, the set P_i^{red} of edges which has to be stored in node j of the new subtree. Let $M_j = |P_j^{red}|$. Thenn it takes time $O(m_j \log m_j)$ to construct the data structure with property $(*)$ for node j and hence time

$$\begin{aligned}\sum_{j\in I} O(m_j \log m_j) &= \sum_{j\in J} O(m_j \log n)\\ &= O(|I| + \sum_{j\in I} m_j \log n)\\ &= O(k + m \log n)\end{aligned}$$

to construct these data structures for all nodes in I. This finishes the description of the rebalancing algorithm. Note that L- and R-values of all edges remain unchanged since the path decomposition is not changed; only its arrangement in the super-tree is changed. Thusno action is required for the L- and R-values. We summarize the discussion in

Lemma 14. *Rebalancing at anode i with k descendants takes time $O(k \cdot \log n)$.*

Proof: We have shown above that rebalancing takes time $O(k + m \log m)$. It therefore suffices to prove that $m = O(k)$. This can be seen as follows. Note first that $k+1$ is the number of faces of $\hat{G}$ between paths P_{l-1} and P_{r+1} of the decomposition. Next consider the following planar graph with $k+1$ nodes. Its nodes are the faces

between P_{l-1} and P_{r+1} and its edges are the duals of the edges in E, i.e., edges connect adjacent faces. This planar graph has m edges and *no* parallel edges since any two finitey faces of the planar subdivision can share at most one edge. This fact can easily be seen by induction on the number of edges added to the planar subdivision. Thus $m = O(k)$ by Lemma 1 of Section 4.10. ∎

We are now ready to derive a bound on the amortized insertion cost.

Lemma 15. *The amortized insertion cost of an insertion is* $O((\log n)^2)$.

Proof: We infer from Lemma 14 that the cost of rebalancing at a node with k descendants is $O(k \log n)$. Hence the total cost of all rebalancing operations required to process n insertions is

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

by (a variant of) Theorem 5 of Section 3.6.3. thus amortized rebalancing cost is $O((\log n)^2)$. Finally, observe that the cost of changing the path decomposition after adding an edge is $O(\log n)$. This proves Lemma 15. ∎

Lemmas 13 and 15 together imply Theorem 4. ∎

Theorem 4 deals only with very limited versions of dynamic behaviour. In particular, deletions, replacement of edges by pairs of edges, and non-connected subdivisions cannot be handled. A treatment of the more general forms of dynamic behaviour would be of great help in the usage of the plane sweep paradigm in three-dimensional space as we will see in Section 4.3.

3.3. Applications

In this section we discuss several applications of Voronoi Diagrams: the all pair nearest neighbor problem, the Euclidian Spanning tree problem and the Euclidian traveling salesman problem. Let $S \subseteq \mathbb{R}^2$. In the all pair nearest neighbor problem we want to find for each $x \in S$ an element $y \in S$ such that $dist(x,y) = dist(x, S - \{x\})$, i.e., the element closest to x. For the two other problems we consider the complete network $N = (S, S \times S, c)$ with vertex set S and edge costs as given by Euclidian distance, i.e., $c(x,y) = dist(x,y)$. Other applications can be found in the exercises. For the three applications discussed here the following lemma is crucial.

Lemma 16. *Let $S \subseteq \mathbb{R}^2$, let $x, y \in S$, $x \neq y$. If $VR(x) \cap VR(y) = \emptyset$ or $VR(x) \cap VR(y)$ is a singleton set then there is $z \in S$ such that $dist(x,z) \leq dist(x,y)$ and $dist(y,z) < dist(x,y)$ and $VR(x)$ and $VR(z)$ have a non-trivial line segment in common.*

Proof: Let $x, y \in S$, $x \neq y$ and assume that $VR(x)$ and $VR(y)$ do not share a non-trivial line segment. Consider the straight line segment L connecting x and y. Let p be the point of intersection of L and the boundary of $VR(x)$ and let z be such that p lies on a common edge of $VR(x)$ and $VR(z)$. It is conceivable that p is an endpoint of this edge. We show $dist(x,z) \leq dist(x,y)$ and $dist(y,z) < dist(x,y)$.

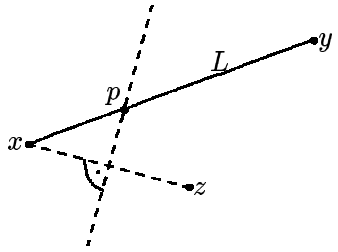


Figure 56.

$dist(y,z) < dist(x,y)$: Note first that $dist(y,z) \leq dist(x,y)$ since y and z lie on the same side of the perpendicular bisectors of x and z. If $dist(y,z) = dist(x,y)$ then y lies on the perpendicular bisector of x and z and hence $y = p$. Thus y, a point of S, lies on the boundary of $VR(x)$, contradiction. We conclude $dist(y,z) \neq dist(x,y)$ and hence $dist(y,z) < dist(x,y)$.

$dist(x,z) \leq dist(x,y)$: We observe that $dist(x,z) \leq 2 \cdot dist(x,p)$, $dist(x,p) \leq dist(p,y)$ since $p \in VR(x)$ and $dist(x,y) \leq dist(x,p) + dist(p,y)$. Thus $dist(x,z) \leq 2 \cdot dist(x,y) \leq dist(x,p) + dist(p,y) = dist(x,y)$. ∎

We can now proceed to the applications.

Theorem 5. *Let $S \subseteq \mathbb{R}^2$, $|S| = n$. Given Voronoi Diagram of S one can solve the all pair nearest neighbor in time $O(n)$.*

Proof: Let $x \in S$ be arbitrary. We infer from Lemma 16 that x has a nearest neighbor y such that $VR(x)$ and $VR(y)$ have a non-trivial line segment in common. Thus we can find the nearest neighbor of x by inspecting all y such that $VR(x)$ and $VR(y)$ have an edge in common; this takes time $O(m(x))$, where $m(x)$ is the number of edges on the boundary of $VR(x)$. We conclude that the total running time is $O(\sum_{x \in S} m(x)) = O(n)$ by Lemma 2. ∎

Theorem 6. *Let $S \subseteq \mathbb{R}^2$, $|S| = n$. Then a minimum cost Euclidian spanning tree of S can be computed in time $O(n \log n)$.*

Proof: We want to compute a minimum cost spanning tree of the network $N = (S, S \times S, c)$ with $c(x, y) = dist(x, y)$. Let $E = \{(x, y);\ x, y \in S, x \neq y$ and $VR(x)$ and $VR(y)$ share a non-trivial line segment$\}$. We show that the edges of a minimum spanning tree can be taken from set E.

Claim: *There is a minimum cost tree T of network N such that all edges of T belong to E.*

Proof: Let T be a minimum cost spanning tree of network N such that T has a maximal number of edges in E. If all edges of T belong to E then we are done. So let us assume otherwise. Then there must be an edge (x, y) of T with $(x, y) \notin T$. Thus $VR(x)$, and $VR(y)$ do not have a non-trivial line segment in common and hence by Lemma 16 there is $z \in S$ such that $VR(x)$ and $VR(z)$ share a non-trivial line segment, $dist(x, z) \leq dist(x, y)$ and $dist(y, z) < dist(x, z)$. Also either $T_1 = (T - \{(x, y)\}) \cup \{(x, z)\}$ or $T_2 = (T - \{(x, y)\}) \cup \{(y, z)\}$ is a spanning tree of N. However, T_1 has more edges in E than T and the same cost as T and T_2 has smaller cost than T. Thus in either case we obtain a contradiction to the fact that T uses edges outside E and has minimum cost. ∎

We conclude from the claim above that it suffices to determine a minimum spanning tree of the network (S, E, c). Set E can be determined in time $O(n \log n)$ by constructing the Voronoi Diagram of S. Also, $|E| = O(n)$ by Lemma 2 and hence a minimum cost spanning tree of (S, E, c) can be determined in time $O(n \log n)$ by Section 4.8, Theorem 2. Actually, in view of Section 4.8, Theorem 4, time $O(n)$ suffices for the final step since (S, E) is a planar graph. ∎

From Theorem 56 we obtain a good approximation algorithm for Euclidian traveling salesman tours.

Theorem 7. *Let $S \subseteq \mathbb{R}^2$, $|S| = n$, and let L_{opt} be the length of an optimal Euclidian traveling salesman tour of S. then a tour of length at most $2 \cdot L_{opt}$ can be found in time $O(n \log n)$.*

Proof: By Theorem 6 we can find a minimum cost Euclidian spanning tree in time $O(n \log n)$. In Section 6.7.1 we have shown that the "once around the tree" tour has length at most $2 \cdot L_{opt}$. The result follows. ∎

8.4. The Sweep Paradigm

The sweeping approach is a very powerful paradigm for solving two-dimensional and some higher dimensional geometric problems. It has mostly been used for intersection problems but has also proved useful for other problems, e.g., triangulation and order problems. The underlaying idea is quit simple. Suppose that we have to solve a problem concerning a set of geometric objects in the plane. A concrete example is the intersection problem of line segments. Plane sweep approaches this problem by sweeping a vertical line from left to right across the plane. It uses a data structure, called the y-structure, to record the status of the sweep at the current position of the sweep line. The status of the sweep is all information about the problem to the left of the sweep line which is relevant to solving the problem to the right of the sweep line. In all our applications this information encompasses at least the intersections of the sweep line at its current position with the geometric objects at hand. Moreover, we will always have these intersections sorted by y-coordinate. Additional information depending on the particular problem to be solved is also associated with he y-structure.

The sweep line gradually moves from left to right. Often, there are only a few positions of the sweep line which can cause a change in the status of the sweep. In our example of intersecting line segments this will be the endpoints of line segments and the points of intersection of line segments. The positions at which the status of the sweep changes are stored in the x-structure. Usually, some points of the x-structure are already known initially (the endpoints of line segments in our example) and some are computed during the sweep (the points of intersection in our example). Thus the sweep advances from point to point in the x-structure. At each point, the y-structure is updated, some output is computed and some additional points are inserted into the x-structure. Of course, the points to be inserted into the x-structure must lie to the right of the sweep line if plane sweep is to work.

The paradigm of plane sweep is captured in the algorithm of Program 10.

```
(1)   initialize x-structure and y-structure;
(2)   while x-structure ≠ ∅
(3)   do p ← min(x-structure);
(4)       Transition(p)
(5)   od.
```

Program 10

In line (3) the point with minimal x-coordinate is selected in the x-structure and deleted from it. In line (4) the sweep line is advanced beyond that point. Of course, the details of procedure *Transition* depend on the specific problem that is to be solved. in this chapter we study three applications of the sweep paradigm: intersection problems in the plane, triangulation problems in the plane and intersection problems in three-dimensional space.

The success of the sweep paradigm in these cases results from the fact that it reduces the dimension of the problem to be solved. More precisely, it solves a static problem in the plane by solving a dynamic problem on the (sweep) line. The latter problem is essentially one-dimensional and therefore simpler and better understood.

8.4.1. Intersecting Line Segments and Other Intersection Problems in the Plane

In this section we will solve a number of intersection problems in the plane. We start with intersecting line segments and then extend the basic algorithm to various other cases: decomposing a nonsimple polygon into simpler parts, intersecting sets of polygons, translating sets of line segments, ... Some fo these applications are treated in the exercises.

Let $L_1, L_2, \ldots, L_n$ be a set of n line segments in the plane. We want to compute all their pairwise intersections, say there are s of them. Our first approach consists of checking all pairs L_i, L_j with $1 \leq i < j \leq n$ for intersection. It runs in time $O(n^2)$. Of course, this algorithm is optimal if $s = \Omega(n^2)$, since s is the size of the output. However, if $s \ll n^2$ then a much better algorithm exists. We will show how to compute all s intersections in time $O((n+s)\log n)$.

Theorem 1. *Let $L_1, \ldots, L_n$ be a set of n line segments in the plane. Then the set of all s pairwise intersections can be computed in time $O((n+s)\log n)$ and space $O(n)$.*

Proof: We use plane sweep, i.e., we sweep a vertical line from left to right across the plane. At any point of the sweep we divide the set of line segments into three pairwise disjoint groups: dead, active and dormant. A line segment is dead (active, dormant) if exactly one (two, zero) of its endpoints are to the left of the sweep line. Thus the active line segments are those which currently intersect the sweep line and the dormant line segments those which have not yet been encountered by the sweep. For the description of the algorithm we assume that no line segment is vertical and that no two endpoints or intersection points have the same x-coordinate. both assumptions are made to simplify the exposition. The reader should have no difficulty in modifying the algorithm in order to make it work without these assumptions.

The y-structure stores the active line segments ordered according to the y-coordinate of their intersection with the sweep line. More precisely, the y-structure is a balanced search tree for the set of active line segments. In our example, line segments $L_1, L_2, \ldots, L_6$ are active.

They are sorted in the y-structure in that order as described in the introduction of this chapter, i.e., the y-structure is a dictionary for the set of active line segments. Any kind of balanced tree can be used for the dictionary. It supports (at least) the following operations in logarithmic time.

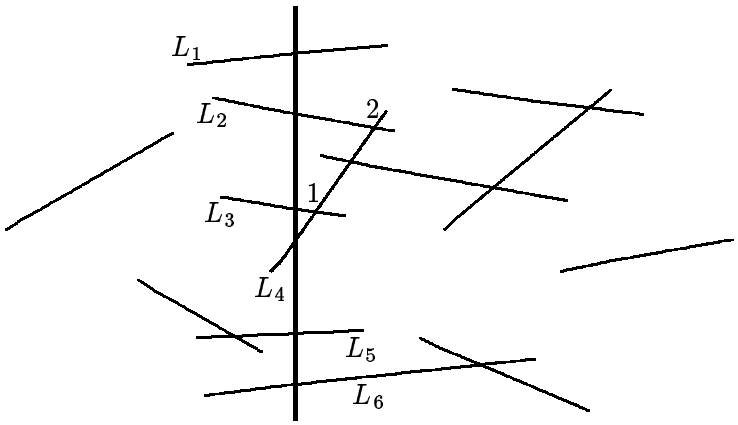


Figure 57.

find(p)	given point p on the sweep line, find the interval (on the sweep line) containing p
Insert(L)	insert line segment L into the y-structure
Delete(L)	delete line segment L from the y-structure
Pred(L), *Succ*(L)	find the immediate predecessor (successor) of line segment L in the y-structure
Interchange(L, L')	interchange adjacent line segments L and L' in the y-structure

It is worthwhile observing that the cost of operations *Pred*, *Succ* and *Interchange* can be reduced to $O(1)$ under the following assumptions. First, the procedures are given a pointer to the leaves representing line segment L as an argument and second, the tree structure is augmented by additional pointers. For the *Pred* and *Succ* operations we need pointers to the adjacent leaves and for the interchange operation we need a pointer to the least common ancestor of leaves L and L'. Note that the least common ancestor of leaves L and L' contains the information which discriminates between L and L' in the tree search. We leave it to the reader to convince himself that the additional pointers do not increase the running time of *Insert*s or *Delete*s.

We describe the x-structure next. It contains all endpoints of line segments (dormant or active) which are to the right of the sweep line. Furthermore, it contains *some* of the intersections of line segments to the right of the sweep line. Note that it cannot contain all of them because the sweep has not even seen dormant line segments yet. The points in the x-structure are sorted according to their x-coordinate, i.e., the x-structure is a heap. For the correctness of the algorithm it is important that the x-structure always contains the point of intersection of active line segments which is closest to the sweep line. We achieve this goal by maintaining the following invariant.

If L_i and L_j are active line segments, are adjacent in the y-structure, and intersect to the right of the sweep line then their intersection is contained in the x-structure.

In our example, point 1 must be in the x-structure and point 2 may be. In the space-efficient version of the algorithm any point below point 2 is not in the x-structure. We have the following consequence of the invariant above.

Lemma 1. *Let p be the intersection of active line segments L_i and L_j. If there is no endpoint of a line segment and no other point of intersection in the vertical strip defined by the sweep line and p then p is stored in the x-structure.*

Proof: If p is not stored in the x-structure then L_i and L_j are not adjacent in the y-structure. Hence there must be an active line segment L which is between L_i and L_j in the y-structure. Since L's right endpoint is not to the left of p, either $\bigcap(L, L_i)$ or $\bigcap(L, L_j)$ is to the left of p, contradiction. ∎

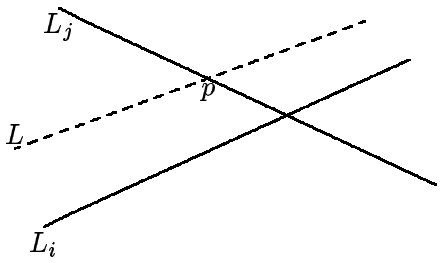


Figure 58.

Finally, we maintain the following invariant about the output. All intersections of line segment which are to the left of the sweep line have been reported.

We are now in a position to give the details of the algorithm.

In the algorithm above the statements in square brackets are not essential for correctness. inclusion of these statements does not increase asymptotic running time, however it improves space complexity from $O(n + s)$ to $O(n)$ as we will see below.

The following example illustrates the algorithm.

We still have to prove correctness and to analyze the running time. For correctness it suffices to show that the invariant holds. Call a point critical if it is either the endpoint of a line segment or an intersection of two line segments. Then the invariant about the x-structure and Lemma 1 ensures that the point p selected in line (4) is the critical point which is closest to and ahead of the sweep line. Thus each critical point is selected exactly once in line (4) and hence all intersections are output in line (21). Furthermore, lines (7), (13), and (17) ensure that the y-structure always contains exactly the active line segments in sorted order and lines (8), (14), and (19) guarantee the invariant about the x-structure. In lines (9)

```
(1)   y-structure/ ← ∅;
(2)   x-structure/ ← the 2n endpoints of the line segments sorted by x-coordinate;
(3)   while x-structure/ ≠ ∅
(4)   do let p be a point with minimal x-coordinate in the x-structure;
(5)      delete p from the x-structure;
(6)      if p is a left endpoint of some segment L_j
(7)      then search for p in the y-structure and insert L_j into the y-structure;
(8)           let L_i, L_k be the two neighbors of L_j in the y-structure;
                 insert ⋂(L_i, L_j) and ⋂(L_j, l_k) into the x-structure, if they exist;
(9)           [delete ⋂(L_i, L_k) from the x-structure]
(10)     fi;
(11)     if p is a right endpoint of some segment L_j
(12)     then let L_i and L_k be the two neighbors of L_j in the y-structure;
(13)          delete L_j from the y-structure;
(14)          insert ⋃(L_i, L_k) into the x-structure if the intersection is
                 to the right of the sweep line
(15)     fi;
(16)     if p = ⋂(L_i, L_j)
         then co L_i, L_j are necessarily adjacent in the y-structure oc
(17)          interchange L_i, L_j in the y-structure;
(18)          let L_h, L_k be the two neighbors of L_i, L_j in the y-structure;
(19)          insert ⋂(L_h, L_j) and ⋂(L_i, L_k) into the x-structure, if they are
                 to the right of the sweep line;
(20)          [delete ⋂(L_h, L_i) and ⋂(L_j, L_k) from the x-structure;]
(21)          output p
(22)     fi
(23)  od
```

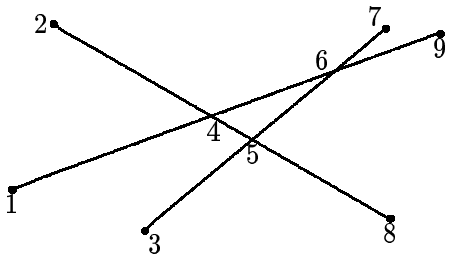
Program 11

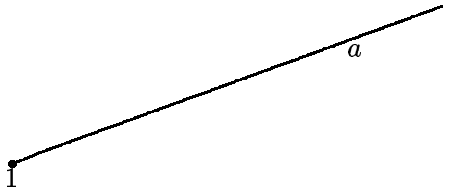
and (20) we delete points from the x-structure whose presence is no longer required by the invariant about the x-structure. This completes the proof of correctness.

The analysis of the running time is quite simple. Note first that the loop body is executed exactly $2n+s$ times, once for each endpoint and once for each intersection. Also, a single execution deletes an elment from a heap (time $O(\log(n+s)) = O(\log n)$ since $s \leq n^2$) and performs some simple operations on a balanced tree of size n (time $O(\log n)$). Thus the running time is $O((n+s)\log n)$.

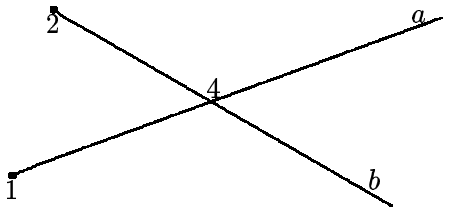
The space requirement is clearly $O(n)$ for the y-structure and $O(n+s)$ for the x-structure. If we include lines (9) and (20) then the space requirement of the x-structure reduces to $O(n)$, since by this modification only intersections of active line segments which are adjacent in the y-structure are stored in the x-structure. Thus space requirement is $O(n)$. ∎

The algorithm above works for segments which are more general than straight line

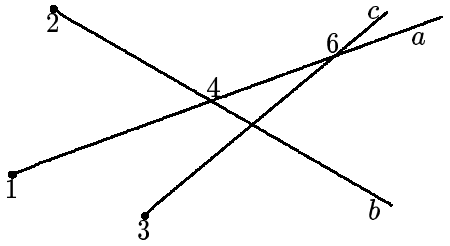




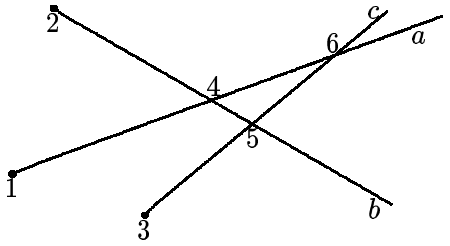
x-structure: 2, 3, 7, 8, 9
y-structure: a
sweep line: between 1 and 2



x-structure: 3, 4, 7, 8, 9
y-structure: a, b
sweep line: between 2 and 3



x-structure: 4, 6, 7, 8, 9
y-structure: c, a, b
sweep line: between 3 and 4



x-structure: 5, $\boxed{6}$, 7, 8, 9
y-structure: c, b, a
sweep line: between 4 and 5

Remark: In the space-efficient version node 6 is deleted from the x-structure and then reinserted by the transition at point 5.

Figure 59.

segments, e.g., for circular segments (Exercise 24). If all line segments are vertical or horizontal then the running time can be improved and reaches $O(s + n \log n)$, Exercise 23.

Our next goal is to extend the algorithm above to more complicated tasks. More specifically, we show how to decompose a polygon into simple parts. Let $x_0, \ldots, x_{n-1}$ be a sequence of points in the plane. Then line segments $L_i :=$

$L(x_i, x_{i+1})$, $0 \leq i \leq n-1$, define a closed curve in the plane. The removal of this curve from the plane divides the plane into $r+1$ polygonal regions $R_0, R_1, \ldots, R_r$ one of which is unbounded; say R_0.

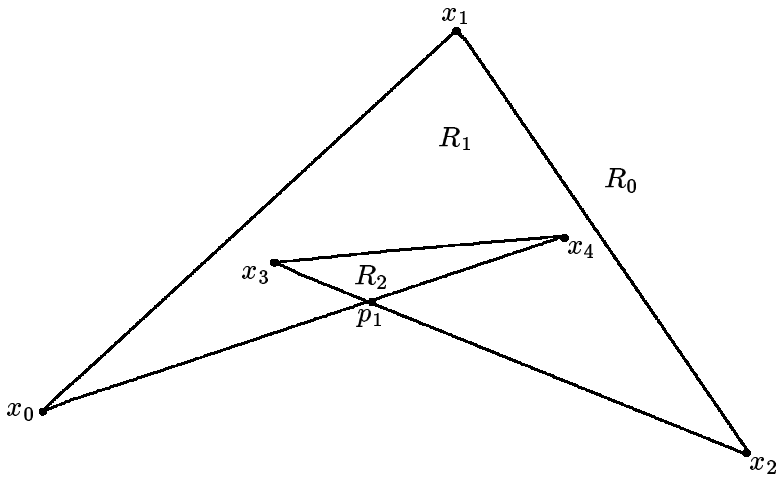


Figure 60.

Each R_i, $1 \leq i \leq r$, is a polygonal region. The boundary of R_i is a simple polygon whose vertices are either among the x_i or among the points of intersection of the line segments L_i. Let $p_1, \ldots, p_s$ be the intersections of the line segments L_i. Then our goal is to compute the boundary vertices in cyclic order for each region R_i. In our example, the output is $R_0 : x_0, p_1, x_2, x_1$, $R_1 : x_1, x_0, p_1, x_3, x_4, p_1, x_2$, and $R_2 : x_3, p_1, x_4$.

We have chosen the problem of polygon decomposition for expository purposes. It is a simple intersection problem. Nevertheless, the underlying ideas required to solve it are applicable to a wide variety of problems, e.g., the intersection of a set of polygons. We know already how to compute the points $p_1, \ldots, p_s$ by plane sweep. We will show now how to extend the y-structure such that the more complicated decomposition problem can be solved. Consider our example and suppose that the sweep line is positioned between x_1 and x_4.

At this point we have line segments $L(p_1, x_2)$, $L(p_1, x_4)$, $L(x_3, x_4)$, $L(x_1, x_2)$ in the y-structure. Note that there is always an even number of line segments in the y-structure since we sweep a closed curved. The four line segments split the sweep line into five intervals, two of which are infinite. For each interval, we conceptually record the tentative name of the regions to which the interval belongs (cf. Fig. 62), i.e., these names are used to illustrate the algorithm but they are not actually stored in the augmented y-structure. Two intervals have the same name if they belong to the same region and this region is connected to the left of the sweep line. Thus the two infinite intervals always have the same name. Also, in our example the intervals between $L(x_1, x_2)$ and $L(x_3, x_4)$ and between $L(p_1, x_4)$ and $L(p_1, x_2)$ have different names.

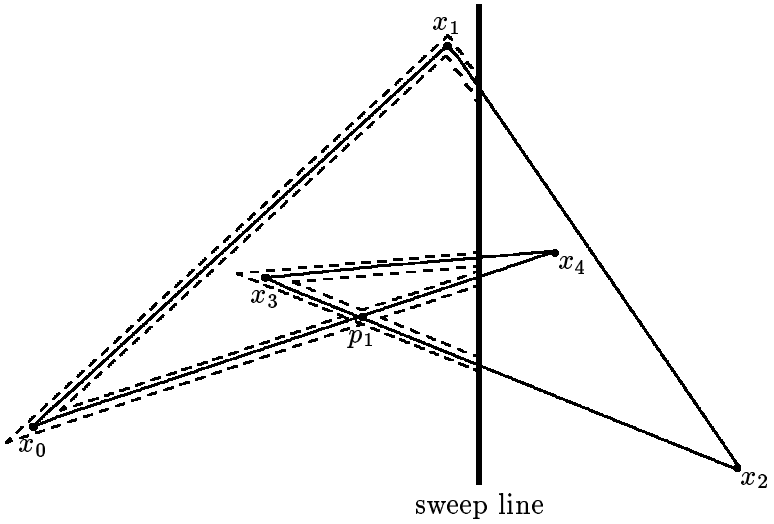


Figure 61.

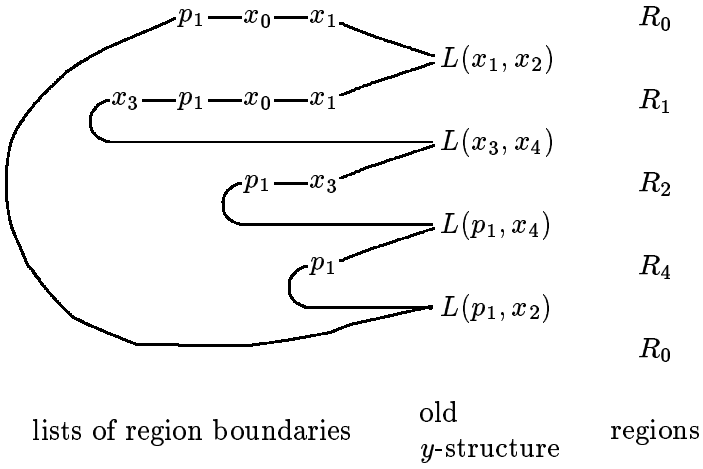


Figure 62.

Finally, for each region we record its boundary in doubly linked lists. In our examples, the boundary of region R_1 consists of vertices x_1, x_0, p_1, x_3, the line segments between them, and parts of the line segments $L(x_1, x_2)$ and $L(x_3, x_4)$. More generally, the region boundaries are stored as follows. With each entry of the former y-structure we associate two doubly linked lists, one for each of the two regions which have the entry (which is a line segment) on their boundary. Each doubly linked list connects two entries of the former y-structure and stores a polygonal chain. This polygonal chain is part of the boundary of one of the regions. More precisely, assume that there is a region R such that k intervals are known to belong to region R. Then $2k$ endpoints $y_1, y_2, \ldots, y_{2k}$ are connected as $y_2 - y_3, y_4 - y_5, \ldots, y_{2k} - y_1$, in order to reflect the fact that the boundary of region R

consists of k polygonal chains, running from y_{2i} to y_{2i+1}, $1 \leq i < k$, and from y_{2k} to y_1. Figure 63 illustrates the case $k = 4$; region R is hatched.

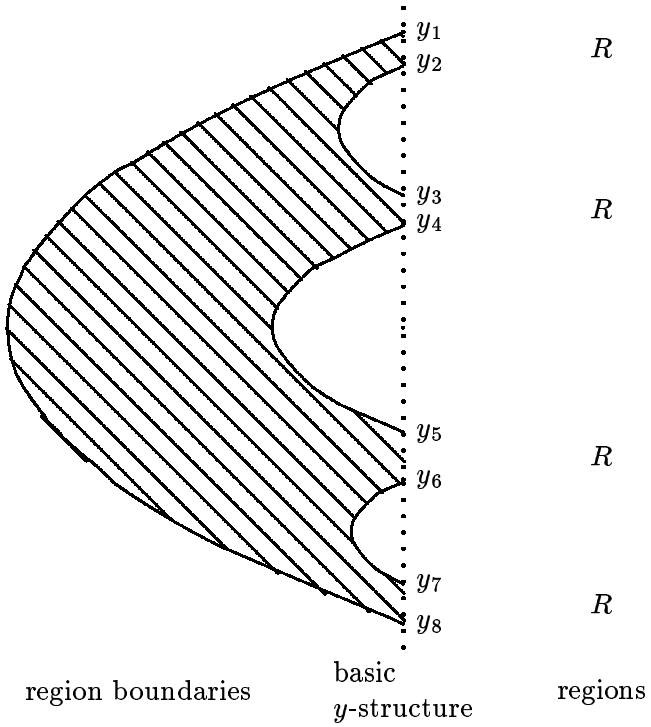


Figure 63.

In our current example, we have the following y-structure for a position of the sweep line between x_3 and p_1.

Thus the boundary of region R_1 consists of the interval between $L(x_0, x_1)$ and $L(x_3, x_4)$ followed by the part of line segment $L(x_3, x_4)$ which extends from the sweep line to vertex x_3, followed by a part of line segment $L(x_3, x_2)$ which extends from vertex x_3 to the sweep line, followed by the interval between $L(x_3, x_2)$ and $L(x_0, x_4)$, followed ... .

This completes the description of the augmented y-structure. The transitions in the plane sweep still remain to be described. Since we sweep a closed curve there are exactly four types of transitions, as illustrated below. Either we scan the common left endpoint of two line segments, the common right endpoint of two line segments, the right endpoint of one line segment and the left endpoint of another or a point of intersection. We refer to the four types as starting point, endpoint, bend, and intersection. For each of the four types we will now describe the additional steps required. We assume without saying that the steps taken in the basic algorithm are carried out.

Case 1: Two starting line segments (start point).

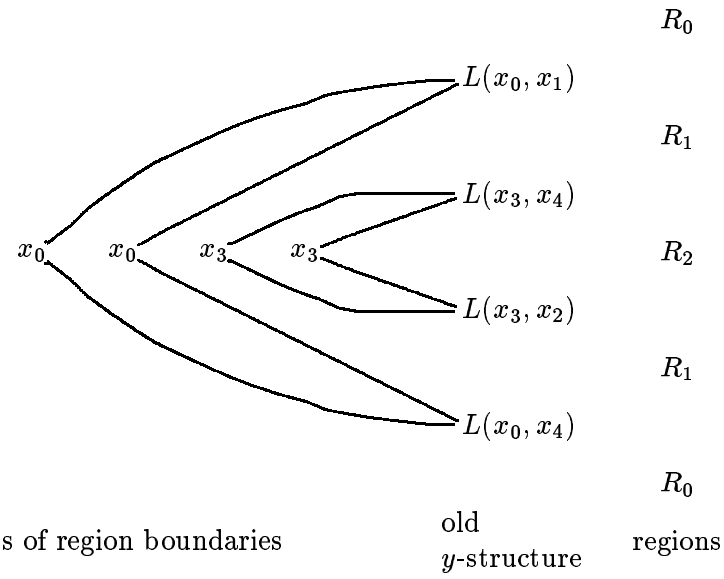


Figure 64.

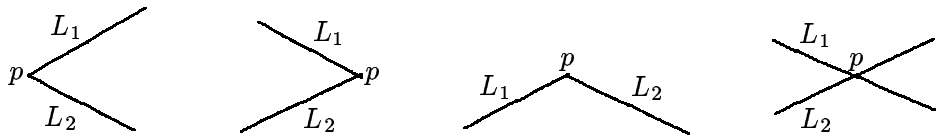


Figure 65.

Let line segments L_1 and L_2 share left endpoint p. We insert line segments L_1 and L_2 into the y-structure and associate a new region with the interval between them. Let L and L' be the two line segments adjacent to p in the y-structure. Then the interval between them is split into two parts both associated with the same region. Furthermore, we record the fact that the part of L_1 extending from the sweep line to p followed by the part of L_2 extending from p to the sweep line belongs to the boundary of R and R_{new} as illustrated in Figure 66.

Case 2: Two terminating line segments (endpoint).
Let line segments L_1 and L_2 share right endpoint p. Let $BLTOP_i$, $BLBOT_i$, $i = 1, 2$ be the boundary list associated with the interval above (below) L_i at entry L_i of the y-structure. Also let R_1, R_2, R_3 be the regions associated with the intervals between L and L_1, L_1 and L_2, and L_2 and L', respectively. Consider region R_2 first. Lists $BLBOT_1$ and $BLTOP_2$ describe part of the boundary of region R_2. We concatenate $BLBOT_1$, point p, $BLTOP_2$ in order to record the fact that the two partial boundaries meet in vertex p. If lists $BLTOP_1$ and $BLTOP_2$ are not identical (a fact which is easily tested in time $O(1)$ by providing direct links between the two endpoints of each boundary list) then no further action with respect to region R_1 is required. If $BLBOT_1$ and $BLTOP_2$ are identical then p is the rightmost point of region R_1 and the scan of region R_1 is completed. We can thus output its boundary.

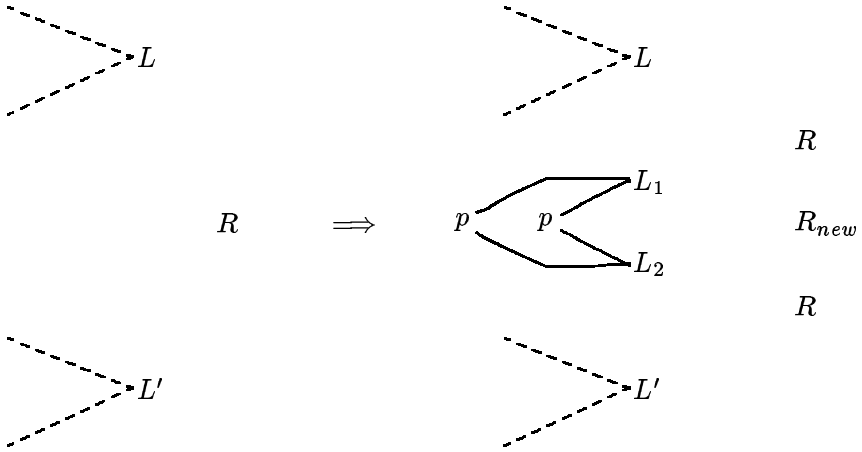


Figure 66.

