2 Computational Geometry—A User's Guide

David P. Dobkin
Diane L. Souvaine
Princeton University

ABSTRACT

During the past decade, significant effort has been devoted to the design and analysis of efficient algorithms for computations which are geometric in nature. Most of this effort has been devoted to problems without direct applications in robotics. The methods used, however, should have a significant impact upon the development of algorithmic methods in robotics. A few of the more promising methods are described below along with some geometric applications.

1. INTRODUCTION

The past decade has brought significant progress in the development of algorithms for solving problems which are geometric in nature (see e.g., Lee & Preparata, 1984). Many of these algorithms depend on methods that were developed to solve problems in other branches of computer science. We believe that this trend will continue and that the methodology of computational geometry will ultimately find significant applications in robotics. Here we present and explain the methods used in developing three families of geometric algorithms. Two of these methods involve the decomposition of a problem into subproblems which can be more easily solved. The third involves transforming a problem in its entirety to a new format which may be more tractable.

The first of these methods is a hierarchical searching method. Here, a geometric problem is preprocessed to provide a coarse representation of the entire problem. Search queries upon the whole are then used to localize the region in which the problem is to be solved. Queries of this type alternate with computations which yield continually finer descriptions of these continually smaller

From Advances in Robotics 1: Algorithmic and Geometric Aspects of Robotics. J. Schwartz and C. Yap, editors. Lawrence Erlbaum Associates, Hillsdale, NJ, 1987.

regions. Efficient algorithms result from balancing the two processes of localizing the search and of increasing the detail. Algorithmic efficiency is then balanced against preprocessing time and storage space requirements. A simple example of this process is the binary search of a sorted array. Here, the region is an interval of array indices and the finer description consists of increasing the density of information.

The second method presented is the divide-and-conquer method. Here, a problem is broken into subproblems which can be solved recursively. A method is then defined for combining solutions to subproblems in order to yield a solution to the entire problem. An extension of sort-merge techniques, this method has found significant application to all types of algorithm design. Attention here is focused on issues involved in applying it to problems which are geometric in nature. In some contexts, these first two methods often work together. For example, a divide and conquer method (i.e., sort-merge) might be used to order elements of an array to allow the use of a hierarchical search method (e.g., binary search). For instance, a Voronoi diagram (Shamos & Hoey, 1975), constructed via divide-and-conquer, yields a planar subdivision that we could search by the methods of section 2.

Finally, the duality method is one which has been used with success by mathematical programmers for the past 4 decades. We focus here specifically on problems involving 2- and 3-dimensional geometries, where duality is used as a transformation. Given two sets A and B and a query regarding their interrelationship, we apply the transform T and answer a different, hopefully easier, query about the relationship between T(A) and T(B). The form of T and its uses are described here.

The premise of this paper is that the algorithm for solving a particular problem is often less important than the methodology which led to the algorithm. We encourage the reader to look upon this paper as he/she would look upon a user's manual for a new programming system. We present methods both in their pure form and as a means to solving geometric problems. To continue the user's manual analogy, the reader might consider the former as an exposition of the macros which come with a particular system. The latter then provide examples of these macros in action. The applications we consider depend on geometrics principles: planar point location, convex hull construction and updating, and computation of polygon, disk, and half-space intersections. Combining these principles with relevant sets of facts for robotics should make both our methods and results useful to robotics workers.

2. HIERARCHICAL SEARCH

2.1. Binary Search

Binary search is an effective technique for locating a particular object within a search domain, providing that the objects have some inherent ordering. A simple

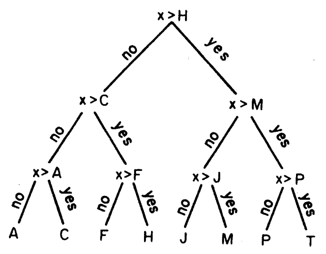


FIG. 2.1. Sample binary search tree

example involves the search via random-access probes for an element in an ordered linear array. The procedure is modeled by the binary tree depicted in Fig. 2.1. The elements of the array are contained in order in the leaves and internal nodes of the tree. Figuratively, the search begins at the root of the tree comparing the desired element to the middle element of the array. Either answer will delete half of the array, and thus half of the tree, from further consideration. The search moves recursively to either the left subtree, comparing to the first quartile, or the right subtree, comparing against the third quartile. The search continues until the search element is found or the frontier of the tree is reached. We might say that the first question provides a coarse idea of the location of the number (vis à vis the median) and each succeeding question provides finer and finer detail; that is, in a binary sense, each question identifies one further bit of the array index of the number.

When we apply this technique to geometric problems, the process may become somewhat more complicated. Unlike the problem above, many geometric problems have no obvious one-dimensional ordering leading to an obvious progression of questions. Preprocessing is necessary to regularize the structure of the original problem. This yields a coarse description of the problem at which the search begins. Searching then involves answering questions based on this coarse description. The resultant answers point us toward further queries based on slightly more detail within a smaller instance of the problem, and so forth until an answer is reached. The preprocessing time must be balanced against the searching time as we describe below.

We consider here a problem to which this technique applies.

Input: A collection of N disjoint polygons in the plane.
Query: For a given point P, find all polygons to which it belongs.

The naive approach to this problem, merely testing the point P against each polygon in turn, quickly becomes impractical as both the number of polygons in search domain and the number of distinct query points increase. We develop a method of preprocessing the polygons to decrease individual query time for cases when the number of queries is large.

At first, we will restrict our attention to polygons which are axis-parallel rectangles. We represent these rectangles by their lower left and upper right vertices. Our development begins with a simple algorithm for this case and works toward a general procedure which applies to arbitrary polygons. All of the key ideas, the algorithmic "macro," can be seen in the simple rectangle case.

2.2. Rectangle Search I: Specific Example

We consider the simple example depicted in Fig. 2.2a where all vertices have integer coordinates. The search domain consists of four rectangles:

$$R_1 = \{(1, 5), (4, 7)\}$$
$$R_2 = \{(0, 0), (2, 4)\}$$
$$R_3 = \{(5, 3), (6, 6)\}$$
$$R_4 = \{(3, 1), (7, 2)\}$$

We begin by finding the smallest rectangle, Q, which encloses all of the search rectangles. In this case $Q = \{(0, 0), (7, 7)\}$. Together, the boundaries of Q and the four rectangles form a planar graph, S, with nineteen vertices and twenty-two edges which subdivides the plane into six regions: the unbounded region, R_5 which is exterior to Q; the four rectangular regions; and the region we call R_0 which is interior to Q but exterior to all four rectangles. Since all of the edges are finite line segments and thus only one region is unbounded, we can call this graph a "finite planar subdivision" (see Fig. 2.2b).

In order to determine which, if any, of the four rectangles contains the query point $P = (x, y)$, we first determine whether P lies in the unbounded region R_5 by making four comparisons to check whether either x or y is less than 0 or bigger than 7. If so, our search terminates. Otherwise, we need to determine the region interior to S which contains P. Ideally, a few quick comparisons would localize our search further, in a pattern similar to one-dimensional binary search. Unions of regions of the planar subdivision, however, produce awkward (i.e., nonrectangular) shapes against which P cannot easily be tested. We need to restructure the search domain. Since all current vertices have integer coordinates, we may refine the subdivision S, to a new planar graph S_0 which forms a complete grid of unit squares. As each of these squares is a subset of exactly one of the regions in S, the representation we choose for S_0 will include a pointer for each square face to its "parent" in S (see Fig. 2.2c).

At first, the task of locating P in one of the forty-nine regions of S_0 seems more complicated than the original problem. The regular shape of the regions, however, benefits us. If we delete an interior vertex, T, and all of the edges adjacent to it, four squares are merged to form one larger square. If we knew that P were included in the larger square and we knew the coordinates of the point T, we could quickly determine to which of the original four squares P belonged.

Therefore, we will create a new subdivision S_1 using the following procedure: sweep a vertical line left-to-right across the plane; stop every second time that it

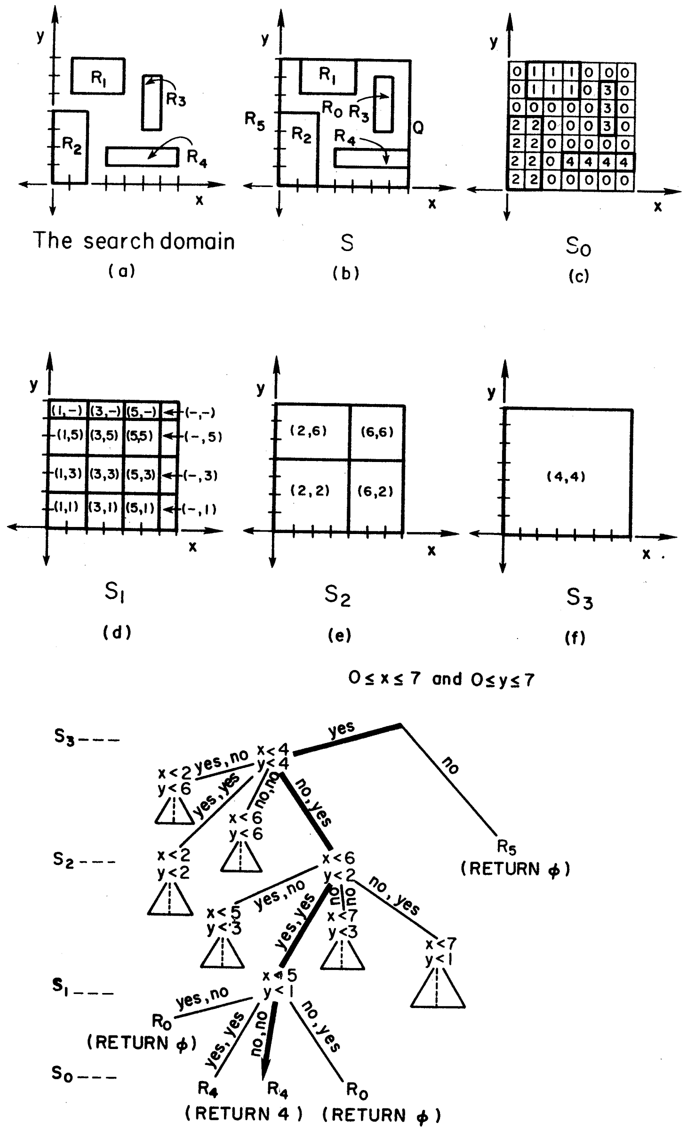


FIG. 2.2. Hierarchy for rectangle search I

intersects (contains) vertices and vertical edges of S_0; move bottom-to-top along the stopped line, deleting every second vertex and its adjacent edges; where necessary, add edges to preserve the boundary of Q. Here, both coordinates of a deleted vertex will be odd numbers. In most cases, four squares will be merged to form a larger square, but along the upper and right boundaries of S_0, two or even no squares may merge (see Fig. 2.2d). The representation of S_1 will associate a tag with each region which not only points to its parents, or originators, in S_0 but also identifies the vertex they all share.[1] This process cuts the size of the search domain from forty-nine internal regions to sixteen. If we repeat the process, we achieve a subdivision S_2 which has four internal regions (Fig. 2.2e) and then a subdivision S_3 with just one (Fig. 2.2f). The sets, S_0, S_1, S_2, S_3 constitute a four-level hierarchy. We measure the size of each subdivision, S_i, by the number of finite regions it contains:

$$|S_0| = 49;\ |S_1| = 16;\ |S_2| = 4;\ |S_3| = 1.$$

Our algorithm to determine which of the original rectangles contains P proceeds as follows: make at most four comparisons to determine whether P lies in the interior or exterior of S_3; if P lies in the exterior, return the null symbol $\varnothing$; if P lies in the interior of S_3, compare x and y against the two coordinates stored in S_3 to determine which region in S_2 contains P; then test P against the coordinate(s) stored there to determine which region in S_1 contains P; test P against the coordinate(s) stored in that region to determine which square in S_0 contains P; look up the original region, R, to which that square belongs; if R is one of the rectangles, $R_1, \ldots, R_4$, return the number of that rectangle; otherwise, return $\varnothing$. Fig. 2.2g demonstrates the path followed through the hierarchy for the case where $P = (11/2, 3/2)$. Four comparisons are executed at the root, followed by two more at each internal node in the tree for a total of ten comparisons.

As already mentioned, this algorithm successively considers finer details (rectangles of smaller area) as it narrows the search domain (by eliminating sets of rectangles).

2.3. Rectangle Search I: General Case

We now extend the algorithm to the more general case where the search domain consists of N axis-parallel rectangles, $R_1, R_2, \ldots, R_N$, whose vertices have real-valued coordinates. First, we project all of the vertices onto the x-axis, yielding at most $2N$ distinct values. Without loss of generality, we assume that the $2N$ values $x_0, x_1, x_2, \ldots, x_{2N-1}$, are distinct and represented in increasing order. Similarly, project all of the vertices onto the y-axis, achieving an ordered list $y_0, \ldots, y_{2N-1}$. Note that the values x_0 and y_0 need not originate with the same

[1]The term "parent" is used loosely, as a region in S_1 generally points to four distinct squares in S_0. In cases of multiple parenting, we shall henceforth use the term "originators."

vertex, and x_0 and x_1 need not define the x-extent of a single rectangle. The smallest rectangle which circumscribes $R_1, \ldots, R_N$ is $Q = \{(x_0, y_0), (x_{2N-1}, y_{2N-1})\}$. S is the planar graph formed by the boundaries of Q and $R_1, \ldots, R_N$; R_{N+1} is the region exterior to Q; and R_0 is the region interior to Q but exterior to $R_1, \ldots, R_N$ (see Fig. 2.3a).

To achieve the regular structure needed for S_0, proceed as follows:

1. for each x_i, $0 \le i \le 2N-1$, draw the line segment from (x_i, y_0) to (x_i, y_{2N-1});
2. for each y_i, $0 \le i \le 2N-1$, draw the line segment from (x_0, y_i) to (x_{2N-1}, y_i);
3. reinterpret the resulting picture as a planar graph, S_0.

Since Q was divided at each x-coordinate and y-coordinate which represented a vertex of the original rectangles, each one of the regions (faces) of S_0 is a subset of exactly one of the regions, $R_0, \ldots, R_N$. Thus, the representation of each face in S_0 will include a pointer to its parent in S.

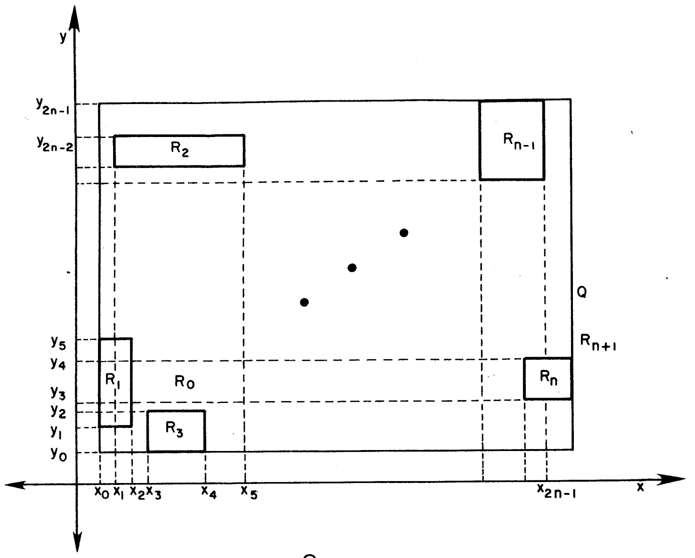
Despite the fact that the regions in S_0 need not be squares, all interior vertices of S_0 will have degree four, as in the previous example. Thus we can use the procedure defined earlier to create subdivisions $S_1, S_2, S_3, \ldots$, each having fewer faces than the previous one. We continue until we reach an S_m which has a single finite region identical to the bounding box Q, (see Figs. 2.3b and 2.3c).

