SHORTEST PATHS AVOIDING POLYHEDRAL OBSTACLES IN 3-DIMENSIONAL EUCLIDEAN SPACE

by
Varol Akman

A Thesis Submitted to the Graduate
Faculty of Rensselaer Polytechnic Institute
in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY
Major Subject: Computer and Systems Engineering

Approved by the
Examining Committee:

Wm. Randolph Franklin, Thesis Adviser

Frank DiCesare, Member

Mukkai S. Krishnamoorthy, Member

James M. Tien, Member

Rensselaer Polytechnic Institute
Troy, New York
June 1985
(For graduation August 1985)

CONTENTS

LIST OF FIGURES

FOREWORD

> For me there is only the traveling on paths that have heart, on any path that may have heart. There I travel, and the only worthwhile challenge is to traverse its full length. And there I travel, looking, looking, breathlessly.
>
> -don Juan

> A tree is best measured when it is down.
>
> -Robert Wilson and David Byrne

My research owes considerably to my adviser, Randolph Franklin. He introduced me to the exciting field of computational geometry and taught me that geometry is fun. I thank him for his unfailing support during all phases of this work, especially when things seemed to be evolving painfully slow.

My former committee member Edwin Rogers kindly let me use the CSLAB facilities. My other committee members, Frank DiCesare, Mukkai Krishnamoorthy, and James Tien also deserve many thanks for their encouragement and advice.

The material reported herein has been supported by the National Science Foundation under grants ECS 80-21504 and ECS 83-51942. Special thanks go to Herbert Freeman and Michael Wozny for employing me in the IPL and the CICG during some periods of my study. My study in the United States has also been supported in part by a Fulbright award and by the Middle East Technical University, Ankara.

This thesis is dedicated to Vinciane Lacroix.

ABSTRACT

Algebraic and geometric algorithms are presented to solve the general and some specific instances of the following problem which is known as FINDPATH in artificial intelligence:

> "Given a set of polyhedra and two external points (source and goal), calculate the shortest path between these points under the Euclidean metric, constrained to avoid intersections with the given polyhedra."

The shortest path is a polygonal path connecting the source and the goal, possibly bending at some edges of the given polyhedra while subtending equal entry and exit angles. The general problem can then be solved exactly using symbolic algebra and is based upon finding the real roots of a system of nonlinear equations (stating the angular equality conditions) by elimination. If approximate solutions are acceptable this can also be done using numerical methods such as continuation, or using optimization methods to minimize a sum of square roots (expressing the length of the shortest path). In addition, two special cases are solved where there exists a single convex polyhedron, and the source and the goal are on its boundary or exterior. These solutions make use of planar developments of polyhedra and polyhedral visibility.

Two Voronoi-based locus methods are also introduced to solve FINDPATH. These methods use preprocessing to speed up subsequent searches for a shortest path. The first method partitions the boundary of a convex polyhedron given only the source on it so that, for a later goal on the boundary, the shortest path is found efficiently. It makes use of standard point location algorithms for a straight-edge planar subdivision once the partitioning is done. The second method, currently more of a descriptive nature than algorithmic, is based on W. R. Franklin's "Partitioning the Plane to Calculate Minimal Paths to any Goal around Obstructions" [Tech. Rep., ECSE Dept., Rensselaer Polytechnic Inst., Troy, NY, Nov. 1982], and for a given source, partitions the free space around polyhedra into regions (bounded by high-order surfaces) so that the shortest paths for all goals in a given region follow the same sequence of edges of the given polyhedra. This method makes use of a recent spatial point location algorithm for arbitrary algebraic varieties.

Finally, an overview of a workbench (called SP) which implements some of the above and thus allows experimentation with shortest paths is given.

1. INTRODUCTION

This introductory chapter has five sections. In section 1.1 we give some prerequisite material and summarize our notation. In section 1.2 we define the problem that is central to this thesis. Section 1.3 describes the motivation for solving this problem. Section 1.4 overviews the relevant work by other researchers. Section 1.5 outlines the remaining chapters of the thesis.

1.1 Prerequisites and Notation

For classical material on polyhedra, graphs, and algebra, we follow Grunbaum[41], Harary[43], and van der Waerden[113], respectively. Standard references on computational geometry are still lacking but Sedgewick[99], Shamos[101], Mehlorn[73], and Lee and Preparata[66] are useful. For concrete complexity, we follow Garey and Johnson[39]. Explanations of the terms and the concepts left undefined in this thesis can be found in these references and will not be repeated here.

With few exceptions, we shall be concerned with R^3, the 3-dimensional real Euclidean space, or 3-space in short. Points of R^3, as well as the corresponding vectors, will be denoted by lower case letters. $d(a,b)$ will denote the Euclidean distance between the points a and b whereas $d(a)$ will denote the length of vector a. $\mathrm{card}A$ will denote the cardinality of set A. The scalar product of vectors $a,b \in R^3$ is denoted by $<a,b>$ while $a \times b$ denotes their vector product. The affine hull of set A is denoted by $\mathrm{aff}A$ whereas $\mathrm{conv}A$ denotes the convex hull. The interior, boundary, and closure of set A are respectively $\mathrm{int}A$, $\mathrm{bd}A$, and $\mathrm{cl}A$. For a polyhedral set A, the totality of the extreme points of A is the vertices of A and is denoted by $\mathrm{vert}A$. 1-faces of A are called the edges of A and are denoted by $\mathrm{edge}A$. Maximal proper faces are the facets of A whose union is equal to $\mathrm{bd}A$. (Departing from Grunbaum's terminology, we shall use the more widespread term "face" instead of "facet.") To prevent potential confusions, we shall use the terms vertex, edge, and path *exclusively* for polygons and polyhedra. For graphs, we shall use the less common terms node, arc, and walk, respectively.

To estimate bounds, the standard notation (reviewed below to overcome the slight nuances in the literature) will be used. Let $f(n)$ and $g(n)$ be functions of integer $n \geq 0$. Then $f(n)=O(g(n))$ implies that there exists c and n_0 such that $|f(n)|<cg(n)$ for $n>n_0$. $f(n)=o(g(n))$ implies that $\lim_{n\to\infty} f(n)/g(n)=0$. $f(n)=\Theta(g(n))$ implies that there exists c_1 and c_2 $(c_1c_2>0)$, and n_0 such that $c_1g(n)<f(n)<c_2g(n)$ for $n>n_0$. $f(n)=\Omega(g(n))$ implies that $g(n)=O(f(n))$.

1.2 Problem Statement

Let $P=\{P_1, \cdots, P_n\}$ be a prescribed set of disjoint polyhedra and $s, g \in R^3$ be distinct points which are *not* internal to any P_i. The class of rectifiable curves which have endpoints s and g and which do not intersect any $\text{int}P_i$ will be denoted by $C(s, g; P)$. For C in this class, $l(C)$ will denote the length of C under the Euclidean (L_2) metric. An interesting problem in computational geometry asks for the shortest one among these curves and will be the subject of this thesis:

> FINDPATH
> INSTANCE: Polyhedra $P=\{P_1, \cdots, P_n\}$ such that $P_i \cap P_j = \Phi$, $i \neq j$, and $s, g \in R^3$ such that $s \neq g$ and s, g do not belong to $\text{int}P_i$, $1 \leq i \leq n$.
> QUESTION: Which $C \in C(s, g; P)$ has the shortest length?

The above version of FINDPATH is an optimization problem and in the sequel, when we refer to FINDPATH without any qualification we shall mean FINDPATH (optimization). The following defines the decision version of FINDPATH:

> FINDPATH (decision)
> INSTANCE: Same as in FINDPATH, and a positive integer k.
> QUESTION: Is there a $C \in C(s, g; P)$ such that $l(C) \leq k$?

We shall return to the interplay between these two versions of FINDPATH after the following theorem which must be intuitively clear:

THEOREM 1.1 There exists a $C' \in C(s, g; P)$ such that for all $C \in C(s, g; P)$, $l(C') \leq l(C)$. Moreover, every such C' is a polygonal path with its possible bend points belonging to some edges of some members of P.

Proof. Our proof will follow a similar theorem of Chein and Steinberg[14] in 2-space. Let C be any curve in $C(s, g; P)$. If any part of C lies outside $H = \text{conv}P'$ where $P' = \{s, g\} \cup (\bigcup_1^n \text{vert}P_i)$, let x be the first point at which C leaves H, and y_1 be the point at which C first returns. Let F be the face of $\text{bd}H$ which contains y_1 and let y be the last point where C enters H on F. Then $C = C_1C_2C_3C_4$, where C_1, C_2, C_3, and C_4 are respectively curves from s to x, from x to y_1, from y_1 to y, and from y to g, with C_1 lying inside H, C_2 lying outside H, and C_4 starting at $\text{int}H$ (figure 1.1). Let C_2' be the shorter polygonal path on $\text{bd}H$ between x and y_1, C_3' be the line segment y_1y, and $C' = C_1C_2'C_3'C_4$. (Here y_1y is meaningful since F is convex.) Then $l(C_3') \leq l(C_3)$ and by the convexity of H, $l(C_2') \leq l(C_2)$; so $l(C') \leq l(C)$. Furthermore C' does not intersect the members of P and thus

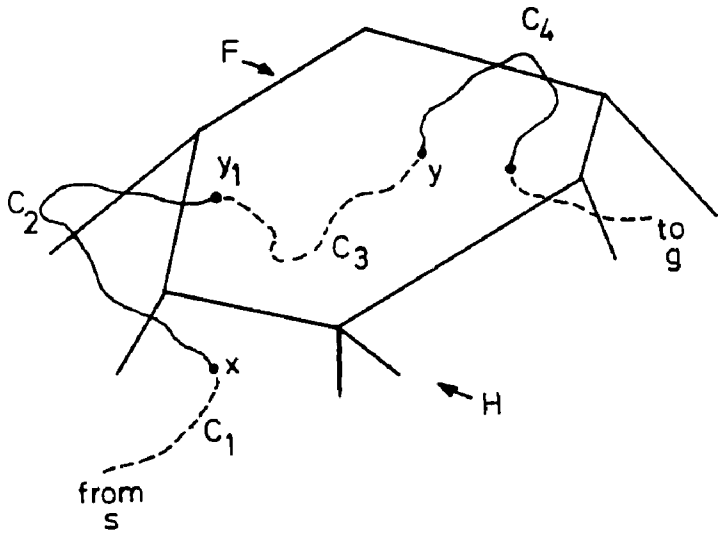


Figure 1.1 Curves C_1, C_2, C_3, and C_4 of theorem 1.1.

belongs to $C(s,g;P)$. We shall therefore restrict our attention to curves C lying entirely within H.

Decompose $H-(\bigcup_1^n \text{int}P_i)$ into 3-simplexes in such a way that any pair of simplexes either have no common points or share a face (of arbitrary dimension). If C, lying within H, first enters the interior of a 3-simplex of the decomposition at x and leaves it for the last time at y, then $C=C_1C_2C_3$ where C_2 is a curve from x to y. Letting $C_2'=xy$ and setting $C'=C_1C_2'C_3$, we have $l(C')\leqslant l(C)$. Clearly, $C'\in C(s,g;P)$ since if C' cuts a face of this 3-simplex, so does C (figure 1.2).

Now we can restrict our attention to polygonal paths within H. Suppose $C=b_1\cdots b_t$ is such a path with $b_1=s$ and $b_t=g$. If there is a bend point of C not belonging to an edge of a polyhedron in P, let b_i, $1<i<t$ be the first such point. Consider the intersections of all polyhedra in P with the triangle $T=b_{i-1}b_ib_{i+1}$. The intersections are a set of polygons (possibly degenerated to line segments or points) in aff$T$. Let V denote the totality of the vertices of these polygons and consider $H_i=\text{conv}(\{b_{i-1},b_{i+1}\}\bigcup V)$. If $V=\Phi$ then let $C_2'=b_{i-1}b_{i+1}$; otherwise bdH_i consists of $b_{i-1}b_{i+1}$ and another polygonal path C_2' where $l(C_2')\leqslant l(b_{i-1}b_ib_{i+1})$ by convexity. Accordingly, $l(C')\leqslant l(C)$ if $C'=C_1C_2'C_3$ and C' also belongs to $C(s,g;P)$ (figure 1.3). Continuing likewise, we can reduce the number of bend points of C which are not on the edges of any P_i while simultaneously reducing $l(C)$, until we obtain the required polygonal path C'. □

It is noted that the polygonal path C' found in theorem 1.1 is not necessarily unique. We shall call any such path an s-to-g *shortest path*. Thus, FINDPATH asks for the characterization (i.e., the determination of the bend points) of an s-to-g shortest path. The following theorem shows that listing all such paths will in general require exponential time in card$P$:

THEOREM 1.2 There exists instances of FINDPATH where there are $\Theta(2^n)$ s-to-g shortest paths.

Proof. We shall demonstrate this with an example workspace in 2-space (figure 1.4). It is easy to extend this example to 3-space by erecting prisms for each polygon. In figure 1.4 we require that $n\geqslant 4$ and even, and all odd-numbered polygons are large enough so that a shortest path cannot tour around them. We also require that all even-numbered polygons are squares whose centers of mass are on sg; similarly, the nonreflex vertices of P_1 are aligned on sg. It is seen that for the described workspace there are exactly $2^{n/2}$ s-to-g shortest paths. □

Now we return to the relationship between FINDPATH and FINDPATH

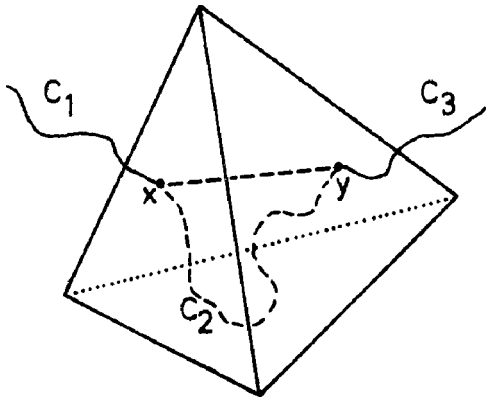


Figure 1.2 A curve intersecting the 3-simplex of theorem 1.1.

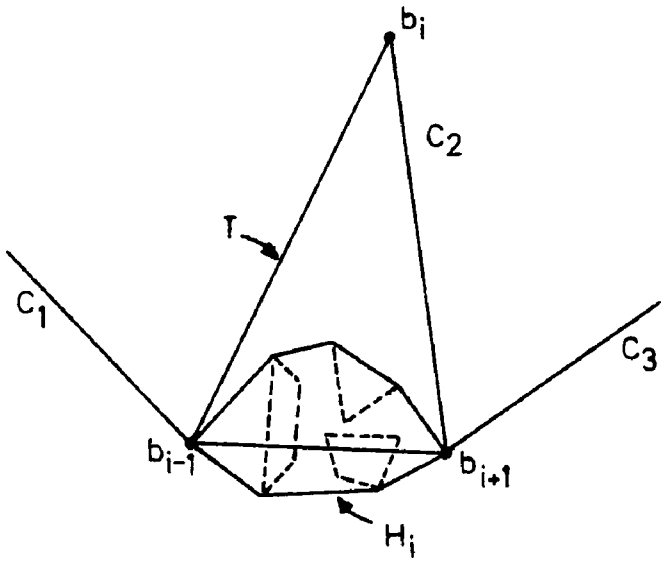


Figure 1.3 The intersections of P_i with the triangle $b_{i-1}b_i\,b_{i+1}$ of theorem 1.1 shown with dashed lines.

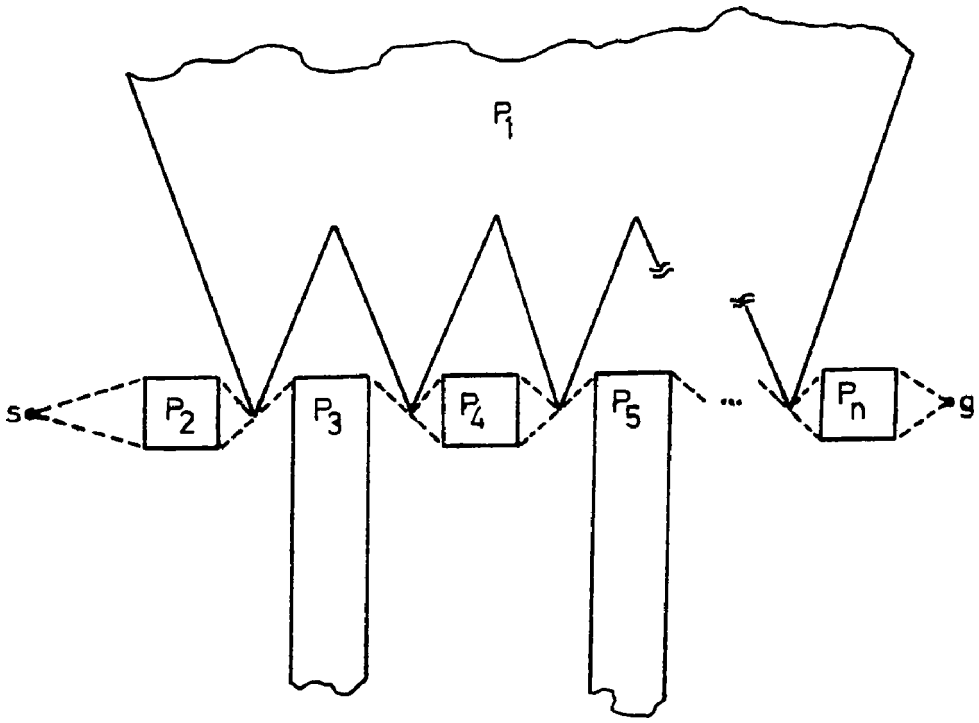


Figure 1.4 The workspace of theorem 1.2 which gives rise to $2^{n/2}$ s-to-g shortest paths.

(decision). It may be thought that each of the optimization, evaluation (i.e., the one that asks *only* the length of the shortest path), and decision versions of FINDPATH, in this order, is not harder than the preceding ones. That is, if we are given the bend points of a shortest path then we can compute its length, and certainly compare this number with an integer. However, the cost of performing the latter operation deserves more attention. Consider the following problem which we shall call:

> FINDPATH (suboptimality)
> INSTANCE: Same as in FINDPATH, and an s-to-g polygonal path C which is free of intersections with the given polyhedra, i.e., $C \in C(s,g;P)$.
> QUESTION: Is C suboptimal, i.e., is there a second obstacle-avoiding s-to-g polygonal path $C' \in C(s,g;P)$ where $l(C') \leqslant l(C)$?

Clearly, FINDPATH (suboptimality) requires a way to compare the lengths of two paths, $l(C)$ and $l(C')$, which are basically a sum of square roots. The obvious method of repeated squaring to eliminate the square roots has exponential worst-case complexity. In fact, there is no known method to guarantee the correctness of this comparison with only a polynomial effort in the number of square roots, r, involved. Garey et al[38] cite the best known upper bound on the number of places of accuracy as $O(m\,2^r)$ where m is the number of digits in the original symbolic expression. (Nevertheless, Mignotte considers the following related problem in [74]. If a and b are given algebraic numbers and if we know the approximations a' and b' for them, can we decide whether $a = b$? That is, is there a "numerical" proof of $a = b$ in contrast to an algebraic proof which can be difficult in many cases? He proves that this is indeed possible showing that a numerical attack is feasible to simplify radicals.)

In view of this, one may be tempted to think that problems such as FINDPATH or GEOMETRIC TRAVELING SALESMAN[85] are intractable simply because of the evaluation issues stemming from the irrational square roots. In [85] Papadimitriou shows that this is not really so, at least for the GEOMETRIC TRAVELING SALESMAN problem. He defines a new distance metric $d'(x,y) = \text{ceil}(d(x,y))$ and shows that the problem remains NP-complete under this metric.

1.3 Motivation

Like many other problems in computational geometry (GEOMETRIC TRAVELING SALESMAN being a prime example) FINDPATH can be described in simple terms even to a novice whereas its solution seems to involve many

difficult problems. Aleksandrov[3], Courant and Robbins[17], Efimov[26], and Lyusternik[72] study some relevant problems such as planar developments of polyhedra but their arguments are frequently informal and nonconstructive. Saaty[93] defines a specific instance of FINDPATH that we shall call BOUNDARY FINDPATH (cf. section 3.1) and hints at the usefulness of polyhedral developments to solve it but does not attempt to formulate a solution. Jacobson and Yocom[51, 52] and Jacobson[53] deal with interesting problems in the proximity of FINDPATH such as shortest paths within polygons or hypercubes, yet their approach is far from being algorithmic. Thus, while TRAVELING SALESMAN has received wide attention for a long time (cf. Lawler et al[54]), FINDPATH has essentially been overlooked until the last couple of years.

The recent creation of interest in FINDPATH owes to the advent of *task-level* (or *model-based*) robot programming systems such as the SHRDLU program of Winograd[114]. Although SHRDLU was essentially built to study natural language understanding, its geometric nature makes it a good candidate to demonstrate our point. SHRDLU simulates a robot manipulator working in a *blocks world*, a workspace solely cluttered with simple objects such as cubes and pyramids. Users requesting a particular action state it in simple English. Thus, requests such as "Pick up a red block," or "Stack up both of the red blocks and either a green cube or a pyramid" call for goals to be achieved by SHRDLU. A geometric path planner designs and submits these plans to a manipulator controller which eventually simulates them. During the design, the path planner utilizes a geometric model of the workspace.

The path planner comprises three important functions. The function FINDSPACE tries to find a place for a given block so that it will be stable on its new support. If FINDSPACE cannot determine a suitable place, MAKESPACE intervenes to clear away some obstacles. Finally, FINDPATH moves a block to a given location while avoiding clashes with the other blocks. It is possible to reduce an "object" FINDPATH problem of this sort to a "point" FINDPATH problem, i.e., the version we shall study in this thesis. Lozano-Perez[71] cites an approach which is known as the *configuration space (Cspace)* method for the reduction. In simple terms, *Cspace* method maps an object movement problem to a point movement problem by simultaneously growing the obstacles in the workspace and shrinking the moved object to a *reference point*. The method gives exact results for movements consisting of only translations. An approximate scheme which allows rotations is also possible.

1.4 Related Research on Motion Planning

Directly relevant work on 3-dimensional FINDPATH is quite recent. We shall review works by Franklin[29], Sharir and Schorr[102], O'Rourke et al[83], and Mitchell and Papadimitriou[75] in more detail in sections 4.2 and 4.3. On the other hand, there is a wealth of material on 2-dimensional FINDPATH and in the general area of what we shall call *motion planning* [2]. We shall, somewhat artificially, divide this material into three classes and review only the most important ones in each class:

1.4.1 More Practical Works

Howden presents a simple program[50] which determines in discrete 2-space if a sofa can be moved from a source to a goal inside a house. He calls this the SOFA PROBLEM. Eastman[21] defines the representational requirements for space planning problems. Boyse[8] considers the problem of interference detection among polyhedral objects. He discusses two types of interference checking: detection of intersections among stationary objects and detection of collisions among moving objects. The former check is crucial in the preliminary phase of FINDPATH where the disjointness of the obstacles must be guaranteed. Udupa[110] maintains the free space as a variable resolution description of the feasible positions of the reference point found via *Cspace* method. Safe paths are found by recursively introducing new goals into a linear path until the complete path is confined to the free space. Sussman[105] and Pfister[87] work on heuristics for solving FINDSPACE; their work is in the spirit of Winograd's.

Lozano-Perez and Wesley present an algorithm[70] for 2-dimensional FINDPATH where there are polygonal obstacles with forbidden interiors. Their algorithm uses the fact that the shortest path is a polygonal path whose bend points are some vertices of the given polygons. It first constructs a *visibility graph* (*Vgraph*) whose nodes consist of $\{s, g\} \cup (\bigcup_{1}^{n} \text{vert} P_i)$. An arc in *Vgraph* connects a pair of nodes corresponding to a pair of vertices visible to each other and carries a weight equal to the distance between these vertices. It is easy to search *Vgraph* using a shortest path algorithm for graphs (cf. Tarjan[107]) once it is available. Lee and Preparata mention a result of Lee[65] claiming that 2-dimensional FINDPATH with arbitrarily oriented disjoint line segments is solvable in $O(m^2 \log m)$ time where m is the number of segments. Special configurations of the segments make more efficient techniques available. Tompa gives several $O(m)$ algorithms[109] for the case of presorted vertical rays each extending either to $+\infty$ or $-\infty$. Sharir and Schorr also have an $O(m \log m)$ algorithm[102] if the obstructions are all vertical lines with several point apertures. Since *Vgraph* on m nodes where

$$m = \sum_{1}^{n} \text{card}(\text{vert} P_i)$$

has in general $\Omega(m^2)$ arcs, Lozano-Perez and Wesley's graph search has an $\Omega(m^2)$ lower bound even assuming that *Vgraph* is available. Lee and Preparata, in their quest for an $o(m^2)$ time algorithm, propose two improvements. They show that there are $O(m \log m)$ solutions[65] when the given segments form a simple polygon to which the source and the goal are internal, and when the given segments are disjoint and parallel.

A polygon in R^2 that can translate and rotate has three degrees of freedom whose legal values are constrained by the other obstacles in the workspace. In [71] Lozano-Perez considers these degrees of freedom to be three Cartesian coordinates of a point that must move while avoiding obstacles in R^3 and then uses an approximate 3-dimensional FINDPATH algorithm. His student Nguyen gives a heuristic algorithm[78] for 2-dimensional FINDPATH. His algorithm is based on describing the free space as a network of "linked cones." Cones capture the freeways and the bottlenecks between obstacles while the links capture the connectivity of the free space. In an earlier work[10] Brooks represents the free space as a union of overlapping cones and gives an algorithm which finds "good" paths for convex polygonal bodies. The paths are good in the sense that the distance of closest approach to an obstacle over the path is usually far from minimal over the class of homotopic paths. Brooks and Lozano-Perez[9] describe an implementation of 2-dimensional FINDPATH with rotation. Donald[19] presents a technique for hypothesizing a "channel volume" through the free space containing a class of collision-free paths. His channel volume starts out by surrounding the moving object at the source and grows toward the goal.

Although they use the L_1 (Manhattan or rectilinear) metric, two other algorithms for 2-dimensional FINDPATH are also important. In [61] Larson and Li give a Dijkstra-type algorithm for finding the minimum length path between the source and the goal in the presence of polygonal barriers. Lipski[67] deals with n horizontal and vertical line segments in the plane. He gives an $O(n \log^2 n)$ time algorithm (which can be improved to $O(n \log n \log\log n)$ using more complicated data structures) to find, for a given source segment, the shortest paths to all other goal segments.

1.4.2 Complexity-theoretic Works

Donald[20] is a fine tutorial on the geometric complexity of motion. Reif, in a pioneering paper on the complexity of motion planning[92] treats the MOVER'S PROBLEM, another name for the SOFA PROBLEM. He approximates the sofa and the house as systems of linear inequalities which are

slightly perturbed by ϵ. His algorithm gives an indeterminate answer if the sofa, shrunk by a factor of $1-\epsilon$, can be moved from the source to the goal but when grown by a factor of $1+\epsilon$ the move is impossible. He then extends the problem to allow the sofa to consist of multiple polygons linked together at several vertices and proves that this instance is PSPACE-hard.

In a series of seminal papers, Schwartz and his students study the complexity of the PIANO MOVER'S PROBLEM, yet another name for the SOFA PROBLEM. In [95] Schwartz and Sharir give a topological analysis of the space of positions of a polygon moving in the plane in the presence of polygonal obstacles. In [96] they work with the manifold of allowable positions of a manipulator and reduce the motion planning problem to finding the connected components of an algebraic manifold. They partition this manifold into connected subregions of simple structure which they call "cells." Their technique makes use of Tarski's decision method for elementary algebra and geometry[108] and its more efficient refinements introduced by Collins and his coworkers[16, 4, 5]. Their algorithm is polynomial in the number of edges of the obstacles but exponential in the number of degrees of freedom. Schwartz and Sharir[97] solve the following instance: given several 2-dimensional discs inside a polygon, find a continuous motion connecting two specified configurations of these objects during which they avoid collision with the polygonal edges and with each other. Their algorithm is polynomial in the number of edges of the polygon but exponential in the number of moving discs. This is expected since in [104] Spirakis and Yap proved the strong NP-hardness of moving many discs. Sharir and Ariel-Sheffi[103] treat the case of a "spider" with several arms. This is a 2-dimensional robot consisting of several arms all jointed at one common endpoint and free to rotate past each other. In [98] Schwartz and Sharir solve the motion planning problem for a line segment moving amidst polyhedra. A paper by O'Dunlaing et al[79] describes other approaches to motion planning using notions such as "retraction" which has interesting connections with differential topology (cf. Schwartz[94]).

Hopcroft and his associates worked on several motion planning problems. In [45] Hopcroft et al investigate the 2-dimensional SOFA PROBLEM in which the object is a manipulator with an arbitrary number of joints. They give two polynomial algorithms: one for moving the manipulator confined within a circle from a given configuration to another, and the other for moving it from its initial position to a position in which its tip reaches a given point inside the circle. In a closely related work[46] they prove that a planar linkage can be constrained to stay inside a bounded region with a straight-edge boundary by only adding a polynomial number of new links to it. They also show that the question of whether a planar linkage in some initial configuration can be moved so that a designated joint reaches to a given point in the plane is PSPACE-hard. Hopcroft and Wilfong[47] deal with the motion of objects in

contact. In another paper[48] they study the motion planning for multiple objects where each object is a polygon with edges parallel to the axes of R^2. They show that for a rectangular workspace this problem is in PSPACE. Since in [49] Hopcroft et al proved that this problem (which they called the WAREHOUSEMAN'S PROBLEM) is PSPACE-hard, the problem is PSPACE-complete.

1.4.3 Voronoi-based Works

A detailed review of Voronoi diagrams is beyond the limits of this thesis. Section 4.1 summarizes some essential material and points out to major references.

O'Dunlaing and Yap[80] use the Voronoi method for planning the motion of a disc in a 2-dimensional region bounded by polygonal obstacles. They construct a Voronoi diagram using the obstacles. The diagram is a planar graph of straight and parabolic edges and is characterized as the set of points which are equidistant from at least two obstacles. Kirkpatrick[57] gives an $O(n \log n)$ time algorithm for this problem where n is the number of edges of the obstacles: a somewhat simpler but $O(n \log^2 n)$ time algorithm is given by Lee and Drysdale[63]. If there is a continuous collision-free motion of the disc between the source and the goal positions of its center then O'Dunlaing and Yap show that there is another motion of the same sort during which the center of the disc moves entirely along the Voronoi diagram. In a descendant of this work, O'Dunlaing et al tackle the case of a moving line segment instead of a disc[81, 82]. They present an algorithm running in time slightly greater than $O(n^2 \log n)$ for constructing a skeleton representation of a generalized Voronoi diagram. Theirs is a variant of Kirkpatrick's continuous skeletons[57].

1.5 Organization of the Thesis

After this introduction, the rest of the thesis is structured as follows. Chapter 2 discusses the general solution of FINDPATH and several ways of speeding it. Chapter 3 deals with two special instances of FINDPATH, both in the presence of a single convex obstacle. Chapter 4 offers two Voronoi-based approaches to solve FINDPATH more efficiently after some preprocessing. Chapter 5 describes the features of SP, a workbench written to explore and to experiment with our shortest path algorithms. (Appendix summarizes the internals and the available functions of SP.) Finally, chapter 6 cites important open problems along with our conclusions.