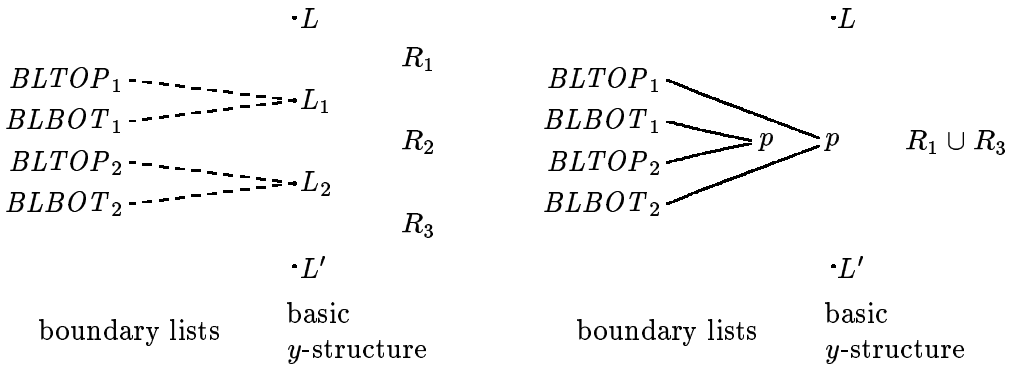


Figure 67.

Let us turn to regions R_1 and R_3 next; $R_1 = R_3$ is possible. Regions R_1 and R_3 merge in point p and we can join their boundaries by concatenating $BLTOP_1$, vertex p, $BLBOT_2$.

Finally, we delete line segments L_1 an L_2 from the y-structure.

Case 3: One ending, one starting line segment (bend).
Let line segment L_1 end in vertex p and let line segment L_2 start in p. Let $BLTOP$ ($BLBOT$) be the boundary list associated with the region above (below) L_1 at entry L_1 of the y-structure. The only action required is to replace L_1 by L_2 in the y-structure and to add point p to lists $BLTOP$ and $BLBOT$.

Case 4: Point of intersection. Let line segments L_1 and L_2 intersect in vertex p. Conceptually, divide L_1 and L_2 into parts L_1', L_1'', and L_2', L_2'' such that L_1', L_2' end in p and L_1'', L_2'' start in p. This shows that case 4 reduces to case 1 followed by case 2.

We have now arrived at a complete description of the algorithm. The analysis is straightforward. At every transition point, we spend time $O(1)$ in addition to the

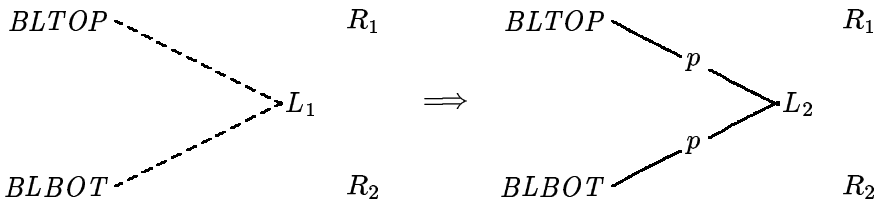


Figure 68.

time spent in the basic algorithm in order to manipulate a few linked lists. Thus running time is still $O((n+s)\log n)$ where s is the number of intersections of line segments. We summarize in

Theorem 2. *let $x_0, x_1, \ldots, x_{n-1}$ be a sequence of points in the plane. Then the simple regions defined by the line segments $L_i = L(x_i, x_{i+1})$, $0 \leq i \leq n-1$, can be determined in time $O((n+s)\log n)$ and space $O(n)$, where s is the number of pairwise intersections of segments L_i.*

Theorem 2 can be refined and varied in many ways, cf. Exercise 25 to 28. For example, we might start with many instead of one closed curve or we might want to compute a function different from intersection, e.g., symmetric difference. The latter variant of the problem arises frequently in the design of integrated circuit.

8.4.2. Triangulation and its Applications

In this section we study the problem of triangulating a simple polygon and, more generally, of decomposing a simple polygon into convex parts. A decomposition into convex parts or triangles is desirable within computational geometry because convex polygons are much easier to handle than general polygons. It is also desirable in other fields, e.g. numerical analysis, although additional restrictions are often imposed, in particular on shape of the convex parts or triangles. For example, one might want to avoid either small or large angles. Apart from the fact that it is interesting in its own right, the triangulation problem also allows us to illustrate techniques for speeding up plane sweep.

Let P be a simple polygon with vertex set $x_0, x_1, \ldots, x_{n-1}$ in clockwise order. A triangulation of vertex set $\{x_0, \ldots, x_{n-1}\}$ is a maximal set of non-intersecting straight line segments between points in this set. Aa triangulation of polygon P is a triangulation of its vertex set such that all edges of the polygon are edges of the triangulation. An **inner triangulation** of P consists of all triangles of a triangulation which are inside P.

In Figure 69 polygon edges are shown solid and triangulation edges are shown dashed.

Throughout this section we use n to denote the number of vertices of polygon P and c to denote the number of cusps (concave angles), i.e., $c = |\{i;\ \angle(x_{i-1}, x_i, x_{i+1}) >$

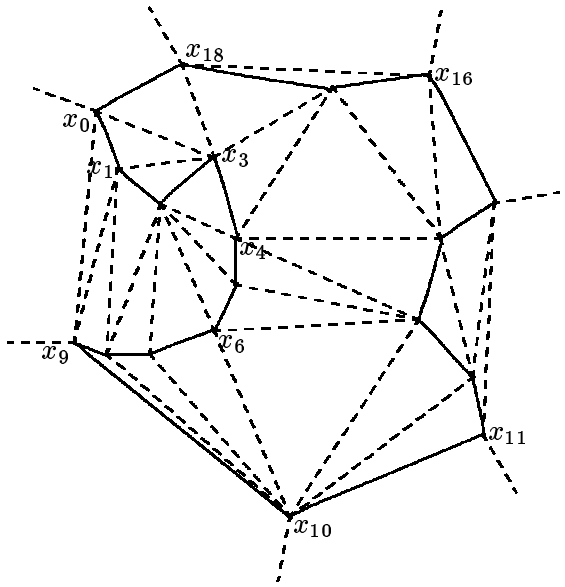


Figure 69.

$\pi\}|$. In our example, we have $c = 9$. We will present an $O(n \log n)$ triangulation algorithm which we later refine to an $O(n + c \log c)$ algorithm.

Theorem 3. *Let P be a simple polygon. Then an (inner) triangulation of P can be constructed in time $O(n \log n)$.*

Proof: We will first show how to construct an inner triangulation; a simple modification of the algorithm will then yield a triangulation. We use plane sweep.

The x-structure contains all vertices of the polygon sorted according to the x-coordinate. As in the decomposition algorithm above we classify vertices in three groups: starting vertices, end vertices and bends. The transition at point p depends on the type of the vertex.

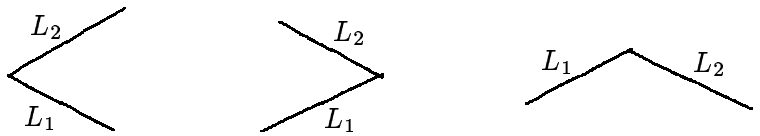


Figure 70.

As always, the y-structure contains the active line segments. Since we sweep a closed curve, the number of active line segments is always even. The active line segments dissect the sweep line into an odd number of intervals which we label out and in alternately. The two infinite intervals are labelled out. The in-intervals

(out-intervals) correspond to regions inside (outside) the polygon. Figure 71 shows the y-structure after processing point x_{18}.

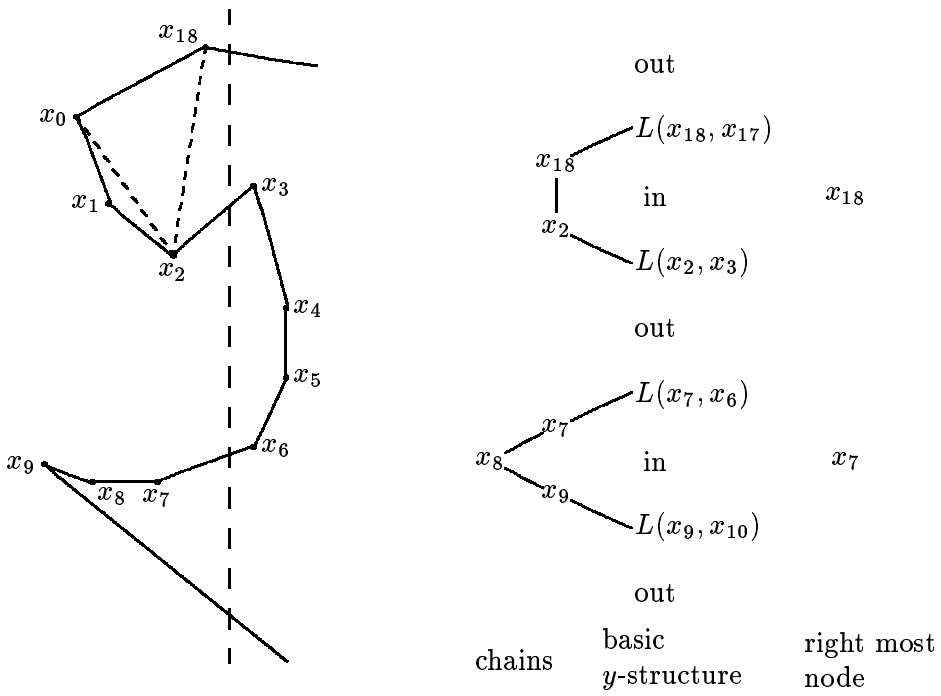


Figure 71.

With every in-interval we associate a polygonal chain, say $v_1, \ldots, v_k$ of polygon vertices. Then v_1 and v_k are endpoints of active line segments, and $L(v_i, v_{i+1})$, $1 \leq i < k$, is a triangulation edge, i.e., either an edge of the polygon P or an edge constructed in the triangulation process. Furthermore, we maintain the invariant
x_i oder v_i? that $\angle(x_i, x_{i+1}, x_{i+2}) \geq \pi$ for $1 \leq i \leq k-2$, i.e., the triangulation of chain $v_1, \ldots, v_k$ cannot be locally extended. Finally, for each in-interval, we provide a pointer to the rightmost node on the chain associated with that interval. note that the x-coordinates of the nodes on a chain decrease as we follow the chain starting at the rightmost node and proceeding towards one of its ends. This follows immediately from the fact that all nodes on a chain (except the two endpoints) are cusps.

We are now ready to give the details of procedure *Transition*. Let p be the point selected from the x-structure.

Case 1: p is a starting point
Let L and L' be the active line segments immediately above and below point p.

Case 1.1: p lies in an out-interval
This case is particularly simple. We split the out-interval between L and L' into three intervals of types out, in, out and associate a chain consisting of node p only with the in-interval. Also, p is the rightmost node of that chain.

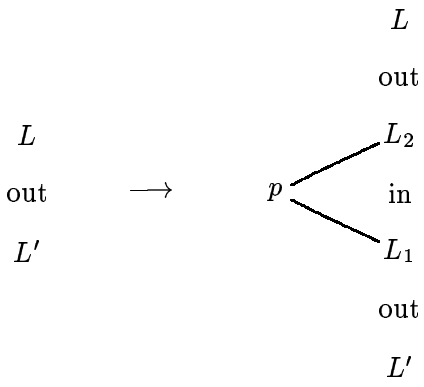


Figure 72.

Case 1.2: p lies in an in-interval
Let $v_1, \ldots, v_k$ be the chain associated with the in-interval and let v_l be its rightmost node. We can certainly add edge $L(v_l, p)$ to the triangulation. Also, we follow the chain starting at v_l in both directions and add edges to the triangulation as long as possible, i.e., until we reach points v_i (preceding v_l on the chain) (and v_j (following v_l)) such that $L(v_i, p)$ can be added to the triangulation but $L(v_{i-1}, p)$ cannot, i.e., until either $i = 1$ or $\angle(v_{i-1}, v_i, p) \geq \pi$. Then we split the in-interval into three intervals of in, out, in and associate chain $v_1, \ldots, , v_i, p$ with the upper in-interval and chain $p, v_j, \ldots, v_k$ with the lower in-interval. Also, p is the rightmost point of both intervals. Figures 73 and 74 illustrate the transition and the effect on the y-structure.

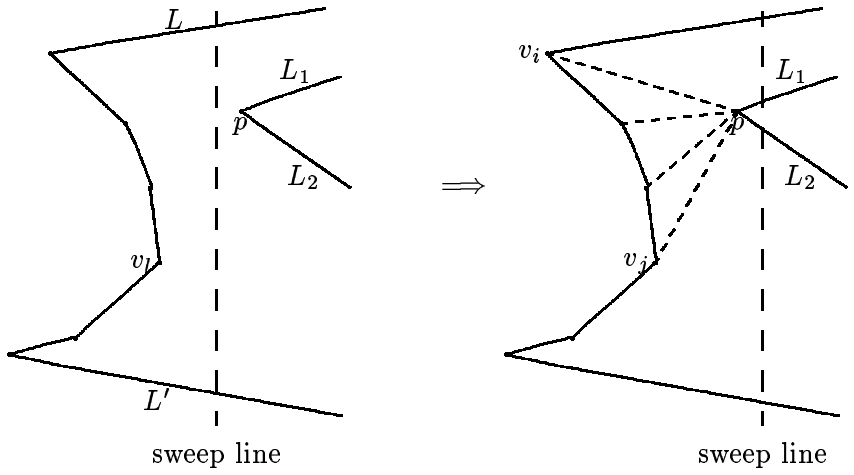


Figure 73.

The rightmost element on each chain is indicated by an arrow. The time complexity of this transition is $O(\log n +$ number of triangulation edges added).

Case 2: p is a bend

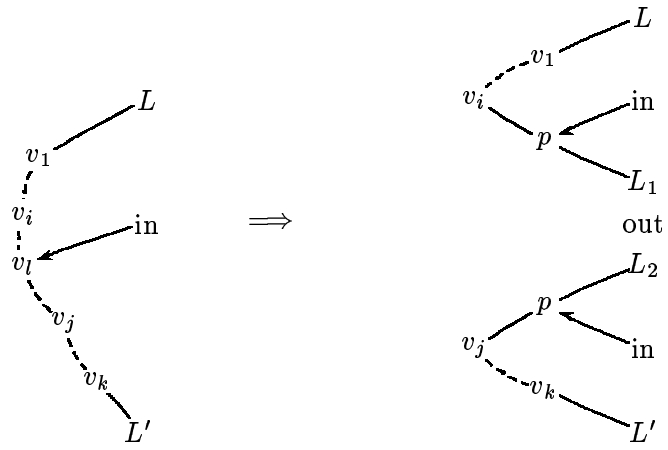


Figure 74.

Let L_1 be the edge ending in p and let L_2 be the edge starting in p. Then L_1 is on the boundary of an in-interval. Let $v_1, \ldots, v_k$ be the chain associated with that in-interval, where v_1 is the other endpoint of line segment L_1. We add triangulation edges $L(p, v_2), \ldots, L(p, v_i)$ until $\angle(p, v_i, v_{i+1}) \geq \pi$ and transform the chain associated with the in-interval into $p, v_i, v_{i+1}, \ldots, v_k$. Also, p is the new rightmost node of the chain. The cost of this transition is $O(1+$ number of edges added to the triangulation). Note that line segment L_1 can be accessed directly, given point p.

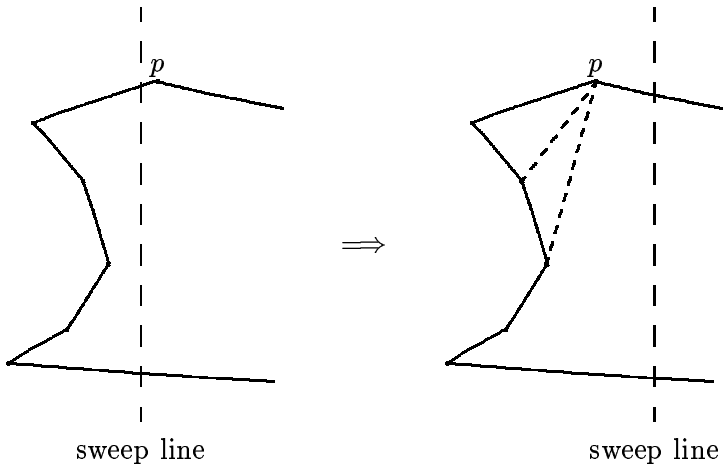


Figure 75.

Case 3: p is an end node
Let L_1 and L_2 be the two line segments ending in p and let L and L' be the line segment adjacent to L_1 and L_2.

Case 3.1: The interval between L_1 and L_2 is an in-interval

Let $v_1, \ldots, v_k$ be the chain associated with the in-interval. We add edges $L(p, v_i)$, $2 \leq i \leq k-1$, to the triangulation and delete line segments L_1 and L_2 from the y-structure. The cost of this transition is clearly $O(\log n +$ number of edges added to triangulation).

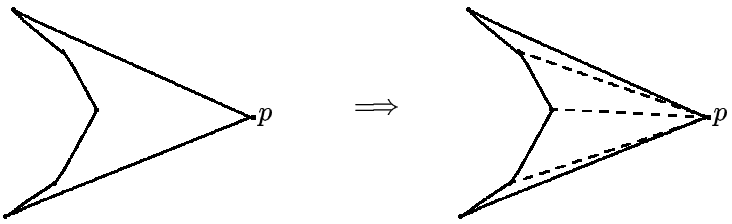


Figure 76.

Finally we prove the correctness of this transition, i.e., we have to show that the edges of the chain and the newly constructed edges are pairwise non-intersecting. This follows from the fact that $p, v_1, \ldots, v_k, p$ is a simple polygon and that $\angle(v_{i-1}, v_i, v_{i+1}) \geq \pi$ for $2 \leq i \leq k-1$.

Case 3.2: The interval between L_1 and L_2 is an out-interval

Let $v_1, \ldots, v_k$ be the chain associated with the in-interval between L and L_1, and let $w_1, \ldots, w_h$ be the chain associated with the in-interval between L_2 and L'. Vertex p can be interpreted as a bend for both chains, i.e., we can add triangulation edges $L(p, v_{k-1}), \ldots, L(p, v_i)$ until $\angle(p, v_i, v_{i-1}) \geq \pi$ and edges $L(p, w_2), \ldots, L(p, w_j)$ until $\angle(p, w_j, w_{j+1}) \geq \pi$. Then we merge both in-intervals by deleting edges L_1, L_2 from the y-structure. Also, we associate chain $v_1, v_2, \ldots, v_i, p, w_j, w_{j+1}, \ldots, w_k$ with rightmost node p with the new in-interval. The cost of this transition is clearly $O(\log n +$ number of edges added to the triangulation).

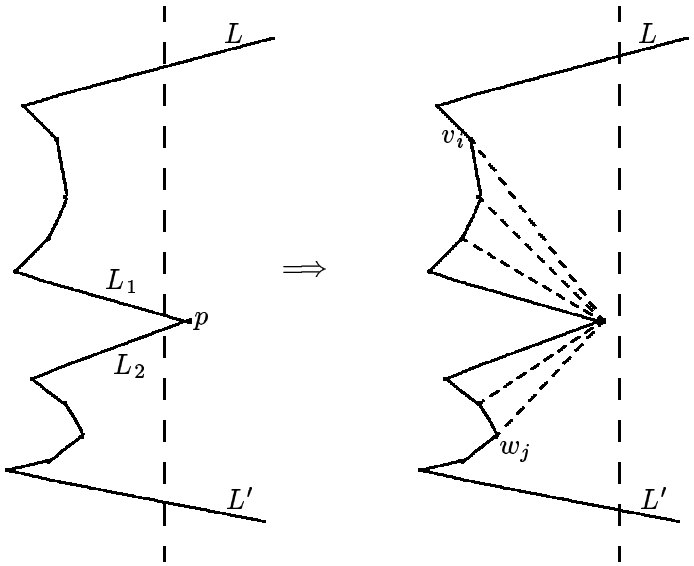


Figure 77.

In summary, we infer that the cost of every transition is bounded by $O(\log n +$ number of edges added to the triangulation). Since there are only n transitions and since the number of edges in the triangulation is at most $3 \cdot n$ (the triangulation is a planar graph) we infer that total cost is $O(n \log n)$. This includes the cost for sorting the n entries of the x-structure. Thus an inner triangulation of a simple polygon P can be constructed in time $O(n \log n)$.

We will now extend the algorithm to the construction of a triangulation. In a preprocessing step we determine the convex hull of polygon P. This takes time $O(n)$ by Section 2, Theorem 2. Next we modify the y-structure and associate chains with in-intervals and out-intervals. The chains associated with the out-intervals describe partial triangulations of the outside of P. Finally, we associate hull edges with the two infinite out-intervals; namely the upper and the lower hull edge which are intersected by the sweep line (cf. Figure-78).

It is now easy to modify the transitions described above such that a triangulation is constructed. Basically, we only have to treat cases 1.1 and 1.2 and 3.1 and 3.2 symmetrically. We leave the details to the reader. ∎

We will next describe an improvement of the triangulation algorithm. The $O(n \log n)$ running time of the algorithm stems from two sources; it takes time $O(n \log n)$ to sort the vertices of the polygon and it takes time $O(n \log n)$ to sweep the plane. The latter statement has to be taken with a grain of salt. One transition at end and starting points do have cost $O(\log n + \ldots)$, transitions at bends have cost $O(1 + \ldots)$. Thus the cost of the sweep is $O(n + s \log s)$, where s is the number of starting and endpoints. The next lemma shows that $s = O(c)$, where c is the number of cusps.

Lemma 2. $s \leq 2 + 2 \cdot c$.

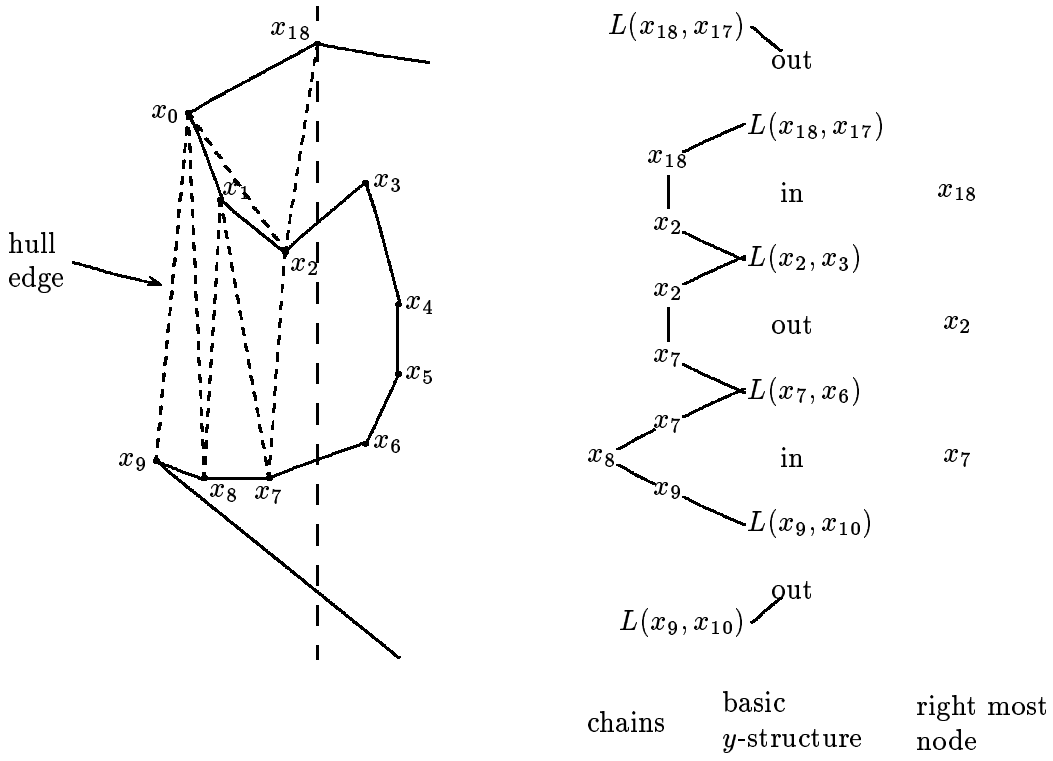


Figure 78.

Proof: Let P be a simple polygon and let s_1 be the number of starting points. Then $s_1 \leq 1 + c$, since any two starting points must be separated by a cusp. A similar argument shows that the number of endpoints is bounded by $1 + c$. Thus $s \leq 2 + 2 \cdot c$. ∎

We could reduce the total cost of the algorithm if we advanced the sweep line through the set of starting and endpoints only. Then the cost of initializing the x-structure would drop to $O(s \log s)$ and hopefully total running time would drop to $O(n + s \log s)$.

Theorem 4. *Let P be a simple polygon with n vertices, s start and endpoints and c cusps. Then P can be triangulated in time $O(n + s \log s) = O(n + c \log c)$.*

Proof: The main idea is to store only starting and endpoints in the x-structure and to give up the strict regimen of the y-structure. Rather, we allow the y-structure to lag behind the sweep line. As before the y-structure stores an ordered set of line segments which defines a set of intervals. For each interval, we have its own local sweep line, some of which might lag behind the global sweep line. The global sweep line refers to the sweep line of our basic algorithm. With each interval we associate a chain and a pointer to the rightmost node on the chain as before. Also, the angles at the vertices of the chain are concave as before.

We require two additional invariants. Consider two adjacent intervals in the y-structure. Then the local sweep lines associated with the two intervals must touch a common edge, namely the edge separating the two intervals in the y-structure. The second invariant refers to the ordering of line segments in the y-structure. Conceptually follow the polygon P starting from each of the line segments stored in the y-structure until the global sweep line is reached. In this way we associate a point on the sweep line with each line segment in the y-structure. We require that the order of the line segments in the y-structure coincides with the order of the associated points on the global sweep line.

Figure 79 illustrates these definitions. Line segments $L_1, \ldots, L_6$ are stored in the y-structure.

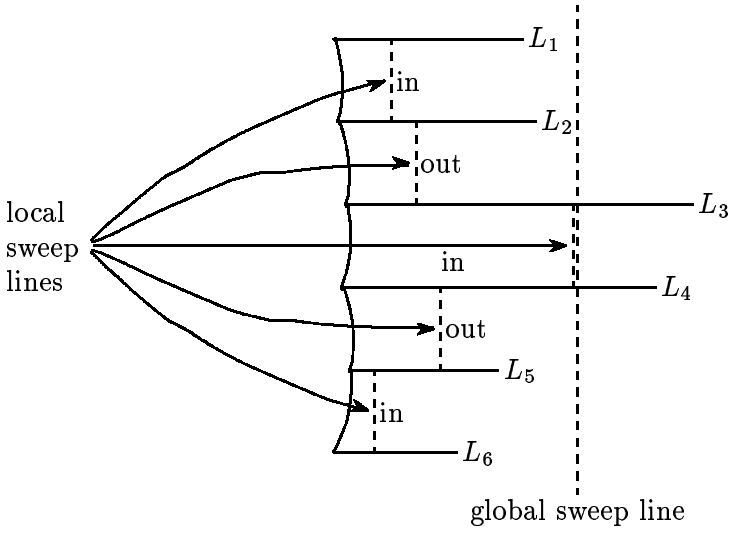


Figure 79.

The x-structure stores the s starting and endpoints in increasing order of x-coordinate. Thus it takes time $O(s \log s)$ to build up the x-structure. Suppose now that we select vertex p from the x-structure and that we want to locate p in the y-structure. Recall that the y-structure is basically a balanced search tree for an ordered set of line segments. When we search for p in the y-structure we compare p with line segments in order to locate p relatively to intervals. Since the line segments stored in the y-structure do not necessarily intersect the sweep line such a comparison is potentially meaningless. We proceed as follows.

Suppose that we have to compare p with line segments L which is stored in the y-structure. Then L borders two intervals I_1 and I_2 with associated chains CH_1 and CH_2. Our immediate goal is to close the gap between the local sweep lines for intervals I_1 and I_2 and the global sweep line. We illustrate this process for interval I_1. We extend the chain on both ends all the way up to the sweep line. Note that only bends are encountered in this process. We handle them exactly in the same way as we did in the proof of Theorem 3. This strategy is correct because

the transition at bends was completely local and since the order of intervals is the same as in the corresponding state of the previous algorithm. Suppose now that we extend chain CH_1 all the way to the global sweep line. Then we also might have to extend the chain above CH_1 in order to ensure that the local sweep lines of adjacent intervals touch the same edge, ... It is important to observe that no structural changes in the tree structure underlying the y-structure underlying the y-structure are necessary. This follows from the invariant about the order of the line segment in the y-structure. Thus the process of determining the position of p relative to an interval takes time $O(1+$ number of edges constructed) and hence the search for p in the y-structure takes time $O(\log s+$ number of triangulation edges constructed). Note that the number of intervals in the y-structure is at most $2 \cdot s$ since intervals only split at start points.

When we have finally determined the position of point p in the y-structure we have also closed the gap between a number of local sweep lines and the global sweep line. In particular, the interval containing p and the two adjacent intervals are processed all the way up to the global sweep line. We can therefore process (start or endpoint) p exactly as we did above.

In summary, there are s transitions. Each transition has cost $O(\log s+$ number of triangulation edges drawn). Since only $O(n)$ triangulation edges are drawn altogether, total running time is $O(n + s \log s) = O(n + c \log c)$by Lemma 2. ∎

We will now turn to applications of triangulation. Our first application is an extension of our linear time algorithm for intersecting convex polygons (Section 1, Theorem 5).

Theorem 5. *Let P be a simple n-gon and let Q be a convex m-gon. Assume that a triangulation of P is available. Then $P \cap Q$ can be computed in time $O(m + n)$.*

Proof: Let T be a triangulation of P. We first extend T to a planar subdivision T' by adding a set of non-intersecting rays starting at the vertices of the convex hull of P, i.e., we also divide the infinite face into triangles. T' can clearly be obtained from T in time $O(n)$. Also, T' has only $O(n)$ edges. Since every edge of T' intersects Q at most twice the number of intersections of edges of T' and edges of Q is $O(n)$.

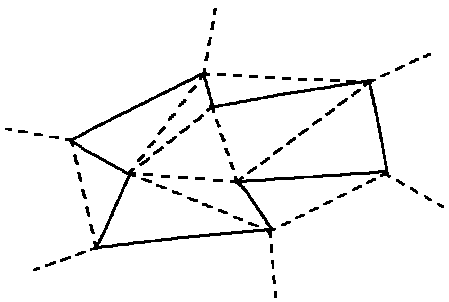


Figure 80.

Let $v_1, \ldots, v_m$ be the vertices of Q. We can certainly find the triangle containing v_1 in time $O(n)$. Also, knowing the triangle containing v_i, we can find all intersections between T' and line segment $L(v_i, v_{i+1})$ in time $O(1 + s_i)$, where s_i is the number of such intersections. We refer the reader to Section 1, Theorem 5 for details. Hence the total time needed to find all points of intersections is $O(m - \sum s_i) = O(m + n)$ by the argument above. ∎

Another application is the decomposition of a simple polygon into a nearly minimum number of convex parts. Let P be a simple polygon and let D be a subset of the set of line segments defined by the vertices of P. D is a decomposition into convex parts if the line segments in D are pairwise non-intersecting, if they are all in the interior of P, and if all the regions defined by $P \cup D$ are convex. Figure 81 shows a decomposition into a minimal number of convex parts; the edges of D are dashed. A decomposition into a minimal number of convex parts can be computed by dynamic programming (Exercise 31) in time $O(n^2 \cdot c^2)$. Here c is the number of cusps.

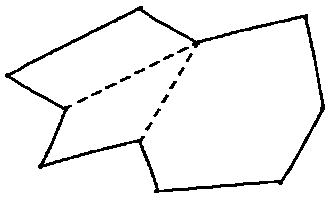


Figure 81.

Theorem 6. *Let P be a simple n-gon, let T be an interior triangulation of P. Let OPT be the minimal number of parts in any convex decomposition of P. Then a decomposition with at most $4 \cdot OPT - 3$ parts can be constructed in time $O(n)$.*
Proof: Observe first that $OPT \geq 1 + \lceil c/2 \rceil$, since at least one partitioning edge is necessary for each cusp. We will partition P into at most $2 \cdot c + 1 \leq 4 \cdot OPT - 3$ parts as follows.

Go through the edges in T in an arbitrary order. Delete an edge of T if this deletion does not create a concave angle in any part of the decomposition. We claim that the decomposition obtained in this way has at most $2 \cdot c + 1$ parts.

This can be seen as follows. Consider any of the remaining edges. Assign any such edge to one of its endpoints. Edge e may be assigned to endpoint p if the removal of e creates a concave angle at p. Of course, p is a cusp of the original polygon. Assume for contradiction that three edges are assigned to any cusp p. Let $e_1, e_2, \ldots, e_5$ be the polygon edges and the three assigned edges in cyclic order. Then $\angle(e_1, e_3) > \pi$ and $\angle(e_3, e_5) > \pi$ and hence $\angle(e_1, e_5) > 2\pi$, contradiction. hence at most two edges are assigned to any cusp and thus we have constructed a decomposition with requiring at most $2 \cdot c + 1$ parts. ∎

Another application of triangulation are visibility problems. Let P be a simple polygon and let m be a point in the interior of P. Let *Vis* be the set of points

visible from m, i.e., $Vis = \{v;\ L(m,v)$ does not intersect $P\}$. Then *Vis* is clearly a simple polygon.

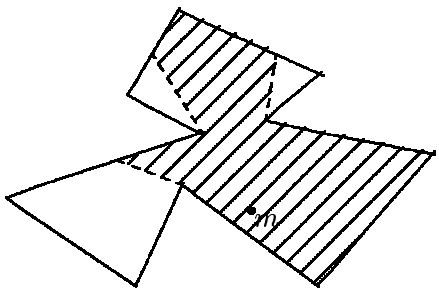


Figure 82.

The goal is to compute the vertices of *Vis* in clockwise order. This task is easily solved in linear time, given the following triangulation of P. Consider the following graph G. Its vertices are the triangles of the triangulation; two triangles (= nodes) are connected by an edge if they share a non-trivial edge in the triangulation. We claim that G is a tree. This can be seen as follows. Let t_1, t_2 be triangles. Since the vertices of t_1 are vertices of P, removal of t_1 splits P into exactly three disjoint parts. Exactly one of these parts contains t_2. Hence the first edge on a simple path from t_1 to t_2 in G is uniquely defined and hence (by induction) there is a unique simple path in G from t_1 to t_2. Thus G s a tree.

Let t be the triangle containing m. Make t the root of G and compute visibility top-down. More precisely, let e be an edge of some triangle t'. If e is an edge of t then all of e is visible. If e is not an edge of t then let e' be the immediate predecessor of e on the path to the root. Knowing the visible part of e' it is trivial to compute the visible part of e in time $O(1)$. Thus all visible parts can be computed in time $O(n)$. We summarize in

Theorem 7. *Let P be a simple n-gon and let m be a point in the interior of P. Given an inner triangulation of P one can compute the visibility polygon with respect to m in linear time $O(n)$.*

8.4.3. Space Sweep

In this section we want to illustrate that the sweep paradigm can also be used to solve three-dimensional problems. More precisely, we will show how to compute the intersection of two convex polyhedra P_0 and P_1 in time $O((n_0+n_1)\cdot\log(n_0+n_1))$, where n_0 (n_1) is the number of vertices of P_0 (P_1). An alternative algorithm having the same time bound is discussed in Exercise 2. The success of plane sweep in two dimensions stems from the fact that it turns a two-dimensional problem into

a one-dimensional problem. In other words, the y-structure is linearly ordered in a natural way and hence balanced search trees can be used successfully in order to maintain it. In three-dimensional space the situation is more complicated in general. The intersection of the sweep plane with the geometric objects is an arbitrary planar subdivision which changes dynamically as the sweep plane advances. Unfortunately, the techniques developed in Section 3.2 for maintaining dynamic planar subdivisions are not strong enough to handle the sweep of general three-dimensional objects. However, there is one special case which we can handle: the sweep of convex polyhedra. The intersection of the sweep plane with a convex polyhedra is essentially a convex polygon, and convex polygons behave appropriately as we saw in Section 1.

W.l.o.g. let no two vertices of one of the two polyhedra have equal x-coordinates. We assume throughout this section that all faces of polyhedra P_i, $i = 1, 2$, are triangles. To achieve this, all polygonal faces are partitioned into co-planar triangles by drawing **improper edges** from their respective point of maximal x-coordinate. Also, we assume the following representation of polyhedra. For each vertex, we have the list of incident edges in clockwise order, for each edge we have pointers to endpoints and to the adjacent faces and for each face we have the list of edges in clockwise order.

Theorem 8. *Let P_0 and P_1 be convex polyhedra with n_1 and n_2 vertices respectively, let $n = n_1 + n_2$. Then convex polyhedron $P_0 \cap P_1$ can be computed in time $O(n \log n)$.*

Proof: $P_0 \cap P_1$ is a convex polyhedron. We divide the set E of edges of $P_0 \cup P_1$ into two disjoint classes E_1 and E_2. E_1 consists of all edges of $P_0 \cap P_1$ which lie on the surface of P_0 and P_1 and E_2 is the remaining set of edges of $P_0 \cap P_1$. Each edge in E_2 is (part of) an edge of P_0 or P_1. Each edge in E_1 is the intersection of a face of P_0 and a face of P_1. Note however, that edges in E_1 can also be part of *edges* of P_0 or P_1.