With the hierarchy, $S_0, \ldots, S_m$ in place, we can determine which of the original rectangles contains P by the following procedure:

```
Begin
    1. decide whether $P$ lies in the bounded or unbounded region of $S_m$
    2. if $P$ is in the unbounded region, return $\varnothing$ and stop
    3. $v_{m-1} \leftarrow$ the vertex associated with bounded region of $S_m$
    4. for $i \leftarrow m - 1$ to 1 step $-1$
            begin
         5. test $(x, y)$ against $v_i$ to determine which of its originators in $S_i$
            contains $P$
         6. $v_{i-1} \leftarrow$ the vertex stored with that originator
       end;
    7. look up in $S_0$ the original region $R_j$ to which $v_0$ points
    8. if $j = 0$, then return $\varnothing$, else return $j$.
End.
```

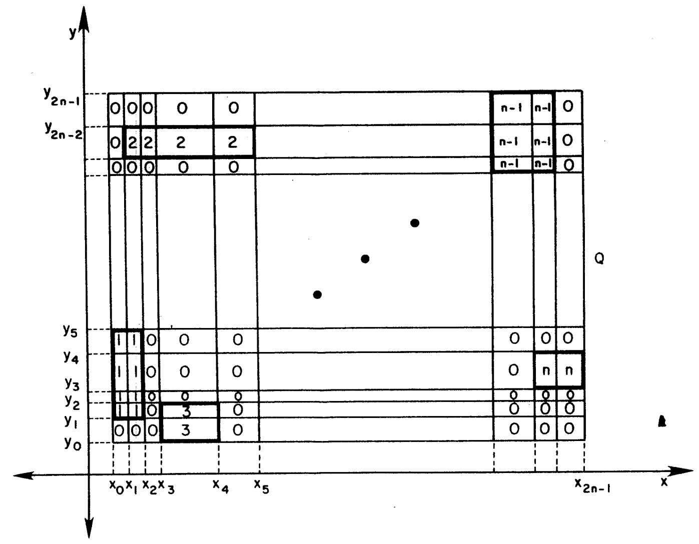
The set-up phase of the algorithm, steps 1–3, and the interior of the loop, steps 5–6, each require constant time.

Thus, the time complexity of the algorithm is linear in m, the number of times the algorithm executes the inner loop. If we reconsider Fig. 2.3b, we can see that S_0 contains $(2N-1)^2 = j^2$ regions ($|S_0| = j^2$). In the process of creating S_1, we eliminate the vertical divisions for $\lfloor j/2 \rfloor$ values of x. In like fashion, we delete the horizontal divisions for $\lfloor j/2 \rfloor$ values of y. Thus, S_1 must contain $(\lceil j/2 \rceil)^2 = k^2$



S

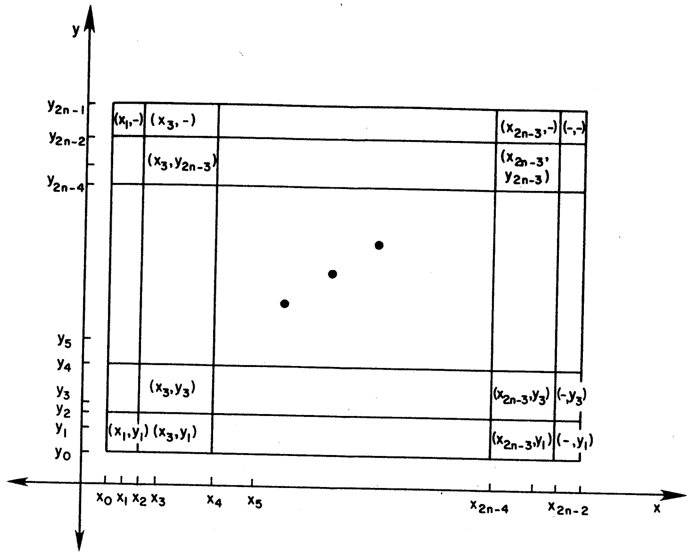
(a)



S_0

(b)

FIG. 2.3. Hierarchy for general case



S_1

(c)

FIG. 2.3.c

regions. In the same manner, the process of creating S_2 removes $\lfloor k/2 \rfloor$ horizontal and vertical divisions, so $(|S_2| = (\lceil k/2 \rceil)^2$ finite regions. Since S_m contains but one finite region, $m = \lceil \log (2N-1) \rceil$.[2]

Thus, in determing the region containing P, this algorithm performs $4 + 2 \lceil \log (2N - 1) \rceil$ comparisons, or $O(\log N)$ comparisons. The naive algorithm requires $4N$ comparisons, or $O(N)$ comparisons. Thus, even for small N (e.g. $N = 3$) this algorithm is preferable, provided the necessary hierarchy can be created easily. Given that S_0 contains $4N^2$ vertices and $(2N - 1)^2$ regions, however, just the representation of the hierarchy must take $\Omega(N^2)$ time. As a result, this algorithm has value only when a large number of points P need testing and when the size of N does not make N^2 prohibitively large.

2.4. Rectangle Search II

The hierarchical algorithm described above actually subdivides Q into more regions than necessary. Our second algorithm offers only a slight improvement in the worst-case space requirement, but is far more efficient on average and should make the later exposition clearer. In the above, all of the vertices of the N rectangles, $R_1, \ldots, R_N$, are projected onto both the x- and y-axes, creating $2N - 1$ finite intervals in each direction. Then, added horizontal and vertical

[2]Note that log x here always means $\log_2 x$.

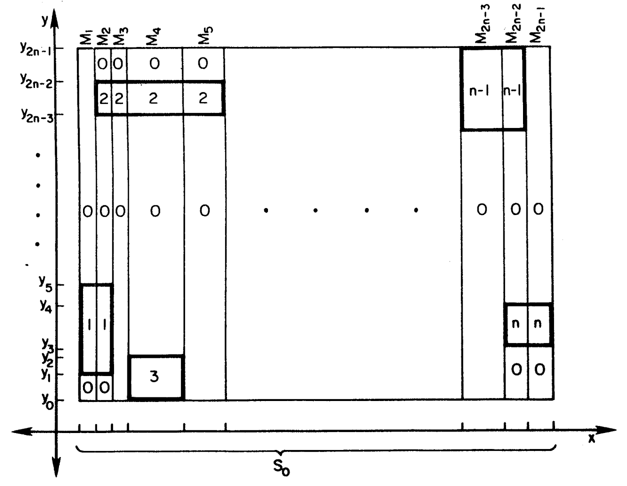


FIG. 2.4. Decomposition for rectangle search II

segments create a total of $(2N - 1)^2$ distinct regions (see Figs. 2.3a and3b). Now we add only those vertical segments containing an edge of a rectangle. The new S_0 contains only the actual intervals along the x-axis, rather than the entire resulting graph. In other words, $S_0 = \{[x_0, x_1], [x_1, x_2], \ldots, [x_{2N-2}, x_{2N-1}]\}$. The sets, $M_1, \ldots M_{2N-1}$, consist of the distinct vertical intervals represented in the vertical slab defined by each respective horizontal interval. In the example of Fig. 2.4,

$$\begin{aligned}
M_1 &= \{[y_0, y_1], [y_1, y_5], [y_5, y_{2N-1}]\},\\
M_2 &= \{[y_0, y_1], [y_1, y_5], [y_5, y_{2N-3}], [y_{2N-3}, y_{2N-2}], [y_{2N-2}, y_{2N-1}]\},\\
M_3 &= \{[y_0, y_{2N-3}], [y_{2N-3}, y_{2N-2}] [y_{2N-2}, y_{2N-1}]\},\\
&\vdots\\
M_{2N-1} &= \{[y_0, y_3], [y_3, y_4], [y_4, y_{2N-1}]\}.
\end{aligned}$$

Each interval of M_i has a tag identifying the original region $R_0, \ldots, R_N$ which it represents. The size of M_i, $|M_i|$, is defined as the number of finite intervals it contains. $|S_0|$ is defined similarly.

Now we search the two dimensions sequentially. Instead of defining regions of positive area, the new S_0 contains only intervals. First, we need to define a hierarchy of sets $S_1, \ldots, S_m$, which will enable us to determine quickly which interval of S_0 contains the x-coordinate of P. Then we need to define a similar hierarchy for *each* of the $M_1, \ldots, M_{2N-1}$. Thus, when the correct x-interval has been chosen, we select the corresponding M_i. A search of that hierarchy will determine the y-interval containing P and thus the original region to which P belongs.

Since the processes are identical, we describe the merge and search methods for the S hierarchy only. Recall, that $S_0 = \{[x_0, x_1], [x_1, x_2], [x_2, x_3], \ldots, [x_{2n-2}, x_{2n-1}]\}$. Beginning with x_1, we select every other endpoint. In each case, we will merge the two intervals sharing that endpoint and insert the new interval into the set S_1. In the representation of S_1, however, each interval will have a pointer back to its originators in S_0 as well as a tag identifying the endpoint which the originators share. We will reiterate this process until we have a hierarchy of sets,

$$\begin{aligned}
S_0 &= \{[x_0, x_1], [x_1, x_2], \ldots, [x_{2N-2}, x_{2N-1}]\}\\
S_1 &= \{[x_0, x_2], |x_2, x_4|, \ldots, |x_{2N-4}, x_{2N-2}|, |x_{2N-2}, x_{2N-1}|\}\\
&\vdots\\
S_m &= \{|x_0, x_{2N-1}|\}
\end{aligned}$$

As before, $m = \lceil \log (2N - 1) \rceil$.

The search process for a point $P = (x, y)$ begins by determining whether x lies in the interval $[x_0, x_{2N-1}]$ (two comparisons). If not, we are done. If so, then we look at the tag value stored in S_m and determine to which originator x belongs (one comparison). We repeat the process, until we know which basic interval in S_0 contains x. (So far we have completed $2 + \lceil \log (2N - 1) \rceil$ comparisons.) We select the appropriate M_i and perform a similar search in its hierarchy. Since M_i has at most $2N - 1$ intervals, this second phase search requires at most $2 + \lceil \log (2N-1) \rceil$ comparisons. When we have isolated the appropriate interval at the bottom layer, we can read off the original region which contains P.

This algorithm preserves the $O(\log N)$ query time of the previous example, but does not achieve the desired reduction in worst-case preprocessing time. The example depicted in Fig. 2.5 maximizes the total number of intervals defined over *all* of the M_i:

$$\begin{aligned}
\sum_{i=1}^{2N-1} |M_i| &= 2 \sum_{i=1}^{N-2} (2i + 1) + 2(2N - 2) + (2N - 1)\\
&= 4 \sum_{i=1}^{N-2} i + 2(N - 2) + 2(2N - 2) + (2N - 1)\\
&= 2(N - 2)(N - 1) + 8N - 9\\
&= 2N^2 + 2N - 13
\end{aligned}$$

Since $|S| = 2N - 1$, the bottom level of this algorithm contains a total of $2N^2 + 4N - 14$ intervals. The previous algorithm has $(2N - 1)^2$ or $4N^2 - 4N + 1$ regions at the bottom level.

In the worst case, the second algorithm represents insignificant improvement; both algorithms may require $\theta(N^2)$ time to represent their bottom level objects and thus may require $\Omega(N^2)$ preprocessing. The example depicted in Fig. 2.5,

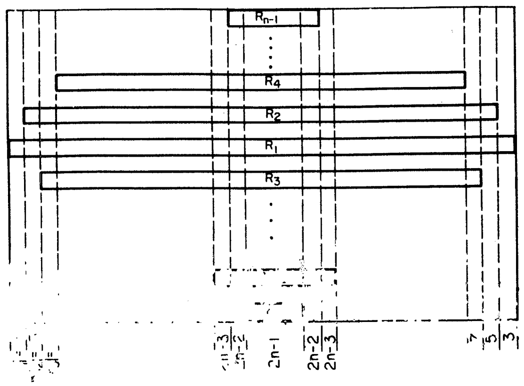


FIG. 2.5. Instance of rectangle search II yielding $O(n^2)$ intervals

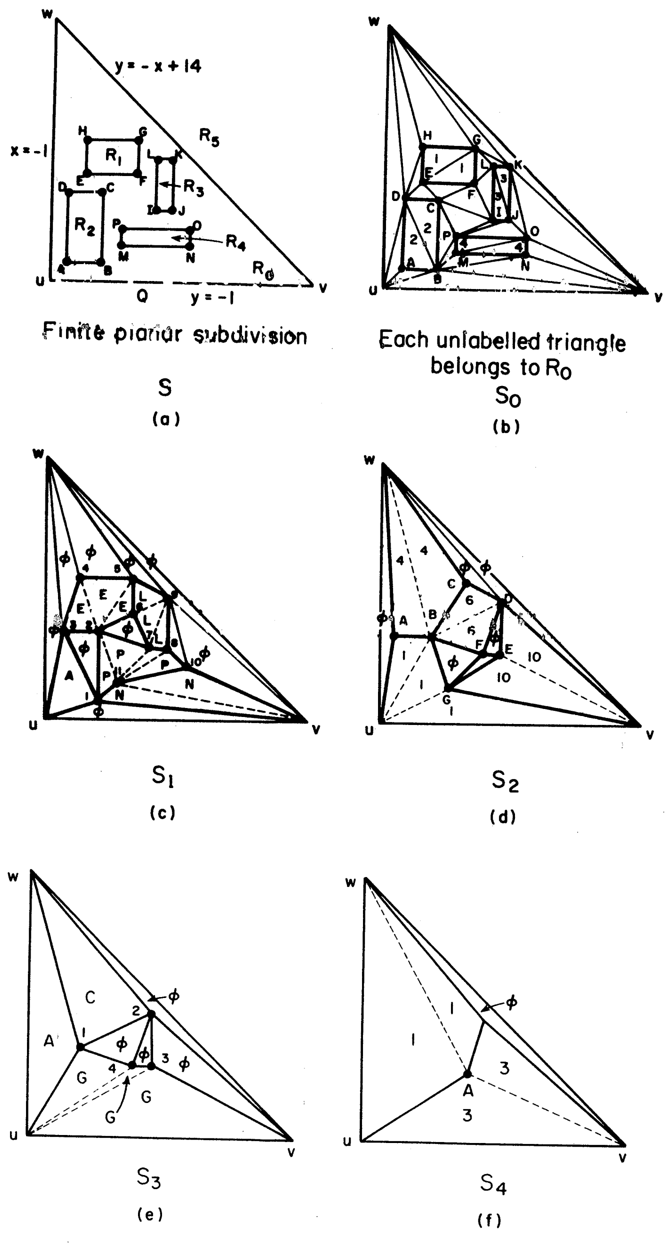
however, is quite unusual. On average, the second algorithm should substantially outperform the first. It also describes a "slabbing technique" of searching coordinates sequentially which is a valuable "macro" in computational geometry. In addition, it helps to clarify the criteria for effective hierarchies.

2.5. Rectangle Search III: Specific Example

In each of the cases above, the regularization process created too many new regions. The result was an algorithm using quadratic space. Our goal is to develop an algorithm which requires subquadratic preprocessing time and storage space while maintaining an $O(\log N)$ query time. What we need is a subdivision which creates as few new regions as possible while having uniform faces conducive to hierarchical abstraction. Euler's formula shows the feasibility of this approach: a planar graph on n vertices can have at most $2n - 4$ regions. Thus a regularization process which adds no vertices can add only a linear number of new regions.

Let us return to the four rectangle example of Fig. 2.2a. This time, we choose a large triangle Q which contains all of the rectangles in its interior. S is defined as the finite planar subdivision formed by the boundaries of Q and the four rectangles (see Fig. 2.6a). All vertices have alphabetic labels. Instead of adding vertical and horizontal segments which must create new vertices, we add a sequence of disjoint edges between preexisting vertices until every region is a triangle (see Fig. 2.6b). In the representation of the triangular subdivision, S_0, each face is associated with an integer which identifies its parent in S.

