COMPUTATIONAL GEOMETRY

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
In Candidacy for the Degree of
Doctor of Philosophy
by
Michael Ian Shamos
May, 1978

ABSTRACT

COMPUTATIONAL GEOMETRY

Michael Ian Shamos

Yale University, 1978

This thesis is a study of the computational aspects of geometry within the framework of analysis of algorithms. It develops the mathematical techniques that are necessary for the design of efficent algorithms and applies them to a wide variety of theoretical and practical problems. Particular attention is given to proving lower bounds on running time and to analyzing the average-case performance of geometric algorithms. The approach taken is to isolate a computational feature that is common to a large class of problems. It turns out, for example, that determining whether any two of N line segments in the plane overlap is an essential step in many intersection applications. An optimal algorithm for this problem, therefore, becomes an important geometric tool that can be used to build other, more complicated, fast algorithms. This method is employed in a unified attack on the problem of the convex hull, various geometric search problems, finding the intersection of objects and questions invoiving the proximity of points in the plane. What emerges is a modern, coherent discipline that is successful at merging classical geometry with computational complexity. Among the major new results presented are a convex hull algorithm with expected running time that is linear in the number of input points, an O(N log N) algorithm for linear programming in two variables (which is superior to the Simplex method), and an O(N log N) algorithm for constructing a minimum spanning tree on a finite set of points in the plane.

Preface

This thesis is a sprawling study of the interaction between geometry and computing. It is an examination of the issues that arise in solving geometric problems by machine at high speed and the fact that such devices have only recently been built obliges us to consider aspects of geometric computation that simply do not occur in classical mathematics, and new methods are required. Fortunately, Geometry is a highly intuitive subject, and aside from some technical material on analysis of algorithms, the ideas of the thesis are accessible to high-school students. While many a Ph.D. candidate would be unsettled by the idea that his work is so elementary, I consider it a distinct advantage because the thesis argues for simplicity and elegance in the design and construction of algorithms. I setting out to write this volume I had a story to tell. The tale is a long one, full of detours, intrigues and episodes, but it must be told fully. The reader who has the energy to follow will see an ancient gem shine in contemporary light, with none of her sparkle lost to Time.

Acknowledgements

"For this relief, much thanks ..."

- Shakespeare, *Hamlet.*

This thesis describes the results of four years of sustained work. During that time, many people have helped shape it, encouraged, begged, cajoled, dared, nagged, prodded, implored and inspired me to write, and some have sympathetically eased me through the pure physical hell of preparing it for publication. To all, a hearty "thank you". In some cases their contributions were greater than what I would expect out of friendship or scholarly interest, and it is a pleasure for me to acknowledge these individually.

My greatest debt by far is to my wife, Julie. It is customary to thank one's

spouse for "putting up" with long hours, lost weekends, and the like, but mine carefully orchestrated the entire performance! Having learned once in a course on research and development management that the creative scientist is "a curious admixture of the childlike and the mature", she applied this principle to me. For continued sustenance, perpetual inspiration and undying love, this work is for You.

My academic inheritance is from my parents, who fostered my inquisitiveness and instilled in me a passion for the unknown. Dad, nothing can compare with the joy of discovering something that no one else in the world has seen before, then telling them about it. So it was with Computational Geometry.

It was my Grandfather, a civil engineer and our family patriarch, who showed me the beauty and simplicity of Geometry, the elegance and logic of Euclid, before I was old enough to understand. This early indoctrination had its effect, though, and the weighty volume you are holding is the result.

I owe a great debt to John Wheeler, my undergraduate advisor at Princeton, for showing me that I could make discoveries by myself and that nothing more was required than books, pencils, paper and a quiet place to think. His geometric view of physics showed me the tremendous scope and influence of Geometry, and deepened my perception of the subject.

It was not until I came to Yale as a graduate student that I acquired the tools to produce any new geometry. The atmosphere in the new Department of Computer Science was congenial and conducive to research: few courses, lots of machine time, and an opportunity to work closely with the faculty. My thesis committee consisted of David Dobkin, Stan Eisenstat and Martin Schultz. David insisted that I study the foundations of both geometry and computer science before commencing research. (I never learned Galois Theory, but it turned out not to be necessary.) He shaped the thesis by demanding a coherent theory rather than a collection of isolated results. Stan taught me combinatorics, divide-and-conquer, and the need for thoroughness and a command of the literature. He had an uncanny way knowing just what method to apply to a particular problem and many citations in the text testify to his intuition. Martin was the first to suggest that the work I was doing might constitute a Ph.D. thesis. It did, four years later. His encouragement at the early stages was crucial to me.

Many fellow students have made substantial contributions, either by suggesting their own solutions or criticizing mine. Fred Friedman, Rob Schreiber and Andy Sherman were always willing to tackle a geometry problem, and something interesting always came out of our discussions. Dan Hoey was a true collaborator. We worked together, often at the strangest hours, and he provided the key insights that led to the most important results in this thesis. How close can colleagues be unless they have discovered the generalized Voronoi diagram together?

After leaving Yale I found a most hospitable environment at Carnegie-Mellon. Joe Traub saw merit in this work and now Computational Geometry has a home. At CMU Jon Bentley and I are now studying higher-dimensional geometric and statistical problems.

On the financial side, the IBM Corporation was a substantial benefactor. As a graduate student, I was on educational leave from IBM and was also the recipient of the IBM Fellowship at Yale. Bell Laboratories invited me to spend a delightful three weeks with virtually no responsibilities other than the pursuit of geometry. My work has been supported by the Office of Naval Research and by the Advanced Research Projects Agency of the Department of Defense, while benefits from the Veterans Administration allowed me to devote all of my energies to Geometry. My parents were also major contributors to this campaign.

This document was prepared at Carnegie-Mellon using the PUB compiler and the SPACS graphic drawing system. So much of the effort was automated that toward the end I began to feel more like an I/O device than an author. Brian Reid and Mark Sapsford wrote macros and modified PUB as I went along, gently persuading it to accept my demands (150 diagrams). The orthography was carefull checkd by SPELL, a program that detects and corrects spelling errors at high speed. Its knowledge of the outside world, though, is somewhat limited. Seeing one of my references to John Tukey, it innocently suggested that "Turkey" was what I wanted! (I swear.) With such potent debugging tools at his disposal, the author is able to deny all responsibility for mistakes in the final manuscript. Any residual errors are the result of hardware or softw
?ILL MEM REF at user PC 452306
?Job aborted.

Chapter 1
Introduction

"He who would know what geometry is must venture into its depths."
- J.J. Sylvester, *A Probationary Lecture on Geometry.*

1.1. The Beginning

Geometry is a subject that has captured the imagination of Man for at least 2500 years. It is at the very foundation of Art, Architecture, and Mathematics, and plays a central role in a host of other areas. Computer Science, by contrast, is a newcomer among such established fields, and it has not yet had the opportunity to benefit from their richness. By the same token, Geometry, developing as it did long before the invention of computers, is laden with ideas, results, and prescriptions that are not easily translated into the modern setting of Analysis of Algorithms. It is now recognized that solving problems on a computer does not merely involve rewriting known formulas in some programming language, but that significant issues arise in problem representation, data structures, algorithm design, and computational complexity [Aho (74)]. It is no surprise that straightforward transcription of classical results does not necessarily produce the best algorithms. And why should it, since until recently the only computations that were feasible were those that could be performed with pencil and paper? The need for fast algorithms is apparent only within the framework of high-speed computers and large quantities of data. What ancient geometer could have imagined problems involving millions of points? The purpose of this thesis is, therefore, to establish a discipline of "Computational" Geometry by recasting classical results into explicit and efficient algorithmic form.

1.2. Thesis outline

The reader may expect three things from this thesis: the synthesis of a new discipline, an exposition of its methods, and a large number of new results. The new subject is *Computational Geometry*, which is based on the premise, for which ample justification will be given, that classical mathematics needs to be augmented in order for us to be able to solve geometric problems efficiently on a computer. We address specifically the need for efficient algorithms and present and a set of techniques for designing them. Our approach is simplistic, but powerful: it consists of studying a class of geometric problems, isolating their common algorithmic component, *and analyzing that component completely*. For example, we will see that a large number of intersection problems can be solved rapidly if we are able to determine whether any two of N line segments in the plane cross each other. To develop a basic computational tool that will lead to the solution of many problems, we subject the line segment problem to the most exhausting scrutiny, eventually obtaining an optimal algorithm (in Section 5.4). Considerable time is spent in distilling out the fundamental problems, motivating the techniques used to solve them, and in deriving methods for proving lower bounds. Once this is done, the analyzed problem becomes an instrument that is used to prove new results and build other, more complex, algorithms. Each chapter comprises an application of this paradigm to a new area of Computational Geometry.

The Introduction includes a short review of the history of Geometry from an algorithmic standpoint, to determine its strenghths and weaknesses.

Chapter Two is an examination of such basic issues as problem specification, representation of geometric objects, and computational models. All of these arise as we attempt to treat an elementary problem: "Given a plane polygon, is it convex?" The objective is to develop an efficient (and correct!) test for convexity. The reader will see that even so simple-looking a problem has many non-trivial ramifications.

Chapter Three is a study of convex hull algorithms. We present and analyze the

algorithms of Graham and Jarvis, including some expected-time results. We then develop a divide-and-conquer hull algorithm that generalizes to three dimensions, has linear average-case performance, and is optimal in the worst-case sense. The methods used in the expected-time analysis are of considerable interest, since they involve unexpected theorems from stochastic geometry. We show a lower bound on the convex hull problem by demonstrating that any hull algorithm must be able to sort. A section is devoted to some of the many applications of hull-finding, including L_∞ and isotonic regression in statistics. We discuss the problem of finding the farthest pair of points of a finite plane set and are able to produce an O(N log N) algorithm based on finding the convex hull. Analysis shows this algorithm to run in linear expected time for a wide class of input distributions. This and all succeeding chapters conclude with a list of unsolved problems.

Chapter Four, "Inclusion Problems", is an introduction to geometric searching that covers preprocessing methods, time-storage tradeoffs, and applications of binary search. The fundamental geometric query is to determine whether or not a new point lies in a given polygon. The complexity of the preprocessing depends on whether the polygon is convex or not, and on how quickly the query must be answered. Star-shaped polygons are introduced as a special class for which the inclusion question can be answered easily.

Chapter Five is concerned with three central problems concerning the intersection of geometric objects. One is to *form* the intersection, another is to *detect* whether two objects intersect (this is easier, in general), and the third is to construct the common intersection of N objects. We give a linear algorithm for the intersection of two convex polygons and use it as the merge step of a divide-and-conquer algorithm for the intersection of N half-planes. By exploiting the connection between this problem and linear programming in two variables, we are able to show that the Simplex algorithm is not optimal in two dimensions. We present in its place an O(N log N) algorithm for two-variable linear programming whose expected running time is O(N) for certain input distributions. We prove that finding the intersection of two plane N-gons may require quadratic time, while O(N log N) time suffices to determine whether or not they intersect. Applications of the intersection problems to computer graphics and pattern recognition are also discussed.

It is in Chapter Six, "Closest-Point Problems", that genuine synthesis occurs. We

pose eight seemingly unrelated problems, all involving the proximity of points in the plane, and introduce a single geometric structure that solves all of them efficiently. In so doing we make use of techniques from all of the previous chapters. We obtain an O(N log N) algorithm for finding a minimum spanning tree on N points in the plane. Because the minimum spanning tree problem has been studied by many previous investigators and has diverse applications, we consider this algorithm to be the most important single result of the thesis. The two closest points of a set, the nearest neighbor of each point, the smallest circle enclosing the set, and a triangulation of the N points can all be found in O(N log N) time by recourse to this single structure, called the Voronoi diagram. After giving an optimal algorithm for its construction, we show how it can be generalized to solve the k-nearest neighbors problem and answer a number of other important computational questions.

Each chapter contains a list of unsolved problems that suggest avenues for further research. The last chapter indicates several ways in which the entire subject of Computational Geometry can be generalized and extended. A short Appendix details one of its early failures -- an attempt to apply the methods of algebraic complexity.

1.3. Historical Perspective

This section presents a computer scientist's view of the historical development of geometry. We shall pay particular attention to the germination of algorithmic and complexity notions in the work of the early geometers and lament the fact that such promising ideas were not destined to flower. Since this thesis proposes a major reworking of classical geometry to make it explicitly computational, a short review of traditional geometric thinking is not out of place. In particular, we take the view that classical mathematics is not a computational discipline, and that computer science must develop theoretical and algorithmic foundations on its own.

Egyptian and Greek geometry were masterpieces of applied mathematics. It is well established that the original motivation for tackling geometric problems was the need to tax lands accurately and fairly and to erect buildings [Eves (72)]. As often happens, the mathematics that developed has permanence and significance that far

transcends Pharaoh's original revenue problem. It is a field in which intuition abounds and new discoveries are within the compass (so to speak) of amateurs.

It is popularly held that Euclid's chief contribution to geometry was his exposition of the axiomatic method of proof, a notion that we will not dispute. More relevant to this discussion, however, is the invention of *Euclidean construction*, a schema which consists of an algorithm *and its proof*, intertwined in a highly stylized format. The Euclidean construction satisfies all of the requirements of an algorithm: it is unambiguous, correct, and terminating. After Euclid, unfortunately, geometry continued to flourish, while analysis of algorithms faced 2000 years of decline. This can be explained in part by the success of reductio ad absurdum, a technique which made it easier for mathematicians to prove the existence of an object by contradiction, rather than by giving an explicit construction for it (an algorithm).

The Euclidean construction is remarkable for other reasons as well, for it defines a collection of allowable instruments (ruler and compass) and a set of legal operations (primitives) that can be performed with them. The Ancients were most interested in the closure of the Euclidean primitives under finite composition. In particular, they wondered whether this closure contained all conceivable geometric constructions (e.g., the trisection of an angle). This is a computer science question -- do the Euclidean primitives suffice to perform all geometric "computations"? In an attempt to answer this question, various alternative models of computation were considered by allowing the primitives and the instruments themselves to vary. Archimedes proposed a (correct) construction for the trisector of a 60-degree angle with the following addition to the set of primitives: Given two circles, A and B, and a point P, we are allowed to mark a segment MN on a straightedge and position it so that the straightedge passes through P, with M on the boundary of A and N on the boundary of B [Eves (72)]. In some cases, restricted sets of instruments were studied, allowing compasses only, for example. Such ideas seem almost a premonition of the methods of automata theory, in which we examine the power of computational models under various restrictions. Alas, a proof of the insufficiency of the Euclidean tools would have to await the development of Algebra.

The influence of Euclid's *Elements* was so profound that it was not until Descartes that another formulation of geometry was proposed. His introduction of coordinates enabled geometric problems to be expressed as algebraic ones, paving the way for

the study of higher plane curves and Newton's calculus. Coordinates permitted a vast increase in computational power, bridged the gulf between two great areas of Mathematics, and led to a renaissance in constructivist thinking. It was now possible to produce new geometric objects by solving the associated algebraic equations. It was not long before computability questions arose once again. Gauss, now armed with algebraic tools, returned to the problem of which regular polygons with a prime number of sides could be constructed using Euclidean instruments, and solved it completely [Kazarinoff (70)]. At this point a close connection between ruler and compass constructions, field extensions, and algebraic equations became apparent. In his doctoral thesis, Gauss showed that every algebraic equation has at least one root (Fundamental Theorem of Algebra) [Courant (41)]. Abel, in 1828, went on to consider the same problem *in a restricted model of computation*. He asked whether a root of every algebraic equation could be obtained using only arithmetic operations and the extraction of *n*th roots, and proved that the answer was negative. While all constructible numbers were known to be algebraic, this demonstrated that not all algebraic numbers are constructible. Shortly thereafter, he characterized those algebraic equations which *can* be solved by means of radicals, and this enabled him to discuss the feasibility of specific geometric problems, such as the trisection of the angle [Kazarinoff (70)].

1.3.1 Complexity Notions in Classical Geometry

Euclidean constructions for any but the most trivial problems are very complicated because of the rudimentary primitives that are allowed. An apparently frequent pastime of the post-Euclidean geometers was to refine his constructions so that they could be accomplished in fewer "operations". In was not until the twentieth century, however, that any quantitative measure of the complexity of a construction problem was defined. In 1907, Emile Lemoine established the science of *Geometrography* by codifying the Euclidean primitives as follows [Lemoine (07)]:

1. Place one leg of the compass at on a given point.
2. Place one leg of the compass on a given line.
3. Produce a circle.
4. Pass the edge of the ruler through a given point.
5. Produce a line.

The total number of such operations performed during a construction is called its *simplicity*, although Lemoine recognized that the term "measure of complication" might be more appropriate. This definition corresponds closely to our current idea of the *time complexity* of an algorithm, although in Lemoine's work there is no functional connection between the size of the input (number of given points and lines) in a geometric construction and its simplicity. Indeed, Lemoine's interest was in improving Euclid's original constructions, not in developing a theory of complexity. At the former he was remarkably successful -- Euclid's solution to the Circles of Appolonius problem requires 508 steps, while Lemoine reduced this to fewer than two hundred [Coolidge (16)]. Unfortunately, Lemoine did not see the importance of proving (or perhaps was unable to prove) that a certain number of operations were *necessary* in a given construction, and thus the idea of a lower bound eluded him.

Hilbert, however, appreciated the significance of lower bounds. Working in a restricted model, he considered only those constructions performable with straightedge and *scale*, an instrument which is used only to mark off a segment of fixed length along a line. Not all Euclidean constructions can be accomplished with this set of instruments. For those which can, we may view the coordinates of the constructed points as a function F of the given points. Hilbert gave a necessary and sufficient condition for F to be computable using exactly n square root operations, one of the earliest theorems in algebraic computational complexity [Hilbert (99)].

Further evidence suggests that many of our present-day techniques for analyzing algorithms were anticipated by the geometers of previous centuries. In 1672, Georg Mohr showed that any construction performable with ruler and compass can be accomplished with compass alone, insofar as the given and required objects are specified by points [Eves (72)]. (Thus, even though a straight line cannot be drawn with compass alone, two points on the line can each be specified by intersecting two circular arcs.) What is notable about Mohr's proof is that it is a *simulation*, in which he demonstrates that any operation in which the ruler participates can be replaced by a finite number of compass operations. Could one ask for a closer connection with automata theory? Along similar lines is the result that the ruler used in any construction may have any positive length, however small, and yet be able to simulate a ruler of arbitrary length [Eves (72)].

While Lemoine and others were occupied with the time complexity of Euclidean constructions, the question of the amount of *space* needed for such constructions was also raised. While the measure of space that was used does not coincide with our current definition as the amount of memory used by an algorithm, it comes remarkably close and is a quite natural one: the area of the plane needed to perform the construction. In general, the space used depends on the area of the convex hull of the given loci and on the size of the required result, as well as on the size of any intermediate loci that need to be formed during the construction [Eves (72)]. Our point here is that time and space notions are not entirely foreign to Geometry.

When the impossibility of certain Euclidean constructions was demonstrated by Galois, it was realized that this prevented the *exact* trisection of an angle but said nothing about the feasibility of an approximate construction. In fact, asymptotically convergent procedures for the quadrature of the circle and duplication of the cube were known to the ancient Greeks [Heath (21)]. The history of iterative algorithms is indeed a long one.

1.3.2 The Theory of Convex Sets, Metric and Combinatorial Geometry

Geometry in the nineteenth century progressed along many lines. One of these, promulgated by Klein, involved a comprehensive study of the behavior of geometric objects under various transformations, and projective geometry formed an important offshoot. While research on finite projective planes leads to fascinating questions in combinatorial theory and discrete algorithms, this aspect of geometry will not be pursued in this thesis, largely because such topics are at best distantly related to problems involving the properties of point sets and location of objects.[1]

The growth of real analysis had a profound effect on geometry, resulting in formal abstraction of concepts that had previously been only intuitive. Two such developments, metric geometry and convexity theory, provide the principal mathematical tools that we will exploit in later chapters to aid in the design of fast algorithms.

[1] But for a modern treatment see [Karteszi (76)].

Distance is the essential notion of geometry. The *metric*, its generalization, was able to draw geometric concepts and insights into analysis, where the idea of the "distance" between functions gave rise to function spaces and other powerful constructs. Unfortunately, much of what developed withdrew toward the nonconstructive. Function spaces by their nature are not computational objects.

The significance of convexity theory is that is deals analytically with *global* properties of objects and enables us to deal with extremal problems. Unfortunately, many questions in convexity are cumbersome to formulate algebraically, and the subject tends to encourage nonconstructive methods.

Combinatorial geometry is much closer in spirit to our goal of algorithmic geometry. It is based on characterizations of geometric objects in terms of properties of finite subsets. For example, a set is convex iff the line segment determined by every *pair* of its points lies entirely in the set. The inadequacy of combinatorial geometry for our purposes lies in the fact that for most sets of interest the number of finite subsets is itself infinite, which precludes algorithmic treatment. Our job will be to remedy these deficiencies and produce workable mathematics that will lead to good algorithms.

1.4. Prior work in geometric algorithms

Even though we plan to develop Computational Geometry from first principles, the reader should realize that many geometric algorithms already exist. A large number of applications areas provide problems that are inherently geometric and have been examined by many researchers. These include the Euclidean traveling salesman, minimum spanning tree, hidden line, and linear programming problems, among hosts of others. In order to demonstrate the broad scope of computational geometry in a convincing way, we will defer presenting background material on such problems until they occur in the text.

A number of subjects and applications have been deliberately omitted because the author felt that they had been adequately treated elsewhere or that he had nothing intelligent to remark about them. One of these is geometric modeling by

means of spline curves and surfaces, a topic that is closer in spirit to numerical analysis than it is to geometry and has been dealt with in [Bezier (72)], [Forrest (72)], and [Riesenfeld (73)]. We should note that Forrest refers to his discipline as "Computational Geometry".

In a fascinating book called Perceptrons (of which the subtitle is "Computational Geometry"), Minsky and Papert deal with the complexity of predicates that recognize certain geometric properties, such as convexity. The intent of their work was to make a statement about the possibility of using large retinas composed of simple circuits to perform pattern recognition tasks. Their theory is self-contained and does not fall within the algorithmic outlines of this thesis.

We intentionally will not deal with graphics software or geometric editors. While these systems are undoubtedly candidates for some of the algorithms we will develop, they raise issues that are oriented more toward implementation details and the user interface than toward analysis of algorithms. Included in the same class are numerical control software for machine tools, support programs for graphic plotters, map-drawing systems, and software for architectural design and civil engineering.

On hearing the term "Computational Geometry", many people assume that it refers to the problem of proving geometry theorems by computer. While this is a fascinating study, it reveals much more about our theorem-proving heuristics and understanding of proof procedures than it does about geometry per se, and will thus not be treated here.

Chapter 2
Towards Computational Geometry

The purpose of this chapter is to explore some of the basic issues that arise as we attempt to solve geometric problems on a computer. For reasons of exposition, we will focus on the (apparently) simple problem of determining whether or not a given polygon of N sides is convex, and try to produce the fastest possible algorithm. In solving this problem, we will be forced almost immediately to deal with the following questions:

1. Problem specification. What *exactly* is the computational problem to be solved? What mathematical results are relevant? How can a convex polygon be characterized?

2. Problem representation. How is the polygon to be represented? Precisely what input will be provided and what output is expected?

3. Model of computation. What is the theoretical setting in which the algorithms are to be analyzed? What primitive operations are to be allowed and how much will be charged for them?

4. Lower bounds How can lower bounds be obtained for a problem that is inherently geometric (as opposed to algebraic) ? Are the bounds robust over different models of computation?

5. Divide and conquer. Divide-and-conquer is a very powerful algorithmic technique. Does it find application in computational geometry?

6. On-line algorithms. In some problems the data points are not all available at the outset, but arrive in real-time. Is there a fast algorithm that is able to process the information as it is received?

7. Average-case analysis. What is the expected-time behavior of geometric algorithms? What mathematics is necessary for this analysis?

The process of defining the problem begins in this case with elementary definitions from convexity theory which lead to theorems characterizing convex polygons. These theorems, however, do not suggest efficient algorithms, and we must derive equivalent characterizations which do. An iterative scheme ensues, in which we prove new theorems, analyze the resulting algorithms, and compare them with known lower bounds on running time. By following the development of a simple but optimal algorithm from its raw beginnings, the reader will obtain a preview of the spirit and methods of this thesis.

2.1. When is a polygon convex?

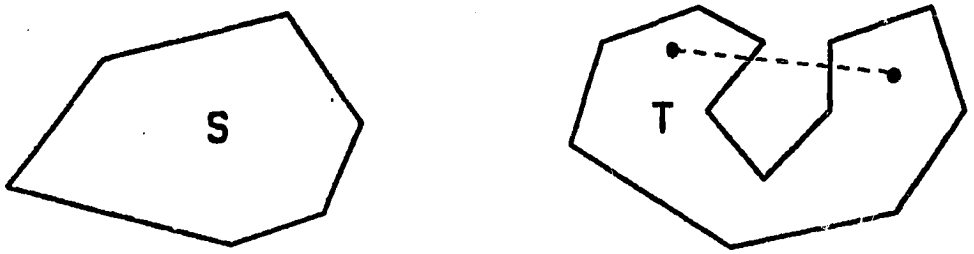


Figure 2.1: S is Convex, T is Not.

Let us start with one definition of a convex set:

Definition 2.1: A set S is *convex* iff the line segment joining every pair of points of S lies wholly within S.

In Figure 2.1, for example, S is convex, while T is not.

This definition is a very powerful one for the purpose of proving theorems. As an algorithmic test to determine whether a set is convex, however, the definition is not immediately useful. In general, S will have an uncountable number of points, so a direct examination of all the line segments they determine is out of the question. A line segment itself has an uncountable number of points, so it is not even clear how one might verify that it is contained in S. Are we therefore constrained to work only with countable or finite sets?

Fortunately not; the only requirement is that S be *finitely specifiable*. For example, while the interior of a plane polygon contains uncountably many points, as does its boundary, the polygon can nonetheless be specified by a finite sequence of vertices.

2.2. Representation of geometric objects

Let us adopt some reasonable conventions regarding simple geometric objects:

A *point* will be represented as a vector of variables containing its Cartesian coordinates. We assert that the choice of coordinate system cannot affect the asymptotic running time of any geometric algorithm, provided that the model of computation (to be specified in the next section) allows the necessary transformations. All this means is that a vector of coordinates in one system can be transformed into any other system in time that depends possibly on the number of dimensions, but *not* on the number of points involved[1].

Stated another way, a set of N points in k dimensions can be put into Cartesian form in O(Nk) time, so asymptotically no time is lost by assuming that the points are already given in Cartesian coordinates.

It is now clear that a natural representation for an unordered set of N points in k

[1]Note that there must not be any "interaction" among the points. In center-of-mass coordinates, for example, the required transformation cannot be performed in constant time.

dimensions is either an N x k array or a list of k-vectors. It is important to realize that a polygon is an *ordered* collection of points, and that a different polygon results if any two points are interchanged.[2] Specifying a polygon unambiguously requires giving its vertices *in the order that they occur on the boundary.* A number of data structures are suitable for storing polygons. In many cases two vectors, for x- and y-coordinates respectively, will suffice. Often, though, processing the polygon will involve the insertion and deletion of vertices, in which case a doubly-linked list will be more economical in terms of time.[3] These representations can be mutually transformed in linear time.

In either the vector or linked-list representation, there are normally 2N different realizations of the same N-gon, since the enumeration can begin with any vertex and proceed either clockwise or counterclockwise. To avoid this multiplicity of representations, we define a canonical form:

Definition 2.2: A polygon is *simple* iff no nonconsecutive sides intersect and consecutive sides intersect only at a single point..

Definition 2.3: A simple polygon is in *standard form* if its vertices occur in counterclockwise order[4], with all vertices distinct and no three consecutive vertices collinear, beginning with the vertex that has least y-coordinate. (If two or more vertices are tied with least y-coordinate, we begin with the one

---

[2]Strictly speaking, a polygon is a closed plane figure whose boundary is a polygonal line. While the polygon and its boundary contain uncountably many points, they both may be specified uniquely by listing the vertices of the polygon. We will often deliberately make no distinction among a polygon, its boundary, and this representation.

[3]In such applications, the use of linked lists will reduce the order of the time complexity at a cost of only a constant factor in space.

[4]"Counterclockwise order" is defined precisely in Appendix A.

that has least x-coordinate.[5] See Figure 2.2). A non-simple polygon is in standard form if its first three vertices occur in counterclockwise order and the first vertex is lexicographically least.

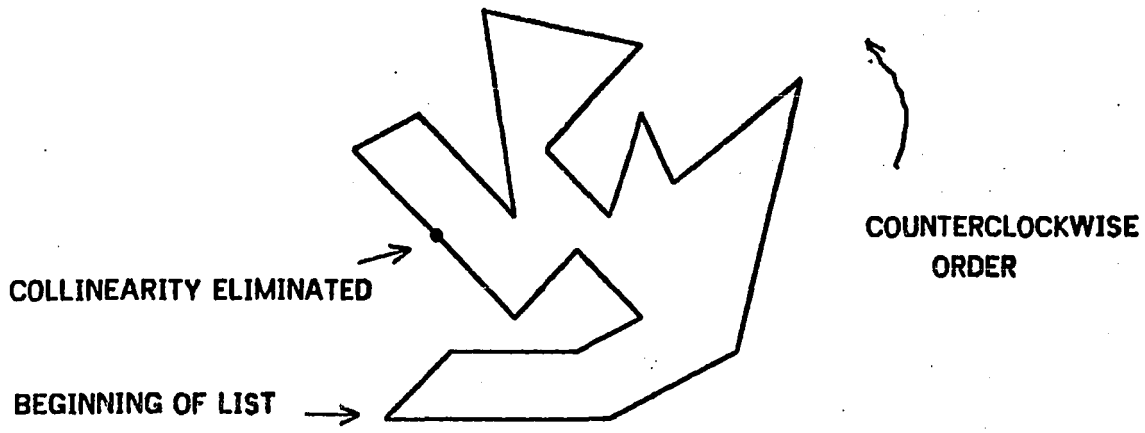


Figure 2.2: A Polygon in Standard Form

Several aspects of the above definition of standard form require explanation. First of all, the definition only applies to *simple* polygons -- those that are not self-intersecting. This is because a single point in the plane may occur legitimately more than once as a vertex of a non-simple polygon (Figure 2.3). As we shall see, testing for simplicity is a non-trivial problem.

The requirement that the vertices be distinct is imposed in order to remove the degeneracy that arises when an edge of the polygon has zero length. This will mean that a quadrilateral with a null edge will be represented as a triangle, since they are identical *as polygons*. Likewise, we eliminate consecutive collinear vertices because to allow them would permit multiple representations of the same plane figure. The counterclockwise orientation is chosen so that, as the boundary of the polygon is traversed, the interior lies to the left. This and the insistence on beginning with the lexicographically least vertex are designed to simplify the presentation of later algorithms and reduce their running time.

[5] That is, the first vertex is lexicographically least in y,x order.

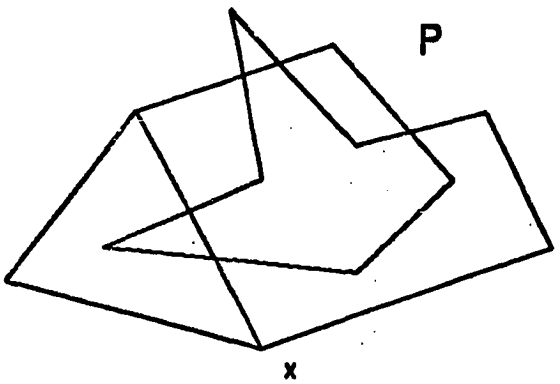


Figure 2.3: The point x occurs three times as a vertex of P.

We now indicate how to convert a simple polygon P to standard form in linear time:

1. If P is in vector form, convert it to circular doubly-linked list representation.

2. Perform a single pass through the list, eliminating duplicate and collinear vertices. Since P is simple, multiple copies of a vertex must be adjacent in the list, which is doubly-linked to allow rapid deletion. To determine whether three points are collinear, we need only compute the area of the triangle determined by them. The points are collinear iff the area is zero. During the same pass, then, consecutive collinear points can be removed. We also record the vertex that is lexicographically lowest.

3. We now need to decide whether the vertices are in clockwise or counterclockwise orientation. This may be done in O(N) time by evaluating the signed area of P, as given by equation (A.1). If the area is positive, the polygon is already in standard form; if it is negative, the representation needs to be reversed. In either case the list pointers can now be modified in linear time to yield P in linked-list standard form.

4. If vector form is now desired, the transformation is trivial.

This detailed explanation of so elementary a process should convince the reader that transformation to standard form is merely a bookkeeping procedure in which no essential computation is buried.

2.2.1 Congruence and similarity

The prime objective of this thesis is to isolate and analyze fundamental geometric problems. Some of these fall into the mainstream of computational geometry, while others are peripheral but raise questions that are simply too tantalizing to ignore. One of these is the problem of congruence, which is included here because of its close connection with representation issues.

Definition 2.4: Two polygons are congruent if they can be made to coincide.

Problem P2.1: (Congruence of Polygons) Given two polygons, are they congruent?

The computational form of this question is really: Do two representations define the same polygon up to rotation and translation? Because rotations are allowed, it does not suffice to transform both polygons to standard form and verify that they are identical.

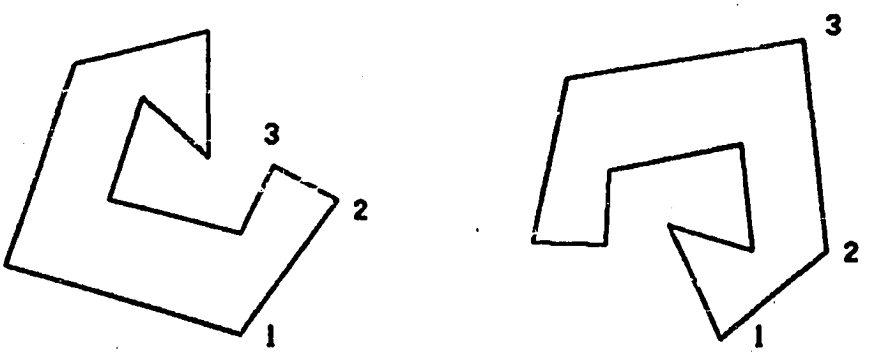


Figure 2.4: Congruent polygons may have different standard forms.

To solve this problem we need a representation for polygons that is rotation- and translation-independent. We can satisfy this requirement by building a circular list containing the *lengths* of the sides of the polygon and its internal angles, stored alternately.[6] Determining whether two polygons in this representation are congruent then reduces to the question of whether two circular lists differ only by a shift. Glenn Manacher [Manacher (76)] realized that this is a simple pattern-matching problem that can be solved in linear time by the Knuth-Morris-Pratt algorithm [Knuth (77)]. Since two polygons are *similar* if they are congruent up to a change in scale, the same procedure can be used to test for similarity in linear time if we first multiply the sides of one polygon by the ratio of their longest (or shortest) edges.

2.3. Model of computation

A *model of computation* specifies the primitive operations that an algorithm may perform and the costs that will be charged for them. It is essential that a model be carefully specified before any attempt is made to prove upper and lower bounds on execution time, since such results have meaning only within a particular computational framework. In choosing one, we normally must make compromises between realism and mathematical tractability, selecting a scheme that represents actual computers as closely as possible, while still permitting thorough analysis. What sort of model is appropriate for geometric applications? Having finally fixed on a specific representation for a polygon, we must now decide which operations may reasonably be performed on that representation. We regularly encounter several types of problems, each requiring a different set of primitives:

1. Subset selection. In this kind of problem we are given a collection of objects and asked to choose a subset that satisfies a certain property. Examples are finding the two closest of a set of N points or finding the vertices of its convex hull. The essential feature of a subset selection problem is that *no new objects need be created*; the solution consists entirely of elements that are given as

[6] For a purely technical reason we store a side of length L as the number L + 2π, and angles in radians. Then sides and angles can never be confused. It should be clear that this representation can be obtained from standard form in linear time.

input. In many cases such a problem can be solved using only comparisons and the four arithmetic operations so that a decision-tree model is appropriate.

2. Computation. Given a number of objects, we may need to compute the value of some geometric parameter. The primitives allowed in the model must be powerful enough to permit this calculation. Suppose, for example, that we are working with a set of points having integer coordinates. In order to find the distance between a pair of points, we not only need to be able to represent irrational numbers, but to take square roots as well. In other problems we may even require trigonometric functions.

The model that we will adopt for most purposes is a random-access machine (RAM) similar to that described in [Aho (74)] but in which each storage location is capable of holding a single real number. The following operations are available at unit cost:

1. The arithmetic operations +, - , x , / .

2. Comparisons between two real numbers. ($<, \leq, =, \neq, \geq, >$).

3. Trigonometric functions, EXP, and LOG. (In general, analytic functions.) While we normally will not employ these functions, they are included in the model to strengthen our lower bound results.

4. Indirect addressing of memory (integer addresses only).[7]

This model is an amalgam of useful features of the straight-line, computation tree, and integer RAM models, and we shall refer to it as a *real RAM*. It closely reflects the kinds of programs that are typically written in high-level algebraic languages

[7]To allow the address calculation mechanism to truncate real values to integers would add the FLOOR function to our set of primitives and make lower bounds almost impossible to prove. The FLOOR cannot be computed in any constant number of arithmetics and comparisons, as an information-theoretic argument will show, although it is often realistic to consider it a primitive on most machines.

such as FORTRAN and ALGOL, in which it is common to treat variable of type REAL as having unlimited precision. At this level of abstraction we may ignore such questions as how a real number can be read or written in finite time.[8]

Knuth has recently popularized the use of Ω, a notational device that distinguishes nicely between upper and lower bounds [Knuth (76)], and which we will adopt.

$O(f(N))$ denotes the set of all functions $g(N)$ such that there exist positive constants C and N_0 with $|g(N)| \leq Cf(N)$ for all $N \geq N_0$.

$\Omega(f(N))$ denotes the set of all functions $g(N)$ such that there exist positive constants C and N_0 with $g(N) \geq Cf(N)$ for all $N \geq N_0$.

Thus $\Omega(f(N))$ is used to indicate functions that are *at least as large* as some constant times $f(N)$, precisely the concept that one needs to describe lower bounds.

It is often observed that a problem can be solved by transforming an instance of it into an instance of a different problem, solving the transformed problem, and translating the result back into the context of the original problem. If this is possible, we say that one problem is *transformable* to the other. This idea is so useful for proving lower bounds that we formalize the process: Given two problems *A* and *B*,

1. The input to problem *A* is converted into a suitable input for *B*.

2. Problem *B* is solved.

3. The output of *B* is transformed into a *correct* solution to *A*.

If the above transformation steps 1 and 3 together can be performed in $O(T(N))$

[8]This is perfectly legitimate -- the same problem arises with straight-line programs

time, then we say that *A* is *T(N)-reducible to B*, written $A \propto_{T(N)} B$.[9] Reducibility is not necessarily a symmetric relation; in the case when *A* and *B* are mutually reducible, we say that they are *equivalent*.

Theorem 2.1: (Lower bounds via reducibility) If problem *A* is known to require T(N) time and *A* is R(N)-reducible to *B* ($A \propto_{R(N)} B$), then *B* requires at least T(N) - O(R(N)) time.

Theorem 2.2: (Upper bounds via reducibility) If problem *B* can be solved in T(N) time and $A \propto_{R(N)} B$, then *A* can be solved in at most T(N) + O(R(N)) time.

The real RAM is so powerful (because of its transcendental functions and the possibility of encoding tricks) [10] that lower bounds are difficult to prove. We will usually establish a lower bound for a problem *P* by showing that another problem *Q*, known to require T(N) time, is linear-time reducible to *P*. The most important known lower bound is that for sorting, which requires $\Omega(N \log N)$ comparisons, worst-case, on a real RAM.[11] This result is our primary source of lower bounds.

The model of computation affects the assertions we may make about coordinate systems. If only arithmetic operations are allowed, for example, then a point in polar form *cannot be transformed at all* to Cartesian coordinates, because of the trigonometric functions involved. This may not prove to be much of a restriction, however. If we only need to compare the polar angles of two points given in

---

[9] Reducibility is usually defined to be a relation on languages, in which case no output transformation is necessary because the output of a string acceptor is either zero or one. See [Karp (72)]. For geometry problems, we need the greater flexibility afforded by the more general definition.

[10] A single memory location can hold a potentially infinite amount of information, but this is also true of the integer RAM model.

[11] This is implied by a stronger theorem of [Friedman (72)], whose model even permits the computation of arbitrary analytic functions.

Cartesian coordinates, for example, this can be done with arithmetics and comparisons only. It is not necessary to compute the angles.[12]

2.4. Convexity Testing

We now have enough background to make computational sense of the geometric problem stated at the beginning of this chapter.

Problem P2.2: (Convexity Test) Given a plane polygon with N vertices in standard form, find a real RAM algorithm to determine whether it is convex.

In order to proceed, we must search for properties of convex polygons that can be used in an algorithmic test.

A *diagonal* of a polygon is a line segment joining two nonconsecutive vertices.

Property 2.1: A polygon P is convex iff every diagonal of P is a subset of P. [13]

While Property 2.1 says that we may restrict our attention to a finite number of line segments, it still does not yield an algorithm because we have no finite test as yet for whether a line segment lies within a polygon. Such a condition is not difficult to obtain:

Property 2.2: A line segment L is a subset of a polygon P iff at least one point of L is interior to P and the interior of L intersects no edge of P.

---

[12]This turns out to be very convenient in many algorithms, including that of Graham (section 3.3).

[13]In this thesis the term "Property" is used to denote a elementary theorem from plane geometry that is stated without proof. Except for the illustrative example in this chapter, no new theorems or algorithms will be based on such "Properties".

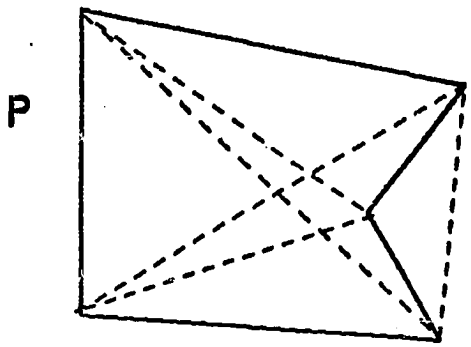


Figure 2.5: No Diagonal Intersects an Edge, yet P is not Convex.

We may now use Properties 2.1 and 2.2 to formulate an algorithm. Each vertex of P is the vertex of an interior wedge determined by the two edges of P that meet at V.[14] A diagonal D emanating from V can lie within P only if D lies within the wedge. This can be tested in a constant number of operations by examining the angles that D forms with the two coincident edges. It remains only to form each diagonal, of which there are $N(N-3)/2 = O(N^2)$ if P has N vertices, and test it against each of the N edges of P for intersection. Each of these tests can be performed in a constant number of operations, independent of N. It should be clear that this algorithm can be implemented to run in $O(N^3)$ steps.

Property 2.1 is a perfectly good characterization in combinatorial geometry. In traditional mathematics the value of such results is judged by their conciseness, elegance, and the number of new theorems they enable us to prove. In computational mathematics we must adopt a different measure of success, one based on algorithmic efficiency. We will constantly strive to develop definitions and theorems that will be of direct use in speeding up algorithms. This goal is both practical and aesthetic since a procedure that uses the fewest possible number of steps makes a minimal demand on computational resources and at at the same time satisfies the artistic requirement of simplicity that runs through all of Mathematics.

[14] Since P is in standard form.

How does our convexity test measure up? Could it be optimal in the sense that any algorithm to determine whether a polygon is convex must perform at least cN^3 steps? Any correct algorithm for a problem provides an *upper bound* on the time sufficient to solve it, though not necessarily a least upper bound. To show optimality of a specific procedure, we must prove a *lower bound*, a theorem stating that *no* algorithm could succeed using fewer operations. (That a single algorithm suffices to establish an upper bound while lower bounds must apply to all conceivable algorithms may account for the relative scarcity of optimality results.)

Definition 2.5: A lower bound on an algorithm is said to be *trivial* if it refers only to the time necessary to read the input or write its output.

For the problem of testing convexity only a trivial lower bound is available:

Theorem 2.3: Any algorithm that determines whether or not a polygon having N vertices is convex must perform at least cN operations, for some constant c>0.

Proof: We show that each vertex must be examined. For suppose that the algorithm reports that P is convex without having processed some vertex V. V can be moved, without changing the outcome of the convexity test, so that P is no longer convex. (See Figure 2.6.) Thus the algorithm cannot answer correctly without having examined V.

We now know that a linear number of operations are required to perform a convexity test and that $O(N^3)$ operations suffice. Judging from the simplicity of the lower bound proof, one might expect that a more refined argument could improve it. Similarly, a more clever algorithm could conceivably test convexity in less than cubic time. Because there is a gap between the bounds, at least one of them is not the best possible, and work still remains before we can be satisfied with the analysis of this problem.

In an effort to find a better algorithm we must dispense with Property 2.1. While succinct and elegant, it is based on inspection of all of the diagonals of a polygon, and any algorithm which uses the theorem directly is condemned to perform at least

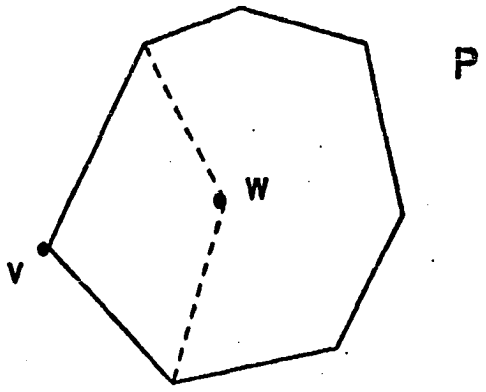


Figure 2.6: Convexity Testing. Every vertex must be examined.

$O(N^2)$ operations, just because there are that many diagonals. While $O(N^2)$ would be an improvement over N^3, it provides no hope of closing the gap entirely. At this stage we must develop alternative characterizations of convex polygons, use them to find algorithms, and analyze the running times of these algorithms. The following attempt shows that even erroneous conjectures can lead to interesting and profitable research ideas:

Nontheorem. A polygon P is convex iff no interior angle is reflex.

(Recall that an angle is *reflex* if it exceeds 180 degrees.) This result, if true, would lead to an easy linear-time test for convexity -- it would suffice to examine N triples of consecutive vertices and compute the angle determined by each, verifying that no angle is reflex. Figure 2.7 shows, however, that the characterization is incorrect.

The problem is that, as shown in the figure, a nonsimple polygon may have no reflex angles and still fail to be convex. However, we may make use of the following:

Property 2.3: A polygon is convex iff it is simple and has no reflex interior angles.

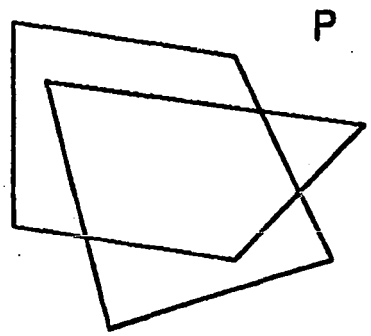


Figure 2.7: No angle is reflex, but P is not convex.

If our polygon is known in advance to be simple, we may check its convexity in linear time by examining each interior angle. If this is not known, though, we must determine whether or not P is simple. This can be done by testing every pair of nonconsecutive sides for intersection, but then we are back to a quadratic algorithm! The issue of whether a polygon is simple arises in many other geometric problems. (For example, the convex hull of a simple polygon can be found in linear time, but N log N time is required if the polygon is not known to be simple.) This problem is treated fully in Section 5.4. As far as the convexity test is concerned, though, the possibility of self-intersection makes Property 2.3 not as useful as it might appear to be initially.

Rather than continuing with the simplicity idea, let us try an alternate approach, based on the fact that the vertices of a convex polygon seem to occur in angular order around its boundary. Traversing the boundary of P induces a direction on each edge, converting it into a vector. (See Figure 2.8). We may then speak unambiguously of the *angle* defined by an edge as the angle it subtends with the positive x-axis.

Theorem 2.4: A polygon P is convex iff in standard form its edge angles are non-decreasing.

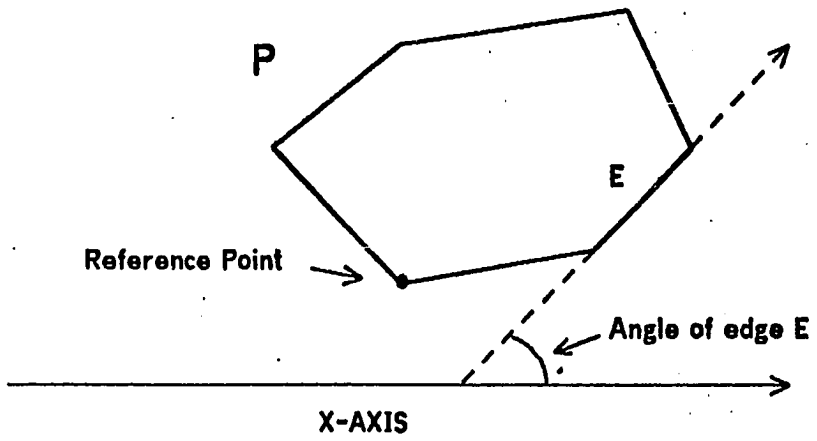


Figure 2.8: Defining edge angles of a polygon.

Proof: In standard form, the first vertex of a polygon is lexicographically lowest in y,x order, and the vertices occur in clockwise sequence. Thus the first edge of P has positive angle. If the conditions of the theorem are met, then P is simple or it would not be a closed polygon. Similarly, P has no reflex angles so Property 2.3 applies! To show necessity, consider the first vertex V at which monotonicity fails. Then either the interior angle at V is reflex or an edge at V intersects some other edge of P. Monotonicity must be strict because in standard form consecutive collinear vertices have been eliminated.

Our work is now done because Theorem 2.4 yields a linear-time algorithm: P can be put into standard form in linear time, and monotonicity of angle can be verified in a single pass through the vertex list.

Theorem 2.5: Whether a plane polygon is convex can be determined in O(N) time, and this is optimal.