In the example of Figure 83 the edges in E_1 (E_2) are shown as wiggled (heavy) lines. The edges in E_1 are naturally grouped into connected components (if two edges share an endpoint then they belong to the same component). Our example deals with only one component. We compute $P_0 \cap P_1$ in a two step process. At the first step we will compute at least one point (on an edge) of each component by space sweep and at the second step we will use these points as starting points for a systematic exploration of $P_0 \cap P_1$. We denote the problem of computing at least one point of each component by ICP'. We assume that each point in the solution set to ICP' is given as the intersection of an edge of P_i with a face of P_{i-1}, $i = 0$ or $i = 1$.

Lemma 3. *Given a solution to ICP' one can compute $P_0 \cap P_1$ in time $O(n)$.*

Proof: The basic idea is to explore $P_0 \cap P_1$ starting from the solution set to ICP' by any of the systematic methods for exploring graphs. More precisely, we proceed

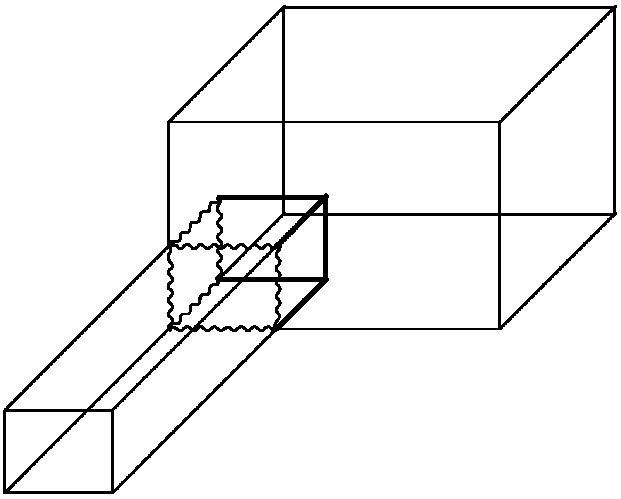


Figure 83.

as follows. Let S be the solution set to ICP'. We run through the points in S in turn. for every $x \in S$, we construct the set of edges in $E_1 \cup E_2$ incident to x and arrange them in queues Q_1 and Q_2, respectively. Edges in E_1 are given as the intersection of a face of P_0 and a face of P_1. These faces are not co-planar. Edges in E_2 are part of an edge of P_0 or P_1. We represent them by a pointer to an edge of P_0 or P_1 and the set of their endpoints which are already constructed. Note that an edge in E_2 shares either two or one or zero endpoints with edges in E_1. The other endpoints are vertices of P_0 or P_1. Having processed the points in S we construct E_1 by considering the edges in Q_1 in turn. For each edge removed its/their? from Q_1, we construct all edges in $E_1 \cup E_2$ incident to their endpoints and add them to Q_1 or Q_2, respectively (if they have not been added before). In this way all edges in E_1 are found. Furthermore, Q_2 contains all edges in E_2 which share at least one endpoint with an edge in E_1. We now construct all edges in e_2 by regarding the edges in Q_2 in turn. For each edges removed from Q_2, we have processed either one or two of its endpoints. If only one endpoint has been processed then we add all edges incident to the other endpoints (which is a vertex of P_0 or P_1) to Q_2 (if they were not added before). Note that all endpoints of edges in E_2 which are not vertices of P_0 or P_1 were constructed when processing S or computing E_1.

Net we have to describe how to construct all edges in $E_1 \cup E_2$ incident to a node x. Special care has to be taken when x lies on co-planar faces D_0 of P_0 and D_1 of P_1. We therefore treat this case first. Let F_0 (F_1) be the convex polygon formed by the union of all faces of P_0 (P_1) which are co-planar to D_0 (D_1). We intersect F_0 and F_1 in time $O(\deg_0(F_0) + \deg_1(F_1))$ using the methods of Section 1 and process all vertices of the intersection as described in one of the cases below. Also, all the edges of the intersection are added to E_1; they do not have to be added to Q_1. In the sequel, all points of S which belong to either F_0 or F_1 can be ignored. We achieve this by marking all faces comprising F_0 and F_1 as done before. The special

treatment of co-planar faces is required for the following reason. If co-planar faces were treated like other faces it is conceivable that a large number of intersections would be discovered which are in the interior of a face of $P_0 \cap P_1$ as illustrated in Figure 84. The diagram shows co-planar faces of P_0 (——) and P_1 (- - -) and their inessential intersections.

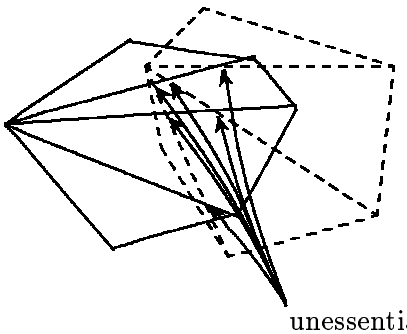


Figure 84.

This finishes the description of the treatment of co-planar faces. We return now to the discussion on the construction of all edges in $E_1 \cup E_2$ incident to a point x. Let x be given as the intersection of an edge e bordering faces F' and F'' of P_i and face F of P_{i-1}. We have to distinguish several cases.

Assume first that x is a vertex of either P_0 or P_1, say a vertex of P_j. Then x lies either in the interior of a face of P_{1-j} or on an edge of P_{1-j} or is a vertex of P_{1-j}. If x lies in the interior of a face, say D, we first check whether D is co-planar with one of the faces of P_j. If so, and if the faces concerned are not "done", we use the method described above. In addition (otherwise, respectively) we can certainly compute all edges in $E_1 \cup E_2$ incident to x in time $O(\deg_j(x))$ by intersecting D with all the faces of P_j incident to x. We use $\deg_j(v)$ to denote the degree of x in polyhedron P_j. In general, there will be two edges in E_1 incident to x and a number of edges in E_2 as indicated in Figure 85.

$\deg_j(x)$?

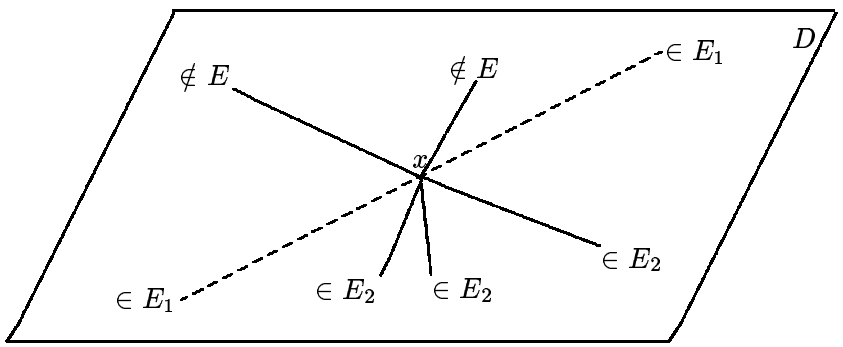


Figure 85.

If x lies on an edge of P_{1-j} but is not a vertex of P_{1-j} then x lies on the boundary of exactly two faces of P_{1-j}. Basically, we only have to intersect both of them with the faces of P_j incident to x. Again, time $O(\deg_j(v))$ suffices. So let us finally assume that x is a vertex of both polyhedra. Let r be a ray extending from x into the interior of P_0, and let D be a plane perpendicular to r and intersecting r in a point different from x. Let $e_1, \ldots, e_m$ be the edges of P_0 incident to x in clockwise order, extended to rays. Convexity implies that all these rays intersect D; the points of intersection define a convex polygon C_0. Also extend the edges of P_1 incident to x to rays; two adjacent rays bound an infinite "triangular" face. Intersecting these faces with plane D either yields a convex polygon C_1 or an open convex polygonal segment C_1 as shown in Figure 86 or the empty set.

x

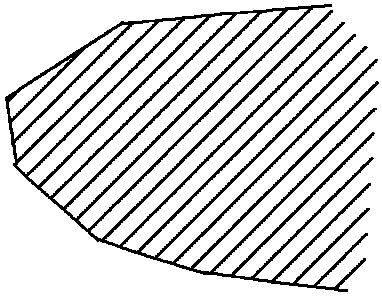


Figure 86.

Applying the methods of Section 1, we can compute the intersection of C_0 and C_1 in time $O(\deg_0(x) + \deg_1(x))$. Having determined the intersection of C_0 and C_1 it si simple to determine all edges of E_1 (E_2) incident to x.

Again, it is possible to detect co-planar faces in this process. They are treated as before. Additional time O(sum of degrees of newly intersected co-planar polygons) is sufficient.

In all three cases we mark vertex x in the appropriate polyhedron/polyhedra. This completes the discussion of x being a vertex of P_0 or P_1.

Assume next that x lies on the boundary of F, i.e., x lies on an edge d of P_{1-i}. Let F''' be the other face of P_{1-i} which has d on its boundary. If none of F or F''' is co-planar to F' or F'' then the structure of $P_0 \cap P_1$ in the vicinity of x is readily computed in time $O(1)$ by intersecting the four faces F, F', F'' and F'''. Otherwise we proceed as described above.

Let us finally assume that x lies in the interior of face F. Then x has to be a proper intersection, and F cannot be co-planar with either F' or F''. $F \cap F'$ and $F \cap F''$ belong to E_1, and the part of e which is directed to the inside of F belongs to E_2.

This completes the description of the construction of all edges in E_1 and E_2 which are incident to a node in the solution set to ICP'. The construction of all such edges clearly takes time $O(n)$. Handling co-planar faces can also be done in total time $O(n)$ since each such face is explored only once.

We use these edges as the basis for the exploration of $P_0 \cap P_1$. First we completely explore E_1.

```
(1)   for i ∈ {0,1}
(2)   do Q_i ← subset of edges of E_i determined by processing the solution set
                 to ICP' as described above
(3)   od;
(4)   while Q_1 ≠ ∅
(5)   do remove an arbitrary edge e from Q_1; e is given as the
          intersection of a face F_0 of P_0 and a face F_1 of P_1;
(6)       let x, y be the two endpoints of e;
(7)       for z ∈ {x, y}
(8)       do if z was not visited before
(9)          then find all edges in E_1 and E_2 incident to z as described above
                  and add them to Q_1 and Q_2 respectively
(10)         fi
(11)      od
(12)  od
```

Program 12

We claim that Program 12 explores all remaining edges in E_1 in time $O(n)$. Note first that Q_1 is initialized with at least one edge of every component of E_1. Hence all remaining edges in E_1 are explored and each edge is explored at most twice, once from either side. We turn to the time bound next. We infer from the preceding discussion that the total time spent in line (9) is $O(n)$. Furthermore each execution of lines (5) and (6) takes time $O(1)$ since faces are triangles. Line (8) remains to be considered. Point z is either a vertex of P_0 or P_1 or lies on an edge of P_0 or P_1. Also, at most two vertices of $P_0 \cap P_1$ can lie on any edge of P_0 or P_1. Hence, if we mark visited vertices of P_0 or P_1 and associate other vertices of $P_0 \cap P_1$ with the edge of P_0 or P_1 on which the vertex lies, then the test in line (8) takes time $O(1)$. Since the number of edges in $P_0 capP_1$ is $O(n)$ the time bound follows.

We finally have to explore the edges in E_2. Recall that every edge in E_2 is part of an edge of either P_0 or P_1. Also note that all endpoints of edges in E_2 which are not vertices of P_0 or P_1 have been determined at this point. Furthermore, all edges in E_2 which have such an endpoint belong to Q_2 at this moment. It is now easy to find the remaining edges in E_2 in time $O(n)$. We leave the details to the reader. ∎

We will now show how to solve ICP' by space sweep. As before, we have two structures. The x structure contains all vertices of P_0 and P_1 in order of increasing x-coordinate. the yz-structure, which replaces the y-structure, stores the status of the sweep. It stores, for each of the two polyhedra, a **crown** which represents the intersection of the polyhedron and the sweep plane.

Let P_i, $i = 1, 2$, be one of the polyhedra. Let e_j, $0 \leq j < n_i$, be the edges of P_i which are intersected by the sweep plane in cyclic order. Here, two edges are adjacent, if they bound the same face. A **prong** is the portion of a face bounded by two consecutive edges e_j and $e_{j+1(\pmod{n})_i}$ and, to the left, by the sweep plane. The set of all prongs is called the **crown**. We also call edges e_j, $0 \leq j < n_i$, the **forward edges** of the crown, the edges connecting the intersections of the e_j's with the sweep plane the **base edges** and the remaining edges the **prong edges**. Prong edges connect tips of prongs.

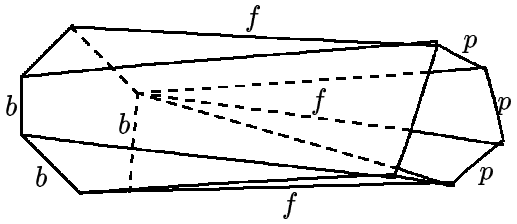


Figure 87.

Figure 87 illustrates these definitions. The different types of edges are indicated by characters b, f and p. We store a crown in a blanced tree as follows. For polyhedron P_i, we select an axis line L_i; for instance the line connecting the vertices of P_i with minimal and maximal x-coordinate, respectively. Let $p \in \mathbb{R}^3$. The cylindrical coordinates (x, *alfa*, *radius*) of p with respect to L_i are defined as follows. First,, x is the x-coordinate of point p. Second, pair (*alfa*, *radius*) forms the polar coordinates in the plane, say E, which goes through p and is parallel to the yz-plane. The origin of the polar system is the intersection of E and L_i and the angle is measured against a fixed direction in that plane; the y-direction. A forward edge of the crown C_i of P_i is represented by the cylindrical coordinates of its endpoints with respect to L_i, $i = 0, 1$, i.e., by a six-tuple (x_0, $alfa_0$, r_0, x_1, $alfa_1$, r_1). If p is a point of this forward edge with x-coordinate x, $x_0 \leq x \leq x_1$, then the pair ($alfa$, r) corresponding to p can be computed in time $O(1)$. Next observe, that although alfas and radii change with x, the cyclic order of forward crown edges remains invariant between transitions. Hence we can store forward edges in a balanced tree, organized with respect to the alfas. For correctness, we assume a $(2, 4)$-tree. Figure 88 illustrates the radial representations of the crown.

We are now ready for the space sweep algorithm. The global structure of the algorithm is as given in the introduction of Section 4. A transition at vertex p of polyhedron P_i is performed as described in Program 13.

Before we analyze the running time of the algorithm we show its correctness.

Lemma 4. *The algorithm of Program 13 correctly solves ICP'.*

Proof: It is clear that a set of intersections is computed. Thus we only need to show that at least one point of each component will be reported.

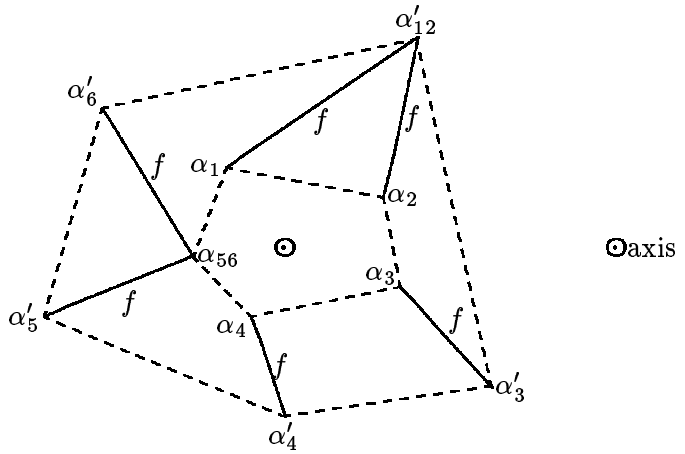


Figure 88.

```
(1)   procedure Transition(p, i):
(2)       update crown C_i of P_i;
(3)       for every face F of P_i having a starting edge at P
(4)       do intersect the starting edges of F with the crown C_{i-1} of polyhedron P_{1-i}
              and report all intersections;
(5)          if no intersection is found
(6)          then choose one forward edge of C_{1-i} and intersect with all faces
                      of P_i which are co-planar with F, report intersections if any
(7)          fi
(8)       od
(9)   end.
```

Program 13

Let S be any component, and let v be a vertex of S with minimal x-coordinate. Clearly y is the intersection of an edge e of polyhedron P_i with a face of polyhedron P_{1-i}. If v is not reported then e must start before F (in the sweep order). Consider the state of the sweep just after the starting vertex p of F was encountered. At this point edge e is a forward edge of the crown C_i of P_i.

Trace (conceptually) component S in the polygon F_p containing face F and crown C_i starting at pont v. Two cases may arise:

Case 1: We are not able to trace S completely i polygon F_p and crown C_i.
Then we either hit a bounding edge of F_p before a base or prong edge of C_i or vice versa. In the former case there is a bounding edge, say e', of F_p which intersects a prong, say PR, of C_i. The intersection of e' and PR was either detected when processing p (if p is the starting edge of e') or will be detected when processing the starting point of e'. Note that PR is still a prong of C_i at this point. In the latter case, we must hit a prong edge, say e'', of C_i. This follows from the fact that v is

in front of the sweep plane and has minimal x-coordinate among all points in S. When the sweep advances to the starting point of e'' polygon F_p is still part of the crown of P_{1-i}. Hence we will pick up the intersection of e'' and F at that point.

Case 2: We are able to trace S completely in F_p and crown C_i.
Since F_p is part of a plane, and since the prongs of C_i are parts of planes S must be a closed curve and therefore runs through all forward edges of C_i. Hence *all* forward edges of C_i intersect F_p, and a point of S is found in line (6) of *Transition*. ∎

Let us now examine the time required for different actions of *Transition*.

Lemma 5. *The updating of crowns P and Q can be done in total time $O(n+m)$.*

Proof: Consider updating C_i at a transition vertex v. Let c_1 edges end in vertex v and let c_2 edges start at vertex v. It is easy to provide direct access to these edges. Also, since we assumed that edges incident to a node are given in cyclic order we know in which order the c_2 edges starting at v have to appear in the crown. Hence we only have to delete c_1 edges (at known positions) from the tree and to insert c_2 edges (at known positions) into the tree. Thus the amortized cost of the insertions and deletions at point v is $O(c_1 + c_2)$ by Section 3.5.3.2. Hence the total time required to update the crown is $O(n+m)$. ∎

Lemma 6. *Let C be a crown with c forward edges and let L be a line which does not intersect the base of C. Then $C \cap L$ can be computed in time $O(\log c)$.*

Proof: The algorithm is a variant of the one used to prove Theorem 2 in Section 1. We view the balanced tree representation of the crown as a hierarchical representation of the crown. let $v_1, \ldots, v_c$ be the intersections of the forward edges with the base plane of the crown in cyclic order and let $w_1, \ldots, w_c$ be the forward endpoints of the forward edges in cyclic order. Note that $v_j = v_{j+1} (= v_{j+2} = \cdots)$ or $w_j = w_{j+1} (= w_{j+2} = \cdots)$ is possible. Let t be the forward endpoint of the axis, i.e., t is the vertex with maximal x-coordinate of the polygon to which crown C belongs. Consider the convex hull of the point set $\{v_1, \ldots, v_c, w_1, \ldots, w_c, t\}$. The faces of the convex hull are the prongs, and the triangles w_j, w_{j+1}, t (with $w_{c+1} := w_1$); $1 \leq j \leq c$. A balanced tree defines a hierarchical representation of this polyhedron as follows. Let D^i be the forward edges which are stored in the i-th level of the tree, $1 \leq i \leq k = O(\log c)$. Then the convex hull of the endpoints of the edges in D^i and point t is the approximation C^i of the crown C. We have $C^k = C$. Also note, that if $e = (v_j, w_j)$ and $e' = (v_h, w_h)$ are adjacent edges in D^i then they determine one or two faces of the approximation C^i, namely either the triangle $v_j, w_j = w_h, v_h$ or the triangle w_j, w_h, t and the quadrilateral v_j, w_j, w_h, v_h which collapses to a triangle if $v_j = v_h$. We can now use essentially the algorithm of Section 1, Theorem 2 to compute $C \cap L$.

We use the following notation. If x is a point of a C^i then a face F of C^{i+1} is **in the vicinity of** x if F and the face of C^i containing x have an edge in D^i

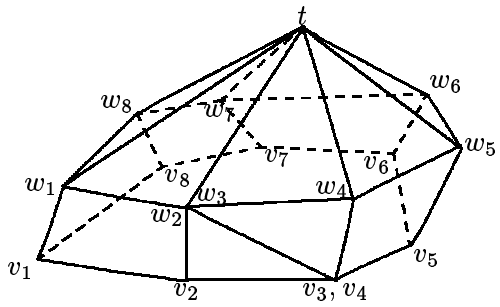


Figure 89.

in common. A point y of C^{i+1} is in the vicinity of x if a face of C^{i+1} to which y belongs is in the vicinity of x. Finally, if x does not intersect C^i then we use p_i to denote the point of C^i which has smallest distance from line L. Note that p_i is not necessarily a vertex of C^i. However, we may assume w.l.o.g. that p_i lies on an edge of C^i. The following lemma captures the heart of the algorithm.

Lemma 7.

a) *If L intersects P^{i+1} but does not intersect P^i then the faces of P^{i+1} which are intersected by L are in the vicinity of p_i.*

b) *If L does not intersect P^{i+1} then p_{i+1} is in the vicinity of p_i.*

c) *If L intersects P^i then the intersections of L and P^{i+1} are in the vicinity of the intersections of L and P^i.*

d) *If L is a line and e is a line segment then the point $y \in e$ closest to L can be computed in time $O(1)$.*

Proof: a) and b): Let E be a plane parallel to L and supporting C^i in point p_i. Then plane E divides the space into two half-spaces one of which contains L and one of which contains C^i. Call the former half-space H. IF L intersects P^{i+1} then $H \cap P^{i+1}$ must be non-empty. By convexity of C^{i+1} only faces in the vicinity of x can possibly intersect H. This proves parts a) and b).

c) Obvious.

d) Let L' be a line containing line segment e. Let L (L') be given by point a (a') and direction $\vec{b}$ ($\vec{b'}$). If L and L' are parallel then the solution is trivial. Otherwise, consider plane E determined by a and directions $\vec{b}$ and $\vec{b'}$. It contains line L and is parallel to L'. Let $\vec{c}$ be the normal direction of the plane, let a'' be the intersection of E and the line of direction $\vec{c}$ through a', and let L'' be the line through a'' parallel to L'. Let x be the intersection of L and L'' and let y be the projection of x onto L' in the direction $\vec{c}$. Then points x, y realize the minimum distance between L and L'. If y is a point of e then we are done. If y does not belong to e then the endpoint of e closest to y realizes the minimum distance. ∎

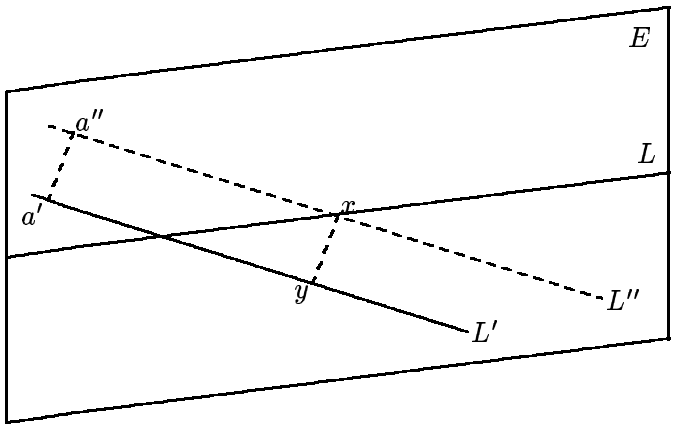


Figure 90.

We infer from the lemma above that $L \cap C$ can be found by following at most two paths in a balanced tree. In each node of the tree $O(1)$ work is required. ∎

Combining Lemmata 3 to 7 we obtain:

Lemma 8. *ICP′ can be solved in time* $O(n_0 \log n_1 + n_1 \log n_0)$.

Proof: The total cost of updating the crowns is $O(n_0 + n_1)$ by Lemma 6. Also, the total cost of line (4) of procedure *Transition* is $O(n_0 \log n_1 + n_1 \log n_0)$ by Lemma 7. The remaining lines of *Transition* have cost $O(n_0 + n_1)$. ∎∎

The goal of this section was to illustrate the use of the sweep paradigm in three-dimensional space and also to provide an algorithm for a basic problem in algorithmic geometry. The alternative algorithm discussed in Exercise 2 is conceptually simpler; in fact the proof of Lemma 7 was inspired by a solution to that exercise. On the other hand, the algorithm given above only needs to store a cross section of the polyhedron and hence uses less space. Also, it is more efficient since the hierarchical representations obtained in a solution to Exercise 2 have depth $c \log n$ for a fairly large constant c. We refer the reader to the related problem discussed in Section 3.2, Lemma 8.

8.5. The Realm of Orthogonal Objects

In this section we explore the geometry of **orthogonal** or **iso-oriented** objects, i.e., objects all of whose edges are parallel to a coordinate axis. Thus in two-dimensional space, we have to deal with points, horizontal and vertical line segments, and rectangles whose sides are axis-parallel. More generally, an iso-oriented object in $\mathbb{R}^d$ is the cartesian product of d real intervals one for each coordinate, i.e., it has the form $\prod_{x=1}^{d}[l_i, r_i]$ where $l_i \leq r_i$ for all i.

As above, we will primarily concentrate on two-dimensional problems. The two major algorithmic paradigms which have been used in this field are plane-sweep and divide-and-conquer. There are many problems which can be solved using either paradigm; the elegance of the solution may however differ drastically as we will see. We devote Section 5.1 to plane-sweep and Section 5.2 to divide-and-conquer algorithms. In the section on plane-sweep we will introduce three new data structures (interval, priority search, and segment trees) which can handle special types of planar range queries more efficiently than range trees (cf. Section 7.2.2); we will also see that some of the algorithms generalize to non-orthogonal objects (Section 5.1.4) and higher dimensions (Section 5.3).

8.5.1. Plane Sweep for Iso-Oriented Objects

Plane sweep is a very powerful technique for solving two-dimensional problems on iso-oriented objects. Recall that iso-oriented objects are points, horizontal and vertical line segments, and axis-parallel rectangles. Typically, the transition points are the points, the endpoints of the line segments and the vertices of the rectangles.

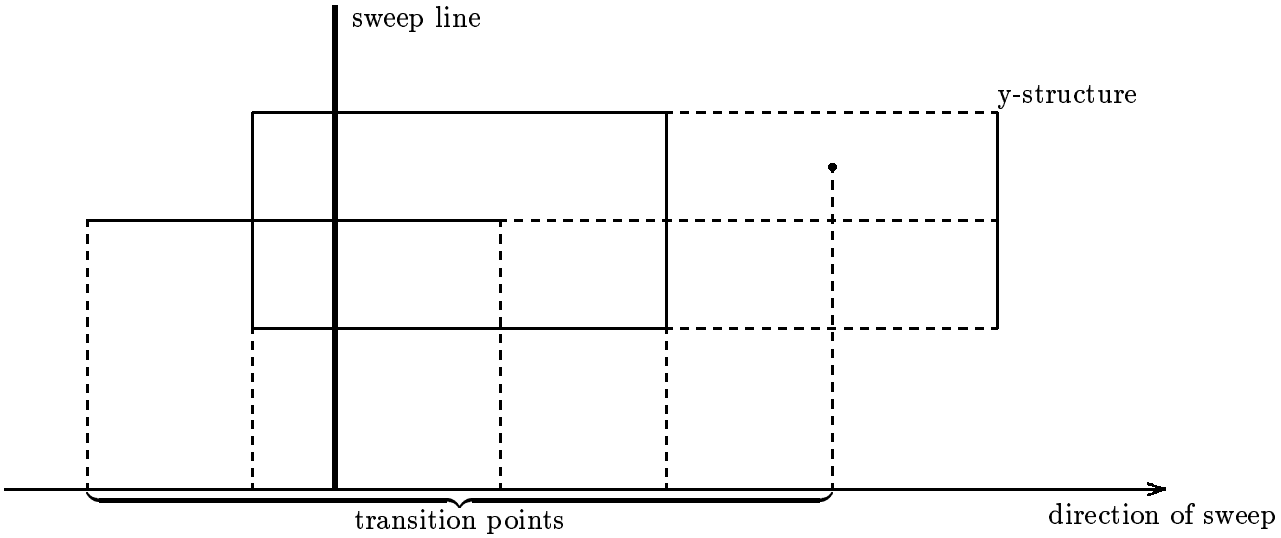


Figure 91.

Also, the y-structure typically contains information about the horizontal line segments and the rectangles which are intersected by the sweep line in its current position. Horizontal line segments correspond to points and rectangles correspond to intervals.

Throughout this section we use the following **notation for intervals** on the real line. Let $x, y \in \mathbb{R}$, $x \leq y$. Then $[x, y]$ denotes the closed, (x, y) the open interval, and $[x, y)$ and $(x, y]$ denote the half-open intervals with endpoints x and y, i.e.,

$$[x, y] = \{z;\ x \leq z \leq y\}$$
$$(x, y) = \{z;\ x < z < y\}$$
$$[x, y) = \{z;\ x \leq z < y\}$$
$$(x, y] = \{z;\ x < z \leq y\}.$$

Note that a closed interval $[x, x]$ with identical endpoints corresponds to a point.

In the course of a plane sweep algorithm for iso-oriented objects one has to maintain dynamic sets of intervals. Whenever the sweep reaches a left (right) side of a rectangle or horizontal line segment one has to insert (delete) an interval into (from) the y-structure. Also, one typically has to query the y-structure at transition points.

Intervals are often conveniently represented as points in two-dimensional space. Then queries about intervals can often be phrased as (special types of) range queries. We illustrate this observation by a few examples. Let $S = \{[x_i, y_i];\ 1 \leq i \leq n\}$ be a set of closed intervals and let $I = [x_0, y_0]$ be a query interval. We can regard S as a set $A(S)$ of points $p_i = (x_i, y_i)$, $1 \leq i \leq n$, in $\mathbb{R}^2$.

Fact 1. *Intervals $[x_0, y_0]$ and $[x, y]$ intersect iff $x \leq y_0$ and $y \geq y_0$*

Hence we can find all intervals in S intersecting the query interval $I = [x_0, y_0]$ by finding all points (x, y) in the associated set $A(S)$ with $x \leq y_0$ and $y \geq x_0$. This corresponds to a range query (cf. Chapter 7) in $\mathbb{R}^2$ with right boundary $x = y_0$ and bottom boundary $y = x_0$ (cf. Fig. 92).

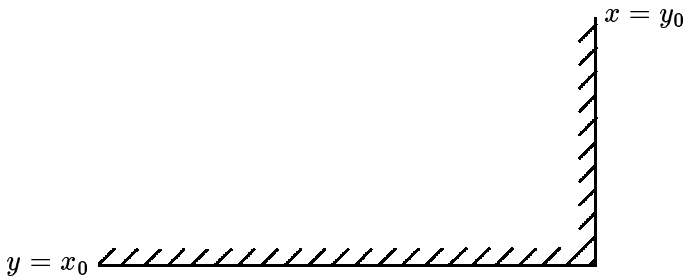


Figure 92.

Of course, we could use range trees (cf. 7.1.2) to store set $A(S)$ but more elegant data structures exist. For the query above we will discuss interval trees in section 5.1.1.

iff **Fact 2.** *Interval* $[x, y]$ *is contained in query interval* $[x_0, y_0]$ *if* $x_0 \leq x \leq y \leq y_0$.

Hence we can find all intervals contained in the query interval $I = [x_0, y_0]$ by finding all points (x, y) in the associated set $A(S)$ with left boundary $x = x_0$ and right boundary $y = y_0$ (cf. Fig. 93). Priority search trees (Section 5.1.2) serve as a good data structure for this type (and more general types) of range query.

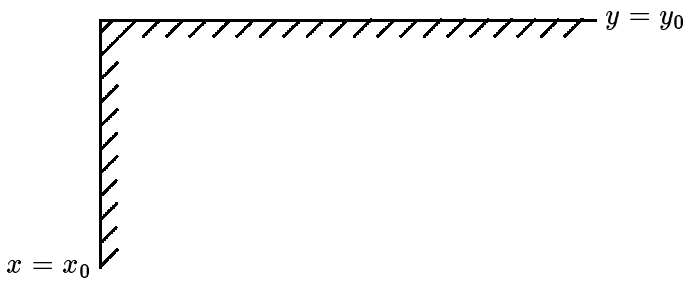


Figure 93.

Remark: Note that all points $(x, y) \in A(S)$ satisfy $y \geq x$. Hence the containment query really searches a bounded region and the intersection query searches an unbounded region. This shows that the two types of queries are different and cannot be transformed into one another by reflection.

A third data structure are segment trees (Section 5.1.3). They are particularly useful if we want to store additional information about intervals. We will use them to solve the measure problem (compute the area covered by the union of a set of rectangles) and a three-dimensional hidden line elimination problem. As to the latter problem we will associate with each rectangle (interval in y-structure) its height ($= z$-coordinate) in order to decide visibility.

All these tree structures are easily explained in the static case. They can all be made dynamic; but at least in the case of interval and segment trees the details of the dynamic version are tedious and the programs become very long. Fortunately, a semi-dynamic version suffices for all plane sweep algorithms. In the semi-dynamic version we consider trees over a fixed universe (usually the y-coordinates of the vertices) and handle only intervals whose endpoints are drawn from the universe. We can handle insertions and deletions; however, the cost of the operations is dictated by the size of the universe and not by the size of the set which is actually stored.

For all three tree structures we use the following common terminology. Let $U \subseteq \mathbb{R}$ be a finite set, called the universe. More generally, U might be any finite ordered set. All trees which we will consider are based on leaf-oriented binary search trees for U, i.e., on binary trees with $|U|$ leaves. The leaves correspond to the elements of ordered set U from left to right (cf. Section 8.5.2). An interior node has at least three fields: pointers to the left and right son and a split field, which is a real number. Thus

```
type node = record ...
                    case stat: status of
                        leaf: (...)
                        nonleaf: (split: real;
                                  lson, rson: ↑node;
                                  ...)
                end
```

where **type** *status* = (*leaf*, *nonleaf*) and the dots indicate additional fields. As usual the split fields direct the search, i.e., the split field of node v is at least as large as all leaves in the left subtree and smaller than all leaves in the right subtree.

For every node v of the tree, we define its xrange $xrange(v)$ as follows. The xrange of the root is the half-open real line, i.e., $xrange(root) = (-\infty, +\infty]$, and if v is the left (right) son of w then $xrange(v) = xrange(w) \cap (-\infty, split(w)]$ ($xrange(v) = xrange(w) \cap (split(w), +\infty]$). Note that a search for $x \in \mathbb{R}$ goes through node v iff $x \in xrange(v)$.

Remark: In the section on segment trees we will slightly deviate from these definitions. There, the split field will be a pair consisting of a real number denoted $split(v)$ and an indicator $Ind(v) \in \{<, \leq\}$. If $Ind(w) = \leq$ then $xrange(v)$, v a son of w, is defined as above. If $Ind(w) = <$ then a search for x proceeds to the left son of w only if $x < split(w)$ and $xrange(v) = xrange(w) \cap (-\infty, split(w))$ ($xrange(v) = xrange(w) \cap [split(w), +\infty]$) for v being the left (right) son of w.

For the description and analysis of the query algorithms in interval, priority search, and segment trees we need some additional notation. Let $I = [x_0, y_0]$ be a query interval. We define node sets P, C and C_{max} with respect to query interval I.

$$P = \{v;\ xrange(v) \cap [x_0, y_0] \neq \emptyset \text{ and } xrange(v) \not\subseteq [x_0, y_0]\}$$

$$C = \{v;\ xrange(v) \subseteq [x_0, y_0]\}$$

$$C_{max} = \{v;\ v \in C \text{ and } father(v) \notin C\}$$

Lemma 1. *Let T be a search tree of height h and let $[x_0, y_0]$ be a query interval. Then*

a) $|P| \leq 2 \cdot h$

b) $|C_{max}| \leq 2 \cdot h$

c) $|C| \leq 2 \cdot (\text{number of leaves } v \text{ with } xrange(v) \subseteq [x_0, y_0])$.