Limiting the number of new regions has sacrificed some of the uniformity. In the S_0 of algorithm I, each face was rectangular and every interior vertex had degree four. To create S_1, we chose a collection of independent (i.e., no two



y = - x + 14
x = -1
y = -1
Q
R_1 R_2 R_3 R_4 R_5 R_0
Finite planar subdivision
S
(a)
Each unlabelled triangle belongs to R_0
S_0
(b)
S_1
(c)
S_2
(d)
S_3
(e)
S_4
(f)

FIG. 2.6. Hierarchy for rectangle search III

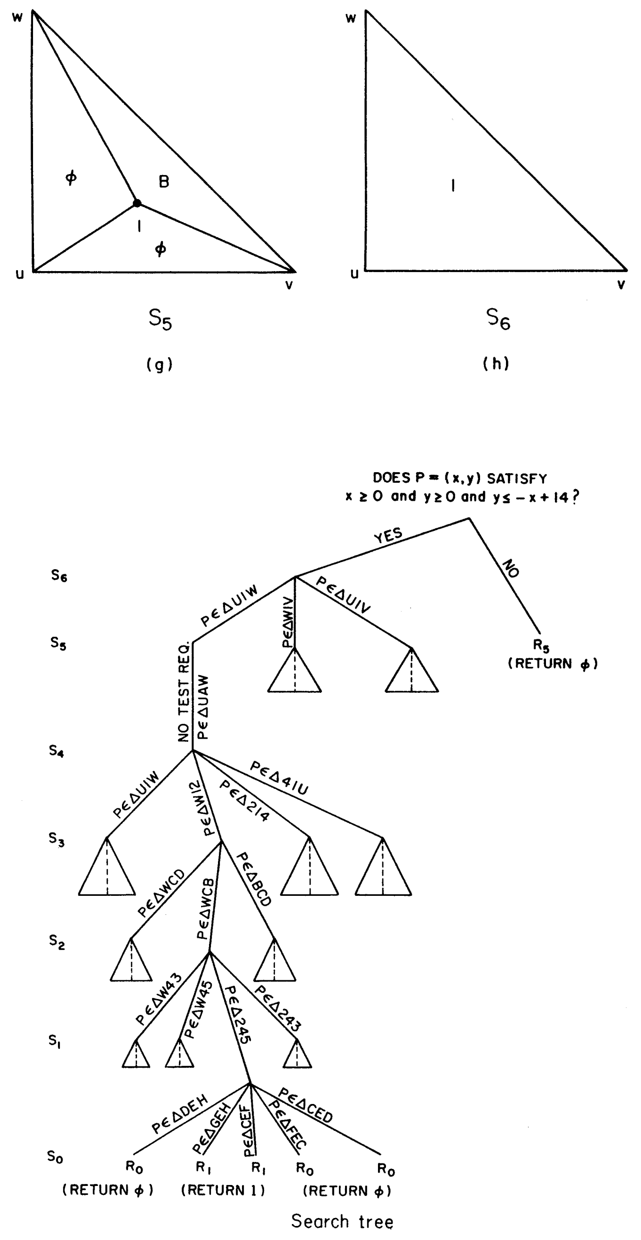


FIG. 2.6.g, h, & i

vertices share an edge), interior vertices from S_0. The deletion of each of these vertices with its adjacent edges automatically produced a new rectangular subdivision. In Fig. 2.6b, consider the five independent vertices A, E, L, N and P. Vertex A has degree 3, while N has degree 4, E and L have degree 5, and P has degree 6. Thus, the deletion of these vertices and their incident edges will produce a new triangle, a quadrilateral, two pentagons and a hexagon (see Fig. 2.6c).

The success of the hierarchical approach depends on having the same type of subdivision at each level in the hierarchy. Consequently, we form a triangular subdivision S_1 by retriangulating the quadrilateral, hexagon and two pentagons using the dotted segments shown in Fig. 2.6c. Note that the interior vertices of S_1 have numeric labels, while the vertices of S_0 are labeled alphabetically. We shall continue to alternate alphabetic and numeric labels throughout the hierarchy to aid in distinguishing between adjacent levels.

The subdivision S_1 contains two kinds of triangles. The triangles in S_1 which are identical to triangles in S_0 are considered "old" triangles. Each "old" triangle will contain a "triangle pointer" to its duplicate, or "parent," representation in S_0. The subdivision S_1, however, has ten fewer regions than does S_0. The three triangles in S_0 sharing vertex A have been replaced by a single triangle in S_1; the four triangles sharing vertex N, by two triangles; the six triangles sharing vertex P, by four triangles; and the five triangles sharing E and L respectively, by three triangles. The representation of each of these "new" triangles contains a "vertex pointer" to the respective vertex in S_0.

The location of a point P in S_0 is easily derived from its position in S_1. Suppose P lies in the triangle of S_1 joining vertices 1, 2, and 3, denoted $\Delta 123$. Then P also lies in ΔBCD of S_0 and belongs to R_2. If P lies in $\Delta 234$ of S_1, then it also lies in the pentagon 23456 formed by the deletion of vertex E from S_0. We test P against the edges $\overline{EC}$, $\overline{ED}$, $\overline{EF}$, $\overline{EG}$ and $\overline{EH}$ to determine the angle at E containing P. Depending on P's location in $\Delta 234$, P lies in one of ΔCED, ΔDEH, ΔHEG, ΔGEF, or ΔFEC. Consequently, P might lie in either R_0 or R_1.

We repeat the process, deleting vertices 1, 4, 6, and 10 and retriangulating to form a subdivision S_2, with 8 fewer faces. The deletion of A, C, and G produces S_3; deleting 1 and 3 leads to S_4; deleting vertex B produces S_5; finally, deleting vertex 1 yields S_6 (see Figs. 2.6d–h).

Given $P = (3, 6)$, the algorithm follows the search tree depicted in Fig. 2.6i. First, we confirm that P lies in the interior of S_6. Next, P is found to lie in $\Delta U1W$ (by testing the ray $1P$ against the edges incident to vertex 1 in S_5). That triangle appears unchanged in S_4 except for the renaming of a vertex, so $P \in \Delta\ UAW$. $\Delta\ UAW$ in S_4 points to the vertex 1 in S_3. Consequently, we consider the four triangles in S_3 which share vertex 1 and determine that $\Delta 12W$ contains P. Moving to S_2, an examination of rays $\overrightarrow{CB}$, $\overrightarrow{CD}$, and $\overrightarrow{CW}$ proves that P belongs to ΔWCB. Subsequently, we determine that P lies in $\Delta 245$ of S_1 and in ΔEHG of S_0. Thus P belongs to R_1. This particular example required a total of 22 comparisons.

For the small example we have been considering, as we might expect, this new algorithm runs more slowly than either algorithm I or the naive algorithm. For a problem with only four original regions, the hierarchy contains seven levels and a cumulative total of 89 regions. Analysis of the algorithm in the general case, however, proves that it is asymptotically superior to the naive algorithm and the two other rectangular algorithms defined above. In addition, we note that nowhere did the algorithm depend on the rectangular shape of the original objects; thus the algorithm applies equally well for a search-domain consisting of general polygons. Furthermore, the searching done by the algorithm was hierarchical in nature, exactly as was the searching done by the first two algorithms.

2.6. General Polygon Search

In this case, our search domain consists of irregularly shaped polygons, rather than rectangles. Since a given polygon could have an unlimited number of sides, we will measure the search domain by the total number of vertices which it contains, rather than by the number of polygons. Similarly, at each level in the hierarchy, $|S_i|$ will represent the number of vertices S_i contains, rather than the number of regions.

The first task is to find a triangle, Q, which surrounds all of the polygons. We use the following approach: Scan all the vertices to determine x_0, the smallest x-coordinate; y_0, the smallest y-coordinate; and b_0, the largest value of $y + x$. The three sides of Q will lie on the lines $y = y_0 - 1$, $x = x_0 - 1$ and $y = -x + b_0 + 1$. Thus Q is an isosceles triangle with a right angle at $(x_0 - 1, y_0 - 1)$. The union of the bounding triangle Q and the given polygons forms a planar graph S of n vertices (three more than the number of original polygonal vertices). We triangulate S arbitrarily to form S_0 (see Figs. 2.6a and b). The triangulation of a planar subdivision on n vertices, S, to form a triangular subdivision, S_0, requires at most $O(n \log n)$ time.

The efficiency of this algorithm will depend heavily on both the number of vertices eliminated at each stage of the hierarchy (number of levels necessary) and the degree of each vertex eliminated (the number of originators a single triangle might have). We will need to make wise choices. Being fully triangulated, S_0 has $3n - 6$ edges and $2n - 4$ regions by Euler's relation. Since every edge is incident on two vertices, the average vertex degree must be $(6n - 12)/n$, or somewhat less than 6. To achieve this average value, at least half of the vertices must have degree smaller than 12. Let V include those vertices of S_0 with degree < 12. Thus, $|V| \geq n/2$. Let us choose an independent subset V' of V. A straightforward elimination procedure applies. Choose any vertex $v \in V$ and place it in V'. Then delete both v and any vertices adjacent to v from V. Since v has at most 11 neighbors, no more than 12 vertices are deleted from V. We can repeat this process at least $|V|/12$ times. Consequently, when V is empty, V' will contain

more than $n/24$ independent vertices. Therefore, let each S_i be achieved by deleting independent vertices of degree ≤ 11 from S_{i-1}. The process stops at S_m which has 1 triangle.

With the hierarchy, $S_0, \ldots, S_m$ in place, which we will call an S-tree, we can determine which of the original polygons contains P by the following procedure:

```
Begin
    1. decide whether $P$ lies in the bounded or unbounded region of $S_m$
    2. if $P$ is in the unbounded region, return $\varnothing$ and stop
    3. $v_{m-1} \leftarrow$ vertex associated with bounded region of $S_m$
    4. $t_{m-1} \leftarrow \varnothing$
    5. for $i \leftarrow m - 1$ to 1 step-1
        6. if $v_i \neq \varnothing$ then begin      (''new'' triangle)
                7) determine with triangle of $S_i$ at $v_i$ contains $P$
                8) $v_{i-1} \leftarrow$ vertex stored at that triangle
                9) $t_{i-1} \leftarrow$ pointer stored at that triangle
           end
        10. else begin          (''old'' triangle)
                11. $v_{i-1} \leftarrow$ vertex stored at $t_i$
                12. $t_{i-1} \leftarrow$ pointer stored at $t_i$
            end
    13. look up in $S_0$ the original polygon $R_j$ to which triangle $t_0$ belongs.
    14. if $j = 0$, then return $\varnothing$, else return $j$
End
```

The set-up phase of the algorithm, steps 1–4, can be completed in constant time. Since the degree of v_i is at most 11, each iteration of the loop, steps 6–12, can require at most 12 comparisons.

Thus, the time complexity of the algorithm is linear in m, the number of times the algorithm executes the inner loop. Recall, however, that S_m has 1 triangle and 3 vertices. Consequently, $|S_m| = 3 \leq (23/24)^m |S_0| = (23/24)^m n$. Therefore, $m \leq (\log n - \log 3)/(\log 24 - \log 23)$. Using the convenience of "O" notation, we may say that $m = O(\log n)$. Thus, the algorithm performs at most 12 comparisons at each of $O(\log n)$ levels. So the search process takes $O(\log n)$ time.

To evaluate the amount of preprocessing time the algorithm requires, first let us consider the amount of time and storage required just to represent the S_i. S_0 has n vertices, three more than the actual search domain. Thus the total number of vertices (which is linear in the number of edges and regions) represented throughout the hierarchy is approximately

$$|S_0| \sum_{i=0}^{m} \left(\frac{23}{24}\right)^i < n \sum_{i=0}^{\infty} \left(\frac{23}{24}\right)^i = 24\ n.$$

Thus, this portion of the problem requires only linear time and space.

The time required to prepare the representation of S_i from that of S_{i-1} or the representation of S_0 from S depends on the complexity of the triangulation process. As we stated above, the latter process, the triangulation of S to form S_0, requires at most $O(n \log n)$ time. In the former case, the vertices to be deleted can be determined in the manner described above in time linear in $|S_{i-1}|$. When a vertex is deleted, the resulting polygon of fewer than 12 sides can be retriangulated in constant time. Thus, given S_0, creating the remainder of the S-tree (S_1, ..., S_m) takes linear time.

Given a fixed collection of N disjoint polygons having a combined total of n vertices, the hierarchical algorithm we have described creates an S-tree in time $O(n \log n)$ and space $O(n)$ which enables any subsequent query to be answered in time $O(\log n)$. Throughout the development of this algorithm, no restriction has been placed on the number of vertices per polygon or on the shapes permissible. A naive approach to the problem would require only $O(n)$ time, so it remains preferable for cases where few queries are anticipated. As the number of queries grows, however, the hierarchical approach becomes vastly superior.

In the sections below, we shall assume that a constant bounds the number of vertices per polygon, making $n = \theta(N)$. In this case, the naive approach requires only time $O(N)$ to complete a query. In the general algorithm above, n may be replaced by N in each of the complexity results. The creation of an S-tree on N polygons, an S-tree of size N, requires preprocessing time $P_S(N) = O(N \log N)$ and space $S_S(N) = O(N)$. Although an S-tree is formed only after the triangulation of the entire search domain, we may say that each of the N polygons costs an average insertion time of $I_S(N) = P_S(N)/N = O(\log N)$. A query costs $Q_S(N) = O(\log N)$ time.

2.7. Dynamic Polygon Search

An interesting addition to the polygon search problem is to allow the search domain to change over time. The S-tree is unsuitable for this problem where polygons are added one at a time with queries interspersed. As the triangulation of S cannot be modified locally, the insertion of the N^{th} polygon would entail discarding the old S-tree on $N - 1$ polygons and building a new S-tree of size N. This is likely to be less efficient than the naive approach, since inserting N polygons sequentially into any initially empty structure would cost an excessive amount of time:

$$O\left(\sum_{i=1}^{N} i \log i\right) = O(N^2 \log N).$$

Polygonal search, however, does not require that all of the data be retained in a single structure. If some of the polygons were represented in one S-tree and the

rest in another, the polygon containing P could be identified by forming the union of the results of searching the two S-trees independently. Thus instead of dismantling an entire S-tree in order to insert the N^{th} polygon, we may simply construct a new S-tree for that polygon. A subsequent query on the search domain will necessitate searching both structures in time $O(\log(N - 1) + \log 1) = O(\log N)$.

A data structure for the insertion-dynamic polygon search problem may consist of a collection of S-trees, rather than a single such tree. Nonetheless, adding a new S-tree upon each insertion does not work. Although the insertion time would remain constant, queries after N insertions to an initially empty structure would require $O(N)$ time. What is needed is a method which balances the cost of creating large S-trees against the cost of searching a large number of trees.

An insertion strategy based on binary counting, which we will call a binary transform, produces a feasible data structure for insertion-dynamic polygon search, the DPS. The first polygon to join the search domain occupies its own S-tree. Upon the insertion of the second polygon, a single S-tree of size 2 is formed. The third polygon forms its own S-tree, but the insertion of the fourth polygon prompts the formation of a single S-tree of size 4. The histogram of Fig. 2.7 illustrates the continuing pattern. A DPS of size 7 consists of three S-trees of sizes 1, 2, and 4, respectively. All three structures are replaced by a single S-tree of size 8 upon the next insertion. In general, if N is represented in binary as $(b_n b_{n-1} \ldots b_0)$, a DPS of size N will contain an S-tree of size 2^i for those i with $b_i = 1$.

A DPS allows a balance between insertion time and query time. First of all, the DPS of size N contains no more than $\log (N + 1)$ S-trees. As no S-tree has size greater than N, a search of one tree requires at most $O(\log N)$ time. Hence, a search of the entire DPS can use at most $O(\log^2 N)$ time.

Analysis of the insertion time is more difficult. Adding the 15^{th} polygon entails building an S-tree of size 1 at almost no cost, while the 16^{th} polygon necessitates the construction of new S-tree of size 16. Instead of figuring the

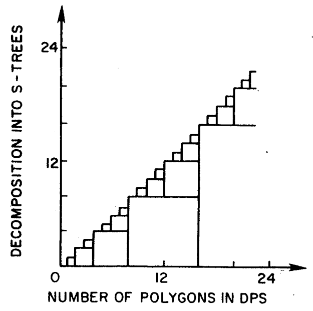


FIG. 2.7. Binary transform

insertion cost of a polygon, $I_{DPS}(16)$, as the immediate expenditure of time associated with adding it to the DPS, however, we will assign to each polygon its share of the cost of building every S-tree to which it ever belongs. Of the polygons in a DPS of size 16, the first polygon to join the search domain has belonged to the largest number of distinct S-trees: one each of sizes 1, 2, 4, 8, and 16. Its cost must bound the average insertion time. $P_S(N)$, however, grows at least linearly in N, and thus $I_S(N) = P_S(N)/N$ is monotone nondecreasing. Therefore, we can bound $I_{DPS}(16)$ as follows:

$$\begin{aligned} I_{DPS}(16) &\le I_s(1) + I_s(2) + I_s(4) + I_s(8) + I_s(16) \\ &= \sum_{i=0}^{4} I_s(2^i) \\ &\le \sum_{i=0}^{4} I_s(16) \\ &\le 5\, I_s(16). \end{aligned}$$

In general, no element of a DPS of size N will have belonged to more than $(1 + \log N)$ different S-trees over the life of the algorithm. We may conclude:

$$I_{DPS}(N) \le \sum_{i=0}^{\log N} I_s(2^i) \le (1 + \log N)\, I_s(N) = O(\log^2 N);$$

$$P_{DPS}(N) = N\, I_{DPS}(N) \le (1 + \log N)\, P_S(N) = O(N \log^2 N).$$

Thus the average insertion time is identical to the query time. As each polygon is represented in exactly one structure, the space requirements remain $O(N)$.

The creation of the DPS represents a macro applicable in other contexts. The next subsection presents a general development of this macro.

2.8. Decomposable Searching Problems

General polygon search is only one example of a class of searching problems suited to the dynamization process defined above. That process depended on the fact that the correct response to a query over a search domain D could be formed by merging the responses to separate queries over subsets of D using constant time. The problems having this attribute are called "decomposable searching problems."