It is difficult to overemphasize the theoretical importance of lower bound results, for they put a problem in perspective and reveal its inherent complexity. The computer scientist cannot rest until his upper and lower bounds coincide, for until then his understanding of the problem is not complete. From a practical standpoint, though, lower bounds are not always as important as they may seem, since they refer to the exact solution of a specific problem in a possibly unrealistic model of computation. The system-builder is usually more interested in fast algorithms; in fact, he wants algorithms that run quickly on problems of a certain size and is often unconcerned about asymptotic behavior. Of course, the true pragmatist would prefer an algorithm that his programmers can understand and code quickly. We will try in this thesis to strike a balance between theoretical and practical considerations.

In our example the trivial lower bound was sharp and our original upper bound was at fault. We will not always be so fortunate -- more powerful techniques will be needed to prove better lower bounds. However, the example of this section is a paradigm for the rest of the thesis, and we will often begin with classical results, probe their inadequacies, prove new theorems, develop algorithms, and establish lower bounds.

If *ad hoc* techniques and theorems had to be developed for each new problem, computational geometry would rapidly become unwieldy. Fortunately, we will often be able to establish an equivalence between a geometric question and a solved problem in computer science, reducing tremendously the number of new results that need to be obtained. We shall spend considerable time exhibiting such correspondences and deriving algorithms from them.

Equivalences among problems are also important because they are a source of lower bounds. It is easy to show by an adversary argument that any decision tree that determines whether or not a list of N elements is sorted must make at least N-1 comparisons, in the worst case. Since this problem can be transformed, using no comparisons at all, into an instance of convexity testing, we know immediately that convexity testing must also require at least N-1 comparisons, worst case. Reducibilities of this form are a powerful means of obtaining lower bounds.

Our goal, then, will be to reshape classical geometry into a computational

discipline by manipulating definitions and theorems until they can be made to yield fast algorithms, exploiting, whenever possible, analogies between geometric problems and well-understood discrete algorithms.

Chapter 3
Convex Hull Algorithms

3.1. The Problem of the Convex Hull

3.1.1 Background

Recalling Definition 2.1, a set *S* is *convex* iff the line segment joining every pair of its points consists entirely of points of *S*. The *convex hull* of *S* is the smallest convex set containing S.[1] Intuitively, the concept of the convex hull is easy to understand. To find the hull of a set of points in the plane, imagine surrounding the set by a large, stretched rubber band. When the band is released, it will assume the shape of the convex hull.[2] (See Figure 3.1.)

The convex hull of a set can be viewed informally in other ways. It is the shape assumed by the package when the set is gift-wrapped. A physicist would say that, from infinity, a set "looks like" its convex hull, by which he means that from sufficiently far away the closest point of the set is also a point of the hull (Figure 3.2).

---

[1] "Smallest" is in the sense of set inclusion. That is, H is the hull of S iff H contains S, H is convex, and no convex subset of H also contains S.

[2] Note that this algorithm operates in linear time! One need only drive nails into a board (at constant cost per nail) and use a large rubber band. Of course, the model of computation we will be using does not include such primitives. The point is that the existence of a fast, intuitive, perceptual or analog procedure can sometimes hinder our understanding of complexity issues. How are we to reconcile this disarmingly efficient carpenter's algorithm with the N log N lower bound to be proved later?

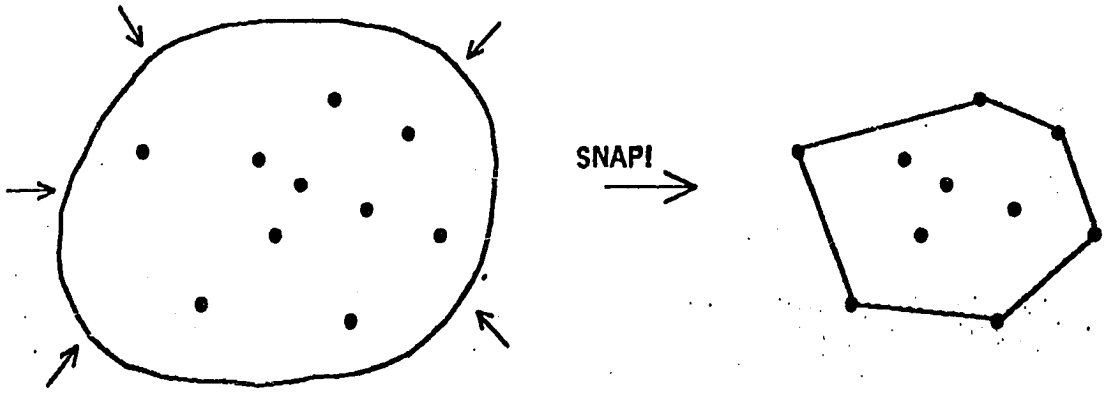


Figure 3.1: A Rubber Band Assumes the Shape of the Convex Hull.

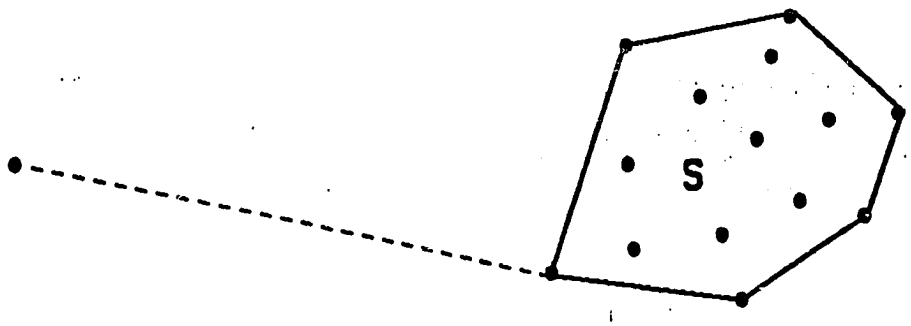


Figure 3.2: At large distances the closest point of S is a hull point.

Unfortunately, such analog devices are unavailable to a computer and we will need a clean definition of the convex hull and a precise specification of how it is to be represented in the machine. We must then crystallize all of our informal notions about hulls into an explicit computational procedure. This is the process of *algorithm design*, during which we combine our conceptual understanding of the problem with knowledge about its mathematical structure in order to decide which algorithmic techniques will be useful. It is by exploiting this structure that we

obtain efficient algorithms. For example, Theorem 2.4, which characterizes convex polygons, enables us to avoid examining all pairs of vertices in determining whether a given polygon is convex.

3.1.2 Mathematical preliminaries

The reader may be wondering at this point why we do not simply consult a geometry textbook, obtain a suitable definition of the convex hull, and follow the prescription for constructing it.[3] Let us try this approach, for it reveals much about the non-algorithmic nature of classical mathematics.

Definition 3.1: [Valentine (64)] The *convex hull* of a set is the intersection of all convex sets containing it.

This definition is a formal characterization of the hull as the smallest convex set containing a given one. While it is elegant and very powerful for proving theorems, it is useless for computational purposes. In general, the number of convex sets involved in the intersection is uncountable, so there is no hope of applying the definition directly -- it is *constructive* but not *computational.* This distinction, while often ignored, is of crucial importance as we undertake to fit geometry into contemporary computer science.

Definition 3.1 is not the only way of defining the convex hull. The following definition, based on a closure operation, corresponds to the intuitive idea of "filling in" a set until it becomes convex:
A convex combination of a set of points p_i is a sum of the form

$$\sum_{i=1}^{N} w_i p_i \ , \text{ where } w_i \geq 0 \text{ and } \sum_{i=1}^{N} w_i = 1 \ . \qquad (3.1)$$

[3] This would have the side benefit of shortening the thesis by 50 pages.

Definition 3.2: [Stoer (70)] The *convex hull* of S consists of all convex combinations of points of S.

It is easy to see that this definition will not help us find the hull by any obvious method in less than exponential time.

3.1.3 Statement of the problem

We are not quite ready to formulate the hull problem computationally because it is still not clear what form the output of a hull algorithm will take. The input is an unordered set of points but what of the result?

Theorem 3.1: [Benson (66)] The convex hull of a set of N points is a convex polygon having at most N sides.

Having specified in the last chapter a model of computation and a way of representing both convex polygons and unordered sets of points, we are at last ready to state two versions of the convex hull problem:

Problem P3.1: (Planar Convex Hull) Given a set of N points in the plane, find its convex hull (that is, the standard form of the *polygon* that defines the hull).

Problem P3.2: (Planar Extreme Points) Given N points in the plane, identify those that are vertices of the convex hull.

Both of these problems can be generalized to k dimensions, but we shall work almost entirely in the plane for the remainder of this thesis.

It should be clear from our discussion of representations that Problem P3.1 is asymptotically at least as hard as P3.2, because the output of P3.1 becomes a valid solution to P3.2 if we merely recopy the polygon produced by the former as an unordered list of points. In the notation of Chapter 2, P3.2 is O(N)-time reducible to P3.1:

$$\text{PLANAR EXTREME POINTS} \propto_N \text{PLANAR CONVEX HULL} ,$$

It is natural to ask whether the former is asymptotically easier than the latter or if they are in fact equal in complexity. This and many other questions will be examined in the succeeding sections. The rest of this chapter is devoted to a study of the complexity of the planar convex hull problem and to algorithms for solving it.

3.2. Early development of a convex hull algorithm

Remembering our earlier nonconstructive, let us now seek mathematical results that will lead to efficient algorithms.

Definition 3.3: A point p of a convex set S is an *extreme point* if no two points a,b ϵ S exist such that p lies on the open line segment (a,b).

Theorem 3.2: [Benson (66)] The set E of extreme points of a finite set S is the smallest subset of S having the property that hull(E) = hull(S), and E is precisely the set of vertices of hull(S).[4]

It follows that two steps are required to find the convex hull of a finite set:

1. Identify the extreme points. (This is Problem P3.2.)

2. Order these points so that they form a convex polygon.

We need a theorem that will enable us to test whether a point is an extreme point.

Theorem 3.3: A point *p* is an extreme point of a plane convex set S unless it lies

[4]Under our definition of standard form, every hull vertex is an extreme point because consecutive collinear vertices are disallowed.

in some triangle whose vertices are in *S* and is not a vertex of the triangle.[5]

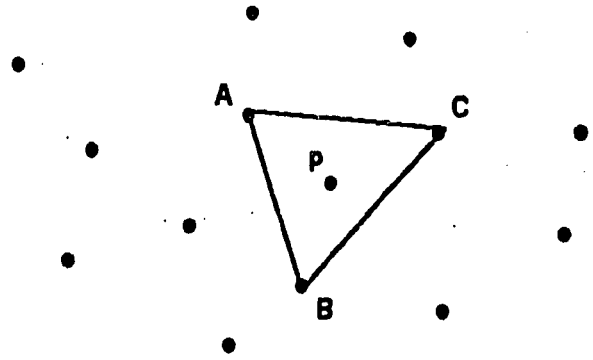


Figure 3.3: Point *p* is not extreme because it lies inside *ABC*.

This theorem provides an algorithm for eliminating points that are not extreme. There are $O(N^3)$ triangles determined by the N points of *S*. Whether a point lies in a given triangle can be determined in a constant number of operations (see Appendix A), so we may learn if a specific point is extreme in $O(N^3)$ time. Repeating this procedure for all N points of *S* requires $O(N^4)$ time. While our algorithm is extremely inefficient, it is conceptually simple and demonstrates that determining whether a point is extreme is decidable.

We have spent $O(N^4)$ time just to obtain the extreme points which must be ordered somehow to form the convex hull. The nature of this order is revealed by the following theorems:

Theorem 3.4: A ray emanating from an interior point of a bounded convex figure *F*

[5]This follows immediately from the proofs of Theorems 10 and 11 of [Hadwiger (64)]. The generalization to k dimensions is obtained by replacing "triangle" by "simplex on k+1 vertices".

intersects the boundary of *F* in exactly one point.[6]

Theorem 3.5: Consecutive vertices of a convex polygon occur in sorted angular order about any interior point.

Proof: Given a convex polygon P, assume that there exist three consecutive vertices ABC of P such that the polar angle of B (measured with respect to some interior point z and line zA) is greater than the polar angle of C. Then any ray from z which intersects edge BC also intersects AB, so by Theorem 3.4 P cannot be convex as claimed.

Imagine a ray, centered at an interior point z of polygon *P*, that makes a counterclockwise sweep over the vertices of *P*, starting from the positive x-axis. As it moves from vertex to vertex, the polar angle[7] subtended by the ray increases monotonically. This is what we mean by the vertices of *P* being "sorted". (See Figure 3.4.)

Given the extreme points of a set, we may find its convex hull by constructing a point z that is known to be interior to the hull and then sorting the extreme points by polar angle about z.

Definition 3.4: The centroid of a finite set of points is their arithmetic mean, $(P_1 + \ldots + P_N)/N$.

Theorem 3.6: The centroid of a set is interior to its convex hull.[8]

---

[6]This is a consequence of [Valentine (64), Theorem 1.10] and the Jordan Curve Theorem.

[7]Polar angles are measured in the usual way, counterclockwise from the x-axis.

[8]That is, if the interior is nonempty. [Benson (66), exercise 25.15]. "Interior" refers to the relative (subspace) topology. The convex hull of two distinct points in

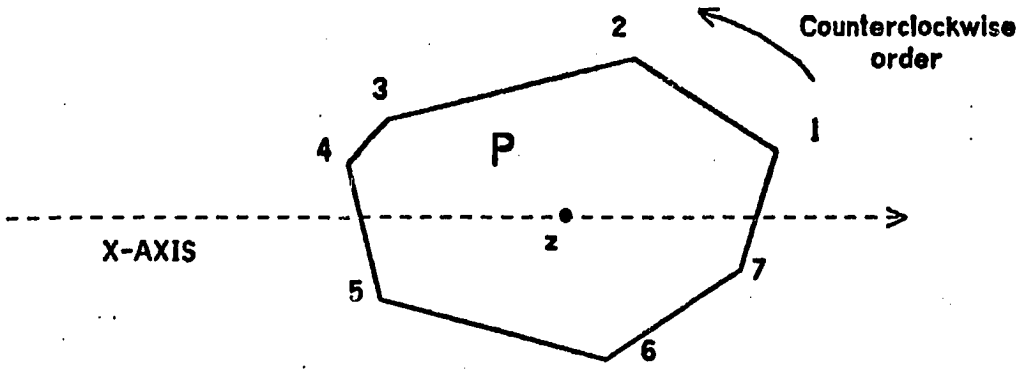


Figure 3.4: The vertices of P occur in sorted order about x.

The centroid of a set of N points in k dimensions can be computed trivially in O(Nk) arithmetic operations.

A different method of finding an interior point is due to Graham, who observes that in the plane the centroid of any three non-collinear points will suffice [Graham (72)]. He begins with two arbitrary points and examines the remaining N-2 points in turn, looking for one that is not on the line determined by the first two. This process uses O(N) time at worst, but almost always takes only constant time -- if the points are drawn from an absolutely continuous distribution, then with probability one the first three points are non-collinear [Efron (65)].

It is now evident how to proceed if we are given the extreme points of a set S. In O(N) time we may find a point *r* that is interior to the hull.[9] It only remains to sort the extreme points by polar angle, using *r* as origin. We may do this by transforming the points to polar coordinates in O(N) time, then using O(N log N) time to sort, but

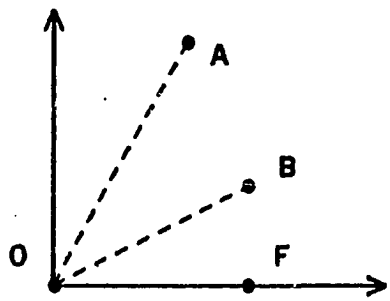
---

R^3 is a line segment whose interior is empty in the metric topology of R^3, but nonempty in the relative topology.

[9]This point is computed, so it is not necessarily one of the given points of S.

the explicit conversion to polar coordinates is not necessary. (This may be of some comfort to those readers who are still uneasy about allowing trigonometric functions in our model of computation.) Since sorting can be performed by pairwise comparisons, we need only determine which of two given angles is greater; we do not require their numerical values. Let us consider this problem in more detail because it illustrates a simple but important geometric "trick" that is useful in many applications.

Problem P3.3: (Angle Comparison) Given two points *A* and *B* in the plane, which one has greater polar angle?



B subtends a smaller angle than A iff it lies in the open wedge AOF.

Figure 3.5: Comparing Polar Angles Without Coordinate Conversion.

Let *F* be any point on the positive x-axis -- the point (1,0), for example. Point *B* subtends a smaller angle than *A* iff it lies in the open wedge *AOF*. (Refer to Figure 3.5.) To be in this wedge, *B* must lie both to the right of the directed line segment *OA* and to the left of segment *OF*. Whether a point (x_1,y_1) lies to the right or left of the directed segment from (x_2,y_2) to (x_3,y_3) can be determined in two multiplications and five addition/subtractions by evaluating

$$\text{AREA} = (x_1-x_3)(y_2-y_3) + (x_2-x_3)(y_1-y_3) \ , \qquad (3.2)$$

which gives twice the signed area of triangle 123. (See Appendix A.) AREA is positive if 1 lies to the left of 23, negative if 1 lies to the right, and zero if

the three points are collinear. The angle comparison involves just two of these tests. Another interpretation of the right-hand side of Equation (3.2) is as the z-component of the cross product of the two vectors OP and OQ, where O = (x_3,y_3), P = (x_1,y_1), and Q = (x_2,y_2).

The details of our first convex hull algorithm are now complete. We have shown that the problem is finite and that it can be solved in $O(N^4)$ time using only arithmetic operations and comparisons.

3.3. Graham's algorithm

An algorithm that runs in N^4 time will not allow us to process very much data. Assuming a processor speed of 10^6 operations per second, the solution of a 1000-point problem would take almost twelve days. Our goal is to reduce this time to a minimum, but could it be that the algorithm of the previous section is optimal? There is nothing in the mathematics so far presented to indicate otherwise. If improvements are to be made, they must come either by eliminating redundant computation or by taking a different theoretical approach. In this section we explore the possibility that our algorithm may be doing unnecessary work.

To determine whether a point lies in *some* triangle defined by a set of N points, is it necessary to try *all* such triangles? If not, there is some hope that the extreme points can be found in less than $O(N^4)$ time. R. L. Graham, in one of the first papers specifically concerned with finding an efficient geometric algorithm [Graham (72)], showed that performing the sort step first enables the extreme points to be found in *linear* time. The method he used turns out to be a very powerful tool in computational geometry.

Suppose that we have already found an interior point and transformed the others (using subtractions only) so that this point is at the origin. We now sort the N points lexicographically by polar angle and distance from the origin. In performing this sort we do not, of course, compute the actual distance between two points, since only a magnitude comparison is required. We could work with distance *squared*, avoiding the square root, but this case is even simpler. If several points

share the same polar angle, only the one farthest from the origin need be retained [Graham (72)]. Also, any points that are coincindent with the origin can be eliminated.

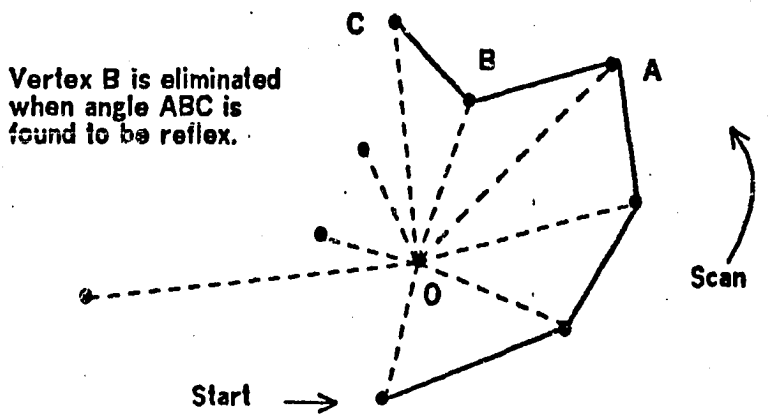


Figure 3.6: Beginning the Graham Scan.

After arranging the sorted points into a doubly-linked circular list and converting the resulting polygon to standard form[10], we have the situation depicted in Figure 3.6. Note that if a point is not an extreme point of the convex hull, then it is interior to some triangle *OAC*, where *A* and *C* are consecutive hull vertices. The essence of Graham's algorithm is a single scan around the ordered points, during which the interior points are eliminated. What remains are the hull vertices in the required order.

The scan begins at the point labelled START which, because of standard form, is the lowest point of the set and hence is certainly a hull vertex. We repeatedly examine triples of consecutive points in counterclockwise order to determine whether or not they define a *reflex angle*, one that is $\geq \pi$. If angle *ABC* is reflex, then *ABC* is said to be a *right turn*, otherwise it is a *left turn*. This can be

[10]The conversion to standard form can be accomplished because the polygon is *simple*. This point is covered more fully in Section 3.5.

determined easily by applying equation (3.2). From Property 2.3, we know that in traversing a convex polygon, we will make only left turns. If *ABC* is a left turn, we may check *BCD* next and no vertices are eliminated. If *ABC* is a right turn, then *B* cannot be an extreme point because it is interior to triangle *OAC*. Having eliminated *B*, we must back up the search to the vertex preceding *A*. This is how the scan progresses, based on the outcome of each angle test:

1. *ABC* is a right turn. Eliminate vertex *B* and let Z be the predecessor of *A*. If A ≠ START then check ZAC, otherwise check ACD.

2. *ABC* is a left turn. Advance the scan and check *BCD*.

The scan terminates when it advances all the way around to *A* again. Note that *A* is never eliminated because it is an extreme point. A simple argument shows that this scan only requires linear time. An angle test can be performed in a constant number of operations. After each test we either advance the scan (case 1), or delete a point (case 2). Since there are only N points in the set, the scan can not advance more than N times, nor can more than N points be deleted. This method of traversing the boundary of a polygon is so elegant and useful that we shall refer to it as the *Graham scan*. A more precise description of the algorithm is given below.

Algorithm A3.1: Graham's Algorithm ("Always go left")

1. Find an interior point p.

2. Using p as the origin of coordinates, sort the points lexicographically by polar angle and distance from p.

3. Arrange them into a circular doubly-linked list in standard form, with START pointing to the initial vertex. The RLINK associated with a node points to its successor in the list and LLINK points to its predecessor (as in [Knuth (68)], page 278). LLINK[START] = START.

4.
```
(Scan)
VPTR←START;
   WHILE (RLINK[VPTR] ≠ START) DO BEGIN
   IF the 3 vertices beginning at VPTR form a left turn
      THEN VPTR←RLINK[VPTR];
      ELSE BEGIN
         DELETE(RLINK[VPTR]);
         VPTR ← LLINK[VPTR];        (Backtrack.)
         END
      END

   (DELETE is a procedure that removes an item from a
      doubly-linked circular list.)
```

5. The list now contains the hull vertices in sorted order.

Theorem 3.7: [Graham (72)] The convex hull of N points points in the plane can be found in O(N log N) time and O(N) space using only arithmetic operations and comparisons.

Proof: From the above discussion, only arithmetics and comparisons are used in Graham's algorithm. Steps 1, 2, and 4 take linear time, while the sort step, 3, which dominates the computation, requires O(N log N) time. O(N) storage suffices for the linked list of points.

While this algorithm is a vast improvement over our first N^4 attempt, its optimality, which was not considered by Graham, remains in doubt. It is this question that we turn to in the next section.

3.4. A convex hull lower bound

> *"Thou hast set a bound that they may not pass over."*
>
> - Psalm 104.

A lower bound on the complexity of a problem defines the minimum number of operations, in a given model of computation, that suffice to solve it. The importance of obtaining such a result should not be underestimated because it may warn us that a proposed line of research will be fruitless. Having found an O(N log N) convex hull algorithm, how are we to proceed? In the absence of a lower bound we have no way of knowing whether a faster algorithm is possible and may spend considerable time trying to find one.

The fact that Graham's algorithm contains a sort step does not imply an Ω(N log N) lower bound for the convex hull problem. It does not even exclude the possibility that some clever implementor may find a way to perform the Graham scan without the initial sort step. What this means is that we should not look at specific algorithms for lower bound ideas, but concentrate instead on structural features of the problem itself. Theorem 3.5 should give us pause, for it states that the vertices of a convex polygon occur in sorted angular order. This seems to imply that any convex hull algorithm must be able to sort, a conclusion that we must demonstrate formally.

Theorem 3.8: Sorting is linear-time reducible to the convex hull problem; therefore, finding the convex hull of N points in the plane requires $\Omega(N \log N)$ time.[11]

Proof: We exhibit the reducibility; the conclusion follows from Theorem 2.1. Given N real numbers $x_1,\ldots,x_N$ all positive, we must show how a convex hull algorithm can be used to sort them with only linear overhead. Corresponding to the number x_i, we construct the point (x_i,x_i^2) and associate the number i with it. All of these points lie on the parabola $y = x^2$. The convex hull of this set, in standard form, will consist of a list of the points sorted by abscissa. One pass through the list will enable us to read off the x_i in order.[12]

Because the transformation involves only arithmetic operations, Theorem 3.8 holds in many computational models; namely, those in which multiplication is permitted and sorting is known to require $\Omega(N \log N)$ time. It applies in all dimensions greater than one[13], as can be seen by considering any set of N distinct points on the intersection of the hyperplane $x = 0$ and the unit hypersphere. While this lower bound is quite elementary, it is included here to introduce reducibility as a way of establishing the connection between a geometric problem and a combinatorial one.

Suppose now that we are only interested in finding the extreme points of a set (Problem P3.2). The reader should see immediately that the above theorem fails to provide a lower bound, since it is based completely on the fact that the hull vertices are sorted and is of no use if we drop the ordering requirement. It is still unknown whether PLANAR EXTREME POINTS is easier than CONVEX HULL.

---

[11] A similar result appeared as a question on the 1972 Yale Department of Computer Science Qualifying Examination.

[12] The author originally proved this theorem by mapping the x_i onto the unit circle. The parabola mapping, suggested by Stan Eisenstat, is superior because it is rational and requires only a single arithmetic operation.

[13] The convex hull of a set of points in one dimension is the smallest interval that contains them, which can be found in linear time.

3.5. A diversion -- simple closed polygonal paths.

Before continuing with more convex hull material, let us look at an unexpected dividend of Graham's algorithm.

Problem P3.4: (Simple Polygonal Path [Gemignani (66)]) Given N points in the plane, are they the vertices of a simple polygon? If so, construct it.

The number of distinct polygons that result from all possible orderings of the points is at most (N-1)!/2. In the set of N! *a priori* orderings, each polygon appears N times because of cyclic shifts and twice because clockwise and counterclockwise orderings yield the same polygon. In Chapter 5, an O(N log N) algorithm is presented to test whether a polygon is simple. Thus we could generate all of these polygons and test to see if any are simple, so the problem has a finite algorithm, but its running time is O(N! N log N).

Theorem 3.9: The shortest closed route through N points in the plane is non self-intersecting unless all the points are collinear.[14]

Finding this shortest circuit is known as the *Euclidean Traveling Salesman Problem* and is the subject of Section 6.1.5. The problem is NP-complete, and the best known algorithm requires $O(N^2 2^N)$ time [Bellman (62)]. Since a shortest circuit always exists, however, we now know that *every* set of points in the plane is the vertex set of some simple polygon.[15] A remarkably fast procedure for finding a simple polygon on any set of points is suggested by the first three steps of Graham's algorithm, in which we compute an interior point and sort the given points by polar angle and radius. If the polar angles of all the points are distinct, the

[14]American Mathematical Monthly, Problem E880, April, 1950, page 261. [Sanders (76)] contains a characterization of those metric spaces in which the theorem remains true.

[15]Except for degenerate sets, as usual, in which all points are collinear.

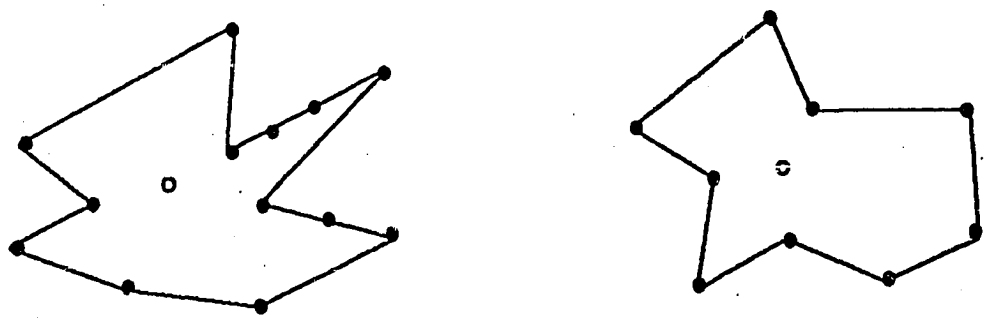


Figure 3.7: Constructing a simple polygon on a finite plane set.

resulting list forms a simple polygon, since the points occur in strictly increasing angular order and thus edge intersections are precluded. If several points share a common polar angle, it is necessary to sort them into a sequence by radius. Suppose that there are k distinct polar angles and that n_j points share the j^{th} angle. Let these points be arranged into a chain C_j such that the first point has least radius and the last point has largest radius. Then concatenating the chains $C_1 \ldots C_k$ in order and joining the last vertex of C_k to the first vertex of C_1 will produce a simple polygon. (See Figure 3.7.)

Theorem 3.10: A simple closed polygonal path through N points in the plane can be found in O(N log N) time, and this is optimal.[16]

Proof: The algorithm described above is easily seen to run in O(N log N) time; the fact that it always produces an SCPP follows from a slight modification of the proof of Theorem 3.5. To demonstrate optimality, we show that

$$\text{SORTING} \propto_N \text{SCPP}$$

[16][Reingold (72b)] contains a statement (without proof) of this theorem and a claim that the problem cannot be solved at all if only comparisons are allowed.

Consider an unordered set in the plane consisting of N points x_i on the x-axis and the single point (0,1). There is only one SCPP through these points, namely the one that begins at (0,1), passes through all of the x_i *in increasing order*, and returns to (0,1). Thus any algorithm which finds this SCPP must sort the x_i.

3.6. Jarvis's Algorithm

Even though we have shown that Graham's algorithm is optimal, there are still many reasons for continuing to study the convex hull problem:

1. The algorithm is optimal in the *worst case* sense, but we have not yet analyzed its expected performance.

2. Because it is based on Theorem 3.5 which applies only in the plane, the algorithm does not generalize to higher dimensions.

3. It is not on-line since all points must be sorted before processing begins.

4. For a parallel environment, we would prefer a recursive algorithm that allows the data to be split into smaller subproblems.

Just as the study of sorting reveals that no single algorithm is best for all applications, we will find the same to be true of hull-finding. Let us continue to explore combinatorial geometry, looking for ideas that may lead to other hull algorithms.

A polygon can equally well be specified by giving its edges in order as by giving its vertices. In the convex hull problem, we have concentrated so far on isolating the extreme points. If we try instead to identify the hull edges, will a practical algorithm result? Given a set of points, it is fairly difficult to determine quickly whether a specific point is extreme or not. Given *two* points, though, it is straightforward to test whether the line segment joining them is a hull edge.

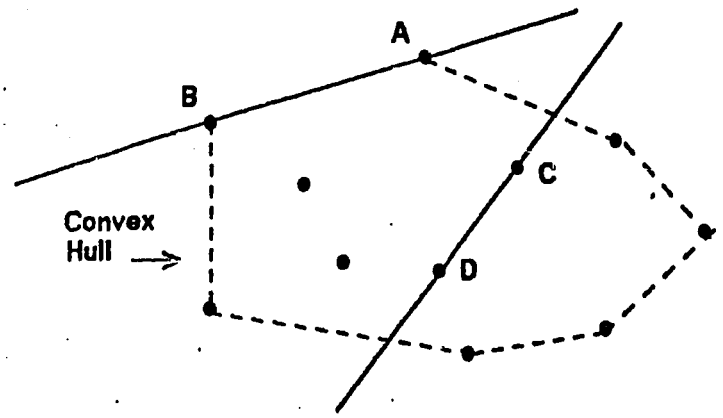


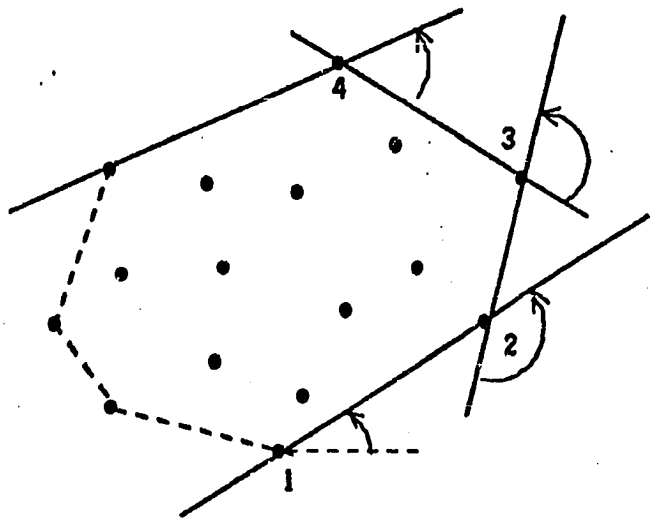
Figure 3.8: A hull edge cannot separate the set.

Theorem 3.11: [Stoer (70), Theorem 2.4.7] The line segment L defined by two points of the set is an edge of the convex hull iff all other points of the set lie on L or to one side of it.

There are $\binom{N}{2} = O(N^2)$ lines determined by all pairs of N points. For each of these lines we may examine the remaining N-2 points and apply equation (3.2) to determine in linear time whether the line meets the criteria of the theorem. Thus in $O(N^3)$ time, we are able to find all pairs of points that define hull edges. It is then a simple matter to arrange these into a list of consecutive vertices.

Jarvis has observed that this algorithm can be improved if we note that, once we established that segment *AB* is a hull edge, then another edge must exist with *B* as an endpoint [Jarvis (73)]. His paper shows how to use this fact to reduce the time required to $O(N^2)$ and contains a number of other ideas that are worth treating in detail.

Assume that we have found the lexicographically lowest point A of the set as in



The algorithm of Jarvis finds successive hull vertices by repeatedly turning angles.

Each new vertex is discovered in O(N) time.

Figure 3.9: The Jarvis March for Constructing the Convex Hull.

Section 2.2. (See Figure 3.9.) This point is certainly a hull vertex. We now wish to find the next consecutive vertex B on the convex hull. This point will be the one that has the least polar angle with respect to A as origin. If we now take B as origin and let the vector AB define the direction of zero polar angle, then the next point C is the one that subtends the smallest polar angle about B, and each successive point can be found similarly in linear time. Jarvis's algorithm marches around the convex hull, finding extreme points in order, one at a time. This process will be referred to as the *Jarvis March*. Each new point is found by measuring angles with respect to the last direction traversed. As we have already seen in Section 3.2, the smallest angle can be found with arithmetics and comparisons alone, without actually computing any polar angles.

Since all N points of a set may lie on its convex hull, and Jarvis's algorithm expends linear time to find each hull point, its worst-case running time is $O(N^2)$, which is inferior to Graham's. However, if h is the actual number of vertices on the convex hull, Jarvis's algorithm runs in O(hN) time which is very efficient if *h* is known in advance to be small. For example, if the hull of the set is a polygon of any constant number of sides, we can find it in *linear* time.

Finding successive hull vertices by repeatedly turning angles is analogous to wrapping a two-dimensional package, but the idea generalizes to k dimensions. Such a "gift-wrapping" algorithm was given [Chand (70a,b)] even before the appearance of Jarvis's paper. In three dimensions it suffices to identify all of the hull edges. Given one edge E, we may find an adjacent edge F by rotating the plane containing E until a new point of the set is reached. If several points are reached simultaneously, they all lie on the same hull face and a two-dimensional problem results for that face. Since each rotation takes only linear time and a convex polyhedron on N vertices has at most 3N-6 edges, the convex hull can be found in $O(N^2)$ time.

In dimensions higher than three, the distinction between finding the hull (including all hyperfaces) and just isolating the extreme points becomes important because the convex hull of N points in k dimensions may consist of up to $\binom{N}{k} = O(N^k)$ hyperfaces. Thus any convex hull algorithm must, in the worst case, exhibit performance that is exponential in dimension. The Chand-Kapur algorithm is a way of organizing the search for hyperfaces so as to reduce overhead.

3.6.1 Average-case analysis

Graham's convex hull algorithm always requires O(N log N) time[17], regardless of the data, because its first step is to sort the input. Jarvis's algorithm, on the other hand, uses time that varies between linear and quadratic, so it makes sense to ask how much time it can be *expected* to take. The answer to this question will take us into the difficult but fascinating field of stochastic geometry, where we will see some of the difficulties associated with analyzing the average case of geometric algorithms.

Since Jarvis's algorithm runs in O(hN) time, where h is the number of hull vertices, to analyze its average-case performance we need only compute E(h), the expected value of h. In order to do this, we must make some assumption about the probability

[17] In any model of computation for which sorting is known to require this much time.

distribution of the input points. This problem brings us into the province of stochastic geometry, which deals with the properties of random geometric objects and is an essential tool for dealing with expected-time analysis[18]. We would like to be able to say, "Given N points chosen uniformly in the plane...", but technical difficulties make this impossible -- elements can be chosen uniformly only from a set of bounded Lebesgue measure [Kendall (63)] -- so we are forced to specify a particular figure from which the points are to be selected. Fortunately, the problem of calculating E(h) has received a good deal of attention in the statistical literature, and we quote below a number of theorems that will be relevant to the analysis of a number of geometric algorithms.

Theorem 3.12: [Renyi and Sulanke (63)] If N points are chosen uniformly and independently at random from a plane convex *r*-gon, then as $N \to \infty$,

$$E(h) = (2r/3)(\gamma + \log_e N) + O(1),$$

where γ denotes Euler's constant.

Theorem 3.13: [Raynaud (70)] If N points are chosen uniformly and independently at random from the interior of a k-dimensional hypersphere, then as $N \to \infty$, E(f), the expected number of *hyperfaces* of the convex hull, is given asymptotically by

$$E(f) = O(N^{(k-1)/(k+1)})$$

This implies that

$E(h) = O(N^{1/3})$ for points chosen uniformly in a circle, and

$E(h) = O(N^{1/2})$ for points chosen uniformly in a sphere.

[18] Consult [Santalo (76)] for a monumental compilation of results in this field.

Theorem 3.14: [Raynaud (70)] If N points are chosen independently from a k-dimensional normal distribution, then as $N \to \infty$, the asymptotic behavior of E(h) is given by

$$E(h) = O((\log N)^{(k-1)/2}) .$$

Theorem 3.15: [Bentley (77c)] If N points in k dimensions have their components chosen independently from any set of continuous distributions (possibly different for each component), then

$$E(h) = O((\log N)^{k-1}) .$$

Many distributions satisfy the conditions of this theorem, including the uniform distribution over a hypercube.

The surprising qualitative behavior of the hulls of random sets can be understood intuitively as follows: For uniform sampling within any bounded figure *F*, the hull of a random set tends to assume the shape of the boundary of *F*. For a polygon, points accumulating in the "corners" cause the resulting hull to have very few vertices. Because the circle has no corners, the expected number of hull vertices is comparatively high, although the author knows of no elementary explanation of the $N^{1/3}$ phenomenon.[19]

[19]The expected number of hull vertices of a set of N points can be expressed as the integral of the N^{th} power of the integral of a probability density. [Efron (65)] What is of interest is not the value of this quantity, but its asymptotic dependence on N.

It follows directly from these results that the expected time used by Jarvis's algorithm can be described by the following table:

Distribution	Average-Case
Uniform in a convex polygon	$O(N \log N)$
Uniform in a circle	$O(N^{4/3})$
Normal in the plane	$O(N (\log N)^{1/2})$

Table 3.1. Average-Case Behavior of Jarvis's Algorithm.

Note that for the normal distribution, Jarvis's algorithm can be expected to take slightly less time than Graham's.

All of the distributions considered in this section have the property that the expected number of extreme points in a sample of size N is $O(N^p)$, for some constant $p < 1$. We shall refer to these as *N^p-distributions.*

Jarvis's original paper outlines some improvements to his procedure that reduce its running time considerably.[20] They are based on the fact that once a point has been found to be interior to the convex hull, it can be eliminated from further consideration. Suppose that we have already found hull points A, B, and C and are searching for vertex D, the successor of C. To do this we will examine N-3 points, *but all those interior to triangle ABC can be deleted during the search.* Likewise, when we scan for vertex E, all points within triangle ACD can be removed forever. Note that the test for inclusion in triangle ACD reduces to asking on which side of line AD the point lies. Figure 3.10 illustrates this process.

[20]These modifications were not taken into account in our average-case analysis because they do not affect the asymptotic average running time for centrally symmetric distributions.

During the scan for vertex D, all points interior to triangle ABC are eliminated from further consideration.

In the scan for E, the points inside triangle ACD are removed, etc.

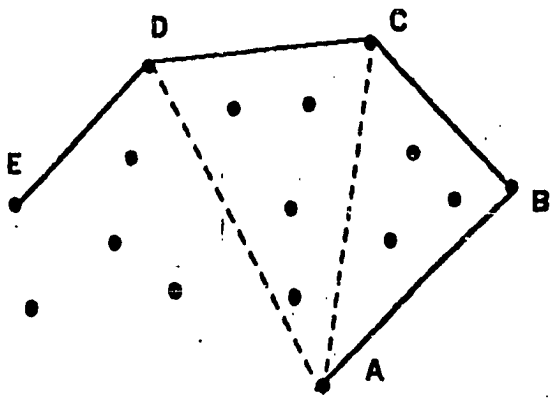


Figure 3.10: Eliminating points that are interior to the hull

3.7. A linear expected-time algorithm

We now develop in detail a linear expected-time convex hull algorithm that is also worst-case optimal. It is based on a divide-and-conquer idea that is simple but remarkably powerful, and which generalizes to higher dimensions.

A first step in applying the method of divide-and-conquer, which we shall refer to from here on as D&C, is to invoke the principle of balancing [Aho (74), page 65], which suggests that a computational problem should be divided into subproblems of nearly equal size. Suppose that in the convex hull problem, we have split the input into two parts, A and B, each containing half of the points. If we now find hull(A) and hull(B) separately but recursively, how much additional work is needed to form hull(A ∪ B), that is, the hull of the original set? To answer this we may use the relation

$$\text{hull}(A \cup B) = \text{hull}(\text{hull}(A) \cup \text{hull}(B)) \quad . \tag{3.3}$$

HULL(S) =
HULL(HULL(A) U HULL(B))

By dividing S into two subsets and finding their hulls recursively we can reduce the problem to finding the hull of the union of two convex polygons.

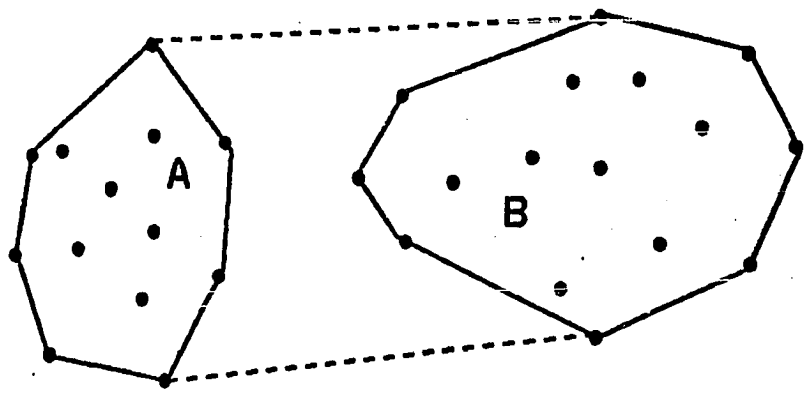


Figure 3.11: Forming the hull by divide-and-conquer.

While at first glance equation (3.3) seems to involve more work than just finding the hull of the set directly, it is crucial to note that hull(A) and hull(B) are *convex polygons*, not just unordered sets of points. (See Figure 3.11.)

Problem P3.5: (Hull of Union of Convex Polygons) Given two convex polygons, find the convex hull of their union.

Algorithm A3.2: Convex Hull

1. Partition the original set S arbitrarily, but as equally as possible, into two subsets A and B.

2. Recursively find the convex hulls of A and B.

3. Merge the two hulls together to form hull(S).

Let U(N) denote the time needed to find the hull of the union of two convex polygons, each having N/2 vertices. If T(N) is the time required to find the convex hull of a set of N points, then applying equation (3.3) gives

$$T(N) \leq 2T(N/2) + U(N) \quad , \quad T(1) = \text{const.}, \tag{3.4}$$

whose solution depends on the form of U(N):

Lemma 3.1: If T(N) obeys the recurrence relation (3.4),

If $U(N) = O(N)$, then $T(N) = O(N \log N)$.

If $U(N) = O(N/\log N)$, $T(N) = O(N \log \log N)$.

If $U(N) \leq O(N^p)$, $p < 1$, then $T(N) = O(N)$.

Proof: An easy inductive exercise as in [Aho (74), Chapter 2].

It now remains to develop an algorithm for forming the hull of the union of two convex polygons. We will make use of the property that the vertices of a convex polygon occur in sorted order about any interior point (Theorem 3.5) and the fact that the Graham scan runs in linear time if presented with a list of vertices that are sorted by polar angle.

Algorithm A3.3: Hull of the union of convex polygons

1. Find a point *p* that is interior to A. (For example, the centroid of any three vertices of A.) This point *p* will be interior to hull(A U B).

2. Determine whether *p* is interior to B. This can be done in O(N) time by the method of Section 4.2. If *p* is not interior, go to step 4.

3. (*p* is interior to B; see Figure 3.12.). By Theorem 3.5, the vertices of

both A and B occur in sorted angular order about p. We may then merge the lists in O(N) time [Knuth (73)] to obtain a sorted list of the vertices of *both* A and B. Go to step 5.

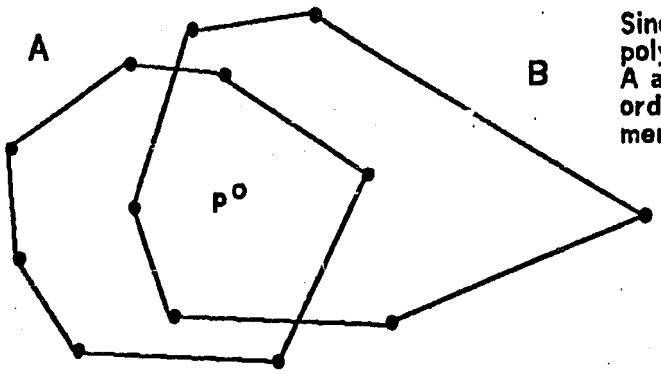
4. (p is not interior to B; see Figure 3.13.) As seen from p, polygon B lies in a wedge whose apex angle is $\leq \pi$. This wedge is defined by two vertices u and v of B, which can be found in linear time by the following procedure: Construct a horizontal line through p. If this line intersects B to the right of p, then B lies in the wedge determined by the vertices of B that have greatest polar angle $< \pi/2$ and least polar angle $> 3\pi/2$. The case in which B lies to the left of p is similar. If the horizontal line through p does not intersect B and B lies above it, the wedge is determined by the vertices that subtend the largest and smallest polar angles about p. (The case in which B lies below is analogous.) The two vertices defining the wedge B into two chains of vertices that are monotonic in polar angle about p, one increasing in angle, the other decreasing. Polygon A together with the two chains of B constitute three sorted lists that contain a total of N elements. These can be merged in O(N) time[21] to form a list of the vertices of A ∪ B, sorted about p.

5. The Graham scan (step 4 of Algorithm A3.1) can now be performed. This requires only linear time to yield the hull of A ∪ B.

Theorem 3.16: [Shamos (75a)] Algorithm A3.3 correctly finds the convex hull of the union of a convex n-gon and a convex m-gon in O(n + m) time.

Proof: Once we have the vertices in sorted angular order about p, correctness follows from Graham's algorithm. Step 1 requires only constant time and correctly finds an interior point of A by Theorem 3.6. Step 2 requires O(n) time and its correctness is a consequence of Theorem 4.2. Step 3 or 4 (whichever is executed) requires O(n + m) time. Since only m + n vertices remain at Step 5, O(n + m) time suffices for the Graham scan (Section 3.3).

[21] Since two sorted lists of length N can be merged in O(N) time, the same is true of three lists.



Since p lies inside both polygons, the vertices of A and B occur in sorted order about p and can be merged in linear time.

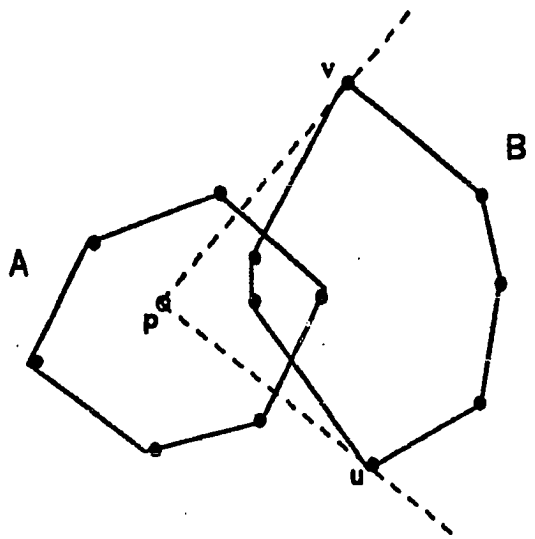
Figure 3.12: Point p lies inside B.

Theorem 3.17: Algorithm A3.2 runs in O(N log N) time, in the worst case.

Proof: Lemma 3.1 and Theorem 3.16.

The recurrence relation (3.4) describes the worst-case running time of our convex hull algorithm in any dimension k, if U(N) is taken to be the time required to form the hull of the union of two convex polytopes in R^k, each having N vertices. Preparata and Hong [Preparata (77b)] discovered a variant of Algorithm A3.3 independently and have used it to show that U(N) = O(N) even in three dimensions, and they are thus able to obtain an O(N log N) convex hull algorithm in R^3.

We now show that Algorithm A3.3 runs in linear expected time over a wide class of probability distributions, if some care is taken in its implementation. The proof is based on the fact that the expected time needed to merge the two subproblem



As seen from p, polygon B lies in a wedge defined by vertices u and v, which partition B into two chains of vertices that may be merged with the vertices of A in linear time.

Figure 3.13: Point p is exterior to B.

solutions is *sublinear*, requiring only $O(N^p)$ time for some $p < 1$. Since the expectation operator distributes over addition, we may rewrite (3.4) as

$$T^*(N) \leq 2T^*(N/2) + U^*(N) \qquad (3.5)$$

where starred quantities denote expected values. This equation will be correct for a particular probability distribution if we are able to guarantee that the subproblems are also from this distribution, which will justify writing the *same* function T^* on both sides. This can be assured by assigning points *randomly* to the subproblems, so the analysis reduces to determining $U^*(N)$.