2. SOLUTION OF THE GENERAL INSTANCE

This chapter has two sections. In section 2.1 we study FINDPATH in its full generality and give a simple yet very inefficient algorithm. Section 2.2 offers some methods to speed this algorithm up.

2.1 A Brute-force Algorithm

In this section we shall give an algorithm which, although very inefficiently, is able to solve FINDPATH. Throughout this thesis, we shall assume, without loss of generality, that sg itself is *not* the shortest path. This can be checked by applying standard line-polyhedron intersection detection algorithms.

We shall first solve the following problem which will be the main ingredient of our algorithm for FINDPATH:

> OPTIMAL LINE VISIT
> INSTANCE: Lines $L=\{L_1, \cdots, L_n\}$ and points s, g in R^3. (Each line is specified by a pair of points.)
> QUESTION: What is the s-to-g shortest path constrained to pass through the sequence of lines $L_1, \cdots, L_n$ in that order?

It is clear that the shortest path is also a polygonal path in this case. Let $b_1, \cdots, b_n$ denote the bend points of the shortest path on $L_1, \cdots, L_n$, respectively. For notational ease, we shall take $b_0 = s$ and $b_{n+1} = g$. The unit vectors along each L_i will be denoted by u_i: their direction can be arbitrary.

LEMMA 2.1 For $1 \leqslant i \leqslant n$, the angle between the vectors $b_{i-1}b_i$ and u_i is equal to the angle between $b_i b_{i+1}$ and u_i (figure 2.1).
Proof . This is well-known; Sharir and Schorr[102] give a proof. □

LEMMA 2.2 The solution of OPTIMAL LINE VISIT is unique.
Proof . Sharir and Schorr[102]. □

With the aid of these lemmas, it is easy to formulate a solution for FINDPATH. From lemma 2.1, the bend points of the shortest path satisfy the following vector equation for all $1 \leqslant i \leqslant n$:

$$<b_{i-1}b_i, u_i> / \ \mathrm{d}(b_{i-1}, b_i) = <b_i b_{i+1}, u_i> / \ \mathrm{d}(b_i, b_{i+1}) \qquad (1)$$

Denoting the pair of points defining L_i by f_i and g_i, we also have the following vector equation for all $1 \leqslant i \leqslant n$:

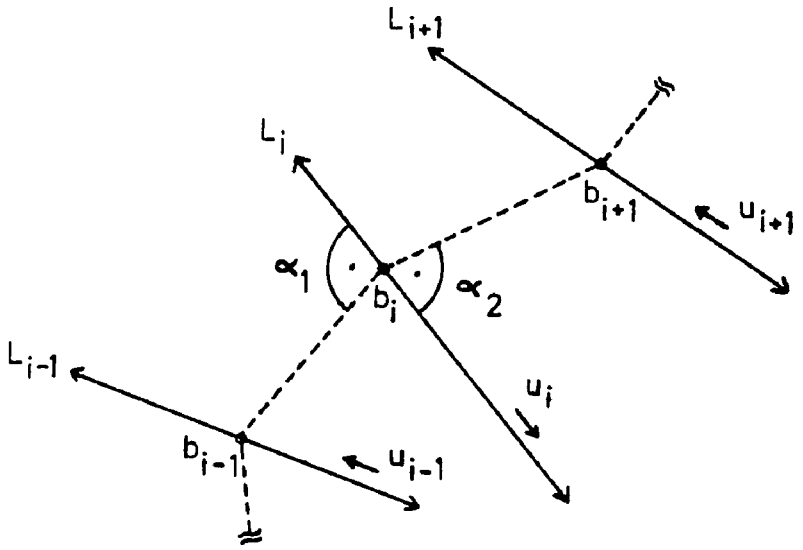


Figure 2.1 The angles α_1 and α_2 are equal if these bend points belong to a shortest path.

$$b_i = f_i + x_i (g_i - f_i) \tag{2}$$

where x_i is L_i's parameter. Substituting the values of b_i found in (2) in (1) and using the fact that $u_i = f_i g_i / \mathrm{d}(f_i, g_i)$, we obtain n nonlinear equations in n unknowns as follows:

$$\begin{gathered} Q_1(x_1,x_2)=0 \\ Q_2(x_1,x_2,x_3)=0 \\ \cdots \\ Q_i(x_{i-1},x_i,x_{i+1})=0 \\ \cdots \\ Q_{n-1}(x_{n-2},x_{n-1},x_n)=0 \\ Q_n(x_{n-1},x_n)=0 \end{gathered} \tag{3}$$

In (3) we note three properties. First, each equation is a quartic (after removing the square roots caused by u_i's) in at most three variables. Second, the polynomial Q_i, $1<i<n$ is dependent only on three unknowns: x_{i-1}, x_i, x_{i+1}. Third, both Q_1 and Q_n have only two unknowns since b_0 and b_{n+1} are constant points. Since we have n independent equations in n unknowns in (3), we can solve this system to determine the real root $(x_1, \cdots, x_n)$ using the classical theory of elimination (cf. van der Waerden[113] and Collins[15] for a detailed description: Ku and Adler[60] and Moses[77] are good short expositions). Once the x_i's are known, the bend points are easily computed using (2).

The idea of elimination is based on the polynomial resultants. We assume that the reader is familiar with the latter and omit several definitions and theorems. The crux of the elimination is the following:

THEOREM 2.1 Let F be an algebraically closed field. Let

$$A(x_1, \cdots, x_r) = \sum_0^m A_i(x_1, \cdots, x_{r-1}) x_r^i,$$

$$B(x_1, \cdots, x_r) = \sum_0^n B_i(x_1, \cdots, x_{r-1}) x_r^i$$

be members of field $F[x_1, \cdots, x_r]$, of positive degrees m and n in x_r, and let $C(x_1, \cdots, x_{r-1})$ be the resultant of A and B with respect to x_r. (This

will be denoted by $C=\text{res}(A,B)$.) If $(a_1,\cdots,a_r)$ is a common zero of A and B, then $C(a_1,\cdots,a_{r-1})=0$. Conversely, if $C(a_1,\cdots,a_{r-1})=0$, then at least one of the following holds:

$$A_m(a_1,\cdots,a_{r-1})=\cdots=A_0(a_1,\cdots,a_{r-1})=0,$$
$$B_n(a_1,\cdots,a_{r-1})=\cdots=B_0(a_1,\cdots,a_{r-1})=0,$$
$$A_m(a_1,\cdots,a_{r-1})=B_n(a_1,\cdots,a_{r-1})=0,$$
$$\text{For some } a_r\in F, A(a_1,\cdots,a_r)=B(a_1,\cdots,a_r)=0.$$

Proof . Collins[15]. □

Theorem 2.1 suggests the following method to calculate the solution of a system of n polynomial equations in n unknowns. Let $A_i(x_1,\cdots,x_n)=0$, $1\leqslant i\leqslant n$ be the given system. Let $B_i(x_1,\cdots,x_{n-1})=\text{res}(A_i,A_n)$, $1\leqslant i\leqslant n-1$. Note that the variable x_n does not appear in B_i's. Now, let $C_i(x_1,\cdots,x_{n-2})=\text{res}(B_i,B_{n-1})$, $1\leqslant i\leqslant n-2$. Neither x_n nor x_{n-1} appear in C_i's. Continuing in this manner, one eventually obtains a polynomial $Z_1(x_1)=0$. By theorem 2.1, if $A_i(a_1,\cdots,a_n)=0$ for $1\leqslant i\leqslant n$, then $Z_1(a_1)=0$. Similarly, we can compute polynomials $Z_2,\cdots,Z_n$ such that $Z_2(a_2)=0,\cdots,Z_n(a_n)=0$ whenever $(a_1,\cdots,a_n)$ is a zero of the original system $A_1=0,\cdots,A_n=0$. If Z_i's are all nonzero polynomials then there are only a finite number of n-tuples $(a_1,\cdots,a_n)$ such that $Z_1(a_1)=0,\cdots,Z_n(a_n)=0$. and every solution of the given system will be among these n-tuples. Thus it is sufficient to find the roots of these univariate polynomials and then determine which n-tuples are solutions of the initial system. Heindel[44] describes a system which employs Sturm's theorem[113] to calculate within any prespecified (yet unlimited) accuracy all the real zeros of a univariate integral polynomial, and this is exactly what we want since complex roots are not meaningful in our setting.

Although the above method is straightforward there are some problems that make it difficult to implement in a computer for large values of n. The following theorem explains this:

THEOREM 2.2 Let R be a commutative ring with identity and let A and B be polynomials of positive degree over R. Then there exist polynomials S and T over R such that $AS+BT=\text{res}(A,B)$ and $\deg S<\deg B$, $\deg T<\deg A$. ($\deg A$ denotes the degree of polynomial A.)
Proof . Collins[15]. □

This means that $\text{res}(A,B)$ can have powers up to $2\deg A\deg B$. Thus, in the above discussion the final univariate polynomials Z_i may be in the worst-case

of order doubly-exponential in n. (There is certainly a similar growth in the size of the coefficients of the resultant, but we think that this is somewhat less important.) In appendix (c) we review Macsyma's *resultant* function for elimination.

It is admitted by experts[55] that elimination is very inefficient for solving a system of nonlinear equations even for small n. Thus numerical methods may be desirable to solve (3). A classical method that is applicable is the Newton-Raphson method. Let J denote the $n \times n$ Jacobian matrix defined by $J_{ij} = \partial Q_i / \partial x_j$. Then the $(r+1)$-st iteration of the Newton-Raphson method is given as $X_{r+1} = X_r - Q(X_r) J^{-1}(X_r)$ where $X = (x_1 \cdots x_n)$, $Q = (Q_1 \cdots Q_n)$. The convergence of the method is dependent on the initial estimate X_0. A typical feature of the method is that the number of correct significant digits roughly doubles at each iteration. In appendix (c) we review our Macsyma function *olvang* which implements this method.

We have seen that purely algebraic schemes, such as elimination, may lead to severe performance penalties. Use of local methods, such as Newton-Raphson, can be a hit-or-miss process. In this case, continuation methods provide the best known technique for computing solutions to polynomial systems. (It is noted however that continuation methods also run quite inefficiently. Morgan[76] states that the run time of his SYMPOL system which is based on continuation is proportional to the "total degree" (the product of the degrees of the individual equations) of the system under investigation.) A major advantage of the method is that it is not affected by the choice of initial estimates of solutions and thus is very reliable. Garcia and Zangwill[37] is the definitive source on the subject of continuation. Similarly, Garcia and Zangwill[36] treat the case of solving polynomial equations using this method. Here we give an overview of the latter.

We want to solve the polynomial system $A_i(x_1, \cdots, x_n) = 0$, $1 \leqslant i \leqslant n$, or written compactly, $A(x) = 0$. We assume that we can solve $G(x) = 0$, a "simplified" system to be specified below. We then form the homotopy:

$$H(x,t) = tA(x) + (1-t)G(x) \text{ where } t \in [0,1]$$

and generate a "flow" from the solutions of $G(x) = 0$ to those of the original system by starting at all solutions for $t = 0$ and following the corresponding paths until $t = 1$. (The details of how this is done are not particularly illuminating and will be omitted here.) Let $c_1, \cdots, c_n$ be complex numbers and let $p(k)$ stand for $\deg A_k$. Then the k-th component of G will be defined as $G_k = x_k^{1+p(k)} - c_k$. The following theorem is very important:

THEOREM 2.3 If for all $x_0 \epsilon C^n$ (n-dimensional complex space) and $t_0 \epsilon [0,1]$ (a) $A(x_0)=0$ implies $\det DA(x_0) \neq 0$, and (b) $H(x_0,t_0)=0$ implies $\text{rank} DH(x_0,t_0)=0$ then all solutions to $A(x)=0$ can be found by the process described above. (DH (resp. DA) is the $2n \times (2n+1)$ (resp. $2n \times 2n$) real matrix of partial derivatives of H (resp. A). $\det A$ denotes the determinant of A whereas $\text{rank} A$ denotes its rank.)
Proof . Garcia and Zangwill[36]. □

One may inquire when the hypothesis of theorem 2.3 holds. The following result is thus relevant:

THEOREM 2.4 The set of all $(c_1, \cdots, c_n) \epsilon C^n$ for which part (b) of the hypothesis of theorem 2.3 does not hold has measure zero in C^n.
Proof . A sketch is given by Morgan[76]. □

The elegance of theorem 2.4 is that we can select $c_1, \cdots, c_n$ in random and reasonably expect them to satisfy the hypothesis of theorem 2.3, assuming that part (a) of it holds. The method may lead to some "bad" paths which diverge to infinity. Since theorem 2.3 guarantees that each solution will have a convergent path this is not a theoretical deficiency; yet it degrades the performance. That is, theorem 2.3 generates $\prod_1^n (1+p(i))$ paths wheras the maximum number of solutions is only $\prod_1^n p(i)$. We did not implement a continuation method in this thesis although we are convinced that it may be the best available approach to solve OPTIMAL LINE VISIT.

Until now, we summarized a way of solving OPTIMAL LINE VISIT based on the solution of a system of equations, either using purely algebraic techniques or numerical analysis. Another method is to apply optimization. Consider the objective function:

$$D(x_1, \cdots, x_n) = \sum_0^n \mathrm{d}(b_i, b_{i+1})$$

This is the total length of an s-to-g path with bend points b_i. To minimize D, we can use the formula $X_{r+1} = X_r - G(X_r) H^{-1}(X_r)$ where G is the gradient vector $(\partial D / \partial x_1 \cdots \partial D / \partial x_n)$ and H is the $n \times n$ Hessian matrix defined by $H_{ij} = \partial^2 D / \partial x_i \partial x_j$. Like Newton-Raphson method, the success of the optimization rests on the good selection of X_0. In appendix (c) we review our Macsyma function *olvdis* written to implement this method.

We shall now consider the following variant of OPTIMAL LINE VISIT:

OPTIMAL EDGE VISIT
INSTANCE: Line segments $E=\{e_1,\cdots,e_n\}$ and points s,g in R^3. (Each line segment is specified by its endpoints.)
QUESTION: What is the s-to-g shortest path constrained to pass through the sequence of line segments $e_1,\cdots,e_n$ in that order?

OPTIMAL EDGE VISIT requires the bend points of the shortest path to belong to the line segments; hence direct application of OPTIMAL LINE VISIT to the lines defined by the given line segments will not in general provide the right answer. Nevertheless, we shall show that OPTIMAL EDGE VISIT is solvable after 3^n applications of OPTIMAL LINE VISIT to instances whose total individual size is n. Denote the line carrying e_i by L_i and apply OPTIMAL LINE VISIT to the obtained instance. If the bend points b_i of the computed path are such that $b_i \epsilon cle_i$, $1\leqslant i\leqslant n$, then we are done. Otherwise, some bend points of the computed path are external to some segments. In this case, it must be clear that the shortest path through the given edges will bend at at least one endpoint of the given edges at which point it will not subtend equal entry and exit angles contrary to lemma 2.1.

Let the endpoints of edge e_i be denoted by e_{i1} and e_{i2}. Let *combs* be the set of all combinations of length n where each combination is obtained by selecting one member from the following sets in that order:

$$\{L_1,e_{11},e_{12}\}$$
$$\{L_2,e_{21},e_{22}\}$$
$$\cdots$$
$$\{L_n,e_{n1},e_{n2}\}$$

A given member of *combs* defines at most n subproblems that should be submitted to OPTIMAL LINE VISIT. The subproblems are simply obtained by isolating the parts of the given combination between consecutive pairs of endpoints.

EXAMPLE Assuming that $n=7$ we should solve 3^7 problems, one for each combination. For instance, one of the combinations is the following: $s,L_1,e_{21},L_3,L_4,e_{52},L_6,e_{71},g$. For this one we must solve four subproblems as follows:

apply OPTIMAL LINE VISIT to instance s,L_1,e_{21}
apply OPTIMAL LINE VISIT to instance e_{21},L_3,L_4,e_{52}
apply OPTIMAL LINE VISIT to instance e_{52},L_6,e_{71}
apply OPTIMAL LINE VISIT (trivially) to instance e_{71},g □

Note that if during the application of OPTIMAL LINE VISIT to a subproblem any of the found bend points turn out to be external to the given segments then we can abort the whole computation for this combination since it will be accounted for in a future combination. Once we are through with a combination we add the lengths of the paths computed for the subproblems together to find the length of the shortest path for this combination. The combination which gives the shortest among the members of *combs* is chosen as the optimal one.

After these preparations, we are now ready to give our algorithm:

Algorithm FINDPATH

1. Let $E = \bigcup_{1}^{n} \text{edge} P_i$ and $EP = 2^E - \Phi$ where 2^E denotes the power set of E.
Let $C' = \Phi$ and $l' = +\infty$.
2. Do the following steps for each permutation of each $T \in EP$:
   2.1 Compute the shortest path C by applying OPTIMAL EDGE VISIT to the edges in T, and s and g.
   2.2 If C intersects any P_i then discard it and continue with step 2. Otherwise, if $l(C) < l'$ then replace C' by C, let $l' = l(C')$, and continue with step 2.

End

This algorithm gives one s-to-g shortest path (C') and its length (l'). To obtain all s-to-g shortest paths in case there are more than one, the following modifications can be made. Initialize a list $cands = \Phi$ at step 1. When a path C is not discarded at step 2.2, let $cands = cands \bigcup C$ and skip the rest of the step. Finally, we add a third step which sorts *cands* in ascending order by path length and takes the initial portion of it containing the equal-length paths.

2.2 Efficiency Considerations

Assume temporarily that we have a black box which efficiently solves OPTIMAL EDGE VISIT numerically. Even in this case, FINDPATH would be extremely inefficient since it tries all permutations of edges to compute the shortest path. (We think here that testing a path against the obstacles for intersection detection is subsumed by the root finding, a reasonable assumption.) Since $n! = \sqrt{2\pi n}\ (n/e)^n (1 + O(1/n))$, the execution time of FINDPATH would still be dominated by $O(n^n)$. Currently, there is no algorithm which does provably better than this bound. In other words, the best known algorithm for FINDPATH is nothing but the naive "try all possibilities" approach.

In fact, even the numerical solution of OPTIMAL LINE VISIT is nontrivial as we already remarked in section 2.1. We shall demonstrate that OPTIMAL LINE VISIT becomes intractable if we relax the requirement that the lines should be visited in the given order. To see that, assume the existence of a polynomial algorithm to solve unordered OPTIMAL LINE VISIT. We shall prove that one can use this as a subroutine to solve GEOMETRIC TRAVELING SALESMAN in polynomial time, a contradiction in the light of the fact that the latter is strongly NP-complete both in the "path" and the "tour" versions[85]. (In the path version we are *not* supposed to come back to our starting city.)

For the proof, let the cities for GEOMETRIC TRAVELING SALESMAN (path) be the points $c_1, \cdots, c_n$ in the plane. Consider the lines passing through c_i's and perpendicular to the plane and denote them by $L_1, \cdots, L_n$. Since we assumed that OPTIMAL LINE VISIT works in polynomial time, we are assured that for a given pair of cities which act as the source and the goal, we can find a source-to-goal shortest path in 3-space efficiently. (This path is obviously confined to the plane.) It starts at the source, visits some permutation of $L_1, \cdots, L_n$, and ends at the goal. Since the number of such pairs is only $O(n^2)$, we can try all of them and select the one which gives the shortest path, which necessarily is the shortest salesman path in the plane. The contradiction follows.

Clearly, what we need to reduce the execution time of FINDPATH is a quick way of deciding which permutations of edges are simply useless. If we can do this then all such permutations can be discarded without ever applying OPTIMAL EDGE VISIT on them. In the following assume that all members of P are convex. Let $e_1, \cdots, e_k$ be a given sequence of edges. The first short-cut is obvious:

LEMMA 2.3 Let e_i, e_{i+1}, $1 \leqslant i < k$ be two edges in the above sequence such that $e_i, e_{i+1} \epsilon \text{ edge} P_j$, for a $P_j \epsilon P$. Let F_1 and F_2 be the lowest indexed faces of P_j which hold e_i and e_{i+1}, respectively. If $F_1 \neq F_2$ then this sequence cannot contribute to the shortest path computation.
Proof . The convexity of P_j guarantees that there is no way to go from e_i to e_{i+1} without intersecting $\text{int} P_j$. □

The second short-cut is as follows:

LEMMA 2.4 An s-to-g shortest path cannot pass through any member of $\bigcup_1^n \text{vert} P_i$. ($P_i$'s are still considered convex.)
Proof . If C is a shortest path on the boundary of a convex polyhedron $P_k \epsilon P$, then Sharir and Schorr[102] prove that C cannot pass through a

vertex of P_k. We shall extend this to the case of several convex polyhedra. If the shortest path C in this case first lands on a face of a polyhedron P_k, then visits a vertex v_j of it, and then takes off then their proof is certainly applicable. Otherwise, the situation is shown in figure 2.2. Now consider two points on C: a in the ϵ-neighborhood of v_j but before it, and b in the ϵ-neighborhood of v_j but after it. Obviously a and b cannot see each other. Let P' be a small subset of P_k including v_j and its say, $c\epsilon$-neighborhood for a constant c, $0<c<1$. It is possible to construct $\text{conv}(P' \bigcup \{a, b\})$ to compute a shorter path from a to b which avoids v_j. □

This may be extended to the case of arbitrary polyhedra. However, we must be careful about the vertices where lemma 2.4 holds in this case. Figure 2.3 shows a nonconvex polyhedron with a "valley" which forces the s-to-g shortest path to pass through vertices v_1 and v_2. It is seen that lemma 2.4 is correct only for convex vertices.

The importance of lemma 2.4 is that it eliminates the need to perform OPTIMAL EDGE VISIT totally in an exclusively convex environment. That is, if we perform a shortest path computation via OPTIMAL LINE VISIT on a sequence of edges in the presence of convex obstacles and find out that the path should pass through some vertices then we can immediately discard this sequence.

Returning to lemma 2.3, can we find additional short-cuts? One such way would be to eliminate a sequence which has two consecutive edges (belonging to different polyhedra) which cannot "see" each other:

DEFINITION Two edges e_1 and e_2 of a polyhedral scene $P = \{P_1, \cdots, P_n\}$ are *visible* to each other if there exists a point of e_1 visible to a point of e_2. Otherwise, they are *invisible* to each other. □

This definition suggests the following decision problem:

> EDGE-TO-EDGE VISIBILITY
> INSTANCE: As in the above definition.
> QUESTION: Are e_1 and e_2 invisible to each other?

We are not aware of an algorithm to solve this problem. Let H be the convex hull (a tetrahedron) of the endpoints of the given line segments. Clearly, e_1 and e_2 are invisible to each other if the object obtained by subtracting all polyhedra from H contains "walls" so as to block any line segment connecting a point of e_1 to a point of e_2. However, we do not know how one can detect this situation. An obvious probabilistic approach (which is vulnerable to an adversary) is to suitably choose point pairs (one of which belonging to e_1 and

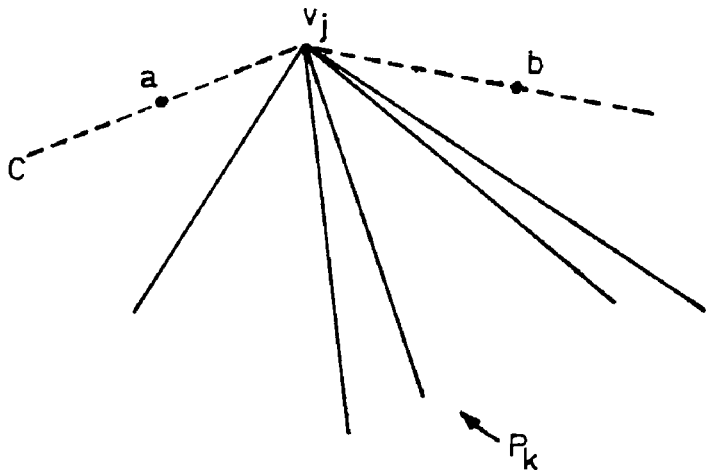


Figure 2.2 A shortest path cannot visit a convex vertex.

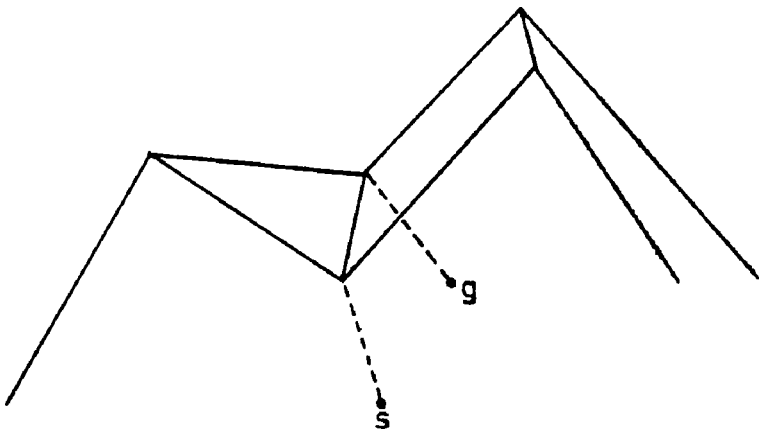


Figure 2.3 A shortest path can visit a nonconvex vertex.

the other to e_2) and to test the line segment connecting them against all polyhedra for intersection. If a prespecified number of tries are always concluded with reported intersections then we may argue that the given line segments are invisible to each other.

Finally, note that if two edges are only *partly visible* to each other then a sequence holding them in consecutive order in FINDPATH cannot in general be discarded without further consideration.

3. SOLUTIONS OF TWO SPECIFIC INSTANCES

This chapter has two sections. Both sections treat the case $P = \{P_1\}$ where P_1 is convex. (In the rest of the chapter we shall simply use P instead of P_1.) Section 3.1 details the case where $s, g \in \text{bd}P$. Section 3.2 uses the solution for that case to solve the instance where s and g are both external to P. This chapter is largely based on a revision of our work in [30, 32].

3.1 Shortest Paths on a Convex Polyhedron

Consider the following specific instance of FINDPATH:

> BOUNDARY FINDPATH
> INSTANCE: Convex polyhedron P, specified points $s, g \in \text{bd}P$ where $s \neq g$.
> QUESTION: Which $C \in \text{bd}P$ has the shortest length?

Before we solve this problem we give an argument as to its importance. Let $P = \{P_1, \cdots, P_n\}$ be a set of *convex* polyhedra. If we are allowed to compute a reasonably short (but *not* the shortest) s-to-g path then we can pursue the following strategy. Let P' be the subset of P such that every member of P' is intersected by sg. (If a polyhedron is intersected by sg only once then it is not included in P'; thus P' consists of polyhedra intersected by sg at precisely two points.) Applying BOUNDARY FINDPATH to each member of P' we obtain an s-to-g polygonal path which comprises two types of curves: line segments through free space, and polygonal paths along the boundaries of the polyhedra between where the path "lands" from free space and "takes off" again (figure 3.1). Repeated optimization of this path is possible and will frequently yield a better (with fewer bend points) and shorter path (figure 3.2) although it is not difficult to come up with cases where repeated optimization might cause clashes.

We shall assume throughout the thesis that the boundary representation is used to define a polyhedron P. In this representation scheme each vertex is defined by its x, y, z coordinates and each face is given as a list of pointers to the vertices, ordered in counterclockwise around the boundary of the face with respect to a point above it. It is convenient to think of vertex labels or face labels as positive integers.

DEFINITION The *face graph (Fgraph)* of a convex polyhedron P is an undirected graph $Fgraph = (FV, FE)$ with unit arc weight, $FV = \{i : F_i \text{ is a face of } P\}$ and $FE = \{(i, j) : F_i \text{ and } F_j \text{ are adjacent}\}$. (Two faces are *adjacent* if they share an edge.) □

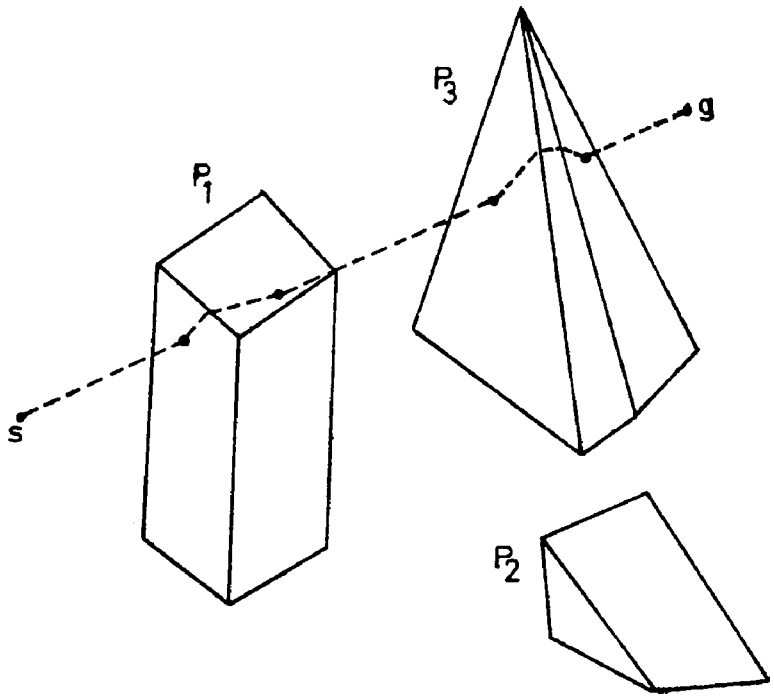


Figure 3.1 A reasonably short path between s and g in the presence of convex polyhedral obstacles.

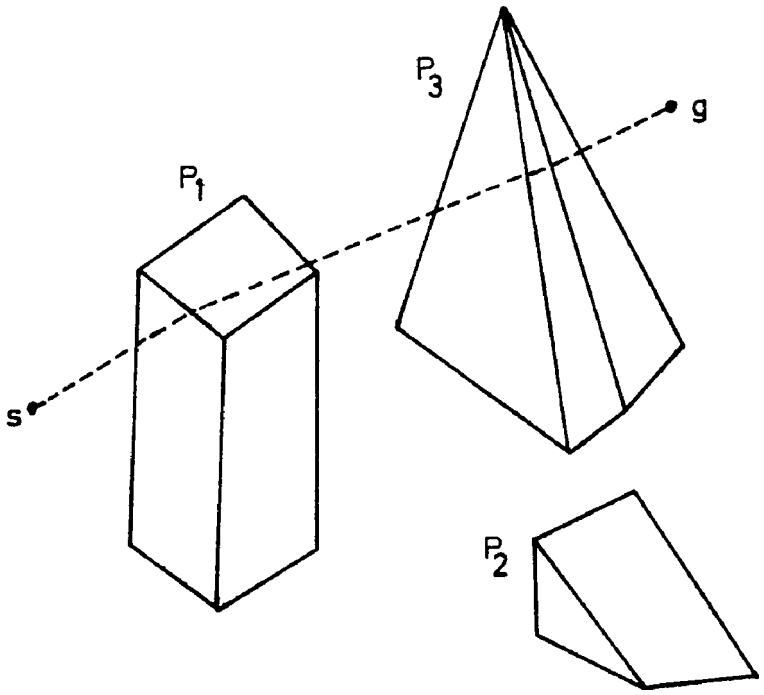


Figure 3.2 Further optimization of the path shown in figure 3.1 yields a shorter path with fewer bend points.

