Computational Geometry and Convexity

Bernard Chazelle

Yale University, PhD Thesis, 1980

Abstract

The purpose of this dissertation is two-fold: To assert the power of convexity as a crucial factor of efficiency in computational geometry and to show how non-convex designs can also benefit from this feature.

Most of the recent results in computational geometry have relied on the attribute of convexity, and have failed to generalize to arbitrary designs. To remedy this flaw, one general approach consists of decomposing the objects into convex pieces, then applying the procedures to each part. We study the problem of finding minimal convex decompositions in two and three dimensions. Among our major results are an $O(n+N^3)$ dynamic-programming algorithm for producing minimal decompositions of non-convex polygons and an $O(nN^3)$ heuristic for decomposing three-dimensional polyhedra. The latter procedure is worst-case optimal in the number of convex parts (within a constant multiplicative factor). In both cases, n denotes the total number of vertices, while N refers to the number of edges which exhibit reflex angles.

We further explore the problem of finding minimal decompositions in three dimensions and prove its effective decidability. We also establish an $\Omega(N^2)$ lower bound on the number of convex parts, and use this result to analyze the performance of the above heuristics.

The second purpose of this study is to show how convexity can be used for greater efficiency. We justify this claim by studying one of the most fundamental questions in computational geometry: "Do two convex objects intersect?" Note that the problem does not call for an actual computation of the intersections, which allows the possibility of sub-linear algorithms. The restriction to a simple detection rather than a complete computation is common in many applications areas where efficiency is the main concern.

We present a class of practical algorithms for detecting intersections of lines, planes, polygons, and polyhedra in two and three dimensions. Their run-times range from $O(\log n)$ for the planar cases to $O(\log^3 n)$ for detecting the intersection of two polyhedra, where n represents the total number of vertices involved.

Table of Contents

1. INTRODUCTION 3

1.1 The Nature of Our Work 3
1.2 Thesis Outline and Main Results 9

2. DECOMPOSING A POLYGON 13

2.1 Introduction 13
2.2 Definitions and Basics 15
2.2.1 The Naive Decomposition 15
2.2.2 A More Efficient Decomposition 21
2.2.3 X-decompositions 24
2.2.4 Y-patterns 32
2.3 The Polynomial-time Algorithm 42
2.3.1 Introduction 42
2.3.2 Superranges 43
2.3.3 Detecting X2-patterns 53
2.3.4 Detecting X3-patterns 53
2.3.5 Detecting X4-patterns 58
2.3.6 The Optimal Convex Decomposition Algorithm 59
2.3.6.1 Introduction 59
2.3.6.2 The Algorithm 62
2.3.6.3 Patching Y-subtrees Together (STEP 2) 64
2.3.6.4 Computing S(i,j) (STEP 3) 67
2.3.6.5 Constructing Y-subtrees (STEP 4) 67
2.3.6.6 Completing the OCD (STEP 5) 72
2.4 An OCD Algorithm Cubic in the Number of Notches 74
2.4.1 Introduction 74
2.4.2 Computing E(i,j) 77
2.4.3 The Proofs of the Lemmas Left Unresolved 83
2.4.4 The Cubic Algorithm 93
2.5 Conclusions and Open Problems 96

3. DECOMPOSING A POLYHEDRON 99

3.1 Introduction 99
3.2 The Naive Decomposition 102
3.2.1 An Exponential Blow-up 104
3.2.2 The Naive Decomposition Revisited 106
3.2.3 A More Efficient Implementation of the Naive Decomposition 119
3.3 A quadratic Lower Bound on the Number of Convex Parts 129
3.3.1 Introduction 129
3.3.2 Description of the Polyhedron P 129
3.3.3 Decomposing P into Convex Parts 130

3.3.4 The Lower Bound on the Number of Convex Parts 138
3.4 Decidability 140
3.4.1 Introduction 140
3.4.2 The OCD Problem Is Decidable 141
3.5 Conclusions 146

4. INTERSECTION OF CONVEX OBJECTS 149

4.1 Introduction 149
4.2 Computing Planar Intersections 153
4.2.1 Notation 153
4.2.2 Fibonacci Search on Bimodal Functions 153
4.2.3 Intersection of a Line with a Convex Polygon - (IGL) 155
4.2.4 Intersection of Two Convex Polygons - (IGG) 157
4.3 Detecting Three-dimensional Intersections 168
4.3.1 Introduction 168
4.3.2 Representation of Three-dimensional Objects 170
4.3.3 Intersection of a Plane with a Polyhedron - (IHP) 173
4.3.4 Intersection of a Polygon with a Polyhedron - (IHG) 174
4.3.5 Intersection of a Line with a Polyhedron - (IHL) 181
4.3.6 Intersection of Two Polyhedra - (IHH) 183
4.4 Conclusions 195

5. EPILOGUE 197

BIBLIOGRAPHY 201

Chapter 1
INTRODUCTION

1.1 The Nature of Our Work

The increasing reliance on computers for solving large problems is a pervasive phenomenon of our age. This trend naturally prevails among the most applied branches of mathematics, and in this regard, geometry is no exception. Historians of science will observe and be intrigued, however, that among the accredited disciplines of computer science, computational geometry stands as a latecomer. Although it has been fostered over the years by the development of related techniques, it was not firmly established as a field until a conscious unifying effort came to assemble the scattered pieces of the edifice into a coherent entity.

We owe this effort to M. Shamos, who gave unity to the field in a Yale PhD thesis and christened the new-born discipline "computational geometry" [Shamos,78]. His work is a large collection of geometric results lending themselves to computer applications. In its claim to breadth and simplicity, it constitutes the basics of the field and amply reflects the extent of its applications. One major contribution of Shamos' work has been to give the field legitimacy by showing how classical geometry failed to address computational issues. Thus, by proving to be practical and original, computational geometry has gained recognition as a well-founded branch of applied mathematics. Its practicality can be seen everywhere, and a quick glance at its applications reveals its intrusion into the most varied domains.

The advent of graphics terminals, epitomized by the current success of video-games, owes its existence to technological breakthroughs as well as a catalogue of geometric software. Typical tasks involved in graphics include windowing, clipping, or positioning, and often raise only very elementary geometric questions. Their profusion, however, has nurtured the field and laid the foundations.

But graphics is by no means the sole contributor to the growing attention given to computational geometry over the past decade. To a large extent, geometry is at the foundation of most arts and techniques, and very few branches of engineering are free of its spell. The range of its applications is extremely wide and eclectic as the few following examples will show.

A typical class of problems encountered in architecture, city planning, or car design involves finding possible arrangements of objects under a set of constraints, possibly expressing intersection or boundary specifications. Optimizing constrained criteria is often purely geometric when for example it involves minimizing wiring costs in network design. In other cases it may still take on a geometric setting and benefit from a geometric approach. Linear programming is a notable example.

Computers and high speed are inherently related. Thus it comes as no surprise that computational geometry often has to deal with dynamic situations. Plane simulation and computer animation are two areas among a host of others where geometrically defined events must be reported in real time. A common attribute of real-time environments is to dictate a quest for efficiency, and this will often add to the difficulty of the problems.

Aside from its practicality, computational geometry has also shed new light on the traditional approach to geometry. To grasp its originality and its departure from classical geometry, we must examine both its nature and its aims.

The need for computational geometry results from the typically non-algorithmic formulations of classical geometry. For example, the convexity of a polygon P is often defined as the property that P must contain a segment if it contains its endpoints. While it is not clear at all how this definition can be of any practical use for testing the convexity of P, restating it as the requirement that consecutive edges not form reflex angles immediately leads to effective methods.

This example may, however, be misleading if one is thereby tempted to limit computational geometry to a passive use of classical geometry. This thesis will provide numerous new results of pure geometry, fully original, yet carrying out the perennial work in the field. There is no mystery to it. All the problems addressed are new inasmuch as computer geometry opens up classical geometry to non-regular structures. Computers have to deal with arbitrary polygons rather than regular polygons, arbitrary curved lines rather than conic sections, and so forth.

Also, the task of translating classical results into algorithmic terms has raised questions related to the analysis of algorithms which would be totally irrelevant in the absence of high-speed computers. Euclid or Monge would certainly have had no grounds for worrying about the number of operations required to find the convex hull of a few thousand points. Thus, a host of new questions have been

raised, and while they undoubtedly relate to computer science, their geometric setting makes the help of classical geometry most promising. Voronoi diagrams provide a conspicuous illustration of non-trivial results of pure geometry whose applications have flowered once expressed in algorithmic terms [Shamos,78]. To summarize, while relying on the immense resources of classical geometry, computational geometry must pursue two goals:

1. Increased computer application.

2. Development of the understanding of the theoretical aspects of the questions addressed.

As one would expect, computer geometry first set about answering the most basic questions. To that end, objects under consideration have often been assumed to be convex. Unfortunately, this approach has been so strikingly exclusive that a Martian skimming through the relevant literature might falsely conclude that all things on earth are convex. Daily observations not only deny that fact, they also raise doubts about the practicality of such assumptions.

To be fair, we must acknowledge the key role played by convex structures. We can trace their importance either directly in nature or as a handy tool in many sciences.

Physical scientists commonly represent phenomena with convex models, and in higher dimensions, linear programming stands as the most notable example of mathematical problems involving convex polyhedra [Dantzig,63]. On the other hand, cartography, land planning, numerical analysis, or economics are some of the numerous fields which often rely on convexity in order to make their problems tractable.

Unfortunately, even the simplest graphics systems will constantly display non-convex objects for which standard geometric procedures have to perform efficiently (inclusion, intersection, area-covering, window-clipping being some landmarks among a host of other common tasks). A possible remedy might be to extend current algorithms to handle non-convex designs. Actually, this alternative is not always viable, especially if the algorithm relies heavily on the attribute of convexity. Then it becomes necessary to devise entirely new procedures. For example, there are logarithmic methods to test the intersection of a convex polygon with a line, whose very principles fall through when the convexity constraint is relaxed.

Yet, as it seems that convexity is so likely to guarantee efficiency, we should be reluctant blithely to dismiss it. On the contrary, we might be tempted to exploit as much convexity as can be found in our

non-convex designs. The above example will illustrate this point. This thesis will demonstrate how to test the intersection of two convex polygons in logarithmic time, while linear time is a lower bound for the non-convex case. Yet, what can be said of non-convex polygons made of two, three or p convex parts? If such decompositions were available, we could naively test the intersection of each pair of convex parts, ending up with an $O(p^2 \log N)$ time algorithm if N is the total number of vertices. Clearly, we can extend this approach to handle a large class of practical problems, hopeful that retaining the power of convexity will yield efficient algorithms.

Of course, this approach requires that, for all practical purposes, the non-convex objects dealt with can be made of relatively few convex parts. Fortunately, this assumption seems to be confirmed by experimental observations from domains as varied as graphics [Newman and Sproull,79], pattern recognition [Pavlidis,68], or tool design [Volecker,77]. For example, it is regarded virtually as a law of nature that polygons with thousands of vertices can be expected to have at most a few tens of reflex angles. (Yes, sometimes, Mother Nature can even be pleasing to computer scientists !)

This stresses the interest lying in a decomposition procedure which would capture this feature. What is needed is an efficient method for decomposing a non-convex object into a minimum number of convex parts. Unfortunately, this operation is anything but trivial. Naive enumerative procedures are doomed to fail and structural facts about the geometry of the problem must be sought. Nevertheless, even if the path to success is a long, devious one, our efforts will undoubtedly be rewarded. Indeed, success would result in the first systematic method for adapting to non-convex designs the host of fast algorithms which constitute the bulk of computer geometry to date.

Concerning the theoretical issues, we plan to achieve two goals:

1. To establish the complexity of a problem which is, at first sight, more likely to be NP-complete than polynomial.

2. To fill a major gap in the current state of computer geometry and present the first significant approach to non-convexity.

In addition to allowing for fast all-purpose algorithms, the study of the decomposition problem is challenging and interesting in its own right. Also, practically speaking, many fields would benefit greatly from a decomposition procedure. Here are some examples:

Tool designers are often obliged to achieve non-convex designs by means of convex pieces.

Typically, the convex pieces ensure the modularity of the tools and must be as few as possible [Volecker,77]. Other applications pertain to pattern recognition, where work on the decomposition problem first started in the late 60's. As a result of the linguists' need for an automatic Chinese character recognizer, engineers designed image processors to filter hand-written signals and produce complex geometric contours. The intent was to decompose the figures into convex parts to be matched against a set of patterns representing the Chinese character designs. Once again, the relative ease of dealing with convex designs as opposed to non-convex ones was the prime concern [Feng and Pavlidis,75].

At this point, we must acknowledge prior work on this problem. Only the two-dimensional case has been investigated and, to our knowledge, only approximations or simplifications of the problem have been solved. Typically, no new vertices have been allowed in the decomposition, that is, all segments must join only the vertices of the original polygon. One possible method consists of regarding the vertices of the polygon as the nuclei of growing convex cells. We grow cells simultaneously toward the interior of the polygon, and freeze borders as they happen to meet. This method has the merit of simplicity, but it does not guarantee a minimal decomposition [Schachter,78].

Other approaches have led to algorithms for solving either approximately or exactly simplifications of the problem [Feng and Pavlidis,75], [Pavlidis,68]. In each of these papers, the introduction of new points to the decomposition has been forbidden which, as we will show, rules out optimal solutions. In any event, the full problem had never been solved before this work, despite the considerable interest and consequent endeavors it has inspired.

Not only all attempts have been unsuccessful, they have also failed to bring any significant insights into the problem, and we are at a loss to acknowledge any direct contribution. One positive consequence of this gap is to make our work highly self-contained.

This thesis certainly does not claim simplicity. But it could hardly be otherwise. We believe that the decomposition problem has been open for so long because it is inherently complex. Aside from solving the problem, we also aimed at a thorough understanding of its intricacies, which also led to several simplifications of our solutions. From a broader perspective, it cannot be questioned that the interest of a difficult problem goes far beyond its own purpose, and an understanding of what makes it difficult is especially enlightening.

A second class of problems which this thesis will investigate concerns what may be regarded as the most important application of computer geometry: To determine whether a pair of convex objects intersect. This problem is well understood in a model of computation where the objects are given as input and their intersection is returned as output. For many applications, however, we may assume

that the objects already exist within the computer and that the only output desired is a single piece of data giving a common point if the objects intersect or reporting no intersection if they are disjoint. Such situations are common in graphics systems, where lots of objects are involved at the same time and the intersection of any two is to be detected. For example, when we wish to clip or window a scene, algorithms of the form given here would be sufficient for identifying those polygons which would require further processing [Newman and Sproull,79]. Similar situations are frequently encountered in real-time environments where the data are constantly updated and fast intersection procedures are required (plane simulation or air traffic control being typical examples). Other applications of intersection algorithms can be found in design rule checking for VLSI designs, trajectory checking for computer-aided design and motion control in computer animation.

In all of these applications, detecting rather than computing intersections is really sought. Although the actual computation of an intersection may require linear time, much greater efficiency can be expected from our own statement of the problem. Indeed, we will show that under the assumption that all objects are convex, this hope is largely fulfilled. Note that the convexity requirement is not that exclusive. As mentioned earlier, our decomposition procedures used in preprocessing will often restore the power of convexity to non-convex designs.

Looking more closely at the benefits of convexity, we are tempted to draw an analogy between the role played by convex objects in geometry and that played by sorted lists in data structures. As sorting is known to speed up searching, it seems that similar benefits should be expected from convexity. Surprisingly enough, this simple idea has never been exploited in a systematic fashion. At the time when most efforts in computational geometry have focused on convex objects, the deep structural facts derived from convexity have seldom been used for greater efficiency.

This thesis intends to fill this gap by presenting a comprehensive study of sublinear intersection algorithms. All possible intersections in two and three dimensions will be investigated, and although few of our algorithms have been proven to be optimal, they constitute the most efficient class of geometric algorithms to date. Their remarkable performance epitomizes the richness of convexity and illustrates the power of our convex decomposition procedures for their ability to allow non-convex designs for efficient treatments. But we also feel that it will go far beyond its illustrative purpose by opening a new direction of research. Indeed, the link between convexity and efficiency revealed here can undoubtedly be used for various other geometric problems. Such problems usually involve the testing of a predicate expressing a geometric property rather than the actual computation of a geometric figure.

An important issue that confronts us concerns the computer representation of geometric objects. In particular it is well-known that three-dimensional polyhedra can be specified under any planar graph

representation. These modes of description, however, will often be inadequate for exploiting the possibilities of binary or Fibonacci search that convex structures allow. Especially when all the data can be stored in a high-speed random-access memory, other representations will be necessary, sometimes involving preprocessing the input data.

Detecting geometric intersections is a perfect example of a general problem which can be cast under various instances and still be solved with a small number of unifying concepts. Unlike convex decompositions which call for a host of seemingly unrelated results, the intersection problems rely on a few key ideas and relate to one another in a subroutine fashion. It is once again an attribute of convexity to provide such mathematical niceties.

In conclusion, by studying the decomposition of non-convex objects into a minimum number of convex parts, we propose to construct the first bridge between the rich, powerful, alas often imaginary realm of convexity and the barren land of the non-convex world. Working in two and three dimensions, we will illustrate the fruitfulness of this approach and demonstrate the power of convexity by describing a class of fast algorithms for determining whether two objects intersect.

At the beginning of this introduction, we mused over the history of computational geometry and showed how the field was awarded recognition for being practical and original. In addition to these qualities, this thesis clearly demonstrates that the field has also depth and substance. It shows that computational geometry has in store a host of very difficult problems which can nevertheless be solved. And this is, after all, the best promise for future exciting research.

1.2 Thesis Outline and Main Results

Even minimal experience with geometry convinces one of the deceptive simplicity of some geometric proofs. Recall that Poincare was the first to write an acceptable proof of Euler's formula, after it had been fully accepted for over a hundred years on the basis of several seriously flawed "proofs" [Lakatos,76].

Thus, while we recognize the value of intuition as the source of most mathematical discoveries, we will grant no credit to "intuitive evidence" in the course of a proof. Following this principle, we have tried to present complex proofs as rigorously as necessary, while giving as much motivation and intuition as possible before diving into the thick of their intricacies. An unfortunate feature of geometry is that simple phenomena are often hard to describe. With figures and notations as our

basic tools, we have tried to illustrate complex proofs abundantly, while avoiding having this thesis turn into a comic strip (the author concedes that, anyhow, the "comic" would be in the eyes of the beholder...). We were specially concerned with notation, trying to keep it simple and suggestive yet thorough and unambiguous. We deplore the absence of standard terminology in computer geometry, and we strongly believe that this lack will have to be remedied soon, hoping to have somehow contributed to this task.

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

In Chapter 2, we study the planar case of the decomposition problem. We describe a polynomial-time algorithm for decomposing a polygon P into a minimum number of convex parts. If n and N are respectively the total number of vertices in P and the number of vertices showing a reflex angle (the notches of P), the running time of our algorithm is $O(n+N^5\log(n/N))$. Since N is most often very small, the algorithm is of practical use. However, we will describe an improved version of the algorithm, which runs in time $O(n+N^3)$. The description of this improvement is long and complex, but mainly independent of the rest of the algorithm. For this reason, we have presented it in a separate section which can be skipped by the reader without jeopardizing his/her understanding of the sequel.

Chapter 3 investigates generalizations to three dimensions. We begin by showing how a naive decomposition may produce an exponential number of convex parts. This drawback can be remedied with a few simple modifications, and we will present an $O(nN^2(N+\log n))$ time algorithm which guarantees $O(N^2)$ convex parts (n and N are respectively the number of vertices and the number of notches. In three dimensions, a notch is an edge of the polyhedron with its adjacent faces forming a reflex angle). The same algorithm can be improved further by exploiting the attributes of convexity more thoroughly. We will present a version of the algorithm which runs in time $O(nN^3)$ and we will introduce a systematic approach to convexity. Up to a constant factor, all the algorithms given are worst-case optimal in the number of convex parts. We will prove that cN^2 is a lower bound on the minimum number of convex parts. We conclude this chapter with a proof that the general problem of finding an optimal decomposition of a three-dimensional polyhedron is decidable.

Chapter 4 explores the problem of detecting whether two objects intersect. With a decomposition method at our disposal, we may assume that all objects are convex. We describe sublinear algorithms for intersecting any pair of objects from the set: line, polygon, plane, polyhedron. The running times of the algorithms range from $O(\log N)$ when the intersection involves two polygons, to $O(\log^3 N)$ when it involves two polyhedra (N being the total number of vertices). To achieve these time bounds, polyhedra must have a special representation which can be simply obtained from any standard representation in time $O(N^2)$.

We will also discuss future research in the area, open problems, and applications of the results presented in this thesis.

Chapter 2
DECOMPOSING A POLYGON

2.1 Introduction

The problem which we are studying can be simply stated as:

```
Given a simple polygon P, what is the smallest
set of pairwise disjoint convex polygons whose
union is exactly P?
```

If the interior angle formed by two consecutive edges of the polygon P is reflex, then their common vertex is called a notch. This notion is crucial since the absence of notches characterizes the convexity of a polygon. If P is non-convex, let n (resp. N) be the total number of its vertices (resp. notches). The most efficient algorithm for solving this problem which we will present requires $O(n+N^3)$ time and space. Here is an outline of the main directions followed in our analysis.

We begin by observing that each notch can be simply removed by the addition of a polygon to the decomposition. Also, we can show that at most two notches can be removed through the addition of a single polygon, from which it follows that the minimum number of convex parts always lies between $N/2 +1$ and $N+1$. However, to extend these simple observations seems a difficult mathematical problem. To form minimal decompositions, additional points must be introduced as vertices of newly generated polygons (these points are commonly referred to as Steiner points [Melzak,61], [Gilbert and Pollak,68]). This removes the obvious finiteness of the problem and makes simple enumerative procedures impossible. Furthermore, the problem cannot be treated in a local manner. These observations led to the conjecture that the problem was NP-complete.

Next we introduce X-patterns from which minimal decompositions may be generated. An Xk-

pattern is a particular interconnection of k notches which removes all reflex angles at the k notches and creates no new notches. A decomposition obtained by applying p patterns $X_{i_1},...,X_{i_p}$ along with k line segments ($k=N-(i_1+...+i_p)$) used to remove the remaining notches will yield N+1-p convex parts. It is clear that the decompositions which use the most X-patterns also minimize the number of convex polygons. This can be viewed as a generalized matching problem and seems to lend itself to a dynamic programming approach [Bellman,57]. There is exactly one type of X2-pattern and one type of X3-pattern, as shown in Figure 2.2. Since this is not the case for all k, and since the task of determining whether some given notches can be combined to form an X-pattern seems too involved, a direct use of X-patterns does not yield a polynomial-time algorithm.

Instead, we use structural facts about the decomposition to limit the types of interconnections occuring at notches, and achieve a polynomial-time dynamic programming algorithm. One crucial fact allows us to assume that, except for X4-patterns, no two Steiner points are adjacent. This leads to the introduction of a more constrained type of interconnection called a Y-pattern. We will show in Section 2.3 how to construct Y-patterns in polynomial-time and achieve our goal with a decomposition algorithm which runs in time $O(n+N^5\log(n/N))$. Finally, in Section 2.4, we will present an improvement which speeds up the algorithm to $O(n+N^3)$ time. Although it has not been proven to be optimal, the number of notches in most practical polygons is small [Pavlidis,79], and the algorithm is of practical use in its current form.

2.2 Definitions and Basics

Throughout this work, all polygons will be assumed to be simple, meaning that only consecutive edges can intersect. Let P be a simple polygon with vertices $w_1, w_2, ..., w_n$ and reflex interior angles occurring at the N notches $v_1, v_2, ..., v_N$ which form a subset of the w_i. Let B(P) denote the boundary of P. From here on, all indices of vertices (resp. notches) are taken modulo n (resp. N). Vertices will always be taken to occur in clockwise order, and the "set of points between v_i and v_j in clockwise order" will refer to the set of points visited in a clockwise traversal of the boundary of P from v_i to v_j. Angles will always be oriented, and by (ab,ac), we actually mean the angle (vector ab, vector ac). By convention, all angles will be measured counterclockwise between 0 and 360 degrees - See Figure 2.1. Finally, if S is a line segment, line(S) will denote the infinite line passing through S.

A decomposition of P is a set of polygons $P_1, ..., P_k$ whose union gives P, and such that the intersection of any two if non-null consists totally of edges and vertices. A decomposition is said to be convex if all its polygons are convex. The goal of this research being to minimize k, we define an optimal convex decomposition (OCD) of P as any convex decomposition realizing the minimum value of k.

Let v be a notch and a,b its adjacent vertices on the boundary of P (a preceding b in clockwise order). We define the range R(v) of v as the set of points u such that the segment vu lies totally in P, and neither the angle (va,vu) nor (vu,vb) is reflex. Thus, R(v) forms a sub-polygon of P - See Figure 2.3.

2.2.1 The Naive Decomposition

A simple way to obtain a convex decomposition consists of drawing a line from a notch so that it removes the reflex angle. This gives two polygons which can be decomposed by iterating on this process. We observe that the decomposition may not be optimal - See Figure 2.4.

To visualize the naive decomposition, we can imagine that we resolve the reflex angle of each notch in turn by drawing a line from the notch within its range until we first hit another line. The line hit can be either a segment of P or a line previously drawn. Let us call the whole segment thus drawn a "new" line of the decomposition. Since we can draw the line along any direction within the range of the notch, we can always manage not to hit any line at either of its endpoints. This detail is of importance for its two main consequences:

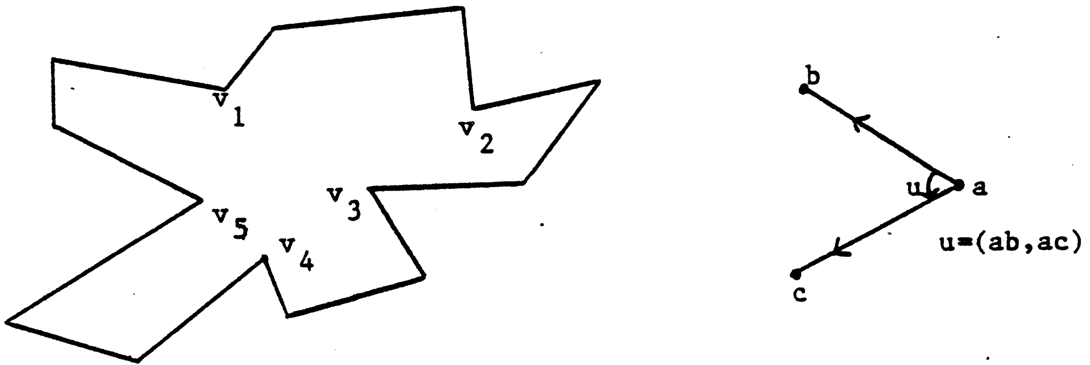


Figure 2.1: Conventions on notches and angles.

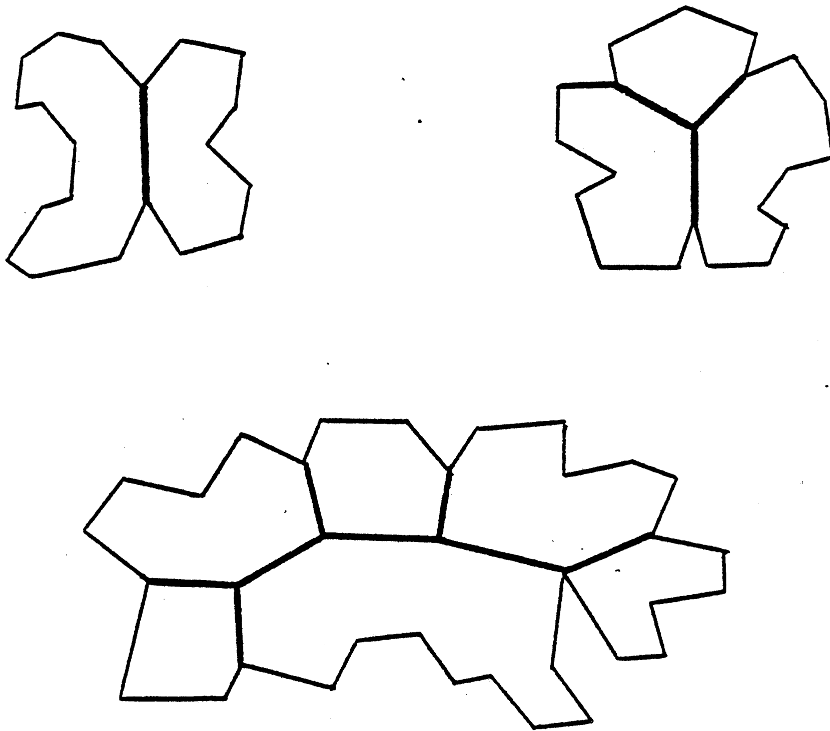


Figure 2.2: An X2-pattern, an X3-pattern, and an X6-pattern.

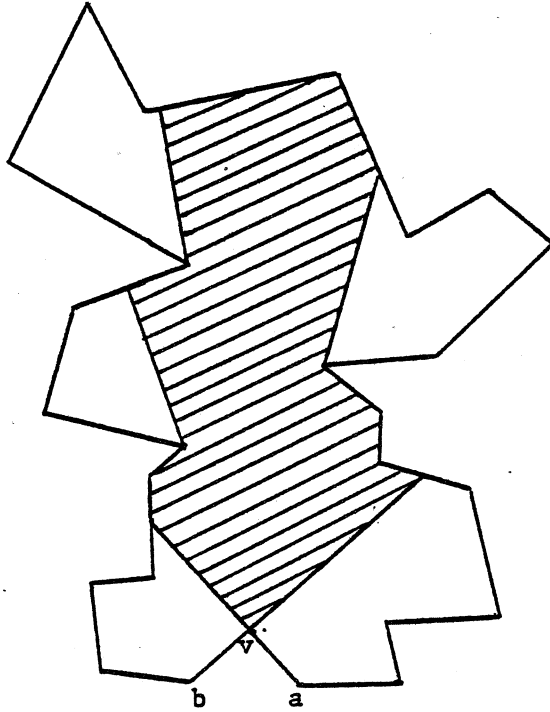


Figure 2.3: The range R(v) of a notch v.

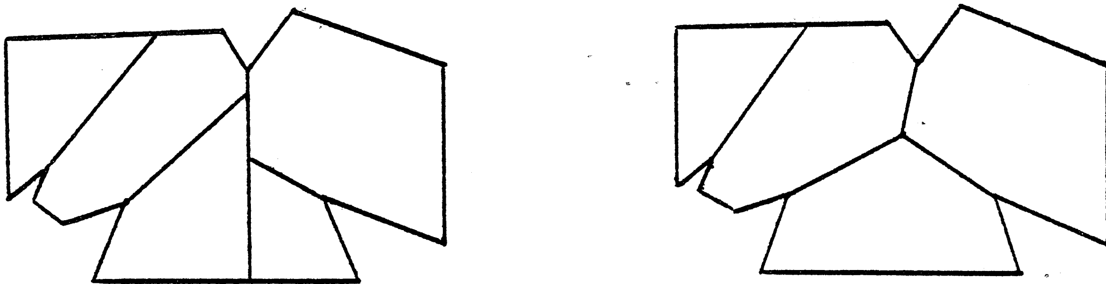


Figure 2.4: The naive decomposition and an improvement.

1. Each new line removes exactly one notch.

2. No more than 2 new lines can intersect at the same point and no pair of new lines can intersect at their endpoints.

We omit the proof which proceeds by a trivial induction on the number of notches.

It follows from 1. that exactly N new lines must be introduced, and since each new line drawn adds one polygon to the decomposition, we conclude that:

Theorem 1: The naive decomposition applied to P produces exactly $N+1$ convex parts.
The edges of P along with all the lines introduced in the decomposition form a planar graph. Note that a line of the decomposition can contribute several edges to the graph. Fact 2. implies that all the vertices of the graph have degree 2 or 3. By induction, all the convex polygons have at least one notch of P as a vertex. Also, since a notch has degree 3, each polygon has a segment of B(P) on its boundary (note that it is not necessarily an edge of B(P)). This fact shows that the subgraph of the decomposition consisting of added edges forms a forest of binary trees (an edge is said to be added if it lies on a new line). Binary trees are defined here as free trees with vertices of degree 1 or 3. The leaves of a binary tree are its vertices of degree 1 [Knuth,68].

Before proceeding with a truly algorithmic description of the naive decomposition, we must answer the basic question:

How do we represent a convex decomposition?

According to the definition, we might want to have each convex polygon represented by a list of its vertices in clockwise order (polygon representation). An alternative, however, is to regard the decomposition as a planar graph and to represent it by its adjacency lists (graph representation). In this representation, we assume that a clockwise order of the edges lying on B(P) is available. Since the algorithms we will give for the naive decomposition will produce graph representations, we need the following result:

Theorem 2: A graph representation of the naive decomposition can be used to obtain a polygon representation in time O(n).

Proof: We identify the edges of the graph with the edges of the convex polygons (Note that two consecutive edges on a polygon may be collinear). Let ab be an edge of a convex polygon Q of the decomposition with a,b in clockwise order. The next vertex c of Q in

clockwise order is the unique vertex of the graph, adjacent to b, distinct from a, such that (ba,bc) is not reflex. Since the degree of all vertices is 2 or 3, the graph representation allows us to determine c in constant time. Also, as mentioned above, each convex polygon has at least one edge which lies on B(P). Thus we can scan the edges of the decomposition graph lying on B(P) in clockwise order, computing the convex polygon adjacent to each of them. Marking the edges lying on B(P) as they are visited will avoid duplicating the polygons.

What is the running time of this method? Since no edge is visited more than twice, it is proportional to the total number of edges in the decomposition graph. This number is exactly $2N+n$ since each new line adds two to the edge count, namely itself and one from the line it intersects (which it splits in two). □

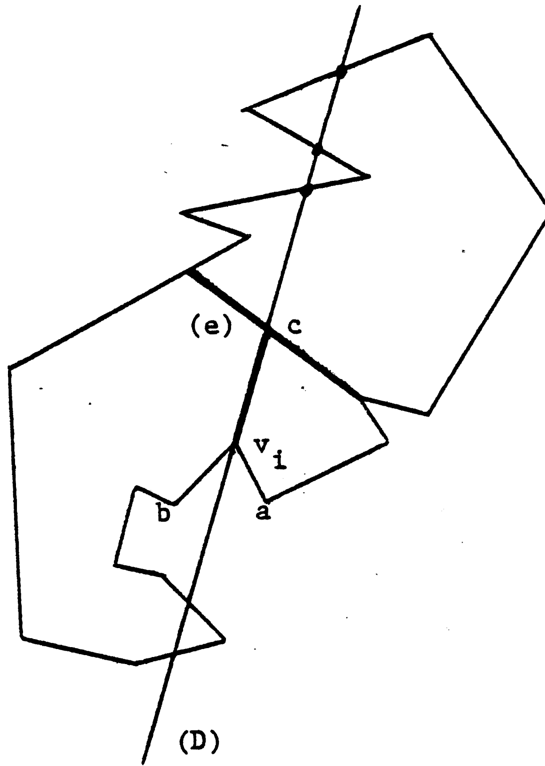
We now turn to effective ways of computing the graph representation of a naive decomposition. We compute the graph G of the decomposition in stages.

1. Initially, G is P.

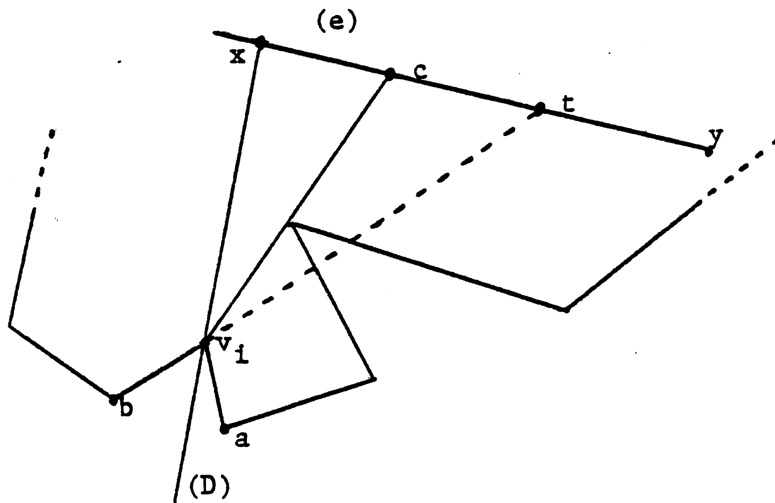
2. For each notch in turn, resolve the reflex angle by introducing a new line, updating G accordingly.

To execute Step 2, let D be the bisector of $(v_i a, v_i b)$ with a, v_i, b being consecutive vertices of P in clockwise order. We must insert into G the segment $v_i c$ defined as follows: $v_i c$ lies in both P and D, and intersects with edges of G only at v_i and c. To determine c, we may compute all the intersections $c_1,\ldots,c_p$ of D with the edges of G, such that $(v_i a, v_i c_j) < 180$. Then c is the c_j closest to v_i - See Figure 2.5-a. Let $e=(x,y)$ be the edge of G on which c lies. Assume for the moment that c is distinct from x and y. We update G by setting a new entry c with adjacent vertices: v_i,x,y. Also, we must correct the entry x (resp. y) by replacing its adjacent vertex y (resp. x) by c.

If now c is x (the case $c=y$ is similar), there is a chance that c becomes of degree 4 and we must repair this anomaly. Let t be the intersection of line(bv_i) with xy if it exists or y otherwise - See Figure 2.5-b. Any line from v_i to xt resolves the reflex angle at the notch v_i, and we only have to ensure that the line does not intersect other lines of the decomposition. To do so, consider the largest triangle $v_i xc$ (with c on xt) which contains no point of the decomposition in its interior. From the



a) Removing the notch v_i.



b) Ensuring degree 3 for c.

Figure 2.5:

convexity of the decomposition, it follows that if c is not t, the segment v_ic must pass through a notch of P. Thus we can compute c in O(N) operations by testing all the notches v_j which lie on the segment v_iz_j, where z_j is the intersection of xy with line(v_iv_j).

We have seen in the proof of Theorem 2 that G has n+2N edges, then since intersecting each bisector with the edges of G takes O(n) time and correcting the possible anomalies requires O(N) operations at each time, the naive decomposition can be carried out in O(nN) time.

2.2.2 A More Efficient Decomposition

We will next anticipate a little and present an improvement which relies on results of Chapter 4. We need to prove some preliminary facts.

Lemma 1: Let $y_1,...,y_N$ be positive integers such that $y_1+...+y_N \leq n$, then $\log y_1 + ... + \log y_N \leq N\log(n/N)$.

Proof: It is a trivial consequence of the fact that the function log is monotone increasing and concave. □

The improvement we propose for the naive algorithm involves an O(n) time preprocessing of P, which will be supposed to be applied once and for all throughout this chapter. Before describing it, we give a few definitions:

A <u>convex polygonal line</u> is a sequence of vertices $\{a_1,a_2,...,a_{p-1},a_p\}$ such that $(a_ia_{i-1},a_ia_{i+1}) \leq 180$ degrees for all i; 1<i<p. It is called a <u>convex chain</u> if $\{a_1,...,a_p,a_1\}$ forms a convex polygon. Note that $a_1,...,a_p$ corresponds to a clockwise traversal.

If we have random access to its p vertices, it will be shown in Chapter 4 how to intersect a convex chain with a line in O(log p) time, reporting the 0, 1, or 2 points of the intersection. Unfortunately, between each pair of notches (v_i,v_{i+1}), the boundary of P is certainly a convex polygonal line L_i but not necessarily a convex chain. This motivates the following preprocessing.

We partition the boundary of P between two consecutive notches into successive convex chains as follows: Let $L_i=\{y_1,...,y_p\}$ be the convex polygonal line given in clockwise order, with $y_1=v_i$ and $y_p=v_{i+1}$. If neither the angle (y_1y_k,y_1y_2) nor the angle (y_ky_{k-1},y_ky_1) is reflex for any k between 2 and p, L_i is a convex chain and remains unchanged. Otherwise, let j_1 be the smallest k such that $\{y_1,...,y_k,y_{k+1},y_1\}$ is a non-convex polygon, that is, such that either (y_1y_{k+1},y_1y_2) or $(y_{k+1}y_k,y_{k+1}y_1)$

is reflex - See Figure 2.6-a. We define C_1 as the convex chain $\{y_1,...,y_{j_1}\}$. Then we apply the same procedure recursively on the remaining part of L_i. We define C_2 as $\{y_{j_1},...,y_{j_2}\}$ with j_2 being the smallest $k>j_1$ such that $(y_{j_1}y_{k+1},y_{j_1}y_{j_1+1})$ or $(y_{k+1}y_k,y_{k+1}y_{j_1})$ is reflex. We iterate on this process until we reach y_p, thus partitioning L_i into t consecutive convex chains $C_1,...,C_t$. Repeating for each pair of notches (v_i,v_{i+1}) and renumbering the C_j, we partition the whole boundary of P into m consecutive convex chains $C_1,...,C_m$ in clockwise order.

Letting x_i,x_{i+1} be the endpoints of C_i in clockwise order (with $x_1=x_{m+1}=v_1$), we call the x_i the pseudo-notches of P. Note that a pseudo-notch may be any vertex of P. See Figure 2.6-b. Clearly, we can allow random access to the vertices of each L_i and execute the whole preprocessing in O(n) time and space. We know that $N \leq m \leq n$. It is crucial for the following, however, to show that $m=O(N)$. Actually we have :

Theorem 3: The number m of pseudo-notches in P cannot be greater than $2(1+N)$.

Proof: Consider the vertices w_i of P which are not notches of P, and let U be the sum of all the angles $(w_iw_{i+1},w_{i-1}w_i)$. Similarly, for all the vertices w_j of P which are notches, let V be the sum of all $(w_{j-1}w_j,w_jw_{j+1})$. It is a classical result of geometry that [Coxeter,61]

$$(1)\quad U\text{-}V = 360$$

Now for each convex chain $C_i=\{a_1,...,a_p\}$ such that only the vertex a_1 may be a notch of P, let a_{p+1} be the vertex of P adjacent to a_p in clockwise order. We define U_i as the sum of all angles $(a_ja_{j+1},a_{j-1}a_j)$ for $j=2,...,p$. By construction, the polygon $\{a_1,...,a_p,a_{p+1},a_1\}$ has a reflex angle either at a_{p+1} or a_1. Therefore if c (resp. d) is the angle $(a_{p+1}a_1,a_pa_{p+1})$ [resp. $(a_1a_2,a_{p+1}a_1)$] counted between -180 and +180 degrees, negative if there is a reflex angle at a_{p+1} (resp. a_1), positive otherwise, we have

$$(2)\quad U_i = 360 - (c+d) \geq 180$$

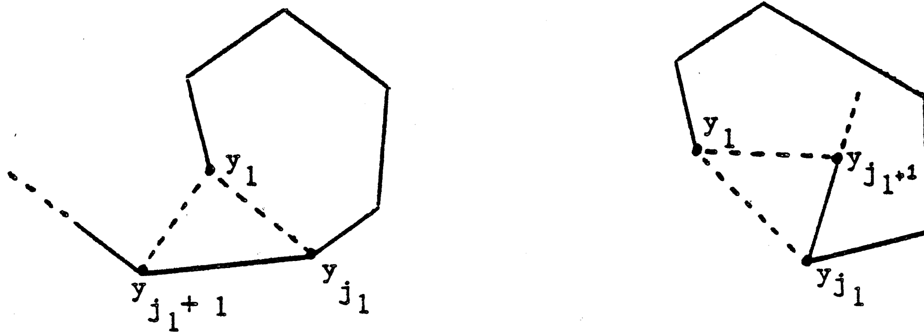
Since none of the U_i accounts for the reflex angles of P, the sum of all the U_i cannot exceed U. Also, if between a pair of consecutive notches, P consists of a single convex chain, no U_i is defined on this portion of P, whereas if it consists of p chains, p-1 U_i's are defined. Consequently, exactly m-N quantities U_i are defined. These facts, combined with (2) imply

$$180(m\text{-}N) \leq \text{Sum of all } U_i \leq U$$

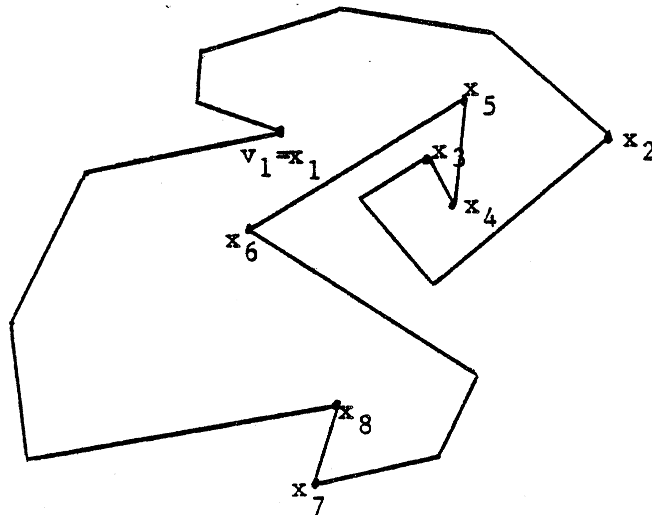
and since from (1)

$$U = 360+V \leq 180(2+N)$$

We finally have $m \leq 2(1+N)$. □



a) The two cases for defining a pseudo-notch.



b) The pseudo-notches of a polygon.

Figure 2.6

We can now state our main result:

Theorem 4: The naive decomposition of P can be done in $O(n+N^2\log(n/N))$ time and $O(n)$ space.

Proof: After preprocessing P as described above in O(n) time, we can speed up the first version of the naive decomposition as follows: To intersect D with the edges of G, we first intersect D with all the edges previously added to G, then with all the chains $C_1,...,C_m$. This will take $O(2N+\log k_1+...+\log k_m)$ time, with k_i being the number of edges in C_i. Since $m=O(N)$ and $k_1+...+k_m\leq n$, Lemma 1 shows that this running time is bounded by $O(N\log(n/N))$. □

2.2.3 X-decompositions

We begin the discussion of OCD's by defining a type of decomposition to be used in our analysis. A polygon is said to be interior to P if it lies inside P, and at most a finite number of its points lie on the boundary of P. Also, we naturally call the number of segments emanating from a vertex in the decomposition the degree of that vertex.

Definition 2.1: An X-decomposition is any convex decomposition containing no interior polygon, and such that no vertex is of degree greater than 3, except for the notches which may be of degree at most 4.

In order to show that there always exists an OCD which is also an X-decomposition, we need some preliminary results.

Lemma 2: If a reflex angle u is subdivided into p non-reflex angles u_i ($u=u_1+...+u_p$ and $0<u_1,...,u_p\leq 180$), such that $u_1+u_2, u_2+u_3, ..., u_{p-1}+u_p>180$, then p cannot be greater than 3.

Proof: It is trivial and is omitted. □

Next we turn to a question regarding graph embeddings. Let G be a connected planar graph embedded inside P, such that the graph H formed by G and P does not have any vertices of degree 1. Let $a_1,...,a_p$ denote the vertices of G encountered in a clockwise traversal of B(P) - See Figure 2.7-a. G divides the boundary of P into p pieces $b_1,...,b_p$, where b_i is the portion of B(P) between a_i and a_{i+1}. Let F_i be the face of H adjacent to b_i and lying in P. We have the following.

Lemma 3: All the faces F_i are distinct.

Proof: F_i is a polygon lying inside P which contains b_i on its boundary. Suppose that $F_i = F_j$. Then F_i contains both b_i and b_j on its boundary, and it is possible to draw a curved line L in F_i which connects b_i and b_j without intersecting any edge of G - See Figure 2.7-b. L partitions P into two regions which both contain vertices of G. Since no edge of G intersects L, G cannot then be connected. This contradicts our assumption and completes the proof. ☐

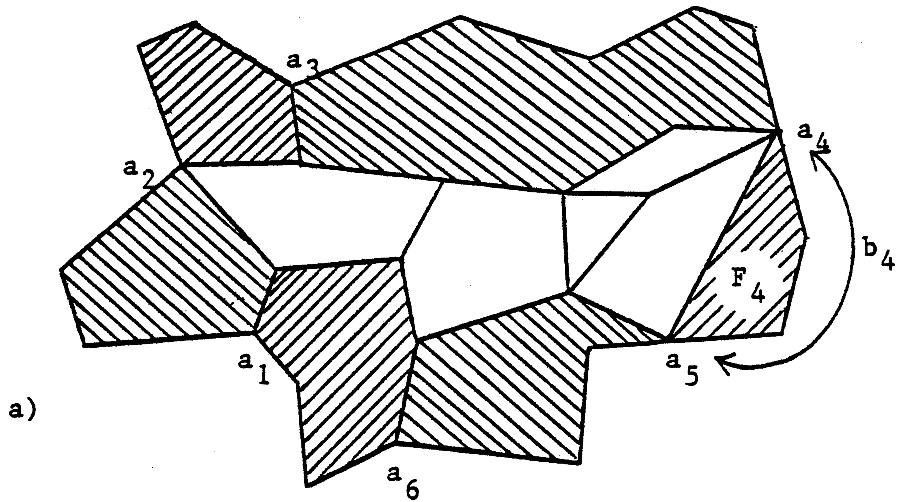
We are now in a position to prove our earlier claim.

Theorem 5: Any OCD can be transformed into an OCD which is also an X-decomposition.

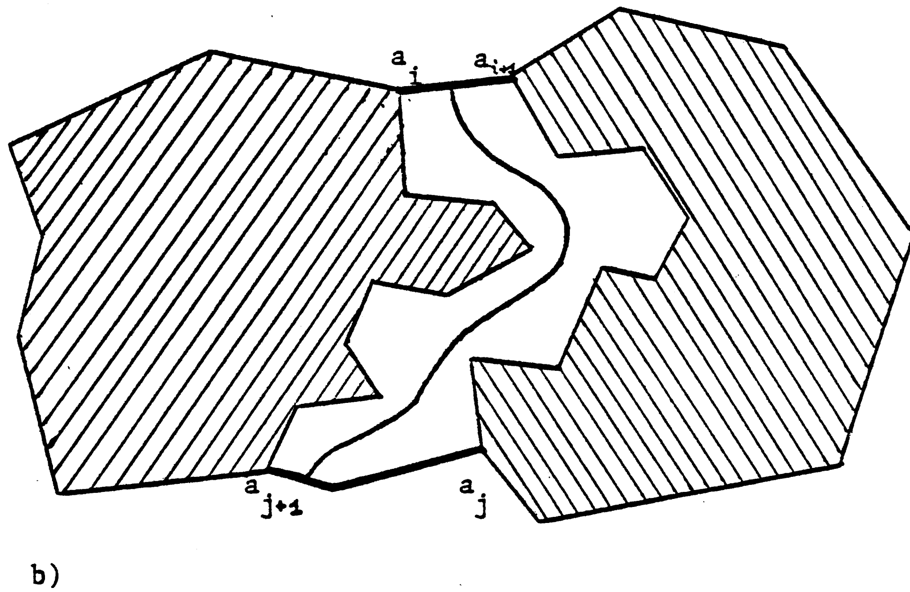
Proof: Consider an OCD of P which is not an X-decomposition. We apply geometric transformations on its edges to make it into an X-decomposition, the result of each of these transformations being always an OCD. We first show how to satisfy the degree requirements. This process may introduce interior polygons, but we next describe a procedure to remove all interior polygons without increasing the degree of any vertex.

1) Regarding the decomposition as a graph consisting of added edges and edges lying on B(P), we can always assume that only the vertices of P may be of degree 2. Let xy be an added edge. Since from the convexity of the decomposition, y is at least of degree 3, xy can be rotated slightly around y without making any angle around y reflex. This is the crux for reducing the degree of vertices. Indeed, let x be a notch of degree greater than 4 and $y_1,...,y_k$ (k>4) its adjacent vertices, with y_1 and y_2 belonging to the boundary of P. From Lemma 2 we know that there exists some i distinct from 1 or 2 such that (xy_{i+1},xy_{i-1}) is not reflex - See Figure 2.8-a. Then we can move xy_i along xy_{i-1} or xy_{i+1} to form a new segment $x'y_i$ with x' chosen close enough to x so as to preserve convexity. We iterate on this process until the notch x becomes of degree 4.

Suppose now that x is not a notch but still lies on the boundary of P (it may or may not be a vertex of P). If x is of degree greater than 3, let $y_1,...,y_k$ (k>3) be its adjacent vertices with y_1,y_2 on the boundary of P. Since (xy_1,xy_2) is not reflex, we can apply the same

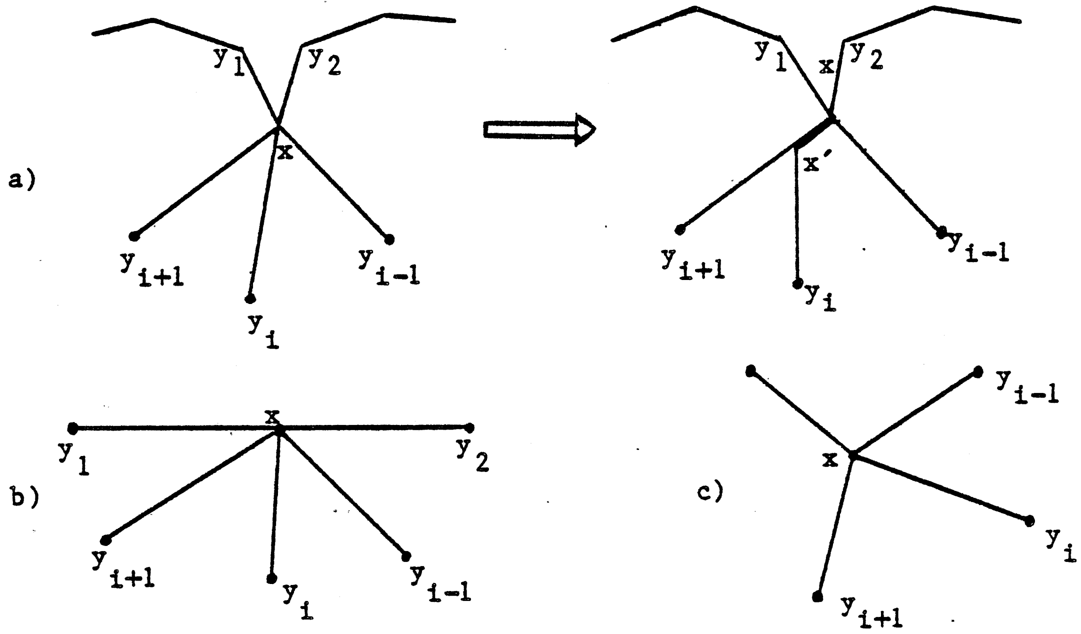


a)



b)

Figure 2.7: Counting faces in P.



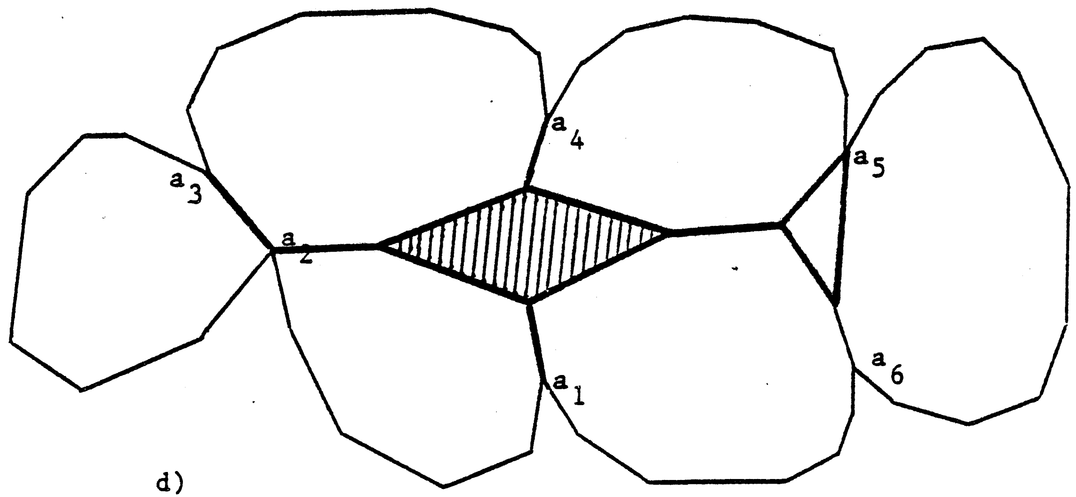


Figure 2.8: a,b,c) Satisfying degree requirements for X-decompositions.
d) Removing interior polygons.

technique as above on any edge xy_i (i distinct from 1,2) until x is made of degree 3 - See Figure 2.8-b. Finally, if x is a vertex of degree greater than 3 which does not lie on the boundary of P, all its adjacent edges are added edges, say $xy_1,...,xy_k$ (k>3). Applying Lemma 2 for u=360 degrees establishes the existence of an edge xy_i such that (xy_{i+1},xy_{i-1}) is not reflex - See Figure 2.8-c. The same method above will still reduce the degree of x to 3. As mentioned earlier, these operations may introduce interior polygons. However, the procedure we next describe will remove all of them.