Many common problems belong to this class. An element s belongs to a set $A \cup B$ if s belongs to A or s belongs to B. The nearest neighbor to a point x in a set $A \cup B$ is the closer of its nearest neighbor in A and its nearest neighbor in B. The smallest number in $A \cup B$ which is larger than a given number x is equal to the

smaller of the two numbers satisfying this criterion separately in A and in B. The number of points in a set $A \cup B$ whose x-coordinate has a given value x_0 equals the sum of the number of such points in A with the number of such points in B.

Once we have a static data structure and a static algorithm for any one of these problems, we can apply the binary transform to achieve an insertion-dynamic structure and algorithm. At any point, the insertion-dynamic algorithm will query at most $\log(N + 1)$ of the static structures, implying that $Q_D(N) \le Q_S(N) \log(N + 1)$, as long as $Q_S(N)$ is monotone nondecreasing. When we have a insertion-dynamic structure of size N, each element of that structure has belonged to at most $(1 + \log N)$ structures during the course of the algorithm. Provided that $P_S(N)$ grows at least linearly with N, each element's cost is bounded above by $(1 + \log N)\, I_S(N)$, insuring that $P_D(N) \le (1 + \log N)\, P_S(N)$. Each object is stored in only one static structure. If we assume that the static algorithm requires at least linear space, then we may conclude that the space requirements have remained unchanged: $S_D(N) \le S_S(N)$. In summary, an insertion-dynamic algorithm achieved by applying the binary transform to a static algorithm achieves the following results:

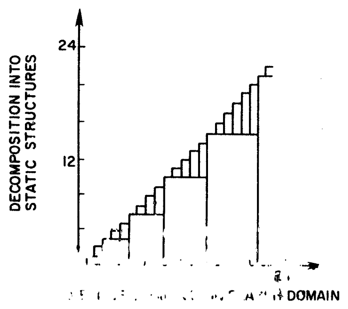
$$\begin{aligned} &Q_D(N) \le Q_S(N) \log(N + 1) \\ &P_D(N) \le P_S(N)\,(1 + \log N) \\ &S_D(N) \le S_S(N) \end{aligned}$$

Due to these performance results, we call the binary transform an admissible $(\log(N + 1), 1 + \log N)$ transform.

The chief asset of the binary transform is that it evenly assesses an $O(\log N)$ dynamization penalty to the query and average insertion times. Other transformations trade improved query time for slower overall processing, and vice versa. We will consider two examples: the triangular transform and the dual triangular transform.

The triangular transform allows the insertion-dynamic structure to contain at most two static structures at a time. Consequently, $Q_D(N) \le 2\, Q_S(N)$. To achieve this limit, it relies on the sequence of triangular numbers, $T_i = i(i + 1)/2$ for $i = 1, 2, \ldots$ If the search domain has size N, let T_j be the largest triangular number less than N. Then an insertion-dynamic data structure of size N contains one static structure of size T_j and one of size $N - T_j$. Suppose one more element is inserted. If $N + 1 = T_{j+1}$, then the previous two structures are discarded and a single structure of size T_{j+1} is created. If $N + 1 < T_{j+1}$, then just the structure of size $N - T_j$ is dismantled, and those objects together with the new one are formed into a new static structure of size $N + 1 - T_j$. [See Figure 2.8].

To assess the total processing time, first consider an example where $T_5 = 15$ elements belong to the data structure. Then, over the course of the algorithm, the first and second elements to join the search domain have belonged to five distinct structures: one each of sizes 1, 3, 6, 10, and 15. The third element has belonged only to structures of size 3, 6, 10, and 15. The fourth element, however, has



Triangular transform

FIG. 2.8. Triangular transform

belonged to structures of size 1, 2, 6, 10, and 15. We can read off this history by tracing a horizontal line through the histogram of Fig. 2.8 at a given height. Examining the histogram shows that no element has belonged to more than five structures.

In general, after $T_j = N$ elements have been inserted, each element has belonged to at most j structures. By the definition of the triangular numbers, we may conclude that $j < \sqrt{2N}$, and thus, $P_D(N) \leq P_S(N)\sqrt{2N}$. Consequently, the triangular transform may be described as an admissible $(2, \sqrt{2N})$ transform. The query time has suffered only a constant dynamization factor, while processing time is slower by a factor of $O(\sqrt{N})$, making this transform a good choice for cases where we expect a large number of queries relative to insertions.

The reverse triangular transform, or the dual triangular transform, allows each element to belong to at most two static structures over the life of the algorithm. As each of these structures must have size less than or equal to N, $I_D(N) \leq 2\, I_S(N)$ and $P_D(N) \leq 2\, P_S(N)$. If the search domain contains $N = T_j$ objects for some j, then the data structure contains one structure of size i for all $1 \leq i \leq j$. If $N > T_j$, then the structure contains one structure of size i for all $1 \leq i \leq j$ in addition to $N - T_j$ $(\leq j)$ structures of size 1 (see Fig. 2.9). Thus, at any time, the insertion-dynamic structure contains at most $2j < 2\sqrt{2N}$ structures, ensuring that $Q_D(N) \leq Q_S(N)\, 2\sqrt{2N}$. In summary, the dual triangular transform is an admissible $(2\sqrt{2N}, 2)$ transform.

The three transforms we have discussed have distinct advantages and disadvantages. The binary transform is best in cases where a comparable number of insertions and queries are expected. The triangular transform and the dual triangular transform are best, respectively, when a greater proportion of queries or a greater proportion of insertions are expected. There are an unlimited number of other transforms available. For example, in both triangular transforms, the number 2 was arbitrarily chosen as the limit for the number of active structures and as the limit on the number of structures per single element over time. The choice of the number 3 instead, or any other integer, would have led to different

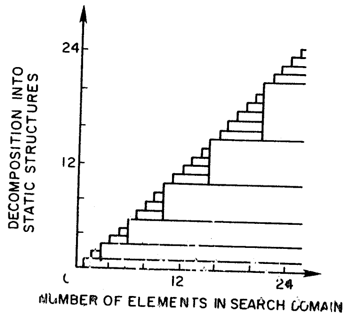


FIG. 2.9. Dual triangular transform

Dual triangular transform

transforms. In fact, we may have a transform isomorphic to any counting scheme.

These transforms may be applied to any decomposable searching problem, and the worst case complexity results we have derived will always hold. For some specific problems, however, the dynamization penalties will not in fact accrue. It may be possible, for example, to merge some of the static structures without completely dismantling them. Similarly, a tighter analysis of the query time may be possible. Nonetheless, the upper bounds readily available as a result of using a standard transform have great value. It is impossible, however, to create efficient transforms applicable to all decomposable searching problems which will convert static algorithms to dynamic algorithms allowing deletions.

2.9. Notes

Garey, Johnson, Preparata, and Tarjan (1979) produced an algorithm for triangulating monotone polygons in linear time. Chazelle and Incerpi (1984) have developed a new scheme for decomposing a simple polygon into a collection of monotone polygons. A modification of that approach leads to an $O(n \log n)$ algorithm for decomposing all of the faces of a planar subdivision on n vertices into monotone polygons (Chazelle, 1984b).

The problem solved here by hierarchical search was first stated by Knuth (1973). A first solution was proposed in Dobkin and Lipton (1976). This paper introduced the notion of x-axis and y-axis projection which have proved useful in other contexts (Bentley & Ottmann, 1979; Brown, 1981; Shamos & Hoey, 1976). The topic of planar subdivision search was extended by Preparata, 1981; Lipton & Tarjan, 1979; Lipton & Tarjan, 1980; Kirkpatrick, 1983. Our development is most similar to that of Kirkpatrick (1983). A similar technique was used in (Dobkin & Kirkpatrick, 1985).

The dynamization techniques and the classification of decomposable searching problems are due to Bentley and Saxe (Bentley, 1979; Bentley & Saxe, 1980). Some of the methodology introduced here is also seen in the order-decomposable methods mentioned in the next section.

3. HIERARCHICAL COMPUTATION

3.1. Divide-and-Conquer Computation

The most powerful technique in algorithm design is the divide-and-conquer technique, as it can be applied to problems of vastly different structures. The sorting of natural numbers is a classic example of the divide-and-conquer procedure. The goal is to sort a set of n integers. To do so, we divide the original set into two equal or nearly equal subsets. Each subset is sorted (by applying the method recursively) with the results (two sorted sets) merged to form a sorted sequence describing the entire set.

A simple example involves producing an ordered list of grades from an alphabetized list of students and their averages. A divide-and-conquer tree begins as a binary search tree, with the original objects ordered in the leaves and with each non-leaf node containing a pointer to the largest element in its left-subtree (see Fig. 2.10). In binary search, however, we begin at the root and descend the tree, considering one node at each level and deciding to progress to either the right or the left subtree. The leaf reached determines the solution. A divide-and-conquer algorithm involves both descending and ascending the tree, visiting each node in each direction. The division is done as we pass down the tree, distinguishing different levels in the hierarchy. For any nonleaf node, the algorithm sorts its left subtree and then sorts its right subtree. The conquering part of the computation is done as we pass up the tree. The algorithm merges the sorted sequences stored at the children and stores the result of the computation at the given node. At the conclusion, the complete sorted sequence is stored at the root.

We may choose how much space we wish to utilize during the course of this algorithm. When we merge two sequences, we create new storage for the result. We may either retain the original sequences or destroy them. In this first case,

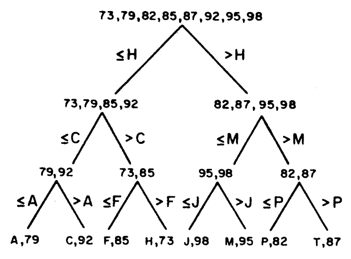


FIG. 2.10. Sample divide-and-conquer tree

the algorithm will need $O(n \log n)$ space, whereas the second case uses only linear space. Investing the extra space allows easy retrieval of previous states of the computation. In some applications, the ability to retrace previous steps easily has enough significance to outweigh the cost in space. In general, however, the linear-space approach is preferable, as it is here.

The efficiency of the algorithm depends on the amount of work required at the leaves of the tree, the complexity of the merge phase, and the overall size of the tree. Typically, this yields a recurrence of the form $T(n) = 2\,T(n/2) + M(n)$ with $T(1) = C$. In this case, no work is necessary at the leaves ($T(1) = 0$), and the merge algorithm is linear in the size of the subtree: $M(n) = n - 1$. Thus $T(n) = 2\,T(n/2) + n - 1 = O(n \log n)$.

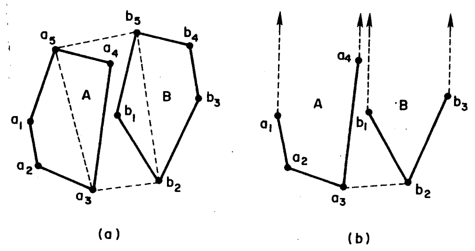
Having briefly described the divide-and-conquer method, we now consider its specialization to the hierarchical methods best suited to geometric problems. Not only do they apply to static geometric problems, but they also serve as a basis for dynamization techniques which apply to a large subclass of geometric problems.

3.2. Two-dimensional Convex Hulls

Over the years, numerous algorithms have been developed for the computation of the convex hull of a set of N points in the plane (see Chapter 5). The algorithm we describe here demonstrates the hierarchical strategy of computing. If the points have been sorted according to the value of their x-coordinate, the set can be divided in two at the median value. The division step now involves computing the hulls of the left and right collections of points. Conquering involves merging the left and right hulls.

Merging two disjoint convex hulls requires determining two distinct segments which are tangent to both hulls and then deleting all vertices and edges within the quadrilateral defined by those tangent segments (see Fig. 2.11a). To simplify our exposition, we consider only the problem of finding the lower tangent and maintaining the "bottom convex hull" or "bc-hull." The bc-hull contains the bottom part of the convex hull, the portion extending from the leftmost vertex to the rightmost vertex in the counter-clockwise direction. The bc-hull, however, augments the bottom part of the hull with vertical rays in the positive direction at

FIG. 2.11. Merging two convex hulls versus merging two bc-hulls



its two endpoints.[3] (See Fig. 2.11b.) Once two bc-hulls have been merged and the corresponding two top convex hulls (tc-hulls) have been merged, the complete convex hull can be formed in constant time.

Given two bc-hulls A and B, each consisting of N vertices with A lying to the left of B, we divide the merge process into three parts. First, we seek a "bridge," i.e., a segment tangent to both A and B. Any vertex on either hull could be an endpoint of the bridge, or a "bridge point." A variant of binary search is used to locate the actual bridge points. Once the bridge points a and b have been identified, we split A at a and B at b. Finally, we form a new hull by concatenating the left portion of A, including a, and the right portion of B, including b. In the chain of vertices representing the new hull, a and b are adjacent to each other, indicating the bridge segment.

The vertices of each bc-hull, C, are represented in the leaves of a concatenable queue. The term "concatenable queue" denotes a binary search tree which has been implemented in such a way as to enable efficient searching, splitting and concatenating. The concatenable queue, or Q-structure, retains all of the features of a binary search tree; each nonleaf node α still contains a pointer, V[α], to that vertex in its left subtree having the largest x-coordinate. Thus the middle vertex of the portion of a bc-hull represented by a single subtree can be identified in constant time. We associate with Q_c, the Q-structure representing the bc-hull C, two additional values: M_c stores the maximum x-coordinate of any vertex of C; m_c contains the minimum x-coordinate.

The search process begins by assigning to a and b the vertices represented at the roots of Q_A and Q_B, the middle vertices of the respective hulls. Define $\overrightarrow{a_r}$ and $\overrightarrow{b_r}$ as the rays extending from a and b rightward toward the next vertex on the hull. Similarly, $\overrightarrow{a_l}$ and $\overrightarrow{b_l}$ represent the rays extending from a and b leftward toward the previous vertex on the hull. The angles formed at a and at b guide our search. Until a bridge is found, we repeatedly remove a fraction of the vertices of one or both hulls from consideration as a bridge endpoint. The size of the four angles, $\angle(\overrightarrow{ab}, \overrightarrow{a_l})$, $\angle(\overrightarrow{ab}, \overrightarrow{a_r})$, $\angle(\overrightarrow{b_r}, \overrightarrow{ba})$, $\angle(\overrightarrow{b_l}, \overrightarrow{ba})$, determine whether $\overline{ab}$ is itself the bridge and, if not, which vertices may be removed.[4]

We consider four cases determined by the presence or absence of reflex angles. If there are no reflex angles (Fig. 2.12a), then the segment $\overline{ab}$ is itself the bridge and the search is finished. The second category includes all cases where one or both of $\angle(\overrightarrow{ab}, \overrightarrow{a_l})$ and $\angle(\overrightarrow{b_r}, \overrightarrow{ba})$ is reflex. If $\angle(\overrightarrow{ab}, \overrightarrow{a_l})$ is reflex, then neither a nor any vertex to the right of a can be a bridge point. Thus the bridge point must lie in the left subtree of Q_A (see Fig. 2.12b). If $\angle(\overrightarrow{b_r}, \overrightarrow{ba})$ is reflex, then both b and all vertices of B left of b can be removed from future consideration and the bridge point must lie in the right subtree of Q_B (see Fig. 2.12c).

[3]Alternately, we can say that the bc-hull of a set P is defined as the convex hull of the union of P with the point $(0, \infty)$. The tc-hull is the convex hull of $P \cup \{(0, -\infty)\}$.

[4]All angles are measured in the counter-clockwise direction from the first ray to the second.