EXAMPLE Figure 3.3(a) shows the face graph of a cube. In figure 3.3(b), the face graph of a parallelepiped is given to note that *Fgraph* only preserves the adjacency information. Figure 3.3(c) shows that two faces with a common point only are *not* considered adjacent by this definition. □

DEFINITION Let $C \epsilon \mathrm{bd}P$ be an s-to-g polygonal path. The sequence of faces that C enters defines the *face visit sequence* of C which will be denoted by $\mathrm{fvs}C$. □

Thus $\mathrm{fvs}C$ is a walk in *Fgraph* between the nodes corresponding to faces F_s and F_g, the faces of P containing s and g, respectively. An immediate consequence of the above definition is:

LEMMA 3.1 Let $C' \epsilon \mathrm{bd}P$ be an s-to-g shortest path. Then $\mathrm{fvs}C'$ is a simple walk in *Fgraph*.
Proof. If this is not true then C' enters a face at least twice. Recalling the fact that the faces of P are all convex, we can then further shorten C', a contradiction. □

From now on, all face visit sequences will therefore be assumed to be simple.

In addition to *Fgraph*, a useful construct that will be used by BOUNDARY FINDPATH is the planar development of a given face visit sequence. It is well-known that the boundary of a convex polyhedron has the structure of a planar graph. Therefore, the totality of the faces of P, situated in 3-space in certain mutual relationship, can be represented in 2-space (specifically the xy-plane) by a system of polygons identified with the faces of P. Frechet and Fan[35] refer to this as the "planar polygonal schema" while Abelson and DiSessa[1] prefer the term "atlas" noting the similarity of this to a road atlas. The relationship between a planar development and the planar polygonal polygonal schema will be apparent after the following description of how to obtain the latter.

In the xy-plane associate with each face of P a polygon having the same metrical form. (Two polygons have the same *metrical form* if they can be made to coincide by translations and rotations.) Define the *glue* relationship between the pairs of edges of these polygons such that two glued edges come from the *same* edge of P. Figure 3.4 illustrates this for a cube. Each edge in the planar polygonal schema is glued to exactly one edge.

DEFINITION A *planar development* corresponding to a face visit sequence $1, \cdots, k$ is a union of polygons $F_1, \cdots, F_k$ of the planar polygonal schema of P. In the planar development two polygons F_i and F_{i+1}, $1 \leqslant i < k$ are united along the edge that they are glued to each other, and do not overlap.

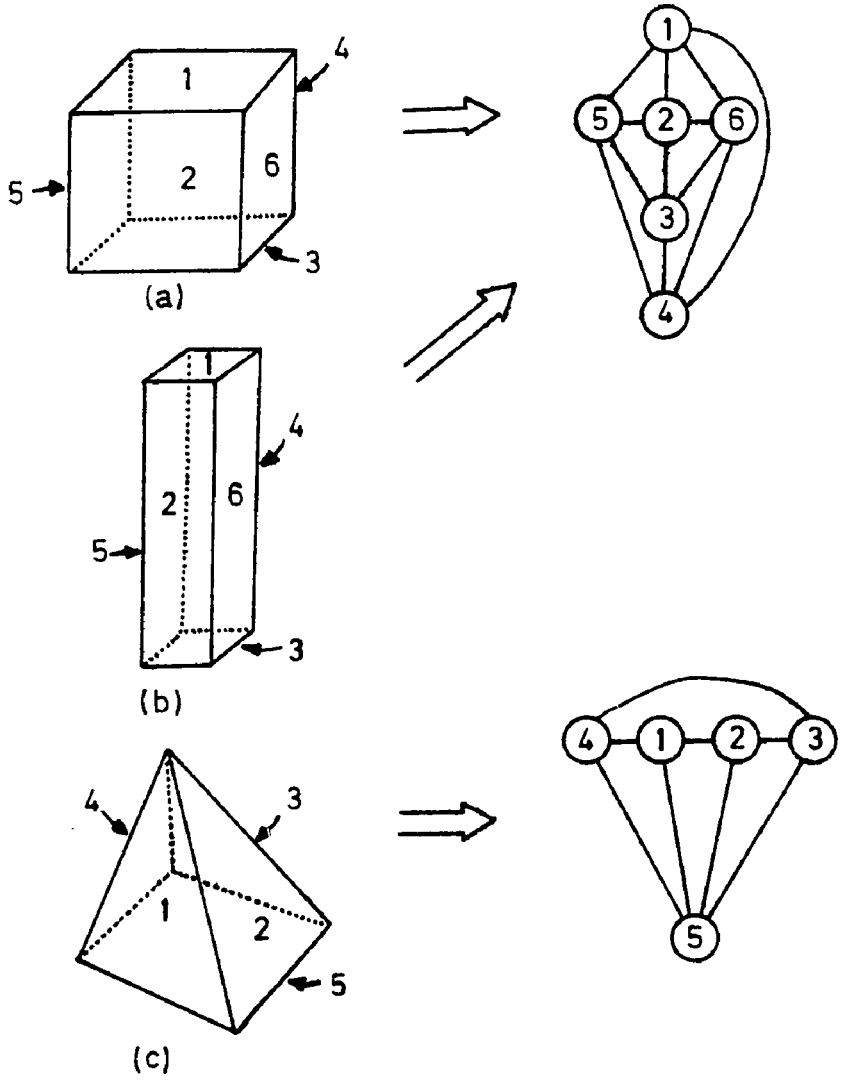


Figure 3.3 Face graphs of convex polyhedra:
(a) cube, (b) prism, (c) pyramid.

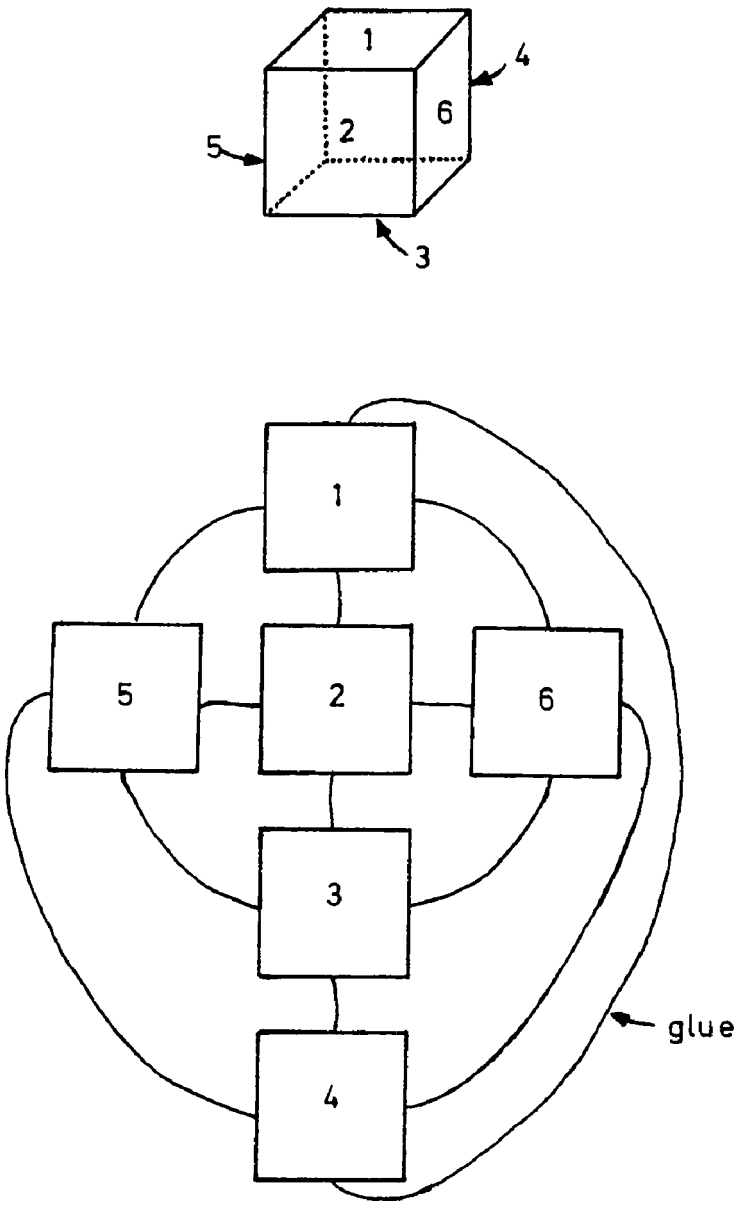


Figure 3.4 The planar polygonal schema of a cube.

□

DEFINITION The *image* of a point on a polyhedron under a planar development is the point in the plane that it ends up under the development. □

DEFINITION A planar development is *legal* if the line segment connecting the images of the source and the goal is internal to the development. □

EXAMPLE Figure 3.5 shows several planar developments computed and drawn by SP, our shortest path workbench. The objects (and their file names) are as follows. Figure 3.5(a): cube (*cube.DAT*), figure 3.5(b): icosahedron (*icosa.DAT*), figure 3.5(c) and (d): dodecahedron (*dodeca.DAT*). It is noted that the last development is not legal. □

It should be apparent that once a planar development is built, it can be moved to any position and orientation in the plane without changing the intrinsic geometry of the paths. We shall now give a procedural definition of a planar development:

DEFINITION To compute the planar development of a face visit sequence $1, \cdots, k$, start with F_1. If affF_1 is parallel to the xy-plane then translate P by a suitable amount so that F_1 is now in the xy-plane; otherwise, let the dihedral angle between affF_1 and the xy-plane be D and and rotate P about the line aff$F_1 \bigcap xy$-plane by D to map F_1 to the xy-plane. The remaining faces F_i are inductively handled as follows. Let e be the common edge of F_{i-1} and F_i whose dihedral angle is D. Rotate P by D about aff$e$ to place F_i to the xy-plane while avoiding overlaps with F_{i-1}'s polygon which is already there. □

Now we are ready for:

Algorithm BOUNDARY FINDPATH

1. Let F_s and F_g be the faces of P including s and g, respectively. Assume that $F_s \neq F_g$; otherwise the shortest path is $C' = sg$.
2. Let FVS be the set of all simple walks in $Fgraph$ of P, between the nodes corresponding to F_s and F_g. Initialize $vs' = \Phi$ and $l' = +\infty$.
3. For each member of FVS do the following steps:

   3.1 Compute the planar development corresponding to this face visit sequence. Let s' and g' be the images of s and g in the xy-plane under the same development. (They can be computed along with the planar development.)

   3.2 If the development in step 3.1 is not legal then continue with step 3. Otherwise, if $d(s', g') < l'$ then replace vs' with the current face visit sequence, let $l' = d(s', g')$, and continue with

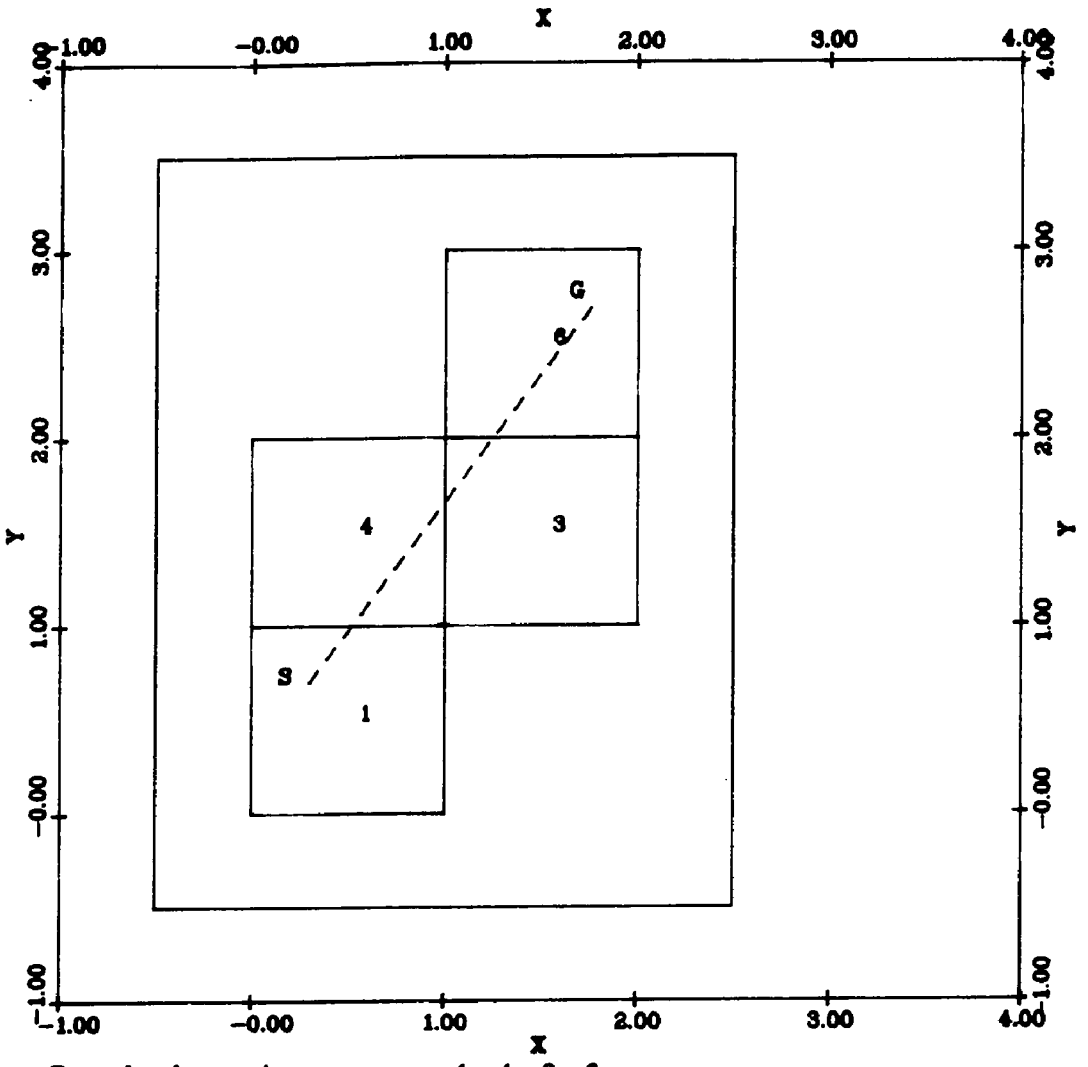
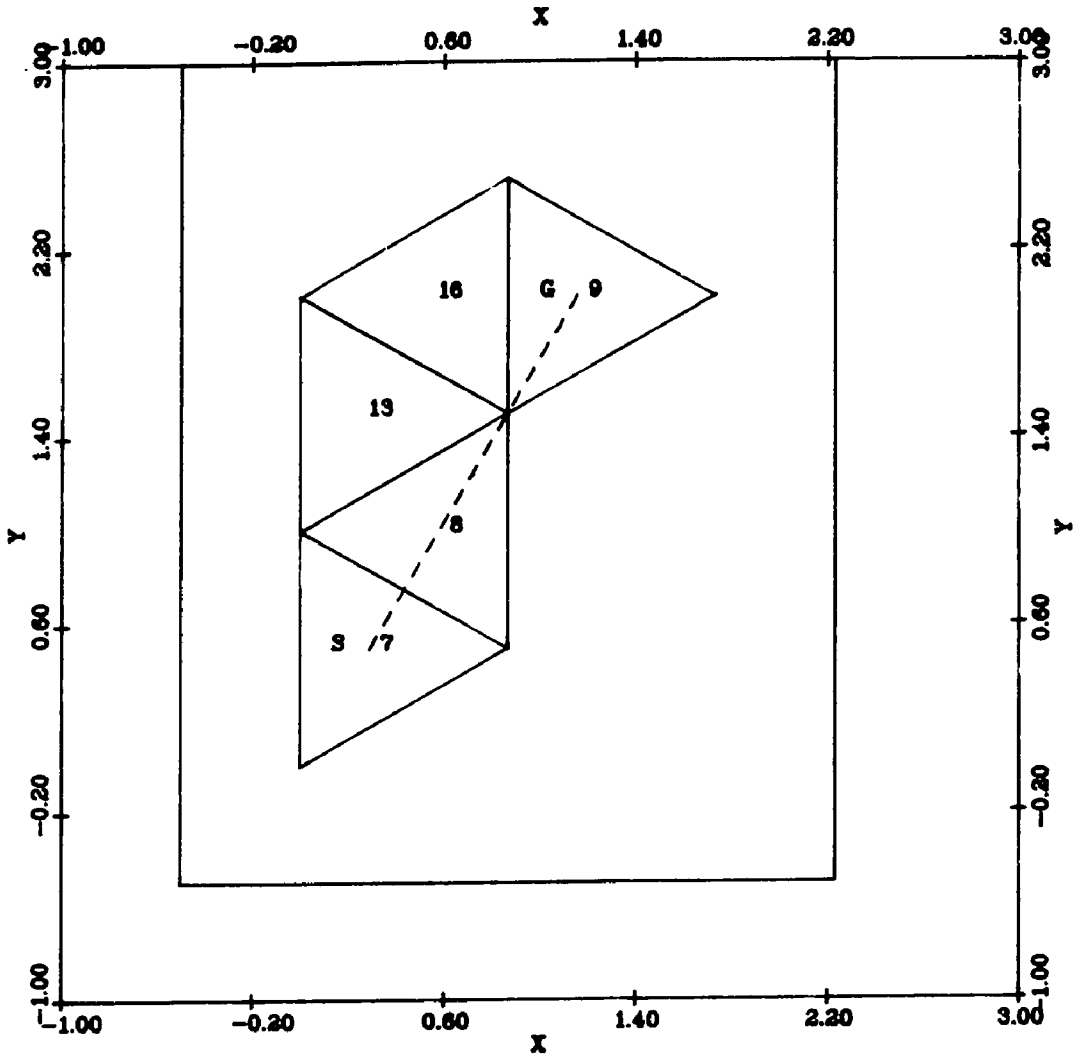
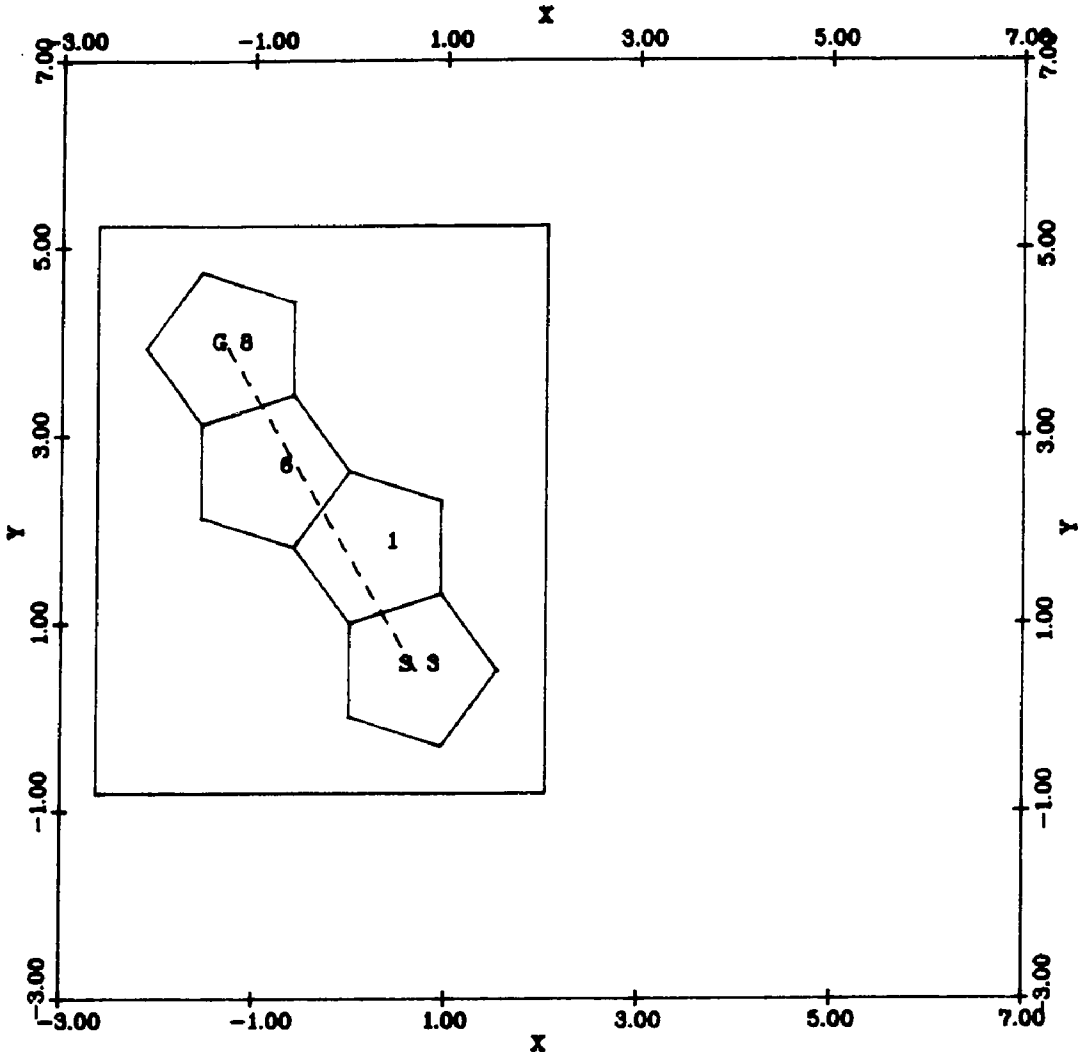


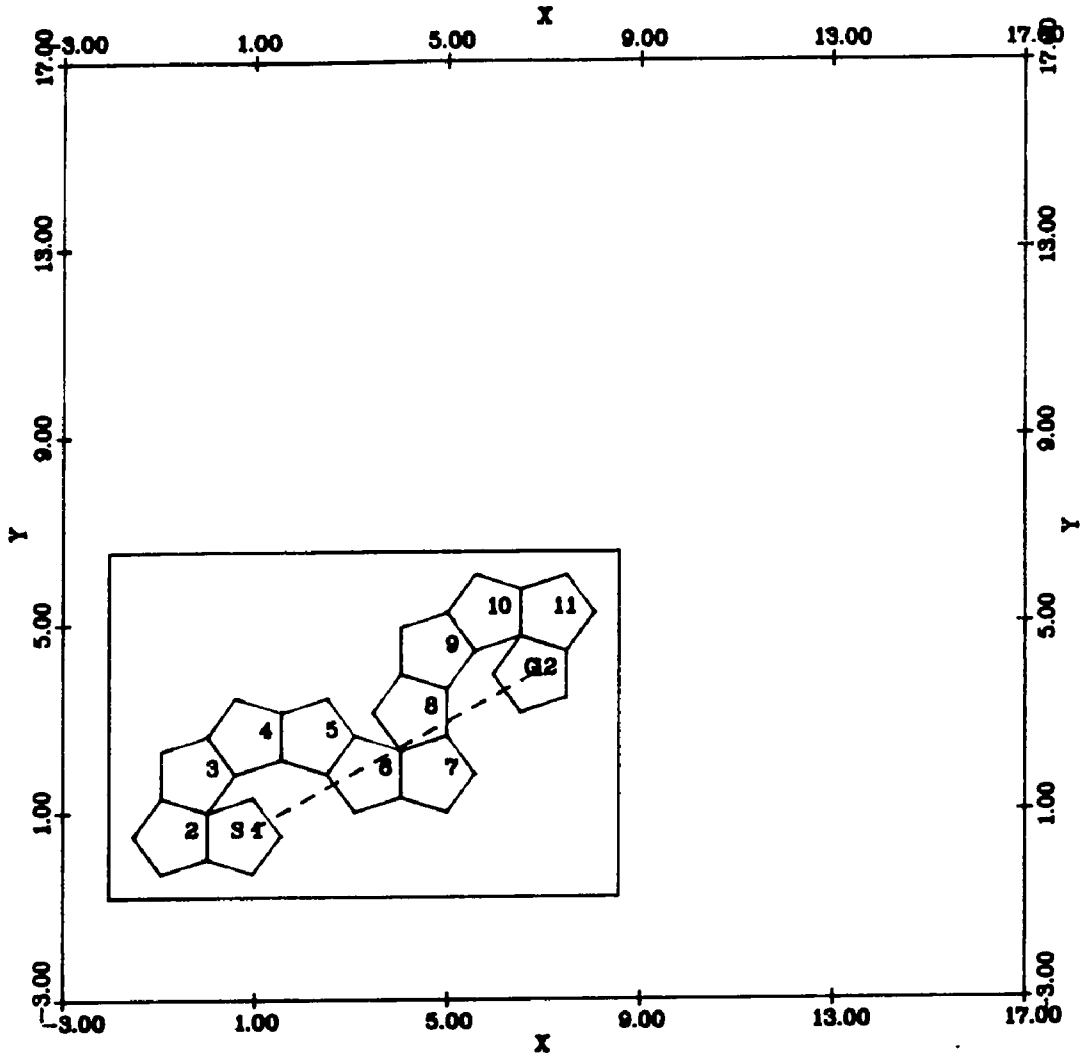
Figure 3.5 Some planar developments computed and drawn by SP. The objects that are developed are as follows: (a) cube, (b) icosahedron, (c) and (d) dodecahedron.



Face development sequence:	7	8 13 16	9	
Source face no.:	7			
Goal face no.:	9			
Source point coords.:	0.948	0.230	0.504	
Goal point coords.:	0.192	1.206	0.504	
Source point coords. (plane):	0.289	0.500		
Goal point coords. (plane):	1.154	2.000		
Source-to-goal distance:	1.732			
Bounding box extents:	-0.500	2.232	-0.500	3.000



Face development sequence:	3	1	6	8	
Source face no.:	3				
Goal face no.:	8				
Source point coords.:	0.380	1.447	0.616		
Goal point coords.:	0.996	−0.447	1.612		
Source point coords. (plane):	0.688	0.500			
Goal point coords. (plane):	−1.276	3.927			
Source-to-goal distance:	3.950				
Bounding box extents:	−2.627	2.039	−0.809	5.236	



Face development sequence:	1	2	3	4	5	6	7	8	9	10	11	12
Source face no.:			1									
Goal face no.:			12									
Source point coords.:			0.688		0.500			0.000				
Goal point coords.:			0.688		0.500			2.227				
Source point coords. (plane):			0.688		0.500							
Goal point coords. (plane):			6.782		3.927							
Source-to-goal distance:			6.991									
Bounding box extents:			-2.039		8.558			-0.809			6.545	

step 3.
4. At this point l' is the length of the shortest path and vs' is the face visit sequence that should be used to to compute the shortest path itself. To do this, first compute the planar development of vs' (and s' and g') and intersect $s'g'$ with all pairwise common edges of the polygons in the development. The intersection points in the plane are then easily used to compute the bend points of the shortest path C'. We know from the planar development of vs' the distance of an intersection point from a vertex in the plane. All we need is to identify the vertex of P in three dimensions that led to this vertex. Then marking the point which is away from this vertex the same distance over the edge touched by the shortest path we locate the bend point for one intersection. The others are found completely analogously.
End

An efficient way of checking whether a planar development is legal follows. Let $e_1, \cdots, e_t$ be the sequence of edges that are glued in the development. Then the development is legal if $s'g'$ intersects every e_i. Note that this is always easier than testing if $s'g'$ is internal to the planar development's boundary.

To list the simple walks between two nodes of a graph we can use the algorithm of Yen[115] which works in $O(kn^3)$ if there are n nodes in the graph and we require the first k simple walks in increasing walk length. Katoh et al[56] give an improved algorithm for the same task with a time complexity $O(kf(n,m))$ under the assumption that shortest walks form one node to all others can be found in $f(n,m)$ time where m is the number of arcs in the graph. Since $f(n,m)$ is either $O(n^2)$ or $O(m \log n)$ in the worst case, this algorithm is more efficient than Yen's.

Determining the value of cardFVS is difficult. Garey and Johnson[39] state that the following problem is NP-hard:

> K-th SHORTEST PATH
> INSTANCE: Graph $G=(V,E)$, positive integer lengths l_e for each $e \in E$, specified nodes $s,t \in V$, positive integers b and k.
> QUESTION: Are there k or more distinct simple walks from s to t in G, each having total length b or less?

They also mention that K-th SHORTEST PATH remains NP-hard even if $l_e = 1$ for all $e \in E$, and is solvable in pseudo-polynomial time (polynomial in card V, k, and $\log b$) and accordingly, in polynomial time for any fixed k (e.g., Yen's algorithm). The difficulty of K-th SHORTEST PATH basically resides in the following counting problem which was proven to be #P-complete by Valiant[111]:

S-T PATHS (SELF-AVOIDING WALKS)
INSTANCE: Graph $G=(V,E)$, specified nodes $s,t \in V$.
QUESTION: What is the number of walks from s to t that visit every node *at most* once?

Cartwright and Gleason[11] give an algorithm to compute this number but their algorithm uses large matrices and is inefficient even for small values of card V.

The problem of counting (s,t)-walks in (s,t)-planar graphs is also #P complete[90]. (A graph is called *source-sink planar*, or (s,t)-*planar* in short, if it has a planar representation with nodes s and t on the boundary.) Provan[91] states that the approximation problem for (s,t)-walks is unsolved, in the sense that the following problem is open:

> "For *any* fixed $\epsilon<1$, does there exist a polynomial algorithm which for a given (s,t)-planar graph G, will give an approximation N_0 for the number N of (s,t)-walks in G which satisfies $|N-N_0|<\epsilon N$?"

On the other hand, for *any* fixed $\epsilon>0$, can it be proven that the above problem is NP-hard? Currently, the only known approximations are to count the minimum length walks or to enumerate as many walks as possible (using a large value of k in Yen's algorithm, for example).

It is not hard to find a convex polyhedron which has an exponential number of simple walks in its *Fgraph* (figure 3.6). If there are l lateral faces of this pyramid-like object (not counting the small triangular faces) then the number of simple walks between the nodes F_s and F_g in the figure is $\Omega(2^{l/2})$. This result also shows that our BOUNDARY FINDPATH algorithm is of exponential time complexity in the number of faces of P. Section 4.3 describes polynomial algorithms by other researchers for this problem. Unfortunately, theirs do not seem to admit practical implementations. In the light of this, the algorithm presented in this section is applicable for objects of moderate complexity. We can also try to test only a certain section (e.g., first few in increasing walk length) of the face visit sequences between the source and the goal faces for an object with many faces with the hope that the shortest path is generated by a short face development sequence. The shortest path rendered by the legal developments found among the developments that these sequences give may be taken as the true shortest path although this too is certainly vulnerable to an adversary (figure 3.7).

