FAST GEOMETRIC ALGORITHMS

by

Mark T. Noga

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science and Applications

January, 1984
Blacksburg, Virginia

To Naida

WORDS OF THANKS

"I'll note you in my book of memory."

- William Shakespeare, *King Henry VI*

This thesis describes the culmination of five years of prolonged study at Virginia Polytechnic Institute. During that time, I have benefitted from the close companionship of many people from all different parts of the United States and the world. Without your concern, understanding, encouragement, wisdom, and prodding, completion of this document would have been an all but impossible task. To all my friends I would like to say *"thank you."* Some of you gave more than what one would normally expect out of friendship and curiosity, and it is an honor for me to acknowledge these individually (and not in any special order).

Occasionally you meet an individual who has the ability to take several seemingly unrelated results and somehow combine them to form what is often a colorful and unique solution to a particular problem. Doug Smith is one of these rare individuals. We spent many hours discussing, amongst other things, why art is pleasing to the eye (the entropy of art), or what features programming languages would contain in the year 2050 (or even if they would exist in a form equivalent to today's languages). Doug was always willing to discuss spatial problems because of his keen interest in art. His contributions were the basis for several of the most important results in this thesis.

Another fellow student who had a profound influence on my work was Barry Fritchman or *"Super Fritch"* as he was known within the Department of Computer Science. Barry was very helpful when it came to VAX system procedures and protocols. Among the qualities Barry possessed was the ability to quickly spot an incorrect approach to a problem. This no doubt saved considerable time in my research effort.

Two of my neighbors in "closets" F and H, Dwight Barnette and Irene Stein, also deserve my warmest thanks for their encouragement and stimulation during those times when I firmly believed that it would be impossible to finish testing out my theories and ideas. Both Dwight and Irene were very enthusiastic about major sporting events. In fact, Irene was so proficient at picking Super Bowl and World Series winners that we accorded her the special name of *"The Swami."*

For technical assistance and support, preparing work orders, typing letters, money for computer accounts, etc., a big thanks goes to Donna Burford, Barbara Love, Allison Taylor, Sandy Birch, and Dee Stater. Dee always made sure I received my assistantship check on time (or at least she tried, subject to the whims of the payroll department). Donna and Allison were always willing to listen to my little problems and help me out of a tough spot when I needed it. Occasionally, they even drove me to one of the local markets to buy groceries.

Looking for a job is an especially time-consuming process. This, coupled with the additional responsibility of completing a dissertation, can lead to many long work days. That is why I am especially grateful to two former Virginia Tech colleagues, Tom Laffey and Bill McCormack. Both helped to guide me in my search for an appropriate position in the "Silicon Valley" area of Northern California where they now work. Bill also served for a short time on my Ph.D. committee and I would like to thank him for his help then.

The following names stand out amongst the many people I have had the pleasure to know while at Virginia Tech. Most were either roommates, personal friends, or fellow graduate students. Sometimes I feel as if a book could be written about the varied experiences we've shared together: Pat Bixler, Erik Turner, Ned Okie, Andie Bretzius, Mike Stinson, Dave Taylor, Bonnie Maier, Shuhab Ahmed, Peter Forbes, Art and Denise Leifer, Peggy Laffey, Dave Kanazawa, Teresa Asid, Peggy Bertsch, Mike and Betsy Heruska, Lois Remsen, Jill Foreman, Betsy Decker, Hilary Zaloom, Jeff and Valerie Facemire, T.C. Pong, Prasanna Mulgonkar, Bob Moose, and Diane Trahan of Arnold's.

The family is always an integral part of any student's life. In my case two sisters, Julie and Lisa Noga, always welcomed me with open arms and affection whenever I had the opportunity to return to my home in Minnesota. They never ceased to amuse me with all their experiences, little jokes, and tidbits of gossip about family and friends.

It is easy to take for granted the effort your parents spend in raising you during the formative years. We as their children believe that they have a moral responsibility to ensure a proper education, diet, and medical care at least until the age of 18. In my case I have placed an additional ten years of responsibility upon their shoulders. Yet they have rarely, if ever, complained and have encouraged me in my academic adventures. I am deeply indebted for their love and generous support throughout the college years.

I would like to thank Donald Allison, David Roselle, Bob Haralick, Roger Ehrich, and John Roach for serving as the members of my dissertation committee. Donald was the chairman of this distinguished group and it was he who suggested that my early work on convex hull algorithms might be expanded to form the basis for the set of topics discussed inside this document. The time and effort he has spent editing this document has been considerable. It would be remiss of me not to thank him for his many contributions. Much of the growth in the Computer Science Department at Virginia Tech in prestige, and in quality and size of faculty, can be attributed to his leadership as Department Chairman. Financial aid, in the form of research assistantships, part-time hourly wages, and a Tennessee Eastman Scholarship were always available because of his committment to Ph.D. (graduate student) research. David helped in several ways. As Dean of the Graduate School he was no doubt instrumental in helping me win a Cunningham Summer Fellowship. Also, his encouragement and enthusiasm during the early stages of research were critically important to me. The original research proposal called for work involving the Euclidean metric only. Bob suggested that I consider problems in the L_1 (or Manhatten) metric, such as the diameter of a set. His other contribution involved the statistical nature of sorting where he insisted upon both a broader test-bed for performance profiling and a clearer definition of where the majority of computation takes place in sorting algorithms. Roger made several contributions involving the style and form of the manuscript. These have helped to produce a "very readable" report of my work. John agreed to serve on my Ph.D. committee on short notice for which I am very grateful. He also provided me with a place to live during the final days of this research effort.

Naida Seemann led a short life of some twenty years. At the time of her tragic death she was a theater arts major at the University of Minnesota, Duluth. She was both a fine musician and actress. My memories are of a beautiful fair-haired girl of Norwegian descent, a symbol of the Minnesota Lake Region. I am proud to say that she was my friend and close companion. It is to her that I dedicate this dissertation.

M.T.N.

Blacksburg, VA

January, 1984

TABLE OF CONTENTS

Chapter 1
INTRODUCTION

"That life is a mystery gives us hope
we may one day understand it."
- Van Over, *Total Meditation*

1.1. The Interplay between Geometry and the Computing Sciences

Geometry has been at the heart of many of man's greatest discoveries and accomplishments throughout the past several thousand years. The ancient Egyptians, Mayans, and Greeks were among several prominent groups who instigated scientific study and advanced the theorems which are the foundation of this discipline, the most central of the Mathematical Sciences. The Great Pyramid in the Valley of Kings near the Nile Delta is perhaps regarded as the most striking example of the importance geometric insight played in the architectural and spiritual development of ancient cultures.

The Computing Sciences, on the other hand, is by comparison a very new field, having a relatively short history of some forty years. Since the majority of Geometry developed long before the advent of computing machinery, the field is a composite of ideas that is not readily translated into computer algorithms. While this may seem obvious now, the designers who developed the first computers were of the opinion that once their machines were actually functioning, rewriting and transcribing known mathematical results into actual machine language encoded algorithms would be a rather trivial problem [Wilkes (81)]. What they soon discovered is that straightforward transcriptions do not usually produce the best algorithms. Significant issues arise in problem representation, data organization, algorithm design, and obtaining actual bounds on the number of elementary operations required to perform a computation.

These considerations ultimately gave rise to the field of *Analysis of Algorithms* in which the main objective is to develop fast algorithms which operate within the framework of today's high speed computers. An important branch of this field is *computational geometry* which centers mainly on the development of fast geometric algorithms. Computational geometry is perhaps the fastest growing sub-area of Analysis of Algorithms. Not only is this area intuitively attractive, but many applications have surfaced which have further promoted additional research efforts. Most of the work has focused upon restructuring the theories of the Ancients into explicit algorithmic form (see [Shamos (78)] for the details). The tools applied to this task have been modern data structures and some previous results from the study of non-geometric algorithms. We will follow the same pattern. Our attack will focus upon several new problems, as well as an investigation of how to improve existing fast geometric algorithms. Our goal will be to provide the reader with a set of tools which will allow him to successfully attack more substantial problems.

1.2. Literature Synopsis

Computational Geometry is the study of the design, exposition, and analysis of problems involving points, lines, and objects in two and three dimensional space. The emergence of this field has been strongly motivated by several interrelated factors. One is that there are a number of application areas such as computer graphics, robotics, and remote sensing, that require manipulation of geometric objects in spatial real-time environments. For instance, a user might see two overlapping figures on a cathode ray tube and request the computation of their intersection. Another factor stems mainly from the set of "famous" problems in Classical Geometry. A good illustration is the smallest circle that will enclose a given set of points. Classical Geometry tells us that at least two of the points on the perimeter must be vertices of the *convex hull* of the set. (The convex hull is the minimum-area convex polygon containing the set of points; Fig. 1.1.) However, it does not tell us how to compute algorithmically the convex hull and choose those vertices which are on the circle.

Research in Computational Geometry has developed along several major lines. We will attempt to classify the different categories and provide some examples of the types of problems in each area.

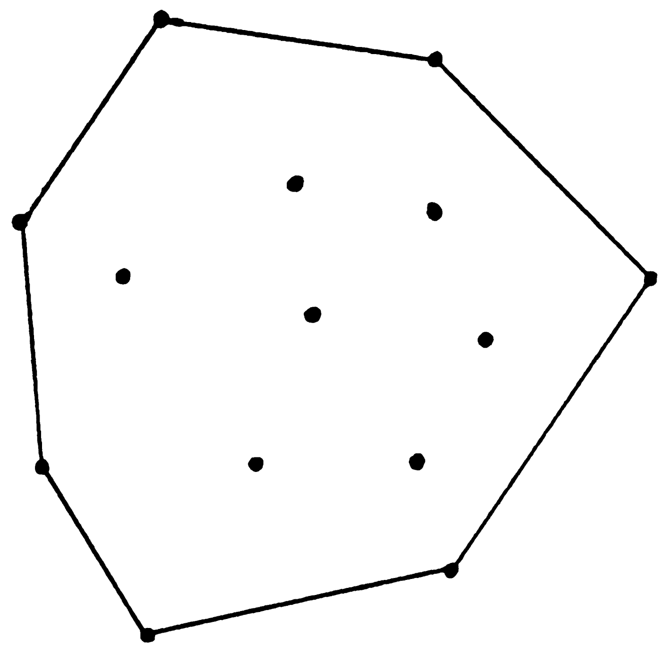


Fig 1.1. The convex hull of a set of points in the plane.

1.2.1. Minimization and Maximization Problems

Certainly the most famous problem, and the one that has had a profound effect on the growth of Computational Geometry is the computation of the convex hull (see definition above). This geometric structure has many applications, but most of all it appears to be a universal tool in *Pattern Recognition.* (Pattern Recognition involves the extraction of information from stochastic transformations of objects and the identification of the object that gave rise to that particular realization [Ehrich (84)].) For instance, it can be used for normalizing patterns, defining decision rules in classification, and obtaining triangulations of points sets [Lawson (77)].

In Chapter Four we give an expository review of our work in [Noga (81)] which contains a description of several of the best methods for computing the hull. A performance evaluation is included which may be of some value to researchers using convex hull algorithms for specific applications. For further details concerning the applications of convex hulls consult [Toussaint (78b)].

Several other problems which have received a significant amount of attention are determining the two points of a set which are farthest apart, *i.e.,* the *diameter of a set,* finding the rectangle of minimum area that will encase a set of points (or a polygon), and finding a tour through a set of points that is minimal with respect to distance. We will discuss all of these problems in Chapters Five and Six. Consult [Shamos (75a)], [Shamos (78)], [Freeman (75)], and [Golden (80)] for additional details.

Other equally important but lesser known problems are finding the ellipse of minimum area covering a given set of points [Silverman (81)], determining the circle of minimum diameter that will cover the set [Shamos (78)], finding the minimum distance between two convex polygons [Schwartz (81)], and inscribing polygons within other polygons which maximize some measurement of distance or area [Dobkin (79)].

Given a rectangular board, can a set of polygons be rearranged in such a way so they do not intersect yet fit within the boundary of the board? This problem has a number of important applications including space planning, template layout, and cutting stock. Although Operations Researchers have advanced a number of special case solutions, the general problem remains unsolved. Recently, it has been shown that some optimal packing problems are NP-complete [Fowler (81)]. Therefore, it is not clear whether any existing Computational Geometry technique will be of any value in attacking this problem.

1.2.2. Closest Point Problems

This large class of problems involves questions concerning the proximity of points in the plane. The basic strategy has been to use a geometric structure called the *Voronoi diagram.* (The Voronoi diagram of a set S of n points p_i, $1 \leq i \leq n$, in the Euclidean plane, is a partitioning of the plane into n polygonal regions, one region associated with each p_i. The Voronoi region $V(p_i)$ associated with each p_i consists of all points closer to p_i than to any of the other p_j, $j \neq i$; Fig. 1.2.) The major problems include, (i) given n points in plane, find the two that are closest together, (ii) for each of n points find its nearest neighbor, and (iii) with preprocessing allowed, how quickly can the nearest point be found with respect to a new given point p. Algorithms can be found in [Shamos (78)].

A related problem involves triangulating a set of points. Because it is often desirable to obtain triangles which have small overall length or weight, where the latter is defined to be the sum of the Euclidean length of all edges of the triangulation, the problem reduces to one of determining the nearest neighbors of a point. It turns out that the Voronoi Diagram can be used to obtain the *Delaunay triangulation* (the dual of the Voronoi Diagram) which has a locally optimal property that usually yields a near minimum weight triangulation.

Three graph theoretic structures which are subsets of the Delaunay triangulation are the *Gabriel graph*, the *relative neighborhood graph*, and the *minimum spanning tree.* All three are based upon slightly different definitions of what it means for two points to be relatively close. In Section 6.13 definitions of these structures can be found along with additional references containing algorithms for their computation.

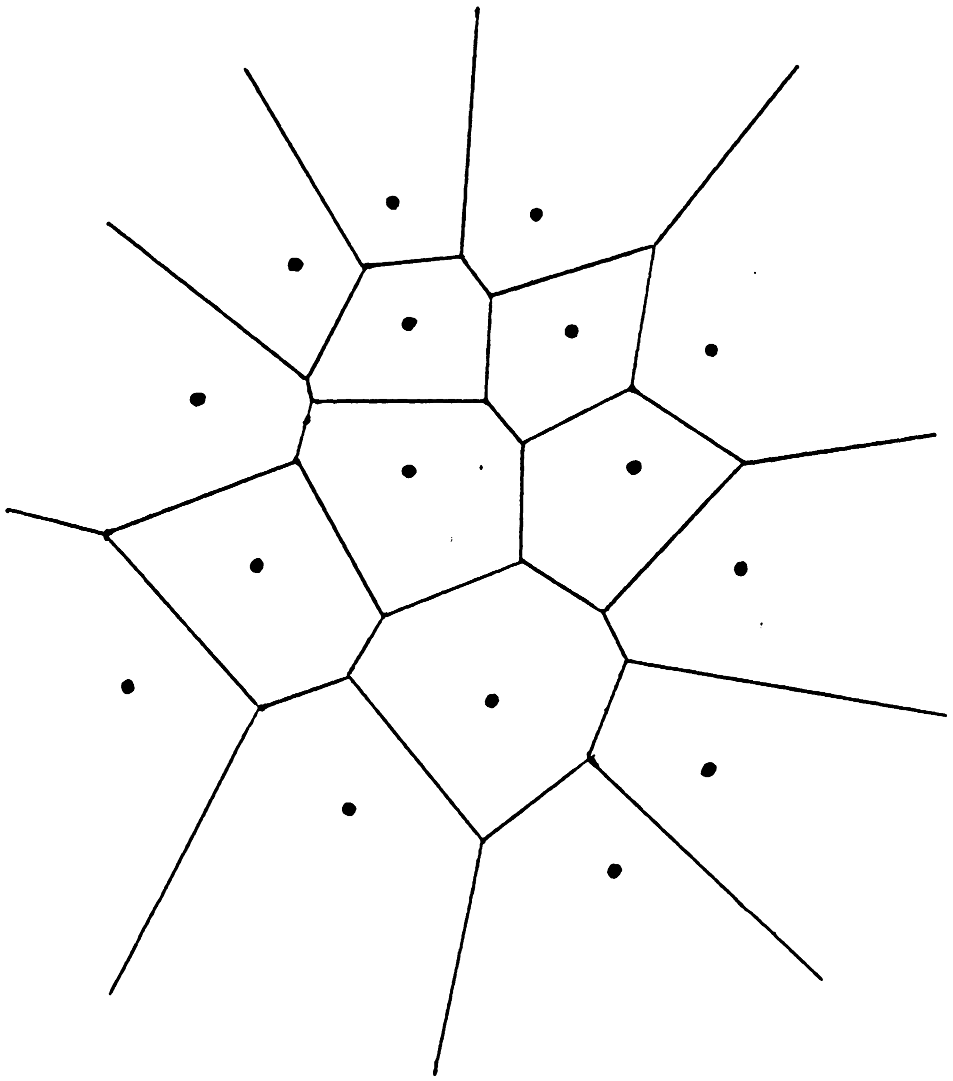


Fig. 1.2. The Voronoi diagram of a set of points.

1.2.3. Inclusion Problems

This class of problems involves determining the location of an object with respect to other (possibly surrounding) objects. The primary problem is, given a simple polygon P and a new point z, determine whether or not z is interior to P. It turns out that the solution depends to some extent on whether P is a convex polygon and whether preprocessing is allowed [Shamos (78)]. As Shamos points out in his thesis [Shamos (78)], *"The importance of this problem stems from the fact that almost all geometric searching, at some level, can be reduced to testing polygon inclusion."* Shamos' analysis leads him to consider the possibility of maintaining databases and efficient search structures to handle queries that will be repeatedly performed on the same polygon.

Inclusion in a planar straight-line graph is a problem that shares a close relationship with polygon inclusion testing; Fig. 1.3. Given a planar straight-line graph and a new point z, how quickly can the region containing z be found? Lipton and Tarjan [Lipton (77)] have proved a very powerful result, the *planar separator theorem,* which they used to yield an asymptotically fast algorithm as follows. The vertices of any n-vertex planar graph G can be partitioned in linear time into three sets A, B, and C such that no edge of G joins a vertex of A with a vertex in B, neither A nor B contains more than 2n/3 vertices, and C contains $O(n^{1/2})$ vertices. Other investigations have been carried out by Lee and Preparata [Lee (76)].

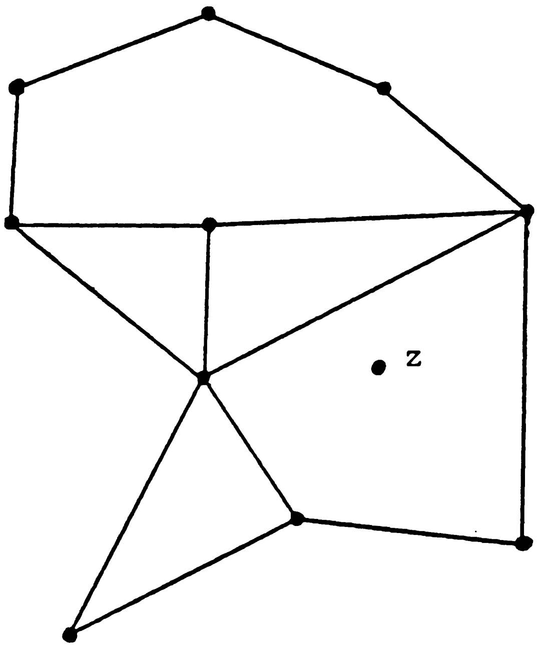


Fig. 1.3. A planar straight-line graph.
In what region does z lie?

Another problem which arises frequently in geographic applications is that of orthogonal range searching: Given n points in the plane, how many lie inside a given rectangle whose sides are parallel or perpendicular to the axes of the implied coordinate system. The major contribution to solving this problem, which involves a concept called *vector domination,* has been given by Bentley and Shamos [Bentley (77a)]. A related problem is the planar fixed-radius near-neighbor problem: preprocess a set S of n points in the plane so that all points of S lying within some fixed radius r of a new point can be listed efficiently. Solutions have been given by Bentley, Stanat, and Williams in [Bentley (77b)] and improved upon in [Chazelle (83)].

1.2.4. Intersection and Visibility Problems

Intersection problems are intricately related to the inclusion problems because two figures intersect only if one contains a point of the other. The importance of efficiently determining whether there is an intersection amongst two or more geometric objects has become increasingly important in a number of industrial applications, including VLSI chip design, CAD/CAM, computer graphics, and robotics.

Considerable research has involved a pivotal problem in computer graphics, *the hidden-line problem.* The idea is to produce a two dimensional picture of a three dimensional scene taking into account that some objects may be partially or totally obscured from the viewpoint of an observer; Fig. 1.4. This naturally leads to the question of how to remove lines hidden by the surfaces of objects which are closer to the observer. The literature on this problem is considerable with [Desens (69)], [Galimberti (69)], [Newman (73)], [Sutherland (66)], [Warnock (69)], and [Watkins (70)] containing the bulk of the major results.

A fundamental question associated with hidden-line removal is to form the intersection of two polygons. The difficulty of the problem depends on the type of polygons involved in the intersection. If, for instance, both are known to be convex, then an algorithm which takes time proportional to the total number of vertices in both polygons suffices. On the other hand, for star-shaped and simple polygons the number of intersection points can be proportional to the square of the total number of vertices in both polygons which implies that any algorithm to solve this problem must take at least kn^2 operations in the worst case, for some positive constant k [Shamos (78)].

Separability is a classical question in combinatorial geometry. Given two sets of points, does there exist a hyperplane that separates them. The importance of finding a fast algorithm comes from the field of pattern recognition where it is desirable, if possible, to obtain a two-variable linear classifier, that is, a linear function f such that a single comparison will suffice to determine the sample to which a point belongs [Meisel (72)]. Because two sets are linearly separable if and only if their convex hulls do not intersect, the problem reduces to the intersection of two convex polytopes [Shamos (75a)].

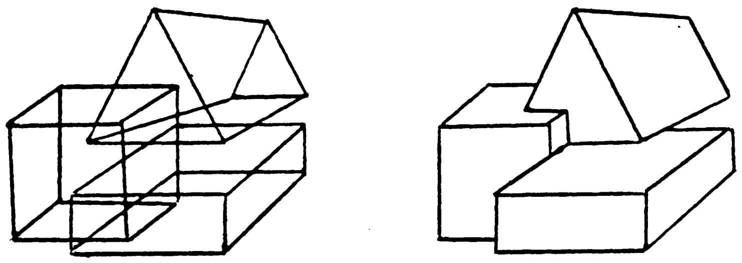


Fig. 1.4. Elimination of hidden lines.

An interesting problem in component layout is given n line segments in the plane, determine whether any two intersect. A related problem is the determination of the intersection of two simple polygons P and Q. Algorithms for these problems are given by Hoey and Shamos in [Shamos (76)].

Nearly all of the research in Computational Geometry has involved stationary sets of objects (points) in two or three dimensions. A recent investigation by Ottmann and Wood [Ottmann (84)] initiates the study of sets of points moving in one dimension with constant velocity and direction (either left or right). The authors refer to these as *"dynamical sets of points."* They propose efficient algorithms to solve the following coincidence problems: (1) Determine all pairs of points which collide at some time in the futures, and (2) for a given time t, determine all points which collide at time t. They also consider anihilation problems and order problems such as: (3) Assuming two colliding points anihilate each other, determine the order of anihilation, and (4) determine for a given time t, the sorted order of points at time t. The method used to solve all of these problems comes from the solution of the half-line intersection problem in two dimensions. It turns out that this problem is closely related to another well-known problem, determining all intersection points amongst n line segments. Solutions to this problem can be found in [Bentley (79)] and [Brown (81)].

Two other classes of problems intricately linked to intersection are visibility and decomposition. Decomposition in the most general sense involves breaking a simple polygon or polyhedron into a (hopefully minimal) number of non-overlapping parts. Chazelle [Chazelle (80)] has shown that breaking a simple polygon into a minimal number of convex parts depends on the number of vertices N showing a reflex angle (an interior angle of greater than 180 degrees). Since N is most often very small, the algorithms Chazelle presents are of practical use. Special case decomposition algorithms of some interest have also been designed to break polygons into star-shaped [Avis (81a)] or triangular subsets [Garey (78)]. Visibility has been investigated in [Chazelle (80)], [El Gindy (81)] and [Avis (81b)]. Algorithms have been devised for determining the region of a polygon P visible from a point inside P, and the interior of a polygon which is visible to an observer standing on (or patrolling along) a specified edge. In Chapter Seven we examine a related problem, determining the superrange of star-shaped and monotone polygons.

1.2.5. Summary

We have briefly touched upon a number of important problems that fall within the realm of computational geometry. The emphasis has been on pointing the reader in the direction of where "state-of-the-art" algorithms can be found in the literature. For further information the reader is referred to [Toussaint (80a)] and [Toussaint (82)] for a rather complete survey of the area.

1.3. Thesis Outline

The reader may expect three things from this thesis, a continuing synthesis of a new discipline, an exposition of several new algorithms, and a re-evaluation and speed-up of historically fast methods. We address specifically the need for efficient geometric algorithms, isolate the common algorithmic components, and present the techniques for designing them. Not only do we examine in detail several problems of significance, but we take considerable time actually implementing and comparing our algorithms to other methods if they exist. Our work is therefore somewhat more pragmatic than the majority of previous research in Computational Geometry.

The attempt is made to present the reader with the results in a single expository style unburdened by much of the rigor of present day mathematical proofs. While we recognize the value of intuition as the source of most mathematical discoveries, we will try to give as much motivation as we can before diving into the details of any proof. Figures are provided whenever we believe they will help to clarify the issues at hand.

The following is an outline of the course we have chosen to follow:

Chapter Two is an examination of some of the basic representation issues and computational models that will form the basis of our work.

Chapter Three is a study of distributive sorting and selection algorithms. While sorting is not a geometric problem per se, the large number of geometric algorithms requiring a sort step indicates the need for a very fast sort. Ample evidence will be given that the method we develop will be very efficient over a wide class of inputs. Selection, the problem of finding the k^{th} smallest element in an unsorted vector, will be investigated as a natural extension of our work on distributive sorting.

Chapter Four, "Hull Algorithms," contains an overview of four of the leading convex hull algorithms and a new algorithm for L_1 (city-block) geometry. The algorithms are those of Graham, Jarvis, Eddy, and Akl-Toussaint. Qualitative comparisons are made to determine under what circumstances each algorithm is likely to give its best and worst case performance. Empirical tests on a set of standard distributions are included to support the analysis. Suggestions are made which should be valuable regarding actual implementations.

Chapter Five commences with a discussion of a simple technique which eventually leads to modifications of two well-known algorithms for the minimum-area encasing rectangle and diameter of a set of points. The new technique we label the *Highpoint Strategy* because it may be used to list the vertex (vertices) which is (are) perpendicularly highest above each edge of a convex polygon in O(n) time. Continuing our work in L_1 geometry, we also present an O(n) algorithm to find the diameter of a set.

Chapter Six is a detour into the world of heuristic algorithms. Our work centers on geometrically motivated approximation algorithms for the

Euclidean and L_1 Traveling Salesman Problems. The central theme is to use hull algorithms to form convex rings each representing an optimal tour of some subgroup of points and then to merge these rings in a way that insures very few intersections yet produces a "close to" optimal tour. The emphasis is not so much on computationally efficient algorithms, but more on the quality of tours produced. Benchmarks are included to indicate how our algorithms perform against nearest insertion, farthest insertion, and the convex hull heuristic of Stewart [Golden (80)]. Avenues for future research are also outlined.

Chapter Seven provides further evidence that for certain classes of polygons, algorithm design and implementation is indeed easier. The question we answer in the affirmative is whether the superrange of a vertex (the vertices which can be seen from a specific vertex v) is an inherently easier task for polygons which are either star-shaped or monotone.

The final Chapter contains a summary of the most important findings of our work and entertains a discussion of future directions of research in Computational Geometry.

"Speed it seems to me, provides the
one genuinely modern pleasure."
- Aldous Huxley

Chapter 2

ALGORITHM DESIGN ISSUES

2.1. Introduction

The purpose of this Chapter is to examine some of the issues that arise in the design and analysis of algorithms and, in particular, geometric algorithms. The word *algorithm* has come to refer to a precise method which may by implemented on a computer for the solution of a problem. Although this is the definition we are most interested in, its meaning is actually more general in that it may refer to any special method for solving a certain kind of problem.

An algorithm has several properties which allows direct translation into a set of programming language statements. First, an algorithm is *finite,* it must consist of a reasonable number of steps, each of which may be carried out by one or more computer operations. Second, each step of an algorithm must be *definite,* which implies that the action to be carried out must be clearly specified to the machine or the user involved. The third property is *effectiveness* each step must be such that it can, at least in principle, be carried out by a person using pencil and paper in a finite amount of time. The fourth property is concerned with input/output; an algorithm must always produce one or more outputs and may have zero or more inputs which are supplied by the user or an external source. Finally, all algorithms must *terminate* after a finite number of operations.

When studying computer algorithms a number of important issues arise, including how to neatly and concisely express algorithms (terms such as "software engineering" and "elements of programming style" are often used to describe this technique), how to prove that an algorithm will terminate with the correct answer, how to determine the actual number of operations an algorithm uses during its execution, and testing and profiling computer programs. While it is beyond the scope of this dissertation to delve very deeply into any one of these areas, we will take the time to examine how geometric objects can be stored by computing devices. Additionally, we will define a model of computation under which our algorithms can be analyzed and discuss why the number of operations needed for a particular problem may be independent of specific algorithms.

2.2. Specification and Representation

One of the problems in dealing with geometric objects is that often they are mathematically defined by an infinite set of points. A *simple polygon* for instance is the union of n mutually adjoining line segments (a polygonal line L) and the interior points enclosed by L. Immediately we are faced with a dilema since we cannot hope to store an uncountable number of points within any physical device such as a computer. Does this mean that we are constrained to work only with countable or finite sets?

The answer is (fortunately) no, because often an object is *finitely specifiable.* In the case of polygons, we only need to list the ordered sequence of vertices.

The representation of a point inside the computer will involve storing the coordinates as an aggregate data type. In general, the choice of coordinate system cannot affect the asymptotic running time of any geometric algorithm since the model of computation will allow for the necessary transformations in constant time (see next Section).

A set of points in k dimensions may be specified by an unordered N by k array or a list of N vectors of size k. Line segments may be specified by their endpoints. However, care must be taken in the specification of a polygon by giving its vertices in the order in which they appear on the boundary. In two dimensional space a single array of size n by two will usually suffice. Often, however, new vertices will have to be added, in which case a doubly linked-list will save time since deletions and insertions will require constant time. In any case, the transformation from an array to a linked-list can be done in time proportional to the number of elements (linear time).

Geometric algorithm designers [Preparata (77)], [Shamos (78)] have developed a *standard-form* for simple polygons which is an attempt to avoid the multiplicity of representations that may result by listing any one of the n vertices first, followed by the remaining (n - 1) vertices in counterclockwise of clockwise order. A simple polygon is in standard form if its vertices occur in counterclockwise order, with all vertices distinct, and no three consecutive vertices collinear, beginning with the vertex that has the least y-coordinate. If two or more vertices have identical least y-coordinates, then the one that has the least x-coordinate is listed first.

The requirement that vertices be distinct is made in order to remove the degeneracy of zero length edges. A quadrilateral with a null edge would therefore be represented as a triangle. Collinear vertices are removed to avoid the problem of multiple representations of identical polygons. Listing the lexicographically least vertex first is designed to allow for the easy interface and comparison amongst several geometric algorithms.

It is possible to convert a polygon in non-standard form to one that is in standard form in linear time. The process is essentially a simple bookkeeping procedure [Shamos (78)]. Throughout this dissertation we will assume without mention that polygons presented as input are in standard form. Likewise, the output of our algorithms will also be in standard form.

2.3. Model of Computation

We are now ready to define the model (or machine) of computation under which our algorithms will run. This will allow us to determine the cost of various arithmetic operations and to estimate the total running time of an algorithm. The idea of using a theoretical machine where the cost of primitive operations can be measured accurately is made for several reasons. Foremost among these, is a need to prove upper and lower bounds on execution time. Another reason is to avoid dependence upon the time required by any one machine to perform a particular computation. It is better to make the time depend on the relative speeds of several machines. Therefore, a scheme that represents present day computers as closely as possible, while still permitting thorough analysis is preferred [Aho (74)].

The operations which can be accomplished by one or a few clock cycles on present day computers include arithmetic operations on integers: addition, multiplication, and division. Others might include arithmetic on floating point numbers (reals), comparisons, variable assignment, execution of procedure calls, and read and write operations. We augment the basic set of arithmetic operations to allow for the computation of square roots and trigonometric functions, since these may be required to represent the distance between two points or the orientation of geometric objects. (These functions may be computed by a suitable combination of the basic arithmetics.) We will assume that each of the basic operations takes one unit of time unless otherwise stated. The basic unit of storage is *the word* and it is capable of holding one integer or one real number.

The model we have adopted is similar to the random access machine (RAM) specified by Aho, Hopcroft, and Ullman in their book: *"The Design and Analysis of Computer Algorithms"* [Aho (74)].

2.4. *The Analysis of Algorithms*

2.4.1. *Analyzing Some Simple Programs*

With the computational model we have chosen, obtaining the computing time of an algorithm will be quite easy, especially when the algorithm has been coded in a high-level algebraic-like programming language such as FORTRAN, C, or PASCAL. The general procedure is to obtain a frequency count of the number of basic operations through an a priori analysis based on the size of the input data.

For example, consider the following three PASCAL programs:

```
PROGRAM one (input,output);
  VAR x,y : real;
  BEGIN
    read (x);
    read (y);
    x := x + y;
    write (x);
  END.

PROGRAM two (input output);
  VAR sum, i, n, number : integer;
  BEGIN
    read (n);
    sum := 0;
    FOR i := 1 TO n DO
```

```
      BEGIN
        read (number);
        sum := sum + number
      END;
    write (sum)
  END.

PROGRAM three (input);
  VAR i, j : integer;
      matrix : ARRAY [1 .. 100, 1 .. 100] OF integer;
  BEGIN
    read (n);
    FOR i := 1 TO n DO
      FOR j := 1 TO n DO
        IF i = j THEN matrix[i,j] := 1
                 ELSE matrix[i,j] := 0
  END.
```

Program one has four basic operations and thus the frequency count is 4. Program two sums n integer numbers. The frequency count is $(2n + 3)$ (we do not usually include a count for the looping statements, although to implement these in a low level language would require a comparison, jump, and addition for each iteration of the FOR loop). The final program initializes an n by n identity matrix. The frequency count for this program is $(2n^2 + 1)$.

The three programs illustrate the idea of *orders of magnitude.* Program one will always take constant time no matter what the inputs are, program two will always take time proportional to n operations, and program three takes time proportional to n^2 operations. Given three algorithms for solving the same task whose orders of magnitude are n, n^2, $n^2\log n$, we will naturally prefer the first since the second and third are progressively slower for large n. For small values of n we must be careful about making statements concerning time comparisons amongst several algorithms. In these cases the constant of proportionality in front of the leading term of the function which describes the running time of a brute-force algorithm is usually much smaller than for a more sophisticated algorithm which has a smaller order of magnitude. The prime goal for the algorithm designer is to produce an algorithm which is an order of magnitude faster than any other, or else to try to prove that it is not possible.

2.4.2. Asymptotic Notation

We will adopt a notation which is convenient for analyzing algorithms in terms of orders of magnitude: this is the *big-O* notation popularized by Knuth [Knuth (76)]. A function $f(n) = O(g(n))$ ("read as f of n equals big oh of g of n") if and only if there exist two positive constants c and n_0 such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$. This notation gives a feeling as to how much time a computation may take as a function of the number of inputs n. What we will usually attempt to determine through an a priori analysis is the function g(n). If f(n) is the actual time that the program takes, then we can usually estimate the value of c by actual performance profiling. For example, program two is O(n). After running three tests one of size 100, another of size 200, and the third of size 300, we might record the times of 1 second, 2 seconds, and 3 seconds respectively. Therefore, an accurate value of c would be 0.01. The value of c is dependent on the computer used in the computation. When we say that an algorithm is O(n) we mean that for increasing values of n, the resulting times will always be less then some constant times |g(n)|.

2.4.3. Performance Profiling

Performance profiling involves testing several programs (algorithms) to see which one is the fastest, or alternatively attempting to pinpoint which part of a program is running slowly. The reader should be aware that there are several problems with performance testing in general. One is that in a multiprogramming environment the clock times always include a certain fraction of the time needed to swap out the user's program on disk. This time will vary depending upon the number of jobs that are currently active in the system, and there is usually no way of determining how much time this takes. A second problem is that the operations used in the various algorithms will have different running times on different hardware. For example, on the CDC series of computers, floating point operations are known to execute much faster than on IBM series computers. Compilers can also affect, sometimes hideously, the running time of what is believed to be a very efficient implementation of an algorithm. We have used the IBM FORTRAN H compiler with the optimizing option (OPT=2) for most of our tests. Without the optimizing option most of the times were at least twice as large, and the relative times were in a few cases different.

2.5. Metric Distance

Most results in this dissertation will involve problems where the distance between two points (i,j) is given by the formula

$$d_2(i,j) = (|x_i - x_j|^2 + |y_i - y_j|^2)^{1/2}.$$

This is known as the *Euclidean* or *L_2 norm.* In a few cases we will consider problems in which another notion of distance applies. This is the *L_1 (Manhatten or city-block) norm* where the distance is given by

$$d_1(i,j) = |x_i - x_j| + |y_i - y_j|.$$

The L_1 and L_2 norms are special cases of the *pth order Minkowski metric (or L_p norm)* where distance is defined as:

$$d_p(i,j) = (|x_i - x_j|^p + |y_i - y_j|^p)^{1/p}, \quad 1 \le p \le \infty.$$

Throughout, we will assume the L_2 metric unless otherwise stated. In those cases when the L_1 metric is in effect we will denote the space in which the points reside by R_1. Likewise, when the L_2 metric is employed we will refer to the points as lying in the space R_2.

There are two major differences between the geometries of R_1 and R_2. One is that distance cannot be represented by a straight line. Instead the appropriate representation is a staircase-like sequence of connected orthogonal line segments; Fig. 2.1. Travel between two points A and B is restricted to rectilinear paths only. (This is the reason the L_1 norm is often referred to as the city-block metric.) The second difference is the uniqueness of these paths. If A and B do not have identical x or y-coordinates, then there is an infinite number of different orthogonal line segment sequences between these points whose distance equals $d_1(A,B)$.

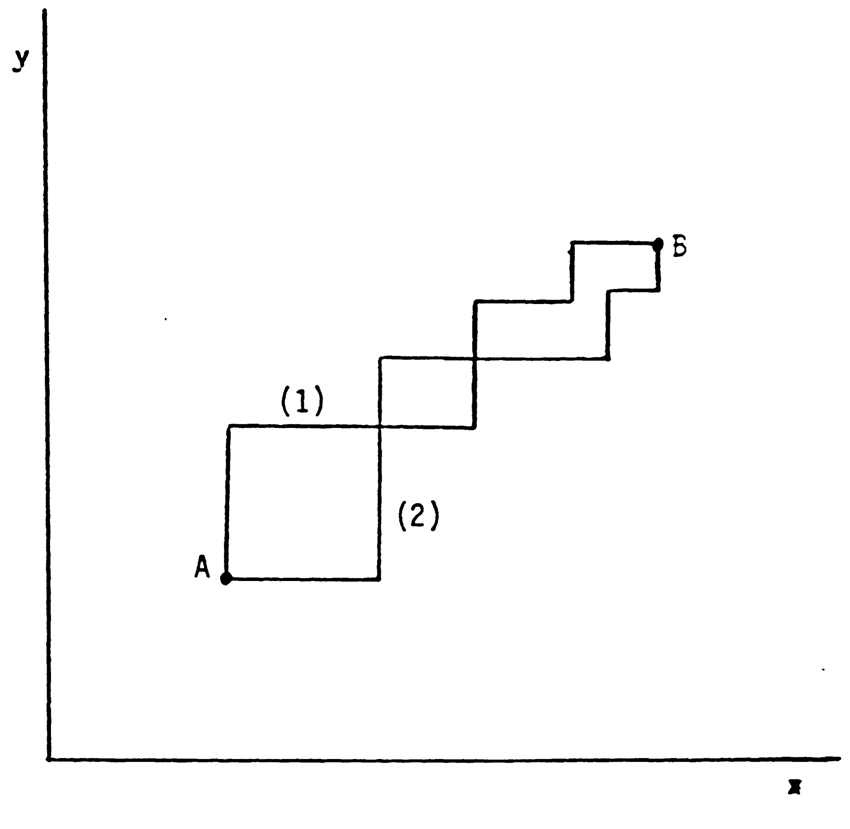


Fig. 2.1. Paths (1) and (2) represent different sequences of orthogonal line segments that could be used to connect points A and B.

Chapter 3

SORTING AND SELECTION BY DISTRIBUTIVE PARTITIONING

In this Chapter we apply the principle of distributive partitioning to solve two important interrelated problems, sorting a set of objects and selecting the k^{th} smallest element from a set. In the past few years distributive partitioning has been shown by a number of researchers to produce very fast sorting algorithms -- even faster that Quicksort which had been considered to be the fastest internal array sorter. Because distributive partitioning performs so spectacularly in conjunction with sorting, we also examine how the method can be applied to the problem of selection.

3.1. Introduction to Sorting

"Seldom he smiles, and smiles in such a sort ..."

- William Shakespeare, *Julius Caesar*

Sorting is the process of re-arranging a given set of objects into a given *order.* It is a problem that is both practically important and theoretically interesting. School children are taught to put the alphabet into order even before they learn how to combine letters to form words. Commercial applications abound in warehouses, phone books, cataloguing, and government filing systems. It has been estimated by various authors that somewhere between 20 and 25 percent of all data processing involves sorting. Hence, the economic impact of a new and faster sorting algorithm can be significant.

In the design and analysis of algorithms many general techniques were first conceived in the construction of sorting algorithms. Because sorting is an ideal subject to demonstrate the differences amongst a great diversity of algorithms, all having the same purpose, the topic is usually discussed quite extensively in most books on algorithm design. Sorting also serves to show how a very significant improvement may be obtained by the development of sophisticated algorithms when simple methods are readily available.

Sorting methods are usually classified as being *internal,* where the data resides in the random access memory of the computer, or *external,* where the data resides on any other physical device excluding random access memory such as disk or tape. In this Chapter we will focus primarily on one particular method of internal array sorting called *distributive partitioning sorting.* Our interest in this method stems from our desire to speed-up the Graham convex hull algorithm, which will be discussed in the next Chapter.

Before proceeding we introduce some definitions and notation to be used throughout the Chapter. We are given items

$$a_1, a_2, \ldots, a_k$$

where

$$\mathrm{key}(a_{k_1}) \le \mathrm{key}(a_{k_2}) \le \ldots \le \mathrm{key}(a_{k_n}).$$

"key" can be thought of as an *ordering function* which must be evaluated to determine the relative position of two items. In practice the key is usually the item itself or computed and stored as an explicit component of each item. We will call the collection of components (including the key) associated with each item a *record.*

It is common in practical applications (and some of these exist in computational geometry) to have multiword keys and even larger components comprising each record. If records are more than just a few words long it is best to keep an extra array of *pointers* to refer to the records indirectly. During the sorting process, instead of exchanging two records, only two one word pointers need to be exchanged. This type of sorting is often called *pointer sorting* or *address table sorting,* and can save time through decreased data movement.

Later on in the Chapter we will present the results of a series of extensive tests designed to assess the relative performance of our hybrid distributive partitioning algorithm (described in the next Section) versus a sophisticated Quicksort algorithm. These tests were carried out on several familiar distributions including the uniform, standard normal, gamma ($\alpha = 2.0$, $\beta = 1.0$), and exponential. Most of the tests involve straight exchange of keys sorting. Pointer sort versions of both methods were compared for uniformly dstributed keys.

3.2. Distributive Partitioning Sorting

3.2.1. Towards a New Sorting Algorithm

In a 1978 paper Robert Sedgewick [Sedgewick (78)] made the following statement: *"there is one (sorting) algorithm called Quicksort which has been shown to perform well in a variety of situations. Not only is this algorithm simpler than many sorting algorithms, but empirical and analytic studies show that Quicksort can be expected to be up to twice as fast as its nearest competitors. The method is simple enough to be learned by programmers who have no previous experience with sorting and those who do know other sorting methods should also find it profitable to learn about Quicksort."* (The Quicksort Sedgewick refers to is actually an improved version of the original method proposed in 1961 by C.A.R. Hoare [Hoare (61a)].)

It seemed that sorting was a dead problem. No new faster algorithms had been invented over the previous ten years, even though many researchers had tried to find such an algorithm, which suggested that possibly none existed. About the same time that Sedgewick's paper appeared a young Polish computer scientist, W. Dobosowiecz, suggested in another paper, *"Sorting by Distributive Partitioning"* [Dobosowiecz (78)], that since only comparison based methods (such as Quicksort) had been examined extensively in most studies it was reasonable to inspect the possibilities of another class of sorting algorithms based upon distributive methods.

Dobosiewicz then described an algorithm which has an O(n) expected case running time on uniformly distributed keys and a worst-case of O(nlogn). While this method runs faster than Quicksort for n > 250 on a CDC 6400 computer, preliminary tests we conducted on an IBM 370/158 indicated that performance was only slightly better for n > 1000. Both our tests and Dobosowiecz's were carried out using optimizing FORTRAN compilers. The difference in performance can be traced to the CDC machine's fast floating point hardware. As we shall see, the DPS algorithm can take advantage of this hardware because arithmetic operations must be executed to distribute each item into its proper position.

After re-evaluating Dobosowiecz's work several authors [Kowalik (81)] [van der Nat (80)] [Meijer (80)] concluded that his original procedure could be speeded-up by the process of ***hydridization.*** (Hydridization is defined to be the mixing of several different varieties to produce a stronger species.) These authors advanced several combination methods which sort partly by distribution and partly by comparison. One step of the original DPS algorithm required a time consuming O(n) selection algorithm to find the median. Besides avoiding expensive median calculation these hybrid sorts enjoy other important features. They are quite simple to state, analyze, and program. Furthermore, the linear expected-case running time is not limited to uniformly distributed inputs.

One drawback of DPS is the space required to sort. In most implementations between 3n and 4n storage locations are needed for input strings of size n. In the early days of computing having enough random access memory to carry out a computation was often a very important consideration. Today most modern "mainframes" have several megabytes of real storage, thus making DPS practical for applications of 100,000 records or more. The bottleneck rests in how fast operations can be carried out. This is commonly referred to as the *time-space tradeoff* by algorithm designers. What it means is that in some cases, a process can be speeded-up by inventing a new algorithm which manipulates a more sophisticated (space consuming) data structure or an old algorithm's data storage mechanisms can be replaced by more efficient ones.

Our algorithm, which was communicated earlier in [Allison (82)], is a super hybrid which includes some of the best features of the Kowalik-Yoo, van der Nat, and Meijer-Akl hybrids plus an additional idea employed by Sedgewick [Sedgewick (78)] to speed up Hoare's Quicksort.

3.2.2. The Algorithm

The sort, which we label Usort, because if performs most efficiently on uniform distributions of keys, requires three passes, first a distributive pass and then two comparison based passes. In the first pass the n records are partially sorted into $\lfloor n/m \rfloor$ boxes (where $\lfloor x \rfloor$ is the floor function, and m is a positive constant less than n/2), with record A_i placed into box j according to the formula

$$j := \lfloor ((key(A_i) - min)/(max - min))(\lfloor n/m \rfloor - e) + 1 \rfloor.$$

The quantity e (taken to be very small relative to the size of n) is needed to insure that the record with the maximum valued key is placed into box $\lfloor n/m \rfloor$ instead of box $(\lfloor n/m \rfloor+1)$ [Allison (81)]. The contents of two consecutive boxes are such that all the keys in the first one are guaranteed to be smaller than all the keys in the second one.

For the second pass each box which contains k or more records is partitioned by Sedgewick's "median-of-three" Quicksort [Sedgewick (78)] until all partitions are of size (k-1) or smaller. The value chosen for k should be about 9, although this value may vary depending upon the relative speed of comparisons and arithmetic operations on any one particular machine. The third pass consists of a single Insertion sort (see [Wirth (76)] for a description on Insertion sorting) over the entire input vector. In this way the stacking overhead associated with an Insertion sort on each individual box or partition can be avoided. Sedgewick has already used this technique with good success in conjunction with his "median-of-three" Quicksort.

3.2.3. Implementation and Storage Requirements

The records are not moved during the first pass. Instead, a singly linked-list is used to represent the items of each box. The linked-list requires a total of $(\lfloor n/m \rfloor+n)$ storage locations: $\lfloor n/m \rfloor$ for the list heads and n locations for the links. Assuming that the records are in array A, efficient pseudo code for distributing the records into $\lfloor n/2 \rfloor$ boxes, where each record is considered to be a real-valued key, would be

```
ndiv2 := n/2;
for i := 1 to ndiv2 do { initialize list heads }
  list_head[i] := 0;
constant := (ndiv2 - .001)/(A[max] - A[min]); { e = .001 }
for i := 1 to n do
  begin
    j := (A[i] - A[min]) * constant + 1.0; { expression truncated }
    link[i] := list_head[j];
    list_head[j] := i
  end
```

As a result of this computation each empty list will have a list head equal to zero and each non-empty list will have a terminating zero link. By making one pass through all the lists, the contents of the boxes may be quickly rearranged into a destination array B where passes two and three may be efficiently carried out. The total array storage required is $(3n+\lfloor n/m \rfloor)$.

3.2.4. Worst-case and Average-case Time Complexity

The worst-case time complexity will occur in the unlikely event that the keys follow a factorial distribution. All of the records with the exception of the one with largest key will fall into the first box and Quicksort will have to be applied to (n-1) records. Since the worst-case time complexity of Quicksort is $O(n^2)$ for groups of size n [Wirth (76)], it follows that the worst-case complexity of Usort is also $O(n^2)$.

We analyze the expected-case time complexity under the assumption that the input sequence consists of uniformly distributed keys. It takes O(n) time to find A(min), A(max), initialize the list heads, distribute the records into the created intervals, and rearrange the records into a form suitable for the second and third passes of the algorithm. For the second pass the time to sort a single box consisting of i records will be proportional to $i\log_2 i$, the expected-case running time of Quicksort [Wirth (76)]. Since the input is uniformly distributed the probability that an item belongs to a given group is $1/(n/m) = m/n$. The probability that a single box will consist of i items is obtained from a binomial distribution (the notation C(n,i) means the combination of n things taken i at a time)

$$P(i) = C(n,i)(n/m)^i[1 - (m/n)]^{n-i}.$$

The expected time to sort a single box of k or more items is

$$\Sigma_{i=k}^{n} (i\log_2 i)C(n,i)(m/n)^i[1 - (m/n)]^{n-i}.$$

The time to sort all the boxes is, therefore,

$$(n/m)\Sigma_{i=k}^{n} (i\log_2 i)C(n,i)(m/n)^i[1 - (m/n)]^{n-i}$$

$$= (n/m)\Sigma_{i=k}^{n}([(i\log_2 i)m^i n(n-1)\ldots(n-1+1)][1 - (m/n)]^{n-i})/(i!n^i)$$

$$< (n/m)\sum_{i=k}^{n} [(i\log_2 i)m^i]/i! = n\sum_{i=k}^{n} (\log_2 i)m^{i-1}/(i-1)!$$

$$< n\sum_{i=1}^{\infty} (\log_2 i)(m^{i-1})/(i-1)! < nme^m = O(n)$$

After the second pass all partitions and boxes will have (k-1) or less records. For insertion sort the worst-case will be when there are n/(k-1) of these groups, each of size (k-1). The time to order all these groups is

$$[n/(k-1)](k-1)^2 = n(k-1) = O(n),$$

since k will be small and fixed for any implementation of Usort. Summing over all steps the total time taken is O(n).

