1997

Collision-free path planning

Shiang-Fong Chen
Iowa State University

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Collision-free path planning

by

Shiang-Fong Chen

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Mechanical Engineering

Major Professor: James H. Oliver

Iowa State University

Ames, Iowa

1997

Graduate College
Iowa State University

This is to certify that the Doctoral dissertation of

Shiang-Fong Chen

has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Major Professor

Signature was redacted for privacy.

For the Major Program

Signature was redacted for privacy.

For the Graduate College

TABLE OF CONTENTS

ACKNOWLEDGEMENTS v

ABSTRACT vi

1. INTRODUCTION 1
   - 1.1 Background 1
   - 1.2 Overview 3
   - 1.3 Organization of This Study 4

2. LITERATURE REVIEW 5
   - 2.1 Hierarchical Approximate Cell Decomposition Approach 5
   - 2.2 Voronoi Diagram Approach 6
     - 2.2.1 Standard Voronoi diagram 7
     - 2.2.2 Moving object is a disc and obstacles are polygons 9
     - 2.2.3 Moving object and obstacles are polygons 10
     - 2.2.4 Approximating generalized Voronoi diagram 10
     - 2.2.5 Translation and rotation 10
   - 2.3 Potential Field Approach 11
   - 2.4 Network Representation Approach 11
     - 2.4.1 Translation 11
     - 2.4.2 Translation and rotation 13
   - 2.5 Other Approaches 15
   - 2.6 Conclusion 16

3. DATA STRUCTURES 17
   - 3.1 Edge Information 17
   - 3.2 Vertex Classification 18
   - 3.3 Vertex Information 20
   - 3.4 Intersection Information 21
   - 3.5 Relative Position of a Vertex to an Edge 22

4. PASSAGE-NETWORK CONSTRUCTION FOR ONE ROTATION LEVEL 23
4.1 Methodology Overview 23
4.1.1 Contour construction 23
4.1.2 Free-space slicing 25
4.1.3 Network construction for a single level 25
4.2 C-space Obstacle Construction 27
4.3 Edge and Vertex Information Setting 27
4.4 Slicing Procedure 30
4.4.1 Updating information 30
4.4.1.1 Updating the edges in the current slab 31
4.4.1.2 Updating the intersection information and vertex information .31
4.4.2 Contour finding 34
4.4.2.1 Detail of the contour finding algorithm 35
4.4.2.2 Contour vertex lying on some edge 37
4.4.3 Plane graph 46
4.4.4 Detail of the slicing algorithm 48
4.5 Network Construction for a Single Level 50
4.5.1 Data structures 50
4.5.2 Network construction 54

5. 3D PASSAGE NETWORK CONSTRUCTION 56
5.1 Proper Rotation Links 56
5.2 3D Network Construction 59
5.2.1 Cell finding 59
5.2.2 3D network algorithm 61
5.3 Motion Planning Algorithm 62

6. RESULTS AND CONCLUSIONS 64
6.1 Implementation and Comparisons 64
6.2 Conclusions and Discussions 75

APPENDIX. EXACT DESCRIPTION OF THE B-VORONOI DIAGRAM OF A HOMOTHETIC ROBOT MOVING THROUGH TWO OBSTACLES .77

REFERENCES 94

ACKNOWLEDGEMENTS

This dissertation could not be accomplished without many people's assistance. First, I am grateful to my advisor, Professor James Oliver for his support, trust, and guidance in many ways. I also thank him for his valuable advice and full support in my job-hunting. I also appreciate the discussions with Professor David Fernandez-Baca. His precious comments and help made this work possible. I am also thankful to Professor Daniel Ashlock, Professor James Bernard, and Professor Donald Flugrad for serving on my Committee.

The support and humor of my colleagues at the Iowa Center for Emerging Manufacturing Technology also enhanced the life in the basement of Black Engineering.

I also appreciate the prayers and warm caring of the brothers and sisters of the church. I also thank my parents for their encouragement and love. Finally, I would like to thank my Lord Jesus Christ for His unsearchable riches and bountiful supply.

ABSTRACT

Motion planning is an important challenge in robotics research. Efficient generation of collision-free motion is a fundamental capability necessary for autonomous robots.

In this dissertation, a fast and practical algorithm for moving a convex polygonal robot among a set of polygonal obstacles with translations and rotations is presented. The running time is $O(c((n+k)N+n\log n))$, where c is a parameter controlling the precision of the results, n is the total number of obstacle vertices, k is the number of intersections of configuration space obstacles, and N is the number of obstacles, decomposed into convex objects. This dissertation exploits a simple 3D passage-network to incorporate robot rotations as an alternative to complex cell decomposition techniques or building passage networks on approximated 3D C-space obstacles.

A common approach in path planning is to compute the Minkowski difference of a polygonal robot model with the polygonal obstacle environment. However such a configuration space is valid only for a single robot orientation. In this research, multiple configuration spaces are computed between the obstacle environment and the robot at successive angular orientations spanning π. Although the obstacles do not intersect, each configuration space may contain intersecting configuration space obstacles (C-space obstacles). For each configuration space, the algorithm finds the contour of the intersected C-space obstacles and the associated passage network by slabbing the collision-free space. The individual configuration spaces are then related to one another by a heuristic called "proper links" that exploit spatial coherence. Thus, each level is connected to the adjacent levels by proper links to construct a 3D network. Dijkstra's algorithm is used to search for the shortest path in the 3D network. Finally, the path is projected onto the plane to show the final locus of the path.

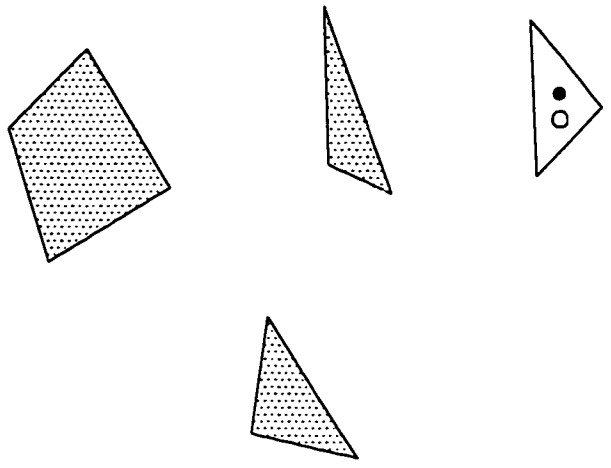
1. INTRODUCTION

1.1 Background

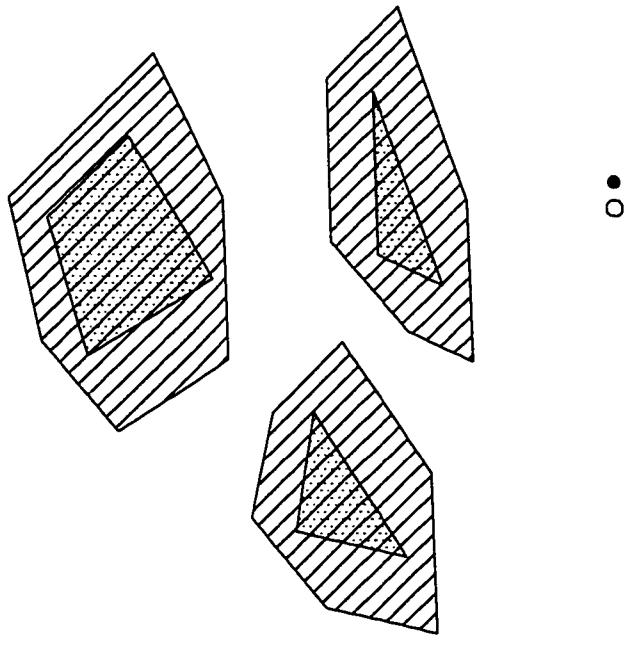
Motion planning is a major problem in robotics. The objective is to plan a collision-free path for robots moving through a workspace populated with obstacles [1-102]. Efficient generation of collision-free motion is a fundamental capability necessary for autonomous robots. The typical goal is to specify a desired function at a very high level, then allow the robot to plan and execute the motion by itself.

The concept of *configuration space*, presented by Lozano-Perez in 1983 [54], is widely used in motion planning. A *configuration* of a robot R is the description of any placement of R in the workspace by a set of independent parameters that characterize the position of a reference point fixed in R. The configuration space is the space of all configurations of R in the workspace. The configuration space for planar polygons is three dimensions, while that of solid polyhedra is six dimensions, including three translations and three rotations. If the robot is a polygon in $\Re^2$, the configuration of the robot is specified by (x, y, θ), where (x, y) is the position of the reference O of the object and θ is its rotation. If the orientation of the robot is fixed, (x, y) is sufficient to specify the configuration.

Those regions of the configuration space which are not reachable by the robot are referred to as *configuration space obstacles* (also called *C-space obstacles*). The complement of the C-space obstacles in the environment is called *free space* (*FP*). Thus, the configuration space approach considers the robot as a single point and the obstacles as "expanded fat obstacles". The expanded fat obstacles are the configuration space obstacles. Thus, the motion planning problem is reduced to moving a single point among the configuration space obstacles as an alternative to moving a 2D object among the polygonal obstacles. For example, in Figure 1.1 (a), the obstacles are the dot shaded objects and the robot is a triangular object with reference



(a) obstacle environment



(b) configuration space

Figure 1.1. Obstacle environment and configuration space

point O. This environment can be reduced to Figure 1.1 (b) where the obstacles are the fat shaded objects, C-space obstacles, and the robot has been shrunken to a point O.

Simply speaking, the motion-planning problem can be stated as follows:

Given an initial configuration R_1 and a goal configuration

R_2 of a robot, determine whether there exists a collision-free

motion trajectory to move the robot from R_1 to R_2.

If so, plan such a motion.

1.2 Overview

This work is built upon the slabbing method proposed by Ahrikencheikh and Seireg [1], which finds an optimal motion for *a point* among a set of *non-overlapping* obstacles. Here, we extend the slabbing method to the motion planning of *a convex polygonal robot* with translations and *rotations*, which also allows *overlapping* configuration space obstacles.

The *contour* is the boundary of the union of a set of intersected C-space obstacles. Successive configuration spaces are computed for every δ radians of angular rotation spanning from $-\pi/2$ to $\pi/2$. Each δ is referred to as a *rotation interval*, and the successive configuration spaces are referred to as *rotation levels*. The remaining orientations are symmetric to the range $[-\pi/2, \pi/2]$, and therefore need not be considered. The individual configuration spaces are then related to one another by a heuristic called *proper rotation link* that exploits spatial coherence to construct a 3D network.

The major steps of the algorithm are as follows.

Begin

Step 1: Find the contour of the intersected C-space obstacles.

Step 2: Find the associated passage network for each rotation level.

Step 3: Connect each rotation level to construct a 3D network.

Step 4: Search for the shortest path in the 3D network.

End

This algorithm has been fully implemented and the experimental results show that it is more robust and faster than other approaches.

1.3 Organization of This Study

The following sections are organized as follows. Chapter 2 gives a literature review. Chapter 3 introduces the data structures used by the algorithm. Chapter 4 presents the algorithm for finding the contour of the intersected C-space obstacles and the algorithm for slabbing *FP*. Chapter 5 describes and analyzes the algorithm for constructing the 3D passage network. Chapter 6 gives implementation results and conclusions.

2. LITERATURE REVIEW

There are several methods that have been investigated in the past, which build on the configuration space approach, to find a path. Some of these are reviewed in the following sections.

2.1 Hierarchical Approximate Cell Decomposition Approach

Hierarchical approximate cell decomposition is one of the most popular approach to path planning [5], [9], [20], [21], [27], [41], [42], [70], [102]. It can deal with both translations and rotations. The concept of this approach is very simple. Configuration space is divided into rectangloid cells with edges parallel to the axes of the space. Cells are labeled as EMPTY or FULL depending on whether they lie entirely outside or entirely inside the C-space obstacles. Those cells being partially inside the configuration obstacle are labeled as MIXED. A 2-D example is shown in Figure 2.1. At each level of approximation, a search algorithm is used to find a set of EMPTY rectangloid cells connecting the initial and goal configurations. If such EMPTY set cannot be found, some MIXED cells are subdivided into smaller cells, and then are labeled as EMPTY, FULL, or MIXED. Another search for a sequence of EMPTY cells is executed again. This iterative process ends when a path is found or no path can be found through the EMPTY cells of greater than the prespecified size.

D. Zhu and J.-C. Latombe speed the algorithm by using *bounding and bounded approach* to decompose MIXED cells which generates a much smaller MIXED area and a larger EMPTY/FULL area [102]. M. Barbehenn and S. Hutchinson improve the algorithm by using a *dynamically maintained single-source shortest path tree* which is based on the idea that the connectivity graph changes slightly at each iteration [5].

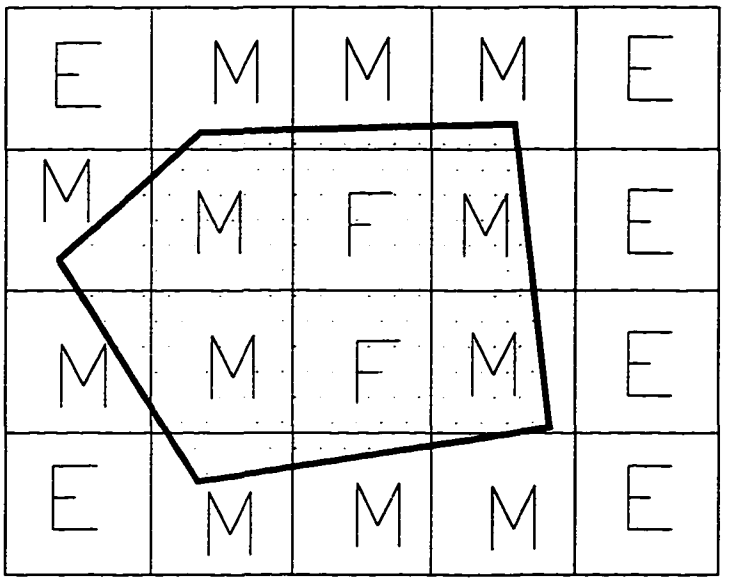


Figure 2.1. 2D cell decomposition

2.2 Voronoi Diagram Approach

Some researchers approach motion planning problems with a *Voronoi diagram* [4], [14], [25], [39], [52], [62], [63], [64], [71], [75], [76], [78], [89]. The Voronoi diagram approach is one of the *road map* methods. A Road map is a graph of highways, along which it is safe to travel, between the obstacles. With such a road map, a motion can be found by moving the robot from the start configuration to some nearby highway. Then, the robot follows the map to somewhere near the goal configuration and leaves the highway and moves to the goal configuration.

The Voronoi diagram approach is usually used in planning a *high-clearance* motion. Voronoi diagrams partition the plane into several regions. Those regions are called *Voronoi cells*. Each cell associates with one unique closest point or object of a given obstacle set S, so the Voronoi diagram is the locus which are equidistant to at least two obstacles. The vertices of the diagram, which are equidistant to more than two obstacles, are called *Voronoi vertices*.

Voronoi diagrams have many applications, for example, in the field of robotics, computer graphics, motion planning, biology, and geography and so on.

The *retraction* method is used with the Voronoi diagram approach. The term *retraction* corresponds to a continuous map from a topological space X to a subset A of X such that every point of A is mapped onto itself and every point in $X - A$ is mapped onto some point in A [60]. After the configurations R_1 and R_2 of the robot are given, a retraction of R_1 and R_2 onto configurations R_1' and R_2' on the Voronoi diagram can be computed. If R_1' and R_2' are connected by some path entirely on the Voronoi diagram, the robot can move from R_1 to R_2.

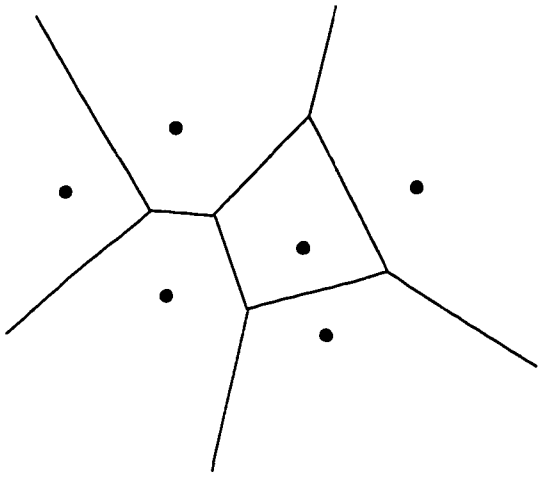
Different kinds of Voronoi diagrams are reviewed in the following sections.

2.2.1 Standard Voronoi diagram

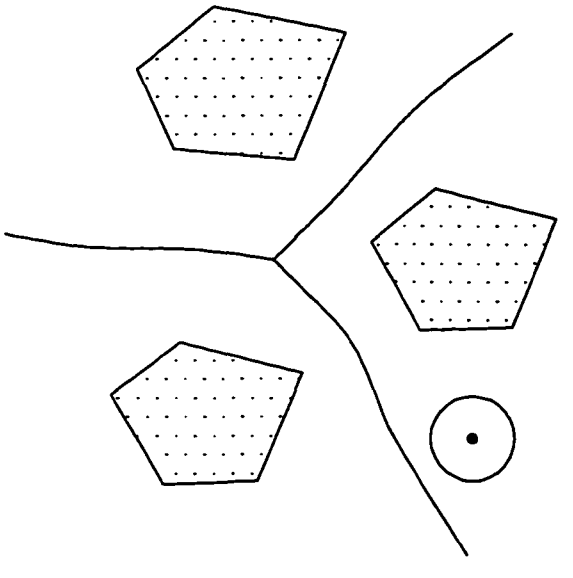
If the obstacles are points in a plane, the standard Voronoi diagram of those points partitions the plane into several convex polygonal regions (see Figure 2.2 (a)). Given two points, p_i and p_j, the set of points closer to p_i than to p_j is the half-plane containing p_i that is defined by the perpendicular bisector of $\overline{p_i p_j}$. Let us denote this half-plane by $H(p_i, p_j)$. The points closer to p_i than to any other point, which is denoted by $V(i)$, is the intersection of N - 1 half-planes. That is

$$V(i) = \bigcap_{i \neq j} H(p_i, p_j).$$

$V(i)$ is called the *Voronoi polygon associated with* p_i. The line segments are called *Voronoi edges*. The Voronoi diagram of a set of N points in the plane can be constructed in $O(N\log N)$ time [68]. After the Voronoi diagram is computed, the robot can trace the Voronoi edges to produce a high-clearance path.

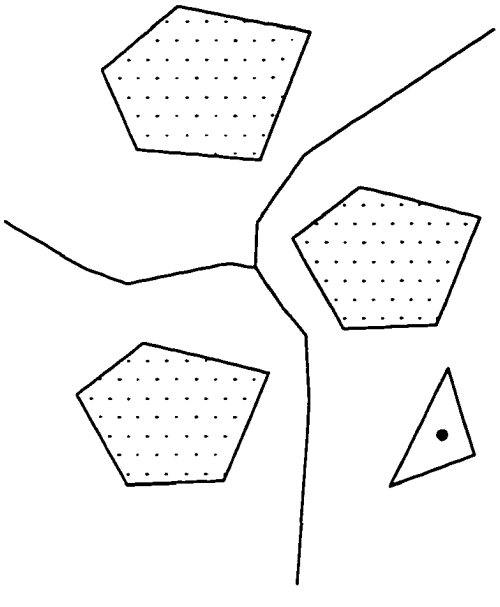


(a) obstacles are points



(b) moving object is a disc

Figure 2.2. Voronoi Diagram



(c) moving object is a polygon

Figure 2.2. (continued)

2.2.2 Moving object is a disc and obstacles are polygons

When the moving object is a disc, the diagram is the loci of the centers of all maximal circumscribed circles (also called *external skeleton* [38]), and the partitions of the plane will be smooth curves (see Figure 2.2 (b)). D. Kirkpatrick gives an $O(n\log n)$ time algorithm to construct the skeleton of arbitrary n-line polygonal figures [38]. Since the moving object is a disc and the radius of the disc can be adjusted to touch at least two obstacles, the loci of the external skeleton are equidistant to at least two obstacles in the Euclidean metric. This method can be combined with the configuration space approach. If a high clearance motion is required for moving a polygonal robot, configuration space can be computed first to shrink the robot to a point, then find the external skeleton of the configuration space. Thus, if the robot is moved along the external skeleton, it will always have the highest clearance to the obstacles.