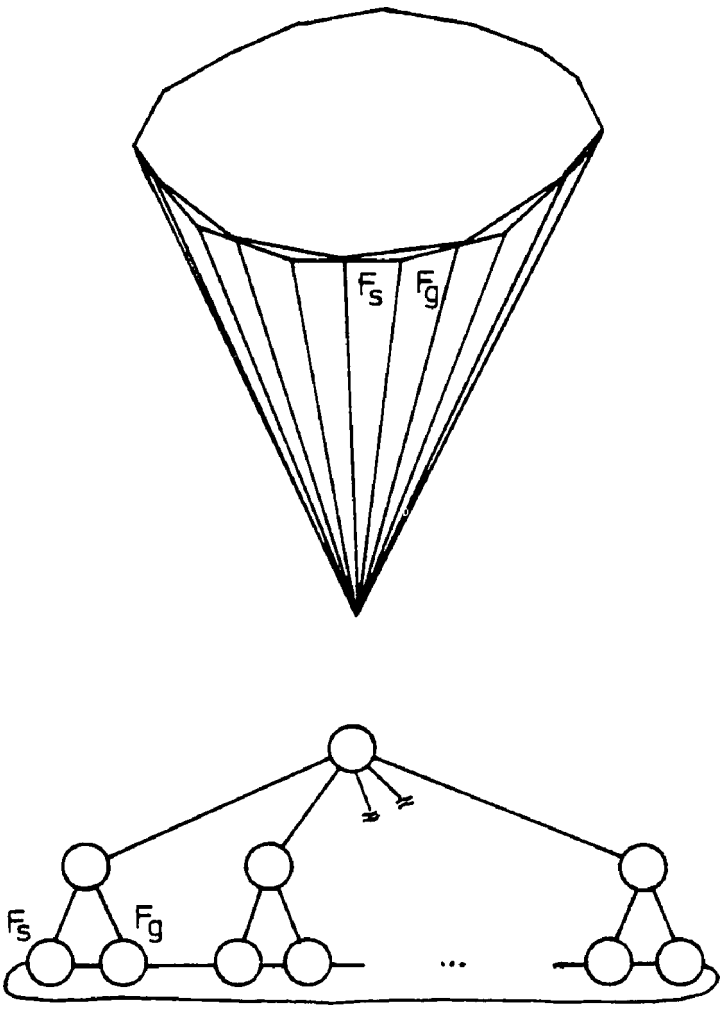


Figure 3.6 There exist an exponential number of simple walks between nodes F_s and F_g in *Fgraph* of this object.

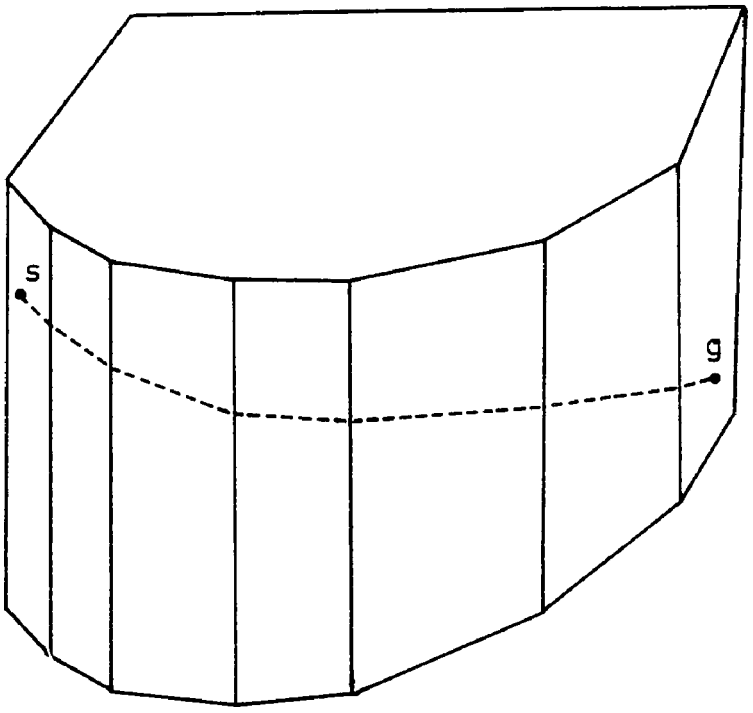


Figure 3.7 Shortest face visit sequences do not always render the shortest paths.

3.2 Shortest Paths around a Convex Polyhedron

Now we inspect the following variant of BOUNDARY FINDPATH:

> EXTERIOR FINDPATH
> INSTANCE: Convex polyhedron P and points s, g where at least one of them is external to P, $s \neq g$.
> QUESTION: Which $C \in C(s, g; P)$ has the shortest length?

Without loss of generality, we shall treat the case where s and g are both outside P. In this case, the following fact is useful:

LEMMA 3.2 Let $H = \text{conv}(\{s, g\} \bigcup \text{vert}P)$. Then an s-to-g shortest path for EXTERIOR FINDPATH is entirely on $\text{bd}H$.
Proof . Similar to theorem 1.1. □

Thus, once H is computed using standard 3-space convex hull algorithms, we can apply BOUNDARY FINDPATH to the instance made of H, s, and g. (Preparata and Hong[88] give an algorithm to find the 3-dimensional convex hull in $O(n \log n)$ time for n points.) However, there is a slight difficulty with this approach. Assuming that we want to know which edges of the original polyhedron the shortest path touches, we must keep extra information with H, i.e., which vertex of H comes from which vertex of P. Below, we shall give a more direct method to obtain H while keeping this information implicitly using a visibility-based approach. (Sutherland et al[106] give an excellent overview of polyhedral visibility.)

DEFINITION For a convex polyhedron P and a point x external to it, the *silhouette edges* of P are members of

$$\{e : e \in F_1 \text{ and } e \in F_2 \text{ where } F_1 \in F_{vis} \text{ and } F_2 \in F_{invis}\}$$

Here, F_{vis} (resp. F_{invis}) is the set of visible (resp. invisible) faces of P from viewpoint x. □

Clearly, $F_{vis} \bigcap F_{invis} = \Phi$ since a face of a convex polyhedron is either completely visible or completely hidden to an observer. Let $E_{sil,s}$ (resp. $E_{sil,g}$) denote the silhouette edges of P from s (resp. g). It is clear that the faces of H will be the union of three *disjoint* sets:

$$\text{bd}H = F_{tri,s} \bigcup F_{tri,g} \bigcup (F_{invis,s} \bigcap F_{invis,g})$$

Here $F_{tri,s}$ (resp. $F_{tri,g}$) is a set of triangular faces each characterized by an edge of $E_{sil,s}$ (resp. $E_{sil,g}$) and s (resp. g). In essence, these are the lateral faces of a pyramid-like object with (generally nonplanar) basis $E_{sil,s}$ (resp. $E_{sil,g}$) and apex s (resp. g). It is noted that the geometric complexity of object H is the same as with P.

We conclude this section with an algorithm to compute the silhouette edges of P from a point x external to it:

Algorithm SILHOUETTE
1. Compute $F_{vis,x}$, the visible faces of P from viewpoint x, by checking line segments xc_i where c_i is the center of mass of face F_i against all F_j, $j \neq i$ for intersection. $F_{vis,x}$ consists of all F_i which do not cause any intersection.
2. Let the totality of the visible edges of P from x be $E_{vis,x} = \{e : e \in F$ where $F \in F_{vis,x}\}$. Sort $E_{vis,x}$ and eliminate *both* of duplicate members. The remaining edges are precisely the edges of $E_{sil,x}$.
End

Figure 3.8 demonstrates the working of EXTERIOR FINDPATH on a simple object.

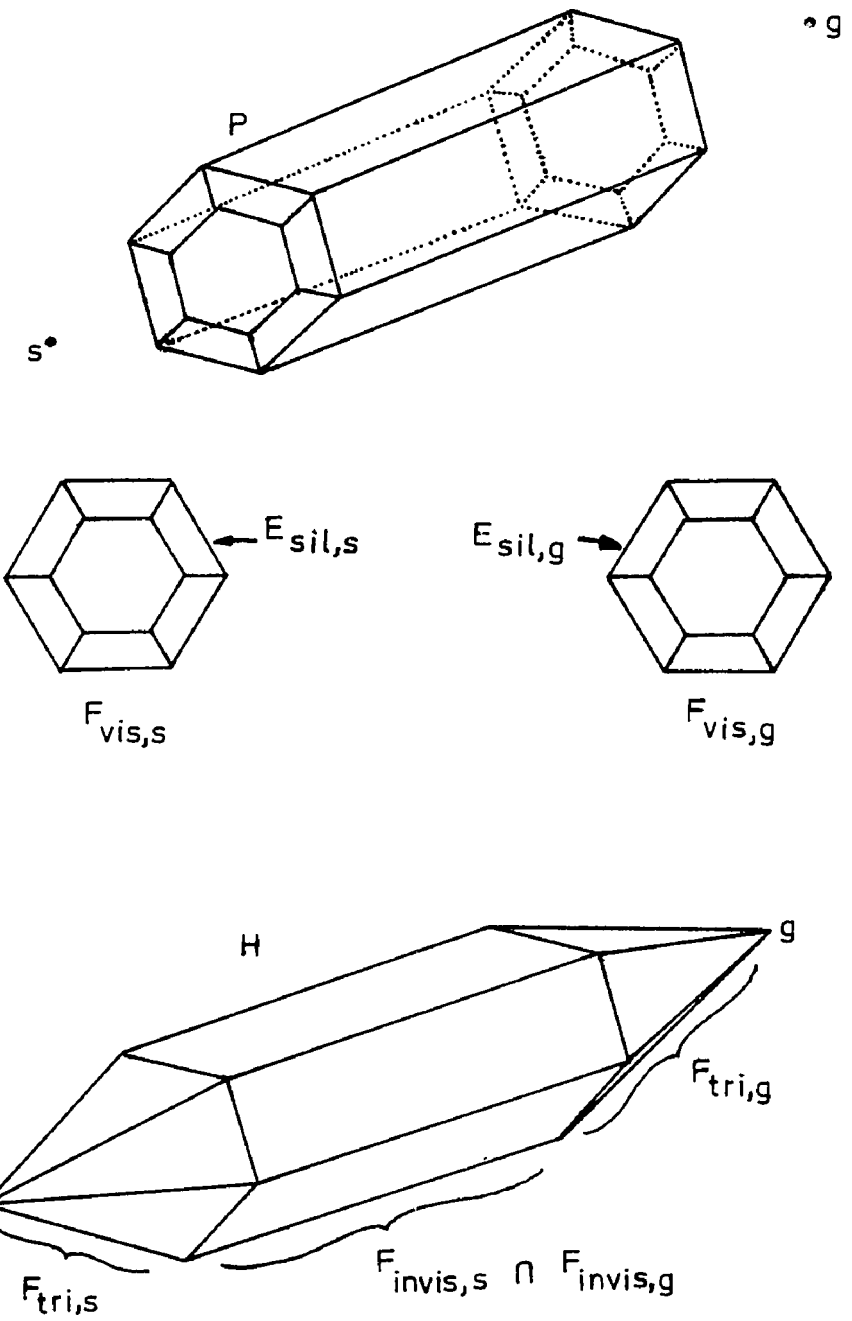


Figure 3.8 Demonstration of how EXTERIOR FINDPATH works via silhouettes.

4. TWO VORONOI-BASED TECHNIQUES

This chapter has three sections. In section 4.1 we introduce an extension of Voronoi diagrams on the boundary of a convex polyhedron and give an algorithm to compute it. Similarly, in section 4.2 we introduce the partitioning of the free space around polyhedra into regions. Section 4.3 evaluates other 3-dimensional FINDPATH algorithms of similar nature. This chapter is based on work reported in [33] and [34].

4.1 Partitioning the Boundary of a Convex Polyhedron

Consider the following variant of BOUNDARY FINDPATH:

> BOUNDARY FINDPATH (locus)
> INSTANCE: Same as in BOUNDARY FINDPATH except that g is not given. However, it is guaranteed that, when specified, g will be on $\mathrm{bd}P$.
> QUESTION: Preprocess P so that for any specified $g \in \mathrm{bd}P$ the s-to-g shortest path is computed efficiently.

A practical case which would benefit from BOUNDARY FINDPATH (locus) is as follows. Consider a truck on the surface of a mountain modeled as a convex polyhedron, say, a pyramid. Suppose that the truck is required to carry material from a fixed location on the surface to several points, say, construction sites. Then, it is reasonable to compute the shortest routes for the truck (approximated as a point) more efficiently than can be achieved by repeated applications of BOUNDARY FINDPATH for each specified destination.

This is a powerful paradigm of computational geometry known as the *locus approach* and is studied in Overmars[84]. In solving a problem using this approach, we are allowed to spend some initial effort (i.e., preprocessing) to construct a data structure which will let us answer future requests (i.e., queries) quickly. To be effective, this assumes two things. First, the number of the query points must be large to validate such an initial effort. Second, the data structure must embody succinctly the locus of the required solution and must enjoy the existence of a fast search procedure to retrieve it.

We shall now summarize one such data structure suitable for solving BOUNDARY FINDPATH (locus). The data structure is known as the *Voronoi diagram* and was first introduced to computational geometry by Shamos[101]. Let $S = \{x_1, \cdots, x_n\}$ be a subset of R^2. For $1 \leqslant i \leqslant n$ let

$$\mathrm{regn}x_i = \{y : \mathrm{d}(x_i, y) \leqslant \mathrm{d}(x_j, y) \text{ for all } j\}$$

be the *Voronoi region* of point x_i. The Voronoi diagram of S, denoted by vor$S$, partitions the plane into card$S = n$ regions, one for each member of S. The (open) Voronoi region of point x_i consists of all points of R^2 closer to x_i than any other point of S. For $1 \leqslant i, j \leqslant n$, letting $H_{ij} = \{y : d(x_i, y) \leqslant d(x_j, y)\}$ (the halfspace defined by the perpendicular bisector of $x_i x_j$), it is seen that for $i \neq j$, reg$x_i = \bigcap H_{ij}$. Thus, regx_i is a convex polygonal region and vor$S$ is equal to the union of the boundaries of all regx_i. For every vertex x of vor$S$ there are at least three points x_i, x_j, x_k in S such that $d(x, x_i) = d(x, x_j) = d(x, x_k)$. The Voronoi diagram for a set of n points has at most $2(n-2)$ vertices and $3(n-2)$ edges[73].

The Voronoi diagram of S can be computed in $O(n \log n)$ time [100] and this is optimal with respect to a wide range of computational models[59]. Unfortunately, practical Voronoi programs which are both fast and reliable are difficult to write mainly due to the special cases in the diagram that are to be handled precisely. Among the published algorithms, those by Avis and Bhattacharya[6], Lee[64], and Guibas and Stolfi[42] seem to be more promising for practical use. On the other hand, it is possible to construct slower implementations which handle special cases without much effort.

We now summarize how to search Voronoi diagrams in logarithmic time in the number of the edges, n, of the diagram, i.e., we cite methods which let one find the point $x \in S$ such that for a query point $y \in R^2$, $d(x, y)$ is minimum. The search methods we shall review are more general than searching Voronoi diagrams in that they are based on searching a *planar subdivision*, i.e., a straight-edge embedding of a planar graph. The underlying problem is generally known as *planar point location* in computational geometry. Let us call a subset of the plane *monotone* if its intersection with any line parallel to the y-axis is a single interval (possibly empty). A subdivision is monotone if all its regions are monotone. Mehlorn[73] shows that a *simple* planar subdivision (a subdivision with only triangular faces) can be searched in time $O(\log n)$ after spending preprocessing time $O(n)$ and storage space $O(n)$. He also gives the following property to show that the last two bounds apply to *general* subdivisions also, with a small penalty in preprocessing time:

LEMMA 4.1 If the searching problem for simple planar subdivisions with n edges can be solved with search time $O(\log n)$, preprocessing $O(n)$, and space $O(n)$, then the searching problem for general subdivisions with n edges can be solved with the same search time and space but preprocessing $O(n \log n)$. If all faces of the generalized subdivision are convex then $O(n)$ preprocessing is sufficient.

Proof. Mehlorn[73]. (Lee and Preparata[62] also show that an arbitrary subdivision with n edges can be refined to a monotone subdivision having at most $2n$ edges in $O(n \log n)$ time.) □

Now we shall review some planar point location algorithms in the literature which either achieve these bounds or come close. Dobkin and Lipton[18] were the pioneers to obtain an $O(\log n)$ query time but they use $O(n^2)$ space. Preparata[89] modifies their method to prove that $O(n \log n)$ space is sufficient. His solution is implementable. In an important paper Lee and Preparata[62] give an algorithm which is based on the construction of separating chains. Their algorithm achieves $O(n \log n)$ preprocessing and $O(n)$ space yet has a query time of $O(\log^2 n)$. The constants hidden in these expressions are small and make their algorithm practically useful. (Their algorithm works for monotone subdivisions only.) Edelsbrunner and Maurer[22] give a space-optimal solution which works for general subdivisions (and even families of nonoverlapping subdivisions). Their query time is $O(\log^3 n)$. Lipton and Tarjan[68, 69] give a method with $O(\log n)$ query time and $O(n)$ preprocessing and space (and thus optimal in all respects). Although based on a brilliant graph separator theorem which has many far-reaching consequences, they admit that their method is of only theoretical interest because of the serious implementation difficulties. Kirkpatrick[58] gives another method with the same bounds. His method builds a hierarchy of subdivisions and seems to be implementable. Finally, Edelsbrunner et al[24] give a substantial improvement of the technique of Lee and Preparata and again attain the optimal bounds in all three respects. The importance of their method is that it seems to admit an efficient implementation along with extensibility to subdivisions with curved edges.

Now we are ready to present our algorithm for BOUNDARY FINDPATH (locus). In the following we show the partitioning for only a face of P (other than F_s) which we shall denote by F_g. To partition bd$P$, we apply the following algorithm for each face. (Note that no partitioning is necessary for F_s due to convexity.)

Algorithm BOUNDARY FINDPATH (locus): Preprocessing
1. Find all simple face visit sequences between F_s and F_g using *Fgraph* of P.
2. For each face visit sequence found in step 1, compute the image of s with respect to the planar development which starts with face F_g and ends with F_s. (Note that we are *not* required to compute the whole development, but just the image point.)
3. Compute the Voronoi diagram of the image points calculated in step 2 using standard Voronoi programs mentioned above. It is required that with each image point (Voronoi center) we store the face visit sequence which is used to arrive it.
4. Clip the diagram obtained in step 3 with respect to "window" F_g so as to preserve only those parts of it within the polygon F_g. (Any of the standard graphics algorithms such as the one due to Sutherland and Hodge[28] can be used for clipping.)

End

Assume that for a given P, the above preprocessing is carried out for all faces (except F_s). Thus, we have a family of planar subdivisions for each face such that for each region of a given subdivision we know the ordered sequence of faces to be developed to the plane if g specified within that region. More specifically, we can apply the following algorithm when we are queried with a new g:

Algorithm BOUNDARY FINDPATH (locus): Querying
1. Compute the face F_g holding a given g. If $F_g = F_s$ then the shortest path is trivially sg.
2. Using standard planar point location algorithms mentioned above locate g inside the planar subdivision belonging to F_g.
3. Since we stored the face development sequence used to arrive this region we can now use it to compute the s-to-g shortest path. Note that there may be cases where g will be shared by at least two regions and thus there will be at least two shortest paths each of which is obtained via a different development sequence.
End

Figure 4.1 shows the Voronoi partitioning on a face of a cube using the above algorithm. This figure was drawn by SP. It is noted that in figure 4.1(a) the given face is partitioned into 4 regions whereas in figure 4.1(b) this number becomes 6. This effect was obtained by simply moving the source to another location on the source face.

In general, the number of regions a given face F_g is partitioned by this algorithm is expected to be small. An informal argument for this is as follows. Consider the images of s in the plane of F_g. If the face visit sequence for a particular image is long then the image will usually end up in a point farther away from F_g compared to another image obtained by a shorter visit sequence. Thus, only a small portion of the possible face visit sequences between F_s and F_g can render images in the plane close to F_g and contribute to the Voronoi diagram on it.

4.2 Partitioning the Free Space around Polyhedra

The inspiration for the work to be described in this section is Franklin's extension of Voronoi diagrams in the plane in the presence of barriers (line segments)[29]. Here he generalizes the Voronoi concept by allowing opaque barriers that a shortest path must avoid. With this environment, a shortest path will always be a polygonal path through the free space bending at the endpoints of the barriers. The ordered list of endpoints that a shortest path

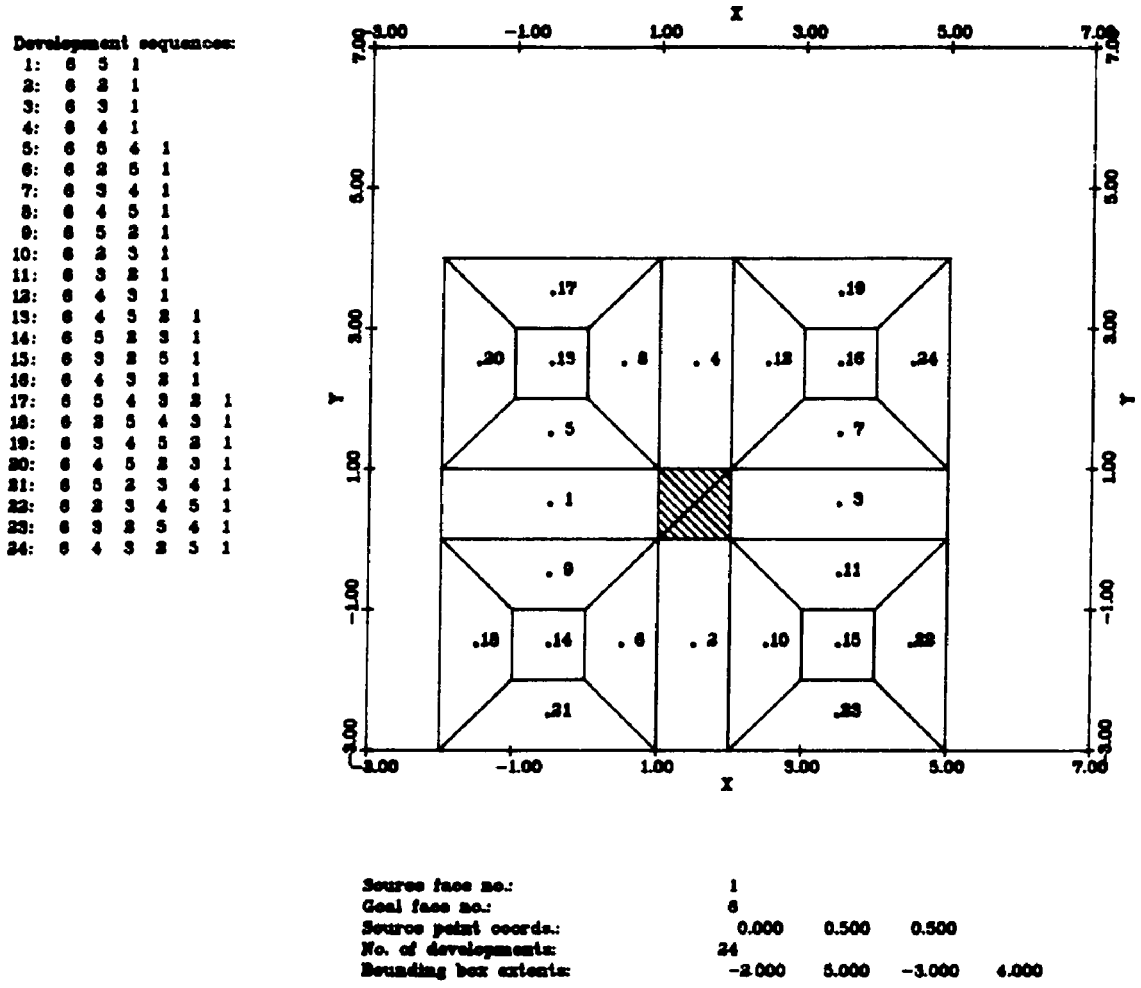
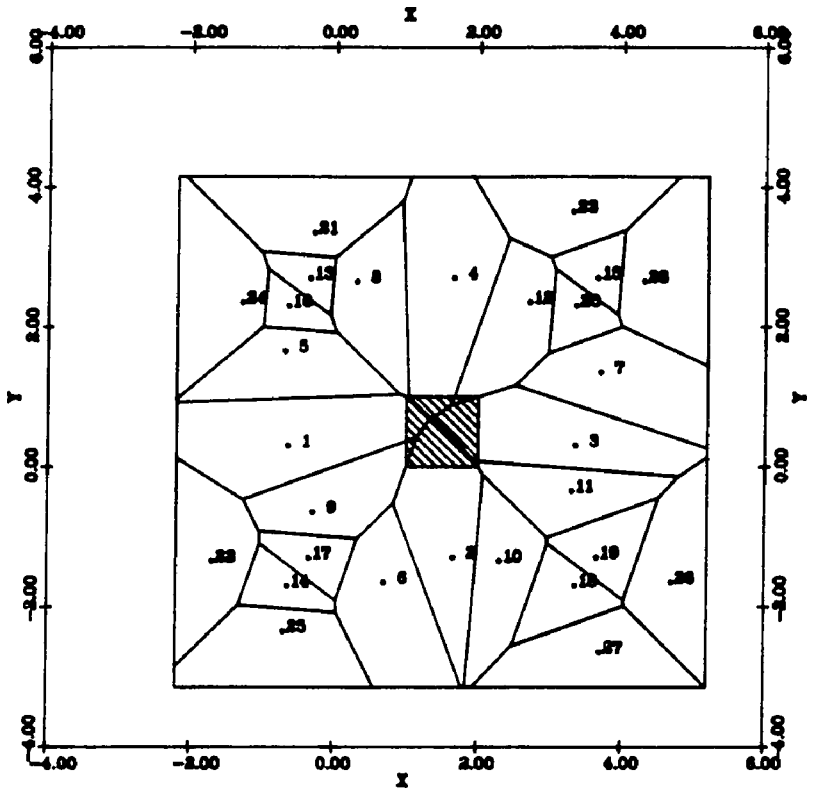


Figure 4.1 Demonstration of BOUNDARY FINDPATH (locus) on a face of a cube. In figure 4.1(a) there are 4 regions on the goal face (the shaded polygon) as a result of Voronoi partitioning. Figure 4.1(b) shows the effect of moving the source on the source face to another location.

Development sequences:

1:	6	5	1			
2:	6	2	1			
3:	6	3	1			
4:	6	4	1			
5:	6	5	4	1		
6:	6	2	5	1		
7:	6	3	4	1		
8:	6	4	5	1		
9:	6	5	2	1		
10:	6	2	3	1		
11:	6	3	2	1		
12:	6	4	3	1		
13:	6	5	4	3	1	
14:	6	2	5	4	1	
15:	6	3	4	5	1	
16:	6	4	5	2	1	
17:	6	5	2	3	1	
18:	6	2	3	4	1	
19:	6	3	2	5	1	
20:	6	4	3	2	1	
21:	6	5	4	3	2	1
22:	6	2	5	4	3	1
23:	6	3	4	5	2	1
24:	6	4	5	2	3	1
25:	6	5	2	3	4	1
26:	6	2	3	4	5	1
27:	6	3	2	5	4	1
28:	6	4	3	2	5	1



Source face no.:	1				
Goal face no.:	6				
Source point coords.:	0.000	0.300	0.650		
No. of developments:	28				
Bounding box extents:	-2.200	5.200	-3.150	4.150	

passes through is called its *contact list* . Franklin's construction works as follows. Regions are constructed so that every goal point in a region has a shortest path to the source with *identical* contact lists. Each region will have an *associated vertex* which is defined as the first vertex on its contact list. When a new barrier is introduced to the environment, two halflines extending from its endpoints are added. These boundaries are called *shadow lines* and are the projections of the endpoints by the associated vertex of the region it is in. A second type of boundary is added to separate the two new regions created. This will in general be a hyperbola and is called a *ridge curve* . A goal on a ridge curve has two shortest paths to the source. If a boundary extends to infinity nothing interesting happens. However, sometimes a boundary intersects a barrier or another boundary. In the former case, the boundary is *cut* (i.e., stops at or blocked by the barrier). In the latter case, when two boundaries hit each other they join to create a *junction* .

We now give an example to illustrate Franklin's partitioning. The reader is referred to Verrilli[112] who presents a rather complete implementation and gives several interesting computer-generated examples.

EXAMPLE In figure 4.2. there are two barriers ab , cd in the plane and s is given as shown. In this case the plane is partitioned into five regions. R_ϕ holds the goals directly reachable (visible) from s . R_a holds the goals which cause a shortest path to bend at endpoint a of ab . R_b holds the goals which cause a shortest path to bend at b . R_d holds points which give rise to shortest paths bending at d . Finally, R_{ac} describes shortest paths bending first at a , and then at c while going from s to g . It can be easily shown that the boundaries between R_a and R_b , and R_d and R_{ac} are hyperbolic portions. All other boundaries are made of straight lines. □

A crucial property of the diagram in figure 4.2 (and of any Franklin's partitioning in 2-space around barriers) is that a bend point *acts* as a source point for a later region. (For instance, a acts as a source for the points of R_{ac} .) Thus the source is continuously "pushed back" and this is the underlying reason for all boundaries being either line segments or hyperbolic sections.

We now emulate Franklin's approach in 3-space where the regions will have the following property: all points of a given region are reached from the source after bending at the *same* edges of the obstructions in the *same* order. We shall call this problem FINDPATH (locus). Our treatment will *not* be algorithmic since we have not yet answered all the questions posed by this problem. In this environment, it is seen that the analogous constructs to those outlined above will be *associated edge* , *shadow plane* , and *ridge surface*. A *contact list* will now hold the ordered set of edges instead of endpoints. We shall use the terms *cut* and *junction* without modification although now they

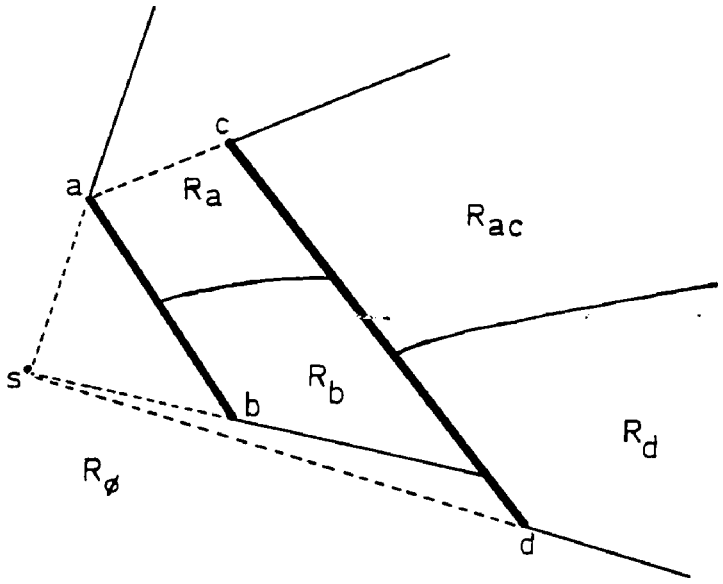


Figure 4.2 Franklin's partitioning in 2-space in the presence of linear barriers.

refer to surfaces instead of curves.