By Theorem 3.16, U(N) is linear in the total number of vertices in the hulls to be merged, so $U^*(N)$ is linear in the *expected* number of vertices involved. All of the distributions discussed in Section 3.6.1 have the property that $E(h) = O(N^p)$, for some $p < 1$. Thus,

$$T^*(N) \leq 2T^*(N/2) + O(N^p) = O(N) . \qquad (3.6)$$

Let us review the algorithm, this time including implementation details:

Algorithm A3.4: Linear Expected-time Convex Hull.

1. Apply a random permutation to the input points and arrange them in a 2 x N array of x- and y-coordinates. Note that a subproblem may be specified by giving two pointers into this array, denoting the left and right indices of a sequence of points, and that each such subproblem is random [Knuth (71)].

2. Recursively find the convex hulls of the first and last N/2 points. In the recursive calls to the hull procedure, it is essential to pass only two pointers to the point array. Copying entire subproblems would use O(N log N) time overall. It is essential to avoid this.

3. The solutions obtained to the subproblems in Step 2 are expected to be *small*, i.e., for a large class of distributions, only $O(N^p)$ points remain. These hulls can be merged in $O(N^p)$ expected time by using Algorithm A3.3. Because they are small, the subproblem solutions may themselves be passed by copy to the hull-union procedure, which is not recursive.

The algorithm can also be implemented bottom-up to avoid explicit recursion by working on sets of four points, then eight, etc. This would be the implementation of choice, but the algorithm is much easier to understand recursively.

Theorem 3.18: [Bentley and Shamos (77b)] For the distributions discussed in Section 3.6.1, the convex hull of a sample of N points can be found in O(N) expected time in both two and three dimensions.

The result in three dimensions follows from the linear polyhedron merge of [Preparata (77b)].

3.8. The On-Line Problem

Each of the convex hull algorithms we have examined thus far requires all of the data points to be present before any processing begins. In many geometric applications, particularly those that run in real-time, this condition cannot be met and some computation must be done as the points are being received. In general, an algorithm that cannot look ahead at its input is referred to as *on-line*, while one that operates on all the data collectively is termed *off-line*.

Problem P3.6: (On-Line Convex Hull) Given N points in the plane, $p_1,\ldots, p_N$, find their convex hull in such a way that after p_i is processed, the hull of the first i points can be output in O(i) time.

The algorithm must maintain some representation of the hull and update it as points arrive; the question is whether this can be done without sacrificing O(N log N) worst-case running time for processing the entire set.

The answer was recently shown by Preparata to be affirmative:

Theorem 3.19: [Preparata (77e)] The convex hull of N points in the plane can be found on-line with an interpoint processing delay of at most O(log N).

3.9. The Hull of a Simple Polygon

Sklansky has shown that a scan similar to Graham's can be applied to a *simple* polygon.

Theorem 3.20: [Sklansky (72)] The convex hull of a *simple* polygon can be found in O(N) time.

3.10. Convex Hull Applications

"You know my methods, apply them."

- Sherlock Holmes.

This section is devoted to a discussion of applications that require computation of the convex hull. New problems will be formulated and treated as they arise in these applications. Their variety should convince the reader that the hull problem is important, both in practice and as a fundamental tool in computational geometry.

3.10.1 Statistics

The connection between geometry and statistics is a close one because a multivariate statistical sample can be viewed as a set of points in Euclidean space. In this setting, many problems in statistics become purely geometric ones. For example, linear regression asks for a hyperplane of best fit in a specified norm. Certain problems in voting theory reduce to finding which k-dimensional hypersphere contains the most points of a set.[22] A survey of geometric techniques in statistics is given in [Shamos (76a)]. Determining the convex hull is a basic step in several statistical problems which we treat separately in the next few paragraphs.

3.10.2 Robust Estimation

A central problem in statistics is to estimate a population parameter, such as the mean, by observing only a small sample drawn randomly from the population. We say that a function t is an *unbiased* estimator of a parameter P if E[t] = P, that is, the expected value of t is precisely P [Hoel (71)]. While the sample mean is an unbiased estimator of the population mean, it is extremely sensitive to *outliers*,

[22] [Johnson (77)]. The word "reduce" may be misleading here because the final problem is NP-complete.

observations that lie abnormally far from most of the others. It is desirable to reduce the effects of outliers because they often represent spurious data that would otherwise introduce errors in the analysis. A related property that a good estimator should enjoy is that of *robustness*, or insensitivity to deviations from the assumed population distribution. Many such estimators have been proposed [Andrews (72a)]. An important class, known as the Gastwirth estimators [Gastwirth (66)], are weighted means of symmetrically-placed order statistics and are based on the fact that we tend to trust observations more the closer they are to the "center" of the sample.

Consider N points on the line. A simple method of removing suspected outliers is to remove the upper and lower α-fraction of the points and to take the average of the remainder. This is known as the α-trimmed mean, and is a special case of the Gastwirth estimator,

$$T = \sum_{i=1}^{N} w_i x_{(i)} , \qquad \sum_{i=1}^{N} w_i = 1 , \tag{3.7}$$

where $x_{(i)}$ denotes the i^{th} smallest of the x_j. The α-trimmed mean is just the case

$$w_i = 1/(1 + (1-2\alpha)N) , \quad \alpha N \le i \le (1-\alpha)N .$$

$\alpha = 0.2$ (The upper and lower 20% of the points are removed.)

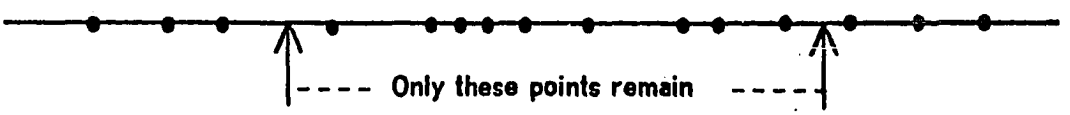


Figure 3.14: The α-trimmed mean.

Any trimmed mean can be computed in O(N) time (using a linear selection algorithm) and any Gastwirth estimator in O(N log N) time (by sorting), but what are their analogs in higher dimensions? Tukey has suggested a procedure known as "shelling", or "peeling", which involves stripping away the convex hull of the set,

then removing the convex hull of the remainder, and continuing until only $(1-2\alpha)N$ points remain [Huber (72)] This procedure motivates our next definition and problem:

Definition 3.5: The *depth* of a point p in a set S is the number of convex hulls that have to be stripped from S until p is removed. Points which lie on a hull but are not extreme have the same depth as extreme points. The depth of S is the depth of its deepest point.

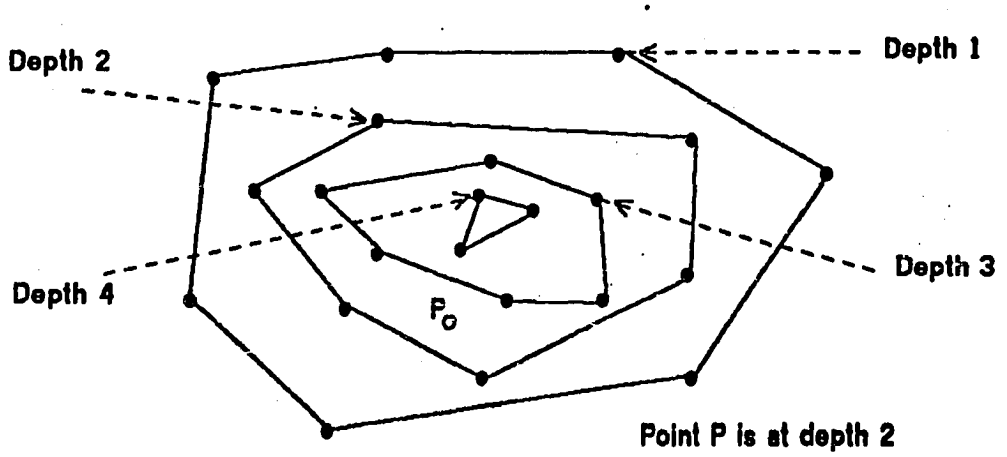


Figure 3.15: The Depth of a Point

Problem P3.7: (Depth of a Set) Given a set of N points in the plane, find the depth of each point.

Theorem 3.21: Any algorithm that determines the depth of each point in a set must make $\Omega(N \log N)$ comparisons in the worst case.

Proof: Consider a one-dimensional set. Knowing the depth of each point, we can sort the set in only O(N) additional comparisons. In any dimension we can force a depth algorithm to sort and thus prove that

SORTING $\propto_N$ DEPTH .

Suppose we are given N real numbers $x_i \geq 0$. In two dimensions, create the set of 4N points $(x_i,0) \cup (0,x_i) \cup (-x_i,0) \cup (0,-x_i)$. It is easily seen that the depth number of a point is the rank of its corresponding x_i. The construction generalizes to higher dimensions but requires $2^k N$ points in dimension k.

The question remains as to whether the depth of a *set* can be found faster than the depths of all of its points. The following results show that under a restricted model of computation this is impossible even in one dimension. We will explore the connection between depth and a basic question in set theory.

Problem P3.8: (Element Uniqueness) Given N real numbers, are they all distinct?

The problem may be solved easily in O(N log N) comparisons by sorting, but there is no obvious way to show that sorting is required.

Theorem 3.22: [Dobkin (76b)] Determining whether N real numbers are distinct requires Ω(N log N) comparisons if only polynomial functions of bounded degree can be computed.

Theorem 3.23: Any algorithm which finds the depth of a one-dimensional set must make at least Ω(N log N) comparisons in the worst case.

Proof: We show that ELEMENT UNIQUENESS $\propto_N$ SET DEPTH, and the result follows from Theorem 3.22. (See Section 5.4.2.) Given N real numbers x_i, find the depth of the one-dimensional set S they determine. If N is odd, then the x_i are distinct iff DEPTH = (N-1)/2. If N is even, then let M be a number that is larger than any element of S. Such an element can certainly be found in O(N) time by computing 1 + max{S}. Now the x_i are distinct iff DEPTH(S $\cup$ M) = N/2.

Unfortunately, no known algorithm achieves the $\Omega(N \log N)$ lower bound in two or more dimensions. The procedure below runs in $O(N^2)$ time and operates by repeated application of the Jarvis march (Section 3.6).

Algorithm A3.5: Set Depth

```
DEPTH ← 0;
WHILE  S is nonempty  DO BEGIN
    DEPTH ← DEPTH + 1;
    Find Hull(S) using Jarvis' Algorithm;
    Label all hull vertices with DEPTH and
        delete them from S;
    END
```

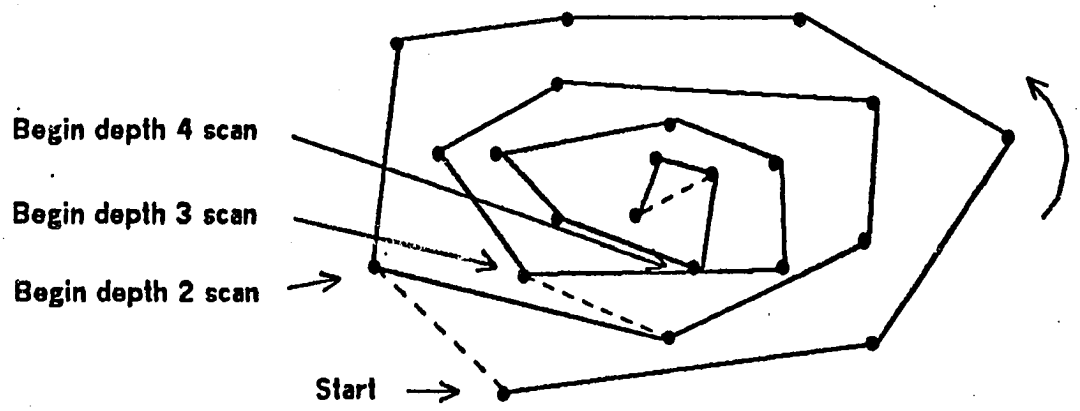


Figure 3.16: Finding depths by repeating the Jarvis March.

Theorem 3.24: Algorithm A3.5 runs in $O(N^2)$ time in the worst case.

Proof:

Let h_i be the number of vertices of depth i. By the results of Section 3.6, the i^{th} iteration of the WHILE-loop can be performed in ch_iN_i time, where N_i is the number of vertices remaining in S before the i^{th} iteration and c is a constant independent of N:

$$N_i = N - \sum_{j=1}^{i-1} h_j \quad .$$

The total run time is given by

$$T(N) = \sum_i ch_iN_i = \sum_i ch_i \left(N - \sum_{j=1}^{i-1} h_j \right) \leq O(N^2) .$$

To use this algorithm for peeling, we need run it only until $2\alpha N$ points have been trimmed. An O(N log N) algorithm would be of great interest for this problem. There are indications that peeling may also be useful for detecting outliers [Friedman (78)].

3.10.3 Chebyshev approximation

In this section we study Chebyshev approximation of a finite set of points in the plane and obtain a fast algorithm for the case in which the approximating function is also linear. The distance between a point and the value of the approximating function at the point is called the *deviation*. In the Chebyshev, or L_∞, norm this is the y-distance between the point and the approximating line $y = ax + b$. We want to minimize the maximum deviation by finding parameters a and b defined by

$$\min_{a,b} \max_i \; | y_i - ax_i - b | \; . \qquad (3.8)$$

The problem is now completely geometric:

Problem P3.9: (Chebyshev Approximation) Given N points in the plane, find the line L that minimizes the maximum y-distance to any point.[23]

Naturally, this problem has been studied a great deal, and a combinatorial characterization of the minimizing line exists:

Theorem 3.25: [Rice (64), Corollary 3-5] A line L is the Chebyshev approximant to a finite set S of points in the plane iff it maximizes the deviation of a best Chebyshev approximation among all subsets consisting of 3 points of S.

Any set of points satisfying Theorem 3.25 will be known as *Chebyshev points*. The best approximation to three points can be found by solving a system of linear equations in the three variables *a*, *b*, and *d*, the deviation:[24]

$$\begin{aligned} ax_i + b - y_i &= -d \\ ax_j + b - y_j &= d \\ ax_k + b - y_k &= -d \end{aligned}$$

Since this set of equations can be solved in a constant number of operations, Theorem 3.25 provides us with an $O(N^3)$ algorithm for Chebyshev approximation.

We can do much better by exploring the relationship between the minimax approximation and the convex hull of the given point set.[25] In this connection it will be useful to define a different geometric device:

Definition 3.6: A line L is a *line of support* of set S if it meets the boundary of S and S lies entirely on one side of L.

---

[23]The required line is unique if all of the points have distinct x-coordinates, which we assume throughout this section. See [Rice (64), page 60].

[24]Again, only if x_i, x_j, and x_k are distinct. This follows directly from the Chebyshev Equioscillation Theorem. [Davis (63), Theorem. 7.6.2].

[25]The author is indebted to Gideon Yuval for this suggestion.

In Figure 3.17, line L supports the polygon, but lines M and N do not.

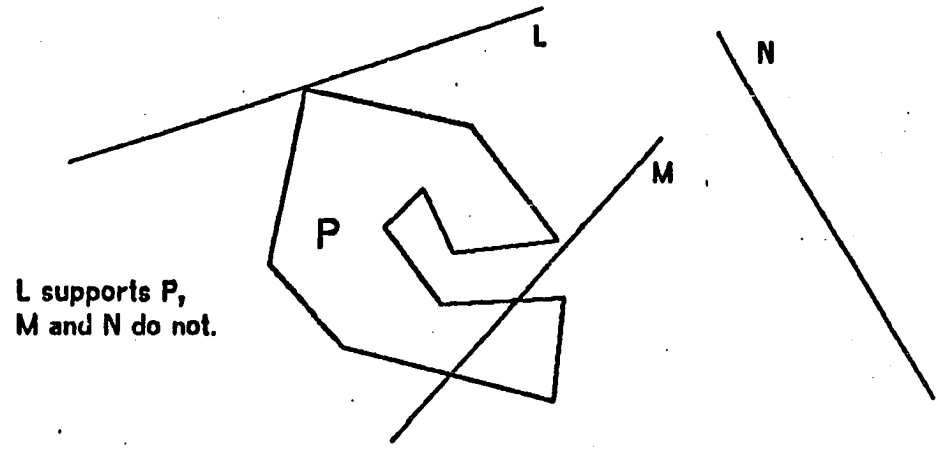


Figure 3.17: Line L is a supporting line of P.

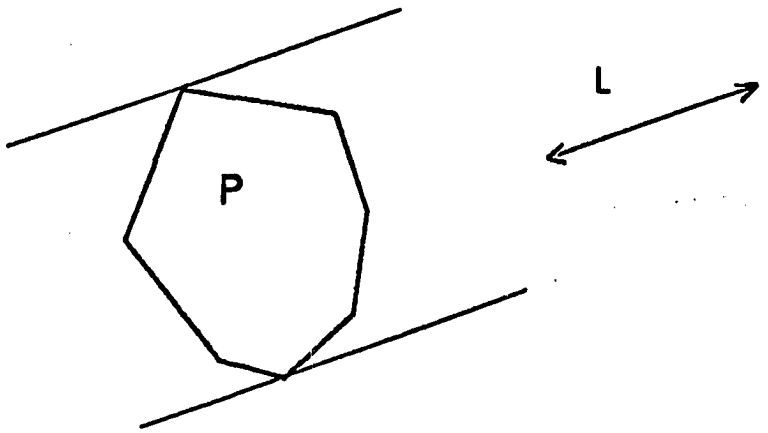


Figure 3.18: The two lines of support parallel to L.

Theorem 3.26: Every line of support of a closed and bounded convex set passes through an extreme point.

Proof: Let L support S. Consider the intersection T of L and S. Since S is closed and bounded, T is either an interval or a single point. If T is a single point it must be extreme or there would have to be points of S on both sides of L. If T is an interval, its endpoints are extreme.

Theorem 3.27: [Yaglom (61), page 8] Parallel to a given direction, a bounded convex figure possesses exactly two lines of support.

Theorem 3.28: The supporting lines of P parallel to a given direction can be found in O(log N) time.

Proof: Let the given direction lie at angle β with respect to the x-axis. A line of support through vertex v_i lies between edges $v_{i-1}v_i$ and v_iv_{i+1} in angle. Since the edges of a convex polygon occur in sorted angular order, we may find the positions of angles β and $\pi + \beta$ in O(log N) time by binary search, if the vertices are stored in a linear list or balanced tree.

The following theorem enables us to work only with hull points, which is not the case in L_1 and L_2 approximation [Shamos (76a)]:

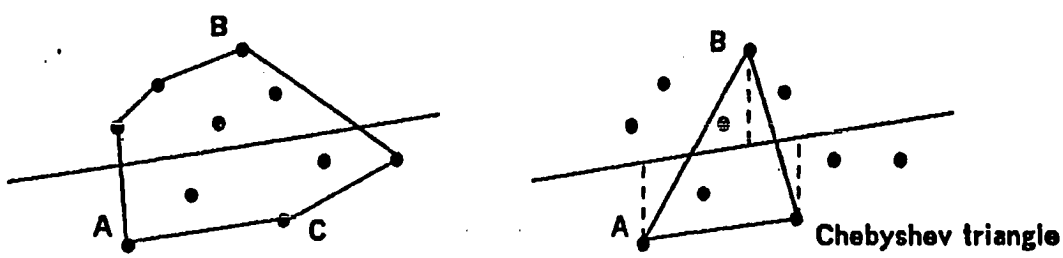


Figure 3.19: The Chebyshev points are also hull vertices.

Theorem 3.29: There exists a set of Chebyshev points of a finite set that are vertices of its convex hull and two of these points are consecutive on the hull.

Proof: Let us call the three Chebyshev points the *Chebyshev triangle.*[26] Consider the lines of support of the Chebyshev triangle parallel to the Chebyshev line. We claim that they also support the original set, for if not, then there is some point whose deviation from the approximating line exceeds that of any Chebyshev point, which is impossible. This shows that at least two of the Chebyshev points are extreme. By the equioscillation property, the third point, *C*, must have a y-deviation equal to that of one of the other two points, say *A*. Thus the line of support through *A* also passes through *C*, so *C* must be extreme. It is immediate that segment *AC* is an edge of the convex hull of *S*. (Figure 3.19.)

Theorem 3.29 is the basis of the following algorithm:

Algorithm A3.6: Linear Chebyshev approximation in two variables

1. Find the convex hull of the set in O(N log N) time.

2. Traverse each hull edge in order, finding the opposing vertex though which a parallel supporting line passes, and record the y-distance from this vertex to the extension of the hull edge. This scan requires only O(N) time. The hull edge and opposing vertex that achieve maximum y-distance give the Chebyshev points, from which the optimal approximating line can be found in constant time.

Theorem 3.30: O(N log N) time suffices to perform Chebyshev approximation on N points in the plane.

[26] Also an area in the Sea of Okhotsk into which Russian mathematicians have been known to disappear. See Durok, S., Zh. Nedostat. Mat. 7(1932), pp. 22-23.

Proof: By constructing a parallel line of support, Algorithm A3.6 finds, for every pair of consecutive hull vertices, a third point that maximizes the deviation. By Theorem 3.29, this is sufficient to find at least one Chebyshev triangle.

Theorem 3.31: [Cf. Theorem 3.18] Chebyshev approximation in the plane can be performed in O(N) expected time;

3.10.4 Least-squares isotonic regression

The problem of isotonic regression is to find a best isotone (that is, monotone non-increasing or non-decreasing) approximation to a finite point set. The error norm usually chosen is L_2, or least-squares, because of its connection with maximum likelihood estimation.[27] In other words, we are seeking an isotone function f that minimizes

$$\sum_{i=1}^{N} (y_i - f(x_i))^2 \ . \qquad (3.9)$$

A best least-squares isotone fit is a step function, as illustrated in Figure 3.20 [Barlow (72), Theorem 1.1]. It should be realized that the approximating function is defined only at the x_i, though for predictive purposes it will often be useful to extend its domain to the entire real line. It "only" remains to determine the locations and heights of the steps. The following method is taken from [Barlow (72)].

Suppose the data have been ordered by x-coordinate. (In many experimental situations, sorting is not necessary because the independent variable is time or the points are taken in order of increasing x.) Define the cumulative sum diagram (CSD) to be the set of points $P_j = (j, s_j)$, $P_0 = (0,0)$, where s_j is the cumulative sum of the y's:

[27] Isotonic regression is discussed at length in [Barlow (72)].

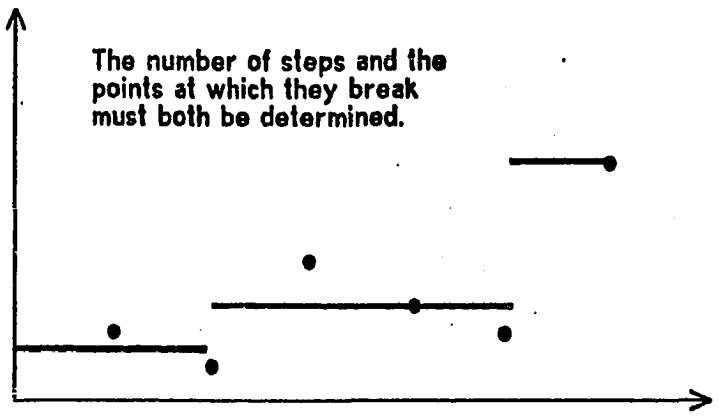


Figure 3.20: A best isotone fit is a step function.

$$s_j = \sum_{i=1}^{j} y_{(i)} \ . \qquad (3.10)$$

The slope of the line segment joining P_{j-1} to P_j is just y_j.

The *lower convex hull* of a set is the lower of the two chains into which the boundary of the hull is partitioned by the points p and q of least and greatest x-coordinate, respectively. It is the supremum of all convex functions whose graphs lie below the set.[28] It should be clear that, given the complete hull, the lower hull can be found in linear time. There is a close relationship between the hull problem and isotonic regression: The isotonic regression of a point set is given by the slope of the lower convex hull of its cumulative sum diagram [Barlow (72)]. Thus,

$$\text{ISOTONIC REGRESSION} \propto \text{CONVEX HULL} \ .$$

Theorem 3.32: Least squares isotonic-regression can be performed on a set of N points in the plane in O(N log N) time. If the points are ordered by abscissa, then O(N) time suffices.

[28] Another non-constructive definition.

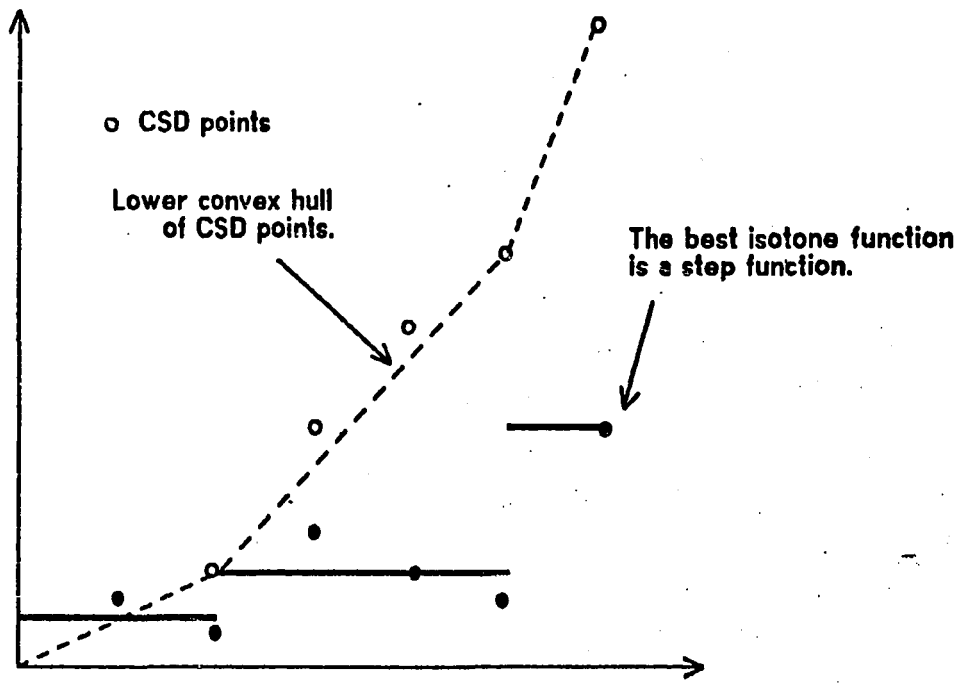


Figure 3.21: The lower convex hull of the CSD defines the isotonic fit.

Proof: We show that once the data are ordered by abscissa, the lower convex hull can be found in O(N) time. If the x_l are ordered, the CSD points (x_l,y_l) can be computed in O(N) time by Equation (3.10) and these are also ordered by abscissa a fortiori. Using any point on the positive y-axis as origin, the Graham scan of Section 3.3 can be run in O(N) time to construct the lower convex hull of the CSD. If the data are not ordered, then, by a simple extension of Theorem 3.8, Ω(N log N) time will be required in the worst case.

3.10.5 Clustering

To quote from [Hartigan (75)], clustering is the "grouping of similar objects". A *clustering* of a set is a partition of its elements that is chosen to minimize some measure of dissimilarity. Hartigan's book contains a large number of different such measures and procedures for clustering using them. We will focus on point data in two dimensions, where we assume that the x and y variables are scaled so that Euclidean distances are meaningful. A measure of the "spread" of a cluster is the maximum distance between any two of its points, called the *diameter* of the cluster. We feel intuitively that a cluster with small diameter has elements that are closely related, while the opposite is true of a large cluster. One formulation, then, of the clustering problem is

Problem P3.10: (Minimum Diameter k-Clustering) Given N points in the plane, partition them into k clusters $C_1, ..., C_k$ so that the maximum cluster diameter is as small as possible.

It is difficult to imagine how to solve this problem unless we at least have an algorithm for determining cluster diameter. This motivates

Problem P3.11: (Diameter of a Set) Given N points in the plane, find two that are farthest apart.

This problem is seemingly so elementary that it is difficult to perceive that there is any real issue involved. After all, we can compute the distance between each of the N(N-1)/2 pairs of points in a completely straightforward manner[29] and choose the largest of these to define the diameter. What is left to investigate? Is it possible that this $O(N^2)$ procedure is not the best possible algorithm?

---

[29]That is, if the model of computation allows square roots. Even if it does, a better way is to compute any monotonic function of distance, say the distance squared. $D(a,b) = (x_a-x_b)^2 + (y_a-y_b)^2$ can be computed in two multiplications and three addition/subtractions. This suffices to find any order statistic among the distances.

The only lower bound that suggests itself is the trivial one: We must spend O(N) time just to examine each point once, for we cannot be sure of the diameter otherwise. (It is tempting to believe that examination of all *pairs* of vertices is necessary, but this is not the case.) Let us try instead to reduce the upper bound.

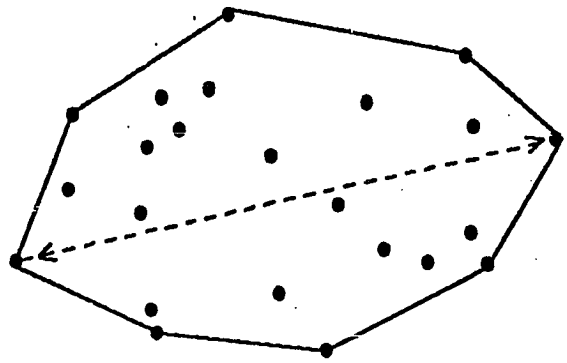


Figure 3.22: Diam(S) = Diam(Hull(S)).

Theorem 3.33: [Hocking (61)] The diameter of a set is equal to the diameter of its convex hull.

In the worst case, of course, all of the original points of the set may be vertices of the hull, so we will have spent O(N log N) time without eliminating anything. The convex hull, however, is a *convex polygon*, not just a set of points, so we have a different problem:

Problem P3.12: (Diameter of a Convex Polygon) Given a convex polygon, find its diameter.

We have immediately that

$$\text{SET DIAMETER} \propto_{N \log N} \text{CONVEX POLYGON DIAMETER} .$$

Theorem 3.34: [Yaglom (61), page 9] The diameter of a convex figure is the greatest distance between parallel lines of support.

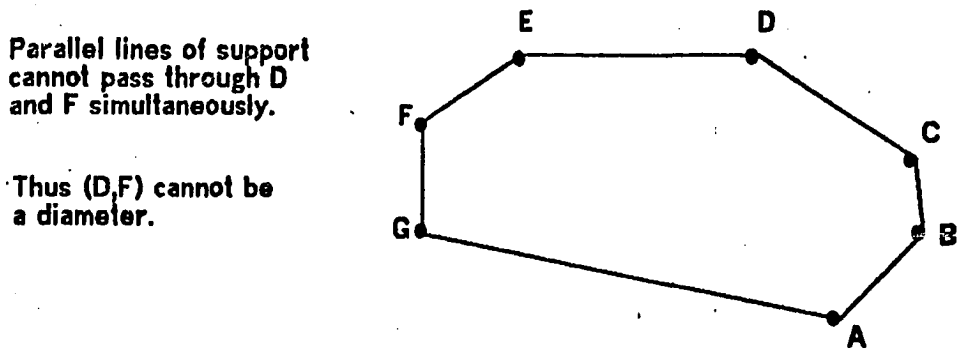


Figure 3.23: Not all vertex pairs are antipodal.

Consult Figure 3.23 and notice that parallel lines of support cannot be made to pass through *every* pair of points. For example, no lines of support through vertices D and F can be parallel. *This means that DF is not a diameter.* A pair of points that does admit parallel supporting lines will be called *antipodal.* Because of Theorem 3.34, we need only consider antipodal pairs. The problem is to find them without examining *all* pairs of points.

Referring now to Figure 3.24, observe that lines L and M are parallel lines of support through vertices A and D, respectively. This means that (A,D) is an antipodal pair. As the lines are rotated slightly counterclockwise about these vertices, they remain lines of support. This is true until one of the lines becomes coincident with an edge of the polygon. Here M, when rotated to position M', hits vertex E before L reaches B, so (A,E) becomes an antipodal pair.

Now M' will rotate about E while L' continues to rotate about A, and the next antipodal pair produced is (B,E). Continuing in this way, we will certainly generate all antipodal pairs, since the parallel lines will move through all possible angles. Determining the new pair at each step involves only a simple angle comparison.

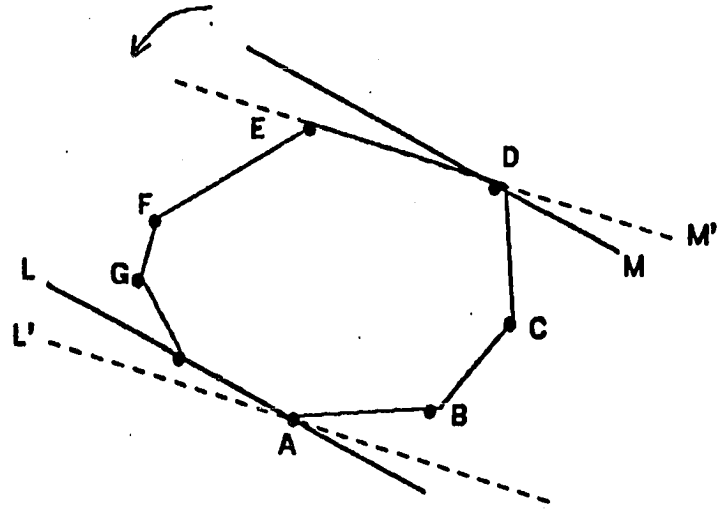


Figure 3.24: Generating antipodal pairs of vertices.

Algorithm A3.7: Antipodal Pairs

Input: A convex polygon P, in standard form.
Output: All antipodal pairs of vertices of P.

We assume that all indices are reduced modulo N (so that N+1 = 1) and that ANGLE(m,n) is a procedure that returns the clockwise angle swept out by a ray as it rotates from a position parallel to the directed segment p_m, p_{m+1} to a position parallel to p_n, p_{n+1}.

1. (Find an initial antipodal pair by locating the vertex opposite p_1.)

```
I ← 1;   J ← 2;
WHILE (ANGLE(I,J) < π) DO J ← J + 1;
CURRENT ← I;
OUTPUT (I,J) as an antipodal pair;
```

2. Now proceed around the polygon, taking account of possibly parallel edges. Line L passes through p_I, p_{I+1}; M passes through p_J, p_{J+1}.

```
    Loop on J until all of P has been scanned:
WHILE (J ≠ N) DO BEGIN
    IF   (ANGLE(CURRENT,I+1) ≤ ANGLE(CURRENT,J+1))
        THEN BEGIN
            J ← J + 1;          (Move line M)
            CURRENT ← J;
            END
        ELSE BEGIN
            I ← I + 1;          (Move line L)
            CURRENT ← I;
            END
    OUTPUT(I,J);        (Report an antipodal pair)

(Now take care of parallel edges)

    IF   (ANGLE(CURRENT,I+1) = ANGLE(CURRENT,J+1))
        THEN BEGIN
            OUTPUT(I+1,J);  OUTPUT(I,J+1);  OUTPUT(I+1,J+1);
            IF   CURRENT = I
                THEN J ← J + 1:
                ELSE I ← I + 1;
            END
    END
```

Theorem 3.35: Algorithm A3.7 produces all antipodal pairs of an N-vertex polygon P in O(N) time.

Proof: Suppose that a pair of parallel lines of support of P pass through vertices A and B. We show that there also exists a pair of parallel supporting lines through A and B in which one of the lines coincides with an edge of P. Since Algorithm A3.7 finds all pairs of supporting lines with this property it successfully produces all antipodal pairs of vertices. Let L and M be parallel lines of support through A and B, respectively. If neither is coincident with an edge of P, rotate both counterclockwise simultaneously until one reaches an edge. L and M still pass through A and B and the support property has not been lost. That the algorithm runs in O(N) time follows from the fact that either I or J or both are incremented during each execution of the WHILE-loop and we always have $I < J \leq N$.

Because enumeration of all antipodal pairs suffices to find the diameter of a polygon (by Theorem 3.34), we have the following:

Corollary 3.1: The diameter of a convex polygon can be found in O(N) time.

Corollary 3.2: [Cf. Theorem 3.20] The diameter of a simple polygon can be found in O(N) time.

Theorem 3.36: [Cf. Theorem 3.7] The diameter of a set of N points in the plane can be found in O(N log N) time.

Theorem 3.37: [Cf. Theorem 3.18] The diameter of a set of N points chosen from an N^p-distribution in the plane (Section 3.6.1) can be found in O(N) expected time.

3.11. Unsolved problems

1. What is the complexity of the k-dimensional hull problem? That is, how does the number of operations required to find all vertices, edges, and faces of the hull depend on N and k?

2. How quickly can the extreme points of a k-dimensional set be found? See [Dobkin (76b)].

3. What is the behavior of the higher-order moments of E(h) under various probability distributions. This would enable us to analyze the expected behavior of algorithms having a worst-case performance that is not linear in N.

4. How difficult is it to determine whether two polytopes are congruent? If the polytopes are three-dimensional and convex, this is related to isomorphism of planar graphs (by Steinitz's theorem [Grunbaum (67)]).

5. What is the complexity of Chebyshev approximation in higher dimensions? (Even k = 3 would be interesting, and it appears to be a difficult problem.)

6. How difficult is it to find the diameter of a set in k dimensions? (Again, k = 3 is a significant challenge.)

7. Isotonic regression in two independent variables? (Assume the regression function is to be monotonic in each variable separately.) What about other norms?

8. Find a depth algorithm that runs in less than quadratic time. What about the depth of a three-dimensional set?

9. Is there a fast algorithm for the convex hull of a simple polyhedron? k-dimensional polytope?

10. Problem P3.10.

3.12. Summary

The convex hull problem affords us the opportunity to develop computational geometry from its foundations, for we must deal directly with the non-constructive nature of combinatorial geometry. Three major convex hull algorithms are presented in this chapter. Graham's algorithm operates by sorting the points of a set by polar angle about an interior point and then eliminating vertices not on the boundary of the hull in a linear-time scan. It always requires O(N log N) time. Jarvis's algorithm is based on the principle of "gift-wrapping" and finds each successive hull vertex in O(N) time by repeatedly turning angles about the boundary of the set. While this may require $O(N^2)$ time in the worst case, it runs very quickly if the number of hull vertices is small. By using a fast algorithm for forming the hull of the union of two polygons as the recursive step of a divide-and-conquer procedure and exploiting a property of random point sets, we are able to produce an algorithm that runs in O(N) expected time without sacrificing O(N log N) worst-case behavior.

A lower bound of $\Omega(N \log N)$ for the hull problem is shown by demonstrating that sorting is reducible to hull-finding. This also provides a lower bound on the time necessary to find a simple closed polygonal path through N points in the plane, a bound which can be achieved by a variant of Graham's algorithm. The last section indicates some of the applications of hull-finding. We have been able to combine the earlier methods of this chapter with additional geometric tools to produce efficient algorithms for trimming, Chebyshev approximation in the plane, and various diameter problems. The diameter of an unordered plane set can be found in O(N log N) time, while the diameter of a simple polygon can be found in linear time. This result makes use of the fact that the convex hull of a simple polygon can be formed in linear time. By merging these results with our linear expected-time hull algorithm of the last section, we are able to produce a linear average-case diameter algorithm.

"have worn me out with several applications ..."
- Shakespeare, *All's Well That Ends Well.*

Chapter 4
Inclusion Problems

4.1. Introduction to Geometric Searching

"He prepared it, yea, and searched it."
- Job 28:27.

Among the most important of geometric problems are those that involve *searching*, or, in the most elementary language, *determining the location of an object*. This chapter develops the basic tools of geometric search that will be used in the succeeding chapters. We begin with a general discussion of searching and relevant complexity measures, then treat in detail the problem of *inclusion*:

Problem P4.1: (Polygon Inclusion) Given a simple polygon[1] P and a new point z, determine whether or not z is interior to P.

The difficulty of solving P4.1 depends on whether P is convex and whether preprocessing is allowed. The importance of the problem stems from the fact that almost all geometric searching, at some level, can be reduced to testing polygon inclusion.[2]

We will discuss a general search paradigm called, informally, a *query*. Given a collection of geometric data, we want to know if it possesses a certain property (say, convexity). In the simplest case the question will only be asked once, in which event it would normally be wasteful to do any preconditioning in the hope of

[1] If P is not simple, the problem may not be well-defined.

[2] For example, we will see in Chapter 6 that determining the nearest neighbor of a point is equivalent to finding which of a set of polygons contains it.

speeding up future queries. A one-time query of this type will be referred to as *single-shot*. Many times, however, queries will be performed repeatedly on the same database, and it may be worthwhile to arrange the information into an organized structure to facilitate searching. This can be accomplished only at some expense, though, and our analysis must focus on four separate cost measures:

1. Query time. How much time is required, in both the average and worst cases, to respond to a single query?

2. Storage. How much memory is required for the data structure?

3. Preprocessing time. How much time is needed to arrange the data for searching?

4. Update time. Given a new piece of data, how long will it take to add it to the data structure?

The various tradeoffs among query time, preprocessing time, and storage are well-illustrated by the problem of range searching[3], which arises frequently in geographic applications and database management:[4]

Problem P4.2: (Range Searching) Given N points in the plane, how many lie in a given rectangle?[5] That is, how many points (x,y) satisfy $a \leq x \leq b$, $c \leq y \leq d$?

It is clear that a single-shot range query can be performed in linear time, since we need only examine each of the N points to see whether it satisfies the

[3][Knuth (73)], page 550.

[4]The work described in this section was performed jointly by the author and Jon Bentley and is reported in [Bentley (77a)].

[5]The rectangle is assumed to have its sides parallel to the coordinate axes.

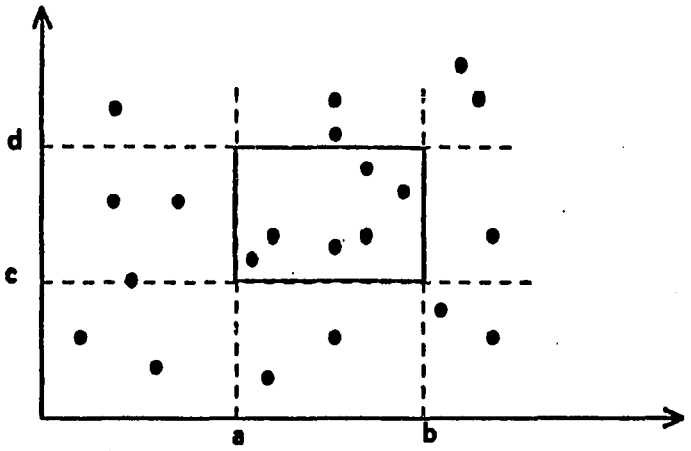


Figure 4.1: A Range Query. How many points lie in the rectangle?

inequalities defining the rectangle. Likewise, linear space suffices because only the 2N coordinates need to be saved. There is no preprocessing time and the update time for a new point is just a constant.

What kind of data structure can be used to speed the processing of multiple queries? It seems to difficult to organize the points so that an arbitrary new rectangle can be accomodated easily. We also cannot solve the problem in advance for *all* possible rectangles because of their infinite number. The following solution is an example of the *locus method* of attacking geometry problems, one in which we look for critical regions within which the answer does not vary and store these compactly.

A rectangle itself is an unwieldy object; we would prefer to deal with points. This suggests that we might replace the rectangle query by four subproblems, one for each vertex, and combine their solutions to obtain the final answer. In this case the subproblem associated with a point p is to determine the number of points Q(p) of the set that satisfy both $x \leq x_p$ and $y \leq y_p$, that is, the number of points in the southwest quadrant determined by *p*. (See Figure 4.2.)

The concept we are dealing with here is that of *vector domination*:

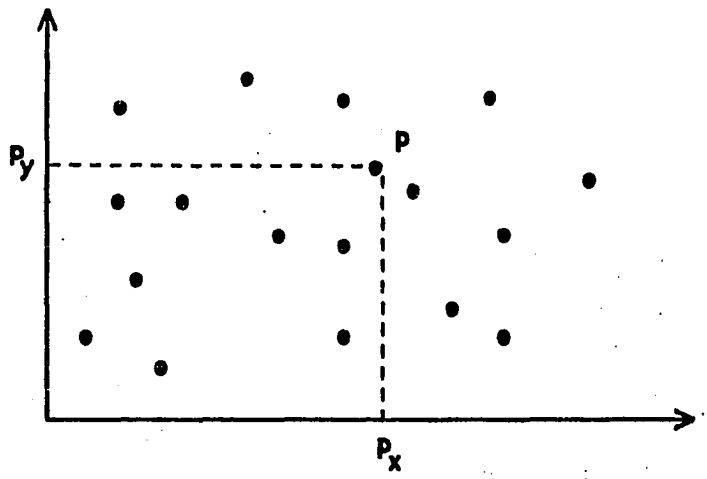


Figure 4.2: How many points lie to the southwest of P?

Definition 4.1: Given two points *A* and *B* in the plane, we say that *A dominates B* if $x_A \geq x_B$ *and* $y_A \geq y_B$; that is, *A* is greater than or equal to *B* in both coordinates.

In the plane, W is dominated by V iff it lies in V's southwest quadrant. Q(p) is thus the number of points dominated by *p*. The connection between domination and range queries is apparent in Figure 4.3. The number N(*ABCD*) of points contained in rectangle *ABCD* is given by

$$N(ABCD) = Q(A) - Q(B) - Q(D) + Q(C) \qquad (4.1)$$

This follows from the combinatorial principle of inclusion-exclusion [Liu (68)]. All points in the rectangle are certainly dominated by *A*. We must remove those dominated by *B* and also those dominated by *D*, but this will cause some points to be eliminated twice -- specifically, the ones dominated by *both B* and *D* -- and these are just the points lying in C's southwest quadrant.

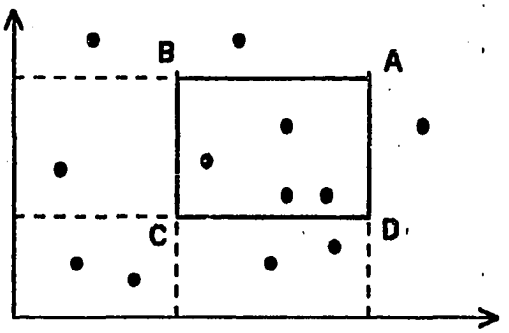


Figure 4.3: A range search as four domination queries.

We have thus reduced the problem of range searching to one of performing four point domination queries. The property that makes these queries easy is that there are nicely-shaped regions of the plane within which the domination number Q is constant.

Suppose we drop perpendiculars from the points to the x-and y-axes, and extend the resulting lines indefinitely. This produces a mesh of $(N+1)^2$ rectangles, as shown in Figure 4.4.

For all points p in any given rectangle, $Q(p)$ is a constant. This means that domination searching is just a matter of determining which region of a rectilinear mesh a given point lies in. This question is particularly easy to answer. Having sorted the points on both coordinates, we need only perform two binary searches, one on each axis, to find which rectangle contains the point. Thus the query time is only O(log N). Unfortunately, there are $O(N^2)$ rectangles, so quadratic storage is required. We must now compute the domination number for each rectangle. This can readily be done for any single rectangle in O(N) time, which would lead to an $O(N^3)$ algorithm overall for preprocessing, but this can be reduced to $O(N^2)$ [Bentley (77a)].

While the above procedure answers queries rapidly, the storage and preprocessing time required can be prohibitive. To address this problem, we now sketch an algorithm from [Bentley (77a)].

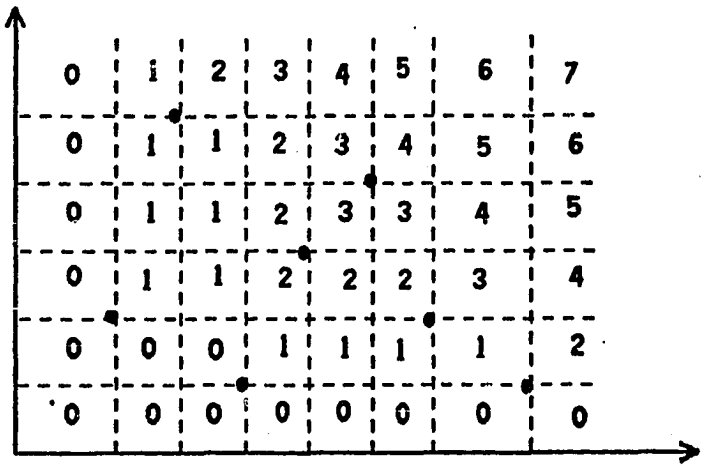
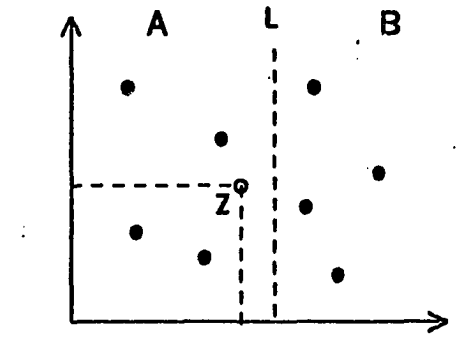
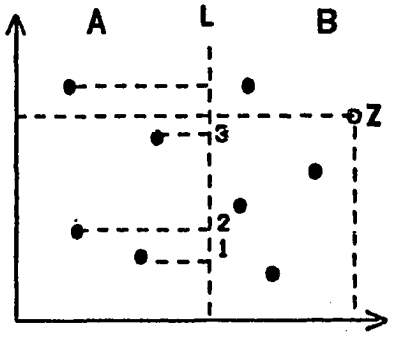


Figure 4.4: Mesh of rectangles for domination searching.



If Z lies to the left of L we need only find its rank in A.



If Z lies to the right we need its rank in B and its y-rank in A.

Figure 4.5: The two cases of ECDF searching in the plane.