2.2.3 Moving object and obstacles are polygons

If the moving object and the obstacles are polygons, and we use the convex distance function mentioned in [52] to define the distance, the Voronoi diagram of those polygons are called *B-Voronoi diagram*, and the partitions of the plane may be concave polygonal regions (see Figure 2.2 (c)). D. Leven and M. Sharir give an $O(n\log n)$ time algorithm to construct a *B*-Voronoi diagram for a purely translation motion, where n is the total number of obstacle corners. More details about the *B*-Voronoi diagram are given in the APPENDIX.

2.2.4 Approximating generalized Voronoi diagram

J. Vleugels and M. Overmars give an easier algorithm to compute an approximating Voronoi diagram [95]. They subdivide the space into primitive cells and test the distance between the obstacles and cells. Those cells having the same distance to at least two obstacles are on the Voronoi diagram. However, since the testing sequence for these cells is usually from left to right and top to bottom, this approximating algorithm has difficulty finding the connectivity relationship of the cells lying on the Voronoi diagram, after all cells are tested.

2.2.5 Translation and rotation

The Voronoi diagram approach is commonly used to plan translational motion. If the robot is allowed to rotate, the problem becomes more complicated. Chew and Kedem have developed a high-clearance motion for a convex polygonal object moving among polygonal obstacles in the plane, allowing both rotation and translation [14]. This algorithm takes $O\left(k^4 n\lambda_3(n)\log n\right)$ time, where k is the number of edges of the moving object and n is the number of corners, and edges of the obstacles and λ_3 is one of the almost-linear functions related to Davenport-Schinzel sequences. They compute the *B*-Voronoi diagram in (x, y, θ) space. The Voronoi boundaries will change gradually as θ changes to generate ruled surfaces.

They construct a skeleton which contains all the information necessary to do high-clearance motion planning. Then any search technique may be used to find the path. No implementation results are reported in their paper.

2.3 Potential Field Approach

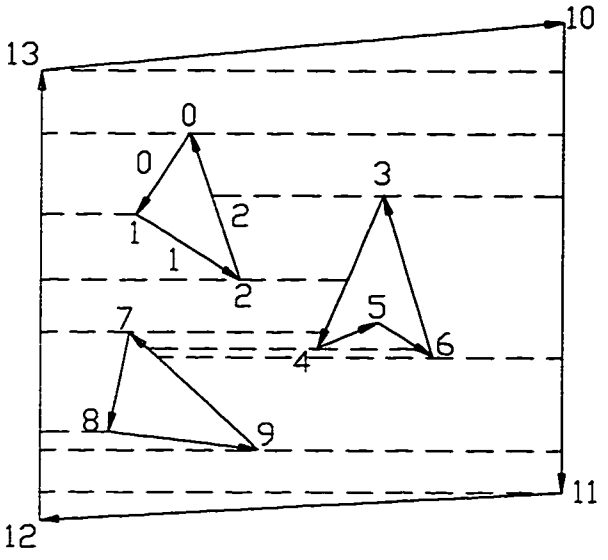
The potential field method is a completely different approach. The idea is to treat the goal configuration as an "attractive" field and the obstacles as a "repelling" field [15], [29], [32], [37], [55], [65], [97]. The motion planning is performed by repeatedly computing the most promising direction of motion, and moving in this direction by some step size. However, it is a very complex task to choose adequate potential functions and there is no guarantee that a collision-free path will always be found.

2.4 Network Representation Approach

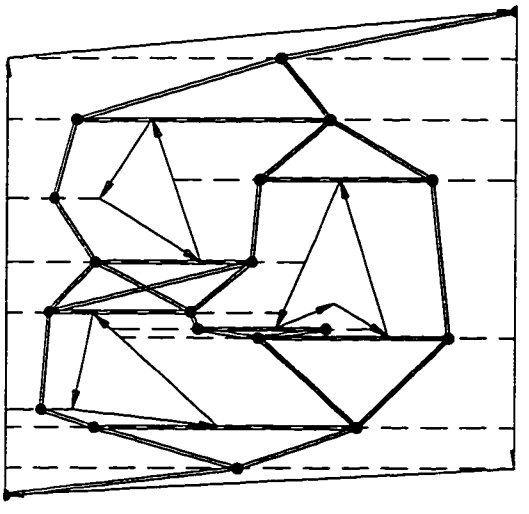
The network representation approach finds adjacency functions between the objects, and then uses any search technique to find a collision-free path embedded in the network.

2.4.1 Translation

T. S. Ku and B. Ravani use a horizontal slicing technique to construct a connectivity graph among non-overlapping polygonal objects [43]. Ahrikencheikh et al. also construct a passage network by slicing the space [1], [2]. One horizontal slicing and its associated passage network are given in Figure 2.3. Ahrikencheikh et al. construct a passage network to find the optimal and conforming motion for *a point* in a constrained plane. Their algorithm allows non-convex but *non-overlapping* obstacles. Obviously, there are no rotation problems in moving a point. They first sort all the vertices according to their descending y-coordinate order. Then they slab the FP by the horizontal lines passing through those sorted vertices. The detail of the algorithm is given below.



(a) horizontal slicing



(b) passage network

Figure 2.3. Horizontal slicing and passage network

Begin

Step 1: Sort all vertices according to their *y*-coordinate where the first vertex in the list has the highest y-coordinate.

Step 2: Initialize the red-black tree to have no edges.

Step 3: Extract the first vertex of sorted list.

Step 4: Add to red-black tree all edge(s) where one end point is the selected vertex, and other end point has lower *y*-coordinate.

Step 5: Delete from red-black tree all edge(s) where one end point is the selected vertex, and the other end point has higher *y*-coordinate.

Step 6: Horizontally slice the free space.

Step 7: If all vertices have been selected then stop; otherwise go to Step 3.

Step 8: Construct the passage network.

End

2.4.2 Translation and rotation

Ahrikencheikh et al. transform the case of a point moving in a 3 D space with stationary 3D polyhedral obstacles into the problem of a 2D polygon moving among 2D polygonal obstacles with translations and rotations. First, they build the 3D *polyhedral* C-space obstacles by computing the 2D C-space obstacles at different critical angles then connect the adjacent 2D C-space obstacles by 4-edge faces. One orientation subrange $[\theta_1, \theta_2]$ is given in Figure 2.4. Actually, this method can only compute an approximate 3D *polyhedral* C-space obstacle, since the boundary of the polyhedron should be ruled surfaces (see Figure 2.5). Thus, collisions can still occur if they try to find an optimized path through the edges of the polyhedron. Next, they construct the passage-network on the convex edges of the polyhedrons.

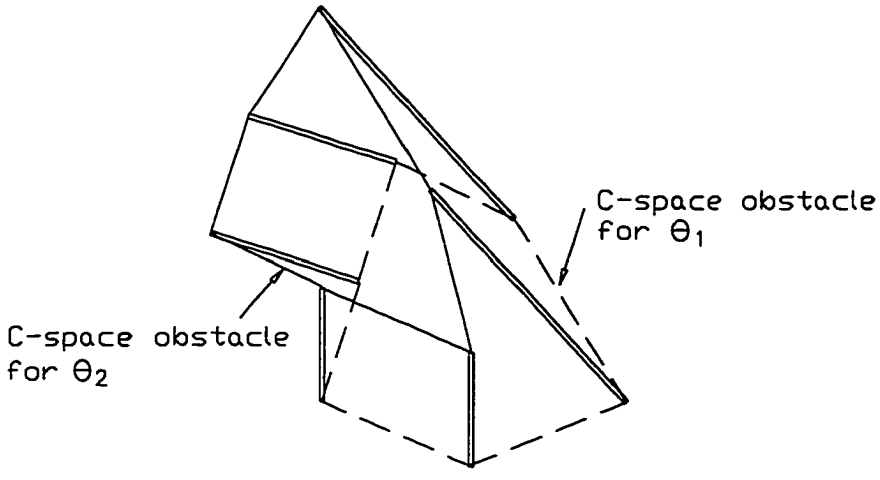


Figure 2.4. Approximate polyhedral C-space obstacle

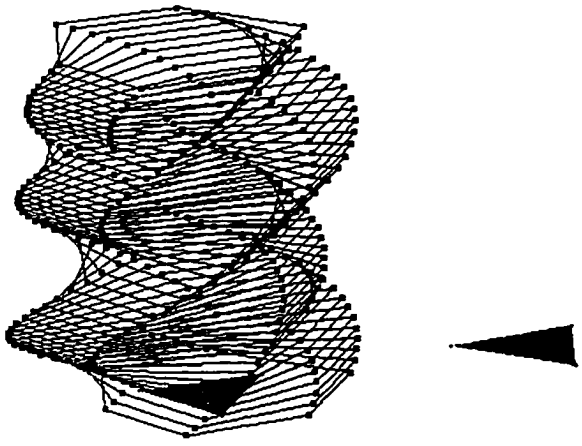


Figure 2.5. Polyhedral C-space obstacle

The convex edges are the "gates" of the possible passages. Then, they unfold the faces of the polyhedron to construct the shortest path. The algorithm takes $O(n^6)$ time to construct an optimized path. If there are many obstacles in the environment, this algorithm becomes extremely complicated and difficult to implement.

2.5 Other Approaches

J. M. Vleugels et al. combine a neural network and deterministic techniques to solve this problem [94]. The network represents *random* configurations of the robot and, from this information, constructs a road map of possible motions in the work space. The algorithm constructs a network that approximates a Voronoi diagram in configuration space. The only information required for this algorithm is whether the robot in a particular configuration intersects an obstacle. It is easily generalized to higher-dimensional configuration spaces, but there is no complexity analysis reported.

M. H. Overmars and P. Svestaka use a probabilistic learning approach to solve motion planning [67]. They split the motion planning process into two phases: the learning phase and the query phase. In the learning phase they construct a probabilistic roadmap in configuration space. This roadmap is a graph where nodes correspond to randomly chosen configurations in free space and edges correspond to simple collision-free motions between the nodes. In the query phase they use the road map to find paths between different pairs of configurations. This method can be applied on free flying robots, planar articulated robots, and car-like robots.

Lozano-Perez use a slicing technique to find a path within a θ rotation range [54]. He divides the complete range of θ values into k smaller ranges, approximates the C-space obstacles of those ranges, and then projects them onto the x-y plane. These slice projections are the C-space obstacles of the area swept out by the moving object over the range of orientations of the slice. Since the swept area under rotation of a polygon is not polygonal, the swept area is approximated by the union of polygons. Then, visibility graphs are used to find a path.

Because the slice projections are approximations of the C-space obstacles, this algorithm is not guaranteed to find a solution.

H. Martinez-Alfaro uses B-spline and simulated annealing methods to plan collision-free paths for robots [56]. He models objects with minimum surrounding area or volume ellipsoid shape. A cost function is developed for the simulated annealing algorithm. The algorithm can get a smooth path by using B-spline curves. However, it is slow.

Takahashi and Schilling used heuristic techniques to find a path for moving a rectangle by generalized Voronoi diagrams (GVD) [89]. Two reference points on the mobile object, corresponding to the front and rear wheels of an automobile, trace the shortest GVD path. This method is also computationally intensive and it only allows *rectangular* moving objects.

2.6 Conclusion

Some of the algorithms reviewed above are hard to implement when the environment is complicated, e.g. Voronoi diagram approach, cell decomposition approach, building passage-networks on 3D polyhedral C-space obstacles, etc. Some of them can not guarantee the existence of a path, e.g. potential fields approach. Some of them have high computational complexity, e.g. cell decomposition approach and Voronoi diagram approach. Actually, most approaches have their own advantages and disadvantages. Thus, the user needs to choose the approach most suitable for the application.

This study tries to find a fast and easily implementable algorithm to solve the motion planning problem. This work takes the advantage of 2D cases to solve the 3D cases. This not only simplifies the implementation, but also facilitates efficient computation.

3. DATA STRUCTURES

The algorithm requires that no two vertices have the same *y*-coordinates. Thus, the obstacle environment is assumed to be surrounded by a skewed bounding box. The region outside the bounding box is treated as an obstacle. The edges in such a C-space obstacle are ordered *clockwise*. The edges in other C-space obstacles are ordered *counterclockwise* (see Figure 3.1). The *Target vertices* and *target edges* are the vertices and edges currently under consideration.

3.1 Edge Information

The edge information of the C-space obstacles is stored in an array *einfo*, which contains the fields *vtop*, *vbottom*, and *object*. It contains the information corresponding to the edge's top vertex, i.e. the vertex with a higher *y*-coordinate, its bottom vertex, i.e. the vertex with a lower *y*-coordinate, and the object which the edge belongs to, respectively. The information for edge *j* is stored in the *j*-th entry of array *einfo*. The declaration of this structure is:

```
struct EdgeInfo{
        int vtop, vbottom;
        int object;
}.
```

The index of an edge vector is the same as the index of its start vertex. The edges in the current horizontal slice are stored in *eInCurrentSlab*. The field of the node in *eInCurrentSlab* containing the index of an edge is referred to as *eindex*.

For example, in Figure 3.1, edge *6* is in object *1* and the two end vertices of edge *6* are vertices *6* and *7* and vertex *6* has a higher *y*-coordinate. Thus, the edge information for edge *6* is:

$$einfo[6].vtop = 6;\ einfo[6].vbottom = 7;\ einfo[6].object = 1.$$

Similarly, the edge information for edge *22* is:

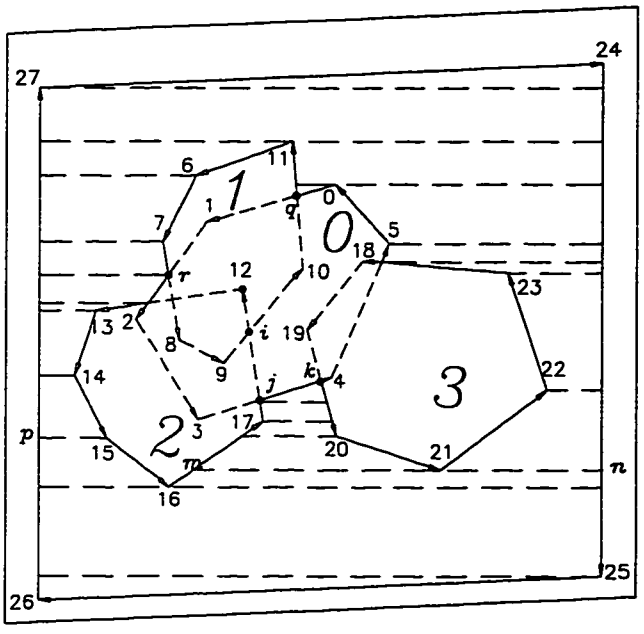


Figure 3.1. Intersected C-space obstacles

einfo[22].*vtop* = *23*; *einfo*[22].*bottom* = *22*; *einfo*[22].*object* = *3*.

The information for *eInCurrentSlab* when the slabbing goes down to vertex *0* is:

$$26 \Leftrightarrow 6 \Leftrightarrow 10 \Leftrightarrow 24 .$$

The information of the three fields in *vinfo* are obtained before any slabbing. The information in *eInCurrentSlab* are obtained and updated during the slabbing procedure.

3.2 Vertex Classification

The vertices in the configuration space are classified into six types.

1) Up_convex

A vertex is *up_convex* if and only if the vertex is convex and its two adjacent vertices both have lower *y*-coordinates.

For example, in Figure 3.1, vertices *11*, *0*, *18*, and *12* are "up_convex".

2) Down_convex

A vertex is *down_convex* if and only if the vertex is convex and its two adjacent vertices both have higher *y*-coordinates.

For example, in Figure 3.1, vertices *9*, *3*, *21*, and *16* are "down_convex".

3) Up_concave

A vertex is *up_concave* if and only if the vertex is concave and its two adjacent vertices both have higher *y*-coordinates.

For example, in Figure 3.1, vertices *q*, *26*, etc. are "up_concave".

4) Down_concave

A vertex is *down_concave* if and only if the vertex is concave and its two adjacent vertices both have lower *y*-coordinates.

For example, in Figure 3.1, vertices *24*, *k*, etc. are "down_concave".

5) Left

A vertex is *left* if and only if its front vertex has a lower *y*-coordinate and its back vertex has a higher *y*-coordinate.

For example, in Figure 3.1, vertices *6*, *7*, *r*, *19*, *20*, etc. are "left".

6) Right

A vertex is *right* if and only if its front vertex has a higher *y*-coordinate and its back vertex has a lower *y*-coordinate.

For example, in Figure 3.1, vertices *5*, *23*, *22*, *j*, *10*, etc. are "right".

When the target vertex is on the contour and if it is a "down_concave" or "up_concave" vertex, there is no slice going through it. If the contour vertex is "up_convex" or "down_convex", there is one slice going from it to its closest right and closest left edges. If the contour vertex is "left", there is one slice going from it to its closest left edge. If the contour vertex is "right", the slice goes from it to its closest right edge.

3.3 Vertex Information

The algorithm requires that no two vertices have equal y-coordinates, including the vertices of the C-space obstacles and the intersections of the C-space obstacles. The information of the vertices, including the vertices in C-space obstacles and the intersections of the C-space obstacles, is stored in an array *vinfo*. Based on the data structure developed by Ahrikencheikh and Seireg [1], *vinfo* contains the fields *vfront*, *vback*, *vleft*, *vright*, *efront*, *eback*, *eleft*, and *eright* which contain the information corresponding to its front vertex, its back vertex, its left vertex, its right vertex, its front edge, its back edge, its left edge, and its right edge, respectively. The "front" and "back" are the relative sequences of the edges or vertices in the C-space obstacles. The "left" and "right" are the relative positions of the edges or vertices in the horizontal slicing. Since the C-space obstacles are the "expanded fat obstacles" of the original obstacles, they might overlap although the real obstacles do not overlap. Besides the above fields, one more field *oncontour* is needed, which is a boolean value indicating whether the target vertex is on the contour or not. The vertex information for vertex j is stored in the j-th entry of the array *vinfo*. The declaration of the structure is:

```
struct VertexInfo {
        int   vfront, vback;
        int   vleft, vright;
        int   efront, eback;
        int   eleft, eright;
        Boolean oncontour;
        }.
```

If the vertex does not have right vertex, right edge, left vertex, or left edge, the values of the corresponding fields are set to be -1.

All the vertices in C-space obstacles are sorted by non-increasing y-coordinate order and their indices are inserted in that order into a linked list *ylist*. The field of the node in *ylist* con-

taining the index of a vertex is referred to as *vindex*.

For example, in Figure 3.1, the vertex information for vertex *21* is:

vinfo[21].*vfront* = *22*; *vinfo*[21].*vback* = *20*;

vinfo[21].*vleft* = *m*; *vinfo*[21].*vright* = *n*;

vinfo[21].*efront* = *21*; *vinfo*[21].*eback* = *20*;

vinfo[21].*eleft* = *16*; *vinfo*[21].*eright* = *24*;

vinfo[21].*oncontour* = TRUE.

The vertex information for vertex *15* is:

vinfo[15].*vfront* = *16*; *vinfo*[15].*vback* = *14*;

vinfo[15].*vleft* = *p*; *vinfo*[15].*vright* = *-1*;

vinfo[15].*efront* = *15*; *vinfo*[15].*eback* = *14*;

vinfo[15].*eleft* = *26*; *vinfo*[15].*eright* = *-1*;

vinfo[15].*oncontour* = TRUE.

The information in *ylist* is:

24 ⇔ 27 ⇔ 11 ⇔ 6 ⇔ 0 ⇔ 1 ⇔ 7 ⇔ 5 ⇔ 18 ⇔ 10 ⇔ 23 ⇔ 12 ⇔ 13 ⇔ 2 ⇔ 19

⇔ 8 ⇔ 9 ⇔ 14 ⇔ 4 ⇔ 22 ⇔ 3 ⇔ 17 ⇔ 20 ⇔ 15 ⇔ 21 ⇔ 16 ⇔ 25 ⇔ 26

Only *vfront*, *vback*, *efront*, and *eback* fields are set before the slicing procedure. The remaining fields are determined when the slabbing procedure is processed.

3.4 Intersection Information

Since the C-space obstacles might intersect, and the slicing lines intersect some edges, each edge needs a linked list *ptonEdge* to store its intersection information. The first element of the list is the highest vertex of the edge, and the last element is its lowest vertex.