Many "real" applications involve the ordering of data which exhibit near uniform behavior. With regard to Usort, this means that after the first pass O(n) boxes will contain at least one item, but with low probability no single box will overload (become too populous). For distributions meeting these requirements Usort will be very fast. In the event that several boxes contain a significant number of items (with respect to the size of the input vector), the time to sort will be reasonable because of Quicksort's $O(n\log_2 n)$ expected-case time complexity. Pass two is a "fail-safe" mechanism which insures against all but the most pathological cases.

3.2.5. Modifications for Pointer Sorting

Usort is easily modified for pointer sorting. After the first pass an array of length n is needed to initialize the pointers. The auxiliary array B referred to in Section 3.2.3 is not needed. The storage requirements are therefore identical to the "straight-exchange of records" version. In passes two and three, comparisons will take the form

$$A[pointer[i]] < A[pointer[j]]$$

with the pointers being exchanged depending upon the outcome of the test. This indirect reference causes more overhead than a normal comparison between keys; however, the expected-case time complexity of Usort with pointers is still $O(n)$ for uniformly distributed keys.

3.2.6. Comparison With Other Distributive Methods

We expect our method to be somewhat faster than the other three distributive sorting hybrids for the following reasons:

(1) The van der Nat algorithm uses recursion and merge sorting. In most high level languages the code generated to handle the recursion produces extra overhead that could be avoided by using an explicit stack [Horowitz (76)]. Converting the van der Nat algorithm to a non-recursive version is not a trivial matter. The merge sort prevents a worst-case of $O(n^2)$, but it does produce some overhead.

(2) The Kowalik and Yoo method can be described recursively, and can be implemented quite easily using a stack unlike the van der Nat algorithm. However, we do not subscribe to the idea of applying a distributive pass more than once for the following reason. As a rough rule of thumb, one call to Quicksort takes the same amount of time as two calls to DPS on the same size vector. If DPS is called again on a large subfile (relative to the size of the original input vector) then it would have been better to call Quicksort in the first place.

(3) The Meijer and Akl method is closest in design to our method. They use Heapsort on boxes that contain more than the threshold number of items, while we use Quicksort. Thus their method has a worst-case of O(nlogn) which is better than our worst-case performance. However, our average-case performance should be better since Quicksort is known to be much faster than Heapsort on input vectors of size 100 or less [Wirth (76)]. Because of the low probability of the worst-case of Quicksort occuring we prefer our method.

3.2.7. Early Test Results and Discussion

The two algorithms described above, Usort (USORT) and Usort with pointers (UPSORT) were coded in FORTRAN and run on an IBM 3032 (FORTX,OPT=2) in March 1981. These were tested against FORTRAN implementations of Sedgewick's Quicksort (QSORT) and Quicksort with pointers (QPSORT) [Sedgewick (78)]. Copies of all tested programs can be found in Appendix 1. Pseudo-random variates were generated over the interval (0,1) by IMSL uniform random number generator GGUBS [IMSL (80)]. 5 realizations of 100 runs were made on all sample sizes. The timings taken were averaged and are summarized in Table 3.1. Standard deviations were calculated for each set of 5 test runs and appear in the parentheses to the right of each entry.

We experimented with different m and k before choosing values 2 and 8. We did not find these values to be a critical factor in obtaining good running times for the algorithm. For example, with m = 3 and k = 10 the total time was only about 2% higher. However, we did find that Usort ran 10% slower when m = 1. The reason for this behavior is that many of the lists were empty and maintenance of empty lists incurs extra overhead. This anomaly had also been reported by Kowalik and Yoo [Kowalik (81)] for their distributive paritioning hybrid sort.

Table 3.1. Sorting time (average of 5 realizations of 100 runs in hundredths of a second). Standard deviations within parentheses.

n	QSORT	USORT	QPSORT	UPSORT
250	39.4(.49)	28.6(.49)	58.8(.40)	42.8(.40)
500	87.4(.80)	57.8(.40)	130.2(.40)	86.6(.49)
1000	191.0(1.79)	114.8(.98)	286.2(1.17)	175.4(1.85)
2000	416.0(.63)	233.4(.80)	629.2(4.26)	350.8(1.86)
4000	894.0(4.04)	472.0(4.56)	1366.6(13.57)	702.4(6.83)

What we found is that Usort is a very fast sorting method for uniformly distributed keys; the times for USORT are 53-73% of the QSORT times on the sample sizes used in the test. As stated in [Allison (82)], better performance can be expected on machines with fast floating point hardware such as the CDC Cyber Series [van der

Nat (80)]. Our results compared favorably with Kowalik and Yoo's [Kowalik (82)]. However, they used sample sizes of 5000, 10000, and 50000 items. Extrapolation of the data in Table 3.1 would lead to even more impressive results than those obtained by Kowalik and Yoo.

3.2.8. Later Test Results

In 1981 Devroye and Klincsek [Devroye (81)] proved that distributions which have exponentially dominating tails, such as the normal, gamma, and exponential, can also be sorted in O(n) time by recursive distributive partitioning methods. Thus, any box containing more than a threshold number of items would be recursively sorted again by DPS, as opposed to our hybrid which invokes Quicksort to partition overly crowded boxes.

The acquisition of a new computer at Virginia Tech (an IBM 3081) in addition to our earlier study and the Devroye and Klincsek report indicated that a more elaborate test mechanism was needed to determine the performance characteristics of our hybrid on a variety of distributions. Instead of obtaining CPU timings, counts of certain fundamental operations were obtained by placing software counters into the USORT code. The same procedure was also carried out for Sedgewick's Quicksort. The set of operations included:

1. *Arithmetics* (Addition, Subtraction, Multiplication, Division)

2. *Assignment* (Replacing the value of a variable by either the value of another variable, or the value of a constant)
3. *Comparison* (Comparing the value of either two variables or a variable and a constant)
4. *Exchange* (equivalent to three assignments -- included because of other test involving the number of exchanges used by Quicksort)

Sample statements containing the various operations follow:

```
(1)  U = U + 1
(2)  I = L + 1
(3)  B(J-1) = V
(4)  SIZEUP = R - I + 1
(5)  J = R
(6)  B(R) = SWITCH
(7)  B(MIDDLE) = B(I)
(8)  STACK(1,DEPTH) = I
(9)  IF (V .GT. B(I)) GO TO 110
(10) IF (B(I) .LE. B(L)) GO TO 20
(11) IF (I .EQ. 0) RETURN
```

Statement (1) would count as one arithmetic operation. The assignment involved would not be counted since this operation is always performed as a side effect of an arithmetic operation. Statement (2) would also count as one arithmetic, statement (3) as one arithmetic, and statement

(4) as two arithmetics. Statements (5), (6), (7), and (8) are examples of assignment. The execution of any one of these would cause one to be added to the assignment counter. Statements (9), (10), and (11) would each be tallied as one comparison. No distinction was made between the use of simple variables and array variables or their types (real, integer, boolean) in any of the operations. An exchange would take the form

```
(12) SWITCH = A(J)
     A(I) = A(J)
     A(J) = SWITCH
```

Each exchange is equivalent to three assignments. These operations can only take place in Quicksort or the Quicksort pass of Usort. A separate count was included for comparison with other studies [Wirth (76)], [Sedgewick (78)] where the number of these critical operations has been either measured or theoretically determined.

In some cases the code contained composite statements such as

```
(13) IF (A(I+1) .GT. B) GO TO 10
(14) IF (A(I+1) .EQ. B(J-1)) I = J
```

In these (13) would count as one comparison and one arithmetic, and (14) would count as one comparison, two arithmetics, and one assignment. Do loops such as

```
DO 10 I = 1,N
```

```
      A(I) = B(I)
   10 CONTINUE
```

require special treatment. Rewriting the loop in a more primitive form yields

```
      I = 1
   10 IF (I .GT. N) GO TO 20
      A(I) = B(I)
      I = I + 1
      GO TO 10
   20 ---------
```

Thus, the DO loop construct requires (in this case) N comparisons and additions (not including the N assignments which must also be counted). Similar calculations can be made for all DO loops. The statements inside the loop body can be multiplied by the repetitive factor of the loop to yield the number of times they will be executed.

The tests were restricted to four distribution types: uniform, standard normal, gamma ($\alpha = 2.0$, $\beta = 1.0$), and exponential. IMSL routines GGUBS, GGNML, and GGAMR were used for random number generation [IMSL (80)]. For each sample size - distribution type combination, preliminary screening tests were undertaken to determine the optimum number of boxes which would minimize the number of fundamental operation counts; see Tables 3.2 through 3.7. Each entry in the tables represents the number of operations required for 1 run. These were derived from an average of 100 runs. We have not included the results from every one of our preliminary tests, instead the emphasis has been on providing the reader with the evidence which shows that USORT is not particularly sensitive to the number of boxes chosen for the distributive pass. We first determined the value of k such that kn is the number of boxes producing the approximate number of fewest operations for 100 sets of data over 100 runs. These are the "starred" entries in Tables 3.2 through 3.7. Tests were then run for the identical data using (k-.45)n, (k-.15)n, (k+.15)n, (k+.45)n, and (k+1.05)n boxes. (In some cases this data was already available from the optimization phase.) Because of cost, exceptions were made for 4000 and 8000 items where data was taken for (k-.45)n, (k+.45)n, and (k+1.05)n boxes only. Checking the Tables it appears that in our attempt to locate the optimum number of boxes we did not always succeed. For example, for the uniform distribution with 500 items we found the optimum at .65n boxes. It turns out that .50n boxes produces a marginally better result; see Table 3.3. This, we believe, supports our claim that USORT is not particularly sensitive to the number of boxes chosen for the distributive pass.

We now turn our attention to the actual number of specific operations required by USORT and QSORT for the various sample size - distribution type combinations. For each of the "starred" entries in Tables 3.2 through 3.7, an operation count breakdown is given in Tables 3.8 through 3.13 along with the corresponding operation count breakdown for QSORT. CPU timings on an IBM 3081 have also been included to allow the reader to more easily determine the relative

Table 3.2. Operation counts for Usort for different numbers of boxes, n = 250 items. Starred entries indicate approximate optimum as determined by our preliminary screening tests.

Uniform Distribution

No. of boxes	Operation count
.20n	10779
.50n	6831
.65n*	6817
.80n	6951
1.10n	7324
1.70n	8156

Normal Distribution

No. of boxes	Operation count
.35n	8689
.65n	7138
.80n*	7153
.95n	7294
1.25n	7638
1.85n	8440

Gamma Distribution
($\alpha = 2.0$, $\beta = 1.0$)

No. of boxes	Operation count
.45n	9294
.75n	7554
.90n*	7571
1.05n	7590
1.35n	7832
1.95n	8631

Exponential Distribution

No. of boxes	Operation count
1.10n	8340
1.40n	8290
1.55n*	8389
1.70n	8502
2.00n	8843
2.60n	9653

Table 3.3. Operation counts for Usort for different numbers of boxes, n = 500 items. Starred entries indicate approximate optimum as determined by our preliminary screening tests.

Uniform Distribution

No. of boxes	Operation count
.20n	21186
.50n	13407
.65n*	13606
.80n	13880
1.10n	14628
1.70n	16299

Normal Distribution

No. of boxes	Operation count
.40n	19513
.70n	14588
.85n*	14493
1.00n	14738
1.30n	15432
1.90n	17032

Gamma Distribution
($\alpha = 2.0$, $\beta = 1.0$)

No. of boxes	Operation count
.75n	16497
1.05n	15395
1.20n*	15594
1.35n	15756
1.65n	16506
2.25n	18127

Exponential Distribution

No. of boxes	Operation count
1.20n	18396
1.50n	17523
1.65n*	17598
1.80n	17670
2.10n	18195
2.70n	19686

Table 3.4. Operation counts for Usort for different numbers of boxes, n = 1000 items. Starred entries indicate approximate optimum as determined by our preliminary screening tests.

Uniform Distribution

No. of boxes	Operation count
.20n	40165
.50n	27923
.65n*	27322
.80n	27756
1.10n	29253
1.70n	32582

Normal Distribution

No. of boxes	Operation count
.50n	40994
.80n	30550
.95n*	30120
1.10n	30192
1.40n	31515
2.00n	34687

Gamma Distribution
(α = 2.0, β = 1.0)

No. of boxes	Operation count
.90n	34764
1.20n	32299
1.35n*	32273
1.50n	32654
1.80n	33953
2.40n	37167

Exponential Distribution

No. of boxes	Operation count
1.40n	40103
1.70n	37085
1.85n*	37380
2.00n	37284
2.30n	37799
2.90n	40675

Table 3.5. Operation counts for Usort for different numbers of boxes, n = 2000 items. Starred entries indicate approximate optimum as determined by our preliminary screening tests.

Uniform Distribution

No. of boxes	Operation count
.20n	79325
.50n	56087
.65n*	55831
.80n	55449
1.10n	58469
1.70n	65151

Normal Distribution

No. of boxes	Operation count
.55n	96130
.85n	62193
1.00n*	60684
1.15n	61056
1.45n	63537
2.05n	69987

Gamma Distribution
(α = 2.0, β = 1.0)

No. of boxes	Operation count
.95n	80038
1.25n	67260
1.40n*	66192
1.55n	66882
1.85n	69026
2.45n	75001

Exponential Distribution

No. of boxes	Operation count
1.90n	83711
2.20n	80326
2.35n*	79873
2.50n	79655
2.80n	81679
3.40n	87401

Table 3.6. Operation counts for Usort for different numbers of boxes, n = 4000 items. Starred entries indicate approximate optimum as determined by our preliminary screening tests.

Uniform Distribution

No. of boxes	Operation count
.20n	156977
.65n*	112451
1.10n	117486
1.70n	130696

Normal Distribution

No. of boxes	Operation count
.75n	160516
1.20n*	126477
1.65n	132243
2.25n	144608

Gamma Distribution
($\alpha = 2.0$, $\beta = 1.0$)

No. of boxes	Operation count
1.25n	153449
1.70n*	138367
2.15n	145363
2.75n	157566

Exponential Distribution

No. of boxes	Operation count
2.35n	174686
2.80n*	169832
3.25n	174441
3.85n	185129

Table 3.7. Operation counts for Usort for different numbers of boxes, n = 8000 items. Starred entries indicate approximate optimum as determined by our preliminary screening tests.

Uniform Distribution

No. of boxes	Operation count
.25n	286431
.70n*	225760
1.15n	235970
1.75n	262887

Normal Distribution

No. of boxes	Operation count
.85n	371209
1.30n*	261386
1.75n	270389
2.35n	294241

Gamma Distribution
($\alpha = 2.0$, $\beta = 1.0$)

No. of boxes	Operation count
1.45n	334510
1.90n*	293939
2.35n	300958
2.95n	325385

Exponential Distribution

No. of boxes	Operation count
2.90n	373660
3.35n*	368468
3.80n	377345
4.40n	397996

Table 3.8. Operation Counts for Quicksort and Usort, n = 250 items. AR = arithmetics, AS = assignments, CP = comparisons, EX = exchanges, Time is CPU seconds on an IBM 3081.

Uniform Distribution

	QSORT	USORT (.65n boxes)
AR	2952	2811
AS	675	1555
CP	2855	2467
EX	420	0
Time	.0016	.0014

Normal Distribution

	QSORT	USORT (.8n boxes)
AR	2970	2925
AS	672	1614
CP	2875	2589
EX	420	0
Time	.0016	.0014

Gamma Distribution
(α = 2.0, β = 1.0)

	QSORT	USORT (.9n boxes)
AR	2953	3044
AS	670	1679
CP	2866	2683
EX	424	1
Time	.0017	.0015

Exponential Distribution

	QSORT	USORT (1.55n boxes)
AR	2971	3365
AS	677	1837
CP	2872	3189
EX	420	1
Time	.0016	.0017

Table 3.9. Operation Counts for Quicksort and Usort, n = 500 items. AR = arithmetics, AS = assignments, CP = comparisons, EX = exchanges, Time is CPU seconds on an IBM 3081.

Uniform Distribution

	QSORT	USORT (.65n boxes)
AR	6496	5605
AS	1348	3089
CP	6429	4930
EX	962	0
Time	.0035	.0027

Normal Distribution

	QSORT	USORT (.85n boxes)
AR	6503	6027
AS	1345	3313
CP	6435	5307
EX	959	1
Time	.0036	.0030

Gamma Distribution
(α = 2.0, β = 1.0)

	QSORT	USORT (1.2n boxes)
AR	6462	6353
AS	1343	3477
CP	6398	5829
EX	963	1
Time	.0038	.0030

Exponential Distribution

	QSORT	USORT (1.65n boxes)
AR	6495	7014
AS	1348	3814
CP	6427	6609
EX	960	2
Time	.0036	.0034

Table 3.10. Operation Counts for Quicksort and Usort, n = 1000 items. AR = arithmetics, AS = assignments, CP = comparisons, EX = exchanges, Time is CPU seconds on an IBM 3081.

Uniform Distribution

	QSORT	USORT (.65n boxes)
AR	14219	11132
AS	2698	6143
CP	14315	9756
EX	2157	0
Time	.0077	.0055

Normal Distribution

	QSORT	USORT (.95n boxes)
AR	14165	12971
AS	2686	7071
CP	14320	12109
EX	2161	2
Time	.0083	.0062

Gamma Distribution
(α = 2.0, β = 1.0)

	QSORT	USORT (1.35n boxes)
AR	14206	12971
AS	2686	7071
CP	14320	12109
EX	2161	2
Time	.0083	.0062

Exponential Distribution

	QSORT	USORT (1.85n boxes)
AR	14183	14721
AS	2695	7983
CP	14288	13957
EX	2165	3
Time	.0078	.0071

Table 3.11. Operation Counts for Quicksort and Usort, n = 2000 items. AR = arithmetics, AS = assignments, CP = comparisons, EX = exchanges, Time is CPU seconds on an IBM 3081.

Uniform Distribution

	QSORT	USORT (.6n boxes)
AR	30798	22309
AS	5398	12266
CP	31456	19430
EX	4787	0
Time	.0165	.0111

Normal Distribution

	QSORT	USORT (1.0n boxes)
AR	30705	24857
AS	5392	13586
CP	31384	22239
EX	4799	2
Time	.0169	.0123

Gamma Distribution
(α = 2.0, β = 1.0)

	QSORT	USORT (1.4n boxes)
AR	30776	27211
AS	5387	14825
CP	31452	24980
EX	4794	2
Time	.0176	.0129

Exponential Distribution

	QSORT	USORT (2.35n boxes)
AR	30789	31508
AS	5389	16961
CP	31476	31033
EX	4806	4
Time	.0169	.0154

Table 3.12. Operation Counts for Quicksort and Usort, n = 4000 items. AR = arithmetics, AS = assignments, CP = comparisons, EX = exchanges, Time is CPU seconds on an IBM 3081.

Uniform Distribution

	QSORT	USORT (.65n boxes)
AR	66337	45686
AS	10786	25109
CP	68631	39831
EX	10539	0
Time	.0355	.0227

Normal Distribution

	QSORT	USORT (1.2n boxes)
AR	66405	52025
AS	10787	28321
CP	68700	47348
EX	10539	3
Time	.0366	.0256

Gamma Distribution
(α = 2.0, β = 1.0)

	QSORT	USORT (1.7n boxes)
AR	66190	56756
AS	10775	30834
CP	68517	53619
EX	10555	3
Time	.0379	.0273

Exponential Distribution

	QSORT	USORT (2.8n boxes)
AR	66130	66925
AS	10781	35911
CP	68451	67645
EX	10555	5
Time	.0364	.0332

Table 3.13. Operation Counts for Quicksort and Usort, n = 8000 items. AR = arithmetics, AS = assignments, CP = comparisons, EX = exchanges, Time is CPU seconds on an IBM 3081.

Uniform Distribution

	QSORT	USORT (.7n boxes)
AR	141698	91364
AS	21571	50151
CP	148192	80687
EX	22983	0
Time	.0761	.0478

Normal Distribution

	QSORT	USORT (1.3n boxes)
AR	142125	106111
AS	21572	57637
CP	148618	97455
EX	22984	3
Time	.0786	.0544

Gamma Distribution
(α = 2.0, β = 1.0)

	QSORT	USORT (1.9n boxes)
AR	142141	116371
AS	21583	63056
CP	148621	111995
EX	22981	3
Time	.0814	.0572

Exponential Distribution

	QSORT	USORT (3.35n boxes)
AR	141841	142126
AS	21569	75875
CP	148360	148496
EX	23002	5
Time	.0783	.0730

performance characteristics amongst the different distribution types.

Usort required fewer total operations for every distribution type except the exponential on sample sizes of 250 and 500 items. The CPU timings mirror this observation. As the number of items increased, the number of boxes must also be increased to obtain optimum behavior for all distribution types except the uniform. The reason this must be done can be attributed to certain boxes becoming more and more overcrowded as the sample size increases. For example, in the normal distribution, boxes near the median will tend to become crowded, while boxes several standard deviations from the median will contain relatively fewer items. To stop this overcrowding the number of boxes can be increased which in turn implies that more empty lists must be maintained. However, this appears to be cheaper than passing several overcrowded boxes to the Quicksort phase or fuller boxes to the Insertion sort pass. The low number of exchanges recorded for Usort support this conclusion.

For each sample size - distribution type combination the time for USORT and QSORT can be modeled by a pair of equations in three variables. For example, take the uniform distribution and 250 items. Under this model of computation

$$T_{QSORT} = 2952AR + 1935AS + 2855CP$$
$$T_{USORT} = 2811AR + 1555AS + 2467CP.$$

This formulation allows us to answer questions about the relative speed of the two algorithms based upon the three types of operations: arithmetics, assignments, and comparisons. In the example above USORT required fewer operations for all three types. Therefore, USORT will always be faster than QSORT on any machine. As another example, consider the exponential distribution with 500 items:

$$T_{QSORT} = 6495AR + 4128AS + 6427CP$$
$$T_{USORT} = 7014AR + 3814AS + 6609CP$$

For Machine A let the time of an assignment be one-half the time of an arithmetic (AS = .5AR), and let the time for a comparison be the same as an arithmetic (CP = AR). Then, T_{QSORT} reduces to 14986AR versus 15530AR for T_{USORT}. Clearly, QSORT is the faster of the two algorithms on Machine A. On the other hand, for Machine B let the time of an assignment be three times slower than an arithmetic (AS = 3AR), and the time of a comparison equivalent to the time of an arithmetic (CP = AR). In this case T_{QSORT} reduces to 25306AR versus 25065AR for T_{USORT} which makes USORT just slightly faster then QSORT.

It should be clear that we have established a model which allows us to be very analytical about the time required to sort. What makes the model particularly attractive is that it allows us to pinpoint where the bulk of the computation takes place in the two algorithms. Furthermore, if other types of operations, such as jumps and subroutine calls need to be monitored, these can easily be added to the model. The model allows us to ascertain the speed of USORT and

QSORT on any new machine. Because computer architectures are always being modified and updated, the model takes on increased value as a forecasting tool. This could conceivably lead to a decision on whether to purchase a new computer.

Another question arises from the tabulated data. What about recursively partitioning boxes by DPS? Would this technique speed-up the running time of Usort for distributions such as the exponential, with perhaps a small loss in performance for uniformly distributed items? Although we did not implement a recursive DPS algorithm, the number of exhanges in the Tables indicate that for uniformly distributed items such an algorithm would perform with almost equivalent speed to our hybrid (because there would be very few recursive calls). For the other distribution types more recursive calls would have to be made because of the higher population of some boxes. This would cause some items to be distributed again (or several times). It appears that the total number or items redistributed would have to be very roughly around n or smaller if a recursive DPS algorithm is to run as fast as QSORT, since QSORT was never twice as slow as USORT in any of our tests (CPU time).

A final question is: Can we determine a good choice for the number of boxes regardless of distribution type, or number of items being sorted? The answer seems to be yes if we only consider the uniform, normal, and gamma distributions. A good choice might be around 1.2n boxes. Using this figure, Usort required 47389 arithmetics, 25772 assignments, 45942 comparisons, and 0 exchanges on a sample size of 4000 uniformly distributed items, still significantly better than Quicksort. On the other hand, for the gamma distribution the number of operations required was 70541 arithmetics, 38297 assignments, 55618 comparisons, and 17 exchanges, which led to a running time just slightly less than Quicksort's time.

Future research should consider a number of important issues. One is: Will a recursive DPS algorithm be as effective as Quicksort on distributions where the expected number of items falling into one or several boxes is quite large? Second, a study should be carried out to theoretically estimate the average number of operations taken by Usort for different distribution types. A third avenue of future research might involve non-uniformly sized boxes. This would require an initial pass over the n items trying to gain some statistical information about their distribution.

What impact will our results have on geometric algorithms? It turns out that the distribution of angles that must be sorted in the Graham convex hull algorithm will in many cases follow a rectangular density, not too unlike the standard normal distribution. Therefore, we can recommend Usort in these situations.

Some authors have downplayed the significance of distributive sorting methods [Baase (78)] [Huits (79)]. However, our results indicate to the contrary. We believe that any significant increase in the sorting speed of internal files (say on the order of 5% or more) is of major importance to the computing community.

3.3. *Selection by Distributive Partitioning*

"I have called this principle, by which each slight variation, if useful, is preserved, by the term Natural Selection."

- Charles Darwin, *Origin of Species*

3.3.1. *Introduction to Selection*

The problem of *selection,* finding the k^{th} smallest element from an unordered set, is closely related to sorting. An obvious statistical application involves finding the median. One popular selection algorithm is based upon the partition operation of Quicksort [Floyd (75)]. Therefore, it is natural to question if the technique of distributive partitioning might also yield a particularily efficient selection algorithm. In what follows we propose, analyze, and test a new selection algorithm based upon distributive partitioning.

3.3.2. *The Algorithm*

Consider the problem of finding the k^{th} smallest element in the set A with elements a_1, a_2, ..., a_n. The method of distributive partitioning is used to place the elements into boxes b_j; j = 1, 2, ..., B where B is the number of boxes, so that all the elements in box b_j

are less than all the elements in box b_{j+1}. A count is kept of the unsorted elements in each box and only the box containing the k^{th} smallest element needs to be considered further.

Algorithm KSMALL

Step 1. Find the maximum (max) and minimum (min) elements in A.

Step 2. If max = min, stop. The k^{th} smallest element has been found.

Step 3. Distribute the elements of A into boxes $b_1, b_2, \ldots, b_B$ and count the number of elements $|b_j|$ in box b_j. An element will belong to box b_j if j is the integer result of the computation

$$j = \lfloor [(a_i - \min)/(\max - \min)](B - e) + 1 \rfloor .$$

The quantity e (taken to be very small relative to the size of the elements) is needed to insure that the minimum and maximum are placed into the first and last boxes respectively.

Step 4. If $|b_1| \geq k$, then the k^{th} smallest element must be in box b_1. If $|b_1| + |b_2| \geq k$, then the k^{th} smallest element must be in box b_2. In this way the box which contains the kth smallest element is found. Steps 1 through 4 are then repeated recursively on this box.

The implementation of algorithm KSMALL can be efficiently carried out by representing the elements of each box as a singly linked list. Also, the recursion can be eliminated by using an explicit stack.

3.3.3. Complexity Analysis for Uniform Distributions

If the time for a comparison is C, a multiplication M, a division D and an addition/subtraction S then the average case time complexity is the sum of the times for the various steps. For each step the times are as follows:

Step 1: $T_1(n) = (3/2)Cn$

Step 3: $T_2(n) = (2S + M)n + 2S + D$

Step 4: $T_4(n) = (1/2)SB$

The total time complexity will therefore satisfy the recurrence relation

$$T(n) = ((3/2)C + 2S + M)n + 2S + D + (1/2)SB + T(n/B)$$

The solution of this recurrence relation is O(n).

For a uniform distribution represented by this recurrence relation it may be easily shown that a suitable choice for the number of boxes at each recursive step is $c\sqrt{n}$ where c depends on the speed of the various arithmetic and logical operations.

It is clear that for a uniform distribution and $k > (1/2)n$ a slight improvement may be achieved by starting the box count in step 4 at box b_B rather than box b_1.

The worst case complexity will occur in the unlikely event that the value of the elements follow a factorial distribution. In this case with

$B \leq n$ all elements except the largest will fall into the first box and the selection algorithm will have to be applied recursively to $(n-1)$ elements. The worst case complexity will therfore be $O(n^2)$. This behavior is similar to the Quicksort method for sorting elements in which the partitioning element is chosen so as to reduce the unsorted set by one element.

3.3.4. Results and Discussion

The selection algorithm described above (KSMALL) was coded in FORTRAN and run on an IBM 3032 (FORTX,OPT=2). It was tested against the algorithm SELECT [Floyd (75)] which has average case time complexity of 1.5n and is known to be superior to both FIND [Hoare (61)] and PICK [Blum (73)]. The FORTRAN version of SELECT, which was used, included the improvements suggested in [Brown (76)]. In the test the median element of a uniform distribution was found. The initial input vector was also ordered so that all elements less than or equal to a_k occupied positions $a_1, \ldots, a_{k-1}$ and all elements greater than or equal to a_k occupied positions $a_{k+1}, \ldots, a_n$. For KSMALL the median element will take longest to find since, if $k \geq (1/2)n$, counting of the boxes can begin with box b_B.

Table 3.14. Selection time (secs. for 100 runs)

n	KSMALL	SELECT
500	0.60	0.36
1000	1.16	0.71
2000	2.29	1.24
4000	4.51	2.32
8000	9.07	4.57

The results of the test are given in Table 3.14. These show that, as expected, KSMALL is clearly O(n) but that it is not as efficient as SELECT. This is due to the individual sampling of each element in KSMALL rather than the group sampling carried out in SELECT.

The timings in Table 3.14 were obtained with B, the number of boxes at each recursive step, set equal to $c\sqrt{n}$. For the IBM 3032 a suitable value of c was found to be 4.0. For other machines this value will be different since it depends on the relative timings of multiplications and additions.

KSMALL was not tested against other O(n) selection algorithms because of the theoretical superiority of SELECT (order 1.5n) over FIND (order 3.39n), PICK (order 5.43n) and the unimplemented algorithm described in [Schonhage (76)] (order 3n). We expect that the performance of KSMALL is comparable to that of FIND. It is clearly outperformed by SELECT in the case of selection of a single element.

3.3.5. Multiple Selection

Multiple selection involves finding more than one k^{th} smallest element in an unordered set. Let m denote the number of k^{th} smallest items that will be found. For this problem we would expect KSMALL to outperform SELECT (we are of course refering to algorithms based upon KSMALL and SELECT) when m is larger than (say) 5 since the initial distributive pass of KSMALL partitions the elements into many boxes of approximately the same size. SELECT, on the other hand, essentially distributes the items into two boxes (not necessarily of the same size). Therefore, repeated application of KSMALL on several very small sized array partitions (or boxes) will be faster than repeated application of SELECT on much larger array partitions.

A performance test was conducted on an IBM 3081 (FORTX, OPT=2) to see if indeed a multiple selection algorithm based upon KSMALL (KSMALLM) is faster than a similar algorithm based upon the technique used in SELECT (SELECTM). The results of the test which appear in Table 3.15 support our qualitative analysis. The time to sort n items using USORT is also given in the table. It is clear that there is only a small range of m values for which it will be better to call KSMALLM than USORT. On the IBM 3081 it looks as though m can be no larger than 5 percent of n before it is better to sort.

Implementation of both KSMALLM and SELECTM involved updating an inorder binary search tree to store away and recall the position of previously partitioned boxes or array segments as required by each algorithm. In this way we avoided searching all but the minimal number of elements for each k^{th} smallest item found.

Table 3.15. Selection time (secs. for 100 runs)

n,m	SELECTM	KSMALLM	USORT
500,4	0.19	0.21	0.27
500,8	0.27	0.23	0.27
500,32	0.44	0.30	0.27
1000,8	0.53	0.42	0.55
1000,16	0.68	0.44	0.55
1000,64	1.06	0.62	0.55
2000,16	1.50	0.83	1.11
2000,32	1.82	0.92	1.11
2000,128	2.57	1.25	1.11
4000,32	3.87	1.70	2.27
4000,64	4.55	1.90	2.27
4000,256	5.96	2.49	2.27

Chapter 4

HULL ALGORITHMS

4.1. Definition of the Convex Hull

Consider a set of points in the plane. Is there a fundamental geometric entity that defines the boundary of the point set? Our search for such a structure is motivated by our desire to treat all the points of the set as a whole entity. For example, we might want to determine whether a line ℓ passes through the interior (or "belongs with") a set of points; Fig. 4.1. This might lead us to formulate the following definition: a line ℓ belongs with a set of points S if there exist at least two points on opposing sides of ℓ. Computing whether ℓ belongs with a set S could involve checking the relative position of every point in S against the line ℓ. Intuitively, it seems that we could eliminate some of these points from the test if we knew the special points that make up the boundary of the set. It turns out that these special points can be characterized by a fundamental geometric property called *convexity*.

If a line is moved in from infinity towards the point set it will eventually strike a point in the set. If enough of these lines are moved in from all different directions then the boundary points will be defined. If we connect up these points so they form a non-intersecting simple polygon, then this polygon will have the property of convexity. All of the points in the point set will be on the same side of each edge

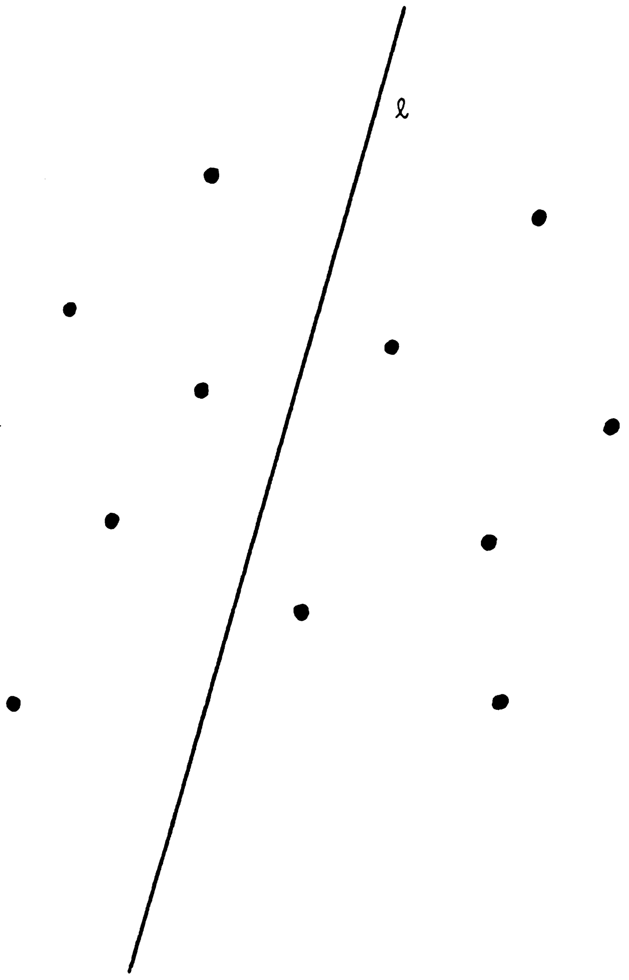


Fig. 4.1. Does ℓ belong with the set of points?

of this polygon. Two points inside any convex polygon will have the property that if a line is drawn between them then it will be entirely contained within the polygon. Triangles, Squares, and Rectangles are all convex polygons.

Those points which make up the vertices of the convex polygon define an infinite locus of points which has been given a special name: *the convex hull.* Any point that is an element of the convex hull is an interior point of the set. Two formal (and equivalent) definitons of the convex hull are often found in the literature. These are (1) the convex hull of a set of points in the plane is the minimum area convex polygon containing the set and (2) the convex hull is the shortest perimeter simple polygon that contains the set of points; Fig. 1.1.

4.2. Representation and Other Considerations

Before discussing how to compute the hull we must consider how to represent (or store) this geometric structure. Any algorithm to compute the hull will have to mark those points in the original set that make up the hull. As input we can expect the points to be contained in an array of records (in a language like PASCAL) or two arrays, one holding the x coordinates, and the other holding the y coordinates (in a language like FORTRAN). Output could take several forms: one could be an array of points that are the convex hull vertices (not in any particular order), another possible representation would be an array with the additional property that the vertices are sequenced as they appear in order along the perimeter of the hull.. Still another possibility is to link-up, using integer pointers, those points in the original input arrays that are vertices of the hull (this type of representation has many applications in computer science and is often refered to as a *linked-list*).

Depending upon the application we might want to use any one of these output representations. The most useful is the linked-list, and the one we subscribe to, because all of the other representations can be efficiently obtained from it in at most O(n) time. Many other geometric algorithms require as a preprocessing step convex hull computation. For these the linked-list representation has proven to be the most flexible because often additional points must be merged into the hull and the linked-list provides a convenient mechanism for doing this in constant time. In any case, it has been shown [Shamos (78)] [Yao (81)] that computing the ordered sequence of vertices on the hull *(the ordered convex hull)* is not any harder than just identifying the vertices. We will see that in the algorithms we discuss that as a side effect of finding the vertices we can always build the ordered hull.

Since the word "ordered" has crept into our discussion it should come as no surprise that convex hull algorithms are closely related to sorting algorithms. In fact, every known convex hull algorithm has an analog sorting procedure. Finding the convex hull is really a two-dimensional sorting problem. However two things make computing the convex hull somewhat different than sorting. One is that usually when finding the hull we are forced to throw away some of the points -- those that will not be on the hull. The second difference is that instead of comparisons between two keys we have to use more expensive functions to determine where whether a point is on the hull.

In the next three Sections we will investigate four different algorithms for computing the ordered hull. We will see that the major difference in their relative speeds depends largely on the number of vertices that finally end up on the hull as well as the initial distribution of points in the plane.

4.3. The Graham Algorithm

The first method we examine is due to R.L. Graham [Graham (72)] (see [Schechter (82)] for an interesting biography on Graham's life). Graham's landmark paper is historically significant because it describes not only a method to solve a geometric problem, but an efficient one as well. Up until the early 1970's researchers had concentrated on finding "a" solution to a problem. They did not even consider addressing the problems of time and space complexity in their solutions. With the maturing of the field of the analysis of algorithms and the importance of real-time computing this philosophy has changed.

4.3.1. The Method

In a nutshell the algorithm is very simple (leaving out a few details which we will examine shortly). Perform a polar sort on the point set about a point z which is guaranteed to be on the hull or in the interior of the hull; Fig. 4.2a. The ordered points may then be thought of as the vertices of a simple polygon (just connect up the ordered points to form the edges of the polygon); Fig. 4.2b. All that remains is to take this polygon and eliminate any vertex whose common edges form a reflex angle (greater than 180 degrees) with respect to the interior of the polygon. (It is helpful to think that a new polygon is being redrawn each time a point is eliminated.)

4.3.2. Implementation Details

By considering how to implement the Graham convex hull algorithm we can observe the process of step-wise refinement in the solution of a problem.

The first problem we encounter is how to order the points. There is at least one obvious solution. Take the points and convert them to (r,θ) polar coordinates and use θ as the key for the sort. Knowing that this will involve the use of costly trig functions immediately suggests that we should look for an alternative. A candidate for such a function is to let z be the bottommost point of the set (min y-coordinate) [Anderson (78)] and order the points by a variation of the

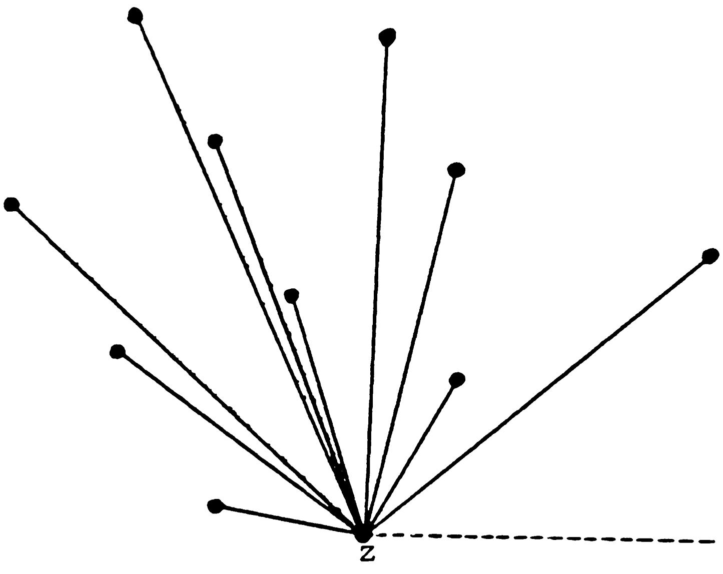


Fig. 4.2a. Ordering the points by polar angle.

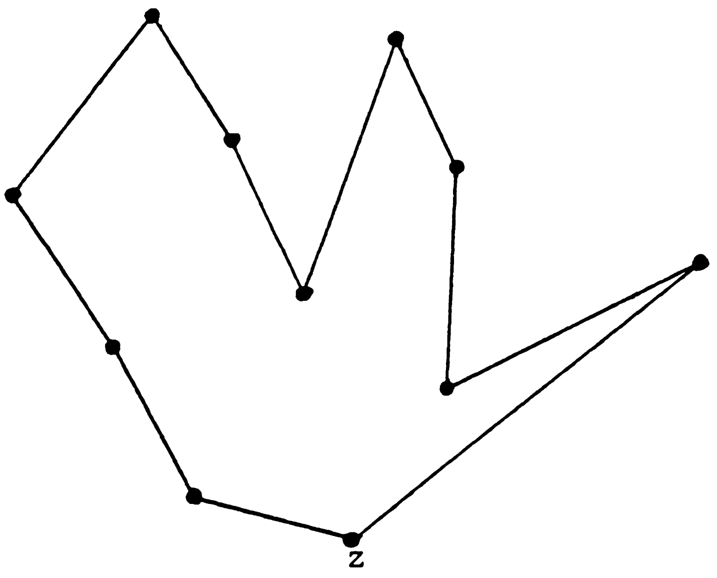


Fig. 4.2b. Forming a simple polygon from the ordered points of fig. 4.2a.

tangent formula [Noga (81)] as follows:

```
if $(x_i \neq x_z)$ or $(y_i \neq y_z)$
  then $\psi_i := -(x_i - x_z)/(|x_i - x_z| + (y_i - y_z))$          (1)
  else $\psi_i := -2$
```

The major advantage of using this formula, besides cost, is that division by zero will never occur. We could have used any point as the anchor for the sort and still used a variant of the above equation. However, using the bottommost point has another beneficial side effect as we shall see later.

From our discussion in Chapter Three it follows that the actual sorting step should be carried out by a pointer sort. In this way we can avoid physically exchanging the records representing a point and its angular value.

In the next stages of the algorithm we will be throwing away points. This is where we can use the linked-list implementation discussed in Section 4.2 to store the hull. We make the assumption that all points are initially on the hull and then traverse the linked-list deleting those points that are interior to the hull. The pointers from the sort step can be conveniently used to form the list, which will be doubly linked so we can move either counterclockwise or clockwise on the polygon.

It is not hard to see that if several points have the same angular value then only the outermost of these points can possibly be a vertex of the hull (the point farthest from the bottommost point). By making one pass through the linked-list we can examine each subsequence of identical angular values and retain only that point which is farthest distant from the bottommost point.

After this, we are ready to eliminate any vertex where a reflex angle may be found. We can do this by sequentially scanning around the polygon. For each set of 3 consecutive vertices (i,j,k) we pass, the following test is performed:

```
if $(x_j - x_i)(y_k - y_i) > (y_j - y_i)(x_k - x_i)$
  then accept j as being on the hull; set i := j, j := k,
       and k := (its adjacent counterclockwise neighbor).
  else j cannot be on the hull; backtrack by setting
       j := i and i := (its adjacent clockwise neighbor).
```

Conceptually, the test involves checking to see if j and z are on different sides of an infinite line passing through points i and k. Depending upon the outcome of the test, j is either temporarily accepted as being on the hull, or j is deleted from any further consideration as a possible hull vertex (backtrack).

A convenient place to start the so-called *Graham Scan* [Shamos (78)] is with the vertex immediately counterclockwise from the bottommost point. The scan can stop whenever k reaches the bottommost point because i and j will also be on the hull and any subsequent tests would only involve actual hull vertices. (This is the other advantage of using the bottommost point z as the anchor for the sort step.)

In the process of applying each test of the Graham Scan we can never backtrack more than n times (we can't delete more vertices than we started out with) and we can never "accept" more than n times. Therefore, the maximum number of these tests we can perform is at most 2n which immediately implies that the Graham Scan requires O(n) time in the worst-case.

4.3.3. Complexity Analysis

All steps of the Graham algorithm take O(n) time except sorting the points which requires O(nlogn) time in the worst-case. Thus, the Graham algorithm has a worst-case running time of O(nlogn). Expected case performance is between O(n) and O(nlogn) if we use a DPS sorting algorithm (see Chapter Three). It is not hard to see that the Graham algorithm will perform (subject to the sorting step) slightly faster when most of the points are on the hull because the Graham Scan will be faster. That is, the cost of removing points incurs some overhead.

4.4. Package Wrapping - The Jarvis Algorithm

4.4.1. The Method

This method due to R.A. Jarvis [Jarvis (73)] parallels the way a human being would go about finding the hull. Imagine that each point of the set is represented by a small peg which has been inserted into a flat piece of wood. Take a piece of string and attach it to the bottommost peg so that the string rests on a line that parallels the x-axis; Fig. 4.3. By sweeping the string upward and keeping it taut a peg will be contacted; this peg represents a point that must be on the hull. Continue "sweeping" until another peg is reached, and so on, until the bottommost peg has been contacted again. Each peg where the string changes direction represents a point that is on the convex hull. *Package wrapping* is the name often associated with this method because it can be extended to 3-dimensional sets of points.

4.4.2. Implementation Details

It is not hard to see that finding each new vertex will involve a minimum angle computation. We can use a line parallel to the x-axis passing through the bottommost point as the reference line for the first computation. In all successive minimum angle computations we can use a line collinear to the edge formed by the last two points included in the hull as the reference line.

The method outlined above is certainly correct and can easily be programmed in a high level language. It is a nice method if we don't have to consider efficiency. A closer examination of its features reveals that it will require a significant number of angle computations. For example, consider the case where n = 1000 and the number of points on the hull will be 15. Then the number of these angle computations will be approximately 15,000. Compare this with the Graham algorithm where we would do 1000 angle computations, 1000

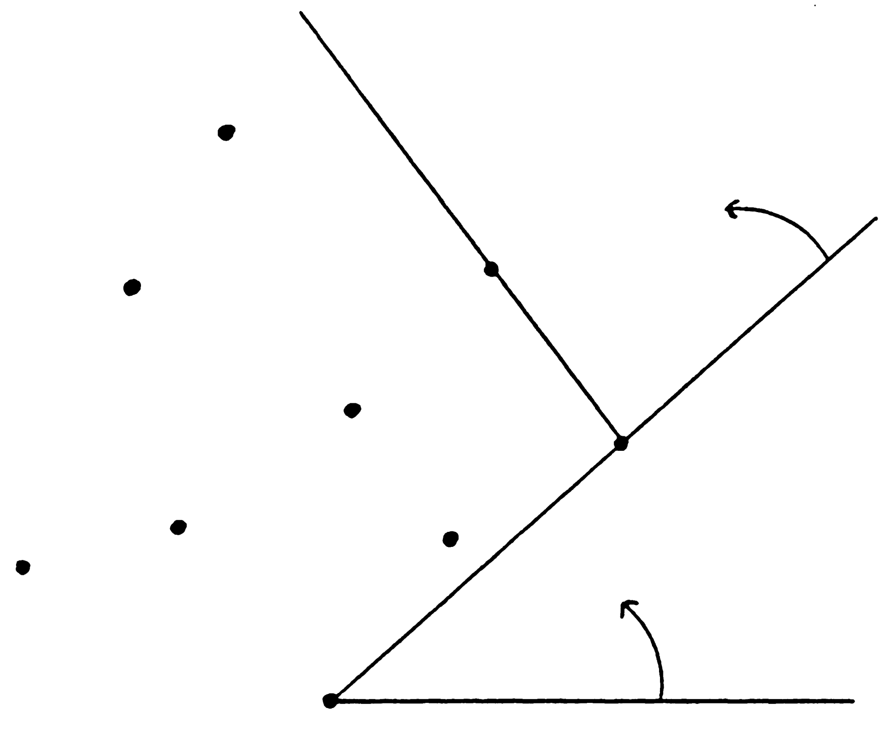


Fig. 4.3. The Jarvis sweep.

distributive sort computations, and at most 2000 point deletion computations. The reader can plainly see that the Graham algorithm is more efficient than package wrapping even on what is admittedly a best-case situation for the latter method.

Usually, when we have a very simple iterative algorithm that is inefficient, the best way to speed it up is to introduce an intermediate step which does a transformation of the inputs or eliminates those that cannot contribute to the final result(s). Such is the case with package wrapping.

Observe that if we draw a line segment from the bottommost hull vertex to the most recently included hull vertex then this line partitions the set of points into one group of points that could still be on the hull and another group containing those points that cannot be on the hull (not including those that have already been found to be on the hull); Fig. 4.4. That is, any point found to lie in the region delineated by the partial hull and a line ℓ from the initial origin cannot be on the hull. After each step we could check all the points to see what side of the line ℓ they are on and delete the interior points. However, Jarvis noticed that extra time may be saved by storing the angular displacement of each point with respect to the bottommost point. Then when a new vertex is determined its angular displacement can be compared to the angular displacement of all the remaining points. Those points that have a smaller displacement can then be thrown away since they can no longer be on the hull. The ordering formula we used in the Graham Algorithm (formula (1)) be conveniently used to compute the angular displacement of each point.

We could be pretty happy with our algorithm now. After all, we have a way to eliminate about half of the angle computations that would have otherwise been done over the course of finding all the hull vertices. However, there is a rather subtle way to do even better. The idea is to use a half-line extended parallel to the positive x-axis as the reference line for each minimum angle computation; Fig. 4.5. The points to be considered in the computation for finding the next hull vertex can then be thought of as falling into a quadrant system where the last found hull vertex is the origin of the system; Fig. 4.6. In this scheme evaluating the angle associated with a point can be avoided if the point's quadrant label is larger that the quadrant label of the current point. Determining the quadrant label for a point j involves looking at the sign of the quantities $(x_j - x_L)$ and $(y_j - x_L)$, where L is the index of the vertex that was last placed into the partial hull. If angles need to be computed then for quadrants 1 and 3 we can use:

$$\text{angle}_j := -(x_j - x_L)/((x_j - x_L) + (y_j - y_L)), \qquad (2)$$

and for quadrants 2 and 4:

$$\text{angle}_j := -(y_j - y_L)/((x_j - x_L) + (y_j - y_L)), \qquad (3)$$

Note that formulas (2) and (3) are variations of formula (1).