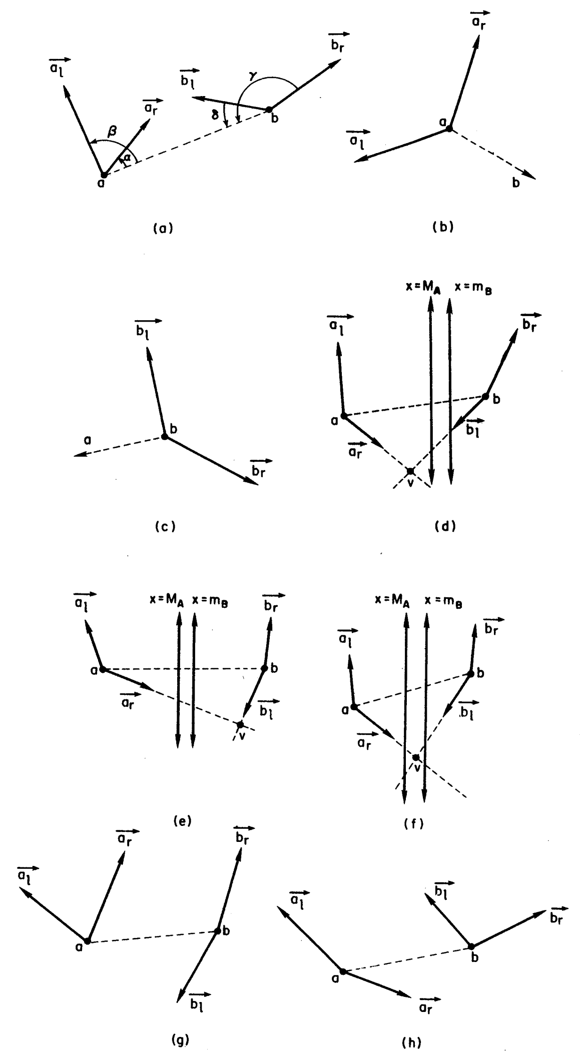


FIG. 2.12. Case analysis for merging bc-hull A with bc-hull B

The third category involves both $\angle(\overrightarrow{ab}, \overrightarrow{a_r})$ and $\angle(\overrightarrow{b_l}, \overrightarrow{ba})$ being reflex. In this case, we determine the intersection point, $v = (x, y)$, of a_r and b_l. We let M_A represent the maximum x-coordinate of A, and let m_B represent the minimum x-coordinate of B. If $x < m_B$, then no vertex of B could lie below $\overrightarrow{a_r}$. Consequently, neither a nor any point left of a could be a bridge point and they can be removed from future consideration (see Fig. 2.12d). Similarly, if $x > M_A$, then no vertex of A may lie below $\overrightarrow{b_l}$ and so b and all vertices to the right of b can be rejected (see Fig. 2.12e). If $M_A < x < m_B$, then we may delete both sets (see Fig. 2.12f).

In the final category, only one of $\angle(\overrightarrow{ab}, \overrightarrow{a_r})$ and $\angle(\overrightarrow{b_l}, \overrightarrow{ba})$ is reflex. If $\angle(\overrightarrow{b_l}, \overrightarrow{ba})$ is reflex, $\overrightarrow{ab}$ forms an angle smaller than 180° with both $\overrightarrow{a_l}$ and $\overrightarrow{a_r}$ and thus a remains a potential bridge point. In other words, no point of A lies below $\overleftrightarrow{ab}$. But since $\overrightarrow{ba}$ is interior to the angle from $\overrightarrow{b_r}$ to $\overrightarrow{b_l}$, neither b nor any point right of b can lie on the bridge (see Fig. 2.12g). If $\angle(\overrightarrow{ab}, \overrightarrow{a_r})$ is reflex, then neither a nor any point left of a can be a bridge point (see Fig. 2.12h).

In each of the last three categories, we cut in half the number of potential bridge points on at least one of the bc-hulls. We may repeatedly set a and b to the midpoints of the remaining vertices of A and B, respectively, by moving to the leftson or rightson of the current node in Q_A or Q_B. We then rerun the algorithm. Since the height of both Q_A and Q_B is $O(\log N)$, at most $O(\log N)$ iterations are required.

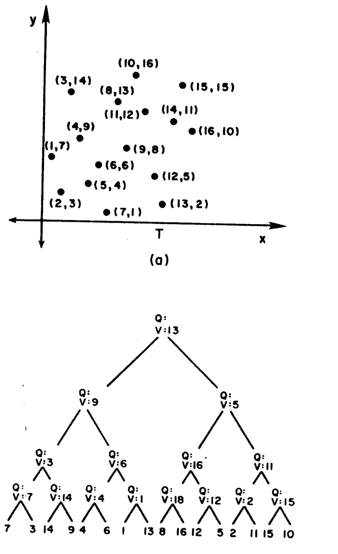
We may formalize the algorithm as follows:

```
MERGE (A,B,C)
Begin
   1. $\alpha \leftarrow$ the root of $Q_A$
   2. $\beta \leftarrow$ the root of $Q_B$
   3. done $\leftarrow 0$
  repeat
    begin
    4. $a \leftarrow V|\alpha|$
    5. $b \leftarrow V|\beta|$
    6. determine which of $\angle(\overrightarrow{ab},\overrightarrow{a_l}),\angle(\overrightarrow{ab},\overrightarrow{a_r}),\angle(\overrightarrow{b_r},\overrightarrow{ba}),\angle(\overrightarrow{b_l},\overrightarrow{ba})$, is reflex
    7. if none are, then done $\leftarrow 1$
    8. else if $\angle(\overrightarrow{ab},a_l)$ or $\angle(\overrightarrow{b_r},\overrightarrow{ba})$ is reflex, then
       begin
       9. if $\angle(\overrightarrow{ab},\overrightarrow{a_l})$ is reflex, then $\alpha \leftarrow$ leftson($\alpha$)
       10. if $(\overrightarrow{b_r}\overrightarrow{ba})$ is reflex, then $\beta \leftarrow$ rightson($\beta$)
       end
    11. else if $\angle(\overrightarrow{ab},\overrightarrow{a_r})$ and $\angle(b_l\overrightarrow{ba})$ are reflex, then
       begin
       12. $(x, y) \leftarrow$ the intersection of $\overrightarrow{a_r}$ and $\overrightarrow{b_l}$
       13. if $x < m_B$, then $\alpha \leftarrow$ leftson($\alpha$)
       14. if $x > M_A$, then $\beta \leftarrow$ rightson($\beta$)
       end
    15. else if either $\angle(\overrightarrow{ab},\overrightarrow{a_r})$ or $\angle(\overrightarrow{b},_l\overrightarrow{ba})$ is reflex, then
       begin
       16. if $(\overrightarrow{ab},\overrightarrow{a_r})$ is reflex, then $\alpha \leftarrow$ rightson($\alpha$)
       17. if $\angle(\overrightarrow{b_l},\overrightarrow{ba})$ is relfex, then $\beta \leftarrow$ leftson($\beta$)
    end
  until done
```

```
   18. $Q_C \leftarrow$ concatenate(leftsplit($Q_A$,a), rightsplit($Q_B$,b))
   19. $M_C \leftarrow M_B$
   20. $m_C \leftarrow m_A$
End
```

The set-up phase of the algorithm, steps 1–3, the interior of the loop, steps 4–17, and the final two steps each require constant time. As the loop is iterated $O(\log N)$ times, the algorithm through step 17 requires only $O(\log N)$time. Step 18 requires a choice similar to the one we mentioned in section 3.1. We may leave Q_A and Q_B intact and merely copy the left portion of Q_A and the right portion of Q_B before concatenating the two sections. As the copying itself requires $O(N)$ time, the merge algorithm as a whole requires linear time. Alternatively, we may split the actual representations of Q_A and Q_B and then concatenate the appropriate sections. In this case, we lose the ability to recapture immediately the previous states of the computation, but step 18 now requires only $O(\log N)$ time, preserving an $O(\log n)$ worst-case complexity for the complete merge algorithm.

Having fully detailed the merge process, we return to the hierarchical algorithm as a whole. The entire process can be modeled by a divide-and-conquer tree, a T-tree. Suppose we wish to determine the bc-hull of the set of points, P, depicted in Fig. 2.13a. First, we sort the points in ascending order of x-coordi-



Beginning of divide-and-conquer tree
(b)

FIG. 2.13. An instance of the convex hull problem

nate. This example includes one point for every x-coordinate from 1 to 16 inclusive, and henceforth each point will be identified by its y-coordinate. The points are incorporated into the leaves of a balanced binary search tree (e.g., a BB[a]-tree). Mirroring the binary search tree, each nonleaf node a includes a pointer $V[a]$ to the rightmost vertex of its left subtree. The T-tree for P, T_P, is formed by augmenting each nonleaf node, a, with a Q-structure, Q_a, which represents the bc-hull of all points in the subtree at a (see Fig. 2.13b).

The Q-structures are formed gradually as we move up the tree. At the bottom level, the tree induces a natural pairing of points. The bc-hull of a pair of points consists of a segment between them with a positive vertical ray added at each endpoint (see Fig. 2.14a). The Q-structures on the first level of the tree describe the bc-hulls of these pairs. At each succeeding level of the tree, sibling bc-hulls are merged according to the process described above to form the Q-structure at the parent node (see Figs. 2.14b-e). Figure 2.14f represents T_P at the termination of the algorithm. The root contains the bc-hull of the entire set P. We can then repeat the same process to form the tc-hull. The merger of these hulls gives the convex hull of the original set of points.

For a point set of size n, this algorithm requires $\theta(n \log n)$ preprocessing time to sort the points. The recurrence relation, $T(n) = 2\,T(n/2) + M(n)$ describes the time complexity of the remainder of the algorithm. The bc-hull of a set of $n/2$ points must have no more than $n/2$ vertices. Thus, when the partial bc-hulls remain intact, each level of the T-tree requires $O(n)$ storage and the merge technique uses linear time. Consequently, the overall algorithm will run in $\theta(n \log n)$ time, using $O(n \log n)$ space. If we choose not to preserve the partial bc-hulls, the final T-tree contains only the complete bc-hull. At intermediate stages in the algorithm, the T-tree contains only the bc-hulls of disjoint sets. Consequently, the algorithm uses only linear space. Each merge requires only logarithmic time. Consequently,

$$T(n) = 2\,T\left(\frac{n}{2}\right) + M(n) = \sum_{i=1}^{\log n} \frac{n}{2^i} \log 2^i = O(n)$$

In order words, after the initial sorting this algorithm will require only linear time. Since the problem of finding convex hulls corresponds to sorting and thus any algorithm must require $\Omega(n \log n)$ time, either algorithm can be considered optimal.

3.3. Dynamic Convex Hulls

In the previous example, we determined the bc-hull of a fixed set of points. In many applications, however, we may wish to maintain the bc-hull of a set which allows insertions and deletions. In other situations, balanced binary search tree accommodates insertions and deletions easily. The appropriate location is found

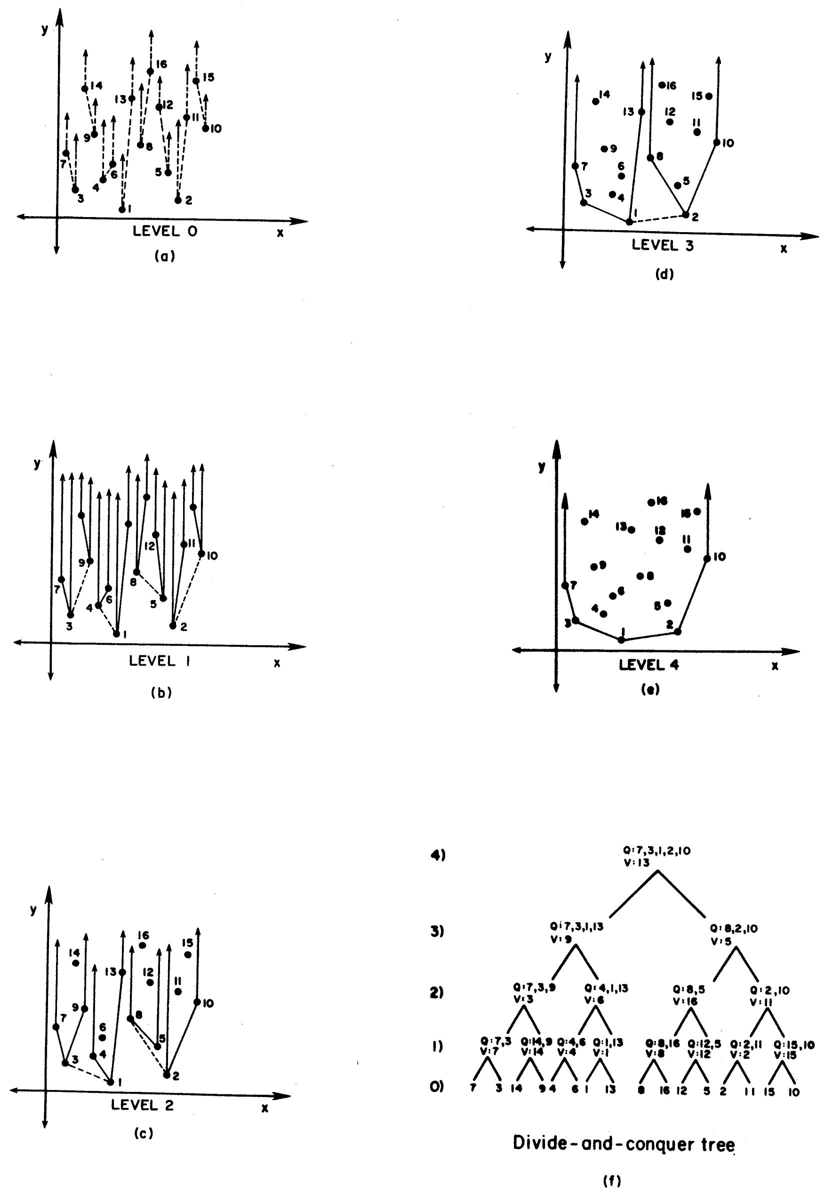


FIG. 2.14. Building the bc-hull of the instance in Fig. 2.13

by beginning at the root and descending the tree until the relevant location is found. After inserting or deleting the point, the path to the root is retraced, rebalancing the tree and updating any pointers as necessary. The entire process requires $O(\log n)$ time. This case is more complicated.

Suppose that we have the complete T-tree for the original set of points and that all partial hulls have been retained. After locating the appropriate leaf position in the T-tree for the insertion or deletion, we can rebuild the be-hull on level 1 in constant time.. Its sibling hull remains unchanged, so we can merge the two hulls to form the parent Q-structure. At each node as we ascend the tree, the sibling hull is already available. Provided no rebalancing is necessary, a single merger will move us one level higher in the tree. Suppose we arrive at a node and determine that rebalancing is required. A BB(α)-tree can be rebalanced using no more than two local rotations per node as we ascend the tree. To update the Q-structures following these rotations, we will have to reform at most two of the bc-hulls by merging some of the partial bc-hulls in a new order. Nonetheless, only a constant number of merges will be required at each node. Each merge requires time linear in the size of the subtree.

Therefore, the total insertion/deletion time can be $ID(n) = O\left(\sum_{i=1}^{\log n} \frac{n}{2^i}\right) = O(n)$, which is too large to be considered efficient.

Suppose that in building the original T-tree we had chosen to conserve space and had thus destroyed the partial hulls. After each insertion or deletion, we would then have to process the entire tree from level 0 to the root, again using $O(n)$ time.

We must try a new approach. In a new T^* structure, the new Q-structure at a node α will be formed from the actual portions of the children Q-structures, Q_γ and Q_δ, using logarithmic time. At α however, we retain a pointer, $B[\alpha]$, to the position of the left bridge point in Q_γ. In addition, we associate with the nodes γ and δ the remaining fragments of Q_γ and Q_δ, denoted $Q_\gamma{}^*$ and $Q_\delta{}^*$ (see Fig. 2.15). Consequently, we could split Q_α after the left bridge point, as indicated by $B[\alpha]$, and reform the complete Q_γ and Q_δ in logarithmic time.

We can modify the insert/delete algorithm as follows: begin at the root and split the current bc-hull and form the bc-hulls of its two children; compare the desired point against the values of V at the root and move to the left or right son accordingly. At each interior node, again split the Q-structure and form the Q-structure of its children before moving to one child or the other. The descent will require at most $O(\log n)$ time at each node, or $O(\log^2 n)$ time overall. In the ascent, we will perform at most a constant number of merges at each node, using only $O(\log n)$ time as we do not have to preserve entire partial hulls. Thus the ascent also requires at most $O(\log^2 n)$ time. Consequently, we can maintain the bc-hull of a set of n points at a cost of $O(\log^2 n)$ per insertion/deletion.

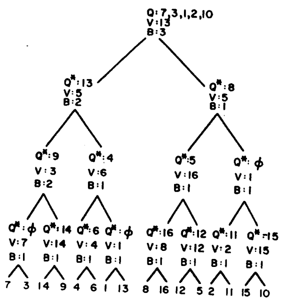


FIG. 2.15. Dynamic T-tree

3.4. Order Decomposable Problems

The two dimensional bc-hull problem is a single example of a class of set problems suited to the dynamization process defined above. That process entailed ordering the set and then forming the solution over the entire set by merging the solutions of the first i and the last $n - i$ elements, for any $1 \leq i \leq n$, in time $M(n)$. This suggests the name "$M(n)$-order decomposable" for any set problem having an $M(n)$ merge algorithm dependent on some ordering scheme, *ORD*. A static divide-and-conquer algorithm takes time $O(ORD(n) + T(n))$ where $T(n) = 2\,T(n/2) + M(n)$.