2) Consider the subgraph H of the decomposition consisting of the added edges. We pick an interior polygon of the OCD and let G be the connected component in H to which the edges of this polygon belong - See Figure 2.8-d. Note that an interior polygon of the OCD is a face surrounded by a cycle in H. Let $a_1,...,a_k$ denote the vertices of G (in clockwise order) which lie on the boundary of P. Let K be the graph obtained by removing G from the graph of the OCD. Since G lies in P and is a connected component in H, it lies entirely in one face of K which we denote Q. We observe that all the a_i's lie on the boundary of Q. Also, since G is connected, Lemma 3 shows that the faces of the OCD adjacent to each portion of B(Q) between a_i and a_{i+1} and contained in Q are all distinct. Therefore there are at least k of them, and since G also contains the face corresponding to the interior polygon, the OCD has at least k+1 faces in Q. The polygon Q may not be convex, but since we had a convex decomposition of P before removing G, all the notches of Q must be notches of P, that is, some of the a_i's. Now, instead of keeping the decomposition of Q induced by the OCD, we apply the naive decomposition to it, which will introduce at most k+1 polygons. Since the boundary of each of these polygons must contain some segment on B(P) adjacent to a notch a_i, none of them can be interior to P. Repeating for all remaining interior polygons eliminates them and completes the proof.

□

Once again, we regard the added edges of an X-decomposition as forming a subgraph, and more precisely, a forest of trees, since there is no interior polygon in the decomposition. After removing all the edges of the decomposition which lie on B(P), the vertices of the forest are of degree 1 or 3, except for those vertices which are located at the notches of P and which may be of degree 1 or 2. We will

pay special attention to those trees where all the vertices of degree 1 or 2 are notches of P, and which we call X-patterns. X-patterns are defined more generally as follows:

Definition 2.2: A planar embedding of a tree lying in P is called an X-pattern if:

1. All vertices are of degree 1, 2, or 3.

2. The vertices of degree 1 or 2 coincide with notches of P, and the adjacent edges resolve the reflex angle at the notch.

3. None of the 3 angles around any vertex of degree 3 is reflex.

An X-pattern with k vertices of degree 1 or 2 is called an Xk-pattern. Vertices of degree 1,2,3 are respectively called N1-, N2-, N3-nodes. For simplicity, we refer to the vertices of degree 1 or 2 as the notches of the X-pattern. Informally, an Xk-pattern is an interconnection of k notches used to remove them, while introducing k-1 additional polygons to the decomposition - See Figure 2.2. An X-decomposition is said to have p X-patterns if the forest of trees which it forms contains p X-patterns. X-patterns are of great interest for us because of the following.

Theorem 6: An X-decomposition with p X-patterns consists of at least $N+1-p$ convex polygons.

Proof: Let S,t,k be respectively the number of polygons, trees, and vertices of the trees lying on the boundary of P. It is easy to prove the relation

$$(1)\ S = k-t+1$$

by induction on t. The case $t=1$ being trivial, assume that the introduction of t-1 trees involves k_1 vertices on B(P) and creates $S_1=k_1-(t-1)+1$ polygons. Introducing the last tree into the decomposition will account for exactly $k-k_1-1$ additional polygons, leading to a total of $S=S_1+k-k_1-1=k-t+1$ polygons and proving (1).

Each of the t-p trees which are not X-patterns has at least one vertex lying on B(P) which is not a notch. Therefore,

$$(2)\ t-p \leq k-N$$

Combining (1) and (2) completes the proof. □

Actually, we can show that any X-decomposition with p X-patterns can always be transformed into

an X-decomposition with exactly N+1-p polygons. Remove all the trees of the decomposition which are not X-patterns. Relation (1) in the proof of theorem 6 shows that this leaves N_1+1-p polygons, where N_1 is the total number of notches involved in the p X-patterns. Now, resolve the reflex angle at the remaining notches by applying the naive decomposition to these N_1+1-p polygons. This guarantees an X-decomposition and adds N-N_1 polygons, thus leading to a total of N+1-p convex polygons . See Figure 2.9 for an X-decomposition involving an X3-pattern and an X4-pattern. In general, a set of X-patterns is called <u>compatible</u> if no pair of edges taken from two distinct patterns intersect. For example, the X-patterns of an X-decomposition are always compatible. The previous results show that if an OCD has p X-patterns, any set of p compatible X-patterns will lead to an optimal decomposition through the following procedure:

1. Apply the p X-patterns.

2. Remove the remaining notches with the naive decomposition (to "remove" means here "to resolve the reflex angle at").

Note that those p X-patterns do not have to be trees of an X-decomposition. It may happen that applying the naive decomposition to finish off the work will add edges to the X-patterns, and as a result, the former X-patterns will become subtrees of the final decomposition. Of course, the naive decomposition may transform the original X-patterns but cannot add or remove X-patterns. All the previous results can be summarized in the following:

Theorem 7: Let p be the maximum number of compatible X-patterns in P. An OCD of P has exactly N+1-p convex polygons, and can be obtained by applying p compatible X-patterns and removing the remaining notches with the naive decomposition.

Comparing the results of Theorem 1 and 7, we observe that the effect of each X-pattern is to save one polygon over the naive decomposition. This remark permits us to establish bounds on the minimum number of convex parts in a decomposition.

Theorem 8: For any polygon P, an optimal convex decomposition consists of at least $1+\lceil N/2 \rceil$ convex parts and at most N+1.

Proof: It is a direct consequence of the necessity for an X-pattern to involve at least two notches. □

Before closing this section, we will establish a general result on the topology of X-patterns, which will be used often later on. We call a segment xy a <u>divider</u> of P if it lies in P and if x and y are the

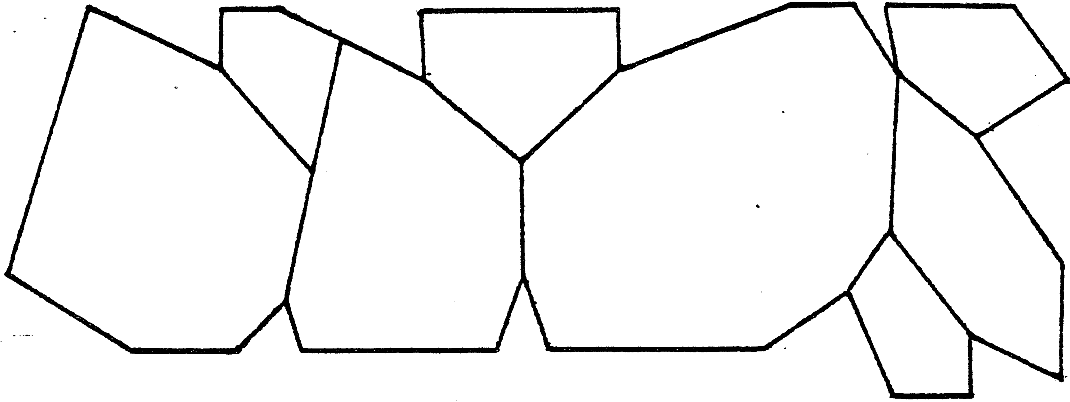


Figure 2.9: An X-decomposition involving an X3- and an X4-pattern.

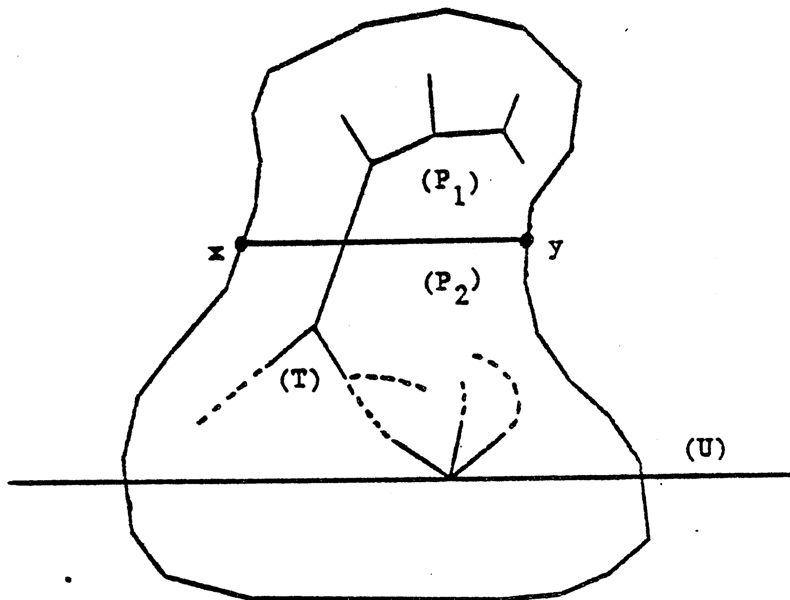


Figure 2.10: Intersection of an X-pattern with a divider of P.

only intersections of xy with the boundary of P. It is clear that a divider partitions P into two polygons P1 and P2.

Lemma 4: If an X-pattern intersects a divider xy, it must contain notches in both P1 and P2.

Proof: Suppose that the X-pattern has all its notches in P1. Let T be the portion of the X-pattern lying in P2. Since the X-pattern intersects xy, T is not empty. In general, T consists of a set of disconnected trees. Let U be the line parallel to xy which intersects T at a maximum distance to line(xy) - See Figure 2.10. All of T lies between U and line(xy). Since T does not contain any notches, the intersection of U and T must contain at least one N3-node of the X-pattern. This N3-node must then exhibit reflex angles, which leads to a contradiction. □

2.2.4 Y-patterns

Unfortunately, to determine if k given notches can be combined to form an Xk-pattern seems very difficult and makes the existence of an efficient algorithm based on these patterns extremely unlikely. To remedy this flaw, we extend our work to Y-patterns which allow for such an algorithm. A Yk-pattern is essentially an Xk-pattern where no two Steiner points (i.e., vertices outside of B(P)) are adjacent. For this reason, Y-patterns are likely to be easier to construct than X-patterns.

Definition 2.3: A Yk-pattern is an Xk-pattern made up of vertices of type N1,N2, and N3 (Fig. 2.11-a) such that:

1. No edge joins two nodes of type N3.

2. In any path containing 3 consecutive nodes of respective type N2,N3,N2, the N2-nodes lie on opposite sides (i.e., the two pairs of edges of P which emanate from the N2-nodes lie on opposite sides of the path).

Note that the N3-nodes of a Y-pattern are its Steiner points. A Y7-pattern and its representation are

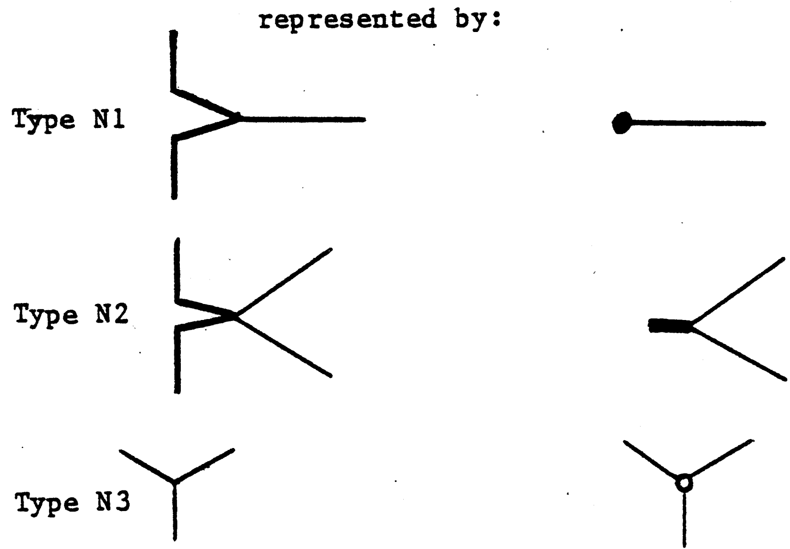
given in Figure 2.11-b. To understand condition 2, we note that, for instance, if the two N2-nodes were pointing downward, we would not have a Y-pattern.

Of course, Y-patterns are worth considering only if they can be effectively used in X-decompositions. Indeed, we can show that, except for X4-patterns, all Xk-patterns can be transformed geometrically into Y-patterns without increasing the number of convex polygons, and without even affecting the other patterns in the decomposition. These transformations, called reductions, involve stretching, shrinking, or rotating the lines of the original pattern.

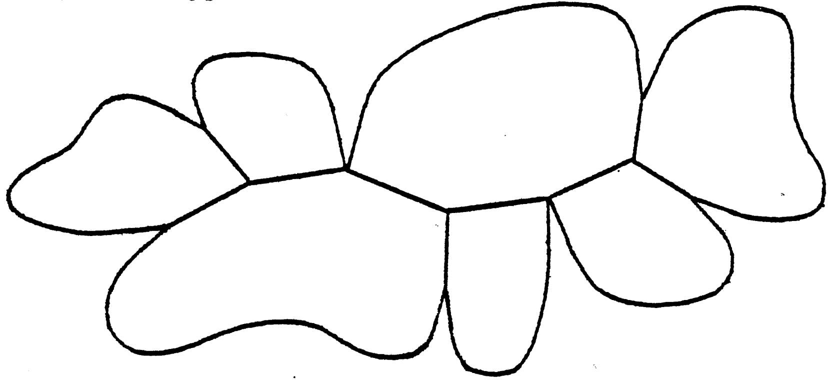
All the transformations simply involve moving N3-nodes. They can be best visualized by imagining an X-pattern as a mechanical system of levers. Levers can be shrunk, stretched, but must remain constantly straight. Furthermore, no angle is allowed to become reflex. At all times, the tree which is being transformed remains an X-pattern. However, the X-pattern may gain or lose vertices in the process. Figure 2.12 shows the reduction of an X3-pattern in an OCD. Note that the final tree can be considered either as an X3-pattern or as an X2-pattern between a and b, with the edge from c coming from the naive decomposition. In the latter case, we consider that the pattern "loses" two vertices (one N1 and one N2-node). Since we obtain an OCD by maximizing the number of X-patterns, in this case, the knowledge of the X2-pattern between a and b is sufficient to produce an OCD. Note however that this X2-pattern cannot form a tree in any OCD but only a subtree.

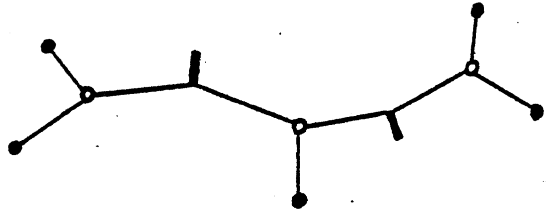
Theorem 9: In an X-decomposition, any X-pattern which is not geometrically reducible to an X4-pattern can be reduced to a Y-pattern. (This Y-pattern may be a subtree in the X-decomposition.)

Proof: As we said earlier, the method involves applying continuous transformations (reductions) to the edges of an X-pattern, while preserving the convexity of all polygons at all times. Before proceeding, we have to ensure that reductions can be carried out freely without merging two patterns in the process, thus possibly increasing the number of polygons in the decomposition. To see that, consider an X-pattern T. By definition, no line of T can intersect any other line which does not belong to T or P. We claim that any reduction of T will preserve this property. Suppose that in the course of a reduction, a point of T gets to intersect with an edge L not in T, then this point can always be assumed to be a vertex of T, therefore, 3 edges of T emanate from it and all lie on the same side of L, thus exhibiting a reflex angle between adjacent edges in the pattern, and showing that the reduction was illegal. See the example of Figure 2.13. Now, we can focus on our prime goal:



a) Node type.





b) a Y7-pattern and its representation.

Figure 2.11

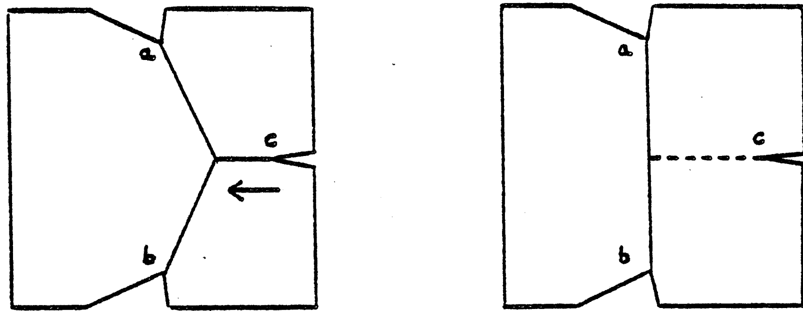


Figure 2.12: Reductions on X-patterns.

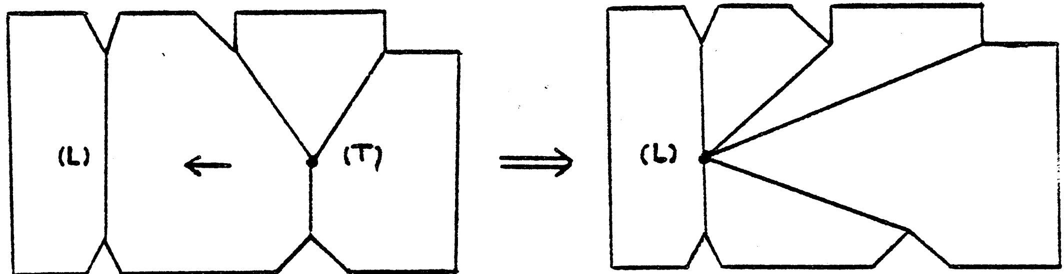


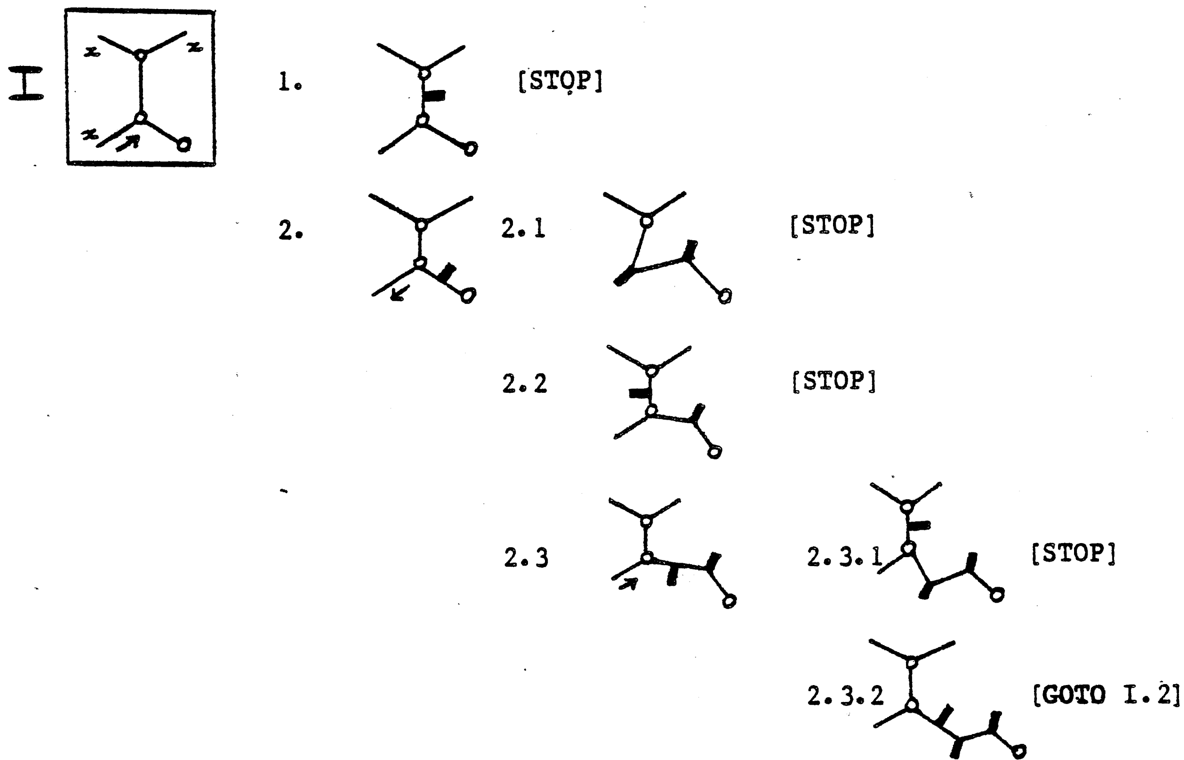
Figure 2.13: An illegal reduction.

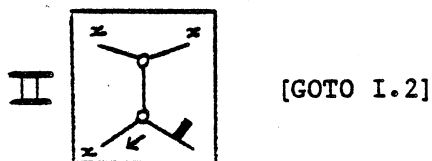
Given any X-pattern not reducible to X4, we show that we can reduce it to a tree satisfying successively conditions 1 and 2 in the definition of Y-patterns.

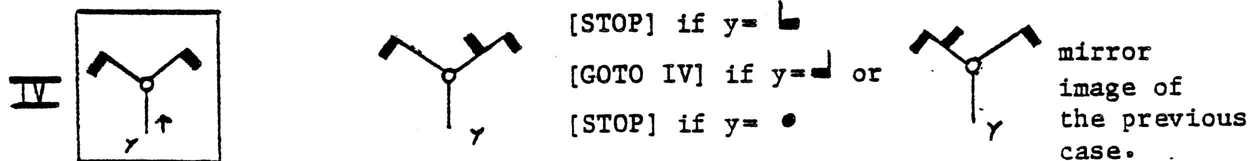
1) Figures 2.14-I,-II,-III show the 3 possible cases where two N3-nodes are adjacent and indicate the corresponding sequence of reductions to apply. Explaining the details of the first case should be sufficient. We move one of the N3-nodes in the direction indicated by the arrow. This node is the only vertex of the X-pattern to move, which in turn causes the motion of exactly 3 edges. Either we get to 1) and we are finished, or we get to 2) and another reduction leads to 2.1),2.2), or 2.3). We then iterate on this process as indicated in the figure. Convergence is guaranteed since each reduction adds a different N2-node. We note that the figure investigates all cases except those representing extreme instances of X-patterns, namely, the cases illustrated in Figure 2.15 (knowing that case 2 represents 2 edges emanating from the same notch, one of which lies in the range of the notch, the figure should be self-explanatory). To handle these 4 cases, it suffices to note that, in each of them, we can prune one edge from the pattern along with the adjoining subtree and still preserve the non-reflexivity of all the angles (see Lemma 2). We then iterate on the same procedure decribed above. This operation removes at least one N1-node from the pattern, but since case 2 of Figure 2.15 introduces a new N1-node, convergence is not obviously guaranteed.

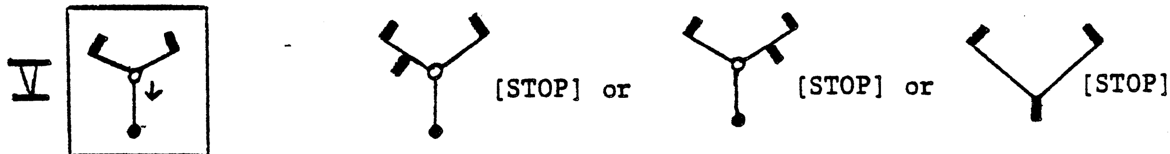
To see that the process will always converge, we only have to show that when in case 2 the subtree pruned involves a single notch, this notch can never be reintroduced by subsequent reductions. Figure 2.16-a represents this situation with the dashed line being pruned. The only way to reintroduce x into the pattern (or to actually introduce any notch between x and y in clockwise order) is to make it a N2-node first. This involves at least another notch between x and y and is therefore impossible (for a more formal proof, see Remark below).

Figures 2.14-II,-III handle all the remaining cases in a similar fashion. This procedure is to be applied as long as two N3-nodes are adjacent in the pattern. Figures 2.14-I,-II,-III)









x = node of any type.
y = node of type N1 or N2.

Figure 2.14: The proof of Theorem 9.

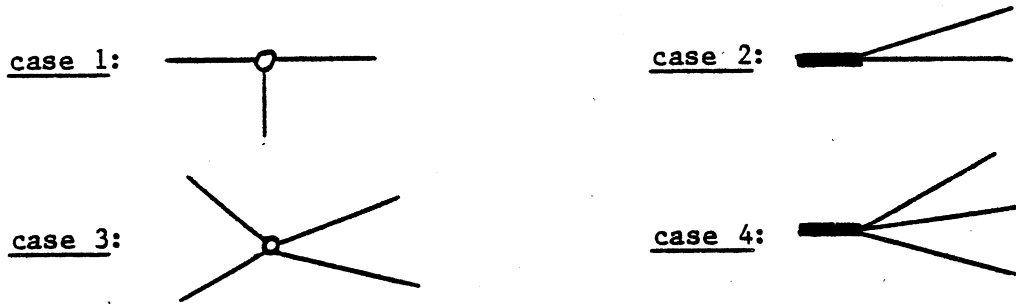


Figure 2.15: Extreme instances of X-pattern vertices.

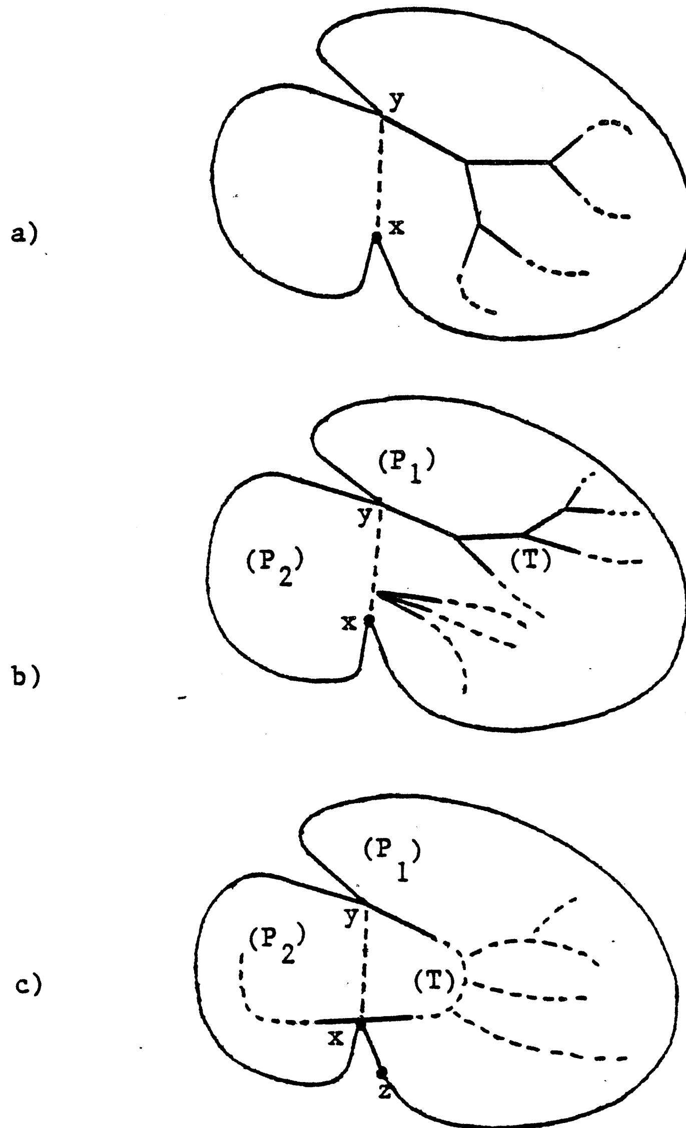


Figure 2.16: Proving that reductions always converge.

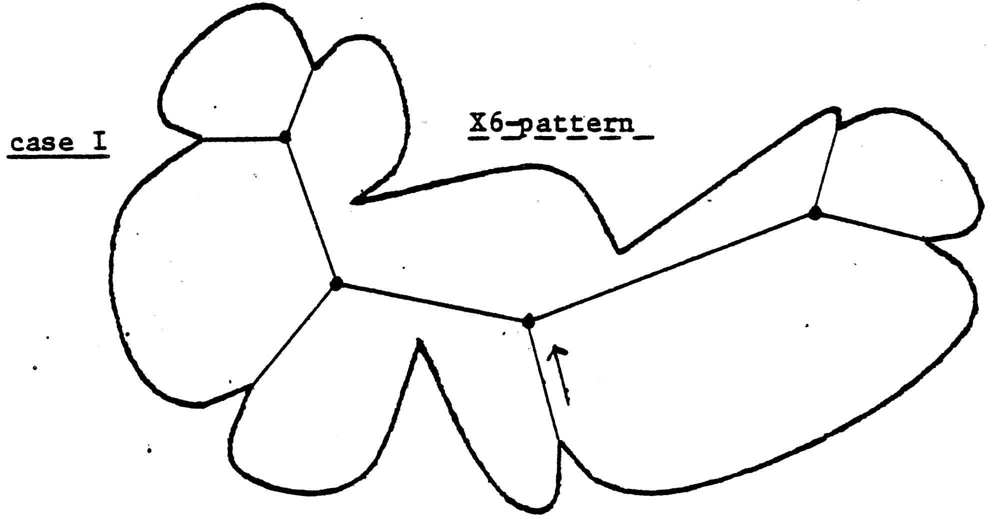
cover all cases since the pattern cannot be reduced to an X4. Each application of the procedure decreases the number of these "Steiner" edges by at least one, thus ensuring convergence. See Figure 2.17 for an example of reductions turning an X6-pattern into a Y7-pattern. Note that X-patterns "gain" vertices in the transformations of Figure 2.14 and "lose" vertices in the pruning of the extreme instances of Figure 2.15.

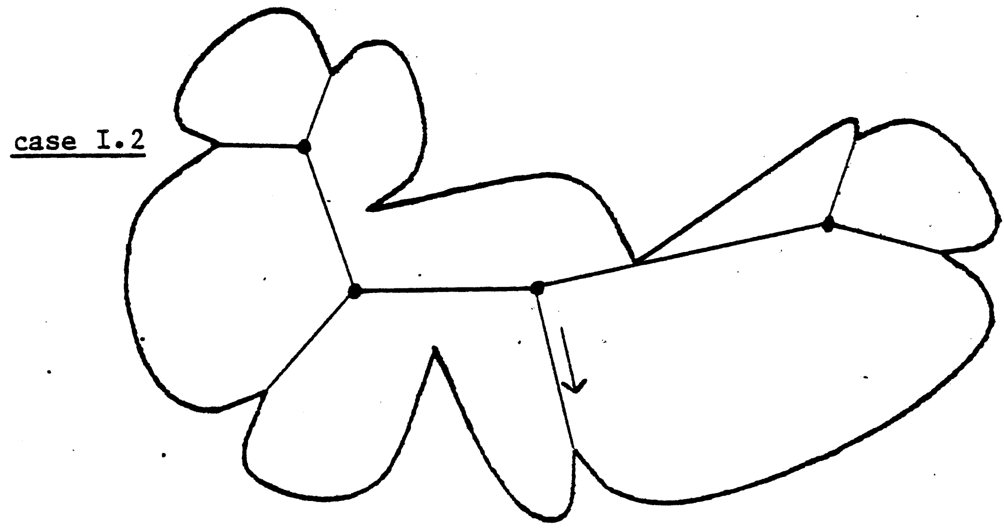
2) Finally, once condition 1 holds, we satisfy condition 2) by treating the two possible cases IV),V) as indicated. Convergence is guaranteed for the same reasons as above. Note that if we fall into one of the 4 cases of Figure 2.15, chopping one edge off will automatically make condition 2) hold locally. □

This theorem sets the stage for our algorithm.

Remark:

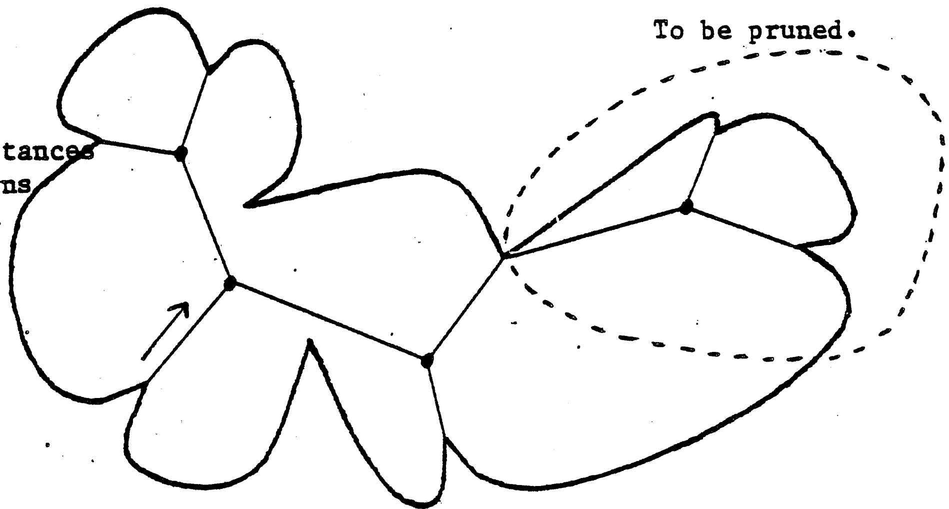
The edge xy partitions P into two polygons P1 and P2, one of which (say P1) contains the X-pattern T under consideration. We begin by observing that T will always lie entirely in P1 after any series of reductions. Indeed, the first contact of T with P2 would occur with an N3-node of T lying on the segment xy and its 3 adjacent edges lying in P1 - See Figure 2.16-b. Thus T would exhibit a reflex angle, which is impossible. We can now prove our claim. If the notch x is to be reintroduced into T through subsequent reductions, it must first become an N2-node of T with the 2 adjacent edges being collinear. This is a consequence of the reductions of Figure 2.14. Since x was originally an N1-node of T, the angle (xz,xy) is non-reflex - See Figure 2.16-c. This implies that T must lie partly in P2, which contradicts our remark above. □



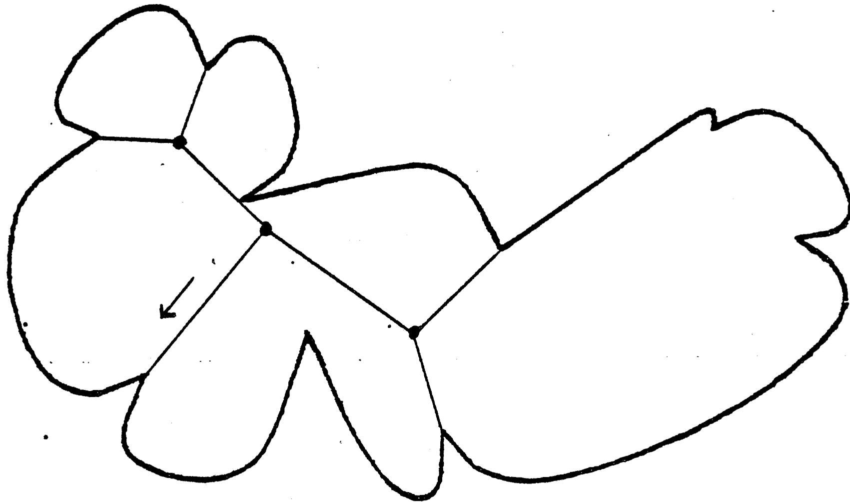


(Figure 2.17 .../...)

case 2 of
extreme instances
of X-patterns
Prune, then
fall into
case I.



case I.1
then
case II.



case I.2.2

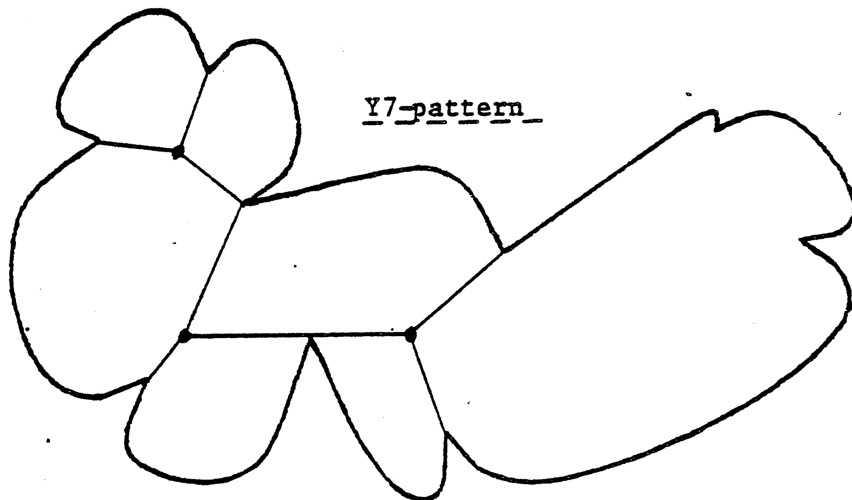


Figure 2.17: Example of reductions turning an X6-pattern into a Y7-pattern.

2.3 The Polynomial-time Algorithm

2.3.1 Introduction

From the results of the previous section, we observe that an optimal decomposition can be based on Y-patterns and X4-patterns. This reduces the problem of computing an OCD to that of determining a maximum set of compatible patterns, where patterns designate here X4- or Y-patterns. Before proceeding with the description of a polynomial-time algorithm for producing the graph of an OCD, we must answer a basic question previously raised in the discussion of the naive decomposition: How hard is it to derive a polygon representation of an OCD from its graph representation?

Theorem 10: The graph of an OCD can be used to obtain a polygon representation of the OCD in time O(n).

Proof: From the subgraph of an OCD consisting of the added edges, we can obtain the full graph of the decomposition in time O(n): Simply add the vertices of P and update the adjacencies of the leaves of the graph. Then, since an X-decomposition is a forest of binary trees just as well as the naive decomposition, the proof of Theorem 2 is still valid.

□

One difficult problem is that patterns are not given for free. They must be computed and questions of the kind: " Does there exist a pattern connecting k given notches?" must be answered. In order to preserve the flow of the presentation, we have chosen to postpone the description of the decomposition algorithm and even any mention to it until we have solved a certain number of purely geometric problems. These problems will arise constantly later on and methods for solving them will be used as subroutines by the main algorithm. If those subroutines are necessary to the algorithm, their description is not essential to its understanding, however, and we prefer to present them separately.

The type of problems which we will tackle concerns the possibility of an X-pattern between given notches. We will successively solve this problem for X2-, X3-, and X4-patterns (Sections 2.3.3,4,5). Although only X4- and Y-patterns will be eventually considered in computing OCD's, methods for detecting X2- and X3-patterns will be used for constructing Y-patterns by patching two or three "Y-subtrees" together. We will explain these operations in detail at the end of Section 2.3.4.

The problem of detecting the possibility of an X3-pattern between three given notches is essentially equivalent to that of finding a point of P visible from the three notches so that the three edges to the notches do not exhibit reflex angles. This problem can be easily solved if O(n) time is allowed. However, our claim to achieve an $O(n+N^3)$ time decomposition algorithm rules out such trivial procedures.

Instead, we will present a preprocessing which computes a description of the region of P visible from each notch. This preprocessing takes O(Nlog n) time for each notch. This may be surprising since the visible region may involve on the order of n vertices. The crux is that only significant vertices (e.g, notches) of the visible region have to be computed for our purposes. This economical description is called the superrange of the notch. We will devote Section 2.3.2 to describing superranges and showing efficient ways of computing them. The algorithm which we will present is fairly involved, but the reader can skip its description without jeopardizing his/her understanding of the decomposition algorithm.

Finally, we will present the optimal convex decomposition algorithm in Section 2.3.6.

2.3.2 Superranges

Let v be a notch of P and $t_1,...,t_p$ be the list in clockwise order of all the notches visible from v, that is, such that vt_i lies totally in P. Note that scanning $t_1,...,t_p$ corresponds to a clockwise traversal on the boundary of P but also a clockwise sweep around v. This simple remark will be of the utmost importance later on. If D_i is the semi-infinite line (or half-line) starting from v with the direction from v to t_i, and D_0 (resp. D_{p+1}) is the half-line passing through the edge of P starting from v in clockwise order (resp. counterclockwise order), $D_0,...,D_{p+1}$ partition the region of P visible from v into p+1 simple polygons all adjacent to v. Typically a polygon is comprised between D_i,D_{i+1} and a convex polygonal line on B(P). Call a_i and b_i the endpoints of this convex line with b_i following a_i in clockwise order (note that this line is not necessarily a convex chain) - See Figure 2.18.

For each notch v, we define a data structure called the superrange of v, denoted SR(v), and used to represent the domain of P visible from v. SR(v) is the ordered list:

$$SR(v) = \{(a_0,b_0),...,(a_p,b_p)\}$$

Next we describe an efficient method for computing a superrange.

Theorem 11: The superrange of any notch can be computed in O(Nlog n) time after O(n) preprocessing.

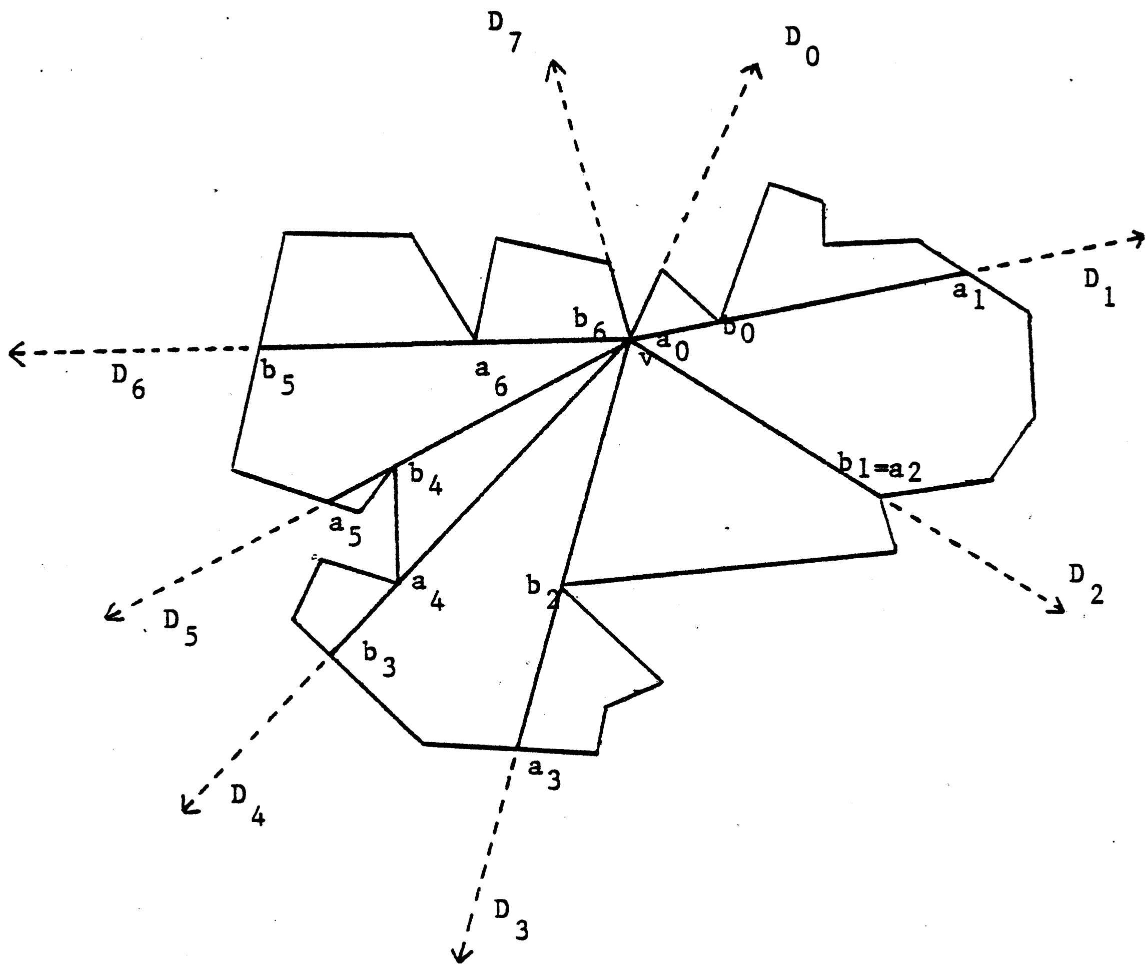


Figure 2.18: The superrange of a notch v.

Proof: We successively present an algorithm for computing the superrange, prove its correctness, then establish its running time. The algorithm uses two stacks of points with the usual primitives pusha, popa, topa for stack1 and pushb, popb, topb, for stack2. Initially both stacks are empty, and when the algorithm terminates, stack1 (resp. stack2) contains the list $a_0,...,a_p$ (resp. $b_0,...,b_p$) in this order.

Wlog, we assume that v is v_1. The algorithm considers in turn subchains of $C_1,...,C_m$ forming a partition of the boundary of P in clockwise order. Informally, a subchain starting at A is defined by observing the motion of vx as x scans the vertices of P in clockwise order starting from A. As long as vx sweeps angles in the same direction, that is, either clockwise or counterclockwise around v, and as long as x is not a pseudo-notch, the points of the boundary so far scanned belong to the subchain. Letting B represent the other endpoint of the subchain, Figure 2.19-a illustrates the 4 possible configurations with their respective designation.

Since we compute a subchain by sweeping the boundary of P until we change direction or we first hit a pseudo-notch, the 4 cases can be interpreted as follows:

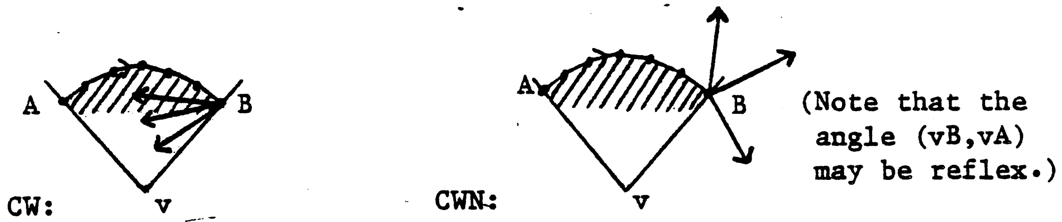
1. CW: Clockwise motion. Change of direction. B is not a notch.

2. CWN: Clockwise motion. Either B is a notch and there may or may not be a change of direction, or B is not a notch and there is no change of direction.

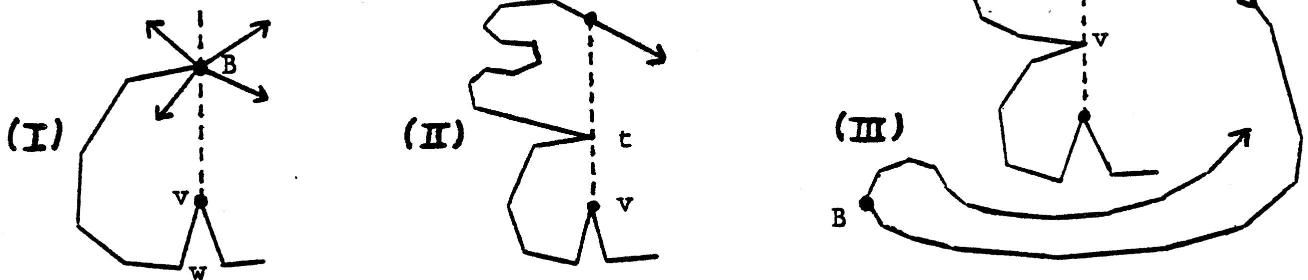
3. CCW: Counterclockwise motion. Change of direction. B is a notch.

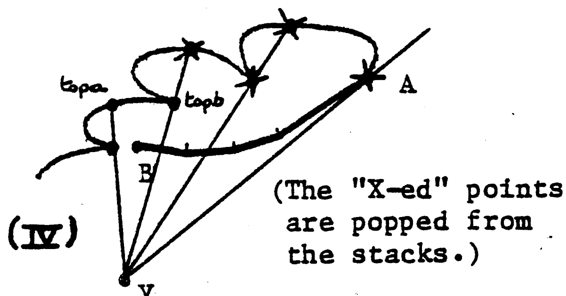
4. CCWN: Counterclockwise motion. Either B is a notch and there is no change of direction, or B is not a notch and there may or may not be a change of direction.

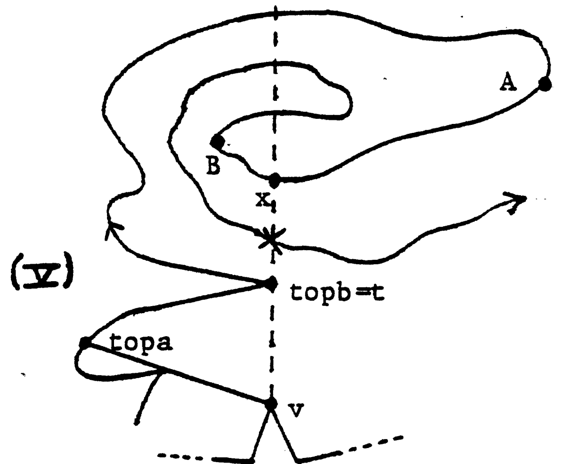
It is easy to see that these 4 cases cover all possibilities. The importance of these particular cases in computing superranges becomes apparent when we give an informal

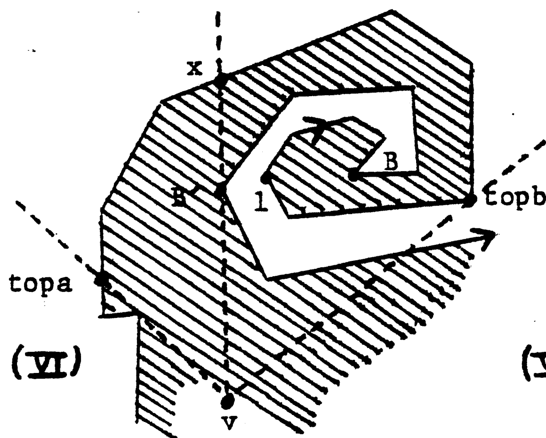


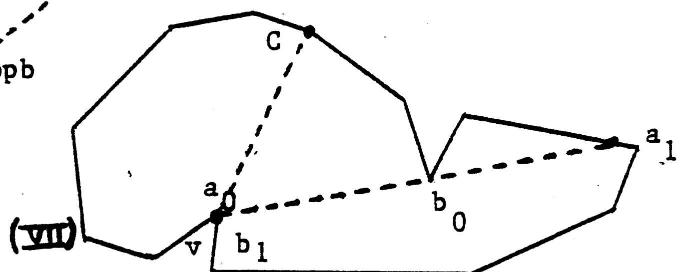
a) The 4 types of subchains.











STACK1	STACK2
a_1	b_1
c	b_0
a_0	c

b) Computing the superrange of v.

Figure 2.19

interpretation of them: CW and CWN are both clockwise motions, with CW corresponding to a situation where the boundary of P curves below the subchain and subsequently hides part of it from v. CCW and CCWN proceed counterclockwise. In both cases, the subchain currently scanned is not visible from v but CCW is the only case where the next subchain may be partly visible.

In all cases, we obtain B by either minimizing or maximizing the angle (vx,vA) for all vertices x of C_i. As will be shown in Chapter 4, this can be done in O(log k) operations if C_i has k vertices. C being a subchain, we define NEXT(C) as the subchain following C in clockwise order on the boundary of P. We adopt the convention that if C terminates at v, that is, is the last subchain before v, then NEXT(C) is 0. Thus, given the endpoints of C, we can determine the endpoints of NEXT(C) in O(log k) time if k is the number of vertices of the convex chain C_i where NEXT(C) belongs.

We also define l(C) as A if C is of type CW or CWN and B if it is of type CCW or CCWN. Since these 4 types can be distinguished in constant time, l can be computed in constant time.

We present some notation: "x intersect y" designates the intersection of x and y. By "hline(ab)", we mean the half-line originating from a and passing through ab. Also, if a and b lie on the same convex chain C_i (b following a in clockwise order), Chain(a,b) designates the portion of C_i comprised between a and b in clockwise order. Finally, we let w be the vertex of P following v in clockwise order. We can now present our algorithm:

Throughout the algorithm,

```
"Case 1" refers to the case where "C is of
         type CWN and the intersection of
         C with hline(vt) is not empty".

"case 2" =  "C is of type CW and the intersection
         of C with hline(vt) is not empty".

"case 3" =  "C is of type CCW".
```

Recall that B designates the ending vertex of the subchain C in a clockwise traversal.

```
Algorithm SR(v)

t=w , C=C1
while C not 0 begin
switch to the corresponding case
into which C falls:
case 1:
  pusha(C intersect hline(vt))
  pushb(B)
  t=B
  C=NEXT(C)
  while C not in case 1 or in case 2
      begin
           C=NEXT(C)
      end
  break

case 2:
  pusha(C intersect hline(vt))
  pushb(B)
  l=B
  C=NEXT(C)
  while true
    begin
    if (vl,vl(C))<180   then l=l(C)
    while x= C intersect vtopa not empty
        begin
             popa , popb , t=topb
        end
    if topb lies on vx
        then
             C=NEXT(C)
             while (C not in case 1 or 2) or
                   (C intersect hline(vt) does
                   not exist or does not lie on vx)
                   begin C=NEXT(C) end
             break
    else if C is in case 3 and l=l(C)
        then
             x= Intersection of hline(vB)
                with Chain(topa,topb)
             popb , pushb(x)
             t=topb , C=NEXT(C)
             break
        else C=NEXT(C)
    end
 end
 postprocessing:
  Eliminate from both stacks the pseudo-notches
  which are not notches of P.
```

REMARK: In statements such as "x= C intersect line", if the intersection is a whole

segment of C, either of its endpoints is to be assigned to x. However, to ensure $v=a_0=b_p$, we should choose the point nearest v.

Proving the correctness of the algorithm is delicate and requires great care. As a short-hand, we will say that a chain hinders a point M if it prevents it from being visible from v by intersecting the segment vM. The algorithm considers each subchain in turn and

1. Declares the current subchain to be visible from v if either no previous chain hinders it or, if some hinders it, so does a subchain to be examined later on.

2. Also, to guarantee correctness, all previous subchains declared visible so far must be checked to see if the current subchain hinders them or not.

The algorithm distinguishes between two cases:

I) The first subchain to be examined falls into case 1 or case 2. Since the polygon formed by closing the subchain with an edge joining its endpoints is convex (fundamental property of a convex chain), no point previously visited can hinder B, and t can be moved clockwise accordingly - See Figure 2.19-b-I. At all times, t is the point representing the biggest motion clockwise around v so far. Assume that the second subchain considered is not in case 1 or 2. Then since it must start counterclockwise (as seen from v), no subsequent subchain can be visible from v until one crosses hline(vt) - See Figure 2.19-b-II. At this point, the algorithm switches either to the first or the second case. In either case, there are 2 possibilities:

Either no previous subchain hinders C, and the previous procedure is still valid, or the chain "spirals" around v - See Figure 2.19-b-III. In this case, we just ignore this fact and declare the subchain visible from v. We still satisfy condition 1. since it must also be hindered by at least one subsequent subchain.

II) Assume that C falls in case 2. The subchain NEXT(C) hinders part of C and the

information stored in the stacks must be corrected. This is done in the block "while true begin ... end". First, as long as C and vtopa intersect (vtopa is the segment from v to topa), the whole subchain from topa to topb is hindered by C and the stacks must be popped - See Figure 2.19-b-IV. Now 3 cases must be distinguished according to the relative position of C and vtopb.

1) "If topb lies on vx" (Fig. 2.19-b-V), the correction is done. We reset the algorithm to case 1 or case 2 by looking for the first subsequent subchain to cross the segment xt.

2) "C is in case 3 and l=l(C)". Note that l is the point visited from the moment we entered case 2 corresponding to the biggest motion counterclockwise. Clearly, C may be in case 3 and yet have its endpoint B not visible from v if l is distinct from l(C). See the example of the subchain ending at B in Figure 2.19-b-VI. In the same figure, on the other hand, the subchain ending at B' permits us to finish the correction, since at that stage, l=l(C). We next fall into case 1 or case 2. It is helpful to observe that the a_i's and b_i's are computed only during clockwise sweeps and corrected when the motion goes counterclockwise.

3) Finally, if neither of the above alternatives occurs, we can directly switch to the next subchain.

Since convex chains are considered rather than the whole convex polygonal lines running between consecutive notches, the stacks may contain other points than the a_i,b_i defined in the superrange, namely, pseudo-notches which are not notches of P - See Figure 2.19-VII. Of course, each of these points appears in both stacks and can be removed with a simple scan through the stacks.

Finally, we establish the running time of the algorithm by observing first that no point popped from a stack can be pushed onto it again. Also, each subchain has first to be computed, then may be submitted to a constant number of intersection tests and cause exactly one intersection test with another convex chain (statement "x= Intersection of

hline(vB) with Chain(topa,topb)"). Therefore, the total running time is bounded by O(Nlog n) since each of the m=O(N) convex chains defines at most 3 subchains. The proof is now complete since we have seen earlier how to compute the convex chains in O(n) preprocessing. □

The notion of superrange can be of great use for many geometric problems and is, thus, interesting in its own right. To understand its fruitfulness in our specific problem of decomposition, we need to introduce a function of two arguments R(v,D), where v is a notch of P and D is a half-line emanating from v which contains at least a segment vx lying inside P. This ensures that at least a piece of D is visible from v. Let D_i, D_{i+1} be the 2 half-lines introduced in the definition of the superrange between which D lies. Then, if (vb_i,va_i) is reflex, R(v,D) is set to 0, otherwise it is set to the segment vy where y is the intersection of D with a_ib_i. To simplify the notation, we also define R(v,va) as R(v,hline(va)). We clearly have:

Lemma 5: Once the superrange of each notch has been computed, R(v,D) can be evaluated in O(N) time, for any notch v.

Remark: Since the superrange is an ordered list sweeping the plane clockwise around v, D_i and D_{i+1} can be located through a binary search, thus leading to an O(log N) running time for computing R(v,D). This may be interesting for reasons of efficiency, but will not, however, change the order of magnitude of the decomposition algorithm presented later on.

From here on, we assume that, in a preprocessing stage, the superrange of each notch has been precomputed. The motivation for the superrange appears clearly in the following.

Theorem 12: If v is a notch of P and vx the edge of an X-pattern, x lies on the segment R(v,vx).

Proof: The theorem is obvious if x lies on the boundary of P, since it is then a notch of P and R(v,vx) is exactly vx. Suppose that x is a Steiner point and vx contains R(v,vx) - See Figure 2.20. Let (a_i,b_i) be the pair of SR(v) such that vx intersects a_ib_i. The segment a_ib_i is a divider which partitions P into two polygons P1 and P2, with, say, P1 containing x. Since the portion of B(P) between a_i and b_i is a convex chain, P1 is a convex polygon, therefore the X-pattern cannot have notches in P1, which contradicts Lemma 4. □

We will next show how to use the previous results to compute X2-, X3-, and X4-patterns efficiently.