The only detail that remains is a mechanism for deleting points and

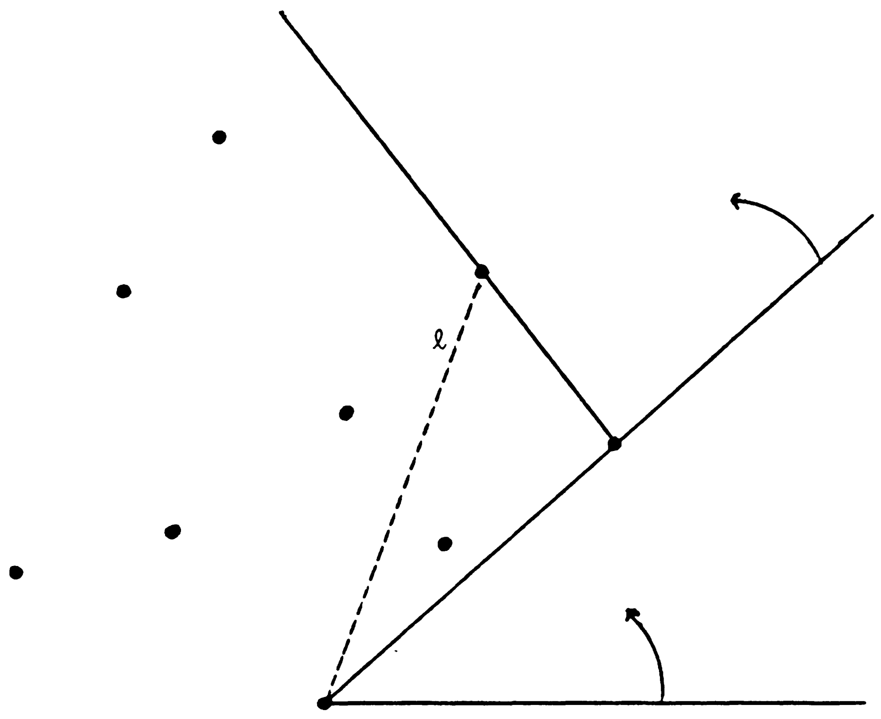


Fig. 4.4. Jarvis point deletion mechanism.

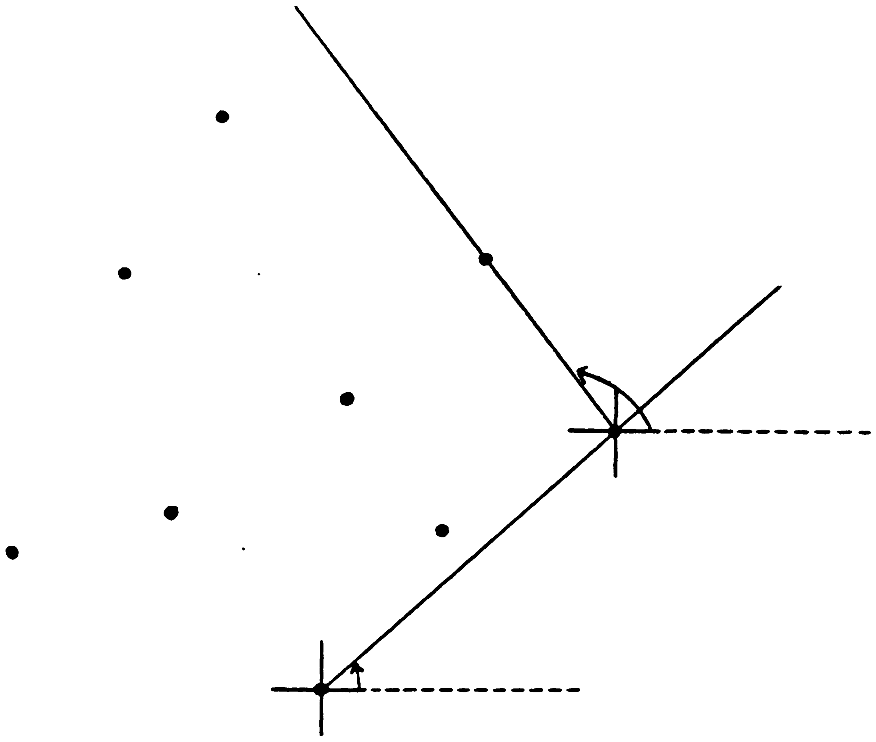


Fig. 4.5. Using a half-line extended parallel to the positive x-axis as the reference line for each minimum angle computation.

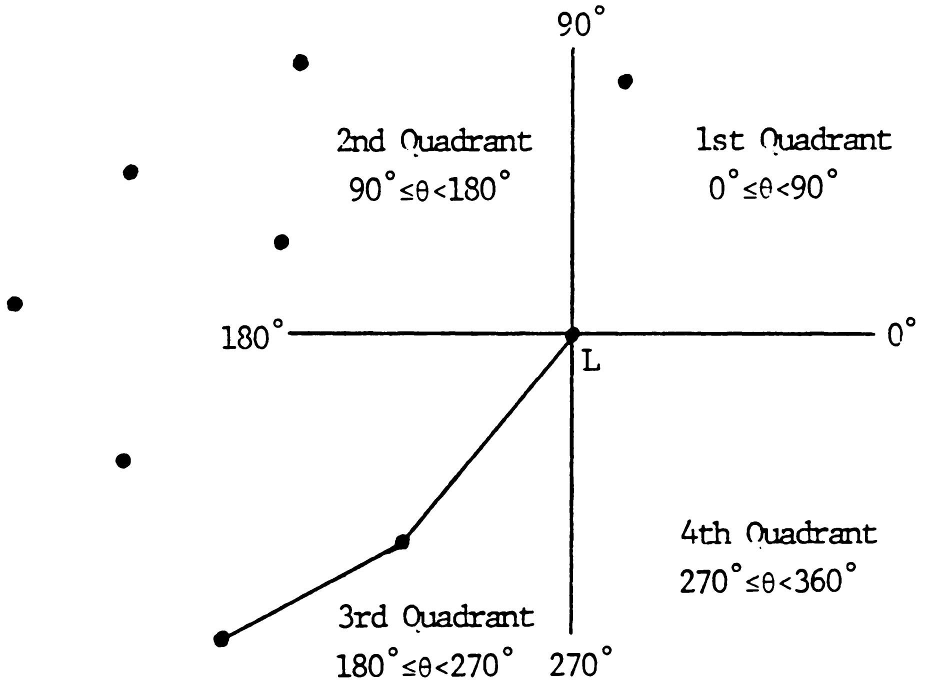


Fig. 4.6. Quadrant label method.

storing the hull. A good way is to hold the set of points in a linked-list and place any points that are on the hull into an auxillary array H. Initially H will be empty. Once the basic algorithm is complete the list pointers and the array H can be used to reform a linked-list that contains the convex hull.

4.4.3. Complexity Analysis

It should be clear from the implementation details we have given that there are two main factors that affect the performance of the Jarvis algorithm. One is the number of points that are on the hull, and the second is the distribution of points within the interior of the hull. Thus, it is hard to make any quantitative statements about the performance of the algorithm. If we make some very strict assumptions about the distribution of inputs and the exact location of the hull vertices in the plane then we can get a fairly good estimate of the number of operations the algorithm will take [Noga (81)]. However, any estimate will involve some rather laborious probabilistic calculations and therefore it is better to do a performance evaluation of the algorithm on a set of sample distributions.

Statements about the best and worst case are much easier to make. We always need to make at least 2n angle computations to identify the first 3 vertices on the hull. If all (or almost all) of the points fall into the partial hull defined by these three points then no more (or almost no more) angle computations will be required. Therefore, the best case is clearly O(n). The worst-case will occur when all n points are the vertices of the hull. In this case the improvements we have suggested will be of no help (in fact they will slow down the algorithm). The number of angle computations will be n(n-1)/2 implying that the algorithm has a $O(n^2)$ worst-case running time.

4.5. The Eddy Algorithm

4.5.1. General Method

The weakness of the Jarvis algorithm is that if most of the points are inside the hull there is still no guarantee it will run quickly. What is needed is a method whereby most of the interior points are sure to be discarded after the first few steps. Such a method exists [Eddy (77a)] and its operation is analagous to the well-known sorting algorithm Quicksort [Hoare (61a)]. The general idea is very simple: find a few points that are on the hull and delete all the points falling into the region formed by these points. This idea can then be carried on recursively by making the region bigger (by finding another point on the hull) or using either the Graham algorithm or package wrapping on the remaining points.

4.5.2. Implementation

The full recursive version is very easy to implement if we make use of a language that will allow recursion such as PASCAL, PL/I, or C. Even if we cannot use recursion we can think of the algorithm recursively and then use an explicit stack in the implementation.

The first step is to locate two points that are certain to be on the hull. A linked-list is then created and initialized with these points. Practical choices are the points with the minimum y-coordinate B, and the maximum y-coordinate T. (B and T are acronyms for the words "bottom" and "top.") The directed line $\overrightarrow{BT}$ is then used to partition S into two sets, α and β, for points lying above line $\overrightarrow{BT}$, and the other for points lying below line $\overrightarrow{BT}$. During the partitioning step the indices of the points farthest above and below $\overrightarrow{BT}$, L and R ("left" and "right"), are computed and placed into the linked list. The points L and R are on the hull because if we take the line segment BT and move it in the direction of its positive or negative normal then the last point BT contacts will be a boundary point of the set; Fig. 4.7.

A convenient way to find the position of a point k relative to a line from point i to point j is to use the quantity

$$S = x_k(y_i - y_j) + y_k(x_j - x_i) + y_j x_i - y_i x_j. \qquad (4)$$

If k is *above* ij then S will be positive; if k is *below* ij, S will be negative, and if k is *on* ij then S will evaluate to zero. The magnitude of S will be in direct proportion to the distance k is above the directed line segment ij. (This formula is related to the one used in the Graham scan.)

The reader may notice that the original problem has been broken into two subproblems of (hopefully) the same size. This is known as *divide and conquer.* Divide and conquer is important not only in the design of efficient computer algorithms, but also in just about any mental task a human performs. Most books on analysis of algorithms contain a fairly lengthy discussion of the technique. In the case of our problem it allows us to concentrate on finding the hull above $\overrightarrow{BT}$ with just the points in the set α, since none of the points in the set β will ever be in the partial hull from B to T. Furthermore, once we find a way to compute the hull above $\overrightarrow{BT}$ it should be obvious that we can use the same strategy for computing the hull below $\overrightarrow{BT}$.

Observe that any points falling inside triangle BLT cannot be on the hull. To eliminate these points the procedure is to first isolate the points that are above the directed line segment $\overrightarrow{BL}$, then next find all points that are above $\overrightarrow{LT}$. Of course, to really be efficient, while determining which points are above $\overrightarrow{BL}$ we can keep track of that point X which is farthest above $\overrightarrow{BL}$. Likewise, when finding the points above $\overrightarrow{LT}$, the highest point Y above $\overrightarrow{LT}$ can be retained; Fig. 4.7.

Clearly, point X must be on the hull and any points falling inside of triangle BXL cannot be on the hull. That is, the subproblem with triangle BXL is in the exact same form as the original problem with triangle BLT. This immediately suggests a concise recursive solution to

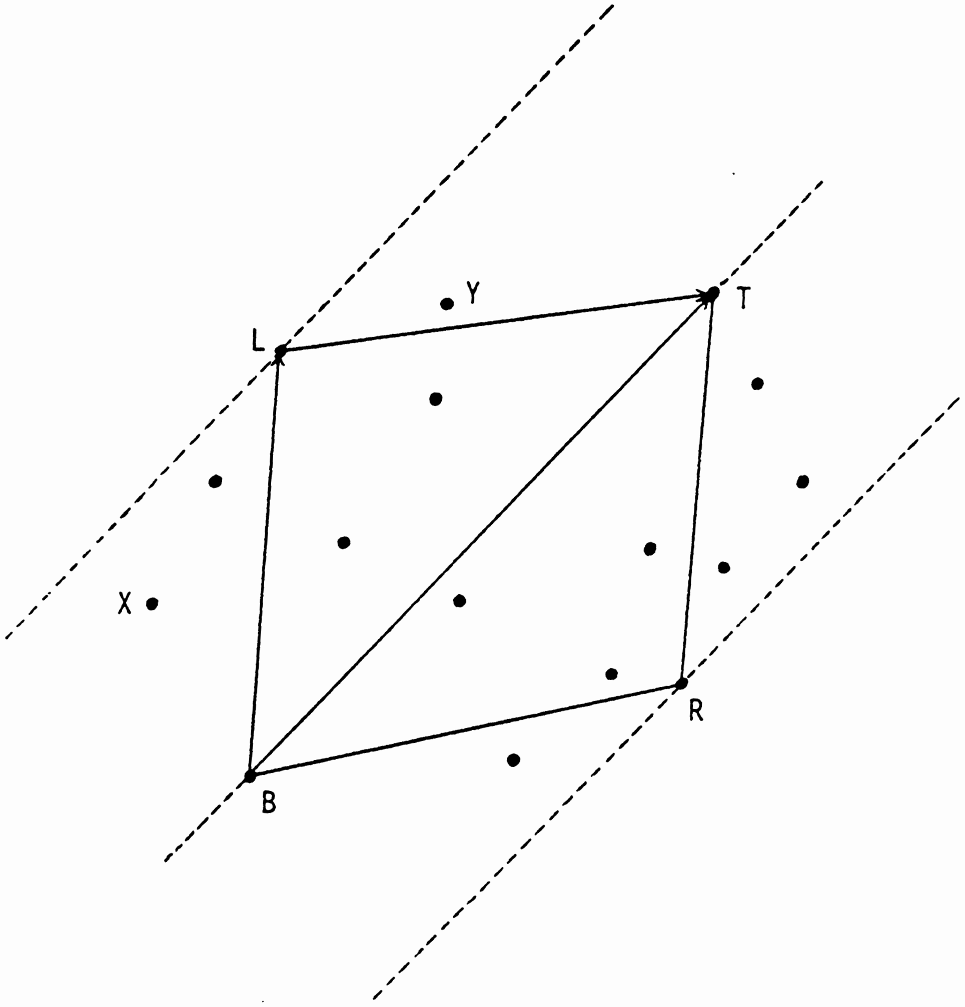


Fig. 4.7a. The partitioning process of the Eddy algorithm -- first level of recursion.

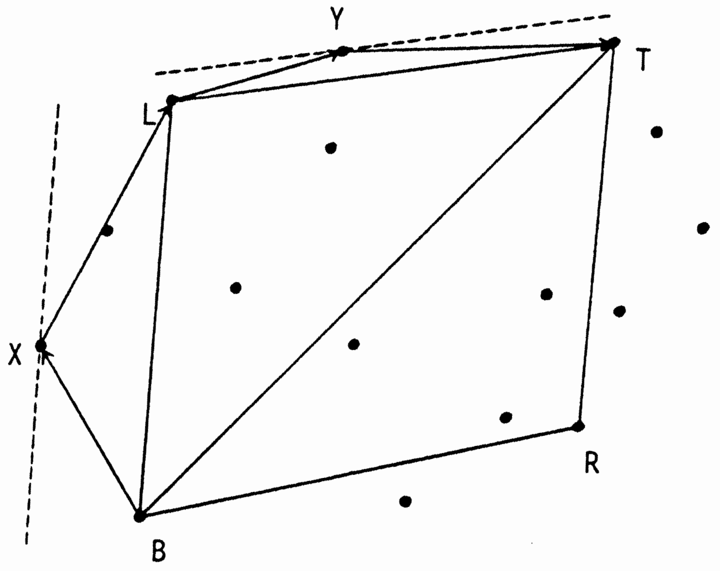


Fig. 4.7b. The partitioning process of the Eddy algorithm -- second level of recursion.

the original problem of finding the hull above line $\overrightarrow{BT}$ given the set of points α. Two procedures, Find_above_1 and Find_above_2, are all that are needed. Find_above_1 will locate the highest point above line $\overrightarrow{BL}$ and insert this point into the linked list between B and L. It will also, as a side effect, return all the points in α that are below $\overrightarrow{BL}$. Find_above_2 is similar in that it locates the highest point above $\overrightarrow{LT}$ with the points returned from Find_above_1 and inserts this point into the linked list between L and T. However, unlike Find_above_1, procedure Find_above_2 does not need to return any of the points below line $\overrightarrow{LT}$ since these points are inside triangle BLT.

Now, if Find_above_1 calls itself recursively with the points above line $\overrightarrow{BL}$ and then calls Find_above_2 with the points returned from the recursive call, then the hull above $\overrightarrow{BL}$ can be computed. Likewise to compute the hull above $\overrightarrow{LT}$, Find_above_2 should call Find_above_1 with the points above $\overrightarrow{LT}$ and then call itself recursively with the points returned from Find_above_1. This system of procedures is a good example of *indirect recursion.* See [Hofstadter (79)] for a description of this technique.

4.5.3. Complexity Analysis

How fast is the Eddy algorithm? If points are drawn from a uniform distribution inside a circle or square it will take only a few recursive calls to eliminate all but a few points in the set. The following informal argument demonstrates that the algorithm will be O(n) in these cases.

It should be clear that the speed of the algorithm is dependent on the number of times the position of a point is checked relative to some line. This will involve a call to a function which evaluates S in equation (4). The initial split of the points requires n evaluations of this function. To eliminate the points in triangles BLT and TRB requires an additional 2n evaluations of equation (4). Note that at least n/2 points will be eliminated (in the expected sense) by this step because triangles BLT and TRB take up at least 1/2 of the area in which the points are distributed. This leaves at most n/2 points above lines BL, LT, TR, and RB. But the area argument is recursive so therefore it will take $(n + n/2 + n/4 + \ldots) = 2n$ or fewer evaluations of S in equation (4) to find all of the remaining hull vertices. Thus, the total number of evaluations of equation (4) will be at most $5n = O(n)$.

What happens to the running time of the Eddy algorithm when all (or almost all) of the points are on the hull? In this case performance will be similar to Quicksort. We will be recursively partitioning the points without throwing any away. Each partitioning problem will give rise to two subproblems of approximately size n/2. This analysis leads to a recurrence relation of the form

$$\begin{aligned} T(n) &= c, && n = 1, \\ T(n) &< T(an) + T(bn) + O(n), && n > 1. \end{aligned}$$

where a and b are random variables and $a + b = 1$. At each step of the recursion as long as both a and b are greater than zero, then this recurrence has a solution of $O(n\log n)$ [Devai (79)].

In a worst-case situation all of the points will be above the partitioning line at each stage of the algorithm. This means that the original problem will be reduced by one point for each recursive call. This leads to a recurrence relation of the form

$$
\begin{aligned}
T(n) &= c, & n &= 1,\\
T(n) &= T(n-1) + O(n), & n &> 1,
\end{aligned}
$$

which has a solution of $O(n^2)$.

4.5.4. The Akl-Toussaint Algorithm

The above analysis suggests that a hybrid of the Eddy algorithm might perform reasonably well. The hybrid should do two things: (1) if the points are uniformly distributed over some region in the plane then it should perform about the same as the Eddy algorithm. (2) If most of the points are on the hull it should do a better job than Eddy (or at least avoid the $O(n^2)$ worst-case situation). With this in mind S.G. Akl and G.T. Toussaint [Akl(78b)] designed a convex hull algorithm which essentially involves the first few steps of the Eddy algorithm and then uses a variant of the Graham algorithm to find the remaining hull vertices. (Remember that a hybrid combines the most important features of two (or several) algorithms with the hope that a faster algorithm will result.) As we saw in Chapter Three on sorting and selection, this idea proved to be the key to the development of a very fast sorting algorithm.

The first step involves finding the four extreme points xmin, ymin, xmax, and ymax. Any points which fall inside the quadrilateral region formed by these points may then be eliminated; Fig. 4.8.

Next, we find the point k_i in extremal region i whose coordinates (x_{k_i}, y_{k_i}) maximize the quantity

$$m_1 x_{k_i} + m_2 y_{k_i}$$

where,

m_1 = +1 for regions 2 and 3,

m_1 = -1 for regions 1 and 4,

m_2 = +1 for regions 1 and 2,

m_2 = -1 for regions 3 and 4.

This will allow all points falling inside each of the four triangles (xmin,k_1,ymax), (ymax,k_2,xmax), (xmax,k_3,ymin), and (ymin,k_4,xmin) to be removed from any further consideration as possible extreme points of the hull.

These first few steps are almost identical to the Eddy algorithm. Again the idea is to discard any points that fall into interior triangular regions. After these two steps we are left with eight regions in which there may be possible convex hull vertices. In each region we can sort the points by (say) y-coordinate and then apply the Graham scan.

We have avoided any discussion of implementation details since these are simply a combination of the ones found with the Eddy and Graham

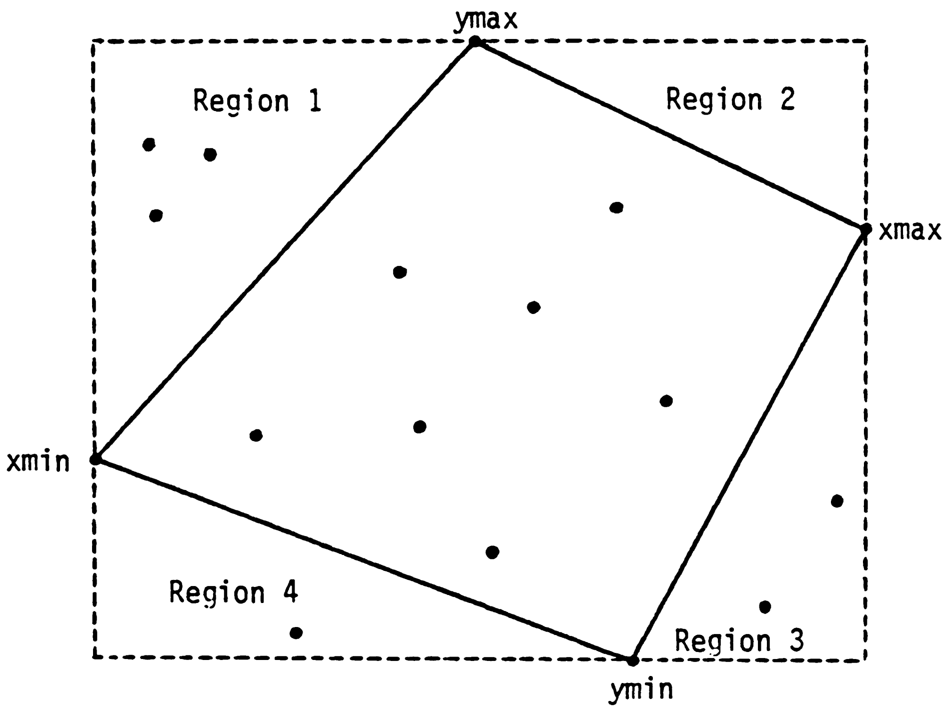


Fig. 4.8. Point deletion process of the Akl-Toussaint algorithm.

algorithms.

A qualitative analysis of the algorithm leads to a "Catch 22" situation. If we encounter a distribution of points that is good for the Eddy algorithm then the Akl-Toussaint algorithm will not perform any faster since there is some overhead in switching to the Graham scan in each of the eight regions. On the other hand, if most of the points are on the hull (not a particularly good situation for the Eddy algorithm) then Akl-Toussaint will outperform Eddy. However, it will never be as fast as the Graham algorithm because of the time wasted in the first few steps.

The major advantage which this algorithm provides is stability. Time complexity will always be between O(n), the best-case of the Eddy algorithm, and O(nlogn), the worst-case of the Graham algorithm.

4.6. Performance Evaluation

4.6.1. Introduction

We have discussed several algorithms for finding the convex hull of a set of points in the plane. Throughout, we have informally compared these algorithms and qualitatively analyzed those situations which appear to be good or bad for each algorithm. What we need now is a quantitative measure of how these algorithms perform on a uniform set of inputs. In some cases we have also given theoretical bounds on the running time of these algorithms assuming a standard distribution of points in the plane. A quantitative test will also serve to experimentally verify these results.

4.6.2. Experimental Procedure

We have obtained coded versions of the Eddy and Akl-Toussaint algorithms from the authors [Eddy (77b)] [Akl (79)] and written our own programs to carry out the Graham and Jarvis algorithms (see Appendix 1). All of these were written in FORTRAN and run on an IBM 3032 (FORTX, OPT=2).

Since the language is FORTRAN, both the Eddy and Akl-Toussiant algorithms were non-recursive. In other languages these algorithms could be coded using recursive subroutines. However, one must be careful of these situations since some languages are notoriously inefficient in the handling of recursion. In analysis of algorithms jargon this is known as *the overhead of recursion.* There is a systematic procedure to convert any recursive subroutine into an equivalent nonrecursive version. After applying this transformation further simplifications can be made thereby producing even more gains in efficiency [Horowitz (76)]. It turns out that removing the recursion from the Eddy and Akl-Toussaint algorithms is no harder than converting Quicksort (which can be coded quite elegantly using recursion) to a non-recursive routine.

The next thing to consider is the type of data we will need to test our algorithms. The following distributions represent a varied cross-section which should provide a realistic benchmark for testing.

(a) Uniform inside a square,

(b) Uniform inside a circle,

(c) Uniform inside an annulus (inner radius 9/10 of outer radius),

(d) Uniform on a circle.

Summarizing our previous analysis, the Graham algorithm should work about the same on all four distributions; (d) will run a little faster than the other three. The Jarvis Algorithm should work reasonably well on distribution (a), (b) and (c) should be somewhat slower, and (d) will be $O(n^2)$. The Eddy and Akl-Toussaint algorithms should handle distributions (a) and (b) very quickly. On (c) Eddy should be a bit slower than for (a) or (b) since there will be more points on the hull; Akl-Toussaint should be slower for the same reason but it is hard to estimate exactly how much. On (d) Akl-Toussaint should be faster than Eddy because of the hybridization; however, the time for Eddy should still be acceptable.

Timings were recorded for sample sizes of 100, 250, 500, 1000, 2000, and 4000 points. 100 runs were made for sample sizes of 100, 250, 500, and 1000 points, 50 runs for 2000 points, and 25 runs for 4000 points. The only exception was in the case of the Jarvis algorithm where only 1 run was made on distribution (d) for all sample sizes. The results of the performance tests appear in Tables 4.1, 4.2, 4.3, and 4.4. All times are in seconds and have been adjusted for 100

Table 4.1. Computation time: Uniform inside a square

N	Graham	Jarvis	Eddy	A-T
100	.47	.94	.43	.41
250	1.18	2.62	.97	.91
500	2.39	6.02	1.91	1.73
1000	4.78	13.21	3.65	3.37
2000	9.78	29.02	7.36	6.54
4000	19.96	60.60	14.48	12.88

Table 4.2. Computation time: Uniform inside a circle

N	Graham	Jarvis	Eddy	A-T
100	.44	1.00	.43	.41
250	1.22	3.31	1.02	.97
500	2.30	7.98	1.90	1.97
1000	4.80	19.36	3.80	3.96
2000	9.62	48.92	7.22	8.20
4000	19.96	95.84	14.44	17.24

Table 4.3. Computation time: Uniform inside an annulus

N	Graham	Jarvis	Eddy	A-T
100	.45	1.93	.65	.77
250	1.17	6.65	1.51	2.05
500	2.24	16.37	2.84	4.35
1000	4.69	52.63	5.59	9.50
2000	9.44	104.86	10.82	20.74
4000	19.76	262.48	21.60	45.72

Table 4.4 Computation time: Uniform on a circle

N	Graham	Jarvis	Eddy	A-T
100	.41	6.90	1.25	.88
250	1.01	42.50	3.54	2.30
500	2.01	164.00	7.80	4.82
1000	4.02	674.00	16.78	10.28
2000	8.16	2615.00	35.86	21.60
4000	17.28	9357.00	75.52	46.36

runs.

4.6.3. Discussion

The results of the test for the most part back-up our earlier qualitative analysis. The Graham algorithm transforms a two-dimensional sorting problem into one dimension. Remember, all convex hull algorithms must be able to sort. In the Graham algorithm the sorting step is explicit and we were fortunate enough to have at our disposal a very fast sorting technique, DPS. Therefore, it is not too surprising that the Graham algorithm stands-up comparatively to the other methods.

Even after all the improvements suggested in Section 4.4, the performance of the Jarvis algorithm was disappointingly slow. The reason simply is that points on the interior of the hull are not eliminated quickly enough. Another way to diagnose the problem is to realize that the Jarvis algorithm is really straight selection sorting (with throwaway) in disguise. It is clear that the Graham algorithm with DPS and no throwaway will in almost all cases be faster than a straight selection sort with intermittent throwaway.

Recall, that package wrapping is believed to be very similar to the way humans attempt to find the convex hull. The fact that it runs slower than all of the other methods in the test indicates that computer systems have a long way to go in emulating human visual perception.

This just adds to the growing body of evidence in Artificial Intelligence which indicates that radically different architectures and software must evolve before machines are ever able to reason spatially as fast as human beings.

There were really no surprises in the running time of the Eddy algorithm on the various distributions. Performance was very good on distributions (a) and (b); on (c) it took more recursive calls and thus more time to eliminate the points, and on (d) no points were thrown away so performance appeared to follow the theoretical bound of O(nlogn).

On distribution (a), uniform in a square, Akl-Toussaint seems to be slightly faster than the Eddy algorithm. There are two possible explanations. One is that the Eddy program is inefficient in some respects (we did not check for this). The other is that the Akl-Toussaint algorithm may do a slightly better job of eliminating points in the first few steps than the Eddy algorithm. If this is the case the following statements due to Bentley and Shamos [Bentley (78)] suggest that very few points were leftover for the Graham part of the hybrid.

"For uniform sampling within any bounded figure F, the hull of a random set will tend to assume the shape of the boundary of F. If F is a polygon, points accumulating in the corners will cause the resulting hull to have very few vertices. Because the circle has no corners, the expected number of vertices is comparatively high."

If the second reason is correct, then the Bentley-Shamos statements also explain why Eddy outperformed the Akl-Toussaint algorithm on distributions (b) and (c). Using pencil and paper one can easily verify that many points will be left over after the first few steps of the Akl-Toussaint algorithm for these distributions, especially for distribution (c). The results on distribution (d) verify why hybrid algorithms are superior to "one-type" methods. In a way the hybrid is intelligent because in effect it recognized that after a few steps no points were being discarded, so it switched to a different method. (This was not part of the code, but this idea could be easily implemented.)

We spent considerable effort implementing and profiling our Graham and Jarvis implementations to ensure that they were very efficient. This is a subjective statement since in general there is no way to prove that one particular implementation is the most efficient [Knuth (74)]. However, we do feel safe in concluding that the results indicate that the Graham algorithm is the best general purpose convex hull finder.

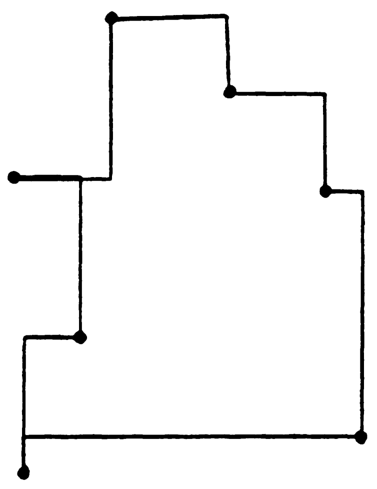
4.7. The L_1 Hull

4.7.1. Definition

The convex hull is a geometric structure in the space R_2 which, as we shall see in the next few Chapters, has a number of important applications. It is natural to ask whether there is an analog structure in the Manhatten metric which (for lack of a better name) we might call the *L_1 hull.* To be consistent with the definition of the convex hull this structure would have to be an L_1 convex polygon containing the set of points with minimal interior area and perimeter.

What is convexity in the L_1 metric? Recall that an L_1 line between any two points (i,j) in R_1 is a staircase-like sequence of connected orthogonal line segments whose distance is equivalent to $d_i(i,j)$. Following the definition of convexity in R_2, an L_1 polygon is said to be convex if all pairs of interior points can be connected by some L_1 line that is entirely contained within P. Figures 4.9 and 4.10 contain some convex and nonconvex L_1 polygons.

Note that in general there are many minimum perimeter L_1 convex polygons that can contain a given set of points; Fig. 4.11. And, it is easy to see that the length of each of these polygons will always be d_1(ymin,xmax) + d_1(xmax,ymax) + d_1(ymax,xmin) + d_1(xmin,ymin). However, as Figure 4.12 illustrates, to find the polygon of minimum area requires that we compute the four paths P_1 = (ymin, ..., xmax), P_2 = (xmax, ..., ymax), P_3 = (ymax, ..., xmin), and P_4 = (xmin, ..., ymin), where each P_i in the set $\{P_1, P_2, P_3, P_4\}$ is an orthogonal line segment sequence that is maximal with respect to the number of points it may contain from S. The reason we don't want the same point in two different paths is that in applications involving the L_1 hull we want to avoid redundancy. Note from Figure 4.13 that the ordering of vertices on the L_1 hull is not necessarily unique for some sets of



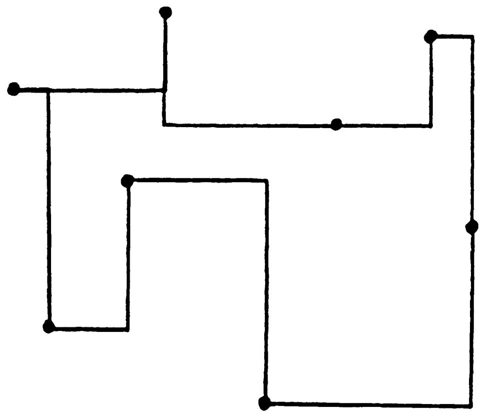
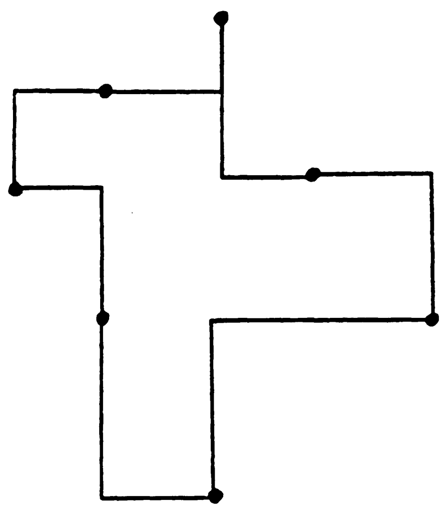


Fig. 4.9. Non-convex L_1 polygons.



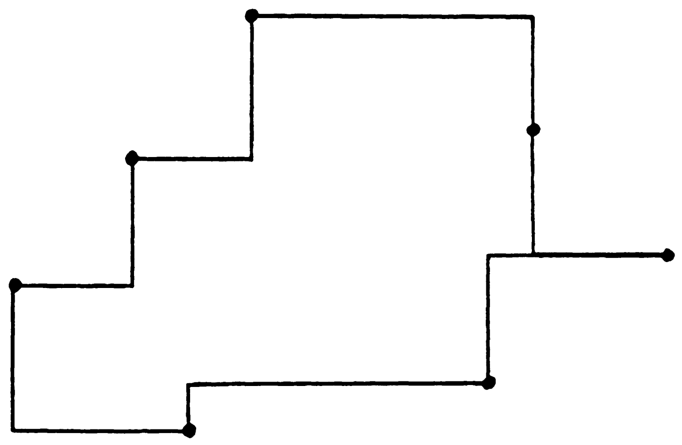


Fig. 4.10. Convex L_1 polygons.

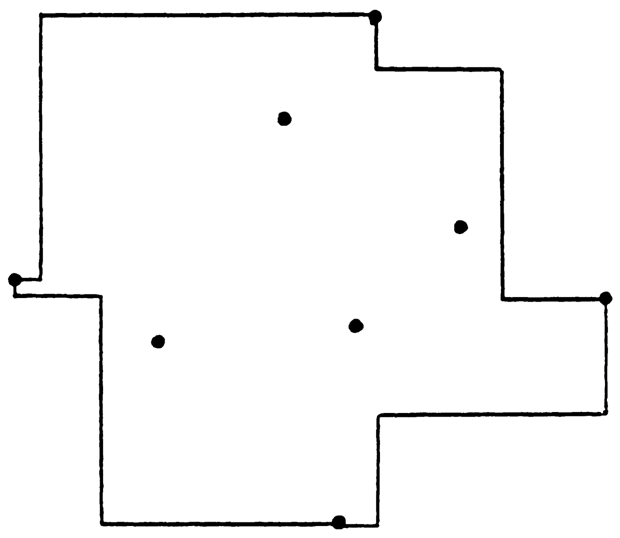
points. Thus, we define a canonical form in which path P_1 is computed first, followed by paths P_2, P_3, and finally P_4.

4.7.2. L_1 Hull Algorithm

Let us see if we can discover a way to efficiently compute the L_1 hull. It follows from the definition that if we can find a method for computing one of the paths (say P_1) then we can use the same procedure on each of the other three paths.

Consider the rectangular region delineated by the points ymin and xmax as shown in Figure 4.14. It should be apparent that if there are any points in this region then at least one of these points must be on path P_1 (i.e., on the hull). The question is which one? Note that if there were a point near corner b of the shaded region then this point would surely be on the hull. Because, this point would have the highest combined total distance above and below the directed line segments (a,ymin) and (a,xmax).

Our observation suggests that the point p closest to corner b of the shaded region is on the hull. That is, the point that is farthest above (a,ymin) plus farthest below (a,xmax). We can prove that p is on the hull by contradiction. If p is not on the hull then there must be a point in region 4. But then this point would be the point farthest above (a,ymin) plus farthest below (a,xmax).



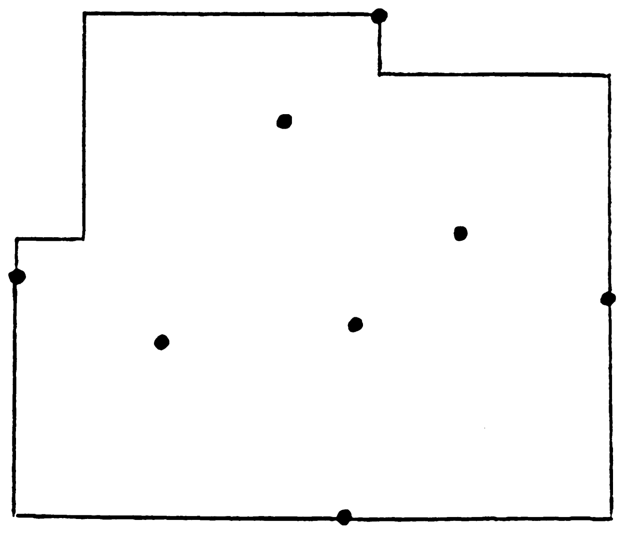


Fig. 4.11. Minimum perimeter L_1 polygons containing the set of points.

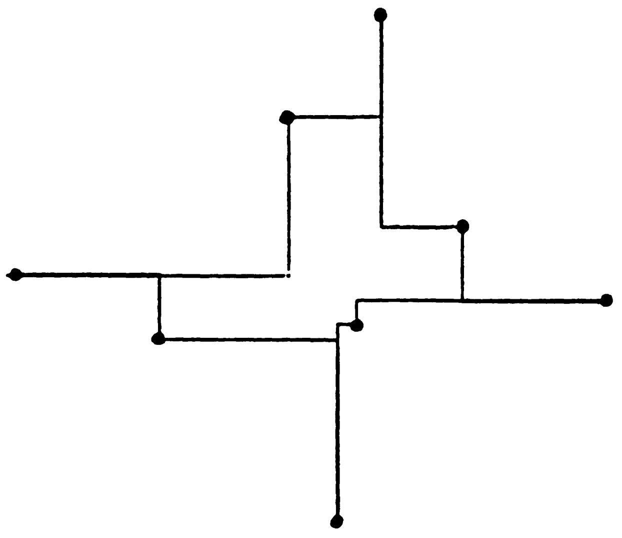
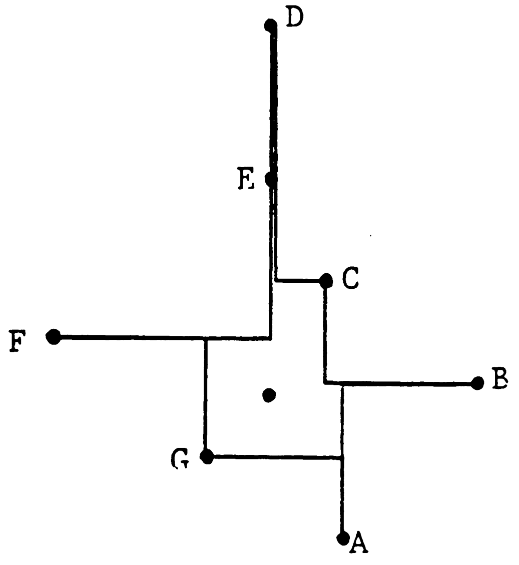
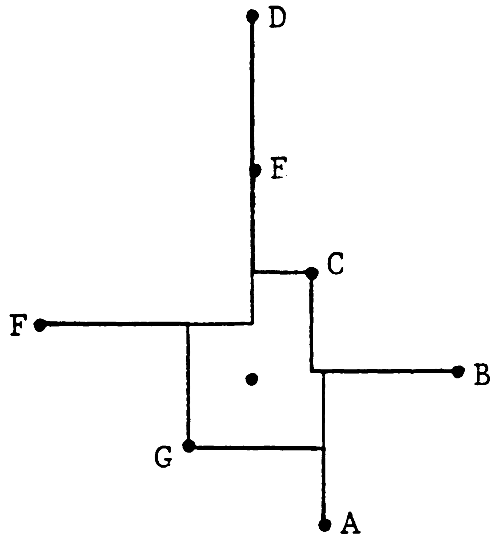


Fig. 4.12. The L_1 hull of the set of points in fig. 4.11.



$H_1 = (A,B,C,D,E,F,G)$



$H_2 = (A,B,C,E,D,F,G)$

Fig. 4.13. Both H_1 and H_2 are valid L_1 hulls for the set of given points.

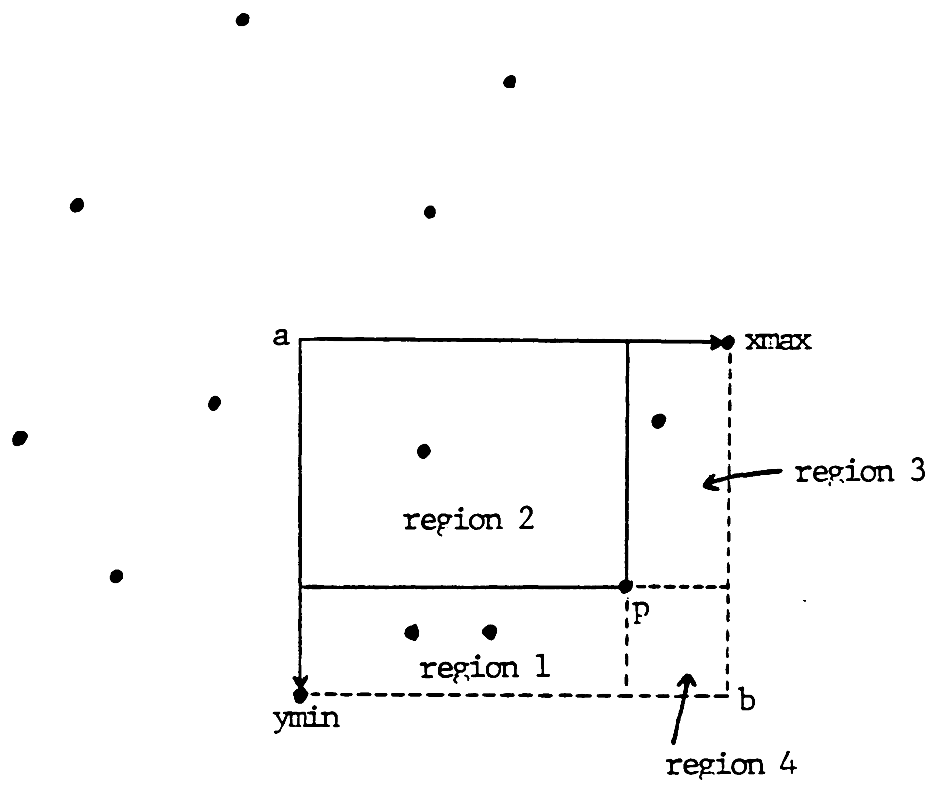


Fig. 4.14. Illustrating the proof that point p is on the L_1 hull.

It should be clear that finding p splits the original problem into two new subproblems. If any points are left in region 1 than at least one of these points must be on the hull. The same is true in region 3. Because the subproblems in regions 1 and 3 are exactly identical to the original problem it follows that computing P_1 can be carried out by use of a recursive procedure. The argument that this procedure will find the correct path from ymin to xmax is inductive and follows trivally from the proof given above. For the sake of completeness we give a psuedo-code version of the procedure for computing the path P_1 from ymin to xmax.

```
PROCEDURE PATH (S, n, ymin, xmax);
  BEGIN
    STEP 1: Let $S_1$ contain all of the points k  S such that
      $x_k \geq x_{ymin}$ and $y_k \leq y_{xmax}$;
    STEP 2: Let $n_1 = |S_1|$;
    STEP 3: IF $n_1 > 0$ THEN
      BEGIN
        STEP 4: Find the point pmax  $S_1$ that maximizes the
          function $(x_{pmax} - x_{ymin}) + (y_{xmax} - y_{pmax})$;
        STEP 5: Insert pmax into H between ymin and xmax, and
          delete pmax from S and $S_1$.
        STEP 6: PATH ($S_1$, $n_1$, ymin, pmax);
        STEP 7: PATH ($S_1$, $n_1$, pmax, xmax)
      END
  END.
```

The procedure for computing the other paths is essentially identical except for a simple modification of the function in step 4, and corresponding changes in the parameters in steps 5, 6, and 7.

4.7.3. Analysis

The worst-case of the L_1 Hull algorithm is $O(n^2)$ since each pair of recursive calls (steps 6 and 7) may result in a subproblem of size n-1 points.

Consider a uniform distribution of points in the plane. Refering to Figure 4.14, it is clear that for path P_1 we can expect to find p in the extreme lower corner (near b) of the rectangle delineated by ymin and xmax. This implies that region 2 will be much larger than the combined areas of regions 1 and 3, and thus at least 50 percent of the points (but usually much more) will be eliminated after each pair of recursive calls. This leads directly to the recurrence relation

$$T(n) = k, \qquad n = 1,$$
$$T(n) = T(an) = T(bn) + O(n), \qquad n > 1$$

where $a \geq b \geq 0$ are random variables, and $\max(a + b) < 1$. This means that a given percentage of the points are thrown away at each step of the recursion which gives a solution of O(n) [Devai(79)]. The same argument holds for the remaining three paths which implies that the expected time to compute the L_1 hull will be O(n).

4.7.4. Final Notes

Our interest in the L_1 hull was motivated by the wealth of applications that abound for the convex hull. We note that the L_1 hull contains (at least) all points on the convex hull. Furthermore, the convex hull must encase the L_1 hull. Could it be that the L_1 hull is more suitable for some of the applications where the convex hull is presently being used? This appears to be a promising avenue for future research.

Chapter 5

MINIMUM ENCASING RECTANGLES AND SET DIAMETERS

"I have lived in this world just long enough to look carefully the second time into things that I am the most certain of the first time."

- Josh Billings

5.1. Introduction

In this Chapter we wish to examine several interrelated problems that have long been of interest to researchers in computational geometry.

Problem 1: *(Diameter of a Set)* Given a set of points in the plane, determine the two points that are farthest apart.

Problem 2: *(Diameter of a Polygon)* Given the vertices of a simple non-intersecting polygon, determine the two vertices that are farthest apart.

Problem 3: Given a set of points in the plane, determine the minimum area rectangle that will encase the set of points.

Problem 4: Given the vertices of a simple non-intersecting polygon, determine the rectangle of minimum area that will encase the polygon.

These problems have wide theoretical appeal probably because they can be so simply stated. However, they also have a number of important applications, mainly in pattern recognition and operations research.

Consider the problem of having a machine distinguish amongst several different objects moving along an assembly line conveyer belt. These objects could be in any order and there may be more than one of each object. (We assume that the objects cannot overlap and that there is adequate spacing amongst the objects.) One procedure would be to take a picture of each object using a TV camera placed at some fixed distance immediately above the belt. This picture could then be processed to get a digitized image of each object. Because each object will have a minimum encasing rectangle of known area, or diameter of known length it follows that in many cases we can use the algorithms for problems 1 through 4 to classify the objects. The procedure is guaranteed to work under all two dimensional rotations and translations of the objects; it does not matter in what position the objects sit on the conveyer belt, only that the same set of surfaces is always exposed to the lens of the camera.

Another example of a problem where it is necessary to compute the diameter of a set is *clustering*. A clustering of a set is a partition of its elements that is chosen to minimize some measure of dissimilarity. In two dimensions, a measure of the "spread" of a cluster is the maximum distance between any two of its points, called appropriately the diameter of the cluster. Shamos [Shamos (78)] points out that a cluster with small diameter has elements that are closely related, while the opposite is true of a large cluster. He then goes on to formulate the clustering problem as follows: given n points in the plane, partition them into k clusters $C_1, \ldots, C_k$ so that the maximum cluster diameter is as small as possible. Solving this problem will involve an algorithm for determining cluster diameters, hence the motivation for finding a very efficient algorithm to determine the diameter of a set.

References [Haims (70)], [Adamowicz (72)], and [Eastman (71)] discuss several other applications of the minimum encasing rectangle including template-layout problems, cutting stock, optimal space planning and packaging problems.

5.2. Problem Synthesis

At first glance it might seem that the diameter and minimum area encasing problems are not really related, besides the obvious fact that both are optimization problems. Also, it seems that finding the diameter of a set (or polygon) is so simple that we should not even consider solving this problem. Certainly, we can compute the distance between each pair of points, of which there are $n(n-1)/2$, and choose the largest of these as the diameter. Could it be that this $O(n^2)$ procedure is not the fastest algorithm for determining the diameter? On the other hand, there is nothing about the minimum area encasing rectangle problem that immediately suggests a simple algorithm for its solution. What is the common thread that binds these two problems?

It should be clear that working with all the points in the set is not really necessary. Those points that are on the interior of the set are not needed to find the minimum area encasing rectangle. The points that are really of concern are the boundary points of the set. Therefore, it appears that finding the convex hull and working with this structure might be one way to proceed. Recall that the convex hull is the minimum area convex polygon containing the set of points. Since any encasing rectangle of the set is also convex it follows that the minimum area encasing rectangle must also encase the convex hull.

Could it be that finding the convex hull is also useful in efficiently determining the diameter of a set? If it is, then the diameter's endpoints would have to be two vertices of the hull. We can prove this theorem by contradiction. Assume that at least one of the endpoints of the diameter is not on the hull. Now, if we extend the diameter line segment so that it intersects an edge of the hull, then the distance between one of the endpoints of the convex hull edge and one of the diameter endpoints must be greater than the length of the diameter by the triangle inequality; Fig. 5.1., a contradiction.