A number of common problems belong to this class. Suppose we wish to compute the maximal elements of the planar set $A \cup B$ where all points a_i in A are left of all points b_j of B. Recall that p is a maximal element of a set C iff for all $q \in C$, either $x_p > x_q$ or $y_p > y_q$. $MAX(C)$ represents the sequence of maximal elements of C in decreasing order of y-coordinate and in ascending order of x-coordinate. Figure 2.16a highlights the maximal elements within the sample sets A and B by linking them together with a polygonal chain of horizontal and vertical segments. Let y_B represent the maximum y-coordinate in B. Then $MAX(A \cup B) = concatenate(\{p \in MAX(A) | y_p > y_B\}, MAX(B))$. Splitting $MAX(A)$ at the appropriate spot and concatenating both require $O(\log n)$ time. Consequently, the two-dimensional maximal element problem is $O(\log n)$-order decomposable.

Consider the problem of determining the intersection of a collection of n lower half-planes: for $1 \leq i \leq n$, a line l_i: $y = a_i x + b_i$ bounds the half-plane $H_i = \{(x, y) | \, y < a_i x + b_i\}$ from above. Sort the half-planes according to the slopes of the bounding lines. Suppose we have the border A of the intersection region defined by the first i half-planes and the border B of the region defined by the last $n - i$ (see Fig. 2.16b). Find the intersection point of the two contours in time $O(\log n)$. Split them there, and concatenate the front piece of the first with the

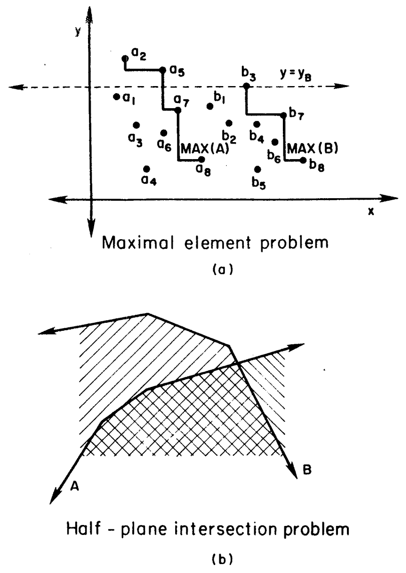


FIG. 2.16. Other examples of order-decomposable problems

tail piece of the second, also using $O(\log n)$ time. Consequently, the lower half-plane intersection problem is $O(\log n)$-order decomposable.

In either case, a dynamic algorithm operates on a balanced T-tree. The leaves contain the ordered elements of the set, and each nonleaf node α contains a pointer $V[\alpha]$ to the largest element in its left subtree. At each nonleaf node, we compute the solution over its subtree. At a node α with children γ and δ, we form Q_α from Q_γ and Q_δ using $O(M(n))$ time. As we merge the two children Q-structures, we will store a string of information describing the steps we take as $B[\alpha]$, a tag at node α. As the merge process itself took only $O(M(n))$ time, writing a string which describes that process cannot require more than $O(M(n))$ time or $O(M(n))$ space. The unused pieces of Q_γ and Q_δ are stored at γ and δ as $Q_\gamma{}^*$ and $Q_\delta{}^*$, respectively. Only the root will retain a complete Q-structure.

To insert or delete an object, we locate the appropriate leaf using binary search. At every node α on the path from the root to the leaf, we use the information stored in $B[\alpha]$ to split the Q-structure and recreate the complete Q-structures at the children, using only as much time as the original merge. Thus the process of descending the tree requires time

$$DOWN(n) = O(M(n) + M(n/2) + M(n/4) + \ldots) = O\left(\sum_{i=0}^{\lfloor \log n \rfloor} M\left(\frac{n}{2^i}\right)\right).$$

If $M(n) = \Omega(n^\epsilon)$ for $\epsilon > 0$, then $DOWN(n) = O(M(n))$. Otherwise, $DOWN(n) = O(M(n) \log n)$. After updating level 0 of the tree, we ascend the tree reforming Q-structures on the path from the leaf to the root. Even with rebalancing, only a

constant number of merges will be performed at each level. Consequently, $UP(n) = O(DOWN(n))$.

We may conclude that any $M(n)$-order decomposable problem has a dynamic algorithm which accommodates both insertions and deletions in time $ID(n) = O(M(n))$, if $M(n)$, $= \Omega(n^\epsilon)$ for $\epsilon > 0$, and in time $ID(n) = O(M(n) \log n)$ otherwise. In other words, whenever there is a "cheap" merging algorithm for a set problem, the problem can always be dynamized efficiently.

3.5. Notes

Divide-and-conquer techniques are as prevalent in computational geometry as they are elsewhere in the design of algorithms. A common data structure which derives from this approach is the Voronoi diagram which is used for closest point problems (see Chapter 3).

The first convex hull algorithm was proposed by Graham (1972). His method is simple, but does not easily allow dynamization. We follow the development of Preparata and Hong (1977) here. This algorithm also extends to three dimensions. The problem of computing hulls has been widely studied (see Chapter 5).

Overmars and van Leeuvwen (Overmars & van Leeuwen, 1981; van Leeuwen & Overmars, 1981) were the first to consider methods of computing dynamic convex hulls. Their work also led to the generalization to order-decomposable problems (Overmars, 1981, 1983). Our development here follows theirs. Similar techniques are used in (Dobkin & Munro, 1985). For details on the use of the BB(α) data structure, see (Reingold, Nievergelt, & Deo, 1977).

4. GEOMETRIC TRANSFORMATIONS

4.1. Introduction

In this section, we discuss methods of transformation. Typically, transformations change geometric objects into other geometric objects (e.g., take points into lines) while preserving relations which held between the original objects (e.g., order or whether they intersected). A number of geometric problems are best solved through the use of transformations. The standard scheme is to transform the objects under consideration, solve a simpler problem on the transformed objects, and then use that solution to solve the original problem. No single transformation applies in all cases; a number of different transformations have been used effectively. Here, we describe three commonly used transformations in 2-space and in 3-space and demonstrate their applications.

4.2. Mathematical Background

The domain of all of our two-dimensional transformations will be the projective plane. The projective plane is an enhanced version of the Euclidean plane in which each pair of lines intersects. The projective plane contains all points of the Euclidean plane (the "proper points"). We introduce a set of "improper points" with one point associated with every direction in the plane. Two parallel lines, then, intersect at that "improper point" indicated by the direction of the parallel lines (this can be thought of as a point at infinity). All "improper points" are considered to lie on the same line: the "improper line" or the "line at infinity." Thus, any two lines in the projective plane intersect at exactly one point: two nonparallel proper lines intersect at a proper point (i.e., one of the Euclidean plane); two parallel proper lines intersect at the improper point bearing the same direction; and a proper line intersects the improper line at the improper point defining the direction of the proper line. Likewise, between every two points passes exactly one line: There is a proper line passing through every pair of proper points; a proper line passes through a given improper point and a given proper point; and the improper line passes through any two improper points.

In three dimensions, projective space will be obtained similarly. All points in E^3 are called "proper points." Define one "improper point" for every direction in space. Collectively, the "improper points" form the "improper plane." Therefore, each proper plane will intersect the improper plane in an improper line; a pair of parallel proper planes intersect in an improper line; and a pair of nonparallel proper planes intersect in a proper line. Given any two lines, either they are skew or else they intersect in a single point. Any line and any point not on the line define a plane.

In general, the actual algorithms used to solve problems rely solely on Euclidean geometry. Therefore, although all the transformations will map the projective plane (space) onto itself, we will wish to choose a transformation which maps the objects under consideration to "proper" objects. Thus, the parameters of the original problem will dictate which transformations are appropriate. Throughout the section, we will use the following notation: The images of a point V, a line h, and a triangle XYZ under a transformation B will be denoted by V_B, h_B, and XYZ_B, respectively.

4.3. Point/Line Duality I

The appropriate transformation is selected based on relationships among the original objects and the given problem to be solved. Our choice is guided by the properties to be preserved under the transformation.

We first return to the two-dimensional problem of determining the intersection of a collection of N lower half-planes where for $1 \le i \le N$, a line $l_i : y = a_i x + b_i$ bounds the half-plane $H_i = \{(x, y) \mid y < a_i x + b_i\}$ from above. A redundant half-plane is defined as one which contains the entire intersection region in its interior. One approach to the problem consists of identifying and eliminating the redundant half-planes. For each remaining half-plane H_i, a portion of l_i lies on the boundary of the intersection region. To determine the boundary, we then sort

the l_i in descending order of slope in $O(n \log n)$ time; determine the intersection points between pairs of adjacent lines in linear time; now the boundary consists of the left half of the first line up to its intersection with the second line, the segments from one intersection point to the next, and the half of the final line extending to the right from its intersection with the second to last line.

To complete the algorithm, we must define a process for determining redundant half-planes. A half-plane H_i is redundant iff there are half-planes H_j and H_k such that the slopes satisfy the inequality $a_j < a_i < a_k$ and such that l_i lies above the intersection point of l_j and l_k We say that the pair l_j, l_k forms a certificate of redundancy for l_i. Given a collection of N half-planes, the naive method of deciding the redundancy of a given half-plane entails comparing its bounding line against all other pairs of lines in time $\theta(N^2)$. Determining all redundancies requires $\theta(N^3)$.

Clearly, a transformation which allows for efficient identification of redundant half-planes must in some way preserve both the above/below relationship and the concept of slope. We show that the transformation T which maps the point (a, b) to the line $y = ax + b$ satisfies these criteria. To do so, we must first determine the image of a line l: $y = cx + d$ under the transformation T. As defined above, T maps the points $(n, cn + d)$ and $(m, cm + d)$, both of which lie on l, to the lines $y = nx + cn + d$ and $y = mx + cm + d$, respectively. Both of these lines pass through the point $P = (-c, d)$ (see Fig. 2.17). In other words, T maps the set of all points lying on l to the set of all lines passing through P, or the "pencil of lines" through P. More simply, we just say that T maps $y = cx + d$ to the point $(-c, d)$.

That T maps $y = cx + d$ to the point $(-c, d)$ rather than to the point (c, d) destroys the symmetry of the transformation. We prefer transformations which are self-inverting. Nonetheless, T remains valuable because it preserves both the above/below relationship and the concept of slope. First of all, the negative of the slope of a line survives as the x-coordinate of the image point. Consequently, if the slope of l exceeds the slope of k, then the point l_T lies to the left of k_T. Secondly, the vertical distance from $P = (a, b)$ to the line l: $y = cx + d$ is defined as the distance from P to the point $(a, ac + d)$ which lies on l, $b - (ac + d)$. The

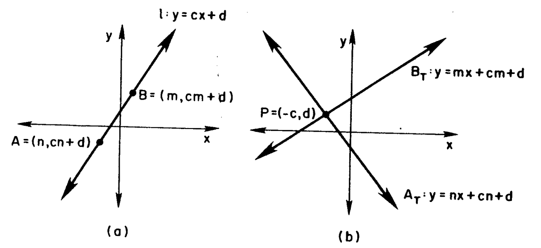


FIG. 2.17. A line (a) and its dual (b) under the transformation T

ALTERNATIVE: $(a,b) \longrightarrow y = 2ax - b$ self-dual

$y = cx + d \longrightarrow (\frac{c}{2}, -d)$

vertical distance from the line $P_T: y = ax + b$ to the point $l_T = (-c, d)$ is defined as the distance from the point $(-c, b - ac)$ lying on P_T to the point l_T, $(-ac + b) - d$. The two quantities are identical in both sign and magnitude.

Before applying T to the half-plane intersection problem, we must determine the image of a vertical line under T. Each point (m, a) on the line $x = m$ is mapped to a line $y = mx + a$, a line with slope m. All lines with slope m, however, intersect at the improper point P_m associated with m. Thus, T maps the vertical line $x = m$ to the pencil of lines through P_m, or just to P_m. Conversely, each line $y = mx + a$ which passes through P_m is mapped to a point $(-m, a)$ which lies on the line $x = -m$. Consequently, the image of P_m under T corresponds to $x = -m$. In general, we prefer to ignore improper points. Whenever a given problem involves vertical lines, we either choose to use a transformation other than T or we rotate the object space slightly until no vertical lines remain.

The transformation T preserves the notion of slope and vertical distance and thus can help to identify the redundant half-planes within a collection where none is bounded by a vertical line. Consider the collection of lower half-planes bounded above by the lines h, j, k, l, m, and n and their images or "duals," under T: h_T, j_T, k_T, l_T, m_T, and n_T (see Figs. 2.18a and b). Portions of m, k, h, and l together form the boundary of the intersection region R. Thus, these four lines cannot correspond to redundant half-planes. The line j does correspond to a redundant half-plane: the slope of j has a value between the slopes of k and h, and j lies above the intersection point of k and h. Consequently, j_T must lie between $\overleftrightarrow{\text{the ho}}$rizontal values of k_T and h_T. in addition, j_T must lie above the line $\overleftrightarrow{k_T\text{H}}$ T the dual of the intersection point of k and h. Similarly, since h, l, forms a certificate of redundancy $\overleftrightarrow{\text{for}}$ n, n_T must lie between the horizontal values of h_T and l_T and above the line $\overleftrightarrow{h_T l_T}$.

We may conclude that a line p in the object plane has certificate of redundancy q, r if, and only if p_T lies directly above the line segment $\overline{q_T r_T}$ in the image plane. A line p which has no such certificate corresponds to a significant half-plane. The only points p_T which lie above no segment $\overline{q_T r_T}$ are those points on

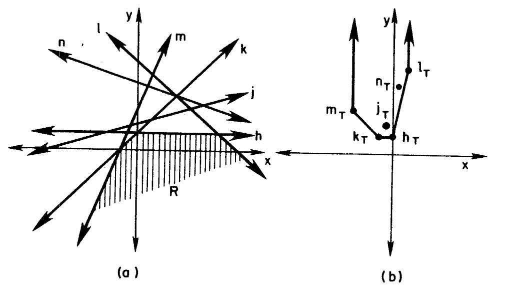


FIG. 2.18. Computing the intersection of a set of lower half-planes (a) by finding the bc-hull of the dual set of points (b)

the bottom convex hull, or bc-hull, of the set of all image points. In the example of Fig. 2.18a, we observe that m_T, k_T, h_T and l_T form the bc-hull, while j_T and n_T lie the interior.

The transformation T enables us to distinguish the significant and redundant half-planes in a collection of size N in time $O(N \log N)$: finding the dual of each bounding line takes time $O(N)$; determining the bc-hull of the daul points takes time $O(N \log N)$. Those half-planes corresponding to points on the bc-hull are significant. The half-planes corresponding to points in the interior are redundant.

We note that our algorithm for using a geometric transformation to solve the half-plane intersection problem consisted of three parts. We first identified the geometric techniques we might use (in this case redundancy). Next, we identified the invariants required by a transformation (notion of above/below and slope). Finally, we found an appropriate transformation and solved the problem. This is a classic example of how geometric transformations are used.

Had we wished to solve the lower half-space intersection problem, we could have used a three dimensional version of T:

$$(a,b,c) \xrightarrow{T} z = ax + by + c$$

$$z = ax + by + c \xrightarrow{T} (-a, -b, c)$$

ALTERNATIVE

$(a, b, c) \longrightarrow z = 2ax + 2by - c$

$z = ax + by + c \rightarrow \left(\frac{a}{2}, \frac{b}{2}, -c\right)$

self-dual

Here, T still preserves vertical distance, distance with respect to the x-coordinate, as well as the directional cosines, which represent the rates at which z changes with respect to x and y. The points on the bottom part of the convex hull of the transformed problem will correspond to the nonredundant half-spaces within the original problem. The details are left to the reader.

A number of other intersection problems depend on the above/below relationship and/or the concept of slope. A line segment intersects a line if, and only if one endpoint lies above the line and the other lies below the line. A ray $\vec{r}$ intersects a line l if, and only if one of the following conditions holds:

1. $\vec{r}$ extends to the right with greater slope than l and the vertex lies below l;
2. $\vec{r}$ extends to the right with smaller slope than l and the vertex lies above l;
3. $\vec{r}$ extends to the left more steeply than l and the vertex lies above l;
4. $\vec{r}$ extends to the left with smaller slope than l and the vertex lies below l.

The intersection of two segments, two rays, or a ray and a segment can be determined by a sequence of tests involving slopes and/or vertical separation.