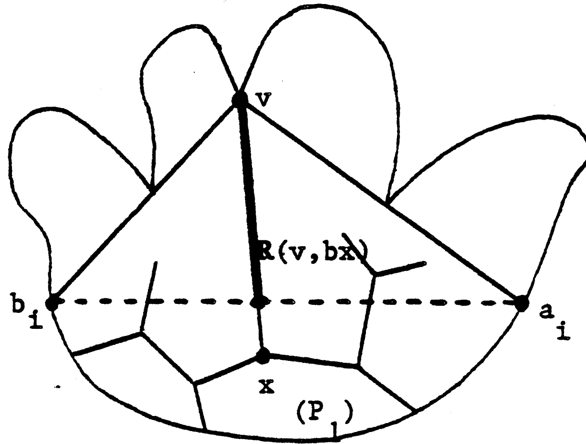


Figure 2.20: R(v,D) expresses the longest edge vx with direction D of an X-pattern.

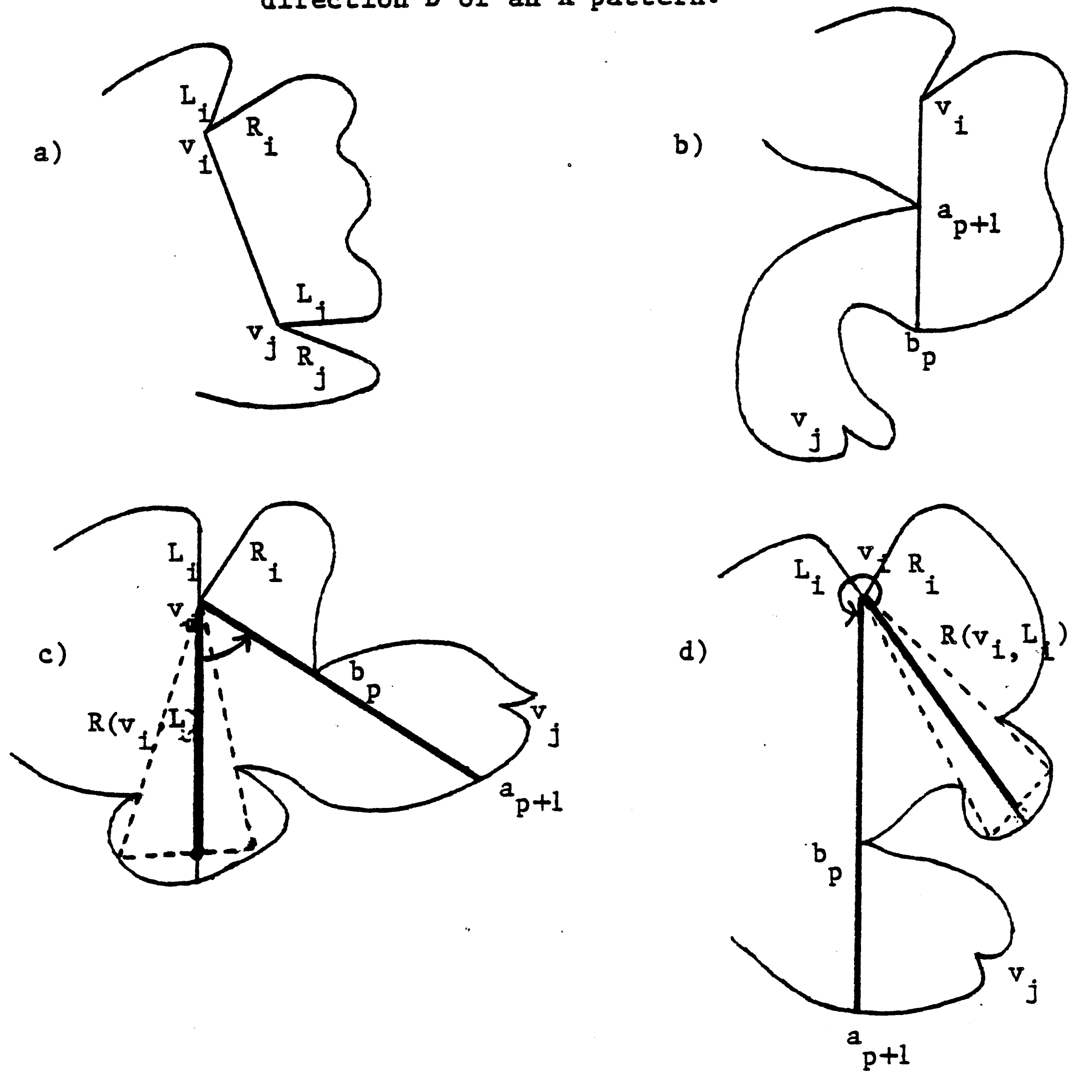


Figure 2.21: The definition of r_{ij} and l_{ji}.

2.3.3 Detecting X2-patterns

Theorem 13: Given two notches v_i,v_j, the possibility of an X2-pattern between v_i and v_j can be detected in constant time, after a preprocessing taking $O(N^3+N^2\log n)$ time.

Proof: The preprocessing involves computing the superrange of each notch, then determining the segments $R(v_i,v_iv_j)$ for all pairs of notches v_i,v_j. From Theorem 11 and Lemma 5, it follows that this can be done in $O(N^3+N^2\log n)$ time. Now the result of theorem 12 implies that an X2-pattern between v_i and v_j is possible if and only if $R(v_i,v_iv_j)=v_iv_j$, and the segment v_iv_j removes the reflex angle at both v_i and v_j. □

2.3.4 Detecting X3-patterns

We now turn to the more complex problem of computing X3-patterns. We need some additional preprocessing that we next describe: Call R_i and L_i the two edges of P adjacent to v_i (R_i following L_i in a clockwise traversal of the boundary). For our purposes here, we actually consider R_i and L_i oriented towards v_i, thus permitting us to define $R(v_i,R_i)$ and $R(v_i,L_i)$ without ambiguity. Similarly, to give full meaning to angles of the kind $(v_ix,R(v_i,D))$, we always assume $R(v_i,D)$ to be a segment oriented in the direction of D.

The preprocessing involves computing the superrange of each notch. Then, for each pair of notches v_i,v_j, we define the two quantities r_{ij} and l_{ji} as follows: v_ir_{ij} is basically the rightmost segment in the range of v_i which can be seen from v_j. More precisely, if both R_i and L_i lie on the same side of line(v_iv_j) with $(v_iv_j,R_i)<180$ (Fig.2.21-a), we determine the pairs (a_p,b_p) and (a_{p+1},b_{p+1}) of $SR(v_i)$ such that v_j occurs between b_p and a_{p+1} in a clockwise traversal of B(P). We assume that a_{p+1} does not lie strictly between v_i and b_p (Fig.2.21-b). We may have $v_j=b_p=a_{p+1}$, however. Then, let t be the segment $R(v_i,L_i)$ if $(R(v_i,L_i),v_ia_{p+1})<180$ (Fig.2.21-c), else v_ia_{p+1} (Fig.2.21-d). Actually if we now have $(v_iv_j,t)<180$, we define t as $R(v_i,v_iv_j)$. Finally, if $(t,R_i)<180$, we define r_{ij} as the endpoint of t other than v_i. If any of the conditions above fails, r_{ij} is 0.

We repeat the same process on v_j with respect to v_i. If R_j and L_j lie on the same side of line(v_iv_j), we first determine the pairs (a_p,b_p) and (a_{p+1},b_{p+1}) from $SR(v_j)$ such that v_i occurs between b_p and a_{p+1} in clockwise order. We also suppose that b_p does not lie strictly between v_j and a_{p+1}. Then we

let t be $R(v_j,R_j)$ if $(v_jb_p,R(v_j,R_j))<180$ or v_jb_p otherwise. Similarly, if $(t,v_jv_i)<180$, t is changed to $R(v_j,v_jv_i)$, so that we can define l_{ji} as the endpoint of t other than v_j if $(L_j,t)<180$. In all other cases, l_{ji} is 0.

With the superrange of each notch at our disposal, we can compute each r_{ij} and l_{ji} in O(N) time, summing up to an $O(N^3+N^2\log n)$ preprocessing time.

We can now achieve the first of our goals:

Theorem 14: After some preprocessing taking $O(N^3+N^2\log n)$ time, the possibility of an X3-pattern between 3 given notches can be detected in constant time.

Proof: We will show that v_i,v_j,v_k are the notches in clockwise order of an X3-pattern if and only if:

1. v_i,v_j,v_k occur in clockwise order on the triangle $v_iv_jv_k$.

2. $r_{ij},r_{jk},r_{ki},l_{ik},l_{kj},l_{ji}$ are all distinct from 0.

3. The points $A = v_kl_{kj}$ intersect v_jr_{jk}, $B = v_il_{ik}$ intersect v_kr_{ki} and $C = v_jl_{ji}$ intersect v_ir_{ij} are well defined.

4. The polygon $Q = v_iCv_jAv_kBv_i$ is simple and has a non-empty kernel. (Recall that the kernel of a polygon Q is the region of Q visible from every point in Q [Shamos,78]).

All these conditions can be easily tested in constant time with the preprocessing described above.

We say that a point x is <u>range-visible</u> from a notch v if it lies in its range, that is, if the segment vx lies totally inside P and removes the reflex angle at v. We define the <u>open triangle</u> (ax,ay) as the convex region swept by a half-line pivoting in clockwise order around a from ax to ay.

Let S be the Steiner point of an X3-pattern between v_i,v_j,v_k. The first condition is clearly necessary. To prove that the second is such, we will just show that r_{ij} is not 0, all of the other cases being similar. Since the 3 edges of the pattern must lie in the triangle $v_iv_jv_k$, the first requirement illustrated in Figure 2.21-a is obvious. Reconsidering the pairs $(a_p,b_p),(a_{p+1},b_{p+1})$, it is equally clear that the configuration of Figure 2.21-b cannot lead to an X3-pattern since we must have $(Sv_j,Sv_i)<180$, where S is the Steiner point. Indeed, Sv_j must intersect b_pa_{p+1} with possibly $v_j=b_p$, since S must be visible from both v_i and v_j. This remark shows that not only the configurations of Figure 2.21-c,-d are the only ones possible, but also that S cannot lie in the open triangle (v_ia_{p+1},R_i). The other conditions to satisfy in order to define r_{ij} express the fact that S lies in the triangle $v_iv_jv_k$ as well as in the range of v_i. Also, since we must have $(v_iS,v_iv_j)<180$, it is legitimate to set t to $R(v_i,v_iv_j)$ if $(v_iv_j,t)<180$. Finally, if (t,R_i) is reflex, no point visible from v_j can be range-visible from v_i and we can set r_{ij} to 0. Thus when a Steiner point exists, all these conditions will be satisfied and S cannot lie in the open triangle (v_ir_{ij},R_i).

As mentioned in defining the superrange, $a_0b_0a_1b_1...a_pb_p$ occur in clockwise order on the boundary of P, therefore we must have $(v_il_{ik},v_ir_{ij})<180$ when l_{ik} and r_{ij} are distinct from 0, since v_i,v_j,v_k occur in clockwise order. It follows that if A,B, and C exist, the polygon Q must be simple. To prove that these points are well defined, we first show that Sv_j intersects v_ir_{ij}. Since we have seen that Sv_j must intersect v_ia_{p+1}, thus implying that v_ir_{ij} is not defined as $R(v_i,v_iv_j)$, we only have to show that Sv_j intersects v_ir_{ij} in the case where this segment is defined as $R(v_i,L_i)$. Since S cannot lie in the open triangle $(R(v_i,L_i),R_i)$, Sv_j must intersect v_iB (Fig.2.22-a), and it cannot intersect Br_{ij}, since S would then belong to a convex polygon where no Steiner point can lie - See Theorem 12. This proves our claim, and shows that r_{ij} (as well as l_{ji} by a similar reasoning) lies outside the triangle Sv_iv_j - See Figure 2.22-b. Finally, as we know that S cannot lie in the open triangles (v_ir_{ij},R_i) and (L_j,v_jl_{ji}), we conclude that both angles (v_iS,v_ir_{ij}) and (v_jl_{ji},v_jS) are < 180, which, combined with the previous result, establishes that v_ir_{ij} and v_jl_{ji} intersect. This proves the existence of the point C, as well as A and B, by symmetry. Finally, since S can

lie only in the open triangle $(v_i l_{ik}, v_i r_{ij})$, the same result about v_j and v_k proves that it actually lies in the kernel of Q.

The 4 conditions having been proved necessary, we next show that they are sufficient. Assume that they are all satisfied - See Figure 2.22-c. Since $v_i r_{ij}$ is range-visible from v_i, and so is $v_j l_{ji}$ from v_j, condition 3 shows that C is range visible from both v_i and v_j. It follows that the boundary of P cannot intersect with $v_i C$ or $v_j C$, and by symmetry, cannot intersect with the edges of Q. Therefore, any point of its kernel is range-visible from v_i, v_j, v_k, and is the Steiner point of a possible X3-pattern. Note that all 3 angles around the Steiner point are ensured to be <180 since the kernel of Q lies inside the triangle $v_i v_j v_k$. □

Corollary 14: (Corol. of Theor. 14)

1) It turns out that we will not have to use the previous result directly since only X4-patterns and Y-patterns will compose our OCD's. However, we can show how a minor modification of the method presented above permits us to patch Y-subtrees together in constant time. More precisely, the 3 edges adjacent to a Steiner point of a Y-pattern can be viewed as forming an X3-pattern, where the angles at the 3 notches would have been slightly modified. The three remaining pieces of the Y-pattern emanating from the three notches are called Y-subtrees. Conversely, given three Y-subtrees denoted Y_i, Y_j, Y_k adjacent to v_i, v_j, v_k respectively, we might ask whether they can be patched together to form a Y-pattern. Let t_i, t_j, t_k be the three edges of the Y-subtrees adjacent to v_i, v_j, v_k respectively - See Figure 2.23. It is clear that the Y-subtrees can be patched if and only if the 4 conditions of theorem 14 are satisfied, where in all statements with R_i, R_j, R_k, these segments have been replaced by t_i, t_j, t_k respectively. Note that this detection will operate in constant time only if $R(v_i, t_i)$, $R(v_j, t_j)$, $R(v_k, t_k)$ have been precomputed, since these segments are needed for the detection (see proof of Theorem 14).

2) A second application of Theorem 14 relates to the construction of Y-patterns from several subtrees. Suppose that two Y-subtrees Y_i and Y_j are given with the configuration of Figure 2.23. Let S be a point visible from v_i, v_j, v_k such that the angle $(v_k v_j, v_k S)$ is maximum and, with the adjunction of Y_i and Y_j, no reflex angle exists at v_i, v_j, or S. Clearly, if there exists a Y-subtree Y_k such that Y_i, Y_j, Y_k can be patched as indicated above, we can always assume S to be the Steiner point between v_i, v_j, and v_k. The problem is now to determine S if such a point exists. Once again, Theorem 14 shows that S is defined if and only if the 4 conditions are satisfied. Now all occurrences of R_i and R_j should be replaced by t_i and t_j respectively, and all statements involving R_k should simply be dropped. Since S has to be in the kernel of Q, S is the vertex x of this kernel that maximizes the angle

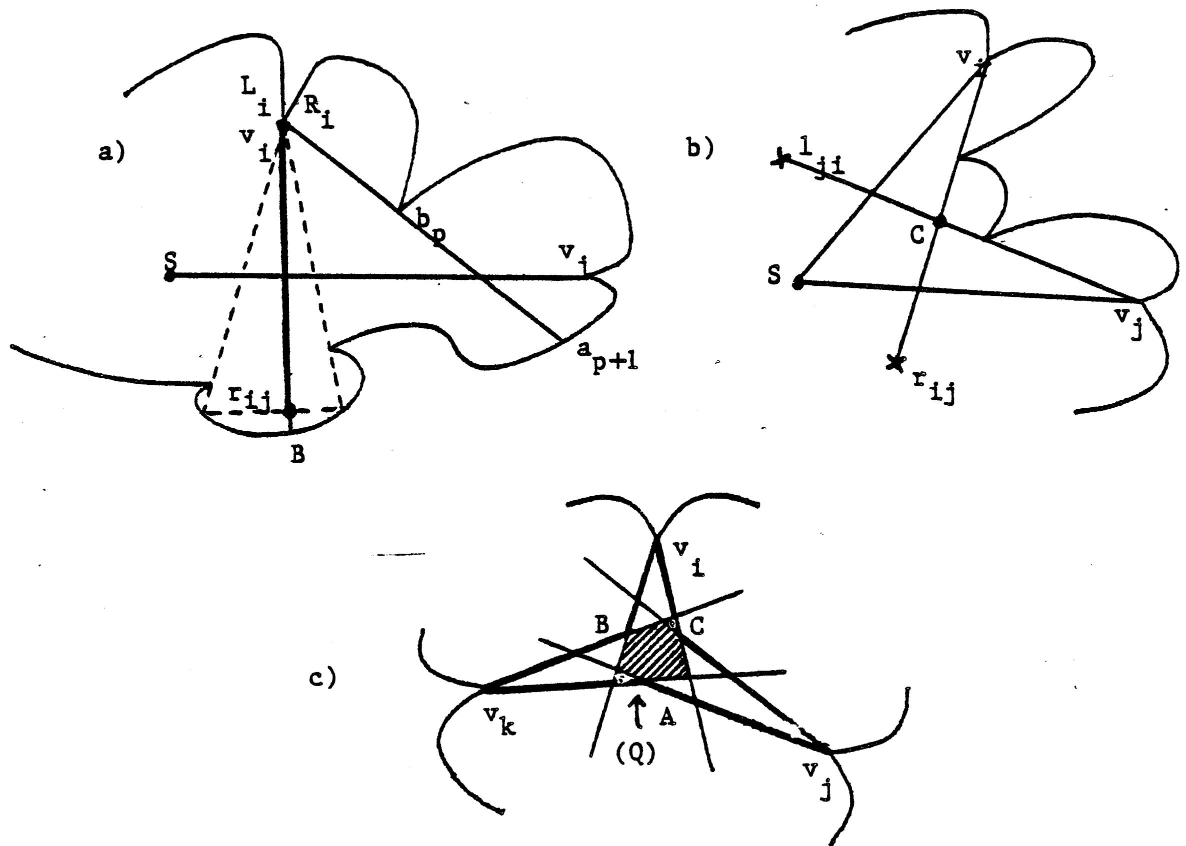


Figure 2.22: Detecting X3-patterns.

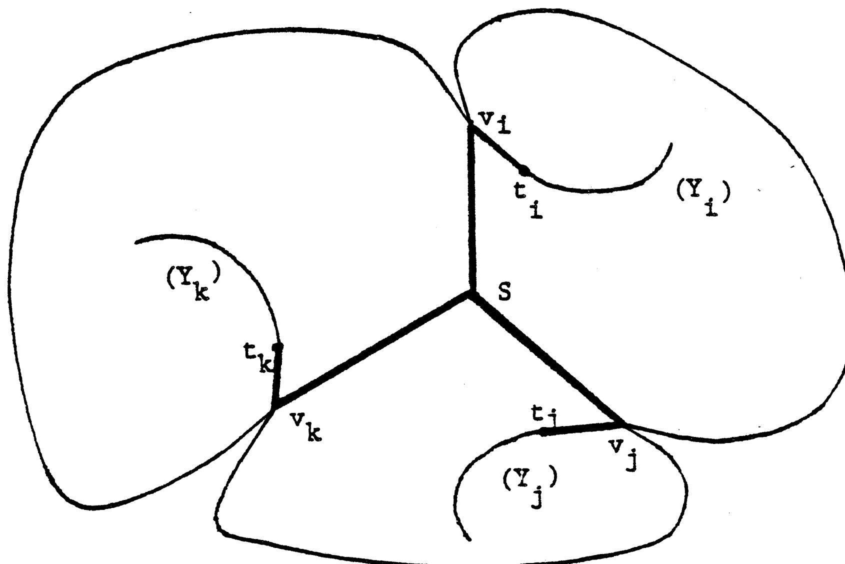


Figure 2.23: Patching Y-subtrees is similar to detecting X3-patterns.

$(v_k v_j, v_k x)$. With $R(v_i,t_i)$ and $R(v_j,t_j)$ precomputed, the 4 conditions can be tested and the kernel computed (if it exists) in constant time. Since the kernel cannot have more than 6 edges, S will also be determined in constant time.

All the results of this corollary still apply for the "counterclockwise" configuration, where all the Y-subtrees connect the notches to the left instead of the right.

2.3.5 Detecting X4-patterns

At this point, we wish to present a simple method for detecting X4-patterns. We will describe a much more efficient algorithm in a later section, which we have judged too involved to be introduced here.

Theorem 15: After some preprocessing taking $O(N^3+N^2\log n)$ time, the possibility of an X4-pattern between 4 given notches can be detected in $O(N\log(n/N))$ time.

Proof: v_i, v_j, v_k, and v_l being the notches given in clockwise order, two types of X4-patterns have to be considered and tested successively - See Figure 2.24-a,-b. We only describe the method for the case where v_i and v_j are adjacent to the same Steiner point, the other case being treated similarly.

We will see later on that, for our purposes, we can always assume that no reduction of the pattern can lead to an X3-pattern or a Y5-pattern with a notch between the two Steiner points. In this case, it is easy to see that applying the two reductions of Figure 2.24-b,-c successively, will take A to the intersection of $v_i r_{ij}$ and $v_j l_{ji}$ (recall that these points have been defined in the preprocessing of Theorem 14). This comes from the fact that, by definition, $v_i r_{ij}$ (resp. $v_j l_{ji}$) is a segment range-visible from v_i (resp. v_j) with a minimum angle $(v_i r_{ij}, v_i v_j)$ (resp. $(v_j v_i, v_j l_{ji})$, and which must intersect $v_j A$ (resp. $v_i A$). A being now $v_i r_{ij}$ intersect $v_j l_{ji}$ with similarly B= $v_k r_{kl}$ intersect $v_l l_{lk}$, the existence of an X4-pattern implies that AB does not intersect the boundary of P and removes all reflex angles at A and B. Reciprocally, these conditions are sufficient. All r_{uv}, l_{uv} have been precomputed in the preprocessing, and ensuring that AB removes the reflex angles at A

and B takes constant time. Finally, we must test for the intersection of AB with the boundary of P. This can be done by intersecting each convex chain $C_1,...,C_m$ with AB in logarithmic time, leading to an O(Nlog(n/N)) execution time, as seen in the proof of theorem 4. □

2.3.6 The Optimal Convex Decomposition Algorithm

2.3.6.1 Introduction

The procedure for determining a maximum set of compatible patterns which can be used in a decomposition is based on a dynamic programming approach for the following reasons: Suppose that an oracle informs us that a certain X4 or Y-pattern belongs to an OCD. If this pattern has k notches, it decomposes P into k subpolygons $P_1,...,P_k$, and finding an OCD for each of them will give us an OCD of P.

To do so, we compute maximal sets of compatible patterns for each P_i. Since the notches of P_i are also notches of P, any X-pattern of P_i is also an X-pattern of P. Conversely, we want to show that any X-pattern of P involving only notches in P_i is also an X-pattern of P_i. This is of the utmost importance since dynamic programming proceeds bottom-up, therefore we will have to find a maximal set of patterns involving notches of P_i before even knowing the exact shape of P_i. Thus we must prove the following.

Let $z_1,...,z_k$ be the notches of an X-pattern T in clockwise order around the boundary of P, and let $v_i,v_{i+1},...,v_j$ (denoted V(i,j)) be the notches of P between z_u and z_{u+1} in clockwise order ($z_u=v_{i-1}$, $z_{u+1}=v_{j+1}$). We will show that no X-pattern S which has all of its notches in V(i,j) can intersect T.

Assume that S intersects an edge e of T. Consider the unique divider D of P passing through e. Recall that D is defined as the shortest segment collinear with e having both endpoints on B(P). D partitions P into two polygons P1 and P2 - See Figure 2.25. Since the path of T between z_u and z_{u+1} is a convex polygonal line, it lies entirely in P1 or P2 (say P1). It follows that all the notches in V(i,j) are notches of P1. Finally, from Lemma 4, we conclude that S must have notches in P2, which leads to a contradiction and proves our claim.

We can now define <u>S(i,j)</u> for every pair of notches v_i,v_j, as a maximum set of compatible X4- or Y-patterns which may be applied between notches in V(i,j) only. The ultimate goal being to find

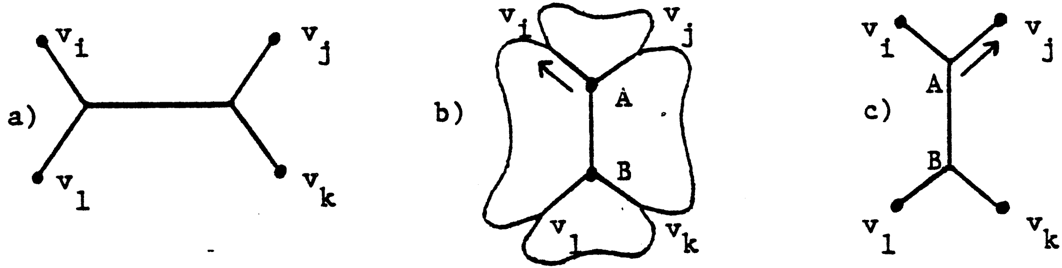


Figure 2.24: A simple method for computing X4-patterns.

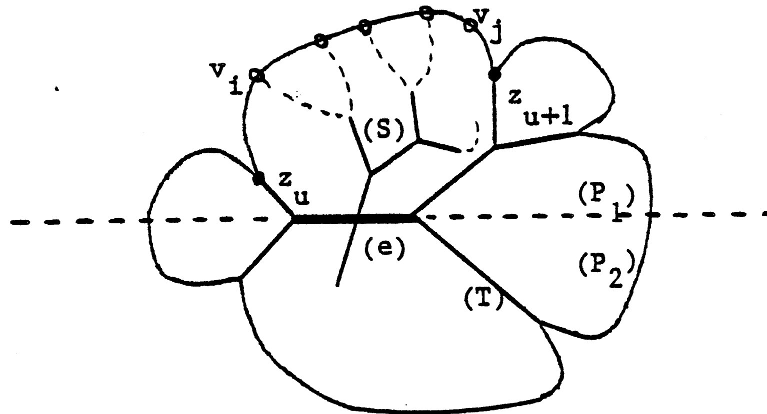


Figure 2.25: The interaction between X-patterns.

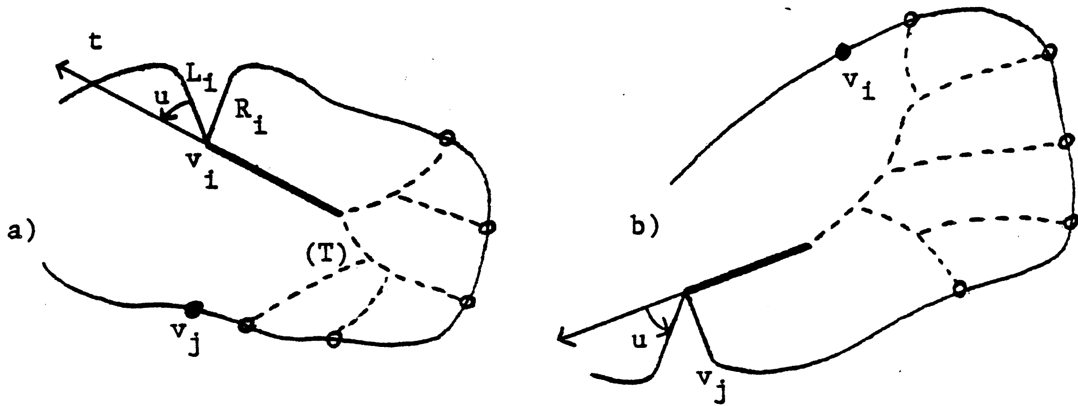


Figure 2.26: Defining B(i,j) and F(i,j).

S(1,N), the dynamic programming algorithm computes S(i,j) from {S(k,l)|V(k,l) strictly included in V(i,j)}. This can be done directly if v_i and v_j do not have to to be connected to the same pattern. We simply test all combinations {S(i,k),S(k+1,j)} for all v_k in V(i,j-1). Otherwise, we have to distinguish whether v_i and v_j have to be connected together to an X4- or a Y-pattern.

To handle the latter case, we compute all possible Y-patterns with a dynamic programming approach, that is, we eliminate the patterns which could not belong to an OCD. We compute Y-subtrees and Y-patterns by patching smaller Y-subtrees, using previous results on the detection of X2- and X3-patterns. In order to prevent the number of computations from blowing up, however, we keep only the Y-subtrees that are candidates for belonging to an OCD. A Y-subtree is considered not to be a candidate if, at the time it is computed, we are ensured of the existence of an OCD which does not use this Y-subtree (although we may not know this OCD explicitly yet). As a short-hand, we say that a pattern or a Y-subtree lies in V(i,j) if all its notches do. It remains now to formalize the intuition given here, and present a polynomial-time algorithm.

Consider a Y-pattern which is used in an OCD and has v_i as an N2-node (recall the definition of an N2-node in Figure 2.11). v_i splits it into two Y-subtrees, and there exists j such that

1. One of the Y-subtrees lies in V(i,j) whereas the other lies in V(j+1,i).

2. All the other patterns in the OCD lie totally inside or outside V(i,j).

We want to consider the candidacy of this Y-subtree at the time S(i,j) is computed. We first observe that if $v_i, v_{i_1}, \ldots, v_{i_m}$ is a list of its notches in clockwise order, we can ignore its candidacy if we do not have:

$$3.\ |S(i,j)| = |S(i+1,i_1-1)| + .. + |S(i_{m-1}+1,i_m-1)| + |S(i_m+1,j-1)|$$

(Of course, if $v_m = v_j$, the rightmost term must be omitted.)

Indeed, if 3. does not hold, the right-hand side is strictly smaller than |S(i,j)|. Also, the assumption that a Y-pattern using this Y-subtree is the only one to overlap both in V(i,j) and V(j+1,i) (conditions 1. and 2.) implies that removing the pattern from the OCD and replacing all the patterns lying in V(i,j) by those of S(i,j) will yield a decomposition at least as good.

Let t be the edge adjacent to v_i in the Y-subtree lying in V(i,j). We give t an orientation towards v_i. Among all the Y-subtrees in V(i,j) having v_i as an N2-node, satisfying relation 3, and such that $u=(L_i,t)<180$, we can consider the Y-subtree T which minimizes the angle u as the only candidate, since all the candidates must cause the same savings between their notches - See Figure 2.26-a. We

then define B(i,j) as the pair $(R(v_i,t),T)$. If there is no such subtree, B(i,j) is the pair $(R(v_i,R_i),0)$. Carrying out the same reasoning counterclockwise in V(i,j) with now v_j as an N2-node, we define F(i,j) in a similar fashion - See Figure 2.26-b.

2.3.6.2 The Algorithm

Having established our notation, we are now in a position to present our algorithm. We assume a function <ARG> for assembling Y-subtrees in computing S(i,j). ARG is in general a pair of Y-subtrees taken from B(u,v) or F(u,v). If these two subtrees can be patched together to form a Y-pattern, Y, the function <.> returns (C,Y), where C is the maximum number of compatible patterns which can be applied in V(i,j), including Y. We return to a discussion of this function after a presentation of our algorithm. However, before proceeding with a formal description, we will give a brief overview of the algorithm.

After all the necessary preprocessing in STEP 1, we set up a double loop to implement the dynamic programming scheme. Each step involves computing S(i,j) for a given value of i and j. First we compute the best Y-pattern which connects v_i and v_j (STEP 2). This involves patching Y-subtrees previously computed and selected as candidates. STEP 3 computes A as a maximal set of compatible patterns in V(i,j), where v_i and v_j cannot belong to the same pattern. Then we allow for an X4-pattern connecting v_i and v_j and compute B. Finally, the Y-pattern of STEP 2 is used to compute C and the maximal set among A,B,C can be chosen as S(i,j). STEP 4 computes the Y-subtrees which are considered candidates and lie in V(i,j). These subtrees are to be used later on in STEP 2. Once a maximal set of compatible patterns for P has been determined, we can finish off the decomposition with the naive method (STEP 5).

Algorithm CD

STEP 1: "Preprocessing"

Check that P is simple and non-convex. Make a list of the notches $v_1,...,v_N$ and the convex chains $C_1,...,C_m$. For each notch v_i, compute $SR(v_i)$, $R(v_i,R_i)$, and $R(v_i,L_i)$. For each pair of notches v_i,v_j, determine the pairs (a_p,b_p) of $SR(v_i)$ as defined in the preprocessing of Theorem 14, and compute $R(v_i,v_iv_j)$. Initialize B(i,i) (resp. F(i,i)) to $(R(v_i,R_i),0)$ (resp. $(R(v_i,L_i),0)$) and S(i,i) to the empty set.

```
for  d=1,...,N-1
     for  i=1,...,N
              let j=i+d [mod N] and do steps 2,3,4.
```

STEP 2:

Compute the best Y-pattern connecting v_i and v_j as the maximum element of $U_i Yi, i=1,...,4$ (the elements of these sets being pairs (C,Y-pattern), the maximum is taken with respect to C), where for all v_k in V(i+1,j-1):

```
Y1= { <F(i,k),B(k,j)> }
Y2= { <B(i,k-1),F(k,j)> }U{ <B(i,j-1),F(j,j)> }
Y3= { <B(i,k-1),B(k,j-1)> }
Y4= { <F(i+1,k),F(k+1,j)> }
```

STEP 3:

Compute S(i,j) as the maximum of A,B,C (with respect to the cardinality of the sets).

A= Max[S(i,k) U S(k+1,j)] for all v_k in V(i,j-1) corresponding to taking the best Y-patterns in V(i,k) and V(k+1,j).

B= Max[$\{x_{i,a,b,j}\}$US(i+1,a-1)US(a+1,b-1)US(b+1,j-1)] for all X4-patterns $x_{i,a,b,j}$ connecting v_i, v_a, v_b, v_j, with v_a and v_b in V(i,j).

C= { the Y-pattern computed in STEP 2 (having notches $v_i, v_{i_1}, ..., v_{i_p}, v_j$ in clockwise order) }
US(i+1, i_1-1)U...US(i_{p-1}+1, i_p-1)US(i_p+1,j)

STEP 4:

```
Compute B(i,j) and F(i,j).
```

STEP 5:

```
Finish the decomposition using the naive algorithm,
adding one polygon for each remaining notch.
```

The remainder of Section 2.3.6 is devoted to justifying the steps of the algorithm and determining its time of execution.

2.3.6.3 Patching Y-subtrees Together (STEP 2)

We define the function <ARG> to take two Y-subtrees and construct a Y-pattern if these subtrees can be patched together. This process forms the core of STEP 2 of the algorithm CD.

ARG is any argument of the kind: (F(i,k),B(k,j)), (B(i,k-1),F(k,j)), (B(i,k-1),B(k,j-1)), or (F(i+1,k),F(k+1,j)), with v_i, v_k, v_j occurring in clockwise order. We outline the algorithm for only the first three cases, with the last case following directly from this description.

```
case <F(i,k),B(k,j)> - (Fig.2.27-a)
  Let F(i,k)=(r,T) and B(k,j)=(s,V).
  if (r,s)<180 and T not 0 and V not 0
    then  return(|S(i,k)|+|S(k,j)|+1 , Y-pattern:TUV)
    else  return(0)
```

```
case <B(i,k-1),F(k,j)> - (Fig.2.27-b)
  Let B(i,k-1)=(r,T) and F(k,j)=(s,V).
  if an X2-pattern is possible between v_i and v_j
  with the notches adapted to T and V respectively, in the
  sense of Corollary 14
    then return(|S(i,k-1)|+|S(k,j)|+1,Y-pattern:v_i v_j UTUV)
    else return(0)

case <B(i,k-1),B(k,j-1)> - (Fig.2.27-c)
  Let B(i,k-1)=(r,T) and B(k,j-1)=(s,V).
  r and t allow to test the possibility of patching
  T and V with the notch v_j (Corollary 14.1).
  if the patching is possible, call W the resulting
  Y-pattern.
    then return(|S(i,k-1)|+|S(k,j-1)|+1 , W)
    else return(0)
```

We omit the last case, which we have, however, illustrated in Figure 2.27-d.

Because the Y-subtrees in the B(u,v) and F(u,v) satisfy relation 3., the number that <ARG> returns along with a Y-pattern represents the maximum number of compatible patterns which can be applied in V(i,j), once this Y-pattern has been applied.

We state this formally as:

Lemma 6: STEP 2 computes in O(N) time a Y-pattern connecting v_i and v_j (if any exists) such that the number of compatible patterns which can be applied in V(i,j) between its notches is maximum.

Proof: Corollary 14.1 shows that each call on <ARG> takes constant time, therefore STEP 2 requires O(N) time. Note that all patterns and Y-subtrees being represented by adjacency lists, merging any pair of them takes constant time. We next have to show that

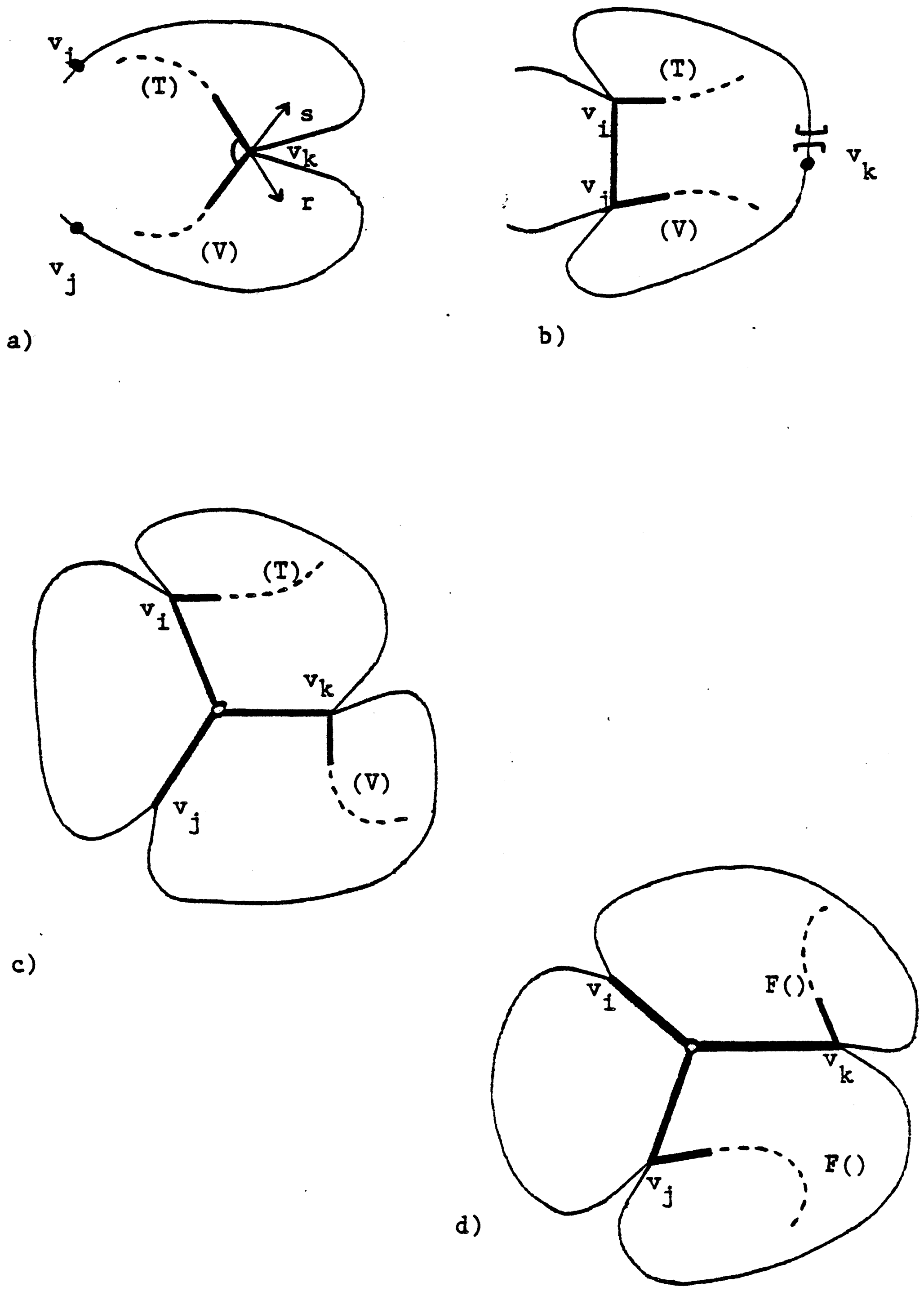


Figure 2.27: Computing the function <ARG>.

STEP 2 investigates all possibilities of Y-subtrees connecting v_i and v_j. Consider the path from v_i to v_j in the Y-pattern which we are looking for. If it contains an N2-node, it will be detected in Y1. Otherwise, one N3-node may appear on this path, and all such candidates will be considered in Y3 and Y4. Finally, if no N2- or N3-node exists, this will be handled in Y2. Note that for reasons explained earlier, it is legitimate to consider only the Y-subtrees in the B's and F's, since there are the only remaining candidates at this stage. □

2.3.6.4 Computing S(i,j) (STEP 3)

Suppose that we have a procedure to compute the quantity B of STEP 3 in Xfour(n,N) time. Then it follows that

Lemma 7: STEP 3 computes S(i,j) in time $O(N^3+N^2Xfour(n,N))$.

Proof: The running time follows from the fact that A and C can be computed in O(N) time. To show the correctness of the computation, we assume by induction that S(k,l) has been computed for all v_k,v_l in V(i,j) (except for S(i,j)). We first find a maximal set of compatible patterns without allowing any to have both v_i and v_j as vertices. Then, we allow one X4 and finally one Y-pattern to have these two notches as its vertices. In the latter case, Lemma 6 justifies considering only the Y-pattern computed in STEP 2. □

Note that Theorem 15 shows how to detect an X4-pattern in O(Nlog(n/N)) time. We can then set Xfour(n,N) to $O(N^3\log(n/N))$ at this stage.

2.3.6.5 Constructing Y-subtrees (STEP 4)

We compute B(i,j) and F(i,j) in STEP 4 of the algorithm by iteratively patching Y-subtrees via functions Y(i,ARG) and Y'(i,ARG), where ARG is an argument of the form B(a,b) or (B(a,b),B(c,d)) (or the same with F). We outline these functions with the B's only, the other cases being symmetric "in the opposite direction". - See Figure 2.28.

case Y(i,B(a,b))

Let B(a,b)=(r,T) and t be the edge of T adjacent to v_a. v_a,v_b,v_i must occur in clockwise order.

if v_iv_a does not intersect with the boundary of P (except at v_i and v_a), removes the notch at v_a adapted to T, and satisfies: $w=(R_i,v_iv_a)<180$.

then return(Y-subtree:v_iv_aUT)

else return(0)

case Y(i,(B(a,b),B(c,d)))

Let B(a,b)=(r,T) and B(c,d)=(s,V).

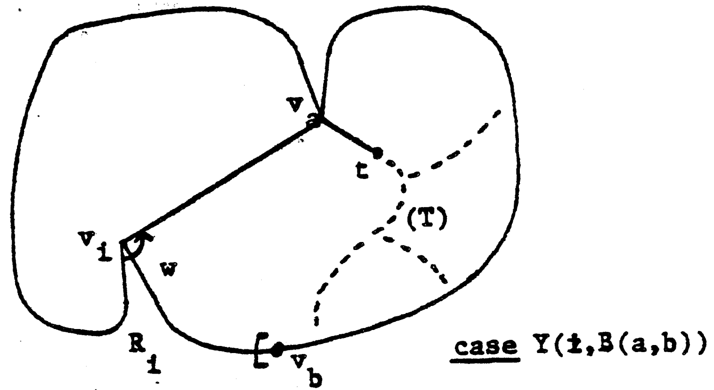
v_i,v_a,v_b,v_c,v_d must occur in clockwise order. With the notches v_a and v_c adapted to T and V respectively, check if an X3-pattern is possible between v_i,v_a,v_c. If yes, compute the Steiner point S which maximizes $w=(R_i,v_iS)$ - (Corollary 14.2). Finally,

if w<180

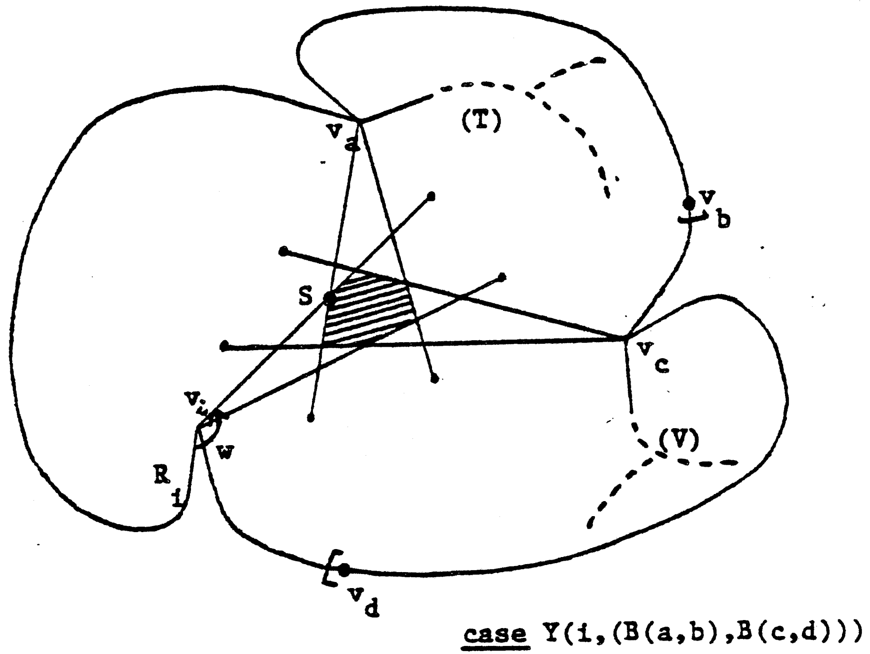
then return(Y-subtree: Sv_iUSv$_a$USv$_c$UTUV)

else return(0)

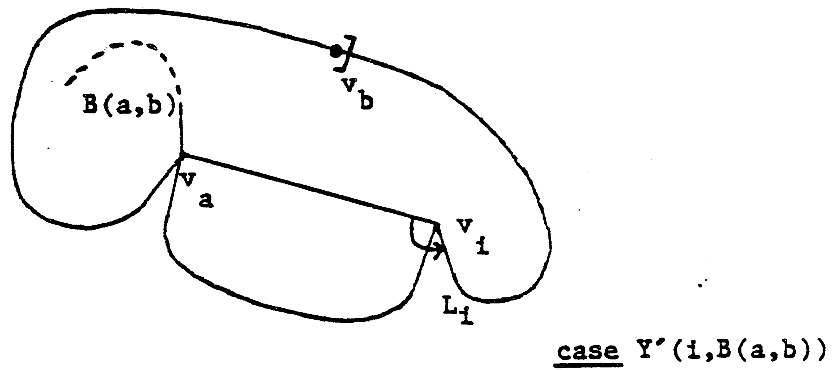
We define Y'(i,B(a,b)) as we did Y(i,B(a,b)) with (R_i,v_iv_a) replaced by (v_iv_a,L_i) - See Figure 2.28 for an illustration of the difference. We define Y'(i,F(a,b)) by a similar process.



case Y(i,B(a,b))



case Y(i,(B(a,b),B(c,d)))



case Y′(i,B(a,b))

Figure 2.28: Computing Y(i,ARG) and Y′(i,ARG).

We are now ready to implement STEP 4 of the algorithm CD. We describe only the computation of B(i,j), the case F(i,j) being similar in the opposite direction. We compute the 4 sets B1,B2,B3,B4 as follows:

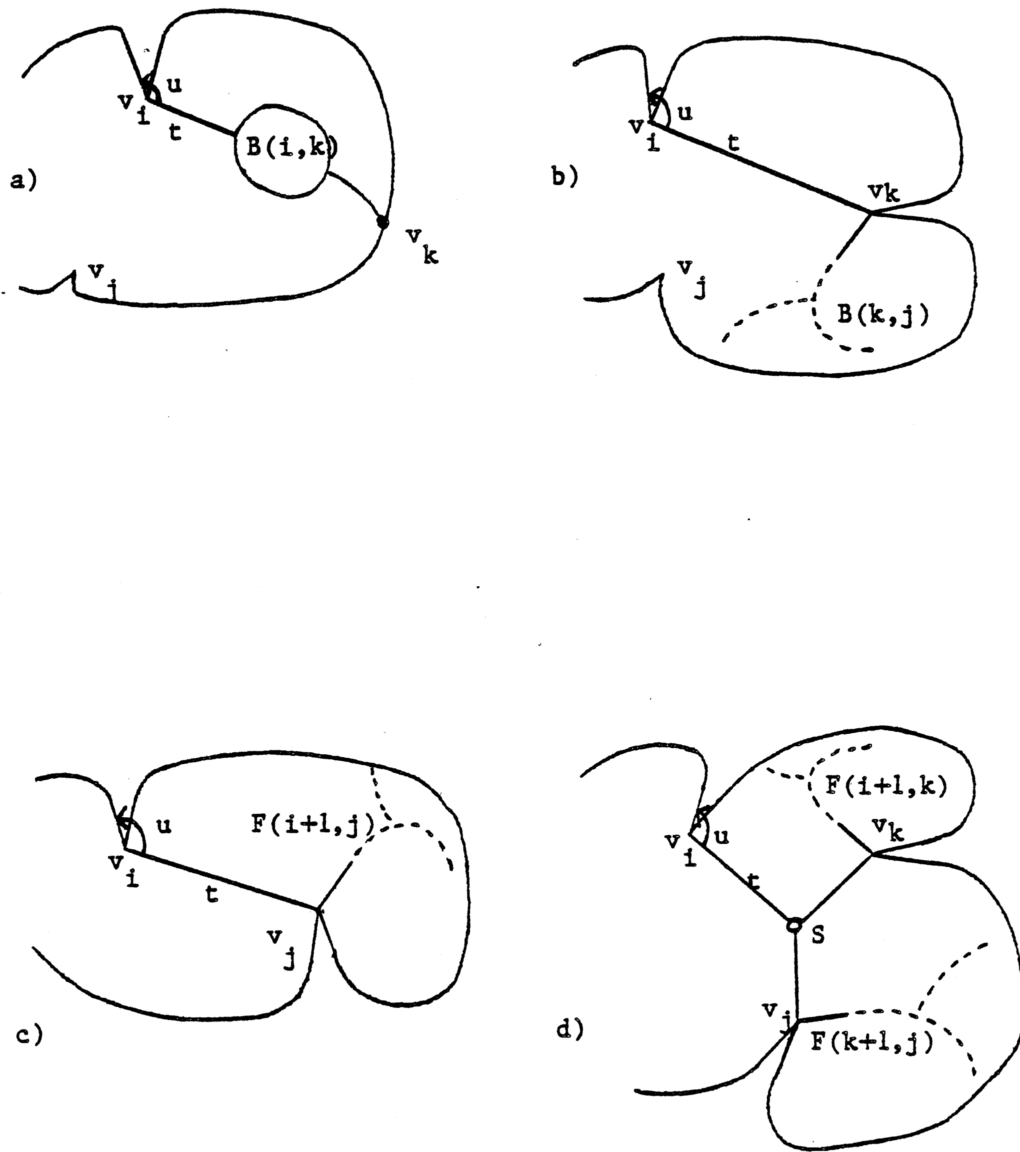
```
Let C be the value of |S(i,j)| computed in STEP 3.

B1= {Y-subtree of B(i,k)} for all v_k in V(i,j-1) s.t
    |S(i,k)|+|S(k+1,j)|=C

B2= {Y'(i,B(k,j))} for all v_k in V(i+1,j-1) s.t
    |S(i+1,k-1)|+|S(k,j)|=C

B3= {Y(i,F(i+1,j))} if |S(i+1,j)|=C

B4= {Y(i,(F(i+1,k),F(k+1,j)))} for all v_k in
    V(i+1,j-1) s.t |S(i+1,k)|+|S(k+1,j)|=C

Let T be the Y-subtree of B1UB2UB3UB4 which maximizes
the angle u=(t,L_i), where t is the edge of T which is
adjacent to v_i (Fig.2.29).

B(i,j)= (R(v_i,t),T)
```

We can now show that:

Lemma 8: STEP 4 computes B(i,j) and F(i,j) in O(N) time.

Proof: Once again, for reasons of symmetry, we only deal with B(i,j). Corollary 14 and Lemma 5 show that the Y and Y' functions can be evaluated in constant time (note that for this purpose, in addition to a Y-subtree, each B and F contains the segment R(v,t), where t is the edge of the subtree adjacent to v. Also, we should not tamper with the representation of the Y-subtrees used in the Y and Y' functions since they may have to be used later on. It is clear, however, that Y-subtrees can be merged in constant time with a simple system of pointers, while avoiding copying any of them.

B1 through B4 evaluate all possible Y-subtrees adjacent to v_i and lying in V(i,j), and



<u>Figure 2.29</u>: Computing $B(i,j)$.

keep a single candidate by maximizing the angle u (Fig.2.29). Once again, we can show by induction that it is legitimate to consider only the Y-subtrees in the B's and F's. B1 considers all the subtrees which do not connect v_i and v_j (Fig.2.29-a). To handle the other cases, B2 and B3 compute the subtrees whose vertex adjacent to v_i is an N2-node. The two possible configurations are illustrated in Figure 2.29-b,-c. Finally, if this vertex is an N3-node, B4 will detect all such candidates. □

2.3.6.6 Completing the OCD (STEP 5)

The last step of the algorithm CD consists of removing the remaining notches with the naive decomposition. Since an optimal set of compatible patterns involves a total of O(N) edges, the result of Theorem 4 still applies, and shows how to complete the decomposition in time $O(N^2\log(n/N))$. From previous results, we can collect the orders of magnitude of each step's running time.

preprocessing	:	$n+N^3+N^2\log n$
STEP 2	:	N^3
STEP 3	:	$N^3+N^2Xfour(n,N)$
STEP 4	:	N^3
STEP 5	:	$N^2\log(n/N)$

Thus, we can state our main result:

Theorem 16: The algorithm CD requires $O(n+N^3+N^2Xfour(n,N))$ operations.

Proof: It suffices to show that among n, N^3, and $N^2\log n$, the last term is never dominant, for n large enough. Suppose that n and N^3 are smaller than $N^2\log n$, then $N<\log n$, so $n<\log^3 n$, which is not possible for n large enough. □

At this stage, we know that

$$Xfour(n,N)=O(N^3\log(n/N))$$

which, combined with Theorem 10, leads to

Theorem 17: An OCD of P can be obtained under a graph or a polygon representation in $O(n+N^5\log(n/N))$ operations, using $O(n+N^3)$ storage.

Proof: It remains to show that $O(n+N^3)$ space is sufficient. This is obvious since all the trees and subtrees stored in B(i,j), F(i,j), and S(i,j) involve O(N) edges. □

2.4 An OCD Algorithm Cubic in the Number of Notches

2.4.1 Introduction

In the previous section, we have established an upper bound of $O(N^3\log(n/N))$ on Xfour(n,N), which led to an unsatisfying time bound. We now show how this may be reduced to O(N) time after $O(N^3)$ preprocessing, thus decreasing the total running time of CD to $O(n+N^3)$. Since there are potentially on the order of N^4 X4-patterns in P, we cannot compute all of them, and to achieve cubic time, we have to show that only $O(N^3)$ of them may be considered candidates. Theorem 9 states that all X-patterns not reducible to an X4-pattern can be reduced to Y-patterns. We can show that those "irreducible" X4 can be assumed to have a specific form, which we refer to as loose.

Definition 2.4: An X4-pattern is said to be loose if its edges can be moved so that each edge adjacent to a notch v_i lines up to R_i or L_i (16 configurations are possible) - See Figure 2.30.

We now formalize our earlier claim:

Theorem 18: Every X4-pattern which is not reducible to a Y-pattern can be reduced to a loose X4.

Proof: Call HULL(T) the convex hull of an X-pattern T. It is clear that every X4-pattern T can be reduced to an X4-pattern V such that no further reduction of V can lead to another X4-pattern lying strictly in HULL(V) (i.e., where at least one notch lies in the interior of HULL(V)). We show that if T cannot be reduced to a Y-pattern, V must be loose. Assume that one of the 16 configurations cannot be achieved for V (See Figure 2.31 for the situations where this can arise). Now, by applying the reductions given by the arrows of Figure 2.31, we either reach a Y-pattern or an X4-pattern lying strictly in HULL(V), contradicting our hypothesis. Actually, another possibility is to reduce to the alternate case of Figure 2.31, which can arise only a finite number of times. □

Because of this result, we can always assume that the X4-patterns we have to deal with in STEP 3 are loose and collinear with the right edge of each notch involved. More precisely, if S is a Steiner point adjacent to v_i, R_i is collinear with Sv_i. From here on, all the X4-patterns considered will be

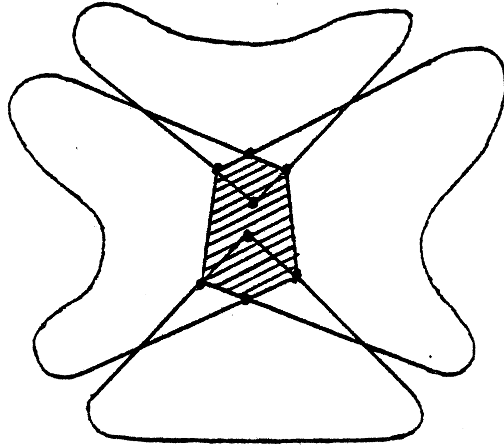


Figure 2.30: A loose X4-pattern with its 16 extreme configurations.