Before slabbing the *FP*, the intersection information in *ptonEdge* for each edge has only two elements, one is the top vertex of this edge, and the other one is its bottom vertex. For example, in Figure 3.1 the intersection information for edge *17* is *ptonEdge*[17]: 12 ⇔ 17

and the intersection information for edge *3* is *ptonEdge*[3]: $4 \Leftrightarrow 3$. After the slabbing procedure, edge *17* has been determined to have two intersection points, i and j, so the intersection information for edge *17* becomes *ptonEdge*[17]: $12 \Leftrightarrow i \Leftrightarrow j \Leftrightarrow 17$. The intersection information for edge *3* becomes *ptonEdge*[3]: $4 \Leftrightarrow k \Leftrightarrow j \Leftrightarrow 3$.

3.5 Relative Position of a Vertex to an Edge

For a given edge E with top vertex (x_1, y_1) and bottom vertex (x_2, y_2), a given vertex $v\,(x, y)$ is in the *positive* x-direction of E if the cross product

$$(x - x_2, y - y_2) \otimes (x_1 - x_2, y_1 - y_2) > 0.$$

Vertex v is in the *negative* x-direction of this edge if

$$(x - x_2, y - y_2) \otimes (x_1 - x_2, y_1 - y_2) < 0.$$

Vertex v is on edge E if

$$(x - x_2, y - y_2) \otimes (x_1 - x_2, y_1 - y_2) = 0.$$

For example, in Figure 3.1 if the current slice is the one passing through vertex *19*, the edges in *eInCurrentSlab* are *24*, *22*, *4*, *9*, *17*, *7*, *2*, *13*, and *26*. Vertex *19* is in the positive x-direction of edges *9*, *17*, *7*, *2*, *13*, and *26*, but it is in the negative x-direction of edges *24*, *22*, and *4*.

4. PASSAGE-NETWORK CONSTRUCTION FOR ONE ROTATION LEVEL

The complement of the area enclosed by the contour in the configuration space is the free space. Since the passage network is constructed in the free space, the information of the free space should be determined first (see Figure 4.1). In this chapter, algorithms for finding the contour of a set of intersected C-space obstacles and constructing the network for a single rotation level are given.

4.1 Methodology Overview

The passage network construction consists of two major steps:

Begin

Step 1: Find the contour and slice the free space;

Step 2: Construct the passage network for a single level.

End

4.1.1 Contour construction

In this study, the contour of a set of intersected C-space obstacles is constructed by using a slabbing technique, which takes $O((n+k)N)$ time, where n is the total number of obstacle vertices, k is the number of intersections, and N is the number of obstacles, decomposed into convex objects. The slabbing technique is used instead of Kedem and Sharir's $O(n\log^2 n)$ algorithm [35] (as shown in Figure 4.2), for two reasons. First, the slabbing method is simple, as shown by its extensive use in solving geometric intersection problems [61], [66]. Second, the passage network construction is based on slabbing, and thus the contour construction can be conveniently carried out simultaneously. The second reason is that experiments show that

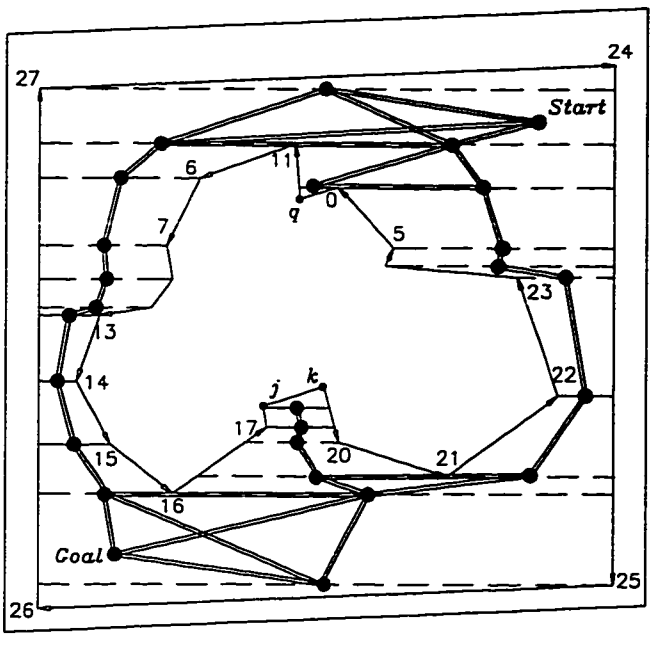


Figure 4.1. The contour and the passage network of Figure 3.1

Divide and conquer algorithm for calculation of $\cup K_i$. (K_i is the C-space obstacle)

Step 1. Calculate and preprocess all the K_i's.

Step 2. Recursively find $G = \underset{i \in g}{\cup} K_i$ and $H = \underset{i \in h}{\cup} K_i$, where

$$g = \{1, \ldots, |m/2|\}, \; h = \{|m/2| + 1, \ldots, m\}.$$

Step 3. Find the contour of $K = G \cup H$, using the Ottmann-Widmeyer-Wood approach [66].

Figure 4.2. Kedem and Sharir's Algorithm for constructing a contour.

finding the contours with this method takes less than 1% of the total running time. Thus, attempting to optimize this step does not pay off.

Some vertices of the contour are from the C-space obstacles, e.g., vertices *11, 6, 7*, etc. in Figure 4.1, while others are from the intersections of the C-space obstacles, e.g., vertices *j, k*, etc. The contour may contain several disjoint regions, see, e.g., Figure 4.3. In order to find all contour vertices and their adjacency relations, the edges in ***eInCurrentSlab***, the intersection information in *ptonEdge*, and the vertex information in *vinfo* need to be updated during the slicing.

4.1.2 Free-space slicing

The free space is divided by the horizontal sweeping lines going through the contour vertices to its closest positive and/or negative x-direction obstacle edges. Using the terminology used by Ahrikencheikh and Seireg [1], the intersections of the sweeping lines and the obstacle edges are called "secondary" vertices. The vertices of C-space obstacles and the intersections of the C-space obstacles, which are on the contour, are called "primary" vertices (see Figure 4.3).

Thus, the free-space slicing procedure can be defined as finding the secondary vertices associated with their primary vertices. And the gate is the segment corresponding to one primary vertex and one of its secondary vertices. By the hypothesis that no two vertices have equal y-coordinates, each slab can have only one, two, three, or four gates (see Figure 4.4). The gates are the possible passages for the moving object.

4.1.3 Network construction for a single level

The mid-point of the gate is the node of the network. The network is constructed by connecting the nodes of adjacent gates together (see Figure 4.1). Such a network is called a "passage network". The x-y distance of the connected nodes is the weight of the link.

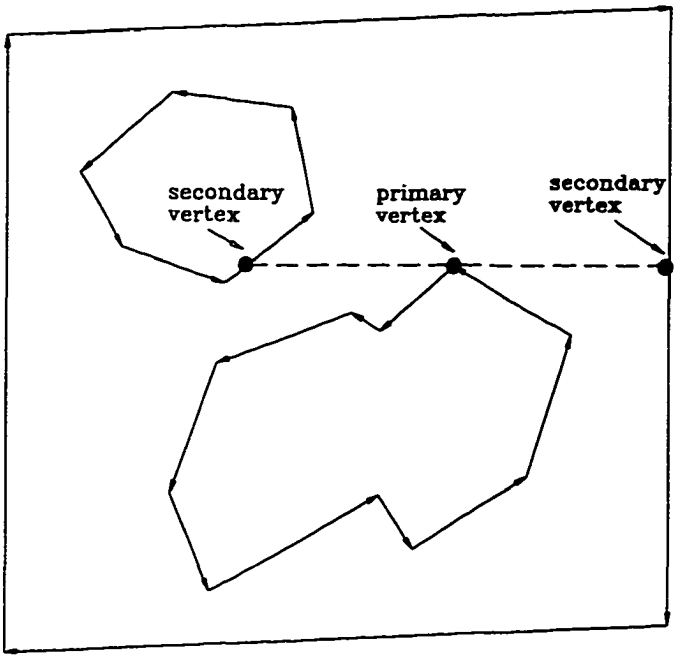


Figure 4.3. Primary and secondary vertices

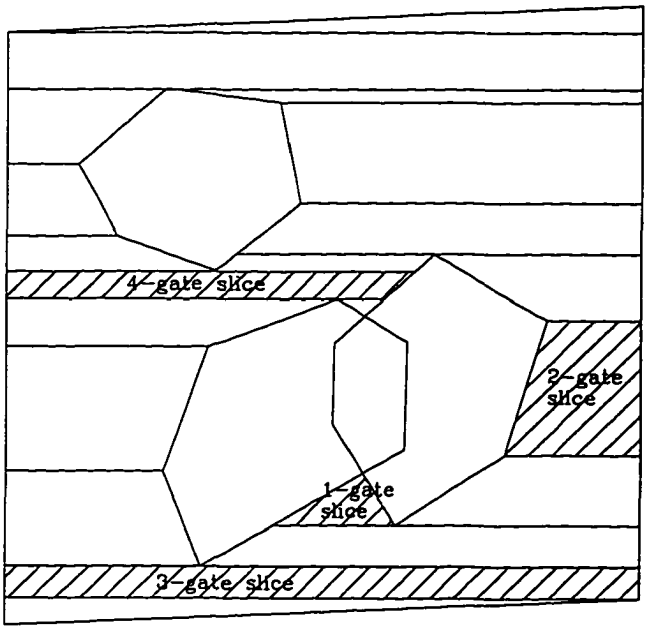


Figure 4.4. Gates

4.2 C-space Obstacle Construction

The first step of the motion planning algorithm is to find the C-space obstacles.

Let $A_1, \ldots, A_N$ be N convex polygonal obstacles, and let B be a convex robot. If B is rotated around the origin by 180°, it is denoted by B^N (see Figure 4.5). The C-space obstacles are obtained by the Minkowski sum of A_i and B^N [26]:

$$A_i + B^N = \{a + b | a \in A_i, b \in B^N\} \ , \ i = 1 \ldots N.$$

The complement of the C-space obstacles is the collision-free configuration space.

One easier way to compute the C-space obstacles is to view each edge of a polygon as a vector directed counterclockwise around the polygon. Then, the edges of $A_i + B^N$ are the edges in A_i and B^N merged in their slope order (see Figure 4.6). Actually, $A_i + B^N$ can also be obtained by moving the reference point O of B^N around the boundary of A_i (see Figure 4.7).

All vertices in the C-space obstacles are sorted according to non-increasing y-coordinate and inserted into the linked list *ylist* in that order.

4.3 Edge and Vertex Information Setting

After the C-space obstacles have been computed, the information for the edges and vertices must be set. Before the slabbing procedure, only *vfront*, *vback*, *eback*, and *efront* fields of *vinfo* and *vtop*, *vbottom*, and *object* fields of *einfo* are assigned. The information in *eInCurrentSlab* is empty. The information in *ptonEdge* for each edge is only its top and bottom vertices as described in Chapter 3.

All edges and vertices are numbered sequentially. Thus, if the index of the last edge of C-space obstacle k is i, the first edge of C-space obstacle k+1 will be (i+1). The detail of the algorithm for setting edge and vertex information is described as follows.

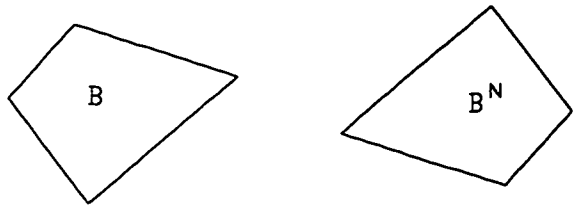


Figure 4.5. B and B^N

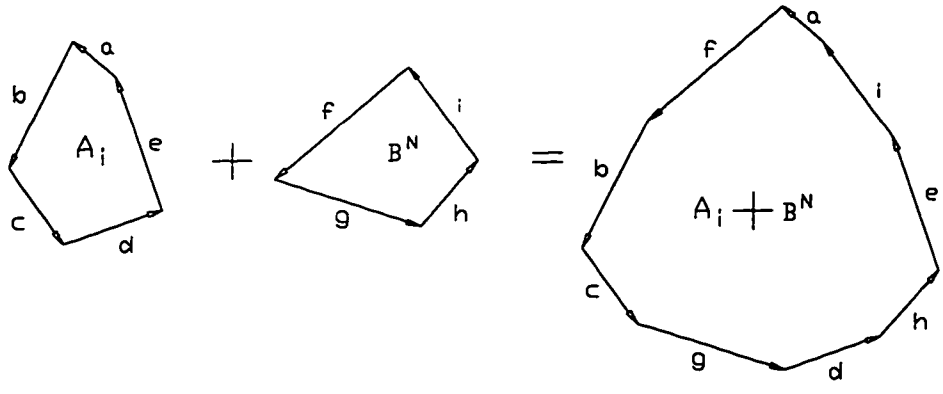


Figure 4.6. Merged in slope order

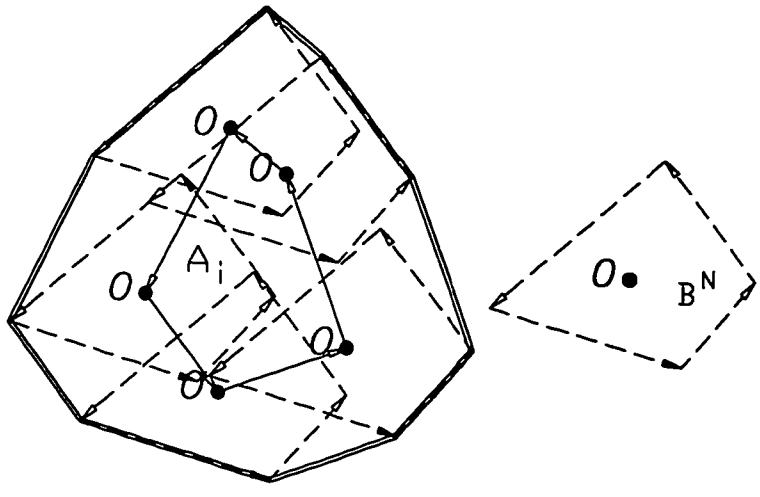


Figure 4.7. B^N Moves around A_i

```
Algorithm Set_E_V_Info
Begin
1. n = 0; /* the index of the first target edge and target vertex*/
2. for each C-space obstacle c do
3.    for each vertex v in c do
4.          einfo[n].object = c;
5.          vinfo[n].efront = n;
6.          if v is the last vertex of c then
7.              vinfo[n].vfront = the first vertex of c;
8.              vinfo[n].vback = n - 1;
9.              vinfo[n].eback = n -1;
10.             if the y-coordinate of vertex n < the y-coordinate of the first vertex of c do
11.                 einfo[n].top = the first vertex of c;
12.                 einfo[n].bottom = n;
13.             else
14.                 einfo[n].top = n;
15.                 einfo[n].bottom = the first vertex of c;
16.         else
17.             vinfo[n].vfront = n + 1;
18.             if the y-coordinate of vertex n < the y-coordinate of vertex n+1 then
19.                 einfo[n].top = n + 1;
20.                 einfo[n].bottom = n;
21.             else
22.                 einfo[n].top = n;
23.                 einfo[n].bottom = n + 1;
24.             if v is the first vertex of c then
```

25. *vinfo*[*n*].*vback* = the last vertex of *c*;

26. *vinfo*[*n*].*eback* = the last edge of *c*;

27. **else**

28. *vinfo*[*n*].*vback* = *n* -1;

29. *vinfo*[*n*].*eback* = *n* -1;

30. increment *n* by 1;

End

4.4 Slicing Procedure

The slicing procedure partitions *FP* into several triangles or quadrilaterals, each of which is referred to as a *cell*. The boundaries of the cells which do not belong to any C-space obstacles are the *gates*. There are at most four gates in one cell. The slicing procedure processes the elements obtained by merging the vertices of the C-space obstacles with the intersections of the C-space obstacles by decreasing *y*-coordinate. The major steps of this procedure is as follows.

Begin

Step 1. **for** every vertex v_i, including the vertices in the C-space obstacles and intersections of the C-space obstacles **do**

Step 2. **if** v_i is on the contour **then**

Step 3. slice *FP* through v_i;

End

4.4.1 Updating information

The horizontal slicing procedure slices the *FP* by non-increasing order of the vertices on the contour. In what follows, the information that needs to be updated during the slicing is described.

4.4.1.1 Updating the edges in the current slab

For a given target vertex, we examine if there is any edge adjacent to the target vertex and below the horizontal line which passes through the target vertex. If so, the edge is inserted into *eInCurrentSlab*. If there is any edge adjacent to the target vertex and above the horizontal line, then the edge is deleted from *eInCurrentSlab*. For example, in Figure 4.8, when the slicing procedure goes from vertex *7* to vertex *5*, since edge *5* is above the horizontal line, which passes through vertex 5, and edge *4* is below the line, edge *5* is deleted from *eInCurrentSlab* and edge *4* is added into *eInCurrentSlab*. Thus, the information in *eInCurrentSlab* is updated from $24 \Leftrightarrow 26 \Leftrightarrow 10 \Leftrightarrow 5 \Leftrightarrow 1 \Leftrightarrow 7$ to $24 \Leftrightarrow 26 \Leftrightarrow 10 \Leftrightarrow 1 \Leftrightarrow 7 \Leftrightarrow 4$.

4.4.1.2 Updating the intersection information and vertex information

If the inserted edge intersects any edge currently in *eInCurrentSlab*, the intersection point is inserted into a list *intlist* ordered by non-increasing *y*-coordinate. For example, in Figure 4.8, when vertex *7* is the target vertex, edge *7* is inserted into *eInCurrentSlab*, and it is determined to intersect with edge *1* at point *a*. At this moment *intlist* has only one element *a*. When vertex *18* is the target vertex, edges *18* and *23* are inserted into *eInCurrentSlab*. Edge 23 is intersected with edge 4 at point *b*, so point *b* is inserted into *intlist*. Since point *b* has a higher *y*-coordinate, the information in *intlist* becomes $b \Leftrightarrow a$. Once the intersection vertex in *intlist* has been the target vertex, this element is deleted from *intlist*. If an intersection point is on the contour, *FP* will be sliced through this intersection point, e.g., points *r* and *j* in Figure 4.10. The intersection information *ptonEdge* of the two intersected edges and the vertex information *vinfo* about the vertices on the two edges will be updated.