Let L be a line having median x-coordinate among the points (Figure

4.5.). (Degenerate cases can be handled by the methods of Section 6.4.) L can be found in O(N) time. Let A be the set of N/2 points to the left of L and B the set of points to the right. Given a new point Z we want to determine the number of points (in both A and B) that it dominates. In a single comparison against L we can determine whether Z lies in A or in B. If Z lies in A (the diagram on the left in Figure 4.5) it cannot possibly dominate any point of B, so we may confine our attention to a subproblem of half the size of the original. The recurrence describing this situation is just

$$T(N) = T(N/2) + 1 .$$

If we learn from the first comparison that Z lies in B then the problem is only slightly more complicated (the right diagram in Figure 4.5). We must find the number of points in B that are dominated by Z, which can be done in time T(N/2). We then add to that the number of points in A dominated by Z. Since, however, the x-coordinate of Z is known to be greater than that of any point of A, this number is merely the number of points of A that lie below Z. If we project the points of A onto L and sort them in advance (as part of the preprocessing) we will be able to locate Z in this ordering in O(log N) time by binary search. Thus the recurrence that results when Z is in B is

$$T(N) \leq T(N/2) + O(\log N) .$$

It is immediate that $T(N) = O(\log^2 N)$, even if the second case arises after each comparison.

The storage requirement of this algorithm is easy to analyze in view of its recursive structure. In two dimensions we need to store two data structures on N/2 points and one linear list of length N/2. Thus,

$$S(N,2) = 2S(N/2,2) + O(N) = O(N \log N).$$

The preprocessing time is described by precisely the same relation.

Theorem 4.1: [Bentley (77a)] Range searching in the plane can be performed using any of the following combinations of resources:

Query	Storage	Preprocessing time
$O(\log N)$	$O(N^2)$	$O(N^2)$
$O(\log^2 N)$	$O(N \log N)$	$O(N \log N)$
$O(N)$	$O(N)$	$O(N)$

Table 4.1. Resources Required for Range Searching.

The time-storage-preprocessing tradeoffs illustrated in this theorem are typical of many geometric search problems.

4.2. Inclusion in a convex polygon

We now return to inclusion problems, the simplest of which is

Problem P4.3: (Convex Inclusion) Given a convex polygon *P* and a new point *z*, is *z* interior to *P* ?

We can dispose of the single-shot problem immediately, and the result holds for non-convex polygons as well:

Theorem 4.2: Whether a point z is interior to a simple N-gon can be determined in O(N) time, without preprocessing.

Proof: The Jordan Curve Theorem for Polygons [Courant (41)] states that a simple polygon partitions the plane into two disjoint regions, the *interior* and the *exterior*, that are separated by the boundary of the polygon. Consider a horizontal line H that passes through z. (See Figure 4.6). Since the polygon is bounded, the extremities of H must lie in the exterior region. In general, H will intersect the polygon P. (If it does not, then z is exterior.) Let R

be the number of points to the right of z in which H intersects the boundary of P. Consider moving a point x left on H from $+\infty$ towards z. Initially, x lies outside P. As an edge of P is crossed, x becomes interior. The test point z is thus inside or outside P as R is either odd or even. R can be determined by comparing H with each of the N edges of P, but some care is require to count the number of intersections properly. Two difficulties arise. First, if H coincides with an edge of P then two intersections are to be counted unless z lies on the edge, in which case the algorithm terminates. The other problem is that H can intersect an edge without crossing it, as shown in Figure 4.7. This can only occur at a vertex of P, and it is necessary to determine whether both edges incident with the vertex lie on the same side of H. If so, the intersection must be counted twice (or not at all).

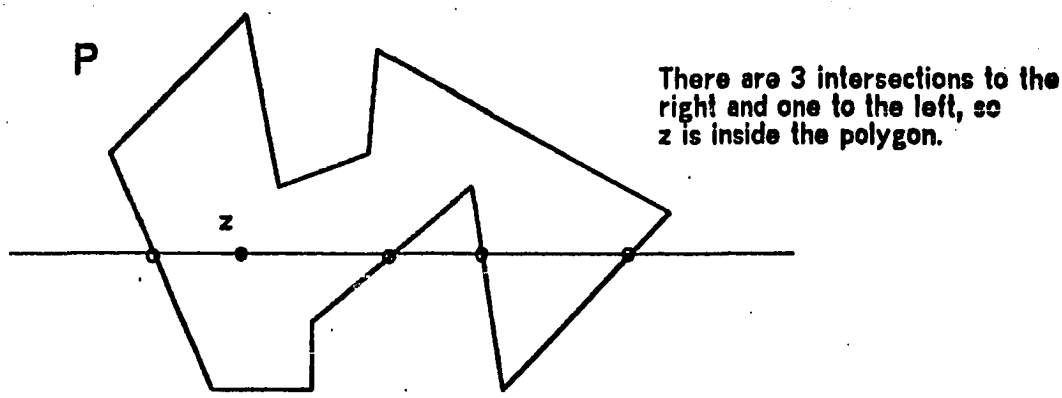


Figure 4.6: Single-shot inclusion in a simple polygon.

For repeated queries with preprocessing allowed, we develop a special method that relies on the convexity of the polygon. Recalling Theorem 3.5, the vertices of a convex polygon occur in angular order about any interior point. Find such a point *O* and consider the N rays from *O* that pass through the vertices of P. (Figure 4.8.) These rays partition the plane into N pie-shaped wedges. Each

The mountain range illustrates that care must be taken in counting intersections. There are three intersections to the right of z and one to the left, but this time z is outside!

The resolution is that the intersections at A and B must be counted twice.

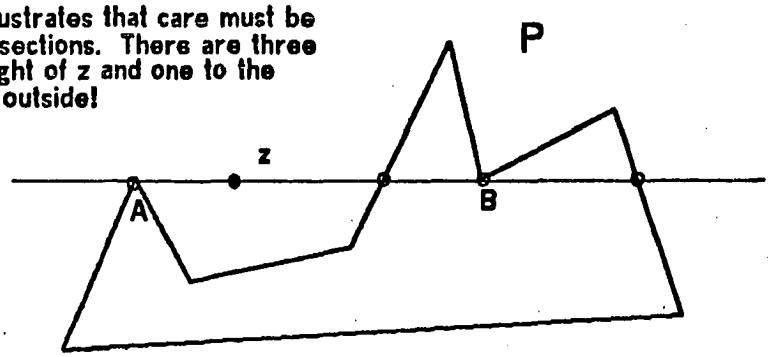
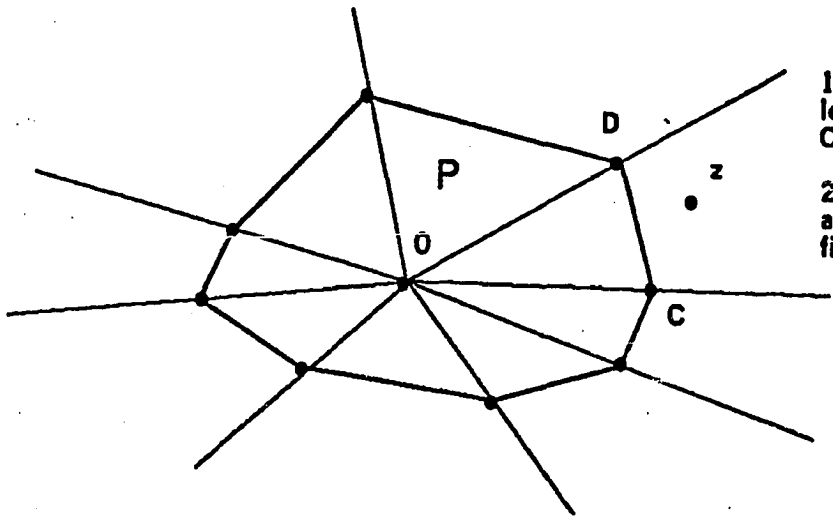


Figure 4.7: Intersections must be counted properly.



1. By binary search we learn that z lies in wedge COD.

2. By comparing z against edge CD we find that it is exterior.

Figure 4.8: Division into wedges for the convex inclusion problem.

wedge is divided into two pieces by a single edge of P. One of these pieces is wholly interior to P, the other wholly exterior. Treating *O* as the origin of polar coordinates, we may find the wedge in which *z* lies by a single binary search, since the rays occur in angular order. Given the wedge, we need only compare *z* to the unique edge of P that cuts it, and we will learn whether *z* is interior.

Algorithm A4.1: Convex Inclusion

Preprocessing:

1. Find an interior point O.
2. Arrange the vertices of P in a structure suitable for binary searching. (A vector, for example.)

Search:

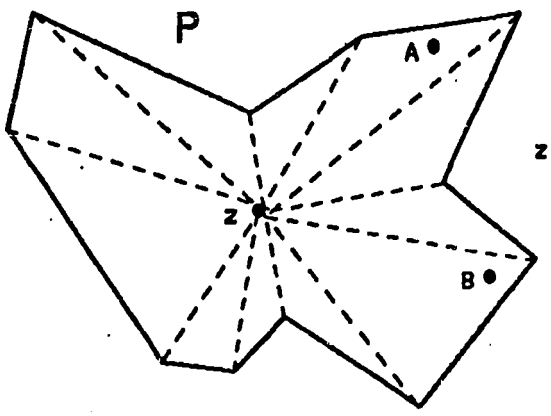
1. Given a new point z, determine by binary search the wedge in which it lies. Point z lies between the rays defined by p_i and p_{i+1} iff angle zOp_{i+1} is a right turn and zOp_i is a left turn. In this way we can find the wedge without computing any polar angles.
2. Once p_i and p_{i+1} are found, then z is interior iff $p_ip_{i+1}z$ is a left turn.

Theorem 4.3: The inclusion question for a convex N-gon can be answered in O(log N) time and O(N) space, given O(N) preprocessing time.

Proof: It is clear that after O(N) preprocessing, Algorithm A4.1 runs in O(log N) time. We now show that it decides inclusion correctly. A point z is only reported to be interior to P if a vertex p_i has been found such that the directed path $p_ip_{i+1}z$ is a left turn. In this case z is interior to triangle Op_ip_{i+1} and thus is interior to P. Point z is only reported to be exterior to P if no vertex p_i is found such that z is interior to or on the boundary of triangle Op_ip_{i+1}. Since these triangles partition P, z cannot be interior to or on the boundary of P. Thus in both cases the algorithm answers correctly.

4.3. Star-shaped Inclusion

What property do convex polygons possess that enable them to be searched quickly? In order to be able to apply binary search, the vertices must occur in sequence about some point. This property is also shared by some non-convex polygons, as shown by Figure 4.9. Maruyama was inspired to call these polygons "angularly simple" [Maruyama (72)], but we will adhere to classical terminology:



z can "see" all other points of P.

Figure 4.9: A Star-shaped Polygon.

Definition 4.2: A polygon P is said to be *star-shaped* if there exists a point z such that, for all points p of P, the line segment zp lies entirely within P.

Casually speaking, P is star-shaped if there exists a point that can "see" all of the other points.

Theorem 4.4: [Penney (72)] A polygon P is star-shaped iff there exists some point $z \in P$ such that, for all vertices v of P, $zv \subset P$.

This theorem suggests that a finite algorithm may exist to determine whether or not P is star-shaped since it is a criterion that depends only on the *vertices* of P (rather than on all points of P).

Definition 4.3: A polygon P is said to be *star-shaped with respect to a point z* iff $zp \subset P$ for all points p of P.

Note that a star-shaped polygon is not necessarily star-shaped with respect to all of its interior points. In Figure 4.9, for example, the segment joining A and B does not lie within P, so P is not star-shaped with respect to A (or B, for that matter).

To determine whether or not a point is interior to a star-shaped polygon, we may use Algorithm A4.1 directly if an appropriate origin *O* from which to base the search can be found. The set of feasible origins within P is also the locus of points with respect to which P is star-shaped and is called its *kernel*:

Definition 4.4: The *kernel* of a polygon P is the locus of points $z \epsilon P$ such that $zp \subset P$ for all points p of P.

Thus, any point in the kernel will serve as origin. The kernel of a polygon is shown in Figure 4.10.

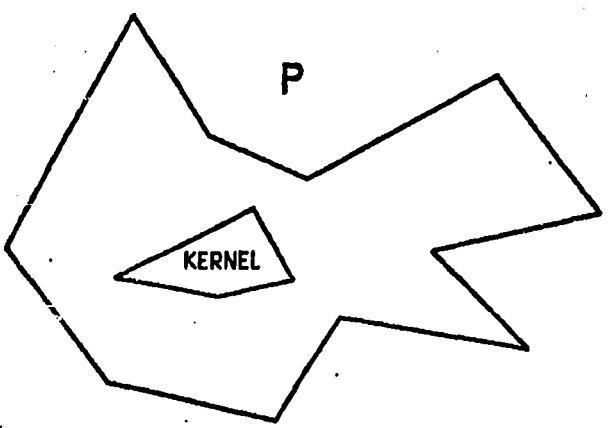


Figure 4.10: The Kernel of a Polygon

The next theorem shows why we need not be very particular in selecting the origin if P is convex.

Theorem 4.5: Kernel(P) = P iff P is convex.

Proof: 1. (P convex => Kernel(P) = P). If P is convex, then by Definition 2.1, $zp \subset P$ for all points $z,p \in P$. But then each point z of P lies in Kernel(P) by Definition 4.4.

2. (Kernel(P) = P => P convex). Consider any two points x,y of P. If Kernel(P) = P, then in particular $x \in$ Kernel(P) and thus $xy \subset P$ by Definition 4.4, so P is convex;

Thus convex polygons are star-shaped with respect to all interior points.

Theorem 4.6: The kernel of a star-shaped polygon is itself a convex polygon having no more vertices than the original.[6]

Problem P4.4: (Kernel of a Polygon) Given a polygon, how quickly can its kernel be found?

We will return to this problem in Section 5.5.1, where we show that the kernel can be found in O(N) time. We thus have a fast way of finding a reference point about which to create wedges for inclusion searching.

Theorem 4.7: The inclusion question for a star-shaped polygon can be answered in O(log N) time and O(N) storage, given O(N) preprocessing time.

Proof: Given a star-shaped polygon P, a point O in its kernel can be found in O(N) time by Theorem 5.14. Now execute Algorithm A4.1 with the point O as just obtained, and the proof of Theorem 4.3 applies, with minor modifications.

[6] A proof is given in Section 5.5.1.

4.4. Inclusion in a Simple Polygon

Plane polygons can be arranged in a hierarchy that is strictly ordered by the subset relation:

$$\text{CONVEX} \subset \text{STAR-SHAPED} \subset \text{SIMPLE} \qquad (4.2)$$

Suppose we know that a polygon is not star-shaped. How difficult is it to solve the inclusion problem? One approach is motivated by the fact that every polygon is a union of some number of star-shaped polygons.[7] For example, the polygon in Figure 4.11 is a union of four star-shaped ones.

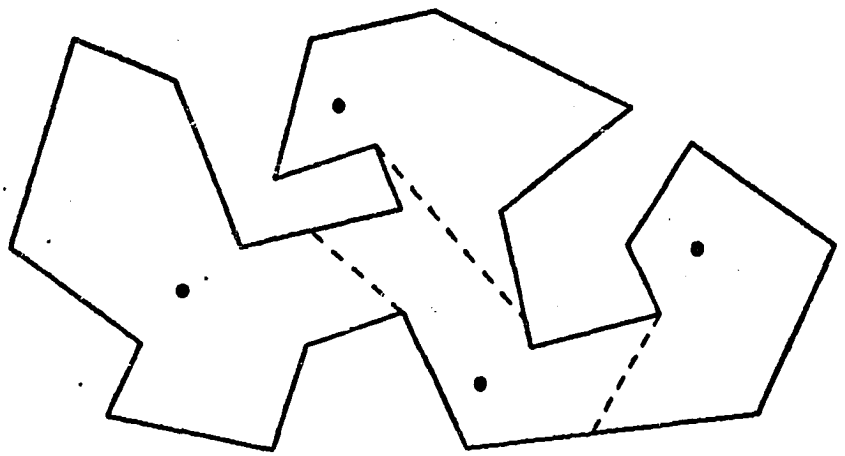


Figure 4.11: A Simple Polygon as a Union of Star-Shaped Polygons.

[7] [Maruyama (72)] has used this idea to define the complexity of a polygon as the least k for which it can be decomposed into a union of k star-shaped polygons.

We may then apply the search algorithm for star-shaped polygons k times to obtain an O(k log N) inclusion procedure for simple polygons: The test point z is compared with each of the k origins in turn, until it is found to be interior to some star-polygon or until all k polygons have been searched. Unfortunately, Chvatal has shown that k may be as large as $\lfloor N/3 \rfloor$, so the search may take as much as linear time. [Chvatal (75)]. (See Figure 4.12.) Even though k can be O(N), the maximum time required for such a search is O(N), not O(N log N). Here is the reason: Once the polygon P has been partitioned into disjoint star-shaped subsets, a planar graph results. If we partition using only chords of P, then the total number of edges in all resulting polygons cannot exceed 6N-12, since no segment lies in more than two polygons and there are at most 3N-6 segments in all. If the number of edges in polygon i is n_i, then we have

$$n_1 + n_2 + \ldots + n_k \leq 6N-12 , \quad n_i \geq 3 , \; k \leq N .$$

The totai search time is given by

$$\sum_{i=1}^{k} \log n_i ,$$

which we want to maximize subject to the above constraints. The sum attains a maximum when k is as large as possible and the n_i are as equal as possible, i.e. when k = N/3 and n_i = 3, in which event the sum is O(N). (See [Hardy (67)].) Thus, in the worst case, the search will be no better than the single-shot algorithm of Section 4.2.

The difficulty with the above method is that is does not make effective use of binary search because there is no way to decide which of the star-shaped subsets should be searched. Furthermore, it is not clear how to obtain a minimal decomposition.[8] Just as for range queries, we would like to be able to define regions that are either entirely inside or outside the polygon, but ones that can be searched quickly. This fundamental idea of *creating new objects to permit binary*

[8] In fact, we know of no polynomial-time algorithm.

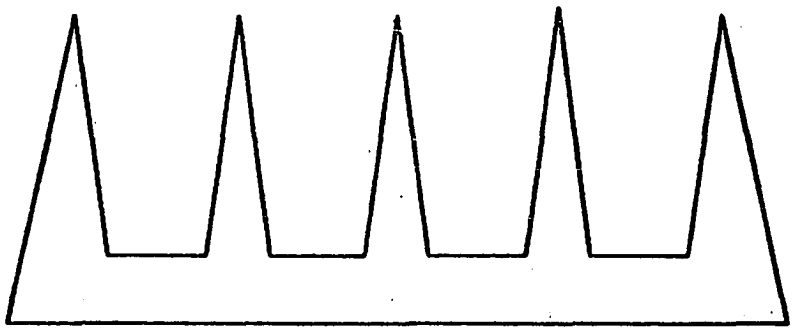


Figure 4.12: The Alligator Counterexample.

searching is due to Dobkin and Lipton. [Dobkin (76a)]. They realized that constraining oneself to work with only the original data (in this case the edges of the polygon) is unnecessarily restrictive. We use their methods, combined with the locus approach, to devise a fast algorithm.

Given a simple polygon *P*, consider drawing a horizontal line through each of its vertices, as in Figure 4.13. This divides the plane into N+1 horizontal slabs. If we sort these slabs by y-coordinate as part of the preprocessing, we will be able to find in O(log N) time the slab in which a new point *z* lies.

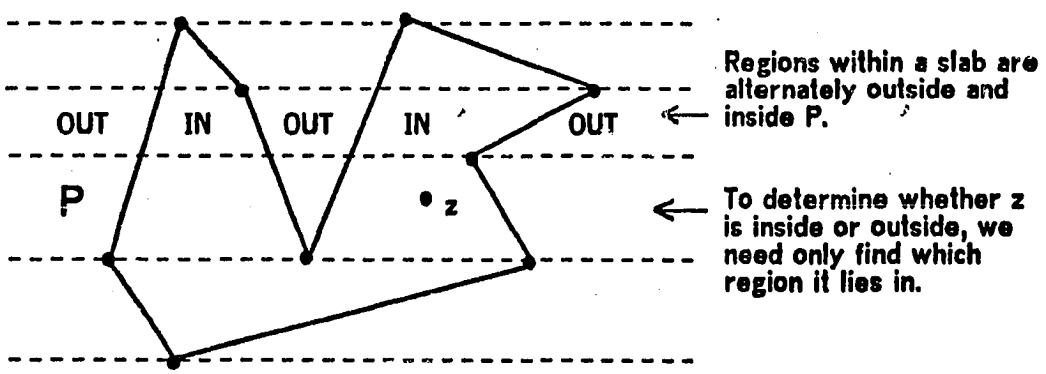


Figure 4.13: The vertices of a polygon define horizontal slabs.

Now consider the situation within a slab, which contains segments of the edges of P. These segments define trapezoids that are either inside or outside of P.[9] Since P is simple, its edges intersect only at vertices, and, since each vertex defines a slab boundary, *no segments intersect within a slab.* (See Figure 4.14.)

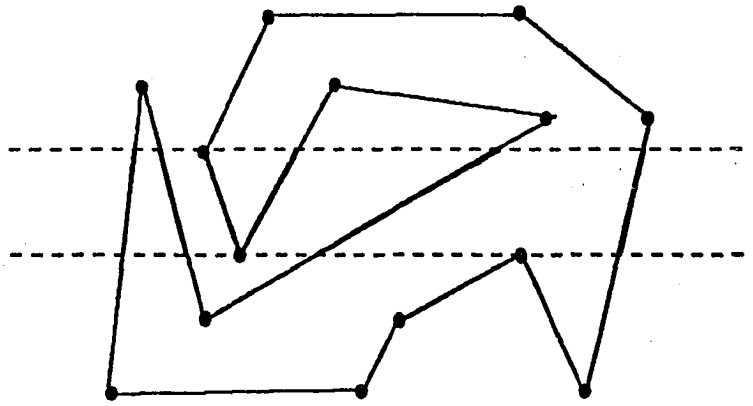


Figure 4.14: Within a slab, segments do not intersect.

The segments can thus be totally ordered by the LEFT-RIGHT relation, and we may use binary search to determine in O(log N) time the trapezoid in which *z* falls. This will give a worst-case query time of O(log N).

It only remains to analyze how much work is done in preconditioning the polygon and storing it. Naively, it seems that we must sort all of the line segments in every slab. Furthermore, each slab may have O(N) segments, so it appears that $O(N^2 \log N)$ time and $O(N^2)$ storage will be required. We will show how to reduce the preprocessing time to $O(N^2)$. Nothing can be done (in this algorithm) to reduce the storage used since there exist polygons that need quadratic space (Figure 4.15).

Notice that a given edge of the polygon may pass through many slabs. This

[9]The trapezoids can degenerate into triangles, as Figure 4.14 shows.

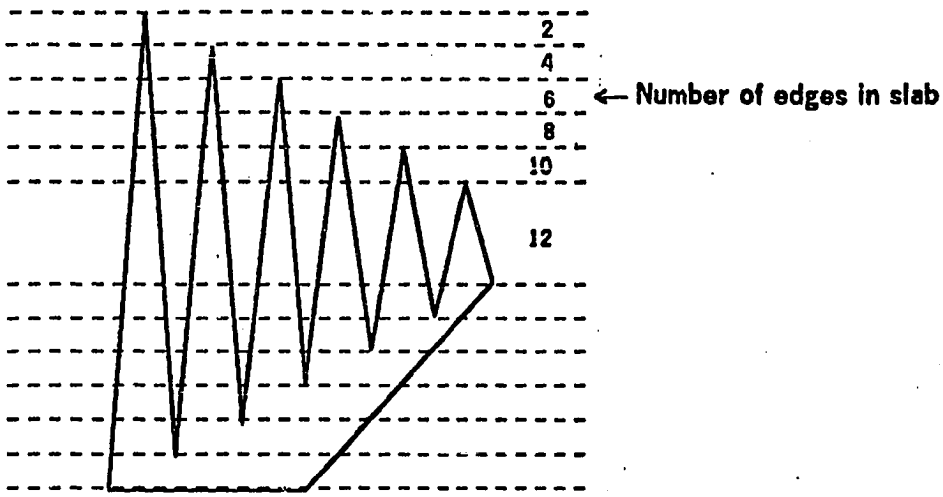


Figure 4.15: The slabs may contain a total of $O(N^2)$ segments.

observation is the key that allows us to reduce the preprocessing time. We will process the slabs in ascending order, beginning with the lowest, having first sorted the vertices of P by y-coordinate. The segments belonging in a given slab can be found in O(N) time. Enter them into a balanced binary tree, based on the left-right ordering, in O(N log N) time. Moving up to the next slab boundary L, some edges of P (at least one) will terminate, while the others continue into the next slab. As we enter the next higher slab from below, additional edges may be introduced. Because *P* is simple, all continuing edges will retain their same respective positions in left-right order. It is only necessary to delete from the tree those edges that terminate at L and introduce the ones that begin at L. Since each of the N edges is inserted in the balanced tree exactly once and deleted once, O(N log N) time suffices for these operations. To create the slab data structure, however, it is necessary to output the segments of each slab in order. This can be done in linear time for each slab, for a total of $O(N^2)$ time. Here is the preprocessing algorithm:

Algorithm A4.2: Preprocessing for Polygon Inclusion

```
1.  Sort the vertices of P by y-coordinate.
        VERTEX[I] will be the I'th lowest vertex.

2.  Set up the slabs.  The I'th slab from the bottom is
    just a list of edges of P.  (Slab 0 is empty.)
        The edges are maintained as leaves of a balanced tree.
        INSERT is a procedure that makes tree insertions.
        DELETE performs deletions.

    FOR I ← 1 UNTIL N DO BEGIN
(Case 1:  VERTEX[I] is the lower endpoint of edges E and F)
            INSERT(E);  INSERT(F);  GO TO READOUT;
(Case 2:  VERTEX[I] is the upper endpoint of edges E and F)
            DELETE(E);  DELETE(F);  GO TO READOUT;
(Case 3:  VERTEX[I] is the upper endpoint of edge E
            and the lower endpoint of edge F)
            DELETE(E);  INSERT(F);
READOUT:  Output the edges into the I'th slab list
            without deleting them from the tree.
    END
```

Thus we have

Theorem 4.8: The inclusion question for a simple polygon can be answered in O(log N) time using $O(N^2)$ storage, given $O(N^2)$ preprocessing time.

In earlier work [Shamos (75b)], the author conjectured that the storage requirement for simple inclusion could be reduced to O(N) while increasing the query time to only $O(\log^2 N)$. This was verified by Lee and Preparata, who also showed that the preprocessing time is only O(N log N) [Lee (76b)].

Their method is based on a simple but elegant idea. A polygonal line will be said to be a *monotone chain with respect to line L* iff its vertices retain the same order when projected onto L. Monotone chains are of interest because they can be

searched in O(log N) time by binary search. A *set* of chains will be called monotone iff they are all monotone with respect to some line L. The Lee-Preparata algorithm constructs a monotone set of chains for a polygon P such that the region between adjacent chains is either wholly inside or wholly outside of P. At most N chains are created and these have a total of O(N) edges. The inclusion search is accomplished by finding the pair of chains between which the test point lies. This is done by a binary search of the chains and each chain may require O(log N) time to examine. Thus $O(\log^2 N)$ time suffices. For future reference, we state their result as a theorem:

Theorem 4.9: Inclusion in a simple polygon can be determined in $O(\log^2 N)$ time using O(N) storage, given O(N log N) time for preconditioning.

4.4.1 Location in a planar embedding

A problem that is intimately related to polygon inclusion concerns *planar straight-line graphs*, planar graphs that have been drawn in the plane using only straight line segments as edges. [10]

Problem P4.5: (Planar Graph Search) Given a planar straight-line graph and a new point *z*, how quickly can the region containing *z* be found?

If N is the number of vertices of the graph, then both the number of edges and regions are O(N) [Harary (71)]. Algorithm A4.2 can be applied essentially without modification to yield an O(log N) slab search method, because the non-intersection property of segments within a slab is retained. The analog of Theorem 4.8 is

Theorem 4.10: Location in a planar straight-line graph can be determined in O(log N) time and $O(N^2)$ storage, given $O(N^2)$ preprocessing time.[11]

---

[10] A theorem of [Fary (48)] states that <u>every</u> planar graph can be embedded in the plane as a planar straight-line graph.

[11] [Dobkin (76a)] contains a similar result but with $O(N^2 \log N)$ preprocessing time.

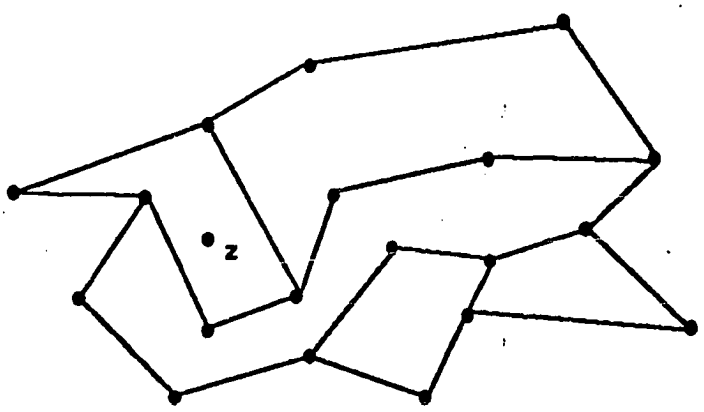


Figure 4.16: A planar straight-line graph. In which region does z lie?

Likewise, Lee and Preparata demonstrate that their method also carries over:

Theorem 4.11: [Lee (76b)] Location in a planar straight-line graph can be determined in $O(\log^2 N)$ time and O(N) storage, given O(N log N) preprocessing time.

Recently, Lipton and Tarjan proved a very powerful result which they termed the "Planar Separator Theorem":

Theorem 4.12: (Planar Separator) [Lipton (77a)] The vertices of any N-vertex planar graph G can be partitioned in O(N) time into three sets A, B, and C such that no edge of G joins a vertex in A with a vertex in B, neither A nor B contains more than 2N/3 vertices, and C contains $O(N^{1/2})$ vertices.

They were able to use this fact to obtain a planar graph searching algorithm with asymptotically optimal time and space performance:

Theorem 4.13: [Lipton (77b)] Location in a planar straight-line graph can be determined in O(log N) time and O(N) storage, using O(N log N) preprocessing time.

Method	Query	Storage	Preprocessing Time
Slabs	O(log N)	$O(N^2)$	$O(N^2)$
Lee & Preparata	$O(\log^2 N)$	O(N)	O(N log N)
Lipton & Tarjan	O(log N)	O(N)	O(N log N)
Single-shot	O(N)	O(N)	O(N)

Table 4.2. Summary of resources required for simple polygon inclusion and planar graph searching.

4.5. Unsolved problems

1. How difficult is it to determine whether a point is inside or outside a convex polytope? (We know that O(log N) time suffices and is necessary, but suppose we only allow storage that is linear in the number of edges of the polytope?) [Lee (76b)] achieves $O(\log^2 N)$ search time and O(N) storage for the three-dimensional case.

2. Give an algorithm to find the least number of convex polygons whose union is a given simple polygon.

3. Give an algorithm to determine the least number of star-shaped polygons whose union is a given simple polygon.

4. Complications arise if a collection of figures is given whose members are not pairwise disjoint. For example, given N rectangles in the plane, find *all* the ones that include a new point z.

4.6. Summary

This chapter introduces geometric searching and notions relating to it. All known techniques of geometric search involve restructuring a problem so that binary search may be applied. This often involves the creation, either explicitly or implicitly, of new objects that must be arranged in a data structure. Whether this preprocessing is justified depends on the query response time that must be achieved, the amount of storage available, and the number of queries that are to be handled.

We have shown that one of the fundamental problems of geometric search is *Inclusion in a polygon* and the complexity of answering it depends sensitively on the structure of the polygon. Determining whether a point is interior to a convex polygon is directly equivalent to binary search. We introduce the *kernel* of a polygon as the locus of points suitable as origin for searching a star-shaped polygon. Simple polygons and polytopes in any dimension can always be searched in O(log N) time if sufficient storage and preprocessing time are available, there being an apparent tradeoff between speed of search and storage and preconditioning expense.

The problem of the *range query* typifies advanced search applications, in which one must reorganize the input considerably in order to develop an efficient algorithm. [Knuth (73)] remarks that "no really nice data structures seem to be available" for this problem. The key to such a data structure is in reducing a rectangle query to four single-point queries and applying the principle of inclusion-exclusion.

Chapter 5
Intersection Problems

5.1. Introduction

Much of the motivation for studying intersection problems stems from the simple fact that two objects cannot occupy the same place at the same time. An architectural design program must take care not to place doors where they cannot be opened or have corridors that pass through elevator shafts. In computer graphics, an object to be displayed obscures another if their projections on the viewing plane intersect. A pattern can be cut from a single piece of stock only if it can be laid out so that no two pieces overlap. The importance of developing efficient algorithms for detecting intersection is becoming apparent as industrial applications grow increasingly more ambitious. A single integrated circuit may contain tens of thousands of components, a complicated graphic image may involve one hundred thousand vectors and an architectural database often contains upwards of a million elements. In such cases even algorithms that are only quadratic in the number of objects are unacceptable.

Another reason for delving into the complexity of intersection algorithms is that they shed light on the inherent complexity of geometric problems and permit us to address some fundamental questions. For example, how difficult is it to tell whether a polygon is simple? One would be justified in investigating such a topic even if it had no practical applications, but we will find no shortage of uses for the algorithms of this chapter. Because two figures intersect only if one contains a point of the other[1], it is natural that intersection algorithms should involve inclusion testing. We may thus consider intersection problems to be natural extensions of inclusion problems.

---

[1]Depending on how boundary intersections are defined.

5.1.1 The Hidden-Line Problem

A pivotal problem in computer graphics, and one that has absorbed the energy of many researchers[2], is the *hidden-line problem*. A two-dimensional image of a three-dimensional scene is necessarily a projection. We may not, however, merely project each object onto the plane of the observer, for some objects may be partially or totally obscured from view. In order to produce a faithful display, those lines which a real observer cannot see must be eliminated from the picture. Figure 5.1 shows a scene before and after hidden lines have been removed.

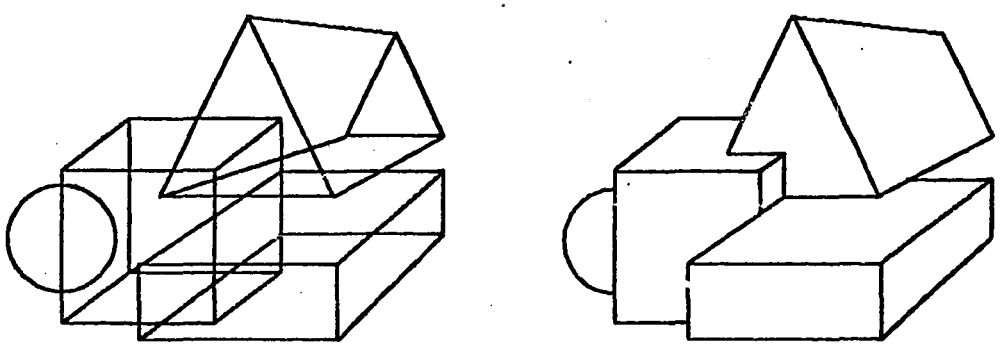


Figure 5.1: Elimination of Hidden Lines.

One object obscures another if their projections intersect, so detecting and forming intersections is at the heart of the hidden-line problem. A considerable investment has been made in developing hardware to perform this task, which is particularly difficult in practice because of the real-time requirements of graphic display systems and the fact that objects are usually in motion. There is no commercial system available that is able to display a moving scene with hidden lines removed [Andries Van Dam, personal communication, April, 1977]. In view of the

[2][Desens (69)], [Freeman (67)], [Galimberti (69)], [Loutrel (70)], [Matsushita (69)], [Newman (73)], [Sutherland (66)], [Warnock (69)], [Watkins (70)].

effort that has gone into graphic hardware development, it is surprising that the *complexity* of the hidden-line problem has received so little study, for it is here that the potential gains are the greatest. Building a box with a program implemented in microcode can, at best, achieve a speedup of a constant factor over software.[3] An improvement in the algorithm, though, can reduce the *order* of the running time, so that the speedup improves with increasing problem size.

It is important to keep in mind, however, that optimality in computer science is usually taken to mean *asymptotic* optimality, and it may occur that an "optimal" algorithm runs more slowly than a naive one for all practical problems.[4] Careful implementation and examination of the relevant constants of proportionality must be performed before one algorithm can be said to be superior to another. Machine experiments performed by Steven Reiss, Dan Hoey, Kevin Brown, and the author indicate that the $O(N \log N)$ algorithms in this thesis begin to surpass the performance of their $O(N^2)$ counterparts for problem sizes in the range of 50-100 points.

In many cases, particularly for vector graphic devices, scene components are represented as polygons. If the projections of two objects are the polygons A and B, and A lies nearer to the viewer than B, what must be displayed is A and $B - (A \cap B)$. A basic computational problem in hidden line removal is thus to *form the intersection of two polygons.* In practice we must do more than this since the image will consist of many separate polygons, all of which must be displayed, but one of the fundamental operations is pairwise intersection [Sutherland (66)]. In this chapter we obtain tight bounds for intersecting convex, star-shaped, and simple polygons. The polygon problem is an example of the first type of intersection problem we will consider:

Problem P5.1: (Form Intersection) Given two objects, form their intersection.

---

[3] In some cases, of course, a constant factor is all that is needed.

[4] For example, this is the case for Schonhage-Strassen integer multiplication [Aho (74)].

5.1.2 Pattern Recognition

One of the major techniques of pattern recognition is classification by *supervised learning*.[5] Given N points, each of which is identified as belonging to one of m samples, we wish to preprocess them so that a new (unidentified) point can be correctly classified. Figure 5.2 is a two-dimensional example in which the axes represent the weights and heights of a group of people of the same age. Males are designated by "M", females by "F". The point "u" represents a person whose weight and height are known. Can we classify "u" based on these quantities alone? What sort of decision rule should be used?

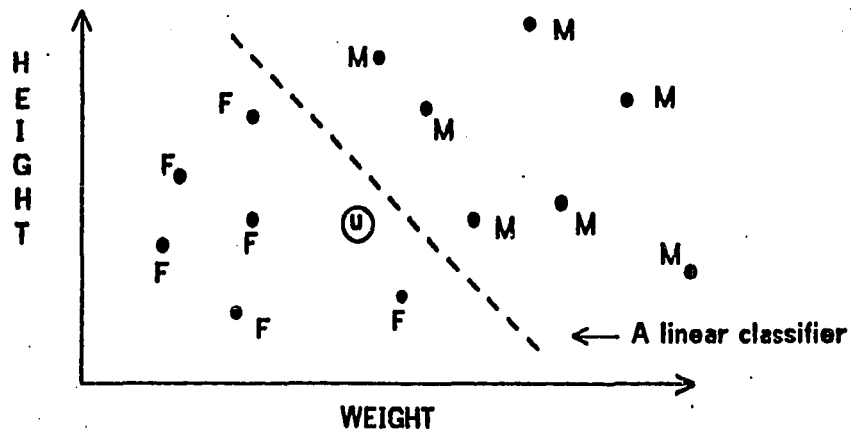


Figure 5.2: A Two-Variable Classification Problem

It is desirable, if possible, to obtain a *linear classifier* [Meisel (72)], that is, a linear function *f* such that a single comparison will suffice to determine the sample to which "u" belongs:

$$\text{IF } f(x_u,y_u) > T \quad \text{THEN } u \in M; \quad \text{ELSE } u \in F;$$

[5][Andrews (72b)], [Duda (73)], [Meisel (72)].

In the above expression, T is a threshold value. In k dimensions, the locus $f(x_1,...,x_k) = T$ is a hyperplane; in two dimensions, it is a straight line. A linear classifier performs well if it *separates* the two samples such that all points of M lie on one side and all points of F lie on the other.

Definition 5.1: Two sets are said to be *linearly separable* iff there exists a hyperplane H that separates them.

Determining the existence of a linear classifier is thus a matter of deciding whether the training samples are separable.

5.1.2.1 Separability

Separability is a classical question in combinatorial geometry. In 1903, P. Kirchberger proved the following elegant theorem, which resembles Theorem 3.3 in spirit:

Theorem 5.1: [Kirchberger (03)] Two finite plane sets P and Q are linearly separable iff every subset of four or fewer points of P ∪ Q is separable[6].

Since there are $O(N^4)$ such subsets, any algorithm based on this characterization is likely to be extremely inefficient. The theorem does not suggest a method for constructing the separating line, but merely gives a criterion for its existence.

Theorem 5.2: [Stoer (70), Theorem 3.3.9.] Two sets are linearly separable iff their convex hulls do not intersect.

This theorem is illustrated in Figure 5.3.

In developing a geometric algorithm, we have no *a priori* basis for preferring one of these theorems to the other. Only analysis will enable us to decide which will yield the faster procedure. Fortunately, we already know how difficult it is to form convex hulls. We must now solve a special case of

---

[6]The generalization to k dimensions involves the separability of every subset of k+2 or fewer points. See [Rademacher (50)].

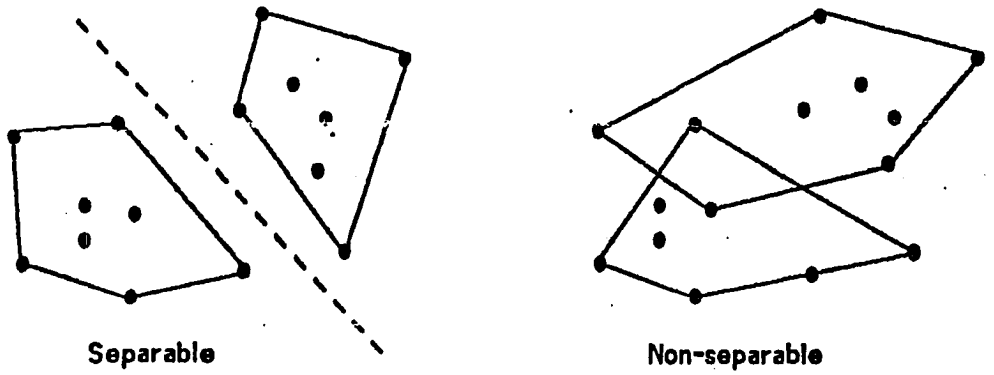


Figure 5.3: Two sets are separable iff their convex hulls are disjoint.

Problem P5.2: (Intersection Test) Given two polygons, do they intersect?

While this is no harder than actually forming the intersection, it may turn out to be easier in certain cases.

5.1.3 Wire and Component Layout

With microminiaturization proceeding at a fantastic pace, the number of components on chips, conductors on boards, and wires in circuitry has grown to the point that such hardware cannot be designed without the aid of machines. The number of elements in a single integrated circuit may easily exceed ten thousand, and each must be placed by the designer subject to a variety of electronic and physical constraints. The programs that assist in this process are largely heuristic and often produce solutions that are not feasible because two components overlap or two conductors cross.[7] Heuristic methods are used because some of the problems involved in component placement are NP-complete [Garey (76b)] The designs must therefore be subjected to exhaustive verification that involves

[7] See [Akers (72)], [Hanan (72)], and [Hanan (75)].

pairwise comparisons of all items on the chip, an expensive and time-consuming operation. This motivates the following theoretical problem:

Problem P5.3: (Pairwise Intersection) Given N objects, determine whether any two intersect.

We will, of course, be looking for an algorithm that avoids testing each object against every other. The solution to this problem, which we develop in Section 5.4, has extensive practical and theoretical applications.

5.1.4 Linear Programming

Linear programming can be viewed as a third type of intersection problem. The feasible region of a linear program is the intersection of the half-spaces determined by its constraint set. The objective function is maximized at some vertex of this convex polyhedral region. This is a convex hull problem of an entirely different nature than that studied in Chapter 3. Here we are given not the vertices of the set, but a collection of half-spaces that bound it, and are asked to *find* the vertices. We must find the *common intersection* of N objects. How difficult is this?

In one dimension linear programming is trivial. It may be formulated as

$$\text{Maximize } ax + b \quad \text{subject to} \quad a_i x + b_i \leq 0, \quad i = 1,\ldots,N. \tag{5.1}$$

The feasible region is either null, an interval, or a half-line because it is an intersection of half-lines: Each half line extends either to minus infinity or plus infinity. Let L be the leftmost point of the positively oriented half-lines and let R be the rightmost point of the negative ones. If $L > R$, the feasible region is null. If $L = R$ it is the interval [L,R]. Clearly L and R can be found in linear time, so linear programming in one dimension is an O(N) process. The complexity of linear programming in higher dimensions is going to depend on how quickly we are able to form the intersection of half-spaces. This question is taken up in greater depth in Section 5.5.1, and is mentioned here only to introduce the problem of common intersection.

5.2. Intersection of Convex Polygons

Problem P5.4: (Convex Intersection) Given two polygons, P with m vertices and Q with n vertices, form their intersection.

We assume without loss of generality that $m \leq n$.

Lemma 5.1: The intersection of N half-planes is a convex polygonal region having at most N sides.

Proof: That the intersection is a convex polygonal region is Theorem 6.5 of [Benson (66)]. We must show that the number of sides does not exceed N, which is easily established by induction. For N=1 the theorem is true, since a half-plane is bounded by a single edge. Assume the result to be true for N = k and consider the effect of intersecting one more half-plane with the existing region R. If this last half-plane coincides with one of the first k, no new sides are introduced. Otherwise, the half-plane intersects the boundary of R in at most two points, each of which creates a new vertex of R. However, the complement of H contains at least one vertex of R (since H coincides with no other half-plane), so at most one new vertex is added to R for each new half-plane.

(This completes the proof of Theorem 4.6 because we have shown that the kernel is an intersection of at most N half-planes and thus has at most N sides.)

Theorem 5.3: The intersection of a convex m-gon and a convex n-gon is a convex polygon having at most $m + n$ vertices.

Proof: The intersection of P and Q is the intersection of the $m + n$ interior half planes determined by the two polygons. By Lemma 5.1, this intersection is a convex polygonal region having at most $m + n$ sides. Since P and Q are both bounded, so is their intersection, which must therefore be a closed polygon.

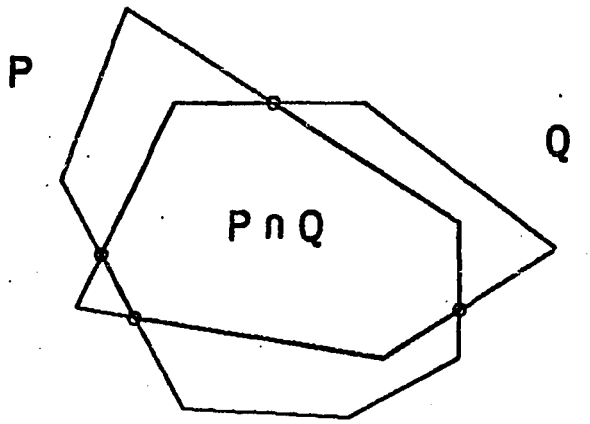


Figure 5.4: Intersection of Convex Polygons

We present an algorithm due to Dan Hoey that is based on the following observations: Let *a* and *b* be the leftmost and rightmost points of P, respectively. These points partition P into two chains of vertices, each of which is sorted by x-coordinate. The *upper chain* P_u consists of those vertices that occur in the counterclockwise sequence from *b* to *a* and the *lower chain* P_l consists of the sequence from *a* to *b*. Similarly, Q is partitioned into upper and lower chains Q_u and Q_l. Since the vertices in all four chains occur in sorted order, the chains can be merged in O(m + n) time to obtain a sorted list of all the vertices of P and Q together. Vertical lines drawn through these m + n vertices divide the plane into at most m + n + 1 slabs as in Figure 5.5. They are already ordered. The same is true of Q. We need only merge these four chains to yield a sorted list of all the vertices. (This is analogous to the procedure employed in Theorem 4.2.)

The intersection of a slab and either of the polygons is a trapezoid. Thus, within any single slab, the intersection of P and Q is an intersection of trapezoids, which can be found in constant time. We now show that all of the pieces can be found and fitted together in O(m+n) time. Let the vertical lines described above be denoted LINE[1:m+n] and for each element of LINE let us set up pointers to all edges of P and Q intersected by that line. If a line passes through a vertex of one

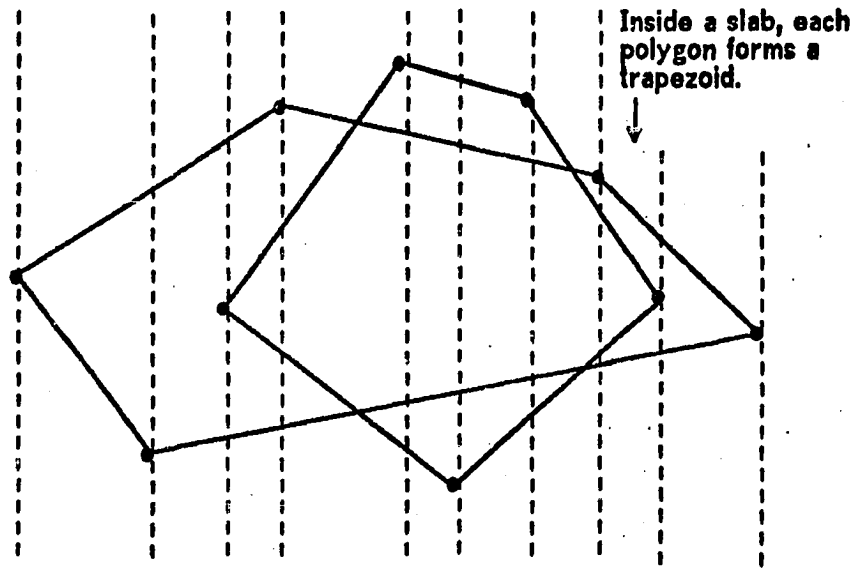


Figure 5.5: Slabs Defined by the Vertices of Two Convex Polygons.

of the polygons we set up a pointer only to the right-hand edge so intersected. Since a line can intersect a convex polygon in at most two points, no line will have more than four associated pointers, one each for edges in P_u, P_l, Q_u, and Q_l. Let the pointers be called PU, PL, QU, and QL, respectively. In other words, PU[I] designates that edge of the upper chain P_u that is intersected by the vertical line LINE[I]. If no intersection occurs, the corresponding pointer is NIL. All the pointers can be created in O(m+n) time by "merging" each chain with the sorted array LINE.

We now process the slabs sequentially by passing through LINE. The P trapezoid lying to the right of LINE[I] is bounded by LINE[I], LINE[I+1], and the edges indicated by PU[I] and PL[I]. The Q trapezoid is found in the same way. The intersection of these trapezoids has at most six vertices, two of wich lie on LINE[I], two on LINE[I+1], and two within the slab. Let R be the intersection of P and Q. We will form the upper and lower chains of R during the pass through the slabs by appending vertices of the intersection to the appropriate chain. When this procedure is finished, R can be put into standard form. We thus have

Theorem 5.4: (Dan Hoey, private communication) The intersection of a convex m-gon and a convex n-gon can be found in O(m + n) time.