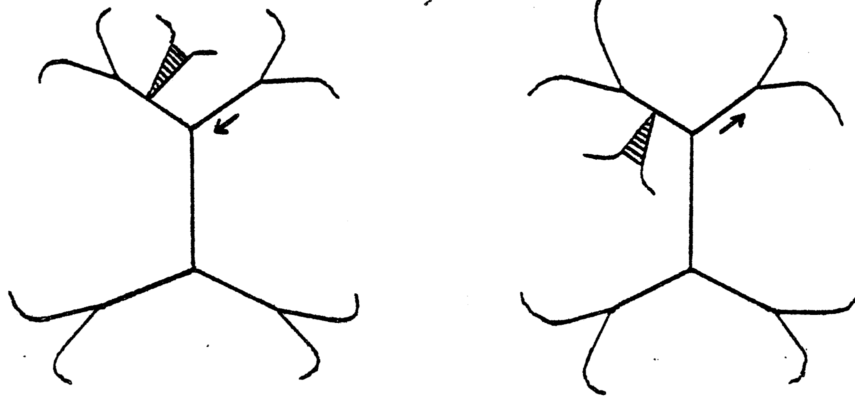


Figure 2.31: X4-patterns can be assumed to be loose.

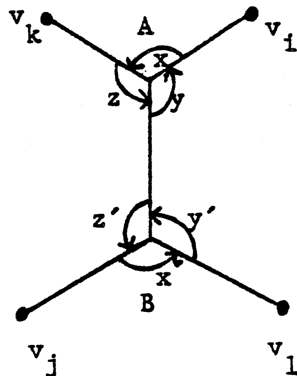


Figure 2.32: Characterization of an X4-pattern.

assumed to have this configuration. Before addressing our main problem, we give a basic characterization of X4-patterns which we will use throughout.

Lemma 9: v_k, v_i, v_l, v_j being 4 notches of P in clockwise order, the tree of Figure 2.32 forms an X4-pattern if and only if:

1. A,B are the only two intersections between edges.

2. Angles x,y,z,x',y',z' are less than 180 degrees.

3. No edges of the tree intersect with the boundary of P (except at the notches).

Proof: This characterization is fairly straightforward. It is important to notice, however, that if we consider the convex hull of the X4-pattern, a clockwise order of its 4 vertices corresponds to a clockwise order of the notches on the boundary of P. This topological fact will be used throughout. □

We represent this situation by the statement $X4(v_k,v_i,v_l,v_j,A,B)$. We will often use this statement with nodes replaced by * to represent the set of all possible X4 having the *ed elements filled in.

Next, we introduce some operations to be added to the preprocessing at STEP 1. Let r_i be the segment $R(v_i,R_i)$ and (a_p,b_p) be the pair of $SR(v_i)$ such that $a_p b_p$ intersects r_i (note that this pair has to be determined in order to compute $R(v_i,R_i)$ - See Lemma 5). For all v_k between b_p and v_i in clockwise order, compute A_{ik}, the intersection of r_i and r_k if it exists (recall that not only the intersection may not exist, but the segments r_u may be 0 - See the definition of the R function above). We always have:

1. $(A_{ik}v_i, A_{ik}, v_k) < 180$ and $A_{ik}v_i$ and $A_{ik}v_k$ intersect B(P) only at v_i and v_k respectively.

2. For each v_i, the set of A_{ik}'s contains all possible Steiner points adjacent to v_i of the loose $X4(*,v_i,*,*,*,*)$ of Figure 2.32.

Next, for each v_i, the points A_{ik} are sorted on r_i and maintained in a sorted list. In the following, r_i will be viewed either as a geometric segment or as a list of sorted points A_{ik}. The data structure chosen for r_i should allow constant time access to A_{ik} as well as 2-way scans of the list r_i. Note that

this computation can be done in $O(N^2 \log N)$ operations, and that the A_{ik} may not be defined for all i,k.

We wish to apply the idea of patching subtrees to the construction of X4-patterns. In the configuration of Figure 2.32, the edge v_kA is to be patched with the rest of the pattern. To generate X4-subtrees, we extend the concept of B's and F's, and define the sets E(i,j). Basically, E(i,j) will contain enough information to decide, in constant time, if for a given v_k, there exists a v_l such that $X4(v_k,v_i,v_l,v_j,*,*)$. E(i,j) will be computed after S(i,j) by considering all v_l between v_i and v_j and determining for each all the v_k that can be patched to form an X4-candidate.

Since, for each v_l, there are potentially on the order of N such v_k, we cannot even look at all of them, should E(i,j) be computed in O(N) time. Fortunately, for each v_l, we can express the corresponding sét of v_k by a single piece of data, which can be computed in constant time.

It remains now to formalize the intuition given above. E(i,j) is defined as the set of pairs (A_{ik},A_{jl}) with all k distinct such that $X4(v_k,v_i,v_l,v_j,A_{ik},A_{jl})$ with the added property that if an OCD contains a loose X4-pattern, $X4(v_k,v_i,*,v_j,*,*)$, then there exists an OCD containing $X4(v_k,v_i,v_l,v_j,A_{ik},A_{jl})$ where (A_{ik},A_{jl}) belongs to E(i,j). This allows the set E(i,j) to be used for our purposes without overlooking X4-candidates.

We next show that such sets can be found satisfying this property, and that each of them can be computed in O(N) time, given the preprocessing described above.

2.4.2 Computing E(i,j)

Recall that E(i,j) is to be computed after S(i,j).

I) "Selecting candidates on r_i and r_j"

To begin, we determine the points A_{ik} such that v_k belongs to V(j+1,i-1) (note that if the pair (a_p,b_p) of $SR(v_i)$ used to compute r_i is such that b_p belongs to V(j,i), all the vertices on r_i will already satisfy this condition).

Next, we keep only the A_{ik} which lie on the other side of the infinite line passing through R_j from the edge L_j. This ensures that:

3. v_j, v_k, v_i occur in clockwise order and the angle $z'<180$ (Fig.2.32).

We operate the same selection on the list r_j. Namely, we determine the A_{jl} such that v_l belongs to $V(i+1,j-1)$ and which lie on the other side of line (R_i) than L_i, thus ensuring :

4. v_i, v_l, v_j occur in clockwise order and $y<180$.

We now retain in r_j only points A_{jl} for which

$$|S(i,j)| = |S(i+1,l-1)| + |S(l+1,j-1)|$$

By doing this, we keep only the candidates for Steiner points of an OCD. Like the Y-subtrees in the $B(i,j)$, the candidates have to cause the same savings in $V(i,j)$, that is, $|S(i,j)|$.

Finally, we update the lists r_i and r_j with their respective points thus selected, maintaining the sorted order. Let us rename the points of r_i (resp. r_j) from v_i (resp. v_j), $A_1,...,A_p$ (resp. $B_1,...,B_q$). This entire step can be done in $O(N)$ time given the preprocessing, and yields a list of all possible Steiner points for optimal X4-patterns. It is now clear that $E(i,j)$ can be found satisfying the specifications given earlier. Each A_k must be paired with the B_l such that A_k and B_l are the Steiner points of the same optimal X4 and the angle (A_kB_l,A_kv_i) is maximum - See Figure 2.33. We must now give a precise procedure to accomplish this task.

II) "Computing a region of safety"

Next, we compute the "region of safety" for added edges to ensure condition 3. of Lemma 9. Since r_i and r_j do not intersect, neither do A_1A_p and B_1B_q. Moreover $A_1A_pB_1B_q$ forms a convex quadrilateral (see relations 3. and 4.). Our next step is to compute two convex chains $C=(c_1,...,c_s)$ and $D=(d_1,...,d_t)$ running from r_i to r_j and r_j to r_i respectively. These chains will have the property that a "middle edge" between r_i and r_j lies totally in P if and only if it lies totally between the two chains - See Figure 2.37-a. Informally, C (resp. D) is the convex hull of the pieces of B(P) which pass through A_1B_q (resp. A_pB_1), thus delimiting an area of safety for potential middle edges. To preserve the flow of the presentation, we will prove the following result in the next section.

Lemma 10: The convex chains C and D can be computed in $O(N)$ time after an $O(n+N^2\log n)$ time preprocessing.

When the procedure of Lemma 10 determines that any segment drawn between A_1A_p and B_1B_q intersects the boundary of P, it will set C or D to 0. Otherwise, it effectively returns two convex chains C and D with the segment c_1d_t (resp. d_1c_s) containing A_1 and A_p (resp. B_1 and B_q). Also, as stated

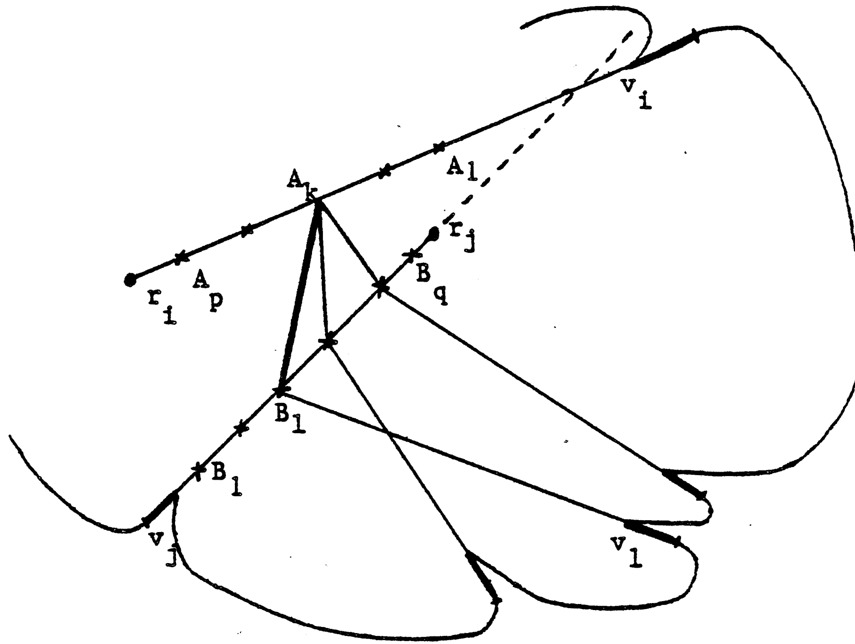


Figure 2.33: Computing E(i,j).

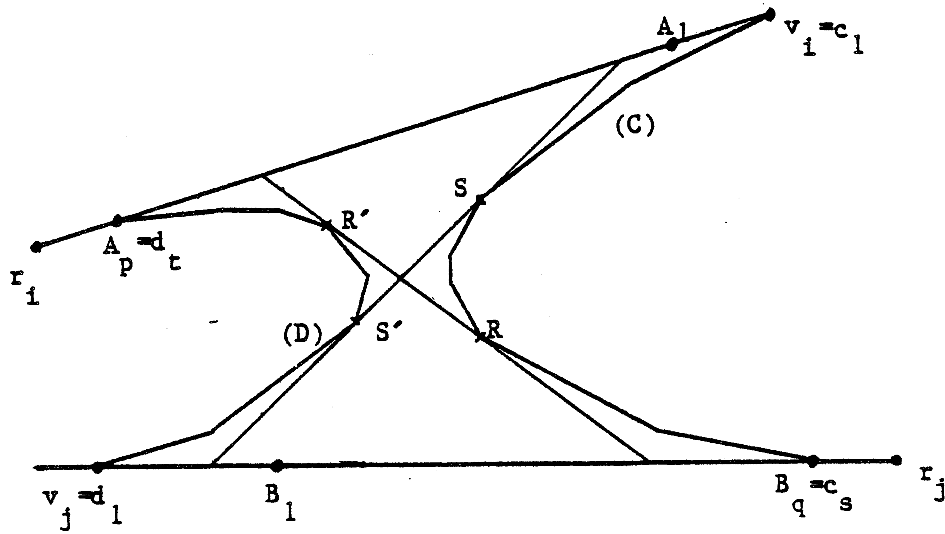


Figure 2.34: The limits on middle edges RR′ and SS′.

earlier, a segment joining A_1A_p and B_1B_q will not intersect the boundary of P if and only if it does not intersect C or D.

It will take O(N) operations to test whether C and D intersect, since both have O(N) vertices [Shamos,78]. In this case, no middle edges are possible and E(i,j) is 0. Otherwise, we can compute RR' and SS' as the limits put upon the middle edges by the polygon P - See Figure 2.34. SS' is computed by beginning at c_1 and d_1 and moving through D until all of D lies above line(d_kc_1), then moving through C until all of C lies below line(d_kc_l). We repeat until the process converges, which must occur after O(N) operations, since no vertex of C or D is visited more than once. Here is a more formal description of the procedure. RR' is computed in similar fashion.

<u>computing SS'</u>

```
k=l=1
while (d_k d_k+1, d_k c_l)<180 or
      (c_l c_l+1, c_l d_k)<180
   begin
         while (d_k d_k+1, d_k c_l)<180
                 begin k=k+1 end
         while (c_l c_l+1, c_l d_k)<180
                 begin l=l+1 end
   end
S=c_l, S'=d_k
```

III) "<u>Computing wedges for possible middle edges</u>"

We are now in a position to compute the wedges where middle edges must lie. We begin by observing that any point in the list r_i not lying between the intersections of these segments with the lines passing through RR' and SS' may be removed, which can be done in O(N) time. Although r_i is now a sublist of $A_1,...,A_p$, we still call its vertices A_1 through A_p for simplicity. Since the former list will no longer be used, there will be no ambiguity.

Now, for every B_l on r_j (recall that $B_l = A_{ju}$ for some u), we define f(l) as the intersection of line(r_i) with the half-line starting at B_l, collinear with R_u, and not passing through R_u. If this intersection does not exist, f(l) is 0 - See Figure 2.35. Also, for each A_k on the list r_i, let A'_k (resp. A''_k) denote the point on the segment r_j which is the closest to B_1 (resp. B_q) and such that the segment $A_kA'_k$ (resp.

$A_kA''_k$) does not strictly intersect D (resp. C) (i.e., the intersection consists of a segment or a single point). We now view r_j as a list of vertices and we find the two vertices B_l ($=g_1(k)$) and B_m ($=g_2(k)$) which lie on the segment $A'_kA''_k$ and are the closest to A'_k and A''_k respectively - See Figure 2.36. If B_l and B_m do not exist, we define $g_1(k)$ and $g_2(k)$ to be 0.

At this stage, we need two results which we will also prove in the next section.

Lemma 11: The functions f, g_1, g_2 can be set up to operate in constant time, and this can be done in O(N) operations.

Lemma 12: Let $V=\{1,...,q\}$ and $W=\{1,...,p+q\}$, $W_1 U W_2 = W$, $|W_1|=q$, $|W_2|=p$, and let f, g_1, g_2 be defined with: $f: V \to W_1$ a bijection, and $g_1, g_2: W_2 \to V$ non-decreasing, with $g_1(i) \leq g_2(i)$ for all i in W_2. Define $x: W_2 \to V$ such that for all i in W_2, x(i) is the smallest integer in V with $g_1(i) \leq x(i) \leq g_2(i)$ and $i \leq f(x(i))$.if such an integer exists and 0 otherwise.

If f, g_1, g_2 are computable in constant time, then so is x after O(p+q) preprocessing.

IV) "Computing E(i,j)"

Lemma 11 permits us to compute all the values of f, g_1, g_2 in O(N) time. Note that if f(l) is 0, no middle edge adjacent to B_l is possible since it must lie in the open triangle $(B_lB_q, B_lf(l))$. Therefore, we can eliminate those B_l from r_j. Once again, we still represent the resulting list by $B_1,...,B_q$. Similarly, if $g_1(k)=g_2(k)=0$, A_k cannot be a Steiner point and we eliminate all such A_k from r_i. Note that the values of g_1 and g_2 should be computed after that last selection on r_j. Then we merge the points f(l) with the remaining vertices A_k, forming the set W. Strictly speaking, f maps l not to a point on r_i but to the corresponding index in W. We can always assume f to be a one-to-one mapping. Also, we define V as the set of vertices left on r_j and g_1, g_2 will map k to the corresponding indices in V. Because of the removals, g_1 and g_2 obey the two conditions of Lemma 12. Finally, we define W_1 as the sublist of W corresponding to the f(l) and W_2 as the complement in W, that is, the indices corresponding to the A_k. It is easy to see that all the removals, merges, and settings of functions can be done in O(N) time. Moreover, all the conditions of Lemma 12 have been met.

Thus we can set up the function x in O(N) time. The last step consists of keeping in E(i,j) all the pairs $(A_k, B_{x(A_k)})$ such that $x(A_k)$ is not 0 with A_k in r_i and $(A_kv_u, A_kB_{x(A_k)}) < 180$ $(A_k = A_{iu})$. Note that $x(A_k)$ is a short-hand for x(t) with t the element in W_2 corresponding to A_k. Recall that if any of the previous computations fails, E(i,j) is 0.

We can now state our main result:

Theorem 19: It is possible to compute E(i,j) for any i,j; $1 \leq i,j \leq N$ in time O(N) after $O(n+N^2 \log n)$ preprocessing.

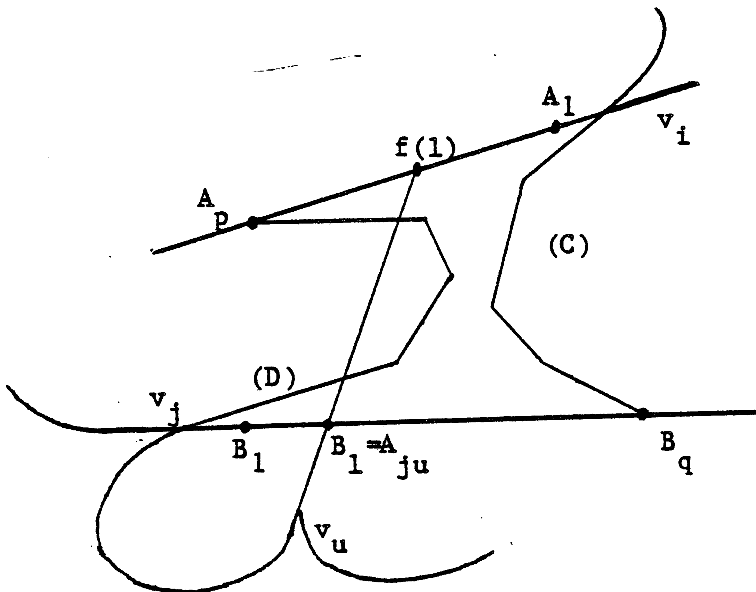


Figure 2.35: The function f.

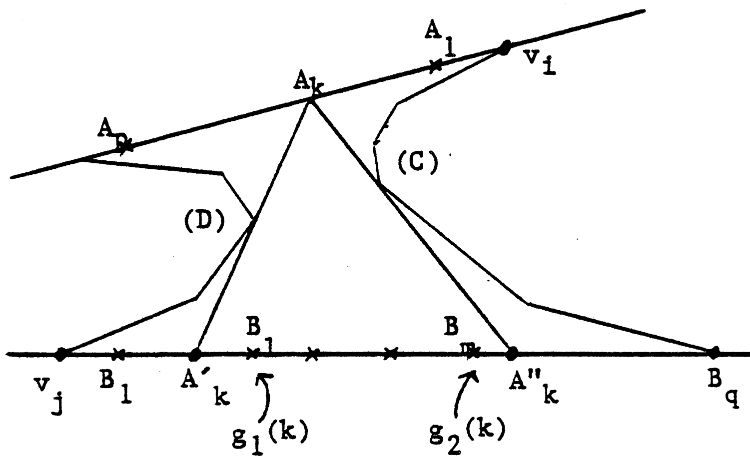


Figure 2.36: The functions g_1 and g_2.

Proof: The preprocessing involves setting up the lists r_i, which amounts to $O(N^2 \log N)$ time because of the necessary sorting, and for the need of Lemma 10, computing the superranges, which takes time $O(n+N^2 \log n)$.

Given i and j, E(i,j) is then computed in O(N) operations. We review the main phases of the procedure and establish its correctness. In the preprocessing stage, 1. ensures that the angles x and x' are <180, and that condition 1. of Lemma 9 is satisfied. For i,j fixed, 3. and 4. show that y,z'<180. Then, considering the savings, by an argument now classical, we eliminate the Steiner points which are not candidates. Next, we pair A_k on r_i with B_l on r_j ($l=x(A_k)$).

By definition, $x(A_k)$ is the smallest integer in V (i.e., the vertex of r_j that maximizes the angle y) lying on the segment $[g_1(k),g_2(k)]$ (i.e., ensuring condition 3 of Lemma 9), and such that A_k lies on the segment A_1F with F the point on r_i corresponding to $f(x(A_k))$ (i.e., ensuring y'<180). Finally, since $x(A_k)$ maximizes the angle y and y+z is a constant for all B_l, if z>180, no vertex of r_j can be paired with A_k. If $z \leq 180$, all the conditions of Lemma 9 are satisfied, and $A_kB_{x(A_k)}$ can be kept as the only middle edge candidate to be adjacent to A_k.

Since we know that this edge is indeed the middle edge of a loose X4-pattern, only savings considerations will later decide if this edge belongs to an OCD. This is a slight difference with the Y-subtrees of B(i,j) and F(i,j), where both the savings and the geometry had to be tested to determine their candidacy. □

2.4.3 The Proofs of the Lemmas Left Unresolved

We now justify our earlier claims and successively show how to compute the region of safety, and set up the functions f,g_1,g_2,x, all of this in O(N) time.

Lemma 10:

The convex chains C and D can be computed in O(N) time after an $O(n+N^2 \log n)$ time preprocessing.

Proof: C and D are clearly computed in exactly the same manner and we may give the details for C only.

We assume that all superranges have been precomputed, which requires $O(n+N^2 \log n)$ time as shown in Theorem 11. Let (a_p,b_p) be the pair of $SR(v_j)$ such that the segment a_pb_p intersects r_j. Recall that this pair is uniquely defined and must be computed to obtain r_j. If a_p is a notch, we let w be a_p, otherwise w is the notch of P next to a_p counterclockwise. We may always assume that adequate preprocessing allows to determine w in constant time, given a_p. If v_i lies between w and v_j in clockwise order, we simply define C as v_iB_q (Fig. 2.37-c). Otherwise things are somewhat more complex, and before describing an algorithm formally, we will give an overview of the method.

Let L be the line collinear with the edge of C adjacent to v_i (i.e., the first edge of C counterclockwise). L will essentially wrap around the obstacles created by the boundary of P in a counterclockwise motion - See Figure 2.37-d. Let V be the polygonal line on the boundary of P between v_i and w in clockwise order. We will show that all the "obstacles" (which are the vertices of C) are notches of V. Therefore we can expect to wrap around C entirely in O(N) operations if L can pivot around each vertex of C in constant time on average.

Let x be a vertex of C with L_1 (resp. L_2) designating the line L before (resp. after) pivoting around x - See Figure 2.37-d. We first locate L_1 in the superrange of x, then we scan SR(x) counterclockwise until we hit a vertex b_k which lies on V. We can show that in general b_k is also the next vertex of C. Recall that locating L in SR(x) involves finding the pair (a_j,b_j) such that L intersects a_jb_j. To ensure an O(N) running time, we cannot actually locate L_1 in SR(x). Instead, we determine a notch y nearby which will serve the same purpose. This notch is to be determined at the time when L_1 is computed. Thus we define

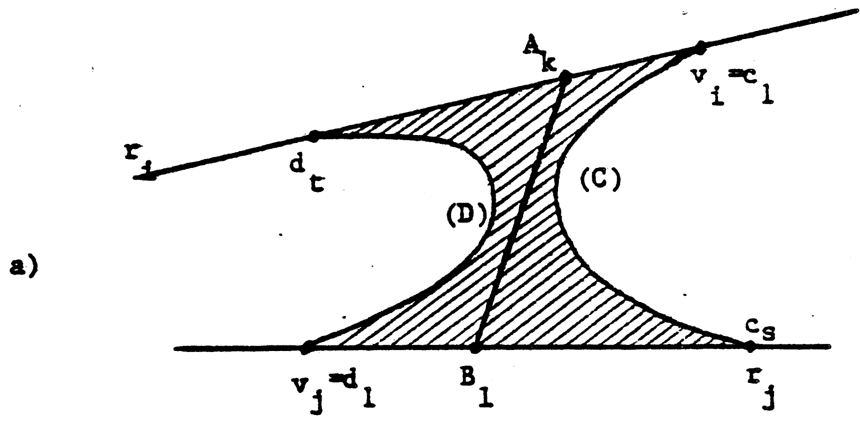
a function NEXT which maps (x,y) to (a_k,b_k). More generally, NEXT maps any pair of notches x,y (x in V) to the pair (a_k,b_k) of SR(x) computed as follows:

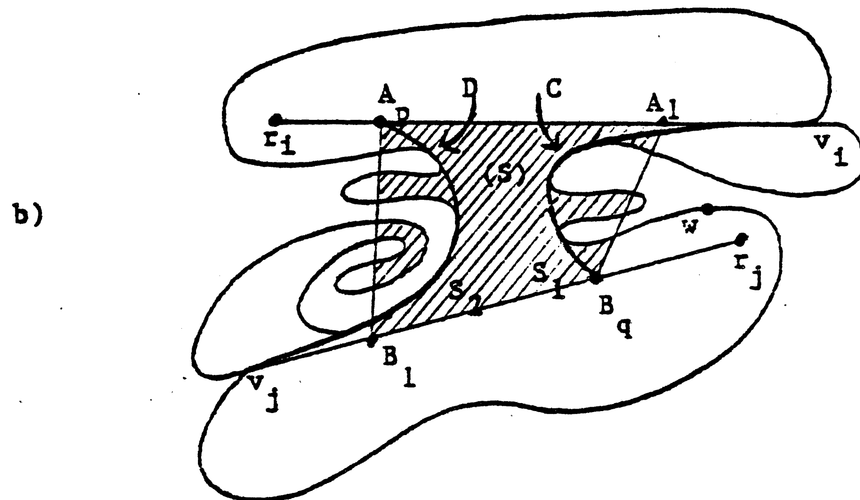
1. Find the two pairs (a_j,b_j) and (a_{j+1},b_{j+1}) of SR(x) such that y lies between b_j and a_{j+1} in clockwise order.

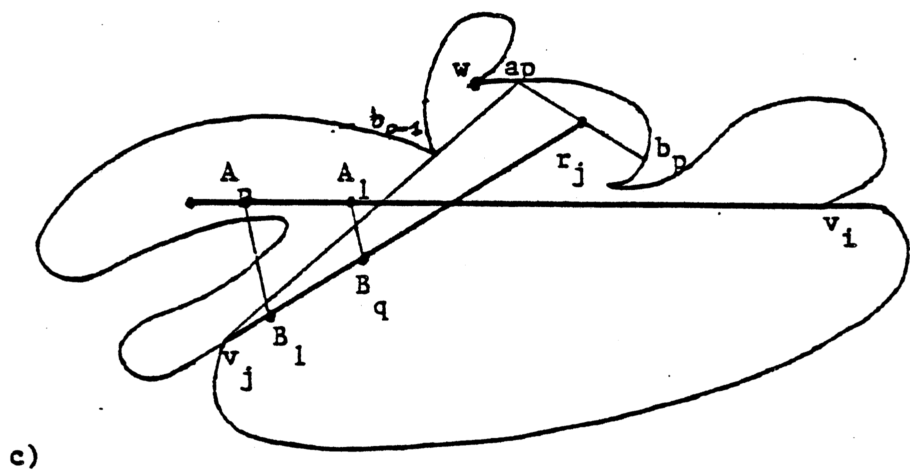
2. Scan the pairs of SR(x) counterclockwise starting at (a_j,b_j) (i.e., $(a_j,b_j),(a_{j-1},b_{j-1}),\ldots$) and determine the pair (a_k,b_k) such that b_k is a notch of V whereas a_{k+1} lies outside of V. If we fail to find such a pair, return(0).

3. When NEXT is evaluated and a pair (a_k,b_k) is actually returned, the function sets a global variable cnext to be a_{k+1} if it is a notch, else the notch of P next to a_{k+1} counterclockwise (Fig.2.37-d).

We first show that the function NEXT is well defined and can be evaluated in time proportional to the number of pairs scanned in step 2. Recall that in any superrange SR(x), the pairs (a_j,b_j) realize a partition of the set of all notches, and more precisely, any notch y lies between b_j and a_{j+1} in clockwise order, for some j. Thus, once SR(x) has been computed, we can extend the preprocessing to assign each notch of P to its corresponding pair b_j,a_{j+1}. A simple scan through the notches of P will do it in O(N) time. Finally, noticing that we can test if a notch lies in V in constant time, and that cnext is also found in constant time for reasons seen above, we achieve our claims above.







(Figure 2.37 .../...)

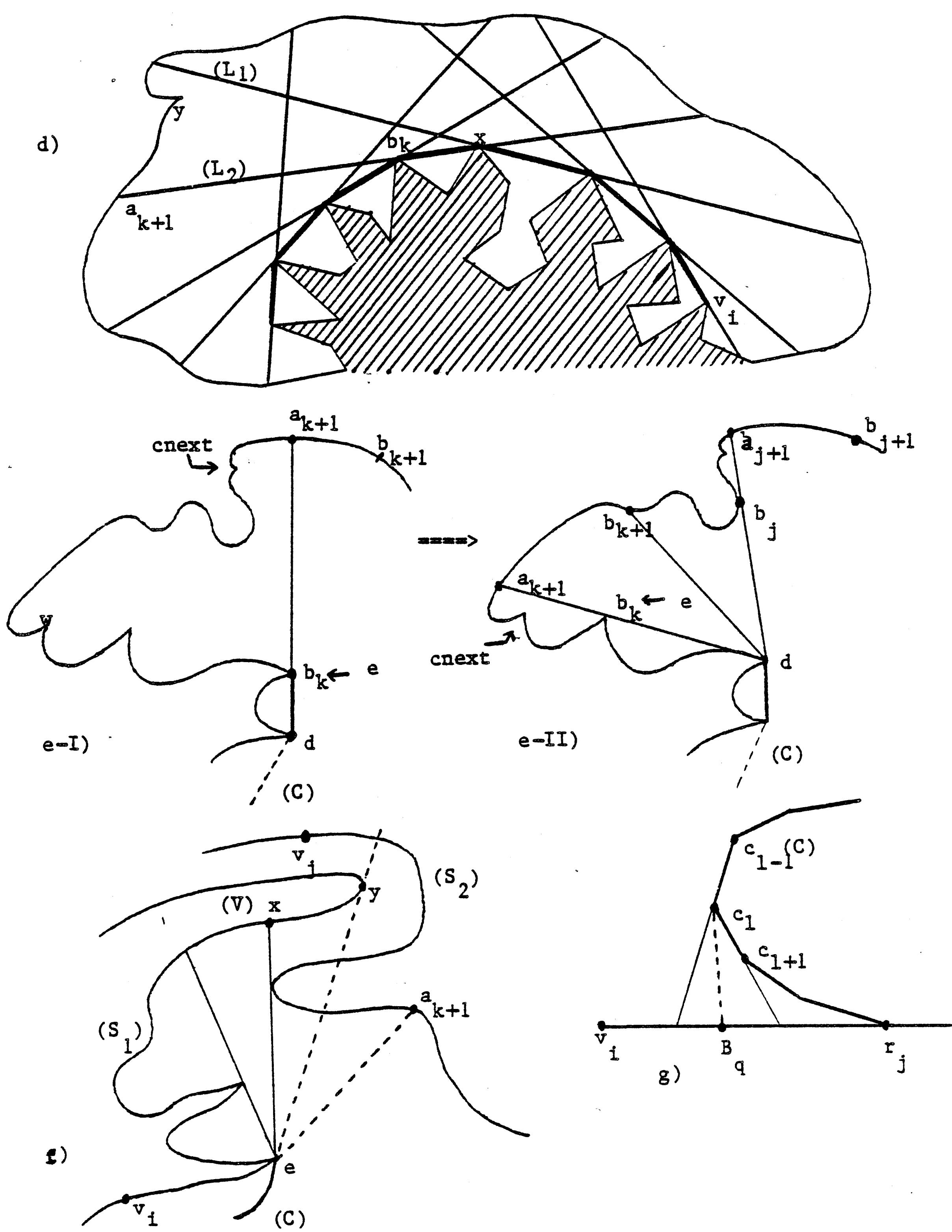


Figure 2.37: Computing the chains C and D.

We are now ready to set out the algorithm for computing C. Let a_1b_1 be the segment intersecting r_i with (a_1,b_1) in $SR(v_i)$, and t be the notch next to a_1 counterclockwise.

```
if v_i lies between w and v_j
      then  return(C=v_iB_q)
C={v_i},d=v_i,e=NEXT(v_i,t)
while e not 0
      begin
            if (dB_q,de)<180
                  then  return(C=CU{B_q})
            d= e
            C= CU{d}
            e= NEXT(e,cnext)
      end
return(C=0)
```

To see that the algorithm runs in O(N) time, it suffices to note that the notch cnext moves counterclockwise on the boundary of P, then a total of O(N) pairs (a_j,b_j) will be examined in all the superranges considered by the function NEXT.

We now show that a middle edge does not intersect the boundary of P if and only if it does not intersect C or D. We first observe an important fact of topology. In order to connect r_i and r_j with a middle edge lying totally in P, the intersection of P and the quadrilateral $A_1B_qB_1A_p$ must contain a polygon S with edges A_1A_p and B_1B_q. Note that this intersection may actually consist of several polygons. The boundary of S consists of these two segments joined by two polygonal lines belonging to the boundary of P, S_1 and S_2. S_1 has all its vertices in V and S_2 in V', where V' is defined as V, switching the role of v_i and v_j - See Figure 2.37-b.

To avoid computing S_1 and S_2 explicitly (they may have on the order of n vertices), we first notice that no middle edge can intersect the convex hull of S_1 without intersecting S_1.

The same observation on S_2 leads to computing the convex hulls C_1 and C_2 respectively. For convenience, we can replace the vertex A_1 in S_1 (resp. B_1 in S_2) by v_i (resp. v_j), and still preserve the initial property, that is, a middle edge lies totally in P if and only if it does not strictly cross C or D. We now turn to the actual computation of C and D.

The first case considered assumes that v_i lies between w and v_j in clockwise order. S is then reduced to the single intersection point of r_i and r_j and C can be set to v_iB_q - See Figure 2.37-c. Note that we can always assume that in this case r_i and r_j intersect, otherwise the lists r_i and r_j would be empty (see conditions 3. and 4. in Step I). If v_i does not lie between w and v_j, V is not empty, and we will prove by induction that C is actually the convex hull of S_1 or 0 if no middle edges are possible. Figure 2.37-e illustrates the computation of the next edge of C. To ensure convexity, all of S_1 must lie on the same side of the line passing through this edge. Therefore the next vertex of C after the vertex labelled e in Figure 2.37-e-I must be the point x of V, visible from b_k, which minimizes the angle (ea_{k+1},ex). b_k and a_{k+1} are the vertices in SR(d) returned by the previous call on NEXT.

Since the endpoints of V are notches of P, x must be one of the vertices listed in SR(e), and, to obtain it, it suffices to locate the pair (a_{j+1},b_{j+1}) in SR(e) between which a_{k+1} lies, then start a scan through SR(e). We only have to perform a counterclockwise scan since, by induction hypothesis, a_{k+1} does not lie in V, therefore the next vertex of C must be some a_l or b_l in SR(e) for $l \leq j$. Once again, the crux is that a counterclockwise scan in SR(e) corresponds both to a counterclockwise scan through the vertices of P and a counterclockwise angular sweep. Note that a_{k+1} is a point of SR(d) which has to be located in SR(e). Since a_{k+1} is not a notch in general, this operation seems too complex.

Instead, we can find the pair b_j,a_{j+1} of SR(e) between which cnext lies (recall that cnext is the first notch after a_{k+1} in a counterclockwise scan through the boundary of P). Since cnext is a notch, this operation takes constant time. Thus the function NEXT will return the next vertex of C. Note that if the point determined by NEXT is not a notch, NEXT

returns 0. Indeed, this point could not be the next vertex of C and the actual next vertex y would not be visible from e. Consequently, C would intersect S_2 and no middle edge would then be possible - See Figure 2.37-f. We observe that if C is well defined, there exists l such that B_q lies in the open triangle $(C_{l-1}C_l, C_lC_{l+1})$ - See Figure 2.37-g. Thus, the algorithm terminates by substituting $C_{l+1}, C_{l+2}, \ldots$ and the remaining vertices of C by B_q, which is clearly legitimate since this last portion of C cannot have any effect on middle edges. □

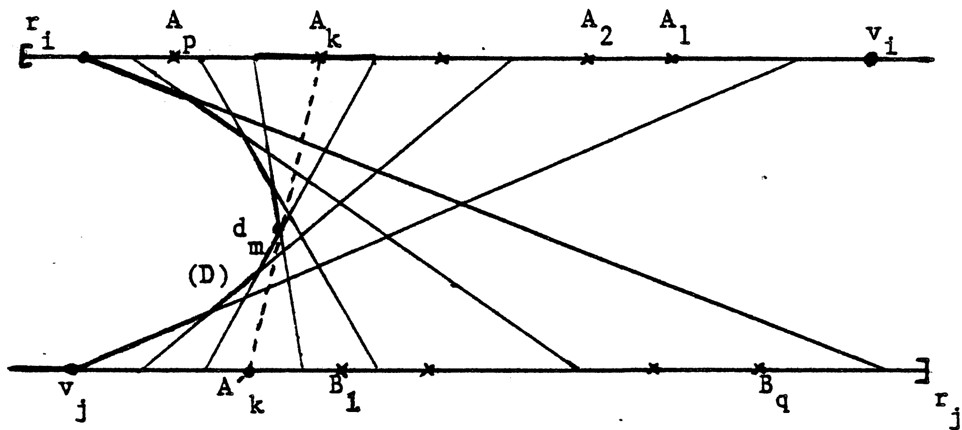
Lemma 11:

The functions f, g_1, g_2 can be set up to operate in constant time, and this can be done in O(N) operations.

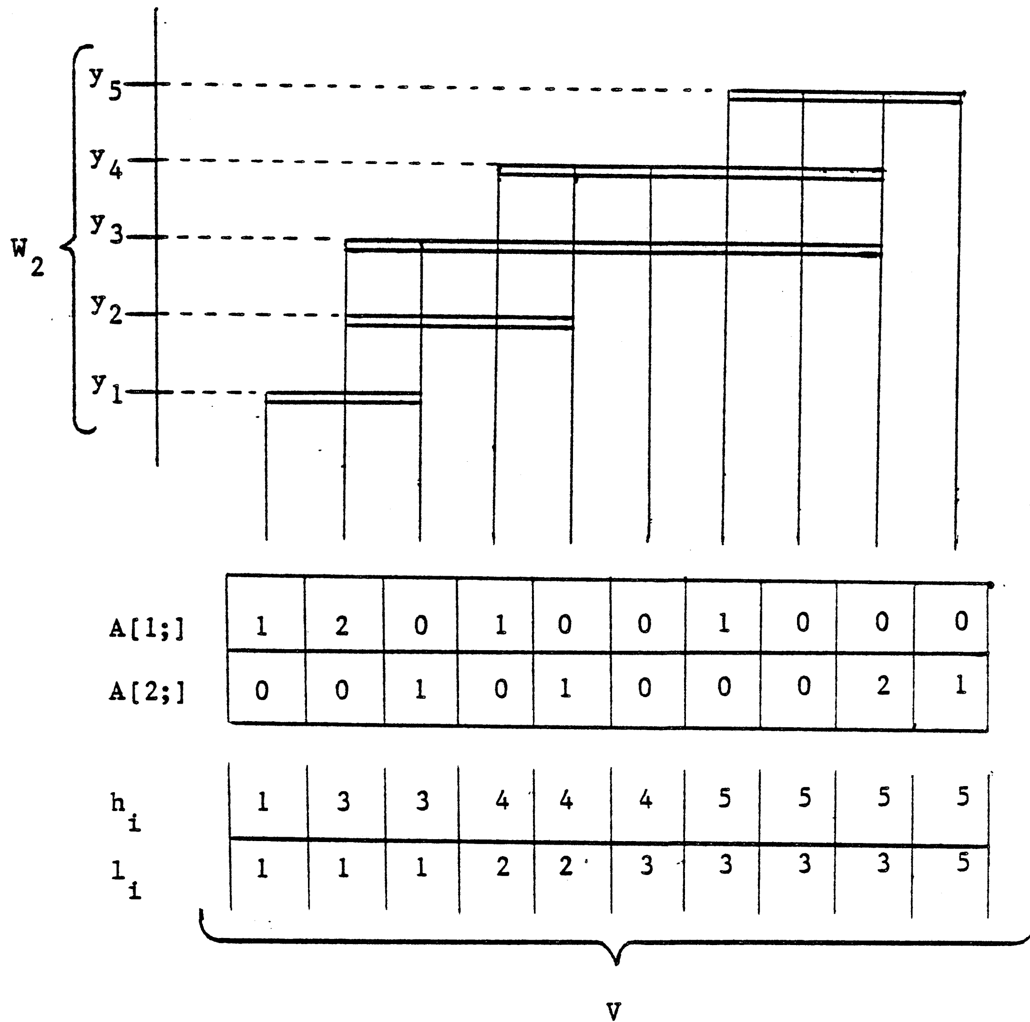
Proof: f(l) can be evaluated directly in constant time by intersecting line(r_i) with the half-line adjacent to B_l, collinear with R_u yet not passing through R_u with $B_l = A_{ju}$. If there is no intersection, f(l) is 0. Next we show how to compute all the values of g_1 and g_2 in O(N) time. We start by computing the intersection of r_i with all the lines $(d_k d_{k+1})$ for consecutive values of k. These points partition r_i into segments, and the previous computation provides a sorted list of their endpoints in O(N) time. To each of these segments corresponds a unique vertex of D. Then, for $A_1, \ldots, A_p$ in turn, we find the segment where A_k lies. Let d_m be the corresponding vertex of D. We compute A'_k by intersecting r_j with line($A_k d_m$) - See Figure 2.38. This also gives us a sorted list of the points A'_k since D is convex. Finally, we can merge the A'_k and B_l in O(N) time, and in one scan through the list, find for each A'_k the nearest B_l on the same side as B_q. We then set $g_1(k)$ to l. We repeat this process with respect to C, defining $g_2(k)$ for each A_k on r_i. Finally, for each A_k, we check that $g_1(k) \leq g_2(k)$. If it is not the case, we set $g_1(k)$ and $g_2(k)$ to 0. All of this clearly requires O(N) time. □

Lemma 12:

Let $V = \{1, \ldots, q\}$ and $W = \{1, \ldots, p+q\}$, $W_1 \cup W_2 = W$, $|W_1| = q$, $|W_2| = p$, and let f, g_1, g_2 be defined with: $f: V \to W_1$ a bijection, and $g_1, g_2: W_2 \to V$ non-decreasing, with $g_1(i) \leq g_2(i)$ for all i in W_2. Define $x: W_2 \to V$ such that for all i in W_2, x(i) is the smallest integer in V with $g_1(i) \leq x(i) \leq g_2(i)$ and $i \leq f(x(i))$ if such an integer exists and 0 otherwise.



<u>Figure 2.38</u>: Computing g_1.



A[1;]	1	2	0	1	0	0	1	0	0	0
A[2;]	0	0	1	0	1	0	0	0	2	1

h_i	1	3	3	4	4	4	5	5	5	5
l_i	1	1	1	2	2	3	3	3	3	5

<u>Figure 2.39</u>: The setting of the function x.

If f,g_1,g_2 are computable in constant time, then so is x after O(p+q) preprocessing.

Proof: Note that the naive method for computing all the values of x runs in O(pq) time. We present an O(p+q) time algorithm for achieving all these computations and establish its correctness.

Let $y_1,...,y_p$ be the elements of W_2 in increasing order ($1 \le y_j \le p+q$). First we consider the set of y in W_2 such that $g_1(y) \le i \le g_2(y)$ for a given i between 1 and q, and observe that it is a contiguous (possibly empty) subset of W_2 since g_1 and g_2 are non-decreasing. We compute the largest y, y_{h_i}, and the smallest y, y_{l_i}, as follows (if there is no such y, we set (l_i,h_i) to 0):

```
Initialize an array A (2 by q) to 0.
for i=1,...,p
     begin
          A[1,g1(yi)]= A[1,g1(yi)]+1
          A[2,g2(yi)]= A[2,g2(yi)]+1
     end
l=1,h=0
for i=1,...,q
     begin
          h=h+A[1,i]
          if l≤h
               then (li,hi)=(l,h)
               else (li,hi)=0
          l=l+A[2,i]
     end
```

The algorithm clearly achieves a time bound of O(p+q). To establish its correctness, we observe that the first loop sets A[1,j] to the number of y in W_2 such that $j=g_1(y)$, that is, the number of intervals $[g_1(y),g_2(y)]$ starting at j. Similarly, A[2,j] counts the number of intervals finishing at j. Then since, as i increases, $g_1(y_i)$ and $g_2(y_i)$ cannot decrease or pass each other, we can derive (l_i,h_i) from (l_{i-1},h_{i-1}) by counting the number of intervals which have to be added and removed. More precisely, the difference h_i-h_{i-1} is exactly the number of y in W_2 such that $i-1<g_1(y) \le i$, which is equivalent to $g_1(y)=i$ and shows that this number is A[1,i]. Likewise, if $i-1<g_2(y_{l_{i-1}})$ we have $l_i=l_{i-1}$. Else if $g_2(y_{l_{i-1}})=i-1$, l_i-l_{i-1} is the number of y such that $g_2(y)=i-1$, that is, A[2,i] - See an example in Figure 2.39.

We are now ready to set out the function x by computing all its values. If $y_{l_1} \le f(1) \le y_{h_1}$,

all x(i) with i between y_{l_1} and f(1) must be set to 1. Now if $y_{l_1} \le y_{h_1} \le f(1)$, all x(i) with i between y_{l_1} and y_{h_1} must be set to 1. In both cases, no other i in W should have x(i) equal to 1. Then we can carry out the same reasoning with 2, assigning this value only to the x(i) which have not been set yet. Since as i increases from 1 to q, l_i and h_i cannot decrease or pass each other (unless $(l_i,h_i)=0$), a possible implementation is:

```
Initialize all x(i) to 0 for all i=1,...,q
M=0
for i=1,...,q
      begin
          a= Max(y_l_i,M)
          b= Min(f(i),y_h_i)
          if a≤b and (l_i,h_i) not 0
              then
                  {for j=a,...,b
                   begin x(j)=i end
                   M=b+1}
       end
```

Note that if i belongs to W_1, the value assigned to x(i) has no meaning. The algorithm runs in time O(p+q), which completes the proof. □

2.4.4 The Cubic Algorithm

We are now prepared to use the information contained in E(i,j) to produce an OCD. E(i,j) may be computed in STEP 4 of the algorithm CD, with the additional preprocessing described earlier. We can now replace the former computation of B in STEP 3 by the following:

- Initialize B as the empty set.

- For all v_k in V(i+1,j-2) such that E(k,j) contains a pair (A_{ki},B_{jl}), assign to B the maximum (with respect to the cardinality) of B and:

  $X4(v_i,v_k,v_l,v_j,A_{ki},B_{jl})$

  U S(i+1,k-1) U S(k+1,l-1) U S(l+1,j-1).

- For all v_k in V(i+2,j-1) such that E(i,k) contains a pair (A_{ij},B_{kl}), assign to B the maximum of B and:

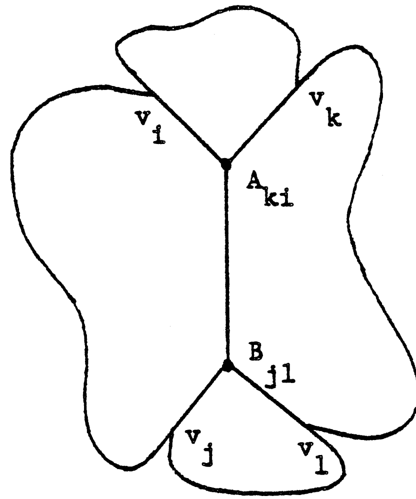
  $X4(v_j,v_i,v_l,v_k,A_{ij},B_{kl})$

  U S(i+1,l-1) U S(l+1,k-1) U S(k+1,j-1).

This allows us to consider both possible topologies for an X4-pattern lying in V(i,j) and adjacent to v_i and v_j - See Figure 2.40. This also shows that B can be computed in O(N) time, hence Xfour(n,N)=O(N). Since the additional preprocessing takes $O(n+N^2 \log n)$ time (Theorem 19), Theorem 16 shows that the total running time of the new CD algorithm is $O(n+N^3+N^2 \log n)$, which has been shown to be $O(n+N^3)$ in the proof of Theorem 16. This leads to the main result of this chapter.

Theorem 20: An OCD of P can be obtained under a graph or a polygon representation in $O(n+N^3)$ operations, using $O(n+N^3)$ storage.

REMARK: It is worthwhile to notice that all S(i,i-1) give optimal yet possibly different decompositions.



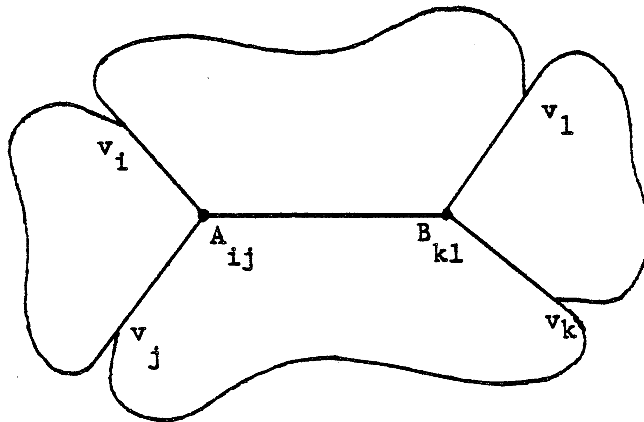


Figure 2.40: Computing B with the topologies of X4-pattern.

2.5 Conclusions and Open Problems

Our main result is an algorithm for decomposing a polygon into a minimum number of convex parts. The algorithm is linear in the number of vertices and cubic in the number of notches, which is fairly acceptable, given the near-convexity of most polygons arising in practice. Unfortunately, the algorithm seems inherently involved and to implement it in its most elaborate form is certainly a formidable task. We might be willing, however, to sacrifice a little efficiency to achieve greater simplicity. Computing only X2-patterns and doing away with superranges may often be found an acceptable compromise. Even the naive decomposition efficiently implemented ($O(n+N^2\log(n/N))$) may turn out the best alternative if we can afford to miss an optimal solution by at worst a factor of two in the number of convex parts.

Of course, only the cubic algorithm reveals the genuine beauty of the problem. Its long development involves solving many subproblems, most of which are interesting in their own right and deserve special attention. For example, the concept of superrange seems very fruitful. It is a very effective means for dealing with visibility problems in general and its fast computation ($O(N\log n)$) makes it very appealing. Recall that it involves a linear preprocessing which consists of partitioning the boundary of the polygon into convex chains. This can be viewed as a first step towards adapting non-convexity to algorithms for convex designs.

Indeed, it is frequent that tasks on polygons only involve dealing with boundaries. In that case, this simple preprocessing is often enough to improve efficiency. Finding the closest pair of points (A,B) with A and B lying on distinct polygons is one such example. For this problem, testing all pairs of chains and applying the ideas of Chapter 4 may be faster than applying the standard algorithm [Shamos,78]. However, lots of problems view polygons inherently as two-dimensional domains rather than closed lines, and thus require the sophisticated treatment of our decomposition procedures. For example, the applications in tool design and pattern recognition mentioned in the introduction are of this nature.

One novelty of the analysis we have given is the introduction of a new complexity measure which accounts for non-convexity. The number of notches of a polygon P expresses how far P is from being convex. If n and N were to be of the same order of magnitude, we could have simplified our analysis greatly, still ending up with polynomial yet impractical algorithms. The distinction between the size of the input and N, the non-convexity measure, led to efficient procedures on the grounds that the former quantity greatly dominates the latter. This is justified from practice and suggests that such a distinction should be made in many other geometric analyses. This would involve submitting current algorithms to such minor modifications like rewriting a polygonal line as a union of convex chains and exploiting the ideas of Chapter 4 to speed up computation.

Other open problems more closely related to the decomposition problem include rewriting polygons as sums and differences of convex parts. First the problem needs to be stated in a more rigorous manner, which might involve some simplifying hypotheses. One version of the problem consists of expressing a polygon as a difference A-B, where A and B are sets of polygons. The goal is then to decompose both A and B into a minimum number of convex parts. Given the most intricate nature of this problem, we believe that only simple efficient heuristics should reasonably be sought.

Chapter 3
DECOMPOSING A POLYHEDRON

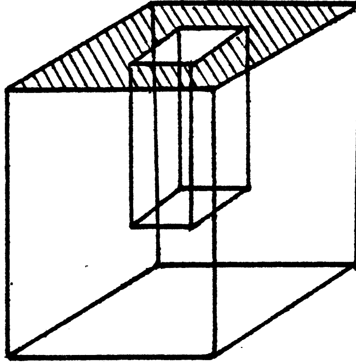
3.1 Introduction

Generalizing the decomposition to three dimensions introduces a host of new difficulties which rule out simple extensions of the decomposition algorithms presented in the previous chapter. Actually, the problem becomes so intricate that polynomial-time procedures seem beyond reach. Efficient heuristics, however, might be contemplated with the hope of producing near-optimal decompositions. Recall that in two dimensions, the naive decomposition produces at most twice as many pieces as the optimal, and might often be found an acceptable compromise, given its simplicity.

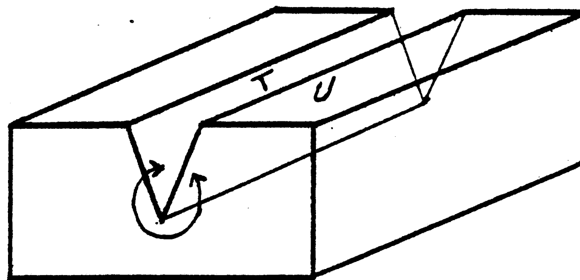
Trying to extend the naive algorithm to three dimensions will give a foretaste of the new difficulties we must face with an additional dimension. Efficient decompositions may yet be achieved and we will proceed to describe them after setting up our notation and addressing the basic question of computer representation of three-dimensional objects.

We define a three-dimensional polyhedron as a finite, connected set of simple plane polygons, such that every edge of each polygon belongs to exactly one other polygon. To exclude degenerate cases (e.g, two cubes connected by a single vertex), we also require that the polygons surrounding each vertex form a simple circuit [Coxeter,73]. Note that this definition does not prevent faces from having holes - See Figure 3.1-a. A face with k holes is said to be of genus k. Similarly, polyhedra may have holes (i.e., "handles"), and we define the genus of a polyhedron as the genus of the surface formed by its boundary [Massey,67]. Note that from the definition polyhedra may not be "hollowed out", that is, have interior boundaries.

Let P be a polyhedron with n vertices $v_1,...,v_n$, p edges $e_1,...,e_p$, and q faces $f_1,...,f_q$. Vertices, edges,



a) A face of a polyhedron with a hole in the middle.



b) A notch of a polyhedron.

Figure 3.1

and faces are said to be adjacent if they have at least one common point. For simplicity, however, we will say that a face and an edge or two faces are adjacent only if they share an entire line segment. When the intersection is reduced to a single point, they are said to be vertex-adjacent. If T and U are two adjacent faces intersecting in a segment L, we define the angle (T,U) as the angle between two segments lying respectively on T and U and perpendicular to L. Recall that there is no natural orientation of angles in euclidean space. Thus, to avoid ambiguity, the angle (T,U) will always be measured between 0 and 360 degrees with respect to a given side of the pair T,U. For example, we can easily extend the notion of notch to three dimensions by first noticing that each face of P has an outer and an inner side. Then we can define a notch as an edge of P with its adjacent faces forming a reflex angle with respect to their inner side - See Figure 3.1-b. Once again, N denotes the number of notches, $g_1,...,g_N$.

As in two dimensions, it is easy to see that the presence of notches is a characteristic of non-convexity [Coxeter,73]. Thus we can view a convex decomposition of P either as a partition of P into convex polyhedra or as a set of "cuts" through P which resolve the reflex angles at the notches. This leads to a trivial extension of the naive decomposition of polygons. We describe the method and analyze its performance in the next section.

3.2 The Naive Decomposition

As in the planar case, a notch can be removed by cutting along a plane adjacent to it so as to resolve the reflex angle between its adjacent faces. More precisely, let g be a notch of the polyhedron P with f_i and f_j its adjacent faces, and let T be a plane passing through g such that the angles (f_i,T) and (T,f_j) measured from the inner side of f_i and f_j are not reflex. The intersection of T and P is in general a set of polygons. These polygons may have holes and the holes may themselves contain other polygons - See Figure 3.5-a. Let S be the unique polygon containing g. We call S a cut of the naive decomposition. It is clear that cutting along S will remove the notch g. Note that in general this operation will break P into two pieces. If P has a non-zero genus, however, removing a notch may simply cut a handle of P and preserve its connectivity. In that case, the polyhedron obtained has two distinct faces with the same geometric location - See Figure 3.2-a. Other intriguing effects may be observed and it is worthwhile to mention some of them.

If the polygon S has holes, removing g may create a handle in the single or in either of the two parts produced - See Figure 3.2-b. Therefore the added genus of all the pieces produced thus far will increase by one. From the previous remarks, we also observe that the operation may preserve one piece while removing a handle and creating another - See Figure 3.2-c. In our analysis we will often deal with polyhedra of arbitrary genus, since the naive decomposition may produce intermediate polyhedra of non-zero genus.