If the intersection point is not on the contour, there is no need to update this information. Four intersection cases are shown in Figure 4.9. The bold segments in Figure 4.9 are the portions on the boundaries of the contour in which the intersection point is on the contour. For case 1 (resp. case 3) of Figure 4.9, there will be a slice that goes from the intersection point to

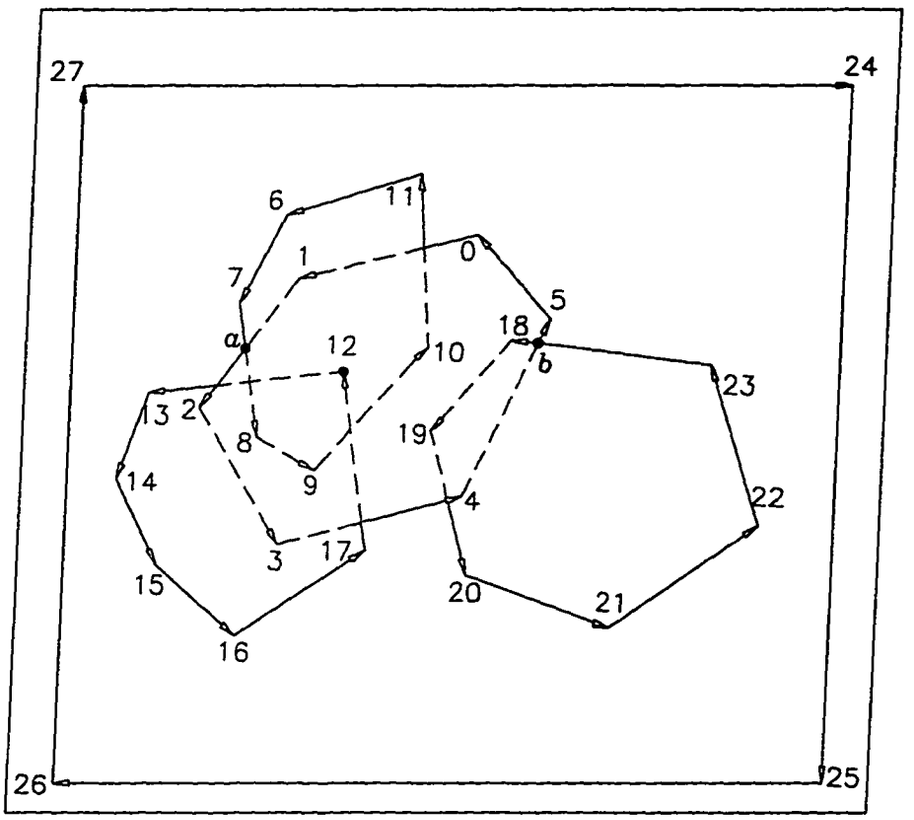


Figure 4.8. One environment example

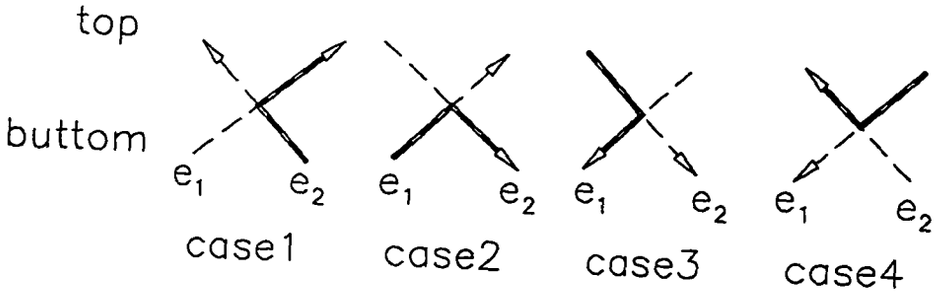


Figure 4.9. Four intersection cases

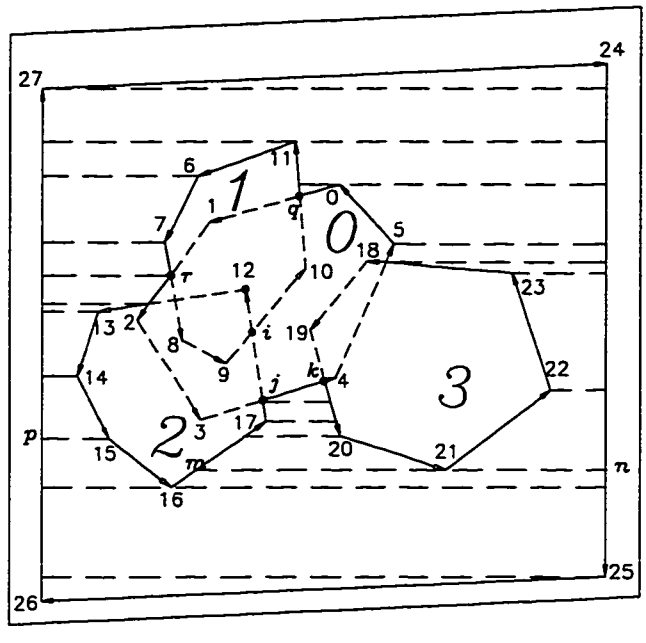


Figure 4.10. Intersected C-space obstacles

the closest right (resp. left) edge. Case 2 and case 4 are degenerate cases with no slicing, since the intersection vertex is concave.

For example, in Figure 4.10, when the slicing procedure goes down to vertex *4*, edge *3* is inserted into *eInCurrentSlab* and edge *4* is deleted from *eInCurrentSlab*. Edge 3 is tested for intersection with edges *19* and *17* at k and j respectively in the current slab, so k and j are stored in *intlist*. When the slabbing procedure goes down to point k, point k is tested to determine if it is on the contour. Since point k is on the contour, it is now necessary to determine what kind of intersection it is, and it is found to be a case 2 intersection. Thus, there is no slice passing through point k. The front vertex of vertex *3* is then updated from vertex *4* to vertex k, the back vertex of vertex k is *3*, the front vertex of k is *20*, and the back vertex of *20* is updated from *19* to k. The intersection information for edge *3* becomes *ptonEdge*[*3*]: $4 \Leftrightarrow k \Leftrightarrow 3$. After vertex k has been the target vertex, it is deleted from *intlist*.

When the slabbing procedure goes down to point *j*, point *j* is tested to determine if it is on the contour. Since point *j* is on the contour and it is a case 1 intersection, there is a slice from vertex *j* to its closest right edge *19*. The front vertex of *17* becomes *j*, the back vertex of *j* is *17*, the front vertex of *j* is *k*, and the back vertex of *k* is updated from 3 to *j*. The intersection information for edge *3* becomes *ptonEdge*[*3*]: $4 \Leftrightarrow k \Leftrightarrow j \Leftrightarrow 3$. Updating all the information for one target vertex takes only constant time.

4.4.2 Contour finding

Since the C-space obstacles are convex "closed loops", each C-space obstacle in the current slab contributes two edges to *eInCurrentSlab*. For example, in Figure 4.10, when the slicing procedure goes to vertex *19*, the objects in the current slab are objects *0*, *1*, *2*, and *3*, and the edges in *eInCurrentSlab* are $24 \Leftrightarrow 26 \Leftrightarrow 4 \Leftrightarrow 7 \Leftrightarrow 9 \Leftrightarrow 22 \Leftrightarrow 17 \Leftrightarrow 13 \Leftrightarrow 2 \Leftrightarrow 19$. Edges *24* and *26* are from the bounding box, edges *4* and *2* are from object *0*, edges *7* and *9* are from object *1*, edges *17* and *13* are from object *2*, and edges *19* and *22* are from object *3*. Thus, if there are *N* objects, there are at most *2N* edges in *eInCurrentSlab*.

For a given target vertex in *ylist* or *intlist*, and two given edges in *eInCurrentSlab* which belong to the same C-space obstacle O_i. If the target vertex is in the positive *x*-direction of both edges or in the negative *x*-direction of both edges, this vertex is outside of O_i. On the other hand, if the target vertex is in the positive *x*-direction of one edge and in the negative *x*-direction of another edge, this vertex is inside O_i, and thus it is not on the contour.

For example, in Figure 4.10, vertex *19* is in the positive *x*-direction of edges *7* and *9*, so it is outside of object *1*. However, vertex *19* is in the positive x-direction of edge *2*, but it is in the negative *x*-direction of edge *4*, so vertex *19* is inside object *0*. Since vertex *19* is inside some object, it is not on the contour.

The major steps for testing whether a vertex is on the contour are described as follows.

Begin

Step 1. mark the target vertex as "on the contour";

Step 2. **for** every pair of edges e_i and e_j in *eInCurrentSlab* which belong to the same C-obstacle **do**

Step 3. **if** the target vertex is in the different x-direction of e_i and e_j **then**

Step 4. mark the target vertex as "not on the contour";

End

4.4.2.1 Detail of the contour finding algorithm

If the target vertex is outside of the C-space obstacle of the bounding box, the target vertex is not on the contour either. If the target vertex is on the contour, we need to find its closest edges for later slicing. The closest edges can be found by the same procedure.

If the target vertex is in the positive x-direction of edge E and edge E belongs to object i, set $obstacle[i] = P$. If the target vertex is in the negative x-direction of edge E, set $obstacle[i] = N$. If the target vertex is neither in the positive nor in the negative x-direction of E, it lies on the edge. In the next section, there is more discussion about the situation where the target vertex lies on some edge.

Procedure *isonContour* determines whether the target vertex is on the contour and computes the closest right and left edges of the target vertex. If the target vertex is not on the contour, its closest right and left edges are set to be -1. Procedure *isonContour* tests all vertices, including the vertices on the C-space obstacles and the intersections of the C-space obstacles.

Procedure isOnContour

Input: the index of the target vertex, *target_vertex*;

Begin

1. *vinfo*[*target_vertex*].*oncontour* = TRUE;

2. **if** *target_vertex* is outside of the C-space obstacle of the bounding box **then**

```
3.    vinfo[target_vertex].oncontour = FALSE;
4.    stop;
5. for every edge e_i in eInCurrentSlab do
6.  if (e_i does not belong to the bounding box) and (target_vertex is not the end vertex of
      e_i) and (if target_vertex is an intersection of two edges, e_i is not one of the edges)
      then
7.          which_obj = einfo[e_i].object;
8.          if target_vertex is in the positive x-direction of e_i then
9.              if obstacle[which_obj] == N then
10.                 vinfo[target_vertex].oncontour = FALSE;
11.                 closest right edge of target_vertex = -1;
12.                 closest left edge of target_vertex = -1;
13.                 stop;
14.             else obstacle[which_obj] = P;
15.                 update the closest left edge of target_vertex;
16.         else if target_vertex is in the negative x-direction of e_i then
17.             if obstacle[which_obj] == P then
18.                 vinfo[target_vertex].oncontour = FALSE;
19.                 closest right edge of target_vertex = -1;
20.                 closest left edge of target_vertex = -1;
21.                 stop;
22.             else obstacle[which_obj] = N;
23.                 update the right closest edge of target_vertex;
24.         else /* target_vertex is on e_i */
25.             onedge = e_i;
End;
```

Procedure *isonContour* takes $O(N)$ time in the worst case to process a target vertex. After the procedure, those vertices whose *oncontour* fields are TRUE are the vertices on the contour, and they are the "primary" vertices mentioned earlier. The closest right (resp. left) vertex for a left (resp. right) primary vertex is set to be -1. If the target vertex is found to be on the contour, *FP* will be sliced by a horizontal line passing through that vertex to its closest left and/or closest right edge(s) according to the vertex type.

4.4.2.2 Contour vertex lying on some edge

If the contour vertex lies on some edge, it might change the type and the closest right and/or left edge(s) of this vertex. Since we assume that no two vertices have the same y-coordinates, when we consider that the vertex lies on some edge, we cannot have the following cases: for one primary vertex, it cannot lie on the end vertex of another edge, as shown in Figure 4.11 (a). The situation is the same for the intersection point (see Figure 4.11 (b)). By assumption, there is no edge going through the intersection point. If the intersection point lies on another edge, then two or more intersection points overlap (see Figure 4.11 (c)).

Here, s denotes the start point of the edge on which the target vertex lies, and a is its end point. The target vertex is denoted by t, its back vertex by b, and its front vertex by c (see Figure 4.12). Different types of vertices lead to different kinds of situations when the vertex lies on another edge.

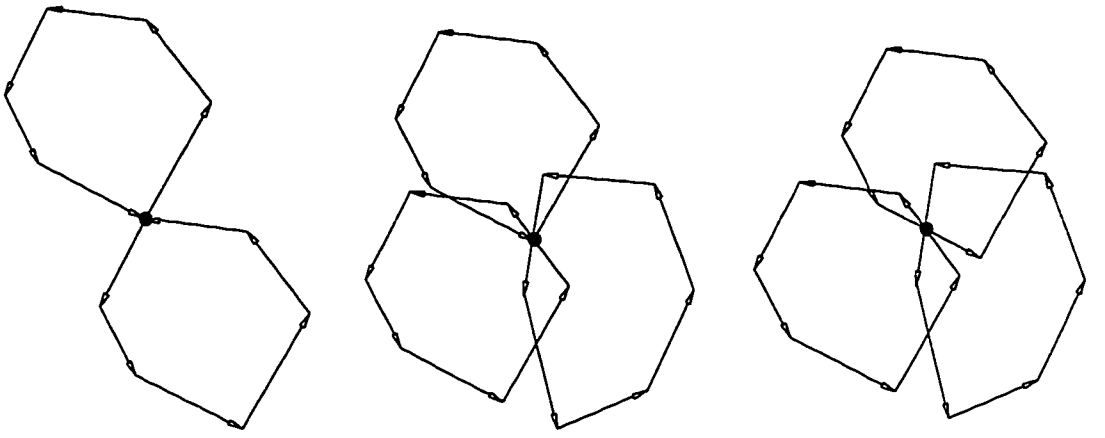
1) Up_convex

If the target is "up_convex", and it lies on some edges, it will be one of the six cases as described in Figure 4.12.

Case (1) and case (2):

If $\vec{sa} \times \vec{sb} > 0$ and $\vec{sa} \times \vec{sc} > 0$, this is case (1) or case (2).

Since the edges of the objects are ordered counterclockwise, even if t lies on edge sa, t is still counted as inside the object which owns sa. Thus, vertex t is not on the contour, so its



(a) two primary vertices (b) one primary vertex and one intersection (c) three intersections

Figure 4.11. Overlapping vertices

closest right and left edges are reset to -1.

Case (3):

If $\vec{sa} \times \vec{sb} < 0$, $\vec{sa} \times \vec{sc} < 0$, and vertex c is in the positive x-direction of edge sa, it is case (3).

Since the edges of the objects are ordered counterclockwise, even if t lies on edge sa, t is still counted as outside the object which owns sa. Thus, vertex t is on the contour, and its closest left edge is updated to edge sa, and its closest right edge is still the one obtained from Procedure *isOnContour*. In this case, the distance between the target vertex t and its closest left edge is 0.

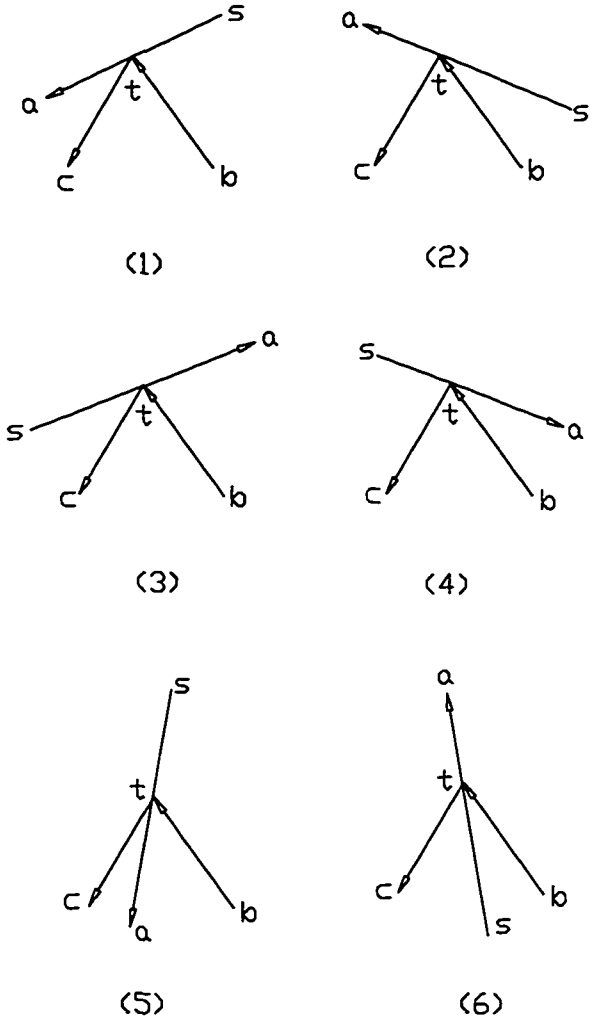


Figure 4.12. Up_convex

Case (4):

If $\vec{sa} \times \vec{sb} < 0$, $\vec{sa} \times \vec{sc} < 0$, and vertex c is in the negative x-direction of edge sa, it is case (4).

In Case (4), vertex t is on the contour as it was in Case (3), except its closest right edge is updated to edge sa, and its closest left edge is still the one obtained from Procedure *isOnContour*. The distance between the target vertex t and its closest right edge is 0.

Case (5):

If $\vec{sa} \times \vec{sb} > 0$ and $\vec{sa} \times \vec{sc} < 0$, it is case (5).

Since the objects are ordered counterclockwise, in this case, vertex *b* is inside the object which owns *sa*, and vertex *c* is outside that object. The type of vertex *t* is updated from "up_convex" to "left". Thus, its closest right edge is set to -1. The back vertex of *t* is updated from *b* to *s*, *vinfo*[*t*].*vback* = *s*. The front vertex of *s* is updated from *a* to *t*, *vinfo*[*s*].*vfront* = *t*.

Case (6):

If $\vec{sa} \times \vec{sb} < 0$ and $\vec{sa} \times \vec{sc} > 0$, it is case (6).

Similarly to case (5), in this case the type of vertex *t* is updated from "up_convex" to "right". Thus, its closest left edge is set to -1. The front vertex of *t* is updated from *c* to *a*, *vinfo*[*t*].*vfront* = *a*. The back vertex of *a* is updated from *s* to *t*, *vinfo*[*a*].*vback* = *t*.

2) Down_convex

If the target is "down_convex", and it lies on some edges, there are also six cases as described in Figure 4.13.

Case (1):

If $\vec{sa} \times \vec{sb} < 0$, $\vec{sa} \times \vec{sc} < 0$, and vertex *c* is in the positive *x*-direction of edge *sa*, it is case (1).

Since the edges of the obstacles are ordered counterclockwise, even if *t* lies on edge *sa*, *t* is still counted as outside the object which owns *sa*. Thus, vertex *t* is on the contour, and its closest left edge is updated to edge *sa*.

Case (2):

If $\vec{sa} \times \vec{sb} < 0$, $\vec{sa} \times \vec{sc} < 0$, and vertex *c* is in the negative *x*-direction of edge *sa*, it is case (2).

In Case (2), vertex *t* is on the contour as it was in Case (1), except its closest right edge is updated to edge *sa*.

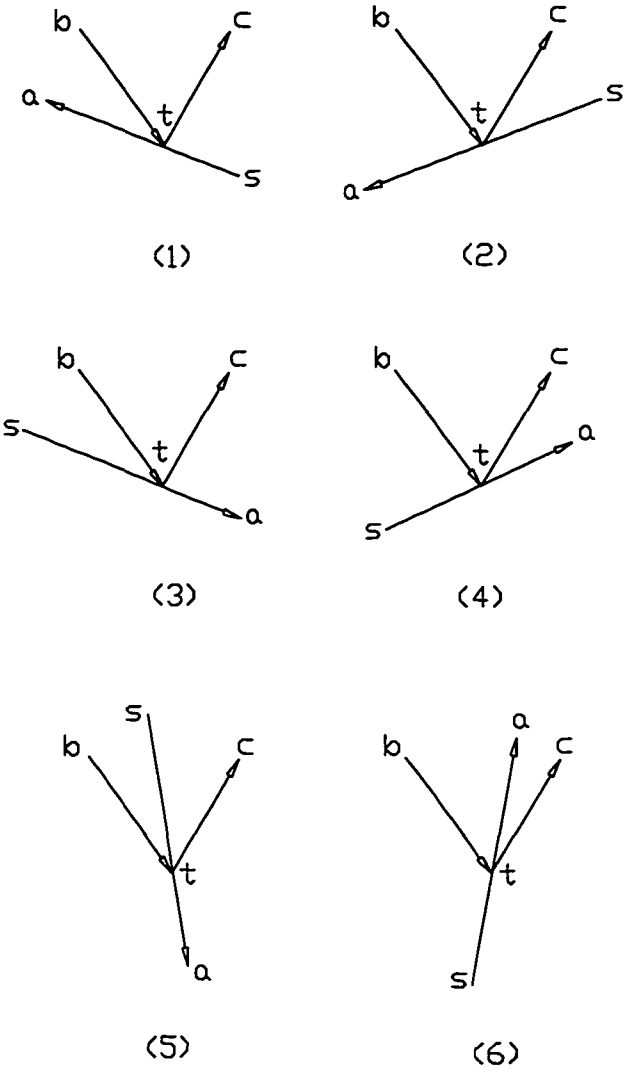


Figure 4.13. Down_convex

Case (3) and case (4):

If $\overrightarrow{sa} \times \overrightarrow{sb} > 0$ and $\overrightarrow{sa} \times \overrightarrow{sc} > 0$, this is case (3) or case (4).

In the two cases, t is counted as inside the object which owns sa. Thus, vertex t is not on the contour, so its closest right and left edges are reset to -1.

Case (5):

If $\vec{sa} \times \vec{sb} < 0$ and $\vec{sa} \times \vec{sc} > 0$, it is case (5).

In this case, the type of vertex *t* is updated from "down_convex" to "left". Its closest right edge is reset to be -1. The front vertex of *t* is updated from *c* to *a*, *vinfo*[*t*].*vfront* = *a*. The back vertex of *a* is updated from *s* to *t*, *vinfo*[*a*].*vback* = *t*.

Case (6):

If $\vec{sa} \times \vec{sb} > 0$ and $\vec{sa} \times \vec{sc} < 0$, it is case (6).

In this case, the type of vertex t is updated from "down_convex" to "right". Its closest left edge is set to -1. The back vertex of *t* is updated from *b* to *s*, *vinfo*[*t*].*vback* = *s*. The front vertex of *s* is updated from *a* to *t*, *vinfo*[*s*].*vfront* = *t*.