Proof: a) Note first that $v \in P$ implies $father(v) \in P$ since $xrange(v) \subseteq xrange(father(v))$. Thus P consists of a set of paths in tree T; this explains the use of letter P. Note next that $v \in P$ and hence $xrange(v) \cap [x_0, y_0] \neq \emptyset$ and $xrange(v) \not\subseteq [x_0, y_0]$ implies $x_0 \in xrange(v)$ or $y_0 \in xrange(v)$; here we also took into account that $xrange(v)$ is an interval. Thus $v \in P$ implies that v either lies on the path from the root to the leaf v_0 with $x_0 \in xrange(v_0)$ or on the path to the leaf w_0 with $y_0 \in xrange(w_0)$. Also, P consists of an initial segment of these paths. This proves $|P| \leq 2 \cdot h$.

b) The bound on the size of C_{max} is derived as follows. First, if $v \in C_{max}$ then v's brother does not belong to C_{max}. Also, if $v \in C_{max}$ and hence $v \in C$ and $father(v) \notin C$ then $xrange(father(v)) \not\subseteq [x_0, y_0]$ and $xrange(v) \subseteq [x_0, y_0]$ and hence $xrange(father(v)) \cap [x_0, y_0] \neq \emptyset$. This shows that $v \in C_{max}$ implies $father(v) \in P$ and hence $|C_{max}| \leq |P| \leq 2 \cdot h$.

c) Since $v \in C$ implies that both sons of v also belong to C we conclude that the nodes in C form a forest of subtrees of T. Since in a binary tree the number of nodes is at most twice the number of leaves the bound follows. ∎

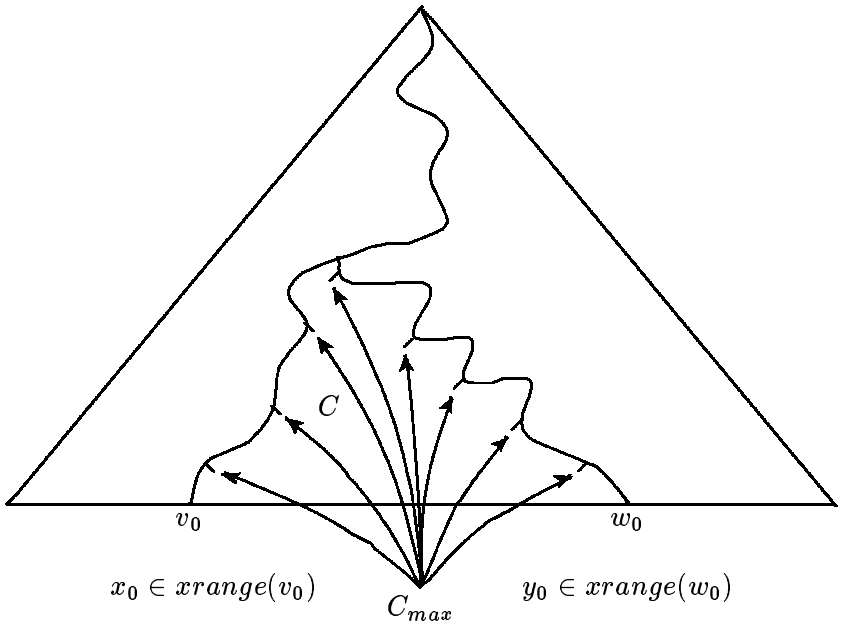


Figure 94.

The pictorial definition (cf. Figure 94) of sets P, C and C_{max} may help the reader's intuition. Set P consists of the paths to leaves v_0 and w_0 with $x_0 \in xrange(v_0)$ and $y_0 \in xrange(w_0)$. Set C is the set of nodes between the two paths and set C_{max} consists of the maximal nodes in C.

8.5.1.1. The Interval Tree and its Applications

This section is devoted to the interval tree. It will allow us to store a set of n intervals in linear space such that intersection queries can be answered in logarithmic time.

Let $U \subseteq \mathbb{R}$ be a finite set and let $S = \{[x_i, y_i]; x_i \in \mathbb{R}, 1 \leq i \leq n\}$ be a set of n closed intervals on the real line. An interval tree T for S (with respect to universe U) is a leaf-oriented search tree for set U, where each node of the tree is augmented by additional information. There are three pieces of information associated with each node v.

a) The node list $NL(v)$ of node v is the set of intervals in S containing the split value of v but of no ancestor of v, i.e.,

$$NL(v) = \{[x, y] \in S;\ split(v) \in [x, y] \subseteq range(v)\}.$$

We store the node list of node v as two sorted sequences: the ordered list of left endpoints and the ordered list of right endpoints. Both sequences are stored in balanced trees; furthermore, we provide for pointers to the maximal (minimal) element of the sequence of right (left) endpoints.

b) A mark bit stating whether the node list of v or any descendant of v is nonempty.

c) For every node v with nonempty node list, a pointer to the next larger (smaller) node in inorder with nonempty node list.

Figure 95 shows an interval tree for set $S = \{[1, 8], [2, 5], [3, 7], [4, 5], [6, 8], [1, 3], [2, 3], [1, 2]\}$ with respect to universe $U = \{1, 2, 3, 4, 5, 6, 7\}$. The split fields are shown inside the nodes and leaves, and the node lists are shown next to the nodes. The mark bits are shown as stars on the top of the nodes. Finally, the doubly linked list of nodes with nonempty lists is indicated by dashed lines.

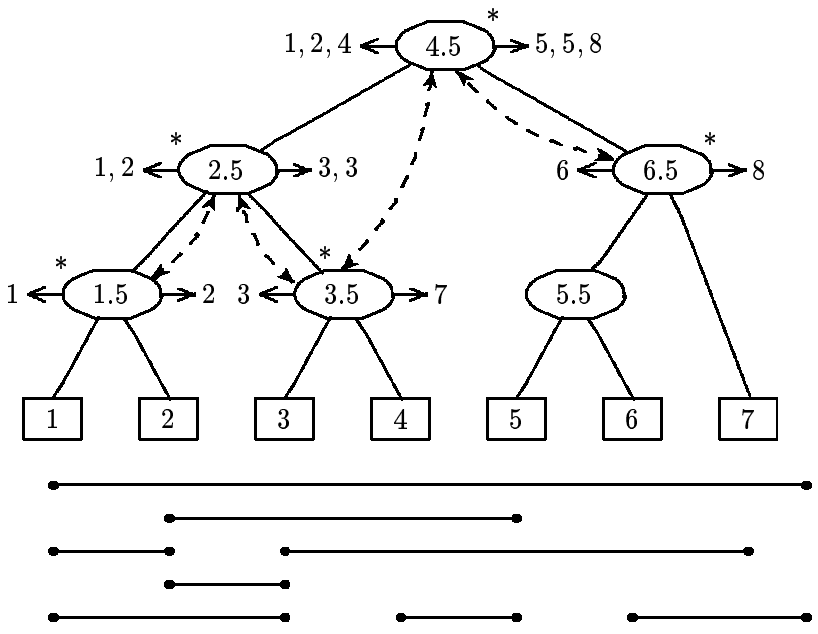


Figure 95.

The main power of interval trees stems from the node lists. The mark bits and doubly linked lists of nodes with nonempty node lists are needed to cope with large universes. If U contains only endpoints of intervals in S then the mark bits and the doubly linked lists are not needed.

The following lemma shows that interval trees use linear space, can be constructed efficiently, and efficiently support insertions and deletions of intervals (with left endpoint in U).

Lemma 2. *Let $U \subseteq \mathbb{R}$ be an ordered finite set, $N = |U|$, and let s be a set of n intervals with left endpoints in U.*

- **a)** *An interval tree for S uses space $O(n + N)$.*
- **b)** *An interval tree for S of depth $O(\log N)$ can be constructed in time $O(N + n \log N n)$.*
- **c)** *Intervals (with left endpoint in U) can be inserted into an interval tree of depth $O(\log N)$ in time $O(\log n + \log N)$. The same holds for deletion.*

Proof: a) The search tree for U clearly uses space $O(N)$. Furthermore, the total space required for the node lists is $O(n)$, since every interval in S is stored in exactly one list. This follows immediately from the definition of the node list.

b) Let T be a complete binary search tree for set U. Tree T can clearly be built in time $O(N)$ and has depth $O(\log N)$. The node list remains to be constructed. We show how to construct the left part of all node lists in time $O(n \log n + n \log N)$, the symmetric algorithm can be used to construct the right parts. Sort the intervals in S in increasing order of their left endpoint and propagate the sorted list of intervals down the tree. More precisely, if list L of intervals arrives at node v then then split L into L_1, L_2 and L_3, where L_1 (L_3) is moved to the left (right) son of v and L_2 is the node list at v. This can clearly be done in time $O(|L|)$ by comparing each interval in L with $\mathit{split}(v)$. Also, if L is sorted according to its left endpoint, then so are L_1, L_2, L_3. If we assign the cost of precessing list L to its members, then cost $O(\log N)$ is assigned to each interval. Hence the total cost of constructing all node lists is $O(n \log n + n \log N)$, where the $O(n \log n)$ accounts for the sorting step. Finally, it is easy to set the mark bits and to construct the doubly linked list of nodes with nonempty node list in time $O(N)$.

c) Let $[x, y]$ be an interval which is to be inserted into S. Our first task is to find the node v into whose node list interval $[x, y]$ has to be inserted. Node v can be found by a simple tree search. If $\mathit{split}(\mathit{root}) \in [x, y]$ then v is the root of the tree. If $\mathit{split}(\mathit{root}) \notin [x, y]$ then either $x > \mathit{split}(\mathit{root})$ and the search proceeds to the right subtree or $y < \mathit{split}(\mathit{root})$ and the search proceeds to the left subtree. Thus v can be determined in time $O(\log N)$. Next we insert x and y into the node list of v. This takes time $O(\log n)$. If the node list of v was nonempty before the insertion then we are done. Otherwise, we find the largest node w with nonempty node list preceding v in inorder using the mark bits. This takes time $O(\log N)$. Finally we insert v into the doubly linked list of nodes with nonempty node list. In summary, insertions take time $O(\log N + \log n)$. We leave the corresponding claim for deletion to the reader. ∎

We can now turn to the main property of interval trees: the efficient support of intersection queries.

Lemma 3. *Let S be a set of intervals and let $I = [x_0, y_0]$ be a query interval. Let $A = \{[x, y] \in S;\ [x, y] \cap [x_0, y_0] \neq \emptyset\}$ be the set of intervals in S intersecting I. Then, given an interval tree of height h for S, one can compute A in time $O(h+|A|)$.*

Proof: Let P and C be defined as in the introduction of Section 5.1, i.e., $P = \{v;\ v$ is a node of T and $\mathit{xrange}(v) \cap I \neq \emptyset$, $\mathit{xrange}(v) \not\subseteq I\}$ and $C = \{v;\ v$ is a node of T and $\mathit{xrange}(v) \subseteq I\}$. Then

$$A = \bigcup_{v \in C} NL(v) \cup \bigcup_{v \in P} \{[x, y] \in NL(v);\ [x, y] \cap I \neq \emptyset\},$$

since $[x, y] \in NL(v)$ and $[x, y] \cap I \neq \emptyset$ implies $\mathit{xrange}(v) \cap I \neq \emptyset$. Also, $v \in C$ clearly implies $NL(v) \subseteq A$. Consider $v \in P$ next. Recall that we organized $NL(v)$ as two ordered lists, the list of left endpoints and the list of right endpoints. Let $x_1 \leq x_2 \leq \cdots \leq x_k$ be the former list and let $y_1 \leq \cdots \leq y_k$ be the latter list. We have to discuss three cases, two of which are symmetric. Suppose first that $\mathit{split}(v) \in I$. Then $NL(v) \subseteq A$ since $\mathit{split}(v) \in [x, y]$ for all $[x, y] \in NL(v)$. Suppose next, that $\mathit{split}(v) \notin I$, say $\mathit{split}(v) \leq x$, the reverse case being symmetric. Then interval $[x_i, y_j] \in NL(v)$ intersects I iff $x \leq y_j$. We can thus find all such intervals by inspecting $y_k, y_{k-1}, \ldots$ in turn as long as they are at least as large as x. Hence we can determine $NL(v) \cap A$ in time proportional to $|NL(v) \cap A|$.

The node sets P and C are easily determined. P consists of the nodes on the search paths to x and y and C is the set of nodes between those paths. Thus the time required to compute A is $O(|P| + |C| + |A|)$. We will now complete the proof by two different arguments, one for the case of a "small" universe U and one for the general case.

Assume first that U contains only endpoints of intervals in S. Then $|C| = O(|A|)$, since all leaves in C are endpoints of intervals in A. Also $|P| \leq 2 \cdot h$ and the time bound follows. Note that we have not used the mark bits and the doubly linked list in this case.

We will now turn to the general case. Clearly, we only need to visit the nodes in C with nonempty node list. Note that there are at most $|A|$ such nodes. We can find one of them using the mark bits in time $O(h)$ and then find all the others by following the doubly linked lists since the nodes in $P \cup C$ occur as a contiguous segment in the doubly linked list. Thus we also have the time bound in the general case. ∎

This finishes the discussion of static or radix interval trees. Dynamic interval trees are based on D-trees; cf. Section 3.6.2. The change required with respect to the preceding discussion is insignificant. We now require that the underlying binary tree is a D-tree for the set of left endpoints of intervals in S, where the weight of a point x is the number of intervals with left endpoint x. Then the total weight (the thickness of the root of the D-tree) is $n = |S|$ and hence the depth of the D-tree is $O(\log n)$. Thus intersection queries can be answered in time $O(\log n + s)$, where s is the size of the answer by Lemma 3.

Next we discuss insertions and deletions. Let $I = [x, y]$ be an interval and suppose that we want to insert I into (delete I from) S. Clearly, we can add (delete) I from the appropriate node list and change the weight of the left endpoint x of I by one in time $O(\log n)$. Next, we have to rebalance the interval tree by rotations and double rotations. Since a double rotation can be realized by two rotations, it suffices to discuss rotation. The underlying D-tree is reorganized as described in Section 3.6.2. Finally we discuss the maintenance of the node lists in rotations.

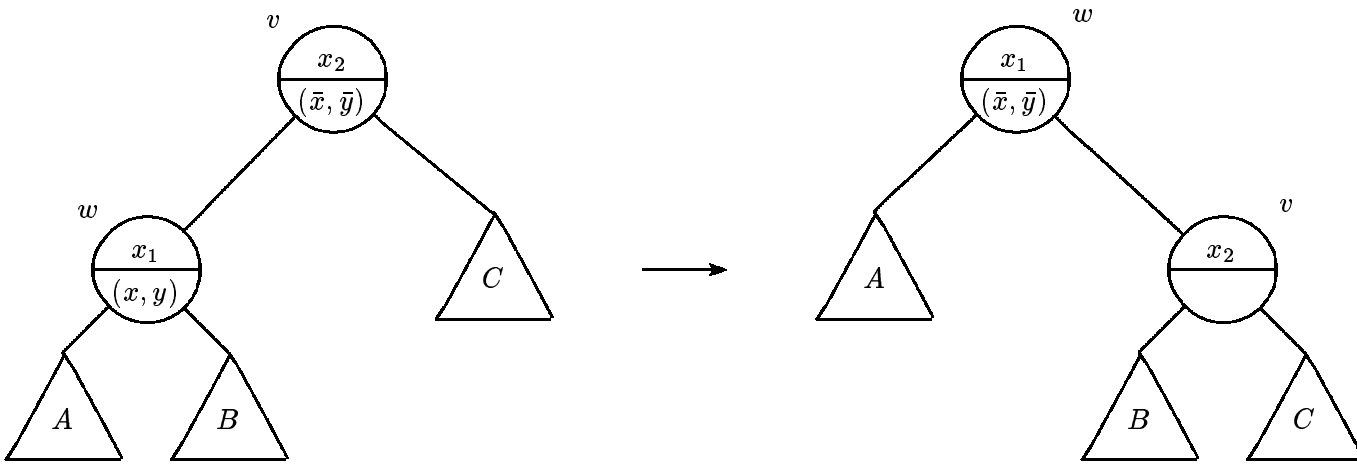


Figure 96.

Note first that the node lists of nodes in subtrees A, B, and C are unchanged. This can be seen as follows. Since $split(u)$ and $split(v)$ are not changed by the rotation, the xranges of nodes in subtrees A, B and C remain unchanged. Hence quantities $split(z)$ and $xrange(z)$ do not change for a node z in these subtrees and thus $NL(z)$ remains the same. The xranges of nodes u and v change. We have $xrange'(v) = xrange(u)$ and $xrange'(u) = xrange(u) \cap (-\infty, split(v)]$, where the prime is used to denote the situation after the rotation. Let $NL = NL(u) \cup NL(v)$ be the union of the node lists of nodes u and v. Then $NL'(v) = \{[x, y] \in NL; split(v) \in [x, y]\}$ and $NL'(u) = NL - NL(v)$. It is easy to see that the sorted representations of node lists $NL'(v)$ and $NL'(u)$ can be computed from the sorted representations of $NL(u)$ and $NL(v)$ in time $O(|NL|)$. Thus a rotation at node v has cost $O(|NL|)$, where $NL = NL(u) \cup NL(v)$.

Lemma 4. *Let v be a node of a dynamic interval tree. Then $|NL(v)| = O(th(v))$, where $th(v)$ is the number of leaves in the subtree with root v in the* BB[α]*-tree (underlying the D-tree underlying the interval tree); cf. Section 3.6.2.*

Proof: If $[x, y] \in NL(v)$ then $[x, y] \subseteq range(v)$. Hence a search for x, the left endpoint of interval $[x, y]$ goes through node v and hence the active x-node is a descendant of v. Thus a fraction of at least $\alpha/2$ of all leaves labelled x in the underlaying BB[α]-tree are descendants of v. Thus $|NL(v)| \leq 2/\alpha \cdot th(v)$. ∎

Lemma 4 together with Theorem 5 of Section 3.6.2 allows us to bound the amortized rebalancing cost of dynamic interval trees.

Lemma 5. *The total cost of n insertions and deletions into an initially empty interval tree is $O(n \log n)$.*

Proof: The cost of inserting (deleting) the n intervals into (from) the appropriate node lists and the weight changes for the left endpoints is clearly $O(n \log n)$. We still have to discuss the total cost of the rotations and double rotations.

By Lemma 4 and the preceding discussion the cost of rotating at a node v of $th(v)$ is $O(th(v))$. Thus the total cost of rotations and double rotations is $O(n \log n)$ by 3.6.2., Theorem 5 with $f(N) = O(N)$. ∎

We summarize our discussion of interval trees in

Theorem 1. *Let S be a set of n intervals.*

a) *Let U be a universe containing left endpoints of intervals in S. A static interval tree of depth $O(\log N)$, $N = |U|$, can be constructed for S with respect to U in time $O(n \log nN)$. It uses space $O(N + n)$ and allows us to compute the set A of intervals in S intersecting a query interval I in time $O(\log N + |A|)$. In addition, insertions and deletions take time $O(\log n + \log N)$.*

b) *In dynamic interval trees we have $N \leq n$. However, the time bounds for insertions and deletions are amortized.*

Proof: Obvious from Lemmas 2 to 5. ∎

We will now turn to an application of interval trees: reporting insertions of rectangles.

Theorem 2. *Let R be a set of n iso-oriented rectangles in the plane. Then the set A of pairs of intersecting rectangles in R can be computed in time $O(n \log n + |A|)$.*

Proof: We use the following notation. For $r \in R$, we denote the projections of r into the x-axis (y-axis) by $I_x(r)$ ($I_y(r)$). Our algorithm for computing A is based on plane sweep.

The x-structure contains exactly the left and right endpoints of projections $I_x(r)$, $r \in R$. The y-structure is an interval tree for the set of intervals $I_y(r)$ of rectangles r which are intersected by the sweep line. The transition at point x of the x-structure is as follows.

(1) Let In be the set of rectangles $r \in R$ which have x as the left endpoint of projection $I_x(r)$;

(2) Insert $I_y(r)$ into the y-structure for all $r \in In$;

(3) For every $r \in In$, enumerate all rectangles r' which are stored in the y-structure and which intersect the y-structure and which intersect the y-projection of r, i.e, $I_y(r) \cap I_y(r') \neq \emptyset$;
(4) Delete all rectangles r from the y-structure which have x as the right endpoint of projection $I_x(r)$.

The algorithm above clearly enumerates only intersecting pairs of rectangles. Finally we prove that it enumerates all intersecting pairs. Let $r, r' \in R$ with $r \cap r' \neq \emptyset$. Let $I_x(r) = [x_1, x_2]$ and $I_x(r') = [x_1', x_2']$. We assume w.l.o.g. that $x_1' \leq x_1$. Consider the transition at $x = x_1$. When line (3) is reached rectangle r' belongs to the y-structure. Also, $I_y(r) \cap I_y(r') \neq \emptyset$ and hence r' is enumerated in line (3) when this line is executed for rectangle r. This proves correctness.

The time bound is also easily established. We may either use dynamic interval trees or static interval trees with U being the set of (bottom) endpoints of intervals $I_y(r)$. Then $|U| \leq n$. Hence we spend time $O(\log n)$ per rectangle $r \in R$ in lines (2) and (4). Also, the cost of line (3) is $O(\log n + |A(r)|)$, where $A(r) = \{r \in R;\ r \cap r' \neq \emptyset\}$. Thus total running time is $O(n \log n + |A|)$. ∎

8.5.1.2. The Priority Search Tree and its Application.

Priority search trees are a mixture of search trees and priority queues (heaps). They support 1 1/2-dimensional range queries with logarithmic running time and linear space. Recall that range trees, cf. Section 7.2, require space $O(n \log n)$ and time $O((\log n)^2)$. A 1 1/2-dimensional range query is given by a rectangle whose bottom side is missing, i.e., it searches for all points (x, y) in a semi-infinite strip $x_0 \leq x \leq x_1,\ y \leq y_1$. Priority search trees can be used instead of interval trees in reporting intersections of rectangles. Other applications are e.g. containment queries or maintaining buddy systems.

Throughout this section we assume $S = \{(x_i, y_i);\ 1 \leq i \leq n\}$ to be a set of points inn the plane. We assume for simplicity that $x_i \neq x_j$ for $i \neq j$, although the theory of priority search trees can also be developed without that assumption. We will indicate the required changes in the discussion below. In many applications this assumption can be obtained by replacing point (x, y) by $((x, y), y)$, by ordering the first component lexicographically, and by taking some care in formulating queries and interpreting their answers. We leave the details to the reader.

Let U be a set containing at least the x-coordinates of all points in S. A priority search tree for S with respect to universe U consists of a leaf-oriented search tree for set U. As always, we use a field $split(v)$ in each node v to direct the searches. In addition, each node v has a priority field $prio(v)$ which holds an elment of S (or nothing). Each elment of S is stored in the priority field of exactly one node and this node must lie on the search path to the x-coordinate of the element, i.e., if $prio(v) = (x, y)$ then $x \in xrange(v)$. The priority fields implement

a priority queue on the y-coordinates of the elements of S, i.e., if $prio(v) = (x, y)$ then $prio(father(v))$ is defined and $y \geq y'$, where $prio(father(v)) = (x', y')$.

Figure 97 shows a priority search tree for set $S = \{(1,4),(2,1),(3,7),(4,2),(5,1),(6,3)\}$ with respect to universe $U = \{1,2,3,4,5,6\}$. In nodes the split (priority) field is given above (below) the horizontal line.

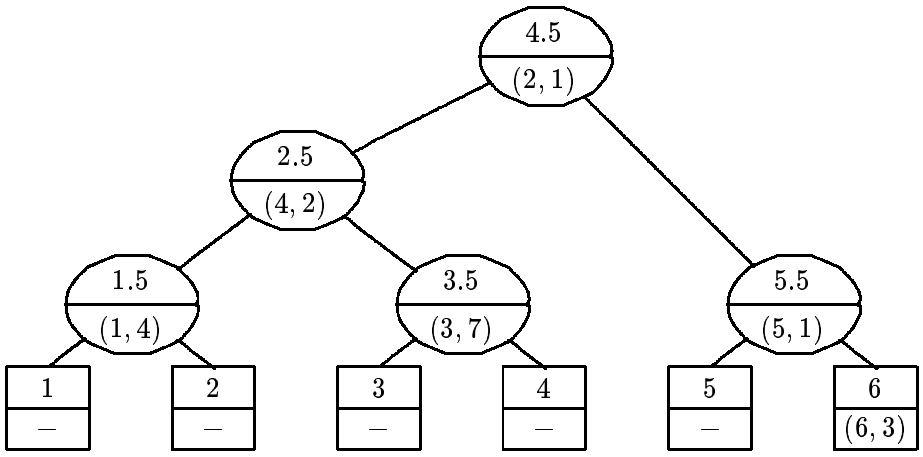


Figure 97.

If set S contains points with identical x-coordinate then the following change is required. With each leaf of the tree we associate a sorted sequence which contains some of the elements having the (value associated with the) leaf as x-coordinate. Thus if $(3, 10)$ also belonged to S then the sorted sequence associated with leaf 3 would contain elements $(3, 7)$ and $(3, 10)$ in that order. We implement the sorted sequence as an ordinary priority queue. All algorithms below become slightly more involved because of that change. We leave the details to the reader.

The algorithms for priority search trees are subtle. We therefore give (almost) complete programs based on the following declarations.

```
type pair   = record x,y: real end;
type status = (leaf, nonleaf);
type node   = record prio: pair;
              split: real;
              case stat: status of
              leaf: ;
              nonleaf: (lson, rson: ↑node)
              end
```

Nodes and leaves are of type *node*. If the priority field of a node v is undefined then we store $(split(v), infinity)$ in it.

Lemma 6. *If S is sorted according to the x-coordinate then a priority search tree for S of depth $O(\log N)$ can be built in time $O(n + N)$.*

Proof: We can clearly build a search tree (based on the complete binary tree with N leaves) for U in time $O(N)$. We still have to fill the priority fields. We do so by playing a knockout tournament on the elements of S. We initialize the tournament by writing element (x, y) of S into the leaf corresponding to x. If no element of S corresponds to a leaf v then the priority field is initialized to $(\mathit{split}(v), \infty)$. Next consider a node v such that the priority fields of v's sons are already defined. We move the priority field of the left son of v to v if it has the smaller y-coordinate and we move the priority of the right son otherwise. This empties the priority field of one of the sons of v which we then fill in exactly the same way. Continuing in this way we finally empty the priority field of a leaf w which we then refill with $(\mathit{split}(w), \infty)$. Thus we can compute $\mathit{prio}(v)$ in time proportional to the height of v. Since the number of nodes of height h is $N/2^h$ we need time

$$O\left(\sum_{h=1}^{\log n} (N/2)^h \cdot h\right) = O(N)$$

to fill all priority fields. ∎

We can use priority search trees for the following queries:

$$\begin{aligned}
\mathit{MinXinRectangle}(x_0, x_1, y_1) &= \min\{x;\ \exists y : x_0 \le x \le x_1 \text{ and } y \le y_1 \text{ and } (x, y) \in S\} \\
\mathit{MaxXinRectangle}(x_0, x_1, y_1) &= \max\{x;\ \exists y : x_0 \le x \le x_1 \text{ and } y \le y_1 \text{ and } (x, y) \in S\} \\
\mathit{MinYinXRange}(x_0, x_1) &= \min\{x;\ \exists x : x_0 \le x \le x_1 \text{ and } (x, y) \in S\} \\
\mathit{EnumerateRectangle}(x_0, x_1, y_1) &= \{(x, y) \in S;\ x_0 \le x \le x_1 \text{ and } y \le y_1\}
\end{aligned}$$

Query $\mathit{EnumerateRectangle}(x_0, x_1, y_1)$ enumerates all points of S which lie in the semi-infinite strip $x_0 \le x \le x_1$ and $y \le y_1$. *MinXinRectangle* (*MaxXinRectangle*) compute the minimal (maximal) x-coordinate of any point in that strip. Finally, *MinYinXRange* computes the minimal y-coordinate of any point in the vertical strip determined by x_0 and x_1.

For the lemmas below we recall the definitions of sets P, C and C_{max} of nodes of a search tree T with respect to query interval $[x_{left}, x_{right}]$.

$$\begin{aligned}
P &= \{v;\ \mathit{xrange}(v) \cap [x_{left}, x_{right}] \neq \emptyset \text{ and } \mathit{xrange}(v) \not\subseteq [x_{left}, x_{right}]\} \\
C &= \{v;\ \mathit{xrange}(v) \subseteq [x_{left}, x_{right}]\} \\
C_{max} &= \{v;\ v \in C \text{ and } \mathit{father}(v) \notin C\}
\end{aligned}$$

Lemma 7. *Let T be a priority search tree (for set S with respect to universe U) of height h. Then operation MinYinXRange takes time $O(h)$.*

Proof: Consider operation $\mathit{MinYinXRange}(x_{left}, x_{right})$. If the output of *MinYinXRange* is undefined then $C = \emptyset$ and hence the searches for x_{left} or x_{right} end in the

same or in adjacent leaves. If the output is defined, say $(x^*, y^*) \in S$ with $y^* MinYinXRange(x_{left}, x_{right})$, then let w be the leaf whose associated value of U is x^*. Leaf w belongs to $P \cup C$. If $w \in P$, then there is clearly a node $v \in P$ such that $prio(v) = (x^*, y^*)$. If $w \notin P$, i.e., $w \in C$, then consider $z \in C_{max}$ such that z is an ancestor of w. Since y^* is the minimal y-value of any point in the query interval and since the priority fields implement a priority queue on the y-values we conclude that there is an ancestor v of z with $prio(v) = (x^*, y^*)$. In particular, $v \in P \cup C_{max}$. Thus in either case there must be a node $v \in P \cup C_{max}$ such that $prio(v) = (x^*, y^*)$ and hence *MinYinXRange* can be answered by inspecting all nodes in $P \cup C_{max}$. Finally, since $|P \cup C| \leq 4 \cdot h$ since $v \in C_{max}$ implies $father(v) \in P$, and since every $v \in P$ lies on the search path for either x_{left} or x_{right} the nodes in $P \cup C_{max}$ can clearly be inspected in time $O(h)$. The details are given in Program 14. In this program we use the additional type

type
condpair
= **record** *p*: *pair*; *valid*: boolean **end**

```
(1)     function MinYinXRange(t↑node, x_left, x_right : real) : condpair;
(2) var cand1, cand2: condpair;
(3) begin       cand1.valid ← cand2.valid ← false;
(4)             if x_left ≤ t↑.prio.x ≤ x_right
(5)             then cand1.p ← t↑.prio;
(6)                  cand1.valid ← true
(7)             else if t↑.stat = nonleaf
(8)                  then   if x_left ≤ t↑.split
(9)                         then cand1 ← MinYinXRange(t↑.lson, x_left, x_right)
(10)                        fi;
(11)                        if t↑.split < x_right
(12)                        then cand2 ← MinYinXRange(t↑.rson, x_left, x_right)
(13)                        fi;
(14)                        if (¬cand1.valid) or (cand2.valid and
(15)                           cand2.p.y < cand1.p.y)
(16)                        then cand1 ← cand2
(17)                        fi
(18)                 fi
(19)            fi;
(20)            MinYinXRange ← cand1
(21) end.
```

Program 14

Several remarks should be made about that program. Note first that only nodes in $P \cup C$ are visited. Also, if t points to a node $v \in C_{max}$ then the test in line (4) returns true and hence no further recursive calls are initiated. Thus only

nodes in $P \cup C_{max}$ are visited and the time bound follows. However, not all nodes in $P \cup C_{max}$ are necessarily visited and hence correctness requires some explanation. Let $v \in P \cup C_{max}$ be such that $prio(v) = (x^*, y^*)$ and let z be any proper ancestor of v. Then $prio(z) = (x, y)$ is defined and $y \leq y^*$. if $y < y^*$ then $x \notin [x_{left}, x_{right}]$ by definition of y^* and hence the test in line (4) returns false. Thus appropriate recursive calls are initiated in lines (9) and/or (11). ∎

Lemma 8. *Let T be a priority search tree (for set S with respect to universe U) of height h. Then operation EnumerateRectangle takes time $O(h + s)$, where s is the size of the answer.*

Proof: Consider operation $EnumerateRectangle(x_{left}, x_{right}, y_{top})$. Let $R = \{(x, y);\ x_{left} \leq x \leq x_{right}, y \leq y_{top}\}$ be the query rectangle. Clearly, all pairs $(x, y) \in R \cap S$ are stored in the priority fields of nods in $P \cup C$. The crucial observation is now that the $v \in C - C_Max$ and $prio(v) \in R$ implies $prio(father(v)) \in R$. This can be seen as follows. Let $prio(v) = (x, y)$ and let $prio(father(v)) = (x', y')$. Then $y' \leq y \leq y_{top}$ and $x_{left} \leq x' \leq x_{right}$ since $father(v) \in C$. Thus $(x', y') \in R$.

The observation above suggests the following algorithm. Visit all nodes in $P \cup C_Max$ and inspect their priority fields. For every $v \in C_{max}$, explore the subtree rooted at v top-down. if descendant w of v is visited and $prio(w) \in R$ then also visit both sons of w. If $prio(w) \notin R$ then visit no son of w. The correctness of this algorithm follows immediately from the observation above.

The bound on the running time follows from the fact that if a node v is visited then either $prio(v) \in R$ or $father(v) \in P \cup C_{max}$ or $prio(father(v)) \in R$. Thus at most $3 \cdot s + 2 \cdot h$ nodes are visited and hence running time is $O(h + s)$. ∎

Lemma 9. *Let T be a priority search tree for set S of height h. Then operations MinXinRectangle and MaxXinRectangle take time $O(h)$.*

Proof: Operations *MinXinRectangle* and *MaxXinRectangle* are symmetric. We therefore only have to consider operation $MinXinRectangle(x_{left}, X_{right}, y_{top})$. Let $R = \{(x, y);\ x_{left} \leq x \leq x_{right}, y \leq y_{top}\}$ be the query rectangle and let $(\tilde{x}, \tilde{y}) \in R \cap S$ be such that $prio(v) = (\tilde{x}, \tilde{y})$ for some $v \in P \cup C_{max}$ and $\tilde{x} \leq x$ whenever $(x, y) = prio(w) \in R$ for some node $w \in P \cup C_{max}$. Thus $(\tilde{x}, \tilde{y})$ is the "best" answer to the query which is stored in the priority field of a node in $P \cup C_{max}$.

It follows from the proof of Lemma 6 (operation *MinYinXRange*) that $(\tilde{x}, \tilde{y})$ is defined iff $R \cap S \neq \emptyset$. Thus we can test in time $O(h)$ whether operation *MinXinRectangle* is defined. Assume next that *MinXinRectangle* is defined. Let $(x^*, y^*) \in S$ be such that $x^* = MinXinRectangle(x_{left}, x_{right}, y_{top})$. If $x^* = \tilde{x}$ then we can find (x^*, y^*) by visiting the nodes in $P \cup C_{max}$. So let us assume that $x^* < \tilde{x}$. Then pair (x^*, y^*) is stored in the priority field of a node w in $C - C_{max}$ and hence there is a node $v \in C_{max}$ which is an ancestor of w. Let $prio(v) = (x', y')$. Then $y' \leq y^* \leq y_{top}$ since the priority fields implement a priority

queue and $x_{left} \leq x' \leq x_{right}$ since $v \in C$. Thus $(x', y') \in R$ and hence $\tilde{x} \leq x'$ by definition of $(\tilde{x}, \tilde{y})$. In particular, we have $x^* \leq \tilde{x} \leq x'$ and hence $\tilde{x} \in xrange(v)$. We conclude that either $(\tilde{x}, \tilde{y}) = (x^*, y^*)$ or $(x^*, y^*) = prio(w)$, where w is a descendant of the unique $v \in C_{max}$ with $\tilde{x} \in xrange(v)$. We finally describe how to find w in the latter case. Note first that if z is a node on the path from v to w then $prio(z) \in R$. Also, z is the right son of its father if and only if $y'' > y_{top}$, where (x'', y'') is the priority field of the left brother of z. Thus we can find w by a simple tree search starting in w. If the search reaches node z and $y'' < y_{top}$, where (x'', y'') is the priority field of the left son of z, then we proceed to the left son, otherwise we proceed to the right son. In this way we can be sure that node w will be found.

In summary, query *MinXinRectangle* can be answered by visiting the nodes in $P \cup C_{max}$ and by following a single path down the tree starting in a uniquely defined node of C_{max}. Thus time $O(h)$ suffices. An elegant realization is given by Program 15.

```
(1)     function MinXinRectangle(t↑node; x_left, x_right, y_top : real) : condpair;
(2) var c: condpair;
(3) begin       c.valid ← false;
(4)             if y_top ≥ t↑.p.y
(5)             then if t↑.stat = nonleaf
(6)                  then   if x_left ≤ v↑.split
(7)                         then c ← MinXinRectangle(t↑.lson, x_left, x_right, y_top)
(8)                         fi;
(9)                         if ¬c.valid and t↑.split < x_right
(10)                        then c ← MinXinRectangle(t↑.rson, x_left, x_right, y_top)
(11)                        fi
(12)                 fi;
(13)                 if     x_left ≤ t↑.p.x ≤ x_right and t↑.p.y ≤ y_top
(14)                        and (¬c.valid or t↑.p.x < c.p.x)
(15)                 then c ← t↑.p; c.valid ← true
(16)                 fi
(17)            fi;
(18)            MinXinRectangle ← c
(19) end.
```

Program 15

Several remarks should be made about that program. Note first that only nodes in $C \cup P$ are visited. Next consider a node $v \in C$. If $prio(v) \notin R$ and hence the test in line (4) returns false then no son of v is visited. If $prio(v) \in R$ then the left son of v is always visited. If $prio(lson(v)) \in R$ then the recursive call made in line (7) returns a valid pair and hence no recursive call is made in line (10). If $prio(lson(v)) \notin R$ then the recursive call made in line (7) aborts immediately because the test in line (4) is negative and hence has cost $O(1)$. Thus at most one

recursive call with nonconstant cost is made by node $v \in C$ and hence only $O(h)$ nodes are visited below each node $v \in Cmax$. It remains to show that at most one node $v \in C_{max}$ initiates recursive calls at all. Let $v \in C_{max}$ be the first node in C_{max} which initiates recursive calls and let $w \in C_{max}$ be to the right of v. Also, let z be the least common ancestor of v and w. Since v (w) is in the left (right) subtree of z the recursive call initiated in line (7) of the procedure when applied to node z returns a valid pair and hence the call in line (10) is not executed. Thus w is never visited, and hence only $O(h)$ nodes are visited altogether.