Let $a, b, c \in R^3$ be noncolinear points. We shall refer to the opaque triangle abc by T. Let s be any point outside aff$T$. We shall start with the following simple case: partition the space into regions such that if a new $g \in R^3$ is specified, then we can tell if g can directly be reached from s, and if not, the edge of T where an s-to-g shortest path bends.

When g is outside the infinite *frustum* obtained by subtracting the finite pyramid described by basis T and apex s from the infinite pyramid described similarly, the shortest path is sg itself. Thus, one of the regions, R_Φ contains all points $g \in R^3$ such that $sg \cap T = \Phi$. (In other words, R_Φ is the set of all visible points from s.) If g is not in R_Φ then three possibilities exist. For all $g \in R_{bc}$ the shortest path bends at bc. For all $g \in R_{ca}$ it bends at ca. Finally, for all $g \in R_{ab}$ it bends at ab. The edges bc, ca, and ab are the associated edges of regions R_{bc}, R_{ca}, and R_{ab}, respectively. Figure 4.3 shows these regions. Intersections of R_Φ with these three regions are the shadow planes for this example.

Now we shall compute the intersections of the pairs of regions. (These will be the ridge surfaces.) For clarity, we take s as the origin of the coordinate system without loss of generality. We shall compute the intersection of regions R_{ab} and R_{bc} and generalize it to the other two cases. If $g \in R_{ab} \cap R_{bc}$ then there exists a shortest path to g either via ab or bc. Label the bend points of this shortest path with ab (resp. bc) by d_{ab} (resp. d_{bc}). These points should satisfy:

$$d(s, d_{ab}) + d(d_{ab}, g) = d(s, d_{bc}) + d(d_{bc}, g) \qquad (1)$$

The left (resp. right) hand side of (1) is equal to $d(s, g_{ab})$ (resp. $d(s, g_{bc})$) where g_{ab} (resp. g_{bc}) is the point obtained by rotating g about affab (resp. affbc) until it coincides with affsab (resp. affsbc) and is in the opposite half-plane compared to s. In the sequel, we shall call any such point obtained by rotating a point about an axis an *image* point. If p is a point in 3-space and p' is its image due to rotation by an angle Θ about axis u passing through the origin then it is well-known from graphics that:

$$p' = <p, u> + (p - <p, u> u)\cos\Theta + (u \times p)\sin\Theta \qquad (2)$$

Applying (2) for the image of g about affab, we obtain:

$$g_{ab} = a + f\, u_{ab} + (ag - f\, u_{ab})\cos\alpha + (u_{ab} \times ag)\sin\alpha \qquad (3)$$

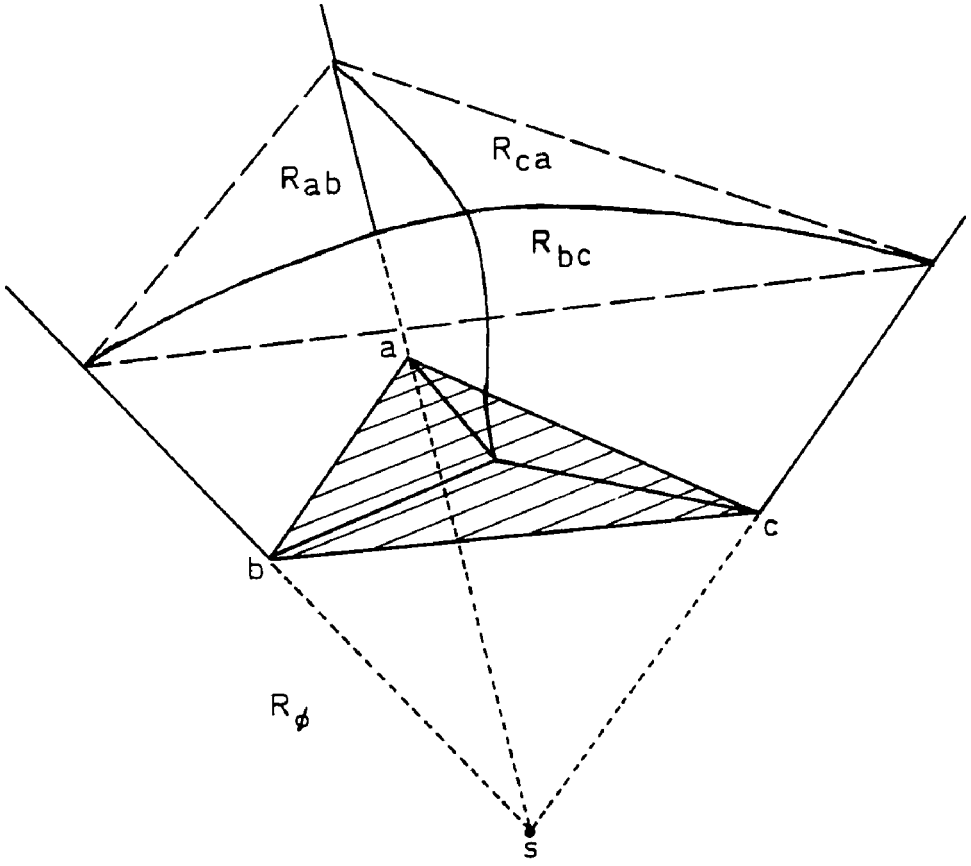


Figure 4.3 Partitioning the space in the presence of a solid triangle.

where α is the dihedral angle between affsab and affgab, $u_{ab} = ab \,/\, \mathrm{d}(a,b)$, and $f = <ag, u_{ab}>$. After some simplifications in (3), which are given in full in Franklin and Akman[34], the following expression for g_{ab} is obtained:

$$\mathrm{d}^2(g_{ab}) = \mathrm{d}^2(a) + \mathrm{d}^2(a,g) + 2(<ag, u_{ab}><a, u_{ab}> + \mathrm{d}(ag \times u_{ab})\mathrm{d}(a \times u_{ab})) \quad (4)$$

To have a geometric interpretation of (4), we expand the scalar and vector products and simplify the resulting expression to get:

$$\mathrm{d}^2(g_{ab}) = \mathrm{d}^2(a) + \mathrm{d}^2(ag) - 2\mathrm{d}(a)\mathrm{d}(ag)\cos(\alpha_1 + \alpha_2) \quad (5)$$

Here α_1 and α_2 are the angles $<gab$ and $<sab$, respectively (figure 4.4). Thus (5) is a statement of the cosine theorem on triangle sag_{ab}.

Up to this point, we found an equation (i.e., (4)) that gives $\mathrm{d}^2(g_{ab})$ in terms of known quantities a and u_{ab}, and unknown g. The equations for $\mathrm{d}^2(g_{bc})$ and $\mathrm{d}^2(g_{ca})$ are found completely analogously with obvious modifications.

To compute the intersections of R_{ab} and R_{bc} we solve the equation $\mathrm{d}(g_{ab}) - \mathrm{d}(g_{bc}) = 0$, or equivalently, $\mathrm{d}^2(g_{ab}) - \mathrm{d}^2(g_{bc}) = 0$. In [34] it is shown that the intersection surfaces are in general ternary (in x, y, z, the coordinates of g) quartics which may degenerate to planes in some cases.

If we want to partition the space behind a solid polygon instead of a triangle then we are required to compute all the potential boundaries between the pairs of regions. It is obvious that the intersections of the regions with the polygon (or the triangle in the previous case) are made of straight edges. In fact the subdivison of the polygon by these edges can be found more simply:

LEMMA 4.2 Let P be a convex polygon with vertices $v_1, \cdots, v_n$ and s a point outside aff$P$. The invisible side of P to s is partitioned into at most n convex regions (each completely containing an edge of P) such that for a goal g specified inside one of the convex regions the s-to-g shortest path is via the associated edge of this region.

Proof . We give a constructive proof which in fact is based on the technique of BOUNDARY FINDPATH (locus) (figure 4.5). Rotate s about affv_1v_2, affv_2v_3, $\cdots$ until it is coincident to aff$P$ and always on the opposite side of a particular edge with respect to int$P$. This is basically a planar polygonal schema of the pyramid with basis P and apex s in aff$P$. Denote the n images obtained in this way by $s_{12}, s_{23}, \cdots$ and construct their Voronoi diagram. When clipped by window P the diagram partitions P into at most n regions which are necessarily convex since P is convex. □

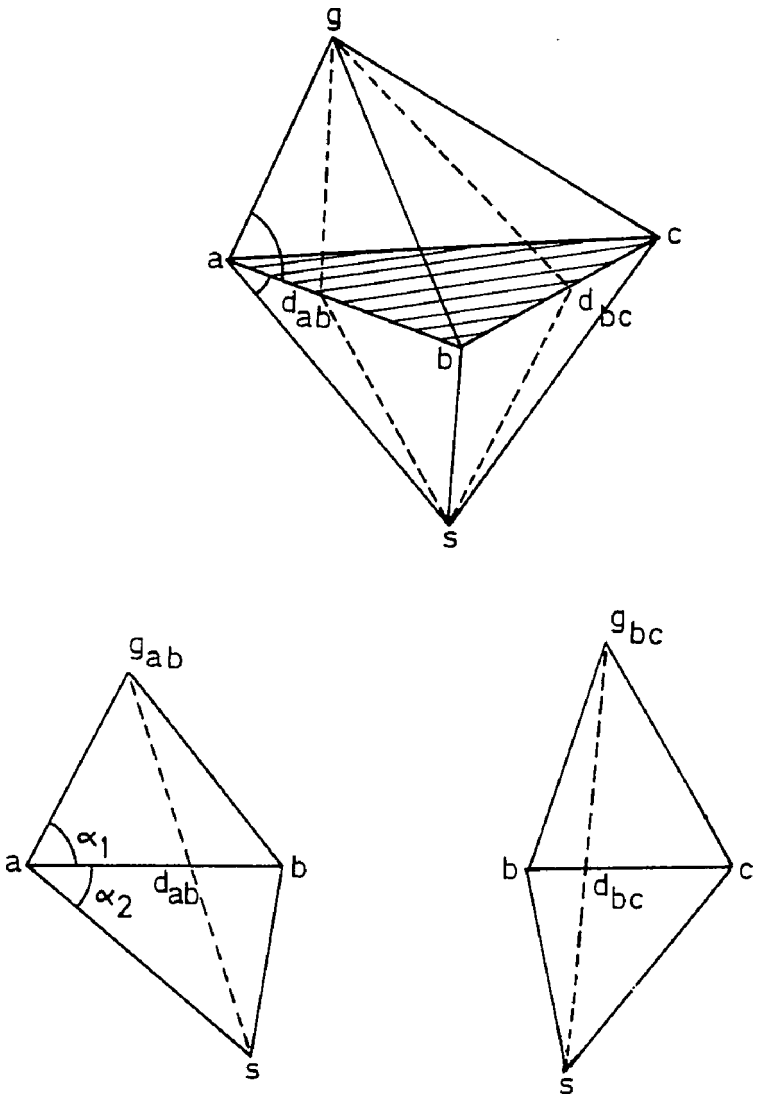


Figure 4.4 Computation of the images g_{ab} and g_{bc} of g.

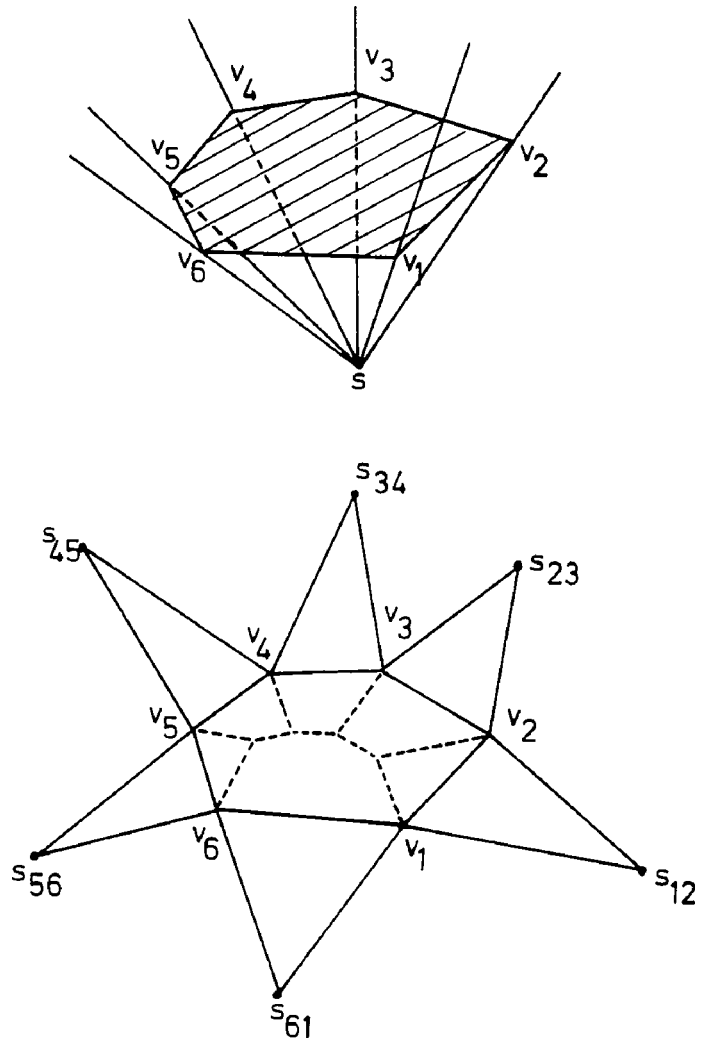


Figure 4.5 Voronoi partitioning on the back face of an opaque polygon.

The extension of the technique outlined for FINDPATH (locus) to several obstacles creates difficult problems. In this case we cannot find a similar property to that of Franklin's partitioning mentioned above, i.e., the regions of the partition no more stay as quartics but grow in degree for each new obstacle causing new regions. We shall now imagine that for a given s we managed to compute a subdivision of the space into n regions bounded either by planes or quartic (or higher-order) curves using the above approach. With each region we assume the existence of a stored label which is simply equal to the contact list for this region. If we are given a new point g, we should first locate the region that g is in, and then use the contact list for this region to compute the s-to-g shortest path. Since the latter operation can be done using OPTIMAL EDGE VISIT we shall concentrate on the former operation.

Let S be a family of n planes in R^3. Dobkin and Lipton[18] show how to represent S in a polynomial amount of space so that whether a given point belongs to any of the given planes can be tested in $O(\log n)$ time. If we approximate the boundaries of our regions with planes then their algorithm is applicable. Chazelle[13] generalizes this for the case where S is a family of algebraic varieties. His algorithm is an adaptation of Collins' quantifier elimination procedure[16]. Let d be a positive constant and let $S=\{Q_1, \cdots, Q_n\}$ be a set of polynomials of degree at most d in three variables with rational coefficients. He considers the problem of preprocessing S so that, for any $(x,y,z) \in R^3$, the predicate "There exists an i, $1 \leqslant i \leqslant n$, such that $Q_i(x,y,z)=0$" can be evaluated efficiently. This problem (and its generalization to spaces higher than R^3) is known as *spatial point location*. If the predicate is true then *any* of the indices i for which $Q_i(x,y,z)=0$ is reported. Otherwise, (x,y,z) is seen to lie in a region over which each Q_i has a constant sign. Assuming that these regions have associated labels, his algorithm retrieves the label corresponding to the region containing (x,y,z). The algorithm uses $O(n^{512})$ preprocessing time (and space) and computes the above predicate in $O(\log n)$ time. Although it meets our requirements completely, the possibility of a practical implementation of Chazelle's method is highly questionable.

4.3 Critique of Recent Algorithms in 3-Space

Sharir and Schorr's work[102] is probably the most detailed study on shortest paths. They mainly consider the case of BOUNDARY FINDPATH (locus) and present an algorithm which works in time $O(n)$ per query where n is the measure of complexity of the polyhedron, say the number of vertices. Their algorithm is based on the idea of "ridge" points on the object. A ridge point on the polyhedron has the property that there exists at least two shortest paths to it from the source. It turns out that the set of ridge points is a union of line segments and that the union of the vertices and the ridge points

is a closed connected set. Defining the union of the latter with the shortest paths from the source to every vertex, one partitions the polyhedron boundary into disjoint connected regions (called "peels") whose interiors are free of vertices or ridge points. The peels can be constructed in $O(n^3 \log n)$ (preprocessing) time using complicated techniques whose implementation seems extremely difficult. The size of the data structure that is created by the preprocessing step is $O(n^2)$.

O'Rourke et al[83] use most of Sharir and Schorr's ideas to extend the problem to a polyhedral surface which is no more supposed to be convex. (However, it should be orientable.) The shortest path that they calculate is only a "geodesic", i.e., it is confined to the surface and thus may not be the true shortest path. Their algorithm runs in $O(n^5)$ time. Furthermore, it does not follow a locus approach: using their algorithm on each new query (goal point) would take $O(n^4)$ time.

Mitchell and Papadimitriou[75] give an algorithm to solve this last problem in a locus setting. (It is noted that they are still computing geodesics, not true shortest paths.) Theirs is an $O(n^2 \log n)$ time algorithm for subdividing the surface of an arbitrary polyhedron (possibly of positive genus) so that the length of the shortest path from a given source to any goal on the surface is obtainable by simple point location. It has striking similarities to Dijkstra's method for shortest paths in graphs. As in our algorithm presented in section 4.1, point location is achieved in time $O(\log n)$, after which the actual shortest path is backtracked in time proportional to the number of faces that it traverses on the boundary.

5. SP - A WORKBENCH TO COMPUTE SHORTEST PATHS

This chapter has two sections. In section 5.1 we give an overview of a workbench (called SP) to compute shortest paths. Section 5.2 offers a review of the kind of shortest path computations that can be done with SP. Appendix explains SP in greater detail. This chapter and appendix are based on [31].

5.1 Overview

SP is a family of programs written in Franz Lisp[27] and Macsyma command language[40] to experiment with shortest paths in 3-space. To be able to use SP effectively a user should have a good knowledge of Lisp and Macsyma and is referred to the above manuals. Appendix (a) describes how to use SP. The most important parts of SP consist of Franz functions. These are reviewed in detail in appendix (b). Macsyma parts of SP are given in appendix (c). There are some Fortran parts of SP which implement graphical output[12]. (Appendix (d) gives a summary.)

SP was designed with the following philosophy. Let W be a workspace (e.g., a bounding box) which includes a set of polyhedral obstacles. SP is given a geometric description of the members of W and from that point on should be able to compute shortest paths inside W between given pairs of points using the algorithms presented in chapters 2 and 3, and the preprocessing strategies given in chapter 4. It is imperative that SP has some interactive graphics facilities and can supply the user with views of W so that she can have an intuitive feeling about the correctness of a particular computation (but see appendix (d)). In that sense, SP resembles to Verrilli's system[112]: it provides the user with facilities to carry out needed computations, but at the same time needs her intervention here and there. Following a rapid prototyping approach, we either simply excluded those computations which we do not currently know how to perform effectively, or reformulated them to be controlled by user advice at certain points.

Currently, one can only work with a single convex polyhedron using the Franz part of SP. There are facilities to implement BOUNDARY FINDPATH, EXTERIOR FINDPATH, and BOUNDARY FINDPATH (locus). It is also possible to implement an approximate FINDPATH algorithm for a workspace with several convex polyhedra as outlined in section 3.1 and depicted in figures 3.1 and 3.2. Using Macsyma parts of SP it is possible to compute shortest paths in a general workspace with many objects (which may be nonconvex) although this is not fully automated in the light of the combinatorial explosion that our FINDPATH algorithm has. (Nevertheless, if the user specifies the list of edges that the shortest path must visit, then the problem is

solved without much effort.) It is also possible (using Macsyma) to work on FINDPATH (locus) although this is not automated yet. (In [34] all computations were done in this way.)

5.2 Some Interesting Scenarios

The prospective user is expected to know the details of SP. Fortunately, the interactive nature of Lisp comes into play whenever one wants to inspect and/or debug the existing data structures. Working with SP is incremental in the sense that one computes things, stops and studies them (by plotting if necessary), and continues. Thus it is best to visualize SP as a sophisticated calculator tailored for shortest path computations.

SP uses simple data structures which are easy to implement and debug. Since it is built on Lisp, the list data structure is the most common in SP with a few exceptions where arrays are used. There are several global variables which exist throughout a session with SP. These will be explained below. However, since Lisp is interactive, a better way to learn about the following (and about SP in general) is to run SP on an example object and to study the created data structures.

We first describe how to solve BOUNDARY FINDPATH. Throughout this section, the reader is referred to appendix (b) to understand the details of the functions that will be mentioned. We denote functions by lower-case italics and (global) parameters by (upper-case) italics enclosed within angular brackets.

To read a polyhedron into SP, *rpoly* is used. A polyhedron file consists of three parts: blurb, vertices, and faces. (The blurb part is only for documentation purposes.) Below the format of a simple polyhedron data file (*tetra.DAT*) is shown:

```
(blurb 2)
(1 (tetrahedron))
(2 (created by akmanv august 15 1984))
(vertices 4)
(1 (0.0000000 0.0000000 0.0000000))
(2 (0.0000000 1.0000000 0.0000000))
(3 (0.8660254 0.5000000 0.0000000))
(4 (0.2886102 0.5000000 0.8165195))
(faces 4)
(1 (1 2 3))
(2 (2 1 4))
(3 (3 2 4))
```

(4 (1 3 4))

As mentioned before, this is the straightforward boundary representation for a tetrahedron. One can check whether an object is topologically and geometrically consistent by using the function *pinteg* in SP. Similarly, *pinfo* gives polyhedral statistics about an object. *Rpoly* creates the global data structures $< VERTICES >$, $< FACES >$, $< NVERTICES >$, and $< NFACES >$, the last two items being the length of the first two lists. An important property of *rpoly* is that it builds the *Fgraph* for the object as soon as the object is read into SP and stores it in $< OMEGA >$. (The function *cromega* does this at any time for the current object.) Once the polyhedron is available, to solve BOUNDARY FINDPATH we specify the faces F_s and F_g and using *yen* compute as many simple face visit sequences as required between these nodes of $< OMEGA >$. As noted in section 3.1, the number of such sequences cannot be efficiently computed a priori; hence *yen* must be run until it starts generating null sequences at which point we know that we obtained all simple sequences. Since for a large (i.e., with many faces) object the number of generated simple sequences may be large, one may want to try only a small portion of the potential sequences. *Yen* returns the sequences in list $< KPATHS >$. It is possible to use the function *cart* which computes the number of simple walks between all pairs of nodes of a graph and returns the result in matrix $< S >$. In this way we know beforehand how many sequences we can get between any two given nodes of $< OMEGA >$ without ever running *yen* .

Once we have the simple face visit sequences, we can try them one by one using *alex* . This function requires the positions of the source and the goal on faces F_s and F_g and works for a single sequence. It creates a planar development in $< UNFOLD >$ and computes the source-to-goal distance $< SRCTOGOL >$ and the images of them in the xy-plane, i.e., $< PLNSRC >$ and $< PLNGOL >$. It should be noted that all planar developments are created in the xy-plane. *Ufinteg* is a function to check whether a planar development in $< UNFOLD >$ is consistent, i.e., all polygons of it have z-coordinates very close to 0. A face visit sequence given to *alex* is stored in $< SIGMA >$. Note that *alex* does not care whether a planar development is legal or not. One can run *wunfold* after running *alex* to create a plot file which may be drawn using Fortran plot utility *drawunf*. Figure 3.5 shows some planar developments obtained in this way.

One can do the above computation automatically via *allgoals* which will compute the images of the goal for a requested number of simple face visit sequences and output only those that are legal. In this case, the global parameter $< MINPATH >$ gives the shortest paths' bend points. $< MINSIGMA >$ is the face visit sequence that gave rise to $< MINPATH >$ whereas

$<MINSTOG>$ holds the length of the shortest path.

If one wants to see a computed path on the object along with the object itself in perspective, the function *prettyplt* is used. This writes a plot file which can be drawn using the utility *drawpp* . Some examples are given in figures 5.1 and 5.2. It is noted that the function *foldbak* takes a planar development which is given in $<UNFOLD>$ and maps it into 3-space to obtain the bend points of the legal path connecting the source and the goal in $<BENDPTS>$. *Prettyplt* draws this path on the boundary of the object.

We now describe how to deal with EXTERIOR FINDPATH, i.e., when the source and the goal are outside the current object. To compute the silhouette edges of a convex polyhedron. SP first creates $<EDGES>$ and $<NEDGES>$ using the function *getedges* . (Note that the edges are implicit in the boundary representation.) Then *fassil* finds the silhouette edges with respect to a given viewpoint (source or goal) and stores them in $<SILE>$. $<CCWSILE>$ has the same information in counterclockwise order when viewed from the viewpoint. *Fassil* also creates $<VISE>$ and $<VISF>$ which respectively hold the visible edges and the visible faces from this point. The latter computations may be individually carried out by the functions *visibled* and *visiblfc*. The major function which implements EXTERIOR FINDPATH is *pyrcons* . This takes a polyhedron and two points outside it, and writes a new polyhedron which now has the source-to-goal shortest path necessarily on its boundary. *Pyrcons* uses the silhouette and the visibility functions mentioned above. The new polyhedron file will then be read and a BOUNDARY FINDPATH computation can proceed as formulated above. Figure 5.3 shows an example object created in this way from a cube.

If there are several convex polyhedra inside workspace W . the function *lseghedra* can be used to compute an approximate path as depicted in figure 3.1. *Lseghedra* finds the intersections of the given polyhedra with the source-to-goal line segment and returns a list of point pairs for each polyhedron intersecting the segment. Once these pairs are obtained, a BOUNDARY FINDPATH computation as described above can be done for each pair and its associated polyhedron. SP does not currently know how to further optimize a path obtained in this manner.

We now describe how to do the face partitioning as required by BOUNDARY FINDPATH (locus) of section 4.1. *Naivor* is a simple Voronoi program which computes the Voronoi polygons for a set of points in the xy-plane. *Vorin* writes the images of a source point for a given number of development sequences and specified source and goal faces. These images can then be processed by *naivor* to obtain the Voronoi partitioning on the goal face. Since the diagram is not finite. a clipping routine called *shclip* is used by *naivor* to

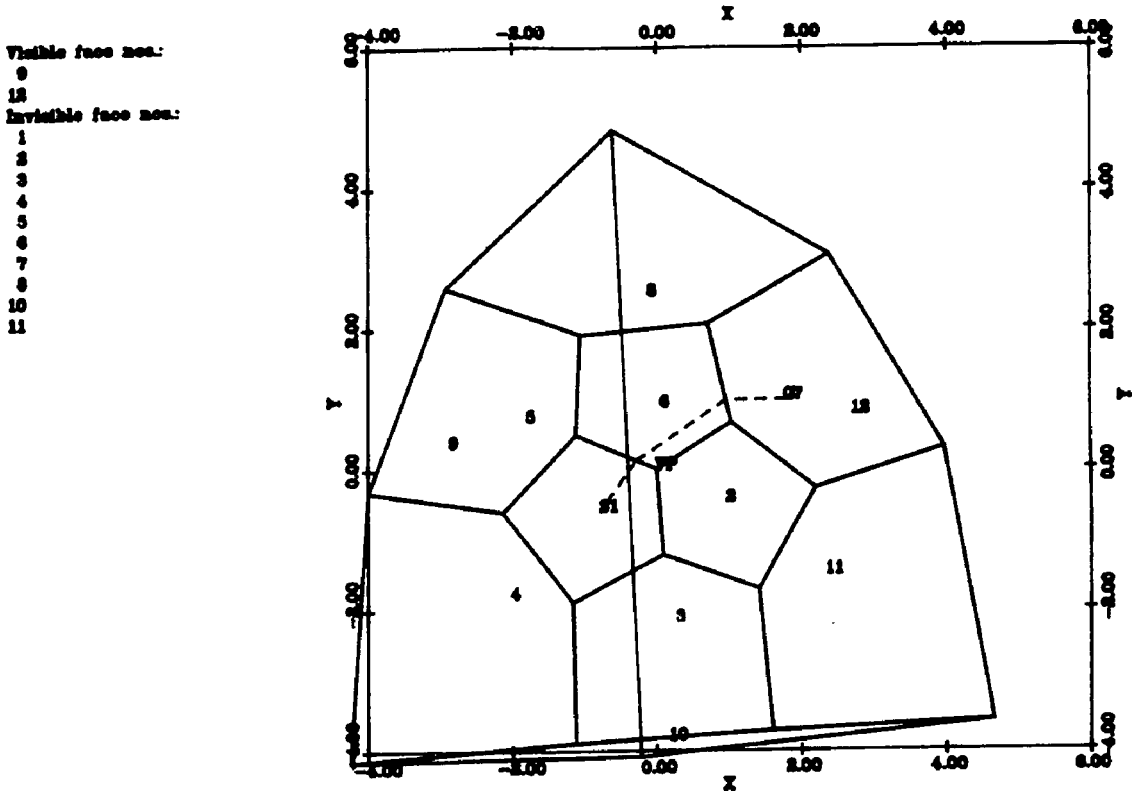
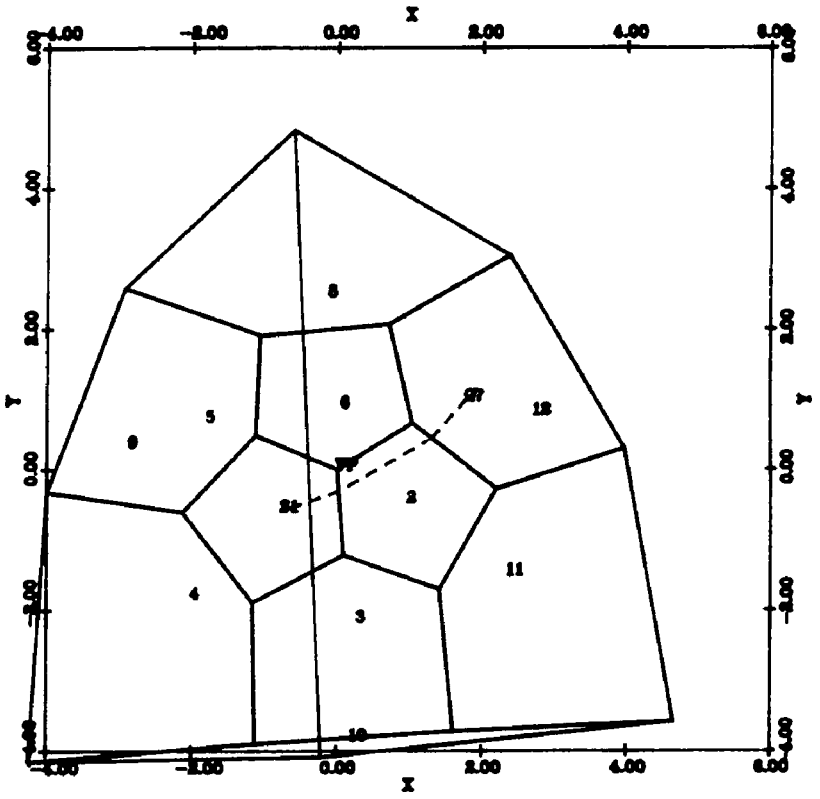
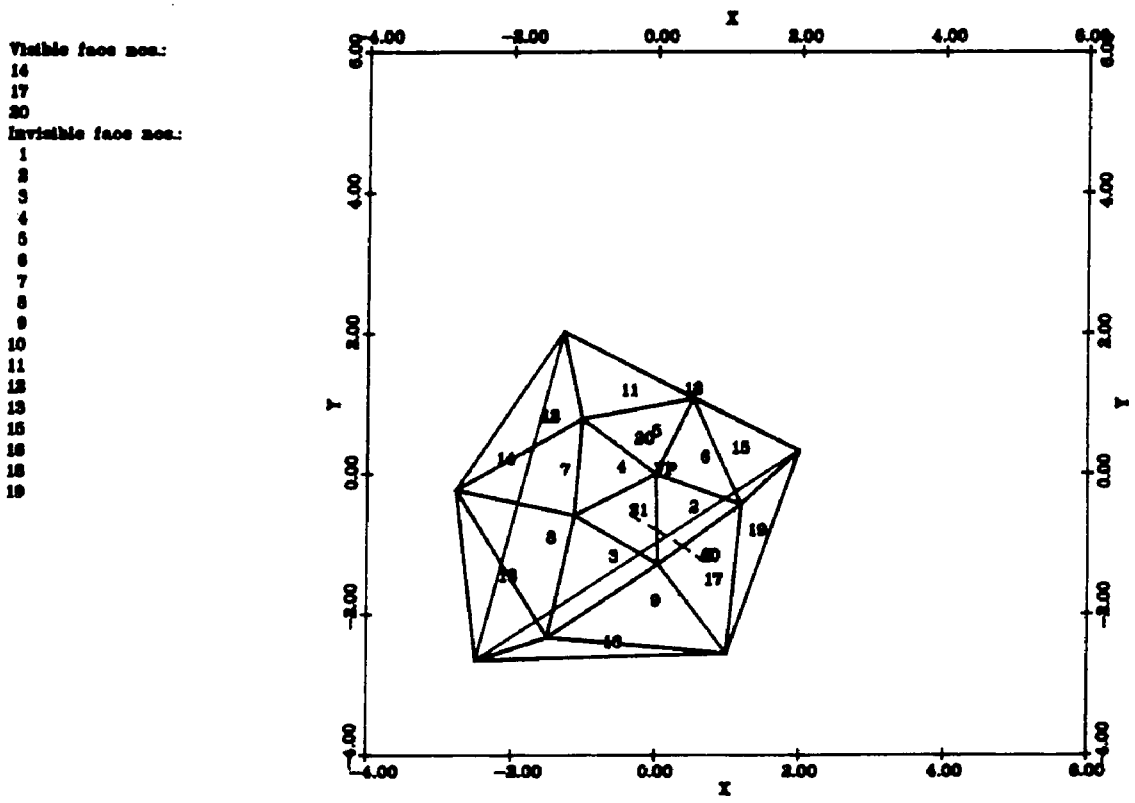


Figure 5.1 Shortest paths on the boundary of a dodecahedron.
Both (a) and (b) are shortest paths. This is a perspective view of the object as computed by SP.