3) Left

If the target vertex is "left", and it lies on some edges, there are also six cases as described in Figure 4.14.

Case (1):

If $\vec{sa} \times \vec{sb} < 0$ and $\vec{sa} \times \vec{sc} < 0$, it is case (1).

In this case, the closest left edge of *t* is reset to be *sa*.

Case (2):

If $\vec{sa} \times \vec{sb} > 0$ and $\vec{sa} \times \vec{sc} > 0$, it is case (2).

In this case, vertex *t* is determined to be inside the object which owns edge *sa*. Thus, its closest left and right edges are reset to -1.

Case(3):

If $\vec{sa} \times \vec{sb} < 0$, $\vec{sa} \times \vec{sc} > 0$, and the *y*-coordinate of vertex *t* is higher than the *y*-coordinate of vertex *a*, it is case (3).

In this case, the front vertex of *t* is updated from *c* to *a*, *vinfo*[*t*].*vfront* = *a*. The back vertex of *a* is updated from *s* to *t*, *vinfo*[*a*].*vback* = *t*.

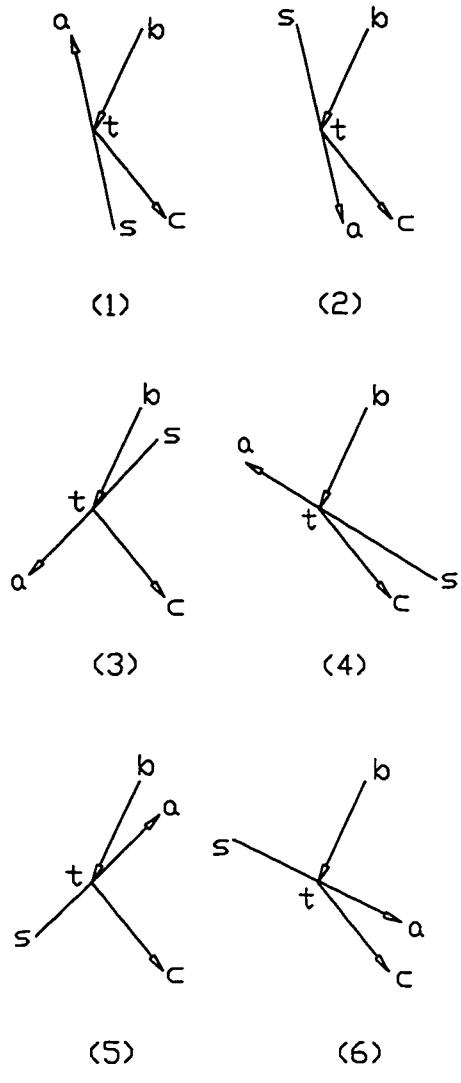


Figure 4.14. Left

Case (4):

If $\vec{sa} \times \vec{sb} < 0$, $\vec{sa} \times \vec{sc} > 0$, and the *y*-coordinate of vertex *t* is lower than the *y*-coordinate of vertex *a*, it is case (4).

In this case, the type of vertex *t* is updated from "left" to "up_concave". Thus, its closest left and right edges are updated to -1. The front vertex of *t* is updated from *c* to *a*,

vinfo[*t*].*vfront* = *a*. The back vertex of *a* is updated from *s* to *t*, *vinfo*[*a*].*vback* = *t*.

Case (5):

If $\vec{sa} \times \vec{sb} > 0$, $\vec{sa} \times \vec{sc} < 0$, and the *y*-coordinate of vertex *t* is higher than the *y*-coordinate of vertex *s*, it is case (5).

In this case, the type of vertex *t* is updated from "left" to "down_concave". Thus, its closest left and right edges are updated to -1. The back vertex of *t* is updated from *b* to *s*, *vinfo*[*t*].*vback* = *s*. The front vertex of *s* is updated from *a* to *t*, *vinfo*[*s*].*vfront* = *t*.

Case (6):

If $\vec{sa} \times \vec{sb} > 0$, $\vec{sa} \times \vec{sc} < 0$, and the *y*-coordinate of vertex *t* is lower than the *y*-coordinate of vertex *s*, it is case (6).

The back vertex of *t* is updated from *b* to *s*, *vinfo*[*t*].*vback* = *s*. The front vertex of *s* is updated from *a* to *t*, *vinfo*[*s*].*vfront* = *t*.

4) Right

If the target vertex is "right", and it lies on some edges, there are also six cases as described in Figure 4.15.

Case (1):

If $\vec{sa} \times \vec{sb} > 0$ and $\vec{sa} \times \vec{sc} > 0$, it is case (1).

In this case, vertex *t* is determined to be inside the object which owns edge *sa*. Thus, its closest right and left edges are set to -1.

Case (2):

If $\vec{sa} \times \vec{sb} < 0$ and $\vec{sa} \times \vec{sc} < 0$, it is case (2).

In this case, the closest right edge of *t* is reset to be *sa*.

Case(3):

If $\vec{sa} \times \vec{sb} < 0$, $\vec{sa} \times \vec{sc} > 0$, and the *y*-coordinate of vertex *t* is higher than the *y*-coordinate of vertex *a*, it is case (3).

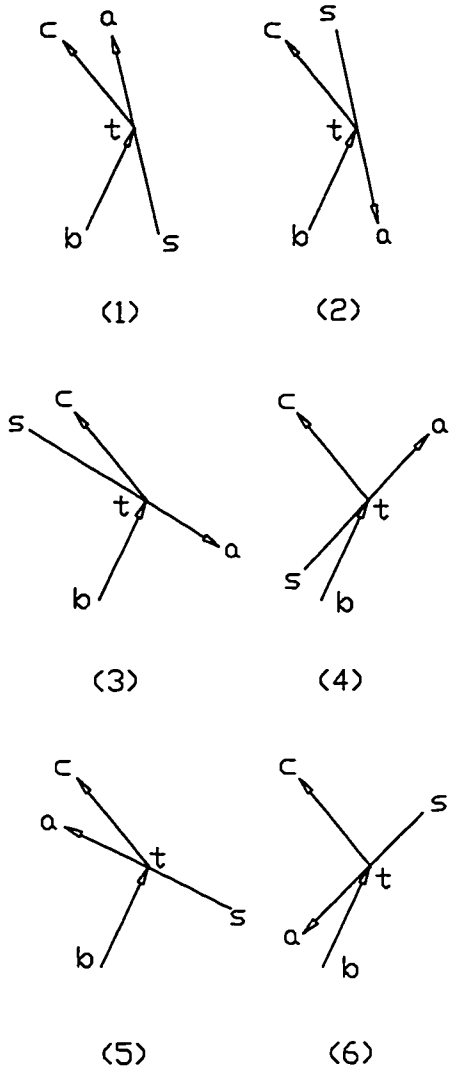


Figure 4.15. Right

In this case, the type of vertex *t* is updated from "right" to "down_concave". Thus, its closest right and left edges are updated to -1. The front vertex of *t* is updated from *c* to *a*, *vinfo*[*t*].*vfront* = *a*. The back vertex of *a* is updated from *s* to *t*, *vinfo*[*a*].*vback* = *t*.

Case (4):

If $\vec{sa} \times \vec{sb} < 0$, $\vec{sa} \times \vec{sc} > 0$, and the *y*-coordinate of vertex *t* is lower than the *y*-coordinate

of vertex a, it is case (4).

In this case, the front vertex of t is updated from c to a, $vinfo[t].vfront = a$. The back vertex of a is updated from s to t, $vinfo[a].vback = t$.

Case (5):

If $\overrightarrow{sa} \times \overrightarrow{sb} > 0$, $\overrightarrow{sa} \times \overrightarrow{sc} < 0$, and the y-coordinate of vertex t is higher than the y-coordinate of vertex s, it is case (5).

The back vertex of t is updated from b to s, $vinfo[t].vback = s$. The front vertex of s is updated from a to t, $vinfo[s].vfront = t$.

Case (6):

If $\overrightarrow{sa} \times \overrightarrow{sb} > 0$, $\overrightarrow{sa} \times \overrightarrow{sc} < 0$, and the y-coordinate of vertex t is lower than the y-coordinate of vertex s, it is case (6).

In this case, the type of vertex t is updated from "left" to "up_concave". Thus, its closest right and left edges are updated to -1. The back vertex of t is updated from b to s, $vinfo[t].vback = s$. The front vertex of s is updated from a to t, $vinfo[s].vfront = t$.

4.4.3 Plane graph

Recall that when the target vertex is on the contour and if it is a "down_concave" or "up_concave" vertex, there is no slice going through it. If the target vertex is "up_convex" or "down_convex", there is one slice going from it to its closest right and closest left edges. If the target vertex is "left", there is one slice going from it to its closest left edge. If the target vertex is "right", the slice goes from it to its closest right edge.

The graph that describes the adjacency information of the slicing is called *plane graph* (see Figure 4.16).

The intersection information and vertex information must also be updated when one slicing line intersects with one edge. For example, after the slicing procedure, the edge information for edge e_0 in Figure 4.16 is:

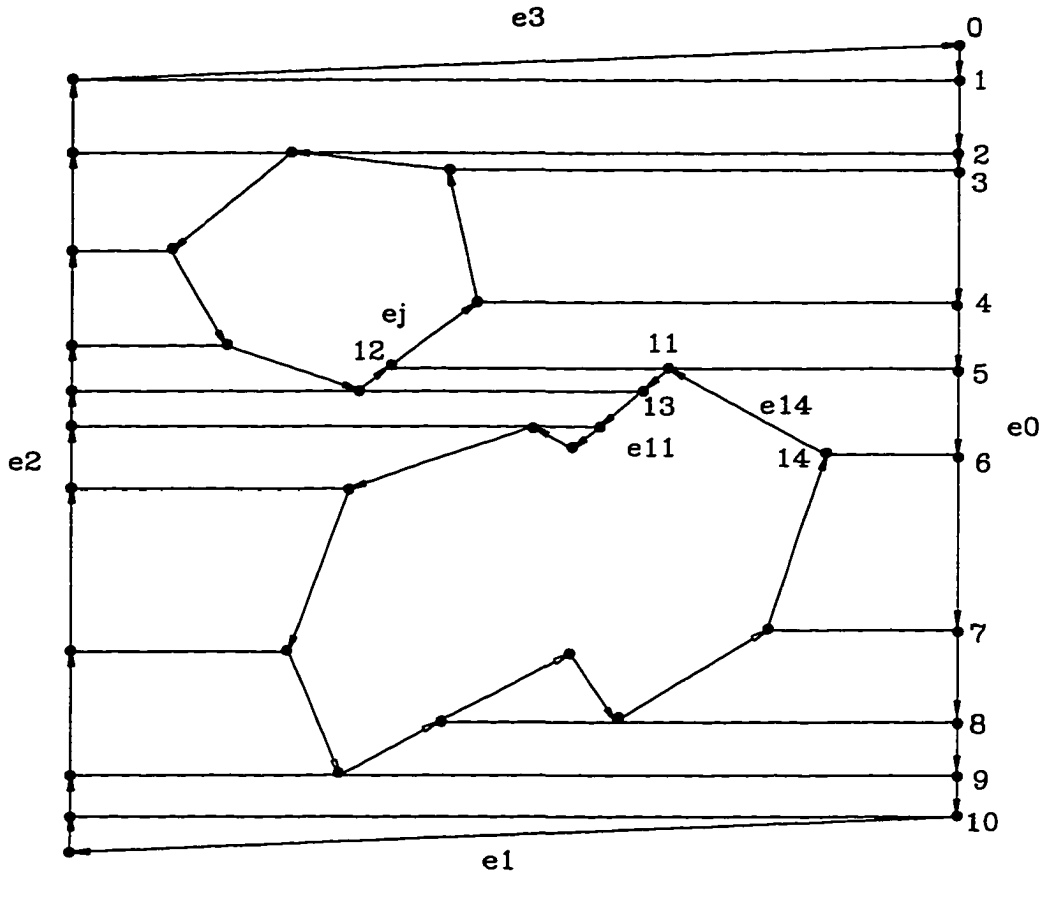


Figure 4.16. Plane-graph

ptonEdge[e_0]: 0 ⇔ 1 ⇔ 2 ⇔ 3 ⇔ 4 ⇔ 5 ⇔ 6 ⇔ 7 ⇔ 8 ⇔ 9 ⇔ 10 .

The vertex information for vertex *11* is:

vinfo[*11*].*vfront* = *13*; *vinfo*[*11*].*vback* = *14*;

vinfo[*11*].*efront* = e_{11}; *vinfo*[*11*].*eback* = e_{14};

vinfo[*11*].*eleft* = e_j; *vinfo*[*11*].*eright* = e_0;

vinfo[*11*].*vleft* = *12*; *vinfo*[*11*].*vright* = *5*.

4.4.4 Detail of the slicing algorithm

This section gives more detail about the slicing algorithm. Procedure *ContourSlice* describes the information update when a slice going through a contour vertex.

Procedure ContourSlice

Input: *target_vertex*;

Begin

1. **if** *target_vertex* is an intersection point **then**
2. Update the intersection information (the information in *ptonEdge*) for the two edges, which intersect at *target_vertex*, according to the intersection cases (see Figure 4.9);
3. Update the vertex information (front and back information) for *target_vertex* and its neighbors;
4. Draw a horizontal line passing through *target_vertex* to its closest left and (or) right edge(s) according to the vertex type of *target_vertex*;
5. Update the intersection information (the information in *ptonEdge*) of *target_vertex*'s closest edge(s), which the slicing line intersects with;
6. Update the vertex information (left and right information) for *target_vertex* and the intersection(s) of the slicing line and *target_vertex*'s closest edge(s).

End

Since each update just takes constant time, procedure *ContourSlice* takes constant time.

The slabbing procedure proceeds according to non-increasing y-coordinate order of the merge of *ylist* and *intlist*. The whole horizontal slicing procedure is described as follows.

Procedure Hslice

Begin

1. **for** every vertex v_i in *ylist* **do**

2. **if** any edge adjacent to v_i is below the horizontal line, which passes through v_i **then**
3. insert the edge into *eInCurrentSlab*;
4. **if** there are intersections between the inserted edge and the edges in *eInCurrentSlab* **then**
5. insert the intersections into *intlist*;
6. **if** any edge adjacent to v_i is above the horizontal line, which passes through v_i **then**
7. delete the edge from *eInCurrentSlab*;
8. **While** ($intlist \neq \varnothing$ and the y-coord. of the first element *fvint* of *intlist* is >= the y-coord. of v_i) **do**
9. **if** isOnContour(*fvint*) **then**
10. ContourSlice(*fvint*);
11. Delete *fvint* from *intlist*;
12. **if** isOnContour(v_i) **then**
13. ContourSlice(v_i);

End;

If the C-space obstacles do not intersect, the edges in *eInCurrentSlab* can be stored in a red-black tree structure to save computational time [1]. However, the edges in *eInCurrentSlab* may intersect, so they are just stored in a linked list here. Step 3 takes constant time to insert an edge but step 7 takes $O(N)$ time to delete an edge, so the total execution time for step 7 is $O(nN)$, since step 1 is repeated n times. It takes $O(N)$ time to find the intersections between each inserted edge and the edges in *eInCurrentSlab*. If there are any intersections, they are inserted into *intlist* by non-increasing y-coordinate. Since the total number of intersections is k, there will be $O(k)$ elements in *intlist* at any time. If the elements in *intlist* are stored in a red-black tree, step 5 will take $O(\log k)$ time for each insertion, for a total of $O(k\log k)$ time. Because k is bounded by N^2, this is $O(k\log N)$. Steps 9 and 10 take a total of $O(kN)$ time. Step

11 takes $O(k\log N)$ time. Steps 12 and 13 take $O(nN)$ time. Summing up all the time bounds, we see that procedure *Hslice* takes $O((n+k)N)$ time.

4.5 Network Construction for a Single Level

The passage network for one orientation is constructed by connecting the mid-points of the adjacent gates, which are the nodes of the network, as described by Ahrikencheikh and Seireg [1].

The network construction procedure for a single level is called ***NetworkperLevel***. Unlike the Ahrikencheikh and Seireg algorithm, we have no extra slices passing through the start and goal configurations. Instead, the start and goal configurations are connected to the nodes of the cell in which they are located (see Figure 4.1).

The major steps for this procedure are:

Begin

Step 1: Create the nodes of the passage network;

Step 2: Connect the nodes of the adjacent gates to each other;

End

4.5.1 Data structures

The nodes of the network are stored in an array *netnode*[*i*][*j*], where *i* is the level of this node, and *j* is the index of this node at level *i*. The data structure is:

```
struct Net{
    int color;
    int vleft, vright;
    int nodeid;
    int level;
    int motion;
```

float *dist*;

Net* *parent*;

Netlist* *next*;

}.

The field *color* is a flag to indicate if this node has been searched or not while we do the network searching. If *color* is 0, that means it is still unsearched. If *color* is 1, that means it has been searched. The fields *vleft* and *vright* are the left and right vertices of this node. The field *nodeid* is the index of this node in array *netnode*, which is the same as the index of the right vertex, stored in *vinfo,* of this node. The field *level* is the rotation level of this orientation in 3D network. The field *motion* indicates the motion of this robot, 1 meaning the motion of this robot from the target node to its parent node is translation first then rotation, and 0 meaning rotation first then translation. The field *dist* stores the distance between this node and the source node, and the initial value of *dist* for each node is -1. The field *parent* stores the parent node of the target node. Each node has only one parent, so after the network searching, we can just follow the *parent* pointers to find the final route. The pointer *next* is a linked list with *Netlis*t data type that stores all adjacent nodes of this target node. The target node is called *center node* with respect to its adjacent nodes. The data structure for the linked list *Netlist* is:

struct Netlist{

int *nodeid*;

int *level*;

int *motion*;

float *weight*;

Netlist* *next*;

}.

The field *nodeid* is the index of this adjacent node. The field *level* is the level of this node. The field *motion* indicates the motion of this robot, 1 meaning the motion of this robot from

the center node to this adjacent node is translation first then rotation, and 0 meaning rotation first then translation. If the two nodes are at the same rotation level, then the field *motion* is ignored. This will be described in more detail in the next chapter. The field *weight* stores the distance between the center node and its adjacent node. The pointer *next* points to the next adjacent node of the center node.

Figure 4.17 shows the relationship between the node and its adjacent nodes, where *ptr*1 to *ptr*n are the adjacent nodes of node *netnode*[*i*][*j*]. The upper half of the box is the data type of the node. For example, in Figure 4.18, suppose the parent of node n_1 of level i is node n_3 of level i-1, and the distance between node n_1 of level i to the source node is 93.5, and node n_1 of level i has been searched, then information for node n_1 of level i is:

netnode[*i*][1].*color* = 1; *netnode*[*i*][1].*vleft* = 4;

netnode[*i*][1].*right* = 1; *netnode*[*i*][1].*nodeid* = 1;

netnode[*i*][1].*level* = *i*; *netnode*[*i*][1].*motion* = 0;

netnode[*i*][1].*dist* = 93.5; *netnode*[*i*][1]->*parent* = *netnode*[*i*-1][3].