So far, we have discovered that instead of working with all the points we can work with the convex hull of the set (or convex hull of the polygon for problems 2 and 4). Of course, it may be that all the points are on the hull in which case it would appear that we have carried out an unnecessary computation without eliminating any of the

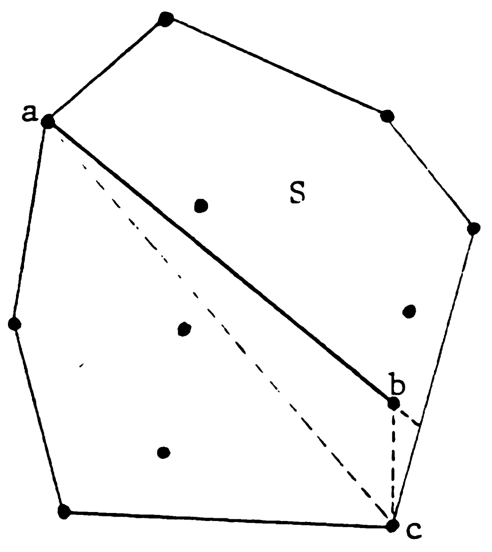


Fig. 5.1. Select any pair of points (a,b) with one of the points (say b) not on the convex hull of S. Then, this pair cannot comprise the diameter since there is a point c on the convex hull of S such that (a,c) is separated by a greater distance.

points. The next step is to determine if there is something about the hull that will allow the minimum encasing rectangle or diameter to be more efficiently computed.

Certainly, there are an infinite number of ways that an encasing rectangle can be drawn so that it might contain the convex hull. And, if any one of these is going to be the minimum encasing rectangle then each of its sides must touch the convex hull at some point. Unfortunately, both of these observations are of little help. What we really need to know is where one edge of the minimum encasing rectangle touches the hull and the direction of this edge. With this information it would be a trivial matter to compute the area and corner points of the rectangle. At present there is no method for determining these. However, Freeman and Shapira have discovered that the minimum area encasing rectangle must have one of its sides collinear to an edge of the convex hull [Freeman (75)]. While this result does not specifically address the problem of edge orientation, it does reduce the minimum encasing problem to one of enumerating all encasing rectangles collinear to each edge of the hull.

This can be done in a very straightforward manner by translating the convex hull so that a selected edge will lie collinear to the positive x-axis with one of its endpoints centered at (0,0). The sides of the minimum area encasing rectangle for each of the translated polygons will be orthogonal to the x and y axis. Therefore, finding the four vertices with the minimum and maximum x and y coordinates within the translated system will be sufficient to compute the area of each encasing rectangle. If there are n edges on the hull then each translation (and encasing rectangle) can be computed in O(n) time. Since a translation is required for each successive edge it follows that the total time to enumerate all encasing rectangle by this method is $O(n^2)$. Again we should ask: Is this brute force technique the most efficient way to compute all encasing rectangles?

What about finding the diameter of the convex hull? Are there only certain pairs of vertices which could comprise the diameter? For example, it would appear that adjacent vertices are unlikely candidates while vertices separated by relatively large distances have a much better chance of forming the diameter pair. Our suspicions are borne out in the following theorem due to I.M. Yaglom and V.G. Boltyanskii [Yaglom (61)]: The diameter of a convex polygon is the greatest distance between parallel *lines of support*. The definition of a line of support is well-known to researchers in classical geometry. For a given polygon P a line of support L meets the boundary of P at one point (or is collinear to an edge of P) such that P lies entirely on one side of L. For a set of points in the plane a line of support L passes through at least one point of the set and all remaining points must either be on L or on one side of L. Analagous definitions exist for planes and hyperplanes of support in three and more dimensions. Figure 5.2 gives several examples of supporting lines for a polygon and set of points.

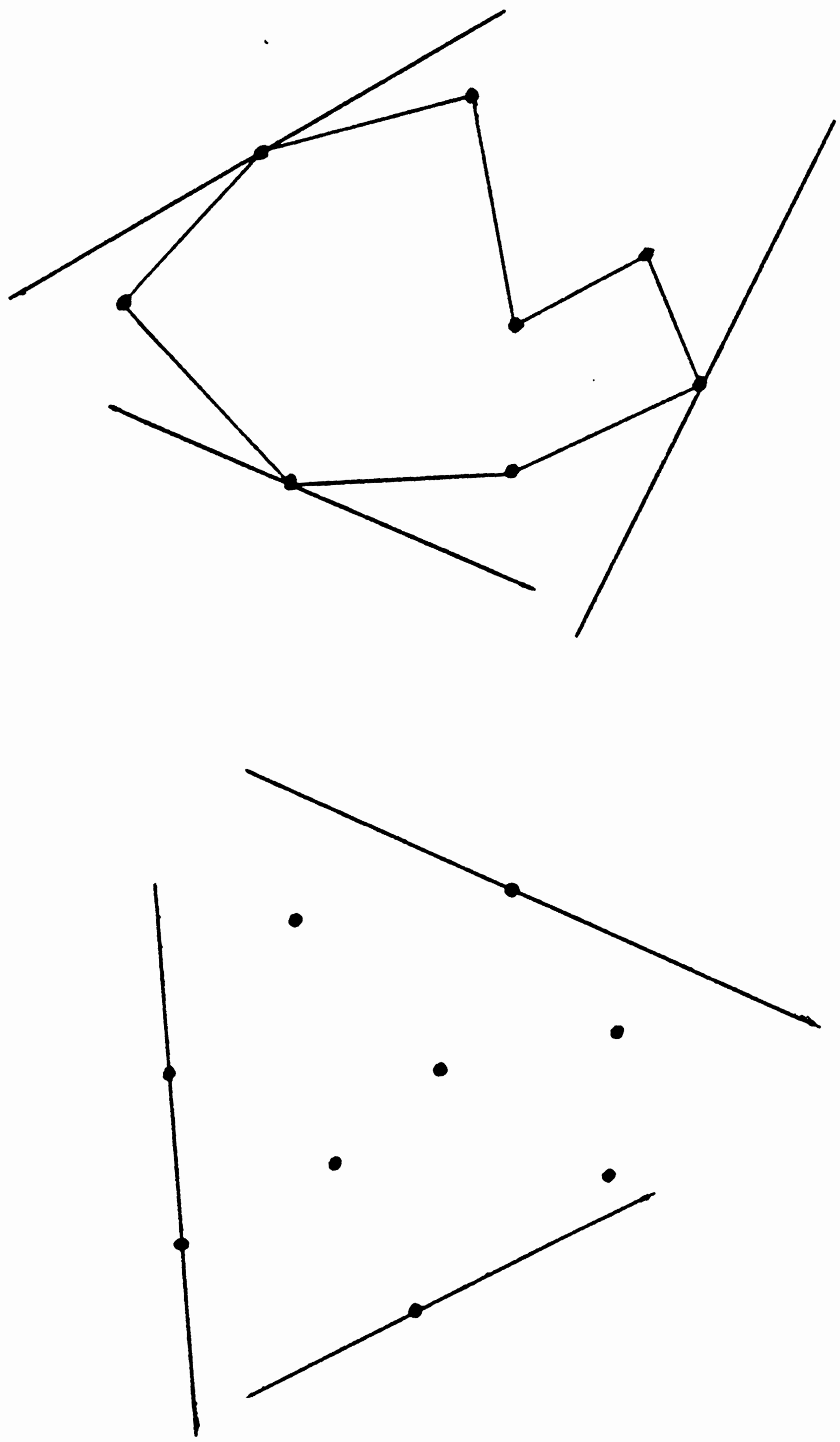


Fig. 5.2. Lines of support.

It should be clear that only certain pairs of vertices admit to parallel lines of support, such pairs of vertices we will call *antipodal.* Shamos [Shamos (78)] has shown that there are only O(n) such pairs for any convex polygon. Therefore, the only question that remains is how to efficiently enumerate these pairs.

Note the similarily of this statement about enumerating antipodal pairs and enumerating encasing rectangles. It turns out that only one general technique is necessary to enumerate all encasing rectangles and antipodal pairs of a convex polygon. This technique, *the highpoint strategy,* runs in O(n) time and when coupled with the Graham convex hull algorithm yields a worst-case O(nlogn) algorithm for all of problems 1 through 4. (Actually the convex hull of a simple polygon can be found in O(n) time [Lee (80)], which means problems 2 and 4 have O(n) worst-case complexity.)

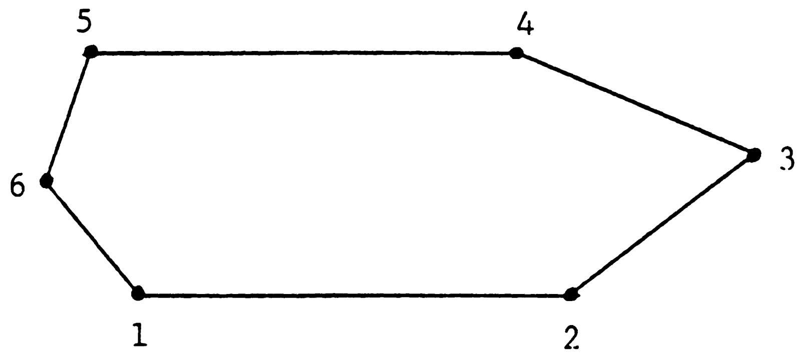
In the next Section we introduce the highest points problem and show how the highpoint strategy can be used to efficiently solve this problem. An understanding of this technique will help in the discussion of the algorithms which apply the strategy to solve the diameter and minimum area encasing rectangle problems.

5.3. Highpoint Strategy

Consider the following simple problem: given a convex polygon determine the vertex point (points) which has (have) the greatest perpendicular distance above each edge; Fig. 5.3. Certainly, an obvious way to attack this problem is to take each edge and record the height of the vertex immediately counterclockwise from it, then continue moving counterclockwise in turn recording the height of each successive vertex until one is reached whose height is smaller than its immediate predecessor p; p is the highpoint for that edge.

The only bad feature of this process is that it runs in $O(n^2)$ time. (On the average each edge will require n/2 height tests to find the highpoint.) Fortunately, there is a way of speeding up the process. Essentially, the idea is that once the initial highpoint H_1 is found for edge 1 it may be used as the starting point to find highpoint H_2 corresponding to edge 2, where edge 2 is adjacent and counterclockwise along the boundary of the polygon from edge 1. H_2 may then be used to find H_3, and in general H_i can be used as the starting point in computing H_{i+1}, where edge $i+1$ is adjacent and counterclockwise to edge i. This idea of moving counterclockwise to the next edge and using the previous highpoint as the starting point for the next highpoint is what we have previously referred to as the highpoint strategy.

We are now ready to present an algorithm to compute all highest points. In the algorithm we make the assumption that no more than two



edge	highpoints
12	4,5
23	5
34	1
45	1,2
56	3
61	3

Fig. 5.3. The highpoints of each edge of a convex polygon.

vertex points on the polygon are collinear, which implies that no edge can have more than two highpoints. (This assumption is nonrestrictive in the sense that the algorithm we present will still have the same complexity if the assumption is removed.) Whenever there are two highpoints corresponding to an edge of the polygon we define these points to be the *left* and *right* highpoints. The left highpoint is the counterclockwise successor of the right highpoint along the polygonal boundary. To compute the height S_p of a point p above an edge with endpoints (i,j) we employ formula (4) which was used in conjunction with the Eddy convex hull algorithm of the previous Chapter.

Algorithm Highest_points

Input: A doubly linked-list containing the ordered sequence of vertices on the convex polygon.

Output: All edges and their highest point(s).

Step 1: Locate the highest point(s) above an initial edge $\overrightarrow{ij}$ of the polygon. This can be carried out by scanning counterclockwise examining each pair of successive vertices A and B until the condition $S_B \leq S_A$ holds, Fig. 5.4. The scan starts with A := cclock(i), B := cclock(A). If $S_A = S_B$ then output A and B as the highpoints (A is the right highpoint, B is the left highpoint); otherwise output A as the lone highpoint.

Step 2: (Highpoint strategy) Move to the next edge. Let i := j; j := cclock(j); and find its highest point(s). Start the scan at the highpoint from the previous edge (or left highpoint if there are two), examining successive pairs of vertices A and B until $S_B \leq$

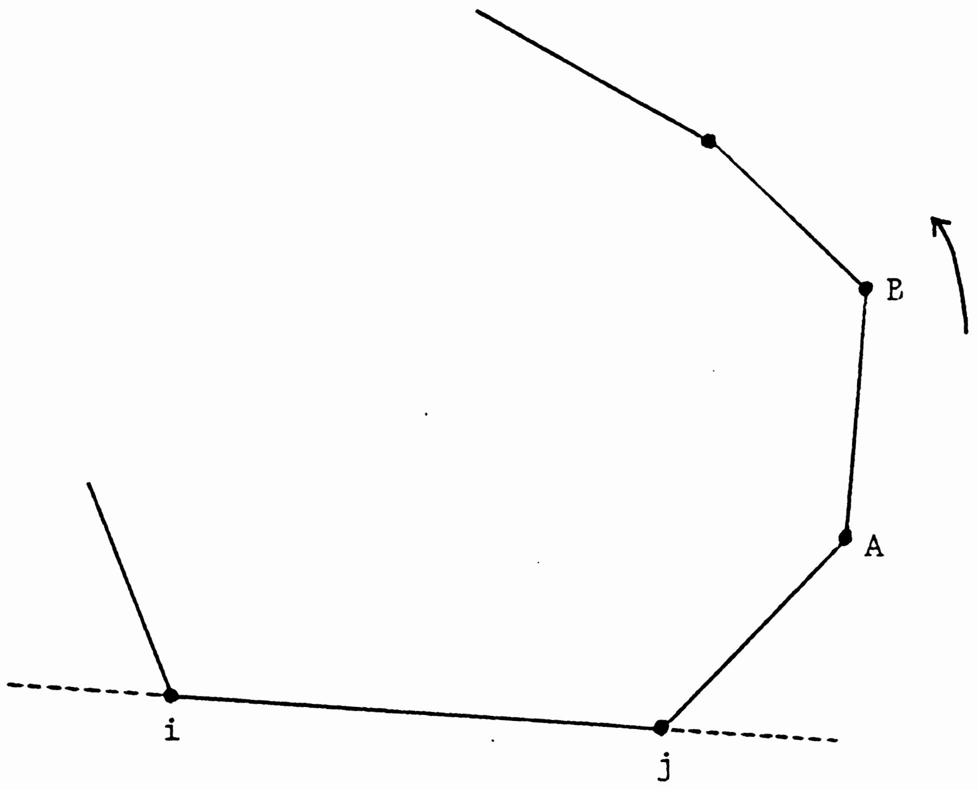


Fig. 5.4. Scanning for the inital highpoint(s).

S_A. Output the highpoint(s) (as in Step 1). Repeat step 2 until all edges have been traversed.

It is not hard to verify that algorithm highest_points produces the highest point(s) above every edge of an n-vertex convex polygon in O(n) time. Step 1 requires n/2 above-line calculations on the average, but never more than n. We can start the scan for the highest point of a new edge at the previous highpoint because all points between the new edge and the old highpoint are perpendicularly less distant than the old highpoint, Fig. 5.5. (Only points in the shaded area can be on the polygon yet not be the previous highpoint.) As each edge is traversed the scan for highpoints commences in a counterclockwise direction, never clockwise. Furthermore, the scan for highpoints can never reach the edge presently being traversed. Since in step 2, n-1 edges are traversed it follows that never more than O(n) vertices of the polygon are examined as possible highpoints. The actual number of scalar product calculations is approximately 3n since a calculation of S_A and S_B must be made for each new edge, and (on the average) a further point will have to be evaluated due to the counterclockwise migration of the highpoints.

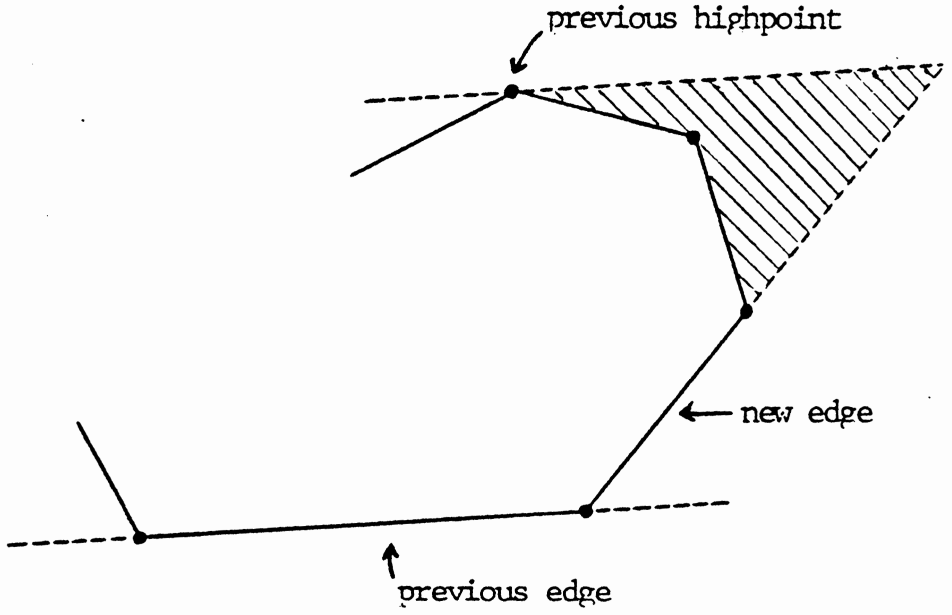


Fig. 5.5 The highpoint(s) for the new edge cannot be in the shaded region.

5.4. Enumerating Encasing Rectangles

The algorithm is a simple modification of the one given in the previous Section. It requires applying the highpoint strategy 3 times to compute each successive encasing rectangle. To compute the initial encasing rectangle we start at any edge of the polygon and scan counterclockwise applying the above-line test to each pair of adjacent edges A and B until $S_B < S_A$, Fig. 5.6. The perpendicular distance from A to line ij may then be computed by solving the following set of simultaneous equations to determine the point C where a perpendicular line from A crosses the line ij:

$$y_C - y_i = m(x_C - x_i),$$
$$y_C - y_A = -(x_C - x_A)/m,$$

where $m = (y_j-y_i)/(x_j-x_i)$ is the slope of line ij. The Euclidean distance formula can be used to calculate the distance ℓ between point A and C, Fig. 5.6. Note that ℓ is the length of one side of the encasing rectangle collinear to edge ij.

The length of the other side of the rectangle may be computed in a manner analagous to the procedure above. Starting at vertex j, scan counterclockwise to find the point D highest above line $\overrightarrow{AC}$, and similarly scan counterclockwise starting at A to find the vertex point E highest above $\overrightarrow{CA}$. Perpendicular lines emanating from D and E may then be dropped onto AC and CA, and their lengths, w_1 and w_2, computed by again solving the appropriate set of simultaneous equations

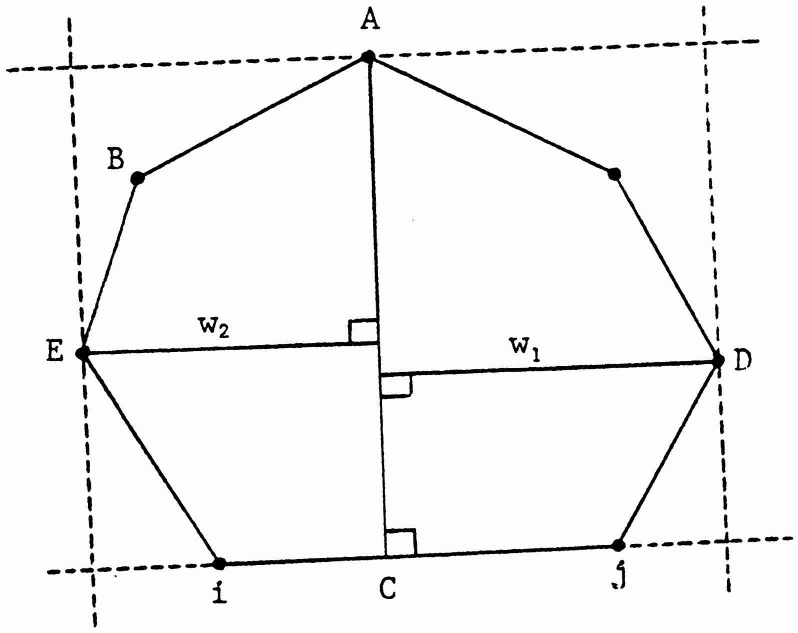


Fig. 5.6. Computing the initial encasing rectangle.

and applying the Euclidean distance formula. The sum of these lengths is the overall width of the encasing rectangle. Hence, the area of the encasing rectangle collinear to edge ij is $\ell * (w_1 + w_2)$.

Taking our cue from algorithm highest_points all remaining rectangles can be computed by judiciously applying the highpoint strategy. Let i := j; j := cclock(j). Use the previously determined highpoints as the starting points for the new highpoint scans. Once these points are found the area of the encasing rectangle can be computed by solving the appropriate sets of simultaneous equations and using the Euclidean distance formula (as before). If the area of this rectangle is smaller than any of the previously computed rectangles its corresponding edge and area replace the smallest of those already held in storage.

The process of enumerating all encasing rectangles, as described above, runs in O(n) time because the algorithm subjects the convex polygon to three passes of algorithm highest_points. The total implication of our work is that it is possible to compute the minimum area encasing rectangle of a set of n points in O(nlogn) time and an n-sided simple polygon in O(n) time.

The minimum area encasing rectangle algorithm was coded in FORTRAN and tested on an IBM 3032 (FORTX,OPT=2). Uniform random variates were generated on the boundary of an ellipse and passed to a modified Graham convex hull algorithm. For all sample sizes all points remained on the hull and were passed to a subprogram which computed the minimum area encasing rectangle of a convex polygon. In Table 5.1 we give the time taken by this subprogram. 5 realizations of 100 runs were made for each sample size. Average times are in seconds and appear in the Table along with their standard deviations. As expected, the results indicate that the minimum area encasing rectangle subprogram runs in O(n) time.

Table 5.1 - Average time to find the minimum area encasing rectangle of a convex polygon with n vertices (standard deviations in parenthesis).

n	min. encasing time
125	3.556 (.0114)
250	7.096 (.0344)
500	14.184 (.0532)
1000	28.116 (.0729)

5.5. Enumerating Antipodal Pairs

Recall that an antipodal pair of vertices on a convex polygon admits to parallel lines of support. To enumerate all pairs efficiently Shamos suggested treating the edges of the convex polygon as vectors and translating them to the origin, Fig. 5.7. In this transformation, edges go to vectors, and vertices to sectors. All antipodal pairs may then be found by extending an infinite line L through the origin and rotating it counterclockwise. The antipodal pair does not change until L passes through some new vector of the diagram. In Fig. 5.7, pair 3,6 turns into 4,6 as L passes through vector $\overrightarrow{34}$; 4,6 turns into 4,1 and so on.

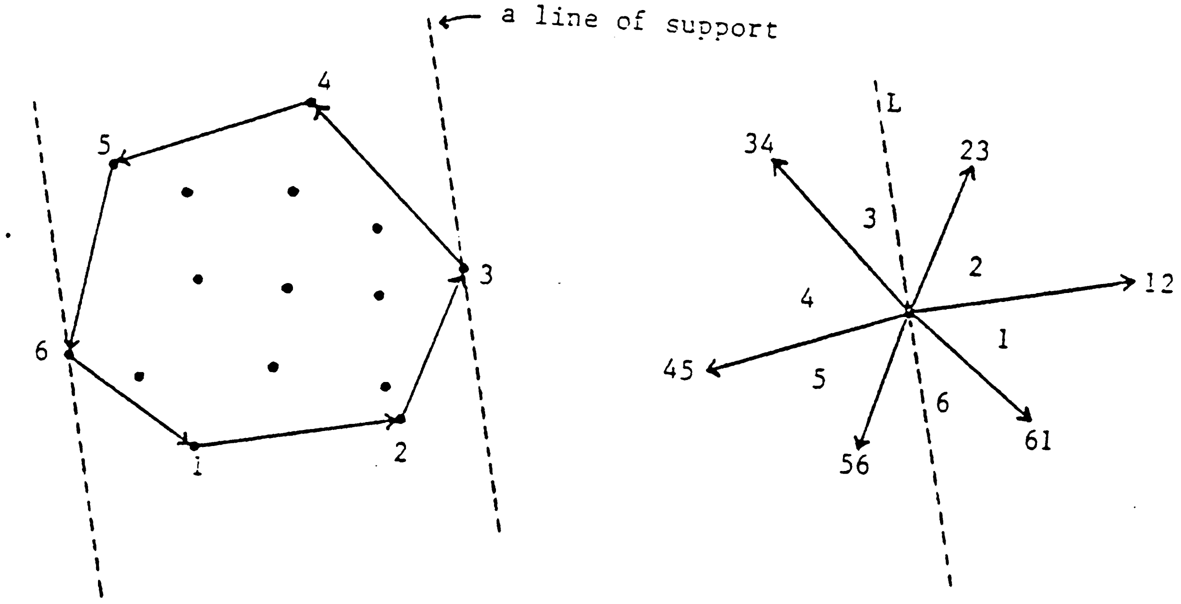


Fig. 5.7. The Shamos antipodal pair finder.

It is clear that, for each vector passed, a new antipodal pair is determined. (If two vectors are simultaneously encountered 4 new antipodal pairs result.) Because there are n vectors to pass, it follows that by scanning sequentially around the diagram (swinging line L through at least 180 degrees), O(n) time is required to compute all antipodal pairs.

While the Shamos diagram is convenient for showing that it is indeed possible to compute all antipodal pairs in O(n) time, the diagram hides the true simplicity of the method. Note that in Fig. 5.8, lines collinear to edges $\vec{AB}$ and $\vec{BC}$ are two lines of support of the convex polygon passing through vertex B. Let the highpoints above these two lines be H_q and H_r, respectively. (H_q is the right highpoint of $\vec{AB}$ if $\vec{AB}$ has two highpoints and correspondingly H_r is the left highpoint of edge $\vec{BC}$ if $\vec{BC}$ has two highpoints.) Because H_r is the highpoint for edge $\vec{BC}$, parallel lines of support can pass through H_r and B. Likewise, since H_q is the highpoint for edge $\vec{AB}$, parallel lines of support can pass through H_q and B. That is, (B,H_q) and (B,H_r) are antipodal pairs. It should be clear that only the chain of vertices on the convex polygon between H_q and H_r will admit to parallel lines of support in conjunction with point B; Fig. 5.8 (shaded region). The argument generalizes and thus for any vertex point B on a convex polygon, the antipodal pairs corresponding to B are the sequence (B,H_q), ..., (B,H_r), where H_q and H_r are the vertices that are the highpoints above each of the edges adjoining B.

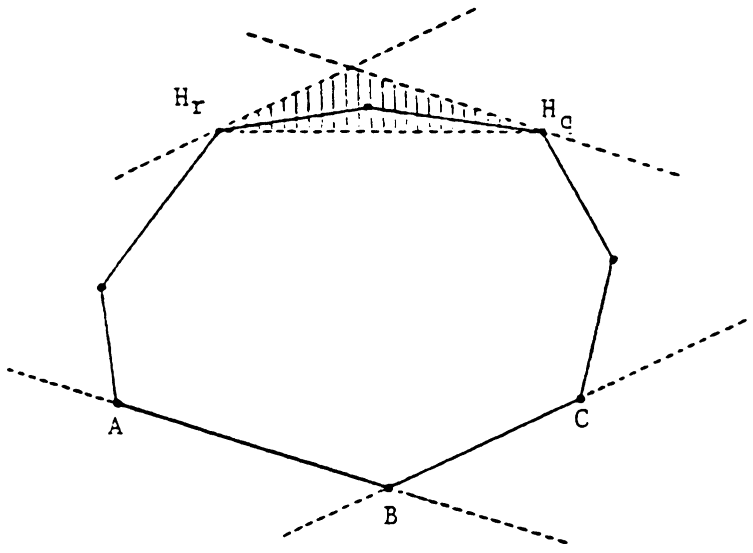


Fig. 5.8. Finding antipodal pairs using the highpoint strategy.

In this way the original problem of finding antipodal pairs has been transformed into one of enumerating highpoints, for which we already have an efficient and simple algorithm. A trivial modification of this algorithm will yield all antipodal pairs and diameter of a convex polygon. We give the algorithm to compute the diameter of a convex polygon.

Algorithm Largest_antipodal_pair

Input: A doubly linked list holding the vertices of the convex polygon.

Output: The endpoints of the diameter and its length.

Step 1: Start with any vertex i on the polygon and its two common edges. Find the highest point above each edge (these points could possibly be the same). Label these points H_q and H_r. Compute the interpoint distances between i and all vertices in the chain H_q ... H_r. Keep only the largest of these distances L_p and its corresponding antipodal pair (A_j, A_k). Let Init_point := H_q.

Step 2: Let i := cclock(i). Find the highest point above each edge adjoining i. Set H_q := H_r (or H_q := clock(H_r) if two highpoints) and use the highpoint strategy to find H_r. Compute the distance between i and all vertices in the chain H_q ... H_r. As these values are computed, compare with the largest pair already in storage, and if necessary reassign L_p and (A_j, A_k). Repeat step 2 until i := Init_point.

The only tricky part of the algorithm is the introduction of the auxiliary variable Init_point. The purpose of Init_point is to prevent the algorithm from scanning entirely around the polygon. This would be wasteful of time because each antipodal pair would be produced twice by the enumeration algorithm. The use of Init_point corresponds to checking if the line L has been rotated through 180 degrees in the Shamos algorithm. Note also, when two edges of the polygon are parallel we must backtrack to produce an extra antipodal pair. This corresponds to the special case in the Shamos algorithm when L passes two edges simultaneously.

5.6. Performance Test

We have argued for the superiority of our method versus the Shamos antipodal pair enumerator on the basis of simplicity. It turns out that in coding both routines this fact is very evident. Two problems with converting the description of the Shamos algorithm to actual code are handling the special case of passing two vectors simultaneously and devising an elegant and efficient method for determining the slope of the edges of a convex polygon.

Concerning the latter problem the avoidance of trig functions was of major importance. Using knowledge gained in implementing the Graham and Jarvis convex hull algorithms we substituted a ratio of sides formula as follows: First we set $y := y_j - y_i$ and $x := x_j - x_i$. Next, if $y \geq 0$ then we set

$$\text{angle}_i := -x/(|x| + y) + 1.0,$$

else we set

$$\text{angle}_i := x/(|x| + |y|) + 3.0.$$

The formulae compute the angular orientation each edge E_{ij} with endpoints (x_i,y_i) and (x_j,y_j) makes with a half-line emanating from point i and extending parallel to the positive x-axis. In this system an edge E_{ij} orientated at 0 radians would have an $\text{angle}_i = 0$, an edge E_{ij} oriented at $\pi/2$ radians an $\text{angle}_i = 1.0$, and edge E_{ij} oriented at π radians an $\text{angle}_i = 2.0$, etc. The relationship between radians and angles is:

$$\text{radians}_i := \text{angle}_i * (2\pi/4.0)$$

FORTRAN versions of algorithm largest_antipodal_pair (LPAIR) and a similar routine based upon the Shamos vector data structure (DIAM) were coded for testing. Uniform random variates were generated for two distributions: uniform on the boundary of an ellipse and uniform on the boundary of a circle. The Graham convex hull algorithm was used to find the convex hull for each sample. In each case all points remained on the hull. Five realizations of 100 runs were made; average times and standard deviations (for just LPAIR and DIAM, not the convex hull routine) appear in Tables 5.2 and 5.3. As can be seen in the Tables, LPAIR runs faster than DIAM. Coupled with its conceptual simplicity and ease of coding it is the method of choice.

Table 5.2 - Average time to find the diameter of a convex polygon; points generated uniformly on the boundary of an elipse. (Standard deviations in parenthesis.)

n	LPAIR	DIAM
125	.488 (.0045)	.534 (.0055)
250	.956 (.0114)	1.048 (.0110)
500	1.874 (.0288)	2.106 (.0195)
1000	3.702 (.0239)	4.198 (.0698)
2000	7.440 (.0908)	8.546 (.2218)

Table 5.3 - Average time to find the diameter of a convex polygon; points generated uniformly on the boundary of a circle. (Standard deviations in parenthesis.)

n	LPAIR	DIAM
125	.482 (.0084)	.524 (.0063)
250	.938 (.0084)	1.036 (.0241)
500	1.862 (.0130)	2.086 (.0219)
1000	3.690 (.0406)	4.082 (.0492)
2000	7.174 (.0440)	8.130 (.0762)

5.7. *The Diameter of a Set in R_1*

We have seen that the convex hull is useful in solving the diameter of a set problem where the distance function is d_2 and the points are in R_2. Could it be that two points on the L_1 hull comprise the diameter of a set in R_1? It turns out that indeed this is actually the case. The proof, which we leave as an exercise to the reader, is by contradiction and almost identical to the proof which showed that the two points which comprise the diameter of a set in R_2 are on the convex hull.

Since the two points which comprise the diameter are on the L_1 hull our first inclination is to see if there exists a paradigm in R_1 to the antipodal enumeration technique discussed in the previous Section. Unfortunately, the counterpart of a line of support in R_2 does not exist in R_1 because of the orthogonal path lines are restricted to in the space.

It appears that in our search for an efficient algorithm computing the L_1 hull will not be of any help. We must look for another property or structure that might be exploited to solve the problem. Since we have already found the "special" points ymin, xmax, ymax, and xmin to be of help in computing the L_1 hull it may be to our advantage to look at these points in the context of the problem at hand. Recall that ymin, xmax, ymax, and xmin were important because they divided the L_1 hull problem into four subproblems. This suggests that region delineation may be the key to the problem of finding the diameter.

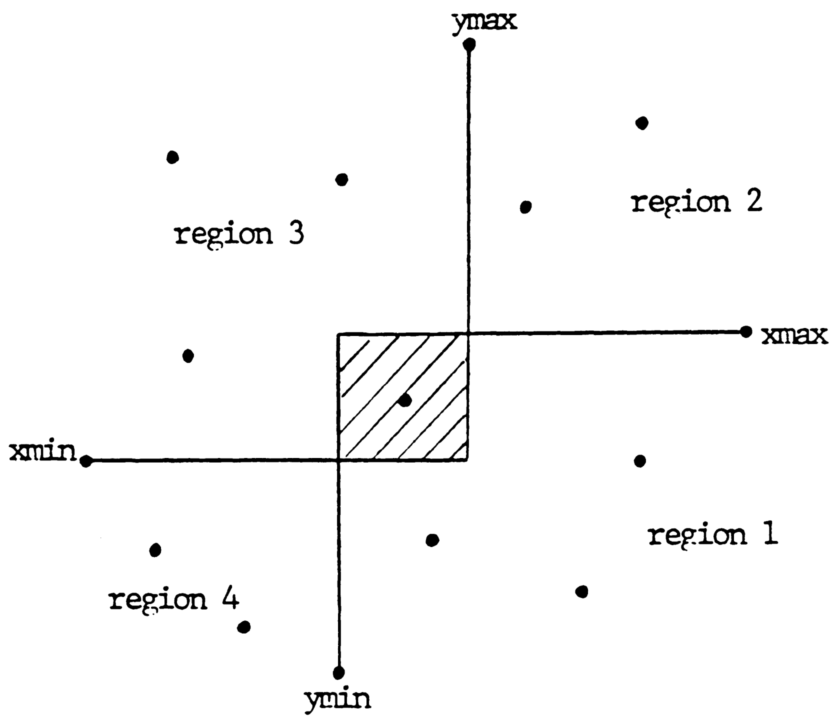
Let region 1 contain all points i of the set such that $x_i \geq x_{ymin}$ and $y_i \leq y_{xmax}$; region 2 contain all points i of the set such that $x_i \geq x_{ymax}$ and $y_i \geq y_{xmax}$; region 3 contain all points i of the set such that $x_i \leq x_{ymax}$ and $y_i \geq y_{xmin}$; region 4 contain all points i of the set such that $x_i \leq x_{ymax}$ and $y_i \leq y_{xmax}$.

From the definition each of the special points belongs to two bordering regions. For example, ymin belongs to both regions 1 and 4. Figure 5.9 reveals that in some cases regions 1 and 3 overlap, or that regions 2 and 4 may overlap. It is also possible that no regions overlap and that there are possibly interior points which do not fall into any of the four designated regions.

Two theorems are almost immediately evident from the examples. One is that if two points a and b are in the same region then they cannot form the diameter pair except when they are both special points. (The proof is trivial.) Second, if two points a and b are in adjacent regions, *i.e.,* 1:2, 2:3, 3:4, 4:1, and they are both non-special then they cannot form the diameter pair. As proof, consider adjacent regions 1 and 2. Let a belong to region 1 and b belong to region 2. Then either $d_1(a,b) < d_1(a,ymax)$ or $d_1(a,b) < d_1(b,ymin)$. Similar proofs exist for the other adjacent regions.

We have made a small discovery, that the diameter consists of two points from opposing regions: either 2:4 or 1:3. Realizing that these regions may overlap, it is natural to wonder if there are pairs within these regions that cannot comprise the diameter. For the moment consider regions 2 and 4, four special cases must be examined: (i) there is no overlap -- $x_{ymin} \leq x_{ymax}$ and $y_{xmin} \leq y_{xmax}$; (ii) there is overlap in the x-coordinate only -- $x_{ymin} >$ xymax and $y_{xmin} \leq y_{xmax}$; (iii) there is overlap in the y-coordinate only -- $x_{ymin} \leq x_{ymax}$ and $y_{xmin} > y_{xmax}$; (iv) there is overlap in both the x and y coordinates -- $x_{ymin} > x_{ymax}$ and $y_{xmin} > y_{xmax}$.

When the regions have no overlap, as in case (i), then the diameter line segment can be made to pass through the inside corner of either region. (For region 2 the inside corner has coordinates (x_{ymax}, y_{xmax}).) Now, if we take all points in region 2 and compute



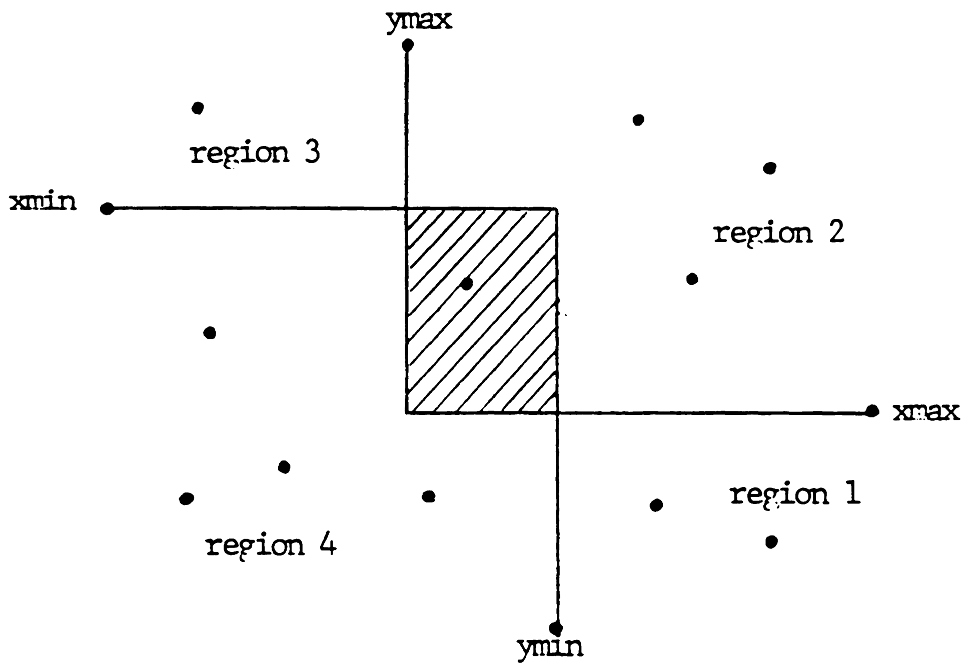


Fig. 5.9. Region delineation.

the distance to one of the two inside corner points and take the largest of these, and likewise in region 4 determine the point which is farthest from the same interior corner point, then these two points must comprise the (possible) diameter among all possible pairs of points in the two regions. The computation takes O(n) time where n is the number of points in both regions.

Next we consider the case where there is overlap in the x-coordinate, as in (ii). Subdivide region 2 such that all points whose x-coordinate $< x_{ymin}$ fall into subregion A, with the remaining points falling into subregion B. Also, subdivide region 4 such that all points whose x-coordinate $> x_{ymax}$ fall into subregion D, with the remaining points falling into subregion C; Fig. 5.10. Among the possible pairs of subregions only the combination of A and D yields no possible diameter pairs, since the distance between ymin and ymax is greater than any combination of two points from these regions. (The pair of points ymin and ymax would be examined as part of regions 1 and 3.) Finding the largest pair in the subregion combinations A:C, B:C, and B:D can be carried out in a similar manner to the inside corner point algorithm described for case (i). Furthermore, the complexity is still O(n) because splitting the points into the various subregions requires O(n) time.

Case (iii), when there is y-coordinate overlap, is isomorphic to case (ii) and therefore the possible diameter pair can again be found in O(n) time.

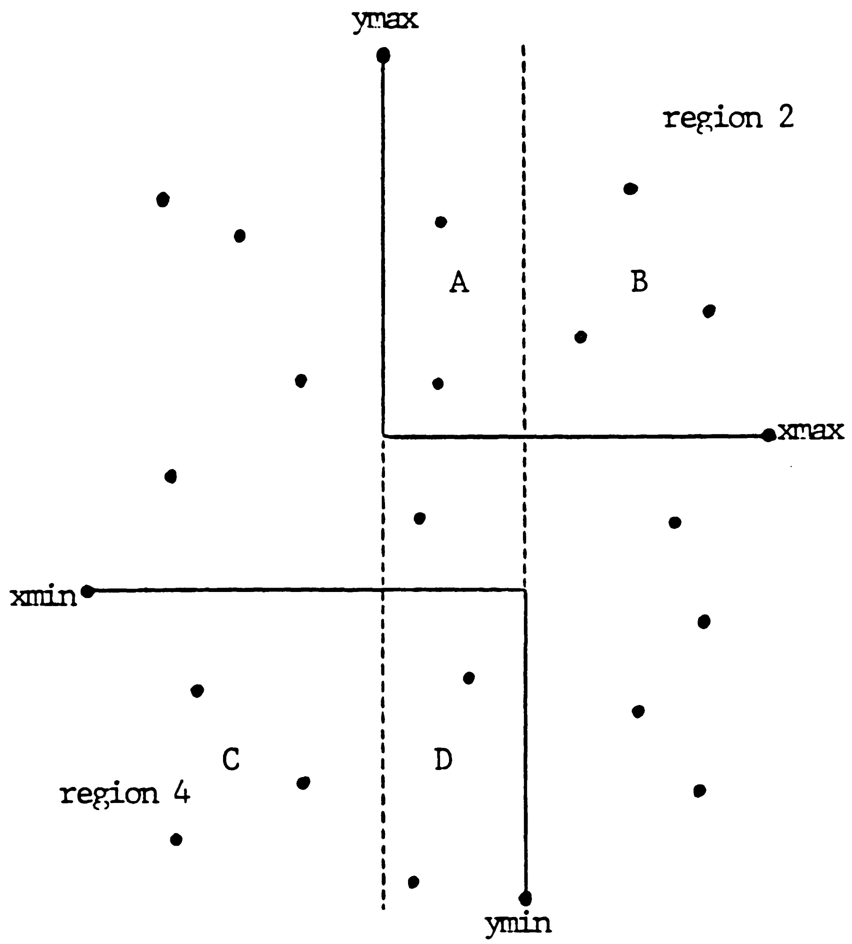


Fig. 5.10. Subdividing regions 2 and 4.

The final case, (iv), is a combination of cases (ii) and (iii). The points in region 2 can be subdivided into one of four subregions as follows: region A contains all points whose x-coordinate $\geq x_{ymin}$ and y-coordinate $\geq y_{xmin}$; region B contains all points whose x-coordinate $< x_{ymin}$ and y-coordinate $\geq y_{xmin}$; region C contains all points whose x-coordinate $\geq x_{ymin}$ and y-coordinate $< y_{xmin}$; region D contains all points whose x-coordinate $< x_{ymin}$ and y-coordinate $< y_{xmin}$. Likewise, region 4 can be subdivided as follows: region E contains all points whose x-coordinate $\leq x_{ymax}$ and y-coordinate $\leq y_{xmax}$; region F contains all points whose x-coordinate $\leq x_{ymax}$ and y-coordinate $> y_{xmax}$; region G contains all points whose x-coordinate $> x_{ymax}$ and y-coordinate $\leq y_{xmax}$; region D contains all points whose x-coordinate $> x_{ymax}$ and y-coordinate $> y_{xmax}$; Fig. 5.11.

The diameter will come from either subregion pair A:E, A:F, A:G, B:F, B:E, C:E, or C:G. It should be evident from Figure 5.11 that any point in region D cannot be an endpoint of the diameter. Since there are seven subproblems that can contain at most O(n) points it follows that the total time required to split the points and determine the largest pair is O(n) (using the method described for case (i)).

For regions 1 and 3 the proof is entirely analgous to the above discussion, and will not be given. Thus, we can compute a possible diameter pair for each opposing pair of regions in O(n) time. Taking the largest of these solves the original problem.

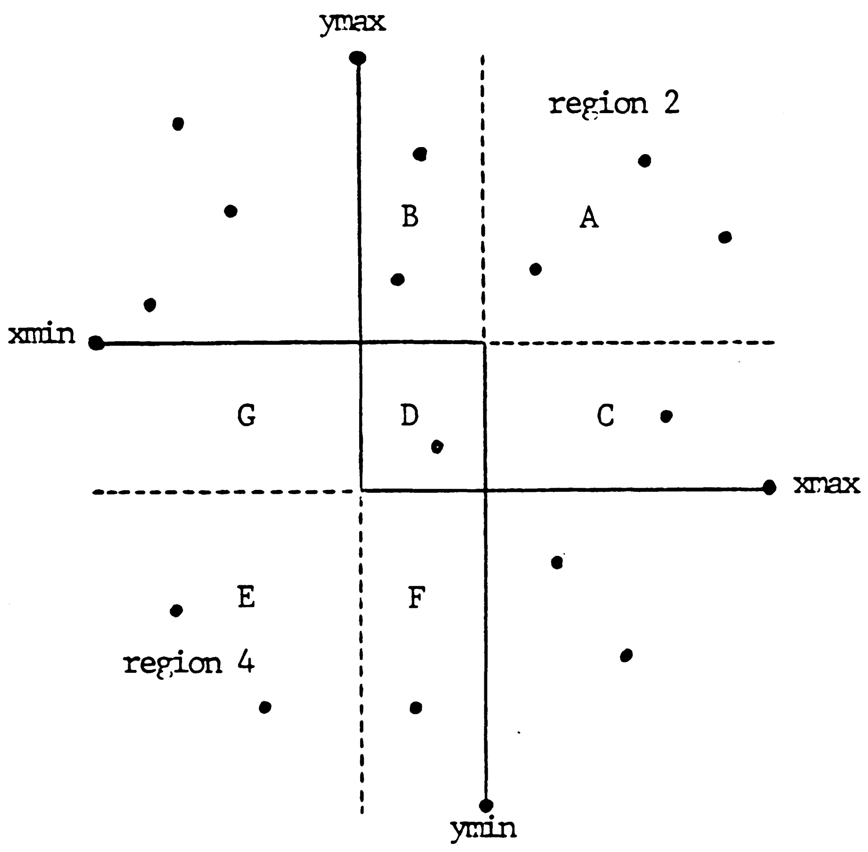


Fig. 5.11. The case where regions 2 and 4 overlap in both coordinates.

That we can always compute the R_1 diameter of a set in O(n) time is somewhat surprising in light of the fact that the R_2 diameter requires O(nlogn) worst-case time. The key to the speed of the algorithm is the non-uniqueness of L_1 line segments. That is, because the L_1 line segment between each possible pair could be made to pass through the same fixed interior point the problem became computationally less burdensome.

Throughout this section we have assumed that the four special points ymin, xmax, ymax, and xmin are unique. Obviously, we have overlooked several degenerate cases as follows:

Case 1: ymin = xmin;

Case 2. ymin = xmax;

Case 3. ymax = xmin;

Case 4. ymax = xmax;

Case 5. ymin = xmin and ymax = xmax;

Case 6. ymax = xmin and ymin = xmax;

Cases 1-4 can all be handled by applying the techniques used to solve the general problem when the special points are all unique. We leave the details to the reader. Cases 5 and 6 are trivial -- one solution is (ymin,ymax).

Chapter 6

THE EUCLIDEAN AND L_1 TRAVELING SALESMAN PROBLEMS

"Although this may seem a paradox, all exact science is dominated by the idea of approximation."

- Bertrand Russell

6.1. Introduction to the Traveling Salesman Problem

One of the most famous and widely researched problems in the scientific literature is the *traveling salesman problem* (TSP). Over the past 50 years computer scientists, engineers, management scientists, operation researchers, and mathematicians have been interested in finding efficient techniques for solving this problem. The vast number of applications in which the problem arises has been the major reason for its serious investigation. An allegorical way of the stating the problem is as follows: *"A number of cities are given along with the cost of traveling between each pair. Starting at one city a traveling salesman (salesperson) wishes to visit each of the remaining cities and return to his point of departure. What itinerary should he follow in order to minimize the cost of his trip?"*

The obvious brute-force algorithm that immediately comes to mind is to enumerate all possible tours and choose the smallest one. Unfortunately, this enumeration technique is infeasible for all but the smallest problems because of its exponential time complexity. If n is the number of cities, then there are exactly (n-1)! different tours, and since

$$n! \sim \sqrt{(2\pi n)}(n^n)(e^{-n}) = O(n^n/e^n)$$

by Stirling's approximation, it follows that even on the fastest modern digital computers it would take centuries to compute a problem of size 25 cities.

There are other exact methods which are somewhat successful on problems of size $n < 45$. Most of these are based upon quickly eliminating those tours which cannot be among the cheapest ones. Branch and bound, dynamic programming, and linear programming are some of the various techniques used in these algorithms. See [Akl (78a)] and [Reingold (77)] for a summary of these methods.

For larger problems another approach has been adopted involving the use of *heuristics* to yield a solution that is near optimal. (A heuristic is defined to be an algorithm which finds a near optimal solution or admits failure and returns no answer.) The relaxation of the optimality constraint frequently allows algorithms to be more efficient, since an all out exhaustive search is only approximated. Good heuristic algorithms are usually not hard to derive because it is a task human beings are called upon to do every day throughout their lives (a simple example would involve crossing a street in heavy traffic). In the case of the TSP it is even possible to show that some heuristics find solutions that are guaranteed to be close to optimal.

The reason we are so quick to dismiss the search for an exact algorithm is that it has been shown by Karp [Karp (72)] that the TSP is a member of a class of problems *(NP-complete or NP-hard problems)* for which no polynomial time algorithm (i.e., reasonable algorithm) is likely to be discovered. Three such problems are the *satisfiability problem* from mathematical logic, determining whether an undirected graph has a complete subgraph of size k, and determining if a digraph has a Hamiltonian cycle.