In spite of those intricacies, we can easily show that repeating this process on each non-convex part will produce a convex decomposition in a finite number of operations. To find out how many convex parts such a decomposition may generate, we first observe that, at any time, any notch of a part is either a notch of P or a subsegment of a notch of P (subnotch). This results from the fact that a cut may intersect other notches thus "duplicating them", while no new notches are ever created. At worst, each cut will intersect all the other notches or subnotches present in the polyhedron considered. If f(N) is the maximum number of cuts which may have to be made to complete the decomposition, then

$$\begin{cases} f(0) = 0 \\ f(n) \leq 2f(N-1)+1 \end{cases}$$

Therefore, at most 2^N-1 cuts are needed, which shows that the procedure will always converge and produce at most 2^N convex parts.

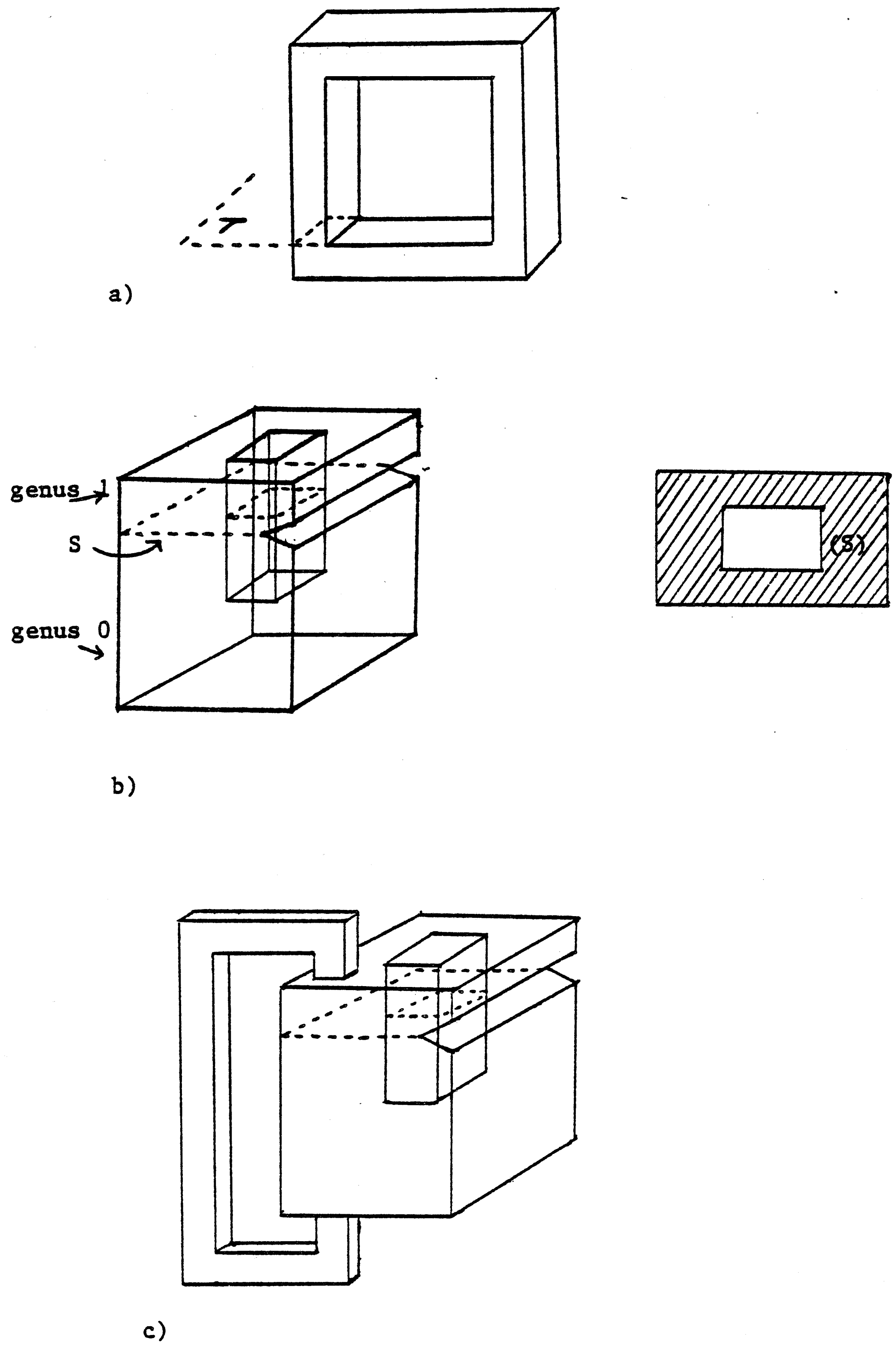


Figure 3.2: Removing a notch.

3.2.1 An Exponential Blow-up

The strategy we have adopted so far is a straightforward extension of the naive decomposition in two dimensions. Unfortunately, the analogy does not go further, especially when one proceeds to count the number of convex pieces which might be generated in the worst case. Indeed, we can show that the upper bound of 2^N established in the previous section is actually tight.

Lemma 13: There exists a class of polyhedra for which the naive decomposition produces 2^{cN} convex parts for some constant c.

Proof: Figure 3.3-a gives the general shape of the polyhedra P_n. P_n is essentially a pentahedron with 2n notches parallel to the yz-plane, $a_1,b_1,...,a_n,b_n$ for decreasing values of x. All the b_i's lie on the hyperbolic paraboloid $L_1T_2L_2T_1$ and the x-coordinates of the b_i lie at regular intervals. The hyperbolic paraboloid, denoted H, is generated by the set of lines passing through L_1 and L_2 and parallel to the yz-plane, or similarly by the set of lines passing through T_1 and T_2 and parallel to the xy-plane [Protter and Morrey,70]. For simplicity, we have only represented the notches b_i in Figure 3.3-a.

We can imagine the notches of P to be carved on the xz-face of the pentahedron as follows: We first cut through the triangle B_ib_i, where B_i is the intersection of the x-axis with the plane passing through b_i and parallel to the yz-plane. Then A_i being chosen on the x-axis, very close to B_i, we define the notch a_i as the intersection of P with the plane passing through b_i and L_1 and the plane passing through A_i and parallel to the yz-plane - See Figure 3.3-b. This allows us to carve in the main pentahedron along the faces thus defined, removing a very thin hexahedron and forming the two notches a_i and b_i.

Since the face passing through a_i and b_i is parallel to L_1, the notch b_i can be removed with a cut intersecting the segment T_1 at any point. In particular, we can remove b_1 with a cut S intersecting the middle point I of T_1 - See Figure 3.3-c. Let J be the middle point of b_1. Since IJ is parallel to the xy-plane and both I and J lie on the hyperbolic paraboloid H, IJ is collinear with a generator of H, that is, lies entirely in H. Therefore the cut S intersects H in a straight line and "duplicates" all the remaining notches $b_2,...,b_n$. The two

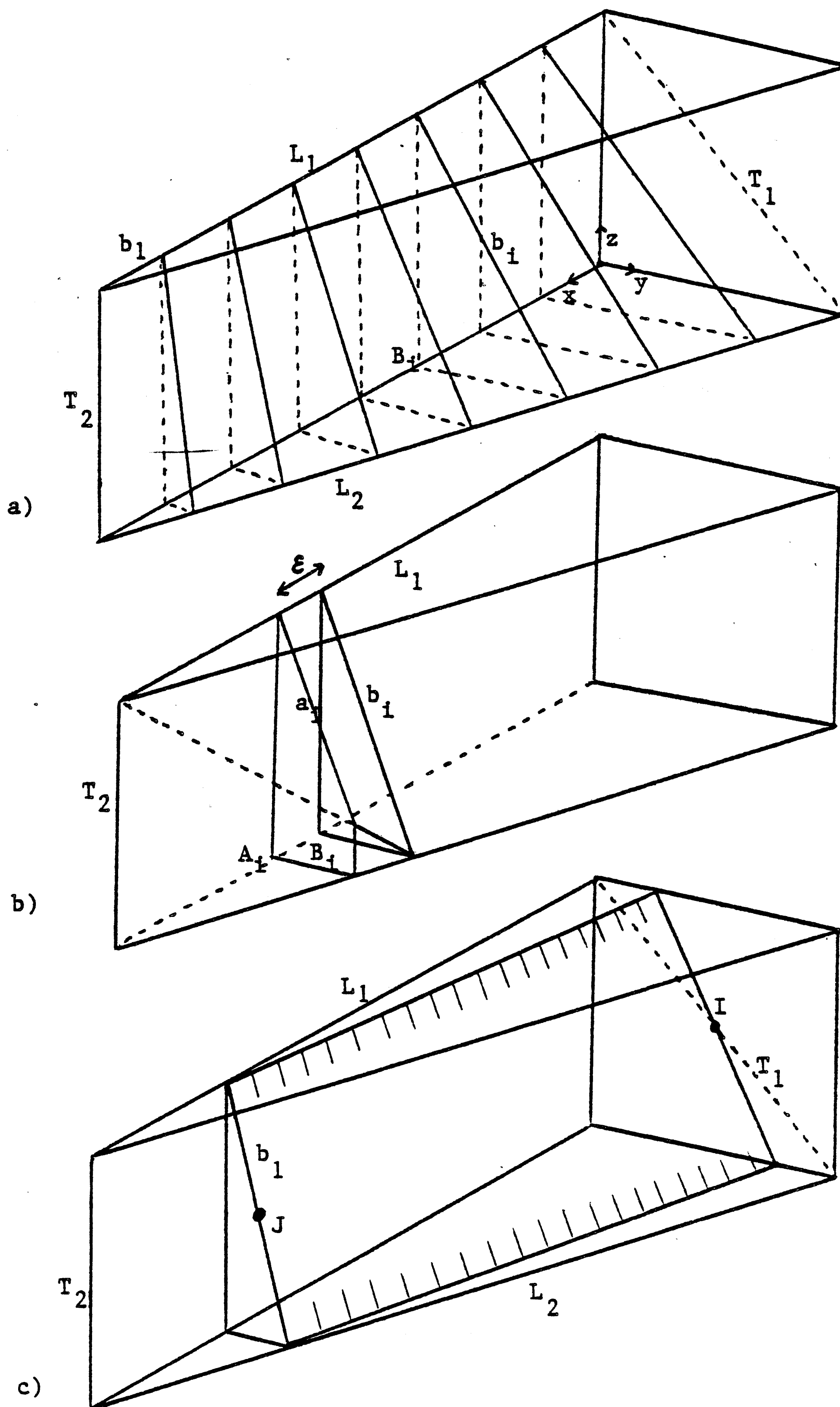


Figure 3.3: A polyhedron decomposed into an exponential number of convex parts.

resulting pieces can be decomposed with the same method, starting at b_2. Iterating on this process will remove all the b_i while generating at least 2^N polyhedra. The a_i notches are finally removed, which yet increases the number of parts and completes our proof. □

3.2.2 The Naive Decomposition Revisited

Although an exponential blow-up is perhaps unlikely in general, the previous method is unacceptable and must be revised. Recall that the naive decomposition of polygons produces at most N+1 convex parts (Theorem 1). An easy fix consists of removing all the subnotches of a notch with the same angle. This will ensure that all the cuts used to remove a notch duplicate a total of at most N-1 other notches, which leads to an $O(N^2)$ upper bound on the number of convex parts.

More precisely, let us define for each notch v_i a plane T_i which resolves its reflex angle. We proceed as before, now ensuring that all subnotches of the notch v_i will always be removed with a cut passing through T_i. This simple requirement improves the worst-case performance dramatically.

Theorem 1: The revised naive decomposition applied to P yields at most $N^2/2 + N/2 + 1$ convex parts.

Proof: We can assume that all the subnotches of a notch are removed consecutively. If we keep all the parts of the decomposition together, all the cuts corresponding to the subnotches of a notch will be coplanar and will intersect every other notch in at most one point. Consequently, at the i-th step, each remaining notch will have been broken up into at most i+1 subnotches, and step i+1 will introduce at most i+1 polyhedra into the decomposition. This completes the proof. □

We must now formalize the intuition given above and describe an effective method for carrying out the naive decomposition. We will show that it can be done in $O(nN^3)$ operations, using $O(nN^2)$ storage. The first issue to investigate is the mode of representation used for describing a polyhedron. Since many practical problems involve dealing with faces rather than edges or vertices, we may assume that the edges enclosing a given face are readily available. More precisely, we require the data strucure chosen to provide three types of lists:

1. Edge-to-face lists: indicate the two faces adjacent to each edge.

2. Face-to-edge lists: give the sequence of edges enclosing each face in clockwise order.

3. Adjacency lists: provide a set of the vertices adjacent to each vertex in clockwise order (as seen from, say, the outside of the polyhedron).

Note that the faces of a non-convex polyhedron may be polygons with holes. In that case, each face-to-edge list should provide clockwise descriptions of the outer as well as of the inner boundaries. We call a graph representation of a polyhedron any representation providing the above lists. We observe that such representations are redundant, since it is possible to go from one set of lists to another. For the sake of simplicity, however, we prefer to make all these lists available to avoid dealing with conversion problems which lie outside of our main concern.

It is easy to see that representing all these lists take O(p) storage (recall that p denotes the number of edges). Suppose that P is of genus 0. Then the boundary of P has a structure of planar graph (possibly disconnected), and we have [Harary,71]

$$p \leq 3n-6$$

This shows in particular that representing P takes also O(n) storage.

Before attacking our main problem, let us consider the problem of dividing up a polyhedron P of genus 0 (i.e., without holes) by a cut S adjacent to an edge e of P. We wish to compute a graph representation of the two polyhedra P_1 and P_2 that S breaks P into. In order to compute S from the knowledge only of e and the plane supporting S, we first prove an important result which is actually rather interesting in general.

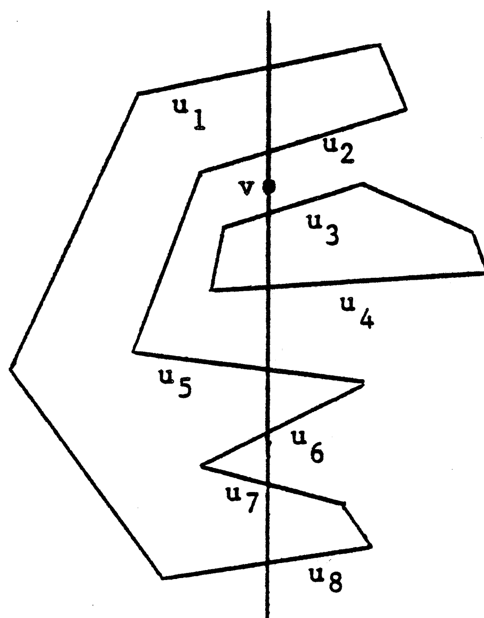
Lemma 14: Let W be a set of simple polygons in the plane such that all the boundaries are pairwise disjoint. We call a maximum of W any polygon of W which is not contained in any other. It is possible to determine all the maxima of W in O(nlogn) time, if n is the total number of vertices in W.

Proof: We first note that the requirement for W specifies that for any pair of polygons, either one contains the other or the two do not intersect. Thus W always has at least one maximum.

The method is directly inspired from Shamos and Hoey's algorithm for intersecting pairs of segments [Shamos and Hoey,76]. The crucial observation is that the intersection of a vertical line L with the maxima of W forms a set (possibly empty) of disjoint

segments. The endpoints of each segment lie on some edges of W, and the vertical line L induces a total ordering R on the set E of these edges. E consists exactly of all the edges of maxima which intersect L - See Figure 3.4-a. We say that two edges of E, consecutive with respect to R, are "linked" if the vertical segment joining them lies in a maximum of W. Note that consecutive pairs of edges in R are alternately linked and not linked. For any point v of L, we define h(v) [resp. l(v)] as the first edge in E above v [resp. below v]. If no such edge exists, h(v) or l(v) is 0. See Figure 3.4-a. The notion of "above" and "below" is of course defined with respect to the vertical line L. Similarly, the order of two edges of W is defined with respect to a common intersecting vertical line. Actually, this order is the same for any vertical line since the edges of W can intersect only at their endpoints. If v is the leftmost vertex of a polygon P of W, P is a maximum if and only if h(v) and l(v) are not linked. This condition is clearly necessary since, if h(v) and l(v) are linked, they belong to the same polygon which cannot be P since v is its leftmost vertex. To see that it is sufficient, we assume that P is not a maximum, then there is a unique maximum Q in W which contains P, and Q must intersect the vertical line passing through v, therefore the intersection is a segment containing v and the pair h(v),l(v) must be linked.

The algorithm proceeds as follows: We sweep a vertical line from left to right, passing through each vertex v in W. The vertices are maintained in sorted order (by x-values) in a set Q. We first check if v is the leftmost vertex of a polygon P of W. If it is, we can decide immediately if P is a maximum by finding whether h(v) and l(v) are linked. If they are, P is not a maximum and all its vertices are deleted from Q. Otherwise, P is a maximum. Actually, since non-maxima are removed as soon as their leftmost vertex is encountered, the polygon containing v is a maximum in all the other cases (i.e., when v is not a leftmost vertex). Then we can simply update the ordering R with the functions "insert" and "delete" as well as the linked pairs with the functions "link" and "unlink". This is fairly straightforward and the algorithm we next present should be self-explanatory.

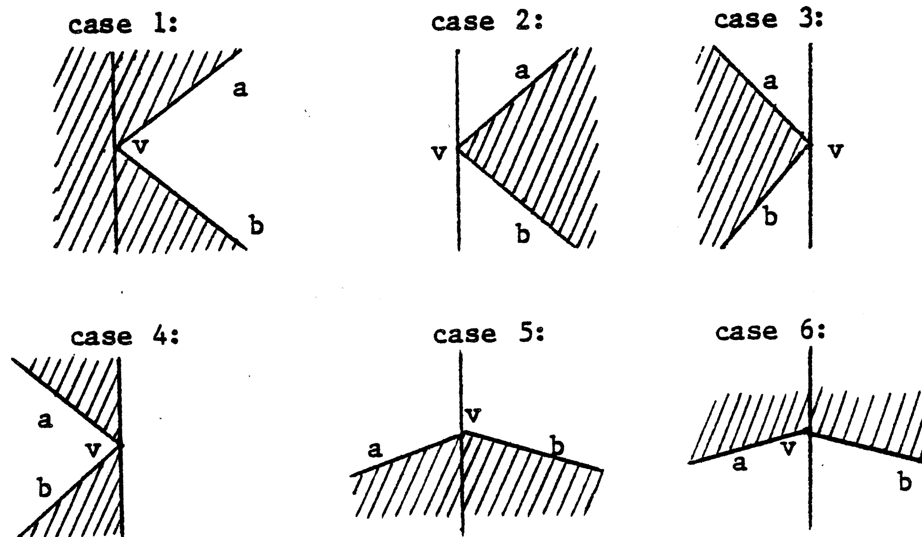


Linked edges: (u_1,u_2),
(u_3,u_4),
(u_5,u_6),
(u_7,u_8).

$h(v) = u_2$

$l(v) = u_3$

a) The ordering R.



b) The algorithm for computing maxima.

<u>Figure 3.4</u>

```
MAXIMUM(W)

  Q = Set of vertices in W stored in order
      by x-values.

  R = Empty set.

  for all v in Q (in ascending x-order)
      begin

      Let P be the polygon to which v belongs.

      if v is the leftmost vertex of P

         and h(v),l(v) are linked

         then  "P is not a maximum"

               delete all vertices of P from Q

         else  "P is a maximum"

               UPDATE(R,v)

      end
```

UPDATE(R,v)

```
Let a,b be the two edges adjacent to v.
Switch to the case corresponding to Fig. 3.4-b.

case 1:
            insert(a),insert(b)
            unlink (h(v),l(v))
            link (h(v),a)
            link (b,l(v))
            break

case 2:
            insert(a),insert(b)
            link(a,b)
            break

case 3:
            delete(a),delete(b)
            unlink(a,b)
            break

case 4:
            delete(a),delete(b)
            unlink (h(v),a)
            unlink (b,l(v))
            link (h(v),l(v))
            break

case 5:
            delete(a),insert(b)
            unlink (a,l(v))
            link (b,l(v))
            break

case 6:
            delete(a),insert(b)
            unlink (h(v),a)
            link (h(v),b)
            break
```

Note that when the algorithm terminates, only the vertices of maxima will remain in Q, thus the maxima can be obtained from Q in O(n) time. To implement the algorithm efficiently, we can store Q as a doubly-linked list with random-access to the nodes, thus allowing constant time deletions. R can be maintained as a balanced tree, so that the functions h,l,insert, and delete perform in logarithmic time. Link(u,v) will simply set two pointers, one from u to v, and the other from v to u, while unlink(u,v) will remove these

pointers. With this implementation, the algorithm requires O(nlogn) time. Note that all the preprocessing needed involves sorting the vertices by x-values and computing the leftmost vertices, all of which also takes O(nlogn) time. □

We can now turn back to the problem of dividing up a polyhedron P. As we noticed earlier, the intersection of P with the plane T supporting the cut S is in general a set of polygons. These polygons may have holes which may themselves contain other polygons of the same kind. In any event, S is defined as the unique polygon containing e as one of its boundary edges - See Figure 3.5-a. We first compute S from which we derive P_1 and P_2.

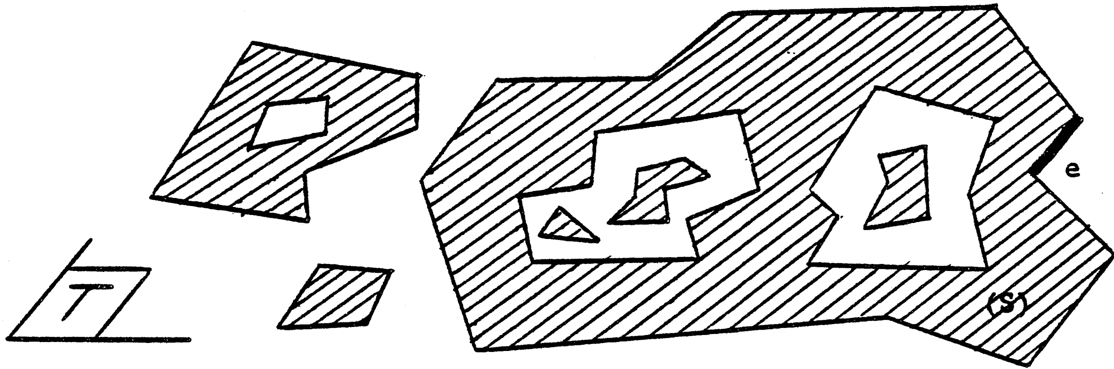
I) - "<u>Computing the intersection of P and T</u>"

Consider each face F of P in turn and report all the edges of F which intersect the plane T, yet do not lie on T. This includes all the edges of the inner and outer boundaries. Let $a_1,...,a_k$ denote the intersections of T with these edges, as they appear in sorted order on the line supporting the intersection of F and T. We call u_i the edge of F intersecting T at a_i. Observing that the intersection of T and F is made up of the segments $a_1a_2,...,a_{k-1}a_k$ (Fig. 3.5-b), we set two pointers for each pair (u_{2i-1},u_{2i}); one from u_{2i-1} to u_{2i} and the other from u_{2i} to u_{2i-1}. Repeating this process for all faces of P will eventually provide doubly-linked lists for all the boundaries of the polygons of the intersection of P and T. Let U denote this set of boundaries. Since each edge is considered at most twice, all these operations take O(p) time, except for the sorts, each of which requiring $O(p_i' \log p_i')$ time, where p_i' is the number of edges intersecting T involved in the face considered. Since each edge appears on two faces, the sum of all the p_i' is less than or equal to 2p', which leads to an O(p'logp') running time (similarly, p' is the number of edges of P intersecting T). Note that the conversion of the doubly-linked lists of u_i into lists of a_i is straightforward in general. Some special cases may yet be encountered, when a_i is the endpoint of u_i and several edges are adjacent to a_i. It is easy to see, however, that those cases can be handled separately without altering the total running time of the algorithm, which is O(p+p'logp').

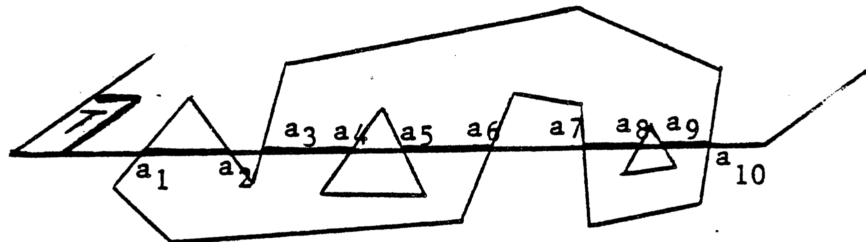
II) - "<u>Computing S</u>"

First we can obtain the outer boundary of S, denoted S^*, by identifying the boundary in U which contains the edge e. To find the inner boundaries is somewhat more involved. We first form the subset W of U consisting of all the boundaries which lie inside S^*. To do so, we can use a variant of the algorithm MAXIMUM used in the proof of Lemma 14.

Q is still the set of all vertices in U, ordered by x-values. The ordering R, however, will now involve the edges of S^* only. As before, the main loop sweeps a vertical line left-to-right passing through each vertex in Q. If v belongs to S^*, we simply maintain the ordering R with the function UPDATE



a) A cut S.



b) The edges of S.

Figure 3.5

defined earlier. Otherwise, we observe that the boundary in U which contains v lies inside S^* if and only if h(v) and l(v) are distinct from 0 and are linked. Thus, we know whether a boundary belongs to W or not as soon as we examine its leftmost vertex. To make the algorithm more efficient, we can thus delete all the vertices of the boundary from Q, after examining its first vertex. Like its look-alike, MAXIMUM, this algorithm requires O(klogk) time, where k is the total number of vertices in Q. Since each of these vertices corresponds to a distinct edge of P, the running time is O(p'logp').

```
Q = Set of vertices in U sorted by x-values.

R = W = Empty set.

for  all v in Q (in ascending x-order)
   begin

      if  v belongs to S*

          then  UPDATE (R,v)

          else  Let B be the boundary in U
                containing v.

                delete all vertices of B from Q.

                if h(v) and l(v) are not 0
                   and are linked

                   then  "v lies inside S*"

                           W = W U {B}

   end
```

We are now ready to apply the result of Lemma 14 to the set W. This will give us exactly all the inner boundaries of S, with a total running time of O(p'logp').

III) - "Computing P_1 and P_2"

The last step is to compute a graph representation of P_1 and P_2. This is a trivial graph transformation, and we only sketch out the procedure. Let Adj(w) be the adjacency list of the vertex w in the graph representation of P. Also, call E the set of edges of P passing through the vertices of S. We can assume E to be readily available, since the edges in E must be determined in order to compute S. Let w be an endpoint of some edge in E. We define P_1 as the polyhedron cut by S which contains w, and we briefly show how to compute P_1 in O(p) time.

1) Adjacency lists of P_1

For each edge ab of E which does not lie on T, let v be the unique vertex of S lying on ab. We can always assume that a lies on the same side of T as w, that is, is a vertex of P_1, whereas b is a vertex of P_2. If v is distinct from a, we replace b by v in the list Adj(a) and delete the list Adj(b). If v=a, we simply delete b from Adj(a) as well as the list Adj(b). Repeating these operations for all the edges of E which do not lie on T has the effect of disconnecting P_1 from P_2. Then, a depth-first search in the resulting graph of P, starting at w, will provide all the vertices of P_1. All the adjacency lists of the vertices common to P and P_1 have already been updated. Finally, since we have a doubly-linked-list description of the boundaries of S, we can set up the adjacency lists of the new vertices, that is, the vertices of P_1 lying on S. All these operations clearly require O(p) time.

2) Face-to-edge lists of P_1

Since the previous lists provide the set of vertices of P_1, we first remove all the faces of P made up entirely of vertices not in P_1. Then, since all the faces of P intersecting S have been previously determined, it is easy to compute a description of the parts of those faces which lie in P_1. Let F be such a face, with $a_1,...,a_k$ being the vertices of S lying on F. Recall that $a_1,...,a_k$ have been computed in sorted order (Step I). We may assume that the boundaries of F are represented by doubly-linked lists with the nodes representing the vertices. Letting u_i be the edge of F passing through a_i and b_i be the endpoint which lies on the same side of T as w, we first delete from the lists all the vertices lying strictly on the other side of T, then we enter the vertices a_i into the lists by linking both ways b_i and a_i as well as a_{2i-1} and a_{2i} - See Figure 3.6-a. Note that we can always assume that u_i does not lie on T, which ensures that b_i is always well-defined. The result of these operations may produce several disconnected lists since F may be broken up into several faces of P_1. Finally, if F has some edges lying on T, the algorithm may produce lists consisting of two vertices, and these degenerate cases should be removed in a postprocessing stage - See Figure 3.6-b. Finally, the face-to-edge lists of S (which have already been computed) must join the set of face-to-edge lists of P_1. Once again, it is clear that all these operations will take O(p) time.

3) Edge-to-face lists of P_1

These lists can be obtained in O(p) time by scanning through the face-to-edge lists once and recording the faces next to each of their boundary edges.

The computation of P_1 and P_2 is complete, and we can conclude:

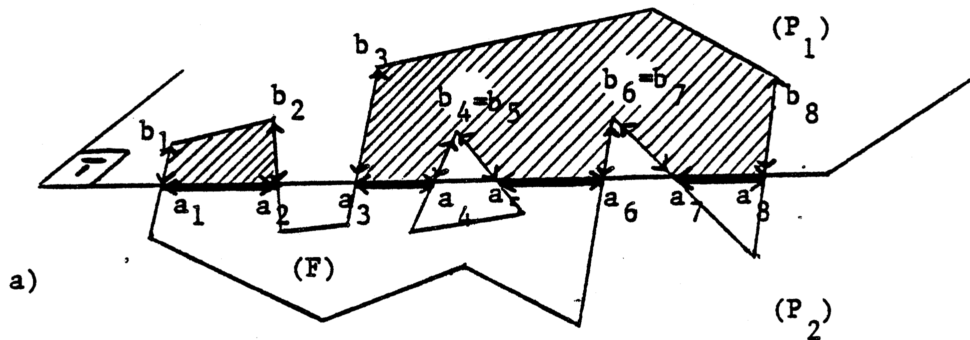
Lemma 15: A polyhedron P of genus 0 can be partitioned with a cut in time $O(p+p'\log p')$, using O(p) storage, with p' being the number of edges of P intersecting the plane supporting the cut.

We have seen that in the course of its action, the naive decomposition may produce polyhedra containing holes. For that reason, we wish to generalize the previous result to polyhedra of arbitrary genus. Now instead of breaking P into two pieces, a cut may simply decrease its genus by one or have some of the effects described at the beginning of Section 3.2 (e.g, removing a handle and creating another). To handle those cases, we may first "cut" each edge of P which intersects S by updating the adjacency lists accordingly. We then test the connectivity of the graph by doing a depth-first search with the adjacency lists. If it is no longer connected, the cut breaks P into two separate pieces P_1 and P_2 which can be computed as indicated above. Otherwise, we update the lists of the representation in a similar way; the only major difference being the introduction of two faces corresponding to the cut. We omit the details of these operations which are very elementary.

In our analysis, we were careful to use the number of edges p and not the number of vertices as the measure of the input size. Indeed, Euler's formula has to be corrected for higher genuses and the relation $p \leq 3n-6$ is no longer valid. However, it is easy to verify that the number of edges always gives the size of the description of P (up to a constant factor). We are now ready to present our first version of the naive decomposition. It is merely a repeated application of the procedure described above.

Theorem 2: The naive decomposition of a polyhedron P of genus 0 can be done in $O(nN^2(N+\log n))$ time, using $O(nN^2)$ storage.

Proof: The algorithm proceeds by removing each notch in turn. In an O(p) preprocessing stage, we can assign to each notch a plane resolving its reflex angle. Then all the subnotches of a same notch will be removed consecutively with cuts lying on this plane. This guarantees $O(N^2)$ convex parts in the end as has been shown in Theorem 1. Each cut can be implemented with the procedure of Lemma 15 and the generalization for higher genera we just mentioned. Consider the partial decomposition before the notch g is removed. Let $P_1,\ldots,P_k$ be the (non-convex) polyhedra in the current decomposition which contain a segment of g as a subnotch (we know that $k \leq N$). Let p_i be the number of edges in P_i and p_i' the number of edges intersecting the plane supporting the cut used to remove g. From Lemma 15, we know that we can remove the subnotch of g in P_i in time $O(p_i+p_i'\log p_i')$. Next we will evaluate the maximum number C of edges present at any time in the decomposition. We distinguish two kinds of edges: First the edges which are



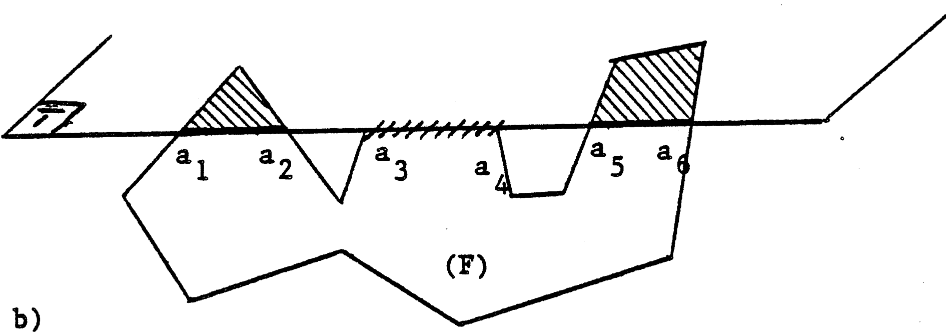


Figure 3.6: Computing the faces of P_1.

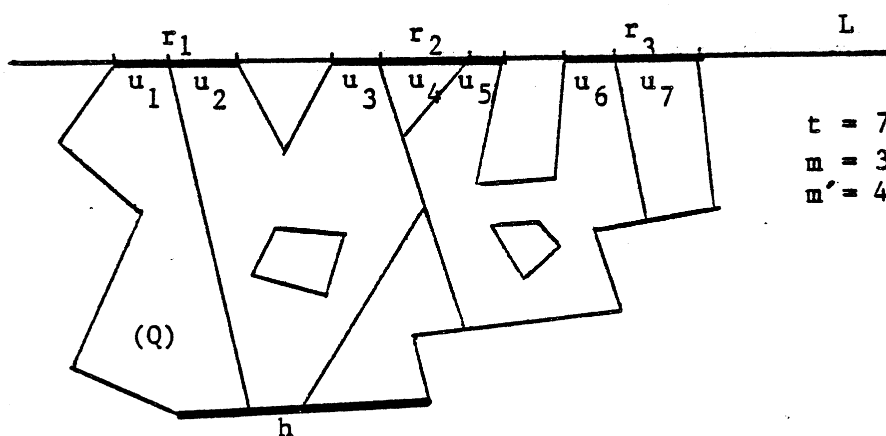


Figure 3.7: Counting the number of edges in the decomposition.

pieces of edges of P. Since each edge of P can be divided into at most N+1 segments, the number C_1 of such edges cannot be greater than p(N+1).

The other edges are intersections of cuts with faces (or parts of faces) of P or intersections between cuts. Since all cuts lie on N possible planes and all faces of P lie on q possible planes, the C_2 edges we are now considering lie on at most qN possible lines. Next we show that each of these lines supports at most 3N edges (we do not believe that this upper bound is tight). Let L be such a line and $u_1,...,u_t$ be the edges of the decomposition which lie on L. The edges $u_1,...,u_t$ form m disconnected segments $r_1,...,r_m$ on L, each segment consisting of contiguous edges u_i ($1 \leq m \leq t$) - See Figure 3.7. Let m' be the number of endpoints common to two consecutive u_i; we have

$$(1)\ m+m' = t$$

L is the line passing through the intersection of a cut S with a face of P or the intersection of two cuts S and S'. In either case, let h be the notch passing through the cut S. The union of all the cuts used to remove h forms a polygon Q possibly with holes. Moreover all the segments r_i are edges of Q and each notch of Q corresponds to a distinct notch of P. At this point, we must anticipate a little and use a result which we will prove in the next section (Lemma 18). This result states that the line L cannot intersect Q in more than 2N segments. Therefore we have

$$(2)\ m \leq 2N$$

Since the "interior" endpoints are all intersections of cuts with L, we also have

$$(3)\ m' \leq N$$

Combining (1),(2),(3) shows that $t \leq 3N$, which proves our claim and implies that

$$C_2 \leq 3qN^2$$

Since each edge of P is adjacent to at most 2 faces of P while a face has at least 3 enclosing edges, we have

$$3q \leq 2p$$

showing that

$$C_2 = O(nN^2)$$

since p=O(n) (P is of genus 0). Our counting argument considered each u_i as the intersection of a cut or a face with a cut. Therefore each edge u_i will be counted exactly twice in $p_1+...+p_k$, hence

$$p_1+...+p_k \leq C_1+2C_2$$

Finally, since $C_1 \leq p(N+1)$ and $p=O(n)$, we have

$$p_1+...+p_k = O(nN^2)$$

Also, since at most 2 edges intersecting a given plane in a single point can be collinear, the maximum number of edges which can intersect a given plane is bounded by the maximum number of lines L, therefore

$$p_1'+...+p_k' = O(nN)$$

It follows that all the subnotches of g can be removed in time $O(nN(N+\log n))$, using $O(nN^2)$ storage. Since N notches must be removed, the proof is now complete. □

3.2.3 A More Efficient Implementation of the Naive Decomposition

As we mentioned earlier, N is greatly dominated by n in most practical cases. This suggests asking the following question: In the running time of the naive decomposition, can the factor nlogn be reduced to n without increasing the amount of storage? The answer is positive, and our next task will be to describe an $O(nN^3)$ time decomposition algorithm using $O(nN^2)$ space.

The improvement relies more on a careful analysis of the running time than on drastic modifications. Once again, we will make convexity the key to efficiency. Basically, we will use the concept of convex chains which we introduced in the previous chapter, where we showed how closely notches and convex chains were related in two dimensions. For example, we will replace linear scans of convex chains by Fibonacci searches. It is no surprise that added complexities arise with polyhedra and we must start with an investigation of how notches and convex chains relate in three dimensions.

We have observed that the intersection of P with a plane T consists in general of several polygons $R_1,...,R_k$ which may have holes and may be embedded within one another. It is intuitively apparent that the larger k is, the more notches P need have. To formalize this statement, we first prove a stronger result valid for a particular case. We say that T is tangent to P if all of P lies on the same side of T, and the intersection of P and T is not empty. In this case, the polygons R_i are faces of P. Let N^* be the number of notches in P which do not intersect T. In the following, P may be of any genus, that is, have any number of holes.

Lemma 16: If T is tangent to P and $k>1$, we have $k \leq 2N^*$.

Proof: Wlog, let T be the horizontal plane, with all of P lying below T. Although all the vertices of P which lie on T are coplanar, we can always assume that among the other vertices, no two lie on a line parallel to T. If this is not the case, we rotate T around some axis by a very small angle so as to satisfy this requirement. The angle can always be chosen small enough to ensure that the intersection of P and T still consists of k polygons $R_1,...,R_k$.

Let $v_1,...,v_l$ be a list of the vertices of P lying strictly below T, sorted vertically in descending order. If we translate the plane T downwards along a vertical axis in a continuous motion, we observe that the polygons R_i undergo continuous transformations. New polygons may appear in the process, some may vanish from T, while others may merge. Eventually all of them will disappear from T. The crucial observation is that since P is connected, no R_i will disappear before merging at least once. Therefore there will be at least k/2 merges in the process (actually, it would be easy to show that there will be at least k-1 merges). Note that the merges can occur only when T reaches a vertex v_i. Let T_i be the corresponding plane (i.e., the horizontal plane passing through v_i). Since all the v_i have distinct heights, at most one merge can occur at T_i. Suppose that R and Q are two polygons merging on T_i. The vertex common to R and Q is the unique vertex of P lying on T_i, that is, v_i - See Figure 3.8. Let P^* be the part of P lying below T_i and containing v_i. By construction, P^* has at least one notch e_i emanating from v_i. Since v_i is a vertex of P, e_i is also a notch of P. Also, detecting the notch e_i when we hit its highest endpoint ensures that all the e_i thus defined will be distinct. Therefore we have exhibited at least k/2 notches in P. All these notches lie below v_1, then do not intersect T, which completes the proof. □

We are now ready to generalize this result to the cases where T is not necessarily tangent to P.

Lemma 17: Let $R_1,...,R_k$ be the polygons of the intersection of P and T (P is of any genus). We have $k \leq 2N$ if $k>1$.

Proof: Let us cut P along each polygon R_i. This partitions P into at most k+1 polyhedra, all tangent to T (we may have strictly fewer than k+1 if P has holes). Also,

since P is connected, each polygon R_i is the face of at least one polyhedron which has at least another face tangent to T (assuming k>1). It follows that among these polyhedra, we can find j of them, say $P_1,...,P_j$, such that each has at least two faces tangent to T and each R_i is the face of at least one of them. Let N_i be the number of the notches in P_i which do not intersect T and k_i the number of faces adjacent to T. Since P_i has at least 2 faces adjacent to T, we can apply Lemma 16, $k_i \leq 2N_i$. Since $k_1+...+k_j \geq k$ and all the notches of P involved in the j quantities $N_1,...,N_j$ are distinct, we have $k \leq 2N$. □

We have an immediate corollary of the lemma in two dimensions.

Lemma 18: Let N be the number of notches of a concave polygon P of any genus. No line can intersect P in more than 2N segments.

Proof: Consider the cylinder with base P and generators perpendicular to P, and apply Lemma 17. □

Now that we have related the number of polygons obtained by intersecting P and T to the number of notches in P, we can modify the algorithm of Lemma 14 to find maxima faster and use these results to analyze the new method.

Lemma 19: Let W be a set of k simple polygons of genus 0 such that all the boundaries are pairwise disjoint. It is possible to determine all the maxima of W in time $O(n+(k+N)\log(k+N)\log n)$, where n is the total number of vertices in W and N is the total number of notches.

Proof: We preprocess each polygon W as follows: Partition its boundary into convex chains $C_1,...,C_m$, starting at the vertex with minimum x-coordinate (recall the definition of a convex chain and how to do the partitioning in Section 2.2.2). We observe that each convex chain C_i can be partitioned into at most 3 subchains with the property that the x-coordinates of a subchain appear in increasing or decreasing order. We compute all such subchains for all polygons in W and associate to each of them a list of its vertices in ascending x-value. We can compute all these lists $L_1,...,L_p$ in O(n) time and store them in O(n) space, allowing random-access to each vertex - See Figure 3.9. Let W_i be any polygon of W and N_i denote its number of notches, m_i its number of convex chains, and p_i its number of subchains. We know that $p_i \leq 3m_i$, and by Theorem 3, we have $m_i \leq 2(1+N_i)$. It follows that $p_i \leq 6(1+N_i)$ and summing up for all polygons in W yields:

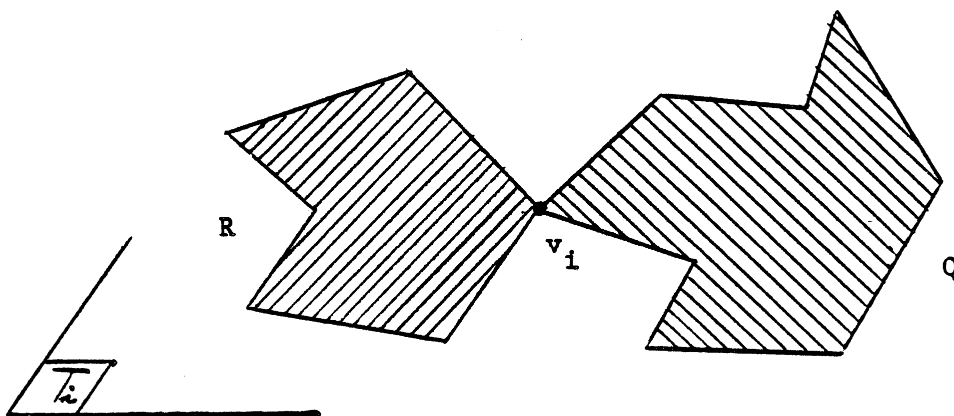
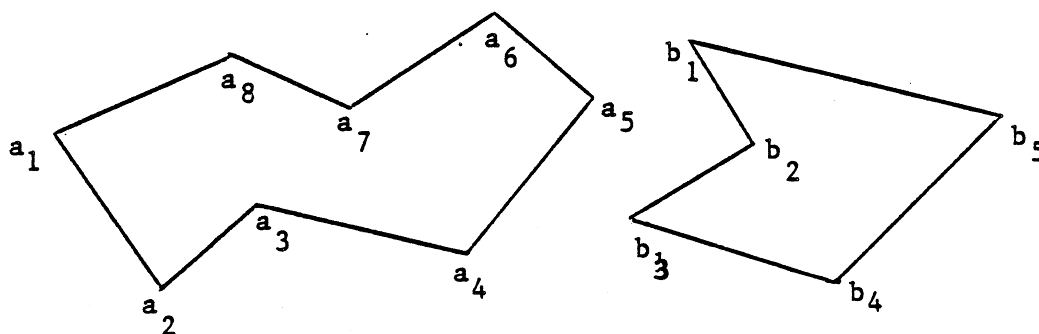


Figure 3.8: The merging of two polygons R and Q.



$L_1 = a_1a_2a_3$ $L_5 = b_1b_2$

$L_2 = a_1a_8a_7$ $L_6 = b_1b_5$

$L_3 = a_3a_4a_5$ $L_7 = b_3b_4b_5$

$L_4 = a_7a_6a_5$ $L_8 = b_3b_2$

Figure 3.9: The subchains in W.

(1) $p \leq 6(k+N)$

The algorithm is strictly similar to the one described in Lemma 14. The only difference is that edges are now replaced by subchains. Since a subchain can intersect a vertical line in at most one point, and two subchains can intersect only at their endpoints, a vertical line L intersecting polygons in W induces the same ordering R on the set of subchains intersecting L as it did on the set of edges intersecting L. The procedure MAXIMUM can be easily adapted to maintain that ordering. Q is now the set of endpoints of $L_1,...,L_p$ sorted by x-values. The major difference arises when a point of L has to be searched in the balanced tree representing R (functions h and l). In order to find if it lies above or below a certain subchain, we must compute the intersection of the subchain with L, which can be done in O(log n) time with a binary search. This will add a factor logn to the running time of MAXIMUM. The procedure UPDATE(R,v) is strictly similar. We can still distinguish among the various cases in constant time, by comparing the position of the edges adjacent to v. The balanced tree has at most p nodes, therefore the algorithm will process each vertex of Q in time O(lognlogp). Since all the preprocessing takes linear time except for sorting the endpoints of the subchains, which takes O(plogp), the total running time of the algorithm will be O(n+plogp+plogplogn), that is, O(n+(k+N)log(k+N)logn) because of relation (1). □

Actually, we need a somewhat stronger result than Lemma 19. Since the boundary C of a polygon P delimits two regions of the plane (Jordan's theorem [Munkres,75]), say P and Q, we observe that any vertex of C convex for P (i.e., not a notch) is a notch for Q and vice-versa. The vertices of C which are notches with respect to Q are called conotches. Thus, any vertex of P is either a notch or a conotch. Likewise, as we defined convex chains, we can now introduce the cochains or convex chains with respect to Q - See Figure 3.10-a. It is clear that cochains can be computed in exactly the same way as convex chains (see Section 2.2.2). We have a result similar to Theorem 3.

Theorem 3: A polygon with m cochains and q conotches satisfies:

$$m \leq 2(1+q)$$

Proof: Let v be a vertex of the polygon which lies on its convex hull. Since v must be a conotch, we can enclose the boundary C of the polygon with a rectangle passing through v - See Figure 3.10-b. This defines a polygon R whose boundary consists of C and of 5

additional edges. Moreover all the conotches of C are notches of R. Therefore we can apply Theorem 3 and find that the maximum number of convex chains in R is bounded by $2(1+q)$, and so is the maximum number of cochains in C. □

This simple remark permits us to define the convex chains in the proof of Lemma 19 either with respect to the inside or the outside of the polygons. We generalize this result as follows:

Lemma 20: Let W be a set of k simple polygons in the plane such that all the boundaries are pairwise disjoint. For each polygon W_i, let N_i be either its number of notches or its number of conotches and N be the sum of all N_i. It is possible to determine all the maxima of W in time $O(n+(k+N)\log(k+N)\log n)$ with n being the total number of vertices in W.

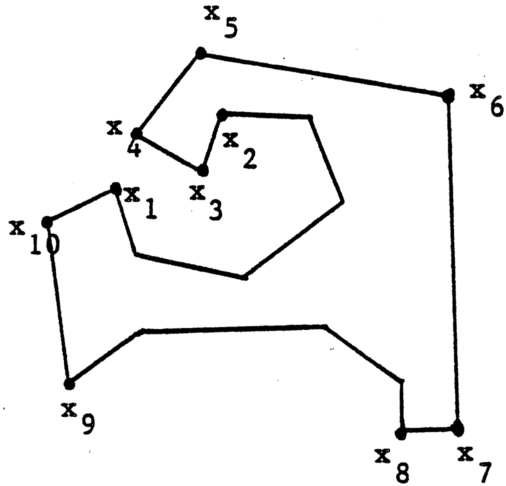
Proof: For each polygon of W, we choose to partition its boundary either into convex chains or cochains. The result of Lemma 19 is then strictly applicable since Theorem 3 shows that the relationship between conotches and cochains is the same as between notches and convex chains. □

We are now in a position to make a cut through P more efficiently (in most cases) than in Lemma 15.

Lemma 21: A polyhedron P of arbitrary genus can be split with a cut in time $O(p+N\log N\log p')$ using $O(p)$ storage, with p' being the number of edges in P intersecting with the plane supporting the cut.

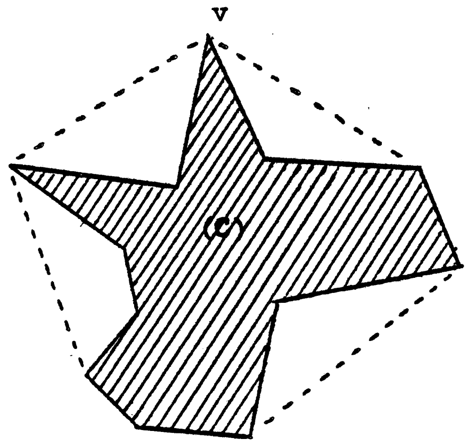
Proof: Note that we used the verb "split" in the statement of the lemma to refer to the fact that P may not be broken up into pieces. We review the various phases of the algorithm presented in Lemma 15.

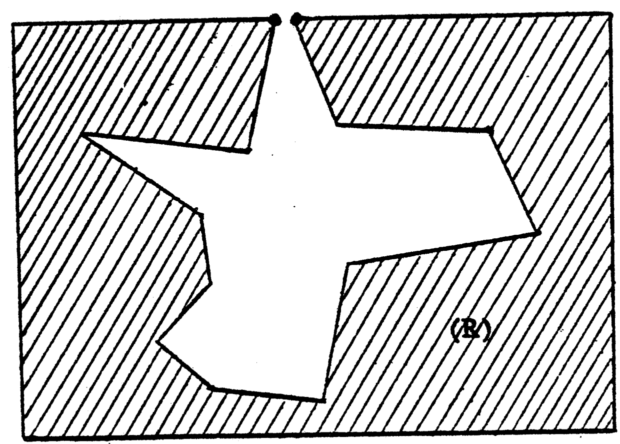
STEP I is unchanged. A more careful analysis of the running time will yet bring some improvement. Let $F_1,...,F_k$ be the faces of P intersecting T, with p_i (resp.N_i) the number of edges (resp. notches) in the polygon F_i. From Lemma 18, we know that F_i intersects P in at most $MAX(1,2N_i)$ segments. It follows that sorting the endpoints of these segments as required by STEP I will take $O(N_i\log N_i)$ time. Since each notch of the polygon F_i is adjacent to a notch of P and a notch of P is adjacent to at most 2 notches in $F_1,...,F_k$, we



10 cochains $x_1x_2, \ldots, x_{10}x_1$

a) The cochains of a polygon.





b) Counting cochains.

Figure 3.10

have $N_1+...+N_k \leq 2N$. Therefore, the execution of STEP I will require $O(p+k+N\log N)$, that is, $O(p+N\log N)$ time.

STEP II. Let $C_1,...,C_t$ denote the boundaries involved in the intersection of T and P, and $P_1,...,P_k$ the polygons of this intersection ($k \leq t$). We compute all the boundaries of the cut as we previously did, now applying the revised version of the procedure MAXIMUM and of its variant (Lemma 19). However, for all the inner boundaries of the intersection, we compute cochains and not convex chains (Lemma 20). Note that we can detect in constant time if a boundary is inner or outer after an $O(p)$ time preprocessing. The result of Lemma 20 shows that all these operations can be carried out in $O(p'+(t+N)\log(t+N)\log p')$ time, since the notches of the P_i are adjacent to distinct notches of P. More precisely, each notch of an outer boundary or conotch of an inner boundary lies on a distinct notch of P. From Lemma 17, we know that if $k>1$, we have $k \leq 2N$. Therefore the intersection of T and P consists of at most $1+2N$ outer boundaries. On the other hand, each inner boundary has at least 3 conotches lying on 3 distinct notches of P. This shows that $t \leq 1+3N$. We can finally conclude that the running time of STEP II is $O(p'+N\log N\log p')$.

STEP III is unchanged and requires $O(p)$ time, which completes the proof. □

Finally we are ready to achieve our main goal.

Theorem 4: The naive decomposition of P can be done in $O(nN^3)$ time and $O(nN^2)$ space.

Proof: We simply have to review the proof of Theorem 2 with a new analysis of the running time. The method for carrying out the decomposition is similar except for the removal of each subnotch, which is now done with the procedure of Lemma 21. Consequently, we can now remove the subnotch of g in P_i in time $O(p_i+N\log N\log p_i')$. Since $p_1'+...+p_k'=O(nN)$ and $k \leq N \leq n$,

$$\log p_1'+...+\log p_k' = O(k\log(nN)) = O(N\log n)$$

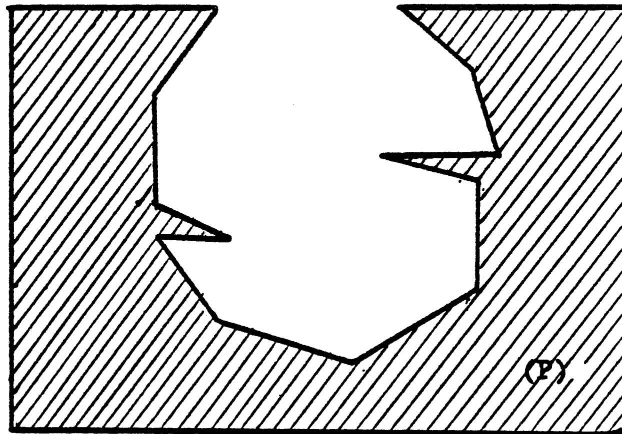
On the other hand, we have $p_1+...+p_k=O(nN^2)$, which shows that the notch g can be entirely removed in time $O(nN^2+N^2\log N\log n)$, and completes the proof. □

This more complex implementation of the naive decomposition has the substantial advantage of being linear in the number vertices. This can be of great interest for problems of big size, where the number of notches is very small compared to the total number of vertices. With this presentation, we primarily intended to illustrate how the attribute of convexity can be exploited to increase efficiency in three dimensions.

In conclusion, we have presented two versions of the naive decomposition, both of which are worst-case optimal (within a constant multiple) in the number of convex parts produced (this will be shown in the next section). Although the algorithms are conceptually fairly simple, complex data structure manipulations have to be performed, which makes their description somewhat involved. We have actually done a very rough analysis of the execution times, and we believe that, in practice, the algorithm may perform much faster than claimed.

REMARK:

We believe that the result of Theorem 3 can also be used for decomposing polygons as sums and differences. This problem was briefly mentioned at the end of Chapter 2, where we simply pointed out its difficulty and the need for simple heuristics. The goal is basically to express a concave polygon P as a difference A-B, where A and B are concave polygons (or sets of polygons) which must be themselves decomposed into a minimum number of convex parts. This type of decomposition can be very economical especially if P contains lots of notches. Intuitively we can see that most of the notches of B are likely to be conotches of P - See Figure 3.11. Therefore if P has few but long cochains, we may simply apply the naive decomposition of Chapter 2 to the polygons of B. Theorem 3 showing that the analysis of Section 2.2 is still valid, this method will be very efficient when P contains few conotches.



P = A−B

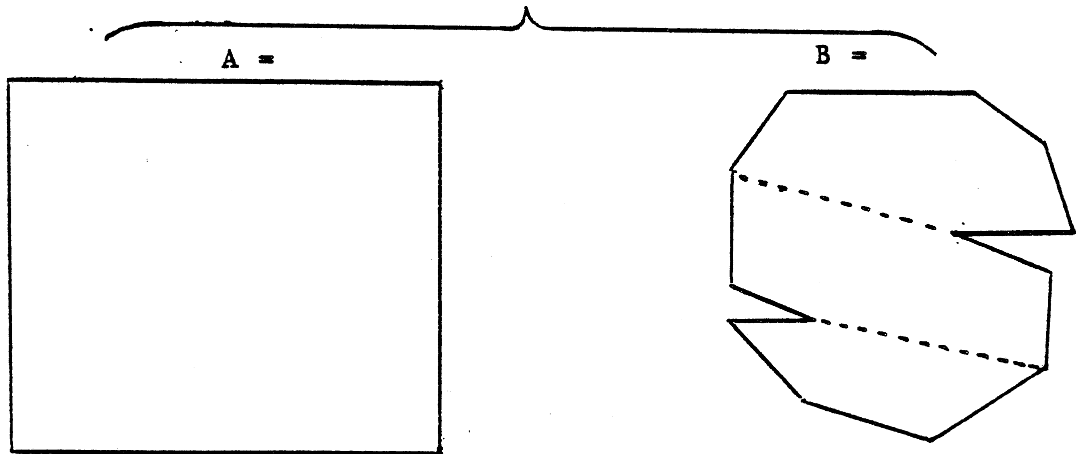


Figure 3.11: Decomposition as sums and differences.