We finally prove correctness. Let $v \in C \cup P$ be such that $prio(v) = (x^*, y^*)$ and let $w_0, w_1, \ldots, w_k$ be the path from the root of T to $v = w_k$. Then $prio(w_i).y \leq y^* \leq y_{top}$ for all i. Hence if w_i is visited and w_{i+1} is the left son of w_i then w_{i+1} is also visited. Assume next that w_{i+1} is the right son of w_i. Let z be the left son of w_i. Then clearly $prio(z).x \leq split(w_i) < x^*$ and hence $prio(z).y > y_{top}$ by definition of (x^*, y^*). Thus,if w_i is visited then the visit of node z fails in line (4) and hence no valid pair is returned in line (7). We therefore visit node w_{i+1} in line (10). This shows that node v is visited by the search and hence x^* is correctly computed. ∎

Lemmata 7 to 9 show that priority search trees support a number of query operations in logarithmic time. We will now turn to insertions and deletions. As in the case of interval trees we treat the radix priority search trees first.

Lemma 10. *Let T be a priority search tree for set S with respect to universe U and let h be the height of T.*

a) *Let $(x, y) \notin S$ such that $x \in U$ and such that there is no $(x', y') \in S$ with $x = x'$. Then we can add (x, y) to S in time $O(h)$.*

b) *Let $(x, y) \in S$. Then we can delete (x, y) from S in time $O(h)$.*

Proof: a) The insertion algorithm is quite simple. It uses variables $t \uparrow node$ and $p : pair$.

```
(1) t ←↑root; p ← (x, y);
(2) while t↑.stat = nonleaf
(3) do          if p.y < t↑.prio.y
(4)             then exchange p and t↑.prio
(5)             fi;
(6)             if p.x ≤ t↑.split then t ← t↑.lson else t ← t↑.rson fi
(7) od
(8) t↑.prio ← p.
```

Program 16

The insertion algorithm follows a path down the tree. The path is determined by the x-value of p. Whenever p's y-value is smaller than the y-value of the priority field of the current node we store p in that node, pick up the priority field of the node and continue inserting it. This shows correctness as well as the bound on the running time.

b) Deletion of (x, y) form S is also quite simple. An ordinary tree search allows us to find node v with $prio(v) = (x, y)$. We delete point (x, y) from the priority field of node v and then refill it by the smaller (with respect of y-value) priority field of the sons of v, which we then in turn refill .... Thus the deletion algorithm is identical to the algorithm used to fill priority fields when building a tree from scratch and hence it runs in time $O(h)$. ∎

We summarize our discussion of radix priority search trees in

Theorem 3. *Let $U \subseteq \mathbb{R}$, $N = |U|$ and let $S \subseteq U \times \mathbb{R}$, $n = |S|$. Then a priority search tree for S with respect to U can be built in time $O(N + n \log n)$. It supports insertions of points in $U \times \mathbb{R}$, deletions, and queries MinXinRectangle, MaxXinRectangle, MinYinXRange in time $O(\log N)$. Furthermore, operation EnumerateRectan takes time $O(\log N + s)$, where s is the size of the answer.*

We next turn to dynamic priority search trees. For $S \subseteq \mathbb{R} \times \mathbb{R}$ let $U(S)$ be the projection of S onto the first coordinate, i.e., the set of x-coordinates of points in S. In a dynamic priority search tree we use a balanced search tree of set $U(S)$ as the underlying search tree structure. Such a tree has depth $O(\log n)$, $n = |S|$. We assume for the following discussion that the underlying search trees are rebalanced by rotations and double rotations, i.e., we may use BB[α]-trees (cf. Chapter 3.5.1), $(2, 4)$-trees realized as red-black trees (cf. Chapter 3, Exercise 27), AVL-trees (cf. Chapter 3, Exercise 25) or half-balanced trees (cf. Chapter 3, Exercise 26). Consider an insertion or deletion. We proceed as described in Lemma 10 except for the following change. An insertion adds an additional leaf to the tree and a deletion deletes some leaf from the tree. In this way we obtain a legal priority search tree except for the possible defect that the underlying search is unbalanced. We use rotations and double rotations to rebalance the tree. Since double rotations can be realized by two rotations we only need to discuss rotations.

Suppose that we perform a rotation at node v with left son w and let $prio(v) = (\bar{x}, \bar{y})$ and $prio(w) = (x, y)$. Then point (x, y) has the smallest y-value of all priority fields in the subtree rooted at v and hence $prio'(w) = (\bar{x}, \bar{y})$. We use primes to denote the situation after the rotation. Also, the split fields of nodes v and w are easy to maintain. However, priority fields $prio(w)$ and $prio'(v)$ cause difficulties. We proceed as follows. We first insert point (x, y) into either tree A (if $x \leq x_1$) or B (if $x > x_1$). It is important to observe that this insertion does not require a structural change of the underlying search tree. It rather updates the priority fields along a path down the tree as described in Lemma 10a), and hence takes time $O(h(v))$, where $h(v)$ is the height of node v. Priority field $prio'(v)$ is filled as

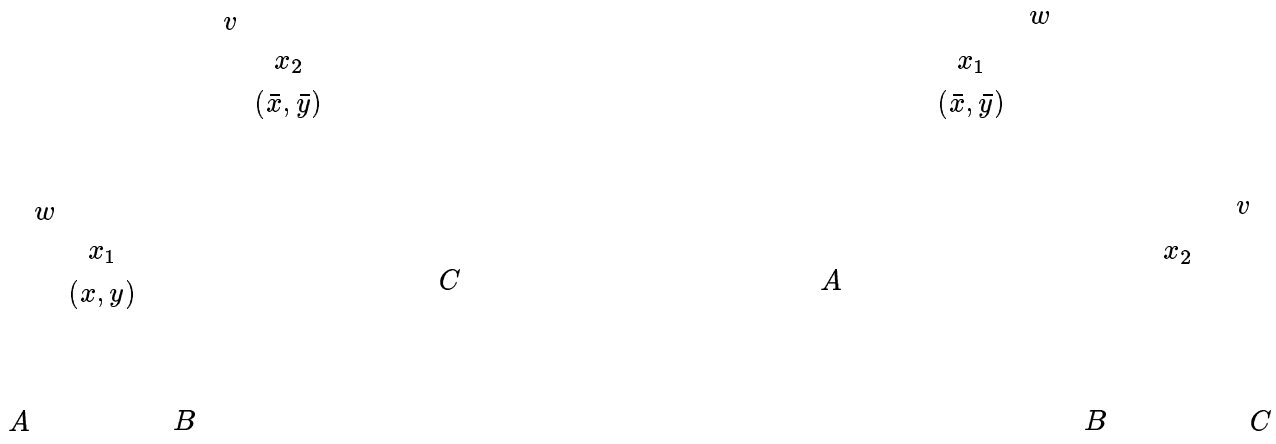


Figure 98.

described in Lemma 10b), i.e., we move the priority field of either the root of B or C to node v and then continue in this way along a simple path down the tree. Thus filling the priority field of node v after insertion takes time $O(h(v))$. We can summarize the discussion in

Lemma 11. *A rotation or double rotation at a node v of height $h(v)$ takes time $O(h(v)$.*

Proof: By the discussion above. ∎

Lemma 11 leads directly to

Theorem 4. *Let $S \subseteq \mathbb{R} \times \mathbb{R}$, $n = |S|$. Dynamic priority search trees based on XYZ-search trees ($XYZ \in \{AVL, \mathrm{BB}[\alpha], \text{red-black}, \text{half-balanced}\}$) support queries MinXinRectangle, MaxXinRectangle, MinYinXRange in time $O(\log n)$ and query EnumerateRectangle in time $O(\log n + s)$, where s is the size of the answer. Moreover, insertions and deletions take time*

$O((\log n)^2)$ *in the worst case*	*if* $XYZ = AVL$
$O((\log n)^2)$ *in the worst case and*	*if* $XYZ = \mathrm{BB}[\alpha]$
$O(\log n)$ *in the amortized case*	*or* $XYZ =$ *red-black*
$O(\log n)$ *in the worst case*	*if* $XYZ =$ *half-balanced*

Proof: The time bounds for the queries follow immediately from Lemmas 7–9. The worst case time bounds for AVL-trees, BB[α]-trees and red-black trees follows from the observation that at most $O(\log n)$ rotations and double rotations of cost $O(\log n)$ each are required to rebalance such a tree. Since $O(1)$ such operations suffice for half-balanced trees e also have the $O(\log n)$ worst case time bound for these trees. We next prove the bound on the amortized cost for BB[α] and red-black trees. For both classes of trees we have shown that the total number of rotations and double

rotations caused by nodes of height h in a sequence of n insertions and deletions is $O(n/c^h)$ for some $c > 1$. This is shown in Section 3.5.1, Theorem 4 for BB[α]-trees and in Section 3.5.3.2, Theorem 10 for red-black trees implementing $(2,4)$-trees. Since the cost of a rotation at a node of height h is $O(h)$ the total cost of all rotations is $\sum_{h\geq 1} O(n/c^h \cdot h) = O(n)$ and hence $O(1)$ per insertion or deletion. Since the cost of an insertion or deletion outside the rebalancing operations is $O(\log n)$ the bound on the amortized cost follows. ∎

There are numerous applications of priority search trees. We first show how to maintain a dynamic set of intervals on the real line.

Theorem 5. *Let S be a set of intervals on the real line, $n = |S|$. Using priority search trees we can insert intervals into S and delete intervals from S in time $O(\log n)$. Also, given a query interval $[x_0, y_0]$ we can enumerate*

a) *all s intervals $[x,y] \in S$ with $[x,y] \cap [x_0,y_0] \neq \emptyset$ in time $O(s + \log n)$.*

b) *all s intervals $[x,y] \in S$ containing the query interval in time $O(s + \log n)$*

c) *all s intervals $[x,y] \in S$ contained in the query interval in time $O(s + \log n)$.*

Proof: For parts a) and b) we represent interval $[x,y]$ by point $(x,y) \in \mathbb{R}^2$ and store the associated set of points in a priority search tree. Then all intervals intersecting the query interval $[x_0,y_0]$ are listed by *EnumerateRectangle*$(x_0, \mathit{infinity}, y_0)$, i.e., we list all rectangles $[x,y] \in S$ with $x_0 \leq y \leq \mathit{infinity}$ and $x \leq y_0$. Similarly, all intervals $[x,y] \in S$ containing the query interval $[x_0,y_0]$, i.e., $x \leq x_0 \leq y_0 \leq y$, can be listed by *EnumerateRectangle*$(y_0, \mathit{infinity}, x_0)$.

For part c) we associate point (x,y) with interval $[x,y]$ and store the set of associated points in a priority search tree. Then all intervals $[x,y] \in S$ contained in the query interval $[x_0,y_0]$, i.e., $x_0 \leq x \leq y \leq y_0$, can be listed by *EnumerateRectangle*(x_y, y_0, y_0). ∎

Theorem 5a) provides us with an alternative proof of Theorem 2. We may replace interval trees by priority search trees for computing the set of s pairs of intersecting rectangles in a set of n iso-oriented rectangles in time $O(n \log n + s)$.

Another class of applications concerns visibility and containment problems. let S be a set of points in $\mathbb{R}^2$; each $(x,y) \in S$ defines a vertical semi-infinite ray with lower endpoint (x,y). We can think of the line segments as being either translucent or opaque. Given a query point $(x_0,y_0) \in S$ we want to find all line segments visible along a horizontal line of increasing x. We store the points (x,y) in a priority search tree.

If the line segments are translucent then the segments visible from (x_0,y_0) are given by $(x_0 \leq x,\ y \leq y_0$, i.e., by *EnumerateRectangle*$(x_0, \mathit{infinity}, y_0)$. In the opaque case, the solution is given by *MinXinRectangle*$(x_0, \mathit{infinity}, y_0)$. Note also, that in either case we can restrict the horizontal line segment starting in (x_0,y_0) to some finite length without changing complexity.

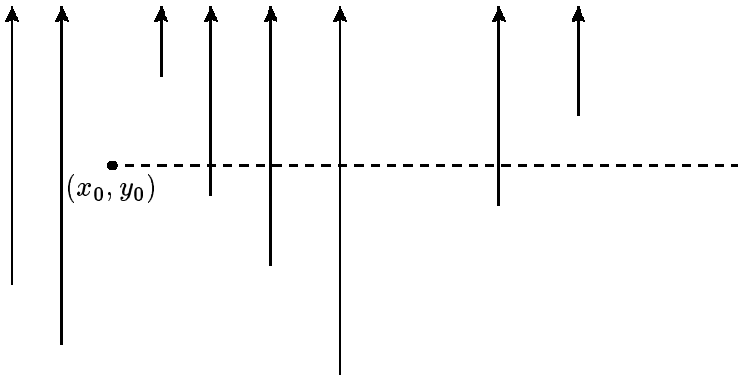


Figure 99.

The problem becomes more complicated if we allow the vertical line segments to be finite. In this case we can obtain an $O((\log n)^2)$ method by a combination of interval trees and priority search trees. We represent the set of vertical line segments as an interval tree T. More precisely, let $\hat{S}$ be the set of projections of the vertical line segments onto the y-axis. Then T is an interval tree for set $\hat{S}$. Let v be a node of T and let $[y_{bottom}, y_{top}]$ be an interval in its node list. Then y_{bottom} (y_{top}) is stored in the left (right) part of v's node list. Next note, that we can view the left (right) part as a set of "semiinfinite" rays extending from (x, y_{bottom}) to $(x, \mathit{split}(v))$, where x is the x-coordinate of interval $[y_{bottom}, y_{top}]$. Thus we can organize both node lists as priority search trees and hence find the visible line segments in v's node list in time $O(\log n + s)$ in the translucent case and $O(\log n)$ in the opaque case. Since the depth of the interval tree is $O(\log n)$ we obtain a total cost of $O((\log n)^2 + s)$ in the translucent case and $O((\log n)^2)$ in the opaque case.

The translucent visibility problem discussed above is an intersection problem. We discuss more general intersection problems in Section 5.1.4. Further applications of priority search trees are discussed in the exercise.

8.5.1.3. Segment Trees

Segment trees are yet another method of storing sets of intervals. They are particularly useful in situations where additional information has to be stored together with intervals. We describe several applications of segment trees in the text and in the exercises, most notably the hidden line elimination problem for three-dimensional scenes (in Section 5.1.4) and the measure problem for a set of polygons.

Let $U \subseteq \mathbb{R}$, $U = \{x_1 < \cdots < x_N\}$, and let S be a set of intervals which have both endpoints in U. A segment tree for S with respect to universe U consists of a search tree with $2 \cdot N + 1$ leaves and some additional information. The leaves correspond to the **atomic** segments $(-\infty, x_1)$, $[x_1, x_1]$, (x_1, x_2), $[x_2, x_2]$, (x_2, x_3), $[x_3, x_3]$, $\ldots$, (x_{N-1}, x_N), $[x_N, x_N]$, $(x_N, +\infty)$ defined by U in increasing order from left to right. Here, (x_i, x_{i+1}) denotes the open interval from x_i to x_{i+1} and $[x_i, x_i]$

denotes the closed interval from x_i to x_i, i.e., point x_i. The split fields of the internal nodes are defined such that a search for $x \in \mathbb{R}$ is directed to the atomic segment containing x, i.e., if (x_i, x_{i+1}) ($[x_{i+1}, x_{i+1}]$) is the rightmost leaf in the left subtree of node v then we follow the pointer to the left subtree if $x < x_{i+1}$ ($x \leq x_{i+1}$). Thus we need to store an additional bit that distinguishes between the two cases in addition to storing the split field $split(v) = x_{i+1}$.

The xrange of a node of a segment tree is defined as in the introduction of Section 5.1. Note that the $xrange(v)$ of node v is the union of the atomic segments associated with the leaf descendants of v. We also associate (and store) with each node v its node list $NL(v)$. The node list of v contains pointers to all intervals of S which cover v's xrange but do not cover the xrange of v's father, i.e.,

$$NL(v) = \{I \in S;\ xrange(v) \subseteq I \text{ and } xrange(father(v)) \not\subseteq I\}.$$

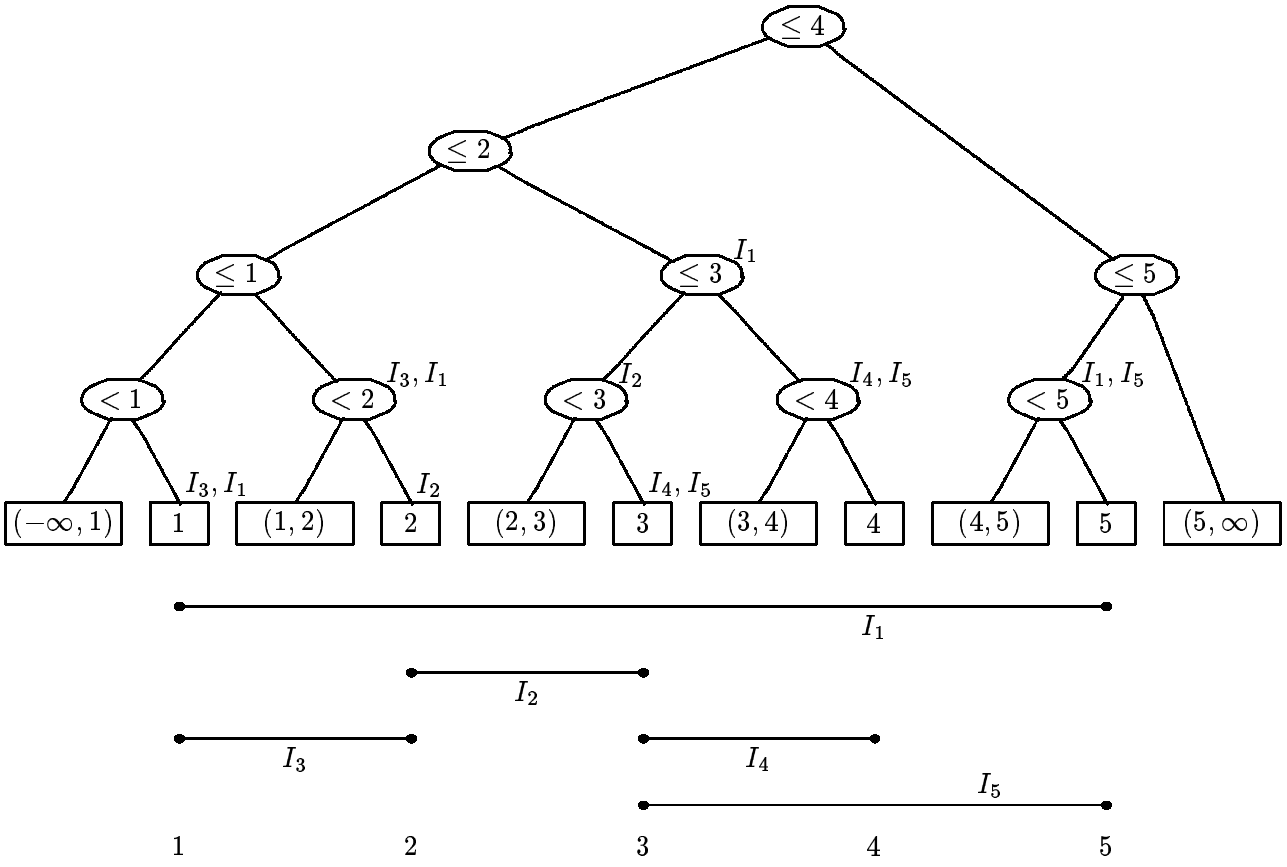


Figure 100.

Figure 100 illustrates these definitions. We have $U = \{1, 2, 3, 4, 5\}$ and $S = \{[1, 5], [2, 3], [1, 2], [3, 4], [3, 5]\}$. The split fields and the condition for following the

pointer to the left son are indicated inside internal nodes. The node lists associated with the various nodes are indicated on top of the nodes.

In a segment tree an interval may be stored in several node lists. Lemma 12 shows that it is stored in at most $2 \cdot h$ node lists, where h is the height of the tree.

Lemma 12. *Let T be a segment tree of height h for S and let $I \in S$. Then I belongs to at most $2 \cdot h$ node lists. Furthermore, I is the disjoint union of the sets $xrange(v)$, where the union is over all nodes v such that $I \in NL(v)$.*

Proof: Let $C = \{v;\ xrange(v) \subseteq I\}$ and let $C_{max} = \{v;\ v \in C \text{ and } father(v) \notin C\}$. Then I belongs to $NL(v)$ if and only if $v \in C_{max}$. In Lemma 2 it was shown that $|C_{max}| \leq 2 \cdot h$. Also, if $v, w \in C_{max}$ then $xrange(v) \cap xrange(w) = \emptyset$. Finally, if $x \in I$ then there is clearly a node $v \in C_{max}$ such that $x \in xrange(v)$. Thus $I = \bigcup\{xrange(v);\ I \in NL(v)\}$ and the union is over disjoint sets. ∎

We can draw several consequences from the lemma above. The first observation is that an interval I is stored in a segment tree as a set of disjoint pieces, the piece $xrange(v)$ is stored in $NL(v)$. Also, a node v represents the identical piece, namely $xrange(v)$, of all intervals I with $I \in NL(v)$. Thus we are free to organize the node lists according to secondary criteria. This explains the flexibility of segment trees. The second consequence is that balanced segment trees require space $O(n \log N)$ and can be constructed in time $O(n \log N)$. Finally, given interval I, one can find all nodes v with $I \in NL(v)$ in time $O(\log N)$. Hence the insertions and deletions are efficient operations in segment trees. The exact time bounds depend on the structure chosen for the node lists.

Theorem 6. *Let $U \subseteq \mathbb{R}$, $|U| = N$, and let S be a set of n intervals with both endpoints in U.*

a) *A segment tree for S of depth $O(\log N)$ can be constructed in time $O(n \log N)$. It requires space $O(n \log N)$.*

b) *If time $g(m)$ is required for inserting an interval into or deleting an interval from a node list of size m and g is nondecreasing then an interval can be inserted into or deleted from a segment tree in time $O(g(n) \log N)$.*

Proof: a) Let T be a complete binary tree with $2 \cdot N + 1$ leaves. It is easy to organize the split fields in time $O(N)$ such that T is the skeleton of a segment tree for S. Next observe that for every $I \in S$ we can find the set of nodes v with $xrange(v) \subseteq I$ and $xrange(father(v)) \not\subseteq I$ in time $O(\log N)$. This follows from the fact that the nodes in C_{max} are sons of the nodes in P which in turn are the nodes on the search paths to the two endpoints of interval I. The space bound follows from the observation that every interval is stored in at most $2 \cdot h = O(\log N)$ node lists.

b) Let I be an interval and suppose that we want to delete I from S. The case of the insertions is symmetric and left to the reader. By part a) we can find set

$A = \{v;\ I \in NL(v)\}$ in time $O(\log N)$. We have to delete I from $NL(v)$ for every $v \in A$. Since the node list of any node can certainly contain at most n intervals, since g is nondecreasing and since $|A| = O(\log N)$, the total cost of deleting I is certainly $O(g(n) \log N)$. ∎

In many applications of segment trees we have $g(n) = O((\log n)^k)$ for some small k. Then insertions and deletions take time $O((\log n)^k \log N)$. We are now ready for the first application of segment trees, the **measure problem of iso-oriented rectangles**. Let $R_1, \ldots, R_m$ be a set of iso-oriented rectangles. We want to compute the measure (area) of $R_1 \cup R_2 \cup \cdots \cup R_m$. We solve this problem by plane sweep. Let U be the set of y-coordinates of the corners of the rectangles R_i, $1 \leq i \leq m$. Then $|U| \leq 2 \cdot m$. For each position of the sweep line let S be the set of rectangles intersected by the sweep line. We store the (projections onto the y-axis of the) rectangles in S in a segment tree with respect to universe U. In Figure 101 the sweep line intersects rectangles R_1, R_3 and R_4.

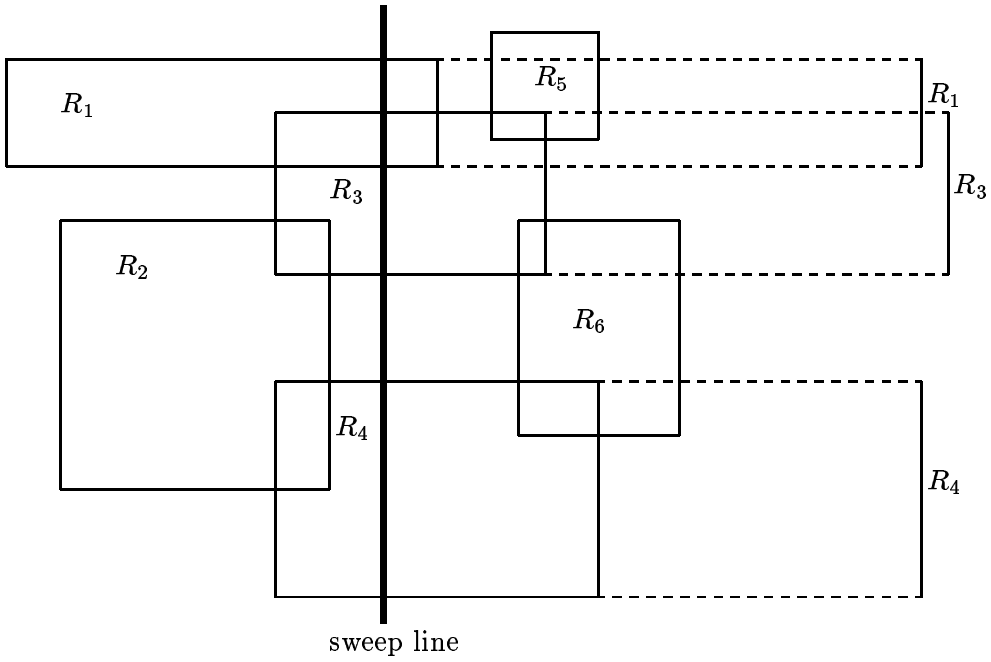


Figure 101.

Their projections (shown in the right half of the diagram) are stored in the segment tree. The node lists are organized in an extremely simple way. We only store the *cardinality* of each node list instead of the complete node list. Thus the space requirement of each node list is $O(1)$. We also store in each node v a quantity $TL(v)$ which is defined as follows. For an interval I let $length(I)$ be the length of I.

Then

$$TL(v) = \begin{cases} length(xrange(v)), & \text{if } NL(v) \neq \emptyset; \\ 0; & \text{if } NL(v) = \emptyset \text{ and } v \text{ is a leaf;} \\ TL(lson(v)) + TL(rson(v)), & \text{if } NL(v) = \emptyset \text{ and } v \text{ is not a leaf.} \end{cases}$$

Then $TL(v)$ is the total length of all atomic segments (in the subtree with root v) which are covered by intervals which belong to a node list of v or a descendant of v. In particular, $TL(root)$ is the length of the union of the intervals stored in S. The x-structure contains the x-coordinates of the corners of the rectangles R_i, $1 \leq i \leq m$, in sorted order. The structure of the plane sweep algorithm is as follows. Suppose that we advance the sweep line to transition point x. Let x_{old} be the previous transition point. Then $TL(root) \cdot (x - x_{old})$ is the area covered by rectangles in the vertical strip given by the scan line positions x_{old} and x. We add this quantity to variable A which sums the total covered area to the left of the scan line. Next, we delete all rectangles R_j from the tree whose right side is at x and insert all rectangles R_j into the tree whose left side is at x. We claim that insertions and deletions take time $O(\log m)$ per rectangle. This can be seen as follows.

We treat the case of a deletion. By Theorem 6 we can find the set $A = \{v;\ I \in NL(v)\}$ in time $O(\log m)$. For all nodes $v \in A$ we have to decrease the counter holding the cardinality of the node list by one. If the counter stays positive then no further action is required for that node. If the counter becomes zero then we need to recompute $TL(v)$ as given by the formula above. Also, we need to propagate the change to the ancestors of v. Since the ancestors of the nodes in A form two paths in the tree the TL-fields of all ancestors of nodes in A can be updated in time $O(\log m)$.

For the sake of completeness we include a detailed program. We assume for this program that an interval with endpoints x and y is stored as half-open interval $(x, y]$. We also assume that atomic segments (x_i, x_{i+1}) and $[x_{i+1}, x_{i+1}]$ are combined to $(x_i, x_{i+1}]$ for all $x_i \in U$. Note that by this convention every interval is still a union of atomic segments. Also, the split field of node v directs a search for x to the left iff $x \leq split(v)$. In the program we maintain the invariant that $xrange(v) = (a, b]$, $I = (x, y]$, $xrange(v) \cap I \neq \emptyset$ and $xrange(father(v)) \not\subseteq I$.

The $O(\log m)$ time bound for procedure *Delete* is readily established from the program text. Only nodes v with $xrange(v) \cap I \neq \emptyset$ and $xrange(v) \not\subseteq I$ generate recursive calls. By Lemma 2 there are at most $2 \log m$ such nodes. This establishes the time bound.

Procedure *Insert* is a minor variant of procedure *Delete*. We only have to replace lines (3) to (8) by

then $v\uparrow.count \leftarrow v\uparrow.count + 1$;
$v\uparrow.TL \leftarrow b - a$

We have now shown that insertions and deletions take time $O(\log m)$ per rectangle. Since a total of m rectangles has to be handled, the cost of the sweep is $O(m \log m)$. The cost of sorting the corners of the rectangles according to x- and y-coordinate

```
(1) procedure Delete(x, y, v, a, b);
(2) begin       if x ≤ a and b ≤ y  co xrange(v) ⊆ I oc
(3) then v↑.count ← v↑.count − 1 fi;
(4) if v↑.count = 0
(5) then        if v↑.stat = leaf
(6)             then v↑.TL ← 0
(7)             else v↑.TL ← v↑.lson↑.TL + v↑.rson↑.TL
(8)             fi
(9) else        if x ≤ v↑.split
(10)            then Delete(x, y, v↑.lson, a, v↑.split)
(11)            fi;
(12)            if y ≥ v↑.split
(13)            then Delete(x, y, v↑.rson, v↑.split, b)
(14)            fi;
(15)            v↑.TL ← v↑.lson↑.TL + v↑.rson↑.TL
(16) fi
(17) end.
```

Program 17

is also $O(m \log m)$. The former sort is required for setting up the x-structure and the latter for computing U. we summarize our discussion in

Theorem 7. *The measure problem for a set of m isooriented rectangles can be solved in time $O(m \log m)$ and space $O(m)$.*

Proof: The time bound is obvious from the discussion above. The space bound follows from the fact that only $O(1)$ words are needed per node of the segment tree. ∎

Further applications of segment trees can be found in Sections 5.1.4 and 5.3 and in the exercises. In particular, we will show how to solve the measure problem for arbitrary polygons and how to do hidden line elimination in Section 5.1.4. In Section 5.3 we will use segment trees for solving intersection problems of iso-oriented objects in three- and higher-dimensional space.

We close this section with a discussion of dynamic segment trees. Let S be a set of intervals on the real line and let U be the set of left and right endpoints of the intervals in S. Let $n = |S|$ and $= |U| \leq 2 \cdot n$. A dynamic segment tree for S is based on a D-tree (cf. Section 8.6.2) with $2 \cdot N + 1$ leaves, one for each atomic segment. As before, we have atomic segments $(-\infty, x_1), [x_1, x_1], (x_1, x_2), \ldots, (x_{N-1}, x_N), [x_N, x_N], (x_N, +\infty)$, where $U = \{x_1 < x_2 < \cdots < x_N\}$. The weights of the atomic segments are defined as follows. Segment (x_i, x_{i+1}) has weight 1, $0 \leq i \leq N$, and the weight of segment $[x_i, x_i]$ is equal to the number of intervals in S which have x_i as an endpoint. As in the case of interval trees we have

Lemma 13. *Let v be a node of a dynamic segment tree based on a D-tree. Then $|NL(v)| = O(th(v))$, where $th(v)$ is the number of leaves below v in the* BB[α]*-tree underlying the D-tree (underlying the dynamic segment tree).*

Proof: Let v be a node of a dynamic segment tree. If $I \in NL(v)$ then $xrange(v) \subseteq I$ and $xrange(father(v)) \not\subseteq I$. Hence at least one endpoint of I is contained in $xrange(father(v))$. We can therefore write $NL(v) = N_1 \cup N_2$, where $N_i = \{I \in NL;$ exactly i endpoints of I are contained in $xrange(father(v))\}$, $i = 1, 2$. A bound on $|N_1|$ is easily derived. We have

$$\begin{aligned} |N_1| &\leq \sum\{weight([x_j, x_j]);\ \text{ the active } [x_j, x_j]\text{-node is a descendant of } v\} \\ &\leq ((1-\alpha)/\alpha \cdot th(father(v)) \leq ((1-\alpha)/\alpha^2) \cdot th(v) \end{aligned}$$

since the fraction $\alpha/(1-\alpha)$ of all $[x_j, x_j]$-leaves are descendants of the active $[x_j, x_j]$-node. Let us turn to N_2 next. If $I \in \mathbb{N}_2$ then $I \subseteq xrange(father(v))$ and hence $xrange(v) \subseteq I$ implies that $xrange(father(v))$ and I have a common endpoint. Hence

$$|N_2| \leq 2 \cdot ((1-\alpha)/\alpha) \cdot th(father(v)) \leq 2 \cdot ((1-\alpha)/\alpha^2) \cdot th(v)$$

by an argument similar to the one above. ∎

We are now in a position to discuss insertions and deletions into dynamic segment trees. We separate two issues. The first issue is to insert or delete an interval without changing the underlying tree structure and the second issue is to rebalance the underlying D-tree by rotations and double rotations.

The total weight of all atomic segments is at most $3 \cdot n + 1$ and hence the depth of a dynamic segment tree is $O(\log n)$. Thus time $O(\log n)$ suffices to locate the set of nodes whose node lists are affected by the insertion. Let us assume that the cost of inserting an interval into or deleting an interval from a node list of size m is $g(m)$, where g is non-decreasing. Then the total cost of updating the node list is

$$\sum_{v \in A} g(|NL(v)| \leq \sum_{v \in A} g(c \cdot th(v)),$$

where A is the set of affected nodes and c is a (small) constant. The set of affected nodes forms (essentially) two paths in the tree and hence for every integer i there are at most d nodes in A with $(1-\alpha)^{-i} < th(v) \leq (1-\alpha)^{-i-1}$ for some (small) constant d. Thus the cost of updating the node lists is

$$C_1 := \sum_{i=0}^{e \log n} d \cdot g((1-\alpha)^{-i}),$$

where $e = -1/\log(1-\alpha)$. In particular, we have $C_1 = O((\log n)^{k+1})$ if $g(m) = O((\log m)^k)$ for some $k \geq 0$ and $C_1 = O(n^\alpha)$ if $g(m) = m^\alpha$ for some $\alpha > 0$.

The cost of rebalancing the underlying D-tree by rotations and double rotations remains to be discussed. Since a double rotation comprises two rotations it suffices to discuss single rotations.

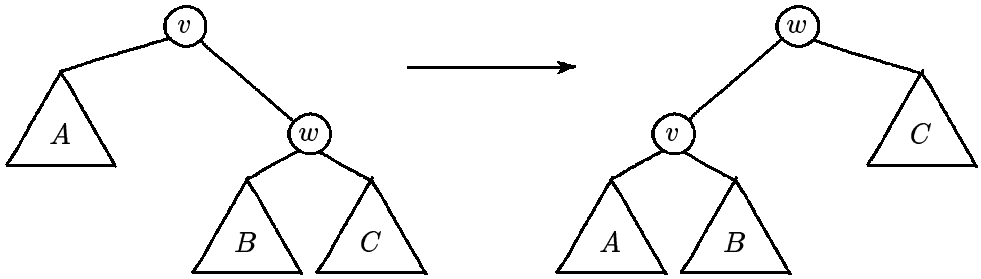


Figure 102.