Visible face nos.:
9
12
Invisible face nos.:
1
2
3
4
5
6
7
8
10
11



Viewpoint:		1.500	0.800	2.500	
Source face no.:		1			
Goal face no.:		7			
Source point:		0.688	0.500	0.000	
Goal point:		-0.118	-0.086	1.811	
Face development sequence:	1 2 7				
Shortest path length:		2.818			
Shortest path bend points:		0.688	0.500	0.000	
		0.000	0.376	0.000	
		-0.498	-0.086	0.998	
		-0.118	-0.086	1.811	
Bounding box extents:		-4.000	4.000	-4.000	4.000



Viewpoint:	0.300	0.800	2.000	
Source face no.:	1			
Goal face no.:	10			
Source point:	0.288	0.500	0.000	
Goal point:	-0.275	0.936	0.504	
Face development sequence:	1 2 10			
Shortest path length:	1.000			
Shortest path bend points:	0.288	0.500	0.000	
	0.000	0.867	0.000	
	-0.215	0.833	0.153	
	-0.275	0.936	0.504	
Bounding box extents:	-3.500	3.500	-3.500	3.500

Figure 5.2 A shortest path on the boundary of an icosahedron. This was computed by SP.

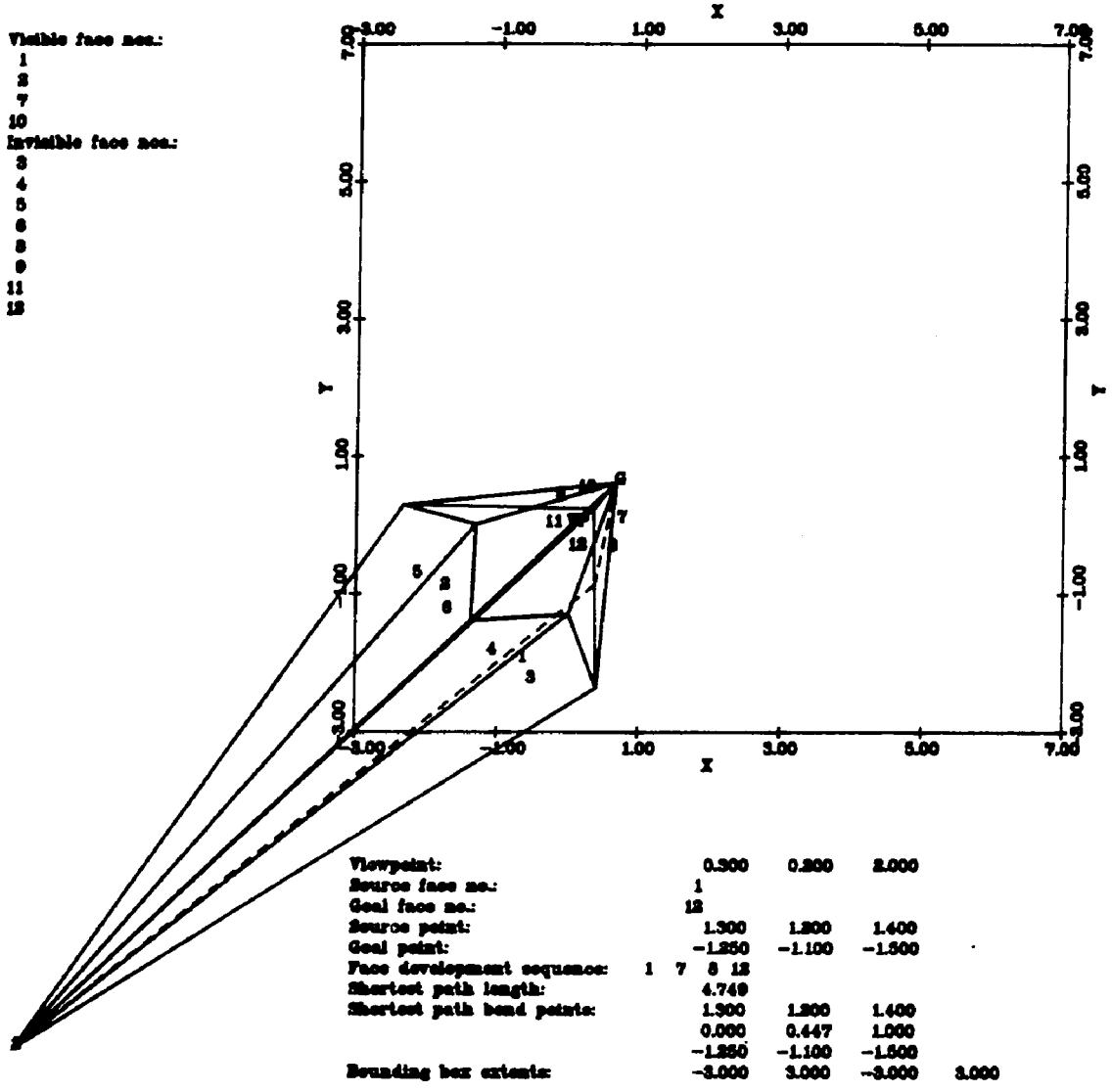


Figure 5.3 A shortest path around a cube. This was computed by SP after computing the new object and then applying BOUNDARY FINDPATH on it.

clip it against a rectangular window. Currently, there does not exist a planar point location routine in SP to compute the region of the clipped diagram for a specified new goal point. (This is not essential and can be incorporated easily.) If one uses the function *pltvor* after running *vorin* . the Voronoi polygons are written to a plot file which can be drawn using *drawvor* utility. Figure 5.4 was generated in this manner.

Finally, we mention some lower level functions that may be used at any point during an interaction with SP. *Dijk* computes the shortest paths in a graph. *Faceqn* gives the face equation of a given face of the current object. *Diang* returns the dihedral angle between two faces. *Centroid* returns the centroid of a given face and is useful when one is trying to find a point inside a face. Conversely, *inside* checks if a given point is inside a given face. The functions *vadd* , *vdiff*, *vtimes* , *vquotient* implement the corresponding 3-dimensional vector functions. *Cross* , *dot*, *magnitude* and *distance* are some other essential functions to work with vectors. *Lsegpln* computes the intersection of a line segment with a plane whereas *plseg* 2 does the same thing for two planar line segments. *Facesof* computes a pair of faces which share a given edge. *Facerot* (resp. *objrot*) rotate a face (resp. the whole object) whereas *facexlat* (resp. *objxlat*) translate a face (resp. the whole object). Using these it is possible to create derivative objects from a given object and to generate new polyhedral data files. The function *wpoly* takes an object currently in the system and writes it to a file in the format required by *rpoly* . A rather complete matrix package incorporating *madd* , *msub* , *mexpo* , *mmul* , *mtpose* , etc. also exists.

The reader is advised to study appendix part (b) to see the other functions that are available in SP and how to use them. It must be noted that we regard SP only as a first approximation to a larger and more general software system which will act as a "Geometer's Workbench." This will entail the integration of additional geometric know-how from classical and computational geometry along with algorithm/data structure animation techniques and better graphics for visualizing complex geometric situations. It is thought that such a system may broaden the way geometry is done in the style Macsyma accomplished for algebra.

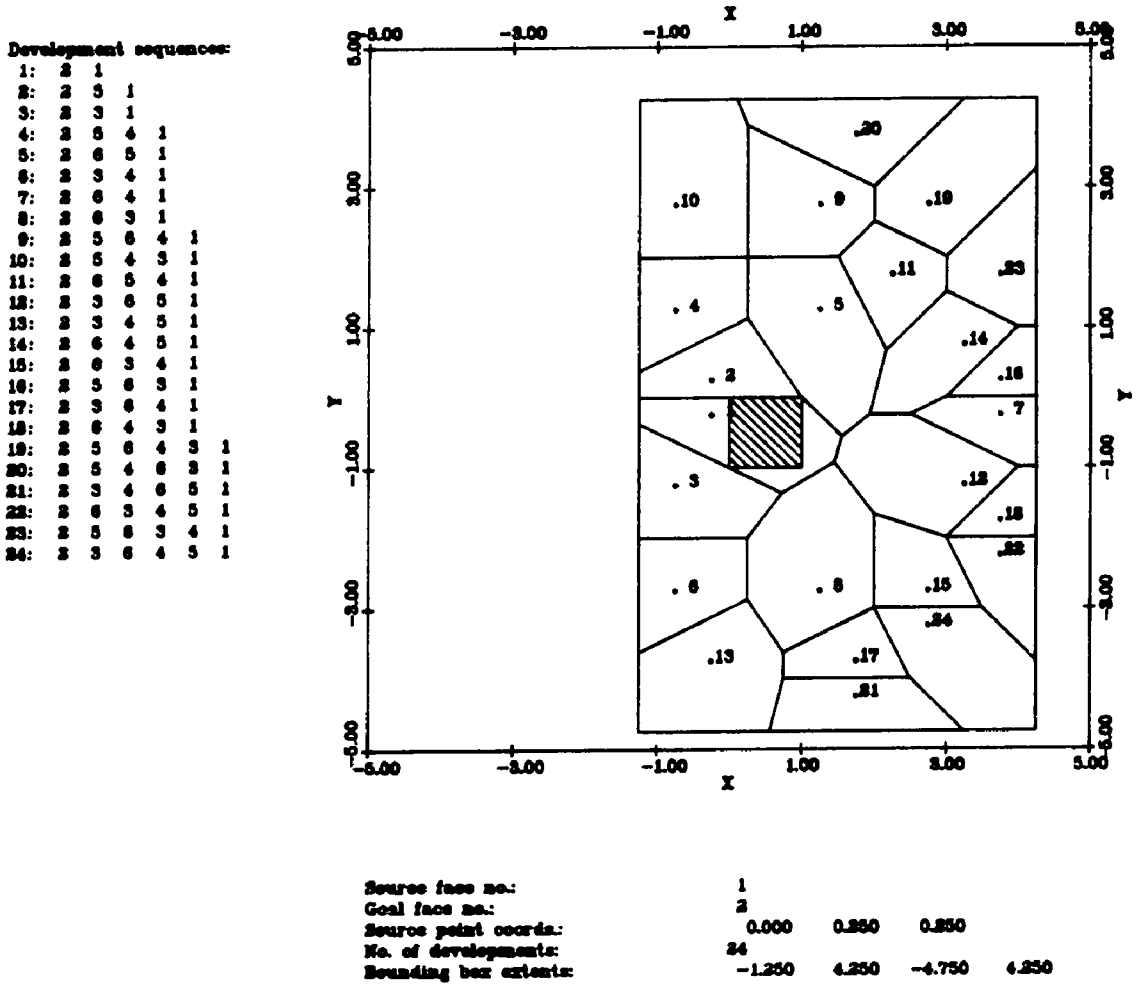
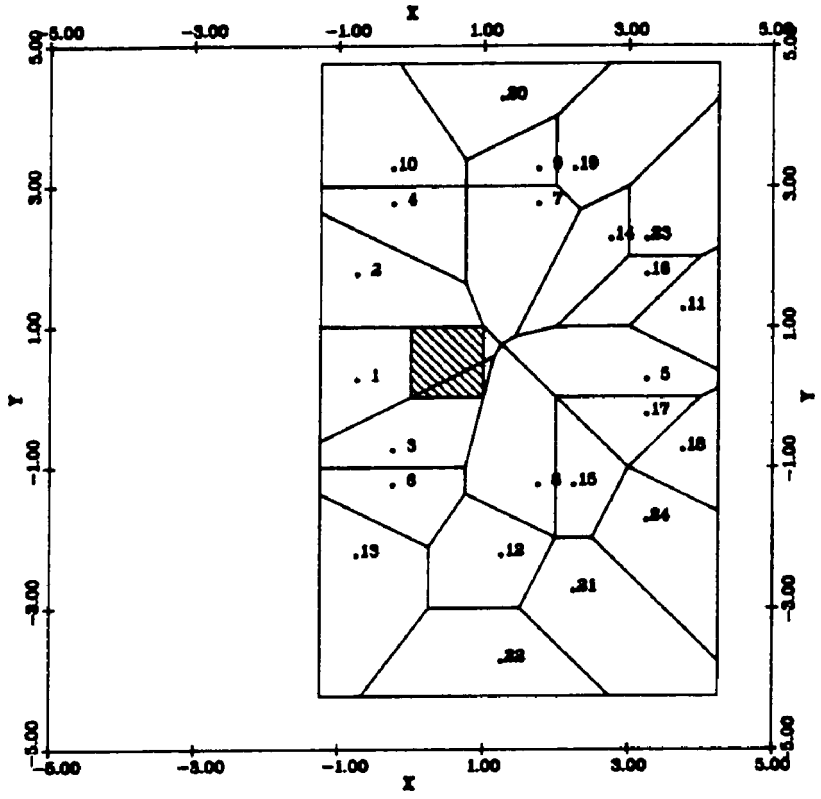


Figure 5.4 This shows the partitioning of the boundary of a cube in the presence of a source on face 1. Parts (a), (b), (c), (d), and (e) respectively show the regions induced on goal faces 2, 3, 4, 5, and 6. These figures were computed by SP.

Development sequences:

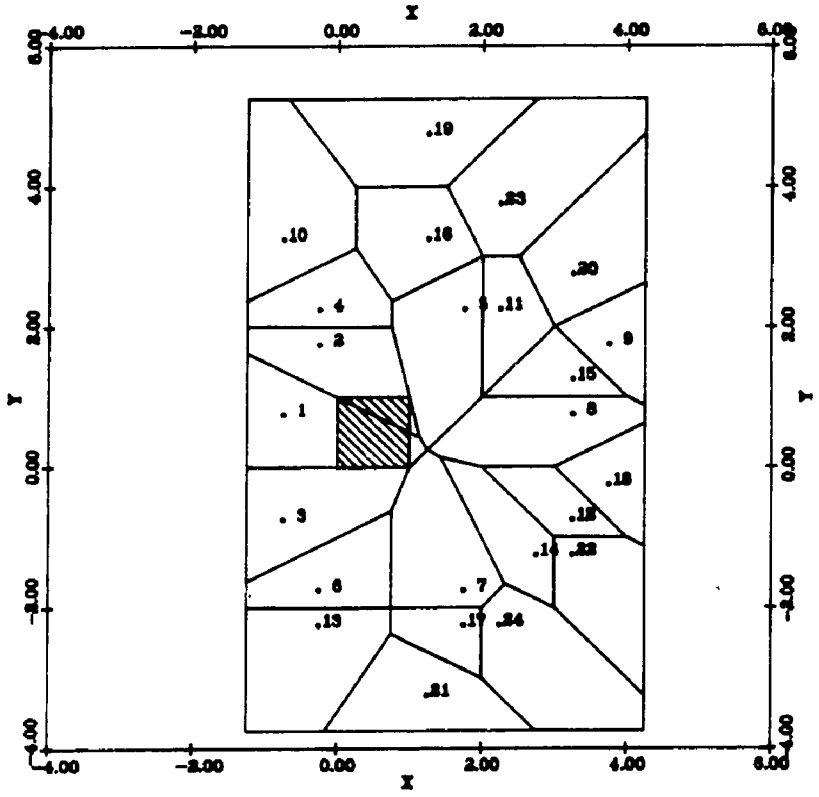
1:	3	1				
2:	3	4	1			
3:	3	2	1			
4:	3	4	5	1		
5:	3	6	5	1		
6:	3	2	5	1		
7:	3	6	4	1		
8:	3	6	2	1		
9:	3	4	6	5	1	
10:	3	4	5	2	1	
11:	3	6	5	4	1	
12:	3	2	6	5	1	
13:	3	2	5	4	1	
14:	3	6	4	5	1	
15:	3	6	2	5	1	
16:	3	4	6	2	1	
17:	3	6	5	2	1	
18:	3	2	6	4	1	
19:	3	4	6	5	2	1
20:	3	4	5	6	2	1
21:	3	2	6	5	4	1
22:	3	2	5	6	4	1
23:	3	4	6	2	5	1
24:	3	2	6	4	5	1



Source face no.:	1			
Goal face no.:	3			
Source point coords.:	0.000	0.250	0.250	
No. of developments:	24			
Bounding box extents:	-1.250	4.250	-4.250	4.750

Development sequences:

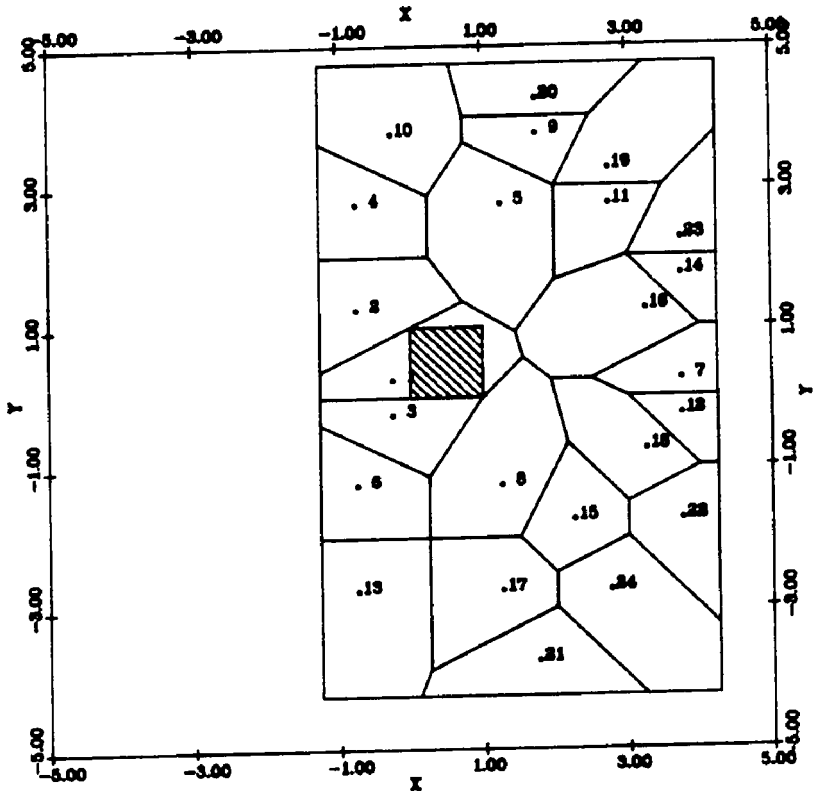
1:	4	1				
2:	4	5	1			
3:	4	3	1			
4:	4	5	2	1		
5:	4	6	5	1		
6:	4	3	2	1		
7:	4	6	3	1		
8:	4	6	2	1		
9:	4	5	6	3	1	
10:	4	5	2	3	1	
11:	4	6	5	2	1	
12:	4	3	6	5	1	
13:	4	3	2	5	1	
14:	4	6	3	2	1	
15:	4	6	2	5	1	
16:	4	5	6	2	1	
17:	4	3	6	2	1	
18:	4	6	2	3	1	
19:	4	5	2	6	3	1
20:	4	6	5	2	3	1
21:	4	3	2	6	5	1
22:	4	6	3	2	5	1
23:	4	5	6	2	3	1
24:	4	3	6	2	5	1



Source face no.:	1			
Goal face no.:	4			
Source point coords.:	0.000	0.250	0.250	
No. of developments:	24			
Bounding box extents:	-1.250	4.250	-3.750	5.250

Development sequences:

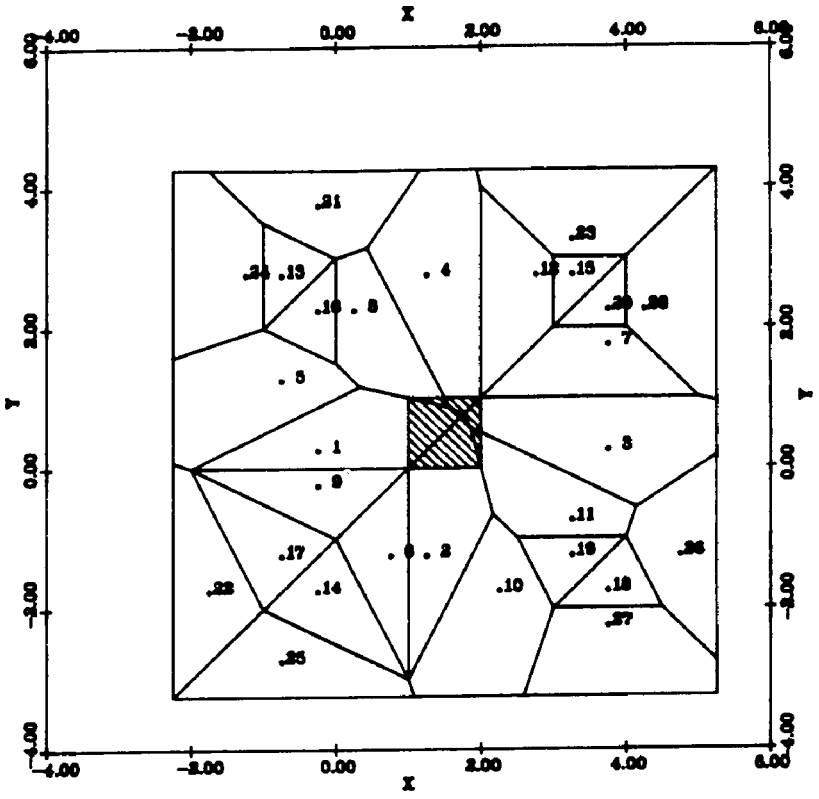
1:	5	1				
2:	5	4	1			
3:	5	2	1			
4:	5	4	3	1		
5:	5	6	4	1		
6:	5	2	3	1		
7:	5	6	3	1		
8:	5	6	2	1		
9:	5	4	6	3	1	
10:	5	4	3	2	1	
11:	5	6	4	3	1	
12:	5	2	6	4	1	
13:	5	2	3	4	1	
14:	5	6	3	4	1	
15:	5	6	2	3	1	
16:	5	4	6	2	1	
17:	5	2	6	3	1	
18:	5	6	3	2	1	
19:	5	4	6	3	2	1
20:	5	4	3	6	2	1
21:	5	2	3	6	4	1
22:	5	6	2	3	4	1
23:	5	4	6	2	3	1
24:	5	2	6	3	4	1



Source face no.:	1			
Goal face no.:	5			
Source point coords.:	0.000	0.250	0.250	
No. of developments:	24			
Bounding box extents:	−1.250	4.250	−4.250	4.750

Development sequences:

1:	6	5	1			
2:	6	2	1			
3:	6	3	1			
4:	6	4	1			
5:	6	5	4	1		
6:	6	2	5	1		
7:	6	3	4	1		
8:	6	4	5	1		
9:	6	5	2	1		
10:	6	2	3	1		
11:	6	3	2	1		
12:	6	4	3	1		
13:	6	5	4	3	1	
14:	6	2	5	4	1	
15:	6	3	4	5	1	
16:	6	4	5	2	1	
17:	6	5	2	3	1	
18:	6	2	3	4	1	
19:	6	3	2	5	1	
20:	6	4	3	2	1	
21:	6	5	4	3	2	1
22:	6	2	5	4	3	1
23:	6	3	4	5	2	1
24:	6	4	5	2	3	1
25:	6	5	2	3	4	1
26:	6	2	3	4	5	1
27:	6	3	2	5	4	1
28:	6	4	3	2	5	1



Source face no.:	1			
Goal face no.:	6			
Source point coords.:	0.000	0.250	0.250	
No. of developments:	28			
Bounding box extents:	-2.250	5.250	-3.250	4.250

6. CONCLUSION

This final chapter has two sections. In section 6.1 we summarize the contributions of this thesis. In section 6.2 we offer directions for future work and cite various open problems.

6.1 Results

As expected in a work of this sort. we now summarize the contributions of this thesis:

- We have shown that there is an algebraic flavor inherent in FINDPATH. The shortest unobstructed path connecting the source and the goal points possibly bends at some edges of the given obstacles while subtending equal entry and exit angles. The general instance of FINDPATH can then be solved using purely algebraic methods such as elimination. We then went on to demonstrate the use of the nonlinear system solvers (of a numeric nature) in solving this problem. In particular, Newton-Raphson and optimization methods were found to be effective in solving OPTIMAL LINE VISIT, a major subproblem of FINDPATH.

- We have given two original algorithms to solve FINDPATH when the source and the goal are located in a workspace which has a single convex polyhedron. In the first case (BOUNDARY FINDPATH) the given points are on the polyhedron's surface. In the second case (EXTERIOR FINDPATH) they are outside the polyhedron. These algorithms use planar developments and polyhedral visibility.

- We have described the advantage of Voronoi diagrams in two locus cases of FINDPATH. In one case, there is a single convex polyhedron and only the source is given. We then partition the boundary of this polyhedron so that future queries with new goals on it can be answered quickly. In doing so, we make use of standard planar point location algorithms for a straight-edge planar subdivision for querying. In the other case, we hint the possibility of a space partitioning scheme to solve the same problem for many polyhedra (albeit very inefficiently for the time being). This partitions the free space around polyhedra into regions bounded by high-order surfaces so that the shortest paths for all goals in a given region follow identical sequence of edges of the given polyhedra. This method makes use of a recent spatial point location

algorithm for arbitrary algebraic varieties.

• We have completed the preliminary version of a workbench to compute shortest paths. The workbench comprises a family of Lisp and Macsyma programs which implement the algorithms and the techniques mentioned above. lets the user to run these programs interactively. and renders graphical output. To the best of our knowledge the shortest path pictures computed by this workbench are the first ones in 3-space.

6.2 Future Research and Open Problems

We obviously answered only a small section of the problems posed by FINDPATH in this thesis. There are many interesting problems deserving further study. Below these are listed in no particular order. (An asterisk identifies an especially difficult problem.)

• FINDPATH (*): Is there an algorithm whose execution time is provably lower than doubly exponential in workspace complexity (e.g., the total number of vertices) to solve FINDPATH? What are the trade-offs if one is willing to solve the problem only approximately[86]?

• OPTIMAL LINE (EDGE) VISIT (*): Unless there are breakthrough achievements in solving GEOMETRIC TRAVELING SALESMAN. this problem is intractable if one relaxes the ordered visit requirement. However. sufficiently fast numerical methods may be designed to solve the ordered version which also seems to be intractable with purely algebraic methods.

• BOUNDARY FINDPATH: Regarding the shortest paths on a convex polyhedron with n vertices. is $O(n^2 \log n)$ a lower preprocessing bound[75] for this problem?

• BOUNDARY FINDPATH (locus): How does the Voronoi diagram on the boundary of a convex polyhedron change when the source moves? Theoretically. this would amount to parametrizing the diagram's edges with respect to the source so that we can guess how they will change metrically while the source moves on the boundary. Note however that the change is by no means continuous. i.e., there will be certain "jump" points at which the diagram on a given face of the polyhedron will gain a new topology. Accounting for this effect seems difficult. On the other hand. Franklin suggested that one can make movies (by animating our

algorithm presented in section 4.1 for different sources) showing the effect of different locations of the source to study this problem experimentally. We think that this is an exciting idea and is implementable within our workbench.

• FINDPATH (locus): We think that the first thing to do to advance our knowledge about this problem is to build an experimental system. This must be similar to Verrilli's program[112] in concept, however with more sophisticated data structures and operations. A major problem which does not exist in Verrilli's 2-dimensional system is to find effective ways of communicating 3-dimensional visual information about surfaces.

• Algebraic surfaces (*): We are particularly interested in finding the intersection of two arbitrary surfaces efficiently and reliably. The latter requirement necessarily dictates a symbolic (rather than numeric) approach to the problem since there may be all kinds of degeneracies. Another relevant problem is to enumerate the regions of 3-space separated by n surfaces which may intersect each other in all conceivable ways[7]. Although we believe that there must exist deep results on the intersections of algebraic varieties in the area of algebraic geometry, the introduction of them to the realm of computational geometry has begun only recently[5].

• Polyhedra with higher genus and curved boundary: Given a boundary description for a polyhedron, one is first required to determine where the holes are. As long as the source and/or the goal is not inside it, a *bounded cavity* (a single-ended hole as in figure 6.1) cannot contribute to the shortest path computation and thus can be filled. Curved objects make things extremely difficult, e.g., there may be an innumerable number of shortest paths. Shortest paths on objects like Moebius bands and Klein bottles are also confusing.