3.3 A quadratic Lower Bound on the Number of Convex Parts

3.3.1 Introduction

The naive decomposition guarantees $O(N^2)$ convex parts and saves us from an exponential blow-up. We may yet wonder whether O(N) parts is not always achievable as it is the case in two dimensions. We next tackle this problem and prove that this $O(N^2)$ upper bound is indeed tight. This result shows that the naive decomposition described in the previous section is worst-case optimal in the number of convex parts (up to a constant multiple).

To achieve our goal, we must exhibit a class of polyhedra which cannot be decomposed into fewer than cN^2 parts. The technique used to derive this lower bound is based on volume considerations. We define a portion (Σ) of the polyhedron P and, observing that a decomposition of P also realizes a partition of (Σ), we study the contribution of each convex part to this partitioning. The crux is to show that a convex part can only have a small piece lying in (Σ), and therefore lots of convex parts are needed to fill up (Σ). To realize this condition, we must carefully design (Σ), giving it a warped shape so that its intersection with any convex object can never occupy too much space. Since (Σ) must be defined by means of straight lines, we give it a shape of hyperbolic paraboloid. Recall that this surface can be generated by two sets of orthogonal lines [Protter and Morrey,70].

3.3.2 Description of the Polyhedron P

P is essentially a rectangular parallelepiped with a series of $N+1$ notches cut through the lower face and $N+1$ similar notches cut through the upper face - See Figure 3.12-a,b. The two faces adjacent to any notch form a very small angle and, for our purposes, can be regarded as a single vertical quadrilateral. Thus, we have $N+1$ such quadrilaterals emanating from the lower face, all of which are vertical, parallel to the plane Oxz, and equidistant. The upper edges of these quadrilaterals are called the bottom notches of the polyhedron P, and are designated BOT0,...,BOTN in ascending y-value. To achieve the desired warping, all the bottom notches lie on the hyperbolic paraboloid $z=xy$. The $N+1$ quadrilaterals emanating from the upper face of P are parallel to the plane Oyz and satisfy the same specifications. Similarly, their lower edges are called the top notches of P and are

designated TOP0,...,TOPN in increasing x-order. All these notches lie on the hyperbolic paraboloid $z=xy+\varepsilon$. We now give a more precise definition of P by characterizing its significant vertices with the system of axes indicated in Fig. 3.12-b. Note that the origin O is the intersection of BOT0 with the vertical plane passing through TOP0. The upper face of the parallelepiped lies on the plane $z=2N^2$ and its lower face, on the plane $z=-2N$. This ensures that all bottom and top notches fit strictly between these two faces. Also the parallelepiped has a depth and width of $N+2$. Figure 3.12-c gives all the coordinates of the top and bottom notches.

3.3.3 Decomposing P into Convex Parts

We define (Σ) as the portion of P comprised between the two hyperbolic paraboloid $z=xy$ and $z=xy+\varepsilon$ and the 4 planes $x=0$, $x=N$, $y=0$, $y=N$. (Σ) is a cylinder parallel to the z-axis, of height ε, whose base is the region of the hyperbolic paraboloid $z=xy$ with $0\leq x,y\leq N$ - See Figure 3.13. Let $Q_1,\ldots,Q_m$ be any convex decomposition of P and let Q_i^* denote the intersection of Q_i and (Σ). Since (Σ) lies inside P, the set of Q_i^* forms a partition of (Σ). Note that Q_i^* may consist of 0,1, or several blocks, most of which are likely not to be polyhedra. Our goal is to prove that $m\geq cN^2$ for some constant c by showing that the volume of Q_i^* cannot be too large. By volume of Q_i^*, we mean the sum of all the volumes of the blocks composing Q_i^*. We first characterize the shape and the orientation of the large Q_i^*'s, which permits us to derive an upper bound on their maximum volume.

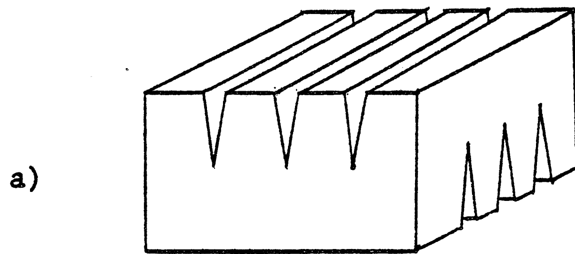
For all i between 0 and N, let BOTi* (resp. TOPi*) denote the vertical projection of BOTi (resp. TOPi) on the plane Oxy. The set of all BOTi* and TOPi* forms a regular square grid of N^2 cells, each cell being itself a one-by-one square. Consider the two points $A:(x_A,y_A,z_A)$ and $B:(x_B,y_B,z_B)$ lying in Q_j^*. We will investigate their possible positions when their vertical projections on the grid lie on two parallel lines which are at a distance 2 of each other. Wlog, we will assume that $x_A\leq x_B$. We have the following result:

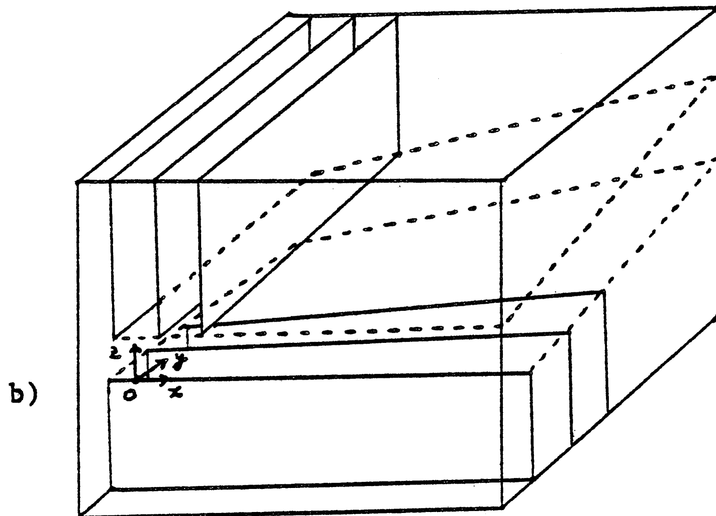
Lemma 22: Let A and B be two points of Q_j^*.

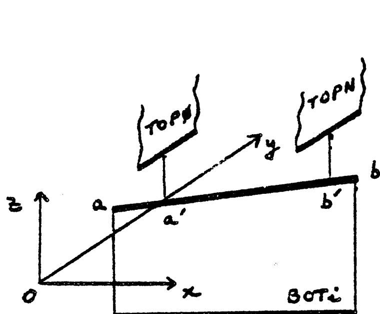
1. If x_A is an integer i with $0\leq i\leq N-2$ and $x_B=x_A+2$, $y_B-y_A\leq 2\varepsilon$.

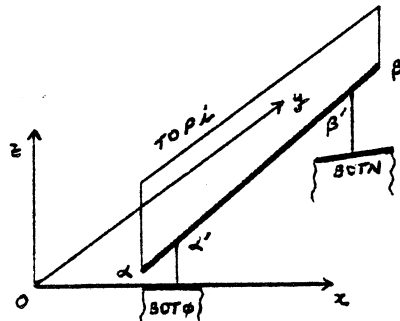
2. If y_A is an integer i with $2\leq i\leq N$ and $y_B=y_A-2$, $x_B-x_A\leq 2\varepsilon$.

Proof: Recall that the line supporting BOTi and TOPi are defined respectively by $(y=i,z=ix)$ and $(x=i,z=iy+\varepsilon)$.









c) $a'\begin{cases}0\\ i\\ 0\end{cases}$ $a\begin{cases}-1\\ i\\ -i\end{cases}$ $b'\begin{cases}N\\ i\\ iN\end{cases}$ $b\begin{cases}N+1\\ i\\ i(N+1)\end{cases}$ $\alpha'\begin{cases}i\\ 0\\ \varepsilon\end{cases}$ $\alpha\begin{cases}i\\ -1\\ \varepsilon-i\end{cases}$ $\beta'\begin{cases}i\\ N\\ iN+\varepsilon\end{cases}$ $\beta\begin{cases}i\\ N+1\\ i(N+1)+\varepsilon\end{cases}$

<u>Figure 3.12</u>: The polyhedron P.

1) Let the coordinates of A and B be respectively $(x_A=i,y_A,z_A)$ and $(x_B=i+2,y_B,z_B)$ with $0\leq i\leq N\text{-}2$. Let T be the middle point of the segment AB, $(x_T=i+1,y_T=(y_A+y_B)/2,z_T=(z_A+z_B)/2)$, and consider the point C on TOPi+1 with coordinates $(x_C=x_T,y_C=y_T,z_C=x_Cy_C+\varepsilon)$. Since Q_i is convex, the whole segment AB lies in Q_i and T lies inside P, therefore $z_T\leq z_C$. Also, since A and B lie in (Σ), $x_Ay_A\leq z_A$ and $x_By_B\leq z_B$, then $(x_Ay_A+x_By_B)/2\leq z_T$. Combining these results yields $(x_Ay_A+x_By_B)/2\leq z_C$, therefore

$$iy_A+(i+2)y_B \leq 2(\varepsilon+(i+1)(y_A+y_B)/2)$$

Hence

$$y_B\leq y_A+2\varepsilon$$

2) The proof is very similar. The coordinates of A and B are respectively (x_A,i,z_A) and $(x_B,i\text{-}2,z_B)$ with $2\leq i\leq N$. The middle point of AB is now defined by $T:(x_T=(x_A+x_B)/2,y_T=i\text{-}1,z_T=(z_A+z_B)/2)$ and lies right above the point of BOTi-1, $C:(x_C=x_T,y_C=y_T,z_C=x_Cy_C)$, therefore $z_C\leq z_T$. Since both A and B belong to (Σ), $z_A\leq x_Ay_A+\varepsilon$ and $z_B\leq x_By_B+\varepsilon$, therefore

$$2(i\text{-}1)(x_A+x_B)/2 \leq 2\varepsilon+ix_A+(i\text{-}2)x_B$$

and

$$x_B\text{-}x_A \leq 2\varepsilon$$

which completes the proof. □

When A is now any point in (Σ) with $0\leq x_A\leq N\text{-}2$ and $2\leq y_A\leq N$, we can still use the previous result to delimit the region where B cannot lie. The shaded area in Figure 3.14 represents the forbidden area. Assume that $x_B\text{-}\lceil x_A\rceil>2$ and let A' and B' be the two points on the segment AB with $x_{A'}=\lceil x_A\rceil$ and $x_{B'}=x_{A'}+2$. Since A' and B' lie in Q_j^*, we can apply the result of Lemma 22 on these two points. It follows that $y_{B'}\text{-}y_{A'}\leq 2\varepsilon$, therefore

$$(y_B\text{-}y_A)/(x_B\text{-}x_A)=(y_{B'}\text{-}y_{A'})/(x_{B'}\text{-}x_{A'})\leq\varepsilon.$$

This shows that B must lie under the line $y=y_A+\varepsilon(x\text{-}x_A)$ as indicated in Figure 3.14. Similarly, we can show that if $\lfloor y_A\rfloor\text{-}y_B>2$, B must lie on the left-hand side of the line $x=x_A+\varepsilon(y_A\text{-}y)$.

We can now attack our main problem, that is, evaluating the maximum volume of Q_j^*. Recall that Q_j^* may be empty or consist of several blocks. Let A be the leftmost point of Q_j^*, that is, the point on a boundary of Q_j^* such that all of Q_j^* lies totally in the halfspace $x\geq x_A$. We will assume that A does not lie too close to BOT0 or TOPN in order to have the points B and C of Figure 3.15 well defined. More precisely, we require that

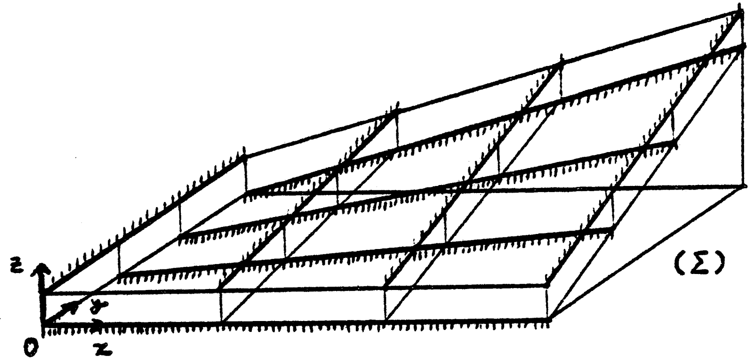


Figure 3.13: The warped region Σ.

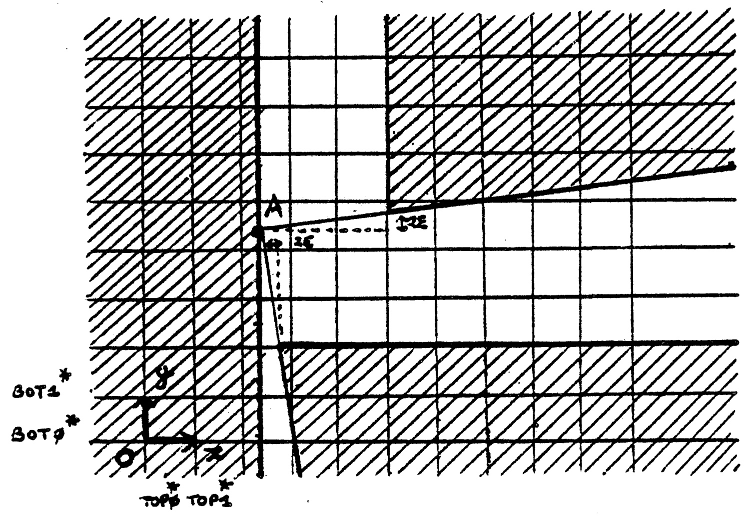


Figure 3.14: The forbidden area.

$$0<x_A<N-2\,,\ 2<y_A<N-3\varepsilon$$

Figure 3.15 is a mere reproduction of Figure 3.14 specifying the regions of interest with respect to A. Note that VA, VB, and VC really denote the intersection of (Σ) with the vertical cylinders whose bases are represented by the shaded areas in Figure 3.15. We have just shown that all of Q_j^* lies in VA U VB U VC. Then we can partition Q_j^* into 3 parts, VA1, VB1, and VC1, defined as the intersection of Q_j^* with VA, VB, and VC respectively.

I) Evaluating the volume of VA1

When there is no ambiguity, we will refer to a three-dimensional object and its volume by the same symbol (in this case VA1). Consider the set of all vertical planes forming a fixed (small) angle θ with the x-axis, that is,

$$\{ Pw : y = x\tan\theta + w \}$$

Let $S(\theta,w)$ be the area of the cross section formed by the intersection of Pw and VA1. The volume of VA1 can be computed by integrating $S(\theta,w)$ along a line normal to the planes Pw.

$$VA1 = \int S(\theta,w)\cos\theta\, dw$$

If we choose θ larger than (Ox,AB) (Fig.3.15), all values of $S(\theta,w)$ will be null beyond A and D, that is, for:

$$w > w_A = y_A - x_A\tan\theta$$

and

$$w < w_D = y_D - x_D\tan\theta$$

Letting $S(\theta)$ be the maximum value of $S(\theta,w)$ for all w, we have

$$VA1 = \int S(\theta,w)\cos\theta\, dw \le (w_A - w_D)S(\theta)\cos\theta$$

and from $y_A - 3 \le y_D$ and $x_D = N$, we derive

$$(1)\quad VA1 \le (3 + N\tan\theta)S(\theta)\cos\theta$$

The condition on θ being easily expressed as

$$(2)\quad \varepsilon < \tan\theta$$

We are now reduced to establishing an upper bound on $S(\theta,w)$. We will find it more convenient to change the system of coordinates so that the point (0,w,0) becomes the new origin and the line ($z=0, y=x\tan\theta+w$) becomes the new x-axis. We express the old coordinates (x,y,z) of any point in terms of the new coordinates (X,Y,Z) as follows:

$$x = X\cos\theta - Y\sin\theta$$

$$y = w + X\sin\theta + Y\cos\theta$$

$$z = Z$$

The hyperbolic paraboloid $z=xy$ is now described by the equation:

$$Z = (X\cos\theta - Y\sin\theta)(w + X\sin\theta + Y\cos\theta)$$

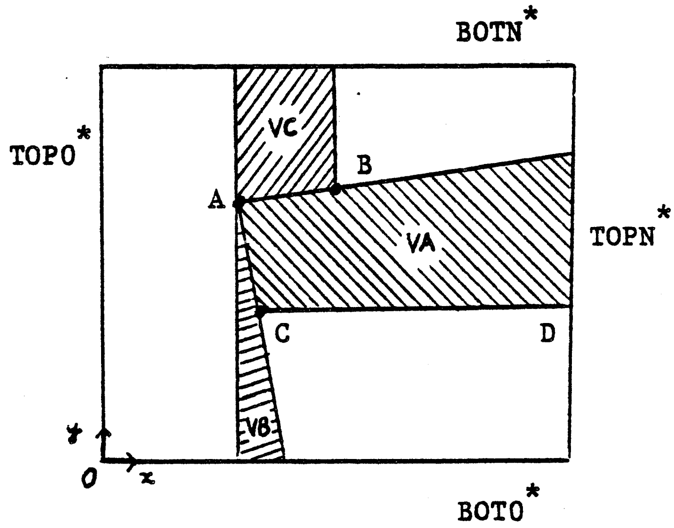


Figure 3.15: Restricting the domain where Q_j^* has to be computed.

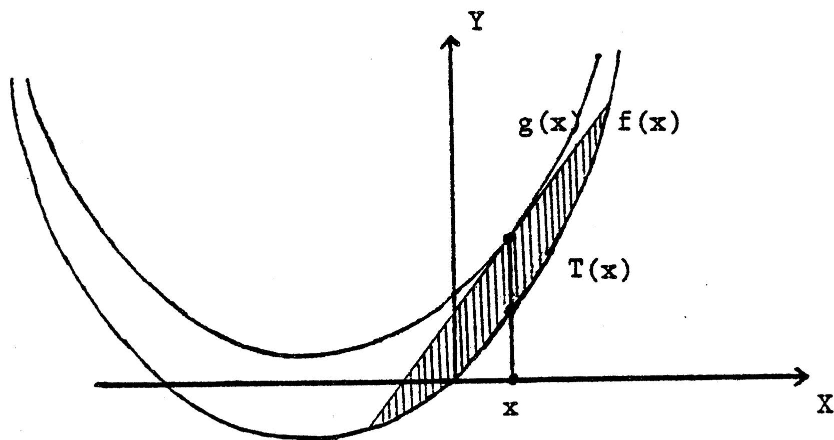


Figure 3.16: The function T(x).

and the intersection of Pw with (Σ) is a strip in the plane $(Y=0)$ comprised between the two parabolas:

$$f : Z = X^2 \sin\theta\cos\theta + Xw\cos\theta$$

$$g : Z = X^2 \sin\theta\cos\theta + Xw\cos\theta + \varepsilon$$

Before proceeding further, we will prove an interesting result about areas covered by parabolas. Suppose that we have two parabolas of the previous type, described by $f(x)=ax^2+bx$ with $a>o$, and $g(x)=f(x)+\varepsilon$. Let $T(x)$ be the area comprised between the parabola f and the tangent to g at x - See Figure 3.16. We can show that

Lemma 23: $T(x)$ is a constant function equal to $4\varepsilon\sqrt{\varepsilon/a}/3$.

Proof: The tangent to g at x has the equation:

$$Y = (2ax+b)(X-x)+ax^2+bx+\varepsilon$$

and intersects the parabola f at the points with X-coordinates x_1 and x_2, solutions of

$$(2ax+b)(X-x)+ax^2+bx+\varepsilon = aX^2+bX$$

that is,

$$aX^2-2axX+ax^2-\varepsilon = 0$$

yielding $x_1=x-\sqrt{\varepsilon/a}$ and $x_2=x+\sqrt{\varepsilon/a}$. It is now straightforward to evaluate $T(x)$,

$$T(x)=\int[(2ax+b)(t-x)+ax^2+bx+\varepsilon-at^2-bt]dt$$
$$=(x_2-x_1)(\varepsilon-ax^2+ax(x_1+x_2)-a(x_1^2+x_1x_2+x_2^2)/3)$$
$$=4\varepsilon\sqrt{\varepsilon/a}/3 \quad \square$$

We will now take a closer look at the structure of the parabolic strip formed by the intersection of (Σ) and Pw which, we know, contains $S(\theta,w)$. Here again, $S(\theta,w)$ designates both the surface and its area. Recall that $S(\theta,w)$ may consist of several disconnected pieces. The intersection of Pw and (Σ) is a connected strip enclosed between two vertical lines $X=a$, $X=b$ (the exact values of a and b are irrelevant for our purposes). Also, as it can be seen in Figure 3.17-a, the upper parabola of this strip, g, intersects the top notches, TOPk, at regular intervals of length $1/\cos\theta$. Let F denote the convex polygon formed by the intersection of Q_j and Pw. We suppose that F is not empty, and we distinguish two cases:

1) No point of F lies above the parabola g (Fig. 3.17-b).

Since F is convex, there exists a line L separating g and F. We observe that the tangent to g parallel to L, L', also separates g and F. Let u be the X-coordinate of the tangent point. Reintroducing the function T defined in Lemma 23, we have $S(\theta,w)\leq T(u)$.

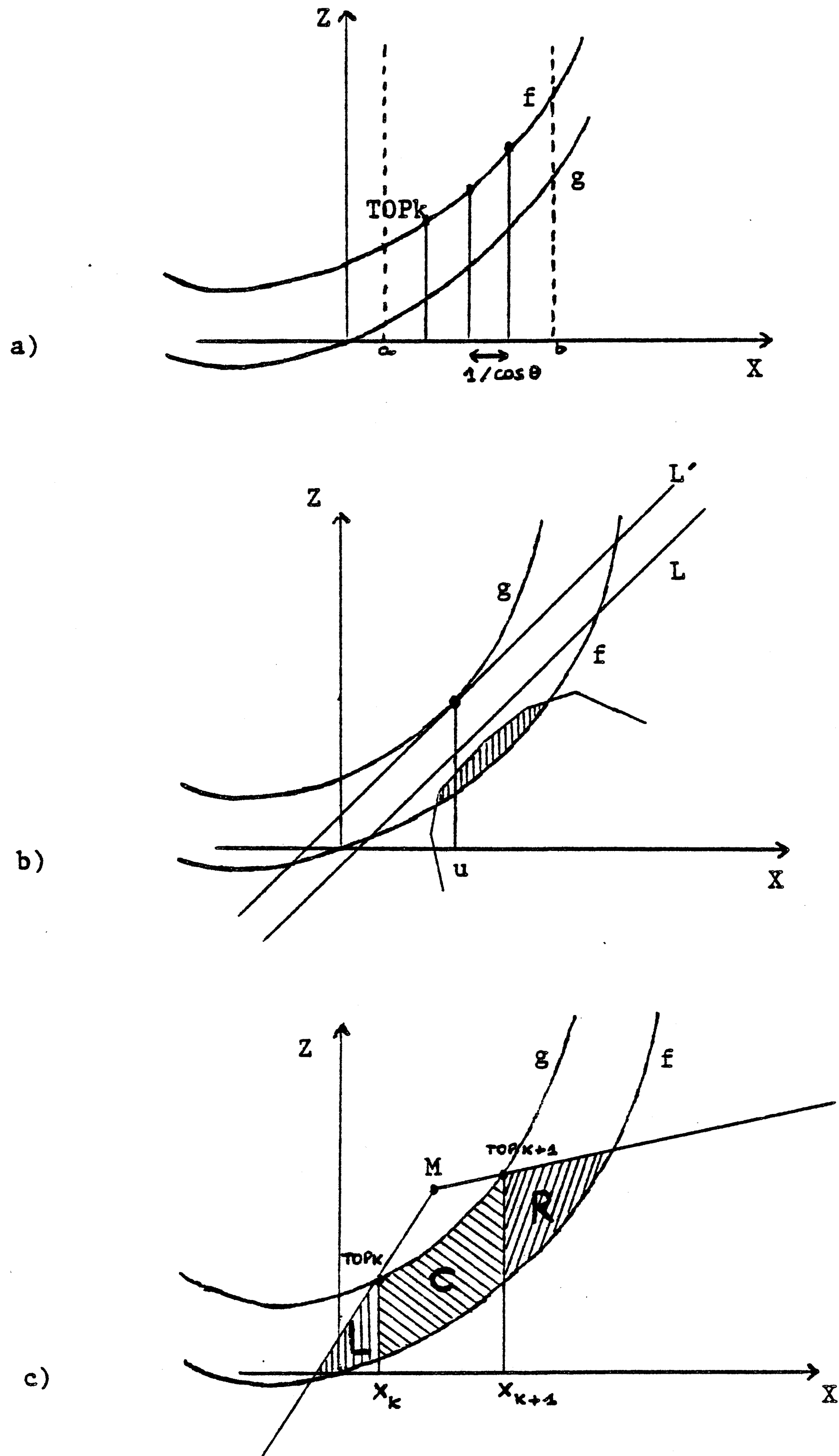


Figure 3.17: Evaluating $S(\theta,w)$.

2) There exists a point M in F lying above g (Fig. 3.17-c).

Using the notation of Fig.3.17-c, it is clear that $S(\theta,w)$ lies totally in L U C U R. Since the areas of L and R are dominated by $T(X_k)=T(X_{k+1})$, and the area of C is exactly $\varepsilon/\cos\theta$, $S(\theta,w)$ satisfies

$$S(\theta,w) \leq 2T(X_k)+\varepsilon/\cos\theta$$

Combining (1) and (2), and using Lemma 23, we find that $S(\theta,w)$ is less than or equal to

$$\varepsilon/\cos\theta+8\varepsilon\sqrt{\varepsilon/\sin\theta\cos\theta}\,/3$$

And from (1), we derive

$$VA1 \leq \varepsilon(3+N\tan\theta)(1+8\sqrt{\varepsilon/\tan\theta}\,/3)$$

II) Evaluating the volume of VC1

Since the hyperbolic paraboloids are symmetric about x and y, the same computation will give an upper bound on VC1. Note that now, no condition like (2) must be set on the angle giving the direction of integration. For convenience, we will take it equal to θ, however. Thus, we have

$$VC1 \leq \varepsilon(3+N\tan\theta)(1+8\sqrt{\varepsilon/\tan\theta}\,/3)$$

III) Evaluating the volume of VB1

The shaded area of Figure 3.15 corresponding to VB has a maximum area of $\varepsilon N^2/2$, therefore the volume of VB is dominated by $\varepsilon^2N^2/2$. This yields an upper bound on VB1

$$VB1 \leq \varepsilon^2N^2/2$$

3.3.4 The Lower Bound on the Number of Convex Parts

We can now prove our main result:

Theorem 5: There exists a constant c and a class of polyhedra involving an arbitrarily large number of notches such that each polyhedron cannot be decomposed into fewer than cN^2 convex parts, if N is the number of notches.

Proof: Recall that the volumes computed in the previous section are only relevant for the points A satisfying

$$0<x_A<N-2 \text{ and } 2<y_A<N-3\varepsilon$$

Let V be the corresponding portion of (Σ). We have

$$V = (N-2)(N-3\varepsilon-2)\varepsilon$$

Since no Q_j can contribute to the volume V more than VA1+VB1+VC1, we can derive a lower bound on the number m of convex parts Q_j.

$$m \geq V/(VA1+VB1+VC1)$$

Assume that N is large enough and that $\varepsilon < \sin\theta < \tan\theta < 1/N^2$. Relation (2) is then satisfied and we have

$$VA1, VC1 < (1+8/3)(3+1/N)\varepsilon < 16\varepsilon$$

Also

$$V > \varepsilon N^2/2$$

Therefore

$$m > \varepsilon N^2/2(32\varepsilon + \varepsilon^2 N^2/2)$$

Hence

$$m > N^2/66 \quad \text{for } N > N_0$$

which completes the proof. □

3.4 Decidability

3.4.1 Introduction

The heuristics described above provide practical answers to the decomposition problem in three dimensions, as we believe that a quest for exact solutions is unreasonable and beyond practicality. We have not, however, addressed the main theoretical issue yet:

```
How hard is it to compute an optimal convex
decomposition of a three-dimensional polyhedron ?
```

Since the number of convex decompositions of a polyhedron is uncountably infinite, no exhaustive enumeration is possible. In two dimensions, we were able to make the problem finite by showing that optimal convex decompositions could be composed of X4- and Y-patterns. Such a result is unlikely here, but we will show that a finite decision procedure is still possible. However, extending this result to a polynomial-time algorithm appears to be very difficult and may well be impossible.

We need to introduce some new notation. Given a convex decomposition of a polyhedron P with n vertices, p edges, q faces, and N notches, we call S the set of convex parts. In the following, S will be regarded as a complex, that is, a set of pairwise disjoint polyhedra sharing common faces, edges, and vertices [Grunbaum,67]. Also, we will assume all complexes to be given in a "normal" form. By this, we mean that each vertex is at least of degree 3, and if two faces F and G of two distinct convex parts intersect in a polygon H, the 3 polygons F, G, and H are identical. We require that any pair of polyhedra should have at most one common face. We can easily see how any decomposition can be turned into a normal form. It is merely a question of defining faces properly. The degree requirement on vertices being trivial, let F be a face of any convex part. We first redefine this face to be the largest polygon containing F and lying totally on the boundary of the polyhedron. This being done for all faces in S, we next refine each face F by considering the intersection of F with all the other faces and decomposing F into the non-empty intersections. Since this operation is symmetric, in the end, each face between two adjacent polyhedra will be a face of both polyhedra, and by convexity, the two polyhedra will have a unique common face. Thus the decomposition will have a normal form. Note that this form implies in particular that if an edge of a convex part lies on the boundary of another polyhedron of the decomposition, it must also be an edge of this polyhedron.

This allows us to define the following sets without ambiguity. V,E,F are respectively the set of all

vertices, edges, and faces involved in S (each of them occurring only once in its set). A vertex, an edge, or a face is said to be external if it lies entirely on the boundary of P (note that it is not necessarily a vertex, edge, or face of P). We designate the sets of external vertices, edges, or faces by V_e,E_e,F_e respectively. Similarly, all the other vertices, edges, and faces involved in S are called internal, and form the sets V_i,E_i,F_i. Finally, we define V_0,E_0,F_0 as the sets of vertices, edges, and faces of P. Note that E_0 (resp. F_0) may not be a subset of E_e (resp. F_e).

For simplicity, we will use the same notation to designate a set or its cardinality when there is no ambiguity. For example, we can write the relations

$$V_0 = n\,,\; E_0 = p\,,\; F_0 = q$$
$$V = V_e + V_i\,,\; E = E_e + E_i\,,\; F = F_e + F_i$$

3.4.2 The OCD Problem Is Decidable

Here is an overview of the proof. We begin by showing that the maximum number of vertices in an OCD is bounded. Then we can define the scheme of a decomposition as a topological description of the corresponding complex, and proceed with a combinatorial enumeration of all possible schemes. Each scheme must be tested for "realizability", that is, it must be determined whether the scheme corresponds to an actual convex decomposition. A system of equations involving the coordinates of the vertices can be set up to express realizability, from which the problem becomes purely algebraic. Finally, we obtain an OCD as any realizable scheme of minimum cardinality.

Before proceeding with a description of the decision procedure, we need some preliminary results. Our first task will be to show that the number of internal faces is polynomial in the number of convex parts. Note that this is trivially false if the convexity requirement is relaxed.

Lemma 24: $F_i \leq S(S-1)/2$

Proof: We identify the polyhedra and the internal faces in S with respectively the nodes and the edges of a graph. An edge will connect two nodes if and only if the corresponding polyhedra share a common face. Since S is given in a normal form, each internal face corresponds to a distinct edge in the graph and there are no multiple edges. The result is then immediate. □

REMARK: This upper bound is tight up to a constant factor, since we observe that there exist classes of decompositions for which we have $F_i \geq kS^2$ for some constant k.

The next result will provide an upper bound on the total number of vertices in S.

Lemma 25: If S is an OCD of P, $V = O(V_0^5)$

Proof: We first ensure that the boundary of P has been triangulated, that is, each face is a triangle. This can be done without adding new vertices to V_0. Since the boundary of P has the structure of a connected planar graph, we have the well-known relations:

$$(1)\ E_0, F_0 = O(V_0)$$

Considering the complex S, we observe that each edge has two endpoints and each endpoint is adjacent to at least 3 edges, therefore $V \leq 2E/3$. Euler's formula can be generalized to complexes, and yields $V = S+1+E-F$ [Grunbaum,67], leading to

$$(2)\ V \leq 2(F-S-1)$$

Like the faces of P, F_e forms a planar graph embedded on the boundary of P. Moreover, the faces of F_e are subfaces of F_0, that is, each face of F_e lies entirely within some face of F_0. Recall that the faces of F_0 are triangles. Let h be the maximum number of faces lying inside a single face of F_0. We have $F_e \leq hF_0$. Inside a triangle of F_0 lies either no edge at all or a planar graph whose edges lie on the boundary of some internal faces. Since internal faces are convex, each contributes to at most one edge of this graph, and relation (1) applied to this planar graph shows that $h = O(F_i)$, therefore $F_e = O(F_0 F_i)$. Since $F = F_e + F_i$, we derive from (2), $V = O(F_0 F_i)$, and from Lemma 24, $V = O(F_0 S^2)$. S being an optimal decomposition, we have already seen that its cardinality satisfies $S = O(N^2)$, and since the notches are edges of P, we have $S = O(E_0^2)$. This shows that $V = O(F_0 E_0^4)$, which combined with relation (1) establishes the lemma. □

REMARK: We do not believe V_0^5 to be optimal.

This result enables us to test out all possible complexes involving $O(V_0^5)$ vertices and select the OCD's. The method is similar to the proof given by Grunbaum to show that it is possible to enumerate all different combinatorial types of d-polytopes with a given number of vertices [Grunbaum,67].

We define the scheme of S or of any complex in general as an enumeration, for each polyhedron in the complex, of all its vertices, edges, and faces (edges and faces are given by the subsets of vertices they involve). Giving the location of the vertices of P along with the scheme of S specifies the

decomposition of P exactly. Let W be a subset of V, or a set of subsets of V, or a set of sets of subsets of V (and so forth); we define L(W) to be the subset of V which contains exactly the vertices involved in W.

We next introduce the notion of abstract scheme. An <u>abstract scheme</u> for P is a family A of non-empty sets $A_1,\ldots,A_p$. Each A_i is a set of subsets of a set $V=\{v_1,\ldots,v_k\}$ with $n=V_0\leq k$. $v_1,\ldots,v_n$ correspond to the n vertices of P and have specified coordinates, whereas each v_i for $i>n$ represents a Steiner point of the decomposition and is assigned three variables x_i,y_i,z_i corresponding to its coordinates. Moreover, we require that $L(A)=V$ and that condition (ST) be satisfied for all $W=A_1,\ldots,A_p$.

<u>Condition (ST)</u> :

1. L(W) does not belong to W.

2. If x belongs to L(W), then the singleton {x} belongs to W.

An abstract scheme of P describes a tentative decomposition of P. Each A_j corresponds to a polyhedron of the decomposition, and the requirements can be interpreted as follows: 1) V indeed represents the set of all vertices. 2) All the vertices of A_j cannot lie on the same face (clause 1. of condition ST). 3) All the vertices of A_j involved in edges or faces are explicitly listed as vertices (clause 2. of condition ST). A scheme of P is an abstract scheme for P. Conversely we want to investigate the conditions for an abstract scheme to be "realizable", that is, to be isomorphic to the scheme of a convex decomposition of P. The idea is to list out all possible abstract schemes which can lead to an OCD, and keep the optimal ones among the subset of those which are realizable. We have the following characterization of realizability.

Lemma 26: The abstract scheme A for a non-convex polyhedron P is realizable if and only if it is possible to find reals (x_i,y_i,z_i) for $i=n+1,\dots,k$ such that for every non-empty family W of non-empty subsets of V which satisfies condition (ST), propositions (A) and (B) are equivalent.

(A) W belongs to A.

(B) (1) For every non-empty subset of L(W), Z, propositions (a) and (b) are equivalent.

(a) Z belongs to W.
(b) There exist reals a,b,c such that
$ax_i+by_i+cz_i=1$ if v_i belongs to Z.
>1 if v_i belongs to L(W)-Z.

(2) Any point in the convex hull of L(W) lies in P.

(3) For all A_i in A, if L(W) and $L(A_i)$ are distinct, no point in the convex hull of $L(A_i)$ lies strictly in the convex hull of L(W).

Proof: Basically, (A)=>(B) ensures that all the polyhedra are convex, disjoint, and lie inside P, whereas (B)=>(A) guarantees that their union gives P. We first show that (1) holds if and only if W is the scheme of a convex polyhedron with vertices L(W). (b) expresses the fact that all the vertices of Z lie on a plane while the others do not, yet lie on the same side. (a)=>(b) means that all the vertices, edges, and faces expressed in W lie on the convex hull of L(W). Also, the requirement [v belongs to L(W) => {v} belongs to W] guarantees that all the vertices of L(W) are vertices of the convex hull of L(W). Conversely, (b)=>(a) shows that W lists exactly all vertices, edges, and faces of the convex hull of L(W). We can now turn to the main characterization:

(A)=>(B) implies that each A_i determines a convex polyhedron (1), which lies inside P (2), and that all of them are pairwise disjoint (3). Here again, (B)=>(A) guarantees that A gives a complete description of all the convex parts of a decomposition. To see that, assume that a point M lies inside P though in none of the polyhedra determined by the A_i. Then consider the polyhedron Q which contains M, lies inside P but outside of the A_i, and

has maximum volume. It is easy to see that Q always contains a tetrahedron made of vertices of Q. This tetrahedron has its vertices in L(A) and satisfies (1),(2),(3), therefore (B)=>(A) shows that its description should be listed in A, which leads to a contradiction and completes the proof. □

We can now prove our main claim.

Theorem 6: The problem of decomposing any polyhedron into a minimum number of convex parts is decidable.

Proof: The crux of the argument is a fundamental result by Tarski on the decidability of first-order sentences in the field of real numbers - See [Tarski,51], but also [Seidenberg,54], [Cohen,69], and [Monk,75]. This result asserts that every statement in elementary algebra (theory of real numbers) containing no free variables is effectively decidable. It is easy to see that the characterization of Lemma 26 can be expressed as such a logical statement. The only difficulty may come from (2) and (3), but can be solved by the following remarks:

If (B) is to be true, then (1) must be true and the W and A_i involved in (2) and (3) are the schemes of convex polyhedra. It follows that all the point in their convex hull can be expressed by a linear combination of the vertices with positive coefficients summing up to at most one. Inclusion inside a convex polyhedron can be expressed by a set of inequalities, and for testing the inclusion in P, we might first decompose P into convex parts (using the naive decomposition for example), and test the inclusion in each part. Having proved that any abstract scheme for P can be shown to be realizable or not, we may simply list all possible abstract schemes involving $O(V_0^5)$ vertices. Lemma 25 shows that if we list them by increasing number of families A_i, the first scheme found realizable will provide an OCD of P. □

The decision procedure described above does not give any practical means for computing OCD's. We have made no attempt to analyze its performance, but from [Monk,75] we conjecture that $2^{2^{O(n)}}$ is an upper bound on its execution time.

3.5 Conclusions

Earlier in our development, we presented a couple of efficient heuristics for decomposing a polyhedron into convex pieces, both of which are worst-case optimal (up to a constant factor) in the number of parts. These methods are refinements of the naive decomposition and unfortunately do not discriminate among particular shapes. For example, it is often the case that two notches will be adjacent and can be removed with the same cut. This simple observation may reduce the number of convex parts by half. More generally, we believe that efficient special-purpose heuristics could be developed along these lines. An interesting case is to restrict the domain of polyhedra to "architectural" designs, where for example all the edges lie on three possible perpendicular directions, one vertical and two horizontal. Another restriction may further force the convex parts to be rectangular parallelepipeds. All these problems are highly practical and are still open.

Returning to the general OCD problem in three dimensions, we observe that generalizations of X-patterns are indeed possible. X-patterns can be viewed as sets of connected faces, and rather amazingly, a single one can remove all the notches of a polyhedron and produce only two convex pieces - See Figure 3.18. If, however, we want our decompositions to be "robust", that is, to remain convex under small perturbations of the polyhedron, such powerful X-patterns are inapplicable. For example, any cut which removes two non-adjacent notches implies that these notches are coplanar and must assume that they will remain so. Thus, a quest for robustness must exclude these types of cuts. Note that in two dimensions, any decomposition is necessarily robust, while in three dimensions, the naive decompositions we described earlier also have this property. We conjecture that a robust decomposition of a polyhedron with N notches must always produce at least $c\sqrt{N}$ convex parts, and that this lower bound is actually achievable.

Only in two and three dimensions is the concept of non-convex polyhedra totally natural. In higher dimensions, convex polyhedra are still easily expressed as intersections of halfspaces, but non-convex polyhedra do not lend themselves to such easy descriptions. One method is to express a polyhedron as a connected union of convex polyhedra. Note that the convex polyhedra may overlap, thus do not necessarily constitute a convex decomposition of the polyhedron. This representation is common in linear programming, when the constraints are expressed by k sets of inequalities and at least one set has to be satisfied. If we can find a convex decomposition of the polyhedron into p parts with $p \ll k$, and if each convex part has relatively few faces, testing the feasibility of a point can be greatly simplified by testing its inclusion in any of the p convex parts. Here again, optimal decompositions do not seem to be within reach in general and perhaps only efficient heuristics should be sought.

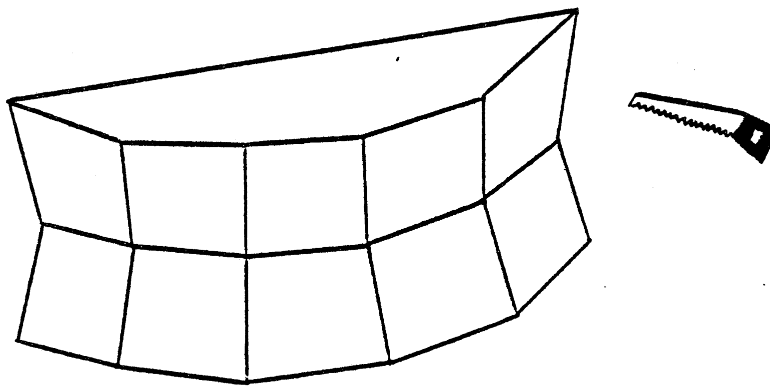


Figure 3.18: A very efficient X-pattern in three dimensions.

Chapter 4
INTERSECTION OF CONVEX OBJECTS

4.1 Introduction

Perhaps the most important application of computational geometry involves determining whether a pair of convex objects intersect. In computer graphics, two objects with intersecting projections on the viewing plane are often submitted to special treatments such as hidden-line elimination. Architectural design must be careful to prevent undesired intersections. Also, the importance of efficient algorithms for detecting intersections has become apparent with the advent of VLSI technology, where several hundreds of thousands of components can be held on a single chip [Mead and Conway,80].

In many cases, however, only detecting an intersection or a non-intersection is required, and we do not have to compute the intersection explicitly. Indeed, many applications require only a knowledge of only portions of the intersection or of the existence of an intersection rather than a complete description of any intersection. For example, in VLSI design, testing the correctness of a circuit layout is a formidable task and only the most "sensitive" portions of the chip can be examined thoroughly. A standard technique consists of covering the main parts of the layout with rectangles and reporting all the intersecting pairs. In a second stage, only the reported pairs of rectangles are examined in detail [Bentley, Haken and Hon,80]. Similar methods are often used in computer graphics, when we wish to clip or window a scene. Then it is often sufficient to identify those polygons which would require further processing [Newman and Sproull,79]. This technique called "boxing" belongs to a widespread class of geometric treatments where typically only a witness to the intersection or non-intersection is required. Such a witness may consist of a point common to both objects or a separating hyperplane if the objects do not intersect. In addition to graphics and VLSI, computer geography, computer-aided design, and computer animation would benefit greatly from

efficient algorithms for solving this type of problem. More generally, many applications to which such algorithms might be useful require a gross procedure which detects the possibility of an intersection from which refined procedures can handle the small number of cases in which an intersection has been reported and must be computed.

Other applications can be found in real-time environments, where the data is constantly updated and any treatment of it must perform extremely fast. Once again, all the data is in memory and only the samples of the input needed for the algorithm have to be accessed. Such situations are common in real-time simulation and optimal control.

We thus wish to depart from the classical problem

```
(1) Given objects P and Q, compute
    their intersection.
```

and restate our basic question as follows

```
(2) Given objects P and Q,  provide
    a witness to their intersection
    or non-intersection.
```

Problem (1) has been well studied, resulting in linear lower bounds and linear or quasilinear upper bounds [Shamos,75,76], [Muller and Preparata,77], [Bentley,78]. Lower bounds for this problem use arguments claiming that linear time is required to read all inputs or report the output. For the problem which we pose, such arguments do not apply. We only require a witness to the intersection or non-intersection of P and Q, and we further assume that the objects we wish to intersect are available (i.e., in memory) so that we cannot rely on input time to yield a linear lower bound.

With a decomposition procedure at our disposal, we can always assume the objects to be convex. Arbitrary designs in two and three dimensions can be first decomposed into convex parts, each of which is then tested for intersection. This approach is justified by the greater efficiency derived from convexity. Indeed, we will present sublinear algorithms for solving problem (2) for all convex designs in two and three dimensions.

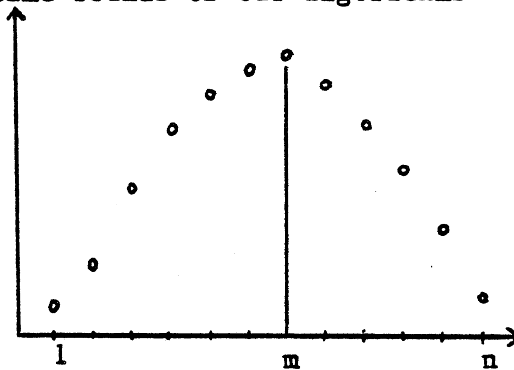
The table in Figure 4.1 summarizes our results and clearly shows that also in geometry, detection is "easier" than computation (this has been known for a long time in data structures). Except for the intersection of a polygon and a line [Dobkin,78], all the algorithms given are original. The time bounds are achieved by using the standard array representation for two-dimensional objects and a special representation of polyhedra which requires $O(n^2)$ operations to reach from the standard representation (where n denotes the total number of vertices). An O(nlogn) preprocessing of the standard representation is actually sufficient, but the running times given here must then be

multiplied by a factor $\log^2 n$ [Dobkin and Munro,80]. Note that convex polyhedra have a structure of planar graph, thus the number of vertices, edges, and faces are linearly dependent, and any of these measures can be used to represent the input size. Although the times given in the table are asymptotic, the constants involved are sufficiently small to make the algorithms viable in practice.

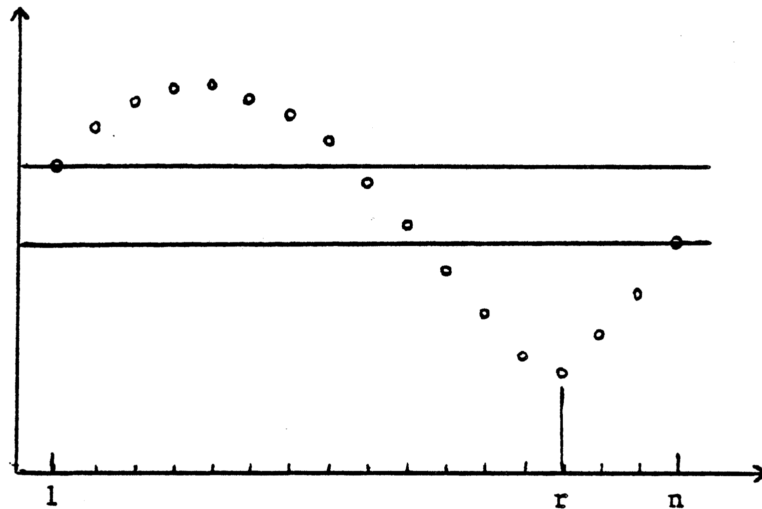
All these algorithms rely on a small number of unifying concepts. Convexity combined with random-access capabilities allows for binary and Fibonacci search, and it is with an explanation of these basic principles that we start our analysis. Section 4.2 is devoted to the two-dimensional case while Section 4.3 investigates the problem cast in three dimensions.

INTERSECTED WITH	LINE	POLYGON	PLANE	POLYHEDRON
LINE	constant	log n	constant	$\log^2 n$
POLYGON		log n	log n	$\log^2 n$
PLANE			constant	$\log^2 n$
POLYHEDRON				$\log^3 n$

Figure 4.1: The time bounds of our algorithms.



a) A unimodal function.



b) A bimodal function.

Figure 4.2

4.2 Computing Planar Intersections

4.2.1 Notation

Polygons are represented by arrays with their vertices given in clockwise order. Polygon P will have vertices $p_1,...,p_p$ and polygon Q vertices $q_1,...,q_q$. All indices of P (resp. Q) are taken modulo p (resp. q) in the obvious fashion. A line will be specified by any two of its points and a segment by its two endpoints. AB will always refer to the segment from A to B, and "line(AB)" will represent the infinite line containing AB. We define d(x,L) as the orthogonal distance from the point x to the line L and h(x,L,v) as the oriented distance from x to L with respect to v. This latter quantity is defined as -d(x,L) if x and v lie on opposite sides of L and as d(x,L) if they lie on the same side. In our representation, both d and h can be computed in constant time. F_i will represent the ith Fibonacci number with $F_0=F_1=1$ and $F_N=F_{N-1}+F_{N-2}$.

4.2.2 Fibonacci Search on Bimodal Functions

A real function f defined on the integers 1,2,...,n is said to be unimodal if there exists an integer m, $1<m<n$, such that f is strictly increasing (resp. decreasing) on [1,m] (resp. [m,n]) as shown in Figure 4.2-a (we may also have $f(m)=f(m+1)$). We can compute m, where the maximum value of f occurs, by a Fibonacci search, which has been shown to minimize the maximum number of function evaluations [Kiefer,53]. This procedure is based on the observation that if $f(F_r)>f(F_{r-1})$ (resp. $f(F_r)<f(F_{r-1})$), then the maximum of f occurs in the interval $[F_{r-1},N]$ (resp. $[1,F_r]$). If r is the integer such that $F_r \le N < F_{r+1}$, we first scale the Fibonacci sequence by a scale factor $k=N/F_{r+1}$, then from the values of f at $\lceil kF_r \rceil$ and $\lceil kF_{r-1} \rceil$ we reduce the problem size by roughly the golden mean (1.618...) and repeat on the remaining interval wherein single queries will produce this reduction. The number of function evaluations needed is about $1.4404..\log_2 n$ or O(logn).

For the purpose of our analysis, we need to extend this algorithm to bimodal functions. A function f defined on 1,2,...,n is bimodal if there exists an integer r, $1 \le r \le n$, for which the sequence $f(r),f(r+1),...,f(n),f(1),...,f(r-1)$ forms a unimodal function - See Figure 4.2-b. The interest in bimodal functions comes from the following fact:

Lemma 27: Let P be a convex polygon with p vertices $p_1,...,p_p$ in clockwise order. For any line L and any point v not in L, the function defined for $i=1,...,p$ by $g(i)=h(p_i,L,v)$ is bimodal.

Proof: From now on we will assume that no more than two vertices of P can be collinear. This requirement does not limit the possible shapes of P but simply ensures that the representation used is not redundant.

Let p_k be the vertex of P which minimizes g(i) for $i=1,...,p$. In case of ties, we choose k so that the only other integer which achieves the same g value is k-1. We can do this because P is convex. We will show that the sequence $g(k),g(k+1),...,g(k-1)$ is unimodal, which suffices to prove the lemma. Let us choose a directing vector r of the line L such that the angle (r,p_kp_{k+1}) is less than 180. Recall that all angles are measured between 0 and 360 degrees in a counterclockwise motion. We define the oriented angles $a_i=(r,p_ip_{i+1})$ and $b_i=(p_ip_{i+1},p_{i-1}p_i)$ for $i=1,...,p$ as in Figure 4.3. By construction, the following relations hold for all i:

$$g(i+1) = g(i)+|p_ip_{i+1}|\sin a_i$$

$$a_{i+1} = a_i-b_{i+1} \ [\text{mod } 360]$$

Since P is convex, all b_i are less than 180 degrees, therefore the sequence $\sin(a_k),\sin(a_{k+1}),...,\sin(a_{k-1})$ will be positive then negative, thus showing that $g(k),g(k+1),...,g(k-1)$ is a unimodal sequence. fl(1)

Since a unimodal sequence has exactly one maximum and one minimum, and each of them is achieved in at most two points which must be consecutive modulo n, bimodal functions have the same property. However, to find the extrema of a bimodal function may not be as easy since we do not know the starting point of its unimodal sequence in advance. One way to get around this difficulty is to consider the line T passing through the points (1,f(1)) and (n,f(n)), that is,

$$T(x) = (x-1)(f(n)-f(1))/(n-1) + f(1)$$

First, if $f(1)=f(n)$ then f(1) is an extremum and f or -f is unimodal, showing that the extrema can be found with the previous method. Otherwise let us assume that $f(1)<f(n)$ (the case $f(1)>f(n)$ being similar). If $f(2)\geq f(n)$ then f(1) is a minimum and the subsequence $f(2),...,f(n)$ is unimodal, which solves our problem. Else, if $f(2)<f(1)$, consider the function g defined as follows:

$$g(x) = \text{Min}(f(x),T(x))$$

g can be evaluated in constant time. Also, the maximum of f being at least as large as f(n), g cannot achieve it, which shows that g first decreases then rises, that is, is unimodal. It follows that the minimum of g (which is also the minimum of f) can be found with a Fibonacci search. Let x be the integer to achieve this minimum value of g (if two consecutive integers achieve the same value, x is defined as the larger). The sequence $f(x+1),f(x+2),...,f(n)$ is unimodal, therefore the maximum of f can also be determined through a Fibonacci search. Since at most two Fibonacci searches have to be performed, we can conclude:

Lemma 28: The extrema of a bimodal function $f(1),...,f(n)$ can be computed in O(logn) time, which involves at most $2.88..\log_2 n+O(1)$ function evaluations.

4.2.3 Intersection of a Line with a Convex Polygon - (IGL)

It is now trivial to derive an algorithm for determining the intersection (null, 1 point, or two points) of an infinite line L and a convex polygon P.

Theorem 1: The intersection of an infinite line with a convex polygon with p vertices can be computed in O(logp) time.

Proof: We can always assume that p_1 does not lie on L, then it follows from Lemma 27 that the function $g(p_i)=h(p_i,L,p_1)$ is bimodal, therefore the algorithm of Lemma 28 allows us to find a vertex of P, w, which minimizes g. We know that P and L intersect if and only if g(w) is negative or zero. In the latter case, w is the unique intersection of P and L. We may find the intersection in the former case as follows.

Consider the sequence $p_1,p_2,...,w$. The first elements lie on the same side of L and the rest of the sequence lies on the other. Then using a binary search we can find, in O(logp) time that edge of P with its two enpoints lying on different sides. This intersection point is then computed in constant time. The same method applied with the sequence $(w,...,p_p,p_1)$ gives the other intersection point. f1(1)

Our algorithm involves approximately 2.8808logp + O(1) computations of g. We observe that it is trivial to extend this algorithm to the case where L is a line segment without increasing the time

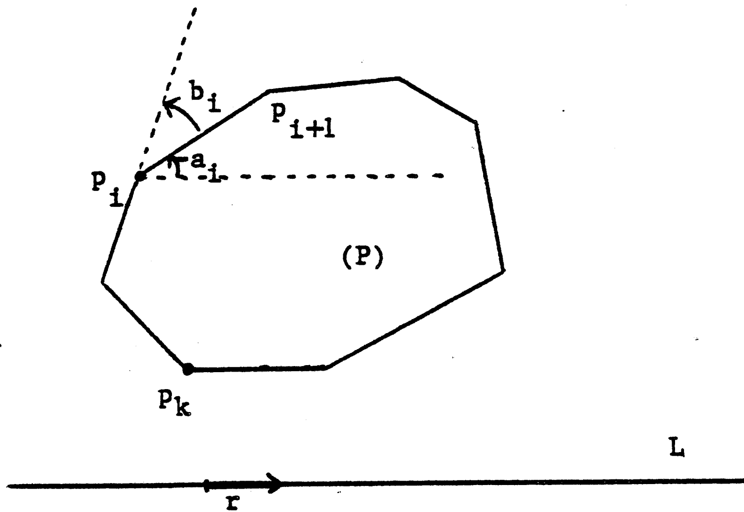


Figure 4.3: The distance from a convex polygon to a line is bimodal.