5.3. Intersection of Star-Shaped Polygons

Because star-shaped polygons have the property, which they share with convex polygons, that there exists a point about which the vertices occur in angular order, we might suspect that their intersection can also be found quickly. This is not the case, as Figure 5.6 shows.

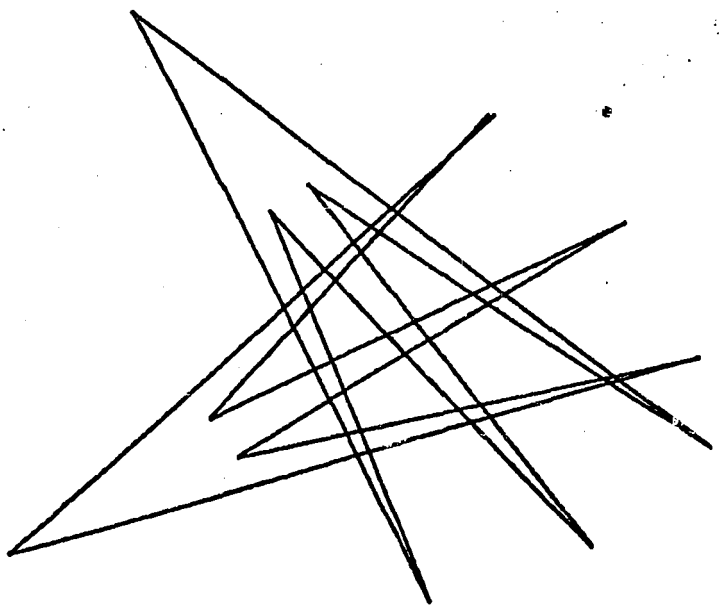


Figure 5.6: Chicken Feet. The Intersection of Two Star Polygons.

The intersection of P and Q is not a polygon itself, but is a union of many polygons. P and Q both have N vertices, and every edge of P intersects every edge of Q, so the intersection has N^2 vertices. This gives a (trivial) lower bound:

Theorem 5.5: Finding the intersection of two star-shaped polygons must require $\Omega(N^2)$ time in the worst case.

This means that the hidden line problem for polygons must also take quadratic time, since merely drawing the intersection requires that N^2 vectors be drawn. In the next section we shall explore the possibility that it may not be necessary to spend this much time if we only want to know whether P and Q intersect at all.

5.4. Intersection of Line Segments

One of the major themes of this thesis is that a large collection of seemingly unrelated problems can be solved by the same method if only their common algorithmic features can be isolated. The present section shows how a diverse set of applications can be unified and reduced to determining whether or not a set of N line segments in the plane are pairwise disjoint. We will always regard a line segment as a closed interval, that is, one which includes its endpoints.

Problem P5.5: (Segment Intersection Test) Given N line segments in the plane, determine whether any two intersect.

Below we discuss a number of applications of this problem.

5.4.1 Applications

5.4.1.1 When do two polygons intersect?

Problem P5.6: (Polygon Intersection Test) Given two simple polygons P and Q, do they intersect?

If P and Q intersect, then either P contains Q, Q contains P, or some edge of P intersects an edge of Q. (Figure 5.7.)

Since both P and Q are simple, any edge intersections that occur must be

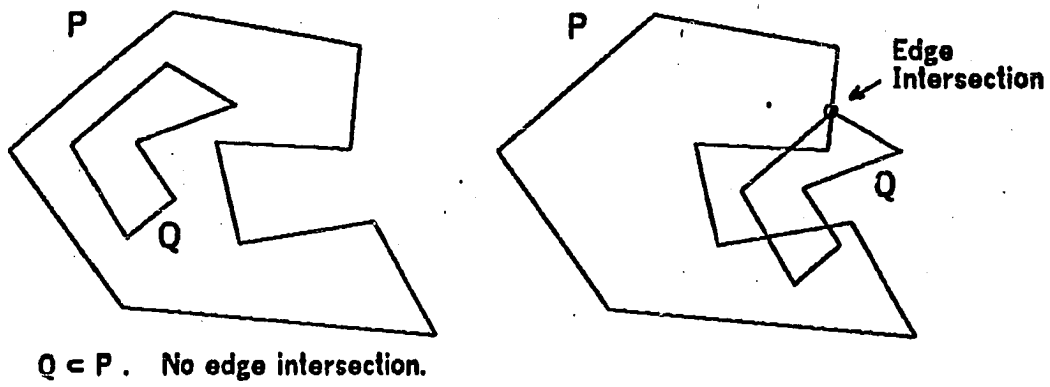


Figure 5.7: Either $P \subset Q$, $Q \subset P$, or there is an edge intersection.

between edges of different polygons. Let T(N) be the time required to solve Problem P5.5. We can then detect any edge intersection between P and Q in T(m + n) operations. If no intersection is found, we still must test whether $P \subset Q$ or $Q \subset P$.

If P is interior to Q, then *every* vertex of P is interior to Q, so we may apply the single-shot point inclusion test of Theorem 4.2 in O(n) time, using *any* vertex of P. If this vertex is found to lie outside Q, we can learn by the same method in O(m) time whether $Q \subset P$. Therefore we have

Theorem 5.6: Intersection of simple polygons is linear-time reducible to line-segment intersection testing:

$$\text{POLYGON INTERSECTION} \propto_N \text{LINE-SEGMENT INTERSECTION}$$

Proof: Let polygons P and Q both have N vertices. In O(N) time we will reduce the problem of determining whether P and Q intersect to four line-segment intersection problems, each involving at most N/2 points. Assume that N is even; if N is odd the argument is only slightly more involved. Partition P into two edge sets P_e and P_o consisting of its even- and odd-numbered edges,

respectively. Likewise, partition Q into edge sets Q_e and Q_o. In each of the four sets thus created there are no edge intersections. Now solve the line-segment intersection problem on each of the four sets $\{P_e \cup Q_e\}$, $\{P_e \cup Q_o\}$, $\{P_o \cup Q_e\}$, and $\{P_o \cup Q_o\}$. If any intersections are found during the solution of these subproblems, then P and Q intersect. If no intersections are found, then either P and Q are disjoint or one contains the other. The latter case can be decided in O(N) time by the method described above.

We have already seen that the complexity of algorithms that deal with polygons can depend on whether the polygons are known to be simple or not. For example, the convex hull of a simple polygon can be found in O(N) time (Theorem 3.20), but Ω(N log N) is a lower bound for non-simple polygons. It would be useful, therefore, to have an algorithmic test for simplicity.

Problem P5.7: (Simplicity Test) Given a polygon, is it simple?

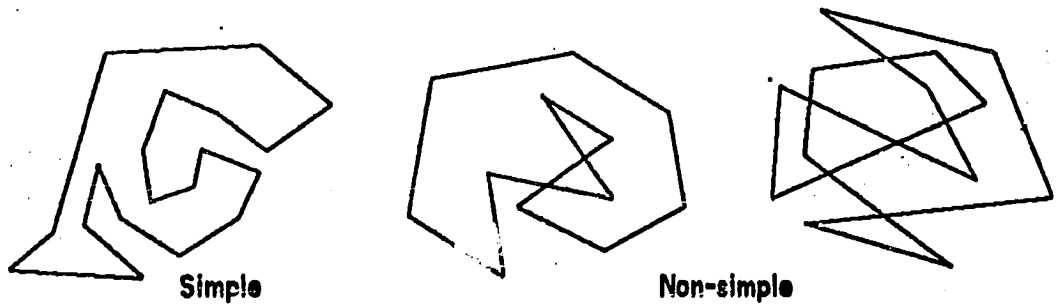


Figure 5.8: Simple and Non-simple Polygons.

5.4.2 A Segment Intersection Algorithm

A reliable guideline for the development of geometric algorithms is to start in one dimension and achieve a complete understanding of the problem in that setting, including lower bounds. This principle seems self-evident, yet it is often ignored by researchers who feel that "elementary" problems are not worth solving. The result is frequently a poorly-understood heuristic algorithm whose performance is attested to by volumes of experimental evidence and timing graphs, but no analysis. While such an approach may produce a quick answer to the problem at hand, it does little to advance our knowledge of algorithm design or complexity. With this in mind, we will proceed to build an intersection algorithm starting at the most basic level.

Suppose we are given N intervals on the real line and wish to know whether any two overlap. This can be answered in $O(N^2)$ time by inspecting all pairs of intervals, but a better algorithm based on sorting comes to mind almost immediately. Let us designate each of the 2N endpoints as either "left" or "right" and sort them lexicographically as follows. A left endpoint at x will be represented by the ordered pair (x, L) and a right endpoint by the pair (x, R). We may ensure that, among all points sharing a given coordinate x, the left endpoints precede the right ones by taking (x, L) < (x, R) in the lexicographic ordering. Now, the intervals themselves are disjoint iff the endpoints occur in alternating order: L R L R ... R L R and no two consecutive points have the same x-value. This check can be performed in O(N) time, once the points have been sorted, so the total time required is O(N log N).
The two questions we will want to deal with are whether this algorithm can be improved and whether it generalizes to two dimensions.

To show a lower bound, we will exhibit a correspondence between the segment overlap problem and a basic question in set theory.

Theorem 5.7: O(N log N) comparisons are necessary and sufficient to determine whether N intervals are disjoint, if only polynomial functions of the inputs can be computed.

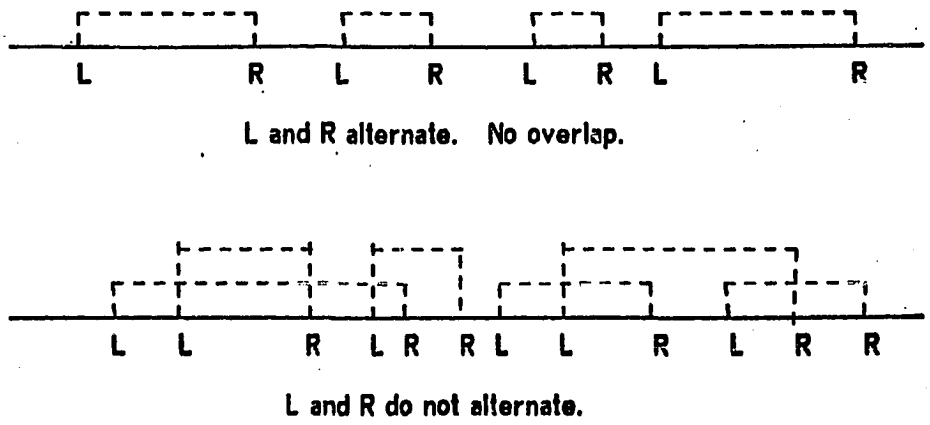


Figure 5.9: Detecting Interval Overlap.

Proof: We show that ELEMENT UNIQUENESS $\propto_N$ INTERVAL OVERLAP. Given a collection of N real numbers x_i, these can be converted in linear time to N (closed but null) intervals $[x_i,x_i]$, which overlap iff the original points were not distinct.

How severe is the restriction to polynomial functions? For one thing, it forbids the use of the FLOOR function, which, as we shall see in Section 6.1.8, is a very powerful operation. No techniques are known that would enable us to prove Theorem 5.7 if the floor function were allowed, but we conjecture that the lower bound would not change. Theorem 5.7 applies *a fortiori* in all dimensions.

5.4.2.1 Two Dimensions

Let us explore what really happens when we sort to detect overlap. The motivation for doing this is that there is no natural liner ordering on line segments in the plane, so a generalization based solely on sorting will have to fail. If we are able to understand the essential features of the algorithm, though, we may be able to extend it to the plane.

Overlap occurs if and only if two segments contain some common point. Each point on the real line has associated with it a set consisting of the intervals that cover it. This defines a function $C: R \rightarrow 2^N$ from the reals to subsets of 1,...,N. The value of C function can change only at the 2N endpoints of the intervals. If the cardinality of $C(x)$ ever exceeds one, an overlap has occurred. To detect this, we first sort the endpoints and then set up a highly primitive data structure that contains just a single object, the "current" interval. Scanning the endpoints from left to right, we INSERT an interval into the data structure when its left endpoint is encountered and DELETE it when its right endpoint is passed. If an attempt is ever made to INSERT when the data structure is already occupied, an overlap has been found; otherwise, no overlap exists. Since the processing of each endpoint in this way takes only constant time after sorting, the checking process requires no more than linear time.

In two dimensions we are obliged to define a new order relation and make use of a more sophisticated data structure.[8]. Consider two non-intersecting line segments A and B in the plane. We will say that A and B are *comparable* if there exists a vertical line through some point on the x-axis that passes through both of them. We define the relation *above at x* in this way: A is *above* B *at* x, written $A >_x B$, if A and B are comparable at x and the intersection of A with the vertical line lies above the intersection of B with that line.[9] In Figure 5.10, we have the following relationships among the line segments A, B, C and D:

$$B >_u D, \quad A >_v B, \quad B >_v D, \text{ and } A >_v D .$$

Segment C is not comparable with any other segment.

---

[8]For purposes of discussion, we will assume that no segment is vertical and that no three segments meet in a point. If either of these conditions is not met, the algorithms we develop will be longer in detail, but not in asymptotic running time.

[9]This order relation and the algorithm derived from it were developed by Dan Hoey. The author provided a proof of correctness of the algorithm and supplied the applications discussed above. These results were reported jointly in [Shamos (76b)].

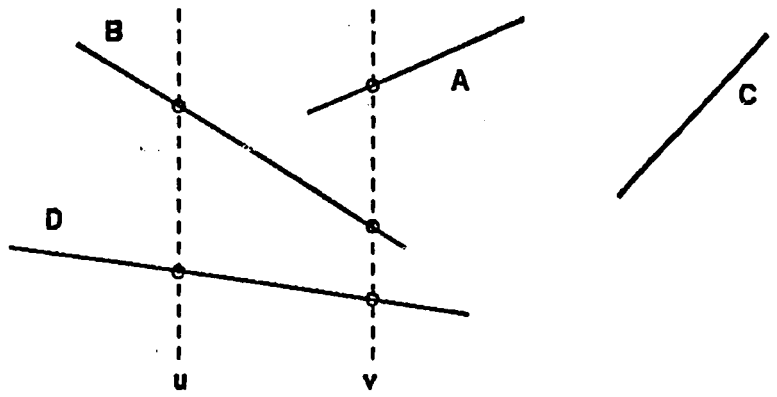


Figure 5.10: An Order Relation Between Line Segments.

Note that the relation $>_x$ is a total order, which changes as x sweeps from left to right. Segments enter and leave the ordering, but it always remains total. The ordering can change in only three ways:

1. The left endpoint of segment A is encountered. In this case A must be added to the ordering.

2. The right endpoint of A is encountered. A must be removed from the ordering because it is no longer comparable with any other segment.

3. An intersection point of two segments A and B is reached. Here, A and B exchange places in the ordering.

Notice that if segments A and B intersect, then there is some x for which A and B are *consecutive* in the ordering $>_x$. Were we able to maintain the ordering as x moves along, we would be able to report all intersections. Since there may be as many as N(N-1)/2 intersections (if every pair of segments cross), this procedure might take $O(N^2)$ time. Below we describe an algorithm which finds an intersection if one exists, but does not go to the expense of keeping complete information about the ordering.

Once we abandon the idea of finding all intersections, things become greatly simplified. Since the segments involved in an intersection must first become neighbors in the total order, we may proceed as in the one-dimensional algorithm. Instead of holding just a single object, however, the data structure must be able to retain a total ordering on as many as N objects and must permit fast updating. After sorting the 2N endpoints, we scan from left to right, inserting a segment into the data structure when its left endpoint is encountered and deleting it when its right endpoint is passed, *checking segments for intersection when they become consecutive in the total order.* This will require a structure T on which we can perform the following operations:

1. INSERT(A,T). Insert segment A into the total order maintained by T.

2. DELETE(A,T). Delete segment A from T.

3. ABOVE(A,T). Return the name of the segment immediately above A in the ordering.

4. BELOW(A,T). Return the name of the segment immediately below A in the ordering.

Algorithm A5.1: Intersection of Line Segments

1. Sort the 2N endpoints lexicographically by x and y and place them in the real array POINT[1:2N].

2. FOR i ← 1 UNTIL 2N DO BEGIN
   P ← POINT[i];
   Let S be the segment of which POINT[i] is an endpoint;
   IF P is the left endpoint of S THEN
     BEGIN
       INSERT(S,T);
       A ← ABOVE(S,T);
       B ← BELOW(S,T);
       IF A intersects S THEN OUTPUT(A,S);
       IF B intersects S THEN OUTPUT(B,S);
     END
   ELSE (P is the right endpoint of S)
     BEGIN
       A ← ABOVE(S,T);
       B ← BELOW(S,T);
       IF A intersects B THEN OUTPUT(A,B);
       DELETE(S,T);
     END
   END

Theorem 5.8: Algorithm A5.1 finds an intersection if one exists.

Proof: Since the algorithm only reports an intersection if it finds one, it will never falsely claim that two segments cross, and we may turn our attention to the possibility that an intersection exists but remains undetected. We will show that the algorithm correctly finds the leftmost intersection point, L. (See Figure 5.11.) If several leftmost intersection points exist, let L be the one

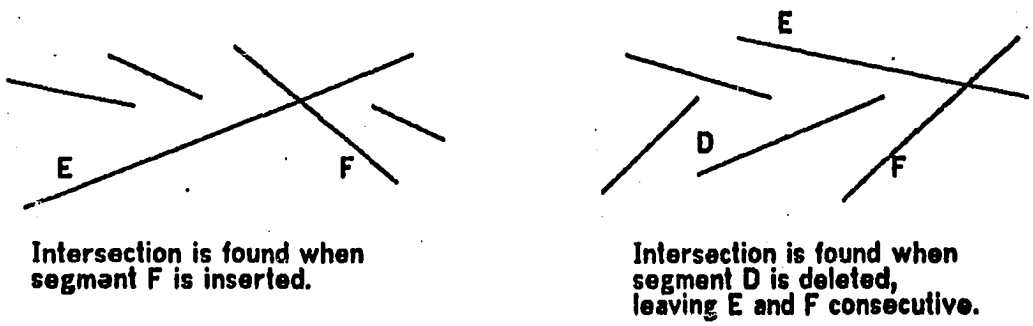
with least y-coordinate. Suppose that L is the intersection of segments E and F, and consider the leftmost point at which E and F become consecutive in the total order. Three cases arise:

1. One of the segments is inserted and the other lies immediately above or below it in the ordering. This case is detected by the first IF block in Algorithm A5.1.

2. Both segments are already in T and an intervening segment is deleted, leaving them consecutive. This is detected by the ELSE block.

3. An intervening segment crosses either E or F. This possibility is ruled out by the fact that L is leftmost and that no three segments intersect in a point.

Thus in any event the intersection is found.



Intersection is found when segment F is inserted.

Intersection is found when segment D is deleted, leaving E and F consecutive.

Figure 5.11: Finding the Leftmost Intersection.

Though simple, Algorithm A5.1 has some curious properties. Even though the leftmost intersection is always found, it is not necessarily the *first* intersection to be found. (The reader may test his understanding of the algorithm by characterizing exactly which intersection *is* found first.) Also, since the algorithm only performs O(N) intersection tests, it may fail to find some intersections. Figure 5.12

illustrates the operation of the algorithm, showing exactly how an intersection is found.

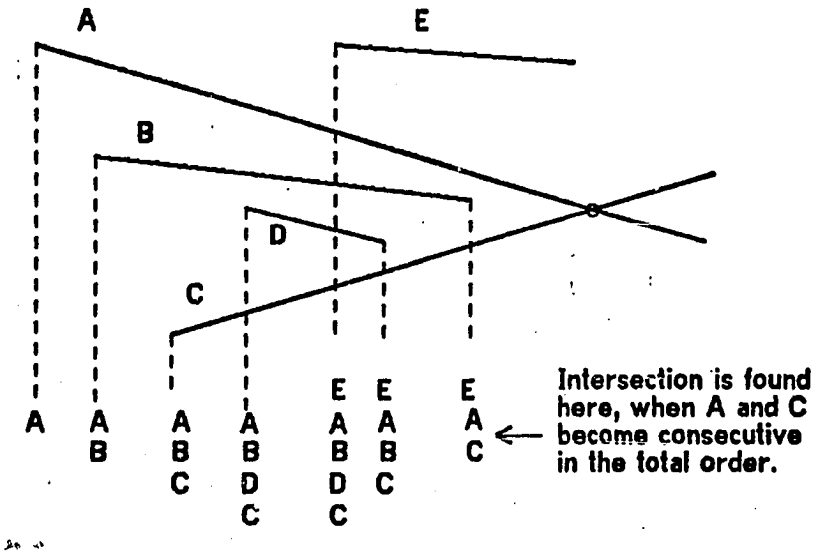


Figure 5.12: The Intersection Algorithm in Operatiom.

Theorem 5.9: Whether any two of N line segments in the plane intersect can be determined in O(N log N) time, and this is optimal.

Proof: We show that Algorithm A5.1 can be implemented in O(N log N) time. The sorting of the 2N endpoints in Step 1 can be accomplished in O(N log N) time. Using a balanced tree scheme, we may implement the operations INSERT, DELETE, ABOVE, and BELOW so that each of them can be performed in O(log N) time, worst-case [Aho (74)]. The FOR-loop of Step 2 contains only a constant number of these operations and the loop is executed 2N times, so O(N log N) time suffices. Optimality was shown in Theorem 5.7. O(N) storage suffices because we need only store a balanced tree with at most N leaves.

Corollary 5.1: [Cf. Theorems 5.6, 5.9] Whether two simple polygons P and Q intersect can be determined in O(N log N) time, where N is the total number of edges in both P and Q.

Further results have been obtained using variations of Algorithm A5.1 and can be found in [Shamos (76b)]:

Corollary 5.2: Whether a given polygon is simple can be determined in O(N log N) time.

Corollary 5.3: Whether a planar straight-line graph on N vertices contains crossing edges can be determined in O(N log N) time.

The algorithms that achieve both of the above results are modifications to the basic intersection algorithm A5.1 that take into account multiple edges that share a common vertex.

Theorem 5.10: Whether any two of N simple (but possibly non-convex) k-gons intersect can be determined in O(Nk log Nk) time.

Theorem 5.11: Whether any two of N convex k-gons intersect can be determined in O(N(k + log N log k)) time.

5.5. Common Intersection

5.5.1 Intersection of Half-Planes

" 'trash', he said, 'but with a kernel in it'."
- Tennyson, *The Princess*.

In Section 4.3, we showed that finding a point in the kernel of a star-shaped polygon is an essential step in the preprocessing needed to answer the inclusion question. At that time, we postponed the development of a kernel algorithm until the necessary tools were available. In this section we transform the kernel problem into one of finding the common intersection of N half-planes, and produce a divide-and-conquer algorithm for its solution.

Each edge of a star-shaped polygon determines a half-plane in which the kernel must lie. (Figure 5.13.) These half-planes are known as the *interior* half-planes or, if the polygon is in standard form, the *left* half-planes.

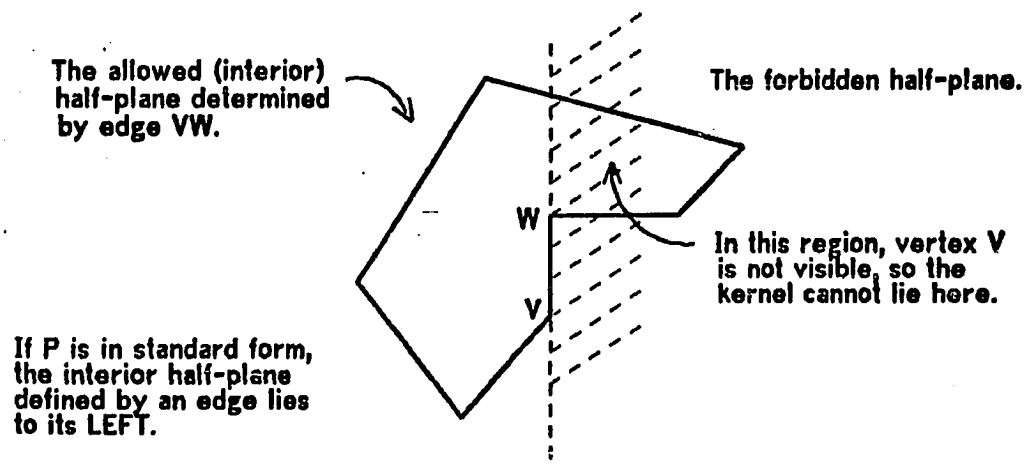


Figure 5.13: Each edge of P determines an allowed half-plane.

Theorem 5.12: [Yaglom (61)] The kernel of a polygon is the intersection of its left half-planes.

Thus immediately we have

$$\text{KERNEL} \propto_N \text{HALF-PLANES}$$

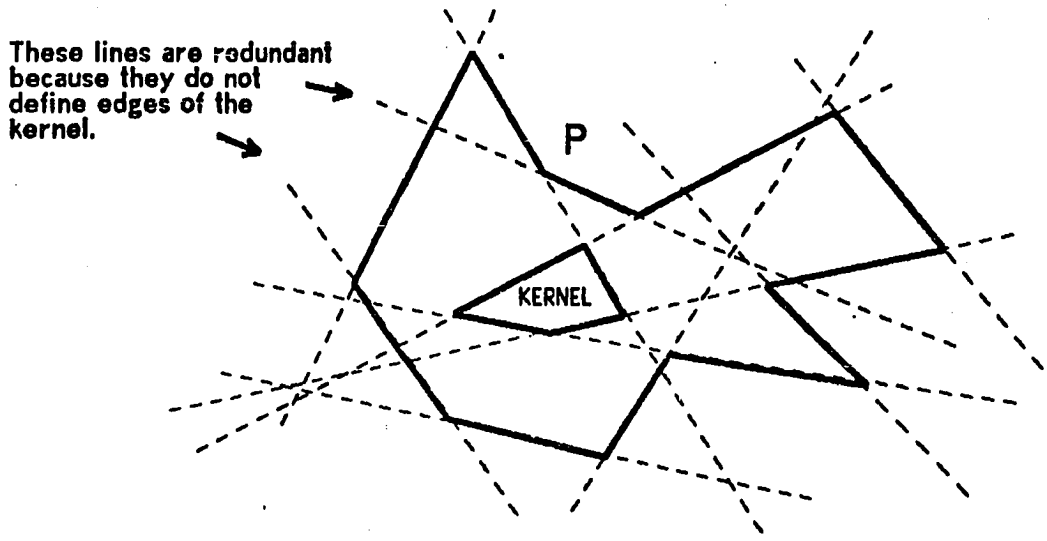


Figure 5.14: The Kernel of a Polygon is the Intersection of its Left Half-Planes.

Possibly the most familiar setting in which the intersection of half-planes arises is in linear programming [Gass (69)]. A two-variable linear program can be formulated as:

$$\text{Maximize } ax + by + c \text{ , subject to } a_i x + b_i y + c_i \leq 0 \text{ , } \quad i = 1,\ldots,N \text{ . } \quad (5.2)$$

The *feasible region* is the set of points (x,y) which satisfy the constraints in (5.2). Each constraint determines a half-plane in which the feasible region must lie. The region itself is the intersection of these half-planes. The objective function ax + by + c defines a family of parallel lines. *The lines of this family that support the feasible region pass through the vertices that minimize and maximize the objective function* [Gass (69)]. (See Figure 5.15.)

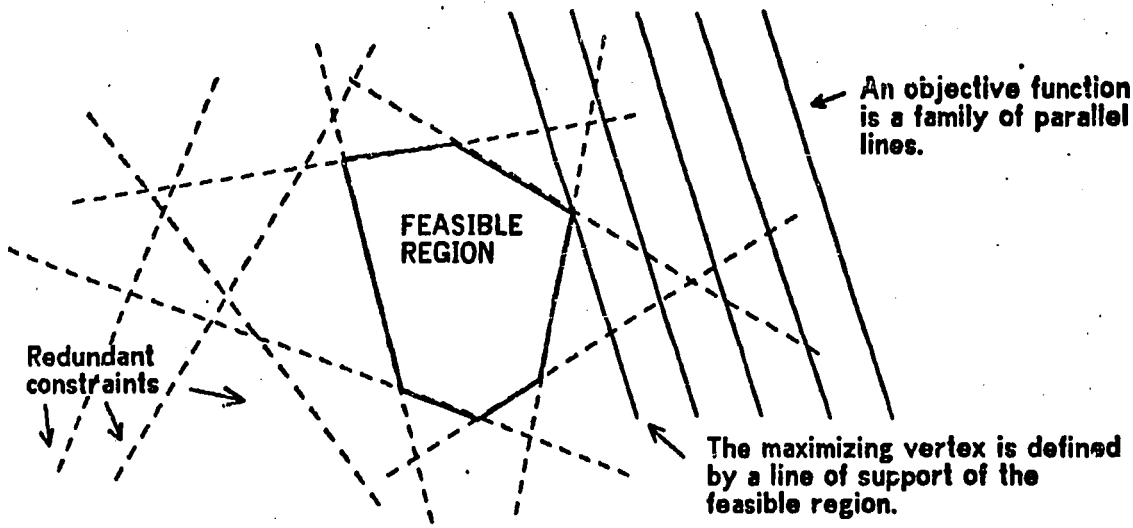


Figure 5.15: A Two-Variable Linear Program.

We already know how to find lines of support of a convex polygon in O(log N) time (Theorem 3.28). Thus,

2-VARIABLE LINEAR PROGRAMMING $\propto_N$ HALF-PLANES

There is a simple quadratic algorithm for forming the intersection of N half-planes. Let us assume that we already have the intersection of the first i half-planes. This is a convex polygonal region of at most i sides, though it is not necessarily closed. Intersecting this region with the next half-plane is a matter of slicing the region

with a line and retaining either the right or left piece. This can be done in O(1) time in the obvious way. The total work required is $O(N^2)$, but the algorithm has the advantage of being on-line.

Let us see if any improvement is possible with a divide-and-conquer approach.[10] Given N half-planes H_i, we want to form the intersection

$$H_1 \cap H_2 \cap \ldots \cap H_N \quad .$$

Because the intersection operator is associative, the terms may be parenthesized in any way we wish:

$$(H_1 \cap \ldots \cap H_{N/2}) \cap (H_{N/2+1} \cap \ldots \cap H_N) \; . \qquad (5.3)$$

The term in parentheses on the left is an intersection of N/2 half-planes and hence is a convex polygonal region of at most N/2 sides. The same is true of the term on the right. Since two convex polygonal regions each having k sides can be intersected in O(k) time by Theorem 5.4, the middle intersection operation in (5.3) can be performed in O(N) time. This suggests the following recursive algorithm:

Algorithm A5.2: Intersection of Half-Planes

INPUT: N half-planes defined by directed line segments
OUTPUT: Their intersection, a convex polygonal region.

1. Partition the half-planes into two sets of equal size.

2. Recursively form the intersection of the half-planes in each subproblem.

3. Merge the solutions to the subproblems by intersecting the two resulting convex polygonal regions.

[10] Stan Eisenstat taught me the value of D&C, and, in particular, suggested that it would work here.

If T(N) denotes the time sufficient to form the intersection of N half-planes by this algorithm, we have

$$T(N) = 2T(N/2) + O(N) = O(N \log N) . \qquad (5.4)$$

This is a "classical" example of D&C.

Theorem 5.13: The intersection of N half-planes can be found in O(N log N) time, and this is optimal.

Proof: The upper bound follows from equation (5.4). To prove the lower bound, we show that

$$\text{SORTING } \alpha_N \text{ HALF-PLANES} .$$

Given N real numbers $x_1,\ldots, x_N$, let H_i be the half-plane containing the origin that is defined by the line of slope x_i tangent to the parabola $y = x^2$. The intersection of these half-planes is a convex polygonal region whose successive edges are ordered by slope. Once this region is formed, we may read off the x_i in sorted order.

Corollary 5.4: The kernel of an N-gon can be found in O(N log N) time.

Note, though, that the *lower* bound of $\Omega(N \log N)$ proved in Theorem 5.13 does not apply to the kernel problem because the edges of a *simple* polygon cannot be in arbitrary positions and the reducibility from sorting fails. There is no reason to believe than any more than linear time is required to find the kernel. In fact, by clever manipulation of edge lists Lee and Preparata [Lee (77a)] have shown:

Theorem 5.14: The kernel of a simple N-gon can be found in O(N) time.

Returning to polygon intersection problems, we have

Theorem 5.15: The common intersection of N convex k-gons can be found in O(Nk log N) time.

Proof: It is straightforward to achieve O(Nk log Nk) time by intersecting the Nk left half-planes of the polygons. To reduce this, we will treat the polygons as N units rather than as a collection of Nk edges. Let T(N,k) be the time sufficient to solve the problem. The intersection of N/2 convex k-gons is a convex polygon of at most Nk/2 sides. Two of these can be intersected in time ckN, for some constant c. So by recursively splitting the problem as in Algorithm A5.2, we have

$$T(N,k) = 2T(N/2,k) + ckN = O(Nk \log N) ,$$

since T(2,k) = O(k) by Theorem 5.3.

Theorem 5.16: A linear program in two variables and N constraints can be solved in O(N log N) time. Once this has been done, a new objective function can be maximized or minimized in O(log N) time.

Proof: The N log N result is an immediate consequence of Theorem 5.13. Minimizing or maximizing an objective function is just a search for supporting lines, and Theorem 3.28 applies.

Let us compare this performance with the that of the Simplex algorithm [Gass (69)]. Simplex operates by moving from vertex to vertex on the feasible region, spending O(N) time for each move (the time required to select the new entering variable). It is easy to see that, in the worst case, Simplex will have to visit every vertex, for a total of $O(N^2)$ time. (In this respect it is very similar to Jarvis's algorithm in Section 3.6.) Furthermore, in order to maximize a new objective function, Simplex must inspect every constraint, so it will use O(N) time. In other words, *Simplex is not optimal.* This is not altogether a surprise, but no other

algorithm is known that is superior to Simplex in any number of dimensions.[11]

5.5.1.1 Expected Time to Intersect Half-Planes

One of the striking features of the Simplex algorithm is that, while its worst case is known to be exponential in dimension (and quadratic in two variables), its behavior is almost always excellent in practice [Klee (65)]. No explanation for this phenomenon is known, but we could hardly claim that Algorithm A5.2 is better than Simplex if it is faster only for some highly unrealistic worst case. To demonstrate once again the power of divide-and-conquer algorithms, we prove that the expected running time of Algorithm A5.2 is linear for a wide class of input distributions.[12] The average case of Simplex in the plane is easy to analyze. If r of the n constraints are relevant (non-redundant), then Simplex requires O(rn) time because it spends O(n) time at each iteration. To obtain a faster algorithm, we invoke the same principle that was presented in Section 3.7, namely, if the expected sizes of the subproblem solutions are small, then the merge step of the D&C algorithm can be performed in sublinear time. This will be the case if many of the half-planes are *redundant*, i.e., do not form edges of the intersection polygon. We now show that most of the half-planes in a random problem can be expected to be redundant.

It is fairly easy to exhibit a reasonable probability distribution for random points in the plane; it is far less obvious how to model a random selection of half planes. [Ziezold (70)] proposes the following. Let K_0 be a bounded convex region of the plane that contains another convex region K_1. If N lines L_i are drawn independently and at random to meet K_0 but not K_1, and we define H_i to be the closed half-plane bounded by L_i that contains K_1, consider E(v), the expected number of vertices of the intersection of all the H_i. (See Figure 5.16.)

---

[11]We must point out that explicit construction of the feasible polytope is not a viable approach to linear programming in higher dimensions because the number of vertices can grow exponentially with dimension. It is already known, however, that Simplex can be beaten in three dimensions [Preparata (77c)].

[12]These results were obtained jointly with Jon Bentley and have appeared in [Bentley (77b)].

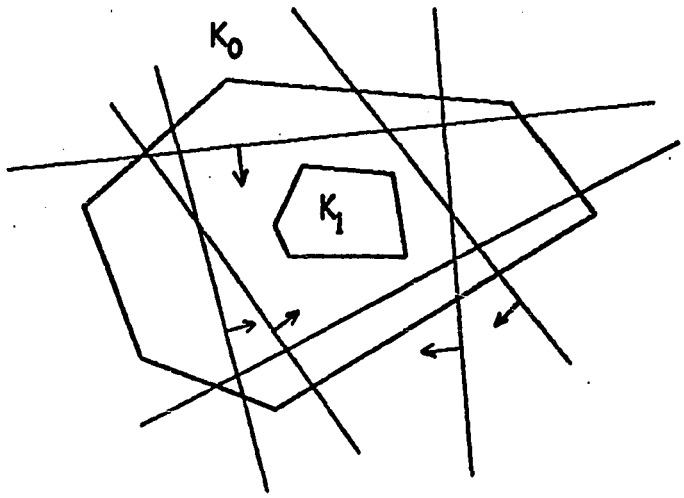


Figure 5.16: A Method of Choosing Random Half-Planes.

Preliminary results were obtained by [Renyi (64)], and Ziezold has shown by duality that E(v) is of the same asymptotic order as the expected number of points on the hull of a set of N points drawn uniformly within K_1. If K_1 shrinks to a point, then E(v) approaches the constant $\pi^2/2$. In any event we will have $E(v) = O(N^p)$, $p<1$, and a linear average-case algorithm for intersecting N half-planes results. This leads immediately to an O(N) expected-time algorithm for linear programming in two variables.

We see that the expected number of *redundant* half-planes -- those that do not define faces of the feasible region -- is very large, which may, in part, account for the excellent observed behavior of the Simplex Method.

5.6. Unsolved problems

1. Suppose that the intersection of two simple N-gons has k edges. (This intersection may consist of disjoint regions.) Does there exist an O(max(k,N log N)) algorithm to construct it?

2. Given N line segments in the plane, how much time is required to *count* the number of pairs that intersect?

3. Of N line segments, suppose that k pairs intersect. Does there exist an O(max(k,N log N)) algorithm to list them?

4. Prove that solving a linear program in two variables requires $\Omega(N \log N)$ time. (Theorem 5.13 implies that $\Omega(N \log N)$ time is necessary to form the feasible region, but this may not be required to maximize a single objective function.)

5. What is the complexity of intersecting two k-dimensional convex polytopes? That is, what is its asymptotic dependence on n and k? How difficult is it to determine whether two (possibly non-convex) polyhedra intersect?

6. A large class of problems concerns the *union* of figures. How difficult is it to find the area of the union of N rectangles whose sides are not necessarily parallel to the coordinate axes? Straighforward application of the principle of inclusion-exclusion yields a 2^N algorithm! What about the area of the union of N circles?[13] Are the various union problems dual to any intersection problems?

[13]The union problems were brought to my attention by Stuart Feldman of Bell Laboratories.

5.7. Summary

The importance of intersection problems stems both from their frequent occurrence in applications and their use as basic tools in computational geometry itself. We divide the problems into three classes, each of which requires a different approach. Forming the intersection of two objects is an essential element in the hidden-line problem and pattern recognition. The intersection of convex polygons can be found in linear time, but, for star-shaped and simple polygons quadratic time may be required. Detecting the intersection of objects is easier, for determining whether two general polygons intersect requires only O(N log N) time. We accomplish this by isolating and solving the problem of detecting whether any two of N line segments intersect. A lower bound of Ω(N log N) follows by showing that element uniqueness is linear-time reducible to line-segment intersection. By making use of a partial order on line segments in the plane, we are able to detect an intersection in O(N log N) time, while finding all intersections could require $O(N^2)$ time. This optimal algorithm is used to test whether a polygon is simple, whether a plane embedding of a graph has any intersecting edges, and whether two simple polygons overlap.

Finally, we use the divide-and-conquer method to find the common intersection of N half-planes. The merge step of this algorithm is the linear-time procedure for intersecting convex polygons developed earlier. The common intersection of half-planes is shown to require Ω(N log N) time in the worst-case, and our algorithm achieves this bound, yielding an O(N log N) kernel algorithm and a fast procedure for solving linear programs in two variables. This latter algorithm, and the one due to Preparata cited earlier, are the only procedures known to be faster than the Simplex method in any number of dimensions. By appealing to recent results in stochastic geometry, we are able to demonstrate that the half-planes algorithm is even better in an average-case sense, running in O(N) expected time.

Chapter 6
Closest-Point Problems

In Section 3.10, we gave an O(N log N) algorithm for finding the two farthest points of a plane set. Finding the two closest points would seem to be a simple extension, but it is not. The two farthest points are necessarily hull vertices, and we may exploit convexity to give a fast algorithm; the two closest points do not necessarily bear any relation to the convex hull, so a new technique must be developed, which is the subject of this chapter. We will be concerned with a large class of problems that involve the proximity of points in the plane, and our goal will be to deal with all of these seemingly unrelated tasks via a single algorithm, one which discovers, processes, and stores compactly all of the relevant proximity information. To do this, we revive a classical mathematical object, the *Voronoi diagram*, and turn it into an efficient *computational* structure that permits considerable improvement over the best presently-known algorithms. While all of the problems to be treated here involve Euclidean distance, all of the example could profitably be analyzed in the L_1 and L_∞ norms. In this chapter all of the geometric tools we have developed, including hull-finding, searching, and polygon intersection, will be brought to bear on these "closest-point" problems.

6.1. The Problems

6.1.1 Closest Pair

Problem P6.1: (Closest Pair) Given N points in the plane, find two that are closest together.[1]

[1]More than one pair may be closest. We will consider finding <u>any</u> such pair as a solution to this problem.

This problem is so easily stated and important that we must regard it as one of the fundamental questions of computational geometry, both from the point of view of applications and pure theoretical interest.

The central algorithmic issue is whether it is necessary to examine every pair of points to find the minimum distance thus determined. This can be done in $O(kN^2)$ time in k dimensions, for any k. In one dimension a faster algorithm is possible, based on the fact that any pair of closest points must be consecutive in sorted order. Thus we may sort the given x_i in O(N log N) steps and perform a linear-time scan that computes $x_{i+1} - x_i$, $1 \le i < N$. This algorithm, obvious as it is, is nonetheless optimal:

Theorem 6.1: O(N log N) comparisons are necessary and sufficient to find the two closest of N points on a line, if comparisons are allowed only between linear functions of the inputs.

Proof: We show that ELEMENT UNIQUENESS $\propto_N$ CLOSEST PAIR, whence the lower bound follows from Theorem 3.22. Given a set of real numbers x_i, treat them as points on the real line and find the two closest. If the distance between them is non-zero, the points are distinct. Since a set in one dimension can always be embedded in *k* dimensions, the lower bound generalizes.

In Section 6.2, we produce an algorithm that achieves this bound.

6.1.2 All Nearest Neighbors

"Better is a neighbor that is near."

- Proverbs 27:10.

Problem P6.2: (All Nearest Neighbors) Given N points in the plane, find a nearest neighbor of each. Two points y and z are both nearest neighbors of x iff $d(x,y) = d(x,z)$.[2]

[2]Note that a point need not have a unique nearest neighbor. It has at most six,

A solution to this problem is a collection of N ordered pairs (a,b), in which b is a nearest neighbor of a. Since one of these N pairs is a closest pair, we have

$$\text{CLOSEST PAIR} \propto_N \text{ALL NEAREST NEIGHBORS},$$

so Theorem 6.1 implies that this problem must require $\Omega(N \log N)$ comparisons as well. In one dimension the same sorting algorithm yields all nearest neighbors, but what happens in higher dimensions? Is it conceivable, for example, that P6.2 is asymptotically no more difficult than P6.1?

The set of neighbor pairs defines a binary relation "$\rightarrow$" on the set of points, where we write $a \rightarrow b$ iff b is a nearest neighbor of a. The graph of this relation is pictured in Figure 6.1. Note that it is not necessarily symmetric, that is, $a \rightarrow b$ does not necessarily imply $b \rightarrow a$.

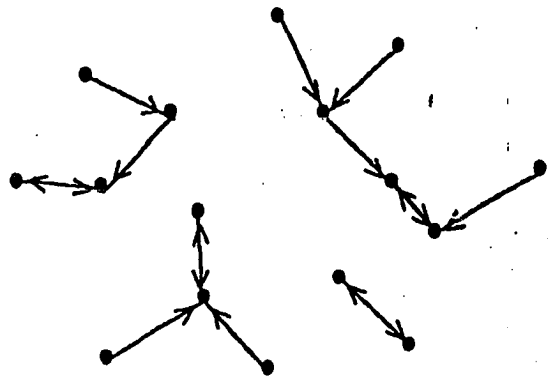


Figure 6.1: The Nearest-Neighbor Relation

A pair which does satisfy symmetry ($a \rightarrow b$ *and* $b \rightarrow a$) is called a *reciprocal pair*. In mathematical ecology, the number of reciprocal pairs is used to detect whether

---

though, in two dimensions, and at most twelve in three dimensions. This maximum number of nearest neighbors in any dimension k is the same as the maximum number of unit spheres that can placed so as to touch a given one. ([Saaty (70)] states that this number is not known for k greater than twelve.)

members of a species tend to occur in isolated couples [Pielou (77)]. The actual number of reciprocal pairs is computed, and the result compared with the number expected under the null hypothesis. Other studies involving nearest-neighbor computations arise in studying the territoriality of species [Pielou (77)], in which the *distribution* of nearest-neighbor distances is of interest, as well as in geography [Kolars (74)] and molecular physics [Brostow (77)]. These distances can also be used for testing the randomness of spatial patterns: a collection of points is random iff the distribution of distances from a random point (x,y) in the plane to the nearest pattern point p_i is identical to the distribution of nearest-neighbor distances among the p_i themselves [Hopkins (54)]. The latter information comes from solving P6.2, and the need for the former motivates our next problem.

6.1.3 Nearest-Neighbor Search

Problem P6.3: (Nearest-Neighbor Search) Given N points in the plane, with preprocessing allowed, how quickly can the one nearest to a new given point *p* be found?

This problem is posed in [Knuth (73)].

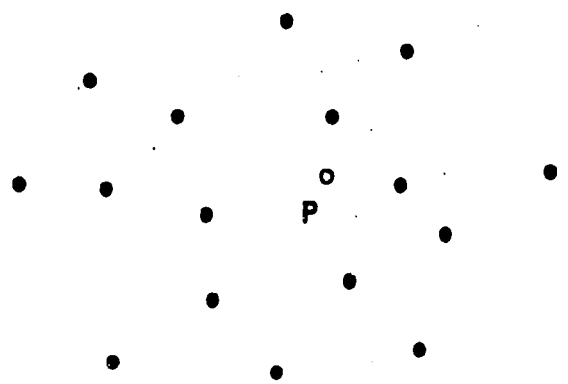


Figure 6.2: To Which Point is P Closest?

We may solve it in O(*k*N) time in *k* dimensions, but we are interested in using

preprocessing to speed the search. There are a multitude of applications for such fast searching, one of which is the *classification problem*, which we spoke of as "supervised learning" in our discussion of linear separability in Section 5.1.2.

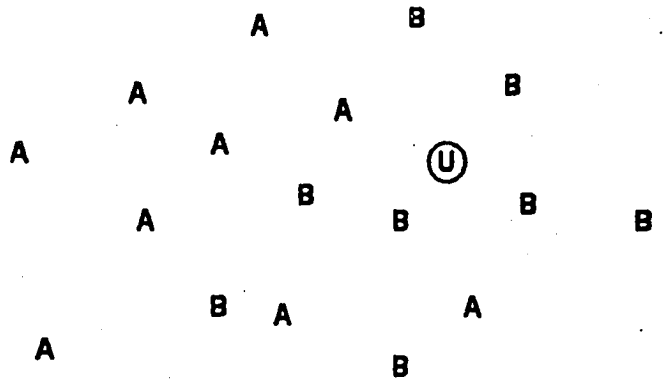


Figure 6.3: The Nearest-Neighbor Rule

An important classification method is the *nearest-neighbor rule* [Duda (73)], which states that when an object must be classified as being in one of a number of known populations, it should be placed in the population corresponding to its nearest neighbor. For example, in Figure 6.3 the unknown point U would be classified "B". If many objects are to be classified against a fixed training set, as is the case in such problems as speech recognition, elementary particle identification, and related pattern recognition problems [Tou (76)], we must be able to perform nearest-neighbor searching quickly.

In one dimension, nearest-neighbor searching is easily seen to be ordinary binary search, and vice-versa. Given N points x_i, we sort them and arrange them in a vector. To find the nearest neighbor of a new point *p*, we perform a binary search, learning, for example, that *p* lies between x_i and x_{i+1}. Whichever of these two points is closer to *p* is its nearest neighbor. Likewise, once we have learned that the nearest neighbor of p is x_i, then we are certain that p lies either between x_{i-1} and x_i or between x_i and x_{i+1}. These cases can be distinguished in a single comparison, and we have performed a binary search. As usual, there is no

immediate generalization to two dimensions because there is no metrically-induced total order on points in the plane, and binary search cannot be applied directly. (For a technique that involves the construction of new search objects, see [Dobkin (76a)].) Previous algorithms based on projections have been observed to run in expected time $O(N^{1/2})$ on test data, but have linear worst-case search time [Friedman (75)]. The reader should consider how apparently unstructured is the configuration of points in Figure 6.2. One of our goals will be to find a structure on these points that will facilitate rapid search.

Since we have seen that in one dimension,

$$\text{BINARY SEARCH} \propto \text{NEAREST NEIGHBOR},$$

we have by the standard information-theoretic argument [Borodin (73)]:

Theorem 6.2: $\Omega(\log N)$ comparisons are necessary to find the nearest neighbor of a point (in the worst case) in any dimension.

In Section 6.5, we give an algorithm that achieves this lower bound for all dimensions. In a model which allows comparisons only between linear functions of the inputs, if we assume that p is equally likely to fall in any of the N+1 intervals determined by the x_i, then Theorem 6.2 bounds the *expected* behavior of any nearest-neighbor search algorithm. This is not true for models of computation which allow non-analytic functions, such as FLOOR, which can be used to compute indices for hashing or direct array access. If FLOOR is allowed and the distribution from which the points are drawn is known and continuous, then interpolation search [Yao (76)] can be used to find nearest neighbors in $O(\log \log N)$ expected time. However, no generalization to two or more dimensions is apparent.

Problem P6.4: (K-Nearest Neighbors) Given N points in the plane, with preconditioning allowed, how quickly can the k points nearest to a new given point be found?