The pointer *next* points to a linked list with a *Netlist* data type which stores the adjacent nodes of *netnode*[*i*][*j*]. Since node n_1 of level i has six adjacent nodes, its *next* field is:

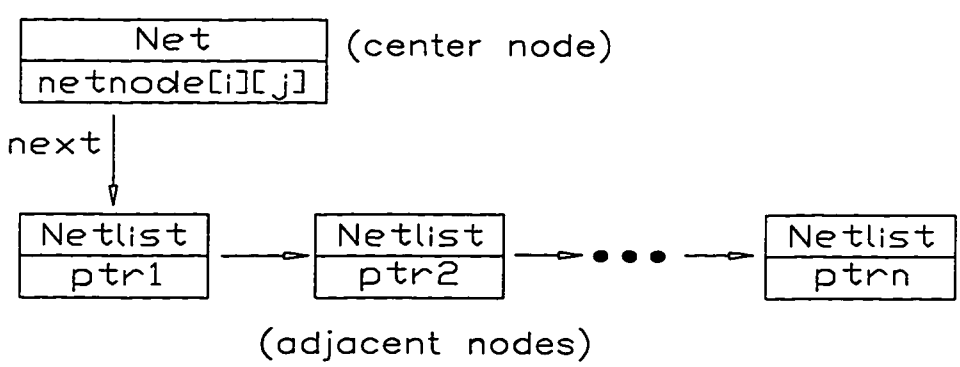


Figure 4.17. Netlist

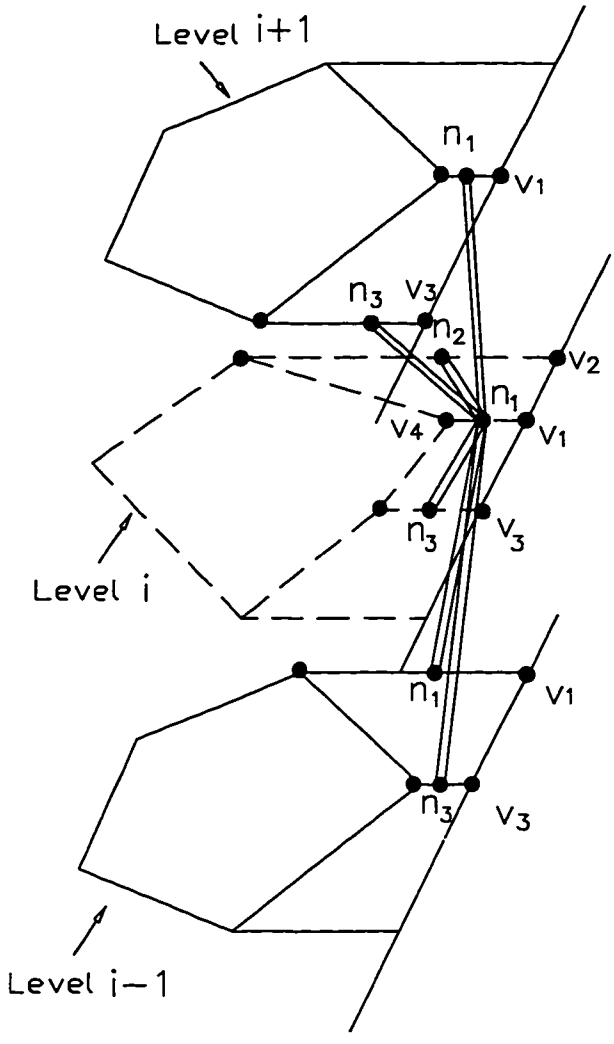


Figure 4.18. Three adjacent levels

netnode[*i*][1]->*next* = *ptr*1->*ptr*2->. . . ->*ptr*6.

The nodes from *ptr*1 to *ptr*6 are with *Netlist* data type. Suppose *ptr*1 is node n_3 of level i+1, and the distance between node n_3 of level i+1 and node n_1 of level i is 10.3, the information for *ptr*1 is:

*ptr*1.*nodeid* = 3; *ptr*1.*level* = i+1;

*ptr*1.*motion* = 0; *ptr*1.*weight* = 10.3;

*ptr*1->*next* = *ptr*2;

Similar data structures are for nodes *ptr*2 to *ptr*n.

4.5.2 Network construction

Since we construct the network for each single level first, some fields in *netnode* are not set until the whole 3D network is constructed.

For both primary and secondary vertices, those with left vertices (or right vertices) must have an associated gate. The vertices that have no left vertices (or right vertices) are the down_concave vertices, or up_concave vertices, so they do not have associated gates. We chose to find the network nodes by scanning vertices that are left vertices.

The network construction for one single level is described as follows.

Procedure NetworkperLevel

Begin

1. **for** every vertex *i* (including primary and secondary vertices) in level *lv* **do**
2. **if** vertex *i* has a left vertex *vlft* **then** /* create the nodes of the network */
3. node *i* of level *lv* = the mid-point of vertex *i* and vertex *vlft*;
4. *netnode*[*lv*][*i*].*color* = 0; *netnode*[*lv*][*i*].*level* = *lv*;
5. *netnode*[*lv*][*i*].*nodeid* = *i*; *netnode*[*lv*][*i*].*parent* = NULL;
6. *netnode*[*lv*][*i*].*vleft* = *vlft*; *netnode*[*lv*][*i*].*vright* = *i*;
7. *netnode*[*lv*][*i*].*dist* = -1; /* initial value */
8. **for** every node *nd* in level *lv* **do**
9. find every adjacent node *adjnd* of *nd* do
10. create a pointer *ptr* with *Netlist* data type;
11. *ptr*->*nodeid* = *adjnd*; *ptr*->*level* = *lv*;
12. *ptr*->*weight* = the distance between *adjnd* and *nd*;
13. insert pointer *ptr* into the linked list pointed by *netnode*[*lv*][*i*]->*next*;

End

Since all C-space obstacles are closed convex sets, there are only $O(N)$ non-convex corners on the contour [78]. That is, there are $O(N)$ intersections on the contour. Hence, the total number of vertices on the boundaries of the contour is $O(n)$. Thus, procedure *Networkper-Level* only takes $O(n)$ time, since it simply goes through the vertices of the contour and connects the adjacent nodes into a network. In Figure 4.1, the passage network is shown with bold links; the dark circles are the nodes of the network.

5. 3D PASSAGE NETWORK CONSTRUCTION

The shape of the C-space obstacles changes with the rotation angle of the moving object. Figure 5.1 shows three configuration spaces in three different robot orientations. Since the robot has been shrunken to a point, the dotted triangle in Figure 5.1 is included to indicate the orientations of the robot. When the robot is at 0° orientation, the C-space obstacles are disjoint. However, when the robot rotates 45°, some of the C-space obstacles overlap. When the robot rotates 90°, the overlapped C-space obstacles separate again. Our approach is to construct snapshots of the rotation levels for different rotation angles, and to link the levels via *proper rotation links*.

In the previous chapter, we have shown how to construct the network for a single rotation level. This chapter will describe how to connect those 2D networks into a 3D network and find a motion path for the robot. The major steps of this algorithm are given below.

Begin

Step 1: Connect the nodes in each level to the nodes of the adjacent levels by the proper rotation links. /* construct the 3D network */

Step 2: Search for the shortest path in the 3D network.

Step 3: Project the shortest path onto x-y plane.

End

5.1 Proper Rotation Links

If the reference point of the robot is placed at point P_1 at orientation θ, the position of the robot is denoted as $P_1(\theta)$. The free space for a given θ is denoted by $FP(\theta)$. We separate the motions of translation and rotation. Thus, if the robot rotates from θ_1 to θ_2, its reference is fixed at the same point. This is denoted by $P_1(\theta_1) \rightarrow R \rightarrow P_1(\theta_2)$, where P_1 is the locat-

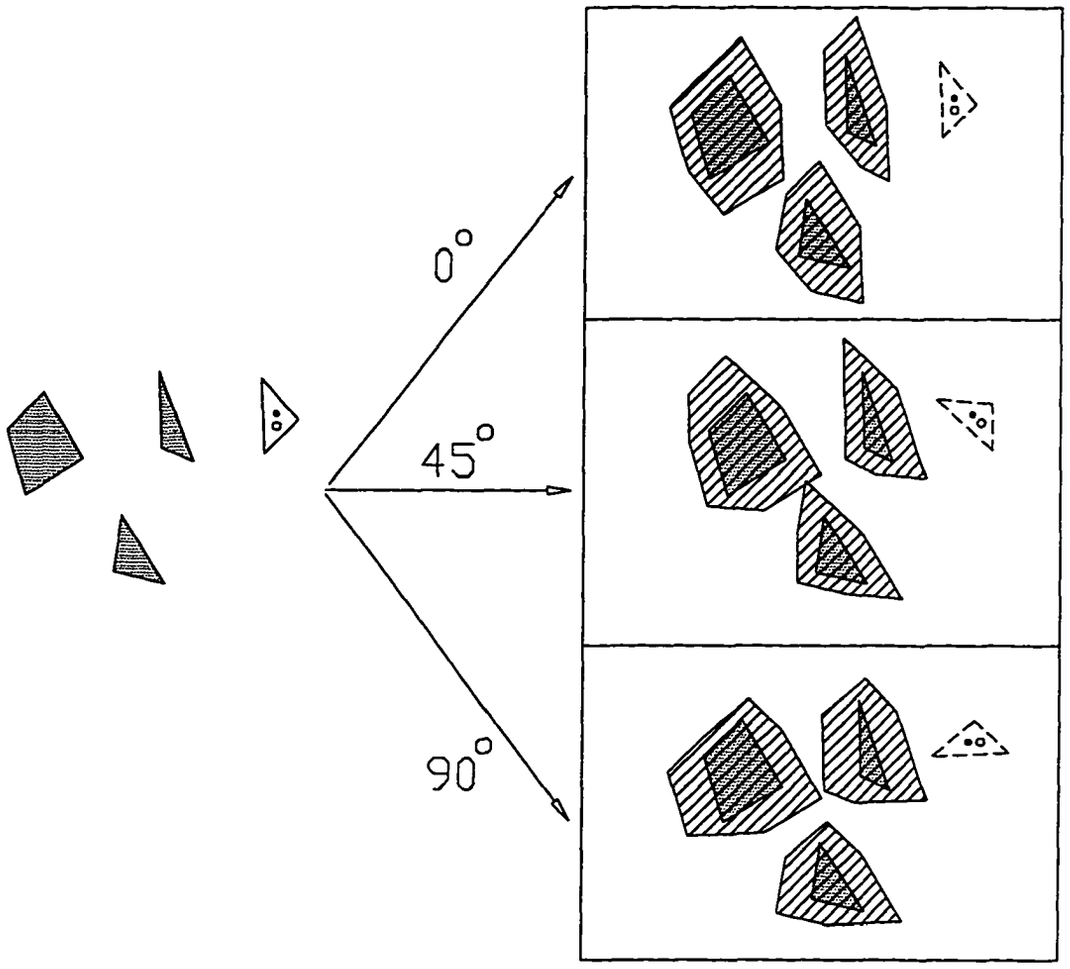


Figure 5.1. Multiple configuration spaces

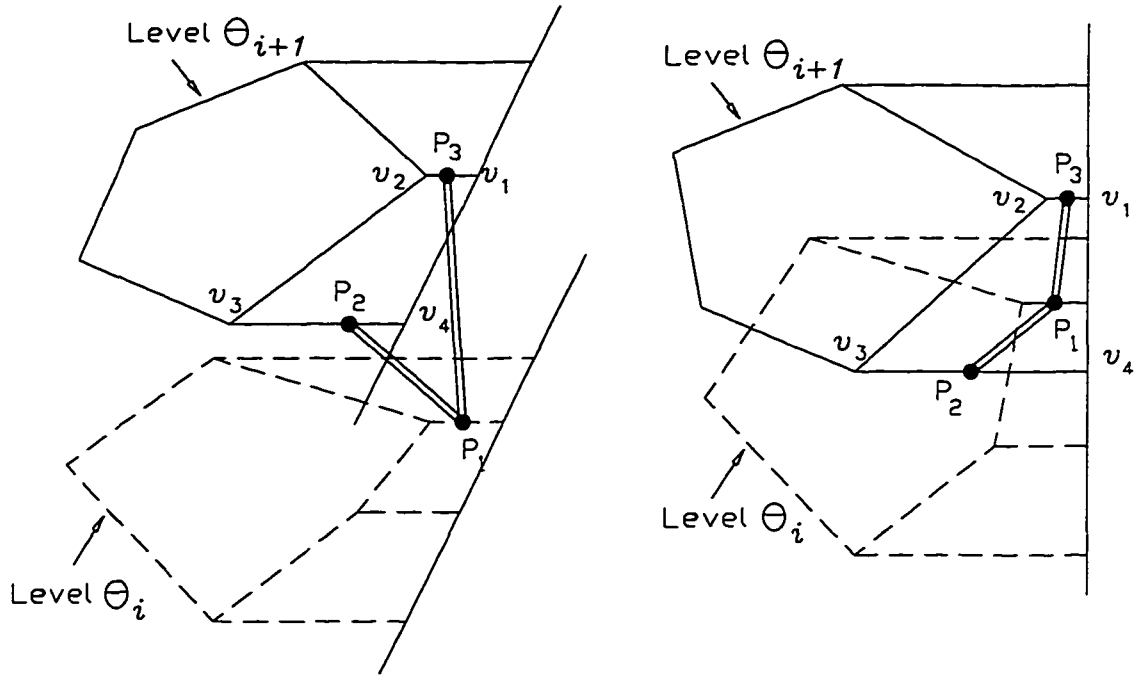
ion of the reference point. The purely translational motion from P_1 to P_2 at level θ_2 is denoted as $P_1(\theta_2) \rightarrow T \rightarrow P_2(\theta_2)$. If the robot rotates from θ_1 to θ_2 at position P_1 then translates to P_2, the motion is denoted as $P_1(\theta_1) \rightarrow R \rightarrow P_1(\theta_2) \rightarrow T \rightarrow P_2(\theta_2)$. If the motion is reversible, the symbol " $\leftrightarrow$ " is used.

Given adjacent levels θ_1 and θ_2, if the orientation interval is small enough, and supposing P is in both $FP(\theta_1)$ and $FP(\theta_2)$ from the top view, then the robot will have a collision

free motion $P(\theta_1) \leftrightarrow R \leftrightarrow P(\theta_2)$. However, if P is in $FP(\theta_1)$ but not in $FP(\theta_2)$, that means after the robot rotates from θ_1 to θ_2, position P will be inside some C-space obstacle. Thus, there will be collision if the robot has a motion $P(\theta_1) \rightarrow R \rightarrow P(\theta_2)$.

Figure 5.2 shows two adjacent rotation levels θ_i and θ_{i+1}. If the two levels are projected onto the x-y plane, P_1 is determined to be in the collision-free cell $v_1v_2v_3v_4$ of level θ_{i+1} (see Figure 5.2 (b)). This means that if the robot is placed at P_1, it will have a collision-free rotation from θ_i to θ_{i+1}, and the robot can move from $P_1(\theta_{i+1})$ to $P_2(\theta_{i+1})$ or $P_3(\theta_{i+1})$ without any collision after the rotation.

The motion that rotates the robot from θ_i to θ_{i+1} about P_1, and then translates it to another place P_2 is referred to as an *RT motion*, and the link connecting $P_1(\theta_i)$ and $P_2(\theta_{i+1})$ is referred to as an *RT link*. If the motion translates the robot first and then rotates it, the motion is referred to as a *TR motion* and the link a *TR link*. Notice that *RT* links and *TR* links are directed. If $P_1(\theta_i)$ to $P_2(\theta_{i+1})$ is an *RT* motion, there is no guarantee that there is a collision-free *RT* motion from $P_2(\theta_{i+1})$ to $P_1(\theta_i)$; however, the reverse step, the *TR* motion from $P_2(\theta_{i+1})$ to $P_1(\theta_i)$, is safe. For example, in Figure 5.2, since P_1 has been determined to be in one collision-free cell of level θ_{i+1}, it will be joined via *RT* links to the nodes of cell $v_1v_2v_3v_4$, $P_2(\theta_{i+1})$ and $P_3(\theta_{i+1})$. However, $P_2(\theta_{i+1})$ and $P_3(\theta_{i+1})$ will link to $P_1(\theta_i)$ with *TR* links. If we link $P_2(\theta_{i+1})$ to $P_1(\theta_i)$ by an *RT* link, it will have collision because P_2 is not in the free space of level θ_i. However, the motions $P_1(\theta_i) \rightarrow R \rightarrow P_1(\theta_{i+1}) \rightarrow T \rightarrow P_2(\theta_{i+1})$, and $P_2(\theta_{i+1}) \rightarrow T \rightarrow P_1(\theta_{i+1}) \rightarrow R \rightarrow P_1(\theta_i)$ are both collision-free.



(a) two adjacent levels

(b) projection of the two levels onto the x-y plane

Figure 5.2. Proper rotation links

5.2 3D Network Construction

Now, we need to find if there is any cell at the adjacent layers which contain the target node. If there exists such cell, the target node will be linked to the nodes on the cell.

5.2.1 Cell finding

For any primary vertex, except for the "up_concave" vertex, there must be one or two cells just below the horizontal line, which passes through the primary vertex.

For example, in Figure 5.3, all the primary vertices are numbered by the Arabic numbers,

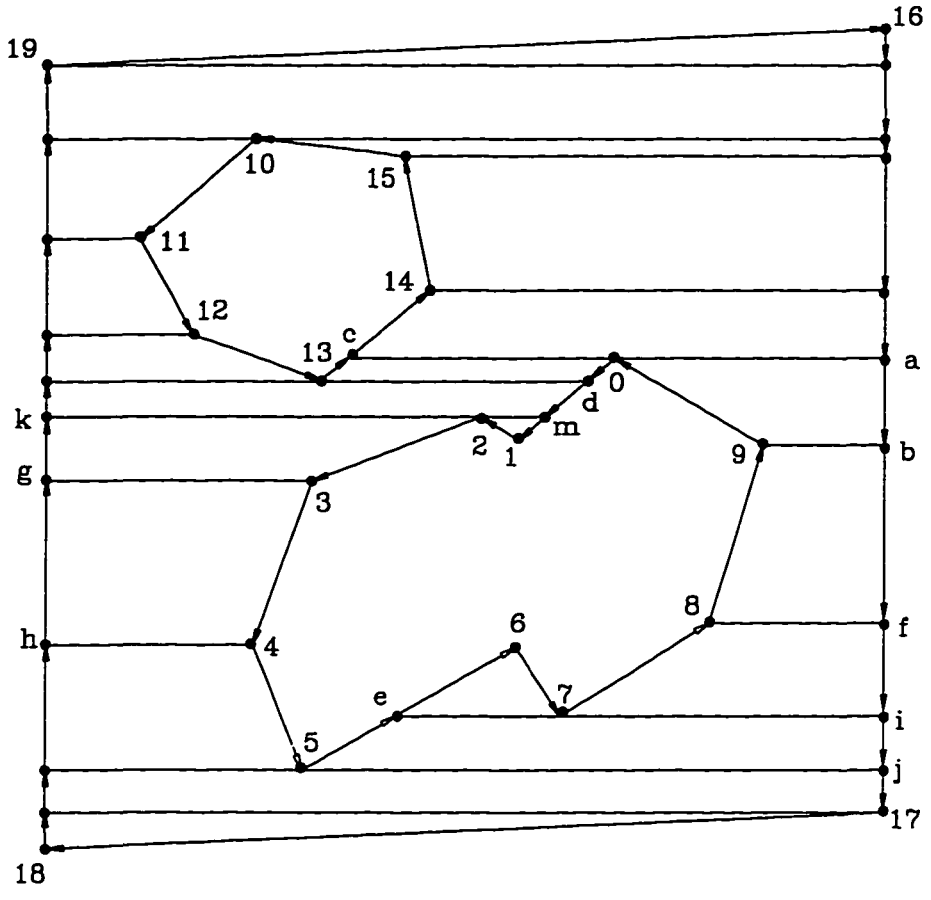


Figure 5.3. Cell finding

and some of the secondary vertices are numbered by letters. If the vertex is a "down_concave" vertex, e.g., vertex *6*, there is only one cell below it. It is a triangular cell *6e7*. Vertex *e* is the back vertex of vertex *6*, and vertex *7* is the front vertex of vertex *6*. Vertex *e* is the left vertex of vertex *7*, and vertex *7* is the right vertex of vertex *e*. Thus, the location of cell *6e7* can be defined.

If the primary vertex is a "down_convex" vertex, e.g., vertex *7* in Figure 5.3, there will be a quadrilateral cell *ie5j* just below it. Vertex *i* is the right vertex of vertex *7*, and vertex *e* is the left vertex of vertex *7*. Vertex *j* is the front vertex of vertex *i*, and vertex *5* is the back vertex of

vertex *e*.

If the primary vertex is a "up_convex" vertex, e.g., vertex *0* and vertex *2* in Figure 5.3, there will be two cells below it. For vertex *0*, there are two quadrilateral cells, *a09b* and *0c13d*. For vertex *2* there is one quadrilateral cell and one triangular cell, *2kg3* and *m21*. Those cells can also be traced by the vertex information stored in *vinfo*.

Similarly, the cell below a right or left primary vertex can be found by the same data structure.

5.2.2 3D network algorithm

If the *motion* field in one adjacent node of a center node is 0, that means it is a *RT* link from the center node to this adjacent node. If the *motion* field is 1, it is a *TR* link. The procedure for constructing the 3D network is as follows.