• FINDPATH for a single nonconvex polyhedron (*): Efficient algorithms for this problem must not exist in the light of the following informal argument. Assume that we can find shortest paths in the presence of a single nonconvex polyhedron efficiently. We shall show that then we can solve the general version with many obstacles efficiently too. To see this, imagine that we tie the given polyhedra together with thin "connectors" to obtain one big nonconvex polyhedron. A connector may be considered as a thin prism which runs between two polyhedra while avoiding others

(figure 6.2). (This is similar to stabbing lines as studied by Edelsbrunner et al[23] or finding transversals of simple objects as introduced by Edelsbrunner[25].) Any answer for this object is an answer for our initial workspace with many obstacles since the probability that a shortest path intersects a connector can be made sufficiently small by reducing the connector thickness.

• Good (rather than optimal) paths: Clearly, no real task-level robot system would like to compute the shortest paths since the latter by definition tend to touch the obstacles. On the other hand, finding good paths which avoid the obstacles comfortably is not a well-defined problem (at least in computational geometry) since the notion of "good" is necessarily subjective[57]. For example, if there exists a torus-like obstacle in the workspace, in general one would require that no robot path pass through its hole. In this case none of the inner paths (no matter how comfortably it avoids the torus) would be good.

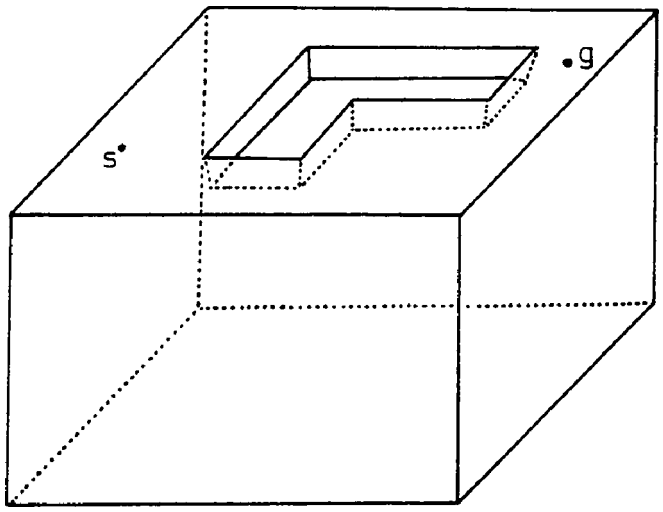


Figure 6.1 A bounded cavity which can be safely filled.

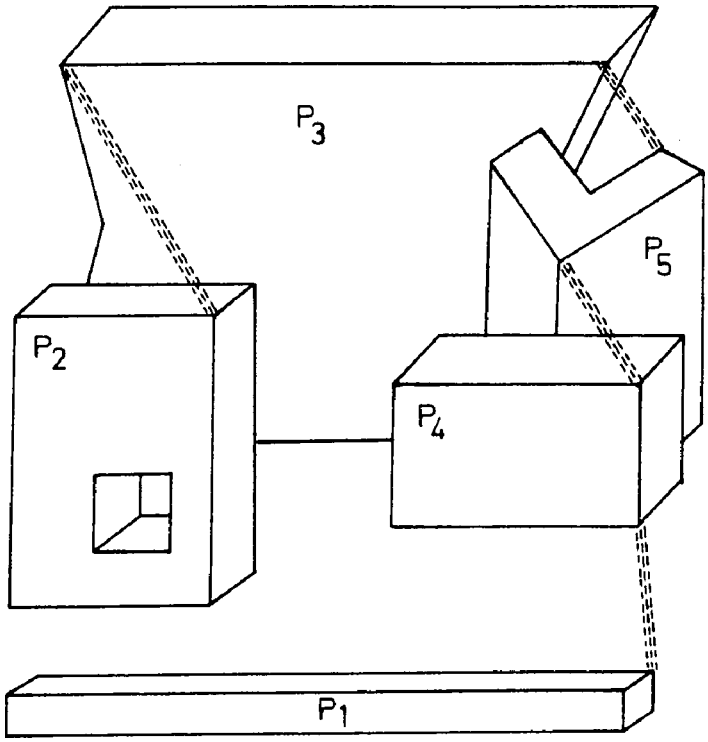


Figure 6.2 A family of connectors in a general workspace constructing a big polyhedron.

REFERENCES

1. H. Abelson and A. DiSessa, *Turtle Geometry. The Computer as a Medium for Exploring Mathematics,* MIT Press, Cambridge, MA, 1982.
2. V. Akman, "Minimal Path Problems in Model-Based Robot Programming: A Literature Survey," Tech. Rep. IPL-TR-042, Dept. of Electrical, Computer, and Systems Eng., Rensselaer Polytechnic Inst., Troy, N.Y., Jan. 1983.
3. A. D. Aleksandrov, *Konvexe Polyeder,* Akademie-Verlag, Berlin, 1958.
4. D. S. Arnon, G. E. Collins, and S. McCallum, "Cylindrical Algebraic Decomposition I: The Basic Algorithm," *SIAM Journal on Computing,* vol. 13, no. 4, pp. 865-877, Nov. 1984.
5. D. S. Arnon, G. E. Collins, and S. McCallum, "Cylindrical Algebraic Decomposition II: An Adjacency Algorithm for the Plane," *SIAM Journal on Computing,* vol. 13, no. 4, pp. 878-889, Nov. 1984.
6. D. Avis and B. K. Bhattacharya, "Algorithms for Computing the d-Dimensional Voronoi Diagrams and Their Duals," in *Advances in Computing Research,* ed. F. P. Preparata, pp. 159-180, JAI Press, 1983.
7. H. Bieri and W. Nef, "A Recursive Sweep-plane Algorithm Determining All Cells of a Finite Division of R^d," *Computing,* vol. 28, no. 3, pp. 189-198, 1982.
8. J. W. Boyse, "Interference Detection among Solids and Surfaces," *Communications of the ACM,* vol. 21, no. 1, pp. 3-9, Jan. 1979.
9. R. A. Brooks and T. Lozano-Perez, "A Subdivision Algorithm in Configuration Space for Findpath with Rotation," Memo No. 684, Artificial Intelligence Lab., Massachusetts Inst. of Technology, Cambridge, MA, Dec. 1982.
10. R. A. Brooks, "Solving the Findpath Problem by Good Representation of Free Space," *IEEE Trans. on Systems, Man, and Cybernetics,* vol. 13, no. 3, pp. 190-197, Mar. 1983.
11. D. Cartwright and T. C. Gleason, "The Number of Paths and Cycles in a Digraph," *Psychometrika,* vol. 31, no. 2, pp. 179-199, Jun. 1966.
12. The Univ. of Michigan Computing Center, *Plot Description System,* Michigan Terminal System (MTS) vol. 11, Ann Arbor, MI, Apr. 1980.
13. B. M. Chazelle, "Fast Searching in a Real Algebraic Manifold with Applications to Geometric Complexity," Tech. Rep. CS-84-13, Dept. of Computer Science, Brown Univ., Providence, RI, Jun. 1984.
14. O. Chein and L. Steinberg, "Routing Past Unions of Disjoint Linear Barriers," *Networks,* vol. 13, pp. 389-398, 1983.
15. G. E. Collins, "The Calculation of Multivariate Polynomial Resultants," *Journal of the ACM,* vol. 18, no. 4, pp. 515-532, Oct. 1971.
16. G. E. Collins, "Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition," *Proc. of the 2nd GI Conf. on Automata Theory and Formal Languages (Lecture Notes in Computer Science 33)* , pp. 134-183, Springer-Verlag, Berlin, 1975.
17. R. Courant and H. Robbins, *What is Mathematics?,* Oxford Univ. Press, New York, 1941.
18. D. P. Dobkin and R. J. Lipton, "Multidimensional Searching Problems," *SIAM Journal on Computing,* vol. 5, no. 2, pp. 181-186, 1976.

19. B. R. Donald, "Hypothesizing Channels through Free Space in Solving the Findpath Problem," Memo No. 736, Artificial Intelligence Lab., Massachusetts Inst. of Technology, Cambridge, MA, Jun. 1983.

20. B. R. Donald, "The Mover's Problem in Automated Structural Design," Manuscript, Lab. for Computer Graphics and Spatial Analysis, Graduate School of Design, Harvard Univ., Cambridge, MA, Aug. 1983.

21. C. M. Eastman, "Representations for Space Planning," *Communications of the ACM*, vol. 13, no. 4, pp. 242-250, Apr. 1970.

22. H. Edelsbrunner and H. A. Maurer, "A Space-optimal Solution of General Region Location," *Theoretical Computer Science*, vol. 16, pp. 329-336, 1981.

23. H. Edelsbrunner, H. A. Maurer, F. P. Preparata, A. L. Rosenberg, E. Welzl, and D. Wood, "Stabbing Line Segments," *BIT*, vol. 22, pp. 274-281, 1982.

24. H. Edelsbrunner, L. J. Guibas, and J. Stolfi, "Optimal Point Location in a Monotone Subdivision," Tech. Rep. 2, DEC Systems Research Center, Palo Alto, CA, Oct. 1984.

25. H. Edelsbrunner, "Finding Transversals for Sets of Simple Geometric Figures," *Theoretical Computer Science*, vol. 35, pp. 55-69, 1985.

26. N. V. Efimov, "Qualitative Problems of the Theory of Deformation of Surfaces," in *Differential Geometry and Calculus of Variations*, vol. 6, pp. 274-423, American Mathematical Society Translations Series, 1962.

27. J. K. Foderaro, K. L. Sklower, and K. Layer, *The FRANZ LISP Manual*, Univ. of California, Berkeley, CA, Jun. 1983.

28. J. D. Foley and A. van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, MA, 1982.

29. W. R. Franklin, "Partitioning the Plane to Calculate Minimal Paths to any Goal around Obstructions," Manuscript, Dept. of Electrical, Computer, and Systems Eng., Rensselaer Polytechnic Inst., Troy, N.Y., Nov. 1982.

30. W. R. Franklin and V. Akman, "Shortest Paths between Source and Goal Points Located on/around a Convex Polyhedron," *Proc. of the 22nd Allerton Conf. on Communication, Control, and Computing*, Monticello, IL, Sep. 1984.

31. W. R. Franklin and V. Akman, "A Workbench to Compute Unobstructed Shortest Paths in 3-Space," *Proc. of the SIAM Conf. on Geometric Modeling and Robotics (Albany, N.Y., Jul. 1985)*, 1985 (to appear).

32. W. R. Franklin, V. Akman, and C. Verrilli, "Voronoi Diagrams with Barriers and on Polyhedra for Minimal Path Planning," *The Visual Computer - An International Journal on Computer Graphics*, Springer-Verlag, 1985 (to appear).

33. W. R. Franklin and V. Akman, "Partitioning the Space to Calculate Shortest Paths to any Goal around Polyhedral Obstacles," *Proc. of the 1st Annual Workshop on Robotics and Expert Systems (Houston, TX, Jul. 1985)*, Instrumentation Society of America, 1985 (to appear).

34. W. R. Franklin and V. Akman, "Euclidean Shortest Paths in 3-Space, Voronoi Diagrams with Barriers, and Related Complexity and Algebraic Issues," *Proc. of the NATO Advanced Study Institute on Fundamental Algorithms for Computer Graphics (Ilkley, Yorkshire, Apr. 1985)*, Springer-Verlag, 1985 (to appear).

35. M. Frechet and K. Fan, *Initiation to Combinatorial Topology*, Prindle, Weber, Schmidt, Boston, MA, 1967.

36. C. B. Garcia and W. I. Zangwill, "Finding All Solutions to Polynomial Systems and Other Systems of Equations," *Mathematical Programming*, vol. 16, pp. 159-176, 1979.

37. C. B. Garcia and W. I. Zangwill, *Pathways to Solutions, Fixed Points, and Equilibria*, Prentice-Hall, Englewood Cliffs, N.J., 1981.

38. M. R. Garey, R. L. Graham, and D. S. Johnson, "Some NP-complete Geometric Problems," *Proc. of the 8th Annual ACM Symp. on Theory of Computing*, pp. 10-22, May 1976.

39. M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-completeness*, W. H. Freeman, San Francisco, CA, 1979.

40. Mathlab Group, *MACSYMA Reference Manual*, 2 vols., Lab. for Computer Science, Massachusetts Inst. of Technology, Cambridge, MA, 1983.

41. B. Grunbaum, *Convex Polytopes*, John Wiley, New York, 1967.

42. L. J. Guibas and J. Stolfi, "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams," *Proc. 15th Annual ACM Symp. on Theory of Computing*, pp. 221-234, 1983.

43. F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, Oct. 1972.

44. L. E. Heindel, "Integer Arithmetic Algorithms for Polynomial Real Zero Determination," *Journal of the ACM*, vol. 18, no. 4, pp. 533-548, Oct. 1971.

45. J. E. Hopcroft, D. A. Joseph, and S. H. Whitesides, "On the Movement of Robot Arms in 2-Dimensional Bounded Regions," *Proc. of the 23rd Annual IEEE Conf. on Foundations of Computer Science*, pp. 280-289, 1982.

46. J. E. Hopcroft, D. A. Joseph, and S. H. Whitesides, "Movement Problems for 2-Dimensional Linkages," *SIAM Journal on Computing*, vol. 13, no. 3, pp. 610-629, Aug. 1984.

47. J. E. Hopcroft and G. Wilfong, "On the Motion of Objects in Contact," Tech. Rep. 94-602, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., May 1984.

48. J. E. Hopcroft and G. Wilfong, "Reducing Multiple Object Motion Planning to Graph Searching," Manuscript, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., 1984.

49. J. E. Hopcroft, J. T. Schwartz, and M. Sharir, "On the Complexity of Motion Planning for Multiple Independent Objects: PSPACE-hardness of the Warehouseman's Problem," Tech. Rep. 14, Courant Inst. of Mathematical Sciences, New York Univ., New York, Feb. 1984.

50. W. E. Howden, "The Sofa Problem," *Computer Journal*, vol. 11, no. 3, pp. 299-301, Nov. 1968.

51. R. A. Jacobson and K. L. Yocom, "Paths of Minimal Length within a Cube," *American Mathematical Monthly*, vol. 73, pp. 634-639, Jun. 1966.

52. R. A. Jacobson and K. L. Yocom, "Shortest Paths within Polygons," *Mathematics Magazine*, pp. 290-293, Nov. 1966.

53. R. A. Jacobson, "Paths of Minimal Length within Hypercubes," *American Mathematical Monthly*, vol. 73, pp. 868-872, Oct. 1966.

54. D. S. Johnson and C. H. Papadimitriou, "Computational Complexity and the Traveling Salesman Problem," in *The Traveling Salesman Problem*, ed. E. L. Lawler, J. K. Lenstra, A. G. Rinooy Kan, John Wiley, New York, (to be published).

55. E. Kaltofen, private communication, 1985.

56. N. Katoh, T. Ibaraki, and H. Mine, "An Efficient Algorithm for k-Shortest Simple Paths," *Networks*, vol. 12, pp. 411-427, 1982.

57. D. G. Kirkpatrick, "Efficient Computation of Continuous Skeletons," *Proc. of the 20th IEEE Annual Symp. on Foundations of Computer Science*, pp. 18-27, Oct. 1979.

58. D. G. Kirkpatrick, "Optimal Search in Planar Subdivisions," *SIAM Journal on Computing*, vol. 12, no. 1, pp. 28-35, Feb. 1983.

59. V. Klee, "On the Complexity of d-Dimensional Voronoi Diagrams," *Archiv der Mathematik*, vol. 34, pp. 75-80, 1980.

60. S. Y. Ku and R. J. Adler, "Computing Polynomial Resultants: Bezout's Determinant vs. Collins' Reduced PRS Algorithm," *Communications of the ACM*, vol. 12, no. 1, pp. 23-30, Jan. 1969.

61. R. C. Larson and V. O. K. Li, "Finding Minimum Rectilinear Distance Paths in the Presence of Barriers," *Networks*, vol. 11, no. 3, pp. 285-304, Sep. 1981.

62. D. T. Lee and F. P. Preparata, "Location of a Point in a Planar Subdivision and its Applications," *SIAM Journal on Computing*, vol. 6, no. 3, pp. 594-606, Sep. 1977.

63. D. T. Lee and R. L. Drysdale, "Generalization of Voronoi Diagrams in the Plane," *SIAM Journal on Computing*, vol. 10, no. 1, pp. 73-87, Feb. 1981.

64. D. T. Lee, "On k-Nearest Neighbor Voronoi Diagrams in the Plane," *IEEE Trans. on Computers*, vol. 31, no. 6, pp. 478-487, Jun. 1982.

65. D. T. Lee and F. P. Preparata, "Euclidean Shortest Paths in the Presence of Rectilinear Barriers," *Networks*, vol. 14, pp. 393-410, 1984.

66. D. T. Lee and F. P. Preparata, "Computational Geometry - A Survey," *IEEE Trans. on Computers*, vol. 33, no. 12, pp. 1072-1101, Dec. 1984.

67. W. Lipski, Jr., "Finding a Manhattan Path and Related Problems," *Networks*, vol. 13, no. 3, pp. 399-409, Sep. 1983.

68. R. J. Lipton and R. E. Tarjan, "A Separator Theorem for Planar Graphs," *SIAM Journal on Applied Mathematics*, vol. 36, no. 2, pp. 177-189, Apr. 1979.

69. R. J. Lipton and R. E. Tarjan, "Applications of a Planar Separator Theorem," *SIAM Journal on Computing*, vol. 9, no. 3, pp. 615-627, Aug. 1980.

70. T. Lozano-Perez and M. A. Wesley, "An Algorithm for Planning Collision-free Paths among Polyhedral Objects," *Communications of the ACM*, vol. 22, no. 10, pp. 560-570, Oct. 1979.

71. T. Lozano-Perez, "Spatial Planning, a Configuration Space Approach," *IEEE Trans. on Computers*, vol. 32, no. 2, pp. 108-120, Feb. 1983.

72. L. A. Lyusternik, *Shortest Paths, Variational Problems*, Macmillan, New York, 1964.

73. K. Mehlorn, *Data Structures and Algorithms (vol. 3: Multi-dimensional Searching and Computational Geometry)*, 3 vols., Springer-Verlag, Heidelberg, Berlin, 1984.

74. M. Mignotte, "Identification of Algebraic Numbers," *Journal of Algorithms*, vol. 3, pp. 197-204, 1982.

75. J. S. B. Mitchell and C. H. Papadimitriou, "The Discrete Geodesic Problem," Manuscript, Dept. of Computer Science, Stanford Univ., Stanford, CA, 1985.

76. A. P. Morgan, "A Method of Computing All Solutions to Systems of Polynomial Equations," *ACM Trans. on Mathematical Software*, vol. 9, no. 1, pp. 1-17, Mar. 1983.

77. J. Moses, "Solution of Systems of Polynomial Equations by Elimination," *Communications of the ACM*, vol. 9, no. 8, pp. 634-637, Aug. 1966.

78. V-D. Nguyen, "The Findpath Problem in the Plane," Memo No. 760, Artificial Intelligence Lab., Massachusetts Inst. of Technology, Cambridge, MA, Feb. 1984.

79. C. O'Dunlaing, M. Sharir, and C. K. Yap, "Retraction: A New Approach to Motion-planning," *Proc. of the 15th Annual ACM Symp. on Theory of Computing*, pp. 207-220, Apr. 1983.

80. C. O'Dunlaing and C. K. Yap, "The Voronoi Method for Motion-planning: I. The Case of a Disc," Tech. Rep. 53, Courant Inst. of Mathematical Sciences, New York Univ., New York, Mar. 1983.

81. C. O'Dunlaing, M. Sharir, and C. K. Yap, "Generalized Voronoi Diagrams for Moving a Ladder: I. Topological Analysis," Tech. Rep. 139, Courant Inst. of Mathematical Sciences, New York Univ., New York, Nov. 1984.

82. C. O'Dunlaing, M. Sharir, and C. K. Yap, "Generalized Voronoi Diagrams for Moving a Ladder: II. Efficient Construction of the Diagram," Tech. Rep. 140, Courant Inst. of Mathematical Sciences, New York Univ., New York, Nov. 1984.

83. J. O'Rourke, S. Suri, and H. Booth, "Shortest Paths on Polyhedral Surfaces," *Proc. of the 2nd Annual Symp. on Theoretical Aspects of Computer Science (Lecture Notes on Computer Science 182)*, pp. 243-254, Springer-Verlag, New York, Jan. 1985.

84. M. H. Overmars, "The Locus Approach," Tech. Rep. RUU-CS-83-12, Computer Science Dept., Univ. of Utrecht, Utrecht, the Netherlands, Jul. 1983.

85. C. H. Papadimitriou, "The Euclidean Traveling Salesman Problem is NP-complete," *Theoretical Computer Science*, pp. 237-244, 1977.

86. C. H. Papadimitriou, "An Algorithm for Shortest Motion in Three Dimensions," *Information Processing Letters*, 1985 (to appear).

87. G. Pfister, "On solving the Findspace Problem, or How to Find Where Things aren't," Manuscript, Artificial Intelligence Lab., Massachusetts Inst. of Technology, Cambridge, MA, Mar. 1973.

88. F. P. Preparata and S. J. Hong, "Convex Hulls of Finite Sets of Points in Two and Three Dimensions," *Communications of the ACM*, vol. 20, no. 2, pp. 87-93, Feb. 1977.

89. F. P. Preparata, "A New Approach to Planar Point Location," *SIAM Journal on Computing*, vol. 10, no. 3, pp. 473-482, Aug. 1981.

90. J. S. Provan, "The Complexity of Reliability Computations in Planar and Acyclic Graphs," Tech. Rep. 83/12, Operations Research and Systems Analysis Curriculum, Univ. of North Carolina, Chapel Hill, NC, Dec. 1984.

91. J. S. Provan, private communication, 1985.

92. J. H. Reif, "Complexity of the Mover's Problem and Generalizations," *Proc. of the 20th Annual IEEE Conf. on Foundations of Computer Science*, pp. 421-427, 1979.

93. T. Saaty, *Optimization in Integers and Related Extremal Problems*, McGraw-Hill, New York, 1970.

94. J. T. Schwartz, *Differential Geometry and Topology*, Gordon and Breach, New York, 1968.

95. J. T. Schwartz and M. Sharir, "On the Piano Mover's Problem: I. The Case of a Two-dimensional Rigid Polygonal Body Moving amidst Polygonal Barriers," *Communications on Pure and Applied Mathematics*, vol. XXXVI, pp. 345-398, 1983.

96. J. T. Schwartz and M. Sharir, "On the Piano Mover's Problem: II. General Techniques for Computing Topological Properties of Real Algebraic Manifolds," *Advances in Applied Mathematics*, vol. 4, pp. 298-351, 1983.

97. J. T. Schwartz and M. Sharir, "On the Piano Mover's Problem: III. Coordinating the Motion of Several Bodies: The Special Case of Circular Bodies Moving amidst Polygonal

Barriers," *International Journal of Robotics Research*, vol. 2, no. 3, pp. 46-75, Fall 1983.

98. J. T. Schwartz and M. Sharir, "On the Piano Mover's Problem: V. The Case of a Rod Moving in Three-dimensional Space amidst Polygonal Obstacles," *Communications on Pure and Applied Mathematics*, vol. XXXVII, pp. 815-848, 1984.

99. R. Sedgewick, *Algorithms*, Addison-Wesley, Reading, MA, 1983.

100. M. I. Shamos and D. Hoey, "Closest-point Problems," *Proc. of the 16th IEEE Annual Symp. on Foundations of Computer Science*, pp. 151-162, Oct. 1975.

101. M. I. Shamos, "Computational Geometry," Ph.D. dissertation, Dept. of Computer Science, Yale Univ., New Haven, CT, 1978.

102. M. Sharir and A. Schorr, "On Shortest Paths in Polyhedral Spaces," *Proc. of the 16th Annual ACM Symp. on Theory of Computing*, pp. 144-153, 1984.

103. M. Sharir and E. Ariel-Sheffi, "On the Piano Mover's Problem: IV. Various Decomposable Two-dimensional Motion-planning Problems," *Communications on Pure and Applied Mathematics*, vol. XXXVII, pp. 479-493, 1984.

104. P. Spirakis and C. K. Yap, "Strong NP-hardness of Moving Many Discs," *Information Processing Letters*, vol. 19, pp. 55-59, 1984.

105. G. J. Sussman, *A Computer Model of Skill Acquisition*, American Elsevier, New York, 1975.

106. I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, "A Characterization of Ten Hidden-surface Algorithms," *ACM Computing Surveys*, vol. 6, no. 1, pp. 1-55, Mar. 1974.

107. R. E. Tarjan, *Data Structures and Network Algorithms*, CBMS-NSF Regional Conference Series in Applied Mathematics, SIAM, Philadelphia, PA, 1983.

108. A. Tarski, *A Decision Method for Elementary Algebra and Geometry*, Univ. of California Press, 1951.

109. M. Tompa, "An Optimal Solution to a Wire-routing Problem," *Journal of Computer and System Sciences*, vol. 23, pp. 127-150, 1981.

110. S. Udupa, "Collision Detection and Avoidance in Computer-controlled Manipulators," Ph.D. dissertation, Dept. of Electrical Eng., California Inst. of Technology, Pasadena, CA, 1977.

111. L. G. Valiant, "The Complexity of Enumeration and Reliability Problems," *SIAM Journal on Computing*, vol. 8, no. 3, pp. 410-421, Aug. 1979.

112. C. Verrilli, "One Source Voronoi Diagrams with Barriers, a Computer Implementation," Tech. Rep. IPL-TR-060, Image Processing Lab., Rensselaer Polytechnic Inst., Troy, N.Y., Feb. 1984.

113. B. L. van der Waerden, *Algebra*, 2 vols., Frederick Ungar, New York, 1977.

114. P. H. Winston, *Artificial Intelligence*, Addison-Wesley, Reading, MA, 1977.

115. J. Y. Yen, "Finding the k-Shortest Loopless Paths in a Network," *Management Science*, vol. 17, no. 11, pp. 712-716, Jul. 1971.

APPENDIX
INTERNALS AND AVAILABLE FUNCTIONS OF SP

This appendix has four parts. In part (a) we describe how to use SP. Parts (b), (c), and (d) document the available Franz Lisp, Macsyma, and plot functions in SP, respectively.

(a) How to Use it

SP source code resides in three subdirectories under user *akmanv* in CSLAB: *phd* , *maxim* , and *mts* . These respectively hold the Franz Lisp, Macsyma, and plot (Fortran) portions of SP. In this part we shall only describe how to use the Franz Lisp portion which comprises the bulk of SP. The remaining portions will be documented in parts (c) and (d).

To run the Franz part of SP, a user should read the file *doit.l* into Lisp thus: (load *'doit.l*). *Doit* does the rest of the work and loads all source files. From this point on the user types SP function names along with their required parameters into Lisp and receives responses. If graphical output is required then the plot utilities in part (d) should be run on the results written into plot files.

In addition to *doit.l* there are 25 files each holding one or more functions: *alex.l* , *allgoals.l* , *cart.l* , *consint.l* , *cromega.l* , *dijk.l* , *face.l*, *fassil.l*, *foldbak.l*, *geom.l* , *lseghedra.l* , *matpac.l* , *naivor.l* , *parms.l* , *pltvor.l* , *prettyplt.l* , *pyrcons.l* , *recons.l* , *rpoly.l* , *shclip.l* , *util.l* , *visi.l* , *vorin.l* , *wunfold.l*, *yen.l* . The file *parms.l* holds some important constants whereas *util.l* has a few utilities. The rest of the files comprise technical code. It should be noted that the contents of the files reflect the historical development of SP and do not share a common underlying theme. In addition to these, there are several polyhedral data files such as *cube.DAT* , *dodeca.DAT* , etc. which may be used for test purposes. (Section 5.2 shows a sample data file.)

In part (b) we summarize each Franz function (there are about 110 of them comprising approximately 3000 lines of moderately commented code) in SP. The name appearing in parentheses after each function is the source file name that contains it. As mentioned in section 5.2, (global) variables are denoted in (upper-case) italics inside angular brackets.

We prefer to work with Lisp in the interpreted mode although this makes things considerably slower. It is possible (although never tried) to compile SP functions under Lisp. In general, almost all of our Franz functions (with the exception of *pltvor* , a Voronoi program, and *vedges* and *vfaces* , polyhedral

edge and face visibility detectors) in SP are fast enough to be executed while the user is interacting with the system. The main reason for the slowness (varying from few minutes to 15 minutes for moderate size objects) of the mentioned programs originate from their naive (but simple to program) algorithmic structure. These are certainly not the most important parts of SP in terms of novelty and we think that due to the extensible design of SP, faster algorithms can always be incorporated if required.

(b) Franz Lisp Functions

The following functions run under Franz Lisp (opus 38.79). The reader is referred to Franz Lisp Manual for more information. (CAVEAT: SP regards any file whose name starts with "foo" or "quux" as temporary and at any time may overwrite it.)

• function: *alex* (*alex.l*)
parameters: <*outf*> <*fvs*> <*s*> <*g*>
Given a face visit sequence <*fvs*> and the 3-dimensional source (<*s*>) and goal (<*g*>) points on the first and the last faces of it. this creates an *xy*-plane planar development in <*UNFOLD*> and writes it into file <*outf*> if <*outf*> is nonnil. <*UNFOLD*> is a list which holds the vertex coordinates of the developed faces for each face specified by <*fvs*>. It is assumed that the faces in <*fvs*> are pairwise adjacent in left-to-right order. *Alex* creates <*PLNSRC*> and <*PLNGOL*> which are the images of <*s*> and <*g*> in *xy*-plane. <*Fvs*> is saved in <SIGMA>. Also created is <*SRCTOGOL*> which is the distance between them in the plane.

• function: *comedge* (*alex.l*)
parameters: <*f1*> <*f2*>
Given two adjacent faces <*f1*> and <*f2*> this returns the common edge between them. The common edge is a pair of vertex numbers.

• function: *xsect* (*alex.l*)
parameters: <*vl* 1> <*vl* 2>
Given vertex lists <*vl* 1> and <*vl* 2> for two faces this returns a pair of vertex numbers (in the order they occur in the face of <*vl* 2>) common to both.

• function: *fnd2pts (alex.l)*
parameters: <*a*> <*b*> <*c*>
Returns a list of two 3-dimensional points (both with zero *z*-coordinate) that are guaranteed to be on line $ax + by + c = 0$ and different.

• function: *consnet* (*alex.l*)
parameters: <*f*> <*vl*>

Returns a list consisting of the 3-dimensional coordinates of the vertices specified in $<vl>$ (vertex numbers for face $<f>$). The first element of the returned list is $<f>$.

• function: *allgoals* (*allgoals.l*)
parameters: $<outf>$ $<fs>$ $<fg>$ $<ndeve>$ $<s>$ $<g>$
Given the 3-dimensional source ($<s>$) and goal ($<g>$) points which are on faces $<fs>$ and $<fg>$ respectively, this writes the images of $<g>$ into file $<outf>$ for the first $<ndeve>$ face visit sequences in $<OMEGA>$ between $<fs>$ and $<fg>$ if $<outf>$ is nonnil. Only the images that correspond to legal developments are written. With each image, *allgoals* writes the specific $<UNFOLD>$ and the 3-dimensional path ($<BENDPTS>$) if the development is legal. The face visit sequence which gives the shortest path is $<MINSIGMA>$ whereas the path itself is $<MINPATH>$ (with length $<MINSTOG>$).