We have (primes denote the situation after the rotation):

$$NL'(w) \leftarrow NL(v)$$
$$NL'(v) \leftarrow NL(root(A)) \cap NL(root(B)),$$
$$NL'(root(A)) \leftarrow NL(root(A)) - NL'(v),$$
$$NL'(root(B)) \leftarrow (NL(root(B)) - NL'(v)) \cup NL(w),$$
$$NL'(root(C)) \leftarrow NL(root(C)) \cup NL(w).$$

The correctness of these formulae is easily established. For example, we have $I \in NL'(v)$ iff I covers A and B but does not cover C,i.e., $NL'(v) = NL(root(A)) \cap NL(root(B))$. The node lists of all other nodes remain unchanged. This follows from the fact that the node list $NL(z)$ of a node z is determined by $xrange(z)$ and $xrange(father(z))$ and that only the xranges of nodes v and w change.

The total size of the node lists $NL(v)$, $NL(w)$, $NL(root(A))$, $NL(root(B))$ and $NL(root(C))$ is $O(th(v))$ by Lemma 13. We will therefore assume that the cost of a single rotation at node v is $O(f(th(v))$ for some non-decreasing function f. Hence the total cost of all rebalancing operations is

$$C_2 := O\left(n \cdot \sum_{i=0}^{e \log n} f((1-\alpha)^{-i}) \cdot (1-\alpha)^i\right),$$

where $e = -1/\log(1-\alpha)$ by Theorem 5 of Section 3.6.2. In particular, if $f(v) = O(m(\log m)^k)$ for some $k \geq 0$ then $C_2 = O(n(\log n)^{k+1})$, i.e., the amortized cost per insertion and deletion is $O((\log n)^{k+1})$. We summarize in:

Theorem 8. *Let S be a set of n intervals. Then the amortized insertion and deletion cost in a dynamic segment tree for S is*

$$O\left(\log n + \sum_{i=0}^{e \log n} (g((1-\alpha)^{-i}) + f((1-\alpha)^i) \cdot (1-\alpha)^i)\right),$$

where g and f are defined as above. In particular, $g(m)$ is the cost of inserting (deleting) an interval into (from) a node list of size m and $f(m)$ is the cost of a rotation at a node of thickness m. In particular, if $g(m) = (\log m)^k$ and $f(m) = m(\log m)^k$ for some $k \geq 0$ then the amortized cost is $O((\log m)^{k+1})$.

Proof: Obvious from the discussion above. ∎

8.5.1.4. Path Decomposition and Plane Sweep for Non-Iso-Oriented Objects

In the preceding Sections 5.1.1 to 5.1.3 we developed special data structures for plane sweep algorithms on iso-oriented objects, namely interval, segment, and priority search trees. In this section we want to show that many plane sweep algorithms can be extended to more general objects, in particular to collections of polygons. The main idea is to transform the collection of polygons into a planar subdivision by adding additional vertices at edge intersections and then to decompose the triangulated subdivisions into paths as described in Section 3.1.2. The path decomposition splits the plane into an ordered set of strips (of varying width) which we can use as atomic segments in a segment tree.

In this section we treat two problems: the measure problem for a union of simple polygons and the hidden line elimination problem. We discuss the first problem at length showing that path decompositions permit to transfer an algorithm from the iso-oriented to the general case. When we turn to the second problem, we take this transfer for granted and treat the general problem directly.

Let $Q_1, \ldots, Q_m$ be a set of simple polygons with a total of n vertices. Our goal is to compute the area covered by the union $Q_1 \cup \cdots \cup Q_m$ of the polygons.

Our approach consists of extending the algorithm of Section 5.1.3 to this more general situation. We can proceed in at least two ways. Making a first attempt, we use dynamic segment trees instead of static segment trees. More precisely, consider an arbitrary position of the sweep line. It intersects some of the edges of polygons $Q_1, \ldots, Q_m$. Take the ordered set of intersected line segments as the universe of the segment tree. In this way, each leaf of the segment tree corresponds to a (half-open) interval between adjacent active line segments. We invite the reader to carry out the fairly complicated details of this approach. It will allow him/her to really appreciate the elegance of the second solution.

For the second approach we first turn the set of polygons into a planar subdivision by adding new vertices at intersections of edges, then triangulate the subdivision and decompose the planar subdivision into an ordered set of strips such that every interval arising in the plane sweep is a union of strips, and finally base the plane sweep on a *static* segment tree for the set of strips. The details are as follows.

Let $Q_1, \ldots, Q_m$ be a set of polygons with a total of n edges. Assume w.l.o.g. that no edge of any polygon is vertical and that there is a total of k intersections

of edges. We can find the k points of intersection in time $O((n+k)\log n)$ by Theorem 1 of Section 4.1. We add the k points of intersection as additional vertices and call the resulting set of vertices V. Then $|V| = n+k$ and edges intersect only at common endpoints, i.e., we have a planar subdivision with vertex set V. Next, we add the edges of the convex hull of set V and then triangulate all finite faces of the planar subdivision. Call the resulting subdivision $\hat{G}$. We can construct $\hat{G}$ in time $O((n+k)\log(n+k))$ by Theorem 2 of Section 2 and Theorem 3 of Section 4.2.

Let s, t be the vertices of minimal and maximal x-coordinate and let $P_1, \ldots, P_p$ be a path decomposition of planar subdivision $\hat{G}$ in the sense of Section 3.2.2, i.e.,

1) each P_i is an x-monotonous (in Section 3.2.2 we required that the path should be y-monotonous) path from s to t,

2) each edge of P_i belongs to at least one path,

3) if vertical line L intersects P_i and P_j and $i < j$ then $L \cap P_i$ is not below $L \cap P_j$.

We have seen in Section 3.2.2 that the implicit representation of a complete path decomposition can be computed in time $O(n+k)$, i.e., we compute integer arrays L and R such that edge e belongs to P_i iff $L(e) \leq i \leq R(e)$. Moreover, $p \leq n+k$.

A path decomposition of $\hat{G}$ divides in a natural way the vertical strip defined by vertices s and t into an ordered set of "horizontal" strips. The i-th strip is the region between paths P_i and P_{i+1}. We use these strips as atomic segments of the segment tree. More precisely, we use a *static* segment tree with universe $[1\,..\,n]$, where integer i corresponds to path P_i. We can now proceed in almost complete analogy to Section 5.1.3, where we treated the measure problem for iso-oriented objects.

We maintain the following data structures. Consider a fixed position of the sweep line. For each polygon Q_i we maintain the set of edges of the polygon intersected by the sweep line in a balanced tree sorted according to the y-coordinate of the intersections. Actually, it suffices to keep the the indices of the paths of the composition containing the edges in a balanced tree. The sweep line intersects polygon Q_i in some intervals. For each interval, which then extends between two paths of the decomposition, say P_a and P_b, we store the interval $(a\,..\,b]$ in the segment tree. As before, actual node lists are not maintained, only their cardinality has to be stored. In addition to $CNL(v)$, the cardinality of v's node list, we store two other fields in every node of the segment tree: $LE(v)$, the length of v's xrange as a function of the position of the sweep line, and $TL(v)$, the total length of the atomic segments below v which are covered. Field $TL(v)$ is also a function of the position of the sweep line. Field $LE(v)$ is defined as follows. If $xrange(v) = (i\,..\,j]$ then $LE(v)$ is the linear function $a + bx$, where $a + bx$ is the vertical distance of paths P_i and P_j at x-coordinate x. Note that $LE(v) = LE(lson(v)) + LE(rson(v))$ if v is not a leaf. Field $LE(v)$ was a constant in the iso-oriented case; we achieve the generalization of plane sweep algorithm from the iso-oriented case by maintaining the fields $LE(v)$ of the nodes of the segment tree. Finally, fields $TL(v)$ are defined

exactly as in the iso-oriented case, i.e.,

$$TL(v) = \begin{cases} LE(v), & \text{if } NL(v) \neq \emptyset; \\ 0, & \text{if } NL(v) = \emptyset \text{ and } v \text{ is a leaf;} \\ TL(lson(v)) + TL(rson(v)), & \text{if } NL(v) = \emptyset \text{ and } v \text{ is not a leaf.} \end{cases}$$

Note that $TL(v)$ is a linear function for every node v of the segment tree and that $TL(root)$ is the linear function which yields the total length of the union of the intervals stored in the tree.

The x-structure of the plane sweep contains the vertex set V of planar subdivision $\hat{G}$ in increasing order of x-coordinate. Suppose now that we advance the sweep line from transition point z_{old} with x-coordinate x_{old} to transition point z with x-coordinate x. The following three actions are required:

1) compute the area covered by the union of the rectangles in the vertical strip between x_{old} and x;
2) update the LE-fields of some of the nodes of the segment tree;
3) update the intervals stored in the segment tree for every polygon Q_i having v as vertex.

Action 1) is very simple. If $TL(root) = a + bx$ is the linear function stored in the TL-field of the root, then $a(x - x_{old}) + b(x^2 - x_{old}^2)/2$ is the area covered by the union of the polygons in the vertical strip between x_{old} and x.

Action 2) requires some care; there is no equivalent in the algorithm for iso-oriented objects. Assume that edges $e_1, \ldots, e_l$ end in vertex z and that edges $e', \ldots, e'_r$ start in vertex z. Assume also that ending and starting edges are ordered from top to bottom.

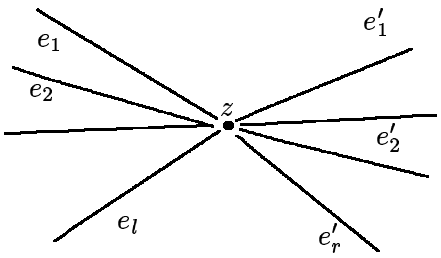


Figure 103.

We know that paths $P_a, \ldots, P_b$, where $a = L[e_1]$ and $b = R[e_l]$ run through vertex z. Also, paths $P_{L[e_i]}, \ldots, P_{R[e_i]}$ use edge e_i, $1 \leq i \leq l$, and paths $P_{L[e'_j]}, \ldots, P_{R[e'_j]}$ use edge e'_j, $1 \leq j \leq r$. It is useful to model the transition through vertex z in two steps. We first move from the left into node z and then leave node z to the right. The first step shrinks the paths between paths $P_{R[e_i]}$ and $P_{L[e_{i+1}]}$ (note that $R[e_i] + 1 = L[e_{i+1}]$), $1 \leq i < l$, to empty sets and the second step expands the strip between paths $P_{R[e'_j]}$ and $P_{L[e'_{j+1}]}$, $1 \leq j < r$, to non-trivial intervals. Therefore,

the following code suffices to make required changes of the LE-field and the induced changes of the TL-fields.

---

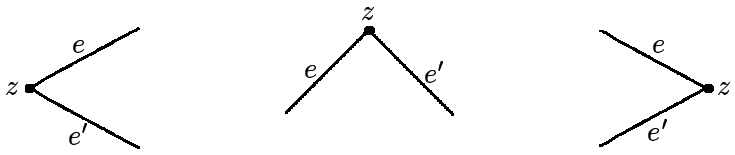
(1) **for** i **from** 1 **to** $l-1$
(2) **do** LE(leaf corresponding to atomic segment $(R[e_i], R[e_i]+1]) \leftarrow 0$;
(3) propagate change in LE-field and induced change in TL-fields towards the ro
(4) **od**;
(5) **for** j **from** 1 tot $r-1$
(6) **do** LE(leaf corresponding to atomic segment $(R[e'_j], R[e'_j]+1]) \leftarrow a_j + b_j x$;
(7) propagate change in LE-field and induced change in TL-fields towards the ro
(8) **od**;
(9) LE(leaf corresponding to atomic segment $(L[e'_1]-1, L[e'_1]]) \leftarrow a_0 + b_0 x$;
(10) LE(leaf corresponding to atomic segment $(R[e'_r], R[e'_r]+1]) \leftarrow a_r + b_r x$;
(11) propagate both changes and the induced changes towards the root.

Program 18

A short remark is necessary for lines (2), (6), (9) and (10). In line (2) we store the function which is constantly zero in the LE-field. In line (6) (similarly in lines (9) and (10)) we store the linear function which yields the distance between edge e'_j and e'_{j+1} as a function of the x-coordinate in the LE-field. In lines (9) and (10) we update LE-fields corresponding to the strip between P_{a-1} and P_a, $a = L[e_1]$, and P_b and P_{b+1}, $b = R[e_l]$.

The cost of action two is $O((l+r)\log(n+k))$ since each execution of lines (3), (7) and (11) takes time $O(\log(n+k))$. Since each edge is handled twice in action two during the entire sweep the total cost of action two is $O((n+k)\log(n+k))$.

Action three is very similar to what we already know from the iso-oriented case. Suppose that z is a vertex of polygon Q_i. In general, z is a vertex of several polygons. Then z is either a starting, bend or ending vertex of Q_i.



start *bend* *end*

Figure 104.

Let e and e' be the two edges of Q_i which are incident to z. Using the tree representation of the set of active edges of polygon Q_i we can easily find the immediate predecessor e_T of e and e' and the immediate successor e_B of e and e' in the top to bottom order of active edges and also insert or delete e and e' depending on what is appropriate. Note that e_T and e_B do not necessarily exist. We can also determine whether or not the region below e belongs to Q_i by inspecting the order

in which edges e and e' appear on the boundary of Q_i. The whole operation takes time $O(\log n)$.

We can now finish the transition by making a few insertions into and deletions from the segment tree. For example, if z is a starting vertex and the region between e and e' does not belong to Q_i then we first delete interval $(L[e_T], R[e_B]]$ from the segment tree and then insert intervals $(L[e_T], R[e]]$ and $(L[e'], R[e_B]]$. All other cases are similar. The precise code for the insertions and deletions is identical to The code given in Section 5.1.3. Thus processing a vertex z takes time $O(m(z) \log n)$, where $m(z)$ is the number of polygons having z as a vertex. Since $\sum_{z \in V} m(z) = O(n+k)$ we conclude that the total cost of action three is $O((n+k) \log n)$. We summarize in

Theorem 9. *Let $Q_1, \ldots, Q_m$ be a set of simple polygons with n edges and k intersections of edges. Then the area of the union of polygons can be computed in time $O((n+k) \log n)$.*

Proof: We infer a bound of $O((n+k)\log(n+k))$ from the preceding discussion. Since $k \leq n^2$ the bound follows. ∎

Our second application of path decomposition to plane sweep algorithms is a hidden line elimination algorithm. Let $Q_1, \ldots, Q_m$ be a set of simple plane polygons in $\mathbb{R}^3$ with a total of n edges. (A polygon Q in $\mathbb{R}^3$ is plane if there is a plane containing all vertices of Q). We assume that polygons Q_i and Q_j, $1 \leq i < j \leq m$, do not intersect except at points having a common boundary. We want to compute the visible parts under orthogonal projection into the xy-plane (from $+\infty$ in the z-direction). A similar algorithm works for perspective projections.

Let Q_i' be the projection of Q_i into the xy-plane, $1 \leq i \leq m$, and let k be the number of intersections of edges of the projected polygons Q_i'. As before we obtain a planar subdivision G with vertex set V, $|V| = n + k$, by adding the intersections as additional vertices, by adding the edges of the convex hull, and by triangulating all finite faces. Also, as before, we construct a path decomposition $P_1, \ldots, P_p$ of the subdivision $\hat{G}$.

We will now show how to solve the hidden line elimination problem by space sweep. More precisely, we sweep a plane, which is parallel to the yz-plane, from $-\infty$ to $+\infty$ in x-direction. Consider a fixed position of the sweep plane PL. It intersects some of the polygons Q_i. Each intersection is a set of straight line segments (note that the Q_i's are not necessarily convex). Let $L_1, \ldots, L_N$ be the set of straight line segments which can be obtained as an intersection of the sweep plane PL and the polygons Q_i, $1 \leq i \leq m$. Note that line segments $L_1, \ldots, L_N$ do not intersect except at common endpoints. Furthermore, let L_i' be the projection of L_i into the xy-plane. The idea of the algorithm consists of maintaining the projections L_i' in a segment tree as follows. We use a *static* segment tree with universe $[1\,..\,p]$. Element j of the universe corresponds to path P_j of the path decomposition, $1 \leq j \leq p$. Consider an arbitrary L_i'. It extends between paths P_a and P_b, say, of the decomposition. We therefore store interval $(a, b]$ in the segment tree. Finally, we

have to discuss the organization of the node lists. Let v be a node of the segment tree with $xrange(v) = (c, d]$. Then node v represents the strip between paths P_c and P_d of the decomposition. Then node list $NL(v)$ of node v is a set of projected intervals; node v represents the sub-interval between paths P_c and P_d. Since the line segments $L_1, \ldots, L_N$ do not intersect except at common endpoints we can order the node list $NL(v)$ according to the z-coordinate. We therefore postulate that $NL(v)$ is maintained as a balanced tree with its elements ordered according to the z-coordinate.

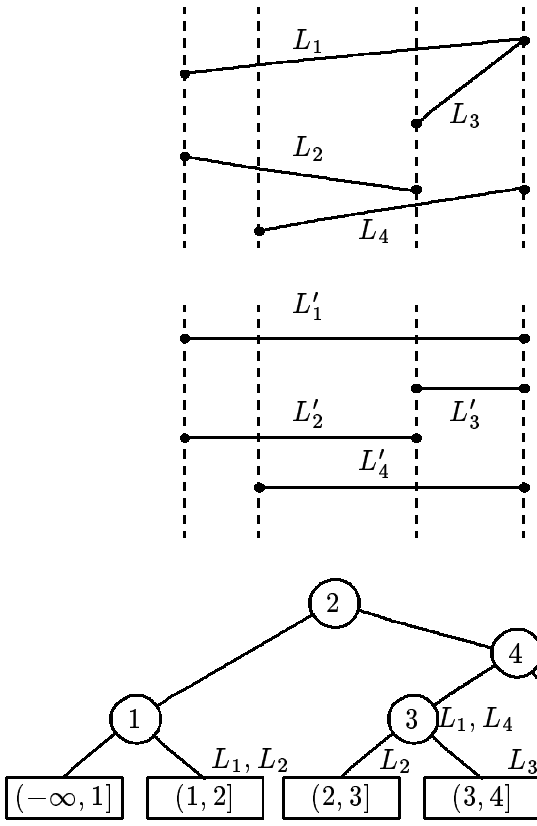


Figure 105.

In the example of Figure 105, we have four line segments L_1, L_2, L_3, L_4 and use a segment tree with universe $[1\,..\,4]$. The ordered node lists are shown in the vicinity of the nodes; e.g., the node list of node 3 is L_1, L_4 in that order.

We can now give the details of the sweep. The transition points are the vertices V in increasing order of the x-coordinate. Suppose that the sweep reaches vertex z. In general, z is a vertex of several polygons Q_i. Two actions are required:

1) Update the state of the sweep;

2) Check for visibility.

The first action is very similar to the third action of the previous algorithm. If z is a vertex of polygon Q_i then it is either a starting bend or ending vertex. In all three cases a few insertions or deletions update the segment. For example, if z is an ending vertex, i.e., two edges, say e and e', of Q_i terminate in z and the region between e and e' does not belong to Q_i, then let e_T and e_B be the immediate successor and predecessor edges in the tree representation (cf. the previous algorithm) of active edges of polygon Q_i. We have to delete intervals $(L[e_T], R[e]]$ and $(L[e'], R[e_B]]$ from the segment tree and we have to insert interval $(L[e_T], R[e_B]]$ into the segment tree. When intervals are inserted into the tree we have to insert the interval into $O(\log n)$ node lists. In each node list we have to insert the interval according to is height at the appropriate position for a cost of $O(\log n)$, each. Thus inserting or deleting an interval has cost $O((\log n)^2)$ and hence the total cost of actions one is $O((n+k)\cdot(\log n)^2)$.

Action two is a check for visibility. It is performed at every transition point after all insertions and deletions required by action one are performed. Let A be the set of edges starting in vertex z. For each edge $e \in A$, we can check its visibility in time $O((\log n)^2)$ as follows. Let $a = L[e]$ and $b = R[e]$, i.e., paths $P_a, \ldots, P_b$ use edge e. Let v be any node of the segment tree which lies on the path from either atomic segment $(b-1, b]$ or atomic segment $(a, a+1]$ to the root. We can clearly check in time $O(\log n)$ whether v's node list contains an interval which covers e by comparing the highest interval in v's node list with edge e and hence we can check in time $O((\log n)^2)$ whether any of these nodes contain an interval which covers e. We claim that if none of these nodes contain an interval which covers e then e is visible. Assume otherwise. Then there is a polygon which covers e. Let I be the intersection of that polygon with the sweep plane and let I' be the projection of I. Then I extends between active edges e' and e'', where $L[e'] \leq L[e]$ and $R[e''] \geq R[e]$. Since I is stored as interval $(L[e'], R[e'']]$ in the segment tree the claim follows.

Visibility of a single edge can thus be checked in time $O((\log n)^2)$ and hence the total time needed for all visibility checks is $O((n+k)\cdot(\log n)^2)$. We summarize in

Theorem 10. *Let $Q_1, \ldots, Q_m$ be a set of simple plane polygons in $\mathbb{R}^3$ with a total of n edges. Assume further, that polygons Q_i and Q_j, $1 \leq i < j \leq m$, do not intersect at common boundary points and that there are k intersections of edges in the orthogonal projection of the polygons into the xy-plane. Then the hidden line elimination problem can be solved in time $O((n+k)\cdot(\log n)^2)$ and space $O(n+k)$.*

Proof: By the discussion above. ∎

Two alternative algorithms for hidden line elimination are discussed in the exercises both of which improve the algorithm described above. The first alternative has the same basic structure but uses zig-zag decompositions instead of complete path decompositions. This reduces the space requirement to $O(n)$. Since k might be as large as $\Omega(n^2)$ this is a significant improvement. Unfortunately, it does not seem to be always possible to use zig-zag decompositions instead of complete path

decompositions; e.g., it is not clear how to use them for the measure problem. The second alternative keeps the space requirement at $O(n+k)$ but reduces running time to $O((n+k)\cdot \log n)$. It constructs a planar subdivision by introducing new vertices at edge intersections, then computes local visibility in the vicinity of vertices and finally determines global visibility by a sweep in z-direction.

8.5.2. Divide and Conquer on Iso-Oriented Objects

In this section we will show how the divide-and-conquer paradigm can be applied to solving problems on sets of iso-oriented objects. In particular it can be used to report intersections of line segments, to report containment of points in rectangles, to report rectangle intersections and to compute the measure and contour of a set of rectangles. In all these cases the divide-and-conquer paradigm yields algorithms which match the performance of the best plane sweep algorithms. In some cases, e.g., the contour problem, the algorithm based on the divide-and-conquer paradigm is conceptually simpler and computationally superior. The section is organized as follows. In 5.2.1 we illustrate the paradigm using the line segment intersection problem and in 5.2.2 we apply it to the measure and contour problem.

8.5.2.1. The Line Segment Intersection Problem

Let S be a set of n horizontal and vertical line segments $L_1, \ldots, L_n$. The line segment intersection problem requires the computation of all pairs of intersecting line segments. In Section 4.1 we have seen (cf. Exercise 23) that the set of s pairs of intersection can be computed in time $O(n \log n + s)$. We will now describe an alternative solution based on the divide-and-conquer paradigm.

Theorem 11. *Let S be a set of n horizontal and vertical line segments. Then the s pairs of intersection can be computed in time $O(n \log n + s)$.*

Proof: Let S_H (S_V) be the set of horizontal (vertical) line segments in S. A horizontal line segment L is given as a triple $(x_1(L), x_2(L), y(L))$, a vertical line segment is given as $(x(L), y_1(L), y_2(L))$.

In order to simplify the exposition and to describe the basic idea more clearly we assume first that all x- and y-coordinates are distinct. We will come back to the general problem at the end of the proof.

We apply the divide-and-conquer paradigm as follows. The divide step divides the plane (and more generally a vertical strip defined by two vertical lines) into two parts by means of a vertical dividing line. In the conquer step we deal with both subproblems and in the merge step we combine the answers. The speed of the method is due to the clever treatment of horizontal line segments. They are only handled in subproblems which "contain" exactly one endpoint of the line segment,

and hence are handled at most $O(\log n)$ times if we divide the problem about equally at each step. Similarly, vertical line segments are only handled in those subproblems which contain them.

A **frame** $F = (f_1, f_2)$ is the region between vertical lines $x = f_1$ and $x = f_2$ with $f_1 < f_2$. Frames define subproblems of our intersection problem. The recursive algorithm *Intersect* (to be described) applied to a frame F does two things.

1) It reports all intersections between vertical line segments contained in the frame and horizontal line segments having at least one endpoint in the frame. Note however that if both endpoints are in the frame then the intersection is actually reported by a recursive call.
2) It computes the sets $VERT(F)$, $LEFT(F)$ and $RIGHT(F)$ to be used at higher levels of the calling hierarchy.

The set $VERT(F)$ is a set of intervals. It contains the projections of all vertical line segments contained in F onto the y-axis. The sets $LEFT(F)$ and $RIGHT(F)$ are sets of points. The set $LEFT(F)$ is the set of intersections of horizontal line segments having one endpoint in frame F with the left boundary of the frame. The set $RIGHT(F)$ is defined symmetrically.

We are now in a position to give a (very high level) description of procedure *Intersect*. In this description we use the word object to refer to either vertical line segments or a left or right endpoint of a horizontal line segment. Procedure *Intersect* is called in the main program with a frame containing all objects.

The correctness of this procedure is readily established. Note first that sets $LEFT$, $RIGHT$ and $VERT$ are correctly computed in all cases. This is clear for lines (3)–(10) and is easy to establish for lines (12)–(17). In line (14) we determine the set of horizontal line segments having one endpoint, each in the two subframes of frame F. In lines (15) and (16) the sets $LEFT$ and $RIGHT$ are computed. Note that a horizontal line segment has one endpoint in F and intersects the left boundary of F if either it has one endpoint in F_1 and intersects the left boundary of F_1 or has one endpoint in F_2, intersects the left boundary of F_2 and does not have the other endpoint in F_1. Finally, the set $VERT$ is certainly correctly computed in line (17).

We finally prove that all intersections are reported correctly in line (18). We need to explain line (18) in more detail first. The set $RIGHT_1 - LR$ is the set of y-coordinates of left endpoints of horizontal line segments which have one endpoint in frame F_1 and intersect the right boundary of frame F. Thus these line segments extend completely through frame F_2 and therefore we can report an intersection with every vertical line segment in frame F_2 whose associated interval in $VERT_2$ contains the y-coordinate of the horizontal line segment. The intersections induced by $RIGHT_1 - LR$ and $VERT_2$ are hence all pairs (H, V) of horizontal line segment H and vertical line segment V such that the y-coordinate $y(H)$ of H is contained in $RIGHT_1 - LR$, the projection $I(V)$ of V onto the y-axis is contained in $VERT_2$, and $y(H) \in I(V)$. The intersections induced by $LEFT_2 - LR$ and $VERT_1$ are defined similarly.

It is now clear that only actual intersections are reported in line (18). We will

```
(1) procedure Intersect(F, VERT, LEFT, RIGHT);
    co          F is input parameter and VERT, LEFT, RIGHT are
                output parameters oc
(2) if exactly one object is contained in frame F
(3) then if the object is a vertical line segment L = (x, y1, y2)
(4)       then VERT ← {[y1, y2]}; LEFT ← RIGHT ← ∅
(5)       else VERT ← ∅;
(6)            if the object is a left endpoint of a
                 horizontal line segment L = (x1, x2, y)
(7)            then LEFT ← ∅; RIGHT ← {y}
(8)            else RIGHT ← ∅; LEFT ← {y}
(9)            fi
(10)      fi
(11) else  choose a vertical line x = f such that about half of the
           objects contained in frame F = (f1, f2) lie to the left of
           the line, let F1 = (f1, f) and F2 = (f, f2);
(12)       Intersect((f1, f), VERT1, LEFT1, RIGHT1);
(13)       Intersect((f, f2), VERT2, LEFT2, RIGHT2);
(14)       LR ← RIGHT1 ∩ LEFT2;
(15)       LEFT ← LEFT1 ∪ (LEFT2 − LR);
(16)       RIGHT ← (RIGHT1 − LR) ∪ RIGHT2;
(17)       VERT ← VERT1 ∪ VERT2;
(18)       report all intersections induced by RIGHT1 − LR and VERT2
           and by LEFT2 − LR and VERT1
(19) fi
(20) end
```

Program 19

next show that all intersections are reported exactly once. Let $H(V)$ be a horizontal (vertical) line segment and assume that H and V intersect. Consider a lowest node, say v, in the tree of recursive calls such that the frame, say F, associated with that recursive call contains V and exactly one endpoint of H. Then $L \cap H$ is not reported in any proper ancestor of v because the frames associated with the ancestors of v contain both endpoints of H. Also $L \cap H$ is not reported in any proper descendant of v because the frames associated with descendants either do not contain v or contain no endpoint of H. Thus $L \cap H$ is reported at most once. Finally, $L \cap H$ is reported in node v because one of the subframes of F contains V, H extends completely through that subframe, and the other subframe contains an endpoint of H. This completes the proof of correctness.

Let us turn to the analysis next. In order to support the divide step we sort the objects by the x-coordinate and store the sorted sequence in an array. Then line (11) and the tests in lines (2), (3) and (6) take time $O(1)$. The sets *LEFT* and *RIGHT* are realized as ordered (according to the y-coordinate) linked lists and

set *VERT* is also realized as a linked list. It contains the intervals in sorted order according to the bottom endpoint. It is now easy to see that lines (14)–(17) take time $O(\bar{n})$, where $\bar{n}$ is the number of objects to be handled. Also line (18) takes time $O(\bar{n} + \bar{s})$, where $\bar{s}$ is the number of intersections reported. In all four lines a simple list traversal algorithm (cf. Section 2.1.4) suffices. We conclude that the cost of a call of *Intersect* is $O(\bar{n} + \bar{s})$, where $\bar{n}$ is the number of objects in the frame and $\bar{s}$ is the number of intersections reported in line (18). Since every intersection is reported exactly once the sum of the $O(\bar{s})$-terms is $O(s)$. For the $O(\bar{n})$-terms we obtain the recursion

$$T(n) = 2 \cdot T(n/2) + O(n)$$

which has solution $T(n) = O(n \log n)$. This proves the theorem for the case that all x- and y-coordinates are unique.

Multiple x-coordinates are treated as follows. Again we sort the objects according to the x-coordinate. Also, for each x-coordinate we precompute the sets *VERT*, *LEFT* and *RIGHT* for the frame consisting of the degenerated rectangle defined by the x-coordinate. it is easy to see that the overall computation takes time $O(n \log n)$ by sorting.

The test in line (2) changes to

(2′) if all objects contained in the frame have the same x-coordinate.

Then lines (3) to (10) can be replaced by a look-up in the set of precomputed solutions. Line (11) changes to

(11′)Choose vertical line $x = f$, where f is the median of the multi-set of x-coordinates of the objects contained in frame F.

We then split the objects into three parts: the objects to the left of (on, to the right of) the dividing line. We apply the algorithm recursively to the objects to the left (right) of the dividing line, the objects on the dividing line are handled by a look-up in the set of precomputed solutions. It is now easy to adapt lines (14)–(18) to the new situation such that they still operate in time $O(\bar{n} + \bar{s})$. Thus we still have the same recursion for the running time and hence the running time is $O(n \log n + s)$.

The extension to multiple y-coordinates is now very simple. We only have to treat sets *LEFT*, *RIGHT*, and *VERT* as multisets. More precisely, with each element of set *LEFT* (the other two sets are treated similarly) we associate an ordered list of all objects having that y-coordinate. The order is arbitrary, but must be uniform (e.g., we might order horizontal line segments according to some arbitrary but fixed linear order of S_H). Then the intersection process in line (14) is still a simple merging process and takes time $O(\bar{n})$. This shows that multiple y-coordinates can also be handled within the $O(n \log n + s)$ bound and hence completes the proof. ∎

A simple variant of the algorithm above can also be used to solve the inclusion problem of points in rectangles and of the intersection problem for sets of rectangles (cf. Exercise 41 and 42).

8.5.2.2. The Measure and Contour Problem

Let $R_1, \ldots, R_n$ be a list of n iso-oriented rectangles in $\mathbb{R}^2$. The measure problem includes the task of computing the area of the union $\mathbb{R}_1 \cup \cdots \cup R_n$ of the rectangles. We developed an $O(n \log n)$ plane sweep algorithm for this problem in Section 5.1.3. The contour problem is also a property of the union of rectangles. It requires the computation of the contour, i.e., the boundary between covered and uncovered area, of $R_1 \cup \cdots \cup R_n$. Figure 106 shows a set of rectangles and its contour.

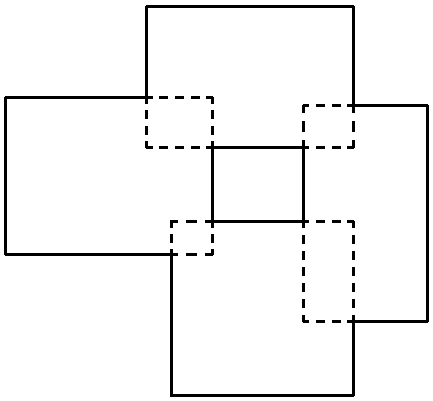


Figure 106.

The main goal of this section is to develop an $O(n \log n + p)$ algorithm for the contour problem based on the divide-and-conquer paradigm. Here p is the number of straight line segments in the contour. It is also possible to achieve this optimal running time by a plane sweep algorithm which however is conceptually more difficult and computationally inferior.

The basic structure of the divide-and-conquer algorithm is again a hierarchical decomposition of the plane into frames. If $F = (f_1, f_2)$ is a frame, let $rect(F)$ be the set of rectangles having at least one vertical edge within frame F. We use V to denote the set of vertical edges of the rectangles. We want to compute two types of information about $rect(F)$: the first type, sets $P(F)$ and $stripes(F)$, describes the contour of the rectangles in $rect(F)$ within frame F, the second type, sets $L(F)$ and $R(F)$, supports the merge step. Let us describe the four sets in more detail: $P(F)$ is the ordered set of y-coordinates of the vertices of the rectangles in $rect(F)$. Let $P(F) = \{y_1 < y_2 < \cdots < y_m\}$, $m \leq 2 \cdot n$. The horizontal lines $y = y_i$, $1 \leq i \leq m$, divide frame F into a sequence of horizontal stripes. The i-th stripe $S_i(F)$ is defined by horizontal lines $y = y_i$ and $y = y_{i+1}$, $1 \leq i < m$. For each such stripe we store an ordered list $list_i = list(S_i(F))$ of intervals $[x_1^i, x_2^i], \ldots, [x_{2k-1}^i, x_{2k}^i]$ such that

1) $f_1 = x_0^i \leq x_1^i \leq x_2^i \leq \cdots \leq x_{2k}^i \leq x_{2k+1}^i = f_2$
2) the area (within the i-th stripe $S_i(F)$) between vertical lines $x = x_{2j-1}^i$ and $x = x_{2j}^i$ is covered, $1 \leq j \leq k$, and the area between vertical lines $x = x_{2j}^i$ and $x = x_{2j+1}^i$ is uncovered, $0 \leq j \leq k$, by the rectangles in $rect(F)$.

In other words

$$\left(\bigcup\{R;\ R \in rect(F)\}\right) \cap ([f_1, f_2] \times [y_i, y_{i+1}]) = \bigcup_{j=1}^{k} ([x^i_{2j-1}, x^i_{2j}] \times [y_i, y_{i+1}]).$$

$stripes(F)$ is now the sequence $(list_1, list_2, \ldots, list_{m-1})$. The sets $L(F)$ and $R(F)$ are sets of intervals. $L(F)$ is the set of intervals obtained by intersecting the rectangles in $rect(F)$ with the left boundary of frame F; $R(F)$ is defined symmetrically. In other words, $L(F)$ is the set of projections onto the y-axis of the right boundaries of those rectangles in $rect(F)$ which have their left boundary outside F. Figure 107 illustrates these definitions.

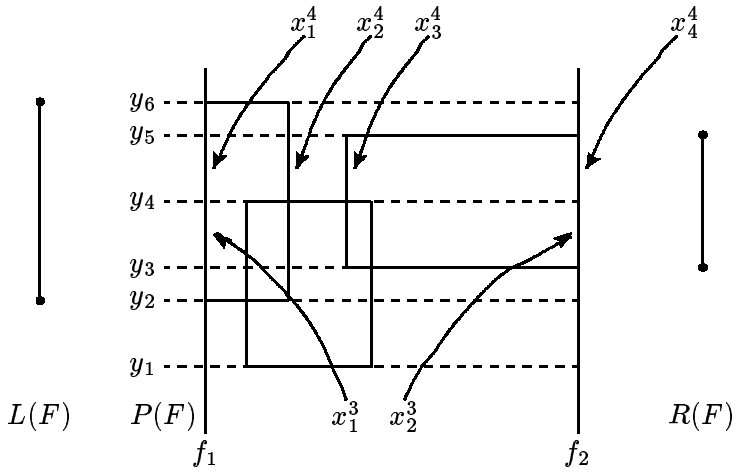


Figure 107.