The key to understanding the class NP is to know that each of the problems in the class can be reduced to anyone of the other problems in polynomial time. This immediately implies that if any one problem in NP could be solved in polynomial time by some algorithm, then all others could also be solved in polynomial time. To date, no one has produced a polynomial-time algorithm for any member of the class.

6.2. Introduction to the Euclidean Traveling Salesman Problem

A special case of the TSP is the *Euclidean traveling salesman problem* (ETSP) which has several applications in management science. Here the cities are points in the plane with given (x,y) coordinates. The problem appears to be simpler than the TSP for several reasons. It is in general unnecessary to store the distance between each pair of cities, because distance can be immediately derived by evaluating the Euclidean distance formula. In the TSP $O(n^2)$ storage is required to store all intercity costs which implies a trivial lower bound of $O(n^2)$ for any heuristic to compute the TSP. On the other hand, the corresponding lower bound for the ETSP is O(n) since only the time to read in the (x,y) coordinates is needed. The ETSP has three other exploitable properties: (i) $d_2(i,j) + d_2(j,k) > d_2(i,k)$ for any three cities $i \neq j \neq k$ (i.e., the triangle inequality), (ii) *symmetry,* $d_2(i,j) = d_2(j,i)$ for all $i \neq j$, (iii) the optimal tour can be shown to be intersection free under the assumption that all of the cities do not lie on the same line [Bellmore (68)].

Another feature that the ETSP possesses is that it is easily visualized and examples can be carried out with paper and pencil. A human can usually obtain a rather good approximation of the optimal tour using these implements. Visualization also facilitates the development of good heuristics, which is a necessity since the ETSP, like the TSP, belongs to the class NP [Garey (76)].

Our research centers on four geometrically motivated algorithms, all based upon convex hull determination. The major theme in all four is to form convex rings of some subgroup of points and then to merge these rings according to a cheapest cost local optimization rule. It has been shown [Eilon (71)] that the order in which points appear on the convex hull is the same in which they will appear on the optimal tour. This observation, which follows from the fact that the optimal tour cannot intersect itself, serves as the impetus for research on convex hull based heuristics.

W. Stewart, Jr. [Golden (80)] has proposed an algorithm which uses as the initial *subtour* the convex hull of the set of points. (A subtour is a tour of some subset of the cities.) Because this algorithm has performed with suprising accuracy we have decided to program and test his algorithm against our methods.

In the next several Sections we give detailed explanations of our algorithms, Stewart's algorithm, and three additional general TSP heuristics, *nearest neighbor, farthest insertion, and nearest insertion.* All of these algorithms have been coded in PASCAL, tested and compared for quality of tours produced. The details of this test are given in Section 6.10.

6.3. Random Hull Peel Heuristic

The first algorithm we propose illustrates the major theme throughout all of our algorithms, which is to merge points from an inner convex hull (ring) into an outer subtour. The outer subtour will always contain or enclose any of the points on the inner ring.

Step one involves finding all convex hull peels as follows:

```
Step 1.1.  i := 1;
Step 1.2.  $Hull_i$ := CH(S);
Step 1.3.  S := S - $Hull_i$;
Step 1.4.  If |S| = 0 then HALT;
Step 1.5.  i := i + 1;
```

Step 1.6. go to Step 1.2.

The procedure effectively computes one hull then strips it away, computes the next hull, strips it away, and so forth until no points remain; Fig. 6.1. The set of convex hull peels produced is stored in an array of linked-lists to allow constant time insertions and deletions. Incidentally, the number of peels induced by this process is called the depth of the set, which is a statistically relevant quantity [Shamos (78)].

The final step is to merge all the convex hull peels into a tour of all the points. First, the initial subtour is assumed to be the outermost convex hull, *i.e.,* T := Hull_1. Hull_2 is then merged into T as follows: A random point k^* is located on Hull_2, k^* is then placed into T (T := T + k^*, Hull_2 := Hull_2 - k^*) between the two adjacent points i^* and j^* that minimize the cost rule R1:

for all (i,j) in T find the (i^*,j^*) *such that*
$d_2(i^*,k^*) + d_2(k^*,j^*) - d_2(i^*,j^*)$ *is minimal.*

For the general step, each new k^* (the point to be inserted) is the counterclockwise successor of the last k^* on Hull_2, and is inserted as above. When no points remain on Hull_2, Hull_3 is stripped and merged into T by using the same process, then Hull_4 is next, and so on until no inner peels remain. We give a summary of the total process below.

Step 2.1. T := Hull_1;
Step 2.2. i := 2;

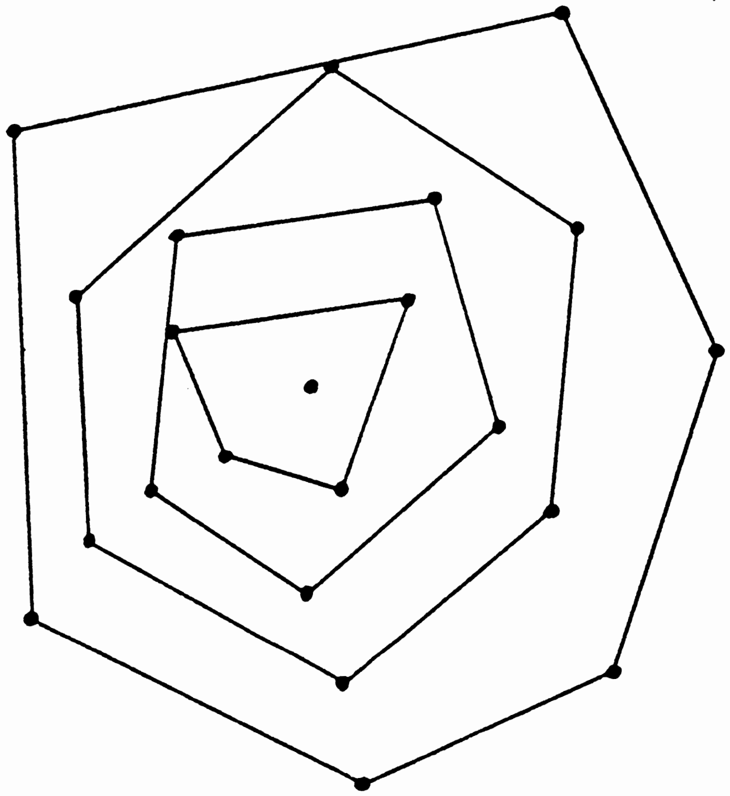


Fig. 6.1. Convex hull peels.

```
Step 2.3.  If $i >$ number of hulls then HALT;
Step 2.4.  Find a random point $k^*$ on $\text{Hull}_i$;
Step 2.5.  Let k*_succ := cclock($k^*$);
Step 2.6.  Insert $k^*$ into T between $i^*$ and $j^*$ according to rule
    R1, and remove $k^*$ from $\text{Hull}_i$;
Step 2.7.  If $k^*$_succ $\neq k^*$ then set $k^*$ := $k^*$_succ; go to step
    2.5;
Step 2.8.  i := i + 1;
Step 2.9.  Go to Step 2.3;
```

We proceed now with a discussion of the complexity of the procedure. The number of convex rings generated by the first step could be as large as $\lceil n/3 \rceil = O(n)$, with three points forming each ring, except for possibly the innermost hull which could have two points or one point. For a uniform distribution of points in the plane empirical tests show that the expected number of hull peels is actually much smaller; see Table 6.1. A theoretical determination of the expected number of hull peels is an open problem in the field of stochastic geometry (geometrical probability). The problem of determining the number of points on the outside hull has been solved for several different planar distributions [Raynaud (70)], [Renyi (68)], [Bentley (78)]. Each solution is dependent on the shape of the region in which the points were generated, and thus an extension of this technique without simplifications seems infeasible for the hull peel problem because there can be no exact shape hypothesis for the outer hull (or any of the hull peels for that matter).

Table 6.1. Average number of convex hull peels for a uniform distribution of points inside a square region in the plane.

n	repetitions	average depth	standard deviation
50	100	6.72	0.552
100	100	10.48	0.594
200	100	16.54	0.642
300	100	21.57	0.714
400	100	26.23	0.815
500	100	30.29	0.844
600	50	34.20	0.606
700	50	37.80	0.881
800	50	41.10	0.814
900	50	44.44	0.760
1000	50	47.42	0.859

To compute all convex hull peels we could repeatedly apply any one of the convex hull algorithms discussed in Chapter Four. In the expected case the Eddy algorithm would be preferred since it works well with any type of uniform distribution of points. Worst-case performance could result from either the maximum number of hull peels or one hull peel. In the first case the algorithm would be called $\lceil n/3 \rceil$ times with performance directly proportional to the sum

$$\sum_{i=3,6,9,\ldots}^{n/3} i = O(n^2).$$

The second case reduces to the worst-case of the Eddy algorithm which is $O(n^2)$.

Average-case performance can be bounded above and below by $O(n^2)$ and $O(n\log n)$. The lower bound follows immediately from the expected performance of the Eddy algorithm when all points are on the hull, which again is analagous to the average case of Quicksort.

The time to insert one point from an inner hull into the outer subtour is proportional to the number of elements in the subtour. Therefore, assuming that there are m points on the initial subtour and (n - m) points remaining on all interior hulls, the time to insert all interior points is given by the sum

$$\sum_{i=1}^{n-m}(m+i-1) = \sum_{i=m}^{n}(i-1) = \sum_{i=m-1}^{n-1} i$$

$$= \sum_{i=1}^{n-1} i - \sum_{i=1}^{m-2} i = [n(n-1)/2] - [(m-1)(m-2)/2] \qquad (1)$$

For a uniform distribution of points $m \sim k\log n$ for some positive constant k and thus (1) reduces to

$$[(n^2 - n)/2] - [(k^2\log^2 n - 3k\log n + 2)/2] = O(n^2).$$

In is clear that unless m is approximately equivalent to n (almost all points are on the hull), (1) will be $O(n^2)$. Hence, the time to compute both steps, finding all hulls, and inserting points into the outer enclosing subtour is clearly bounded above by a function $T(n) = O(n^2)$, and this is also the expected-case time complexity.

It should be evident that a best-case situation is possible in both time complexity and length of tour, and these can occur for the same input instance. If all points fall on the outermost hull (the only hull), then the tour length will be optimal and the time taken to compute the tour will be the expected case of the Eddy algorithm when all the points are on the hull which is O(nlogn).

A similar analysis of the random hull peel heuristic can be carried out under the assumption that the Graham convex hull algorithm is used to find all hull peels. Because the Graham algorithm has a worst-case running time of O(nlogn), step one could take $O(n^2 \log n)$ time. However, the best case, when all points are on one hull, could take as little as O(n) time. The expected-case for a uniform distribution of points will be $O(n^2)$, since for each hull peel found the Graham algorithm will take O(n) time (or less) and thus the insertion time (step 2) will dominate.

6.4. Cheapest Insertion Hull Peel Heuristic

The second of our algorithms might be considered somewhat more sophisticated than the random hull peel heuristic. Instead of inserting points into the subtour starting at a random point on an interior hull, the basic strategy is to always take the point k^* on the present ring which minimizes the cheapest cost rule R2:

> *For each k on the present ring find the (i,j) in the subtour such that $d_2(i,k) + d_2(k,j) - d_2(i,j)$ is minimal. Then, for all triples (i,j,k), determine the (i^*,j^*,k^*) that minimizes $[d_2(i,k) + d_2(k,j)]/d_2(i,j)$.*

The intuitive appeal of this rule is that it is a combination of both cost and ratio. Therefore, the act of creating a new subtour by adding a new point usually involves little angular deviation from the previous subtour.

Step one is again similar to the random hull peel heuristic. All convex hull peels are computed and stored into an array of linked-lists. The outer hull is the initial subtour and we start by inserting points one by one from the 2nd outermost hull until all points on this ring have been exhausted. The procedure is made iterative by moving to the 3rd outermost ring and inserting all points as before. Then the 4th ring is stripped, and the procedure continues until no interior rings remain.

For computing all hull peels the time complexity and implementation details are similar to the random hull peel heuristic. However, the process of efficiently merging the rings is somewhat more complicated. For each point on the present ring the two points (i,j) on the enclosing subtour which are consistent with cost rule R2 are maintained. Whenever a point is inserted, the list of present ring points is scanned for the k^* that has the smallest minimum cost (a minimum of minimums). k^* is then inserted between the two points (i^*,j^*) in the subtour which correspond to the smallest cost. k^* is also deleted from the present ring structure and the list of minimum cost points is updated to reflect the addition of k^* to the subtour and the deletion of the (i^*,j^*) adjacent pair in the tour.

The total time to completely merge an inner convex hull ring with q points to the outer enclosing subtour with p (initial) points is given by the following sum:

$$c_1pq + c_2\sum_{i=1}^{q-1} i + c_3\sum_{i=1}^{q-1}(p+i)(q-i), \qquad (2)$$

where c_1, and c_2, and c_3 are positive constants which reflect the time required by the various operations of each step. The first term represents the time required to build the initial list of minimum cost points, the middle term is the time to find the next point to be inserted for all q points on the inner hull, and the last term is the worst-case time to update and maintain the list of minimum cost points as the inner hull is being stripped.

How much time will it take to merge and strip all the rings? This question cannot be answered without making some simplifying assumptions about the number of rings, the average number of points on each ring, and the average time for updating the minimum cost list. Therefore, we proceed with an analysis of the algorithm under the special conditions of n/k rings, k points on every ring (k divides n evenly), and worst-case update time for the minimum cost points list. Following from formula (2) the total time to insert and strip all inner ring points is given by the following sum

$$\sum_{j=1}^{n/k-1} [c_1jk^2 + c_2\sum_{i=1}^{k-1} i + c_3\sum_{i=1}^{k-1}(jk+i)(k-i)]$$

$$= c_1[k^2((n/k)-1)(n/k)/2] + c_2[k(k+1)((n/k)-1)/2]$$

$$+ c_3[\ [((n/k)-1)(n/k)(k-1)k^2/2] + [((n/k)-1)(k-1)k^2/2]$$

$$- [((n/k)-1)(n/k)(k-1)k^2/4] - [((n/k)-1)(k-1)(k)(2k-1)/6]\ ]$$

$\leq O(n^3)$

In most situations we can expect the actual time to be somewhat smaller since it is highly improbable that inserting and stripping a point will change every one of the minimum cost points already stored in the list.

The maximum time the procedure can take will occur if there are only two rings with $p = 3$ points on the outer ring and $q = (n-3)$ points on the inner ring. Since no points will be excluded from any of the insertion and minimum cost list computations, the maximum number of points will have to be considered at each iteration for merging into the outer ring. Inserting $p = 3$ and $q = (n-3)$ into formula (1) yields

$$c_1[3n-9] + c_2[(n-3)(n-2)/2] + c_3[\ [3n(n-4)] - [9(n-4)]$$

$$- [6(n-4)(n-3)/2] + [n(n-4)(n-3)/2] - [(n-4)(n-3)(2n-7)/6]\ ]$$

$$= O(n^3).$$

6.5. *Dynamic Hull Heuristic*

Another approach that seems worthy of investigation is to first form the outer hull and let this structure be the initial tour. Next, the hull of all the remaining points is formed and the point on this ring which minimizes cost rule R2 is inserted into the outer tour. The procedure generalizes as follows: always maintain one convex hull which encloses all points interior to the present subtour and from this hull select and insert into the subtour the point k^* which minimizes cost rule R2.

The motivation behind the algorithm stems from the simple observation that whenever a point k^* is inserted from the inner convex hull into the outer enclosing subtour it may be that some of its interior points will be outside of the convex region formed by the inner hull minus k^*; Fig. 6.2. Since one of these points might be chosen next (if we were considering all points inside of the enclosing tour for possible insertion) it is highly probable that this point would be on the hull of all interior points. We give a synopsis of the procedure below.

```
Step 1.  T := CH(S);
Step 2.  S := S - T;
Step 3.  If |S| = 0 then HALT;
Step 4.  H := CH(S);
Step 5.  Find the $k^*$ in H and $(i^*,j^*)$ in T which minimize cost
           rule R2;
Step 6.  T := T + $k^*$;
Step 7.  S := S - $k^*$
Step 8.  go to step 3.
```

The implementation of this procedure is rather straightforward except for step 5 where we keep track of any point that was on a previously computed hull. For each of these points the minimum cost distance to insert into the outer tour is maintained as in the cheapest insertion hull heuristic. Whenever a new hull is computed those points which were not in the previous hull are dynamically added to the minimum cost list along with the two points (i,j) on the tour which

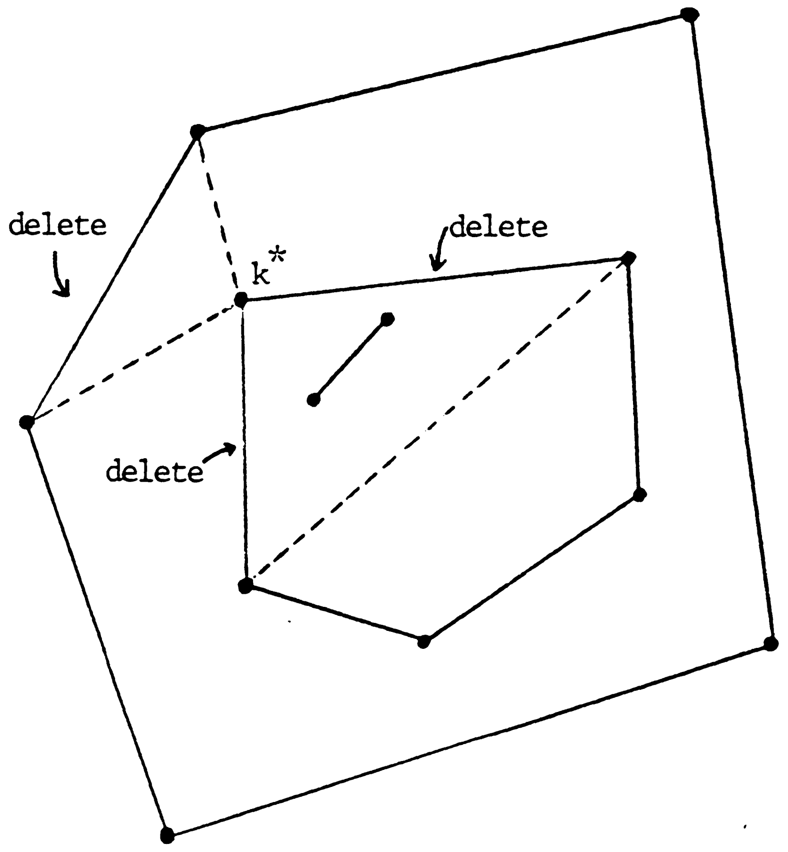


Fig. 6.2. The convex region formed by the insertion of k^* does not include all points on the inner hull peels.

minimize the insertion cost according to rule R2.

The worst-case time taken by the various steps of the algorithm through completion is as follows:

Time of step 1: O(nlogn);

Time of step 2: O(n);

Time of step 3: O(n);

Time of step 4: $O(n^2 \log n)$;

Time of step 5: $O(n^3)$;

Time of step 6: O(n);

Time of step 7: O(n);

Time of step 8: O(n).

In steps 1 and 4 we assume that the Graham convex hull algorithm is used. The Eddy algorithm has $O(n^2)$ worst-case behavior versus the O(nlogn) worst-case time of the Graham algorithm, and therefore in this procedure the Graham algorithm is preferred. Another possibility in step 4 is to use an *on-line* convex hull algorithm [Preparata (79)] to update the present hull after each insertion of k^*. (On-line algorithms do not operate on all the data collectively, instead a structure is always maintained for the data received up to some point in time. When a new data item is encountered the structure is updated to reflect the change.) However, since all points interior to the inner convex hull must be considered for update (to be placed into the next hull), the asymptotic running time would be the same as if the Graham algorithm were used after each insertion of k^*.

6.6. AI Hull Heuristic

The last method we propose is based upon maintaining a small list of candidate points, NT, which may be inserted into the present subtour T. With high probability, one of the points in NT should minimize cost rule R2 amongst all points not yet included in T. The size of NT will, at least in the initial stages of the algorithm, be much smaller than the total number of all points enclosed by the subtour.

As in the other three methods, the initial subtour T is the convex hull of the set of points. Next, we build the list NT that will contain a selected set of points that may be inserted into T. These points will be from all convex hull peels with a depth of k or smaller (excluding of course the outermost hull peel). Points will be inserted one at a time from NT into T according to cost rule R2. After each of these insertions, if a point has been inserted into T with a depth of d, and d is greater than the maximum depth of all points in T, then NT is updated to contain all points with a depth of (d + k) or smaller. k is a threshold value that represents a small number of convex hull peels (say 2 or 3), and can be set by the user for any particular run.

Besides maintaining the lists T and NT as in the heuristics of Sections 6.4 and 6.5, an additional list, NH (Not on Hulls), must also be maintained. The points in this list will always be encircled by the innermost hull included in NT. Whenever NT is updated to contain a new hull (or hulls), then NH is the input to a convex hull finder (the Eddy algorithm) which computes the convex hull of NH. This structure is then added to NT and subsequently subtracted from NH. This process is repeated until the necessary number of hull peels have been added to NT. The appropriate data structure for T is a linked-list, and arrays are proper choices for NT and NH.

The complexity analysis is very similar to the cheapest insertion hull heuristic; worst-case running time will be $O(n^3)$, and expected-case complexity is bounded by $O(n^2)$ and $O(n^3)$. The actual asymptotic running time will be slightly greater than $O(n^2)$, since the probability of a worst-case situation is extremely small.

6.7. The Stewart Hull Heuristic

As stated earlier, Stewart proposed an algorithm for finding the ETSP tour using as the initial subtour the convex hull of the set of points. After this, all points not yet in the subtour are then considered for possible insertion according to cost rule R. The cheapest of these points is then inserted and the process repeated until all points are in the tour. By using additional inner convex hull peels our algorithms exclude those points which have little chance of being inserted next. Stewart's algorithm on the other hand is more of a brute-force method since all points inside each subtour are considered for possible insertion. A psuedo-code version of the general procedure follows:

```
*Step 1.* T = CH(S);
```

```
Step 2.  S := S - T;
Step 3:  If |S| = 0 Then HALT;
Step 4.  Find $k^*$ in S and $(i^*,j^*)$ in T which minimize cost rule
    R2.
Step 5.  T := T + $k^*$;
Step 6.  S := S - $k^*$;
Step 7.  go to step 3.
```

The analysis of Stewart's algorithm is very similar to the cheapest insertion hull heuristic and dynamic hull heuristic (Sections 6.4 and 6.5). Depending upon the set of inputs, the convex hull heuristic takes anywhere from O(nlogn) to $O(n^3)$ time to compute the final tour. When all the points are on the hull or only a few points ($\leq$ logn) are left inside the time complexity will be O(nlogn). However, this is an extremely pathological circumstance, since for most distributions O(n) points will be left inside the hull. This means that the time complexity of the procedure is bounded below and above by $O(n^2)$ and $O(n^3)$. Golden [Golden (80)] states that in practice the algorithm seems to require about $O(n^2 \log n)$ computations. However, he gives no analysis or empirical evidence to justify this result.

Computing the convex hull should be carried out using either the Graham or Akl-Toussaint convex hull algorithms (see Chapter Four). This will insure that if a pathological case does occur (where all the points are on the hull) then the worst-case time complexity will be O(nlogn).

As in the cheapest insertion hull heuristic and dynamic hull heuristic the insertion phase can be carried out by maintaining two lists, one representing the tour T, and the other, NT, which contains all points not yet included in the tour. For each point k in NT, a record is kept of the points (i,j) in T for which

$$[d_2(i,k) + d_2(k,j)]/d_2(i,j)$$

is minimal. In this way, determining the next point to be added to the tour and deciding its relative position in the tour can be carried out in time proportional to one pass through NT. After each insertion, updating NT can be carried out in at most $O(|NT|^2)$ time, but usually much closer to $O(|NT|)$ time. (The fact that updating NT could take as much as $O(|NT|^2)$ time leads to the $O(n^3)$ worst-case behavior of Stewart's algorithm.)

6.8. Nearest Neighbor

The nearest neighbor approximation technique [Bellmore (68)], [Gavett (65)] is an attractive method because of its simplicity. A tour is constructed as follows. First, an arbitrary point is chosen to anchor the tour. Second, find the point which is closest to the point last added and add this point to the tour. Third, repeat the second step until all points have been added to the tour. Fourth, add an edge between the first and last points in the tour.

Note that this simple method is not restricted to just the ETSP, it can just as easily be used to obtain a TSP tour. The method seems to be a "natural" one because when humans are asked to find a ETSP tour one of the criteria they most often invoke is to use those points which were included last into the tour.

Nearest neighbor has been shown in empirical tests to produce tours which are almost always less than 1.5 times the length of the optimal tour [Golden (80)]. Because nearest neighbor is also easily implemented it serves as a good preliminary screening procedure by which any new method may be judged.

A straightforward implementation of the algorithm involves maintaining two lists T and NT in array data structures. T is the tour list, and NT contains all those points not yet included in the tour. Initially T contains one point and NT (n-1) points. As the algorithm progresses points are added to T and deleted from NT. Each insertion and deletion can be done in constant time and therefore the algorithm spends all of its time repeatedly scanning NT. That is, to add the second point to T requires (n-1) distance computations, to add the third point requires (n-2) distance computations, and so on. Hence, the total time required by the procedure is always proportional to the sum

$$\sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = [(n-1)n]/2 = O(n^2).$$

6.9. Nearest and Farthest Insertion

In this Section we examine two well-known approximation methods. These techniques, nearest insertion and farthest insertion, are related to nearest neighbor in that they attempt to add to an existing subtour a point which is closest to or farthest from one of the points already included in the subtour. Rosenkrantz, Lewis, and Stearns [Rosenkrantz (77)] have derived several interesting theoretical results concerning the length of the tours produced by insertion methods. The most important is that nearest insertion always produces a tour that is no longer than twice the length of the optimal tour. The nearest insertion algorithm follows in which we assume that initially the subtour T is empty and S contains the set of points for which the tour will be determined.

Step 1. Start with an arbitrary point i in S and make this point the initial subtour (T := T + i) and delete i from S (S := S - i).

Step 2. If the cardnality of S is equivalent to zero then HALT, T contains the final subtour.

Step 3. Find a point k^* in S such that $d_2(i,k^*)$ is minimal. Add k^* to T (T := T + k^*) and delete k^* from S (S := S - k^*).

Step 4. If the cardnality of S is equivalent to zero then HALT, T contains the final subtour.

Step 5. Find the point k^* in S which is nearest to any of the points in the subtour T.

Step 6. Given k^*, find the edge (i^*,j^*) in T such that cost rule R1 is minimal. That is, find the place in T where k^* can be inserted at minimum cost.

Step 7. Obtain a new subtour by replacing edge (i^*,j^*) in T with edges (i^*,k^*) and (k^*,j^*) (i.e., $T := T + k^*$). Delete k^* from S ($S := S - k^*$).

Step 8. Go to step 4.

The farthest insertion algorithm is the same as nearest insertion except that in step 3 the word "minimal" should be replaced by "maximal," and in step 5 the word "nearest" should be replaced by "farthest."

The intuitive appeal of farthest insertion is that it establishes a general outline of the tour at the outset and then attempts to fill in the details later. Nearby points inserted at the end of the procedure will result in short edges that are less likely to be deleted by some still later insertion. Surprisingly, Rosenkrantz, et.al. [Rosenkrantz (77)] in a series of experiments, found that farthest insertion usually produced better tours than nearest insertion and nearest neighbor. On problems which involved placing 50 points randomly in a unit square, nearest insertion was from 7 to 22 percent worse than farthest insertion, and nearest neighbor was from 0 to 38 percent worse. Thus the ranking was usually farthest insertion best, nearest insertion second, and nearest neighbor last.

Golden, et.al. [Golden (80)] also conducted a set of tests which showed that farthest insertion was a very good performer relative to several other heuristics including nearest neighbor and nearest insertion. Our tests given in the next Section serve to verify these results.

Implementation of both nearest insertion and farthest insertion is similar to several of the convex hull heuristics we have discussed. The idea is again to maintain one list T for those points on the tour, and another list NT for those points not yet included in T. For each point in NT a pointer is kept to the point in T to which it is nearest (for nearest insertion) or farthest (for farthest insertion). Adding a new point into T from NT will require two passes through NT and one pass through T. Because these lists always contain a total of n items, and since there are (n-1) points to insert, the time complexity of the algorithm will be $O(n^2)$.

6.10. Test Results

The eight heuristics, random hull peel heuristic, cheapest insertion hull heuristic, dynamic hull heuristic, AI hull heuristic, Stewart's hull heuristic, nearest neighbor, nearest insertion, and farthest insertion were coded in PASCAL and tested on a VAX 11/780. Uniform random variates were generated inside a unit square for sample sizes of 25, 50, 100, 200, and 400 points. Each of the tours produced was compared to a theoretical lower bound B for tour length based upon Monte Carlo experimentation and total edge length of the minimum spanning tree of the set as reported in [Akl (78a)]:

B = 1.102 * length of minimum spanning tree.

The length of the actual tour was then divided by this quantity to yield a relative performance efficiency measure for quality tours produced by each algorithm:

efficiency = (actual tour length) / B.

50 runs were made for sample sizes of 25, 50, and 100 points; 25 runs were made for 200 and 400 points. The averaged results which appear

*Table 6.2. Average tour lengths of several ETSP heuristics relative to B = 1.102 * length of MST; standard deviations in parentheses.*

n	random hull	cheapest hull	dynamic hull	AI hull
25	1.147(.043)	1.139(.045)	1.133(.044)	1.133(.044)
50	1.143(.041)	1.132(.042)	1.122(.035)	1.111(.039)
100	1.173(.034)	1.161(.034)	1.132(.029)	1.103(.026)
200	1.196(.029)	1.188(.023)	1.162(.018)	1.107(.019)
400	1.237(.013)	1.219(.018)	1.196(.018)	1.105(.015)

n	n. neighbor	far insert	near insert	Stewart
25	1.338(.118)	1.145(.045)	1.278(.075)	1.133(.044)
50	1.300(.093)	1.152(.052)	1.268(.046)	1.111(.039)
100	1.291(.054)	1.168(.034)	1.270(.032)	1.103(.026)
200	1.299(.060)	1.190(.026)	1.259(.019)	1.107(.019)
400	1.277(.039)	1.227(.020)	1.266(.016)	1.105(.015)

in Table 6.2 include sample standard deviations in parenthesis.

Our tests reconfirm (see [Golden (80)]) that the Stewart convex hull heuristic performs remarkably well. The tests also indicate that the AI hull heuristic produces exactly the same results when the depth of the convex hull peels of all points in the NT list is at most (d + 3) where d is the greatest depth of any point included in a subtour (see Section 6.6). This is not surprising since the probability is extremely high that the next point to be inserted into the AI hull heuristic subtour will be identical to the point which will be inserted next into the Stewart convex hull heuristic subtour. Clearly, both the AI hull heuristic and the Stewart convex hull heuristic are superior to all others tested on uniform distributions of points in the plane.

Which of these methods runs faster? In choosing one over the other several factors must be considered. Obviously, the overhead required to compute the convex hull peels must be balanced by the growth of NT in the AI hull heuristic. That is, if NT does not grow too fast from the insertion of points in NH, then the AI hull heuristic would be preferred since the majority of the computation in both algorithms is directly proportional to the size of NT. Unfortunately, we found that NT grows quite rapidly in the AI hull heuristic for all sample sizes, even 400 points. Therefore, the Stewart convex hull heuristic requires fewer total operations and thus operates more efficiently than the AI hull heuristic.

For all sample sizes cheapest hull and dynamic hull performed reasonably well with efficiencies exceeding those of farthest insertion, nearest insertion, and nearest neighbor in all cases. The performance of the random hull heuristic was very similar to that of farthest insertion. Nearest neighbor and nearest insertion seemed to improve slightly for large n, while the performance of farthest insertion, random hull heuristic, cheapest hull heuristic, and dynamic hull heuristic degraded. This degradation, which requires further study, could be caused by the general modus operandi of these methods, which is to form a general outline of the tour in the initial stages and then to include the remaining points by using an insertion scheme. Nearest neighbor and nearest insertion avoid trying to form a general outline early on, and this appears to be advantageous for large n.

In another test we compared the various approximation algorithms on five 100 node problems presented first by Krolak, Felts, and Marble in [Krolak (71)]. The results which appear in Table 6.3 were consistent

Table 6.3. Tour lengths for 100 node problems given in [Krolak (71)].

	problem number				
method	24	25	26	27	28
Best known	21282	22148	20749	21294	22068
Random hull	23750	23996	22844	21998	24808
Cheap hull	23659	23757	21869	21910	24703
Dynamic hull	23205	23718	21320	22319	22901
Stewart	22056	22700	21275	21794	22830
AI hull	22056	22700	21275	21794	22830
N. Neighbor	26856	29155	26327	26950	27587
N. Insertion	26145	27412	26080	25172	26674
F. Insertion	24523	24768	22496	23202	24704

with those involving the uniform distributions of points in the plane. The best know results were obtained from [Golden (80)].

Again, the AI hull heuristic and Stewart's method produced equivalent results and best of all the methods tested. Nearest neighbor, farthest insertion, and nearest insertion performed rather poorly in almost every case. Of the other hull heuristics, the random hull peel heuristic and cheapest insertion hull heuristic performed admirably on problem 27. The dynamic hull heuristic also performed quite well on problems 26 and 28.

Carrying out tests is a very expensive process and thus we were somewhat restricted by the size of the problems attempted. Future research should concentrate on both increased size and different distributions. For example, if the points are uniformly spread throughout an annular ring, one would expect good performance from all the convex hull heuristics. On the other hand, consider a standard normal distribution inside a circular region. The points would be densest in the center, thereby causing some doubt as to how the hull heuristics might perform.

6.11. Improvements for the Stewart and AI Hull Heuristics

We discovered in the last Section that both the AI hull heuristic and Stewart's convex hull heuristic were the best methods for obtaining high quality Euclidean traveling salesman tours. While we were able to argue in Sections 6.6 and 6.7 that in the expected sense these heuristics will exhibit run time performance slightly greater than $O(n^2)$, the worst-case of $O(n^3)$ might be too high a premium to pay to insure a close-to optimal tour. Thus, we submit the following idea which improves worst-case behavior to $O(n^2)$ but maintains the integrity of the tours produced by both approximation schemes.

Recall that worst-case performance was caused by replacing edge (i^*,j^*) by edges (i^*,k^*) and (k^*,j^*) in T. Because each point's minimum edge in NT could be (i^*,j^*) the time to update NT would require a complete scan of T for each point in NT. Consider Figure 6.3, which indicates that if a point $k \neq k^*$ in NT has minimum edge (i^*,j^*) then its new minimum edge with high probability will be either (i^*,k^*) or (k^*,j^*). The only case in which this situation cannot occur is if a sequence of insertions causes the edges of T to form a non-simple polygon. In this case, if (i^*,j^*) is a crossing edge it may be that one of (i^*,k^*) or (k^*,j^*) is not the replacement edge corresponding to a point whose previous minimum edge was (i^*,j^*).

Visual observation of a number of tours constructed for various size n indicated that the probability of T containing an intersection is below 0.05 for uniformly distributed points in the unit square. Furthermore, the number of mutually intersecting edges was always very small, usually two, three, four, or five. Because of these observations, the AI hull heuristic and Stewart's heuristic were modified such that the previous minimum edge (i^*,j^*) was always replaced by either (i^*,k^*) or (k^*,j^*) depending on which one minimizes insertion cost.

The performance of both modified heuristics; Table 6.4, was virtually identical to the results found in Table 6.2. (The input data to both the unmodified and modified routines was identical).

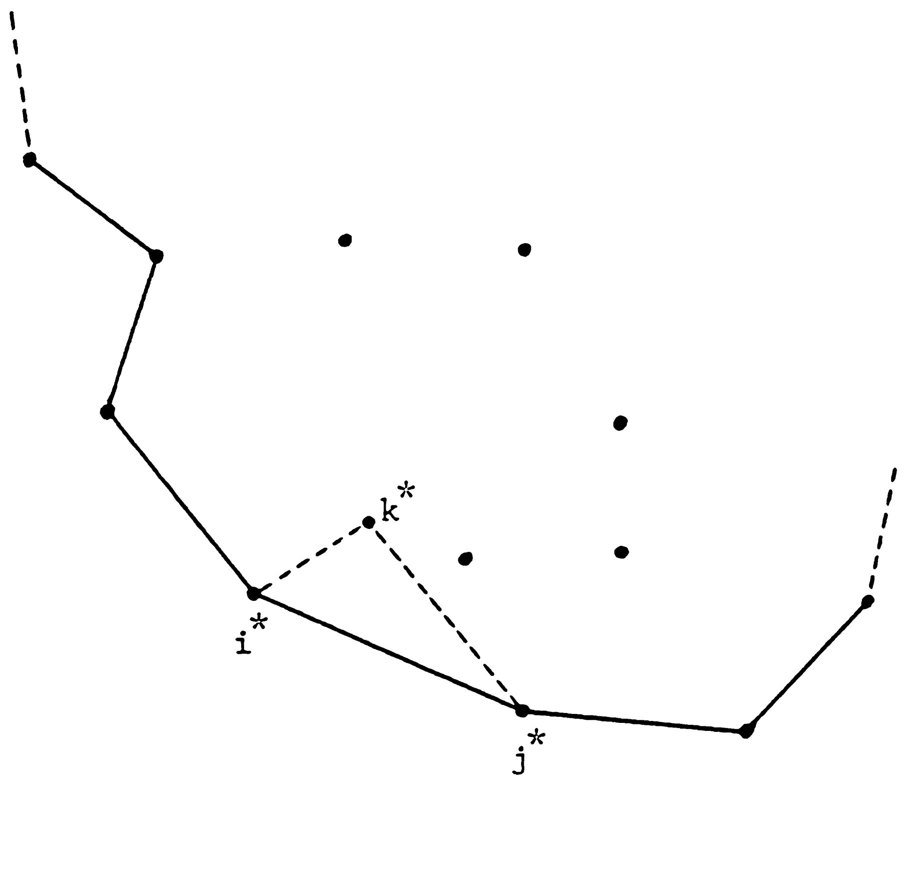


Fig. 6.3. Any point whose minimum edge is (i^*,j^*) is replaced by either (i^*,k^*) or (k^*,j^*) (with high probability).

Table 6.4. Quality of tours for the modified AI hull heuristic and the modified Stewart convex hull heuristic.

n	AI hull heuristic	Stewart's heuristic
25	1.133(.044)	1.133(.044)
50	1.112(.039)	1.112(.039)
100	1.101(.025)	1.101(.025)
200	1.107(.019)	1.107(.019)
400	1.105(.015)	1.105(.019)

Additionally, we again tested all five 100 city problems as discussed in Section 6.10 and recorded results exactly identical to those found in Table 6.3. We can conclude that this scheme works very well for both the AI hull heuristic and Stewart's heuristic. For any other type of cheapest insertion TSP heuristic, we conjecture that a similar modification would give equally satisfying results where the cost between cities is a metric.

6.12. Algorithms for the L_1 Traveling Salesman Problem

One problem that has received little attention in the literature is the counterpart of the ETSP, the L_1 traveling salesman problem (L_1TSP). Here, the intercity distance between any two points (i,j) in S is given by

$$d_1(i,j) = |x_i - x_j| + |y_i - y_j|.$$

Since intercity distance is a metric, the triangle equality and symmetry hold as discussed in Section 6.2. Also, the L_1TSP is NP-hard [Garey

(76)].

The results in Section 6.10 indicate that the AI hull heuristic and the convex hull heuristic of Stewart were the best methods amongst all we examined. Therefore, it is natural to wonder if these procedures can be adapted to the L_1TSP. Recall that in Chapter Four we showed that there is an analog structure in L_1 to the convex hull in L_2 which we appropriately named the L_1 hull. Thus, the only modifications we make to both heuristics is to use the L_1 hull in place of the convex hull, and whenever the distance function is d_2 replace it with d_1.

L_1 versions of the AI hull heuristic and Stewart's hull heuristic were coded in PASCAL and run on a VAX 11/780. Nearest insertion and farthest insertion were also included in the test to determine how the hull heuristics perform against other well known methods. A random number generator was used to generate points uniformly throughout the unit square for several different sample sizes. To measure the quality of each algorithm, a quantity called efficiency was computed as

$$\text{efficiency} = (\text{length of tour})/B,$$

where B = 1.102 * (length of the minimum spanning tree). As explained in Section 6.10, Akl [Akl (78a)] derived this lower bound to benchmark the performance of several ETSP heuristics. It turns out that this bound is also viable in the L_1 metric [Gilbert (65)], [Beardwood (59)]. 50 runs were made for sample sizes of 25, 50, and

100 points; 25 runs were made for 200 and 400 points. The averaged results which appear in Table 6.5, indicate that both hull heuristics produce better quality tours than either nearest or farthest insertion. Also, as we found in the Euclidean metric, both routines perform with equivalent accuracy. An explanation for this behavior has already been given in Section 6.10.

The efficiency relative to B did not prove to be as good in the L_1 metric as in the L_2 metric for all four methods. One possible explanation is that the constant 1.102 was derived by Monte Carlo techniques carried out in the Euclidean metric. In [Gilbert (65)] the author states that this constant should work equally satisfactorily in L_1 as in L_2. However, Tables given there indicate that the constant varies by several percentage points for small n.

We recommend the L_1 AI hull heuristic or the L_1 Stewart hull heuristic for problems of size $n < 500$ points. If a better tour is desired, a composite procedure should be applied. One such procedure would be to first apply either of the L_1 hull heuristics followed by the branch exchange heuristic devised by Lin [Lin (65)].

6.13. Hybrid Convex Hull Heuristics for the ETSP

In our search for a good ETSP approximation algorithm we have restricted our investigation to insertion methods which use only the set of convex hull peels. Future research remains which seems promising

*Table 6.5 - Average tour lengths for several L1TSP heuristics relative to B = 1.102 * length of MST; standard deviations in parentheses.*

n	nearest insertion	farthest insertion	Stewart's heuristic	AI hull heuristic
25	1.248 (.081)	1.238 (.069)	1.168 (.056)	1.168(.056)
50	1.184 (.050)	1.200 (.054)	1.132 (.041)	1.132(.041)
100	1.186 (.036)	1.192 (.042)	1.114 (.026)	1.114(.026)
200	1.178 (.030)	1.181 (.027)	1.119 (.021)	1.119(.021)
400	1.177 (.018)	1.181 (.014)	1.128 (.014)	1.128(.014)

involving the combination of the convex hull with certain graph-theoretic structures. For example, consider the following approximation technique for computing the ETSP tour. Let the convex hull be the initial subtour T. Next, compute a planar graph G for which the vertices of G are the set S and there is a path from vertex i to vertex j for all i ≠ j in G. The third step is to judiciously use the edges of G to merge the vertices of (G - T) into T thereby obtaining a tour of the original set of points.

For step two there are at least four graph structures which may be applied. These are (1) the *Delaunay Triangulation* (DT), (2) the *Gabriel Graph* (GG), (3) the *Relative Neighborhood Graph* (RNG), and (4) the *Minimum Spanning Tree* (MST).

The Delaunay triangulation is a planar graph which triangulates the set of points in S. It is the dual of the Voronoi diagram which was previously defined in Section 1.2.2. Two points i and j are joined by an edge if, and only if, their corresponding tiles share a side, *i.e.,* if i and j are *Voronoi Neighbors.* If this operation is carried out on Fig.

1.2 one obtains the DT of Figure 6.4. One way to compute the DT is to first compute the Voronoi diagram in O(nlogn) time. Once this structure is obtained, the DT can be computed in O(n) time [Toussaint (80a)]. Lee and Schacter [Lee (79b)] have given two algorithms that compute the Delaunay triangulation directly, bypassing the Voronoi diagram step. One is a divide and conquer approach which runs in O(nlogn) time whereas the other is iterative and runs in $O(n^2)$ worst-case time.

The DT may be used to compute the ETSP by taking each edge in T and replacing it by two edges in the DT as follows: For each pair of consecutive points (i,j) in T, find the point k in the DT which is mutually adjacent to (i,j) in the DT. From all triples (i,j,k) take the one, (i^*,j^*,k^*), which minimizes

$$[d_2(i,k) + d_2(k,j)]/d_2(i,j).$$

k^* would then be the next point added to T.

The Gabriel graph of a set of points is formed by joining an edge between any two points which are *Gabriel neighbors.* Two points i and j are neighbors if the circle which passes through both i and j does not contain any other points in S; Fig. 6.5. This definition leads straighforwardly to an $O(n^3)$ algorithm for constructing the GG. Matula and Sokal [Matula (80)] have suggested an O(nlogn) algorithm for computing the GG by first computing the Voronoi diagram.

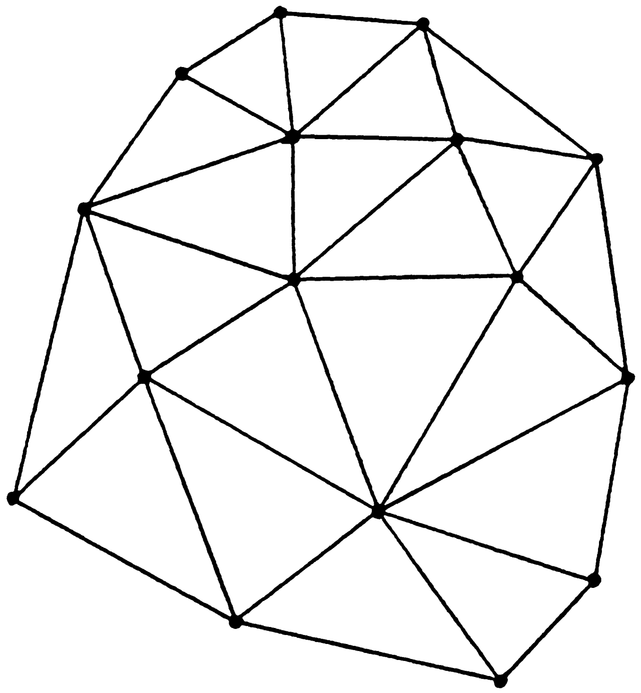


Fig. 6.4. The Delaunay triangulation.

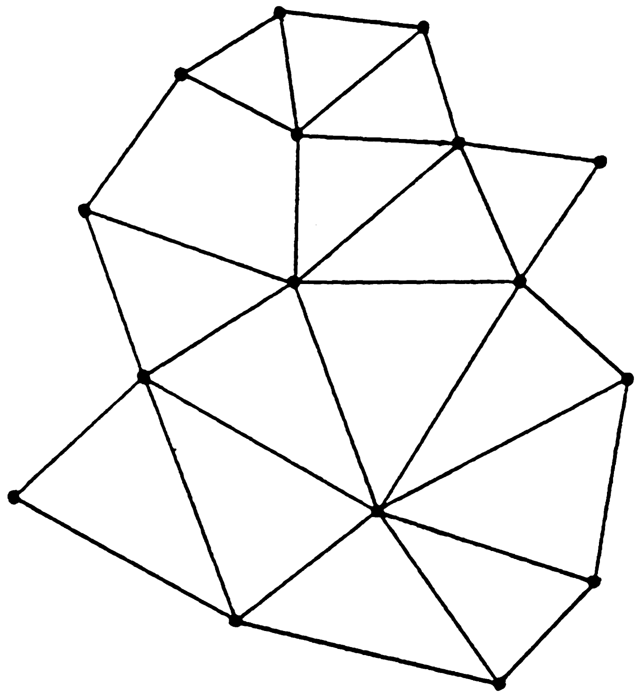


Fig. 6.5. The Gabriel graph.

Another possible way of defining whether two points i and j are neighbors leads to a structure called the Relative Neighborhood Graph (RNG). The definition of the RNG states that two points i and j are joined by an edge if no other points lie inside the intersection of two circles each with radius equivalent to the distance between i and j; Fig. 6.6. The definition suggests a straightforward $O(n^3)$ algorithm for computing the RNG. Toussaint [Toussaint (80b)] has given two other algorithms for determination of the RNG which run in $O(n^2)$ worst-case time. Both are based upon first computing the Delaunay triangulation.

A *tree* is a graph that contains no cycles. A *spanning tree* of a graph G, is a subgraph of G that contains every vertex of G. Given a complete weighted graph G = (S,E), where the edges E represent distance amongst all points in S, the minimum spanning tree is a tree of total minimum edge weight. A MST can be found by choosing the smallest unused edge that does not form a cycle with the edges already chosen and continuing until (|S| - 1) edges have been selected to form the MST. Another scheme devised by Shamos and Hoey [Shamos (75b)] takes O(nlogn) time and involves first computing the Voronoi diagram.

The reader might be wondering whether there is any relationship amongst the four graphs we have discussed. It turns out that the

$$\mathrm{MST} \subset \mathrm{RNG} \subset \mathrm{GG} \subset \mathrm{DT}.$$

This should not be too surprising since the DT can be used to compute all four graph structures. Consult Toussaint [Toussaint (80a)] for

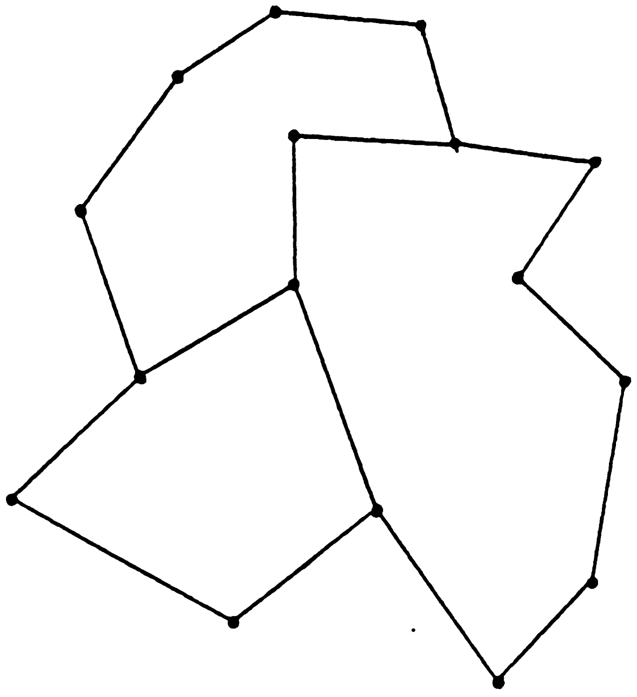


Fig. 6.6. The relative neighborhood graph.

some of the details and further references concerning this relationship.

How would these structures be used to obtain good ETSP tours? Let G be either the MST, RNG, or GG. One procedure would be to consider inserting into T only those points in (G - T) which have an edge in G that joins T. The point to be added first would be the one which minimizes an insertion cost/ratio rule similar to R2.