To show how T can be applied to the problems above, we must first determine the image under T of both rays and segments. Recall that the image of a line is the set of all lines through a point P. It is reasonable to assume that the image of a line segment should be a set of *some* of the lines through a point P. The set may

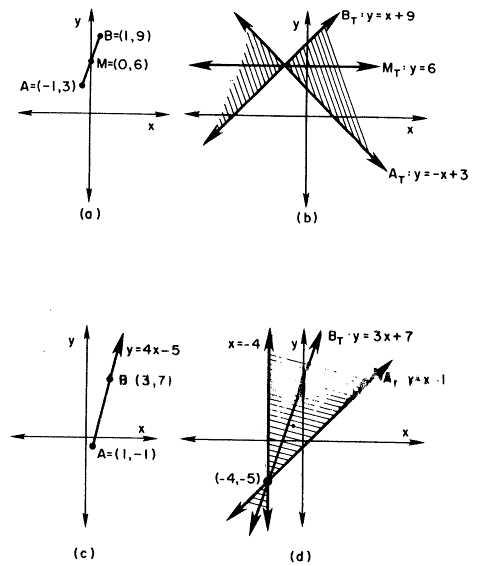


FIG. 2.19. A line segment (a) and its dual (b); a ray (c) and its dual (d)

not contain a vertical line since a line segment $\overline{AB}$ does not pass through a point at infinity. Thus, we may conclude that the image of $\overline{AB}$ consists of the set of all lines lying between A_T and B_T in the counter-clockwise direction (see Figs. 2.19a and b). This region is aptly described as a double wedge. A ray $\overrightarrow{AB}$ may be considered a line segment extending from A through the point B and ending at the improper point having direction $\overrightarrow{AB}$. The image of $\overrightarrow{AB}$, then, is a double wedge formed by A_T and the vertical line to which the improper point is mapped. As demonstrated in Figs. 2.19c and d, testing a point on $\overrightarrow{AB}$ determines which of the two double wedges corresponds to $\overrightarrow{AB}$.

Under the transformation T, we may determine the intersection of segments, rays and lines as follows. A line segment $\overline{AB}$ intersects a line l iff one of A_T and B_T lies above l_T and the other lies below. More simply, $\overline{AB}$ intersects l if l_T lies within the double wedge $\overline{AB}_T$ (see Figs. 2.20a and b). A ray $\overrightarrow{AB}$ extending rightward with slope m intersects a line l iff l_T lies to the right of $x = m$ and above A_T, or l_T lies to the left of $x = m$ and below A_T (see Figs. 2.20c and d). A ray $\overrightarrow{AB}$ extending leftward with slope m intersects l iff l_T lies to the right of $x = m$ and below A_T or to the left of $x = m$ and above A_T. Either case simplifies to $\overrightarrow{AB}$ intersects l iff l_T lies in the double wedge $\overrightarrow{AB}_T$. The segments, $\overline{AB}$ and $\overline{CD}$ intersect iff the image of the line $\overleftrightarrow{AB}$ lies in the interior of the double wedge $\overline{CD}_T$, and vice versa. This is equivalent to saying that $\overline{AB}$ and $\overline{CD}$ intersect iff $\overleftrightarrow{AB}$ intersects $\overline{CD}$ and $\overleftrightarrow{CD}$ intersects $\overline{AB}$.

This concludes our survey of the transformation T. As we noted earlier, there is a general scheme for applying both this and other transformations to geometric problems. While this one is particularly useful for some low-level intersection problems, the other transformations given in this section apply equally well in other contexts.

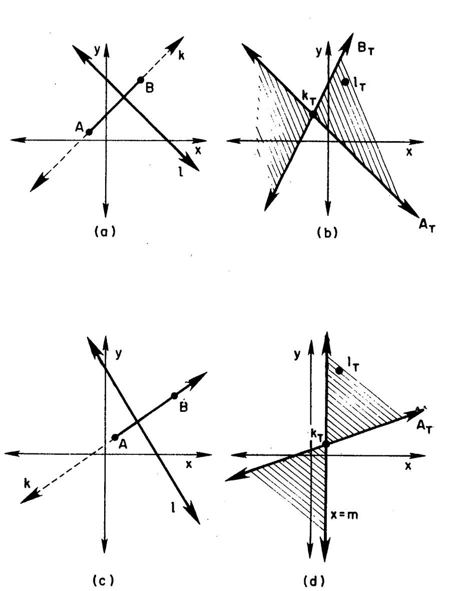


FIG. 2.20. Using duality to check for the intersection of: a line segment and a line (a & b); a ray and a line (c & d)

4.4. Point/Line Duality II

The second transformation we consider is well suited for polygon inclusion/intersection problems. We note that T is not particularly useful in these contexts. Applying T to the problem of whether a line segment s intersected ΔABC would, in fact, complicate the solution process. T maps ΔABC and its interior to the region R which is infinite but contains no vertical ray (see Figs. 2.21a, b). A segment s intersects ΔABC if and only if the following conditions are satisfied: The vertex of the double wedge s_T lies in R and the intersection of s_T and R contains at least one infinite line (see Fig. 2.21c). As neither R nor its complement form convex regions, these conditions are difficult to test.

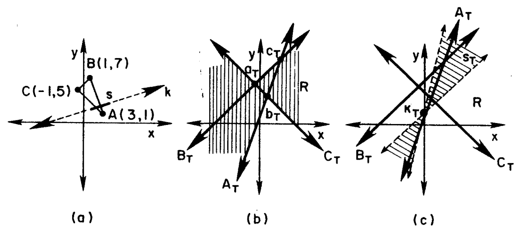


FIG. 2.21. The effect of the transformation T on a triangle and an intersecting line segment

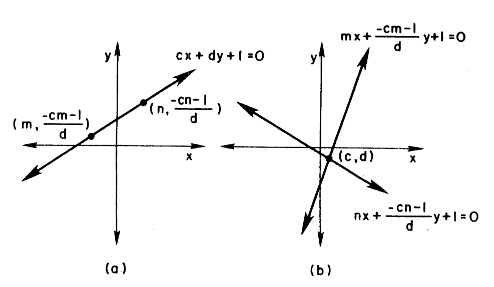


FIG. 2.22. A line (a) and its dual (b) under the transformation S

Our second transformation, S, maps the point (a, b) to the line $ax + by + 1 = 0$. We observe in Fig. 2.22 that S maps each point on the line $cx + dy + 1 = 0$ to a line passing through the point (c, d). Thus S maps the line $cx + dy + 1 = 0$ to the pencil of lines through (c, d), or the point (c, d). Like T, S maps a line segment to a double wedge, this time to one which does not contain the origin. The image of a ray is a double wedge formed by one line passing through the origin and one not.

The transformation S preserves the concept of the distance of an object from the origin. The line l: $ax + by + 1 = 0$ is perpendicular to the line m: $bx - ay = 0$ at the point $(-a/(a^2 + b^2), -b/(a^2 + b^2))$. The line l is precisely $1/\sqrt{a^2 + b^2}$ units away from the origin along the line m. The point (a, b), however, which is dual to l, is precisely $\sqrt{a^2 + b^2}$ units away from the origin along the line m in the opposite direction. Thus, points further from the origin are mapped to lines closer to the origin. That S is also its own inverse (i.e., $(a, b) \xrightarrow{S} ax + by + 1 = 0 \xrightarrow{S} (a, b)$) contributes to its usefulness.

We note that each improper point is mapped to the line through the origin having the same direction and vice versa. Similarly, the origin and the improper line are duals. Consequently, S should not be applied to lines or to segments of lines which pass through the origin. Nonetheless, the mere fact that the domain of a problem contains a line through the origin should not make us abandon S. By translating the axes in one direction or another, we may be able to insure that S will map every object in the domain of our problem to another proper object.

Since the transformation S preserves the concept of distance from the origin, it is most useful for problems where the origin serves a focal role. The image under S of an arbitrary triangle, ΔABC, is just as unwieldy as the image produced by T (see Figs. 2.23a and b). Consider, however, a convex polygon which contains the origin. Each segment from the origin to a point A on the boundary lies completely within the polygon. If the segment has slope m, then S maps each point on the segment to a line on the opposite side of the origin with slope $-1/m$. The line A_S is closest to the origin and the origin itself corresponds to the line at infinity. We may conclude that S maps the segment to the half-plane bounded by A_S which does *not* include the origin. Thus the interior of a convex polygon containing the origin is mapped to the union of a set of half-planes *not* containing the origin, or the exterior of a convex polygon.

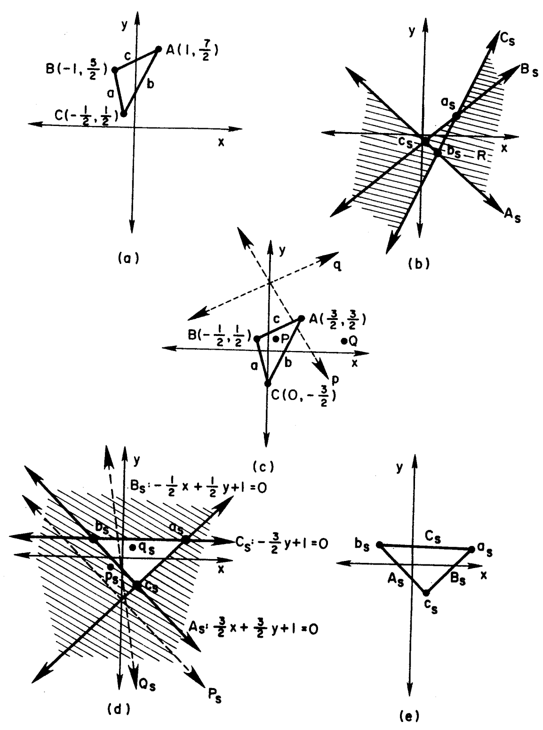

$A(1, \frac{7}{2})$
$B(-1, \frac{5}{2})$
$C(-\frac{1}{2}, \frac{1}{2})$
$A(\frac{3}{2}, \frac{3}{2})$
$B(-\frac{1}{2}, \frac{1}{2})$
$C(0, -\frac{3}{2})$
$B_S: -\frac{1}{2}x + \frac{1}{2}y + l = 0$
$C_S: -\frac{2}{3}y + l = 0$
$A_S: \frac{3}{2}x + \frac{3}{2}y + l = 0$
(a)
(b)
(c)
(d)
(e)

FIG. 2.23. The effect of the transformation S on an arbitrary triangle (a & b) and on a triangle enclosing the origin (c, d, & e)

We can demonstrate this phenomenon on the triangle of Fig. 2.23a, by first translating the axes 1/2 to the left and 2 units up, as shown in Fig. 23c. Now the union of the double wedges a_S, b_S and c_S consists of the entire plane except for the triangle formed by the vertices of a_S, b_S and c_S, which we call ΔABC_S (see Fig. 2.23d). ΔABC contains a point P if and only if P_S is exterior to ΔABC_S. A line p intersects triangle ΔABC if and only if p_S lies outside ΔABC_S. A point Q is exterior to ABC if and only if Q_S lies in ABC_S. And a line q is disjoint from ΔABC if and only if ΔABC_S contains q_S. This tidy symmetry allows us to simplify our definition of the transformation much as we did when we said that the image of a line is a point rather than a pencil of lines. When S is applied to a triangle containing the origin, we say that the image of each segment is a point, rather than a wedge, and the image of each vertex is a segment, rather than a line. In other words, S maps one triangle containing the origin to another triangle containing the origin (see Fig. 2.23e). The significance of the duality, however, is that the interior of one triangle is mapped to the exterior of the other.

This particular duality transformation can be useful in cases where we wish to

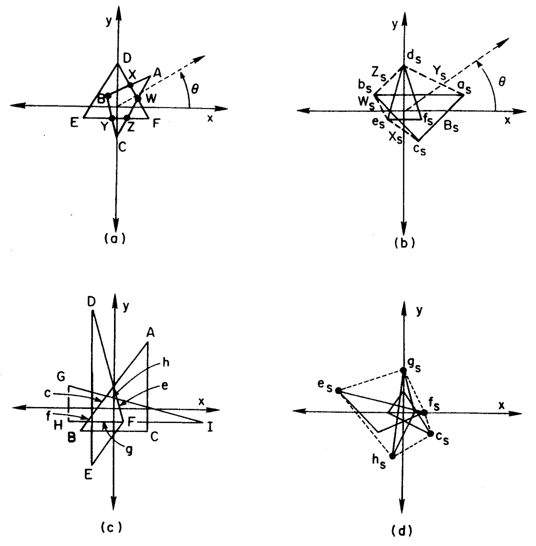


FIG. 2.24. Computing the intersection of triangles (a) by finding the convex hull of the dual triangles (b)

determine the intersection of triangles. In particular, suppose that, having already identified a common point, we wish to compute the intersection of two triangles. We translate the axes so that the common point lies at the origin. Let R represent the set of all points lying on the boundary of either triangle. Then the border of the intersection region can be defined as follows: $I = \{p \in R|$ at some angle $\theta \in [0, 2\pi)$, p is the closest point in R to the origin$\}$. The region of intersection forms a polygon which can be identified by listing a sequence of vertices. In Fig. 2.24a, the sequence of vertices describing the intersection region is $WXBYZ$. At θ, the segment XW is closest to the origin. In the dual problem, we wish to identify the polygon containing those points p which are the farthest from the origin for some θ. The process is simple: Finding the convex hull of ABC_S and DEF_S will yield a sequence of vertices e_S, c_S, a_S, d_S, b_S. The segments joining these vertices, X_S, B_S, Y_S, Z_S, W_S, correspond to the vertices of the intersection region of the original problem (see Fig. 2.24b).

Thus, the transformation S enables us to convert an intersection problem to a convex hull problem. The technique succeeds for any number of convex polygons as long as each contains the origin (see Fig. 2.24c and d). It works in a similar fashion in three dimensions. The three-dimensional version of S maps a point $P = (a, b, c)$ to the plane $ax + by + cz + 1 = 0$ which is perpendicular to line passing through both P and the origin $1/\sqrt{a^2 + b^2 + c^2}$ units on the opposite side of the origin from P. Suppose we apply S to a pair of intersecting planes.

Each plane is mapped to a point. The line formed by the intersecting planes maps to the line passing through the two image points. Consequently, the image of a convex polyhedron containing the origin will be another convex polyhedron containing the origin. The interior of the first, however, will have been mapped to the exterior of the second. Thus to compute the intersection of two or more convex polyhedra, we first find a common point, translate the axes so that this point becomes the origin; apply S; find the convex hull of all of the dual polyhedra; reapply S to obtain the vertices and faces of the intersection polyhedron.

4.5. Inversion (Point/Point Duality)

We turn now to the problem of determining the intersection of a collection of disks. Each of the transformations defined so far has been geared towards problems of straight line geometry, so we need a different type of transformation here. We note, however, that problems in straight line geometry are generally easier to solve than those involving curves. Consequently, a transformation which maps circles to lines could simplify the disk intersection problem.

We define a new transformation G which is similar to S in that it functions as its own inverse and preserves the concept of distance of an object from the origin. In this case, however, instead of mapping points to lines on opposite sides of the origin, G maps the point $P = (a, b)$ to a point which is $1/\sqrt{a^2 + b^2}$ units away from the origin in the same direction as P. Using polar coordinates, G maps (r, θ) to $(1/r, \theta)$. In fact, for G to be a one-to-one mapping of the plane onto itself, we must use polar coordinates and consider the origin as an infinite number of points of the form $(0, \theta)$, $\theta \epsilon (0, 2\pi)$, all of which lie on a circle of radius zero centered at the origin. In this way, the point at the origin denoted by $(0, \theta)$ is mapped to (∞, θ), the improper point having direction θ. The entire null circle is then mapped to the improper line. A circle centered at the origin with finite radius r is dual to a concentric circle with radius $1/r$.

The transformation G pairs each circle passing through the origin with a straight line which does not pass through the origin. As an example, we consider the line l: $y = -3x + 1$ (see Fig. 2.25). We determine the images of the points A, B, C, D, E, F by drawing a ray from the origin through each one of them and picking the point at the appropriate distance. In addition, we note that the rays $\theta = \theta_6$ and $\theta = \theta_0$, which are parallel to the line l, will intersect l at the point at infinity and only at the point at infinity. Thus the origin, or th point $(0, \theta_6) = (0, \theta_0)$, lies on the dual of l.

As demonstrated above, a line whose closest point is R units from the origin in direction θ is transformed to a circle passing through both the origin and $(1/R, \theta)$. A line $\theta = \theta_0$ passing through the origin, however, is mapped to itself: the segment from $(-1, \theta_0)$ through the origin to the point $(1, \theta_0)$ is mapped to the segment from $(-1, \theta_0)$ through the point at infinity to the point $(1, \theta_0)$, and vice

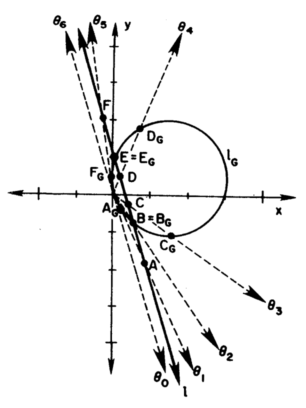


FIG. 2.25. A line and its dual under the transformation G

versa. A circle containing the origin is mapped to another circle containing the origin (see Fig. 2.26a), and a circle not containing the origin is mapped to another circle not containing the origin (see Fig. 2.26b). In all cases of circle/circle duality, the interior of one circle is mapped to the exterior of the other: the interior of a circle through the origin is mapped to a half-plane *not* containing the origin; a half-plane bounded by a line through the origin is mapped to itself.