The k nearest neighbors have been used for interpolation and contouring [Davis (75)] and for classification [Andrews (72b)] (the k-nearest neighbor rule is more robust than just looking at a single neighbor). Though the problem seems to be more

difficult that P6.3, the structure we shall develop in Section 6.3 is capable of solving it.

6.1.4 Euclidean Minimum Spanning Tree

Problem P6.5: (Euclidean Minimum Spanning Tree) Given N points in the plane, construct a tree of minimum total length whose vertices are the given points.

By a solution to this problem we will mean a list of N-1 pairs of points comprising the edges of the tree.[3] Such a tree is shown in Figure 6.4.

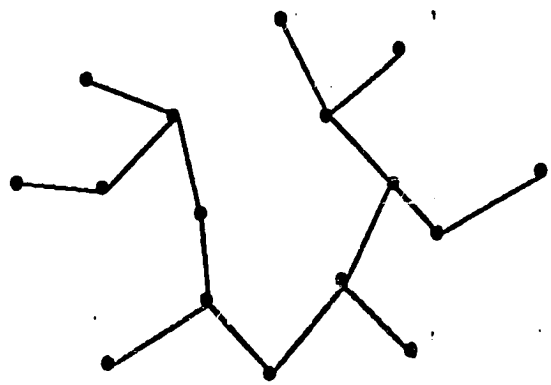


Figure 6.4: A Minimum Spanning Tree on a Plane Set

The EMST problem is a common component in applications involving networks. If one desires to set up a communications system among N nodes requiring interconnecting cables, using the EMST will result in a network of minimum cost [Prim (57)]. A curious facet of Federal law lends added importance to the problem. When the Long Lines Department of the Bell System establishes a communications hookup for a customer, federal tariffs require that the billing rate be proportional to the length of a minimum spanning tree connecting the customer's termini, the distance

[3] A tree on N vertices must have exactly N-1 edges [Harary (71)].

to be measured on a standard *flat* projection of the Earth's surface. This is true regardless of the fact that the Earth is not flat and the Bell System may not choose to set up the actual network as an MST. Nonetheless, the billing is based on a real Euclidean problem, one which must be solved hundreds of times daily, as network configurations are constantly changing.[4]

This law is a Solomon-like compromise between what is desirable and what is practical to compute, for the minimum spanning tree is *not* the shortest possible interconnecting network if new vertices may be added to the original set. With this restriction lifted, the shortest tree is called a *Steiner Tree* (Figure 6.5).[5]

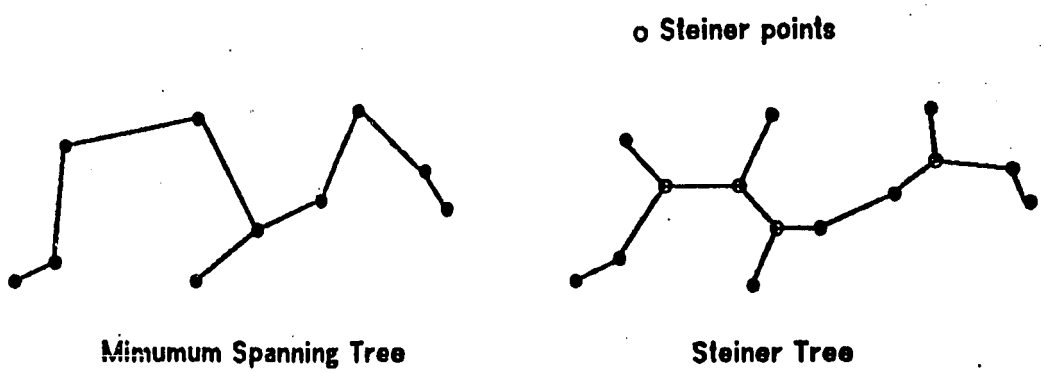


Figure 6.5: A Steiner Tree May Be Shorter Than the MST.

The computation of Steiner trees has been shown by Garey, Graham, and Johnson to be NP-complete [Garey (76a)], and we are unable with present technology to solve problems with more than about 15 points [Boyce (75)]. It is therefore unreasonable for the FCC to require billing to be by Steiner tree. This situation is somewhat reminiscent of the Indiana Bill that, for expediency, fixed the value of pi to be

[4]Thanks go to Stefan Burr for providing this information.

[5]The Steiner tree in Figure 6.5 is taken from [Melzak (73)].

exactly 4.[6]

The minimum spanning tree has been used as a tool in clustering [Gower (69)], [Johnson (67)], [Zahn (71)], determining the intrinsic dimension of point sets [Schwartzmann (75)], and in pattern recognition [Osteen (74)], as well as in minimizing wire length in computer circuitry [Lobermann (57)] and in obtaining approximate solutions to the Traveling Salesman Problem (Section 6.1.5).

The Euclidean Minimum Spanning Tree problem is usually formulated as a problem in graph theory: Given a graph with N nodes and E weighted edges, find the shortest subtree of G that includes every vertex. This problem was solved independently by [Dijkstra (59)], [Kruskal (56)], and [Prim (57)], and the existence of a polynomial-time algorithm (which they all demonstrated) is a considerable surprise, because a graph on N vertices may contain as many as N^{N-2} spanning subtrees[7]. A great deal of work has been done in an attempt to find a fast algorithm for this general problem [Nijenhuis (75)], [Yao (75)], and the best result to date is that O(E) time suffices if $E > N^{1+\epsilon}$, for any $\epsilon > 0$ [Cheriton (76)].

In the Euclidean problem, the N vertices are defined by 2N coordinates of points in the plane, and the associated graph has an edge joining every pair of vertices. The weight of an edge is the distance between its endpoints. Using the best known MST algorithm for this problem will thus require $O(E) = O(N^2)$ time, and it is easy to prove that this is a *lower* bound in an arbitrary graph because the MST always contains a shortest edge of G.[8] Since the edge weights in a general graph are unrestricted, an MST algorithm that ran in less than $O(N^2)$ time could be used to find the minimum of N^2 quantities in less than $O(N^2)$ time, which is impossible. It follows that any algorithm that treats a Euclidean MST problem as being embedded in the complete graph on N vertices is doomed to take quadratic time.

---

[6]"A bill for an act introducing a new mathematical truth", House Bill 246, Legislature of the State of Indiana, 1897.

[7][Moon (67)]. This was first proved by Cayley in 1889.

[8]This was shown in [Kruskal (56)] and [Prim (57)].

What would then lead us to suspect that less time is sufficient? For one thing, the Euclidean problem only has 2N inputs (the coordinates of the points), while the graph problem has N(N-1)/2 inputs (the edge lengths). The Euclidean problem is therefore highly constrained, and we may be able to use its metric properties to give a fast algorithm.

Even if improvement is possible, how fast an algorithm can we expect to obtain? Since the closest-pair problem is linear-time reducible to EMST, Theorem 6.1 implies an $\Omega(N \log N)$ lower bound, but in a restricted model of computation. We can strengthen this result by showing that

$$\text{SORTING} \propto_N \text{EMST} .$$

Consider a set of N points x_i in one dimension. This set possesses a unique EMST, namely, there is an edge from x_i to x_j iff they are consecutive in sorted order. A solution to the EMST problem consists of a list of N-1 pairs (I,J), giving the edges of the tree. We now show how to process this list in O(N) additional time to produce a sorted list of the x_i.

Algorithm A6.1: Linear-Time Sort from Consecutive Pairs

```
        Input   N real numbers X[1:N] and N-1 edges specified
                by two vectors, FROM[1:N-1] and TO[1:N-1].
                It is assumed that X has a unique least element.
        Output  A permutation vector PERM[1:N] such that X[PERM[I]]
                is the I'th smallest value.

   BEGIN
FOR I ← 1 UNTIL N DO SUCC[I] ← 0;
FOR I ← 1 UNTIL N-1 DO SUCC[FROM[I]] ← TO[I];
      Now find LOWEST, the unique point with no predecessor.
LOWEST ← 1;
FOR I ← 2 UNTIL N DO
   IF X[I] < X[LOWEST] THEN LOWEST ← I;
PERM[1] ← LOWEST;
FOR I ← 2 UNTIL N DO PERM[I] ← SUCC[PERM[I-1]];
   END
```

This reducibility, together with the sorting lower bound of Section 2.3, proves

Theorem 6.3: Construction of a minimum spanning tree on N points in any dimension requires $\Omega(N \log N)$ comparisons, even if analytic functions of the inputs may be computed.

6.1.5 Euclidean Traveling Salesman Problem

Problem P6.6: (Euclidean Traveling Salesman) Find a shortest closed path through N given points in the plane.

A shortest tour is shown in Figure 6.6.[9]

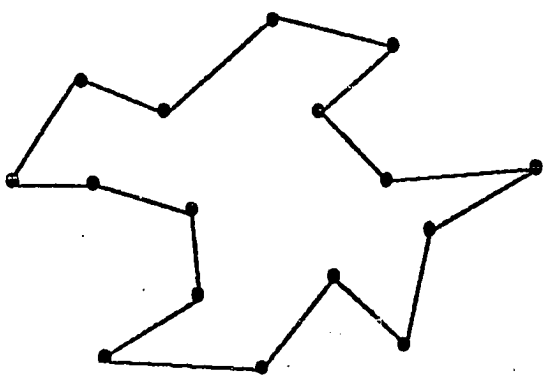


Figure 6.6: A Traveling Salesman Tour

This problem differs from the ordinary traveling salesman problem in the same way that Euclidean minimum spanning tree differs from the MST problem in graphs: The

[9] In fact, the tour was obtained by applying the Christofides heuristic (described below) several times and selecting the best result. It is not known to be a shortest tour for the given set of points.

interpoint distances are not arbitrary, but are inherited from the Euclidean metric. The general TSP is NP-complete.[10] Until recently, no geometry problems were known to be NP-complete, and the possibility existed that properties of the Euclidean metric could be used to produce a polynomial-time algorithm in the plane. Motivated partially by this observation, Garey, Graham, and Johnson [Garey (76)] undertook to prove the NP-completeness of a number of metric problems, with great success, for they succeeded in showing that ETSP is NP-complete, a result that was also proven independently in [Papadimitriou (76)]. We will therefore not attempt an efficient worst-case ETSP algorithm, but will concentrate on the relationship between ETSP and other closest-point problems, with a view toward developing good approximate or probabilistic methods.

Theorem 6.4: A minimum spanning tree can be used to obtain an approximate TSP tour whose length is not more than twice the length of a shortest tour.

Proof: Let MST be the length of a minimum spanning tree, and let TSP be the length of an optimal tour. We show that 2 MST < 2 TSP, which means that, by traversing each edge of the MST twice, we can visit each vertex and return to the starting point without traveling as much as twice the necessary distance. (See Figure 6.7.) Consider removing the longest edge in a TSP tour. The result is a chain through the N points which is a spanning tree, so its length cannot be less than MST. Thus we have

$$\text{MST} \leq (\text{TSP} - E) < \text{TSP} .$$

Multiplying both sides by two gives the necessary inequality.[11]

The next approximate result makes use of a *minimum weighted matching* on a set of points.

---

[10]For definitions relating to NP-complete problems and a proof of the NP-completeness of the TSP, see [Karp (75)].

[11]This result is well-known, and seems to have been discovered independently by many authors (R. M. Karp, private communication).

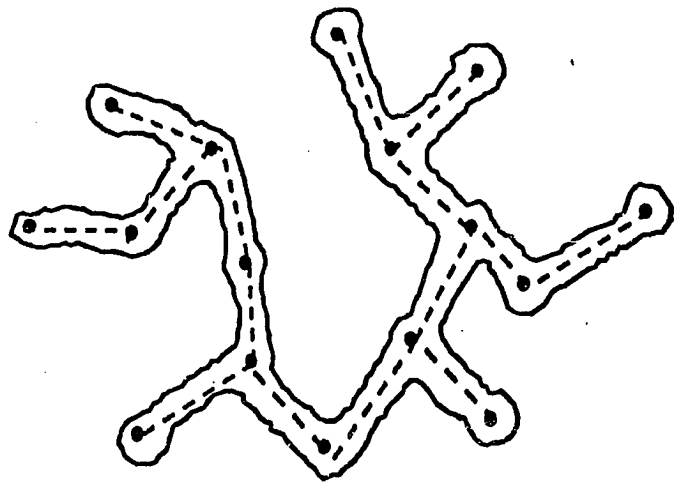


Figure 6.7: An Approximate Traveling Salesman Tour

Problem P6.7: (Minimum Euclidean Matching) Given 2N points in the plane, join them in pairs by line segments whose total length is a minimum.

Such a matching is shown in Figure 6.8.

Edmonds has shown that a minimum weight matching in an arbitrary graph can be obtained in polynomial time [Edmonds (65)], and an $O(N^3)$ implementation is given in [Gabow (72)]. The following result relates minimum spanning trees, matchings, and the traveling salesman problem.

Theorem 6.5: [Christofides (76)] An approximation to the traveling salesman problem whose length is within 3/2 of optimal can be obtained in $O(N^3)$ time if the interpoint distances obey the triangle inequality.

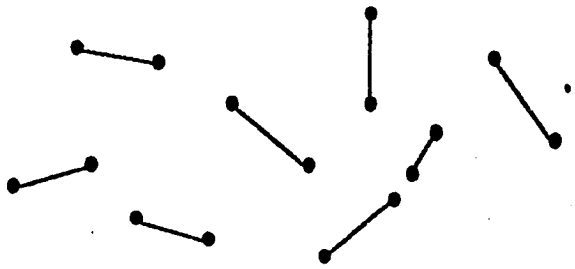


Figure 6.8: A Minimum Euclidean Matching

Proof: [Christofides (76)] The following algorithm achieves the desired result:

1. Find a minimum spanning tree.

2. Find a minimum Euclidean matching on the set of vertices of the MST that are of *odd* degree. (There are always an even number of such vertices in any graph [Harary (71)].) The total length of the edges in this matching is not greater than half the length of the MST.

3. To obtain a TSP tour, traverse the MST, but instead of repeating MST edges, use edges of the matching. This results in a tour that is no longer than 1.5 MST < 1.5 TSP.

In the above algorithm, which is a truly exquisite one, Christofides has used fast procedures (MST and matching) to attack the inherently difficult Traveling Salesman problem.

6.1.6 Triangulation

Problem P6.8: (Triangulation) Given N points in the plane, join them by non-intersecting straight line segments so that every region interior to the convex hull is a triangle.

Being a planar graph, a triangulation on N vertices has at most 3N-6 edges [Harary (71)]. A solution to the problem consists of a list of these edges. A triangulation is shown in Figure 6.9.

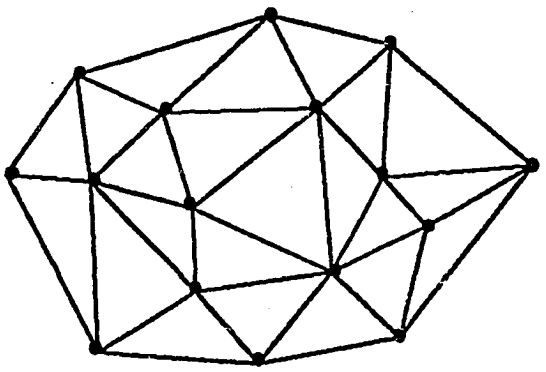


Figure 6.9: Triangulation of a Point Set

This problem arises in the finite element method [Strang (73)], and in numerical interpolation of bivariate data when function values are available at N irregularly-spaced data points (x_i, y_i) and an approximation to the function at a new point (x, y) is desired. In piecewise linear interpolation, the function surface is represented by a network of planar triangular facets.Each point (x, y) lies in a unique facet, and the function value $f(x, y)$ is obtained by interpolating a plane through the three facet vertices. Triangulation is the process of selecting triples that will define the facets.

Many criteria have beeen proposed as to what constitutes a "good" triangulation for numerical purposes [George (71)], some of which involve minimizing the largest angle or minimizing the length of the largest side. Later in this chapter we propose a new method of triangulation based on proximity of points and show that it can be found as rapidly as *any* triangulation on N points. [Lawson (77)] has recently shown that it is equivalent to a contouring procedure due to [McLain (76)], although our method is considerably faster computationally. The method itself is postponed until Section 6.5, when we shall have the tools necessary to present it. Meanwhile, we will content ourselves with another lower bound:

Theorem 6.6: $\Omega(N \log N)$ comparisons are necessary to triangulate N points in the plane, even if analytic functions may be computed.

Proof: We show that SORTING $\propto_N$ TRIANGULATION. Consider the set of N points x_i pictured in Figure 6.10, which consists of N-1 collinear points and another not on the same line. This set possesses only one triangulation, the one shown in the figure. The edge list produced by a triangulation can be used to sort the x_i in O(N) additional operations as in the proof of Theorem 6.3, so $\Omega(N \log N)$ comparisons must have been made.

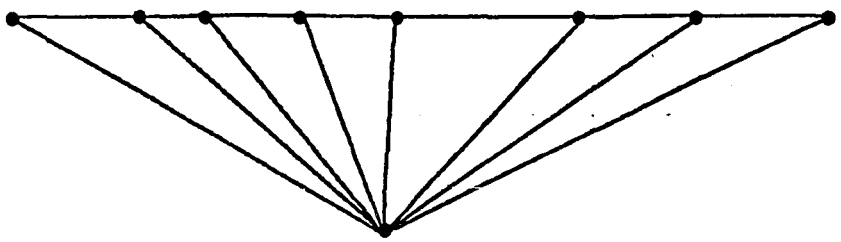


Figure 6.10: Triangulation Lower Bound

6.1.7 Smallest Enclosing Circle

Problem P6.9: (Smallest Enclosing Circle) Given N points in the plane, find the smallest circle that encloses them.

This is a classical problem with an immense literature, the search for an efficient algorithm having apparently begun in 1869 [Sylvester (69)]. The smallest enclosing circle is unique and is either the circumcircle of some three points of the set or defined by two of them as a diameter [Rademacher (57), Chapter 16]. Thus there exists a finite algorithm which examines all pairs and triples of points, and chooses the smallest circle determined by them which still encloses the set. The obvious implementation of this procedure would run in $O(N^4)$ time. This rote method has been improved by Elzinga and Hearn [Elzinga (72a)] [Elzinga (72b)] to run in $O(N^2)$ time and is the best algorithm to date [Francis (74)].

The enclosing circle problem is familiar in Operations Research as a *minimax facilities location problem* in which we seek a point p (the center of the circle) whose greatest distance to any point of the set is a minimum. We may characterize p by

$$\min_{p} \max_{i} \quad (x_i - p_x)^2 + (y_i - p_y)^2 \ . \qquad (6.1)$$

The minimax criterion is used in siting emergency facilities, such as police stations and hospitals, to minimize worst-case response time [Toregas (71)]. It has also been used to optimize the location of a radio transmitter serving N discrete receivers so as to minimize the RF power required [Nair (71)]. Minimization in other metrics is treated in [Shamos (75b)].

A generalization of the smallest enclosing circle brings us into the domain of coverage problems:

Problem P6.10: (k Police Stations) Given N points in the plane, locate *k* circles such that each point is in some circle and the maximum radius of any circle is a minimum.

A solution for $k = 3$ is shown in Figure 6.11. This is again a discrete problem, of which the smallest enclosing circle is the special case $k = 1$.

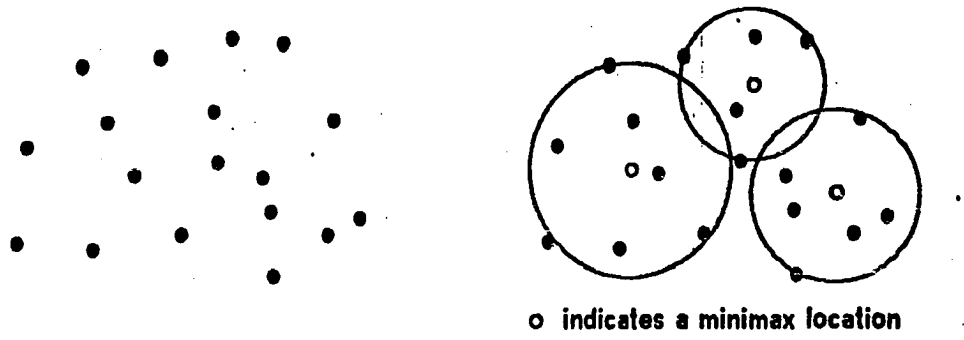


Figure 6.11: Minimax Location of Three Police Stations

A different generalization is:

Problem P6.11: (Smallest Bomb) Given N points in the plane, find the smallest circle that encloses at least k of them.

Another, more belligerent, statement of the problem is: Given N targets of equal strategic importance, what is the smallest bomb that will destroy at least k of them, and where should it be deployed? Or, where should a radio transmitter be set up so that it reaches at least k receivers and uses as little power as possible? Certainly $O(N^4)$ time suffices for all values of k simultaneously, for it is only necessary to construct the circles determined by all subsets of two or three points to determine how many points they contain. There are $O(N^3)$ such circles.

Another set of coverage problems arises if we fix the radius r and ask for the least number of circles of radius r that suffice to cover a finite set. See [Shamos (75b)].

There is good reason to believe that many of these problems are NP-complete since boolean satisfiability is reducible to certain coverage problems, even in the

Euclidean metric. In fact, we know of no case in which properties of the metric can be used to make a problem which is NP-complete without the metric run in polynomial time. (If the problem is already polynomial time-bounded, we have seen numerous examples in which metric properties serve to reduce the running time.)

6.1.8 Largest Empty Circle

Problem P6.12: (Largest Empty Circle) Given N points in the plane, find a largest circle that contains no points of the set yet whose center is interior to the convex hull.

The restriction on the center is necessary, for otherwise the problem would be unconstrained and would not possess a bounded solution. This problem is dual to P6.9 in that it is *maximin*. In other words, we want p as defined by

$$\max_{p \in Hull(S)} \min_{i} \quad (x_i - p_x)^2 + (y_i - p_y)^2 \ . \tag{6.2}$$

The point p is not necessarily unique. Figure 6.12 illustrates a solution.

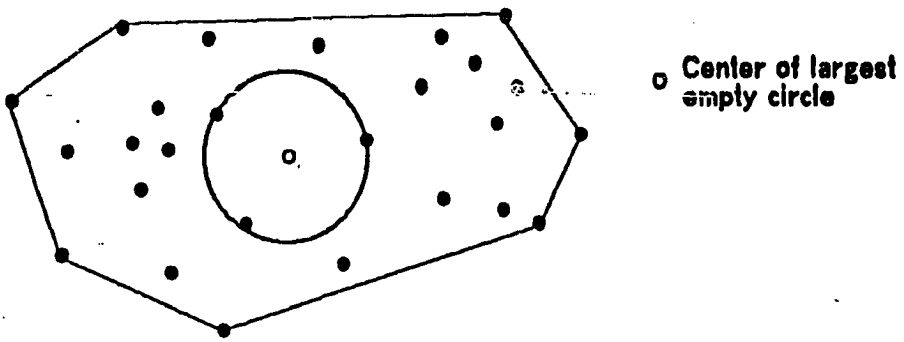


Figure 6.12: A Largest Empty Circle Whose Center is Interior to the Hull.

This is another facilities location problem, but one in which we would like to

position a new facility so that it is as far as possible from any of N existing ones. The new site may be a source of pollution which should be placed as to minimize its effect on the nearest residential neighborhood, or it may be a new business that does not wish to compete for territory with established outlets. Such problems arise frequently in industrial engineering [Francis (74)]. For the present problem, an algorithm has been given [Dasarathy (75)] whose worst-case running time is $O(N^3)$. [12]

In one dimension the problem reduces to finding a pair of consecutive points that are farthest apart, since a "circle" in one dimension is just a line segment. To show a lower bound, we introduce a new problem called GRID.

Problem P6.13: (Grid) Given N real numbers x_i, are they in fact the set of one-dimensional grid points $\{i/N\}$, $1 \le i \le N$? [13]

It is a straighforward to show by the methods of [Reingold (72a)] that the height of any linear tree program which decides GRID must be $\Omega(N \log N)$. This leads to

Theorem 6.7: If comparisons are allowed only between linear functions of the inputs, then $\Omega(N \log N)$ comparisons are required to find the two farthest adjacent points of N points on a line.

Proof: We show that GRID $\propto_N$ FARTHEST ADJACENT POINTS. Given N numbers x_i, determine whether they all lie in the interval [1/N,1]. If so, consider them as points on the line and find the farthest adjacent ones. The answer to GRID is affirmative iff the distance between them is exactly 1/N.

---

[12] A tantalizing problem. Why should we be able to do better for its dual, the smallest enclosing circle? Is convexity helping again? No. We shall see later that the contrast between $O(N^2)$ and $O(N^3)$ only reflects our poor state of knowledge about *both* problems, and O(N log N) time suffices in each case.

[13] This problem and the theorem follwing it were generously contributed by Jon Bentley.

Unexpectedly, [Gonzalez (75)] has demonstrated convincingly the weakness of this lower bound by giving a linear-time algorithm! His method, of course, uses non-linear functions (actually, non-analytic ones), so there is no contradiction, but his algorithm is a gem:

Algorithm A6.2: Max Gap [Gonzalez (75)]

```
Input:   N real numbers X[1:N]  (unsorted).
Output:  MAXGAP, the length of the largest gap between
         consecutive numbers in sorted order.
```

1. Find the MAX and MIN values of X. This can be done in O(N) time. Swap so that X[1] contains MIN and X[N] contains MAX.

2. Create N-1 buckets by dividing the interval from MIN to MAX with N-2 equally-spaced points. In each bucket we will retain HIGH[I] and LOW[I], the largest and smallest values in bucket I.

```
FOR I←2 UNTIL N-2 DO COUNT[I] ← 0;
COUNT[1] ← 1;  COUNT[N-1] ← 1;
```

3. (Hash into buckets)

```
FOR i←2 UNTIL N-1 DO BEGIN
    BUCKET ← 1 + (N-1) * FLOOR( (X[I]-MIN) / (MAX-MIN) )
    COUNT[BUCKET] ← COUNT[BUCKET] + 1;
    IF COUNT[BUCKET] = 1 THEN BEGIN
        LOW[BUCKET] ← X[I];  HIGH[BUCKET] ← X[I];  END
    ELSE BEGIN
        IF X[I] < LOW[BUCKET] THEN LOW[BUCKET] ← X[I];
        IF X[I] > HIGH[BUCKET] THEN HIGH[BUCKET] ← X[I];
    END
END
```

4. Since N-2 points have been placed in N-1 buckets, by the pigeonhole principle some bucket must be empty. This means that the largest gap cannot occur between two points in the same bucket, which justifies our keeping only HIGH and LOW. We now make a single pass through the buckets, keeping the largest gap between each HIGH and the next LOW in a nonempty bucket.

```
MAXGAP ← 0;
LEFT ← HIGH[1];
FOR I ← 2 UNTIL N-1 DO BEGIN
    IF COUNT[I] ≠ 0 THEN BEGIN
        THISGAP ← LOW[I] - LEFT;
        IF THISGAP > MAXGAP THEN MAXGAP ← THISGAP;
        LEFT ← HIGH[I];
    END
END
```

In view of the similarity between MAXGAP and CLOSEST PAIR in one dimension, it is remarkable that a linear algorithm is possible. Unfortunately, no generalization to two dimensions seems to be possible.

6.2. A Divide-and-Conquer Algorithm for Closest Pair

The lower bound of Theorem 6.1 challenges us to find an O(N log N) algorithm for closest pair.[14] There seem to be two reasonable ways to achieve such behavior: sorting and divide-and-conquer. The sorting approach does not appear to lead to anything useful, for the only way to apply it seems to be to project all the points onto one of the axes. One would like to appeal to the principle ("hope" is a better word) that in this projection very few points will separate A and its nearest neighbor. In the worst case, of course, this is not true, as illustrated in Figure 6.13. Points A and B are closest, but they are farthest when projected on the y-axis.

[14]This section is based almost entirely on an idea due to H. R. Strong, for which we are most appreciative.

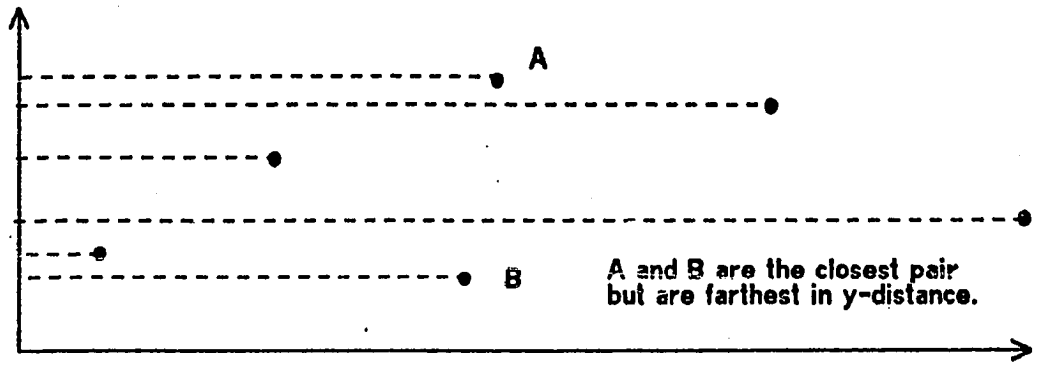


Figure 6.13: The Failure of Projection Methods

A second way to achieve O(N log N) performance is to split the problem into two subproblems whose solutions can be combined in linear time to give a solution to the entire problem. In this case, the obvious way of applying D&C does not lead to any improvement, and it is instructive to explore why it fails.

We would like to split the set into two subsets, A and B, each having N/2 points, and obtain a closest pair in each set recursively. The problem is how to make use of the information so obtained. The possibility still exists that the closest pair in the set consists of one element of A and one element of B, so that there is no clear way to avoid making $N^2/4$ additional comparisons. This leads to a recurrence of the form

$$T(N) = 2T(N/2) + O(N^2) \quad ,$$

whose solution is $T(N) = O(N^2)$. Let us try to remedy the difficulty by retreating to one dimension.

The only O(N log N) algorithm we know on the line is the one which sorts the points and performs a linear-time scan (Section 6.1.1). Since sorting will not generalize to two dimensions, let us try to develop a one-dimensional divide-and-

conquer scheme that will. Suppose we partition a set of points on the line by their median M into two sets A and B with the property that $a < b$ for all $a \in A$ and $b \in B$. Solving the closest pair problem recursively on A and B separately gives us two pairs of points, {c,d} and {e,f}, the closest pairs in A and B, respectively. Let δ be the smallest separation found thus far:

$$\delta = \min (d-c, f-e) .$$

(See Figure 6.14.)

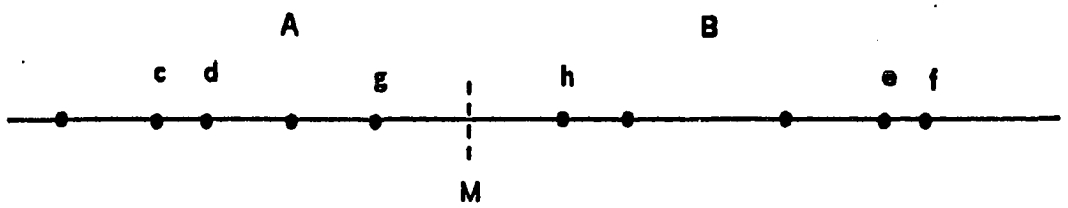


Figure 6.14: Divide-and-Conquer in One Dimension

The closest pair in the whole set is either {c,d}, {e,f}, or some {g,h}, where $g \in A$ and $h \in B$. Notice, though, and this is the key observation, that both g and h must be within distance δ of M if (g,h) is to have a separation smaller than δ.[15] How many points of A can lie in the half-open interval (g,M]? Since every half-open interval of length δ contains at most one point of A, (g,M] contains at most one point. Similarly, [M,h) contains at most one point. The number of pairwise comparisons that must be made between points in different subsets is thus at most one. We can certainly find all points in the intervals (g,M) and (M,h) in linear time, so an O(N log N) algorithm results.

[15] It is clear that g must be the rightmost point in A and h the leftmost point in B, but this notion is not meaningful in higher dimensions so we wish to be somewhat more general.

Algorithm A6.3: Closest Pair in One Dimension

```
        Input:   X[1:N] , N points in one dimension.
        Output:  DELTA , the distance between the two closest.

  RECURSIVE PROCEDURE CPAIR(X,DELTA);
     IF  |X| = 2  THEN BEGIN
         DELTA ← X[2] - X[1];  RETURN;  END
     ELSE
         IF |X| = 1 THEN BEGIN
             DELTA ← ∞ ;  RETURN;  END
     M ← MEDIAN(X);
         Let A be the points < M,
         and B the points ≥ M.
     CPAIR(A,ADELTA);
     CPAIR(B,BDELTA);
       Find the (unique) point G of A that is within ADELTA of M.
       Find the (unique) point H of B that is within BDELTA of M.
     DELTA ← MIN(ADELTA, BDELTA, H - G);
     RETURN;
  END CPAIR
```

This algorithm, while apparently more complicated than the simple sort and scan, provides the necessary transition to two dimensions.

Generalizing as directly as possible, let us partition a two-dimensional set S into two subsets A and B of equal size such that every point of A lies to the left of every point of B. That is, we cut the set by a vertical line L defined by the median x-coordinate of S. Solving the problem on A and B recursively, we obtain δ_A and δ_B, the minimum separations in A and B, respectively. Now let $\delta = \min(\delta_A, \delta_B)$. (See Figure 6.15.)

If the closest pair consists of some $g \in A$ and some $h \in B$, then surely g and h are both within distance δ of L. Thus, $g \in P$ and $h \in Q$. At this point complications arise that were not present in the one-dimensional case. On the line we found at most one candidate for g and at most one for h. In two dimensions every point can be a candidate because it is only necessary for a point to lie within distance δ of L.

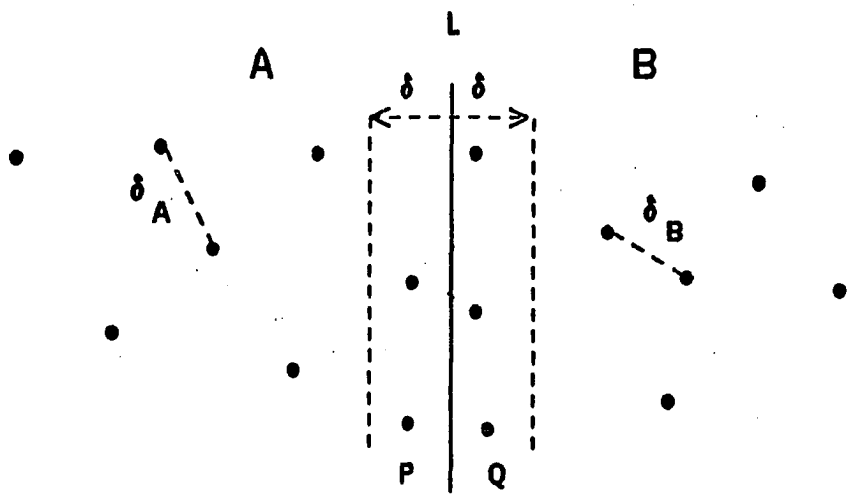


Figure 6.15: Divide-and-Conquer in the Plane

Figure 6.16 shows a set with this property. It again seems that $N^2/4$ distance comparisons will be required to find the closest pair, but we will now show that the points lying within the δ-slabs around L have special structure.

Referring to Figure 6.17, consider any point g in P. We must find all points h in Q that are within δ of g, but how many of these can there be? They must lie in the $\delta \times 2\delta$ rectangle R, and we know that no two points in R are closer together than δ.[16] The maximum number of points of separation at least δ that can be packed into such a rectangle is six, as shown in the figure. This follows from the fact that at most four points with separation at least δ can be packed in a square of side δ and means that, for each point of P, we need only examine six points of Q, not N/2 points. In other words, only $6 \times N/2 = 3N$ distance comparisons are needed in the subproblem merge step instead of $N^2/4$.

[16]The author is deeply grateful to H. R. Strong for this observation, which cleared the way for research on closest-point problems. He developed an $O(N \log^2 N)$ closest-pair algorithm, which we have improved here to $O(N \log N)$.

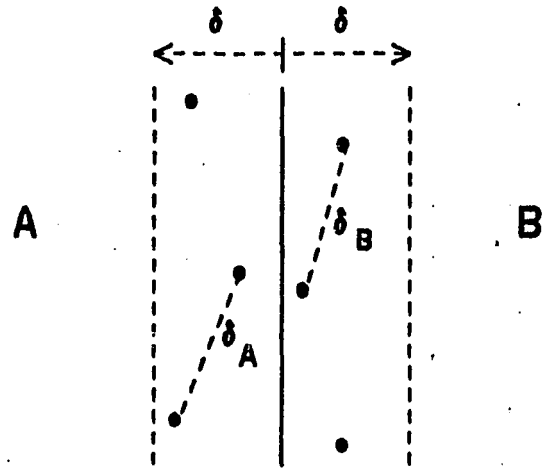


Figure 6.16: All Points May Lie Within δ of L

We do not yet have an O(N log N) algorithm, however, because even though we know that only six points of Q have to be examined for every point of P, we do not know which six they are!

Suppose we project g and all the points of Q onto L. To find the six points of Q that were in rectangle R, we need only look within an interval of twelve points that surround g in this projection. If we sort all the points initially by y-coordinate, this interval can be found for every $g \in P$ in a single pass through the sorted list. The merge step will then run in linear time. Here is the algorithm so far:

1. If $|S|=1$, set δ to ∞, and return. If $|S|=2$, set δ to the distance between the points. Otherwise, sort the points of S by both x- and y-coordinate.

2. Partition S into two subsets, A and B, about the median line M.

3. Find the closest pair separations δ_A and δ_B recursively.

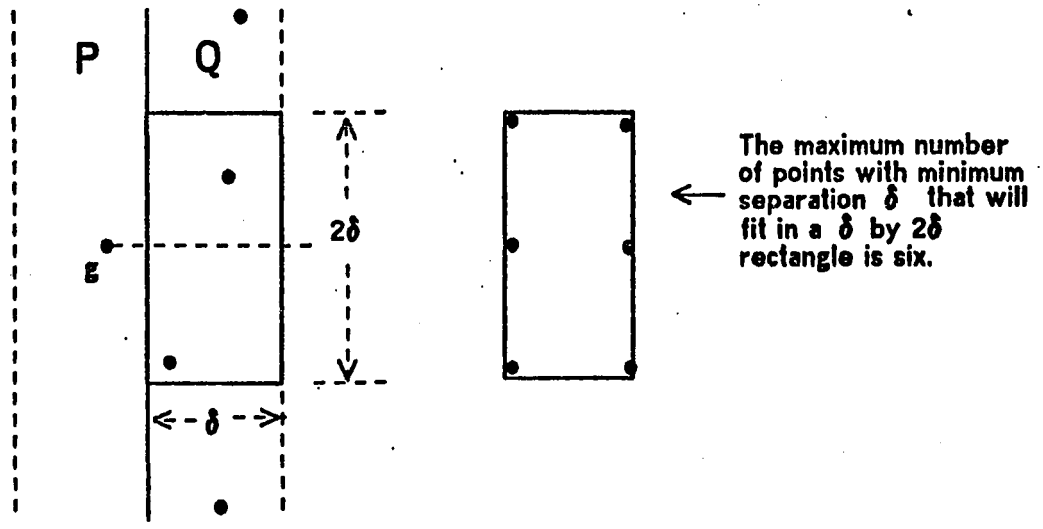


Figure 6.17: For every point in P, only a constant number in Q are examined.

4. $\delta \leftarrow \min(\delta_A, \delta_B)$;

5. Let P be the set of points of A that are within δ of the dividing line L, and let Q be the corresponding set of points of B. Project P and Q onto L.

6. For every point of P, find its six nearest neighbors in the L projection of Q. Let δ_L be the shortest distance between any pair thus obtained.

7. $\delta_S \leftarrow \min(\delta_A, \delta_B, \delta_L)$.

Theorem 6.8: The shortest distance determined by N points in the plane can be found in O(N log N) time, and this is optimal.

Proof: We show that the above algorithm always finds a closest pair. If $|S| = 2$, the algorithm works correctly; we now argue inductively based on the partitioning performed in step 2. Assume the closest pair to be unique so there are three cases to consider. Either both points lie in A, both in B, or one in each. In the first two cases the algorithm works correctly by the inductive hypothesis. In the third case, by the above discussion the algorithm non-recursively find the closest pair of the set with the property that one point lies in A and the other in B. The running time can be computed as follows: Step 1 requires $O(N \log N)$ time, but is only performed once. Steps 2 and 6 take $O(N)$ time, Steps 4 and 7 take constant time, Step 3 takes $2T(N/2)$ time, and Step 5 takes $O(N \log N)$ time. The total running time is described by the recurrence

$$T(N) = 2T(N/2) + O(N) = O(N \log N) .$$

Optimality was shown in Theorem 6.1.

With Dan Hoey, we implemented this algorithm and observed a curious phenomenon: the number of distance computations made was always strictly less than N. That this is always the case was later proved in [Bentley (76a)]. Of course, the behavior of the algorithm is still dominated by the sort step. The structure of this divide-and-conquer scheme has some noteworthy features:

1. The method of division forces the subproblems to have a special property, namely, *sparsity*.

2. The step at which the subproblem solutions are merged takes place in one lower dimension.

3. The merge time is reduced by preprocessing which takes place outside the recursive structure of the algorithm.

Together with Bentley, we have shown [Bentley (76b)] that a straightforward generalization of this algorithm to k dimensions runs in $O(N \log^{k-1} N)$ time, but that this can be reduced to $O(N \log N)$ by imposing constraints on the choice of cut-

planes. Bentley has made a thorough study of multidimensional divide-and-conquer algorithms in his Ph.D. thesis [Bentley (76a)], and has formulated a number of heuristic design principles for discovering new ones. The idea of recurring both on dimension and problem size simultaneously is an especially powerful one.

Theorem 6.1 is significant because it provides hope that all of the closest-point problems discussed above (except the ETSP) can be solved in O(N log N) time. We already knew that CLOSEST PAIR was reducible to P6.2-P6.6, so a quadratic lower bound for CLOSEST PAIR would imply a quadratic lower bound on all of the others. Now that we have a fast algorithm for P6.1, it is reasonable to suppose that the other problems can be solved quickly.

6.3. The Voronoi Diagram

It is one thing to suspect the existence of fast algorithms, but quite another to actually discover them. While our D&C algorithm for finding a closest pair may be encouraging, it does not solve even the all nearest neighbors problem, which would seem to be a simple extension. If we try to set up the analogous recursion for all nearest neighbors, we find that the natural method of splitting the problem does not induce sparsity, and there is no apparent way of accomplishing the merge step in less than quadratic time. To get around this difficulty, let us abandon divide-and-conquer temporarily and study some geometry.

A valuable heuristic for designing geometric algorithms is to look at the defining loci and try to organize them into a data structure. In the case of the closest-point problems we want to solve

Problem P6.14: (Loci of Proximity) Given N points in the plane, for each point p_i, what is the locus of points (x, y) in the plane that are closer to p_i than to any other point?

If we knew these loci, we would be able to solve the nearest neighbor problem directly, since determining the closest point to (x, y) is the same as asking which locus it lies in.

Given two points, p_i and p_j, the set of points closer to p_i than to p_j is just the half-plane containing p_i that is defined by the perpendicular bisector of p_i and p_j. Let us denote this half-plane by $H(p_i,p_j)$. The locus of points closer to p_i than to any other point, which we denote by V(i), is then intersection of such half-planes, a problem we studied in Chapter 5. Thus V(i) is a convex polygonal region having no more than N-1 sides, defined by

$$V(i) = \bigcap_{j \neq i} H(p_i, p_j) . \qquad (6.3)$$

V(i) is called the Voronoi polygon associated with p_i. (See Figure 6.18.) [17].

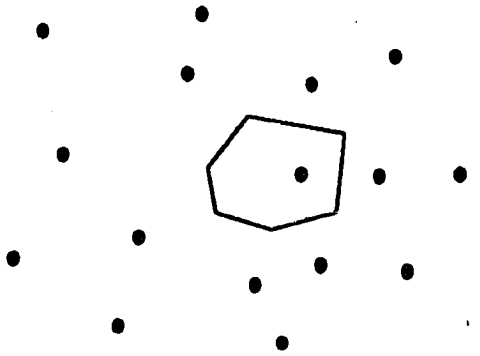


Figure 6.18: A Voronoi Polygon.

These N polygons partition the plane into a convex net which we shall refer to as the *Voronoi diagram.* See Figure 6.19.) The vertices of the diagram are *Voronoi points*, and its line segments are *Voronoi edges.*

Each of the original N points belongs to a unique Voronoi polygon. Thus if

[17] These polygons were first studied seriously by the emigre Russian mathematician G. Voronoi,. who used them in a treatise on quadratic forms [Voronoi (08)]. They are also called Dirichlet regions [Loeb (76)], mosaics [Matern (60)], or Thiessen polygons [Hodder (76)]. Dan Hoey has suggested the more descriptive (and impartial) term "proximal polygon".

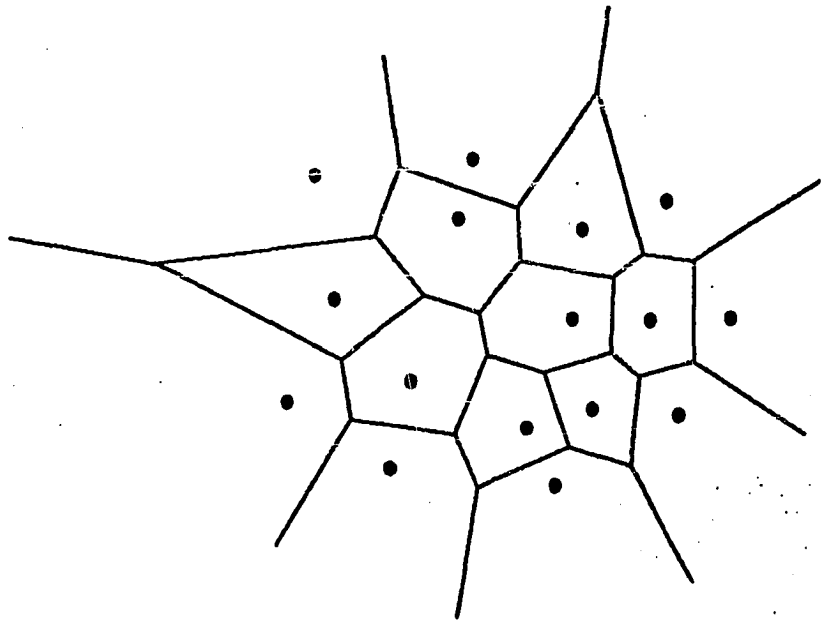


Figure 6.19: The Voronoi Diagram.

$(x,y) \in V(i)$, then p_i is a nearest neighbor of (x, y). The Voronoi diagram contains, in a powerful sense, all of the proximity information defined by the given set.

6.3.1 A Catalog of Voronoi Properties

In this section we list a number of important properties of the Voronoi diagram. We assume throughout that no four points of the original set are cocircular. If this is not true, inconsequential but lengthy details must be added to the proofs and statements of the theorems. Eventually, we will want to use the Voronoi diagram to solve the closest-point problems. This will only be successful if it can be constructed rapidly. A trivial lower bound on the time necessary to do this is the total number of Voronoi points and edges that are present. At first glance the diagram seems very complicated, but the number of elements it contains is small.

Every edge of the Voronoi diagram is a segment of the perpendicular bisector of a pair of points and is thus common to exactly two polygons.

Definition 6.1: The *straight-line dual* of a Voronoi diagram on a set S of N points is a graph whose N vertices correspond to the points of S, in which there is an edge from s to t iff the Voronoi polygons V(s) and V(t) share an edge. (See Figure 6.20).

As an abstract graph, the straight-line dual is the *geometric dual* of the Voronoi graph [Ore (62)].

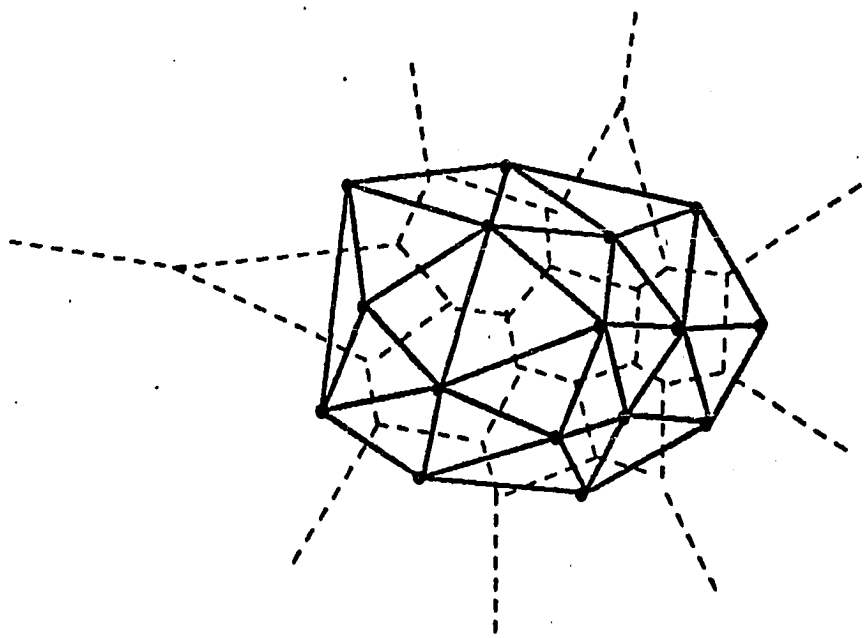


Figure 6.20: The Straight-Line Dual of the Voronoi Diagram.

The dual may appear to be unusual at first glance, since an edge and its dual may not even intersect (look at the edges joining consecutive vertices of the convex hull in Figure 6.20). It importance is due largely to the following theorem of

Delaunay:[18]

Theorem 6.9: The straight-line dual of the Voronoi diagram is a triangulation.[19]

This means that the Voronoi diagram can be used to solve the Triangulation problem, P6.8, but the theorem has a much more significant consequence:

Theorem 6.10: A Voronoi diagram on N points has at most 2N-4 vertices and 3N-6 edges.