The four procedures we have outlined will hopefully produce tours of approximately the same quality as the AI hull heuristic and Stewart's convex hull heuristic. However, the time to compute these tours may require less than $O(n^2)$ time because the Voronoi diagram can be computed in O(nlogn) time. In the case of the convex hull/Delaunay triangulation hybrid, an O(nlogn) algorithm is possible by presorting the list of initial insertion points and then updating this list using a priority queue. Since the number of points in the insertion list will never be greater than n, the sort step will require at most O(nlogn) time, and priority queue update will require at most O(logn) time after each insertion. We envision that these methods would be applicable to the L_1TSP as well as the ETSP.

Chapter 7
THE SUPERRANGE OF STAR-SHAPED AND MONOTONE POLYGONS

7.1. Introduction

There are many other interesting sub-areas of research in computational geometry as we have already seen in chaper 1. So far we have been content in this thesis to examine problems which require convex hull determination (or L_1 hull determination) as part of their solution. In this Chapter we deviate from this pattern to look at some problems that require a somewhat different approach. These problems have been loosely termed *the visibility problems.*

In many application areas the idea of discovering algorithms which will allow the computer to see, in at least some limited sense, is of paramount importance. For example, in Robotics, current research is focusing upon collision avoidance amongst several robot arms working in the same definitional space [Roach (83)]. Naturally, these types of problems lead to the following question: what part of a polygon or polyhedra can an observer see from a specific vantage point? Possibly, the most famous problem whose solution attempts to answer this question from the viewpoint of an observer looking at a set (or scene) of three dimensional artifacts (objects) is the *hidden-line* problem in Computer Graphics.

We will be concerned with visibility among the vertices of polygons, specifically those vertices of a polygon that can be seen from a particular vertex. Let P be a simple polygon in the plane with vertices $(v_0, \ldots, v_{n-1})$ counterclockwise on its boundary. The *superrange* of a vertex $v_i \in P$ is the set of vertices, $S(P,v_i)$, that can be seen from v_i, *i.e.,* those vertices $v_k \in P$ such that all points on the line segment from v_i to v_k lie entirely within (or on) the boundary of P. Our work concerns the introduction of two linear time algorithms for the determination of the superrange of polygons which have the special properties of star-shapedness and monotonicity.

7.2. Previous Research

The definition of vertex superrange can be attributed to Chazelle [Chazelle (80)] who introduced the notion with the intent of formulating an algorithm that would decompose a simple polygon into a minimal number of convex parts. Shamos [Shamos (77)] had earlier defined the *viewability graph* of a polygon as a structure whose nodes are connected by an edge if and only if the associated vertices are visible. Haralick and Shapiro [Haralick (77)] found application of this structure in shape decomposition. Shamos suggested an O(n) algorithm to obtain $S(P,v_i)$, $i = 0, \ldots, n-1$ for each vertex. This would give an $O(n^2)$ algorithm for viewability graph determination. Unfortunately the algorithm is known to fail for certain polygons [El-Gindy (81)].

A structure that is more general than $S(P,v_i)$ is the *visibility polygon* of P. Given a simple polygon P and a point $x \in P$, the visibility polygon of P with respect to x, denoted by V.(P,x), is that subset of P such that for any point y in P, x and y are mutually visible. Toussaint [Toussaint (81)] states that the visibility polygon algorithm of El-Gindy and Avis [El-Gindy (81)] can be adapted to compute S(P,v) in O(n) time and thus the viewability graph can be computed in $O(n^2)$ time.

Several other problems related to superrange determination have been considered by Avis and Toussaint [Avis (81b)]. These problems have been termed "jail-house" problems because their solutions can be used to answer questions about what portions of a polygonal region a patrolling guard can see under the restriction of movement along one edge of the polygon. In [Avis (81b)] three definitions of visibility from an edge are introduced. (i) P is said to be *completely visible* from an edge uv if for every $z \in P$ and every $w \in uv$, w and z are visible. (ii) P is said to be *strongly visible* from an edge uv if there exists a $w \in uv$ such that for every $z \in P$, z and w are visible. (iii) P is said to be *weakly visible* from an edge uv if for each $z \in P$, there exists a $w \in uv$ (depending on z) such that z and w are visible.

Now, consider a guard whose job is to observe the polygon from edge uv. If p is completely visible from uv, the guard can be positioned at any location on uv. If P is strongly visible from uv, then there always exist at least one point on uv where the guard can see all of P. Finally, with weak visibility, the guard will be forced to walk along some section of uv to observe P. Avis and Toussaint have given

O(n) algorithms to solve the patrolling visibility problem under any of the definitions given above.

7.3. Special Case Algorithms

As mentioned previously, the algorithm of El-Gindy and Avis for determining the visibility polygon from a point can be adapted to solve the superrange problem for any simple polygon in linear time. This algorithm is thus optimal in the sense that no algorithm can exhibit a tighter time complexity bound. However, the algorithm is quite complicated in that several stacks must be maintained to keep track of hidden regions generated during a sequential scan of the polygon's vertices.

It has been known for some time that there are polygons with special properties that will allow certain computational problems to be solved in a very straightforward manner. As a general rule, algorithms for these polygons are usually much simpler, easier to implement, and run on the average several times faster than their general case counterparts.

The classes we have alluded to are the convex, star-shaped, and monotone polygons. The algorithm for the superrange of a vertex on a convex polygon is trivial since all of P is visible from any vertex. In the remainder of the Chapter we turn our attention to the latter two types.

7.4. Definitions

A polygon P is said to be star-shaped if there exists at least one point $z \in P$ such that for all $p \in P$, the line segment zp lies entirely within P. Informally, we say that a polygon is star-shaped if there exists a point in P that can "see" all of the other points. From a computational standpoint this definition is not very practical since it implies that every point in P must be examined to determine if P is star-shaped.

Consider the following theorem due to Penny [Penny (72)]: A polygon is star-shaped if and only if there exists some point $z \in P$ such that for all $v_i \in P$, zv_i lies entirely within P. The theorem implies that star-shapedness depends only on the vertices of P, and makes the problem of determining whether a polygon is star-shaped tractable. It is an immediate consequence of this theorem that the sequence of vertices about z (angular ordering of vertices about z) is identical to the sequence of vertices that defines P. And, it is precisely this property that makes determining the superrange of star-shaped polygons simpler than any general-case algorithm.

An important question is how to compute the locus of points with respect to which P is star-shaped, *i.e.,* the kernel of the polygon K(P); Fig. 7.1. It has been shown [Shamos (78)] that K(P) is itself a convex polygon having no more vertices than P. An O(n) algorithm to determine the kernel of a polygon has been given by Lee and Preparata [Lee (79a)]. The basic idea involves using the intersection of

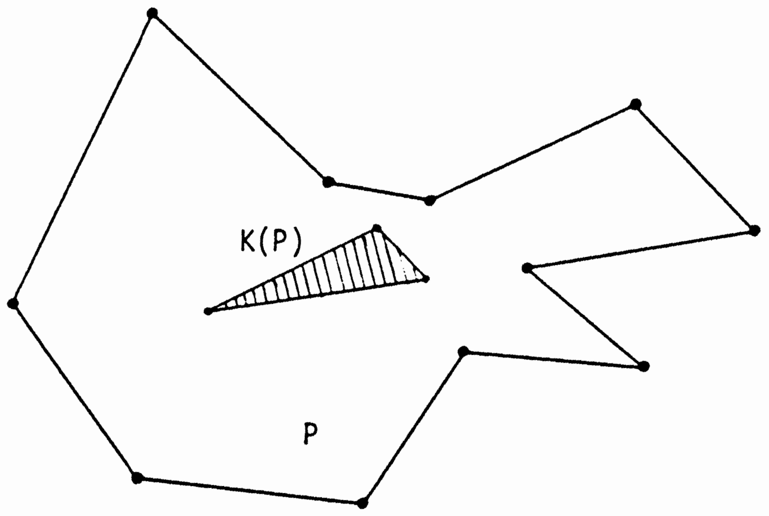


Fig. 7.1. A star-shaped polygon and its kernel.

appropriate half-planes defined by the ordered edges of the polygon.

A chain $C_{i,j} = (v_i, v_{i+1}, \ldots, v_{j-1}, v_j)$ is a sequence of vertices counterclockwise on the boundary of P. $C_{i,j}$ is monotone with respect to a line ℓ if the projections of the vertices $(v_i, \ldots, v_j)$ on ℓ are ordered as the vertices in $C_{i,j}$. A polygon is *monotone* if there exists two chains $C_{i,j}$ and $C_{j,i}$ that are monotone with respect to some line ℓ, as in Fig. 7.2. If a direction is chosen on line ℓ, then one of these chains is monotone increasing and the other is monotone decreasing.

An O(n) algorithm to determine if a polygon is monotone has been given by Preparata and Supowit [Preparata (81)]. This algorithm uses the Shamos [Shamos (78)] antipodal pair vector data structure (see Chapter Five). The algorithm ascertains, with respect to a given reference line, the direction of all lines ℓ for which P is monotone and the two vertices v_i and v_j that define the chains $C_{i,j}$ and $C_{j,i}$.

For the sake of completeness, the relationships amongst the various types of simple polygons is given below.

1. Convex polygons $\subset$ star-shaped polygons $\subset$ simple polygons.
2. Convex polygons $\subset$ monotone polygons $\subset$ simple polygons.
3. Star-shaped polygons $\cap$ monotone polygons $\neq \emptyset$.
4. Star-shaped polygons $\not\subset$ monotone polygons and monotone polygons $\not\subset$ star-shaped polygons.

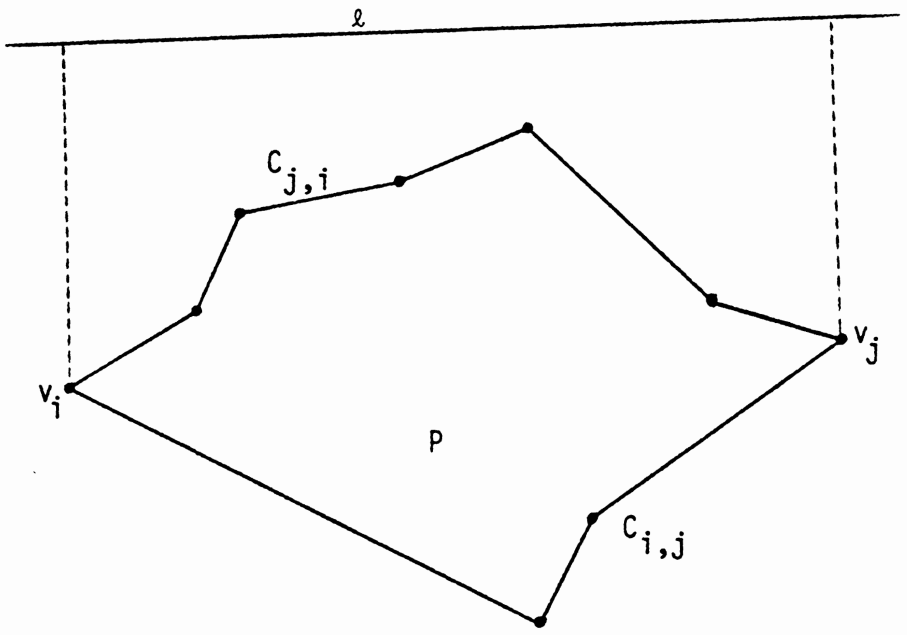


Fig. 7.2. A polygon that is monotone with respect to a line ℓ.

In the remaining Sections of this Chapter we will find it necessary to use the following definition [Bykat (78)] (which we have already found use for in both Chapters Four and Five). A point k is *above* a directed line segment from point i to point j if the quantity

$$S = x_k(y_i - y_j) + y_k(x_j - x_i) + y_j x_i - y_i x_j \qquad (1)$$

is positive. If $S < 0$, then we say k is *below* the directed line segment ij. If $S = 0$, then we say k is *on* the directed line segment ij (or anywhere on an infinite line passing through points i and j). The magnitude of S is in direct proportion to the height point k is above line segment ij.

7.5. The Superrange Algorithm for Star-shaped Polygons

Starting at v_i extend an infinite half-line ℓ through $z \in K(P)$ Fig. 7.3. Then ℓ intersects some edge $v_j v_{j+1}$ of P at a point c and splits the vertices of the polygon into a counterclockwise chain $C_{i,j}$ and a clockwise chain $\bar{C}_{i,j+1}$. Clearly, all vertices in P are in $C_{i,j} \cup \bar{C}_{i,j+1}$. Now, because of the polar ordering of vertices about z, none of the edges in either $C_{i,j}$ or $\bar{C}_{i,j+1}$ intersect ℓ. That is, (a) v_i can see c. Consider the vertex $v_q \in C_{i,j}$ ($v_q \in C_{i,j+1}$) and the line segment t from v_i to v_q. Again because of the polar ordering of vertices about z, (b) none of the edges in $C_{q,j}$ ($\bar{C}_{q,j+1}$) intersect t. Combining (a) and (b) it is immediate that for $v_q \in C_{i,j}$ ($v_q \in \bar{C}_{i,j+1}$) no edges in $\bar{C}_{i,q}$ ($C_{i,q}$) intersect line t.

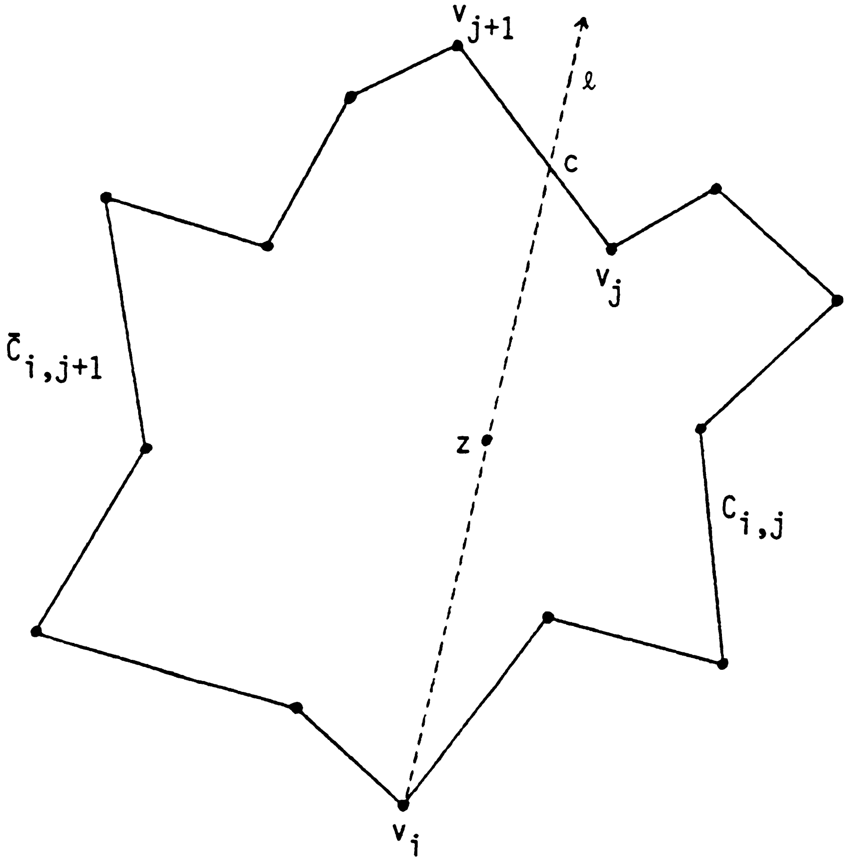


Fig. 7.3. Illustrating the superrange algorithm for star-shaped polygons.

To determine if v_i can see any vertices in $C_{i,j}$ ($\bar{C}_{i,j+1}$) we scan sequentially counterclockwise (clockwise) starting at vertex v_{i+2} (v_{i-2}). v_{i+2} (v_{i-2}) will be visible if it is above (below) the directed line segment from vertex v_i to vertex v_{i+1} (v_{i-1}). Equation (1) can be used to perform this test. For the general step we keep track of the last vertex v_{q*} which is visible from v_i. Initially, $v_{q*} := v_{i+1}$ ($v_{q*} := v_{i-1}$). During the scan, if v_{q*} is above (below) the directed line segment $\overrightarrow{v_i v_q}$, $v_q \in C_{i,j}$ ($v_q \in \bar{C}_{i,j+1}$), then we mark v_q as being visible from v_i , set $v_{q*} := v_q$, and set $v_q := v_{q+1}$ ($v_q := v_{q-1}$); otherwise we simply set $v_q := v_{q+1}$ ($v_q := v_{q-1}$). The counterclockwise (clockwise) scan terminates when v_q has been set equal to v_j (v_{j+1}) and tested for visibility. The correctness of the algorithm follows immediately from (a) and (b).

We give a more detailed version of the algorithm in PASCAL using modern data structures. On input, P is represented by a circular doubly linked-list containing the vertices of the polygon, v_i is a pointer into the doubly linked-list of that vertex for which the superrange $S(P,v_i)$ will be computed, and z contains the (x,y) coordinates of a point inside K(P). The output will consist of a circular doubly linked-list S (with v_i at the head) which will contain the superrange of vertex v_i.

```
PROCEDURE star_superrange (P : linked_list; vi : array_location;
                           z : point; VAR S : linked_list);
{ A TYPE statement corresponding to the parameters in the
PROCEDURE header follows:

TYPE
    array_location = 1 .. n;
    list_node = RECORD
                  cclock, clock : array_location;
                  x, y : real
                END;
    linked_list = ARRAY [ array_location ] OF list_node;
    point = RECORD
              x, y : real
            END; }
VAR
  j, j_plus_1, next, $v_{q*}$ , $v_q$ , temp : array_location;
BEGIN  { star_superrange }
  { we assume that P has at least 4 vertices }
  { find $v_j v_{j+1}$ , the edge on which c lies }
  j := P[$v_i$].clock  { scan clockwise }
  WHILE above_on_line (P[$v_i$].x, P[$v_i$].y, z.x, z.y, P[j].x, P[j].y)
    DO j := P[j].clock;
  j_plus_1 := P[j].cclock;
  { set S equivalent to P }
  S[$v_i$] := P[$v_i$];
  next := P[$v_i$].cclock;
  REPEAT
    S[next] := P[next];
  next := P[next].cclock;
UNTIL next = $v_i$;
{ eliminate any vertices from S that are not in S(P,$v_i$) }
{ first scan counterclockwise }
$v_{q*}$ := S[$v_i$].cclock;
$v_q$ := $v_{q*}$;
REPEAT
  $v_q$ := S[$v_q$].cclock;
  IF NOT above_on_line (S[$v_i$].x, S[$v_i$].y, S[$v_{q*}$].y, S[$v_{q*}$].x,
                                                  S[$v_q$].x, S[$v_q$].y)
    THEN
      BEGIN
        temp := S[$v_q$].cclock;
        S[$v_{q*}$].cclock := temp;
        S[temp].clock := $v_{q*}$
      END
    ELSE
      $v_{q*}$ := $v_q$
UNTIL $v_q$ = j_plus_1;
{ now scan clockwise }
$v_{q*}$ := S[$v_i$].clock;
$v_q$ := $v_{q*}$;
REPEAT
  $v_q$ := S[$v_q$].clock;
  IF NOT below_on_line (S[$v_i$].x, S[$v_i$].y, S[$v_{q*}$].y, S[$v_{q*}$].x,
            S[$v_q$].x, S[$v_q$].y)
          THEN
            BEGIN
              temp := S[$v_q$].clock;
              S[$v_{q*}$].clock := temp;
              S[temp].cclock := $v_{q*}$
            END
          ELSE
            $v_{q*}$ := $v_q$
      UNTIL $v_q$ = j_plus_1;
END;  { star_superrange }
```

The functions above_on_line and below_on_line perform the test to determine whether a point is above and on, or below and on a given line segment (see equation (1)). The first four parameters correspond to the endpoints of the line segment, and the last two parameters represent the coordinates of the point being tested.

Procedure star_superrange runs in O(n) time because each of its four loops never scans through more than n vertices, and the function above_line can be implemented to run in constant time.

7.6. The Superrange Algorithm for Monotone Polygons

Let P be monotone with respect to a line ℓ with counterclockwise chains $C_{j,k}$ and $C_{k,j}$; fig 7.4. Without loss of generality let $v_i \in C_{j,k}$. The details of the algorithm are essentially identical if $v_i \in C_{k,j}$. Each vertex $v_q \in P$ has a projection on line ℓ which we denote by proj(v_q). Let ℓ' be the line passing through v_i and the proj(v_i) on ℓ. ℓ' is clearly normal to line ℓ and partitions the projections of all vertices $v_q \in \{P - v_i\}$ into two ordered sequences. One of these sequences is made up of all vertices in the chain $C_{i+1,r}$; the other involves all vertices in $C_{i-1,r+1}$, where r denotes the index of that vertex whose projection on ℓ is perpendicularily closest to and below ℓ'. (In this sense we are giving ℓ' a direction whereby ℓ' originates at v_i and passes through the interior of P.)

Consider the line segment t from v_i to $v_q \in C_{i+1,r}$. Only those edges in $C_{i+1,r}$ whose endpoints both have projections onto line ℓ between proj(v_i) and proj(v_q) can intersect t see Fig. 7.4. This observation leads to an algorithm where the basic idea is to maintain a *window of visibility* which consists of two lines which will always pass through v_i. One of these two lines we will call the *top line of visibility,* and the other the *bottom line of visibility.* Initially, these will be ℓ' and v_iv_{i+1} respectively. The vertices of $C_{i+2,r}$ are examined against the window starting with the vertex whose perpendicular distance above ℓ' is smallest and working toward the vertex whose perpendicular distance is largest (which will always be vertex k). To

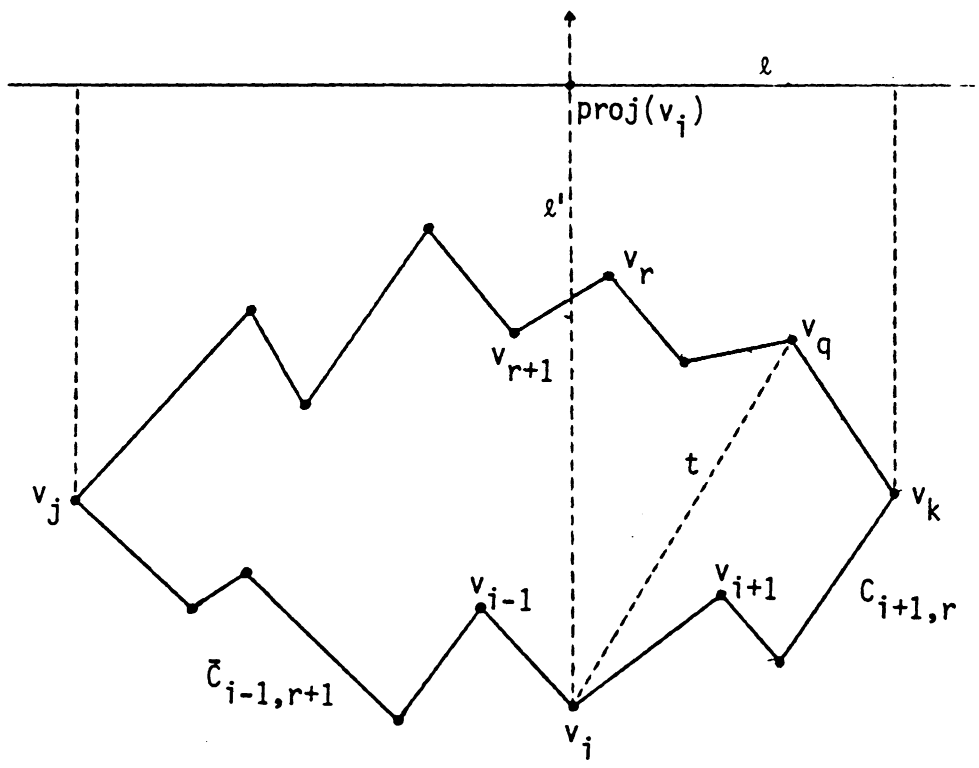


Fig. 7.4. Illustrating the superrange algorithm for monotone polygons.

efficiently implement this step we can work back-and-forth between the vertices in the chains $\bar{C}_{r,k}$ and $C_{i+2,k}$, scanning each chain in sequential order. A psuedo-code version of an algorithm to determine the visibility of any vertex $v_q \in C_{i+1,r}$ from v_i follows. A similar procedure must also be called for the case when $v_q \in \bar{C}_{i-1,r+1}$.

```
PROCEDURE monotone_superrange;

  { We assume the presence of the vertices $v_r$ , $v_{i+1}$ ,
     and $v_{i+2}$ }
  BEGIN

    $v_{bot\_chain}$ := $v_{i+1}$;
    find the point T where edge $v_r v_{r+1}$ crosses $\ell'$
    {$v_i$T defines the top line of visibility};
    B := $v_{i+1}$ {$v_i$B defines the bottom line of visibility};
    REPEAT
      IF $v_{top\_chain}$ is perpendicularily closer to $\ell'$
           than $v_{bot\_chain}$
        THEN
          BEGIN
            IF $v_{top\_chain}$ is below the bottom line of
               visibility THEN HALT {all vertices that are
               visible from $v_i$ have been marked};
            IF $v_{top\_chain}$ is below or on the top line of
               visibility THEN mark $v_{top\_chain}$ as visible from
               $v_i$ and set the top line of visibility to
```

```
              $v_iv_{top\_chain}$, i.e., T := $v_{top\_chain}$;
            $v_{top\_chain}$ := $v_{top\_chain-1}$
          END
        ELSE
          BEGIN
            IF $v_{bot\_chain}$ is above the top line of
              visibility THEN HALT {all vertices that are
              visible from $v_i$ have been marked};
            IF $v_{bot\_chain}$ is above or on the bottom line of
              visibility THEN mark $v_{bot\_chain}$ as visible from
              $v_i$ and set the bottom line of visibility to
              $v_iv_{bot\_chain}$, i.e., B := $v_{bot\_chain}$;
            $v_{bot\_chain}$ := $v_{bot\_chain-1}$
          END
    UNTIL ($v_{top\_chain}$ = $v_k$) AND ($v_{bot\_chain}$ = $v_k$);
    IF $v_k$ is not below the bottom line of visibility and not
       above the top line of visibility THEN mark $v_k$ as
       visible from $v_i$.
  END;
```

The correctness of the algorithm can be established by induction on v_{top_chain} and v_{bot_chain}. The monotone superrange procedure runs in O(n) worst-case time since each line of psuedo code can be implemented to run in constant time and never more than n-1 vertices are examined during the scan of the counterclockwise and clockwise chains ($C_{i+1,k}$ and $\bar{C}_{r,k}$).

7.7. Conclusions

The algorithms we have introduced are very simple and can be implemented with a minimum of effort. They will find application whenever a polygon is known to be star-shaped or monotone. An open question is whether checking for polygon star-shapedness or monotonicity followed by application of our special case superrange algorithms is faster than application of the El-Gindy and Avis visibility from a point algorithm.

Chapter 8

EPILOG

"So where does one find the strength to see

the race to its end? ... From within."

- Eric Liddell, *Chariots of Fire*

8.1. Summary of Research

Over the past twenty years there has been a substantial increase in the number of applications for geometric algorithms as a result of the tremendous growth of computational systems. Unfortunately, in many cases the algorithms advanced to solve specific problems have been ill-conceived, ad hoc techniques which have ultimately led to running times too large for their intended use. The major emphasis in the field of computational geometry has been not only to find computational solutions, but to produce elegant and efficient solutions as well. In this dissertation we have attempted to carry on in the same spirit by producing what we believe are a number of significant, "state-of-the-art" results.

In Chapter Three we examined the process of distributive partitioning and showed how this technique can be embodied in a three pass sorting algorithm which, from our own performance evaluation, as well as external evidence, is quite possibly the fastest general purpose internal array sorter. We then investigated the impact DPS has on the problem of selection: finding the k^{th} smallest element in an unsorted vector. Here, DPS was not found to be as promising as the Floyd-Rivest selection algorithm. However, for the problem of multiple selection we found DPS to be somewhat more efficient than sorting an entire vector of n items when the number of items being selected was small relative to the size of n.

Chapter Four was an investigation of several highly regarded algorithms for the determination of the convex hull of a set of points in the plane. What we found is that the Graham algorithm is perhaps the best of all the methods evaluated in that its performance was remarkably stable over a wide range of point distibutions in the plane. It was in the Graham algorithm that application of the DPS sort paid a huge performance dividend. Two of the other methods tested, Eddy and Akl-Toussaint, were found to be very good performers for point distributions uniformly spread throughout some enclosed planar region. A final method, due to Jarvis was disappointingly slow. A careful analysis showed that even after several improvements, poor performance was a result of the large number of arithmetic computations involved. Recognizing the importance of convex hull algorithms and their many applications, details were given to suggest how these methods can be efficiently implemented.

In the final Section of Chapter Four we showed that the concept of "convex hull" extends quite naturally to another notion of distance between two points, the L_1 metric. An algorithm based upon divide and conquer for computing the L_1 hull was discussed similar in design and performance to the Eddy convex hull algorithm. Further remarks were made indicating that, for some pattern recognition applications, computing the L_1 hull might prove more beneficial than the convex hull.

In Chapter Five a general technique was introduced that proved to be a unifying bond for the solution of two well known problems, that of finding both the diameter and minimum encasing rectangle of a set of points in the plane. The strategy employed involved an efficient means for identifying those points which are perpendicularly farthest above each edge of a convex polygon. It was this technique, along with computing the convex hull of the set of points, which proved to be the crucial link between the two problems. Previously, Shamos and Freeman-Shapira had reported the need to compute the convex hull. However, none of these researchers noticed the strong bond between these two problems. Shamos did find a diameter algorithm asymptotically equivalent to ours, but as argued, the strategy we employed appears to be more efficient and conceptually easier to grasp.

Also included in Chapter Five was an investigation of the problem of computing the L_1 diameter of a set of points. We were able to show that those points forming the diameter pair must be on the L_1 hull of the set, in the same way that the L_2 diameter pair is represented by two points on the convex hull. Unfortunately this result was of little use in the design of a new algorithm. The idea eventually advanced was a divide and conquer region delineation scheme related to the method employed to compute the L_1 hull. This idea led to a worst-case O(n) algorithm for computing both the diameter of a set and the diameter of a L_1 polygon.

Chapter Six was an investigation of four approximation algorithms, all based upon convex hull computation, for finding "very good" Euclidean traveling salesman tours. Our methods were tested against several other heuristics including nearest-neighbor, nearest insertion, farthest insertion, and another highly regarded convex hull heuristic invented by Stewart. We found that one of our methods, the AI hull heuristic, produced nearly identical results to Stewart's heuristic. Some of the other methods we introduced were somewhat better than nearest-neighbor, nearest insertion, and farthest insertion for uniform distributions of points and on a set of well known test problems. However, none proved to be as effective as either the AI hull heuristic or Stewart's method. The latter part of the Chapter concerned two ideas for improving the asymptotic running time of both the AI hull heuristic and the method due to Stewart. As a result of this work, worst-case running time of both algorithms was reduced from $O(n^3)$ to $O(n^2)$. Furthermore, implementation is straightforward for both methods. Because of our earlier work on the L_1 hull we were able to show that the AI hull heuristic and Stewart's heuristic can be adapted quite easily to produce close to optimal tours for the L_1 traveling salesman problem. In the final Section, several avenues of further research were suggested involving the combination of the convex hull along with several other geometric structures, including the Delaunay tessalation, the Relative Neighborhood graph, and the Gabriel graph.

In Chapter Seven we were able to show that some visibility problems for both star-shaped and monotone polygons are inherently easier than for (general-case) simple polygons. The specific problems investigated were for the determination of those vertices which can be seen from a specific vertex on the boundary of a star-shaped or monotone polygon. The algorithms introduced run in O(n) time, identical to the asymptotic running time of the general case superrange algorithm of El-Gindy and Avis. However, the sophistication of their algorithm makes implementation much more difficult than for our methods.

8.2. The Future of Computational Geometry

It is clear that a number of fundamentally important topics remain in computational geometry to occupy researchers for many years to come. At the beginning of the previous decade the assault was directed mainly towards two dimensional problems, with the possible exception of the hidden-line problem. This thesis has also dealt almost exclusively with problems in two dimensions. It now appears that the number of three dimensional applications is increasing, especially in the important areas of robotics and remote sensing. Thus it is time to consider the transition from two to three and higher dimensions. Unfortunately, many problems have more than one solution in the plane, only one of which generalizes to higher dimensions. Often it is not easy to recognize the correct approach when attacking a problem in higher dimensional space. So far, computational geometry in three dimensions is in its infancy. There have been very few problems which have been successfully solved by what are believed to be the most efficient techniques. One is the convex hull where a divide and conquer scheme has been successfully employed by several authors [Shamos (78)], [Preparata (77)], [Toussaint (78a)] to yield O(nlogn) algorithms. Bentley [Bentley (80a)] has also shown how it is possible to determine the two closest of n points in O(nlogn) time in any dimension k. However, the constant of proportionality grows exponentially with k.

In this dissertation we have begun a study of computational geometry involving the L_1 metric. The approach has been to tackle some of the most important problems. However, many of the problems in L_2 which have natural counterparts in L_1 remain to be investigated. Some of these include determining the superrange of an L_1 polygon, determining whether two sets in R_1 are separable, and extending L_1 geometry in general for problems in three and higher dimensions. Studies must be conducted to see if L_1 geometric algorithms can be substituted for algorithms which employ the Euclidean metric. For example, the convex hull is often used as a rough shape descriptor in pattern recognition applications. Would it not be better to use the L_1 hull for the same application, since the L_1 hull contains every point on the convex hull, but uses less space in encasing an object?

Until recently, most of the effort in computational geometry has involved finding exact solutions. But in many applications this may not be entirely necessary. For example, finding two points which are separated by a distance that is within five percent of the actual diameter of a set may be sufficient. Approximation algorithms are almost always faster than exact algorithms. Bentley, Weide, and others [Bentley (80b)], [Weide (78)] at Carnegie-Mellon University have begun to study approximate geometric algorithms with the goal of producing a solution that is within some multiplicative factor of the true or optimal solution. They have shown that finding an approximate convex hull in two and three dimensions can be accomplished by a distributive bin algorithm which is very fast. The quality of their solution is dependent upon the width (or size) and number of bins involved. Obviously, for NP-complete problems, finding good approximation algorithms has always been a necessity. However, if n is large a quadratic algorithm might be as useless as an exponential one. Thus, the study of approximation algorithms becomes increasingly important.

Geometric algorithms for parallel machine architectures is another avenue of research that has hardly been touched. Since many geometric problems are inherently local, such as finding the euclidean minimum spanning tree, computing the convex hull, or approximating a traveling salesman tour, it would appear that dividing these problems into rectangles, finding a subproblem solution using one processor for each rectangle, and then performing a fixup step merging the local solutions to obtain a global solution might be the way to proceed.

In the years to come geometry and geometric algorithms may play a major role in the development of machines which will take the description of a process and manufacture an integrated chip (or set of chips) to solve a particular problem such as finding the diameter of a set, or sorting a vector of numbers. The abilities of these machines could quite possibly depend upon layout design algorithms (heuristics) which are inherently geometric. L_1 geometry algorithms could play a major role since interconnections on present day chips are rectilinear.

8.3. Final Words

The task of this dissertation has been to continue establishing the relatively new discipline of computational geometry. Our present work has given us cause to examine many interesting problems and to propose a number of meaningful results. Certainly, we have raised as many questions as we have answered. Hopefully, the algorithmic tools developed within this work will prove to be valuable in answering these questions which will inevitably lead to the construction of more complicated programs.

Theoretical Computer Science is approximately forty years old, being in a stage similar to Theoretical Mathematics in the 18th Century. Because of the rapid dissemination of knowledge, its growth rate has been nothing less than spectacular. We believe that this dissertation has served to strengthen one small area of this diverse young field.

"Because I do not hope to turn again
Because I do not hope
Because I do not hope to turn"
- T.S. Eliot, *Ash Wednesday*