G enables us to determine easily the intersection of a collection of disks whose boundaries all contain the origin. In the example shown in Fig. 2.27, three arcs bound the intersection of disks D, E, and F: the arc along circle D running in the clockwise direction from x to y; the arc along circle F from y to z; and the arc along circle E from z to x. G can help to determine this boundary. G transforms each of the disks to a half-plane bounded below by a straight line not through the origin. The intersection of these half-planes is bounded by the vertices x_G, y_G, z_G, where x_G is the point of infinity first along D_G and secondly along E_G, and by the segments of D_G, F_G, and E_G which join the vertices. These correspond

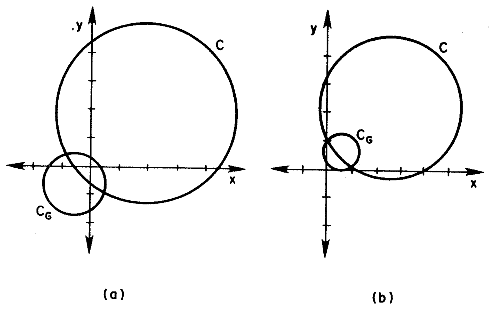


FIG. 2.26. Two circles and their duals

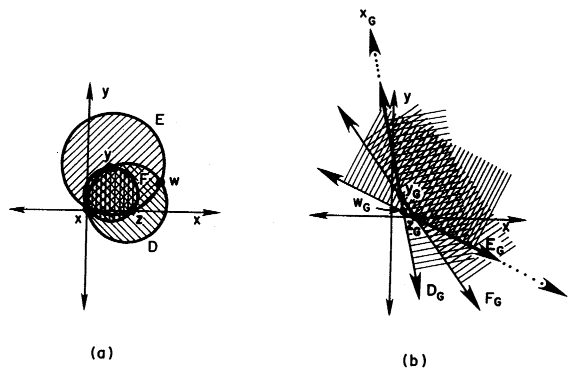


FIG. 2.27. Computing the intersection of a set of circles (a) by finding the intersection of the dual half-planes (b)

exactly to the vertices and arcs of the intersection region in the original. Similarly, the union of the half-planes would correspond to the union of the disks.

A problem involving only disks which all pass through a single point would be quite rare. We may, however, extend G to three dimensions: (r, θ, ϕ). Given a collection of disks in a plane, we can choose an arbitrary point P not in the plane. Then for each disk d, we determine the unique ball which passes through P and which contain d as a cross-section. Next we translate the axes so that the origin coincides with P. G will now map each ball to a half-space not containing the origin. The intersection of the half-spaces will correspond to the intersection of the balls which we can then intersect with the original plane to determine the

TABLE 2.1
Summary of the Transformations Considered in this Section.

	T	S	G
(a,b)	$y=ax+b$	$ax+by+1=0$	$\left(\frac{a}{a^2+b^2}, \frac{b}{a^2+b^2}\right)$
$y=ax+b$	$(-a,b)$	$\left(\frac{a}{b}, \frac{-1}{b}\right)$	$\left(x+\frac{a}{2b}\right)^2+\left(y-\frac{1}{2b}\right)^2=\frac{a^2+1}{4b^2}$
$ax+by+1=0$	$\left(\frac{a}{b}, \frac{-1}{b}\right)$	(a,b)	$\left(x+\frac{a}{2}\right)^2+\left(y+\frac{b}{2}\right)^2=\frac{a^2+b^2}{4}$
$(x-h)^2+(y-k)^2=h^2+k^2$	—	—	$hx+ky=\frac{1}{2}$
preserves	above/below, slope	distance	distance
makes improper	vertical lines	the origin	the origin
useful for	half-plane,ray $\cap$	polygon $\cap$	disk $\cap$

intersection of the disks. The interested reader may find that these or other transformations will simplify his/her favorite problem.

4.6. Notes

The mathematical background section reflects *The VNR Concise Encyclopedia of Mathematics*.

Geometric transformations such as those mentioned here have their roots in the mathematics of the early nineteenth century (Boyer, 1968). Transformations appear in the work of Steiner, and projective geometry originated with Poncelet. Their application to problems of computing dates back to the concept of primal and dual problems in the study of linear programming (see e.g., Papadimitriou & Steiglitz, 1982).

Brown (1979) gives a systematic treatment of transformations and their applications to problems of computational geometry. Since his dissertation, these methods have found vast application (Chazelle, 1983; Chazelle, Guibas, & Lee, 1983; Dobkin & Edelsbrunner, 1984; Edelsbrunner, Kirkpatrick, & Maurer, 1982; Edelsbrunner, O'Rourke, & Seidel, 1983).

The examples given here are derived from Brown's work (1979) as well as from Muller and Preparata (1978). The latter reference gives a complex application of duality to the polyhedron intersection problem, extending their work on halfspace intersection (Preparata & Muller, 1979). The problem is divided there into the subproblems of finding an intersection point and then using that as a basis for computing the entire intersection. The problem of finding the first point was then further studied in Chazelle and Dobkin (1980) and Dobkin and Kirkpatrick (1983).

5. CONCLUSION

We have presented here three different techniques which have proved useful in deriving algorithms for geometric problems: hierarchical search, hierarchical computation, and geometric transformations. Although in each case we supply specific examples, each technique has a wide variety of applications.

Our ability to produce an efficient algorithm using one or more of the given techniques derives from some understanding of the underlying geometry. Each of the first two depends on finding an appropriate data structure or basic unit. For the general polygon search algorithm, we chose a triangular subdivision. For the convex hull algorithm, concatenable queues held the partial hulls. To choose an appropriate geometric transform, the problem must be carefully analysed to determine the properties or characteristics which must be preserved. To solve the half-plane intersection problem, we select a transformation which preserves vertical distance and slope.

Although the three techniques appear to be distinct, they are mutually dependent in many applications. For example, the dynamic convex hull algorithm above depends on hierarchical search for insertions and deletions. In general polygon search, the initial triangulation of S to form S_0 depends on a hierarchical computation of an ordered list of vertices. Geometric transforms can convert a whole problem or part of a problem to a format more accessible to hierarchical decomposition. Together or separately, these three techniques span a wide field of application.

Although all applications here have been geometric in nature, we believe that these methods will have profitable applications in the field of robotics. We invite feedback.

ACKNOWLEDGMENTS

This research was supported in part by the National Science Foundation under Grant MCS83-03926. The second author was supported by an Exxon Fellowship.

We thank D. Barbara, B. Chazelle, S. Friedman, R. Horn, J. Kent, and K. Supowit for reading earlier drafts of this paper and contributing to a more readable document. We are especially indebted to T. Agans for his superb drafting of the numerous figures.

REFERENCES

Many of the references listed here are not cited in the text. We include them here to broaden our survey.

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1974). *The design and analysis of computer algorithms*. Reading, MA: Addison-Wesley.
Bentley, J. L. (1979). Decomposable searching problems. *Inf. Proc. Lett.*, *8*, 244–251.
Bentley, J. L., & Ottmann, T. A. (1979). Algorithms for reporting and counting geometric intersections, *IEEE Trans. Comp.*, *C-28*, 643–647.
Bentley, J. L., & Saxe, J. B. (1980). Decomposable searching problems I: Static-to-dynamic transformation.*Journal of Algs.*, *1*, 301–358. A preliminary version appeared in *Proc. 20th IEEE FOCS*, 1983, 148–168.
Bentley, J. L., & Wood, D. (1980). An optimal worst-case algorithm for reporting intersections of rectangles. *IEEE Trans. Comp.*, *C-29*, 572–577.
Boyer, C. B. (1968). *A history of mathematics*. New York: Wiley.
Brown, K. Q. (1979). *Geometric transforms for fast geometric algorithms*. Carnegie-Mellon. (Tech. Report CMU-CS-80-101.)
Brown, K. Q. (1981). Comments on "Algorithms for reporting and counting geometric intersections," *IEEE Trans. Comp.*, *C-30*, 147–148.
Chazelle, B. (1983). The polygon containment problem. *Advances in computing research* (Vol. I, pp. 1–34). Greenwich, CT: JAI press.
Chazelle, B. (1984a). Intersecting is easier than sorting. *Proc. 16th ACM STOC*, pp. 125–134.
Chazelle, B. (1984b, October). Personal communication with D. Souvaine.

Chazelle, B., Cole, R., Preparata, F., & Yap, C. (1984). *New upper bounds for neighbor searching*, Brown University. (Tech. Report CS-84-11.)

Chazelle, B., & Dobkin, D. P. (1980). Detection is easier than computation. *Proc. 12th ACM STOC*, pp. 146–153.

Chazelle, B., & Dobkin, D. P. (1985). Optimal convex decompositions. In G. T. Toussaint (Ed.), *Machine Intelligence and Pattern Recognition 2: Computational Geometry*. Amsterdam, The Netherlands: Elsevier Science Publishers.

Chazelle, B., Drysdale, R. L., & Lee, D. T. (1986). Computing the largest empty rectangle. *SIAM J. Comp.*, 15, 300–315.

Chazelle, B., Guibas, L. J., & Lee, D. T. (1983). The power of geometric duality. *Proc. 24th IEEE FOCS*, pp. 217–225.

Chazelle, B., & Incerpi, J. (1984). *Triangulation and shape-complexity. ACM Trans. Graphics, 3*, 135–152.

Cole, R., & Yap, C. K. (1983). Geometric retrieval problems. *Proc. 24th IEEE FOCS*, pp. 112–121.

Dobkin, D. P., Drysdale, R. L., & Guibas, L. J. (1983). Finding smallest polygons. *Advances in computing research* (Vol. 1, pp. 181–214). Greenwich, CT: JAI Press.

Dobkin, D. P., & Edelsbrunner (1984). Space searching for intersecting objects. *Proc. 25th IEEE FOCS*, pp. 387–391.

Dobkin, D. P., & Kirkpatrick, D. G. (1983). Fast detection of polyhedral intersections. *Theo. Comp. Sci., 27*, 241–253.

Dobkin, D. P., & Kirkpatrick, D. G. (1985). A linear algorithm for determining the separation of convex polyhedra. *Journal of Algs., 6*, 381–392.

Dobkin, D. P., & Lipton, R. J. (1976). Multidimensional searching problems. *SIAM J. Comp., 5*, 181–186.

Dobkin, D. P., & Lipton, R. J. (1979). On the complexity of computation under varying sets of primitives. *J. Comp. Syst. Sci., 18*, 86–91.

Dobkin, D. P., & Munro, J. I. (1985). Efficient uses of the past. *Journal of Algs., 6*, 455–465.

Dobkin, D. P., & Reiss, S. (1980). The complexity of linear programming. *Theo. Comp. Sci., 11*, 1–18.

Dobkin, D. P., & Snyder, L. (1979). On a general method for maximizing and minimizing among certain geometric problems. *Proc. 20th IEEE FOCS*, 9–17.

Dyer, M. E. (1983). A geometric approach to two-constraint linear programs with generalized upper bounds. *Advances in computing research* (Vol. I, pp. 79–90). Greenwich, CT: JAI Press.

Edelsbrunner, H., Kirkpatrick, D. G., & Mauer, H. A. (1982). Polygonal intersection searching. *Inf. Proc. Lett., 14*, 74–79.

Edelsbrunner, H., O'Rourke, J., & Seidel, R. (1983). Constructing arrangements of lines and hyperplanes with applications. *Proc. 24th IEEE FOCS*, pp. 83–91.

Edelsbrunner, H., Overmars, M. H., & Wood, D. (1983). Graphics in flatland: A case study. *Advances in computing research* (Vol. I, pp. 35–60). Greenwich, CT: JAI Press.

Garey, M. R., Johnson, D. S., Preparata, F. P., & Targan, R. E. (1979). Triangulating a simple polygon. *Inf. Proc. Lett., 7*, 175–179.

Graham, R. L. (1972). An efficient algorithm for determining the convex hull of a planar set. *Inf. Proc. Lett., 1*, 132–133.

Greene, D. H. (1983). The decomposition of polygons into convex parts. *Advances in computing research* (Vol. I, pp. 235–260). Greenwich, CT: JAI Press.

Grünbaum, B. (1967). *Convex polytopes*. New York: Wiley Interscience.

Guibas, L. T., & Yao, F. F. (1983). On translating a set of rectangles. *Advances in computing research* (Vol. I, pp. 61–78). Greenwich, CT: JAI Press.

Karmarker, N. (1984). A new polynomial-time algorithm for linear programming. *Proc. 16th ACM STOC*, pp. 302–311.

Kirkpatrick, D. G. (1980). A note on Delaunay and optimal triangulations. *Inf. Proc. Lett., 10*, 127–128.

Kirkpatrick, D. G. (1983). Optimal search in planar subdivisions. *SIAM J. Comp., 12*, 28–35.

Knuth, D. E. (1973). *Sorting and searching*. Reading MA: Addison-Wesley.

Lee, D. T., & Preparata, F. P. (1977). Location of a point in a planar subdivision and its applications. *SIAM J. Comp., 6*, 594–606.

Lee, D. T., & Preparata, F. P. (1979). An optimal algorithm for finding the kernal of a polygon. *Journal of the ACM, 26*, 415–421.

Lee, D. T., & Preparata, F. P. (1984). Computational geometry—a survey. *IEEE Trans. Comp., C-33*, 1072–1101.

Lipton, R. J., & Tarjan, R. E. (1979). A separator theorem for planar graphs. *SIAM J. App. Math., 36*, 177–189.

Lipton, R. J., & Tarjan, R. E. (1980). Applications of a planar separator theorem. *SIAM J. Comp., 9*, 615–627.

Lloyd, E. L. (1977). On triangulation of a set of points in the plane. *Proc. 18th IEEE FOCS*, pp. 228–240.

Muller, D. E., & Preparata, F. P. (1978). Finding the intersection of two convex polyhedra. *Theo. Comp. Sci., 7*, 217–236.

Overmars, M. H. (1981). Dynamization of order decomposable set problems. *Journal of Algs., 2*, 245–260.

Overmars, M. H. (1983). *The design of dynamic data structures*. Doctoral thesis presented at the University of Utrecht, Druk: Sneldruk Boulevard Enschede.

Overmars, M. H., & van Leeuwen, J. (1981). Maintenance of configurations in the plane. *Journ. Comput. Sys. Sci., 23*, 166–204. A preliminary version appeared in Proc. 12th ACM STOC, pp. 135–145.

Papadimitriou, C. H., & Steiglitz, K. (1982). *Combinatorial optimization: Algorithms and complexity*. Englewood Cliffs, NJ: Prentice Hall.

Preparata, F. P. (1979). An optimal real-time algorithm for planar convex hulls. *C. ACM, 22*, 402–405.

Preparata, F. P. (1981). A new approach to planar point location. *SIAM J. Comp., 10*, 473–482.

Preparata, F. P., & Hong, S. J. (1977). Convex hulls of finite sets in two and three dimensions. *Comm. ACM, 20*, 87–93.

Preparata, F. P., & Muller, D. E. (1979). Finding the intersection of n half-spaces in time *O(n* log *n). Theo. Comp. Sci., 8*, 45–55.

Reingold, E. M., Nievergelt, J., & Deo, N. (1977). *Combinatorial algorithms*. Englewood Cliffs, NJ: Prentice Hall.

Seidel, R. (1984). *A method of proving lower bounds for certain geometric problems*. Cornell University. (Tech. Report 84-592.)

Shamos, M. I. (1975). Geometric complexity. *Proc. 7th ACM STOC*, pp. 224–233.

Shamos, M. I., & Hoey, D. (1975). Closest-point problems. *Proc. 16th IEEE FOCS*, pp. 151–161.

Shamos, M. I., & Hoey, D. (1976). Geometric intersection problems. *Proc. 17th IEEE FOCS*, pp. 208–215.

Supowit, K. J. (1983). Grid heuristics for some geometric covering problems. *Advances in computing research* (Vol. I, pp. 215–234). Greenwich, CT: JAI Press.

THE VNR Concise Encyclopedia of Mathematics. (1977). W. Gellert, H. Kustner, M. Hellwich, & H. Kastner (Eds.). New York: Van Nostrand Reinhold Company, pp. 146–203, 547–561.

van Leeuwen, J., & Overmars, M. H. (1981). The art of dynamizing. *Mathematical Foundations of Computer Science* (pp. 121–131). Berlin; New York: Springer Verlag.