Procedure Construct3DNetwork

Begin

1. **for** every rotation level *i* **do**
2. **for** every node *p* on the passage network of the given level *i* **do**
3. **if** there is any cell *g* in level *i* + 1 which contains *n* **then**
4. **for** every node *m* on cell g **do**
5. /* Link node *p* with node *m* by an *RT* link (node *p* is the center node) */
6. create a pointer *ptr* with *Netlist* data type;
7. *ptr->nodeid* = *m*; *ptr->level* = *i*+1; *ptr->motion* = *0;*
8. *ptr->weight* = the *x-y* distance between *p* and *m*;
9. insert *ptr* into the linked list pointed by *netnode*[*i*][*p*]->*next*;
10. /* Link node *m* with node *p* by a *TR* link (node *m* is the center node) */
11. create a pointer *ptr* with *Netlist* data type;
12. *ptr->nodeid* = *p*; *ptr->level* = *i*; *ptr->motion* = *1*;

13. *ptr->weight* = the *x-y* distance between *p* and *m*;

14. insert *ptr* into the linked list pointed by *netnode*[*i*+1][*m*]->*next*;

15. (repeat steps 3 – step 14 for level *i* - 1 instead of level *i* + 1);

End;

Since there are O(n) nodes at each orientation level, step 2 will be repeated $O(n)$ times. Step 3 is a point location problem, and it can be done by two binary searches. The first search finds the vertical location in $O(\log n)$ time, since there are $O(n)$ slabs in the *FP*. The second search finds the horizontal location in $O(\log N)$ time, since there are $O(N)$ cells in one slab. Thus, step 3 will take $O(\log n)$ time for each target node. Since there are at most four gates for each cell, steps 5 and 14 take constant time. The execution time for step 15 is the same as that of steps 3 through 14. If the rotation interval is δ, there will be π/δ levels; let $c = \pi/\delta$. Thus, the total execution time for procedure *Construct3DNetwork* is $O(cn\log n)$.

5.3 Motion Planning Algorithm

We use Dijkstra's algorithm [16] technique to search the 3D network to find the shortest path. The whole motion planning algorithm is described as follows.

Algorithm Motion-Planning

Begin

1. **for** every rotation level *i* **do**

2. Find the C-space obstacles;

3. Set_E_V_Info();

4. Sort the vertices in C-space obstacles by non-increasing y coordinate and put them in *ylist*;

5. Hslice();

6. NetworkperLevel();

7. Construct3DNetwork();

8. Search for the shortest path by Dijkstra's algorithm;

9. Project the path onto the x-y plane;

End;

If the moving object has $O(1)$ vertices, step 2 of Algorithm *Motion-planning* will take $O(n)$ time to build the C-space obstacles for each level. It takes constant time to obtain the vertex and edge information for each vertex and edge, so steps 3 takes $O(n)$ time per level. Step 4 takes $O(n\log n)$ time to sort n vertices. Step 5 takes $O((n+k)N)$ time, and step 6 takes $O(n)$ time. Thus, steps 1 through 6 take a total of $O(c((n+k)N+n\log n))$ time. Step 7 takes $O(cn\log n)$ time. The number of links originating at each node in a 3D network is at most fourteen: four connect to the previous level, four connect to the next level, and another six links connect to the nodes at the same level. Thus, there are $O(cn)$ links in a 3D network, so it takes $O(cn)$ time to search. Step 9 takes $O(cn)$ time to project the path onto the x-y plane. Thus, the total running time for Algorithm *Motion-Planning* is $O(c((n+k)N+n\log n))$.

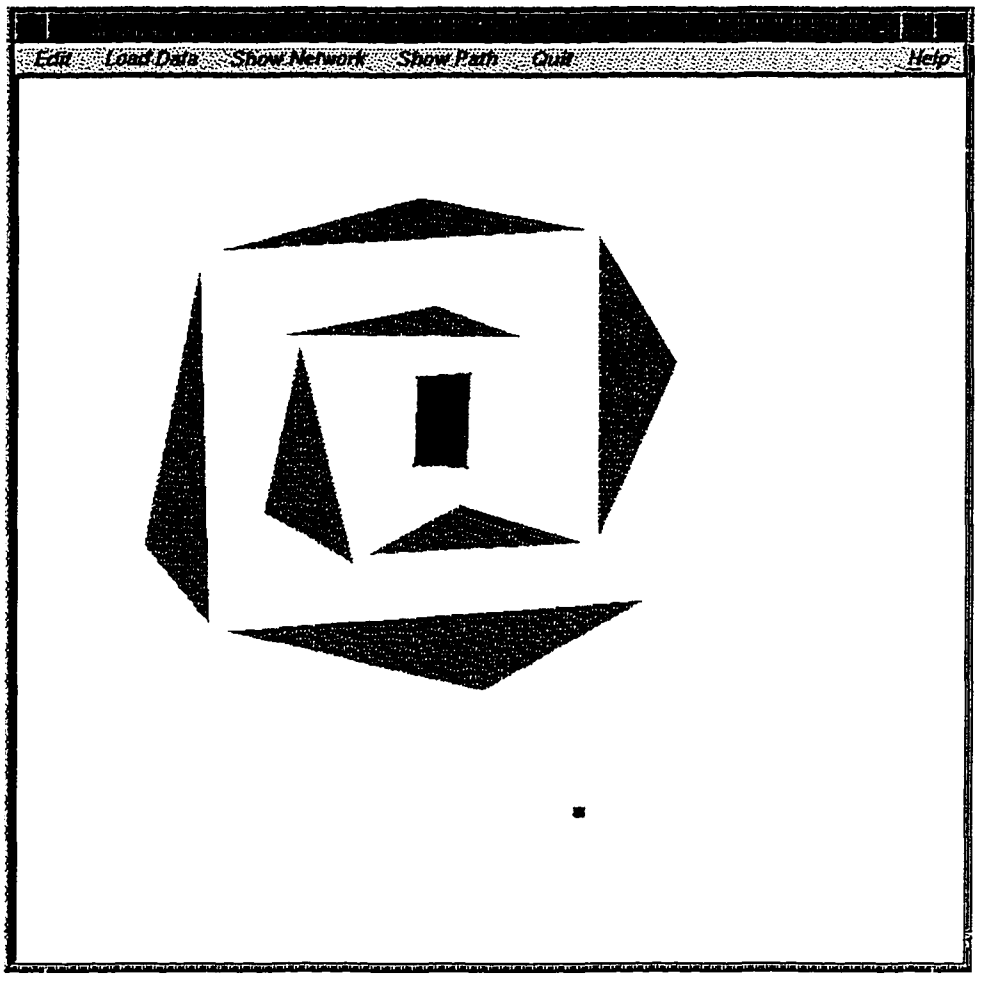
6. RESULTS AND CONCLUSIONS

This chapter will show the implementation results and the comparisons with other path planners. Some conclusions and discussions are also given.

6.1 Implementation and Comparisons

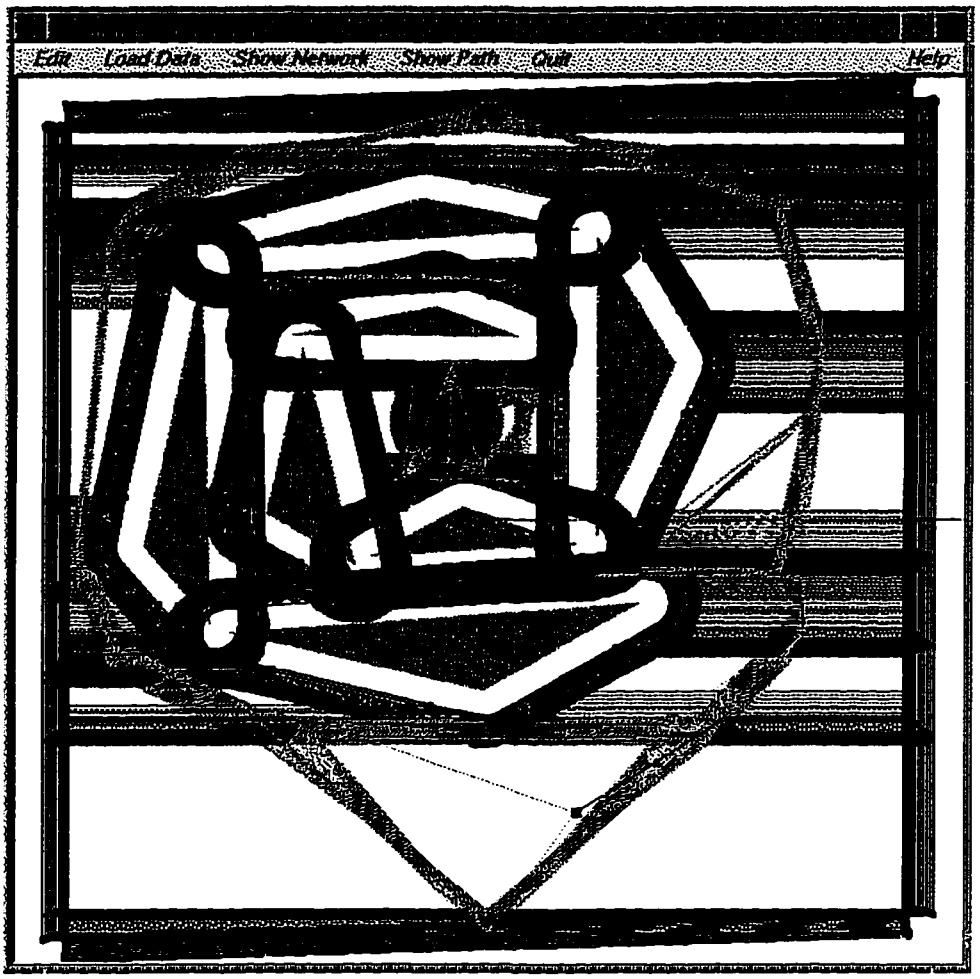
The algorithm has been implemented in C++ on a Silicon Graphics workstation using Open Inventor for graphics display and Xt/Motif for the graphical user interface. Figure 6.1 (a) shows an obstacle environment with a rectangular robot at the center. Figure 6.1 (b) is the top view of the 3D network. Figure 6.1 (c) is a close-up side view of the 3D network. Figure 6.1 (d) shows the final collision-free path projected onto the x-y plane. In Figure 6.2, six environments taken from the literature are shown. Table 6.1 compares this work with the planners developed by Zhu and Latombe (ZL) [102], Barbehenn and Hutchinson (BH) [5], and Vleugels et al. (VKO) [94]. Execution times are affected by different implementations, machines, and experimental conditions.

ZL and BH use the hierarchical approximate cell decomposition approach. The main step of that approach is recursively decomposing the 3D MIXED cells in (x, y, θ). Unlike their approach, the cells generated by our path planner are all 2D EMPTY cells. Thus, we do not need the complicated procedure to recursively decompose the cells. Vleugels et al. [94] use a neural network and deterministic technique to solve the problem. They obtain the times by averaging over 100 runs of their program. Although their results seem good, the variation between the best result and the worst result is very large. Also, choosing the "adequate learning parameters" for each environment is important to obtain the best results.



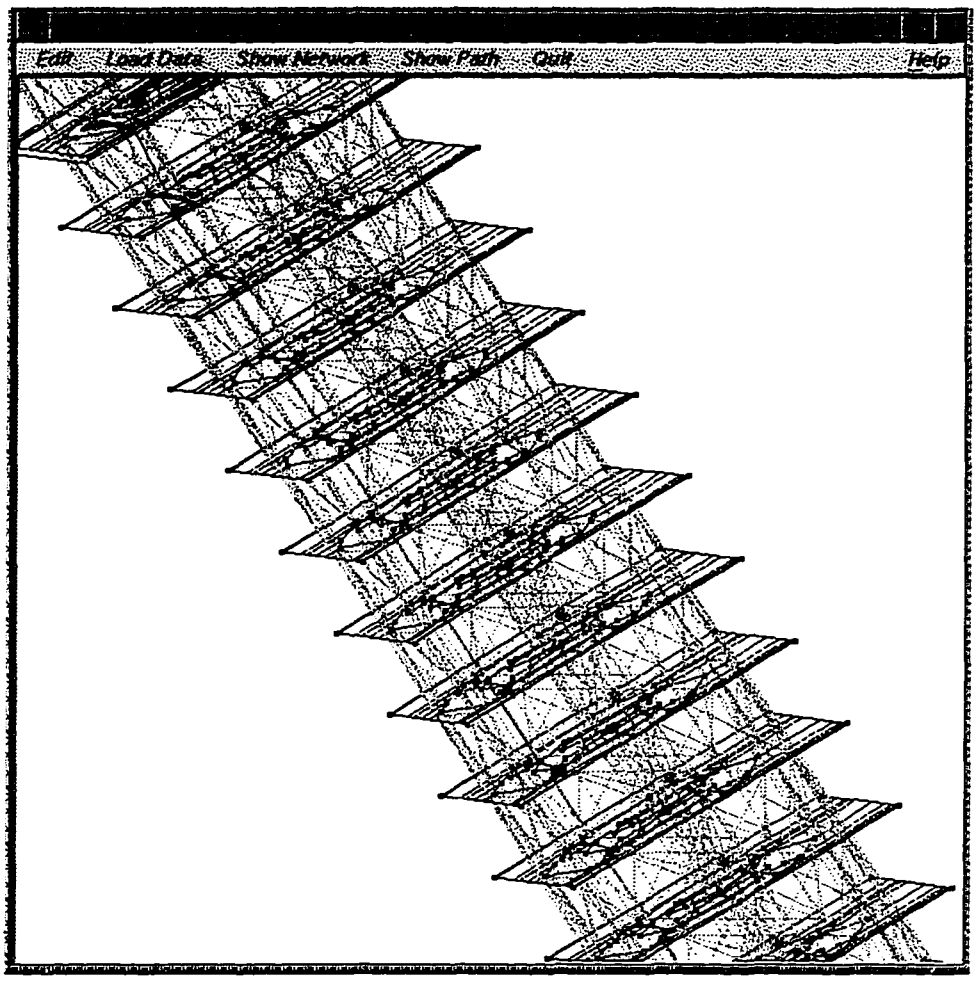
(a) input environment

Figure 6.1. One implementation example



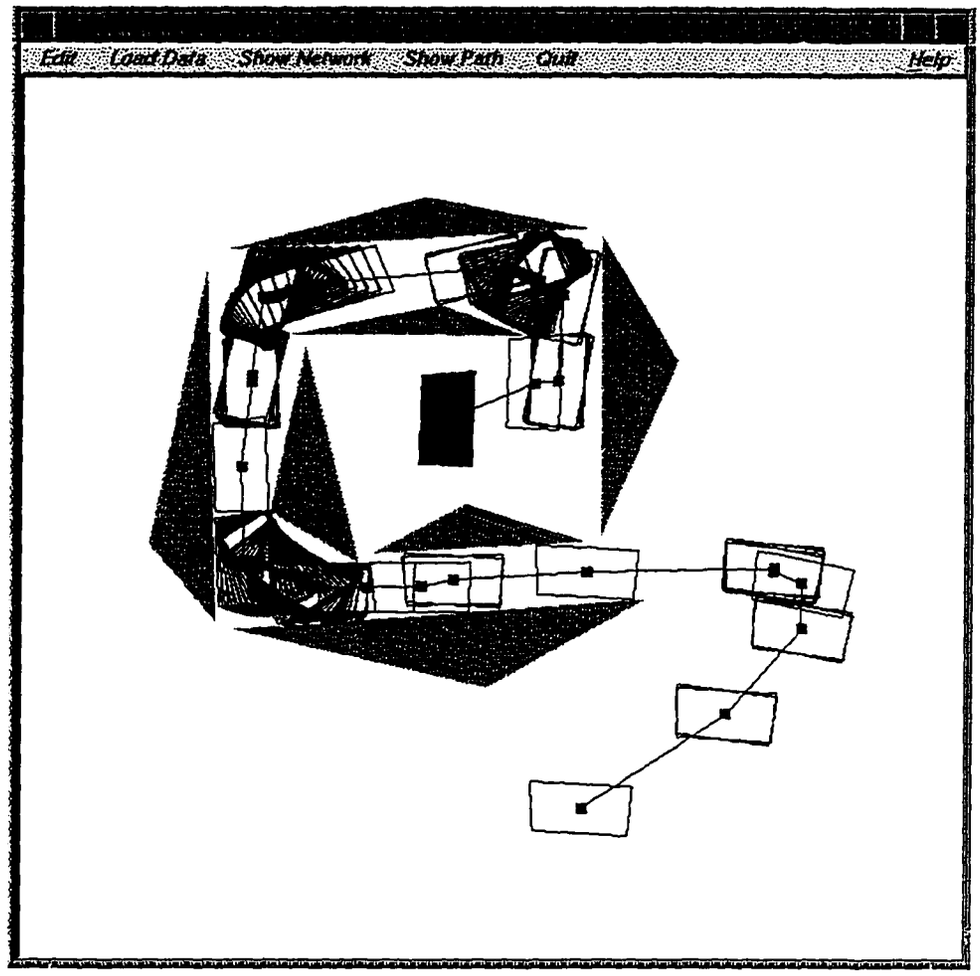
(b) top view of the 3D network

Figure 6.1. (continued)



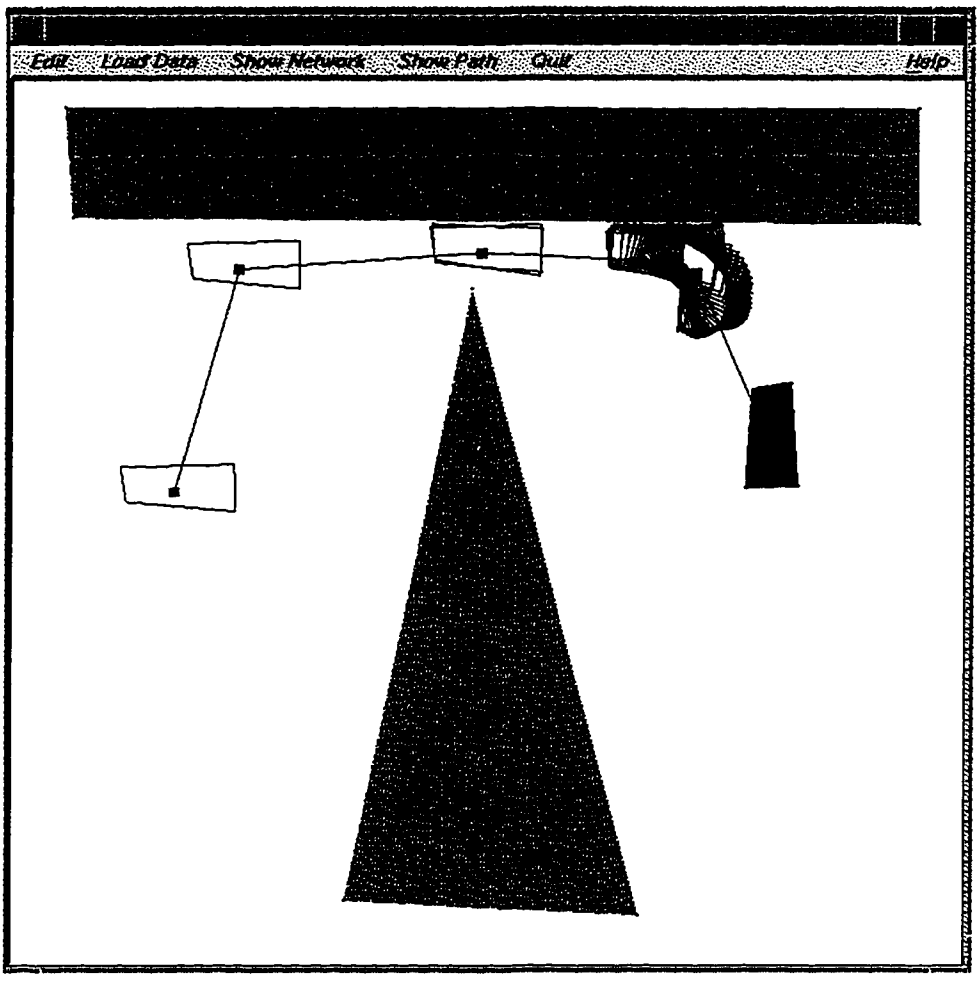
(c) side view of the 3D network

Figure 6.1. One implementation example (continued)