• function: *cntnnil* (*allgoals.l*)
parameter: $<l>$
Returns the number of nonnil items in list $<l>$.

• function: *cart* (*cart.l*)
parameter: $<gr>$
Given a graph $<gr>$ this creates a matrix $<S>$ such that the (i,j)-th entry of $<S>$ is equal to the number of simple walks between nodes i and j, $i \neq j$. The diagonal entries of $<S>$ give the number of cycles of any length containing node i. *Cart* uses specialized functions *ordright*, *ordbottom*, *nones*, and *nnzrows* (all in *cart.l*) whose descriptions are omitted.

• function: *pinteg* (*consint.l*)
parameter: $<inf>$
Returns t if the current polyhedron is consistent. This assumes that $<FACES>$, $<VERTICES>$, $<EDGES>$, $<OMEGA>$, $<NFACES>$, $<NVERTICES>$, and $<NEDGES>$ are available. A consistent polyhedron is defined as one which satisfies the following:

> Euler's formula: $NFACES + NVERTICES = NEDGES + 2$.
> Degree of each node in $<OMEGA>$ is at least 3.
> Each face has at least 3 vertices.
> Each edge belongs to exactly 2 faces.
> The vertices of a face are in the same plane within an error of ϵ.
> All vertices are different.

If file $<inf>$ is nonnil then the polyhedron is read from it. *Pinteg* does not immediately return after it detects an error but checks all of the above

properties.

• function: *cirmember* (*consint.l*)
parameters: <*pe*> <*vl*>
Returns t if the elements of <*pe*> (a pair of edge numbers) occur in <*vl*> (a list of vertex numbers) in circular order. (Two items a and b are in circular order in list l if b immediately follows a in l with wrap around allowed.)

• function: *ufinteg (consint.l)*
parameter: <*unf*>
Returns t if the planar development in <*unf*> is consistent (i.e., z-coordinates of all developed vertices are in the ϵ-neighborhood of 0). Normally, <*unf*> will be equal to <*UNFOLD*>. *Ufinteg* will not stop in a single error; it will flag all of them.

• function: *cromega* (*cromega.l*)
parameters: -
This creates <*OMEGA*> assuming that <*FACES*> and <*VERTICES*> already exist. <*OMEGA*> is a list which shows for each face the list of all other faces that are adjacent to it.

• function: *pxsect* 2 (*cromega.l*)
parameters: <*l* 1> <*l* 2>
Returns t if lists <*l* 1> and <*l* 2> have exactly two elements in common.

• function: *pinfo (cromega.l)*
parameters: <*inf*> <*outf*>
This prints useful information about a polyhedron. If file <*inf*> is nil then the current object is used. If file <*outf*> is nil then the information is written to the terminal and consists of the following:

First <*BLURB*> line for identification purposes.
<*NVERTICES*>.
<*NFACES*>.
Number of arcs in <*OMEGA*>.
Minimum degree of a node in <*OMEGA*>.
Maximum degree of a node in <*OMEGA*>.
Minimum number of vertices in a face.
Maximum number of vertices in a face.
Bounding box extents in x, y, and z directions.

• function: *dijk* (*dijk.l*)
parameters: <*fs*> <*fg*>
This finds the shortest walk(s) between the nodes <*fs*> and <*fg*> in

< *OMEGA* > and returns them in < *SPATHS* >. It also computes < *LSPATHS* > and < *NSPATHS* > which are the length and the number of the shortest walks, respectively. *Dijk* uses specialized functions *getparsps* and *getallsps* (both in *dijk.l*) whose descriptions are omitted.

• function: *padj* (*dijk.l*)
parameters: <*n* 1> <*n* 2>
Returns nonnil if nodes <*n* 1> and <*n* 2> are adjacent in < *OMEGA* >.

• function: *faceqn (face.l)*
parameter: <*f*>
Returns a list consisting of the parameters a , b , c , and d in the face equation $ax + by + cz + d = 0$ of face <*f*>. The normal of the face points outward.

• function: *diang (face.l)*
parameters: <*f1*> <*f2*>
Returns the dihedral angle between faces <*f1*> and <*f2*> in radians. If <*f2*> is one of *'xy* , *'yz* , *'xz* then *diang* gives the dihedral angle between the plane holding <*f1*> and the respective plane.

• function: *getedges (face.l)*
parameters: -
Creates < *EDGES* > and < *NEDGES* > assuming that < *FACES* > and < *NFACES* > are available. < *EDGES* > consists of vertex pairs which define an edge.

• function: *facesof (face.l)*
parameter: < *e* >
Returns a pair of faces which share edge < *e* > (a pair of vertices).

• function: *fassil (fassil.l)*
parameter: < *p* >
Creates the silhouette edges of the current polyhedron from viewpoint < *p* > (assumed to be outside) in < *SILE* >. Also created are < *CCWSILE* >, < *VISE* >, and < *VISF* >. < *CCWSILE* > is the same as < *SILE* > but its edges are such that they follow a counterclockwise order when viewed from < *p* >. < *VISE* > and < *VISF* > hold the visible edges and the visible faces from < *p* >, respectively.

• function: *lexle (fassil.l)*
parameters: <*l* 1> <*l* 2>
Returns t if pair <*l* 1> is lexically less or equal to pair <*l* 2>. The pairs hold integer elements.

• function: *elimdup (fassil.l)*
parameter: <*l* >
Eliminates all occurrences of duplicate members from a sorted list <*l* > and returns the result.

• function: *elimonedup (fassil.l)*
parameter: <l>
Eliminates all but one of the duplicate members from a sorted list <*l* > and returns the result.

• function: *visiblfc (fassil.l)*
parameters: <*f*> <*p*>
Returns t if face <*f*> is visible from viewpoint <*p* >. Due to convexity, <*f*> is visible if any of its interior points (specifically its centroid) is visible from <*p* >. However, to prevent numerical problems, *visiblfc* tests three interior points and does a majority voting.

• function: *visibled (fassil.l)*
parameters: <*e* > <*p* >
Returns t if edge <*e* > (a pair of vertex numbers) is visible from viewpoint <*p* >. Due to convexity, <*e* > is visible if any of its interior points (specifically its midpoint) is visible from <*p* >. However, to prevent numerical problems, *visibled* tests three interior points and does a majority voting.

• function: *inside (fassil.l)*
parameters: <*p*> <*f*>
Returns t if a 3-dimensional point <*p* > is inside or on the boundary of face <*f*>. *Inside* first maps them into the *xy*-plane and carries out all the computation there.

• function: *majority (fassil.l)*
parameter: <*l* >
If the majority of the members of <*l* > are t then return t (breaking ties in favor of nil).

• function: *foldbak (foldbak.l)*
parameters: <*outf*> <*s*> <*g*> <*plns*> <*plng*>
If the planar development in <*UNFOLD* > is legal then this computes the list of bend points (<*BENDPTS* >) of the path connecting the images <*plns* > and <*plng* > of the source and the goal points (<*s* > and <*g* >). If file <*outf*> is nonnil then the result is written to <*outf*>.

• function: *discord (foldbak.l)*
parameters: <*p* 1> <*p* 2> <*d* >

Returns the coordinates of a point on the edge connecting vertex <*p* 1> to vertex <*p* 2> and <*d* > units away from <*p* 1>. Using *discord* a point of <*UNFOLD* > can be mapped into the boundary of a polyhedron, i.e., into 3-space.

• function: *maptounf (foldbak.l)*
parameters: <*l*> <*f*>
Assuming that <*l* > is a pair of vertices and <*f*> is the face which holds them in this order, this returns a list of two points from <*UNFOLD* > such that the first point corresponds to the first vertex and the second point corresponds to the second vertex.

• functions: *vadd vdiff vtimes vquotient (geom.l)*
parameters: <*v* 1> <*v* 2>
Return respectively the element-by-element addition, subtraction, multiplication, and division of 3-dimensional vectors <*v* 1> and <*v* 2>.

• functions: *cross dot* (*geom.l*)
parameters: <*v* 1> <*v* 2>
Return respectively the vector and scalar product of 3-dimensional vectors <*v* 1> and <*v* 2>.

• function: *magnitude* (*geom.l*)
parameter: <*v* >
Returns the length of 3-dimensional vector <*v* >.

• function: *distance* (*geom.l*)
parameters: <*v* 1> <*v* 2>
Returns the distance between 3-dimensional points <*v* 1> and <*v* 2>.

• functions: *specrot genrot* (*geom.l*)
parameters: <*p* > <*a* > <*r* >
Return the image of a 3-dimensional point <*p* > after rotation about axis <*a* > by <*r* > radians. *Specrot* works only with an axis passing through the origin. *Genrot* can handle arbitrary axes (pairs of 3-dimensional points).

• function: *facerot (geom.l)*
parameters: <*f*> <*a*> <*r*>
Returns a new list of vertex coordinates (x and y values only) for face <*f*> after rotation about arbitrary axis <*a* > by <*r* > radians.

• function: *objrot* (*geom.l*)
parameters: <*a* > <*r* >
Returns a new list of vertex coordinates of the current object after rotation

about arbitrary axis $<a>$ by $<r>$ radians.

• function: *facexlat (geom.l)*
parameters: $<f>$ $<v>$
Returns a new list of vertex coordinates (x and y values only) for face $<f>$ after a translation by vector $<v>$.

• function: *objxlat (geom.l)*
parameter: $<v>$
Returns a new list of vertex coordinates which correspond to a translation of the current object by vector $<v>$.

• function: *lineqn (geom.l)*
parameter: $<vp>$
Returns the parameters a, b, c in the equation $ax+by+c=0$ of the line passing through two points specified in $<vp>$. (The points are in the xy-plane within ϵ.)

• function: *almost* 0 *(geom.l)*
parameter: $<x>$
Returns t if $<x>$ is in the ϵ-neighborhood of 0.

• function: *centroid (geom.l)*
parameter: $<f>$
Returns the center of mass of face $<f>$ of the current polyhedron.

• function: *plugpt (geom.l)*
parameters: $<p>$ $<eqn>$
Substitutes a 3-dimensional point $<p>$ into face equation $<eqn>$ and returns the result. $<Eqn>$ consists of the parameters in the equation $ax+by+cz+d=0$.

• function: *lsegpln (geom.l)*
parameters: $<l>$ $<eqn>$
Returns the intersection point of 3-dimensional line segment $<l>$ (a pair of points) and the plane with equation $<eqn>$. This requires that $<l>$ and the plane intersect.

• function: *plseg* 2 *(geom.l)*
parameters: $<l\ 1>$ $<l\ 2>$
Returns t if line segments $<l\ 1>$ and $<l\ 2>$ (point pairs) intersect. The segments are given in the xy-plane.

• function: *triarea (geom.l)*

parameter: <*tri* >
Returns the signed area of triangle <*tri* > (a list of three vertices) which is in the *xy*-plane. The area is positive if the vertices are in counterclockwise order.

• function: *lseghedra* (*lseghedra.l*)
parameters: <*p* > <*s* > <*g* >
Let <*p* > be a list of file names each holding a convex polyhedron. This intersects the source-to-goal segment *sg* with each polyhedron and returns a list of lists which are of the form (*obj x*1 *x*2) where *x*1 and *x*2 are the intersection points of *sg* with the object in file *obj*. Intersections along a vertex or an edge are not counted as proper.

• function: *makadj* (*matpac.l*)
parameters: <*m* > <*gr* >
Makes an adjacency matrix <*m* > from a graph <*gr* >. The format of <*gr* > is equivalent to <*OMEGA* >. In <*m* > the (*i* ,*j*)-th entry is 1 if node *i* is adjacent to node *j* in <*gr* >. Other entries (including the diagonal) are zero.

• function: *mmul* (*matpac.l*)
parameters: <*m* 1> <*m* 2> <*m* 3>
Multiplies integer matrices <*m* 1> and <*m* 2> to obtain <*m* 3>.

• function: *mtpose* (*matpac.l*)
parameters: <*m* 1> <*m* 2>
Transposes integer matrix <*m* 1> to obtain <*m* 2>.

• function: *mzero* (*matpac.l*)
parameter: <*m* >
Returns t if all elements of integer matrix <*m* > are zero.

• function: *mgrind* (*matpac.l*)
parameters: <*m*> <*outf*>
Pretty-prints matrix <*m* > on the terminal or into file <*outf*> if <*outf*> is nonnil.

• functions: *madd msub mdot* (*matpac.l*)
parameters: <*m* 1> <*m* 2>
Respectively add. subtract. and multiply integer matrices <*m* 1> and <*m* 2> element-by-element and return the result in <*m* 1>.

• function: *mident* (*matpac.l*)
parameters: <*m* 1> <*m* 2>
Creates a matrix <*m* 1> and copies integer matrix <*m* 2> to it.

• function: *mmirror* (*matpac.l*)
parameter: <*m* >
Returns t if integer matrix <*m* > is symmetric.

• function: *mexpo* (*matpac.l*)
parameters: <*n* > <*m* 1> <*m* 2>
Raises integer matrix <*m* 1> to power <*n* > and returns the result in <*m* 2>.

• function: *naivor* (*naivor.l*)
parameters: <*inf*> <*outf*> <*ext*>
Writes into file <*outf*> the Voronoi polygons of the points specified in file <*inf*> after clipping them against the window <*ext* >. This uses the specialized functions *vread* and *vwrite* (both in *naivor.l* and omitted here) to read the points and to write the polygons, respectively. <*inf*> holds the *xy* coordinates the Voronoi points (centers), one per line. The first line of it is the number of points. <*Outf*> holds the copy of <*inf*> plus the Voronoi polygons for each point. If <*ext* > is nonnil then it must be equal to the list (%*xmin* %*xmax* %*ymin* %*ymax*) meaning everything will be clipped against a window with the above coordinates. (Default <*ext* > is equal to (-100.0 100.0 -100.0 100.0).) *Naivor* creates the global variables <*VPTS* > and <*NVPTS* > which respectively hold the Voronoi points and their count. Similarly, <*LVOR* > holds the Voronoi polygons which are in counterclockwise order.

• function: *bisect* (*naivor.l*)
parameters: <*p* 1> <*p* 2>
Computes the line bisecting the line segment connecting points <*p* 1> and <*p* 2> (both in *xy*-plane) and returns a list (*m b*) where *m* is its slope and *b* is its *y*-intercept. (If $m = \infty$ then *b* = *x*-intercept.)

• function: *isinside* (*naivor.l*)
parameter: <*p* >
Returns t if point <*p* > in *xy*-plane is inside the window boundary defined by %*xmin* , %*xmax* , %*ymin* , %*ymax* (cf. *naivor*).

• function: *cutbigw* (*naivor.l*)
parameter: <*eqn* >
Returns the intersection points of a line in *xy*-plane with equation <*eqn* > with the "big" window boundary defined by 2%*xmin* , 2%*xmax* , 2%*ymin* , 2%*ymax* (cf. *naivor*). There are always 2 intersections.

• function: *makclbd* (*naivor.l*)
parameters: <*l* > <*eqn* > <*p* >
Returns a big clip window to clip a Voronoi polygon against. It is assumed that *l* is a pair of points which are in counterclockwise order when viewed

from point <*p* >. <*Eqn* > is the equation of the line passing through <*l* >.

• function: *pltvor* (*pltvor.l*)
parameters: <*inf*> <*outf*>
Let <*inf*> be the output of *vorin* . This will read it and prepare <*outf*> which is then submitted to plot utility *drawvor* (cf. part (d)). The format of <*outf*> is as follows:

> Extents: %*xmin* , %*xmax* , %*ymin* , %*ymax* .
> Source face number.
> Goal face number.
> Source point coordinates.
> Number of distinct Voronoi centers.
> Number of vertices of the goal face.
> Coordinates of the vertices of the goal face (one per line).
> Number of faces in this development. (This is followed by faces (one per line) and then Voronoi center coordinates.)
> Number of vertices of a Voronoi polygon. (This is followed by as many vertices as necessary.)

Pltvor detects equal Voronoi centers and discards all but one. It finds a bounding box (with extents %*xmin* , %*xmax* , %*ymin* , %*ymax*) which is used by *naivor* .

• function: *prettylpt* (*prettyplt.l*)
parameters: <*p*> <*outf*>
This writes a file <*outf*> which is used by plot utility *drawpp* (cf. part (d)). Essentially *prettyplt* is used to plot a polyhedron with a 3-dimensional shortest path marked on it from viewpoint <*p* >. The format of <*outf*> is as follows:

> Coordinates of <*p* >.
> Source face number.
> Goal face number.
> Source point coordinates.
> Goal point coordinates.
> Length of face visit sequence.
> Face visit sequence (one face per line).
> Length of the shortest path.
> Length of list <*BENDPTS* >.
> Coordinates of the bend points (one point per line).
> Number of visible faces followed by the below information for each face:
>
> Number of this face.

Centroid coordinates of this face.
Edge endpoint coordinates of this face (one point per line).

Number of invisible faces followed by the below information for each face:

Number of this face.
Centroid coordinates of this face.
Edge endpoint coordinates of this face (one point per line).

• function: *pyrcons* (*pyrcons.l*)
parameters: <*inf*> <*outf*> <*s*> <*g*>
Given two points <*s*> and <*g*> external to a polyhedron P (read from file <*inf*>) this constructs a new convex polyhedron P' and writes it into <*outf*>. P' is such that the shortest path from <*s*> to <*g*> is necessarily situated on it. (In other words, P' is equal to the convex hull of the union of <*s*>, <*g*>, and the vertices of P.)

• function: *assocsrc* (*pyrcons.l*)
parameters: <*e*> <*l*>
If item <*e*> is in list <*l*> then *assocsrc* returns its index.

• function: *holderf (pyrcons.l)*
parameter: <*p*>
Let <*p*> be a 3-dimensional point which is supposed to be on the current polyhedron. This returns the number of the lowest indexed face that holds <*p*> within ϵ.

• function: *onbd* (*pyrcons.l*)
parameters: <*p*> <*f*>
Returns t if point <*p*> is on the boundary of face <*f*>.

• function: *recons* (*recons.l*)
parameters: <*inf*> <*outf*>
This constructs the faces of a straight-edge planar graph specified as a set of its edges. File <*inf*> holds the number of edges and the coordinates of the endpoints of the edges (one edge per line). The output file <*outf*> consists of the number of vertices of the graph followed by the vertex coordinates and the faces of the graph. Each face is specified as a list of pointers to its vertices. All faces are in counterclockwise order except the outer boundary of the graph which is clockwise. *Recons* uses the specialized functions *markvrt*, *unused*, and *follower* (all in *recons.l*) which are omitted.

• function: *tuplex* (*recons.l*)

parameters: <*e* 1> <*e* 2>
Let <*e* 1> and <*e* 2> be two edges (pairs of endpoints). This returns t if the first endpoint of <*e* 1> is lexically less than the first endpoint of <*e* 2>.

• function: *approx* (*recons.l*)
parameters: <*v* 1> <*v* 2>
Returns t if vertices <*v* 1> and <*v* 2> have equal coordinates within ϵ.

• function: *angsort* (*recons.l*)
parameters: <*v* > <*l* >
Sorts the vertices in list <*l* > about vertex <*v* > in descending angular order and returns the sorted list.

• function: *angord* (*recons.l*)
parameters: <*v* 1> <*v* 2>
Returns t if vertex <*v* 1> is after vertex <*v* 2> in increasing angular value in $[0.2\pi)$.

• function: *angle* (*recons.l*)
parameters: <*x* > <*y* >
Returns the angle of vector *xy* in polar coordinates in radians.

• function: *rpoly* (*rpoly.l*)
parameter: <*inf*>
Reads polyhedron data file <*inf*> and creates <*BLURB* >, <*NBLURB* >, <*FACES* >, <*VERTICES* >, <*NFACES* >, <*NVERTICES* >, and <*OMEGA* >. An example data file is shown in section 5.2.

• function: *wpoly* (*rpoly.l*)
parameter: <*outf*>
Writes the current polyhedron into file <*outf*>. This assumes that <*BLURB* >, <*NBLURB* >, <*FACES* >, <*VERTICES* >, <*NFACES* >, and <*NVERTICES* > are available. *Wpoly* can be used to create a new polyhedron file from the current polyhedron (e.g., after some rotation or translation).

• function: *revs* (*rpoly.l*)
parameters: -
Reverses the order of the vertex sequences in <*FACES* >. This is useful when the faces of a polyhedron file are in, say, clockwise order but one needs them in counterclockwise.

• function: *wedges* (*rpoly.l*)
parameter: <*outf*>

Writes the endpoint coordinates of the edges in <*EDGES*> into file <*outf*>. Each line of <*outf*> consists of 6 coordinate values (x 1, y 1, z 1, x 2, y 2, z 2) except the first line which holds <*NEDGES*>.

• function: *shclip* (*shclip.l*)
parameters: <*inp*> <*clipp*>
Let <*inp*> and <*clipp*> be two polygons. <*Clipp*> must be convex whereas <*inp*> may be nonconvex. This clips <*inp*> against clip boundary <*clipp*>. Both polygons should be given in counterclockwise order. The clipped polygon is also in counterclockwise order.

• function: *toleft* (*shclip.l*)
parameters: <*v*> <*bd*>
Returns t if vertex <*v*> is inside the the clip boundary <*bd*> (a pair of points) where "inside" is defined as being to the left of the boundary when one views from the first point of <*bd*> toward the second. It is assumed that the polygon which <*bd*> is a part of is given in counterclockwise order.

• function: *lsegxlin* (*shclip.l*)
parameters: <*l* 1> <*l* 2>
Given line segments <*l* 1> and <*l* 2> which are point pairs this returns the intersection point of <*l* 1> with the line holding <*l* 2> or nil if there is no intersection.

• functions: *vedges ivedges* (*visi.l*)
parameter: <*p*>
Return respectively the set of visible and invisible edges of the current polyhedron from viewpoint <*p*> which is supposed to be external to the polyhedron.

• functions: *vfaces ivfaces* (*visi.l*)
parameter: <*p*>
Return respectively the set of visible and invisible faces of the current polyhedron from viewpoint <*p*> which is supposed to be external to the polyhedron.

• function: *vorin* (*vorin.l*)
parameters: <*outf*> <*fs*> <*fg*> <*ndeve*> <*s*>
Given a source point <*s*> and the source and the goal faces <*fs*> and <*fg*> this writes the images of <*s*> with respect to <*fg*> into file <*outf*> (along with the face visit sequence used to arrive it) for <*ndeve*> developments. (It is guaranteed that when specified <*g*> will be on <*fg*>.)

• function: *wunfold* (*wunfold.l*)

parameters: $< s >$ $< g >$
This is useful for drawing a development. *Wunfold* writes $< UNFOLD >$ into file *foobar* along with $< s >$ and $< g >$. The format of *foobar* is as follows:

Bounding box x and y coordinates.
Number of faces in the development.
The face visit sequence (one face per line).
Source face number.
Goal face number.
Source coordinates (3-space).
Goal coordinates (3-space).
Source image coordinates (xy-plane).
Goal image coordinates (xy-plane).
Source-to-goal distance.
The following is then repeated for each developed face:
    Face number.
    Number of edges for this face.
    $x1$, $y1$, $x2$, $y2$ coordinates for edge endpoints (one edge per line).
    Centroid coordinates for this face in 2-space.

• function: *yen* (*yen.l*)
parameters: $<fs>$ $<fg>$ $<k>$
Returns the first $< k >$ simple walks in $< OMEGA >$ between nodes $<fs>$ and $<fg>$ in $< KPATHS >$ in increasing walk length. Sometimes the length of $< KPATHS >$ may be larger than $< k >$. *Yen* uses the specialized functions *detach* and *getmincands* (both in *yen.l*) which are omitted.

• function: *pcoincide* (*yen.l*)
parameters: $< l\,1 >$ $< l\,2 >$ $< n >$
Returns t if the sublists consisting of the first $< n >$ elements of lists $< l\,1 >$ and $< l\,2 >$ are equal assuming that each of $< l\,1 >$ and $< l\,2 >$ has at least $< n >$ elements.

(c) Macsyma Functions

Macsyma is a large and sophisticated programming system used for performing symbolic (as well as numerical) mathematical manipulations. The reader is referred to the Macsyma Reference Manual for further information. The following descriptions apply to UNIX Macsyma release 304 (MIT and Symbolics, Inc., 1983).

We first summarize how to solve a system of equations with purely algebraic methods in Macsyma. (It is noted that this is basically what OPTIMAL

LINE VISIT requires.) Macsyma built-in function *solve* solves a system of (linear or nonlinear) polynomial equations. To handle the latter *solve* uses the function *algsys* which in turn employs the function *resultant* . Further descriptions of these can be found in the Manual. *Solve* has been tried on the equations described below for *olvang* with success. However, solving many equations with *solve* does not look promising since the function is very slow and sometimes returns with failure after some initial effort which may be quite costly.

In addition to experimenting with elimination, we implemented two numerical functions in Macsyma: *olvang* and *olvdis* . These can be loaded into Macsyma using the command "batch." Before running *olvang* or *olvdis* a file holding the number of lines, the source, the goal, and the line endpoints in the following example format should be batched into Macsyma:

```
N:3:
S:[0.0.0]:
G:[0,4.0]:
P[1]:[1.1.1]; Q[1]:[-1.1,1]:
P[2]:[2.2,1]; Q[2]:[-2.2,1]:
P[3]:[3.3.1]; Q[3]:[-3.3,1]:
```

• function: *olvang*
parameter: $<niter>$
This solves OPTIMAL LINE VISIT using the Newton-Raphson method described in section 2.1 to solve a system of equations stating the equality of the entry and the exit angles that the shortest path makes with the given lines. Each line is parametrized by X_i which is to be determined. Once found, X_i's are used to calculate B_i's (the bend points). The global variable $<EST>$ is a vector holding the starting estimates for the parameters $X = X_1, \cdots, X_n$ of the given lines. (This is initialized by the user before running *olvang* .) After each iteration, *olvang* prints three values on the terminal: $<DELTA>$, $<EPS>$, and $<DIS>$. $<DELTA>$ is an indicator which approaches to zero after each iteration if the computation is converging. *Olvang* stops after $<niter>$ iterations are carried out. Inspecting $<DELTA>$ the user decides to continue with or to abort the computation. In the former case, *olvang* is simply continued with the last value of $<EPS>$ used as a new estimate. In the latter case, a new (and possibly more promising) $<EST>$ must be chosen. $<DIS>$ is the shortest distance at this point during the computation.

The user has additional access to the following global variables used by *olvang* :

$<N>$: number of the lines.
$<S>$: the source point.
$<G>$: the goal point.
$<P[1..N]>$, $<Q[1..N]>$: arrays holding the line endpoints.
$<U[1..N]>$: array holding the unit vectors along the lines.
$<B[1..N]>$: array holding the bend points.
$<AE[1..N]>$: array holding the angle equalities.
$<EVA>$: $<AE>$ evaluated at $X = EST$.
$<RATE>$: constant controlling the rate of convergence.
$<VARS[1..N]>$: array holding the unknown names X_i.
$<JACOB[1..N,1..N]>$: matrix holding the Jacobian.
$<EVJ>$: $<JACOB>$ evaluated at $X = EST$.

$<RATE>$ is normally 1 but may be changed by the user to obtain faster convergence. In general, there are no well-defined rules for guessing good $<EST>$ and $<RATE>$ values.

• function: *olvdis*
parameter: $<niter>$
This solves OPTIMAL LINE VISIT by optimizing the length of the shortest path as described in section 2.1. Each line is parametrized by X_i which is to be determined. Once found, X_i's are used to calculate B_i's (the bend points). The global variable $<EST>$ is a vector holding the starting estimates for the parameters $X = X_1, \cdots, X_n$ of the given lines. (This is initialized by the user before running *olvdis*.) After each iteration, *olvdis* prints three values on the terminal: $<DELTA>$, $<EPS>$, and $<DIS>$. $<DELTA>$ is an indicator which approaches to zero after each iteration if the minimization is succeeding. *Olvdis* stops after $<niter>$ iterations are carried out. Inspecting $<DELTA>$ the user decides to continue with or to abort the computation. In the former case, *olvdis* is simply continued with the last value of $<EPS>$ used as a new estimate. In the latter case, a new (and possibly more promising) $<EST>$ must be chosen. $<DIS>$ is the shortest distance at this point during the computation.

The user has additional access to the following global variables used by *olvdis*:

$<N>$: number of the lines.
$<S>$: the source point.
$<G>$: the goal point.
$<P[1..N]>$, $<Q[1..N]>$: arrays holding the line endpoints.
$<B[1..N]>$: array holding the bend points.
$<RATE>$: constant controlling the rate of convergence.
$<VARS[1..N]>$: array holding the unknown names X_i.

$<GRAD[1..N]>$: array holding the gradient.
$<EVG>$: $<GRAD>$ evaluated at $X = EST$.
$<HESS[1..N,1..N]>$: matrix holding the Hessian.
$<EVH>$: $<HESS>$ evaluated at $X = EST$.

$<RATE>$ is normally 1 but may be changed by the user to obtain faster convergence. In general, there are no well-defined rules for guessing good $<EST>$ and $<RATE>$ values.

It should be remarked that both *olvang* and *olvdis* are very fast compared to the purely algebraic *solve* function of Macsyma. Running them on several moderate size (up to 12 lines) instances of OPTIMAL LINE VISIT takes at most 0.5 CPU minutes and this performance is observable without appreciable difference for both functions. Currently, it is impossible to run *solve* on such large instances since it always fails. (It is noted however that the convergence to the correct result in the case of *olvang* and *olvdis* is dependent on the initial estimate.)

(d) Plot Utilities

The better way to use SP would be to run it on a graphics workstation (such as a SUN) which supports multiple views of text and graphics. At the time of this writing that was not possible. Thus several SP functions write output files which hold plot information about a problem solved using SP. These plot files are then copied to MTS and plotted there using the utilities described below. To copy the files from CSLAB to MTS the file transfer program "mtscom" is used. Once the files are in MTS they are compiled thus: r *ftnx scards=*filename* spunch=*foo*. This compiles the program in *filename* and writes the binary image into *foo*. To run the program type: r *foo*+*plotsys 5=*datafile* 9=*bar*. (Here "*plotsys" is the library for MTS Plot Description System.) This reads its data from *datafile* and writes the plot to *bar*. Then *bar* may be spooled using the program "*ccqueue."

A disadvantage of this approach is the disappearance of the envisioned interactive nature of SP, e.g., it is no more possible to compute and observe things at the same time. Another point is the necessity of several (but in most cases only slightly different) programs for plot files obtained from different SP functions. The following Fortran plot utilities handle three important cases:

• utility: *drawpp*
This accepts the output of Franz function *prettyplt* as input and plots it. The format of the input file is described in part (b) under *prettyplt* .

• utility: *drawunf*
This accepts the output of Franz function *wunfold* as input and plots it. The format of the input file is described in part (b) under *wunfold.*

• utility: *drawvor*
This accepts te output of Franz function *pltvor* as input and plots it. The format of the input file is described in part (b) under *pltvor* .