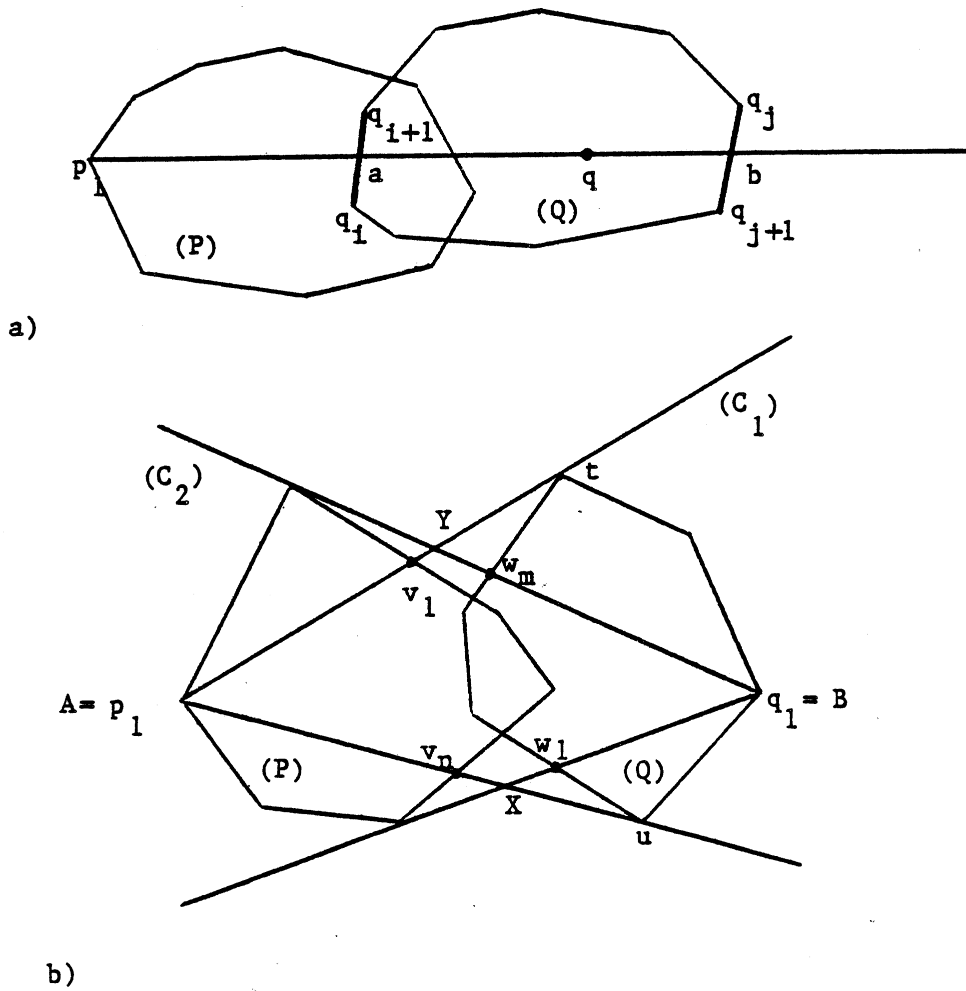


Figure 4.4: We form AYBX as a bounding quadrilateral in which $P \cap Q$ lies if it exists.

bound. Since O(logp) has been shown to be a lower bound on the time complexity for testing the inclusion of a point in a convex polygon, which is constant time reducible to our problem, the algorithm we have described is optimal in the minimax sense [Shamos,78]. In what follows, we will refer to this algorithm as IGL. Though our algorithms are more complex than IGL, they are based on principles similar to those used to derive this algorithm.

4.2.4 Intersection of Two Convex Polygons - (IGG)

The algorithm for computing the intersection of a line and a polygon suggests methods which might be used to speed up algorithms for intersecting two polygons P and Q. If we could determine which of P's sides were closest to Q (and vice versa), we would be able to reduce the problem to a small number of tests of segments intersections. Our method reduces the number of remaining edges of one of the polygons by a factor of 2 at each iteration. The algorithm we present (referred to as IGG) returns NO if P and Q do not intersect and (YES,A) if they do, where A is a point in their intersection. When a NO answer is returned, it is possible to generate in constant time a pair of parallel lines of support separating the polygons.

We begin by limiting the intersection of P and Q to a quadrilateral. This requires O(logpq) steps and also tests for simple intersections (e.g, P contained in Q). From the quadrilateral we form two chains of vertices L_v and L_w which intersect if and only if P and Q intersect. The iterative step of the algorithm is a division in which we eliminate half of either L_v or L_w. This step is reached as one of 5 possible cases determined from the structure of the remaining vertices.

It is important to keep in mind that by intersection of P and Q, we mean the intersection of the area P and Q and not of the polygonal boundaries. We shall prove further that the latter problem requires linear time whereas our problem can be solved in O(log(p+q)) time. We first give a description of the algorithm and prove its correctness, then we establish its running time.

ALGORITHM IGG:

I) - "Cover Q (resp. P) with 2 lines of support intersecting in P (resp. Q)."

a) Compute the intersection of Q with line(p_1q), where q is any point interior to Q, say, the mass center of any 3 vertices of Q. This line always intersects Q in two points a and b which the algorithm IGL can find as well as the edges of Q where a and b lie, say q_iq_{i+1} and q_jq_{j+1} respectively - See Figure 4.4-a.

b) If p_1 lies on the segment ab, it also lies in Q and the algorithm can return (YES,p_1). Otherwise we do a Fibonacci search on the sequence of oriented angles (p_1q,p_1q_k), for all q_k between q_{i+1} and q_j in clockwise order, in order to find the maximum angle. Call t the corresponding vertex of Q. Such a Fibonacci search is legitimate since the sequence is unimodal. If it were not, we could find an ordered list of three consecutive vertices of Q with the angle relative to the middle vertex smaller than both of the others. Then the line joining p_1 to this vertex would cut Q in more than 2 points, contradicting the convexity of Q. Similarly, by considering the sequence $q_{j+1},...,q_i$, we find the vertex u which minimizes the angle (p_1q,p_1q_k). Call C_1 the pencil (p_1u,p_1t) so defined - See Figure 4.4-b.

c) Apply the previous procedure (steps a,b) with q_1 relatively to P. If the algorithm does not return, it will determine another pencil, C_2, centered in q_1 and covering P. Since C_1 (resp. C_2) contains q_1 (resp. p_1), the intersection of C_1 and C_2 is a convex quadrilateral, p_1Yq_1X, as shown in Figure 4.4-b.

II) - Note that the portion of the boundary of P which lies in C_1 is a contiguous polygonal line from the intersection of p_1X and P to the intersection of p_1Y and P, and lies also in C_2. Determine its two endpoints (note that one or even both of these endpoints may be p_1). Renumber the vertices of P so that $L_v=\{v_1,...,v_n\}$ gives the vertices of this polygonal line in clockwise order (we have v_1 on p_1Y and v_n on p_1X). Throughout this chapter, any renumbering is implicit, that is, does not involve any scan through the vertices. It may simply consist of the setting of an arithmetic expression redefining the mapping. The same procedure is carried out with Q defining $L_w=\{w_1,...,w_m\}$. In what follows, we rename the former p_1 and q_1, A and B, respectively, as in Figure 4.4-b. Note that although L_v intersects AY and AX, it may also intersect BX or BY (in at most one point, though).

III) - We have now reduced the original problem to checking the intersection of L_v and L_w.

Let x (resp. y) denote the polygonal line AXB (resp. AYB). To simplify the exposition, for two points F and G, we say that F<G if F and G are both on x or both on y and F is on the path from A to G.

At this stage, we call upon the function INTERSECT(L_v,L_w) defined recursively as follows:

<u>INTERSECT(L_v,L_w)</u>

begin

Assume that n,m>5 using the procedure of the previous section if this is not the case.

$i=\lfloor n/2 \rfloor$, $j=\lfloor m/2 \rfloor$

Let F and G (resp. E,H) denote the two intersections of line(v_iv_{i+1}) (resp. line(w_jw_{j+1})) with the

boundary AYBXA. F (resp. H) is chosen such that v_{i+1} (resp. w_{j+1}) lies on the segment v_iF (resp. w_jH) - See Figure 4.5-a.

"The algorithm distinguishes between cases depending on the selective positions of GF and EH."

<u>Case</u> <u>1</u>:

```
"Either GF or EH lies on the same side of AB".
 (Fig. 4.5-a)

if G and F lie on x

   then  Lv= {v1,...,vi,vn}

   else
       if G and F lie on y

           then  Lv= {v1,vi,...,vn}

if E and H lie on x

   then  Lw= {w1,wj,...,wm}

   else
       if E and H lie on y

           then  Lw= {w1,...,wj,wm}
```

<u>Case</u> <u>2</u>:

```
"From now on, F and E (resp. G and H)
 lie on x (resp. y)."
 (Fig. 4.5-b)

if F<E and G<H

    then  return (NO)
```

Case 3:

"If the segments GF and EH intersect, let I be this intersection."
(Fig. 4.5-c)

```
if G<H and E<F

   then
      if v_i lies on GI

         then L_v= {v_1,v_i,...,v_n}

         else
            if w_j+1 lies on HI

               then L_w= {w_1,...,w_j+1,w_m}

if H<G and F<E

   then
      if v_i+1 lies on FI

         then L_v= {v_1,...,v_i+1,v_n}

         else
            if w_j lies on EI

               then L_w= {w_1,w_j,...,w_m}

else
```

Case 4:

"Av_i and Bw_j intersect"
(Fig. 4.5-d)

```
if Av_i and Bw_j intersect in R

    then  return (YES,R)
```

<u>Case</u> <u>5</u>:

```
(Fig. 4.5-e)
Let R be the intersection of Av_i and HE

if w_j lies on ER

    then
              L_v= {v_1,v_i,....,v_n}
              L_w= {w_1,w_j,....,w_m}

    else
              L_v= {v_1,....,v_i+1,v_n}
              L_w= {w_1,....,w_j+1,w_m}


"Recursive call with parameters of smaller size."

 -  INTERSECT (L_v,L_w)

end
```

Next we show that INTERSECT runs correctly within the given time bound.

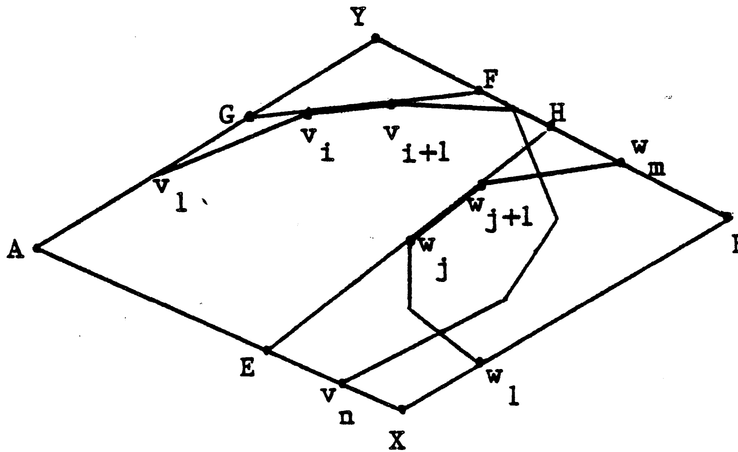
For correctness, it suffices to show that INTERSECT(L_v,L_w) indeed tests for the intersection of L_v with L_w and possibly outputs a point common to P and Q.

CASE 1: Suppose that G and F lie on y (the 3 other cases being similar) - See Figure 4.5-a. By construction, line(BY) intersects P in exactly one point which lies on the same side of B as Y. Now, since P lies totally on the same side of line(GF) as X, the intersection of P with line(BY) lies on the segment BF. Therefore, if L_v and L_w intersect, at least one intersection point lies on the polygonal line $\{v_i,...,v_n\}$. By making L_v equal to $\{v_1,v_i,...,v_n\}$, we reset the initial conditions required by the algorithm. Moreover, we note that since the area delimited by the new setting of L_v is included in P, any intersection point later output will surely be in P. This remark prevails for all the remaining cases.

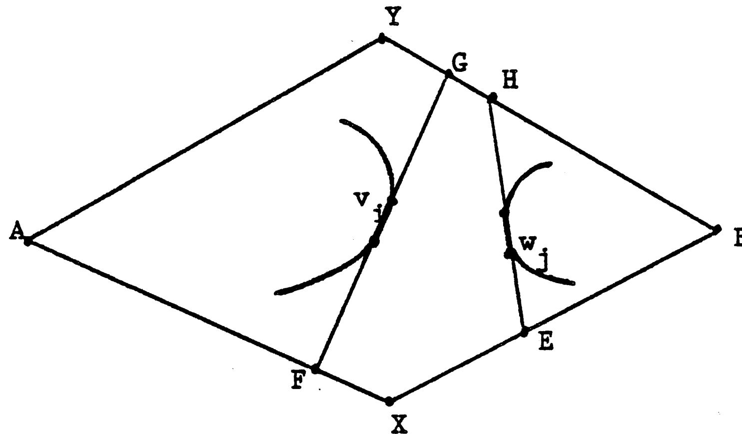
Consider the two polygons delimited by (A,x,FG,y) and (B,x,EH,y) and call V their intersection - See Figure 4.5-b. Since P and Q are convex, their intersection lies totally in V.

CASE 2: Corresponds to V empty - See Figure 4.5-b.

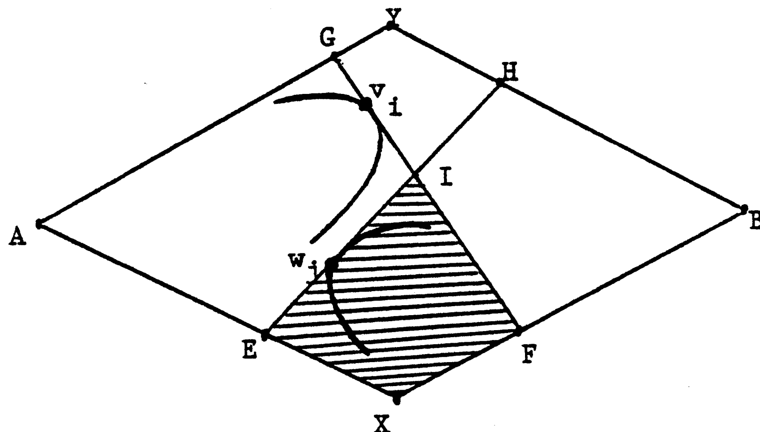
CASE 3: The first if statement supposes that E and F belong to V and the other that H and G belong to V. Since both cases are similar, we treat only the first. Suppose that v_i does not lie in V. Then, since Gv_i lies outside of EHBX, L_w cannot intersect this segment, therefore if L_w intersects



Case 1: GF (resp. EH) lies on the same side of AB, in which case we eliminate half of L_v (resp. L_w).

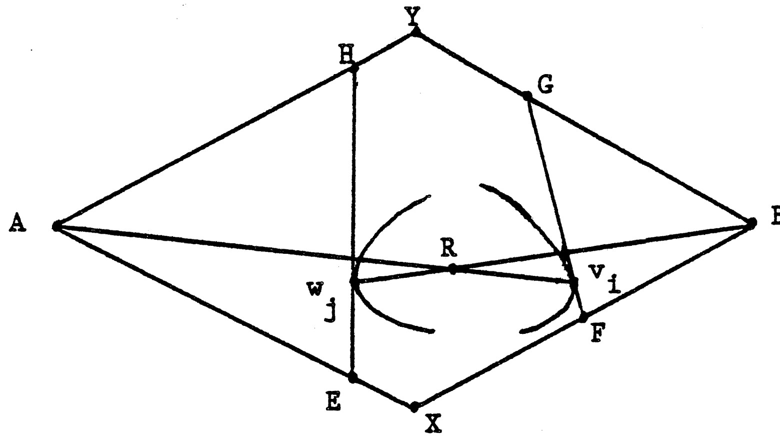


Case 2: A,F,E,B and A,G,H,B occur in this order on x and y, respectively, in which case there is no intersection.

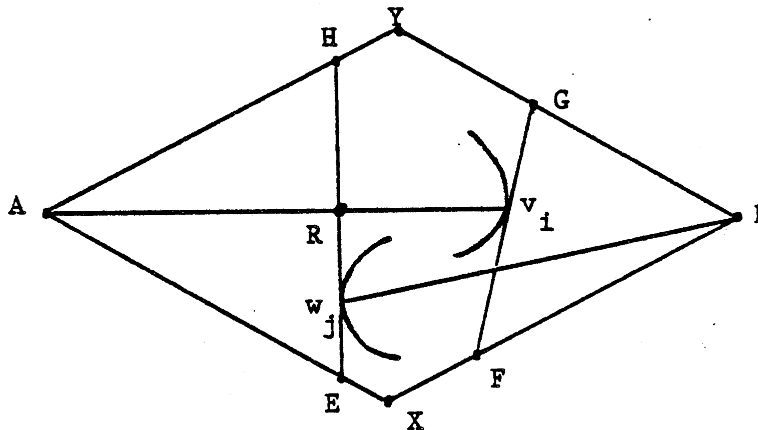


Case 3: GF and EH intersect.

(Figure 4.5 .../...)



Case 4: Av_i and Bw_j intersect, in which case $P \cap Q$ contains their intersection point.



Case 5: Av_i lies strictly "above" (or "below") Bw_j.

Figure 4.5: The algorithm INTERSECT

the polygonal line $v_1,...,v_i$, it also intersects v_1v_i. Thus the new setting of L_v is legitimate. Same with w_{j+1}. From now on, we know that both v_iv_{i+1} and w_jw_{j+1} lie on the boundary of V.

CASE 4: Assumes that Av_i and Bw_j intersect - See Figure 4.5-d. Since these two segments lie in P and Q respectively, their intersection lies in the intersection of P and Q, which is then non-empty.

CASE 5: First, we note that since E lies on x and v_i lies in V, R is well defined. We also know that Av_i and Bw_j do not intersect. The algorithm supposes successively that Av_i lies "above" and "below" Bw_j. The two cases being similar, we treat only the first. L_w cannot intersect the polygonal line $v_1,...,v_i$ without first crossing v_1v_i. Similarly, L_v cannot intersect $w_1,...,w_j$ without first crossing w_1w_j. Conversely, if either L_w crosses v_1v_i or L_v crosses w_1w_j, the intersection belongs to both P and Q. Finally, since $w_1,...,w_j$ (resp. $v_1,...,v_i$) cannot intersect v_1v_i (resp. w_1w_j), the new setting of L_v and L_w is legitimate.

To prove the time bound, we observe that the algorithm runs in constant time between consecutive recursive calls. Every call reduces a problem of size $m+n$ to a subproblem of size $m+n/2$ or $m/2+n$, and when either m or n becomes smaller than 6, the algorithm returns after $O(\log(p+q))$ operations. Therefore, the main algorithm detects the intersection of P and Q in $O(\log(p+q))$ time.

We can regard the intersection of a line with a polygon as a special case of this problem, and the results of the preceding section show that the algorithm described above is optimal in the minimax sense.

We have achieved our main goal. However, we now wish to refine the algorithm IGG so that it produces a pair of parallel separating lines (L_P,L_Q) when it fails to detect an intersection. We have preferred to present this procedure separately since there are applications where this additional information is not needed. Instead of a complicated formal definition, Figure 4.6-a illustrates best what we mean by a pair of separating lines.

Recall that the algorithm IGG fails to detect an intersection in two cases:

(1) It falls into case 2 of the INTERSECT procedure - See Figure 4.5-b. Since P lies in the pencil (BY,BX), it does not intersect line(EH), therefore line(EH) is a separating line L_Q. To compute L_P, we observe that it passes through the vertex of P which minimizes the distance to L_Q. This distance is a bimodal function for the vertices of P, therefore L_P can be determined in $O(\log p)$ time.

(2) Either L_v or L_w (say L_v) is reduced to fewer than 6 vertices ($n<6$). We say that the intersection of line(p_ip_{i+1}) with Q is positive if it is not empty and lies entirely on the same side of p_i as p_{i+1}. If it is not empty and lies totally on the same side of p_{i+1} as p_i, it is called negative. It is clear that if P and

Q do not intersect, any intersection of line(p_ip_{i+1}) with Q (called Q_i) is positive, negative, or empty. Let v_i be the middle vertex of L_v at any stage of the algorithm. We prove by induction that if $v_1=p_k$ and $v_i=p_l$, for all j between k and l-1, the intersection of line(p_jp_{j+1}) with Q (denoted Q_j) is empty or positive.

Since L_v will be eventually reduced to fewer than 6 vertices, we can assume that we never fall into case 2 (Fig. 4.5-b), nor case 4 (Fig. 4.5-d). Let p_r be the vertex v_2. By induction hypothesis, the property can be assumed to be true for all j between k+1 and r-1. It remains to show that Q_r,...,Q_l are all empty or positive. In all cases (1,2,3 - Fig. 4.5-a,c,e), we can verify that since Q lies in the pencil (AX,AY), to be negative, the intersection Q_u must occur in the triangle v_1Gv_i, which is impossible.

Let us now come back to the final stage where L_v has fewer than 6 vertices. Let p_i be the vertex v_2 and p_j the vertex v_{n-1}. We have just showed that Q_{i-1} is empty or positive. Similarly, a symmetric reasoning would show that Q_j is empty or negative. It follows that if some Q_k among Q_{i-1},...,Q_j (note that there are at most 4 of them to consider) is empty, L_P can be set to Q_k - See Figure 4.6-a. Otherwise, there exists a pair (Q_{k-1},Q_k) with Q_{k-1} positive and Q_k negative (Fig. 4.6-b). Observing that the angles (p_kq_1,p_kq_l) form a bimodal sequence for l=1,...,q (we should here measure the angles counterclockwise between -180 and +180 degrees), we can find the vertex x (resp. y) of Q which minimizes (resp. maximizes) that angle, in O(logq) time. Finally, it is clear that L_P can be set to the line passing through p_k and perpendicular to the bisector of (p_kx,p_ky). L_Q is then obtained by minimizing the distance to P, as explained in (1). We can conclude

Theorem 2: An intersection between two convex polygons with p and q vertices, respectively, can be detected in O(log(p+q)) time. A common point is returned when the polygons intersect and a pair of parallel separating lines otherwise.

While the previous algorithm can decide whether P and Q intersect, it is unable to tell whether one polygon lies strictly inside the other, that is, whether the boundaries of P and Q intersect or not. This is because the more general problem of deciding whether two convex polygonal lines intersect requires linear time to be solved. To see this, consider two polygons P and Q given in the complex plane with vertices of P being the roots of $z^n-1=0$ and the vertices of Q the roots of $z^n-[1/2+1/(2\cos(2\pi/n))]^n=0$. It can be easily verified that for any consecutive vertices a,b,c on the boundary of Q, neither the edge ab nor bc intersects the boundary P, whereas the segment ac does - See Figure 4.7. So, any vertex of Q can be moved along a radius to create an intersection, without altering any of the n-1 remaining vertices. Therefore any algorithm checking the intersection of the boundaries of P and Q has to look at all the vertices of Q, yielding the claimed lower bound.

Theorem 3: Testing the intersection of two convex polygonal lines requires linear time.

Thus no general extensions of the algorithm in the plane are possible. However, there are many

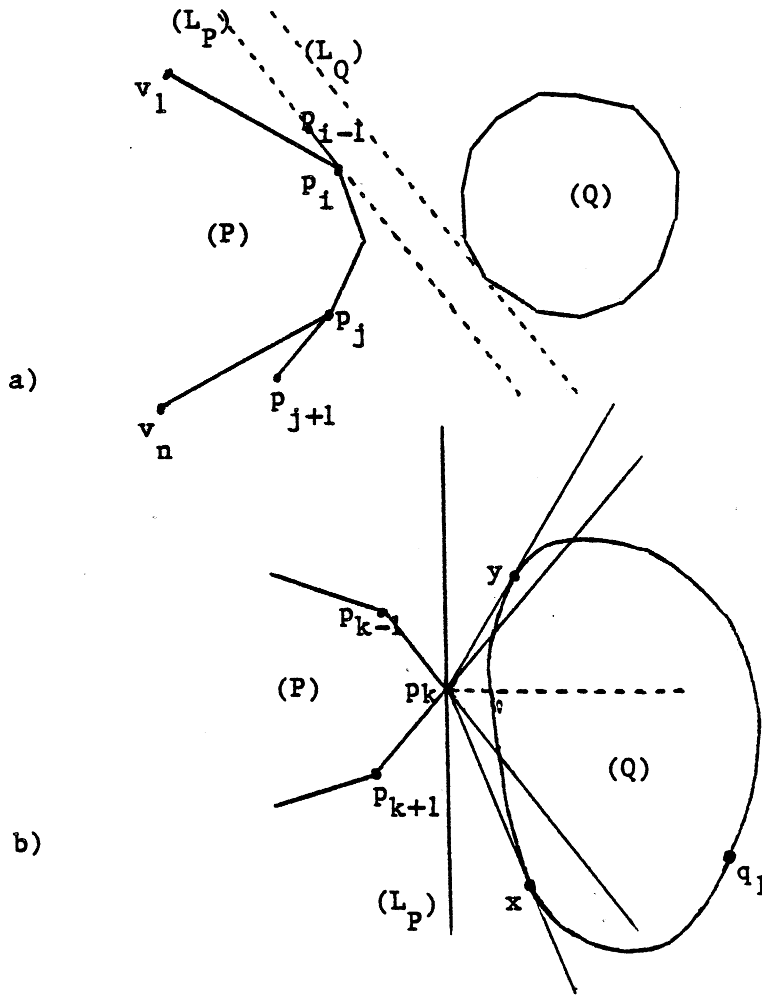


Figure 4.6: Computing a pair of separating lines.

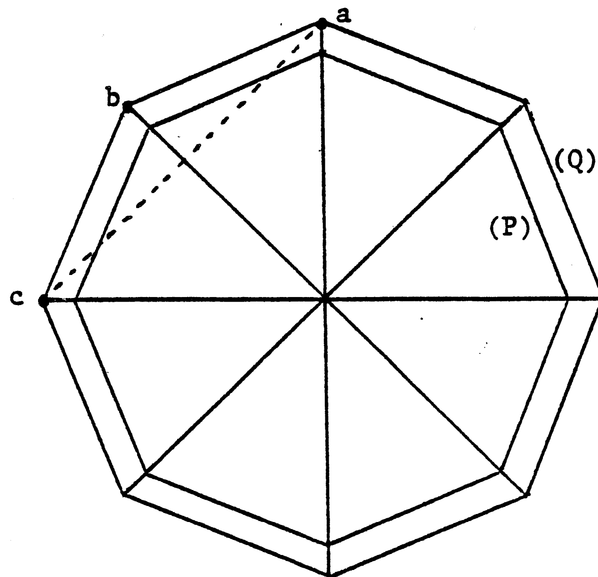


Figure 4.7: Intersecting polygonal lines.

cases where the algorithm can be applied to give a description of the region of intersection (although outputting the vertices of the entire region would require linear time) sufficient for its reconstruction. An interesting open problem would be to determine what information would be available for various possible inputs to the algorithm.

Finally, as with all geometric problems, there remains the issue of finding efficient methods of implementing the algorithm and applying it to problems of practical interest - See [Dobkin,78] for possible applications to a language for computer geometry.

4.3 Detecting Three-dimensional Intersections

4.3.1 Introduction

Although detecting intersections becomes substantially more difficult in three dimensions, the algorithms which we will describe are based on principles similar to those used in the previous sections. We still use Fibonacci searches to find extrema of bimodal functions and answer questions of the form: "Does object A lies entirely on one side of a given hyperplane?". Similarly, binary searches will be our basic means for reducing the size of the problem by a constant factor.

Since all these techniques assume some kind of random-access capabilities, we must give our three-dimensional objects a special representation to provide these features. From the observation that the surface of a convex polyhedron has the structure of a planar graph, it has been a standard method to represent convex polyhedra by a description of the planar graph along with the geometric location of the vertices [Muller and Preparata,77]. Unfortunately this representation does not meet all of our requirements and some preprocessing is needed. We basically wish to represent each polyhedron as a set of parallel convex polygons. These polygons, called preprocessing polygons, form a cross-section of the polyhedron for each vertex. We form this cross-section by intersecting the polyhedron with a plane parallel to the xy-plane and passing through the vertex - See Figure 4.8. This reduces a polyhedron P of p vertices to a set of p convex polygons $P_1,...,P_p$ and p-1 convex caps (we call a cap a convex polyhedron with all the vertices lying on two parallel faces). Since each cap can be tested for intersection in logarithmic time, and projections and intersections of those caps with a plane give convex objects, only O(logp) preprocessing polygons need be considered for all of our purposes, which yields the desired results. Before describing the preprocessing more precisely, we briefly outline the various algorithms which we will present.

1) IHP - (Intersection of a polyhedron P with a plane T)

The projections of P and T on a plane perpendicular to T and the preprocessing polygons form respectively a convex polygon and a line which intersect if and only if P and T intersect. We call upon IGL to test the intersection, which requires $O(\log^2 p)$ operations, since the access to any vertex of the projected polygon involves maximizing a linear combination of the x- or y-coordinates of a preprocessing polygon, that is, maximizing a bimodal function.

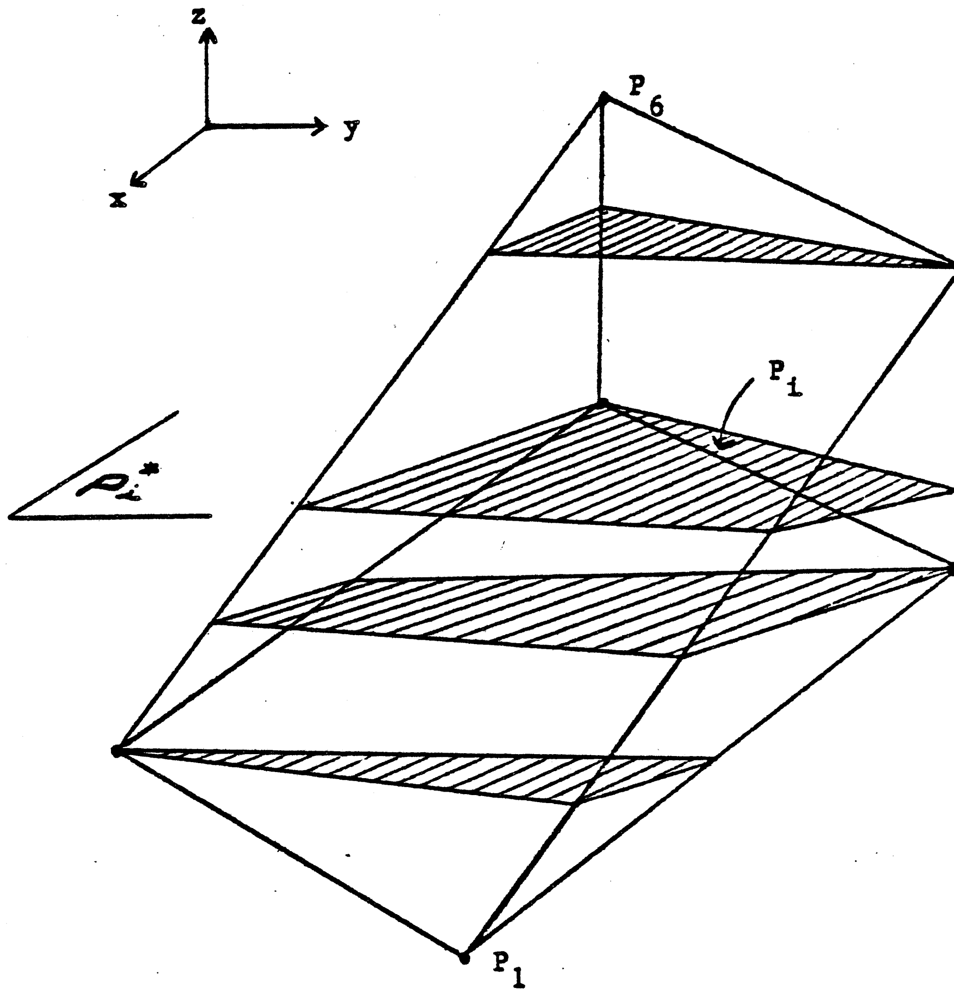


Figure 4.8: Preprocessing three-dimensional objects.

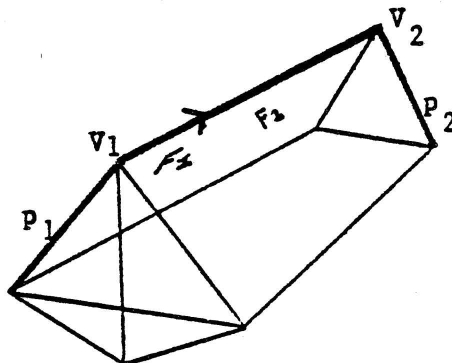


Figure 4.9: The DCEL representation of a polyhedron.

2) IHG - (Intersection of a polyhedron P with a polygon R)

If IHP fails to detect an intersection between P and the plane T supporting R, we are finished. Otherwise, T intersects a set of consecutive preprocessing polygons which we can compute implicitly in $O(\log^2 p)$ time by a binary search whose basic step involves intersecting a polygon with a line. Letting Q be the intersection of P and T, we first test the intersection of R with the subpolygon of Q formed by the preprocessing polygons determined earlier. If we fail, IGG will return a separating line adjacent to Q. We can show that this line is adjacent to two consecutive caps of P which must intersect R if P does. We can test each cap for intersection in turn, thus the whole algorithm runs in time $O(\log^2 N)$, if N is the total number of vertices involved in P and R.

3) IHH - (Intersection of two polyhedra P and Q)

By intersecting P and Q with a plane, a series of binary searches will reduce P successively to a cap, a "slice", and a pentahedron. Each step of the binary searches involves $O(\log^2 N)$ operations, thus leading to an $O(\log^3 N)$ time algorithm, with N the total number of vertices in P and Q.

4.3.2 Representation of Three-dimensional Objects

Before attempting any preprocessing on the polyhedra, we shall assume that they are all represented as planar graphs by doubly-connected-edge lists (DCEL). This requirement is legitimate since any standard planar graph representation can be transformed into a DCEL representation in linear time, and this is faster than the preprocessing we will further impose on the objects. We briefly recall the characteristics of this mode of representation.

Each edge of the polyhedron is represented by a 6-field node (V1,V2,F1,F2,P1,P2), where the edge is directed from vertex V1 to vertex V2. F1 and F2 denote the two faces adjacent to the edge, while P1 and P2 point respectively to the edge which follows V1V2 counterclockwise around V1 and V2 - See Figure 4.9. We note that this representation provides a counterclockwise sequence of the edges incident to a given vertex as well as a clockwise sequence of the edges enclosing a face, both in time proportional to the cardinality of the sequences.

As we mentioned earlier, some preprocessing is required in order to provide the polyhedra with some pseudo random-access capabilities. We will describe a simple version of this preprocessing, which performs in $O(p^2)$ time and space for a polyhedron P with p vertices $p_1, p_2, \ldots$

Let Oxyz be a system of orthogonal coordinates and let P_i^* denote the plane parallel to the xy-plane and passing through the vertex p_i of P. Let us first rename the vertices of P so that $p_1, p_2, \ldots$ corresponds to increasing z-coordinates. This only involves O(plog p) operations to sort the vertices. The intersection of P with the P_i^*'s forms p convex polygons $P_1, \ldots, P_p$, called preprocessing polygons. In general, P_1 and P_p will be reduced to a single point. Also, since several vertices of P may have the same z-coordinates, all the preprocessing polygons are not necessarily distinct, and we repair this flaw by only computing a single preprocessing polygon for each z-coordinate value. Although we may end up with fewer than p polygons, we still designate the set of preprocessing polygons, P_1 through P_p, for the sake of simplicity.

We can compute each P_i as follows: For each face F of P in turn, compute its intersection with P_i^*. It may be empty, or consist of a segment or a single point. In the last two cases, we determine the two edges e_1 and e_2 of F which are adjacent to the endpoints of the intersection. Let F_1 (resp. F_2) be the face of P adjacent to e_1 (resp. e_2) other than F. We can set up double links between F and F_1 as well as F and F_2, and repeating this process for all faces of F will produce a doubly-linked list of the vertices of P_i in order around the boundary - See Figure 4.10. The presence of several vertices of P on the same plane P_i^* may introduce edges of length 0, which can be later removed by a linear scan through the boundary of P_i. Let $x_{i,1}, x_{i,2}, \ldots$ denote a list of the vertices of P_i in clockwise order.

The portion of P comprised between P_i and P_j ($i<j$) is a convex polyhedron denoted $P_{i,j}$. All the algorithms which we will present rely heavily on the properties of the convex caps $P_{i,i+1}$. Since it is almost the case that each vertex of P_i is adjacent to a unique vertex of P_{i+1}, we nearly have a one-to-one correspondence between P_i and P_{i+1}, and these two polygons almost fully describe the cap $P_{i,i+1}$. Unfortunately, the vertices of P_i which are also vertices of P may be adjacent to several vertices of P_{i+1}, and to remedy this discrepancy, we add dummy vertices and dummy edges of length 0. More precisely, let $e_1, \ldots, e_k$ be the edges of P emanating from $x_{i,j}$ and intersecting P_{i+1}^*, as they appear in clockwise order on P_{i+1}. In general $k=1$. If, however, $k>1$, we conceptually duplicate $x_{i,j}$ into k vertices $y_1, \ldots, y_k$ all of which having the same geometric location as $x_{i,j}$. Each y_u, however, is made incident to exactly one edge e_u.

Iterating on this process for all vertices of P_i and all preprocessing polygons, we rename the vertices thus obtained for each P_i, $x_{i,1}^+, x_{i,2}^+, \ldots$ in clockwise order. Similarly, we consider all the edges of P emanating from $x_{i,j}$ and intersecting P_{i-1}^*, and duplicate $x_{i,j}$ accordingly. We thus define a refinement of P_i with respect to the cap $P_{i-1,i}$, renaming all the vertices of P_i, $x_{i,1}^-, x_{i,2}^-, \ldots$ - See Figure 4.11. Note that there is a one-to-one correspondence between $\{x_{i,1}^+, x_{i,2}^+, \ldots\}$ and $\{x_{i,1}^-, x_{i,2}^-, \ldots\}$. We actually require that the preprocessing make this mapping directly accessible (i.e., allowing access to $x_{i,j}^-$ from $x_{i,j}^+$ and vice-versa in constant time). The edges of the cap $P_{i,i+1}$ which run between P_i and P_{i+1} are called the lateral edges of the cap. Similarly, the faces between P_i and P_{i+1} are referred to as lateral

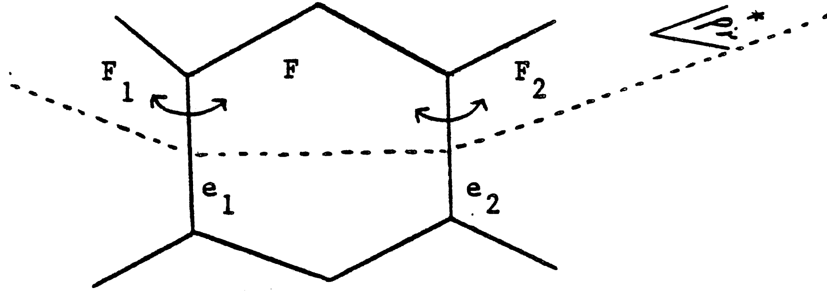


Figure 4.10: Computing the preprocessing polygons.

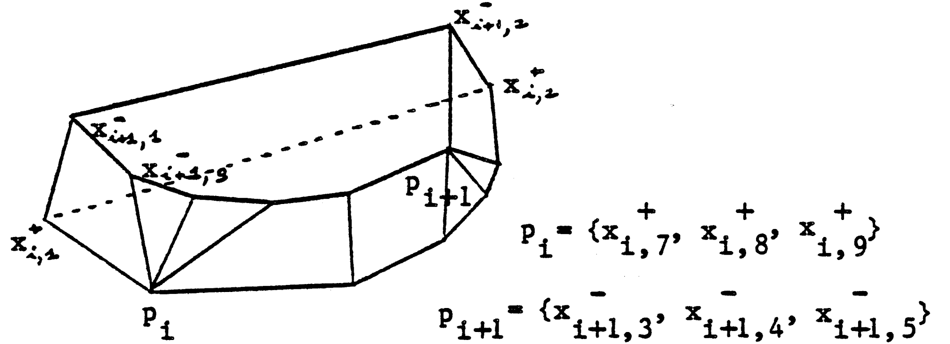


Figure 4.11: The one-to-one correspondence between the vertices of the cap Pi,i+1.

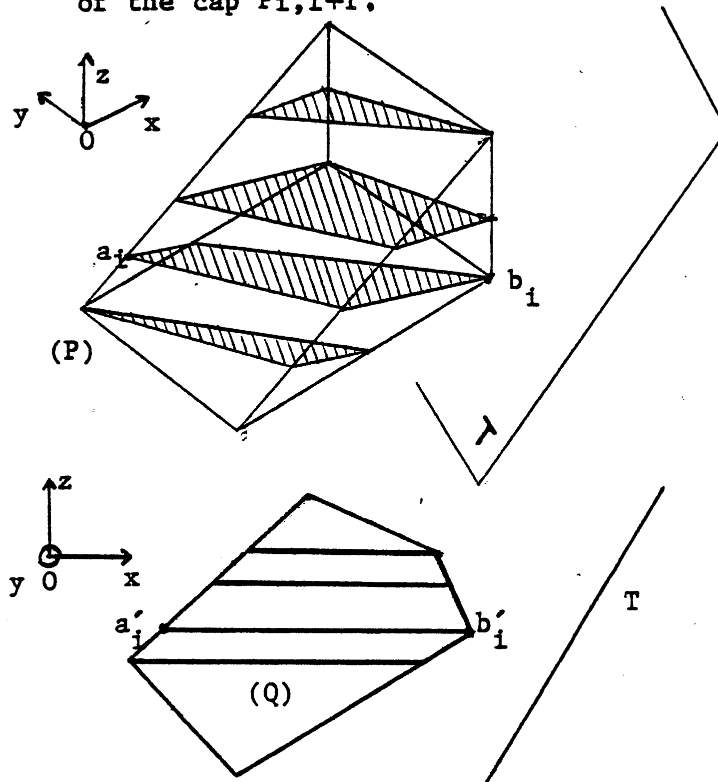


Figure 4.12: The IHP algorithm.

faces. It is clear that all this preprocessing requires $O(p^2)$ time and space, and we can now proceed with our first intersection algorithm.

4.3.3 Intersection of a Plane with a Polyhedron - (IHP)

Let P be a convex polyhedron with p vertices $p_1, p_2, ...$ and T denote the plane under consideration. Recall that the system of coordinates Oxyz is such that the preprocessing polygons of P are parallel to the xy-plane. Since a rotation around Oz involves only linear combinations of the x and y coordinates, we can always assume the y-axis to be parallel to T. The method relies on the observation that P and T intersect if and only if their projection on the xz-plane does.

Let a_i (resp. b_i) be the vertex of P_i with minimum (resp. maximum) x-coordinate. In general, a_i and b_i are unique, although there may be several of them if P_i have some edges parallel to the y-axis. In any case, the orthogonal projection of a_i (resp. b_i) on the xz-plane, denoted a'_i (resp. b'_i) is unique. We first show that the polygon $Q = a'_1 ... a'_p b'_p ... b'_1$ is convex - See Figure 4.12.

Lemma 29: The polygon Q is convex.

Proof: We show that none of the angles $(b'_k b'_{k+1}, b'_k b'_{k-1})$ is reflex. Let B be the intersection of the segment $b_{k-1} b_{k+1}$ with the plane P_k^*. Since P is convex, B lies on P_k, therefore its x-coordinate cannot be greater than the x-coordinate of b_k. The projection of B on the xz-plane being also the intersection of $b'_{k-1} b'_{k+1}$ with P_k^*, it follows that the angle $(b'_k b'_{k+1}, b'_k b'_{k-1})$ is no greater than 180 degrees. We have the same result with the vertices a'_k, and it is easy to conclude that Q is convex. □

We now prove our fundamental result.

Lemma 30: Let L be the intersection of T with the xz-plane. P and T intersect if and only if Q and L intersect.

Proof: If P and T intersect, we distinguish two cases:

1) T intersects with some P_i. Then the intersection of P_i and T is a line segment parallel to the y-axis, and its projection on the xz-plane is a point which lies on the segment $a'_i b'_i$. It follows that Q and L intersect.

2) If T does not intersect with any P_i, it lies strictly between two consecutive preprocessing polygons P_i and P_{i+1}, thus L intersects $a'_i a'_{i+1}$, that is, intersects Q.

Conversely, if L intersects Q, it must intersect one of its edges. Its endpoints are the projections on Oxz of two vertices u and v on the boundary of P, and it is clear that T must intersect the segment uv, that is, intersect P. Note that u and v are not necessarily vertices of P. □

From the previous results, we can easily derive the algorithm IHP.

Algorithm IHP

If P and T do not intersect, the algorithm returns NO, otherwise it returns (YES,A), where A is a point of the intersection.

Lemma 30 shows that we can test the intersection of P with T by applying the IGL algorithm to Q and L. We have an implicit description of Q, since we have random-access to any of its vertices in O(log p) time. This is due to the fact that the x-coordinates of the vertices of any preprocessing polygon form a bimodal function since the polygon is convex. Therefore, any a_i or b_i can be obtained in O(log p) time, from which a'_i and b'_i are computed in constant time. If Q and L do not intersect, IHP will return NO, else IGL provides, in O(log p) time, an edge of Q intersecting L, say $b'_i b'_{i+1}$. Since knowing b'_i and b'_{i+1} implies that b_i and b_{i+1} have been already computed, we can immediately determine the intersection A of T with the segment $b_i b_{i+1}$ and return (YES,A). Note that in this case, the segment $b_i b_{i+1}$ always intersects T. Since the algorithm IGL runs in logarithmic time and each basic step requires O(log p) operations, we can conclude:

Theorem 4: The intersection of a plane with a preprocessed convex polyhedron of p vertices can be detected in $O(\log^2 p)$ operations.

4.3.4 Intersection of a Polygon with a Polyhedron - (IHG)

We start with an analysis of the problem, concentrating only on the most difficult points.

Let P be a preprocessed convex polyhedron of p vertices and R a convex polygon of q vertices. Call Q the intersection of P with the plane T supporting R - See Figure 4.13. By first calling upon IHP, we can check whether Q is empty. Assume that it is not the case. It is equivalent to test the

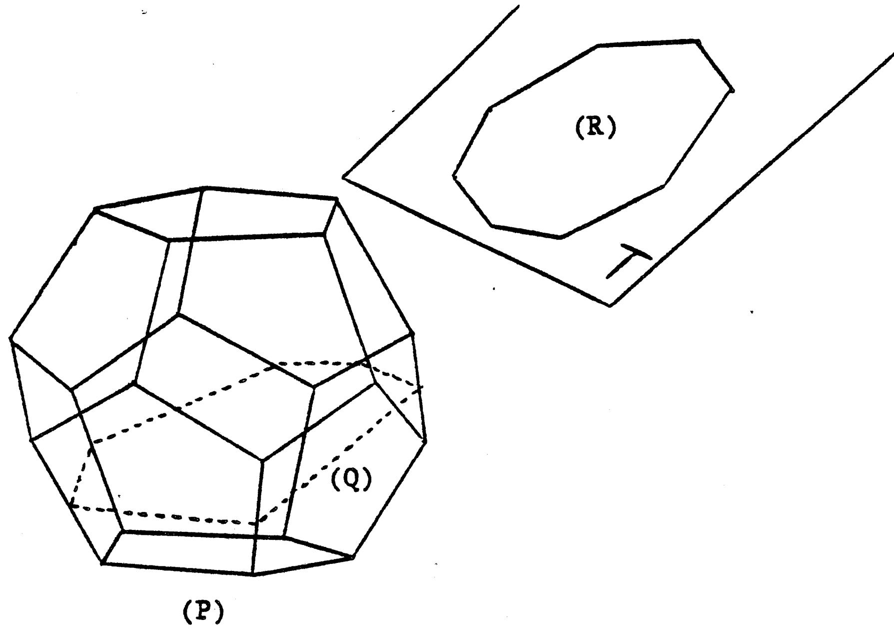


Figure 4.13: Intersection of a polyhedron and a polygon.

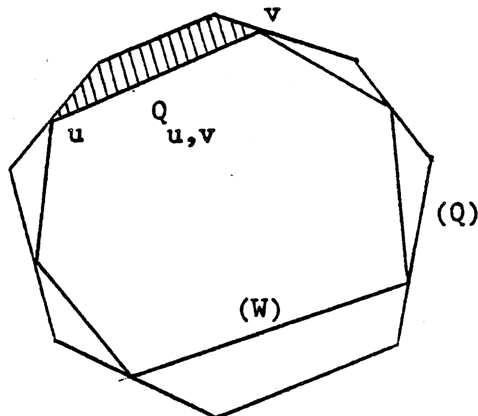


Figure 4.14: The polygons $Q, W, Q_{u,v}$.

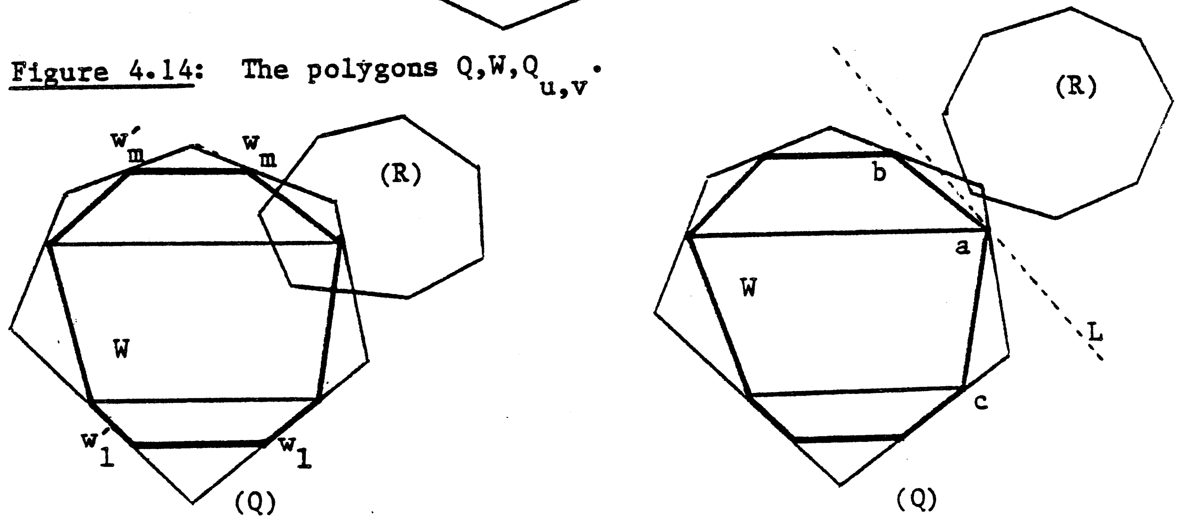


Figure 4.15: The two cases of intersection of Q and R.

intersection of P and R or Q and R. Although Q is not readily available, the preprocessing of P permits us to compute an implicit description of it. We first observe that from the convexity of P, T intersects a set (possibly empty) of consecutive P_i, say, $P_l,...,P_m$ ($l \leq m$). Let w_i, w'_i be the endpoints of the intersection of T and P_i, and W denote the polygon $w'_l...w'_m w_m...w_l$ - See Figure 4.15. Since W is a subpolygon of Q (i.e., W lies inside Q and all its vertices lie on the boundary of Q), it is easy to see that the convexity of Q implies that of W. If u,v are two consecutive vertices of W in clockwise order, we define $Q_{u,v}$ to be the convex polygon (outside of W) delimited by the edge uv and the boundary of Q - See Figure 4.14.

The following result shows how to reduce our main problem to two easier subproblems.

Lemma 31: If Q and W are not empty, P and R intersect if and only if either of the following conditions is satisfied:

1) W and R intersect.

2) Let L be a separating line of W and R passing through a vertex a of W, and let b,c be the vertices of W adjacent to a (b=c if W is reduced to a line segment). R intersects $Q_{b,a}$ or $Q_{a,c}$ - See Figure 4.15.

Proof: It suffices to observe that when R intersects Q but not W, the only parts of Q that L does not separate from R are $Q_{b,a}$ and $Q_{a,c}$. The remainder of the proof is straightforward. □

Case 1 being easy to handle, let us turn to the other case. We wish to compute an implicit description of $Q_{b,a}$ and $Q_{a,c}$ in order to test these polygons for intersection with R. We describe the method for $Q_{b,a}$, the other case being similar. Call $q_1,...,q_k$ the vertices of $Q_{b,a}$, that is, the vertices of Q lying between b and a. $q_1,...,q_k$ are the intersections of the plane T with consecutive lateral edges of some cap $P_{i,i+1}$, say, $e_1,...,e_k$. Since all the edges $e_1,...,e_k$ must pass through consecutive vertices of P_i, $x_1,...,x_k$, it suffices to determine x_1 and x_k to have an implicit description of $Q_{b,a}$. Note that in order to have a one-to-one correspondence between the x_i and the e_i, we must consider P_i with its vertices of the form $x_{i,1}^+, x_{i,2}^+, ...$ We distinguish between two cases:

1) ab is parallel to the preprocessing polygons (horizontal). ab is then the top or bottom edge of W, say, the top edge (wlog). Consider the three-dimensional strip S of $P_{i,i+1}$ formed by all its lateral faces. The intersection of T with this strip is a continuous broken line D running from P_i to P_i without intersecting P_{i+1} - See Figure 4.16-a. Therefore any path from the portion of the boundary of P_i between a and b to P_{i+1} must intersect D. It follows that $x_1,...,x_k$ are exactly all the vertices of P_i

between a and b. To decide whether it is between a and b or b and a in clockwise order around P_i, we simply observe that on one part of the boundary all the lateral edges intersect T, whereas none does on the other. Thus, testing any lateral edge for intersection with T will resolve the ambiguity in constant time.

2) ab is not a horizontal edge of W. $P_{i,i+1}$ now designates the cap lying between a and b. The intersection of T with the strip S consists of two broken lines, one of which runs from a to b - See Figure 4.16-b. Let $x_u x_{u+1}$ (resp. $y_v y_{v+1}$) be the edge of P_i (resp. P_{i+1}), given in clockwise order, which contains a (resp. b). Note that these edges will have already been computed when a and b are. Since we wish to access the edges of $P_{i,i+1}$ from the vertices of P_i, it is important to have a one-to-one correspondence between the vertices of P_i and P_{i+1}, therefore we will consider the polygon P_i (resp. P_{i+1}) with its vertices $x_{i,1}^+, x_{i,2}^+, \ldots$ (resp. $x_{i+1,1}^-, x_{i+1,2}^-, \ldots$). Let x_l be the vertex of P_i in correspondence with y_v, that is, the vertex lying with y_v on the same lateral edge of $P_{i,i+1}$. It is clear that if the lateral edge of $P_{i,i+1}$ passing through x_u intersects T, then $q_1, \ldots, q_k$ are exactly the intersections of T with the lateral edges emanating from $x_{l+1}, x_{l+2}, \ldots, x_u$ - See Figure 4.16-b. Otherwise, if the lateral edge emanating from x_{u+1} intersects T, the vertices $q_1, \ldots, q_k$ of Q are determined by the set of vertices $x_{u+1}, \ldots, x_l$ - See Figure 4.16-c. Finally, if neither of the above cases arises, no lateral edge intersects T between a and b, and $Q_{b,a}$ is reduced to the single edge ab, therefore no testing is necessary.

Putting all these results together and handling the remaining cases is straightforward, and we can now set out the algorithm IHG, whose correctness is established by these results.

<u>Algorithm IHG</u>

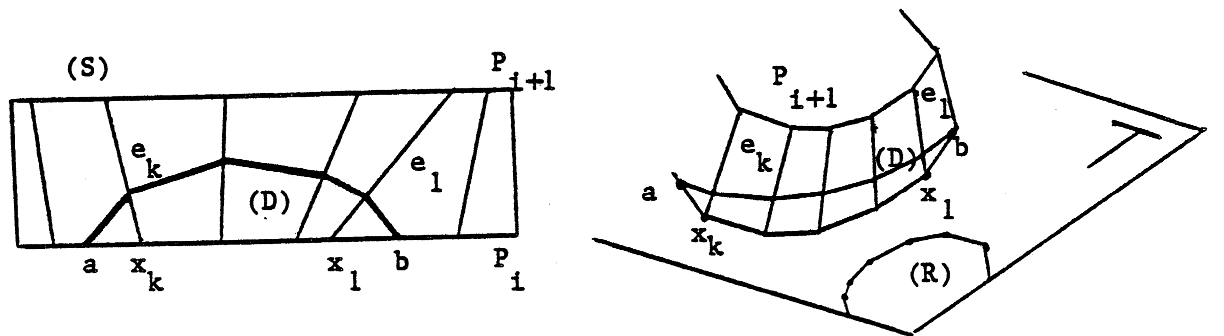
The algorithm takes a convex polygon R and a preprocessed convex polyhedron P as input, and returns NO if P and R do not intersect, or (YES,A) if they do, with A a point of the intersection.