We will show below that the measure and the contour problem are easily solved, once a suitable representation of $stripes(F)$ has been computed for a frame F containing all rectangles. Before doing so we give a recursive procedure MC (for measure and contour) which, given a frame F, computes sets $P(F)$, $stripes(F)$, $L(F)$ and $R(F)$. We assume for simplicity that vertices of different rectangles have different x-and y-coordinates. The modifications required for the general case are described in the preceding section and are left to the reader.

Lines (15) to (18) need some explanation. Note that in line (11) we combine partitions P_1 and P_2 to form the combined partition $P = \{y_1, \ldots, y_m\}$, cf. Figure 108.

In this diagram we have $P = \{y_1, \ldots, y_8\}$, $P_1 = \{y_2, y_5, y_6, y_7\}$ and $P_2 = \{y_1, y_2, y_3, y_4, y_5, y_8\}$. In lines (15) and (17) we refine $stripes_1$ and $stripes_2$ according to the new partition. In our example, the stripe of F_2 between y_5 and y_8 is refined into three substripes: y_5 to y_6, y_6 to y_7, and y_7 to y_8. We obtain the lists $list$ for these substripes by copying the list for the stripes between y_5 and y_8. In the actual implementation, the copy operation will be a pointer operation.

```
(1) procedure MC(F, P, stripes, L, R)
(2) if        F = (f1, f2) contains exactly one vertical edge, say
              l = (x, y1, y2, s), where s ∈ {left, right}
(3) then P ← {y1, y2};
(4)       if s = left
(5)       then R ← {[y1, y2]}; L ← ∅; list1 ← (x, f2)
(6)       else L ← {[y1, y2]}; R ← ∅; list1 ← (f1, x)
(7)       fi
(8) else  choose a vertical line x = f which divides the objects in
          frame F in about equal parts and let F1 = (f1, f) and F2 = (f, f2);
(9)       MC(F1, P1, stripes1, L1, R1);
(10)      MC(F2, P2, stripes2, L2, R2);
(11)      P ← P1 ∪ P2;
(12)      LR ← R1 ∩ L2;
(13)      L ← L1 ∪ (L2 − LR);
(14)      R ← R2 ∪ (R1 − LR);
(15)      refine stripes1 according to P;
(16)      simplify stripes1 according to L2 − LR;
(17)      refine stripes2 according to P;
(18)      simplify stripes2 according to R1 − LR;
(19)      unite stripes1 and stripes2 to form stripes;
(20) fi
(21) end.
```

Program 20

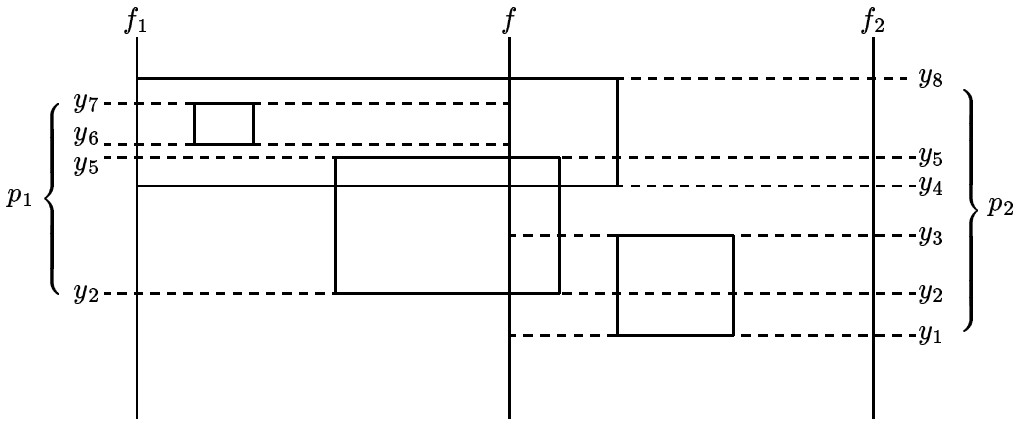


Figure 108.

In lines (16) and (18) we simplify sets $stripes_1$ and $stripes_2$. Consider an interval, say $[y_i, y_j]$ in $L_2 - LR$. It represents a rectangle which extends all over frame F_1. We can therefore replace $list(S_h(F_1))$ of $stripes_1$, $1 \leq h \leq j$, describing

stripe/strip

the stripe between y_h and y_{h+1}, by the trivial list (f_1, f) which indicates that the entire stripe is covered. Set *stripes*$_2$ is simplified in an analogous way based on the intervals in $R_1 - LR$. In our example, we have $L_2 = \{[y_2, y_5], [y_4, y_8]\}$ and $LR = \{[y_2, y_5]\}$. We simplify *stripes*$_1$ by changing the lists of stripes $S_4(F_1)$, $S_5(F_1)$, $S_6(F_1)$, $S_7(F_1)$ to (f_1, f).

Finally, in line (19) we form *stripes* from *stripes*$_1$ and *stripes*$_2$ by concatenating *list*$(S_i(F_1))$ and *list*$(S_i(F_2))$ to form *list*$(S_i(F))$, $1 \leq i < m$, and combining the right interval of *list*$(S_i(F_1))$ and the left interval of *list*$(S_i(F_2))$ if both extend to the dividing line $x = f$.

With regard to the definitions of line (15) to (19) it is easy to prove the correctness of procedure *MC*. We leave the details to the reader.

The running time still remains to be analyzed. We represent sets $P(F)$, $L(F)$, and $R(F)$ by ordered linked lists. $P(F)$ is ordered according to the y-coordinate and the sets $L(F)$ and $R(F)$ are ordered according to the bottom endpoint of the intervals. Then lines (11) to (14) take time $O(\bar{n})$, where $\bar{n}$ is the number of objects in frame F. For the representation of *stripes* we distinguish the measure and the contour problem. We treat the measure problem first because it is simpler.

For the measure problem a very simple data structure suffices. *stripes* is an ordered linear list of reals. If $P = \{y_1 < y_2 < \cdots < y_m\}$, then the i-th element of the list, $1 \leq i \leq m$, is the total length of the intervals in *list*$(S_i(F))$. No other information about *list*$(S_i(F))$ is maintained. It is now easy to see that lines (15) to (19) take time $O(\bar{n})$, where $\bar{n}$ is the number of objects in frame F.

Theorem 12. *The measure problem for a set of n iso-oriented rectangles can be solved in time $O(n \log n)$ and space $O(n)$.*

Proof: The discussion above implies that the running time $T(n)$ of procedure *MC* satisfies the recurrence

$$T(1) = O(1)$$
$$T(n) = 2 \cdot T(n/2) + O(n).$$

Thus $T(n) = O(n \log n)$. Let F be a frame containing all rectangles and apply procedure *MC* to frame F. The application yields $P(F)$ and *stripes*(F) from which one can clearly compute the area of the union of the rectangles in time $O(n)$. This proves the time bound.

The space bound can be seen as follows. Note first that the size of the representations of $P(F)$, $L(F)$, $R(F)$, and *stripes*(F) is $O(\bar{n})$, where $\bar{n}$ is the number of objects in frame F. Note next that at any point of time there are at most $\log n$ incarnations of procedure *MC*, and that the number of objects in the associated frames form a geometric progression. Combining both observations we obtain the space bound ∎

For solving the contour problem we have to make a greater effort. In this case we represent *stripes*(F) as a linked list of pointers. If $P(F) = \{y_1 < y_2 < \cdots < y_m\}$ then the i-th pointer points to the root of a search tree for the endpoints of the

intervals in $list(S_i(F))$, $1 \leq i \leq m$. Again, we can perform lines (15) to (19) in time $O(\bar{n})$, where $\bar{n}$ is the number of objects in frame F. In lines (15) and (17) we extend the list of pointers and perform the copy-operations by setting pointers appropriately. Thus trees may share subtrees. In lines (16) to (18) we redirect some pointers to trivial trees having only two leaves. The set of pointers to be redirected is easily found in time $O(\bar{n})$ by going through ordered lists $P(F)$ and $L_2 - LR$ ($R_1 - LR$) in parallel. Finally, in line (19) we concatenate appropriate lists by creating new root nodes and appropriately defining the two son-pointers.

Theorem 13. *The contour problem for a set of n iso-oriented rectangles can be solved in time and space $O(n \log n)$.*

Proof: The discussion above implies that the running time of procedure *MC* is $O(n \log n)$ since the recurrence $T(n) = 2 \cdot T(n/2) + O(n)$ can be used again.

We apply procedure *MC* as follows. We first sort all rectangle vertices by x- and y-coordinate and then replace the coordinates by integers $1, \ldots, 2n$ in a consistent way. This replacement does not change the topology of the contour but allows us to use bucket sort (cf. Section 2.2.1) in latter stages of the algorithm. Let F be a frame which encloses all rectangles, $F = (0, 2n + 1)$ will do. Then *MC* applied ot F yields $P(F)$ and $stripes(F)$ in time $O(n \log n)$. It remains to describe how to obtain the contour from $P(F)$ and $stripes(F)$.

The contour of a union of iso-oriented rectangles is a collection of contour-cycles. Each contour-cycle is a sequence of alternating horizontal and vertical contour-pieces. Each contour-piece is a fragment of an edge of one of the rectangles.

We show how to find the horizontal contour pieces from $P(F)$ and $stripes(F)$ in time $O(n \log n + p)$, where p is their number. The vertical contour-pieces can then be found by sorting the endpoints of horizontal contour-pieces and the vertical rectangle edges by the x-coordinate. Note that time $O(n+p)$ suffices for the sorting step since bucket sort can be used. Recall that the x-coordinates are the integers $1, \ldots, 2n$. Finally, the contour-cycles can be constructed from the contour-pieces by another application of bucket sort to their endpoints. Again, time $O(n+p)$ suffices.

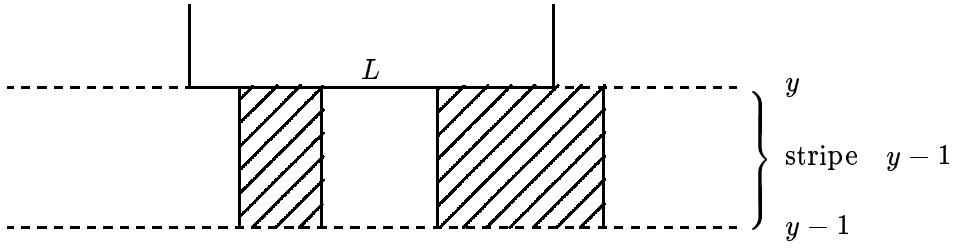


Figure 109.

The horizontal contour-pieces can be found as follows. Let $L = (x_1, x_2 y)$ be any horizontal rectangle edge; x_1, x_2 are integers. We show how to find the k contour-pieces which are fragments of L in time $O(n \log n + k)$. Assume that L is a bottom

side of a rectangle; the reverse case being symmetric. Consider stripe $S_{y-1}(F)$ and let $list = list(S_{y-1}(F))$. Recall that *list* is an ordered list of intervals and that *list* is realized as a search tree for the endpoints of the intervals. Also, note that the tree has depth $O(\log n)$ since the height of the tree grows by one in each level of the recursion. We can therefore find the first endpoint in *list* with x-coordinate $\geq x_1$ in time $O(\log n)$; a simple linear search will then identify the k contour-pieces which are fragments of L in time $O(k)$. This finishes the proof of the time bound. The space bound follows immediately from the time bound. ∎

8.5.3. Intersection Problems in Higher-Dimensional Space

In this section we will study intersection problems in higher-dimensional space. Let $S = \{R_1, \ldots, R_n\}$ be a set of iso-oriented objects in $\mathbb{R}^d$, i.e., each R_i is the cartesian product of d intervals, one for each coordinate,

$$R_i = \prod_{j=1}^{d} [l_{ij}, r_{ij}].$$

We allow intervals to degenerate to a single point. We study the following two problems.

1) Report all pairs R_i, R_j with $R_i \cap R_j \neq \emptyset$. In Sections 5.1.1 and 5.1.2 this problem was solved in two dimensions in time $O(n \log n + k)$. Here k is the number of intersecting pairs. We refer to this problem as the all pair intersection problem.
2) Given iso-oriented query object $Q = \prod_{j=1}^{d} [l_j, r_j]$ report all $R_i \in S$ with $Q \cap R_i \neq \emptyset$. This problem arises in two versions: In the static version, set S is fixed and in the dynamic version, we also allow for insertions and deletions. We refer to this problem as the (static or dynamic) searching problem.

This section is organized as follows. We will first show how to solve the static searching problem by means of static range and segment trees, then extend the solution to the dynamic case by using the dynamic version of the trees. The all pair intersecting problems is finally solved by giving a solution to the searching problem in a sweep algorithm. A solution to the searching problem can be based of the following simple observations. For

$$R_i = \prod_{j=1}^{d} [l_{ij}, r_{ij}]$$

we use

$$R'_i = \prod_{j=1}^{d-1} [l_{ij}, r_{ij}]$$

to denote the projection onto the first $d-1$ dimensions.

Lemma 14.

a) $R_i \cap Q \neq \emptyset$ *iff* $[l_{id}, r_{id}] \cap [l_d, r_d] \neq \emptyset$ *and* $r'_i \cap Q' \neq \emptyset$.

b) $[l_{id}, r_{id}] \cap [l_d, r_d] \neq \emptyset$ *iff either* $l_{id} \in [l_d, r_d]$ *or* $l_d \in [l_{id}, r_{id}]$.

Proof: a) The direction from left to right is obvious. For the other direction, we only have to observe that $x_d \in [l_{id}, r_{id}] \cap [l_d, r_d]$ and $x' \in R'_i \cap Q'$ implies $x = (x', x_d) \in R_i \cap Q$.

b) obvious. ∎

Lemma 14 suggests a recursive solution for the searching problem. For each fixed dimension we need to provide data structures which allow us to find all points contained in a query interval (range trees) and to find all intervals containing a query point (segment trees). The details are as follows.

Suppose first that $d = 1$. We have two trees. The first tree is a priority search tree (cf. Section 5.1.2) and stores the n pairs (= points in $\mathbb{R}^2$) (l_{id}, r_{id}), $1 \leq i \leq n$. It allows us to find all intervals containing a query point l_d, i.e., $l_{id} \leq l_d \leq r_{id}$, in time $O(\log n + k)$, where k is the number of points reported. Furthermore, the tree uses space $O(n)$ and can be constructed in time $O(n \log n)$. The second tree is a balanced search tree for the left endpoints l_{id}, $1 \leq i \leq n$, of the intervals. It uses space $O(n)$, can be constructed in time $O(n \log n)$ and allows us to find the k endpoints contained in query interval $[l_d, r_d]$ in time $O(\log n + k)$. We conclude that for $d = 1$ the static and dynamic searching problem has a solution with query time $O(\log n + k)$, preprocessing time $O(n \log n)$ and space requirement $O(n)$.

For $d > 1$ we have to work slightly harder. Again, we have two trees. The first tree is a segment tree for the intervals $[r_{id}, l_{id}]$, $1 \leq i \leq n$. For each node v of the segment tree we organize its node list $NL(v)$ as a data structure, which solves $(d-1)$-dimensional searching problem. Given query object

$$Q = \prod_{j=1}^{d} [l_j, r_j]$$

this augmented segment tree allows us to find all R_i with $Q' \cap R' \neq \emptyset$ and $l_d \in [l_{id}, r_{id}]$ as follows. We search for l_d in the segment tree. Then the node lists of the nodes of the path of search contain all objects R_i with $l_d \in [l_{id}, r_{id}]$. We use the secondary structures associated with those nodes, i.e., the data structures for the $(d-1)$-dimensional searching problem, to find all R'_i with $Q' \cap R'_i \neq \emptyset$.

Assume inductively that the search time in the $(d-1)$-dimensional structure is $T(n, d-1) = O((\log n)^{d-1} + k)$, where k is the size of the answer. This is true for $d = 2$. Then the search time in the d-dimensional structure is $T(n, d) = O(\log n \cdot (\log n)^{d-1} + k) = O((\log n)^d + k)$ since the $(d-1)$-dimensional solution has to be applied at $\log n$ different nodes. Similarly, if the preprocessing time for the $(d-1)$-dimensional solution is $P(n, d-1) = O(n(\log n)^{d-1})$, then the preprocessing time for the d-dimensional solution is $O(n(\log n)^d)$. This can be seen as follows.

Let $n(v)$, where v is a node of the segment tree, be the cardinality of $NL(v)$. Then $\sum\{n(v);\ v \text{ is a node}\} = O(n \log n)$ since every interval is split into at most $O(\log n)$ pieces and time $O(n(v) \cdot (\log n)^{d-1})$ is spent for constructing the secondary structure for node v. Thus time $O(n(\log n)^d)$ is spent for constructing all secondary structures. Finally, an identical argument shows that the space requirement of the d-dimensional solution is $O(n(\log n)^{d-1})$.

The second tree is an augmented tree for the left endpoints l_{id}, $1 \leq i \leq n$, of the intervals in the d-th dimension Recall (cf. Section 7.2.2) that in a range tree each point is stored in the node lists of all nodes along a path through the tree, i.e., each point is stored in $O(\log n)$ node lists. Let v be a node of the range tree. We organize the node list of node v as a $(d-1)$-dimensional search structure for those objects which have their endpoint l_{id} stored in the node list. An argument similar to the one above shows that the resulting data structure has query time $O((\log n)^d + k)$, preprocessing time $O(n(\log n)^d)$ and space requirement $O(n(\log n)^{d-1})$. We summarize in:

Theorem 14. *The static searching problem for iso-oriented objects in d-dimensional space can be solved in query time $O((\log n)^d + k)$, preprocessing time $O(n(\log n)^d)$, and space $O(n(\log n)^{d-1})$. Here k is the number of objects intersecting the query object.*

Proof: By the discussion above ∎

The extension to the dynamic version is not very difficult. We only have to replace static range and segment trees by their dynamic counterpart. We have:

Theorem 15. *The dynamic searching problem in d-dimensional space can be solved in query time $O((\log n)^d + k)$, insertion and deletion time $O((\log n)^d)$ and space $O(n(\log n)^{d-1})$. The time bounds for insertions and deletions are amortized.*

Proof: The bound on query time and space is derived as in Theorem 14. The bounds on insertion and deletion time are derived in exactly the same way as the respective bounds for range trees (with slack parameter 1) in Lemma 4 of Section 7.2.2. ∎

Now we turn to the all pair intersection problem. We use the sweep paradigm, i.e., we sweep a $(d-1)$-dimensional hyperplane through $\mathbb{R}^d$. The sweep direction is the d-th coordinate axis, i.e., the sweep hyperplane is perpendicular to the d-th coordinate axis. We store $(d-1)$-dimensional projections R_i' of all objects R_i which intersect the sweep hyperplane in a data structure which solves the $(d-1)$-dimensional searching problem. Object R_i is added to the data structure at transition point l_{id} and deleted at transition point r_{id}. Thus $2n$ insertions and deletions with a total cost of $O(n(\log n)^{d-1})$ are required. Also, whenever an object R_i is added to the data structure we query the data structure and report all intersected objects. The total cost of all queries is $O(n(\log n)^{d-1})$. We summarize in

Theorem 16. *The all pair intersection problem for n iso-oriented objects in $\mathbb{R}^d$, $d \geq 2$, can be solved in time $O(n(\log n)^{d-1} + k)$ and space $O(n(\log n)^{d-2})$.*

Proof: Obvious from Theorem 15 and the discussion above. ∎

8.6. Geometric Transforms

Transformations (reductions) play an important role in computational geometry and in algorithm design in general. They allow us to classify problems according to their level of complexity thus giving us a more structured view of the field and also reducing the number of distinct problems by grouping them into classes. In more concrete terms, a transformation of problem A into problem B allows us to use an algorithm for B to solve A and allows us to transfer a lower bound on the complexity of A to a lower bound on the complexity of B. In this book we have extensively used transformations for both purposes. Let us mention just a few situations. The major part of Chapter 6 on NP-completeness is centered around the notion of a reduction, in Chapter 5 we related the complexity of general path problems and matrix multiplication over semi-rings and at various places (e.g., in Section 2 of convex hulls) we derived $\Omega(n \log n)$ lower bounds by reducing the sorting problem to a problem at hand.

Among the many geometric transforms we discuss only two: duality and inversion. The duality transform in $\mathbb{R}^d$ maps points into hyperplanes and hyperplanes into points. It can thus be used to transform problems about points into problems about hyperplanes and conversely to transform problems about hyperplanes into problems about points. We will have the opportunity to use both directions successfully in the sequel.

inversion

The duality transform in $\mathbb{R}^3$ ($\mathbb{R}^2$) transforms spheres (circles) into planes (lines) and vice versa. It can be successfully used to transform problems about circles and spheres into problems about lines and planes. One possible application of this transformation is the construction of Voronoi diagrams; note that vertices of the diagrams are centers of circles passing through at least three points of the underlying set. Thus the problem of constructing Voronoi diagrams can be viewed as a problem about circles.

8.6.1. Duality

The duality transform in $\mathbb{R}^d$ relates hyperplanes and points and more generally k-dimensional subspaces with $(d-k)$-dimensional subspaces.

Let h be a non-vertical hyperplane in $\mathbb{R}^d$, i.e., h intersects the d-th coordinate axis in a unique and finite point. Let the points on h with cartesian coordinates $(x_1, \ldots, x_d)$ satisfy the equation

$$x_d = p_1 x_1 + p_2 x_2 + \cdots + p_{d-1} x_{d-1} + p_d.$$

Then the dual $D(h)$ of hyperplane h is the point $p = (p_1, p_2, \ldots, p_d)$ in $\mathbb{R}^d$. Conversely, $p = (p_1, \ldots, p_d)$ is a point in $\mathbb{R}^d$ then $h = D(p)$ is the hyperplane defined by the equation

$$x_d = -p_1 x_1 - \cdots - p_{d-1} x_{d-1} + p_d.$$

An important property of the duality transform is the preservation of incidence and more generally of vertical distances, i.e., distances in the direction of the d-th coordinate axis. Let h be a hyperplane given by the equation $x_d = q_1 x_1 + \cdots + q_{d-1} x_{d-1} + q_d$ and let $p = (p_1, \ldots, p_d)$ be a point. Then the vertical distance $vd(h, p)$ is defined by

$$vd(h, p) = p_d - (q_1 p_1 + \cdots + q_{d-1} p_{d-1} + q_d).$$

We will also say that p lies above (on, below) h if $vd(h, p) > (=, <)\ 0$. We have

Lemma 1. *Let h be a hyperplane and p be a point. Then $vd(h, p) = -vd(D(p), D(h))$. In particular, p lies on h iff $D(h)$ lies on $D(p)$.*

Proof: Let $p = (p_1, \ldots, p_d)$ and let h be given by the equation

$$x_d = q_1 x_1 + \cdots + q_{d-1} x_{d-1} + q_d.$$

Then hyperplane $D(p)$ is defined by the equation

$$x_d = -p_1 x_1 - \cdots - p_{d-1} x_{d-1} + p_d.$$

Hence

$$\begin{aligned} vd(D(p), D(h)) &= q_d - (-p_1 q_1 - \cdots - p_{d-1} q_{d-1} + p_d) \\ &= -(p_d - (p_1 q_1 + \cdots + p_{d-1} q_{d-1} + q_d)) \\ &= -vd(h, p). \end{aligned}$$

In particular, $vd(h, p) = 0$ iff $vd(D(p), D(h)) = 0$. ∎

We want to mention one more fact concerning the duality transform. For simplicity we restrict the discussion to $d = 3$. For p_1, p_2 distinct points in $\mathbb{R}^d$ let $L(p_1, p_2)$ be the line through p_1 and p_2. We have

Lemma 2. *Let p_1, p_2 be points and let h_1, h_2 be planes in $\mathbb{R}^3$ with $L(p_1, p_2) = h_1 \cap h_2$. Then $D(p_1) \cap D(p_2) = L(D(h_1), D(h_2))$.*

Proof: Since $p_1 \neq p_2$ and $h_1 \neq h_2$ we have $D(p_1) \neq D(p_2)$ and $D(h_1) \neq D(h_2)$. Thus $D(p_1) \cap D(p_2)$ and $L(D(h_1), D(h_2))$ are both lines. Furthermore, from $p_i \in h_j$ we conclude $D(h_j) \in D(p_i)$, $1 \leq i, j \leq 2$, by Lemma 1, and hence the two lines agree. ∎

We are now ready for our first application of duality: the intersection of halfspaces. We will use duality to transform the halfspaces or rather the defining hyperplanes into points. the intersection problem is then transformed into two convex hull problems.

Let h_i, $1 \leq i \leq n$, be a hyperplane. We use h_i^+ (h_i^-) to denote the set of points which are on or above (on or below) h_i. The sets h_i^+ and h_i^- are halfspaces. Let m be an integer with $1 \leq m \leq n$. Our goal is to compute

$$S = \bigcap_{i=1}^{m} h_i^+ \cap \bigcap_{i=m+1}^{n} h_i^-.$$

We previously discussed this problem for $\mathbb{R}^2$ in Exercise 6. In that exercise an $O(n \log n)$ divide-and-conquer algorithm is derived which uses the linear time intersection algorithm (cf. Section 1) for convex polygons in the merge step. The analogous algorithm in $\mathbb{R}^3$ based on the $O(n \log n)$ algorithm for intersecting convex polyhedra (cf. Section 4.3) has running time $O(n(\log n)^2)$. Duality will give us an $O(n \log n)$ algorithm in $\mathbb{R}^3$. From now on we restrict the discussion to $\mathbb{R}^3$.

Let $S^+ = \bigcap_{i=1}^{m} h_i^+$ and $S^- = \bigcap_{i=m+1}^{n} h_i^-$. In view of the $O(n \log n)$ algorithm for computing $S = S^+ \cap S^-$ it suffices to show how to compute S^+ in time $O(m \log m)$. Our algorithm is based on the following observation. The set S^+ is a convex polyhedron whose faces lie on some of the planes h_i, $1 \leq i \leq m$. We call plane h_i redundant (non-redundant) if there is no face (is a face) of S^+ which is contained in h_i. We will use duality to compute the non-redundant planes. Knowledge of the non-redundant planes will then allow us to compute S^+ fairly easily.

So let us assume that plane h_a is redundant. Then $S^+ \cap h_a$ is either empty, a vertex of S^+, or an edge of S^+. In either case, let v be a vertex of S^+ which is closest to h_a. Then there are planes h_j, h_k, h_l such that $v = h_j \cap h_k \cap h_l$ and $h_j^+ \cap h_k^+ \cap h_l^+ = h_j^+ \cap h_k^+ \cap h_l^+ \cap h_a^+$, i.e., planes h_j, h_k, h_l witness the redundancy of h_a. Consider the duals $p_i = D(h_i)$, $1 \leq i \leq m$, of the planes. Then v lies on or above h_a and hence $D(h_a)$ lies on or below $D(v)$. Next observe that $D(v)$ is a plane which contains points $D(h_j), D(h_k), D(h_l)$ and hence is determined by them. This observation suggests the following lemma.

Lemma 3. *h_a is redundant iff $D(h_a)$ is not a vertex of the upper convex hull of point set $\{D(h_i);\ 1 \leq i \leq m\}$. The upper convex hull of a point set consists of those faces (and incident edges and vertices) of the hull which have all points in the set on or below the supporting plane.*

Proof: "⇒": We have argued above that if h_a is redundant then there are planes h_j, h_k, h_l such that $D(h_a)$ lies on or below the plane P determined by points $D(h_j), D(h_k), D(h_l)$. It remains to be shown that the projection of p_a into the xy-plane lies inside the triangle determined by the projections of points p_j, p_k and p_l. Let h_0 be a plane which touches S^+ in vertex v and is parallel to plane h_a. Then $p_0 = D(h_0)$ lies exactly above point p_a. It therefore suffices to show that p_0 lies inside the triangle determined by points p_j, p_k and p_l.

The normal vector $\vec{q}_0 = (q_1^0, \ldots, q_{d-1}^0, -1)$ of plane h_0 lies in the cone defined by the normal vectors $\vec{q}_j = (q_1^j, \ldots, q_{d-1}^j, -1)$, $\vec{q}_k, \vec{q}_l$ of planes h_j, h_k, h_l and hence $\vec{q} = \alpha\vec{q}_j + \beta\vec{q}_k + \gamma\vec{q}_l$ with $\alpha \geq 0$, $\beta \geq 0$, $\gamma \geq 0$ and $\alpha + \beta + \gamma = 1$. Also, plane q_i, $i \in \{0, j, k, l\}$, is given by equation $x_d = q_1^i x_1 + \cdots q_{d-1}^i x_{d-1} + c$ for some constant c. Hence $p_i = (q_1^i, \ldots, q_{d-1}^i, c)$ and therefore $p_0 = \alpha p_i + \beta p_j + \gamma p_l$. This shows that p_0 lies inside the triangle defined by points p_i, p_j and p_l.

$p_i = D(h_i)$?
i, j, l?

"$\Leftarrow$": Suppose that $D(h_a)$ is not a vertex of the upper convex hull. Then there are vertices p_i, p_j, p_k of the upper convex hull such that $p_a = D(h_a)$ lies on or below the triangle with vertices p_i, p_j, p_k. By an argument similar to the one used in the only if part one can show that planes h_i, h_j and h_k witness the redundancy of h_a. ∎

Lemma 3leads to the following algorithm for computing S^+.

1) Compute $p_i = D(h_i)$, $1 \leq i \leq m$, and determine the upper convex hull of point set $\{p_i;\ 1 \leq i \leq m\}$.
2) Use duality to obtain S^+ from the upper convex hull.

The computation of point set $\{p_i;\ 1 \leq i \leq m\}$ takes time $O(n)$. The (upper) convex hull of the point set can be determined in time $O(m \log m)$ by Exercise 13.

We have to describe step 2) in more detail. Let f be a face of the upper convex hull and let h be a plane supporting f. Then $D^{-1}(h)$ is a vertex of S^+ by the proof of Lemma 3. Also, if e is an edge of the upper convex hull separating faces f_1 and f_2 (supported by planes h_1, h_2) then $L(D^{-1}(h_1), D^{-1}(h_2))$ is a line which supports the edge of connecting vertices $D^{-1}(h_1)$ and $D^{-1}(h_2)$ of S^+. (This is a consequence of Lemma 2). We conclude that the structure of the upper convex hull gives us complete knowledge about S^+ and hence S^+ can be computed in time $O(m)$ from the upper convex hull.

Theorem 1. *The intersection of a set of n halfspaces in $\mathbb{R}^3$ can be computed in time $O(n \log n)$.*

Proof: By the preceding discussion. ∎

Our second example concerns problems about point sets in $\mathbb{R}^2$. We will use duality to transform them into problems about sets of lines which we will then be able to solve. Let $S = \{p_i;\ 1 \leq i \leq n\}$ be a set of points in $\mathbb{R}^2$. We consider the following two problems:

a) Decide whether any three of the points are collinear.

b) Compute the smallest area triangle which has vertices in S.

Note that the first problem is a special case of the second; it is tantamount to deciding whether the minimum area triangle has area 0. We derive an $O(n^2)$ algorithm for the first problem based on duality and then extend the algorithm to solve the second problem as well.

Let $h_i = D(p_i)$ be the dual of point p_i, $1 \leq i \leq n$. Since duality preserves incidences we conclude that there are three collinear points in S iff there are three

lines in set $H = \{h_i;\ 1 \leq i \leq n\}$ which have a common point. We decide the latter question by explicitly constructing the planar subdivision which is induced by the lines in H.

We construct the planar subdivision interactively. Let SD_i be the planar subdivision which is induced by lines $H_i := \{h_1, \ldots, h_i\}$. Then SD_1 has only two faces, one edge and no vertex and hence can be clearly constructed in time $O(1)$. We show how to obtain SD_i from SD_{i-1} in time $O(i)$.

Let us recall the representation of planar subdivisions. For each vertex we have the set of incident edges in clockwise order and for each face we have its boundary edges in clockwise order and for each edge we have pointers to its endpoints and the two adjacent faces. In addition to that we assume that for each line h_j we have the list of vertices and edges of the planar subdivision which lie on h_j. Finally, we assume that the lines are sorted by slope.

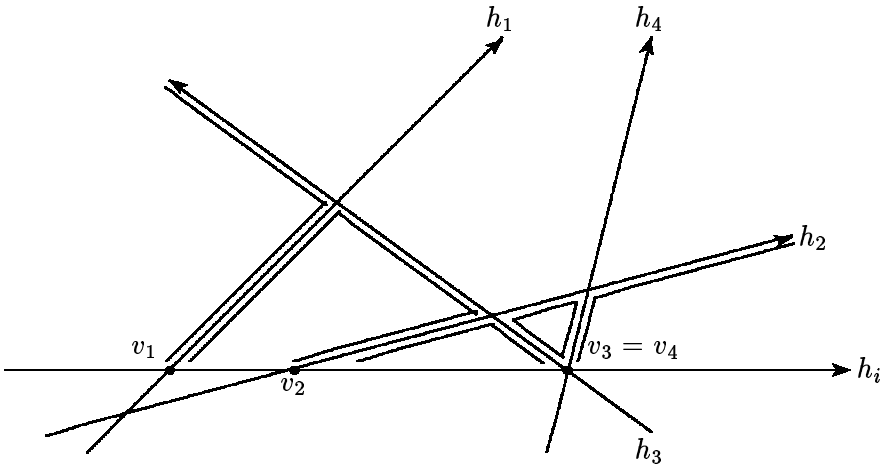


Figure 110.

Suppose now that SD_{i-1} is available and we want to construct SD_i. In Figure 110 h_i is shown as a horizontal line. If all lines in H_{i-1} are parallel to h_i then we can clearly construct SD_i in time $O(i)$ from SD_{i-1}. So let us assume that h_j, $j < i$, and h_i are not parallel. We compute $v_j := h_i \cap h_j$ and locate v_j in the planar subdivision by a linear search through the fragments of line h_j. This takes clearly time $O(i)$.

We locate the other points $v_k := h_i \cap h_k$, $1 \leq k \leq i$, as follows. Line H_i enters two faces of SD_{i-1}, say f and f', from v_j. We find the points, where h_i leaves these faces by a linear search along the boundary of these faces. We make sure that we always follow the part of the boundary which is *above* h_i. In this way we locate two more points of the intersection. From these points we continue the construction in an analogous way. Note that faces above h_i which only touch a single point are traversed completely. In Figure 110 we have indicated the edges visited in this search by a double line. Let m_i be the number of edges visited in this search. Then SD_i can clearly be constructed in time $O(m_i)$ from SD_{i-1}. We have

Lemma 4. $m_i \leq 5i$.

Proof: Let F be the set of faces of SD_i (not SD_{i-1}!) which lie above line h_i and have a fragment of h_i on their boundary. Then m_i is the total number of edges bounding the faces in F. Let E be the multi-set of edges which bound a face in F. We partition E into three disjoint sets. In order to simplify the notation and language we assume w.l.o.g. that h_i is horizontal and that all lines h_j, $j < i$, are oriented in the upward direction. It then makes sense to talk about the interior angle between lines h_j and h_i. It is always between 0 and π.

Let E_1 be the multi-set of those edges in E which have at least one endpoint on line h_i and let E_2 be the remaining set of edges. Since every face in F contributes at most three edges to E_1 (one having both endpoints on h_i and two having one endpoint on h_i) and since $|F| = i$ we conclude $|E_1| \leq 3i$. The edges in E_2 remain to be counted. For this purpose, we partition the edges in E_2 into two groups E_l and E_r. Let $f \in F$ be a face, let $e \in E_2$ be an edge on the boundary of f and let g be that edge on the boundary of f which is supported by h_j, $j < i$. We put e into group E_l if h_j intersects h_i to the left of g and we put e into group E_r otherwise. In Figure 111, the edges E_1 (E_l, E_r) are indicated as solid (dotted, dashed) double lines.

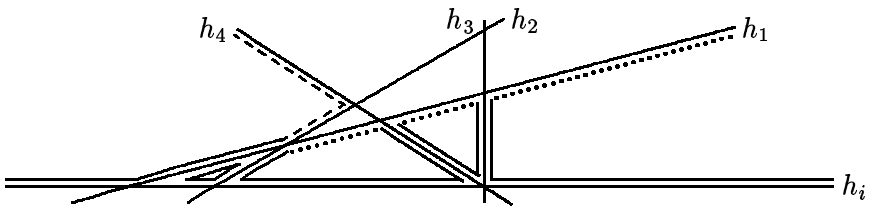


Figure 111.

We show $|E_r| \leq i$. Then symmetry implies $|E_l| \leq i$ and hence $m_i = |E_1| + |E_l| + |E_r| \leq 5i$.