Proof: Each edge in the straight-line dual corresponds to a unique Voronoi edge. Being a triangulation, the dual is a planar graph on N points, and thus has at most 3N-6 edges [Harary (71)]. Therefore, the number of Voronoi edges is at most 3N-6. To compute the number of Voronoi vertices, we observe that there is one such vertex for each *face* of the dual. The number of faces of any planar graph is given by Euler's relation: $F = E - V + 2$. For a given value of V, F will be maximized when E is maximized. Applying this to the dual graph, $V = N$ and $E \leq 3N - 6$, whence $F \leq 2N - 4$.

Since it is the dual of a planar graph, which we shall call the *Delaunay graph*, the Voronoi diagram is itself a planar graph [Harary (71)], and can be stored in only linear space. This makes possible an extremely compact representation of the proximity data. Any given Voronoi polygon may have as many as N-1 edges, but there are at most 3N-6 edges overall, each of which is shared by exactly two polygons. This means that the *average* number of edges in a Voronoi polygon does not exceed six.

---

[18] [Delaunay (34)]. A readable account appears in [Rogers (64)].

[19] In this simple form, the theorem fails when certain subsets of four or more points are cocircular. In this case, however, completing the triangulation will be straightforward. (Recall that if the points are chosen from an absolutely continuous probability distribution, the probability of a cocircularity is zero. A statistician would find it amusing that we are so concerned with events of probability zero.)

We will refer to the faces of the Delaunay graph as *Delaunay triangles* and, since the degree of a vertex t of the Voronoi diagram is equal to the number of edges of the face to which t corresponds in the dual, each vertex of the Voronoi diagram (except the vertex at infinity) is of degree three.

Theorem 6.11: Every nearest neighbor of p_i defines an edge of the Voronoi polygon V(i).

Proof: Let p_j be a nearest neighbor of p_i, and let M be the midpoint of p_ip_j. M cannot be interior to V(i) since it is equidistant from both p_i and p_j. Suppose that M does not lie on the boundary of V(i). Then the line segment p_iM intersects some edge of V(i), say the bisector of p_i and p_k, at N. (Figure 6.21.) Then $p_iN < p_iM$, so

$$p_ip_k = 2p_iN < 2p_iM = p_ip_j ,$$

and we would have p_k closer to p_i than p_j, which is impossible.

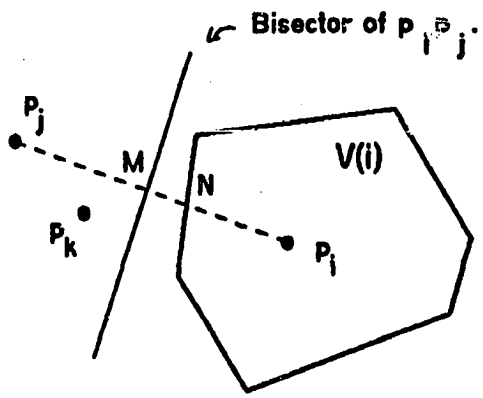


Figure 6.21: Every Nearest Neighbor of p_i Defines an Edge of V(i).

The *circumcenter* of a triangle is the center of the (unique) circle that passes

through the vertices of the triangle. The circumcenter is equidistant from these vertices. Since a Voronoi vertex is of degree three, it is equidistant from three of the original points and is in fact the circumcenter of a Delaunay triangle.

Theorem 6.12: The circumcircle of a Delaunay triangle contains no other points of the set.[20]

Proof: (See Figure 6.22.) Consider the triangle *abc* whose circumcenter is x. (Note that x is not necessarily interior to *abc*.) If the circumcircle *C* contains some other point *d*, then x is closer to *d* than to any of *a*, *b*, or *c*, in which case, by the definition of a Voronoi polygon, it must lie in V(*d*) and not in any of V(*a*), V(*b*), or V(*c*), by the definition of a Voronoi polygon. This is a contradiction, since x is common to all of V(*a*), V(*b*), and V(*c*).

Theorem 6.13: The polygon V(i) is unbounded iff p_i lies on the boundary of the convex hull.

Proof: This follows from the proof of the more general Theorem 6.26 and the fact that the only singleton exposed subsets (Cf. Definition 6.2 are hull vertices. Since only unbounded polygons can have rays as edges, the rays of the Voronoi diagram correspond to pairs of adjacent vertices on the convex hull.

In the next section we will use these properties to construct the Voronoi diagram quickly and employ it solve the closest-point problems. Even though we will be using it for other purposes, it is well to note that construction of Voronoi diagrams is an end in itself in a number of fields. In archaeology, Voronoi polygons are used to map the spread of the use of tools in ancient cultures and for studying the influence of rival centers of commerce [Hodder (76)]. In ecology, the survival of an organism depends on the number of neighbors it must compete with for food and

[20] This phenomenon was first observed by Dan Hoey.

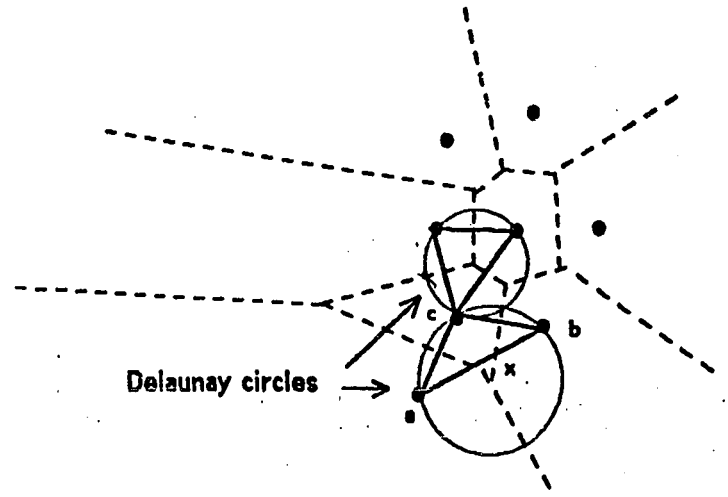


Figure 6.22: The Circumcircle of a Delaunay Triangle is Empty.

light, and the Voronoi diagram of forest species and territorial animals is used to investigate the effect of overcrowding [Pielou (77)]. The structure of a molecule is determined by the combined influence of electrical and short-range forces, which have been probed by constructing elaborate Voronoi diagrams [Brostow (77)].

6.4. Constructing the Voronoi Diagram

We will now see that even though the Voronoi diagram appears to be a complex object, it is eminently suited to attack by divide-and-conquer. The method we employ depends for its success on various structural properties of the diagram that enables us to merge subproblems in linear time.

By "finding" the Voronoi diagram of a set of points, we mean obtaining all of the following data:

1. The coordinates of the Voronoi points.

2. The Voronoi edges (a pair of Voronoi points) incident with each Voronoi point.

3. The two original points that determine each Voronoi edge.

4. A list of the edges of each polygon in cyclic order.

Since each Voronoi polygon is an intersection of N-1 half-planes, it can be constructed in O(N log N) time by Algorithm A5.2. (This is optimal for producing any *single* polygon.) There are N polygons to be formed, so the entire construction can be accomplished in $O(N^2 \log N)$ time. On the other hand,

Theorem 6.14: Constructing a Voronoi diagram on N points in the plane must take $\Omega(N \log N)$ operations, in the worst case.

Proof: We will see later that the closest-point problems are all linear-time reducible to VORONOI DIAGRAM so many proofs of this theorem are possible. We content ourselves here with a very simple one. The Voronoi diagram of a set of points in *one* dimension consists of N-1 bisectors separating adjacent points on the line. From these consecutive pairs, we can obtain a sorted list of the points in linear time by Algorithm A6.1.

We now show that this lower bound can be achieved, which means that constructing the entire diagram is no more difficult than finding a single one of its polygons!

Let us suppose that we have divided a set S, containing N points, into two subsets L and R by a vertical median line M. This means that every point in L lies to the left of every point in R, and every point of R lies to the right of every point in L.[21] Let us now find the Voronoi diagrams V(L) and V(R) of each subset

[21] Unless, of course, two or more points lie on the median line, in which case we assign the upper half to set L and the others to set R.

recursively. If these can be merged in linear time to form the Voronoi diagram V(S) of the entire set, we will have an O(N log N) algorithm. But what reason is there to believe that V(L) and V(R) bear any relation to V(S)?

Consider the locus P of points that are simultaneously closest to a point of L and a point of R. This is just the set of edges of V(S) that are shared between polygons V(I) and V(J), with $p_I \in L$ and $p_J \in R$. We will now show that the locus P is a polygonal line.

Lemma 6.1: Every horizontal line intersects P in at least one point.

Proof: Consider a horizontal line H and any pair of points $u \in L$ and $v \in R$ not both on the median line M. Let the perpendicular bisector of uv intersect H at z. This intersection exists because uv cannot be a vertical line. Denote by LEFT(H,u,v) the set of all points of H that lie to the left of z and by RIGHT(H,u,v) the points that lie to the right. Now examine the intersection LL of LEFT(H,u,v) over all pairs u,v that satisfy the conditions above. Each element of LEFT is a negative half-line, so the intersection is non-empty. Likewise RR, the intersection of the RIGHT(H,u,v) is nonempty. Since H is now known to contain some points that are closest to some point of L and some that are closest to a point of R, by continuity it must also contain at least one point that is equidistant from L and R and which thus belongs to P.

Theorem 6.15: The locus P intersects each horizontal line in at most one point. That is, P is monotonic in y.

Proof: P consists only of segments and rays since it is composed of lines of the Voronoi diagram. We show that in traversing P from $+\infty$ the y-coordinate never increases. For assume the contrary, and let *w* be the first point at which P turns upward. (See Figure 6.23.) Edge *uw* is the bisector of *a* and *b*, *vw* is the bisector of *b* and *c*. Since by construction P separates L and R, we have that *a* and *c* are in R, and *b* is in L, or *a* and *c* are in L and *b* in in R. In either case we have a contradiction. which is a

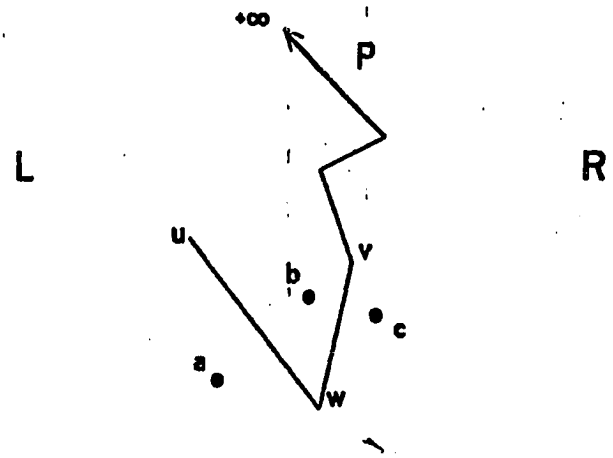


Figure 6.23: Proof That P is Monotonic.

We now know that P is a monotonic polygonal line, so it is meaningful to speak of a point being to the "right" or "left" of P. P has the property that any point to its left is closest to some point of L and any point to the right is closest to some point of R. (See Figure 6.24.)

Consider superimposing the Voronoi diagrams V(L) and V(R). (V(L) and V(R) are shown separately in Figures 6.25 and 6.26, and are superimposed in Figure 6.27.)
Those segments of V(R) that lie to the *left* of P play no role in discriminating proximity between points of L since they pertain only to R, and are thus absent from the final diagram V(S). Likewise, those segments of V(L) that lie to the right of P are also absent from V(S). Given P, the merge step is completed by removing these sets of edges and "stitching" P into the remnants of V(L) and V(R). Here is a rough sketch of the emerging algorithm:

1. Divide S into two subsets L and R by median x-coordinate. This can be done in O(N) time.

2. Find V(L) and V(R) recursively. Time: 2T(N/2).

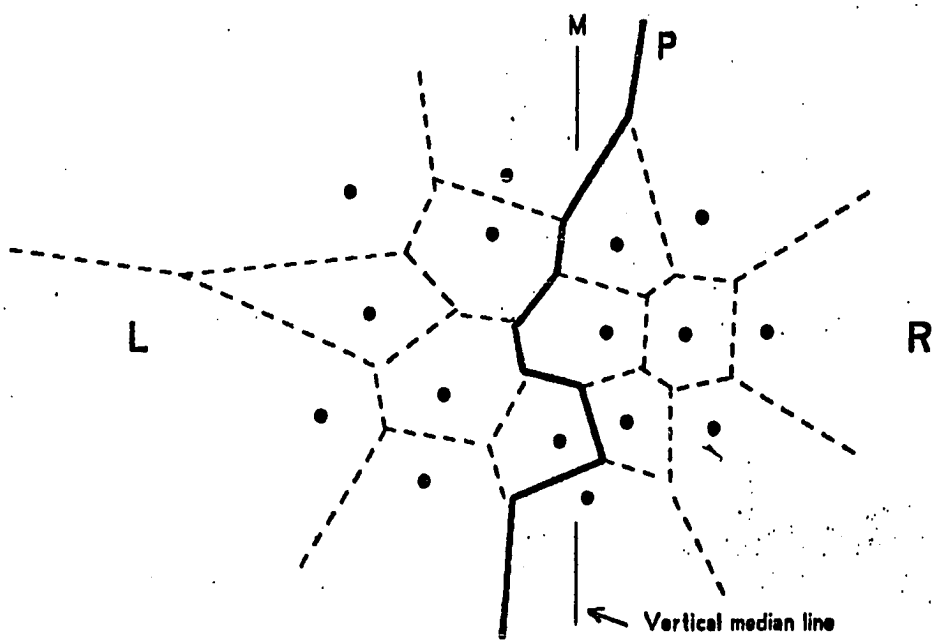


Figure 6.24: The Locus of Points Equidistant from L and R.

3. Construct P, the locus simultaneously closest to a point in L and a point in R.

4. Discard all segments of V(R) that lie to the left of P, and all segments of V(L) that lie to the right of P. The result is V(S), the Voronoi diagram of the entire set.

The success of this procedure depends on how rapidly we are able to find the dividing line P. We will use the monotonicity property of P to enable us to scan V(L) and V(R) downward, without backtracking. Since V(L) and V(R) each contain only O(N) edges, we will be able to find P in linear time.

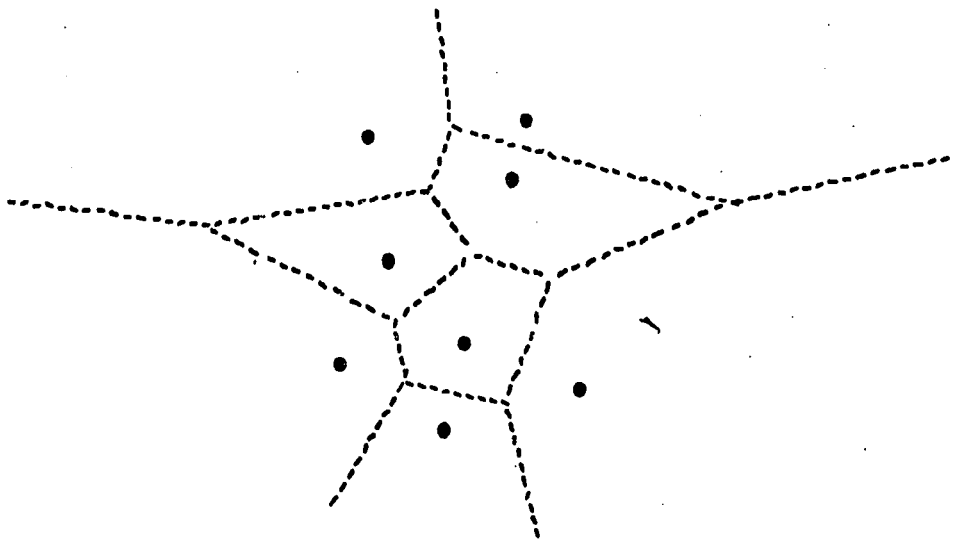


Figure 6.25: The Voronoi Diagram of the Left Set.

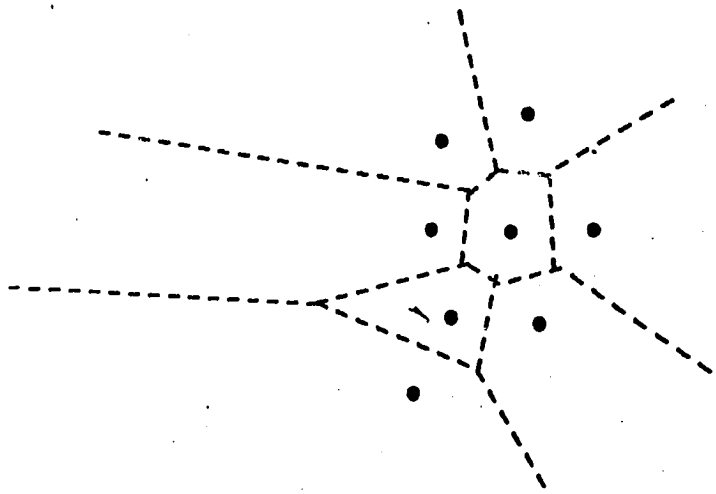


Figure 6.26: Voronoi Diagram of the Right Set.

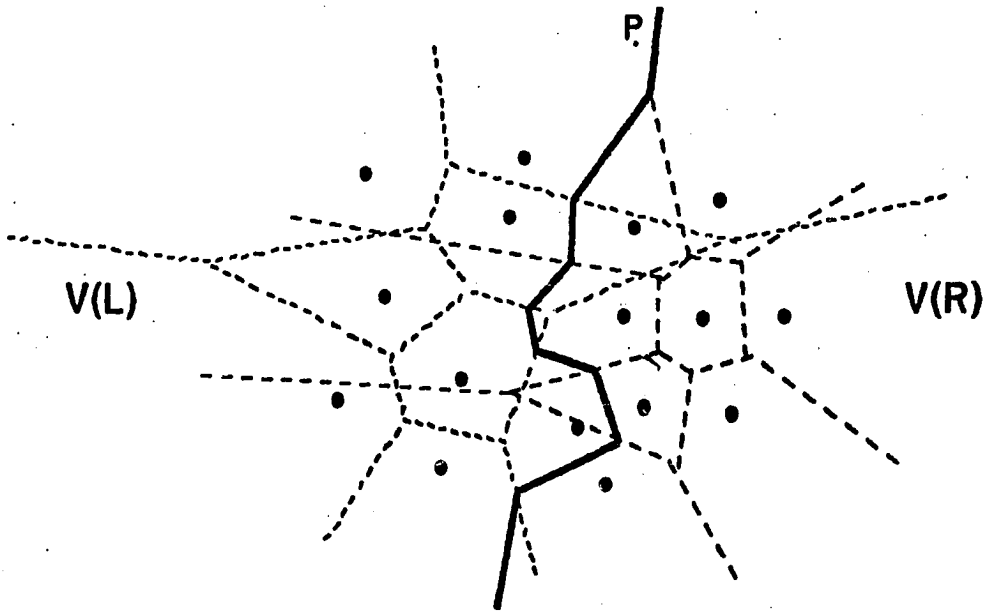


Figure 6.27: V(L), V(R) and P superimposed.

6.4.1 Construction of the Dividing Line

"stepping down by zigzag paths ... "

- Tennyson

"To right or left eternal swervin / They zig-zag on."

- Robert Burns. *To J. S.*

The first step in the construction of P is to find its infinite rays. Since rays of

the Voronoi diagram correspond to pairs of adjacent hull vertices, we must find the two edges of Hull(L U R) that are not present in either Hull(L) or Hull(R).

Theorem 6.16: Given the Voronoi diagram on N points in the plane, their convex hull can be found in linear time.

Proof: Examine the Voronoi edges until a ray r is found. Let us say that the polygon to the left of r (in the directed sense) is V(i). Then p_i is a hull vertex. Scan the edges of V(i) until another ray is found. This will give another hull point p_j, and we now scan V(j), etc., until we return to V(i). An edge is examined only when one of the polygons containing it is scanned. Since each edge occurs in exactly two polygons, no edge is examined more than twice, and linear time suffices.

To find the infinite rays of P we use V(L) and V(R) to obtain Hull(L) and Hull(R) in linear time, then the hull of the union can be found using Algorithm A3.3. (Figure 6.28.) The rays are then the perperdicular bisectors of the segments joining Hull(L) and Hull(R).

It will now be useful to refer to the example in Figure 6.29. The upper ray of P is the bisector of points 7 and 14. Imagine a point z on the ray, moving down from infinity. Initially z lies in polygons V(7) and V(14). It will continue to do so until it crosses an edge of one of these polygons, when it will follow a zigzag course in a different direction. In this case, z encounters an edge of V(14) before it hits any edge of V(7). This means that z is now closer to point 11 than it is to point 14, so it must move off along the 7-11 bisector. It continues until the 6-7 edge of V(7) is reached, and moves off along the 6-11 bisector. Eventually it hits the 10-11 edge of V(11) and proceeds via the 6-10 bisector. This jagged walk continues until the bottom ray of P is reached.

It is straightforward now to show that P can be found in linear time, and we make use of the property that it always moves *downward*. The moving point z always lies in two Voronoi polygons, one in the left set and one in the right set. (This is because both Voronoi diagrams partition the plane.) To find the next point at which P changes direction, it is only necessary to examine these two polygons. If we were

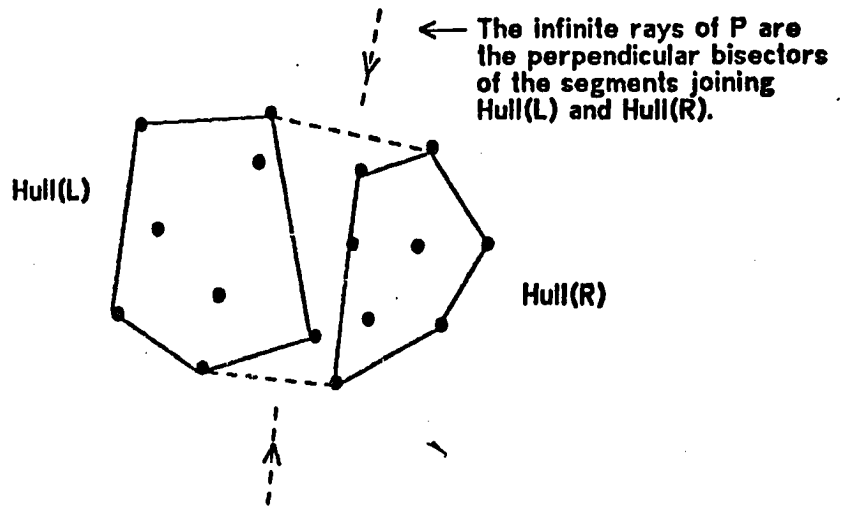


Figure 6.28: Finding the Infinite Rays of P.

able to argue that no Voronoi edge is scanned more than twice, it would follow immediately that only O(N) time is required. If we were to scan all of V(i) each time, far too many edge examinations would be performed; in fact, this procedure could take as much as quadratic time.

The solution is to organize the polygon scanning more sensibly, using the fact that P is monotonic.[22] Say that P begins in polygons V(i) and V(j), with $i \in L$ and $j \in R$. We will continually maintain two pointers, *l* and *r*, to the edges e_l and e_r that P would intersect in V(L) and V(R) if it continued in its present direction. We may find the initial values of *l* and *r* in a single scan of V(i) and V(j). Let e_z be the edge that P intersects first. If $e_z = e_l$, P will bend toward the right as it passes through the edge. To find the next edge that P intersects, we move *l* counterclockwise in the new polygon in V(L) shared by e_z and *r* clockwise in the current polygon in V(R), starting from the last examined edge. Likewise, if $e_z = e_r$, we move the pointer *r*

[22]Material provided by D. T. Lee [Lee (76a)] helped considerably to clarify this paragraph.

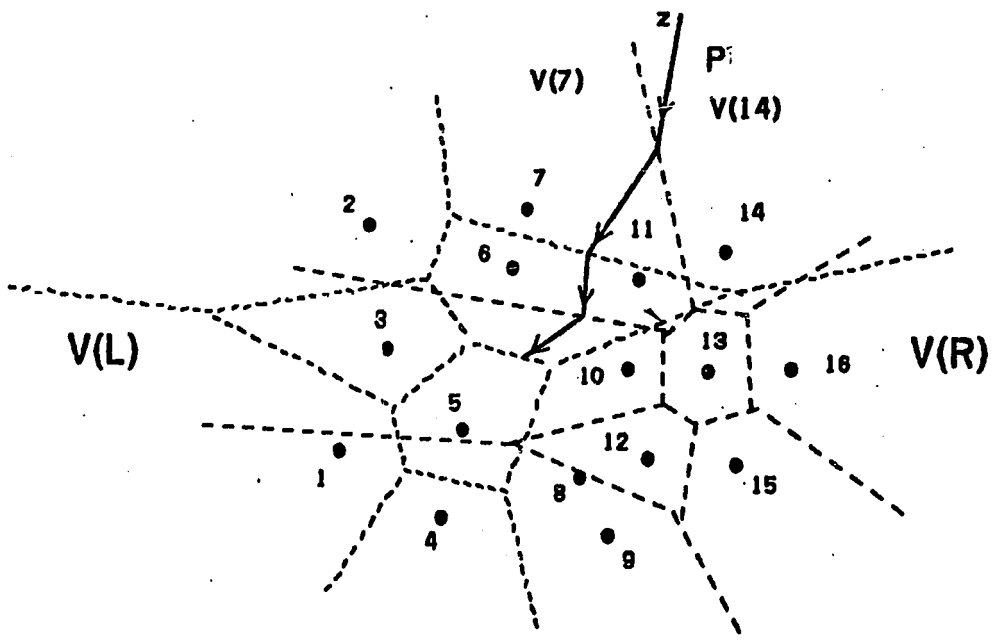


Figure 6.29: A Zigzag Walk to Construct P.

clockwise through the edges of the polygon in V(R) that shares e_z and l counterclockwise in its polygon, beginning from the last edge examined. Note that the polygons in V(L) are always scanned counterclockwise and those in V(R) clockwise.

The scan in any polygon always proceeds from the last edge examined, so no backtracking is ever necessary. At each step in this process, we always do one of the following:

1. Examine a new edge of V(L) and increment l (if it does not intersect P).

2. Examine a new edge of V(R) and increment r (if it does not intersect P).

3. Create a new Voronoi point (where P intersects an edge).

Since there are at most 3N-6 edges in V(L) and V(R) together and at most N vertices in P, the entire construction of P takes only linear time. A complete implementation and formal proof of this procedure appears in [Lee (76a)]. Such a proof can be given only for a specific data structure.

Recall that to form the final Voronoi diagram we must discard all edges of V(L) that lie to the right of P and all edges of V(R) that lie to the left. The edges that are intersected by P form cutsets in both V(L) and V(R) that partition these graphs into two components. The components can be determined by breadth-first traversal in time that is proportional to the number of edges involved. It follows that the process of merging V(L) and V(R) to form V(S) takes only linear time.

Theorem 6.17: The Voronoi diagram of a set of N points in the plane can be constructed in O(N log N) time, and this is optimal.

Proof: For a detailed specification of a data structure, algorithm, and proof of correctness, the reader is referred to the Master's thesis of D. T. Lee [Lee (76a)]. The time required by the recursive merge procedure is described by the recurrence relation $T(N) = 2T(N/2) + O(N) = O(N \log N)$. Optimality was shown in Theorem 6.14.

6.5. Voronoi Applications

We now show how the Voronoi diagram can be used to solve all of the closest-point problems efficiently.

Theorem 6.18: The ALL NEAREST NEIGHBORS problem can be solved in O(N log N) time, and this is optimal.

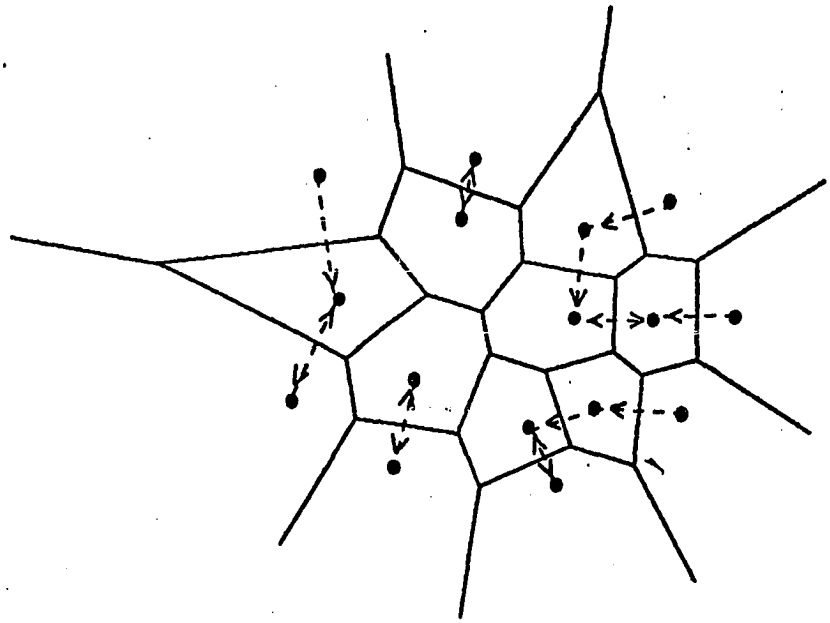


Figure 6.30: The Nearest-Neighbor Relation.

Proof: By Theorem 6.11, every nearest neighbor of a point p_i defines an edge of V(I). To find a nearest neighbor of p_i, it is only necessary to scan each edge of V(I). Since every edge belongs to two Voronoi polygons, no edge will be examined more than twice. Thus, given the Voronoi diagram, all nearest neighbors can be found in linear time. Optimality was shown in Section 6.1.2.

In nearest-neighbor searching, we are given a set of points, and we wish to preprocess them so that given a new point z, its nearest neighbor can be found quickly. However, finding the nearest neighbor of z is equivalent to finding the Voronoi polygon in which it lies. The preprocessing just consists of creating the Voronoi diagram ! Since the diagram is a planar straight line graph, it can be searched using any of the methods given in Section 4.4.1.

Theorem 6.19: [Cf. Theorem 4.10] Nearest-neighbor search can be performed in O(log N) time, using $O(N^2)$ storage and $O(N^2)$ preprocessing time.

Theorem 6.20: [Cf. Theorem 4.11] Nearest-neighbor search can be performed in $O(\log^2 N)$ time, using O(N) storage and O(N log N) preprocessing time.

Theorem 6.21: [Cf. Theorem 4.12] Nearest-neighbor search can be performed in O(log N) time, using O(N) storage and O(N log N) preprocessing time.

To show how to construct a Euclidean minimum spanning tree, we review Prim's algorithm:

Algorithm A6.4: Minimum Spanning Tree [Prim (57)]

```
1. Begin with all points unlabeled except some
       arbitrary point P.

2. WHILE (Some point is unlabeled) DO BEGIN
       Find the shortest edge joining a labeled point P
       with an unlabeled point Q;
       Add edge (P,Q) to the spanning tree;
       Label Q;
   END
```

Ties for the shortest edge may be resolved arbitrarily.

We now use the correctness of Prim's algorithm (not the algorithm itself) to show that the minimum spanning tree on a set of points in the plane is a subgraph of the Voronoi dual.

Theorem 6.22: Every Euclidean minimum spanning tree of a set of points is a subgraph of the Voronoi dual.

Proof: We need only show that each edge added in Step 2 of Algorithm A6.4 is an edge of the dual. Consider any subset U of points of S. It will suffice to prove that a shortest segment joining a point u of U and a point v of S-U is a dual edge. It is clear that uv intersects at least one edge of V(S), since it begins in V(u) and terminates in V(v). However, uv cannot intersect more than one edge of V(S) (except possibly at a Voronoi vertex) because in so doing it would enter some other polygon V(t) and we would then have either tu < uv or tv < uv, which contradicts the hypothesis that uv is shortest. Thus V(u) and V(v) are adjacent polygons, so uv is a dual edge.

Given the Voronoi diagram, the straight-line dual can be constructed easily in O(N) time by merely joining the pair of points that define each Voronoi edge. Since the MST is a subgraph of the dual, it is also a minimum spanning tree of the dual. The dual, however, is a planar graph for which a minimum spanning tree can be found in O(N) time [Cheriton (76)], so we have

Theorem 6.23: A minimum spanning tree on N points in the plane can be found in O(N log N) time, and this is optimal.

Let us say that a figure is *empty* with respect to a point-set S if it contains no points of S. Since we have already seen that the Voronoi dual is a triangulation (see Figure 6.20), it follows that

Theorem 6.24: A triangulation with the property that the circumcircle of every triangle is empty can be found in O(N log N) time, and this is optimal for any triangulation.

Theorem 6.25: The largest empty circle problem can be solved in O(N log N) time.

Proof: Given N points in the plane, consider the function $f(x,y)$, the distance from (x,y) to the nearest given point. Within a Voronoi polygon, f is convex so it attains a maximum at an extreme point of the polygon. For each Voronoi point

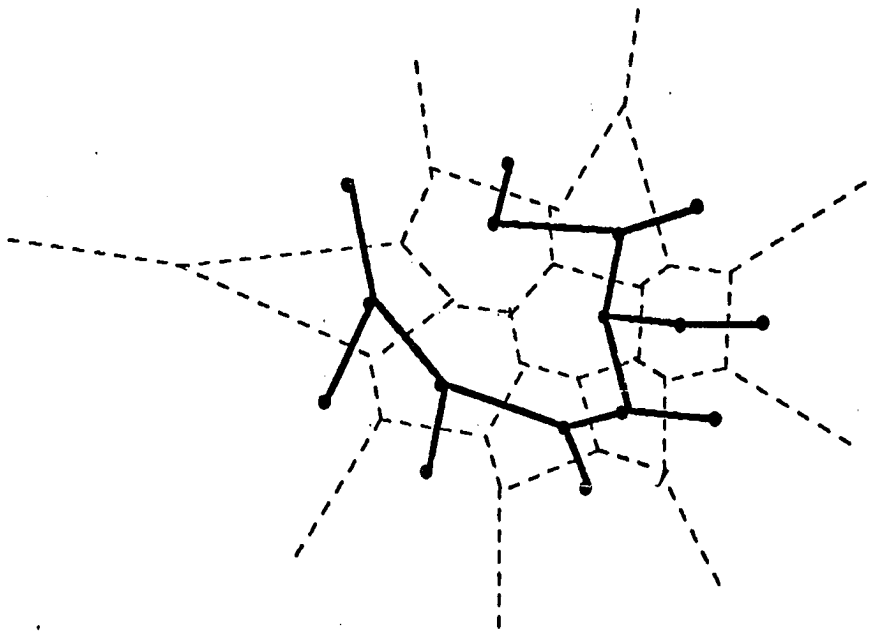


Figure 6.31: The MST is a subgraph of the Voronoi Dual.

P, the value of *f* is just the distance from the point to the "owner" of any of the three polygons that meet at P. Since we have constrained the center of the largest empty circle to lie within the convex hull of the set, this center must occur either at a Voronoi point or at an intersection of a Voronoi edge and a hull edge. (If (x,y) is an interior point of a Voronoi polygon, the convexity of *f* implies that *f*(x,y) can be increased by moving in one of the two directions determined by any line through (x,y).) All of the Voronoi points can be found in O(N log N) time, and each one can be checked for hull inclusion in O(log N) time, so it only remains to show that the hull intersections can be found quickly. Consider any edge E of the hull. Corresponding to E is a ray *r* which coincides with the perpendicular bisector of E (Theorem 6.13). Each such *r* either intersects E or intersects both Voronoi edges adjacent to E, depending on whether or not the circumcenter associated with *r* lies inside or outside of its Delaunay triangle. In examining each ray and its two neighbors, no edge will be scanned more than twice, so O(N) time suffices.

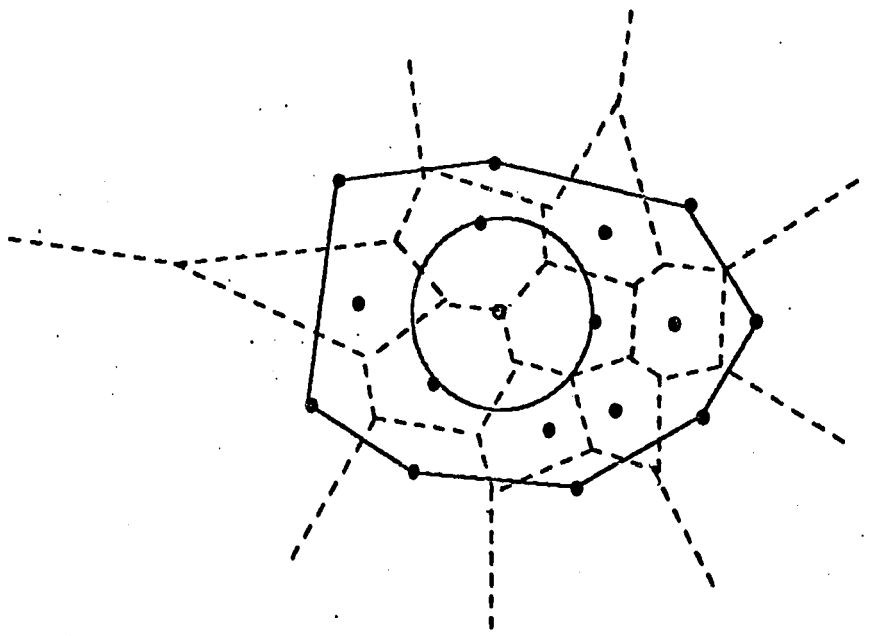


Figure 6.32: The Voronoi Diagram locates a largest empty circle.

It is remarkable that such a diverse collection of problems can be solved by a single unifying structure.

6.6. Generalization of the Voronoi Diagram

The Voronoi diagram, while very powerful, has no means of dealing with farthest points, k-closest points, clustering, and other distance relationships. As such, it is unable to deal with the remainder of the problems we have posed. The difficulty is that we have been working with the Voronoi polygon associated with a single point but such a restriction is not necessary and it will be useful to speak of the *generalized Voronoi polygon* $V^*(T)$ of a subset T of points, defined by

$$V^*(T) = \{x: \forall_{y \in T} \forall_{z \in S-T} \; d(x,y) < d(x,z)\} \qquad (6.4)$$

That is, $V^*(T)$ is the locus of points p such that all points of T are nearer to p than is any point not in T. An equivalent definition is

$$V^*(T) = \cap H(i,j), \; i \in T, \; j \in S-T \;, \qquad (6.5)$$

where H(i,j) is the half plane containing i that is defined by the perpendicular bisector of i and j. This shows that a generalized Voronoi polygon is still convex. It may, of course, happen that $V^*(T)$ is empty. In Figure 6.29, for example, there is no point with the property that its two nearest neighbors are 5 and 13. A set S with N points has 2^N subsets. How many of these can possess non-empty Voronoi polygons? If the number is not large, there will be some hope of performing *k*-nearest-neighbor searching without excessive storage.

Let us define the *Voronoi diagram of order k*, denoted $V_k(S)$ as the collection of all generalized Voronoi polygons of k-subsets of S, so

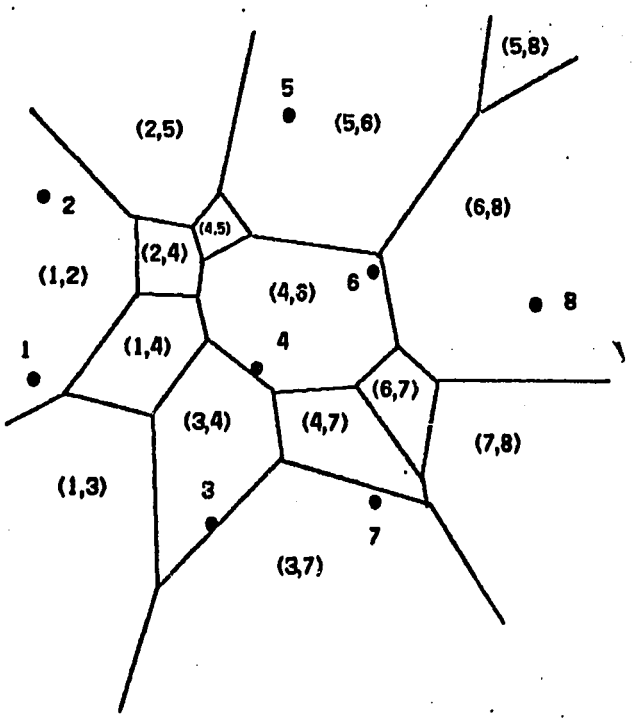
$$V_k(S) = \cup \{ V^*(T) \}, \quad T \subset S, |T| = k \;. \qquad (6.6)$$

In this notation, the ordinary Voronoi diagram is just $V_1(S)$. It is proper to speak of $V_k(S)$ as a "diagram" because its polygons partition the plane (by the same argument as in [Rogers (64)] for the first-order Voronoi diagram). Given $V_k(S)$, the k points closest to a new given point z can be determined by finding the polygon in which *z* lies. Figure 6.33 shows a Voronoi diagram of order two, the set of loci of nearest pairs of points.

In order to obtain bounds on the time and space required to perform k-nearest-neighbor searching, we must compute the number of edges in the order k diagram.

Definition 6.2: A nonempty subset $T \subset S$ is *exposed* iff T and S - T are separable, that is, they lie in complementary half-planes.

Theorem 6.26: The number of unbounded Voronoi polygons (of all orders) of a set of N points is N(N-1).



Some Voronoi polygons of order two are empty. For example, there is no (5,7) polygon.

For a set of N points there are N(N-1)/2 possible polygons. Here, N=8 but only 15 out of the 28 polygons are non-empty.

Figure 6.33: A Voronoi Diagram of Order Two.

Proof: We first show that V(T) is unbounded iff T is exposed, and then count the number of exposed subsets. (This generalizes the idea of the ordinary Voronoi property that unbounded polygons correspond to hull vertices.) If T is exposed, then T and S -T lie in disjoint half-planes which we may take (by rotating and translating the points, if necessary) to be the half-planes $x > 0$ and $x \leq 0$, respectively. Points on the x-axis with sufficiently large coordinates are

closer to some point of T than to any point of S - T, so V(T) must be unbounded. Conversely, if V(T) is unbounded, we can force it to contain all of the positive x-axis (again by rotation and translation) since it must contain a ray. This implies that no vertex of S - T can lie to the right of any vertex of T (as was shown in Section 6.4), so the sets lie in complementary half-planes.

We now exhibit an isomorphism between complementary pairs of exposed subsets and unordered pairs of vertices of S. Given a complementary pair of exposed subsets P and Q, they can be separated by some line L (by definition). Rotate and translate L counterclockwise until it meets some point p of P and q of Q. Since no three points are collinear, p and q are unique. We associate the pair {P,Q} with the vertex pair {p,q}. (See Figure 6.34.) Conversely, given a pair of vertices {p,q}, we can recover the associated subset pair uniquely: Let p lie above or to the right of q and let L be the line passing through p and q. Define P to be the subset consisting of p and the set of points lying to the left of L. Q is the complement of P. Since two exposed subsets are associated with every unodered pair of vertices of S, so there must be N(N-1) exposed subsets.

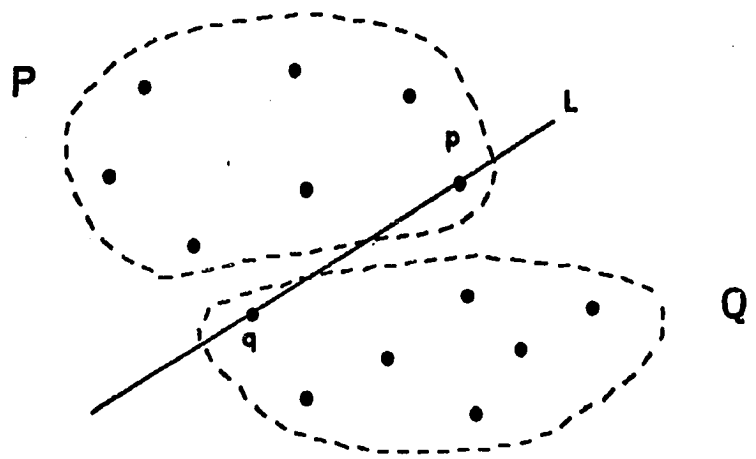


Figure 6.34: Enumerating Exposed Subsets

Theorem 6.27:

The number of bounded Voronoi polygons (of all orders) of a set of N points is $\binom{N-1}{3}$.

Proof: We first show that each triple of points of S determines two generalized Voronoi vertices. Given points a, *b*, and c, let x be their circumcenter and let R be the set of points lying strictly inside their circumcircle. (R may be empty.) Define k = |R|, the number of points in the circle centered at x. Consider the order k+1 Voronoi diagram. The point x is common to the polygons V(R ∪ *a*), V(R ∪ *b*), and V(R ∪ c). In the order k+2 diagram, x is common to V(R ∪ a ∪ *b*), V(R ∪ *b* ∪ c), and V(R ∪ c ∪ a). Since no four points are cocircular, x cannot be a vertex of any diagram of order greater than k+2. Thus there are two Voronoi vertices for every triple of points.

Except for degeneracies, which can only reduce the number of distinct polygons, each Voronoi vertex has degree three (excluding the vertex at infinity). If a planar graph has F faces, V vertices of degree three, and one vertex of degree D, then 2F = 2 + V + D (by Euler's Formula). Letting F_k denote the number of non-empty Voronoi polygons in the order k diagram, V_k the number of Voronoi vertices, and D_k the number of unbounded regions, then summing over k from 1 to N-1 we have

$$2\sum_k F_k = 2N - 2 + \sum_k V_k + \sum_k D_k.$$

But, by Theorem 6.13,

$$\sum_k D_k = 2\binom{N}{2} \quad \text{and} \quad \sum_k V_k = 2\binom{N}{3},$$

so we obtain

$$\sum_k F_k = N - 1 + \binom{N}{2} + \binom{N}{3} = N(N-1) + \binom{N-1}{3}.$$

Since the number of unbounded polygons is N(N-1), the result follows.

Thus, the total number of polygons in all of the Voronoi diagrams combined is $O(N^3)$, not 2^N.

Theorem 6.28: The number of regions in $V_k(S)$ is $O(k(N-k))$. [23]

Because each vertex is of degree three, this means that the number of Voronoi edges is also $O(k(N-k))$. The union of the Voronoi polygons of all orders is precisely the set of perpendicular bisectors of pairs of points of S.

By starting with the order one diagram and successively updating it through orders 2,3,...,k, Lee has been able to prove [Lee (76a)]

Theorem 6.29: The order k Voronoi diagram on a set of N points can be obtained in $O(k^2N \log N)$ time, using $O(k^2(N-k))$ storage.

The next theorem follows from an earlier result on planar graph searching:

Theorem 6.30: [Cf. Theorem 4.12] The k nearest out of N neighbors of a point can be found in $O(\max(k,\log kN)$ search time and $O(k(N-k))$ storage, after $O(k^2N \log N)$ preprocessing.

Note that the search always requires at least O(k) time since k objects are being retrieved.

The generalized Voronoi diagram unifies closest- and farthest-point problems since the locus of points whose k nearest neighbors are the set T is also the locus of points whose N-k *farthest* neighbors are the set S - T. Thus, the order k closest-point diagram is exactly the order N-k farthest-point diagram. Let us examine one of these more closely, the order N-1 closest-point diagram, or the order 1 farthest-point diagram (Figure 6.35.)

Theorem 6.31: Given a set S of N points in the plane, all farthest points of S from an arbitrary point z of the plane lie on Hull(S).

---

[23] This result was stated without proof in [Shamos (75c)]. A proof may be found in [Lee (76a)].

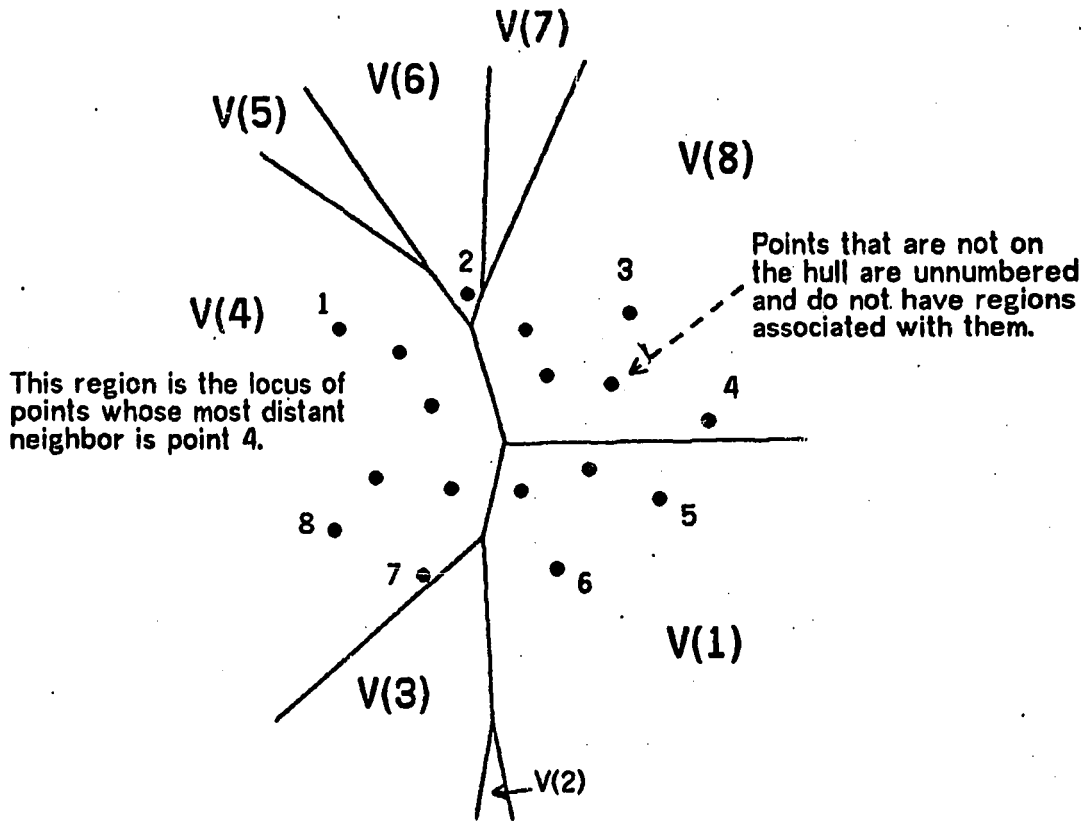


Figure 6.35: The Farthest-Point Voronoi Diagram.

Proof: Let p be some point of S that is farthest from z. Consider L, the straight line that is perpendicular to pz at p. Every point on the opposite side of L from z is farther from z than is p. Thus no point of S can lie in the far half-plane determined by L, or p would not be farthest from z. It follows that L is a supporting line of S and thus p lies on Hull(S).