Appendix 1

COMPUTER PROGRAMS

```
C
C
C
C  ..........................................................................
C
C     SUBROUTINE UPSORT
C
C     PURPOSE
C        TO ORDER AN ARRAY OF ITEMS A INTO ASCENDING ORDER GIVEN THE
C        NUMBER OF ITEMS (N) AND THE POSITION OF THE MINIMUM
C        ITEM (MIN).  AS OUTPUT AN ARRAY OF POINTERS (POINTR) IS
C        PRODUCED INDICATING THE CORRECT ORDERING RELATION AMONGST
C        THE ITEMS OF A, I.E.,
C           A(POINTR(1)) < A(POINTR(2)) < . . . < A(POINTR(N)).
C
C     USAGE
C        CALL UPSORT(A,N,POINTR)
C
C     INPUT
C        A       - REAL; ARRAY OF ITEMS (KEYS) USED IN SORT.
C        N       - INTEGER; NUMBER OF ITEMS TO BE SORTED.
C
C     OUTPUT
C        POINTR - INTEGER; ARRAY OF POINTERS AS DESCRIBED ABOVE.
C
C     METHOD
C        1. FIND THE POSITION OF THE MAXIMUM AND MINIMUM ITEMS
C           IN ARRAY A.
C        2. IF THE MINIMUM AND MAXIMUM ITEMS IN ARRAY A ARE EQUAL,
C           THEN SET POINTR(1) ... POINTR(N) = 1 ... N, AND
C           TERMINATE THE ALGORITHM.
C        3. DISTRIBUTE THE ITEMS OF A INTO THE CREATED INTERVALS.
C           AN ITEM A(I) WILL BELONG TO INTERVAL J AS FOLLOWS:
C
C              J := (A(I) - MIN)/(MAX - MIN) * (N/2 - E) + 1
C
C           E REPRESENTS SOME SMALL NUMBER (SAY .001) TO INSURE
C           THAT THE MAXIMUM ITEM IS DISTRIBUTED INTO BOX N/2
C           (NOT BOX N/2 + 1).  NOTE THAT IN ORDER TO INCREASE
C           SPEED, THE ITEMS ARE NOT MOVED AROUND; BUT THEY ARE
C           LINKED IN LISTS, EACH LIST REPRESENTING ONE GROUP OF
C           ITEMS.
C        4. THE POINTERS ARE INITIALIZED FROM THE CREATED INTERVALS
C           AND EACH GROUP IS SORTED BY QUICKSORT IF IT CONTAINS
C           MORE THAN 8 ITEMS.  AFTER ALL THE POINTERS HAVE BEEN
C           INITIALIZED AND PARTIALLY SORTED AN INSERTION SORT
C           IS USED TO FINISH THE SORTING PROCESS.
C
C     REFERENCES
C        1. DOBOSIEWICZ, W. 'SORTING BY DISTRIBUTIVE PARTITIONING.'
C             INFO. PROC. LETT. 7, NO. 1, JAN. 1978, PP. 1-6.
C        2. MEIJER, H. AND S. G. AKL.  'THE DESIGN AND ANALYSIS OF A
C             NEW HYBRID SORTING ALGORITHM.'  INFO. PROC. LETT. 10,
C             NO. 4-5, 1980, PP. 213-218.
C        3. SEDGEWICK, R.  'IMPLEMENTING QUICKSORT PROGRAMS.'  COMM.
```

```
C           ACM 21, NO. 10, OCT. 1978, PP. 847-857.
C        4. NOGA, M. T.  'CONVEX HULL ALGORITHMS.'  M.S. THESIS,
C           DEPT. OF COMP. SCI., VIRGINIA TECH, 1981.
C
C     DESCRIPTION OF PARAMETERS
C        CONST  - REAL; DISTRIBUTION FORMULA CONSTANT.
C        DEPTH  - INTEGER; SIZE OF STACK (NO. OF UNSORTED PARTITIONS)
C        I,J    - INTEGER; INDEX VARIABLES AND QUICKSORT SCANNERS --
C                 I SCANS RIGHT (INCREMENT), J SCANS RIGHT (DECREMENT).
C        L,R    - INTEGER; LEFT AND RIGHT BOUNDS OF A PARTITION
C        LHEAD  - INTEGER; ARRAY OF LIST HEADS
C        LINK -   INTEGER; ARRAY OF FORWARD LINKS
C        LP8    - LOGICAL; IS TRUE IF THE LOWER PARTITION HAS MORE
C                 THAN 8 ELEMENTS.
C        MAX    - INTEGER; POSITION OF THE MAXIMUM ITEM.
C        MIDDLE - INTEGER; POSITION OF MIDDLE POINTER.
C        MIN    - INTEGER; POSITION OF THE MINIMUM ITEM.
C        NDIV2  - INTEGER; NUMBER OF BOXES (EQUALS N DIVIDED BY 2)
C        NEXT   - INTEGER; USED TO INDEX THROUGH THE CREATED LISTS.
C.       P      - INTEGER; INDEXING VARIABLE.
C        SIZELP - INTEGER; SIZE OF LOWER PARTITION
C        SIZEUP - INTEGER; SIZE OF UPPER PARTITION
C        STACK  - INTEGER; ARRAY HOLDS LEFT AND RIGHT BOUNDS OF
C                 UNSORTED PARTITIONS.
C        SWITCH - INTEGER; USED TO EXCHANGE POINTERS.
C        U      - INTEGER; USED TO COUNT THE NUMBER OF POINTERS IN
C                 EACH OF THE LISTS RELATIVE TO THE STARTING VALUE L.
C        UP8    - LOGICAL; IS TRUE IF THE UPPER PARTITION HAS MORE
C                 THAN 8 ITEMS.
C        V      - REAL; VALUE OF PARTITIONING ELEMENT.
C
C     AUTHOR
C        M. T. NOGA, DEPT. OF COMP. SCI., VIRGINIA TECH, BLACKSBURG,
C        VA 24061.
C
C     DATE
C        MARCH 25, 1981
C
C   ..............................................................
C
C
C
      SUBROUTINE UPSORT(A,N,POINTR)
      INTEGER DEPTH,I,J,L,LHEAD(2250),LINK(4500),MAX,MIDDLE,MIN,N,NDIV2,
     *        P,POINTR(4501),R,SIZELP,SIZEUP,STACK(2,15),SWITCH,U
      REAL A(4501),CONST,V
      LOGICAL LP8,UP8
C
C
C   STEP 1: FIND POSITION OF MAX AND MIN ITEMS
C
      MIN = 1
      MAX = 1
      DO 10 I = 2,N
         IF (A(I) .LT. A(MIN)) MIN = I
         IF (A(I) .GT. A(MAX)) MAX = I
   10 CONTINUE
C
C  STEP 2
C
      IF (A(MIN) .NE. A(MAX)) GO TO 30
      DO 20 I = 1,N
         POINTR(I) = I
   20 CONTINUE
      RETURN
C
C  STEP 3: DISTRIBUTE THE ITEMS INTO THE CREATED INTERVALS
C
   30 NDIV2 = N/2
      DO 40 I = 1,NDIV2
         LHEAD(I) = 0
   40 CONTINUE
      CONST = (NDIV2 - .001)/(A(MAX) - A(MIN))
      DO 50 I = 1,N
         J = (A(I) - A(MIN)) * CONST + 1.0
         LINK(I) = LHEAD(J)
         LHEAD(J) = I
   50 CONTINUE
C
C  STEP 4: POINTERS ARE INITIALIZED AND THEN PARTIALLY REARRANGED BY
C          QUICKSORT
C
      L = 1
      U = 1
      DO 230 P = 1,NDIV2
         IF (LHEAD(P) .EQ. 0) GO TO 230
            POINTR(U) = LHEAD(P)
            NEXT = LHEAD(P)
   60       IF (LINK(NEXT) .EQ. 0) GO TO 70
               U = U + 1
               POINTR(U) = LINK(NEXT)
               NEXT = LINK(NEXT)
               GO TO 60
C
C     BEGIN QUICKSORT
C
   70       IF ((U - L) .LT. 8) GO TO 220
               R = U
               DEPTH = 0
   80          I = L + 1
               J = R
                  MIDDLE = (L + R)/2
                  SWITCH = POINTR(MIDDLE)
                  POINTR(MIDDLE) = POINTR(I)
                  POINTR(I) = SWITCH
               IF (A(POINTR(I)) .LE. A(POINTR(R))) GO TO 90
                  SWITCH = POINTR(I)
                  POINTR(I) = POINTR(R)
                  POINTR(R) = SWITCH
   90          IF (A(POINTR(L)) .LE. A(POINTR(R))) GO TO 100
```

```
                    SWITCH = POINTR(L)
                    POINTR(L) = POINTR(R)
                    POINTR(R) = SWITCH
  100          IF (A(POINTR(I)) .LE. A(POINTR(L))) GO TO 110
                    SWITCH = POINTR(I)
                    POINTR(I) = POINTR(L)
                    POINTR(L) = SWITCH
  110          V = A(POINTR(I))
               GO TO 130
  120          SWITCH = POINTR(I)
               POINTR(I) = POINTR(J)
               POINTR(J) = SWITCH
  130          I = I + 1
               IF (V .GT. A(POINTR(I))) GO TO 130
  140          J = J - 1
               IF (V .LT. A(POINTR(J))) GO TO 140
               IF (J .GT. I) GO TO 120
               SWITCH = POINTR(J)
               POINTR(J) = POINTR(L)
               POINTR(L) = SWITCH
               SIZELP = J - L
               SIZEUP = R - I + 1
               IF (SIZELP .LT. 8) GO TO 145
                    LP8 = .TRUE.
                    GO TO 150
  145               LP8 = .FALSE.
  150          IF (SIZEUP .LT. 8) GO TO 160
                    UP8 = .TRUE.
                    GO TO 170
  160               UP8 = .FALSE.
  170          IF (LP8) GO TO 190
               IF (UP8) GO TO 180
               IF (DEPTH .EQ. 0) GO TO 220
                    L = STACK(1,DEPTH)
                    R = STACK(2,DEPTH)
                    DEPTH = DEPTH - 1
                    GO TO 80
  180               L = I
                    GO TO 80
  190          IF (UP8) GO TO 200
                    R = J - 1
                    GO TO 80
  200          IF (SIZELP .LT. SIZEUP) GO TO 210
                    DEPTH = DEPTH + 1
                    STACK(1,DEPTH) = L
                    STACK(2,DEPTH) = J - 1
                    L = I
                    GO TO 80
  210               DEPTH = DEPTH + 1
                    STACK(1,DEPTH) = I
                    STACK(2,DEPTH) = R
                    R = J - 1
                    GO TO 80
C
C      END QUICKSORT
```

```
C
  220        U = U + 1
             L = U
  230 CONTINUE
C
C  STEP4: INSERTION SORT
C
      POINTR(N+1) = N + 1
      A(N+1) = 2.
      I = N - 1
  240 IF (I .EQ. 0) RETURN
      IF (A(POINTR(I)) .GT. A(POINTR(I+1))) GO TO 250
         I = I - 1
         GO TO 240
  250 SWITCH = POINTR(I)
      V = A(SWITCH)
      J = I + 1
  260 POINTR(J-1) = POINTR(J)
      J = J + 1
      IF (A(POINTR(J)) .LT. V) GO TO 260
         POINTR(J-1) = SWITCH
         I = I - 1
         GO TO 240
      END
```

```
C
C
C
C  ..............................................................
C
C     SUBROUTINE USORT
C
C     PURPOSE
C        SORTS THE N ELEMENT REAL ARRAY A INTO ASCENDING ORDER.
C        AN AUXILLARY ARRAY B IS USED TO SPEED UP THE SORTING
C        PROCESS AND TO HOLD THE FINAL SORTED VECTOR.
C
C     USAGE
C        CALL USORT(A,N,B)
C
C     INPUT
C        A - REAL; ARRAY OF REALS (UNSORTED)
C        N - INTEGER; NUMBER OF ELEMENTS IN A
C        PARMBX - REAL; PARAMETER MULTIPLIED BY N TO GET THE NUMBER
C                 BOXES USED IN THE DISTRIBUTIVE PASS.
C
C     OUTPUT
C        B - REAL; COPY OF INPUT ARRAY A SORTED IN ASCENDING ORDER.
C
C     METHOD
C        1. FIND THE POSITIONS OF THE MAXIMUM AND MINIMUM ITEMS IN
C           ARRAY A.
C        2. IF THE MINIMUM AND MAXIMUM ITEMS IN ARRAY A ARE EQUAL,
C           THEN SET B(1) ... B(N) = A(1) ... A(N), AND
C           TERMINATE THE ALGORITHM.
C        3. DISTRIBUTE THE ITEMS OF A INTO THE CREATED INTERVALS.
C           AN ITEM A(I) WILL BELONG TO INTERVAL J AS FOLLOWS:
C
C              J = (A(I) - MIN)/(MAX - MIN) * (NBOX - E) + 1
C
C           E REPRESENTS SOME SMALL NUMBER (SAY .001) TO INSURE
C           THAT THE MAXIMUM ITEM IS DISTRIBUTED INTO BOX NBOX
C           (NOT BOX NBOX + 1).  NOTE THAT IN ORDER TO INCREASE
C           SPEED, THE ITEMS ARE NOT MOVED AROUND; BUT THEY ARE
C           LINKED IN LISTS, EACH LIST REPRESENTING ONE GROUP OF
C           ITEMS.
C        4. EACH GROUP IS IN TURN PARTIALLY SORTED BY QUICKSORT
C           IF IT CONTAINS MORE THAN 8 ITEMS.  AFTER THIS AN
C           INSERTION SORT IS USED TO COMPLETE THE SORTING PROCESS.
C
C     REFERENCES
C        1. DOBOSIEWICZ, W.  'SORTING BY DISTRIBUTIVE PARTITIONING.'
C             INFO. PROC. LETT. 7, NO. 1, JAN. 1978, PP. 1-6.
C        2. MEIJER, H. AND S. G. AKL.  'THE DESIGN AND ANALYSIS OF A
C             NEW HYBRID SORTING ALGORITHM.'  INFO. PROC. LETT. 10,
C             NO. 4-5, 1980, PP. 213-218.
C        3. SEDGEWICK, R.  'IMPLEMENTING QUICKSORT PROGRAMS.'  COMM.
C             ACM 21, NO. 10, OCT. 1978, PP. 847-857.
C        4. NOGA, M. T.  'FAST GEOMETRIC ALGORITHMS.'  PH.D. THESIS,
C             DEPT. OF COMP. SCI., VIRGINIA TECH, 1984.
```

```
C
C
C     DESCRIPTION OF PARAMETERS
C        CONST  - REAL; DISTRIBUTION FORMULA CONSTANT.
C        DEPTH  - INTEGER; SIZE OF STACK (NO. OF UNSORTED PARTITIONS)
C        I,J    - INTEGER; INDEX VARIABLES AND QUICKSORT SCANNERS --
C                 I SCANS RIGHT (INCREMENT), J SCANS RIGHT (DECREMENT).
C        L,R    - INTEGER; LEFT AND RIGHT BOUNDS OF A PARTITION
C        LHEAD  - INTEGER; ARRAY OF LIST HEADS
C        LINK -   INTEGER; ARRAY OF FORWARD LINKS
C        LP8    - LOGICAL; IS TRUE IF THE LOWER PARTITION HAS MORE
C                 THAN 8 ELEMENTS.
C        MAX    - INTEGER; POSITION OF THE MAXIMUM ITEM.
C        MIDDLE - INTEGER; POSITION OF MIDDLE POINTER.
C        MIN    - INTEGER; POSITION OF THE MINIMUM ITEM.
C        NBOX   - INTEGER; NUMBER OF BOXES
C        NEXT   - INTEGER; USED TO INDEX THROUGH THE CREATED LISTS.
C        P      - INTEGER; INDEXING VARIABLE.
C        SIZELP - INTEGER; SIZE OF LOWER PARTITION.
C        SIZEUP - INTEGER; SIZE OF UPPER PARTITION.
C        STACK  - INTEGER; ARRAY HOLDS LEFT AND RIGHT BOUNDS OF
C                 UNSORTED PARTITIONS.
C        SWITCH - INTEGER; USED TO EXCHANGE POINTERS.
C        U      - INTEGER; USED TO COUNT THE NUMBER OF POINTERS IN
C                 EACH OF THE LISTS RELATIVE TO THE STARTING VALUE L.
C        UP8    - LOGICAL; IS TRUE IF THE UPPER PARTITION HAS MORE
C                 THAN 8 ITEMS.
C        V      - REAL; VALUE OF PARTITIONING ELEMENT.
C
C     AUTHOR
C        M. T. NOGA, DEPT. OF COMP. SCI., VIRGINIA TECH, BLACKSBURG,
C        VA 24061.
C
C     DATE
C        JANUARAY 13, 1984
C
C  .....................................................................
C
C
C
      SUBROUTINE USORT(A,N,PARMBX,B)
      INTEGER DEPTH,I,J,L,LHEAD(32040),LINK(8010),MAX,MIN,N,
     *        NBOX,NEXT,P,R,SIZELP,SIZEUP,STACK(2,20),SWITCH,U
      REAL A(8010),B(8011),PARMBX,CONST,V
      LOGICAL LP8,UP8
C
C
C  STEP 1: FIND POSITION OF MAX AND MIN ITEMS
C
      MIN = 1
      MAX = 1
      DO 5 I = 2,N
         IF (A(I) .LT. A(MIN)) MIN = I
         IF (A(I) .GT. A(MAX)) MAX = I
    5 CONTINUE
```

```
C
C  STEP 2
C
      IF (A(MIN) .NE. A(MAX)) GO TO 15
      DO 10 I = 1,N
         B(I) = A(I)
   10 CONTINUE
      RETURN
C
C  STEP 3: DISTRIBUTE THE ITEMS INTO THE CREATED INTERVALS
C
   15 NBOX = N * PARMBX
      DO 20 I = 1,NBOX
         LHEAD(I) = 0
   20 CONTINUE
      CONST = (NBOX - .001)/(A(MAX) - A(MIN))
      DO 30 I = 1,N
         J = (A(I) - A(MIN)) * CONST + 1.0
         LINK(I) = LHEAD(J)
         LHEAD(J) = I
   30 CONTINUE
C
C  STEP 4: EACH LIST IS DUMPED INTO ARRAY B AND PARTIALLY
C          SORTED BY QUICKSORT
C
      L = 1
      U = 1
      DO 220 P = 1,NBOX
         IF (LHEAD(P) .EQ. 0) GO TO 220
            NEXT = LHEAD(P)
            B(U) = A(NEXT)
   40       IF (LINK(NEXT) .EQ. 0) GO TO 50
               U = U + 1
               NEXT = LINK(NEXT)
               B(U) = A(NEXT)
               GO TO 40
C
C     BEGIN QUICKSORT
C
   50       IF ((U - L) .LT. 8) GO TO 210
               R = U
               DEPTH = 0
   60          I = L + 1
               J = R
                  MIDDLE = (L + R)/2
                  SWITCH = B(MIDDLE)
                  B(MIDDLE) = B(I)
                  B(I) = SWITCH
               IF (B(I) .LE. B(R)) GO TO 70
                  SWITCH = B(I)
                  B(I) = B(R)
                  B(R) = SWITCH
   70          IF (B(L) .LE. B(R)) GO TO 80
                  SWITCH = B(L)
                  B(L) = B(R)
```

```
                   B(R) = SWITCH
   80           IF (B(I) .LE. B(L)) GO TO 90
                   SWITCH = B(I)
                   B(I) = B(L)
                   B(L) = SWITCH
   90           V = B(L)
                GO TO 110
  100           SWITCH = B(I)
                B(I) = B(J)
                B(J) = SWITCH
  110           I = I + 1
                IF (V .GT. B(I)) GO TO 110
  120           J = J - 1
                IF (V .LT. B(J)) GO TO 120
                IF (J .GE. I) GO TO 100
                SWITCH = B(J)
                B(J) = B(L)
                B(L) = SWITCH
                SIZELP = J - L
                SIZEUP = R - I + 1
                IF (SIZELP .LT. 8) GO TO 130
                   LP8 = .TRUE.
                   GO TO 140
  130              LP8 = .FALSE.
  140           IF (SIZEUP .LT. 8) GO TO 150
                   UP8 = .TRUE.
                   GO TO 160
  150              UP8 = .FALSE.
  160           IF (LP8) GO TO 180
                IF (UP8) GO TO 170
                IF (DEPTH .EQ. 0) GO TO 210
                   L = STACK(1,DEPTH)
                   R = STACK(2,DEPTH)
                   DEPTH = DEPTH - 1
                   GO TO 60
  170              L = I
                   GO TO 60
  180           IF (UP8) GO TO 190
                   R = J - 1
                   GO TO 60
  190           IF (SIZELP .LT. SIZEUP) GO TO 200
                   DEPTH = DEPTH + 1
                   STACK(1,DEPTH) = I
                   STACK(2,DEPTH) = J - 1
                   L = I
                   GO TO 60
  200              DEPTH = DEPTH + 1
                   STACK(1,DEPTH) = I
                   STACK(2,DEPTH) = R
                   R = J - 1
                   GO TO 60
C
C     END QUICKSORT
C
  210         U = U + 1
```

```
            L = U
  220 CONTINUE
C
C  STEP 4: INSERTION SORT
C
C
      B(N+1) = 1.0E20
      I = N - 1
  230 IF (I .EQ. 0) RETURN
      IF (B(I) .GT. B(I+1)) GO TO 240
         I = I - 1
         GO TO 230
  240 V = B(I)
      J = I + 1
  250 B(J-1) = B(J)
      J = J + 1
      IF (B(J) .LT. V) GO TO 250
         B(J-1) = V
         I = I - 1
         GO TO 230
      END
```

```
C  .................................................................
C
C     SUBROUTINE QSORT
C
C     PURPOSE
C        SORTS THE N ELEMENT REAL ARRAY A INTO ASCENDING ORDER
C
C     USAGE
C        CALL QSORT(A,N)
C
C     INPUT
C        A - REAL; ARRAY OF REALS (UNSORTED)
C        N - INTEGER; NUMBER OF ELEMENTS IN A
C
C     OUTPUT
C        A - (AS ABOVE) IN ASCENDING ORDER
C
C     METHOD
C        QSORT IS TAKEN FROM A PAPER BY ROBERT SEDGEWICK,
C        BROWN UNIVERSITY, 'IMPLEMENTING QUICKSORT PROGRAMS,'
C        CACM 21, NO. 10, OCT. 1978, PP. 847-857.
C
C        ESSENTIALLY WE USED PROGRAM 2, PAGE 851, AS THE MODEL
C        FOR THE IMPLEMENTATION.  BECAUSE QSORT IS WRITTEN IN
C        FORTRAN WE FOUND IT BENEFICIAL TO USE SOME OF THE ASSEMBLY
C        LANGUAGE MODIFICATIONS GIVEN ON PAGE 853.
C
C     DESCRIPTION OF PARAMETERS
C        DEPTH  - INTEGER; SIZE OF STACK (NUMBER OF UNSORTED PARTITIONS)
C        I,J    - INTEGER; SCANNERS -- I SCANS RIGHT (INCREMENT)
C                 J SCANS LEFT (DECREMENT).
C        L,R    - INTEGER; LEFT AND RIGHT BOUNDS OF A PARTITION
C        LP8   - LOGICAL; IS TRUE IF THE LOWER PARTITION HAS MORE
C                 THAN 8 ELEMENTS.
C        MIDDLE - INTEGER; POSITION OF THE MIDDLE ELEMENT
C        SIZELP - INTEGER; SIZE OF LOWER PARTITION
C        SIZEUP - INTEGER; SIZE OF UPPER PARTITION
C        STACK  - INTEGER; ARRAY HOLDS LEFT AND RIGHT BOUNDS OF
C                 UNSORTED PARTITIONS
C        SWITCH - REAL; USED TO EXCHANGE ELEMENTS
C        UP8   - LOGICAL; IS TRUE IF THE UPPER PARTITION HAS MORE
C                 THAN 8 ELEMENTS
C        V      - REAL; VALUE OF PARTITIONING ELEMENT
C
C     AUTHOR
C        M. T. NOGA, DEPT. OF COMP. SCI., VIRGINIA TECH
C
C     DATE
C        MARCH 9, 1981
C
C  .................................................................
C
C
C
      SUBROUTINE QSORT(A,N)
      INTEGER I,J,L,R,DEPTH,STACK(2,20),MIDDLE,SIZELP,SIZEUP
      REAL A(8011),SWITCH,V
      LOGICAL LP8,UP8
C
C
      L = 1
      R = N
      IF ((R - L) .LT. 8) GO TO 200
      DEPTH = 0
C
C   FIND PARTITIONING ELEMENT V USING MEDIAN OF THREE
C   MODIFICATION, PAGE 851.
C
   15 I = L + 1
      J = R
         MIDDLE = (L + R)/2
         SWITCH = A(MIDDLE)
         A(MIDDLE) = A(I)
         A(I) = SWITCH
      IF (A(I) .LE. A(R)) GO TO 20
         SWITCH = A(I)
         A(I) = A(R)
         A(R) = SWITCH
   20 IF (A(L) .LE. A(R)) GO TO 30
         SWITCH = A(L)
         A(L) = A(R)
         A(R) = SWITCH
   30 IF (A(I) .LE. A(L)) GO TO 40
         SWITCH = A(I)
         A(I) = A(L)
         A(L) = SWITCH
   40 V = A(L)
C
C   INNER LOOP SEQUENCE ... PAGE 853
C
      GO TO 60
   50 SWITCH = A(I)
      A(I) = A(J)
      A(J) = SWITCH
   60 I = I + 1
      IF (V .GT. A(I)) GO TO 60
   70 J = J - 1
      IF (V .LT. A(J)) GO TO 70
      IF (J .GE. I) GO TO 50
C
C   PLACE PARTITIONING ELEMENT INTO CORRECT ORDER
C
      SWITCH = A(J)
      A(J) = A(L)
      A(L) = SWITCH
C
C   SMALL SUBFILES ARE IGNORED AND ONLY THOSE WITH MORE THAN
C   10 ELEMENTS ARE PARTITIONED AGAIN.
C
      SIZELP = J - L
      SIZEUP = R - I + 1
      IF (SIZELP .LT. 8) GO TO 100
         LP8 = .TRUE.
         GO TO 110
  100    LP8 = .FALSE.
  110 IF (SIZEUP .LT. 8) GO TO 120
         UP8 = .TRUE.
         GO TO 125
  120    UP8 = .FALSE.
  125 IF (LP8) GO TO 140
      IF (UP8) GO TO 130
      IF (DEPTH .EQ. 0) GO TO 200
         L = STACK(1,DEPTH)
         R = STACK(2,DEPTH)
         DEPTH = DEPTH - 1
         GO TO 15
  130    L = I
         GO TO 15
  140 IF (UP8) GO TO 150
         R = J - 1
         GO TO 15
  150 IF (SIZELP .LT. SIZEUP) GO TO 160
         DEPTH = DEPTH + 1
         STACK(1,DEPTH) = L
         STACK(2,DEPTH) = J - 1
         L = I
         GO TO 15
  160    DEPTH = DEPTH + 1
         STACK(1,DEPTH) = I
         STACK(2,DEPTH) = R
         R = J - 1
         GO TO 15
C
C  INSERTION SORT
C
  200 A(N+1) = 20
      I = N - 1
  210 IF (I .EQ. 0) RETURN
      IF (A(I) .GT. A(I+1)) GO TO 220
         I = I - 1
         GO TO 210
  220 V = A(I)
      J = I + 1
  230 A(J-1) = A(J)
      J = J + 1
      IF (A(J) .LT. V) GO TO 230
         A(J-1) = V
         I = I - 1
         GO TO 210
      END
```

```
C  ..................................................................
C
C     SUBROUTINE QPSORT
C
C     PURPOSE
C        POINTER SORTS THE N ELEMENT REAL ARRAY A
C
C     USAGE
C        CALL QPSORT(A,POINTR,N)
C
C     INPUT
C        A - REAL; ARRAY OF REALS
C        N - INTEGER; NUMBER OF ELEMENTS IN A
C
C     OUTPUT
C        POINTR - INTEGER; ARRAY IS REARRANGED SO THAT
C                 A(POINTR(1)) < A(POINTR(2)) < ... <
C                 A(POINTR(N)).
C
C     METHOD
C        QPSORT IS TAKEN FROM A PAPER BY ROBERT SEDGEWICK,
C        BROWN UNIVERSITY, 'IMPLEMENTING QUICKSORT PROGRAMS,'
C        CACM 21, NO. 10, OCT. 1978, PP. 847-857.
C
C        THE ALGORITHM USED IS SLIGHT MODIFICATION OF PROGRAM 2,
C        PAGE 851.  ALSO, SEE SUBROUTINE QQSORT FOR A STRAIGHT
C        EXCHANGE OF KEYS IMPLEMENTATION OF THE BASIC ALGORITHM.
C
C     DESCRIPTION OF PARAMETERS
C        DEPTH  - INTEGER; SIZE OF STACK (NUMBER OF UNSORTED PARTITIONS)
C        I,J    - INTEGER; SCANNERS -- I SCANS RIGHT (INCREMENT)
C                 J SCANS LEFT (DECREMENT).
C        L,R    - INTEGER; LEFT AND RIGHT BOUNDS OF A PARTITION
C        LP8    - LOGICAL; IS TRUE IF THE LOWER PARTITION HAS MORE
C                 THAN 8 ELEMENTS.
C        MIDDLE - INTEGER; POSITION OF THE MIDDLE POINTER
C        SIZELP - INTEGER; SIZE OF LOWER PARTITION
C        SIZEUP - INTEGER; SIZE OF UPPER PARTITION
C        STACK  - INTEGER; ARRAY HOLDS LEFT AND RIGHT BOUNDS OF
C                 UNSORTED PARTITIONS
C        SWITCH - INTEGER; USED TO EXCHANGE POINTERS
C        UP8    - LOGICAL; IS TRUE IF THE UPPER PARTITION HAS MORE
C                 THAN 8 ELEMENTS
C        V      - REAL; VALUE OF PARTITIONING ELEMENT
C
C     AUTHOR
C        M. T. NOGA, DEPT. OF COMP. SCI., VIRGINIA TECH
C
C     DATE
C        MARCH 19, 1981
C
C  ..................................................................
C
C
C
```

```
      SUBROUTINE QPSORT(A,POINTR,N)
      INTEGER I,J,L,R,DEPTH,STACK(2,15),MIDDLE,POINTR(7501),
     *        SIZELP,SIZEUP,SWITCH
      REAL A(7501),V
      LOGICAL LP8,UP8
C
C
      L = 1
      R = N
      DO 10 I = 1,N
         POINTR(I) = I
   10 CONTINUE
      IF ((R - L) .LT. 8) GO TO 200
      DEPTH = 0
C
C  FIND PARTITIONING ELEMENT V USING MEDIAN OF THREE
C  MODIFICATION, PAGE 851.
C
   15 I = L + 1
      J = R
         MIDDLE = (L + R)/2
         SWITCH = POINTR(MIDDLE)
         POINTR(MIDDLE) = POINTR(I)
         POINTR(I) = SWITCH
      IF (A(POINTR(I)) .LE. A(POINTR(R))) GO TO 20
         SWITCH = POINTR(I)
         POINTR(I) = POINTR(R)
         POINTR(R) = SWITCH
   20 IF (A(POINTR(L)) .LE. A(POINTR(R))) GO TO 30
         SWITCH = POINTR(L)
         POINTR(L) = POINTR(R)
         POINTR(R) = SWITCH
   30 IF (A(POINTR(I)) .LE. A(POINTR(L))) GO TO 40
         SWITCH = POINTR(I)
         POINTR(I) = POINTR(L)
         POINTR(L) = SWITCH
   40 V = A(POINTR(L))
C
C  INNER LOOP SEQUENCE ... PAGE 853
C
      GO TO 60
   50 SWITCH = POINTR(I)
      POINTR(I) = POINTR(J)
      POINTR(J) = SWITCH
   60 I = I + 1
      IF (V .GT. A(POINTR(I))) GO TO 60
   70 J = J - 1
      IF (V .LT. A(POINTR(J))) GO TO 70
      IF (J .GE. I) GO TO 50
C
C  PLACE PARTITIONING ELEMENT INTO CORRECT ORDER
C
      SWITCH = POINTR(J)
      POINTR(J) = POINTR(L)
      POINTR(L) = SWITCH
```

```
C
C  SMALL SUBFILES ARE IGNORED AND ONLY THOSE WITH MORE THAN
C  8 ELEMENTS ARE PARTITIONED AGAIN.
C
      SIZELP = J - L
      SIZEUP = R - I + 1
      IF (SIZELP .LT. 8) GO TO 100
         LP8 = .TRUE.
         GO TO 110
  100    LP8 = .FALSE.
  110 IF (SIZEUP .LT. 8) GO TO 120
         UP8 = .TRUE.
         GO TO 125
  120    UP8 = .FALSE.
  125 IF (LP8) GO TO 140
      IF (UP8) GO TO 130
      IF (DEPTH .EQ. 0) GO TO 200
         L = STACK(1,DEPTH)
         R = STACK(2,DEPTH)
         DEPTH = DEPTH - 1
         GO TO 15
  130    L = I
         GO TO 15
  140 IF (UP8) GO TO 150
         R = J - 1
         GO TO 15
  150 IF (SIZELP .LT. SIZEUP) GO TO 160
         DEPTH = DEPTH + 1
         STACK(1,DEPTH) = L
         STACK(2,DEPTH) = J - 1
         L = I
         GO TO 15
  160    DEPTH = DEPTH + 1
         STACK(1,DEPTH) = I
         STACK(2,DEPTH) = R
         R = J - 1
         GO TO 15
C
C  INSERTION SORT
C
  200 POINTR(N+1) = N + 1
      A(N+1) = 20
      I = N - 1
  210 IF (I .EQ. 0) RETURN
      IF (A(POINTR(I)) .GT. A(POINTR(I+1))) GO TO 220
         I = I - 1
         GO TO 210
  220 SWITCH = POINTR(I)
      V = A(SWITCH)
      J = I + 1
  230 POINTR(J-1) = POINTR(J)
      J = J + 1
      IF (A(POINTR(J)) .LT. V) GO TO 230
         POINTR(J-1) = SWITCH
         I = I - 1
      GO TO 210
      END
```

```
C
C
C
C  ..................................................................
C
C     SUBROUTINE GRAHAM
C
C     PURPOSE
C        TO FIND THE VERTICES OF THE ORDERED CONVEX HULL GIVEN
C        A SET OF FINITE POINTS IN THE X,Y PLANE.  ON OUTPUT
C        THE CONVEX HULL WILL BE IN STANDARD FORM.
C
C     USAGE
C        CALL GRAHAM(H,L)
C
C     INPUT
C        X,Y - REAL; ARRAYS DESIGNATED TO HOLD THE X,Y COORDINATES
C              OF THE SET OF PLANAR POINTS.
C        N   - INTEGER; NUMBER OF X,Y COORDINATE PAIRS.
C
C     OUTPUT
C        H   - INTEGER; ARRAY HOLDING THE INDICES OF THOSE X,Y
C              POINTS THAT ARE VERTICES OF THE HULL.
C        L   - INTEGER; NUMBER OF CONVEX HULL VERTICES.
C
C     METHOD
C           THE FOLLOWING NOTATION IS ADOPTED:  S IS A SET OF POINTS.
C           EACH POINT, I, IN S, HAS X AND Y COORDINATES X(I) AND Y(I)
C           RESPECTIVELY.  POINTS WILL BE DELETED FROM S AS THE ALGOR-
C           ITHM PROGRESSES UNTIL ONLY THE EXTREME POINTS OF THE HULL
C           REMAIN.
C
C           STEP 1.  FIND THE POINT M, IN S, WHICH IS THE BOTTOMMOST
C           POINT.  IF MORE THAN ONE SUCH POINT EXISTS, CHOOSE THE
C           LEFTMOST.  THUS, ALL POINTS IN S ARE EITHER ABOVE OR TO THE
C           RIGHT OF M.  (SUBROUTINE BOTTOM)
C
C           STEP 2.  COMPUTE A THETA FOR EACH POINT IN S AS
C
C              IF (X(I) NE X(M)) OR (Y(I) NE Y(M)) THEN THETA(I) =
C                 -(X(I) - X(M))/ABS((X(I) - X(M)) + (Y(I) - Y(M)))
C              ELSE THETA(I) = -1.001.
C
C           THE ELSE CLAUSE PREVENTS DIVISION BY ZERO, THEREBY TRAP-
C           PING THOSE POINTS COINCIDENT WITH M.  ALSO, IT IS UNNECES-
C           SARY TO DELETE ANY POINTS FROM S UNTIL THE ORDERING STEP
C           IF COMPLETE.  (SUBROUTINE ANGVAL)
C
C           STEP 3.  ORDER THE POINTS BY ANGULAR VALUE AS COMPUTED ABOVE
C           IN STEP 2.  (SUBROUTINE UPSORT)
C
C           STEP 4.  TO DELETE POINTS FROM S, THETA(I) IS USED TO DEFINE
C           TWO FUNCTIONS, CLOCK(I) AND CCLOCK(I).  CCLOCK(I) RETURNS
C           THE NEXT POINT IN S COUNTERCLOCKWISE FROM I AND CLOCK(I)
C           RETURNS THE NEXT POINT IN S CLOCKWISE FROM I.  IMPLEMENTA-
```

```
C           TION OF THIS STEP WILL INVOLVE INITIALIZING A DOUBLY
C           CIRCULAR LINKED-LIST.
C
C           STEP 5.  ELIMINATE ANY POINT COINCIDENT WITH M, AND IF THERE
C           IS MORE THAN ONE POINT ALONG ANY ONE RAY, THEN REMOVE ALL
C           POINTS BETWEEN M AND THE POINT FARTHEST ALONG THE RAY FROM
C           M.  INCLUSION OF THIS STEP IS ESSENTIAL, OTHERWISE STEP 6
C           (THE CONCAVITY TEST) COULD REMOVE THE WRONG POINT.  (SUBROU-
C           TINE ELIM)
C
C           STEP 6.  SET I = M, J = CCLOCK(I), K = CCLOCK(J) AND
C           DO CONCAVITY TESTING WHILE K NE M
C
C              IF (X(J) - X(I)) * (Y(K) - Y(I)) > (X(K) - X(I))
C                 * (Y(J) - Y(I)) THEN SET I=J; J=K; K=CCLOCK(K)
C              ELSE SET CLOCK(K) = I; CCLOCK(I) = K; J = I; I = CLOCK(J)
C
C           THE CONDITION DETERMINES WHETHER J IS ON THE SAME SIDE OF
C           THE LINE FROM I TO K AS M IS.  (SUBROUTINE CONCAV)
C
C              AT THE COMPLETION OF STEP 6, S WILL CONTAIN THE
C           EXTREME POINTS OF THE HULL.  K WILL BE SET EQUAL TO
C           EVERY POINT IN S, EXCEPT M AND CCLOCK(M), AT LEAST ONCE
C           DURING STEP 6, AND EVERY POINT NOT IN THE HULL WILL
C           CAUSE ANOTHER POINT TO BE REEXAMINED.  THUS STEP 6
C           WILL BE DONE AT MOST (N - 2) + (N - Q) TIMES, WHERE
C           Q IS THE NUMBER OF POINTS ON THE HULL.
C
C           REMARK ON SORTING:  THE ORDERING PROCESS OF STEP 3 MAY
C           BE CARRIED OUT BY USE OF A POINTER SORT.  POINTER SORTING
C           IS MOST EFFICIENT FOR STRUCTURES CONSISTING OF AGGREGATE
C           DATA.  THE SORTING ALGORITHM WE CHOSE IS A DISTRIBUTIVE
C           PARTITIONING HYBRID CALLED UPSORT.
C
C     REFERENCES
C        1. NOGA, M. T.  'CONVEX HULL ALGORITHMS.'
C           MASTER'S THESIS, VIRGINIA TECH, BLACKSBURG, VA. 24061.
C        2. GRAHAM, R. L.  'AN EFFICIENT ALGORITHM FOR DETERMINING
C           THE CONVEX HULL OF A FINITE PLANAR SET.'  INFO. PROC.
C           LETT. 1, JAN. 1972, PP. 132-133.
C        3. ANDERSON, K. R.  'A REEVALUATION OF AN EFFICIENT ALGORITHM
C           FOR DETERMINING THE CONVEX HULL OF A FINITE PLANAR SET.'
C           INFO. PROC. LETT. 7, NO. 1, JAN. 1978, PP. 53-55.
C        4. SHAMOS, M. I.  'COMPUTATIONAL GEOMETRY.'  PH.D. THESIS,
C           YALE UNIVERSITY, 1978.
C
C     DESCRIPTION OF PARAMETERS
C        CCLOCK - INTEGER; THE COUNTERCLOCKWISE OR FORWARD POINTER
C                 OF THE DOUBLY LINKED LIST.  CCLOCK POINTS FORWARD
C                 (OR CONTAINS THE INDEX) TO THE NEXT VIABLE X-Y
C                 COORDINATE PAIR (THOSE X-Y POINTS THAT HAVE NOT
C                 BEEN ELIMINATED).
C        CLOCK  - INTEGER; SAME AS CCLOCK ONLY CLOCKWISE OR BACKWARD.
C        DONE   - LOGICAL; USED TO CHECK WHETHER SUBROUTINE ELIM HAS
C                 DELETED ALL POINTS AS POSSIBLE CONVEX VERTICES.
```

```
C        I       - INTEGER; DO LOOP COUNTER.
C        M       - INTEGER; INDEX OF THE BOTTOMMOST POINT.
C        NP      - INTEGER; EQUALS N-1, USED AS A DO LOOP PARAMETER
C                  IN INITIALIZING DOUBLY LINKED LIST.
C        POINTR  - INTEGER; INSTEAD OF PHYSICALLY SWITCHING THE ELE-
C                  MENTS OF ARRAYS X, Y, AND THETA, THIS ARRAY IS
C                  PERMUTED TO FORM AN ORDERING RELATION OVER THE
C                  X, Y, THETA RECORD (SEE REMARK ON SORTING).
C        THETA   - REAL; THIS ARRAY IS COMPUTED BY SUBROUTINE EVAL.
C                  THETA VALUES ARE USED TO ORDER THE X-Y POINTS
C                  BY POLAR ANGLE.
C
C     AUTHOR
C        MARK NOGA, DEPT. OF COMPUTER SCIENCE, VIRGINIA TECH.
C
C     DATE
C        MARCH 8, 1981
C
C  ..................................................................
C
C
C
      SUBROUTINE GRAHAM(H,L)
      INTEGER CCLOCK(7500),CLOCK(7500),H(7500),I,L,M,N,NP,
     *        POINTR(7501)
      REAL THETA(7501),X(7500),Y(7500)
      LOGICAL DONE
      COMMON/COORDS/X,Y,N
      COMMON/ANGLES/THETA
      COMMON/LINKLS/CCLOCK,CLOCK,M
      COMMON/LISTPT/POINTR
C
C
C  BEGIN SUBROUTINE GRAHAM -- MAIN DRIVER SUBROUTINE.
C  IF THERE IS ONLY ONE POINT IN S TO BE CONSIDERED
C  RETURN WITH THAT POINT, OTHERWISE CONTINUE.
C
      IF(N .NE. 1) GO TO 5
      H(1) = 1
      L = 1
      RETURN
C
C  STEPS  , 2, AND 3
C
    5 CALL BOTTOM(M)
      CALL ANGVAL(M)
      CALL UPSORT(N,M)
C
C  STEP 4: LINK THE SORTED POINTS TOGETHER IN A DOUBLY-
C  CIRCULAR LINKED-LIST.
C
      CCLOCK(POINTR(1)) = POINTR(2)
      CLOCK(POINTR(1)) = POINTR(N)
      CCLOCK(POINTR(N)) = POINTR(1)
      CLOCK(POINTR(N)) = POINTR(N-1)
      NP = N - 1
      DO 20 I = 2,NP
         CCLOCK(POINTR(I)) = POINTR(I+1)
         CLOCK(POINTR(I)) = POINTR(I-1)
   20 CONTINUE
C
C  STEPS 5 AND 6
C
      CALL ELIM(DONE)
      IF(DONE) GO TO 25
      CALL CONCAV
C
C  PLACE THE INDICES OF THE CONVEX HULL INTO H,
C  AND RETURN TO CALLING PROCEDURE.
C
   25 I = M
      L = 1
   30 H(L) = I
      IF(CCLOCK(I) .EQ. M) RETURN
      I = CCLOCK(I)
      L = L + 1
      GO TO 30
      END
C
C
C
C  ...................................................................
C
C     SUBROUTINE BOTTOM
C
C     PURPOSE
C        TO FIND THE BOTTOMMOST POINT M IN A TWO-DIMENSIONAL
C        EUCLIDEAN POINT SET.  IF THERE IS MORE THAN ONE
C        SUCH POINT, THE LEFTMOST IS CHOSEN.  THUS ALL POINTS
C        ARE EITHER ABOVE OR TO THE RIGHT OF M.
C
C     USAGE
C        CALL BOTTOM(M)
C
C     INPUT
C        X,Y    - COORDINATES OF THE PLANAR POINTS
C        N      - NUMBER OF PLANAR POINTS
C
C     OUTPUT
C        M      - INDEX IN X AND Y OF THE BOTTOMMOST POINT
C
C     DESCRIPTION OF PARAMETERS
C        I      - INTEGER; INDEX VARIABLE
C
C  ...................................................................
C
C
C
C
      SUBROUTINE BOTTOM(M)
      INTEGER I,M,N
      REAL X(7500),Y(7500)
      COMMON/COORDS/X,Y,N
C
C
      M = 1
      DO 30 I = 2,N
         IF (Y(I) .GT. Y(M)) GO TO 30
         IF (Y(I) .NE. Y(M)) GO TO 20
         IF (X(I) .GE. X(M)) GO TO 30
   20    M = I
   30 CONTINUE
      RETURN
      END
C
C
C
C    .................................................................
C
C     SUBROUTINE ANGVAL
C
C     PURPOSE
C        SUBROUTINE ANGVAL COMPUTES AN ANGULAR VALUE (THETA) FOR EACH
C        X,Y COORDINATE PAIR.
C
C     USAGE
C        CALL ANGVAL(M)
C
C     INPUT
C        M      - INDEX OF THE BOTTOMMOST POINT
C        N      - NUMBER OF PLANAR POINTS
C        X,Y    - COORDINATES OF THE PLANAR POINTS
C
C     OUTPUT
C        THETA - ANGULAR VALUES
C
C     DESCRIPTION OF PARAMETERS
C        DIFFX,DIFFY - REAL; THE SUBTRACTED X AND Y DIFFERENCES
C                      BETWEEN SOME POINT AND M.
C        I           - INTEGER; INDEX VARIABLE.
C
C    .................................................................
C
C
C
      SUBROUTINE ANGVAL(M)
      INTEGER I,M,N
      REAL DIFFX,DIFFY,THETA(7501),X(7500),Y(7500)
      COMMON/COORDS/X,Y,N
      COMMON/ANGLES/THETA
C
C
      DO 20 I = 1,N
         IF((X(I) .NE. X(M)) .OR. (Y(I) .NE. Y(M))) GO TO 10
         THETA(I) = -1.001
```

```
          GO TO 20
   10     DIFFX = X(I) - X(M)
          DIFFY = Y(I) - Y(M)
          THETA(I) = -DIFFX/(ABS(DIFFX) + DIFFY)
   20 CONTINUE
      RETURN
      END
C
C
C
C  ..............................................................
C
C     SUBROUTINE UPSORT
C
C     PURPOSE
C        TO ORDER AN ARRAY OF ITEMS A INTO ASCENDING ORDER GIVEN THE
C        NUMBER OF ITEMS (N) AND THE POSITION OF THE MINIMUM
C        ITEM (MIN).  AS OUTPUT AN ARRAY OF POINTERS (POINTR) IS
C        PRODUCED INDICATING THE CORRECT ORDERING RELATION AMONGST
C        THE ITEMS OF A, I.E.,
C           A(POINTR(1)) < A(POINTR(2)) < . . . < A(POINTR(N)).
C
C     USAGE
C        CALL UPSORT(N,MIN)
C
C     INPUT
C        A       - REAL; ARRAY OF ITEMS (KEYS) USED IN SORT.
C        N       - INTEGER; NUMBER OF ITEMS TO BE SORTED.
C        MIN     - INTEGER; POSITION IN ARRAY A OF MINIMUM ITEM.
C
C     OUTPUT
C        POINTR - INTEGER; ARRAY OF POINTERS AS DESCRIBED ABOVE.
C
C     METHOD
C        1. FIND THE POSITION OF THE MAXIMUM ITEM IN ARRAY A.
C        2. IF THE MINIMUM AND MAXIMUM ITEMS IN ARRAY A ARE EQUAL,
C           THEN SET POINTR(1) ... POINTR(N) = 1 ... N, AND
C           TERMINATE THE ALGORITHM.
C        3. DISTRIBUTE THE ITEMS OF A INTO THE CREATED INTERVALS.
C           AN ITEM A(I) WILL BELONG TO INTERVAL J AS FOLLOWS:
C
C              J := (A(I) - MIN)/(MAX - MIN) * (N/2 - E) + 1
C
C           E REPRESENTS SOME SMALL NUMBER (SAY .001) TO INSURE
C           THAT THE MAXIMUM ITEM IS DISTRIBUTED INTO BOX N/2
C           (NOT BOX N/2 + 1).  NOTE THAT IN ORDER TO INCREASE
C           SPEED, THE ITEMS ARE NOT MOVED AROUND; BUT THEY ARE
C           LINKED IN LISTS, EACH LIST REPRESENTING ONE GROUP OF
C           ITEMS.
C        4. THE POINTERS ARE INITIALIZED FROM THE CREATED INTERVALS
C           AND EACH GROUP IS SORTED BY QUICKSORT IF IT CONTAINS
C           MORE THAN 7 ITEMS.  AFTER ALL THE POINTERS HAVE BEEN
C           INITIALIZED AND PARTIALLY SORTED AN INSERTION SORT
C           IS USED TO FINISH THE SORTING PROCESS.
C
```

```
C     REFERENCES
C        1. DOBOSIEWICZ, W. 'SORTING BY DISTRIBUTIVE PARTITIONING.'
C             INFO. PROC. LETT. 7, NO. 1, JAN. 1978, PP. 1-6.
C        2. MEIJER, H. AND S. G. AKL.  'THE DESIGN AND ANALYSIS OF A
C             NEW HYBRID SORTING ALGORITHM.'  INFO. PROC. LETT. 10,
C             NO. 4-5, 1980, PP. 213-218.
C        3. SEDGEWICK, R.  'IMPLEMENTING QUICKSORT PROGRAMS.'  COMM.
C             ACM 21, NO. 10, OCT. 1978, PP. 847-857.
C
C     DESCRIPTION OF PARAMETERS
C        CONST  - REAL; DISTRIBUTION FORMULA CONSTANT.
C        DEPTH  - INTEGER; SIZE OF STACK (NO. OF UNSORTED PARTITIONS)
C        I,J    - INTEGER; INDEX VARIABLES AND QUICKSORT SCANNERS --
C                 I SCANS RIGHT (INCREMENT), J SCANS RIGHT (DECREMENT).
C        L,R    - INTEGER; LEFT AND RIGHT BOUNDS OF A PARTITION
C        LHEAD  - INTEGER; ARRAY OF LIST HEADS
C        LINK -   INTEGER; ARRAY OF FORWARD LINKS
C        LP8    - LOGICAL; IS TRUE IF THE LOWER PARTITION HAS MORE
C                 THAN 8 ELEMENTS.
C        MAX    - INTEGER; POSITION OF THE MAXIMUM ITEM.
C        MIDDLE - INTEGER; POSITION OF MIDDLE POINTER.
C        NEXT   - INTEGER; USED TO INDEX THROUGH THE CREATED LISTS.
C        P      - INTEGER; INDEXING VARIABLE.
C        SIZELP - INTEGER; SIZE OF LOWER PARTITION
C        SIZEUP - INTEGER; SIZE OF UPPER PARTITION
C        STACK  - INTEGER; ARRAY HOLDS LEFT AND RIGHT BOUNDS OF
C                 UNSORTED PARTITIONS.
C        SWITCH - INTEGER; USED TO EXCHANGE POINTERS.
C        U      - INTEGER; USED TO COUNT THE NUMBER OF POINTERS IN
C                 EACH OF THE LISTS RELATIVE TO THE STARTING VALUE L.
C        UP8    - LOGICAL; IS TRUE IF THE UPPER PARTITION HAS MORE
C                 THAN 8 ITEMS.
C        V      - REAL; VALUE OF PARTITIONING ELEMENT.
C
C  ..................................................................
C
C
C
      SUBROUTINE UPSORT(N,MIN)
      INTEGER DEPTH,I,J,L,LHEAD(3750),LINK(7500),MAX,MIDDLE,MIN,N,NDIV2,
     *        P,POINTR(7501),R,SIZELP,SIZEUP,STACK(2,15),SWITCH,U
      REAL A(7501),CONST,V
      LOGICAL LP8,UP8
      COMMON/ANGLES/A
      COMMON/LISTPT/POINTR
C
C
C  STEP 1: FIND POSITION OF MAX ITEM
C
      MAX = 1
      DO 10 I = 2,N
         IF (A(I) .GT. A(MAX)) MAX = I
   10 CONTINUE
C
C  STEP 2
```

```
C
      IF (A(MIN) .NE. A(MAX)) GO TO 30
      DO 20 I = 1,N
         POINTR(I) = I
   20 CONTINUE
      RETURN
C
C  STEP 3: DISTRIBUTE THE ITEMS INTO THE CREATED INTERVALS
C
   30 NDIV2 = N/2
      DO 40 I = 1,NDIV2
         LHEAD(I) = 0
   40 CONTINUE
      CONST = (NDIV2 - .001)/(A(MAX) - A(MIN))
      DO 50 I = 1,N
         J = (A(I) - A(MIN)) * CONST + 1.0
         LINK(I) = LHEAD(J)
         LHEAD(J) = I
   50 CONTINUE
C
C  STEP 4: POINTERS ARE INITIALIZED AND THEN PARTIALLY REARRANGED BY
C          QUICKSORT
C
      L = 1
      U = 1
      DO  30 P = 1,NDIV2
         IF (LHEAD(P) .EQ. 0) GO TO 230
            POINTR(U) = LHEAD(P)
            NEXT = LHEAD(P)
   60       IF (LINK(NEXT) .EQ. 0) GO TO 70
               U = U + 1
               POINTR(U) = LINK(NEXT)
               NEXT = LINK(NEXT)
               GO TO 60
C
C     BEGIN QUICKSORT
C
   70       IF ((U - L) .LT. 8) GO TO 220
               R = U
               DEPTH = 0
   80          I = L + 1
               J = R
                  MIDDLE = (L + R)/2
                  SWITCH = POINTR(MIDDLE)
                  POINTR(MIDDLE) = POINTR(I)
                  POINTR(I) = SWITCH
               IF (A(POINTR(I)) .LE. A(POINTR(R))) GO TO 90
                  SWITCH = POINTR(I)
                  POINTR(I) = POINTR(R)
                  POINTR(R) = SWITCH
   90          IF (A(POINTR(L)) .LE. A(POINTR(R))) GO TO 100
                  SWITCH = POINTR(L)
                  POINTR(L) = POINTR(R)
                  POINTR(R) = SWITCH
  100          IF (A(POINTR(I)) .LE. A(POINTR(L))) GO TO 110
```

```
                  SWITCH = POINTR(I)
                  POINTR(I) = POINTR(L)
                  POINTR(L) = SWITCH
  110          V = A(POINTR(L))
               GO TO 130
  120          SWITCH = POINTR(I)
               POINTR(I) = POINTR(J)
               POINTR(J) = SWITCH
  130          I = I + 1
               IF (V .GT. A(POINTR(I))) GO TO 130
  140          J = J - 1
               IF (V .LT. A(POINTR(J))) GO TO 140
               IF (J .GE. I) GO TO 120
               SWITCH = POINTR(J)
               POINTR(J) = POINTR(L)
               POINTR(L) = SWITCH
               SIZELP = J - L
               SIZEUP = R - I + 1
               IF (SIZELP .LT. 8) GO TO 145
                  LP8 = .TRUE.
                  GO TO 150
  145             LP8 = .FALSE.
  150          IF (SIZEUP .LT. 8) GO TO 160
                  UP8 = .TRUE.
                  GO TO 170
  160             UP8 = .FALSE.
  170          IF (LP8) GO TO 190
               IF (UP8) GO TO 180
               IF (DEPTH .EQ. 0) GO TO 220
                  L = STACK(1,DEPTH)
                  R = STACK(2,DEPTH)
                  DEPTH = DEPTH - 1
                  GO TO 80
  180             L = I
                  GO TO 80
  190          IF (UP8) GO TO 200
                  R = J - 1
                  GO TO 80
  200          IF (SIZELP .LT. SIZEUP) GO TO 210
                  DEPTH = DEPTH + 1
                  STACK(1,DEPTH) = I
                  STACK(2,DEPTH) = J - 1
                  L = I
                  GO TO 80
  210             DEPTH = DEPTH + 1
                  STACK(1,DEPTH) = I
                  STACK(2,DEPTH) = R
                  R = J - 1
                  GO TO 80
C
C     END QUICKSORT
C
  220       U = U + 1
            L = U
  230 CONTINUE
```

```
C
C  STEP4: INSERTION SORT
C
      POINTR(N+1) = N + 1
      A(N+1) = 2.
      I = N - 1
  240 IF (I .EQ. 0) RETURN
      IF (A(POINTR(I)) .GT. A(POINTR(I+1))) GO TO 250
         I = I - 1
         GO TO 240
  250 SWITCH = POINTR(I)
      V = A(SWITCH)
      J = I + 1
  260 POINTR(J-1) = POINTR(J)
      J = J + 1
      IF (A(POINTR(J)) .LT. V) GO TO 260
         POINTR(J-1) = SWITCH
         I = I - 1
         GO TO 240
      END
C
C
C
C
C
C
C
C  ..................................................................
C
C     SUBROUTINE ELIM
C
C     PURPOSE
C        (1) ELIMINATES ANY POINTS COINCIDENT WITH M.
C        (2) REMOVES INNERMOST POINTS THAT LIE ALONG THE SAME RAY.
C
C     USAGE
C        CALL ELIM(DONE)
C
C     INPUT
C        CCLOCK,CLOCK - POINTERS OF THE DOUBLY-LINKED LIST
C        M            - INDEX OF BOTTOMMOST POINT
C        N            - NUMBER OF PLANAR POINTS
C        X,Y          - COORDINATES OF PLANAR POINTS
C
C     OUTPUT
C        CCLOCK,CLOCK - AS ABOVE BUT (POSSIBLY) REARRANGED
C        DONE         - IS TRUE IF ALL POINTS EXCEPT THE BOTTOMMOST
C                       HAVE BEEN ELIMINATED FROM CONSIDERATION AS CH
C                       VERTICES.
C
C     METHOD
C        POINTS ARE REMOVED BY USE OF A LINKED LIST.
C
C
C     DESCRIPTION OF PARAMETERS
```

```
C        J,K     - INTEGER; INDICES OF TWO POINTS THAT ARE BEING
C                  CHECKED UNDER CONDITIONS (1) AND (2).
C
C   ...............................................................
C
C
C
      SUBROUTINE ELIM(DONE)
      INTEGER CCLOCK(7500),CLOCK(7500),J,K,M,N
      REAL THETA(7501),X(7500),Y(7500)
      LOGICAL DONE
      COMMON/COORDS/X,Y,N
      COMMON/ANGLES/THETA
      COMMON/LINKLS/CCLOCK,CLOCK,M
C
C
C  ELIMINATE ANY POINT COINCIDENT WITH THE BOTTOMMOST
C  POINT M (THESE POINTS HAVE A THETA = -1.001).
C
      J = M
      K = CCLOCK(J)
      DONE = .FALSE.
   10 IF(THETA(K) .NE. -1.001) GO TO 20
         K = CCLOCK(K)
         IF(K .NE. M) GO TO 10
            CLOCK(J) = M
            CCLOCK(J) = M
            DONE = .TRUE.
            RETURN
   20    CLOCK(K) = J
         CCLOCK(J) = K
C
C  DELETE POINTS ALONG IDENTICAL RAYS.
C
   40 J = K
   45 K = CCLOCK(J)
      IF(K .EQ. M) RETURN
      IF(THETA(K) .NE. THETA(J)) GO TO 40
      IF(Y(K) .LT. Y(J)) GO TO 50
         CCLOCK(CLOCK(J)) = K
         CLOCK(K) = CLOCK(J)
         GO TO 40
   50    CCLOCK(J) = CCLOCK(K)
         CLOCK(CCLOCK(K)) = CLOCK(K)
         GO TO 45
      END
C
C
C
C   ...............................................................
C
C     SUBROUTINE CONCAV
C
C     PURPOSE
C        ELIMINATES ALL CONCAVE VERTICES.
```

```
C
C     USAGE
C        CALL CONCAV
C
C     INPUT
C        CCLOCK,CLOCK - LINKED LIST POINTERS
C        M            - INDEX OF BOTTOMMOST POINT
C        N            - NUMBER OF X,Y POINTS
C        X,Y          - COORDINATES OF POINTS
C
C     OUTPUT
C        CCLOCK,CLOCK - AS ABOVE BUT (POSSIBLY) REARRANGED
C
C     METHOD
C        POINTS ARE REMOVED BY REARRANGING LINKED LIST (CCLOCK,CLOCK).
C
C     DESCRIPTION OF PARAMETERS
C        I,J,K - INTEGER; INDICES OF THREE CONSECUTIVE POINTS.
C        TEMP1 - REAL; TEMPORARY VARIABLE USED TO COMPUTE CONCAVITY
C                TEST FORMULA
C        TEMP2 - REAL; SAME AS TEMP1
C
C  ...................................................................
C
C
C
      SUBROUTINE CONCAV
      INTEGER CCLOCK(7500),CLOCK(7500),I,J,K,M,N
      REAL TEMP1,TEMP2,X(7500),Y(7500)
      COMMON/COORDS/X,Y,N
      COMMON/LINKLS/CCLOCK,CLOCK,M
C
C
C  INITIALIZE; J WILL BE TESTED TO SEE IF IT IS CONCAVE
C  WITH RESPECT TO POINTS I AND K.
C
      I = M
      J = CCLOCK(I)
      K = CCLOCK(J)
C
C  DO CONCAVITY TESTING WHILE K NE M.
C
   10 IF(K .EQ. M) RETURN
         TEMP1 = (X(J) - X(I)) * (Y(K) - Y(I))
         TEMP2 = (Y(J) - Y(I)) * (X(K) - X(I))
         IF(TEMP1 .GT. TEMP2) GO TO 20
C
C     REMOVE J .
C
         CLOCK(K) = I
         CCLOCK(I) = K
         J = I
         I = CLOCK(J)
         GO TO 10
C
```

```
C     KEEP J
C
   20    I = J
         J = K
         K = CCLOCK(J)
         GO TO 10
      END
```

```
C
C
C
C
C
C
C
C
C  ..............................................................
C
C     SUBROUTINE JARVIS
C
C     PURPOSE
C        TO FIND THE CONVEX HULL OF A FINITE PLANAR SET OF POINTS S
C        IN THE TWO-DIMENSIONAL PLANE.
C
C     USAGE
C        CALL JARVIS(H,N)
C
C     DESCRIPTION OF PARAMETERS
C        ANGLE  - REAL; ARRAY HOLDING THE ANGLE FROM EACH POINT IN
C                 S TO THE BOTTOMMOST POINT M.
C        H      - INTEGER; ARRAY HOLDING THE INDICES OF THOSE ELEMENTS
C                 IN S THAT ARE VERTICES ON THE CONVEX HULL.
C        I      - INTEGER; DO LOOP PARAMETER.
C        LANG   - REAL; VALUE OF THE ANGLE FOR THE LAST FOUND CONVEX
C                 HULL POINT.
C        MX,MY  - REAL; THE X,Y COORDINATES OF THE BOTTOMMOST POINT
C                 M (SEE SUBROUTINE BOTTOM).
C        N      - INTEGER; THE NUMBER OF X,Y COORDINATES PASSED
C                 TO SUBROUTINE JARVIS THROUGH COMMON BLOCK BETA.
C        NHE    - INTEGER; THE NUMBER OF HULL ELEMENTS.
C        PIQ    - INTEGER; THE POINT IN QUESTION PRESENTLY BEING
C                 CHECKED FOR POSSIBLE INCLUSION IN H.
C        POSANG - INTEGER; POSITION OF A MINIMUM ANGLE ELEMENT (RE-
C                 TURNED BY SUBROUTINES MINANG AND NEXTPT).  S(POSANG)
C                 WILL BE ADDED TO THE SET OF HULL VERTICES.
C        POSIT  - INTEGER; THE POSITION OF THE BOTTOMMOST POINT M IN
C                 THE X AND Y ARRAYS.
C        PTB    - INTEGER; ARRAY HOLDING THE BACKWARD POINTERS OF THE
C                 LINKED LIST.
C        PTF    - INTEGER; ARRAY HOLDING THE FORWARD POINTERS OF THE
C                 LINKED LIST.
C        VN     - INTEGER; THE NUMBER OF ELEMENTS IN THE LINKED LIST.
C        VNTEMP - INTEGER; DO LOOP CONSTANT HOLDING THE VALUE OF VN
C                 WHEN THERE IS A POSSIBILITY THAT THE VALUE OF VN WILL
C                 BE CHANGED INSIDE THE LOOP.
C        X,Y    - REAL; THE SINGLE DIMENSIONED ARRAYS THAT HOLD THE
C                 THE ORIGINAL X,Y COORDINATE PAIRS OF S, THE POINT SET
C                 FOR WHICH THE CONVEX HULL (VERTICES) MUST BE DETER-
C                 MINED.
C
C     SUBROUTINE AND FUNCTION SUBPROGRAMS REQUIRED
C        SUBROUTINE BOTTOM
C        SUBROUTINE EVALJ
```

```
C         SUBROUTINE MINANG
C         SUBROUTINE NEXTPT
C
C      METHOD
C         THE ALGORITHM USED IS DUE TO R. A. JARVIS, INFORMATION
C         PROCESSING LETTERS 2 (1973), PP. 18-21.
C
C         BEFORE GIVING A STEP BY STEP ACCOUNT OF THE ALGORITHM IT MAY
C         BE INSTRUCTIVE FOR THE READER TO EXAMINE SOME IMPLEMENTATION
C         DETAILS:
C           THE COORDINATES OF S(X,Y) ARE STORED IN REAL ARRAYS X AND
C           Y.  ARRAY H IS USED TO STORE THE POSITION OF THE CONVEX
C           VERTICES.  AS SOON AS AN ELEMENT OF S IS FOUND THAT BELONGS
C           TO THE HULL IT IS DELETED FROM CONSIDERATION FOR ALL SUB-
C           SEQUENT ITERATIONS OF THE ALGORITHM, AND ITS POSITION PLACED
C           INTO ARRAY H.  TO PERFORM THE DELETIONS EFFICIENTLY A DOUBLY
C           CIRCULAR LINKED LIST IS USED.  THEREFORE, EACH TIME AN ELE-
C           MENT IS DELETED FROM THE LIST ONLY TWO POINTERS NEED BE RE-
C           ARRANGED.  DURING PROCESSING VARIABLE VN WILL HOLD THE NUMBER
C           OF ITEMS IN THE LINKED LIST (THE NUMBER OF VIABLE POINTS IN
C           S STILL UNDER CONSIDERATION AS POSSIBLE HULL VERTICES).
C
C         A STEP BY STEP DESCRIPTION OF THE ALGORITHM FOLLOWS WITH
C         NOTES ON IMPLEMENTATION WHERE APPROPRIATE:
C           STEP 1.  FIND THE BOTTOMMOST POINT AND CALL IT M.  M IS
C                    DEFINED TO BE THAT POINT THAT HAS THE MINIMUM Y-
C                    COORDINATE VALUE IN S.  IF TWO OR MORE POINTS IN
C                    S HAVE THE SAME MINIMUM Y VALUE THEN CHOOSE THE
C                    LEFTMOST ONE (I.E., THE POINT WITH THE MINIMUM
C                    X VALUE).
C           STEP 2.  STEP 1 HAS IDENTIFIED THE FIRST POINT ON THE
C                    CONVEX HULL.  PLACE THE POSITION OF M INTO THE SET
C                    OF HULL ELEMENTS, HENCEFORTH TO BE CALLED H, AND DE-
C                    LETE M FROM S USING THE IMPLEMENTATION IDEA DIS-
C                    CUSSED ABOVE.
C           STEP 3.  EVALUATE AND STORE THE ANGLES (IN ARRAY ANGLE) OF
C                    LINES FORM THE FIRST HULL POINT M TO THE OTHER
C                    POINTS IN S.  IN THE PROCESS OF EVALUATING THE
C                    ANGLES, IF A POINT IS FOUND THAT IS CO-INCIDENT
C                    (HAS THE SAME X,Y COORDINATES) WITH M THEN IT
C                    SHOULD BE DELETED FROM S, NO EVALUATION OF AN
C                    ANGLE IS NECESSARY.  TO IMPLEMENT THIS STEP WE
C                    MUST DETERMINE SOME CONVENTION FOR EVALUATING THE
C                    ANGLES.  AN EFFICIENT MEANS OF DOING SO FOLLOWS:
C                    FOR ALL I IN S COMPUTE
C
C                       ANGLE(I) = -(X(I) - MX)/(ABS(X(I)-MX) + Y(I))
C
C                    WHERE MX AND MX ARE THE COORDINATES OF THE BOTTOM-
C                    MOST POINT, AND ABS MEANS 'ABSOLUTE VALUE.'
C           STEP 4.  TO FIND THE NEXT CONVEX HULL VERTEX SCAN ARRAY ANGLE
C                    AND PICK OUT THE MINIMUM ELEMENT.  FOR EQUAL MINI-
C                    MUM ANGLES PICK THE ONE FURTHEST FROM THE ORIGIN.
C                    PLACE THE ASSOCIATED ELEMENT OF S INTO H, AND DELETE
C                    THIS ELEMENT FROM S.  BEFORE DELETING, HOWEVER, IT
```

```
C                    MAY BE WISE, LOOKING AHEAD TO STEP 6, TO STORE THE
C                    DELETED POINT'S X AND Y COORDINATES INTO REALS
C                    XORG AND YORG.  XORG AND YORG WILL BE USED AS A
C                    REFERENCE ORIGIN TO FORM ANGLES (DON'T CONFUSE THE
C                    USAGE OF THE WORD ANGLE WITH ARRAY ANGLE FROM STEP
C                    3) WITH THE REMAINING ELEMENTS OF S.
C          STEP 5.   IF ANY ANGLE ASSOCIATED WITH AN ELEMENT OF S IS
C                    FOUND TO EQUAL THE ANGLE OF THE LAST FOUND CONVEX
C                    HULL VERTEX THEN DELETE THAT ELEMENT FROM S.
C          STEP 6.   USING THE LAST FOUND CONVEX HULL VERTEX AS AN ORIGIN
C                    (XORG,YORG) CALCULATE THE NEXT HULL POINT BY USING
C                    THE IMPROVEMENT IDEA OUTLINED IN SECTION 4 OF THE
C                    JARVIS PAPER.  FOR EQUAL MINIMUM ANGLES (THIS IS NOT
C                    THE SAME ANGLE MENTIONED IN STEP 3) PICK THE ONE
C                    POINT FURTHEST FROM THE ORGIN.  HERE IT IS JUST A
C                    MATTER OF KEEPING TRACK AS WE SCAN THROUGH S WHICH
C                    ANGLE, WITH RESPECT TO THE REFERENCE ORIGIN XORG,
C                    YORG, HAS THE MINIMUM VALUE.  AFTER CALCULATING THE
C                    NEXT HULL VERTEX REASSIGN XORG AND YORG AND PLACE
C                    INTO H, WHILE DELETING IT FROM S.
C          STEP 7.   ELIMINATE ANY ELEMENT OF S HAVING AN ANGLE LESS THAN
C                    OR EQUAL TO THE ANGLE ASSOCIATED WITH THE HULL VER-
C                    TEX FOUND IN STEP 6 ABOVE.  RETURN TO STEP 6.
C          REMARK 1:   BEFORE PERFORMING STEP 1 IT MAY BE ADVANTAGEOUS
C                      TO CHECK IF THE NUMBER OF ELEMENTS IN S EXCEEDS
C                      ONE.  IF NOT THEN IT IS UNNECCESSARY TO CONTINUE.
C          REMARK 2:   ANY OF STEPS 3 THROUGH 7 MAY DECREASE THE NUMBER
C                      OF ELEMENTS IN S TO ONE OR ZERO.  IF ONE POINT
C                      REMAINS THEN ADD THIS POINT TO H.  THE ALGOR-
C                      ITHM TERMINATES WITH H CONTAINING THE ORDERED
C                      CONVEX HULL VERTICES.
C
C     PROGRAMMER
C        MARK NOGA
C
C     DATE
C        FEB. 8, 1979
C
C  ..................................................................
C
C
C
C
      SUBROUTINE JARVIS(H,NHE)
      INTEGER I,H(7500),LP,N,NHE,NP,PIQ,POSANG,POSIT,PTB(7500),
     *        PTF(7500),VN,VNTEMP
      REAL ANGLE(7500),LANG,MX,MY,X(7500),Y(7500)
      COMMON/ALPHA/X,Y
      COMMON/BETA/N
      COMMON/GAMMA/ANGLE
      COMMON/DELTA/PIQ,PTB,PTF,VN
      COMMON/ZETA/LANG,XORG,YORG
C
C
C
```

```
C
C  ******************************************************
C  *                                                    *
C  *                                                    *
C    IF THERE IS ONLY ONE POINT IN S TO BE CONSIDERED
C    RETURN WITH THAT POINT, OTHERWISE CONTINUE.
C  *                                                    *
C  *                                                    *
C  ******************************************************
C
C
      IF(N .NE. 1) GO TO 10
      H(1) = 1
      NHE = 1
      RETURN
C
C
C
C
C  ******************************************************
C  *                                                    *
C  *                                                    *
C    INITIALIZE THE DOUBLE CIRCULAR LINKED LIST.  PTF IS
C    THE ARRAY OF FORWARD LINKS, AND PTB THE CORRESPONDING
C    ARRAY OF BACKWARD LINKS.
C  *                                                    *
C  *                                                    *
C  ******************************************************
C
C
   10 PTF(N) = 1
      NP = N - 1
      DO 20 I = 1,NP
      PTF(I) = I + 1
   20 CONTINUE
      PTB(1) = N
      DO 25 I = 2,N
      PTB(I) = I - 1
   25 CONTINUE
C
C
C
C
C  ******************************************************
C  *                                                    *
C  *                                                    *
C    SUBROUTINE BOTTOM DETERMINES THE X AND Y COORDINATES
C    AND POSITION OF THE BOTTOMMOST POINT M.
C  *                                                    *
C  *                                                    *
C  ******************************************************
C
C
      CALL BOTTOM(MX,MY,POSIT)
C
```

```
C
C
C
C   *******************************************************
C   *                                                     *
C   *                                                     *
C     DELETE M FROM S AND ADD IT TO H.  VN REPRESENTS THE
C     NUMBER OF VIABLE POINTS LEFT IN S.  NHE IS THE NUMBER
C     OF ELEMENTS IN H.
C   *                                                     *
C   *                                                     *
C   *******************************************************
C
C
      PTF(PTB(POSIT)) = PTF(POSIT)
      PTB(PTF(POSIT)) = PTB(POSIT)
      VN = N - 1
      NHE = 1
      H(NHE) = POSIT
C
C
C
C
C   *******************************************************
C   *                                                     *
C   *                                                     *
C     TO PERFORM STEPS 3 THROUGH 7 OF THE MODIFIED JARVIS
C     ALGORITHM WE NEED TO START WITH AN ELEMENT OF S
C     THAT HAS NOT AS YET BEEN DELETED.  THIS IS EASY
C     SINCE WE HAVE ONLY MADE ONE DELETION SO FAR, AND WE
C     KNOW ITS POSITION.  PIQ WILL HOLD THE POSITION OF
C     THE POINT IN QUESTION.
C   *                                                     *
C   *                                                     *
C   *******************************************************
C
C
      PIQ = PTF(POSIT)
C
C
C
C
C   *******************************************************
C   *                                                     *
C   *                                                     *
C     TO PERFORM STEP 3 OF THE MODIFIED JARVIS ALGORITHM
C     EVALUATE AND STORE ANGLES FROM THE FIRST HULL POINT
C     M TO THE OTHER POINTS IN S.  SUBROUTINE EVALJ PER-
C     FORMS THE EVALUATION STORING THE ANGLES IN ARRAY
C     ANGLE.
C   *                                                     *
C   *                                                     *
C   *******************************************************
C
C
```

```
      CALL EVALJ(MX,MY)
      IF(VN .LE. 1) GO TO 100
C
C
C
C
C  ******************************************************
C  *                                                    *
C  *                                                    *
C    THE FOLLOWING BLOCK OF CODE PERFORMS STEP 4 OF THE
C    MODIFIED JARVIS ALGORITHM.  SUBROUTINE MINANG FINDS
C    THE MINIMUM ANGLE ELEMENT AND DELETES IT FROM S.
C    THE DELETED ELEMENT IS PLACED INTO ARRAY H (POS-
C    ANG HOLDS THE POSITION OF DELETED ELEMENT).
C  *                                                    *
C  *                                                    *
C  ******************************************************
C
C
      CALL MINANG(POSANG)
      NHE = NHE + 1
      H(NHE) = POSANG
C
C
C
C
C  ******************************************************
C  *                                                    *
C  *                                                    *
C    STEP 5 IS PERFORMED IN THE FOLLOWING BLOCK OF CODE.
C    DELETE ANY ELEMENT IN S HAVING AN ANGLE EQUAL TO THE
C    LAST FOUND HULL ELEMENT'S ANGLE.
C  *                                                    *
C  *                                                    *
C  ******************************************************
C
C
      VNTEMP = VN
      DO 60 I = 1,VNTEMP
      IF(ANGLE(PIQ) .NE. LANG) GO TO 50
      PTF(PTB(PIQ)) = PTF(PIQ)
      PTB(PTF(PIQ)) = PTB(PIQ)
      VN = VN - 1
   50 PIQ = PTF(PIQ)
   60 CONTINUE
      IF(VN .LE. 1) GO TO 100
C
C
C
C
C  ******************************************************
C  *                                                    *
C  *                                                    *
C    USE SUBROUTINE NEXTPT TO FIND THE NEXT CONVEX HULL
C    POINT AS OUTLINED IN STEP 6 OF THE MODIFIED JARVIS
```

```
C     ALGORITHM.  ONCE AGAIN USE POSANG TO ADD 'NEXT
C     POINT' TO H.
C   *                                                   *
C   *                                                   *
C   *****************************************************
C
C
   70 CALL NEXTPT(POSANG)
      NHE = NHE + 1
      H(NHE) = POSANG
C
C
C
C
C   *****************************************************
C   *                                                   *
C   *                                                   *
C     PERFORM STEP 7 OF THE MODIFIED JARVIS ALGORITHM.
C     THAT IS, ELIMINATE ANY ELEMENT OF S WHOSE ANGLE IS
C     LESS THAN THE LAST FOUND CONVEX HULL POINT.  BY RE-
C     TURNING TO SUBROUTINE NEXTPT WE ARE MAKING THE AL-
C     GORITHM ITERATIVE.
C   *                                                   *
C   *                                                   *
C   *****************************************************
C
C
      VNTEMP = VN
      DO 90 I = 1,VNTEMP
      IF(ANGLE(PIQ) .GT. LANG) GO TO 80
      PTF(PTB(PIQ)) = PTF(PIQ)
      PTB(PTF(PIQ)) = PTB(PIQ)
      VN = VN - 1
   80 PIQ = PTF(PIQ)
   90 CONTINUE
      IF(VN .GT. 1) GO TO 70
C
C
C
C
C   *****************************************************
C   *                                                   *
C   *                                                   *
C     IS THERE ONE POINT LEFT IN S?  IF SO, PLACE THAT
C     POINT INTO H.
C   *                                                   *
C   *                                                   *
C   *****************************************************
C
C
  100 IF(VN .EQ. 0) RETURN
      NHE = NHE + 1
      H(NHE) = PIQ
      RETURN
      END
```

```
C
C
C
C
C
C
C
C
C ...............................................................
C
C     SUBROUTINE BOTTOM
C
C     PURPOSE
C        TO FIND THE BOTTOMMOST POINT M IN A TWO-DIMENSIONAL
C        EUCLIDEAN POINT SET S.  IF THERE IS MORE THAN ONE
C        SUCH POINT, THE LEFTMOST IS CHOOSEN.  THUS ALL POINTS
C        IN S ARE EITHER ABOVE OR TO THE RIGHT OF M.
C
C     USAGE
C        CALL BOTTOM(MX,MY,POSIT)
C
C     DESCRIPTION OF PARAMETERS
C        I      - INTEGER; DO LOOP COUNTER
C        MX     - REAL; X COORDINATE OF THE BOTTOMMOST POINT.
C        MY     - REAL; Y COORDINATE OF THE BOTTOMMOST POINT.
C        POSIT  - INTEGER; HOLDS POSITION OF ARRAY ELEMENT THAT
C                 CURRENTLY IS THE BOTTOMMOST POINT.
C        -------
C        NOTE THAT MX, MY, AND POSIT ARE RETURNED TO THE CALL-
C        ING PROCEDURE.  ALSO, PARAMETERS X, Y, AND N ARE COMMON
C        WITH SUBROUTINE JARVIS.
C
C     SUBROUTINE AND FUNCTION SUBPROGRAMS REQUIRED
C        NONE
C
C     METHOD
C        SEE COMMENT BOXES INSIDE OF THE MAIN BODY OF THE SUB-
C        ROUTINE FOR AN EXPLANATION OF SUBSEQUENT CODE.
C
C     PROGRAMMER
C        MARK NOGA
C
C     DATE
C        FEB. 7, 1979
C
C ...............................................................
C
C
C
C
      SUBROUTINE BOTTOM(MX,MY,POSIT)
      INTEGER I,N,POSIT
      REAL MX,MY,X(7500),Y(7500)
      COMMON/ALPHA/X,Y
      COMMON/BETA/N
```

```
C
C
C
C
C    ******************************************************
C    *                                                    *
C    *                                                    *
C      INITIALIZE, ASSUME THAT THE BOTTOMMOST POINT M IS
C      THE FIRST ELEMENT OF THE ARRAY.  POSIT IS USED TO
C      HOLD THE POSITION OF THE ARRAY ELEMENT THAT CUR-
C      RENTLY IS THE BOTTOMMOST POINT.
C    *                                                    *
C    *                                                    *
C    ******************************************************
C
C
      POSIT = 1
C
C
C
C
C    ******************************************************
C    *                                                    *
C    *                                                    *
C      IF Y(I) IS GREATER THAN THE CURRENT VALUE OF
C      Y(POSIT) THEN IT CANNOT BE THE BOTTOMMMOST POINT.
C      HOWEVER, IF Y(I) IS LESS THAN OR EQUAL TO Y(POSIT)
C      THEN POSIT MAY HAVE TO BE REASSIGNED IF THE NEW
C      PROSPECTIVE M IS BELOW AND/OR TO THE LEFT OF THE
C      CURRENT M.
C    *                                                    *
C    *                                                    *
C    ******************************************************
C
C
      DO 30 I = 2,N
      IF(Y(I) .GT. Y(POSIT)) GO TO 30
      IF(Y(I) .NE. Y(POSIT)) GO TO 20
      IF(X(I) .GE. X(POSIT)) GO TO 30
   20 POSIT = I
   30 CONTINUE
C
C
C
C
C    ******************************************************
C    *                                                    *
C    *                                                    *
C      STORE THE COORDINATES OF THE BOTTOMMOST POINT IN
C      VARIABLES MX AND MY.
C    *                                                    *
C    *                                                    *
C    ******************************************************
C
C
```

```
      MX = X(POSIT)
      MY = Y(POSIT)
      RETURN
      END
C
C
C
C
C
C
C
C
C   ...................................................................
C
C     SUBROUTINE EVALJ
C
C     PURPOSE
C        TO PERFORM STEP3 OF THE MODIFIED JARVIS ALGORITHM (SEE SUB-
C        ROUTINE JARVIS FOR INFORMATION).
C
C     USAGE
C        CALL EVALJ(MX,MY)
C
C     DESCRIPTION OF PARAMETERS
C        DIFFX,DIFFY - REAL; THE SUBTRACTED X AND Y DIFFERENCES
C                      BETWEEN SOME S(I) AND M.
C        I           - INTEGER; DO LOOP PARAMETER.
C        MX,MY       - REAL; X AND Y COORDINATES OF THE BOTTOMMOST
C                      POINT (SEE SUBROUTINE BOTTOM).
C        -----------
C        PARAMETERS COMMON WITH SUBROUTINE JARVIS: ANGLE, PIQ, PTB,
C        PTF, VN, X, Y.  SEE 'DESCRIPTION OF PARAMETERS' SECTION IN
C        SUBROUTINE JARVIS.
C
C     SUBROUTINE AND FUNCTION SUBPROGRAMS REQUIRED
C        SIGN (IBM INLINE FUNCTION)
C
C     METHOD
C        SEE COMMENT BOXES INSIDE OF THE MAIN BODY OF THE SUBROUTINE
C        FOR AN EXPLANATION OF SUBSEQUENT CODE.
C
C     PROGRAMMER
C        MARK NOGA
C
C     DATE
C        FEB. 8, 1979
C
C   ...................................................................
C
C
C
C
      SUBROUTINE EVALJ(MX,MY)
      INTEGER I,PIQ,PTB(7500),PTF(7500),VN,VNTEMP
      REAL ANGLE(7500),DIFFX,DIFFY,MX,MY,X(7500),Y(7500)
      COMMON/ALPHA/X,Y
      COMMON/GAMMA/ANGLE
      COMMON/DELTA/PIQ,PTB,PTF,VN
C
C
C
C
C   *****************************************************
C   *                                                   *
C   *                                                   *
C     IF A POINT COINCIDENT WITH M IS FOUND ELIMINATE IT
C     FROM S (USE THE LINKED LIST IMPLEMENTATION).
C   *                                                   *
C   *                                                   *
C   *****************************************************
C
C
      VNTEMP = VN
      DO 20 I = 1,VNTEMP
      IF((X(PIQ) .NE. MX) .OR. (Y(PIQ) .NE. MY)) GO TO 10
      PTF(PTB(PIQ)) = PTF(PIQ)
      PTB(PTF(PIQ)) = PTB(PIQ)
      PIQ = PTF(PIQ)
      VN = VN - 1
      GO TO 20
C
C
C
C
C   *****************************************************
C   *                                                   *
C   *                                                   *
C     COMPUTE ANGLE(I), FIND NEXT POINT, AND CONTINUE ON
C     FINDING ANGLE(I+1), ETC.
C   *                                                   *
C   *                                                   *
C   *****************************************************
C
C
   10 DIFFX = X(PIQ) - MX
      DIFFY = Y(PIQ) - MY
      ANGLE(PIQ) = -DIFFX/(ABS(DIFFX) + DIFFY)
      PIQ = PTF(PIQ)
   20 CONTINUE
      RETURN
      END
C
C
C
C
C
C
C
C
C   ...............................................................
```

```
C
C     SUBROUTINE MINANG
C
C     PURPOSE
C        TO PERFORM STEP 4 OF THE MODIFIED JARVIS ALGORITHM (SEE SUB-
C        ROUTINE JARVIS FOR DESCRIPTION OF ALGORITHM).
C
C     USAGE
C        CALL MINANG(POSANG)
C
C     DESCRIPTION OF PARAMETERS
C        I         INTEGER; DO LOOP PARAMETER.
C        POSANG -  INTEGER; HOLDS THE POSITION OF THE MINIMUM ANGLE
C                  ELEMENT (RETURNED TO CALLING PROCEDURE).
C                  TATION IN SUBROUTINE JARVIS).
C        --------
C        PARAMETERS COMMON WITH SUBROUTINE JARVIS: ANGLE, LANG, PIQ,
C        PTB, PTF, VN, X, XORG, Y, AND YORG.
C
C     SUBROUTINE AND FUNCTION SUBPROGRAMS REQUIRED
C        NONE
C
C     METHOD
C        INTERNAL DOCUMENTATION GIVES BASICS OF ALGORITHM, CONSULT
C        SUBROUTINE JARVIS FOR A MORE GLOBAL VIEW.
C
C     PROGRAMMER
C        MARK NOGA
C
C     DATE
C        FEB. 7, 1979
C
C  ...................................................................
C
C
C
C
      SUBROUTINE MINANG(POSANG)
      INTEGER I,LP,PIQ,POSANG,PTB(7500),PTF(7500),VN
      REAL ANGLE(7500),X(7500),XORG,Y(7500),YORG
      COMMON/ALPHA/X,Y
      COMMON/GAMMA/ANGLE
      COMMON/DELTA/PIQ,PTB,PTF,VN
      COMMON/ZETA/LANG,XORG,YORG
C
C
C
C
C  *******************************************************
C  *                                                     *
C  *                                                     *
C    INITIALIZE; ASSUME THAT THE MINIMUM ANGLE IS ASSOC-
C    IATED WITH ELEMENT PIQ.
C  *                                                     *
C  *                                                     *
```

```
C  *******************************************************
C
C
      POSANG = PIQ
      PIQ = PTF(PIQ)
C
C
C
C
C  *******************************************************
C  *                                                     *
C  *                                                     *
C    SEARCH THE LINKED LIST FINDING THE MINIMUM ANGLE.
C  *                                                     *
C  *                                                     *
C  *******************************************************
C
C
      DO 20 I = 2,VN
      IF(ANGLE(PIQ) .GT. ANGLE(POSANG)) GO TO 10
      IF(ANGLE(PIQ) .LT. ANGLE(POSANG)) GO TO 5
      IF(Y(PIQ) .LE. Y(POSANG)) GO TO 10
    5 POSANG = PIQ
   10 PIQ = PTF(PIQ)
   20 CONTINUE
C
C
C
C
C  *******************************************************
C  *                                                     *
C  *                                                     *
C    STORE THE MINIMUM ANGLE IN VARIABLE LANG.  ALSO,
C    STORE THE X AND Y COORDINATES OF THE MINIMUM ANGLE
C    ELEMENT.
C  *                                                     *
C  *                                                     *
C  *******************************************************
C
C
      LANG = ANGLE(POSANG)
      XORG = X(POSANG)
      YORG = Y(POSANG)
C
C
C
C
C  *******************************************************
C  *                                                     *
C  *                                                     *
C    DELETE THE MINIMUM ANGLE ELEMENT FROM S USING THE
C    LINKED LIST IMPLEMENTATION (PTF AND PTB).  PIQ MUST
C    BE SET AT THE POSITION OF AN ELEMENT THAT STILL
C    REMAINS IN S.
C  *                                                     *
```

```
C  *                                                       *
C  *********************************************************
C
C
      PTF(PTB(POSANG)) = PTF(POSANG)
      PTB(PTF(POSANG)) = PTB(POSANG)
      PIQ = PTF(POSANG)
      VN = VN - 1
      RETURN
      END
C
C
C
C
C
C
C
C
C  ..................................................................
C
C     SUBROUTINE NEXTPT
C
C     PURPOSE
C        TO FIND THE NEXT POINT ON THE CONVEX HULL FOR SUBROUTINE
C        JARVIS.
C
C     USAGE
C        CALL NEXTPT(POSANG)
C
C     DESCRIPTION OF PARAMETERS
C        DISTX  - REAL; DISTANCE X(PIQ) IS FROM THE ORGIN XORG.  USED
C                 TO FIND THE QUADRANT NUMBER OF X(PIQ).
C        DISTY  - REAL; SAME AS DISTX EXCEPT FOR Y(PIQ).
C        PIQRAT - REAL; HOLDS RATIO OF ELEMENT PIQ.
C        POSANG - INTEGER; THE POSITION OF THE MINIMUM ANGLE ELEMENT
C                 (RETURNED TO CALLING PROCEDURE).
C        QPIQ   - INTEGER; QUADRANT NUMBER OF ELEMENT PIQ.
C        QUAD   - INTEGER; QUADRANT NUMBER OF THE MINIMUM ANGLE
C                 ELEMENT.
C        RATIO  - REAL; RATIO OF THE MINIMUM ANGLE ELEMENT.
C        SGNX   - INTEGER; MATHEMATICAL SIGN OF DISTX.
C        SGNY   - INTEGER; MATHEMATICAL SIGN OF DISTY.
C        ------
C        PARAMETERS IF COMMON WITH SUBROUTINE JARVIS:  ANGLE, LANG,
C        PIQ, PTB, PTF, VN, X, XORG, Y, YORG.
C
C     SUBROUTINE AND FUNCTION SUBPROGRAMS REQUIRED
C        SYSTEM FUNCTION SIGN
C
C     METHOD
C        THE METHOD IMPLEMENTED BELOW IS VERY SIMILAR TO THE POINT
C        ELIMINATION METHOD OUTLINED BY JARVIS (SEE SUBROUTINE JARVIS
C        FOR REFERENCE AND FURTHER EXPLANATION).  THE DIFFERENCES ARE
C        AS FOLLOWS:
C
```

```
C           1.  THERE ARE NO SPECIAL CASES FOR POINTS FALLING ON THE
C               VERTICAL OR HORIZONTAL.  THUS QUADRANT 1 FALLS BETWEEN
C               0 AND 90 DEGREES INCLUSIVE, QUADRANT 2 BETWEEN 90 AND
C               180, 180 INCLUSIVE, QUADRANT 3 BETWEEN 180 AND 270,
C               270 INCLUSIVE, AND QUADRANT 4 BETWEEN 270 AND 360.
C           2.  FOR QUADRANTS 1 AND 3 THE RATIO OF SIDES IS DETERMINED
C               BY THE FORMULA:  -DISTX/(DISTX + DISTY).  FOR QUAD-
C               RANTS 2 AND 4 THE FORMULA IS:  DISTY/(DISTX - DISTY).
C               THESE FORMULAE WERE DERIVED TO PREVENT COMPUTATIONAL
C               OVERFLOW.  THE FORMULAS PROPOSED BY JARVIS ARE DE-
C               FICIENT WITH RESPECT TO THE OVERFLOW PROBLEM.
C
C     PROGRAMMER
C        MARK NOGA
C
C     DATE
C        FEB. 8, 1979
C
C   ..................................................................
C
C
C
C
      SUBROUTINE NEXTPT(POSANG)
      INTEGER I,PIQ,POSANG,PTB(7500),PTF(7500),QPIQ,QUAD,SGNX,SGNY,VN
      REAL ANGLE(7500),DISTX,DISTY,LANG,PIQRAT,RATIO,X(7500),
     *     XORG,Y(7500),YORG
      COMMON/ALPHA/X,Y
      COMMON/GAMMA/ANGLE
      COMMON/DELTA/PIQ,PTB,PTF,VN
      COMMON/ZETA/LANG,XORG,YORG
C
C
C
C
C   ********************************************************
C   *                                                      *
C   *                                                      *
C     INITIALIZE; ASSUME TO BEGIN WITH THAT 'NEXT POINT' IS
C     AT POSITION PIQ.  NOTE:  THE QUADRANT NUMBER (QUAD)
C     AND RATIO ARE COMPUTED HERE IN CASE THIS POINT IS
C     DISCARDED IN FAVOR OF ANOTHER IN THE LOOP OF THE
C     NEXT STEP.
C   *                                                      *
C   *                                                      *
C   ********************************************************
C
C
      POSANG = PIQ
      DISTX = X(PIQ) - XORG
      DISTY = Y(PIQ) - YORG
      SGNX = SIGN(1.,DISTX)
      SGNY = SIGN(1.,DISTY)
      IF(SGNY .LT. 0) GO TO 20
      IF(SGNX .LT. 0) GO TO 10
      QUAD = 1
      RATIO = -DISTX/(DISTX + DISTY)
      GO TO 40
   10 QUAD = 2
      RATIO = DISTY/(DISTX - DISTY)
      GO TO 40
   20 IF(SGNX .GT. 0) GO TO 30
      QUAD = 3
      RATIO = -DISTX/(DISTX + DISTY)
      GO TO 40
   30 QUAD = 4
      RATIO = DISTY/(DISTX - DISTY)
   40 PIQ = PTF(PIQ)
C
C
C
C
C    ********************************************************
C    *                                                      *
C    *                                                      *
C      SEARCH THE REMAINING ELEMENTS OF THE LINKED LIST.
C      IF THE POINT IN QUESTION PIQ HAS A LOWER QUAD-
C      RANT NUMBER REASSIGN POSANG, QUAD, AND RATIO.
C      IF PIQ HAS THE SAME QUADRANT NUMBER THEN A RATIO
C      CHECK MUST BE PERFORMED.  THE RESULT OF THE RATIO
C      CHECK DETERMINES IF PIQ COULD BE 'NEXT POINT.'  UPON
C      COMPLETION OF THE LOOP THE POSITION OF 'NEXT POINT'
C      SHOULD BE CONTAINED IN VARIABLE POSANG.
C    *                                                      *
C    *                                                      *
C    ********************************************************
C
C
      DO 130 I = 2,VN
      DISTX = X(PIQ) - XORG
      DISTY = Y(PIQ) - YORG
      SGNX = SIGN(1.,DISTX)
      SGNY = SIGN(1.,DISTY)
      IF(SGNY .LT. 0) GO TO 60
      IF(SGNX .LT. 0) GO TO 50
      QPIQ = 1
      GO TO 80
   50 QPIQ = 2
      GO TO 80
   60 IF(SGNX .GT. 0) GO TO 70
      QPIQ = 3
      GO TO 80
   70 QPIQ = 4
   80 IF(QPIQ .GT. QUAD) GO TO 120
      IF((QPIQ .EQ. 2) .OR. (QPIQ .EQ. 4)) GO TO 90
      PIQRAT = -DISTX/(DISTX + DISTY)
      GO TO 100
   90 PIQRAT = DISTY/(DISTX - DISTY)
  100 IF(QPIQ .LT. QUAD) GO TO 110
      IF(PIQRAT .GT. RATIO) GO TO 120
      IF(PIQRAT .LT. RATIO) GO TO 110
      IF((ABS(Y(POSANG) - YORG) + ABS(X(POSANG) - XORG)) .GT.
     *   (ABS(DISTY) + ABS(DISTX))) GO TO 120
  110 POSANG = PIQ
      QUAD = QPIQ
      RATIO = PIQRAT
  120 PIQ = PTF(PIQ)
  130 CONTINUE
C
C
C
C
C    *******************************************************
C    *                                                     *
C    *                                                     *
C      STORE THE ANGLE ASSOCIATED WITH THE NEXT POINT IN
C      VARIABLE LANG.  ALSO, REASSIGN THE REFERENCE ORGIN
C      VARIABLES XORG AND YORG.
C    *                                                     *
C    *                                                     *
C    *******************************************************
C
C
      LANG = ANGLE(POSANG)
      XORG = X(POSANG)
      YORG = Y(POSANG)
C
C
C
C
C    *******************************************************
C    *                                                     *
C    *                                                     *
C      DELETE THE ELEMENT FOUND ABOVE (POSANG) FROM S.
C    *                                                     *
C    *                                                     *
C    *******************************************************
C
C
      PTF(PTB(POSANG)) = PTF(POSANG)
      PTB(PTF(POSANG)) = PTB(POSANG)
      PIQ = PTF(POSANG)
      VN = VN - 1
      RETURN
      END
```