Lemma 5. $|E_r| \leq i$.

Proof: We use induction on i. For $i \leq 2$ we have $|E_r| = 0$ and hence the claim is obviously true. So let us assume that $i \geq 3$. Choose j, $j < i$, such that $\angle(h_i, h_j)$ is minimal. Remove line h_j from the planar subdivision and let E'_r be defined with respect to the resulting subdivision as E_r is defined with respect to SD_i. Then $|E'_r| \leq i-1$ by the induction hypothesis. It therefore suffices to show $|E_r| \leq |E'_r|+1$. This can be seen as follows.

Note first that no edge which is supported by h_j can belong to E_r. This follows immediately from the fact that $\angle(h_i, h_j)$ is] minimal.

Next observe that the removal of h_j merges some faces of SD_i. More precisely, there are two faces, say f_1, f_2 in F which are merged to a single face and every other face, say f, of F is either left unchanged or merged with a face which is

not in F. In the latter case f (or the result of merging f with a face not in F) contributes at least as many edges to E'_r as it contributed to E_r. In the former case, the face obtained by merging f_1 and f_2 contributes at most one edge less to E'_r than f_1 and f_2 contributed together to E_r, since the removal of h_j might combine two edges in E_r or combine an edge in E_r with an edge in E_1. Thus $|E_r| \leq |E'_r| + 1$ and the proofs of Lemmata 4 and 5 are completed. ∎∎

We infer from Lemma 4 and the discussion preceding it that SD_i can be computed from SD_{i-1} in time $O(i)$. Hence SD_n can be computed in time $O(n^2)$. Having computed SD_n it is trivial to check in time $O(n^2)$ whether any three lines in H run through a common point. We summarize in

Theorem 2.

a) *The planar subdivision induced by a set of n lines in $\mathbb{R}^2$ can be computed in time $O(n^2)$.*

b) *Given n points in $\mathbb{R}^2$ one can decide in time $O(n^2)$ whether any three of them are collinear.*

Proof: By the discussion above. ∎

In the remainder of this section we extend Theorem 2 to a solution of the minimum area triangle problem. The extension is based on the following simple observation. Let $p_1, \ldots, p_n$ be n points in the plane. For $1 \leq i < j \leq n$ let $near(i,j) = k$ if point p_k has minimal distance from line $L(p_i, p_j)$. Since the area of triangle p_i, p_j, p_k is $dist(p_i, p_j) \cdot dist(p_k, L(p_i, p_j))/2$ we conclude that triangle $p_i, p_j, p_{near(p_i,p_j)}$ has minimum area among all triangles with vertices p_i and p_j. This shows that it suffices to compute function *near* in order to solve the minimum area triangle problem.

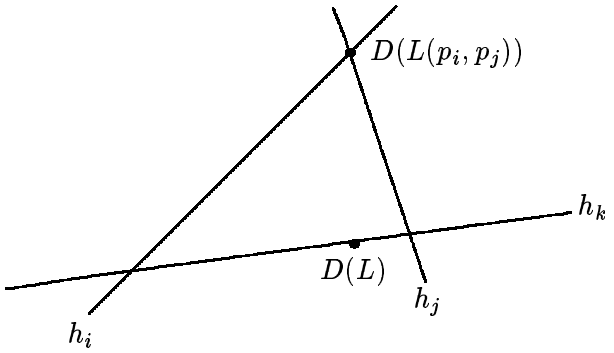


Figure 112.

Let us consider the dual problem. Let $h_i = D(p_i)$ be the line dual to point p_i, $1 \leq i \leq n$, and let i and j be arbitrary. Then point $h_i \cap h_j$ is the dual of line $L(p_i, p_j)$ by Lemma 2; cf. Figure 112. Consider the line L which passes through p_k, $k = near(i,j)$, and is parallel to $L(p_i, p_j)$. The dual $D(L)$ of line L is a point having

the same x-coordinate as the dual $D(L(p_i, p_j))$ of line $L(p_i, p_j)$. This follows from the fact that the x-coordinate of the dual is given by the slope of the line. Since duality preserves incidence we conclude that line $D(p_k)$ passes through $D(L)$. We also conclude that no other line h_l, $l \neq k$, can intersect the vertical line segment connecting $D(L)$ and $D(L(p_i, p_j))$. Hence $D(L)$ lies on the same face as $D(L(p_i, p_j))$. This observation immediately suggests the following algorithm:

Let $h_i = D(p_i)$, $1 \leq i \leq n$. Compute the planar subdivision induced by the h_i's and check whether any three lines have a common point. If not, draw a vertical line through every vertex v of the planar subdivision and find the closest intersection of this vertical line with an edge of the subdivision. This edge (supported by line h_k, say) defines $near(i, j)$, where $v = h_i \cap h_j$.

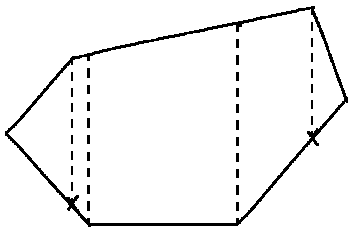


Figure 113.

We finally describe how the vertical lines are handled. We compute the intersection face by face. Since faces are convex polygons we can handle all vertical lines running inside the polygon in time proportional to the number of vertices of the face by a "merge" of the lower and upper part of the face. This shows that function *near* can be computed in time $O(n^2)$ from the planar subdivision and hence proves

Theorem 3. *The minimum area triangle problem in* $\mathbb{R}^2$ *can be solved in quadratic time.*

8.6.2. Inversion

We will now briefly discuss a second transformation which establishes a correspondence between lines (planes) and circles (spheres) in $\mathbb{R}^2$ ($\mathbb{R}^3$). It frequently allows us to transform problems about circles into seemingly simpler problems about lines.

Inversion in $\mathbb{R}^2$ ($\mathbb{R}^3$) is most easily described with respect to a polar (spherical) coordinate system. Let c be the origin of the coordinate system. Then inversion with center c maps a point p with polar coordinates (R, Φ) (spherical coordinates (R, Φ, Psi)) into point $(1/R, \Phi)$ $((1/R, \Phi, \Psi))$, i.e., it inverts the distance from the origin. An important property of inversion is expressed in

Lemma 6.

a) *For all p: $I(I(p)) = p$*

b) *Let C be a circle (sphere) which passes through the center c of the inversion. Then $I(C)$ is a line (plane) which does not pass through c.*

b) *Let P be a line (plane) which does not pass through c. Then $I(P)$ is a circle (sphere) which passes through c.*

Proof: a) is obvious.

b)and c): We only give the proof for $\mathbb{R}^2$ and leave the three-dimensional case to the reader. We first express inversion in cartesian coordinates.

Lemma 7. *Let (x, y) be the cartesian coordinates of point p with respect to origin c. Then $I(p)$ has cartesian coordinates $(x/(x^2 + y^2), y/(x^2 + y^2))$.*

Proof: Let (R, Φ) be the polar coordinates of point p. Then $R^2 = x^2 + y^2$ and $\tan \Phi = y/x$. Let $\bar{x} = x/(x^2 + y^2)$ and $\bar{y} = y/(x^2 + y^2)$. Then $\tan \Phi = \bar{y}/\bar{x}$ and $\bar{x}^2 + \bar{y}^2 = 1/R^2$. Hence $I(p)$ has cartesian coordinates $(\bar{x}, \bar{y})$. ∎

We are now in a position to prove part c). Part b) is proven similarly and is left to the reader. Because of symmetry it suffices to prove part c) for a horizontal line.

Let h be a horizontal line given by the equation $y = a$. Let p with cartesian coordinates (x, a) be an arbitrary point on line h. Then $I(p)$ has cartesian coordinates $(x/d, a/d)$, where $d = x^2 + y^2$. It is now easy to check that $I(p)$ lies on the circle with center $(0, 1/2a)$ and radius $1/2|a|$. Also, all points on this circle are images of points on line h. ∎

Our first application of the inversion transform directly uses Lemma 6. Let C_i, $1 \leq i \leq n$, be a circle in $\mathbb{R}^2$ and let $s_i \in \{+, -\}$, $1 \leq i \leq n$. We use $C_i^{s_i}$ to denote the interior (exterior) of circle C_i provided that $s_i = +$ ($s_i = -$).

Lemma 8. *If $C_1, \ldots, C_n$ are circles in $\mathbb{R}^2$ which all pass through point c and $s_i \in \{+, -\}$, $1 \leq i \leq n$, then*

$$\bigcap_{i=1}^{n} C_i^{s_i}$$

can be computed in time $O(n \log n)$.

Proof: Let I be the inversion with center c. Let $h_i = I(C_i)$, $1 \leq i \leq n$. Then h_i is a line which does not pass through c and $C_i^{s_i}$ corresponds to one of the halfspaces defined by h_i. We denote that halfsapce by $h_i^{s_i}$. Let $P = \bigcap_i h_i^{s_i}$ be the convex polygon defined by the intersection of the halfspaces $h_i^{s_i}$, $1 \leq i \leq n$. We can compute P in time $O(n \log n)$ by Theorem 1. Also, $I(P) = \bigcap_i C_i^{s_i}$. Furthermore,it is easy to compute $I(P)$ since vertices of P are transformed into points and edges are transformed into "circular edges". Thus $I(P)$ can be computed in time $O(n)$ and Lemma 8 is proven. ∎

Lemma 8 hinges on the artificial assumption that all circles have a common point c. This assumption can be replaced if we invest one more idea: embedding into higher dimension.

Theorem 4. *Let C_i be circles in $\mathbb{R}^2$ and let $s_i \in \{+,-\}$, $1 \le i \le n$. Then $\bigcap_i C_i^{s_i}$ can be computed in time $O(n \log n)$.*

Proof: Identify $\mathbb{R}^2$ with the xy-plane of $\mathbb{R}^3$. Let c be a point outside the xy-plane, say $c = (0,0,1)$ and let S_i be a sphere which passes through c and intersects the xy-plane in circle C_i, $1 \le i \le n$. Let $h_i = I(S_i)$, $1 \le i \le n$, be the plane obtained by inversion of S_i with respect to center c. Then $S_i^{s_i}$ corresponds to one of the half-spaces defined by plane h_i. We use $h_i^{s_i}$ to denote that half-space. Then convex polyhedron $P = \bigcap_i h_i^{s_i}$ can be computed in time $O(n \log n)$ by Theorem 1. Also, $I(P) = \bigcap_i S_i^{s_i}$ and hence the intersection of $I(P)$ with xy-plane is the desired solution. We can compute $I(P) \cap (xy\text{-plane})$ in time $O(n)$ by transforming face by face of $I(P)$. The details are left to the reader. ∎

Theorem 4 deals with circles in a very direct way. In Voronoi diagrams circles come up in a more subtle way. Let $S = \{p_1, \ldots, p_n\}$ be a set of n points in the plane and let VD be the Voronoi diagram of S; cf. Section 3.1. Let v be any vertex of the Voronoi diagram. Then v is the center of a circle $C(v)$ which passes through at least three points of S and has no point of S in its interior. Conversely, the center of any such circle is a vertex of the diagram. As in Theorem 4, we identify $\mathbb{R}^2$ with the xy-plane of $\mathbb{R}^3$, choose a point c outside the xy-plane and use $S(v)$ to denote a sphere which passes through c and intersects the xy-plane in circle $C(v)$. Consider the inversion I with center c. Let $h(v) = I(S(v))$. Then $h(v)$ is a plane such that at least three points in $I(S)$ lie on $h(v)$ and all other points in $I(S)$ belong to the same half-space with respect to $h(v)$. In other words, plane $h(v)$ supports a face of the convex hull of point set S.

Construct $I(S)$ and compute the convex hull of $I(S)$. This takes time $O(n \log n)$ by Exercise 13. Let f be an arbitrary face of $I(S)$, let E be the supporting plane, and let $Sp = I(E)$ be he sphere obtained by inversion of E. Then all points o S lie either inside or on Sp or all of them lie outside or on Sp. The two cases are easily distinguished by testing one point of S (whose image under I does not lie on face f) with respect to Sp. In the latter case the center of the circle obtained by intersecting Sp with the xy-plane is a vertex of the Voronoi diagram (in the former case it is a vertex of the farthest point Voronoi diagram). In this way we compute all vertices fo the Voronoi diagram in time $O(n)$, each vertex of the diagram corresponding to a face of the convex hull of $I(S)$. Now the edges are quickly computed. We connect vertices v and w by a (straight-line) edge if the corresponding faces of the convex hull share an edge. This proves

Theorem 5. *The Voronoi diagram of a point set $S \subseteq \mathbb{R}^2$, $|S| = n$, can be computed in time $O(n \log n)$.*

8.7. Exercises

1) A sequence $P_0, P_1, \ldots, P_k$ of polygons is a balanced **outer representation** of convex polygon P if P_0 has at most 4 vertices, $P_k = P$ and P_{i-1} can be obtained from P_i by dropping all other boundary edges and extending the remaining ones. In the example, $P_1 = P$ is shown solid and P_0 is shown in dashed lines. Prove the results of Section 1 using balanced outer representations.

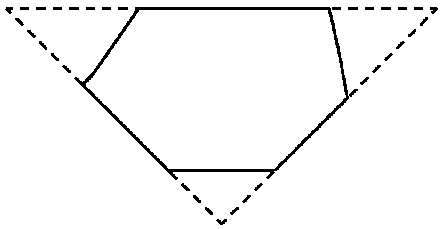


Figure 114.

2) Let P be a convex polyhedron in $\mathbb{R}^3$. An **inner polyhedral representation** of P is an ascending chain $P_0, P_1, \ldots, P_k$ such that P_0 has at most $O(1)$, say 100, faces, $P = P_k$, and P_i can be obtained from P_{i+1} as follows. Let V be a set of independent vertices of degree at most 10 of P_{i+1}. Then P_i is the convex hull of $V(P_{i+1}) - V$. Let n be the number of vertices of P.

a) Show that every convex polyhedron P with n vertices has an inner polyhedral representation $P_0, \ldots, P_k$ with $k = O(\log n)$. [Hint: Lemma 8 of Section 3.2.1 implies that V can be chosen such that $|V| \geq a \cdot |V(P_{ii+1})|$ for some constant $a > 0$.] Show that the representation can be computed in time $O(n)$.

b) Given a balanced inner polyhedral representation of convex polyhedron P and line L show that one can compute $P \cap L$ in time $O(\log n)$.

c) Given a balanced inner polyhedral representation of convex polyhedron P and plane E show that one can decide whether P and E intersect in time $O(\log n)$.

d) Given balanced representations of convex polyhedra P and Q show that one can decide in time $O(n)$ whether $P \cap Q = \emptyset$.

e) Show how to compute $P \cap Q$ for convex polyhedra P and Q in time $O(n \log n)$.

3) Given the balanced representation of convex n-gons P and Q show that one can compute $dist(P, Q) = \min\{dist(x, y);\ x \in P, y \in Q\}$ in time $O(\log n)$.

4) Given convex n-gons P and Q show that one can compute $maxdist(P, Q) = \max\{dist(x, y);\ x \in P, y \in Q\}$ in time $O(n)$.

5) Given the balanced representation of convex n-gon P and point p show that one
$\{p\}$ can compute the balanced representation of $CH(P \cup p)$ in time $O(\log n)$.

6) Show that one can compute the intersection of n half-spaces in time $O(n \log n)$. [Hint: Use divide and conquer and Theorem 5 of Section 1.]

$\forall y \in P$ **7)** Let P be a simple polygonal region. Let $\ker(P) = \{x \in P;\ L(x,y) \subseteq P\}$ be the set of points which can "see" the entire polygon. Show that $\ker(P)$ can be computed in time $O(n \log n)$. [Hint: Show that $\ker(P)$ is an intersection of half-spaces, one for each edge of P.] There is also an $O(n)$ solution.

8) Let $S \subseteq \mathbb{R}^2$ be finite. Show that $CH(S)$ is a convex polygon whose vertices are points of S. [Hint: Let $A = \{(v,w);\ v,w \in S$ and one of the half-spaces defined by line $L(v,w)$ contains all points of $S\}$. Then $CH(S)$ is an intersection of half-spaces, one for each pair in A.]

9) Let $v_0, v_1, \ldots, v_n$ be a simple polygon with $x(v_0) \leq x(v_1) \leq \cdots \leq x(v_n)$. Simplify the algorithm given in the proof of Theorem 1 of Section 2 for simple polygons of this form.

10) let $A, B \subseteq \mathbb{R}^2$, $|A| = n$, $|B| = m$. Show that one can decide in time $O((n+m)\log(n+m))$ whether there is a line which separates A from B.

11) Let $S \subseteq \mathbb{R}^2$, $|S| = n$ and let $\epsilon > 0$. Show how to compute a convex polygon $P \subseteq CH(S)$ in time $O(n + 1/\epsilon)$ such that for all $x \in S$: $dist(x,P) \leq \epsilon \cdot diam(S)$, where $dist(x,P)$ is the minimal distance of x from any point of P and $diam(S)$ is the diameter of S. P may be called an approximate convex hull. [Hint: divide the plane into $k = 1/\epsilon$ vertical strips of width $\epsilon \cdot xwidth(S)$ where $xwidth(S) = \max\{x(v);\ v \in S\} - \min\{x(v);\ v \in S\}$. Determine the points with maximal y-coordinate in each strip and let P be the convex hull of these points.]

12) Design an $O(n \log n)$ divide and conquer algorithm for the convex hull problem in $\mathbb{R}^2$. [Hint: Use the algorithm which is implicit in the proof of Lemma 1 of Section 2.]

13) Design an $O(n \log n)$ divide and conquer algorithm for the convex hull problem in $\mathbb{R}^3$. [Hint: the crucial subroutine takes two non-intersecting convex polyhedra P_1 and P_2 and computes the convex hull of $P_1 \cup P_2$ in linear time.]

14) Let VD be the Voronoi diagram of some point set S, let $y \notin S$ and let $x \in S$ be such that $y \in VR(x)$. Show how to obtain the diagram for $S \cup \{x\}$ in time proportional to the "size of the change" in the diagram. [Hint: Construct the perpendicular bisector of x and y first, say L. Find the intersections of L with the boundary of $VR(x)$. Assume that on of the intersections lies on the boundary of $VR(x)$ and $VR(z)$. Continue moving along the perpendicular bisector of y and z, $\ldots$.]

15) Let VD be the Voronoi diagram of some point set S and let $x \in S$. Show how to obtain the diagram for $S - \{x\}$ in time $O(s \log s)$, where s is the number of edges on the boundary of $VR(x)$. [Hint: Let $e_1, \ldots, e_m$ be the spokes of the Voronoi region of x in circular order as indicated in Figure 115. Let the spokes grow simultaneously into region $VR(x)$ and find a "first" intersection (use a heap for that task); say it is the intersection of e_i and e_{i+1}. Replace e_i, e_{i+1} by the perpendicular bisector of suitable points and continue.]

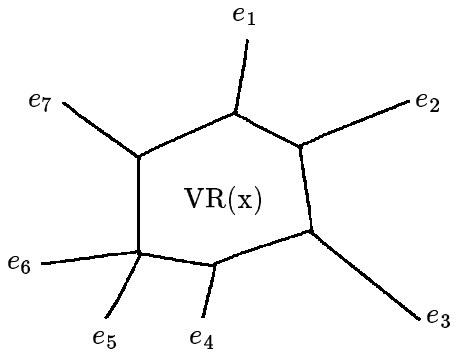


Figure 115.

16) Let S and T be finite subsets of $\mathbb{R}^2$. Develop an algorithm for computing $VD(S \cup T)$ from $VD(S)$ and $VD(T)$ in time $(|S| + |T|)$. [Hint: Use plane sweep.]

The following exercises (17–19) treat alternative methods for the searching planar subdivisions problem. For all these exercises we assume that $\hat{G}$ is a straight line embedding of a planar graph with n vertices.

17) (Slab method). Let $x_1, \ldots, x_n$ be the x-coordinates of the vertices of $\hat{G}$ in increasing order. Divide $\hat{G}$ into $n-1$ slabs by drawing vertical lines through all vertices of $\hat{G}$. In order to locate a point $(x, y) \in \mathbb{R}^2$ in $\hat{G}$, first determine the slap containing point (x, y) by binary search for x in sequence $x_1, \ldots, x_n$. Then locate the position of (x, y) within the slab by binary search of the at most $O(n)$ edges of $\hat{G}$ intersecting the slab. Note that no edges of $\hat{G}$ intersect within a slab and hence the edges can be sorted in a natural way within a slab from top to bottom. Show that this method yields a search structure of depth $O(\log n)$ and size $O(n^2)$. Find an example of a planar subdivision where the space requirement is $\Theta(n^2)$.

18) (Planar separator method). Since $\hat{G}$ is a planar graph the proof of the planar separator theorem (Section 4.10, Theorem 3) guarantees the existence of a cycle $C = x_1, \ldots, x_m$ of length $m \leq 4\sqrt{n}$ in $\hat{G}$ such that the removal of C cuts $\hat{G}$ into two subgraphs both containing at most $2n/3$ nodes, each. Use the slab method to decide whether a point lies inside or outside C. Then use the method recursively for both subgraphs. Show that this approach yields a search structure of depth $O((\log n)^2)$ and size $O(n \log n)$.

19) (Trapezoid method). For the purpose of this exercise define a trapezoid as consisting of two horizontal edges and two non-horizontal edges. Moreover, the two non-horizontal edges are subsegments of edges of $\hat{G}$ and there is no edge of $\hat{G}$ which intersects the interior of both horizontal edges. Refine a trapezoid into some number of trapezoids by

1) drawing a horizontal line through the vertex of $\hat{G}$ which has the median y-coordinate of all vertices in the trapezoid
2) refining the top and bottom half into trapezoids by using those edges of $\hat{G}$ which completely run trough the top or bottom half.

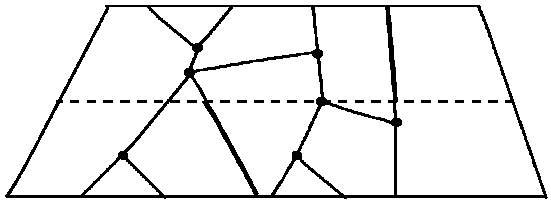


Figure 116.

In our example, the top and bottom half are both cut into two trapezoids as indicated by the heavy lines. Thus in each search step we first locate the point with respect to the horizontal dividing line and then with respect to the non-horizontal dividing lines. Note that the latter number of dividing lines is not bounded by a constant and that the number of vertices in the various sub-trapezoids varies widely. Thus it is efficient to use a weighted binary search (cf. Section 3.4) for the search with respect to the non-horizontal dividing lines; the weight of a sub-trapezoid being the number of vertices of $\hat{G}$ it contains. Show that this method yields a search structure of depth $O(\log n)$ and size $O(n \log n)$. In fact, the depth can be shown to be bounded by $3 \log n + O(1)$ if the method of Section 3.4, Theorem 7 is used for the weighted binary search.

20) Let $S \subseteq \mathbb{R}^2$. Use the Voronoi diagram of S in order to find the largest circle C such that C's center is contained in the convex hull of S and the interior of C contains no point of S. [Hint: the center of C is either a vertex of the Voronoi diagram or a point of intersection of the convex hull of S and an edge of the Voronoi diagram.]

21) Let $S \subseteq \mathbb{R}^2$. For $x \in S$ let $FVR(x) = \{z;\ dist(x,z) \geq dist(y,z) \text{ for all } y \in S\}$ be the set of points z which have x as their farthest neighbor. Show how to compute the farthest point Voronoi diagram in time $O(n \log n)$.

22) Use the farthest point Voronoi diagram (cf. Exercise 21) to find the smallest circle which contains all points of $S \subseteq \mathbb{R}^2$.

23) Given a set of n vertical or horizontal line segments show that one can compute all s intersections in time $O(n \log n + s)$. [Hint: use plane sweep; modify the algorithm given in Section 4.1.]

24) A circular segment is a segment of a circle. Given n circular segments show that one can compute all s intersections in time $O((n + s) \log n)$.

25) Given a set of closed polygonal curves show how to compute the regions defined by the union of these curves.

26) Let PP_1 and PP_2 be sets of simple polygons. Show how to compute all maximal regions which are covered by a polygon in PP_1 and a polygon in PP_2. Assume first that the polygons in PP_i, $i = 1, 2$ are pairwise disjoint. Then drop this assumption. [Hint: Modify the algorithm for decomposing polygons given in Section 4.1.]

27) Let PP be a set of polygons. Compute the boundary of the union of the polygons in PP.

28) Let PP_1 and PP_2 be sets of simple polygons. Show how to compute all maximal regions which are covered by a polygon in PP_1 and no polygon in PP_2. Similarly, compute all maximal regions which are covered by at least three polygons in PP_1 and not more than two polygons of PP_2. For what other logical connectives will your algorithm work? Can you extend your algorithm to more than two sets of simple polygons?

29) Let $L_1, \ldots, L_n$ be a set of *non-intersecting* line segments. Line L_i **dominates** L_j if L_i intersects the infinite "strip" defined by L_j and the vertical rays (extending to $+\infty$) through the two endpoints of L_j. Compute an injective ordering $ord : \{L_1, \ldots, L_n\} \to \{1, \ldots, n\}$ such that $ord(L_i) < ord(L_j)$ if L_j dominates L_i. [Hint: Use plane sweep. Augment the y-structure such that it records the restrictions of ord to the active and dead line segments.]

30) Give the details of the $O(n \log n)$ algorithm for triangulation which is described in Section 4.2.

31) Let P be a simple polygon. Compute a decomposition into a minimal number of convex parts. Use dynamic programming. [Hint: Let P be $x_0, x_1, \ldots, x_{n-1}, x_0$. Call pair (i, j) valid if either x_i or x_j is a cusp and $L(x_i, x_j)$ is inside P. For every valid pair (i, j), $i < j$, and l, $i < l < j$, compute $cost(i, j, l)$, where $cost(i, j, l)$ is the minimal number of convex parts in any decomposition of polygon $x_i, \ldots, x_j, x_i$ which uses edge $L(x_l, x_j)$.]

32) Let $P_1, \ldots, P_k$ be a set of non-intersecting simple polygons. Triangulate $P_1 \cup P_2 \cup \cdots \cup P_k$ using the edges of the polygons.

33) Let P be a simple polygon and let x and y be two points in the interior of P. Show how to compute a shortest path in the Euclidian metric from x to y which runs completely inside P. [Hint: Given a triangulation of P one can find the path in linear time.]

34) Let $S \subseteq \mathbb{R}^2$, $|S| = n$. Prove an $\Omega(n \log n)$ lower bound on the time required to compute a triangulation of S.

35) Let P and Q be planar subdivisions all of which regions are convex. Let n be the number of vertices of $P \cup Q$. Show how to compute $P \cap Q$ in time $O(n \log n + s)$, where s is the number of intersections.

36) Consider the following memory allocation problem. A memory is an array of N cells. We want to dynamically maintain the free cells such that the following requests can be answered efficiently. given integer r (the size of the request) find a block of $s \geq r$ consecutive free cells. In the **best-fit strategy** we want s to be minimal, in the **first-fit strategy** we want the free block to start at the smallest free address. Show that priority search trees can be used to implement either strategy. [Hint: Represent a block of free cells by a pair (block-size, first address in free block).]

37) Let $\hat{G}$ be a planar subdivision with no vertical edge. A **zig-zag decomposition** of $\hat{G}$ is a sequence $P_1, \ldots, P_s$ of x-monotonous paths such that

1) every edge of $\hat{G}$ belongs to exactly one path and only edges of $\hat{G}$ are used in the paths;
2) if $i < j$ and vertical line L intersects P_i and P_j then $L \cap P_i$ is not below $L \cap P_j$;
3) s is minimal among all path systems which satisfy 1) and 2).

a) For a vertex v of $\hat{G}$ let $indeg(v)$ ($outdeg(v)$) be the number of edges entering v from the left (leaving v to the right). Call v a starting vertex of $\hat{G}$ if $indeg(v) < outdeg(v)$. Show that every zig-zag decomposition satisfies

$$s = \sum \{outdeg(v) - indeg(v);\ v \text{ is a start vertex of } \hat{G}\}.$$

Show how to compute a zig-zag decomposition in time $O(n \log n)$ and space $O(n)$, where n is the number of vertices of $\hat{G}$. Make sure that your algorithm does not only compute a set $\{P_1, \ldots, P_S\}$ of paths but a *sequence* $P_1, \ldots, P_s$ of paths which forms a zig-zag decomposition.

b) Let $Q_1, \ldots, Q_m$ be a set of simple plane polygons in $\mathbb{R}^3$ with a total of n vertices. Let k be the number of edge intersections in the projection onto the xy-plane. Show how to use zig-zag decomposition to solve the hidden line elimination problem in time $O((n+k)(\log n)^2)$ and space $O(n+k)$.

c) For a zig-zag decomposition $P_1, \ldots, P_s$ let *First* be the set of first edges of paths $P_1, \ldots, P_s$. Design a plane sweep algorithm which runs in time $O(n \log n)$ and space $O(s)$ and computes set *First* and injective mapping $num : First \to [1..s]$ such that there is a path decomposition $P_1, \ldots, P_s$ with $e \in First$ being the first edge of path $P_{num(e)}$ for all e in *First*. Moreover, design a plane sweep algorithm which given set *First* and mapping *num* maintains a mapping $\overline{num}$ during the sweep such that $\overline{num}$ is defined on the active line segments and $\overline{num}(e)$ is the path which contains edge e. The algorithm should run in time $O(n \log n)$ and space $O(s)$.

d) Use the solution to part c) to refine the space complexity of the hidden line elimination algorithm of part b) to $O(n)$.

e) Can you use zig-zag decomposition for the measure problem of a union of polygons?

38) Design an algorithm for hidden line elimination under perspective projections.

39) This exercise discusses a hidden line elimination algorithm of time complexity $O((n+k)\log n)$ and space complexity $O(n+k)$. Let $Q_1, \ldots, Q_m$ be a set of simple plane polygons in $\mathbb{R}^3$ with a total of n vertices. Let Q_i' be the projection of Q_i into the xy-plane and let k be the number of intersections of edges of the Q_i''s. Obtain planar subdivision $\hat{G}$ with $n+k$ vertices by adding the edge intersections as additional vertices. View every edge of $\hat{G}$ as a pair of two half-edges by conceptually introducing midpoints. For each vertex v of $\hat{G}$ assign locally visible labels and hidden from the half-edges incident to v. A half-edge is locally visible iff none of the polygons having v as a vertex covers it. Otherwise it is hidden.

a) Assume that $\hat{G}$ is connected. Let v_0 be the vertex of $\hat{G}$ with maximal z-coordinate and let S be the maximal connected subgraph of $\hat{G}$ containing v_0 and consisting only of edges both of which half-edges are visible. Show that S is the solution to the hidden line elimination problem.

b) Derive a hidden line elimination algorithm from part a) with running time $O((n+k)\log n)$ and space requirement $O(n+k)$ for the case that G is connected.

c) Extend the solution of part *b*) to the case that $\hat{G}$ is not connected. [Hint: Apply part b) to every component of $\hat{G}$. Use exercise 40 to decide containment of components and use this information to delete covered components.]

40) Let $Q_1, \ldots, Q_m$ be a set of simple polygons with a total of n edges. Report all pairs of intersecting polygons. [Hint: Use path or zig-zag decompositions and extend the algorithm given in Section 5.1.1, Theorem 2.]

41) Given a set S, $n = |S|$, of points and iso-oriented rectangles in $\mathbb{R}^2$ report all pairs (p, R) of point and rectangle with $p \in R$. Design a divide and conquer algorithm to solve this problem in time $O(n \log n + s)$, where s is the number of pairs reported. [Hint: An algorithm similar to procedure *Intersect* of Section 5.2.1

can be used. Let *VERT* be the set of projections of the points in the frame onto the y-axis and let *LEFT* (*RIGHT*) be the set of projections of rectangles which have their right (left) boundary in the frame but their left (right) boundary outside the frame.]

42) Use a solution to the preceding exercise and Theorem 11 of Section 5.2.1 to design an $O(n \log n + s)$ algorithm for the rectangle intersection problem: Given n iso-oriented rectangles compute all pairs of intersecting rectangles.

43) Design a plane sweep algorithm for the contour problem of iso-oriented rectangles. Use segment trees. You should be able to achieve running time $O((n+p) \log n)$ fairly easily; an improvement that realizes $O(n \log n + p)$ is possible but quite involved. Here n is the number of rectangles and p is the number of contour-pieces.

44) Let S, $n = |S|$, be a set of horizontal and vertical line segments. Let pair (L_1, L_2) of elements belong to relation R if $L_1 \cap L_2 \neq \emptyset$. Compute the equivalence classes of relation R in time $O(n \log n)$. The equivalence classes are also called connected components.

8.8. Bibliographic Notes

Section 1 is based on Dobkin/Kirkpatrick (82), except for Theorem 5 which is taken from Shamos (75). The former paper also contains three-dimensional analogues of Theorems 1 to 4; cf. Exercise 2. A solution to Exercise 7 can be found in Lee/Preparata (79).

McCallum/Avis (79) gave the first linear time convex hull algorithm for the vertices of a simple polygon (Theorem 1). Our proof follows Graham/Yao (81). Theorem 2 is by Graham (72) and Theorem 3 is by Shamos (75). The lower bound the convex hull problem in the algebraic decision tree model is by Yao (81) and Ben Or (83). Theorem 4 is due to Preparata (79) and Theorem 5 comes from Overmars/v. Leeuwen (81). A solution to Exercise 13 can be found in Preparata/Hong (79), and a solution to Exercise 11 can be found in Bentley/Faust/ Preparata (82). Finally, on $O(n \log H)$ algorithm for the convex hull problem has recently been given by Kirkpatrick/Seidel (82), where H is the number of vertices of the convex hull.

The $O(n \log n)$ algorithm for constructing Voronoi diagrams is due to Shamos/ Hoey (75). Theorems 5, 6, 7 and Exercises 20, 21, and 22 can also be found there. Lemma 7 and Theorem 2 on searching planar subdivisions is due to Kirkpatrick (83). Section 3.2.2 combines the work of Lee/Preparata (77) who introduced the concept of path decomposition and proved Lemma 10, Harel (80) who showed how to quickly compute lowest common ancestors, and Edelsbrunner (83) who finally reduced search time to $O(\log n)$. Theorem 4 on searching dynamic planar subdivisions

is a joint effort together with O. Fries. The exercises are taken from Kirkpatrick (79) (Exercise 16), Dobkin/Lipton (76) (Exercise 17), Lipton/Tarjan (77) (Exercise 18), Bilardi/Preparata (82) (Exercise 19). An algorithm for searching subdivisions with curved boundaries can be found in Edelsbrunner/Maurer (81).

The sweep paradigm was introduced by Shamos/Hoey (76). Theorem 1 and exercises 23 and 24 are taken from Bentley/Ottmann (79); Brown reduced the space requirement of their algorithm from $O(n+s)$ to $O(n)$. An $O(n(\log n)^2+s)$ algorithm for line segment intersection has recently been found by Chazelle (83). Theorem 2 and Exercise 35 are the work of Nievergelt/Preparata (82). Ottmann/Widmeyer/ Wood (82) show how to solve Exercises 52 to 28, and Guibas/Yao (80) solve Exercise 29. The triangulation algorithm for simple polygons is taken from Hertel/Mehlhorn (83); some of the applications (Theorems 5 and 6) are also from there. Theorem 7 and Exercise 33 are due to Chazelle (82). Chazelle's paper also contains a separator Theorem for simple polygons. Exercise 31 comes from Green (83); a related problem was studied in Chazelle/Dobkin (79). The section of space sweep is based on Hertel/Mehlhorn/Mäntyla/Nievergelt (83). Alternative algorithms for intersecting convex polyhedra can be found in Muller/Preparata (83) and Dobkin/Kirkpatrick (82); cf. Exercise 2.

Interval trees were introduced by McCreight (80) and Edelsbrunner (80) and the proof of Theorem 2 is taken from their papers. The first proof was given by Bentley/Wood (80). Priority search trees are due to McCreight (81) and the entire Section 5.1.2 and Exercises 36 are taken from this paper. Segment trees were introduced by Bentley (77) as a method for solving the measure problem (Theorem 7). Dynamic interval trees and segment trees were discussed by Edelsbrunner (82); the general discussion which also allows intervals to snare endpoints is new. The idea of using path decomposition in plane sweep algorithms comes from Ottmann/Widmeyer (82), they used zig-zag decompositions in the sense of Exercise 37. The algorithm of Exercise 39 was developed by Schmitt (81). The whole Section 5.2 is taken from Güting (83), and Section 5.3 is entirely taken from Edelsbrunner/Maurer (81). A solution to Exercise 43 can be found in Güting (82).

Geometric transforms are discussed at length in Brown (79) and Theorems 1, 4, 5 are by him. Chazelle (83) and Edelsbrunner/O'Rourke/Seidel (83) proved Theorems 2 and 3. The latter paper treats the problem in higher dimensions, too.