Associated with each point p_i is a convex polygonal region $V_{N-1}(I)$ such that p_i is the farthest neighbor of every point in the region. By Theorem 6.31, this diagram is determined only by points on the convex hull and these are all exposed, so there are no bounded regions. The farthest-point diagram can be constructed in O(N log N) time by a procedure analogous to the algorithm for the closest-point diagram. Having found the farthest-point diagrams of the left and right halves of the set, the polygonal dividing line P is exactly the same as in the closest-point case. This time, however, we discard all segments of $V_{N-1}(L)$ that lie to the *left* of P, and those segments of $V_{N-1}(R)$ that lie to the right of P.

Definition 6.3: [Harary (71)] A graph is *outerplanar* iff it is planar and all faces are adjacent to one common face.

An outerplanar graph on N vertices has at most 2N-3 edges [Harary (71)].

Theorem 6.32: Given a finite set of points S in the plane and an additional point z, any point of S that is farthest from z lies on the convex hull of S.

Proof: Let s be a farthest-neighbor of z and consider the perpendicular P to line sz at s. Any point that lies on the opposite side of P from z is farther from z than s is, so no such points can exist and P is a line of support of S (Definition 3.6). But then s lies on Hull(S).

It follows from Theorem 6.32 that a farthest-point Voronoi diagram possesses no closed regions because the only regions correspond to hull vertices and thus must be unbounded by the proof of Theorem 6.26. As a consequence, the straight-line dual of the farthest-point diagram is outerplanar and thus has at most 2N-3 edges.

The farthest-point diagram provides another algorithm for set diameter. The two farthest points of S correspond to some edge of the diagram, and we can find them in only O(N) additional time.

Theorem 6.33: The smallest circle enclosing a set of N points in the plane can be found in O(N log N) time.

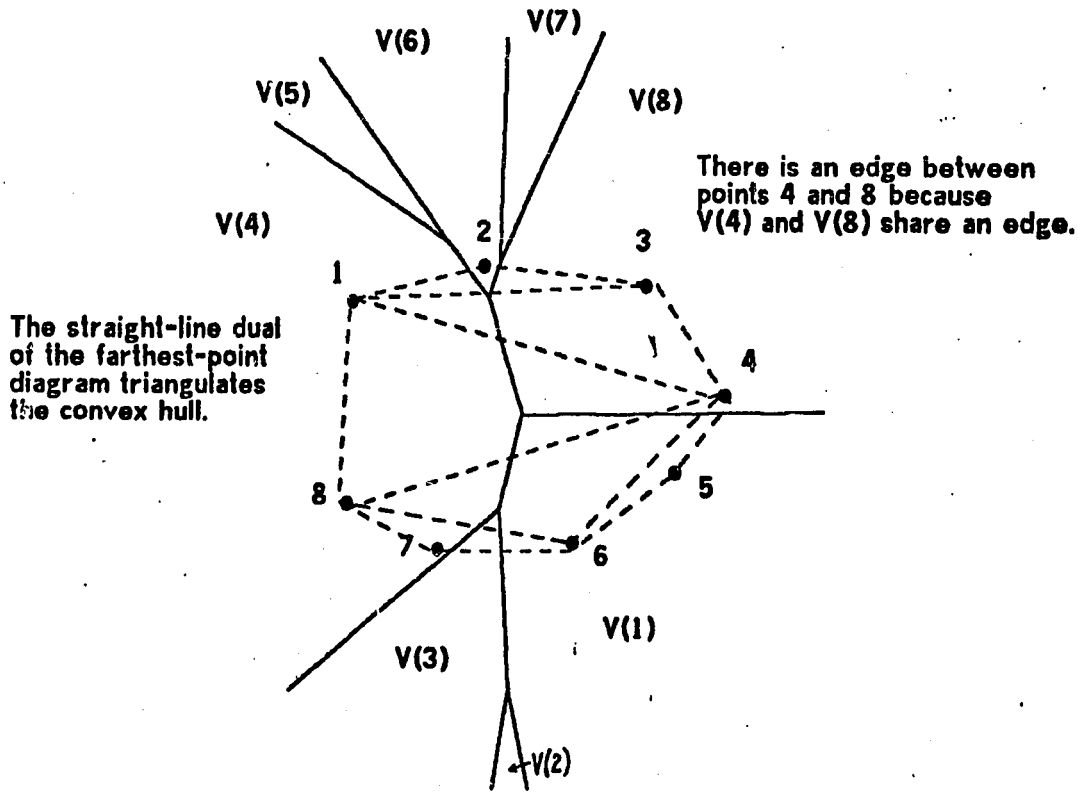


Figure 6.36: The Dual of the Farthest-Point Diagram.

Proof: We know from Section 6.1.7 that the required circle is determined either by the diameter of the set or by three of its points. We can find the diameter in O(N log N) time from the farthest-point diagram and determine whether it encloses the set. If so, we are done. Otherwise, we claim that the center C of

the circle will lie at a vertex of the farthest-point diagram. Let the circle pass through points P, Q, and R. These must be farthest neighbors of C, since if there were a more distant point D from C, the circle would not enclose it. Therefore, C is a common point of the polygons $V_{N-1}(P)$, $V_{N-1}(Q)$, and $V_{N-1}(R)$ and must be a vertex of the diagram. The diagram contains only O(N) points and the circumradius associated with each vertex is the distance from it to any of the three points of whose polygons it is the intersection. The maximum over all vertices of this distance is the radius of the circle.

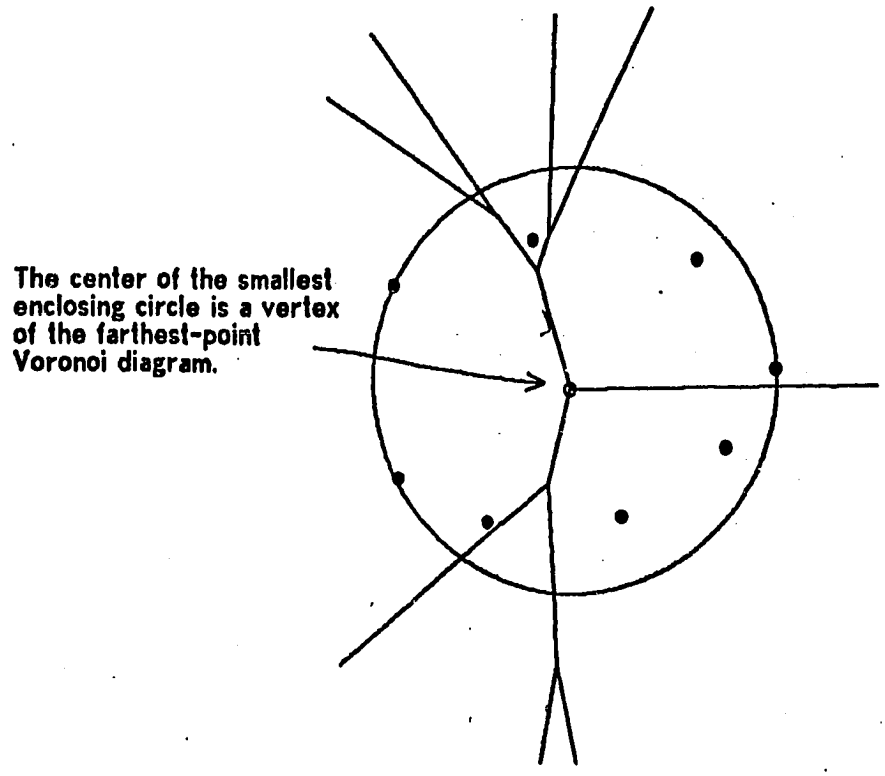


Figure 6.37: The Smallest Enclosing Circle and the Farthest-Point Diagram.

6.6.1. Voronoi Extensions

It is natural to try to extend the Voronoi idea to solve closest-point problems in higher dimensions. Theoretically, there is no difficulty: Voronoi polytopes exist in every dimension, are convex, and partition the entire space. Furthermore, the straight-line dual induces a simplicial partition. For example, in three dimensions the dual partitions space into tetrahedra. From a practical standpoint, though, these polyhedra are not very useful. Preparata has shown that the Voronoi diagram on N points in 3-space may have $O(N^2)$ vertices [Preparata (77a)]. A set which realizes this bound is one having N = 2k points, k of which lie on the unit circle in the x-y plane and the remaining k at the locations $(0,0,2^i)$, i=1,...,k, on the z-axis. This implies a trivial lower bound of $\Omega(N^2)$ time for any algorithm based on Voronoi polyhedra.

Lee and Wong [Lee (77b)] have shown that our Voronoi construction algorithm in the plane generalizes to the L_1 and L_∞ metrics, again yielding O(N log N) algorithms for the closest-point problems.

6.7. Unsolved problems

1. Is there a simple divide-and-conquer algorithm for diameter analogous to Strong's D&C for closest-pair?

2. Is there a fast expected-time algorithm for nearest-neighbor searching in the plane? In particular, can O(log log N) average search time be achieved for reasonable distributions?

3. Can one find the MST for N points in the plane without forming the entire Voronoi diagram? (D&C with a fancy merge step?)

4. Are the edges of a minimum Euclidean matching necessarily edges of the Voronoi dual?

5. How much time is required to construct an order k Voronoi diagram directly?

6. Is a traveling salesman tour necessarily a subgraph of the Voronoi dual?

7. Even though the Voronoi diagram in three-space may have $O(N^2)$ edges, is it still possible to find the minimum spanning tree quickly? (It seems as though one would have to generalize the solution to Problem 3, however.)

8. (Minimum spanning forest) Find a forest of least total length on N points such that each point is incident with at least one edge. (This is not the same problem as minimum matching.)

9. A triangulation of a polygon is a decomposition of its interior into disjoint triangles. Preparata has shown that a simple polygon can be triangulated in O(N log N) time. (This does not follow from Theorem 6.9.) Is O(N) time achievable?

10. For every set of points in the plane, there is an MST on that set having maximum degree five. If we restrict the maximum degree to be two, the traveling salesman path problem results, which is NP-complete. At what point does the degree-constrained MST problem become NP-complete? Degree 2, 3, or 4?

11. (Maximum spanning tree) Given N points in the plane, find a spanning tree of greatest total length. (This is not an idle problem. Maximum spanning trees have been used in clustering by S. C. Johnson.) Unfortunately, the MXST is not a subgraph of the dual of the farthest-point Voronoi diagram. Can the MXST be found in O(N log N) time? (Any efficient implementation of Prim's algorithm will find it in $O(N^2)$ time.)

12. Given the vertices (in order) of a convex polygon, how quickly can the MST be found? D.-T. Lee has shown that the two closest points of a convex polygon can be found in O(N) time.

6.8. Summary

This chapter unifies the whole of computational geometry by combining all of the methods we have developed for searching and intersection into a coherent set of tools for solving problems based on the proximity of points -- the closest-point problems: closest-pair, all nearest neighbors, minimum spanning tree, triangulation, smallest enclosing circle, and largest empty circle. We investigate the computational properties of the Voronoi diagram, a planar graph whose regions are the loci of proximity surrounding each point, and find that the structure can be created, manipulated, and stored efficiently, yielding O(N log N) algorithms for all of the problems. The Voronoi construction is an elementary but complex application of divide-and-conquer, with an involved merge step that is based on geometric features of the diagram. The straight-line dual of the Voronoi diagram is a planar graph that is of special interest because it is a triangulation of the given point set and contains the nearest-neighbors graph and minimum spanning tree as subgraphs. A lower bound on the Voronoi construction and most of the closest-point problems follows from reducibility with either sorting or the element uniqueness problem.

We complete the connection between closest-point and farthest-point problems by defining the Voronoi diagram of order k, which consists of regions that are loci of all points x such that a given k points are the k nearest neighbors of x. This generalization allows efficient solution of the k-nearest neighbors and smallest enclosing circle problems.

Chapter 7
Epilog

7.1. New Directions

The present work, broad though it may be, has barely scratched the surface in several important areas, and I have lived with it long enough to see its shortcomings. Here is list of topics that seem promising for future research.

1. Higher dimensions. This thesis might well have been titled "Computational Geometry in the Plane", but I did not feel compelled to be so restrictive. The transition from two dimensions to N is so easy in linear algebra, why should it not be so in computational geometry? In many applications, even three dimensions would suffice. One difficulty is that a problem may have many solutions in the plane, only one of which generalizes to higher dimensions. It is not easy to recognize such a solution. For example, the only convex hull algorithm in two dimensions that gives rise to an efficient algorithm in three-space is our divide-and-conquer algorithm based on finding the hull of the union of convex polygons. Even D&C has its limitations in multiple dimensions. While Bentley and Shamos [Bentley (76b)] have shown that the two closest of N points can be found in O(N log N) time in any dimension k, the constant of proportionality grows exponentially with k. This is because no D&C scheme has yet been devised which does more than reduce the dimension by one at each step of the recursion. One ordinarily works with recurrences of the form

$$T(N,k) = 2T(N/2,k) + T(N,k-1) + O(N) = O(N \log^k N) \quad . \qquad (7.1)$$

For the closest-pair problem we were able to reduce this to O(N log N) only through a drastic reduction in the number of points remaining at each level of recursion. Bentley's thesis contains a number of valuable heuristics for decomposing higher-dimensional problems [Bentley (76a)].

2. Average-case analysis. The amount of effort being devoted to analyzing the expected behavior of algorithms is increasing dramatically, and the general conclusion seems to be that many algorithms perform much better on the average than worst-case analysis would suggest. Unfortunately, average-case analysis in geometry involves fairly ponderous mathematics that will probably remain inaccessible to many researchers [Santalo (76)].

3. Geometric lower bounds We often exploit the special structure of geometry problems to produce efficient programs; why can we not derive lower bounds from this same structure? One problem is that many geometric questions involve auxiliary functions such as square roots and trigonometric functions which present methods of algebraic complexity will not handle, so new techniques are required. Shamos and Yuval [Shamos (76c)] have shown that determining the average distance between N points in the plane must take $\Omega(N^2)$ operations, even if arbitrary single-valued functions are allowed in addition. This lower bound is non-trivial because the problem has only 2N inputs and a single output.

4. NP-complete problems All of the effort in this thesis was concentrated on studying problems for which efficient algorithms could be derived. When it was begun, no geometric problem was known to be NP-complete, although some were suspected of being difficult (ETSP, Steiner Tree). Garey, Graham and Johnson have made tremendous strides, proving the NP-completeness of optimal linear arrangement, densest hemisphere, Euclidean TSP, and Steiner tree in various metrics. Hopefully, the fact that metric properties do not make these problems tractable will shed some light on the structure of the class NP.

5. Approximation algorithms One way of circumventing NP-completeness is to accept an approximate answer to a problem rather than an exact one. F. K. Hwang has obtained bounds on how well a minimum spanning tree approximates a Steiner tree in different metrics. The Christofides heuristic [Christofides (76)] for the traveling salesman problem discussed in Chapter 6 typifies current work in approximate algorithms. The goal is to produce a solution that is *always* within some multiplicative factor of the true or optimal solution; the cost of obtaining the approximation normally depends on how close an answer is desired. Other approximation schemes for the TSP have been given by [Kim

(75)] and [Rosenkrantz (74)]. Most work in approximate algorithms centers on intractable problems for obvious reasons, although this need not be the case. If N is large, a quadratic algorithm may be as useless as an exponential one. With this in mind, Shamos and Yuval [Shamos (76c)] derived a linear-time approximation for the mean distance between points in the plane and suggested a general method by which such results can be obtained.

6. Probabilistic algorithms Another alternative to NP-completeness is to drop the requirement that an algorithm always produce the correct, or even an approximate, answer. Richard Karp has given an algorithm that almost always produces a traveling salesman tour that is within a factor of $1 + \epsilon$ of optimal and almost always runs in O(N log N) time [Karp (76)]. (Here, "almost always" is to be taken in its precise probabilistic sense.) Rabin has shown, using probabilistic arguments, that if the FLOOR function is allowed, the two closest of N points in the plane can be found in expected time O(N) [Rabin (76)]. This approach seems to be able to yield geometric algorithms of startling efficiency.

7. Parallel algorithms An area of study that has barely been touched is the decomposition of geometric problems for parallel hardware. Suppose that 1024 processors are available. How should one proceed to find a minimum spanning tree? Many geometry problems are inherently local -- Prim's algorithm, for example, shows that one can construct minimum spanning trees via neighborhood search alone, up to a point. The nearest-neighbors graph itself provides at least half of the edges of the MST. This locality suggests a way of splitting the MST problem. Divide the problem into rectangles, find the MST recursively in each, then perform a fixup step to produce the global solution. A large number of other problems have similar local features. The hidden line problem can be divided conveniently among as many processors as are available because the intersections of objects that occur in one region do not affect other regions (modulo such global information as which objects obscure others, etc.) There seem to be many profitable avenues of research in parallel geometric algorithms.

7.2. A Final Note

We have set out to establish a new discipline by asking very elementary questions and being satisfied with nothing short of complete answers to them. These "questions" are in actuality fundamental computational problems that arise throughout geometry, and their "answers" are optimal algorithmic tools used to construct more complicated programs. We achieve great unification by using one structure or algorithm to solve many problems and by using one reducibility idea to prove several lower bounds.

The approach justifies itself by enabling us to derive fast algorithms for a host of problems that were previously treated by less efficient ad hoc methods. Will it work in areas other than geometry? We have applied the precepts expounded in this thesis to computational problems in statistics [Shamos (76a)] with extremely satisfying results. Not only are new results produced, but they come quickly, once the basic problems are isolated. A companion volume titled Computational Statistics is in preparation, and it provides even more evidence of the soundness of our technique.

"Anyone who has studied geometry is infinitely quicker of apprehension."

- Plato, *Republic*.

Appendix A
The Algebraic Approach

> *"Equations are Expressions of Arithmetical Computation, and properly have no place in Geometry."*
> - Newton, *On the Linear Construction of Equations.*

The theory of algebraic complexity is now well-developed [Borodin (73)]. Since analytic geometry allows us to formulate any geometric problem as an algebraic one, does it not stand to reason that that computational geometry should be an offshoot of algebraic complexity? Believing this to be true, the author set off some years ago to study complexity questions in analytic geometry, but was quickly stymied. The chief reason was that *none* of the problems treated in this thesis can be expressed in a purely algebraic way. One cannot, for example, write the coordinates of the convex hull of a set S as a simple function of the coordinates of the points of S. This is true because the convex hull is described by inequalities among the variables, not by equalities. Thus one has no explicit formula to evaluate, so such problems cannot be analyzed by the techniques of algebraic complexity. This is exactly analogous to the inability of differential calculus to deal with constrained optimization problems, in which a maximum can occur on the boundary of a region. The effort must then be concentrated on finding that boundary. This Appendix is devoted to a short description of the one success we *did* have in applying algebraic complexity to geometry.

The area of a triangle is given classically by half the absolute value of the determinant of the three vectors (x_1, x_2, x_3), (y_1, y_2, y_3), and (1, 1, 1), which can be evaluated in a completely straightforward manner in three multiplications and five addition/subtractions. Should we have any reason to believe that this method is optimal?

The corresponding formula for the area of a general polygon is

$$2 \times \text{AREA} = \left| \sum_{i=1}^{N} x_i(y_{i+1}-y_{i-1}) \right| , \qquad \text{(A.1)}$$

where subscripts are reduced modulo N. ($N + 1 \equiv 1$). If this form is evaluated as written, N multiplications and 2N-1 addition/subtractions are required, for a total of 3N-1 arithmetic operations. We may inquire as to whether this optimal by using in independence technique due to [Winograd (70)].

Equation (A.1) is already in the form of an inner product, one of whose factors is the vector of indeterminates x_i. By Winograd's theorem, the number of multiplications required to evaluate (A.1) is at least as large as the rank of the other factor over the real field extended by the indeterminates y_i. This other factor is just the row vector

$$(y_2-y_N , \quad y_3-y_1 , \quad y_4-y_2 , \quad \cdots \quad y_1-y_{N-1})$$

The elements of this vector sum to zero since each y_i appears exactly twice, once with a positive sign and once with a negative sign. Thus the rank is at most N-1. If N is even, the sum of the odd-numbered terms is also zero, and the rank is at most N-2.

These bounds are achievable as follows:

N odd:

$$2\text{xAREA} = \left| \sum_{i=1}^{N-1} (x_i-x_N)\,(y_{i+1}-y_{i-1}) \right| \qquad \text{(A.2)}$$

N even:

$$2\text{xAREA} = \left| \sum_{i}^{N/2} \qquad x_1)\,(y_{2i}-y_{2i-2}) + (x_{2i-2}-x_N)\,(y_{2i-1}-y_{2i-3}) \right| \qquad \text{(A.3)}$$

These formulas are c... under the assumption that the x_i and y_i do not commute multiplicatively.

N =	3	4	5	6	7	8	N odd	N even
Mults (old)	3	4	5	6	7	8	N	N
Mults (new)	2	2	4	4	6	6	N-1	N-2
Adds (old)	5	7	9	11	13	15	2N-1	2N-1
Adds (new)	5	5	11	11	17	17	3N-4	3N-7
Total (old)	8	11	14	17	20	23	3N-1	3N-1
Total (new)	7	7	15	15	23	23	4N-5	4N-9

Table A.1. Operation Counts for Area Computation

Thus we find a reduction in total arithmetics for N=3, 4, and 6. The case of the quadrilateral is quite unexpected, since its area can be found just as easily as that of a triangle. The explicit formula is :

$$2 \times \text{AREA} = |(x_3-x_1)(y_4-y_2) + (x_2-x_4)(y_3-y_1)| . \qquad (A.4)$$

which save four operations over the eleven required classically.

David Kirkpatrick[1] has obtained lower bounds on the number of additions required to evaluate (A.1): 2N-2 for N odd, and 2N-3 for N even. Thus a lower bound on the total number of arithmetics is 3N-3 for N odd and 3N-5 for N even. Since the classical form can be computed in 3N-1 operations, it is never more than four operations away from optimality. The case N=4 is the only known example of a saving of four operations.

The "new" method of calculating areas (Equations (A.2) and (A.3)), has a simple interpretation. The coordinates are transformed by subtraction so that one vertex is at the origin. This enables it to be eliminated from any subsequent multiplications and additions. Unfortunately, the transformation requires O(N) subtractions in order to save only a constant number of multiplications, and so it is asymptotically inferior to the classical expression (A.1).

[1]Private communication to S. C. Eisenstat. March, 1974.

What is the significance of a saving of one or two multiplications? It is conceivable that there are applications in which the areas of triangles must be computed rapidly, but what is more significant is that we have been finding the areas of small polygons for hundreds of years without knowing that the triangle and quadrilateral are of equal complexity and that the determinant formula is not optimal. One need only scratch the surface of computational geometry to encounter the unexpected.

Without the absolute value signs, Equation (A.1) computes a quantity known as the *signed area* of polygon P [Lopshits (70)], which is positive if the vertices of P are in counterclockwise sequence and negative if they are clockwise. We may use these property to *define* the notions of "clockwise" and "counterclockwise" and apply (A.1) to determine the orientation of P in linear time.

Equation (A.1) may also be used to determine whether a point D lies inside or outside a triangle ABC that is in standard form. If the signed areas of ABD, BCD, and CAD are all positive, then D lies inside ABC. If any of the areas is zero, then D lies on the boundary of ABC. Otherwise, D lies outside.

A.1. Unsolved problem

1. Give another example of a geometric problem that can be solved using algebraic techniques.

References

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

[Akers(72)] Akers, S. B., *Routing*, in Design Automation of Digital Systems , Breuer, M., ed. Prentice-Hall (1972).

[Andrews(72a)] Andrews, D. F., Bickel, P. J., Hample, F. R., Huber, P. J., Rogers, W. H. and Tukey, J. W., Robust Estimates of Location, Princeton University Press (1972).

[Andrews(72b)] Andrews, H. C., Introduction to Mathematical Techniques in Pattern Recognition, Wiley-Interscience (1972).

[Artin(67)] Artin, E., Algebraic Numbers and Algebraic Functions, Gordon and Breach (1967).

[Barlow(72)] Barlow, R. E., Bartholomew, D. J., Bremner, J. M. and Brunk, H. D., Statistical Inference Under Order Restrictions, Wiley (1972).

[Bellman(62)] Bellman, R., *Dynamic programming treatment of the traveling salesman problem*, JACM 9 (1962), 61-63.

[Benson(66)] Benson, R. V., Euclidean Geometry and Convexity, McGraw-Hill (1966).

[Bentley(76a)] Bentley, J. L., Divide and Conquer Algorithms for Closest Point Problems in Multidimensional Space, PhD. Thesis, University of North Carolina Department of Computer Science (1976).

[Bentley(76b)] Bentley, J. L. and Shamos, M. I., *Divide-and-Conquer In Multidimensional Space*, Eighth Annual ACM SIGACT Conference (May, 1976), 220-230

[Bentley(77a)] Bentley, J. L. and Shamos, M. I., *A problem in multivariate statistics: algorithm, data structure, and applications*, extended abstract (1977).

[Bentley(77b)] Bentley, J. L. and Shamos, M. I., *Divide and Conquer for Linear Expected Time*, Info. Proc. Lett. (1977). To appear.

[Bentley(77c)] Bentley, J. L., Kung, H. T., Schkolnick, M. and Thompson, C., *On the average number of maxima in a set of vectors*, (1977). Submitted for publication.

[Bezier(72)] Bezier, P., Numerical Control -- Mathematics and Applications, Wiley (1972). Translated by A. R. Forrest.

[Bliss(33)] Bliss, G. A., Algebraic Functions, American Mathematical Society (1933). Reprinted 1966 by Dover.

[Blum(72)] Blum, E. K., Numerical Analysis and Computation: Theory and Practice, Addison-Wesley (1972).

[Borodin(73)] Borodin, A., *Computational complexity: theory and practice*, in Currents in the Theory of Computing , Aho, A., ed. Prentice-Hall (1973).

[Boyce(75)] Boyce, W. M. and Seery, J. B., *STEINER 72, An Improved Version of Cockayne and Schiller's Program STEINER for the Minimal Network Problem*, Technical Report 35. Computing Science, Bell Laboratories (1975).

[Brostow(77)] Brostow, W., Dussault, J.-P. and Fox, B. L., *Construction of Voronoi polyhedra*, (1977). Submitted for publication.

[Cadwell(70)] Cadwell, J. H., Topics in Recreational Mathematics, Cambridge (1970).

[Chaitin(66)] Chaitin, G., *On the length of programs for computing finite binary sequences*, JACM 13 (1966), 547-569.

[Chand(70a)] Chand, D. R. and Kapur, S. S., *An Algorithm for Convex Polytopes*, JACM 17,1 (Jan, 1970), 78-86.

[Chand(70b)] Chand, D. R. and Kapur, S. S., *On Convex Polyhedra*, Math. Mag. (Sep-Oct, 1970), 202-208.

[Cheriton(76)] Cheriton, D. and Tarjan, R. E., *Finding Minimum Spanning Trees*, SIAM J. Comput. 5,4 (Dec, 1976), 724-742.

[Christofides(76)] Christofides, N., *Worst-case analysis of a new heuristic for the travelling salesman problem*, Symposium on Algorithms and Complexity. Department of Computer Science, Carnegie-Mellon University (Apr, 1976)

[Chvatal(75)] Chvatal, V., *A combinatorial theorem in plane geometry*, J. Comb. Theory B 18 (1975), 39-41.

[Coolidge(16)] Coolidge, J. L., A Treatise on the Circle and the Sphere, Oxford (1916). Reprinted 1971 by Chelsea.

[Courant(41)] Courant, R. and Robbins, H., What is Mathematics?, Oxford (1941).

[Dasarathy(75)] Dasarathy, B. and White, L. J., *On Some Maximin Location and Classifier Problems*, Computer Science Conference, Washington, D.C. (1975) (Unpublished lecture).

[Davis(63)] Davis, P. J., Interpolation and Approximation, Blaisdell (1963). Reprinted 1975 by Dover.

[Davis(75)] Davis, J. C. and McCullagh, M. J., Display and Analysis of Spatial Data, Wiley (1972).

[Delaunay(34)] Delaunay, B., *Sur la sphere vide*, Bull. Acad. Sci. USSR(VII), Classe Sci. Mat. Nat. (1934), 793-800.

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

[Dijkstra(59)] Dijkstra, E. W., *A note on two problems in connexion with graphs*, Numer. Math. 1,5 (Oct, 1959), 269-271.

[Dobkin(76a)] Dobkin, David and Lipton, Richard, *Multidimensional Searching Problems*, SIAM J. Comput. 5,2 (Jun, 1976), 181-186.

[Dobkin(76b)] Dobkin, David P., Lipton, Richard J. and Reiss, Steven P., *Excursions Into Geometry*, Technical Report 71. Department of Computer Science, Yale University (1976).

[Duda(73)] Duda, R. O. and Hart, P. E., Pattern Classification and Scene Analysis, Wiley-Interscience (1973).

[Eddy(77)] Eddy, W. F., *A new convex hull algorithm for planar sets*, TOMS (1977). To appear.

[Edmonds(65)] Edmonds, J., *Maximum matching and a polyhedron with 0,1 vertices*, J. Res. NBS 69B (Apr-Jun, 1965), 125-130.

[Edmundson(75)] Edmundson(75), *Definitions of random sequences*, Technical Report TR-360. Department of Computer Science, University of Maryland (Mar, 1975).

[Efron(65)] Efron, B., *The convex hull of a random set of points*, Biometrika 52 (1965), 331-343.

[Elzinga(72a)] Elzinga, J. and Hearn, D. W., *Geometrical Solutions for Some Minimax Location Problems*, Transportation Science 6 (1972), 379-394.

[Elzinga(72b)] Elzinga, D. J. and Hearn, D. W., *The Minimum Covering Sphere Problem*, Mgmt. Sci. 19,1 (Sep, 1972), 96-104.

[Erdos(46)] Erdos, P., *On sets of distances of n points*, Amer. Math. Monthly (1946), 248-250.

[Erdos(60)] Erdos, P., *On sets of distances of n points in Euclidean space*, Magy. Tud. Akad. Mat. Kut. Int. Kozl. 5 (1960), 165-169.

[Eves(72)] Eves, H., A Survey of Geometry, Allyn and Bacon (1972).

[Fary(48)] Fary, I., *On straight-line representation of planar graphs*, Acta Sci. Math. Szeged. 11 (1948), 229-233.

[Forrest(72)] Forrest, A. R., *On Coons and other methods for the representation of curved surfaces*, J. Comp. Graphics and Image Proc. 1 (1972), 341-359.

[Francis(74)] Francis, R. L. and White, J. A., Facility Layout and Location: An Analytical Approach, Prentice-Hall (1974).

[Freeman(67)] Freeman, H. and Loutrel, P. P., *An algorithm for the solution of the two-dimensional hidden-line problem*, IEEE Trans. Elec. Comp. EC-16,6 (1967).

[Friedman(72)] Friedman, N., , Technical Report . Department of Computer Science, University of Toronto (1972).

[Friedman(75)] Friedman, J. H., Baskett, F. and Shustek, L. J., *An Algorithm for Finding Nearest Neighbors*, IEEE Trans. Comp. (Oct, 1975), 1000-1006.

[Friedman(78)] Friedman, J. H. and Rafsky, L. C., *Peeling, skinning, and unwrapping multidimensional point sets*, (1978). Submitted for publication.

[Frost(17)] Frost, P., An Elementary Treatise on Curve Tracing, Fifth Edition, London (1917). Reprinted 1960 by Chelsea.

[Gabow(72)] Gabow, H., *An efficient implementation of Edmond's maximum matching algorithm*, Technical Report 31. Computer Science Department, Stanford Univ. (1972).

[Galimberti(69)] Galimberti, R. and Montanari, U., *An algorithm for hidden-line elimination*, CACM 12,4 (1969).

[Garey(76a)] Garey, M. R., Graham, R. L. and Johnson, D. S., *Some NP-complete geometric problems*, Eighth Annual ACM SIGACT Symposium (May, 1976), 10-22

[Garey(76b)] Garey, M. R., Johnson, D. S. and Stockmeyer, L., *Some simplified NP-complete graph problems*, Theor. Comp. Sci. 1 (1976), 237-267.

[Gass(69)] Gass, S. I., Linear Programming, McGRaw-HHill (1969).

[Gastwirth(66)] Gastwirth, J., *On robust procedures*, J. Amer. Stat. Assn. 65 (1966), 929-948.

[Gemignani(66)] Gemignani, Michael, *On Finite Subsets of the Plane and Simple Closed Polygonal Paths*, Math. Mag. (Jan-Feb, 1966), 38-41.

[George(71)] George, J. A., *Computer Implementation of the finite element method*, Technical Report STAN-CS-71-208. Computer Science Department, Stanford University (1971).

[Gonzalez(75)] Gonzalez, T., *Algorithms on sets and related problems*, Technical Report . Department of Computer Science, University of Oklahoma (1975).

[Gower(69)] Gower, J. C. and Ross, G. J. S., *Minimum Spanning Trees and Single Linkage Cluster Analysis*, Appl. Stat. 18,1 (1969), 54-64.

[Graham(72)] Graham, R. L., *An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set*, Info. Proc. Lett. 1 (1972), 132-133.

[Grunbaum(56)] Grunbaum, B., *A proof of Vazsonyi's conjecture*, Bull. Res. Council Israel 6A (1956), 77-78.

[Grunbaum(67)] Grunbaum, B., Convex Polytopes, Wiley-Interscience (1967).

[Hadwiger(64)] Hadwiger, H., Debrunner, H. and Klee, V., Combinatorial Geometry in the Plane, Holt, Rinehart and Winston (1964).

[Hanan(72)] Hanan, M. and Kurtzberg, J. M., *Placement techniques*, In Design Automation of Digital Systems , Breuer, M., ed. Prentice-Hall (1972).

[Hanan(75)] Hanan, M., *Layout, Interconnection, and Placement*, Networks 5 (1975), 85-88.

[Harary(71)] Harary, F., Graph Theory, Addison-Wesley (1971).

[Hardy(67)] Hardy, G. H., Littlewood, J. G. and Polya, G., Inequalities, Cambridge (1967).

[Hartigan(75)] Hartigan, J. A., Clustering Algorithms, Wiley (1975).

[Heath(21)] Heath, T. L., A History of Greek Mathematics, Oxford (1921).

[Henrici(74)] Henrici, P., Applied and Computational Complex Analysis, Wiley-Interscience (1974).

[Hilbert(99)] Hilbert, D., Foundations of Geometry, (1899). Reprinted 1971 by Open Court.

[Hocking(61)] Hocking, J. G. and Young, G. S., Topology, Addison-Wesley (1961).

[Hodder(76)] Hodder, I. and Orton, C., Spatial Analysis in Archaeology, Cambridge (1976).

[Hoel(71)] Hoel, P. G., Introduction to Mathematical Statistics, Wiley (1971).

[Hopkins(54)] Hopkins, B. and Skellam, J. G., *A new method for determining the type of distribution of plant individuals*, Ann. Bot. Lond. N. S. 18 (1954), 213-227.

[Huber(72)] Huber, P. J., *Robust statistics: a review*, Ann. Math. Stat. 43,3 (1972), 1041-1067.

[Jarvis(73)] Jarvis, R. A., *On the Identification of the Convex Hull of a Finite Set of Points in the Plane*, Info. Proc. Lett. 2 (1973), 18-21.

[Johnson(67)] Johnson, S. C., *Hierarchical clustering schemes*, Psychometrika 32 (1967), 241-254.

[Johnson(77)] Johnson, D. S. and Preparata, F. P., *The Densest Hemisphere Problem*, Technical Report R-757. Coordinated Science Laboratory, University of Illinois (1977).

[Karp(72)] Karp, R. M., *Reducibility among combinatorial problems*, in Complexity of Computer Computations , Miller, R. E. & Thatcher, J. W., ed. Plenum Press (1972).

[Karp(75)] Karp, R. M., *On the computational complexity of combinatorial problems*, Networks 5 (1975), 45-68.

[Karp(76)] Karp, R. M., *The probabilistic analysis of combinatorial search algorithms*, in Algorithms and Complexity: New Directions and Recent Results , Traub, J. F., ed. Academic Press (1976).

[Karteszi(76)] Karteszi, F., Introduction to Finite Geometries, North-Holland (1976).

[Kazarinoff(70)] Kazarinoff, N., Ruler and the Round, Prindle, Weber and Schmidt (1970).

[Kendall(63)] Kendall, M. G. and Moran, P. A. P., Geometrical Probability, Charles Griffin (1963).

[Kim(75)] Kim, C. E., *A Minimal Spanning Tree and Approximate Tours for a Traveling Salesman*, Technical Report TR-431. Department of Computer Science, University of Maryland (Dec, 1975).

[Kirchberger(03)] Kirchberger, P., *Uber Tschebyschefsche Annaherungsmethoden*, Math. Ann. 57 (1903), 509-540.

[Klee(65)] Klee, V., *A class of linear programming problems requiring a large number of iterations*, Numer. Math. 7 (1965), 313-321.

[Knuth(68)] Knuth, D. E., The Art of Computer Programming. Volume I: Fundamental Algorithms, Addison-Wesley (1968).

[Knuth(71)] Knuth, D. E., The Art of Computer Programming. Volume II: Seminumerical Algorithms, Addison-Wesley (1971).

[Knuth(73)] Knuth, D. E., The Art of Computer Programming. Volume III: Sorting and Searching, Addison-Wesley (1973).

[Knuth(76)] Knuth, D. E., *Big omicron and big omega and big theta*, SIGACT News 8,2 (Apr-Jun, 1976).

[Knuth(77)] Knuth, D.E., Morris, R. and Pratt, V., *Fast Pattern Matching in Strings*, SIAM J. Comput. (to appear).

[Kolars(74)] Kolars, J. F. and Nystuen, J. D., Human Geography, McGraw-Hill (1974).

[Kruskal(56)] Kruskal, J. B., *On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem*, Proc. AMS 7 (1956), 48-50.

[Kung(75)] Kung, H. T., Luccio, F. and Preparata, F. P., *On Finding the Maxima of a Set of Vectors*, JACM 22,4 (Oct, 1975), 469-476.

[Lakatos(76)] Lakatos, I., Proofs and Refutations: The Logic of Mathematical Discovery, Cambridge (1976).

[Lawson(65)] Lawson, C. L., *The smallest covering cone or sphere*, SIAM Rev. (1965), 415-417.

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

[Lee(76a)] Lee, D.-T., *On Finding k Nearest Neighbors in the Plane*, Technical Report . Department of Computer Science, University of Illinois (1976).

[Lee(76b)] Lee, D.-T. and Preparata, F. P., *Location of a Point in a Planar Subdivision and Its Applications*, Eighth Annual ACM SIGACT Symposium (May, 1976), 231-235

[Lee(77a)] Lee, D.-T. and Preparata, F. P., *An O(n) algorithm for the kernel of a simple polygon*, (1977). Submitted for publication.

[Lee(77b)] Lee, D.-T. and Wong, C. K., *Voronoi diagrams in L_1 (L) metrics with 2-dimensional storage applications*, (1977). Submitted for publication.

[Lemoine(07)] Lemoine, E., Geometrographie, (1907).

[Lipton(77a)] Lipton, R. J. and Tarjan, R. E., *A separator theorem for planar graphs*, Waterloo Conference on Theoretical Computer Science (Aug, 1977), 1-10

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

[Liu(68)] Liu, C. L., Introduction to Combinatorial Mathematics, McGraw-Hill (1968).

[Loberman(57)] Loberman, H. and Weinberger, A., *Formal Procedures for Connecting Terminals with a Minimum Total Wire Length*, JACM 4 (1957), 428-437.

[Loeb(76)] Loeb, A. L., Space Structures, Addison-Wesley (1976).

[Lopshits(63)] Lopshits, A. M., Computation of Areas of Oriented Figures, D. C. Heath (1963).

[Loutrel(70)] Loutrel, P. P., *A Solution to the Hidden-Line Problem for Computer-Drawn Polyhedra*, IEEE Trans. Comp. C-19,3 (Mar, 1970), 205-215.

[Manacher(76)] Manacher, G., *An application of pattern matching to a problem in geometrical complexity*, Info. Proc. Lett. 5 (1976), 6-7.

[Marden(66)] Marden, M., Geometry of Polynomials, American Mathematical Society (1966).

[Martin-Lof(68)] Martin-Lof, P., Notes on Constructive Mathematics, Almqvist and Wiksell (1968).

[Maruyama(72)] Maruyama, K., *A Study of Visual Shape Perception*, Technical Report UIUCDCS-R-72-533. Department of Computer Science, University of Illinois (1972).

[Matern(60)] Matern, B., *Spatial variation. Stochastic models and their application to some problems in forest surveys and other sampling investigations*, Medd. fran Statens Skogsforskningsinstitut 49 (1960), 1-144.

[Matsushita(69)] Matsushita, Y., *A Solution to the Hidden Line Problem*, Technical Report 335. Department of Computer Science, University of Illinois (1969).

[McLain(76)] McLain, D. H., *Two-dimensional Interpolation from Random Data*, Computer J. 19,2 (1976), 178-181.

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

[Melzak(73)] Melzak, Z. A., Companion to Concrete Mathematics, Wiley-Interscience (1973).

[Moon(67)] Moon, J. W., *Various proofs of Cayley's formula for counting trees*, In A Seminar on Graph Theory , Harary, F., ed. Holt, Reinhart and Winston (1967).

[Muller(77)] Muller, D. E. and Preparata, F. P., *Finding the Intersection of two convex polyhedra*, (1977). Unpublished manuscript.

[Nair(71)] Nair, K. P. K. and Chandrasekaran, R., *Optimal Location of a Single Service Center of Certain Types*, Nav. Res. Log. Quart. 18 (1971).

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

[Nijenhuis(75)] Nijenhuis, A. and Wilf, H. S., Combinatorial Algorithms, Academic Press (1975).

[Ore(62)] Ore, O., Theory of Graphs, American Mathematical Society (1962).

[Osteen(74)] Osteen,, R. E. and Lin, P. P., *Picture Skeletons Based on Eccentricities of Points of Minimum Spanning Trees*, SIAM J. Comput. 3,1 (Mar, 1974), 23-40.

[Papadimitriou(76)] Papadimitriou, C. H and Steiglitz, K., *Some complexity results for the traveling salesman problem*, Eighth Annual ACM SIGACT Symposium (May, 1976), 1-9

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

[Pielou(77)] Pielou, E., Mathematical Ecology, Wiley-Interscience (1977).

[Preparata(77a)] Preparata, F. P., *Steps into computational geometry*, Technical Report . Coordinated Science Laboratory, University of Illinois (1977).

[Preparata(77b)] Preparata, F. P. and Hong, S. J., *Convex hulls of finite sets of points in two and three dimensions*, CACM 2,20 (Feb, 1977), 87-93.

[Preparata(77c)] Preparata, F. P. and Muller, D. E., *Finding the intersection of a set of n half-spaces in time O(n log n)*, (1977). Unpublished manuscript.

[Preparata(77d)] Preparata, F. P., *An O(n log n) algorithm for triangulating a simple polygon*, (1977). Unpublished manuscript.

[Preparata(77e)] Preparata, F. P., *A fast on-line convex hull algorithm*, (1977). Submitted for publication.

[Prim(57)] Prim, R. C., *Shortest Connecting Networks and Some Generalizations*, BSTJ 36 (1957), 1389-1401.

[Rabin(76)] Rabin, M. O., *Probabilistic algorithms*, in Algorithms and Complexity: New Directions and Recent Results , Traub, J. F., ed. Academic Press (1976).

[Rademacher(50)] Rademacher, H. and Schoenberg, I. J., *Helly's theorems on convex domains and Tchebycheff's approximation problem*, Canad. J. Math. 2 (1950), 245-256.

[Rademacher(57)] Rademacher, H. and Toeplitz, O., The Enjoyment of Mathematics, Princeton University Press (1957).

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

[Reingold(72a)] Reingold, Edward M., *On the Optimality of Some Set Algorithms*, JACM 19,4 (Oct, 1972), 649-659.

[Reingold(72b)] Reingold, E. M. and Stocks, A. I., *The Complexity of Contouring*, (1972). Extended abstract.

[Renyi(63)] Renyi, A. and Sulanke, R., *Uber die konvexe Hulle von n zufallig gewahlten Punkten, I*, Z. Wahrschein. 2 (1963), 75-84.

[Renyi(64)] Renyi, A. and Sulanke, R., *Uber die konvexe Hulle von n zufallig gewahlten Punkten, II*, Z. Wahrschein. 3 (1964), 138-147.

[Rice(64)] Rice, J., The Approximation of Functions. Volume I: The Linear Theory, Addison-Wesley (1964).

[Riesenfeld(73)] Riesenfeld, R., *Applications of b-spline approximation to geometric problems of computer-aided design*, Technical Report UTEC-CSc-73-126. Department of Computer Science, University of Utah (1973).

[Rockafellar(70)] Rockafellar, R. T., Convex Analysis, Princeton University Press (1970).

[Rogers(64)] Rogers, C. A., Packing and Covering, Cambridge University Press (1964).

[Rogers(76)] Rogers, D. F. and Adams, J. A., Mathematical Elements for Computer Graphics, McGraw-Hill (1976).

[Rosenkrantz(74)] Rosenkrantz, D. J., Stearns, R. E. and Lewis, P. M., *Approximate algorithms for the traveling salesperson problem*, Fifteenth Annual IEEE Symposium on Switching and Automata Theory (1974)

[Saaty(70)] Saaty, T. L., Optimization in Integers and Related Extremal Problems, McGraw-Hill (1970).

[Sanders(76)] Sanders, D., *Metric Spaces in Which Minimal Circuits Cannot Self-Intersect*, Proc. AMS 36 (Apr, 1976), 383-387.

[Santalo(76)] Santalo, L. A., Integral Geometry and Geometric Probability,

Encyclopedia of Mathematics and its Applications, v. 1. Addison Wesley (1976).

[Schwartzmann(75)] Schwartzmann, Didier H. and Vidal, Jacques J., *An Algorithm for Determining the Topological Dimensionality of Point Clusters*, IEEE Trans. Comp. C-24,12 (Dec, 1975), 1175-1182.

[Shamos(75a)] Shamos, M. I., *Geometric Complexity*, Seventh Annual ACM SIGACT Conference (May, 1975), 224-233

[Shamos(75b)] Shamos, M. I., Problems in Computational Geometry, unpublished manuscript (1975).

[Shamos(75c)] Shamos, M. I. and Hoey, D., *Closest-Point Problems*, Sixteenth Annual IEEE Symposium on Foundations of Computer Science (Oct, 1975), 151-162

[Shamos(76a)] Shamos, M. I., *Geometry and Statistics: Problems at the Interface*, In Recent Results and New Directions in Algorithms and Complexity , Traub, J. F., ed. Academic Press (1976).

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

[Shamos(76c)] Shamos, M. I. and Yuval, G., *Lower Bounds from Complex Function Theory*, Seventeenth Annual IEEE Symposium on Foundations of Computer Science (Oct, 1976), 268-273

[Sklansky(72)] Sklansky, J., *Measuring Concavity on a Rectangular Mosaic*, IEEE Trans. Comp. C-21 (1972), 1355-1364.

[Sommerville(29)] Sommerville, D. M. Y., An Introduction to the Geometry of N Dimensions, Methuen (1929). Reprinted 1958 by Dover..

[Stamey(63)] Stamey, W. L. and Marr, J. M., *Union of two convex sets*, Canad. J. Math 15 (1963), 155. 156.

[Stewart(73)] Stewart, I., Galois Theory, Chapman and Hall (1973).

[Stoer(70)] Stoer, J. and Witzgall, C., Convexity and Optimization in Finite Dimensions I, Springer-Verlag (1970).

[Strang(73)] Strang, G. and Fix, G., An Analysis of the Finite Element Method, Prentice-Hall (1973).

[Sutherland(66)] Sutherland, I. E., *Ten unsolved problems in computer graphics*, Datamation 12,5 (May, 1966).

[Sylvester(69)] Sylvester, J. J., *On Poncelet's approximate linear valuation of surd forms*, Phil. Mag. XX (1869), 203-222.

[Tarjan(72)] Tarjan, R. E., *Depth-first search and linear graph algorithms*, SIAM J. Comput. 1 (1972), 146-160.

[Toregas(71)] Toregas, C., Swain, R., Revelle, C. and Bergman, L., *The Location of Emergency Service Facilities*, Operations Research 19 (1971), 1363-1373.

[Tou(76)] Tou, J. and Gonzalez, R., Pattern Recognition Principles, Addison-Wesley (1976).

[Valentine(64)] Valentine, F. A., Convex Sets, McGraw-Hill (1964).

[Voronoi(08)] Voronoi, G., *Nouvelles applications des parametres continus a la theorie des formes quadratiques. Deuxieme Memoire: Recherches sur les parallelloedres primitifs*, J. reine angew. Math. 134 (1908), 198-287.

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

[Watkins(70)] Watkins, G. S., *A real-time visible surface algorithm*, Technical Report UTECH-CSc-70-101. Computer Science Department, University of Utah (Jun, 1970).

[Winograd(70)] Winograd, S., *On the Number of Multiplications Necessary to Compute Certain Functions*, Comm. Pure and Appl. Math. 23 (1970), 165-179.

[Yaglom(61)] Yaglom, I. M. and Boltyanskii, V. G., Convex Figures, Holt, Rinehart & Winston (1961).

[Yao(75)] Yao, A. C.-C., *An O(E log log V) algorithm for finding minimum spanning trees*, Info. Proc. Lett. 4 (1975), 21-23.

[Yao(76)] Yao, A. C.-C. and Yao, F. F., *The complexity of searching an ordered random table*, Seventeenth Annual IEEE Symposium on Foundations of Computer Science (Oct, 1976), 173-177

[Zahn(71)] Zahn, Charles T., *Graph-Theoretical Methods for Detecting and Describing Gestalt Clusters*, IEEE Trans. Comp. C-20,1 (Jan, 1971), 68-86.

[Zhukhovitsky(66)] Zhukhovitsky, S. I. and Avdeyeva, L. I., Linear and Convex Programming, Saunders (1966).

[Ziezold(70)] Ziezold, H., *Uber die Eckenzahhl zufalliger konvexer Polygone*, Izv. Akad. Nauk. Armjan. SSR. Ser. Mat. 5 (1970), 296-312.

INDEX