REFERENCES

[Adamowicz(72)] M. Adamowicz and A. Albano, *A two-stage solution of the cutting stock problem,* Proc. IFIP Cong. 71, North Holland, Amsterdam (1972), 1086-1091.

[Aho(74)] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley (1974).

[Akl(78a)] S.G. Akl, *An Analysis of Various Aspects of the Traveling Salesman Problem,* Ph.D. Thesis, School of Computer Science, McGill University, Montreal, Quebec, Canada (March 1978).

[Akl(78b)] S.G. Akl and G.T. Toussaint, *A fast convex hull algorithm,* Info. Proc. Lett. 7, no. 5 (1978), 219-222.

[Akl(79)] S.G. Akl, *Personal Communication (1979).*

[Allison(81)] D.C.S. Allison and M.T. Noga, *Selection by distributive partitioning,* Info. Proc. Lett. 11, no. 1 (1980), 7-8.

[Allison(82)] D.C.S. Allison and M.T. Noga, *Usort: an efficient hybrid of distributive partitioning sort,* B.I.T. 22, (1982), 136-139.

[Anderson(78)] K.R. Anderson, *A reevaluation of an efficient algorithm for determining the convex hull of a finite planar set,* Info. Proc. Lett. 7, no. 1 (1978), 53-55.

[Avis(81a)] D. Avis and G.T. Toussaint, *An efficient algorithm for decomposing a polygon into star-shaped polygons,* Pattern Recognition 13, no. 6 (1981), 395-398.

[Avis(81b)] D. Avis and G.T. Toussaint, *An optimal algorithm for determining the visibility of a polygon from an edge,* IEEE Trans. on Computers, v. C-30, no. 12 (Dec. 1981), 910-914.

[Baase(78)] S. Baase, *Computer Algorithms: Introduction to Design and Analysis,* Addison-Wesley (1978).

[Beardwood(59)] J. Beardwood, J.H. Halton, and J.M. Hammersley, *The shortest path through many points,* Proc. of the Camb. Phil. Soc. 55, (1959), 299-327.

[Bellmore(68)] M. Bellmore and G.L. Nemhauser, *The traveling salesman problem: a survey,* Opns. Res. 16 (1968), 538-558.

[Bentley(77a)] J.L. Bentley and M.I. Shamos, *A problem in multivariate statistics: Algorithm, data structure, and applications.* Proc. 15th Allerton Conf. on Communication, Control, and Computing. (Sept. 1977), 193-201.

[Bentley(77b)] J.L. Bentley, D.F. Stanat, and E.H. Williams Jr., *The complexity of finding fixed-radius near neighbours,* Info. Proc. Lett. 6 (1977), 209-212.

[Bentley(78)] J.L. Bentley and M.I. Shamos, *Divide and conquer for linear expected time,* Info. Proc. Lett. 7, no. 2, (1978), 87-91.

[Bentley(80a)] J.L. Bentley, *Multidimensional divide-and-conquer,* CACM 23, no. 4 (1980), 214-229.

[Bentley(80b)] J.L. Bentley, M.G. Faust, and F.P. Preparata, *Approximation algorithms for convex hulls,* Tech. Rep. CMU-CS-80-109, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, PA (1980).

[Blum(73)] M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest, and R.E. Tarjan, *Time bounds for selection,* J. Comput. System Sci. 7 (1973), 448-461.

[Brown(76)] T. Brown, *Remark on algorithm 489,* TOMS 3, no. 2 (1976), 301-304.

[Bykat(78)] A. Bykat, *Convex hull of a finite set of points in two dimensions,* Info. Proc. Lett. 7, no. 6 (1978), 297-298.

[Chazelle(80)] B. Chazelle, *Computational Geometry and Convexity,* Ph.D. Thesis, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Penn. 15213 (July 1980).

[Chazelle (83)] B. Chazelle, *An improved algorithm for the fixed-radius neighbor problem,* Info. Proc. Lett. 16 (1983), 193-198.

[Desens(69)] R.B. Desens, *Computer processing for display of three-dimensional structures,* Tech. Rep. CFSTI AD-706010, Naval Postgraduate School (Oct. 1969).

[Devai(79)] F. Devai and T. Szendrenyi, *Comments on convex hull of a finite set of points in two dimensions,* Info. Proc. Lett. 9, no. 3 (1979), 141-142.

[Devroye(81)] L.P. Devroye and T. Klincsek, *Average time behavior of distributive sorting algorithms,* Computing 26, no. 1 (1981), 1-7.

[Dobkin(79)] D.P. Dobkin and L. Snyder, *On a general method for maximizing and minimizing among certain geometric problems,* Proc. 20th Annual Symposium of Computer Science, San Juan, Puerto Rico (October 1979), 9-17.

[Dobosiewicz(78)] *Sorting by distributive partitioning,* Info. Proc. Lett. 7, no. 1 (1978), 1-6.

[Eastman(71)] C.M. Eastman, *Heuristic algorithms for automated space planning,* Proc. 2nd International Conf. on Artificial Intelligence, British Computer Society, 29 Portland Place, London (1971), 27-39.

[Eddy(77a)] W.F. Eddy, *A new convex hull algorithm for planar sets,* ACM TOMS 3, no. 4 (1977), 398-403.

[Eddy(77b)] W.F. Eddy, *Algorithm 523 CONVEX, a new convex hull algorithm for planar sets,* Collected Algorithms from ACM, (1977), 523P1-523P6.

[Eilon(71)] S. Eilon, C.D.T. Watson, and N. Christofides, *Distribution Management,* Griffin, London (1971).

[Ehrich(84)] R.W. Ehrich, *Personal Communication,* 1984.

[El Gindy(81)] H. El Gindy and D. Avis, *A linear algorithm for computing the visibility polygon from a point,* Journal of Algorithms 2 (1981), 186-197.

[Floyd (75)] R.W. Floyd and R.L. Rivest, *Expected time bounds for selection,* CACM 18, no. 13 (1975), 165-172.

[Fowler(81)] *Optimal packing and covering in the plane are NP-complete,* Info. Proc. Lett. 12, no. 3 (1981), 133-137.

[Freeman(75)] H. Freeman and R. Shapira, *Determining the minimum area encasing rectangle for an arbitrary closed curve,* Comm. ACM 18, no. 7 (July 1975), 409-413.

[Galimberti(69)] R. Galimberti and U. Montanari, *An algorithm for hidden-line elimination,* CACM 12, no. 4 (1969).

[Garey(76)] M. Garey, R.L. Graham, D. Johnson, *Some NP-complete geometric problems,* Proc. 8th SIGACT Symp. on the Theory of Computing (1976), 10-22.

[Garey(78)] M. Garey, D. Johnson, F. Preparata and R. Tarjan, *Triangulating a simple polygon,* Info. Proc. Lett. 7 (1978), 175-179.

[Gavett(65)] J.W. Gavett, *Three heuristic rules for sequencing jobs to a single production facility,* Management Science 11, no. 8 (1965), B166-B176.

[Gilbert(65)] E.N. Gilbert, *Random minimal trees,* Journal of SIAM 13, no. 2 (1965), 376-387.

[Golden(80)] B. Golden, L. Bodin, T. Doyle, and W. Stewart, Jr., *Approximate traveling salesman algorithms,* Opns. Res. 28 (1980), 694-711.

[Graham(72)] R.L. Graham, *An efficient algorithm for determining the convex hull of a finite planar set,* Info. Proc. Lett 1, no. 1 (1972), 132-133.

[Haims(70)] M.J. Haims and H. Freeman, *A multistage solution of the template-layout problem,* IEEE Trans. Syst. Science and Cybernetics SSC-6, no. 2 (Apr. 1970), 145-151.

[Haralick(77)] R.M. Haralick and L.G. Shapiro, *Decomposition of polygonal shapes by clustering,* Proc. IEEE Computer Society Conference on Pattern Recognition and Image Processing, Troy, N.Y. (June 1977), 183-190.

[Hoare(61a)] C.A.R. Hoare, *Quicksort (Algorithm 64),* CACM 4, no. 7 (1961), 321.

[Hoare(61b)] C.A.R. Hoare, *Find (Algorithm 65),* CACM 4, no. 7 (1961), 321-322.

[Hofstadter(79)] D.R. Hofstadter, *Godel, Escher, Bach: An Eternal Golden Braid,* Basic Books (1979).

[Horowitz(76)] E. Horowitz and S. Sahni, *Fundamentals of Data Structures,* Computer Science Press (1976).

[Huits(79)] M. Huits and V. Kumar, *The practical significance of distributive partitioning sort,* Info. Proc. Lett. 8, no. 4 (1979), 168-169.

[IMSL(80)] *International Mathematical and Statistics Library,* Edition 8, (1980).

[Jarvis(73)] R.A. Jarvis, *On the identification of the convex hull of a finite set of points in the plane,* Info. Proc. Lett. 2, no. 1 (1973), 18-21.

[Karp(72)] R.M. Karp, *Reducibility among combinatorial problems,* in Complexity of Computer Computations, R.E. Miller and J.W. Thatcher, Eds., Plenum Press, New York (1972), 85-104.

[Kowalik(81)] J.S. Kowalik and Y.B. Yoo, *Implementing a distributive sort program,* Journal of Information and Optimization Sciences 2, no. 1 (1981), 28-33.

[Knuth(74)] D.E. Knuth, *Structured programming with go to statements,* in Current Trends In Programming Methodology, Vol. I, Raymond T. Yeh, ed., (1977).

[Knuth(76)] D.E. Knuth, *Big omicron and big omega and big theta,* SIGACT News 8, no. 2 (1976).

[Krolak(71)] P. Krolak, W. Felts, and G. Marble, *A man-machine approach toward solving the traveling salesman problem,* CACM 14 (1971), 327-334.

[Lawson(77)] C.L. Lawson, *Software for C^1 surface interpolation,* Tech. Rep. 77-30, Jet Propulsion Laboratory (1977).

[Lee(76)] D.T. Lee and F.P. Preparata, *Location of a point in a planar subdivision and its applications,* Eighth Annual ACM SIGACT Symposium (May, 1976), 231-235.

[Lee(79a)] D.T. Lee and F.P. Preparata, *An optimal algorithm for finding the kernel of a polygon,* JACM 26, no. 3 (1979), 415-421.

[Lee(79b)] D.T. Lee and B.J. Schacter, *Two algorithms for constructing a Delaunay triangulation,* Tech. Rep. 79ASD007, General Electric, Daytona Beach (July 1979).

[Lee(80)] D.T. Lee, *On finding the convex hull of a simple polygon,* Tech. Rep. no. 80-03-FC-01, Dept. of Elec. Engr and Computer Science, Northwestern University (1980).

[Lin(65)] S. Lin, *Computer solutions of the traveling salesman problem,* Bell System Technical Journal 44, (1965), 2245-2269.

[Lipton(77)] R.J. Lipton and R.E. Tarjan, *Applications of a planar separator theorem,* Eighteenth Annual IEEE Symposium on Foundations of Computer Science (Oct., 1977), 162-170.

[Matula(80)] D.W. Matula and R.R. Sokal, *Properties of Gabriel graphs relevant to geographic variation research and the clustering of points in the plane,* Geographical Analysis 12 (July 1980), 205-222.

[Meijer(80)] H. Meijer and S.G. Akl, *The design and analysis of a new hybrid sorting algorithm,* Info. Proc. Lett. 10, no. 4-5 (1980), 213-218.

[Meisel(72)] W.S. Meisel, *Computer-Oriented Approaches to Pattern Recognition,* Academic Press (1972).

[Newman(73)] W.M. Newman and R.F. Sproull, *Principles of Interactive Computer Graphics,* McGraw-Hill (1973).

[Noga(81)] M.T. Noga, *Convex Hull Algorithms,* M.S. Thesis, Dept. of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA, (1981).

[Ottmann(84)] T. Ottmann and D. Wood, *Dynamical sets of points,* Computer Vision, Graphics, and Image Processing (to appear).

[Penny(72)] D.E. Penny, *Perspectives in Mathematics,* Benjamin (1972).

[Preparata (77)] F.P. Preparata and S.J. Hong, *Convex hulls of finite planar and spatial sets of points,* CACM 20, no. 2 (1977), 87-93.

[Preparata (79)] F.P. Preparata, *An optimal real-time algorithm for planar convex hulls,* CACM 22, no. 7 (1979), 402-405.

[Preparata(81)] F.P. Preparata and K.J. Supowit, *Testing a simple polygon for monotonicity,* Info. Proc. Lett. 12, no. 4 (1981), 161-164.

[Raynaud(70)] H. Raynaud, *Sur l'enveloppe convex des nuages des points aleatoires dans R.* Appl. Prob. 7, (1970), 35-48.

[Reingold(77)] E.M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial Algorithms: Theory and Practice,* Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632.

[Renyi(68)] A. Renyi and R. Sulanke, *Zufallige konvexe polygone in einem ringgebeit,* Z. Wahrscheinlichkeits 9, (1968), 146-147.

[Roach(83)] J.W. Roach, *Personal Communication (1983).*

[Rosenkrantz(77)] D.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis, *An analysis of several heuristics for the traveling salesman problem,* SIAM Journal on Computing 6, no. 3 (1977), 563-581.

[Schwartz(81)] J.T. Schwartz, *Finding the minimum distance between two convex polygons,* Info. Proc. Lett. 13, no. 4-5 (1981), 168-170.

[Schechter(82)] B. Schechter, *Ronald Graham: The peripatetic number juggler,* Discover 3, no. 10, (Oct. 1982), 44-52.

[Schonhage(76)] A. Schonhage, M. Paterson, and N. Pippenger, *Finding the median,* J. Comput. System Sci. 13 (1976), 184-189.

[Sedgewick(78)] R. Sedgewick, *Implementing Quicksort programs,* CACM 21, no. 10 (1978), 847-856.

[Shamos(75a)] *Geometric complexity,* Proc. 7th ACM Symposium on the Theory of Computing (May 1975), 224-233.

[Shamos(75b)] M.I. Shamos and D. Hoey, *Closest-point problems,* Sixteenth Annual IEEE Symposium on Foundations of Computer Science, October 1975, 151-162.

[Shamos(76)] M.I. Shamos and D. Hoey, *Geometric Intersection Problems,* Seventeenth Annual IEEE Symposium on Foundations of Computer Science (Oct. 1976), 208-215.

[Shamos(77)] M.I. Shamos, *Problems in Computational Geometry,* Carnegie-Mellon University (1977).

[Shamos(78)] M.I. Shamos, *Computational Geometry,* Ph.D. Thesis, Dept. of Computer Science, Yale University, New Haven, CT, (May 1978).

[Silverman(81)] B.W. Silverman and D.M. Titterington, *Minimum covering ellipses,* SIAM J. Sci. Stat. Comput. 1, no. 4 (Dec. 1980), 401-409.

[Sklansky(72)] J. Sklansky, *Measuring concavity on a rectangular mosaic,* IEEE Trans. on Computers C-21, no. 12 (1972), 1355-1362.

[Sutherland(66)] I.E. Sutherland, *Ten unsolved problems in computer graphics,* Datamation 12, no. 5 (May 1966).

[Toussaint(78a)] G.T. Toussaint, S.G. Akl, and L.P. Devroye, *Efficient convex hull algorithms for points in two and more dimensions,* Tech. Rep. No. SOCS 78.5, School of Computer Science, McGill Univ., (May 1978).

[Toussaint(78b)] G.T. Toussaint, *The convex hull as a tool in pattern recognition,* Proc. AFOSR Workshop in Communication Theory and Applications, Provincetown, Mass. (Sept. 1978), 43-46.

[Toussaint(80a)] G.T. Toussaint, *Pattern recognition and geometrical complexity,* Proc. 5th International Conference on Pattern Recognition, Miami Beach (Dec. 1980), 1324-1347.

[Toussaint(80b)] G.T. Toussaint, *The relative neighborhood graph of a finite planar set,* Pattern Recognition 12, no. 4 (1980).

[Toussaint(82)] G.T. Toussaint, *Computational geometric problems in pattern recognition,* in Pattern Recognition Theory and Applications, J. Kittler, K.S. Fu, and L.F. Pau (eds.), Reidel Publishing Co. (1982), 73-91.

[van der Nat(80)] M. van der Nat, *A fast sorting algorithm, a hybrid of distributive and merge sorting,* Info. Proc. Lett. 10, no. 3 (1980), 213-218.

[Warnock(69)] J.E. Warnock, *A hidden-surface algorithm for computer generated half-tone pictures,* Technical Rep. TR 4-15, Computer Science Dept., Univ. of Utah (1969).

[Watkins(70)] G.S. Watkins, *A real-time visible surface algorithm,* Tech. Rep. UTECH-CSc-70-101, Computer Science Dept., Univ. of Utah (Jun. 1970).

[Weide(78)] B. Weide, *Statistical Methods In Algorithm Design And Analysis,* Ph.D. Thesis, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, PA (Sept. 1978).

[Wilkes(81)] M.V. Wilkes, *The computation lab at Cambridge University: The early years,* Distinguished Visiting Lecture Series, Virginia Polytechnic Institute and State University, Blacksburg, VA (April 1981).

[Wirth(76)] N. Wirth, *Algorithms + Data Structures = Programs,* Prentice-Hall (1976).

[Yaglom(61)] I.M. Yaglom and V.G. Boltyanskii, *Convex Figures,* Holt, Rinehart, and Winston (1961).

[Yao(81)] A.C. Yao, *A lower bound to finding convex hulls,* Journal of ACM 28, no. 4 (1981), 780-787.

FAST GEOMETRIC ALGORITHMS

by

Mark T. Noga

(ABSTRACT)

This thesis addresses a number of important problems which fall within the framework of the new discipline of Computational Geometry. The list of topics covered includes sorting and selection, convex hull algorithms, the L_1 hull, determination of the minimum encasing rectangle of a set of points, the Euclidean and L_1 diameter of a set of points, the metric traveling salesman problem, and finding the superrange of star-shaped and monotone polygons.

The main theme of all our work has been to develop a set of very fast state-of-the-art algorithms which supercede any rivals in terms of speed and ease of implementation. In some cases we have refined existing algorithms; for others we have developed new techniques which add to the present database of fast adaptive geometric algorithms. What emerges is a collection of techniques that is successful at merging modern tools developed in analysis of algorithms with those of classical geometry.