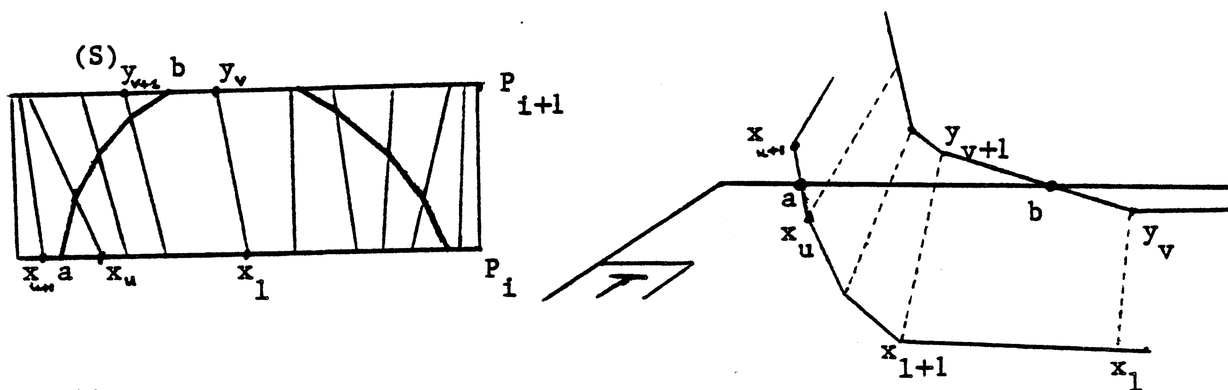
<u>STEP 1</u>

Test the intersection of P with the plane T supporting R by calling upon IHP. If P and T do not intersect, return NO, else the algorithm IHP will provide a point I of the intersection as well as the preprocessing plane P_i^* such that I lies in the cap $P_{i,i+1}$. If IGL indicates that T intersects neither P_i nor P_{i+1}, go to step 2, else go to step 3.

<u>STEP 2</u> "T lies strictly between P_i and P_{i+1}."

Q being the intersection of T and P, the vertices of Q are exactly the intersections of T with all the lateral edges of $P_{i,i+1}$. Therefore P_i gives an implicit description of Q, and it is possible to test the intersection of Q and R with the IGG algorithm, returning NO if it is empty, or (YES,A) if it is not, where A is a point of the intersection returned by IGG.





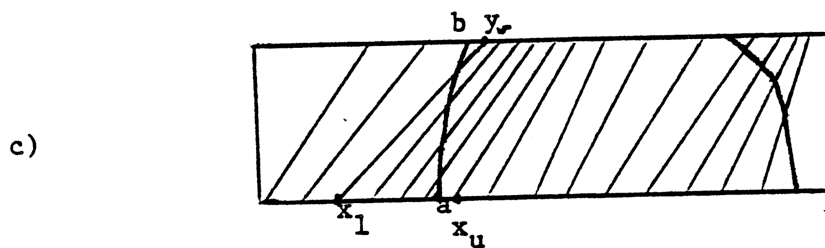


Figure 4.16: Computing $Q_{b,a}$.

STEP 3 "T intersects P_i or P_{i+1}."

Wlog, assume that T intersects P_i. Since T intersects a set of consecutive preprocessing polygons $P_l,...,P_m$, we can determine P_l and P_m through a binary search by testing the intersection of P_k and T with the IGL algorithm. This gives an implicit description of W, from which we can test the intersection of R and W with IGG. Note that to access a vertex of W, we must compute the intersection of T with some preprocessing polygon, using the IGL algorithm. If the intersecion of R and W is not empty, IGG will provide a common point A, and we can return (YES,A). Otherwise, IGG will return a separating line L of W and R passing through W, thus providing the vertices a,b,c.

STEP 4 "If R intersects P, it intersects $Q_{b,a}$ or $Q_{a,c}$."

Apply the procedure described above for $Q_{b,a}$ and $Q_{a,c}$ successively, and test these polygons for intersection with R (IGG), returning NO or a common point accordingly.

Before analyzing the running time of IHG, we wish to extend the algorithm slightly so that it returns a pair of parallel separating lines when P and R do not intersect, that is, a pair of separating lines for Q and R. When IHG returns NO in step 1, no such pair can be defined, but the plane T is itself a separating hyperplane and is a sufficient information for our purposes. In all of the other cases, a non-intersection of P and R is detected after testing both $Q_{b,a}$ and $Q_{a,c}$ for intersection has failed. Instead of testing these two polygons successively, we can simply use the implicit description of $Q_{b,a}$ and $Q_{a,c}$ to test the intersection of $Q_{b,c}$ with R ($Q_{b,c}$ is defined as the union of $Q_{b,a}$, $Q_{a,c}$, and the triangle abc). If no intersection is found, the algorithm IGG will return a pair of separating lines (D,D') for $Q_{b,c}$ and R. Let v be the vertex of $Q_{b,c}$ lying on the separating line D.

If v is distinct from b and c, (D,D') is also a pair of separating lines for Q and R since Q is convex, and fits our purposes - See Figure 4.17-a.

If v is b or c (say b, wlog), D may intersect Q outside of v, thus not separate Q and R. In that case, let d be the vertex of $Q_{b,c}$ adjacent to b and distinct from c. We can show that the line F passing through bd separates Q from R. Then computing a line F' adjacent to R and parallel to F, so that (F,F') forms a pair of separating lines, will take only O(log q) time, as described earlier. We now prove our claim.

Recall that the algorithm has already computed a line L adjacent to the vertex a of Q, and which separates W and R. Call L^+, D^+, F^+ the halfspaces delimited by L, D, F respectively, which do not contain the vertex c - See Figure 4.17-b. Since both L and D separate R from the triangle abc, R lies in the intersection of L^+ and D^+, denoted LD; moreover, L and D intersect in a point called

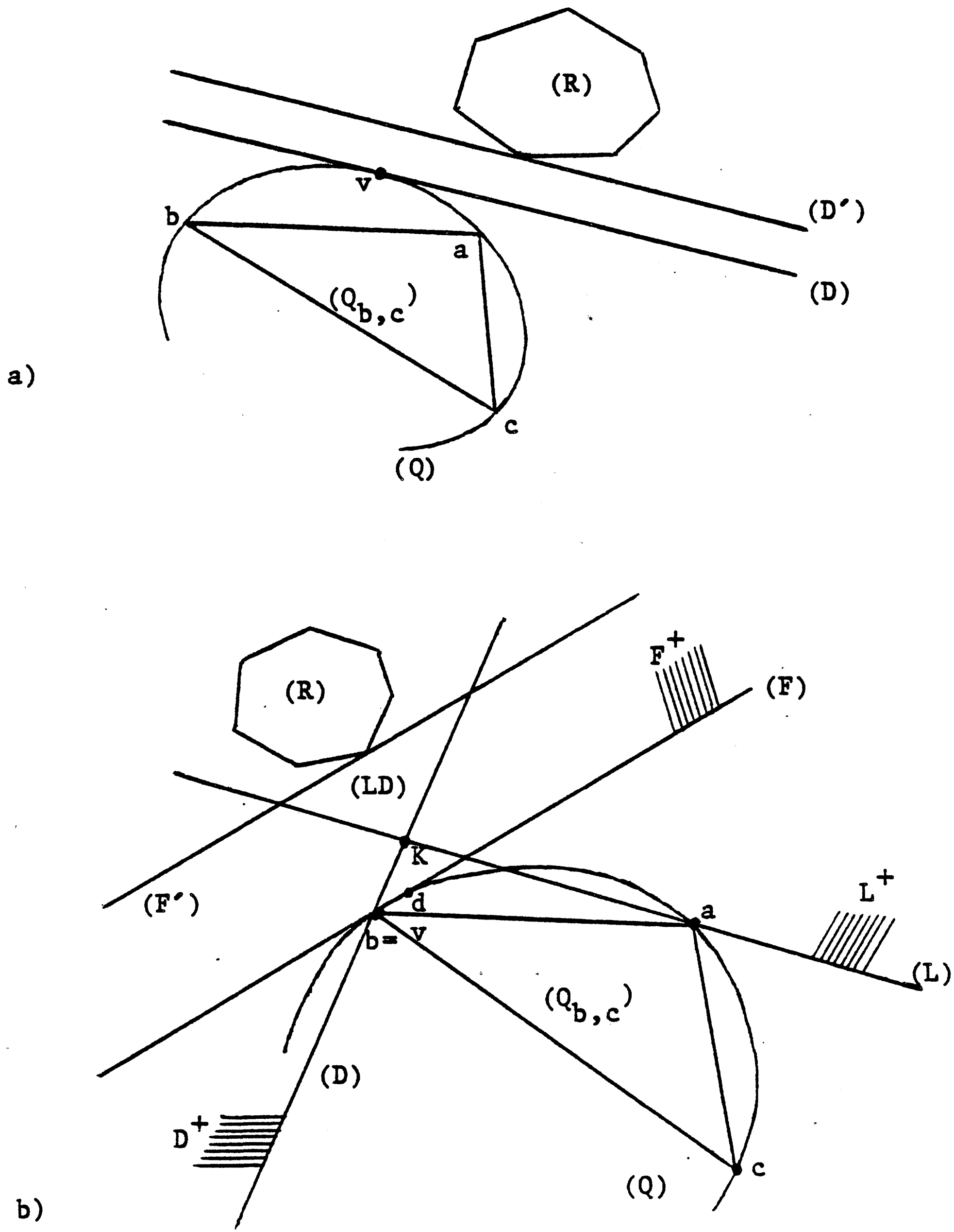


Figure 4.17: Computing a pair of separating lines for Q and R.

K. Finally, since the edge bd lies in the triangle bKa, F intersects L on the segment Ka, therefore LD lies entirely in the halfspace F^+, and so does R. This implies that F is a separating line of R and Q, which proves our claim.

Step 1 calls upon IHP and possibly IGL, thus requires $O(\log^2 p)$ operations. Step 2 is a simple application of IGG and takes $O(\log(p+q))$ time. Step 3 involves a binary search on the preprocessing polygons with a call on IGL at each step, which amounts to $O(\log^2 p)$ time. Testing the intersection of W and R takes $O((\log p)\log(p+q))$ time since each vertex of W is obtained by intersecting T with some P_k (IGL), which takes $O(\log p)$ time. Finally, step 4 performs a constant-time case analysis, then calls on IGG, which requires $O(\log(p+q))$ operations. We can finally state our main result.

Theorem 5: The intersection of a preprocessed convex polyhedron of p vertices with a convex polygon of q vertices can be detected in $O((\log p)\log(p+q))$ operations (or more simply $O(\log^2 N)$ if N is the total number of vertices in both objects).

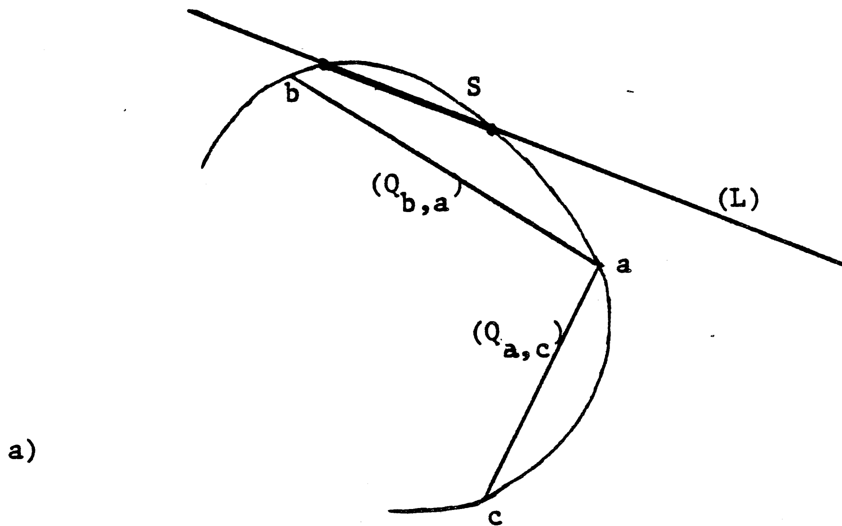
4.3.5 Intersection of a Line with a Polyhedron - (IHL)

We now consider the problem of detecting an intersection between an infinite line (or a line segment) L and a convex polyhedron of p vertices preprocessed as usual. We can contemplate a solution which is a straightforward application of the method described in the previous section.

We first test the intersection of P with any plane T supporting the line L, using IHP. If we fail to detect an intersection, we obviously return NO. Otherwise, we define the polygon Q as usual (i.e., the intersection of P and T), and we compute an implicit description of its subpolygon W formed by the preprocessing polygons of P. We next test the intersection of W and L (IGL), and in the event of a failure compute a separating line adjacent to W and derive the the polygons $Q_{b,a}$ and $Q_{a,c}$. Finally, we test these polygons for intersection with L, calling upon IGL.

Note that in the case of an intersection, we can compute the segment S of L which lies in P, in $O(\log p)$ time. There are essentially two cases to consider:

1) If an intersection is detected while intersecting $Q_{b,a}$ (resp. $Q_{a,c}$) with L, S is exactly the intersection of $Q_{b,a}$ (resp. $Q_{a,c}$) with L, and we can compute it in $O(\log p)$ time (IGL) - See Figure 4.18-a.



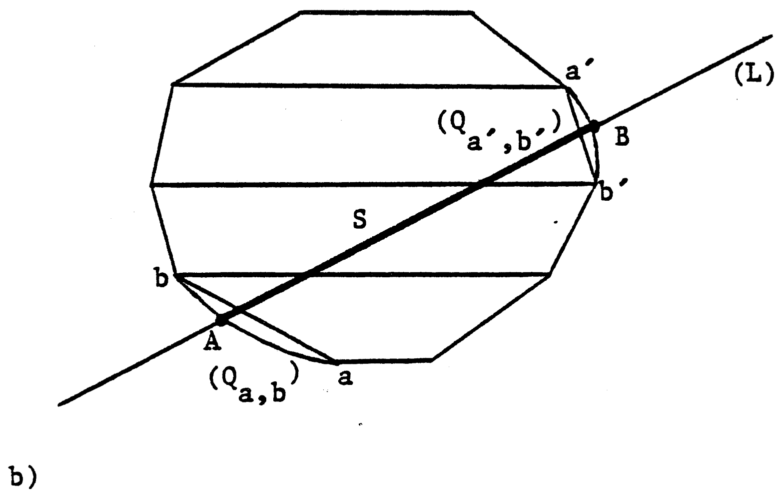


Figure 4.18: The algorithm IHL.

2) If W and L intersect, IGL will provide the two edges of W which intersect L, say, ab and a'b'. Then it is clear that if A (resp. B) is the point on the boundary of $Q_{a,b}$ (resp. $Q_{a'b'}$) which intersects L and does not lie on ab (resp. a'b'), S is the segment AB. To obtain this segment, we need to compute implicit descriptions of $Q_{a,b}$ and $Q_{a'b'}$ and intersect L with these two polygons (see Algorithm IHG for details of the procedure). Finally, IGL will provide A and B in O(log p) time.

The total running time of the algorithm is clearly $O(\log^2 p)$, and we conclude,

Theorem 6: We can compute (explicitly) the intersection of a preprocessed polyhedron of p vertices with an infinite line or a line segment in $O(\log^2 p)$ operations.

4.3.6 Intersection of Two Polyhedra - (IHH)

We now turn to the problem of detecting the intersection of two convex polyhedra P and Q of respectively p and q vertices. We assume that both polyhedra have been preprocessed, yet we do not require that the preprocessing planes of P should be parallel to those of Q - See Figure 4.20-a. Indeed, such a requirement would not be tolerable in dynamic situations where the objects are in motion, as it is often the case in real-time environments for example. The algorithm IHH proceeds by a series of binary searches, all very similar in nature, and reduces P to a cap, a "slice", and a pentahedron successively. For the clarity of exposition, we start our analysis of the problem with some preliminary results related to lines and planes of support. We redefine a line of support of P more precisely as a line having exactly one point or one segment (not necessarily an edge) in common with the boundary of P. Similarly, a plane of support of P is defined as a plane with exactly one edge or one face in common with the boundary of P.

For later purposes, we need to extend the preprocessing of P slightly. We require the existence, for each vertex of P, of an array listing the edges incident to it in, say, clockwise order. This additional constraint requires linear time and space to achieve from a DCEL representation of P, thus does not alter the order of magnitude of the complexity of the preprocessing. We can now state our preliminary result:

Lemma 32: If L is a line of support of P and one edge of P which intersects L is known, it is possible to determine a plane of support of P containing L in O(log p) operations.

Proof: Call v the intersection of L with that edge e of P known to intersect L. We distinguish two cases:

1) v is not a vertex of P (check if v is an endpoint of e); then the plane containing both e and L is a plane of support of P - See Figure 4.19-a.

2) v is a vertex of P; then the plane passing through e and L may unfortunately intersect the interior of P, and further analysis is needed - See Figure 4.19-b. Let $e_1,...,e_k$ be a list of the edges of P adjacent to v, in clockwise order. Recall that the preprocessing of P ensures random-access to these edges. Let U denote a plane parallel to L which intersects the endpoint of e_1 other than v. Call $w_1,...,w_k$ the intersections of the plane U with the infinite lines passing through $e_1,...,e_k$ respectively (note that $e_1 = vw_1$) - See Figure 4.19-c. Since $w_1,...,w_k$ form a convex polygon, the distance from w_i to the plane T passing through L and perpendicular to U gives a bimodal sequence if it is counted positive on one side of T and negative on the other. Thus, its extrema can be found in O(log p) time. Let w_l be one of them; since the plane passing through L and vw_l is a plane of support of the polyhedron formed by $v,w_1,...,w_k$, it is also a plane of support of P, which completes the proof. □

We now turn to the crux of the algorithm IHH. Let us assume that P and Q intersect but neither contains the other. Let T be a plane intersecting P and Q but not their intersection. Call R (resp. S) the intersection of T and P (resp. Q), and let (L_P,L_Q) be a pair of parallel separating lines for R and S respectively. If T_P (resp. T_Q) is a plane of support of P (resp. Q) passing through L_P (resp. L_Q), observing the relative position of T_P and T_Q will indicate on which side of T the intersection of P and Q lies. Indeed, since P and Q intersect but R and S do not, the intersection of P and Q lies entirely in one of the halfspaces delimited by T - See Figure 4.20. To determine which, we first observe that the intersection of P and Q must lie in the intersection H of the halfspace delimited by T_P which contains P with the halfspace delimited by T_Q containing Q. Since L_P and L_Q are parallel, H lies totally on one side of T and the intersection L of T_P and T_Q (which must exist since H is non-empty) may be computed in constant time, and indicates which side of T contains the intersection of P and Q. Note that L is an infinite line parallel to T - See Figure 4.20-b,c. The portion of P which does not lie on the same side of T as L can be rejected since it cannot intersect with Q. This gives us a means to reduce the size of the problem, and conducting this process in a binary search fashion will guarantee efficiency. We now proceed to describe the algorithm.

1) Let P_l be the middle preprocessing polygon of P ($l = \lceil p/2 \rceil$). The first step consists of reducing P to $P_{1,l}$ or $P_{l,p}$. To do so, we test the intersection of the preprocessing plane P_l^* with Q, using the IHP

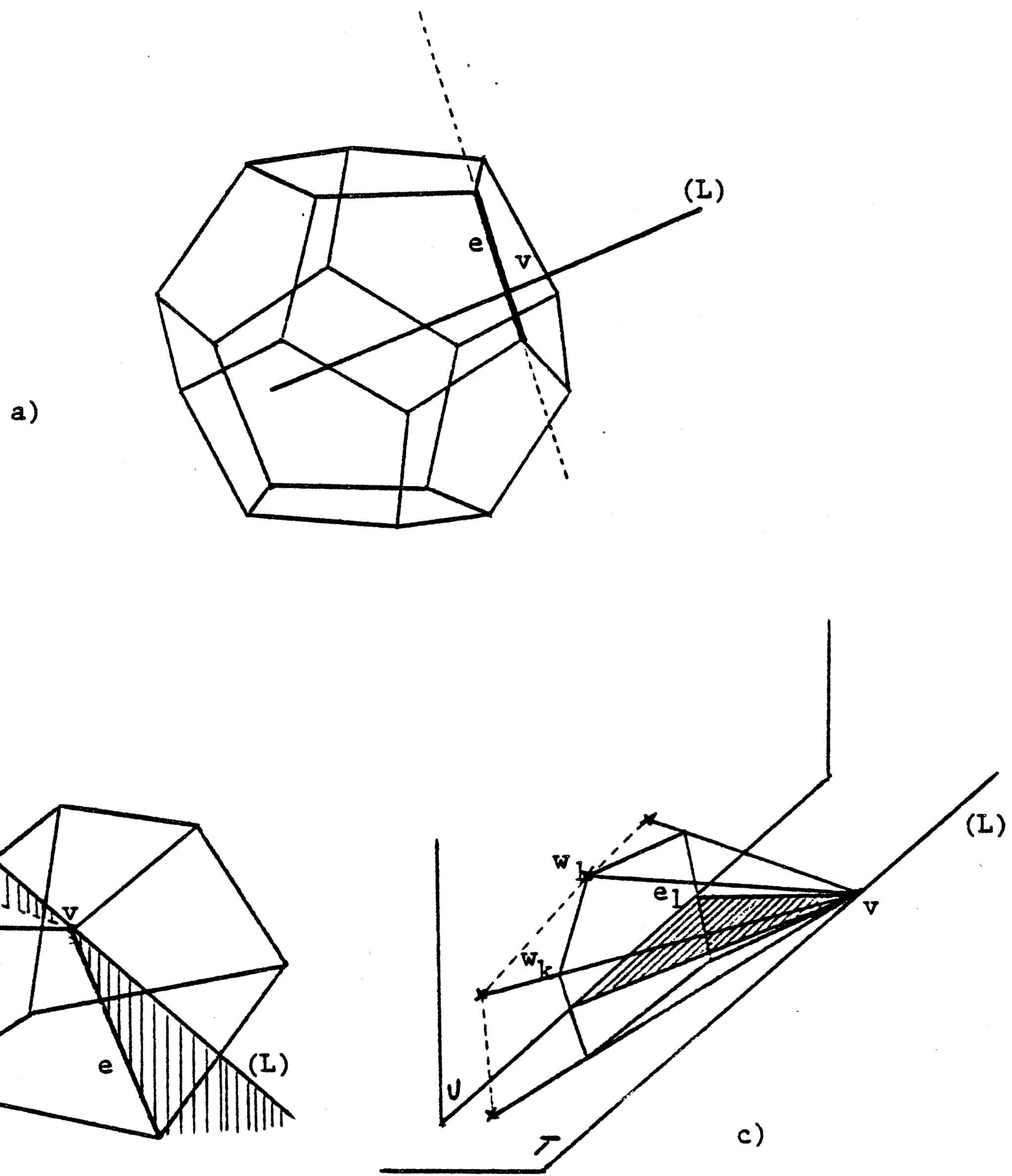
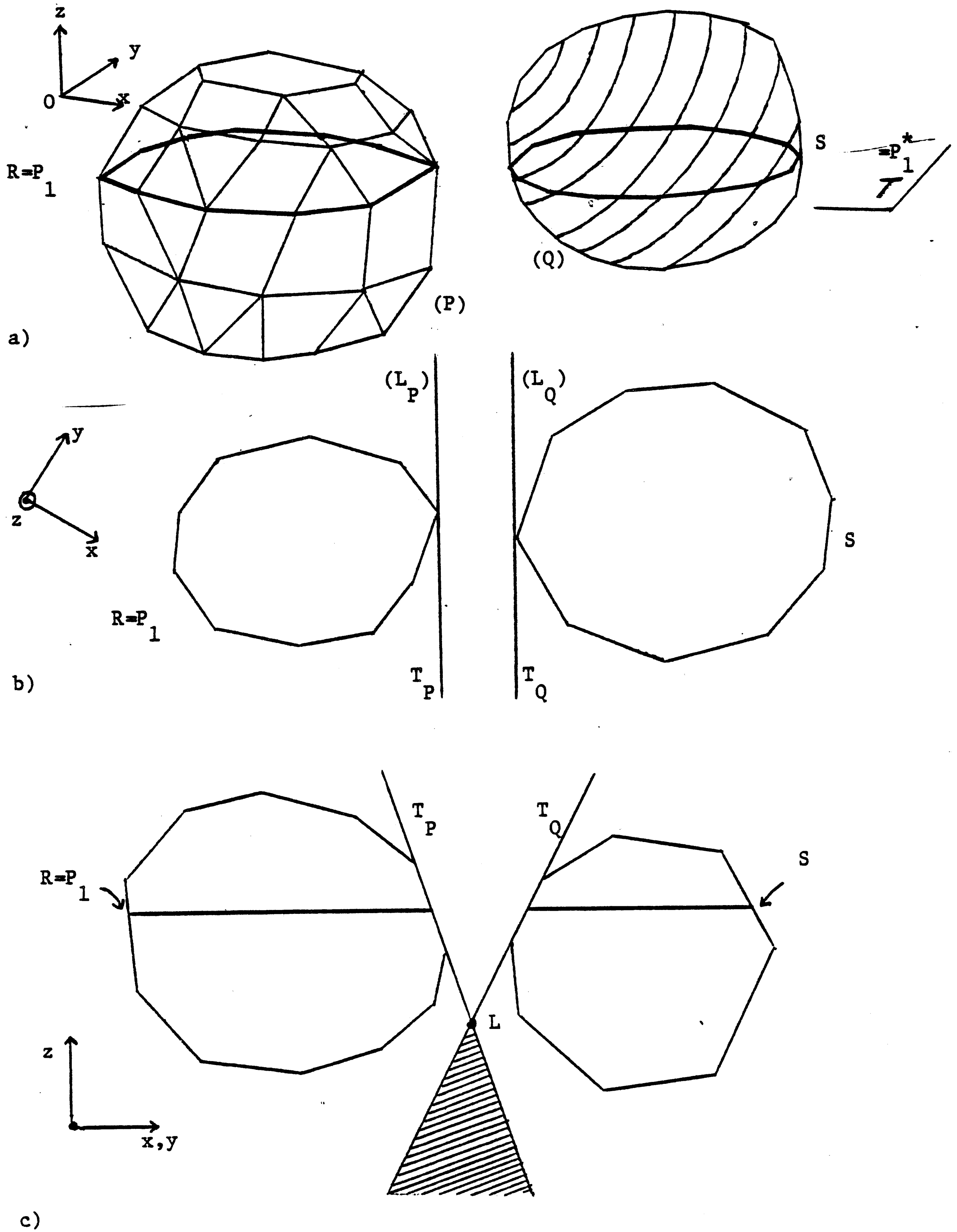


Figure 4.19: Computing a plane of support of P.



<u>Figure 4.20</u>: Reducing the size of P in IHH.

algorithm. If it fails to detect an intersection, Q lies entirely on one side of P_l^* which can be determined in constant time. We then iterate on this process with $P_{1,l}$ or $P_{l,p}$, whichever lies on the same side of P_l^* as Q. If P_l^* and Q intersect, we call upon IHG to test the intersection of Q with the polygon P_l, returning (YES,A) if IHG finds a point A of the intersection, or providing a pair of separating lines (L_P,L_Q) - See Figure 4.20-b. Since in this last case, IHG will also indicate edges of P (resp. Q) which intersect L_P (resp. L_Q), we can apply the result of Lemma 32 and compute a plane of support of P passing through L_P, which we denote T_P. A similar computation will give a plane of support of Q passing through L_Q, T_Q - See Figure 4.20-c. Finally our discussion above shows how locating the intersection of L_P and L_Q with respect to P_l^* permits us to substitute $P_{1,l}$ or $P_{l,p}$ for P accordingly. Of course, if T_P and T_Q do not intersect (i.e., are parallel), neither do P and Q.

Iterating on this process will either produce a point of the intersection or will reduce P to a convex cap $P_{i,i+1}$. Note that we may have $i+1=1$ or $i=p$, in which case the algorithm can return NO since P and Q do not then intersect.

2) It remains now to test the intersection of Q and $P_{i,i+1}$. Let $x_1,...,x_k$ be the vertices of P_i in clockwise order. We choose a lateral edge e of $P_{i,i+1}$, say, an edge passing through x_1, and consider the plane T_j containing both x_j and the edge e. For any u,v, $1<u<v\leq k$, we define $T_{u,v}$ as the portion of $P_{i,i+1}$ comprised between T_u and T_v (i.e., the portion which contains the edge $x_u x_{u+1}$). We have seen in the description of the IHG algorithm how to compute an implicit description of the polygon S_j formed by the intersection of $P_{i,i+1}$ and T_j - See Figure 4.21. Recall that this involves computing the points a and b as well as the lateral edges of $P_{i,i+1}$ which intersect T_j. Having an implicit description of S_j, we can apply the procedure described earlier, using successively IHP with arguments T_j, Q and IHG with arguments S_j,Q. We will either return a point of the intersection of S_j and Q, in which case we are done, or produce a pair of planes of support for P and Q respectively, containing two parallel lines separating S_j and Q.

Once again, locating the intersection of these two planes will permit us to substitute $T_{2,j}$ or $T_{j,k}$ for P accordingly. We can perform a binary search by setting j to $\lceil k/2 \rceil$ initially and iterating on this process. If the algorithm does not terminate before, it will reduce $P_{i,i+1}$ to the convex polyhedron $T_{j,j+1}$ for some j - See Figure 4.22-a.

3) $T_{j,j+1}$ has one face lying on P_i (the triangle $x_1x_jx_{j+1}$) and a parallel face on $P_{i,i+1}$, F. Unfortunately, Figure 4.22-a illustrates only the simplest case since F is not necessarily a triangle. We can, however, remedy this discrepancy easily. Let y_1 be the endpoint of the edge e which lies on P_{i+1} ($e=x_1y_1$) and let $y_1,...,y_n$ denote the vertices of P_{i+1} in clockwise order. F is a convex polygon $y_1,y,y_l,...,y_m,y',y_1$ - See Figure 4.22-b,c. We can determine y and y' in O(log p) time by intersecting P_{i+1} with T_j and T_{j+1}, using the IGL algorithm (which actually must have been done already). If y

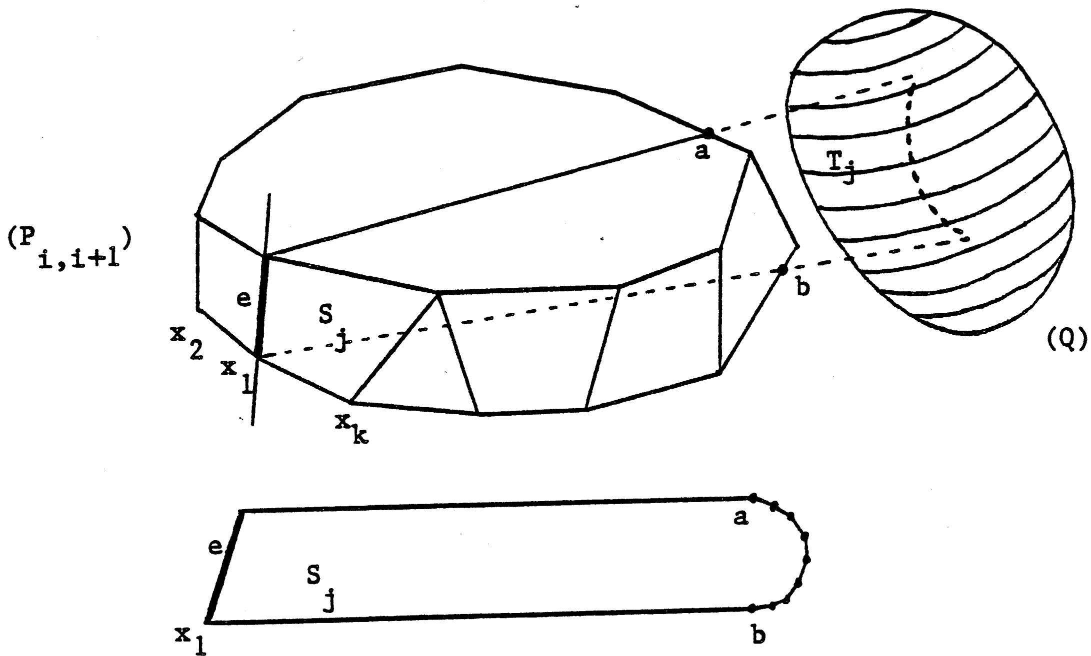


Figure 4.21: Reducing the cap $P_{i,i+1}$ in IHH.

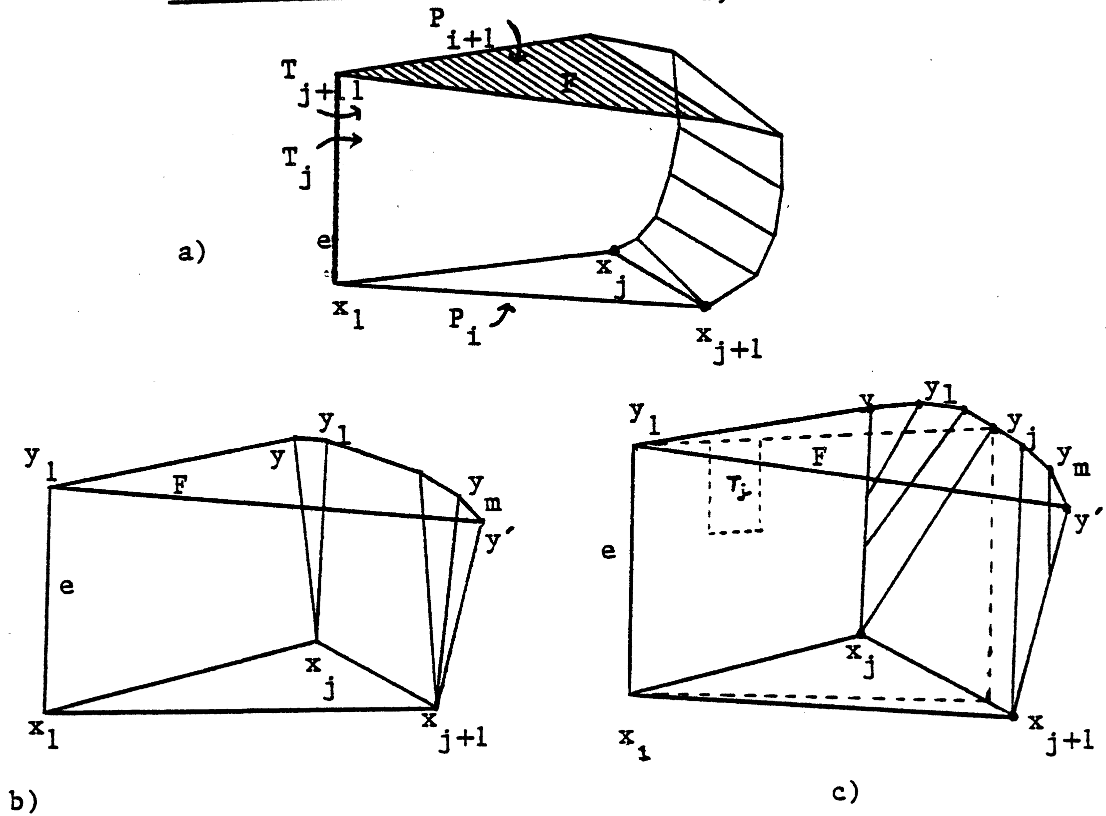


Figure 4.22: The cap $P_{i,i+1}$ after reduction.

and y' do not lie on the same edge of P_{i+1}, we carry on the previous binary search on the planes $T_1',...,T_m'$, where T_j' is now the plane containing e and y_j. If the algorithm does not return, it will reduce P to a convex polyhedron A with two parallel faces on P_i and P_{i+1}, x_1ab and y_1 a'b' respectively, both of which are triangles - See Figure 4.23. Let F_j (resp. F_{j+1}) be the face of A lying on T_j (resp. T_{j+1}). In addition to ab and a'b', A may contain other edges $f_1,...,f_t$ intersecting both F_j and F_{j+1}. These edges lie on consecutive lateral edges of $P_{i,i+1}$, say, $e_1,...,e_t$ in clockwise order. Our next task is to compute an implicit description of this set of edges, that is, to determine e_1 and e_t.

The following fact will permit us to compute e_1 and e_t in constant time. We can always assume that a,b (resp. a',b') occur in clockwise order in a traversal of the boundary of P_i (resp. P_{i+1}). Let g be a lateral edge of $P_{i,i+1}$ intersecting both F_j and F_{j+1}, with g_1 (resp. g_2) the endpoint of g lying on P_i (resp. P_{i+1}). Note that by construction of A, any edge intersecting F_j intersects F_{j+1} as well, and vice-versa. We can observe that if g_1 occurs between x_1 and a (resp. b and x_1) in clockwise order, g_2 must lie between b' and y_1 (resp. y_1 and a') in clockwise order. Wlog, suppose that g_1 occurs between x_1 and a. Let $x_{i,l}^+$ be the vertex of P_i such that a lies on the edge $x_{i,l}^+x_{i,l+1}^+$. Since lateral edges can only intersect at their endpoints, the lateral edge of $P_{i,i+1}$ adjacent to $x_{i,l}^+$ (which is uniquely defined by the preprocessing) also intersects both F_j and F_{j+1}. This shows that lateral edges of $P_{i,i+1}$ intersect F_j and F_{j+1} if and only if the lateral edge adjacent to $x_{i,l}^+$ or $x_{i+1,l}^+$ intersects T_j. This gives us a convenient way to determine e_1 and e_k in constant time with the technique already used in the IHG algorithm. Namely, let $x_{i+1,u}^-x_{i+1,u+1}^-$ be the edge of P_{i+1} which intersects F_j and let $x_{i,m}^+$ be the vertex of P_i in one-to-one correspondence with $x_{i+1,u+1}^-$. It is then clear that e_1 is the edge $x_{i,m}^+x_{i+1,u+1}^-$ and e_t the lateral edge adjacent to $x_{i,l}^+$. All the lateral edges between e_1 and e_t also intersect F_j and F_{j+1}, that is, the edges adjacent to $x_{i,m}^+,x_{i,m+1}^+,...,x_{i,l}^+$. Recall that the one-to-one correspondence between $\{x_{i,1}^+,x_{i,2}^+,...\}$ and $\{x_{i+1,1}^-,x_{i+1,2}^-,...\}$ established in the preprocessing allows random-access to the lateral edges of $P_{i,i+1}$.

4) Having an implicit description of $e_1,...,e_t$, we can define U_j as the plane containing x_1 and e_j and apply the procedure of 2) on this set of planes - See Figure 4.23. We will either return a point of the intersection of P, Q, and U_j, or produce a pair of planes of support from which we can decide which side of U_j contains the intersection of P and Q, if it exists. Note that the intersection of A and U_j is simply a triangle which we can compute in constant time. If the algorithm does not return, it will eventually reduce P to a pentahedron K comprised between T_j, T_{j+1}, two consecutive triangles U_l, U_{l+1} and a lateral face of $P_{i,i+1}$ - See Figure 4.24.

5) Finally, we have to test the intersection of K and Q. To do so, we can test each face of K successively, using the IHG algorithm. If we fail to detect an intersection, we determine whether Q lies entirely inside or outside of K by testing the inclusion of any point of Q in K, which can be done in constant time.

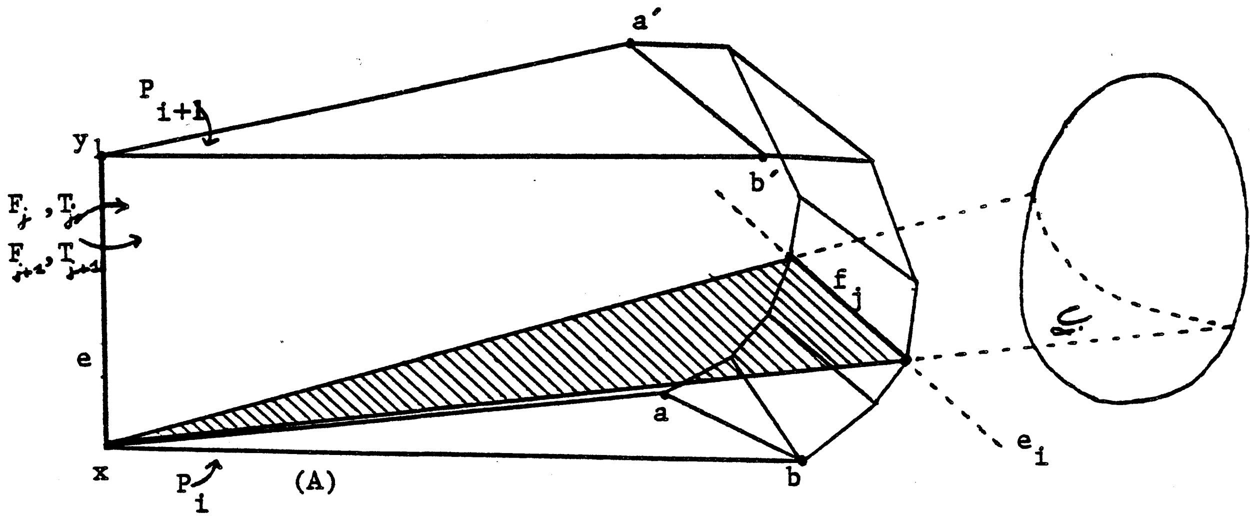


Figure 4.23: Reducing the slice A in IHH.

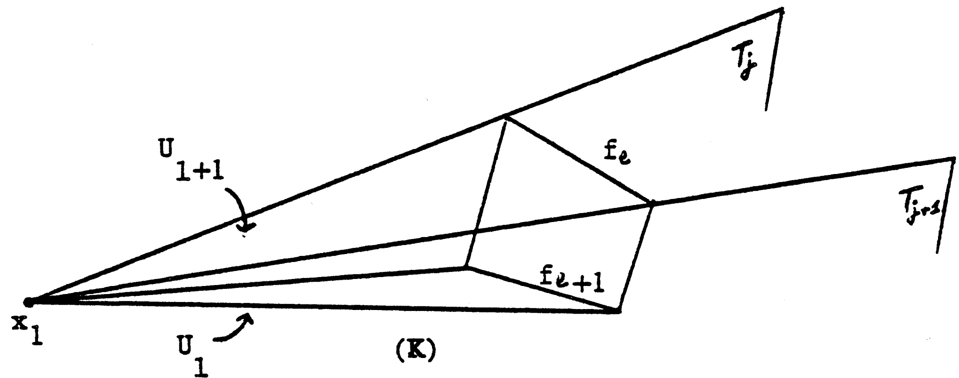


Figure 4.24: The pentahedron K.

We now give a more formal outline of the algorithm IHH, which will also serve as a summary.

Algorithm IHH

The input consists of two preprocessed convex polyhedra P and Q, and the output is NO if P and Q do not intersect or (YES,A) if they do, where A is a point of the intersection.

STEP 1

```
l=1 , m=p

while  l<m-1
    begin

          i= ⌊(l+m)/2⌋

          if P_i* does not intersect Q [IHP]

             then
                  if  q_1 lies above P_i*

                      then  l=i

                      else  m=i

             else
                  if  P_i intersects Q [IHG]

                      then  return (YES,A) , with
                                A the point returned
                                by IHG.

                      else

          "IHG provides a pair of separating lines from
           which T_P and T_Q are computed."

          if  T_P and T_Q do not intersect

             then  return (NO)

          if  T_P and T_Q intersect above P_i*

             then  l=i

             else  m=i

    end

 i=l
```

STEP 2

"P is reduced to a convex cap $P_{i,i+1}$."

Let e be a lateral edge of $P_{i,i+1}$ and T_j be the plane containing e and the vertex x_j of P_i.

Apply the above procedure with respect to the planes T_j.

Finally set j to 1.

STEP 3

"P is reduced to a convex polyhedron $T_{j,j+1}$."

If the face of $T_{j,j+1}$ lying on P_{i+1}^* is not a triangle, apply step 2 with respect to the planes $T_1', \ldots, T_m'$.

STEP 4

"P is reduced to a polyhedron A bordered by two triangles, subpolygons of P_i and P_{i+1} respectively."

Apply the procedure of Step 1 with respect to the planes U_j passing through x_1 and e_j, where x_1 is a vertex of P_i and e_j is the j-th lateral edge of A.

STEP 5

```
"P is reduced to a pentahedron K"
Check if Q lies entirely inside K by testing if q1
does. In the affirmative, return (YES,q1).
Otherwise, apply the IHG algorithm to test if Q
intersects with any of the 5 faces of K. If it is the
case, return (YES,A), where A is a point of the
intersection, else return (NO).
```

We can now state our main result.

Theorem 7: The intersection of two preprocessed convex polyhedra of p and q vertices respectively can be detected in $O((\log p)(\log q)\log(p+q))$ operations (or more simply $O(\log^3 N)$, if N is the total number of vertices in P and Q).

Proof: At this stage, we simply have to evaluate the execution time of the algorithm. We review its various phases and set out their run-times.

1) involves $O(\log p)$ applications of IHP ($\log^2 q$), IHG ($(\log q)\log(p+q)$), and the algorithm of Lemma 32 ($\log pq$).

2) We can obtain an implicit description of S_j in constant time, once the intersection of T_j with P_i and P_{i+1} has been computed ($\log p$). The remainder of this step is similar to the previous one.

3) Same complexity as 2), since computing an implicit description of $y_1,\ldots,y_m$ takes constant time.

4) Same as 2).

5) is essentially a repeated application of IHG to Q and a triangle or a quadrilateral ($\log^2 q$).

The proof is now complete. □

4.4 Conclusions

We have described a complete set of algorithms for detecting intersections in two and three dimensions. In all cases, we have avoided issues of efficiency beyond the asymptotic level. Although the algorithm for computing planar intersections is aymptotically optimal [Shamos,78], we believe that a more sophisticated treatment of bimodal functions will improve its running time. Also, a more refined case analysis might permit us to reduce not only one of the polygons by half but both of them.

In three dimensions, aside from speeding up the preprocessing [Dobkin and Munro,80], we believe that the algorithm IHH would benefit from a more symmetric treatment of the two polyhedra (along the lines of the algorithm IGG, for example). There also remains the question of proving lower bounds, since none of the 3D-algorithms has been shown to be optimal.

In all cases, we believe that improvements can be best discovered by implementing the algorithms and observing their behavior on real problems. There is also the possibility of using the methods presented here as the basis of fast probabilistic algorithms for solving these problems ([Rabin,76], [Bentley and Shamos,77]).

Chapter 5
EPILOGUE

By investigating two classes of problems related to convexity (convex decompositions and detection of intersections), this thesis reveals a fundamental truth in computational geometry: Convexity provides a tremendous factor of efficiency, and even non-convex objects can enjoy its benefits.

Convexity captures a notion of regularity and order in geometry. Thus tracing its presence in non-convex structures will not only bring efficiency but also simpicity. For this reason we believe that the ideas exposed in this volume should be exploited in a systematic way in a great number of geometric problems. The computation of "regions of safety" in Section 2.4 is a good illustration of the possible application of superranges to convex hull problems. Similarly, we observed in Section 3.2 how the mere introduction of convex chains could improve the performance of the sweeping algorithm for computing maxima.

Other topics related to the present work seem promising for future research, and we will mention some of them.

Extending the intersection problem to higher dimensions is very tempting for its possible applications to linear programming. However, we regard prospects for simple generalizations with certain skepticism. Although the method used in three dimensions consists essentially of reducing the problem to two dimensions, the relations between the components of a polyhedron (i.e., faces, edges, vertices) become so intricate in higher dimensions that even extensions to four dimensions seem very challenging. Here again we are faced with the gap between 3 and 4 dimensions that we already encountered in the decomposition problem.

One type of question of great theoretical interest concerns the establishment of lower bounds. In computational geometry, perhaps more than in any other area of computer science, there is a shortage of non-trivial lower bounds. It seems that the traditional approaches to lower bound problems are inadequate and new techniques are undoubtedly required. Almost none of the problems considered

in this thesis has a known non-trivial lower bound. The intersection of convex polygons is one exception, since the problem has a logarithmic lower bound, which is actually achieved by our algorithm.

In most cases, the practical interest of lower bounds is questionable. This is especially true when there is a huge gap between the known lower and upper bounds, as is the case in the decomposition problem. We believe, however, that a study of lower bounds in general will serve a valuable purpose by shedding light on the structure of geometry. For example, one of the only non-trivial lower bounds to be known for the problems considered in this thesis concerns the decomposition in three dimensions, and was established by introducing original, purely geometric techniques (i.e., volume arguments to bound the output size).

In that same range of interest, one might wonder whether the "difficult" instances of the decomposition problem, for which we proposed heuristics (i.e., decompositions in three dimensions and as sums and differences) are actually NP-complete. Only recently has a respectable number of geometric problems been shown NP-complete [Garey, Graham and Johnson,76]. Reductions to NP-complete geometric problems are thus possible. However, the presence of metric constraints tends to make geometric problems often very specific and hard to relate to companion problems.

The design of efficient approximation algorithms is often the only practical alternative for dealing with difficult problems. Traditionally, these algorithms have been meant to produce solutions that always lie within a constant factor of optimal solutions. This requirement may be sometimes too strong to be achieved, especially when exact solutions do not have a fixed order of magnitude. For example, the cardinality of optimal convex decompositions in three dimensions varies between $O(1)$ and $\Omega(N^2)$, and to guarantee optimal solutions within a constant factor may be as difficult as computing exact solutions.

One may relax this requirement, however, and only seek to guarantee a worst-case order of magnitude. This applies to optimization problems, where the goal is now to produce solutions which lie within a constant multiplicative factor from an exact solution on the worst-case input. This is the nature of the result in Section 3.2 concerning the heuristic for the three-dimensional decomposition problem.

In practical situations, the prevailing criterion of a good algorithm is often its expected run-time rather than its worst-case performance. This motivates the study of fast probabilistic algorithms for producing exact or approximate solutions. These algorithms may be inefficient on certain inputs, but are guaranteed a very good average-case performance. Our intersection algorithms illustrate this concept. If the goal is to compute intersections explicitly, these algorithms will be extremely efficient

in the cases of non-intersection. Otherwise they will really be wasted computations, since they can produce at most a few points of the intersection. Currently, in situations where all intersections in a large set of objects have to be computed, standard techniques such as boxing are typically used, and fare much better than the naive method consisting of intersecting each possible pair in linear time [Newman and Sproull,79]. However, if only a few intersections exist, the exhaustive procedure carried out with our detection algorithms may be more efficient. Actually, combinations of these three techniques may well be the best alternative.

Of course, all these issues can be decided only by assuming statistical distributions of geometric objects and performing average-case analyses. Unfortunately, it is likely that the objects involved in practice do not fit into any nice, simple mathematical distribution. Particular shapes are always favored in practical situations, and the choice of relevant probabilistic models is bound to require great familiarity with the domains of application.

BIBLIOGRAPHY

[Aho, Hopcroft and Ullman,74] Aho,A., Hopcroft,J., and Ullman,J., The Design and Analysis of Computer Algorithms, Addison-Wesley Publishing Company, Reading, Massachussetts, 1974.

[Bellman,57] Bellman,R., Dynamic Programming, Princeton University Press, Princeton, New Jersey, 1957.

[Bentley, Haken and Hon,80] Bentley,J., Haken,D., and Hon,R., Fast geometric algorithms for VLSI tasks, COMPCON Spring 80, IEEE Computer Society, 1980.

[Bentley and Ottmann,78] Bentley,J. and Ottmann,T., Algorithms for reporting and counting geometric intersections, Carnegie-Mellon University, August, 1978.

[Bentley and Shamos,77] Bentley,J. and Shamos,M., Divide and Conquer for linear expected time, Information Processing Letters, 1977.

[Chazelle,79] Chazelle,B., Decomposing polyhedra into convex parts, 1979, To appear.

[Chazelle and Dobkin,79] Chazelle,B. and Dobkin,D., Decomposing a polygon into its convex parts, Proceedings of the 11th ACM SIGACT Symposium, Atlanta, Georgia, May, 1979, pp.38-48.

[Chazelle and Dobkin,80] Chazelle,B. and Dobkin,D., Detection is easier than computation, Proceedings of the 12th ACM SIGACT Symposium, Los Angeles, California, May, 1980, pp.146-153.

[Cohen,69] Cohen,P., Decision procedure for real and p-adic fields, Communications on Pure and Applied Mathematics, Vol.22, 1969, p.131.

[Coxeter,61] Coxeter,H., Introduction to Geometry, John Wiley and Sons, Inc., New York-London, 1961.

[Coxeter,73] Coxeter,H., Regular Polytopes, Third edition, Dover Publications, Inc., New York, 1973.

[Dantzig,63] Dantzig,G., Linear Programming and Extensions, Princeton University Press, Princeton, New Jersey, 1963.

[Dobkin,78] Dobkin,D., Keyword structures in a language for computer geometry, Bell Laboratories Technical Memorandum, TM 78-1271-9, June 29, 1978.

[Dobkin and Lipton,76] Dobkin,D. and Lipton,R., Multidimensional searching problems, SIAM Journal on Computing, Vol.5, No.2, June, 1976, pp.181-186.

[Dobkin and Munro,80] Dobkin,D. and Munro,J., Efficient uses of the past, 1980, To appear.

[Dobkin and Snyder,79] Dobkin,D. and Snyder,L., On a general method of maximizing and minimizing among certain geometric problems, 20th Annual IEEE FOCS Conference, San Juan, Puerto Rico, October, 1979, pp.9-17.

[Dobkin and Tomlin,78] Dobkin,D. and Tomlin,D., Cartographic modelling techniques in environmental planning: An efficient system design, Submitted for publication.

[Feng and Pavlidis,75] Feng,H. and Pavlidis,T., Decomposition of polygons into simpler components: Feature generation for syntactic pattern recognition, IEEE Transactions on Computers, Vol.C-24, No.6, June, 1975, pp.636-650.

[Forrest,79] Forrest,A., Private communication to D.Dobkin, March 29, 1979.

[Garey, Graham and Johnson,76] Garey,M., Graham,R., and Johnson,D., Some NP-complete geometric problems, Proceedings of the 8th Annual ACM SIGACT Symposium, Hershey, Pennsylvania, May, 1976, pp.10-22.

[Garey, Johnson, Preparata, and Tarjan,78] Garey,M., Johnson,D., Preparata,F., and Tarjan,R., Triangulating a simple polygon, Information Processing Letters, Vol.7, No.4, June, 1978, pp.175-180.

[Gilbert and Pollak,68] Gilbert,E. and Pollak,H., Steiner minimal trees, SIAM Journal of Applied Mathematics, Vol.16, 1968, pp.1-29.

[Grunbaum,67] Grunbaum,B., Convex Polytopes, Wiley-Interscience, 1967.

[Harary,71] Harary,F., Graph Theory, Addison-Wesley Publishing Company, Reading, Massachussetts, 1971.

[Kiefer,53] Kiefer,J., Sequential minimax search for a maximum, Proceedings of the American Mathematical Society, Vol.4, 1953, pp.502-506.

[Knuth,68] Knuth,D., Fundamental Algorithms, Addison-Wesley Publishing Company, Reading, Massachussetts, 1968.

[Lakatos,76] Lakatos,I., Proofs and Refutations: The Logic of Mathematical Discovery, Cambridge University Press, 1976.

[Lloyd,77] Lloyd,E., On triangulations of a set of points in the plane, 18th Annual IEEE FOCS Conference, Providence, Rhode Island, October, 1977, pp.228-240.

[Lyusternik,63] Lyusternik,L., Convex Figures and Polyhedra, Dover Publications, Inc., New York, 1963.

[Massey,67] Massey,W., Algebraic Topology: An Introduction, Springer-Verlag, New York, 1967.

[Mead and Conway,80] Mead,C. and Conway,L., Introduction to VLSI Systems, Addison-Wesley Publishing Company, Reading, Massachussetts, 1980.

[Melzak,61] Melzak,Z., On the problem of Steiner, Canadian Mathematical Bulletin, Vol.4, 1961, pp.143-148.

[Monk,75] Monk,L., Elementary-recursive Decision Procedures, PhD thesis, University of California, Berkeley, 1975.

[Muller and Preparata,77] Muller,D. and Preparata,F., Finding the intersection of two convex polyhedra, Technical Report, University of Illinois, Urbana, Illinois, October, 1977.

[Munkres,75] Munkres,J., Topology: A First Course, Prentice-Hall, Inc., 1975.

[Newman and Sproull,79] Newman,W. and Sproull,R., Principles of Interactive Computer Graphics, Second edition, McGraw Hill, New York, 1979.

[Nievergelt and Preparata,80] Nievergelt,J. and Preparata,F., Plane-sweep algorithms for intersecting geometric figures, Technical Report, University of Illinois, Urbana, Illinois, 1980.

[Pavlidis,68] Pavlidis,T., Analysis of set patterns, Pattern recognition, Vol.1, 1968, pp.165-178.

[Pavlidis,79] Pavlidis,T., Private communication, 1979.

[Protter and Morrey,70] Protter,R. and Morrey,C., College Calculus with Analytic Geometry, Second edition, Addison-Wesley Publishing Company, Reading, Massachussetts, 1970.

[Rabin,76] Rabin,M., Probabilistic algorithms, in Algorithms and Complexity: New Directions and Recent Results, Traub,J., Ed. Academic Press, 1976.

[Schachter,78] Schachter,B., Decomposition of polygons into convex sets, IEEE Transactions on Computers, Vol.C-27, No.11, November, 1978, pp.1078-1082.

[Seidenberg,54] Seidenberg,A., A new decision method for elementary algebra, Annals of Mathematics, Vol.60, 1954, pp.365-374.

[Shamos,75] Shamos,M., Geometric Complexity, Proceedings of the 7th Annual ACM SIGACT Symposium, Albuquerque, New Mexico, May, 1975, pp.224-233.

[Shamos,78] Shamos,M., Computational Geometry, PhD thesis, Yale University, May, 1978.

[Shamos and Hoey,76] Shamos,M. and Hoey,D., Geometric intersection problems, 17th Annual IEEE FOCS Conference, Houston, Texas, October, 1976, pp.208-215.

[Tarski,51] Tarski,A., A decision method for elementary algebra and geometry (Second edition, Revised), University of California Press, 1951.

[Tomlin,78] Tomlin,D., Private communication to D.Dobkin, 1978.

[Van der Waerden,50] Van der Waerden,B., Modern Algebra, Vols.I,II, Frederick Ungar Publishing Company, New York, 1949, 1950.

[Volecker,77] Volecker,H., Private communication to D.Dobkin, November 14, 1977.