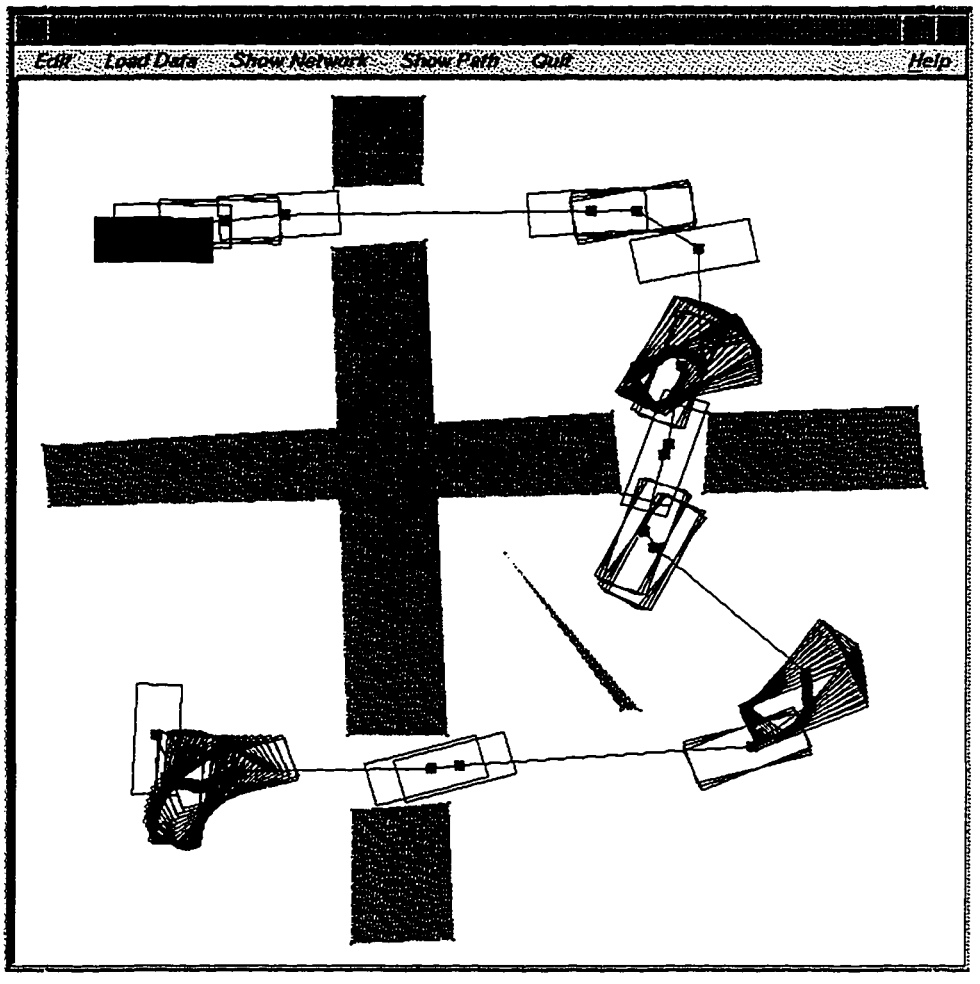
(d) final route

Figure 6.1. (continued)



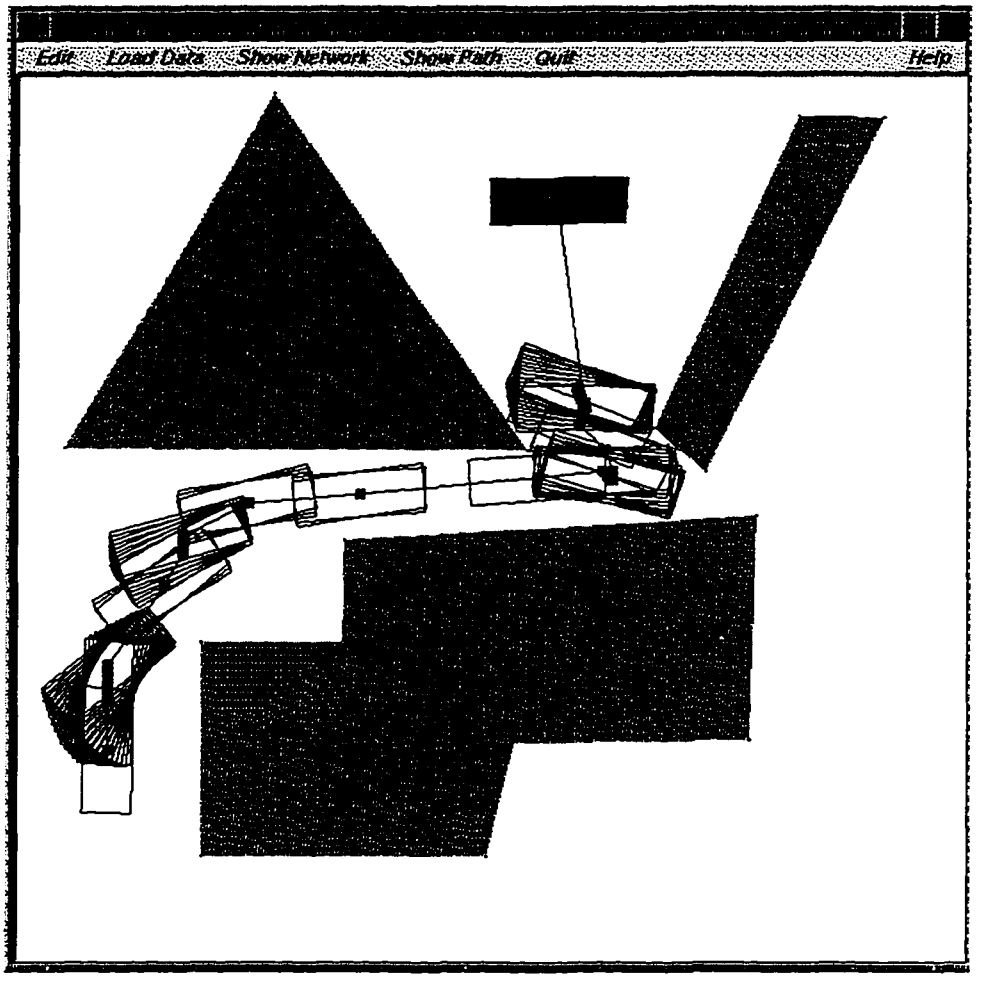
(a) example 1

Figure 6.2. Six environments



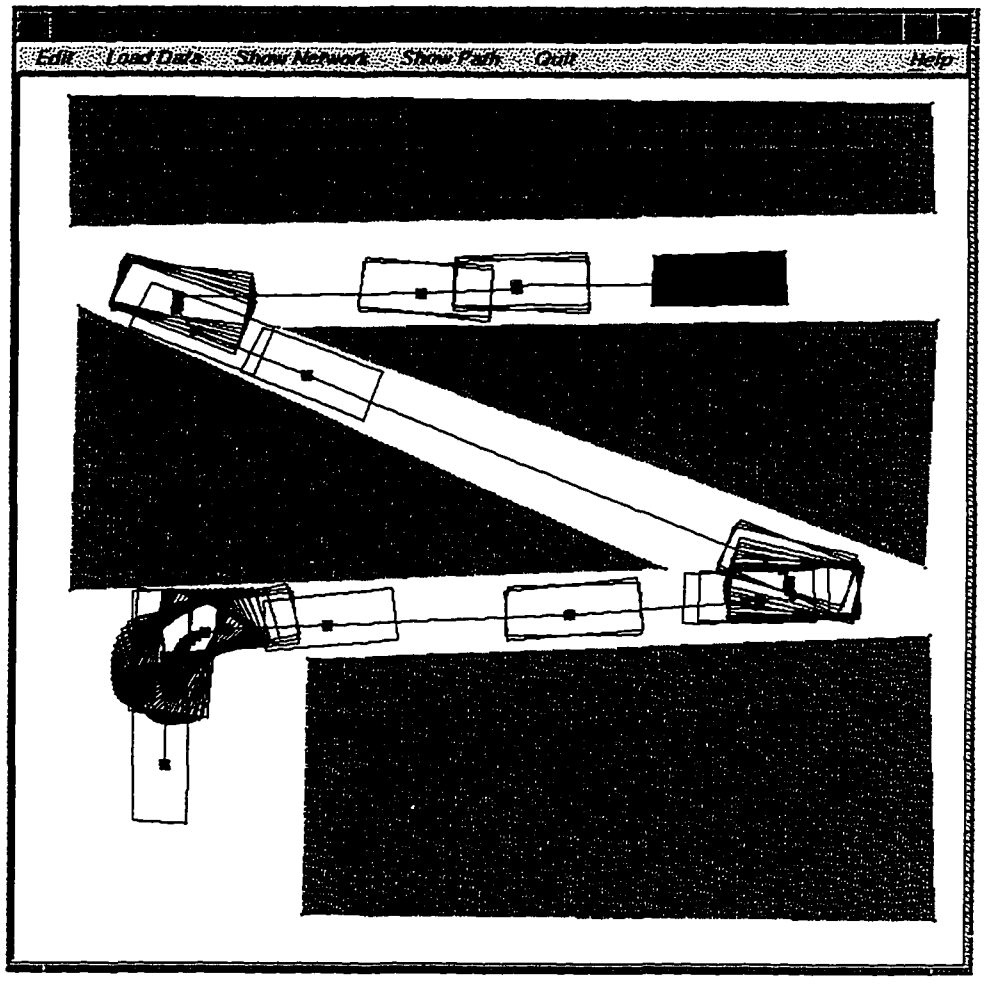
(b) example 2

Figure 6.2. (continued)



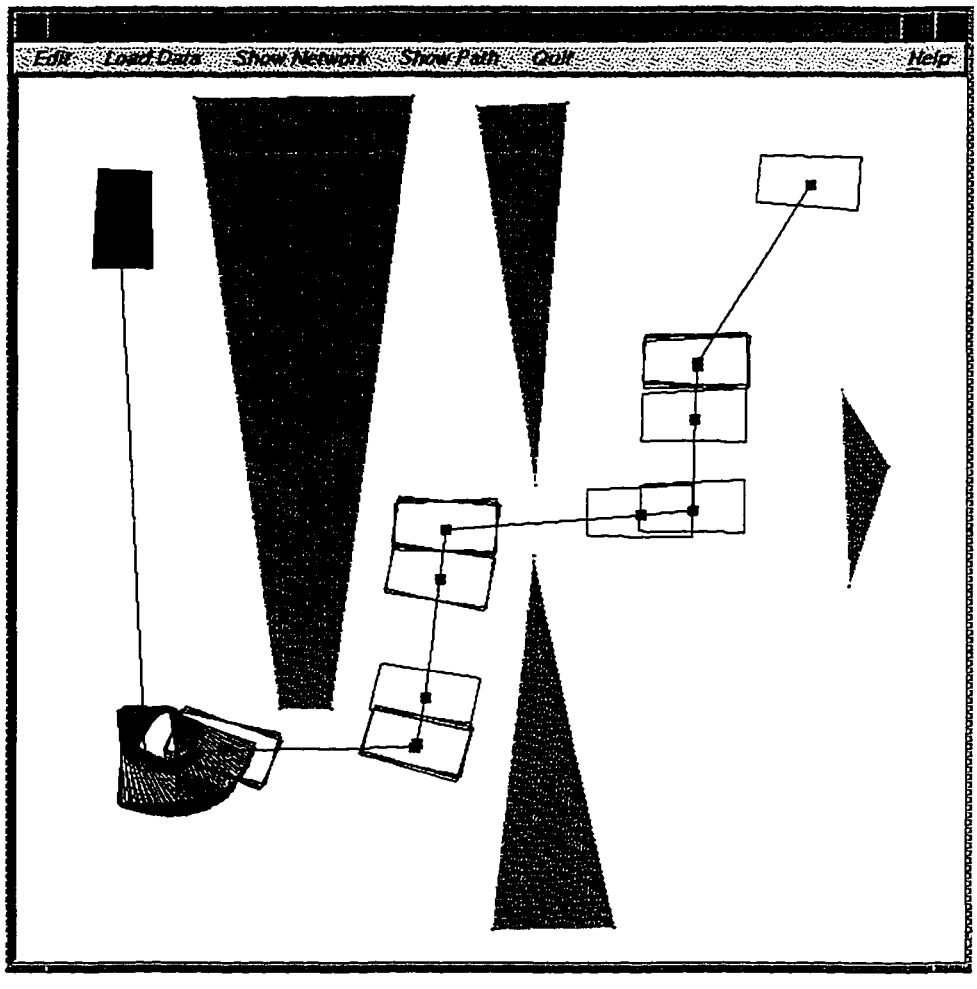
(c) example 3

Figure 6.2. (continued)



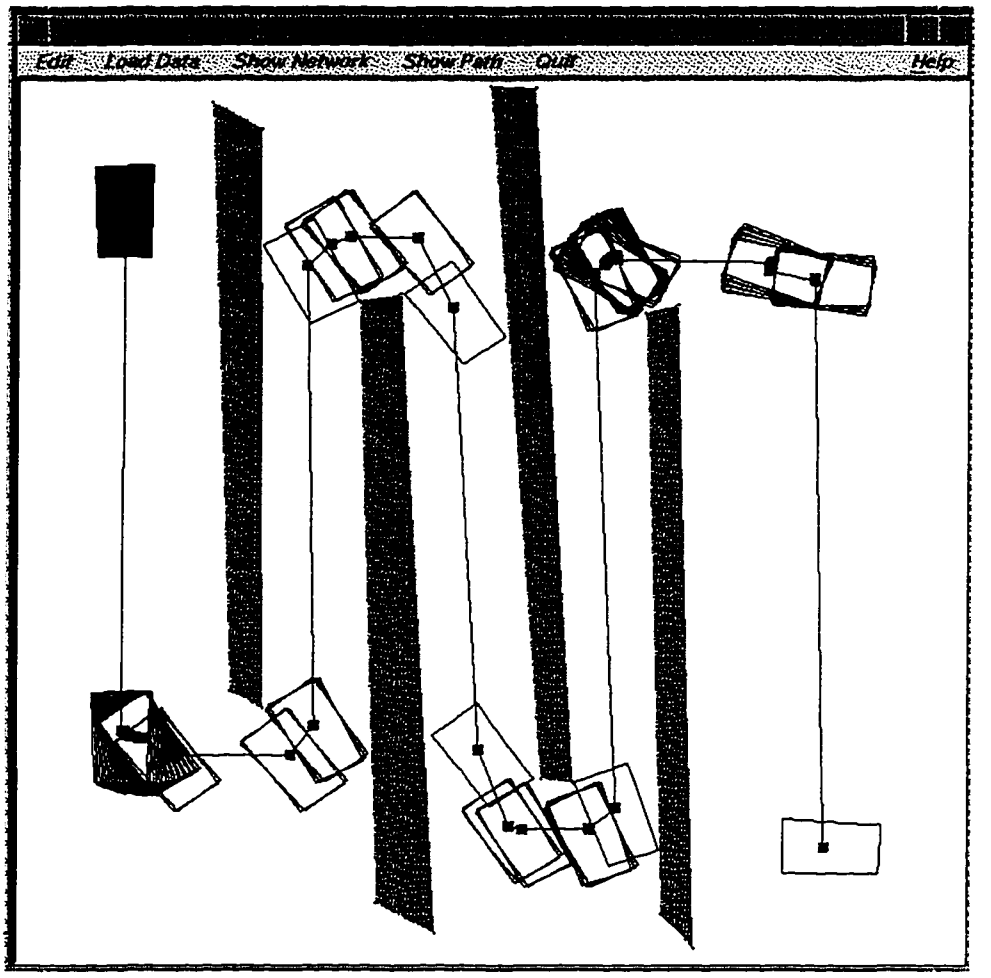
(d) example 4

Figure 6.2. (continued)



(e) example 5

Figure 6.2. (continued)



(f) example 6

Figure 6.2. (continued)

Table 6.1 Comparisons of the six examples

		ZL [102]	BH [5]	VKO [94]	Chen
Machines		Apple Macintosh II	SUN IPC	SGI Indigo R3000 33MHz	SGI Indigo R3000 33MHz
Languages		Allegro Common Lisp	Lucid Common Lisp (v1.3)	C/C++	C++, Xt/ Motif Open Inventor
examples	CPU time(*min*)				
	1	0.6	1.2	N/A	0.012
	2	2.5	8.1	N/A	0.059
	3	5.5	6.5	N/A	0.034
	4	5.0	4.5	N/A	0.024
	5	N/A	N/A	0.083	0.028
	6	N/A	N/A	0.078	0.035

6.2 Conclusions and Discussions

Algorithms for collision-free path planning have become quite valuable in a variety of applications such as robotics, virtual prototyping, assembly planning, and computer graphics. Applied computational geometry also plays an important role in many fields such as medicine, drug design, manufacturing design, feature design, IC board routing design, geography problems, etc.

In this study, an $O(c((n+k)N+n\log n))$ time algorithm is presented for planning a heuristic shortest path. A slabbing technique is used to find the contour of a set of intersected C-space obstacles and a passage network for each rotation level. Successive orientation levels

are connected by the proper rotation links to construct a directed 3D network. Then, Dijkstra's algorithm is used to find the shortest path in the 3D network. Finally, the path is projected onto x-y plane. This algorithm is straight-forward and easy to implement.

This algorithm has several contributions. First, it incorporates robot rotation and translation. Secondly, it allows intersected C-space obstacles and calculates the contour of the intersected C-space obstacles efficiently. Then, it combines slabbing technique and network searching. Experiments show that this approach is significantly faster and simpler than other approaches.

One question that arises here is whether this path planner can always find a collision-free path if such a path exists. We would like to determine it is true or false.

The robot in this study is called a "free flying" robot. That means the robot can rotate and translate freely in the plane among a set of obstacles. Another kind of robot is called a "car-like" robot. The motions of the car-like robot have certain *non-holonomic constraints*, i.e., non-integrable kinematic constraints [6], [22], [23], [24], [33], [46], [47], [48], [49], [59], [67], [72], [81], [82], [83], [88], [91], [98] and are therefore more difficult. Modification of the path planner developed here to adapt to the constraints of car-like robots is another interesting research topic.

APPENDIX. EXACT DESCRIPTION OF THE B-VORONOI DIAGRAM OF A HOMOTHETIC ROBOT MOVING THROUGH TWO OBSTACLES

This Appendix presents a simple *O(n)* time algorithm to move a homothetic robot, i.e., a scaled and translated copy of a 2D robot, through two polygonal obstacles along its *B*-Voronoi diagram, where *n* is the total number of edges. The *B*-Voronoi diagram represents a locus of robot path points from which it can expand or contract to touch the two obstacles simultaneously. The algorithm actually computes the feasible locus, that is, a description of the set of all turning points of the polygonal path.

A.1 Introduction

Motion planning is a fundamental problem in robotics. In general, the goal is to find a collision free path for a robot amidst obstacles. While there have been several important theoretical algorithmic results in the field, many of the procedures developed so far are difficult to implement. Here, a linear time algorithm is presented to find the exact description of the high clearance locus for a homothetic robot moving through two obstacles. This algorithm plays an important role in an $O(n \log N)$ time algorithm by Leven and Sharir [52] for planning a purely translational motion of a convex object among a set of polygonal obstacles in two-dimensional space, where n is the number of obstacle corners and N is the number of obstacles.

Motion planning problems can often be reduced to finding a *high-clearance* path in a Voronoi diagram. Voronoi diagrams partition the plane into several regions called *Voronoi cells*. Each cell is associated with a unique closest point or object of a given set of obstacles, so the Voronoi diagram is the locus which is equidistant to at least two obstacles.

If the obstacles are points in a plane, the standard Voronoi diagram of those points parti-

tions the plane into several convex polygonal regions (see Fig. A.1 (a)). When the moving object is a disc, the diagram is the locus of the centers of all maximal circumscribed circles, and the partitions of the plane will be smooth curves (see Fig. A.1 (b)). If the moving object and the obstacles are polygons, and we use the convex distance function mentioned in [52] to define the distance, then the partitions of the plane will be polygonal arcs (see Fig. A.1 (c)), and the Voronoi diagram of those polygons is called the *B-Voronoi diagram*.

Since the objects in this Appendix are two polygonal obstacles and a polygonal robot, the Voronoi diagram we consider is a polygonal arc, composed of several segments. The intersections between segments are called *turning points*. This algorithm first finds the turning points between segments. The *B*-Voronoi diagram is then obtained by connecting these turning points. In this Appendix, three questions are addressed. First, how to find the first turning point; e.g., point t_1 in Figure A.1 (c). Second, how to find the points between the first one and the last one. Finally, how to find the last turning point; e.g., t_k in Figure A.1 (c).

Section 2 reviews the *B*-Voronoi diagram. Section 3 gives the procedures to find the turning points, and section 4 presents some conclusions.

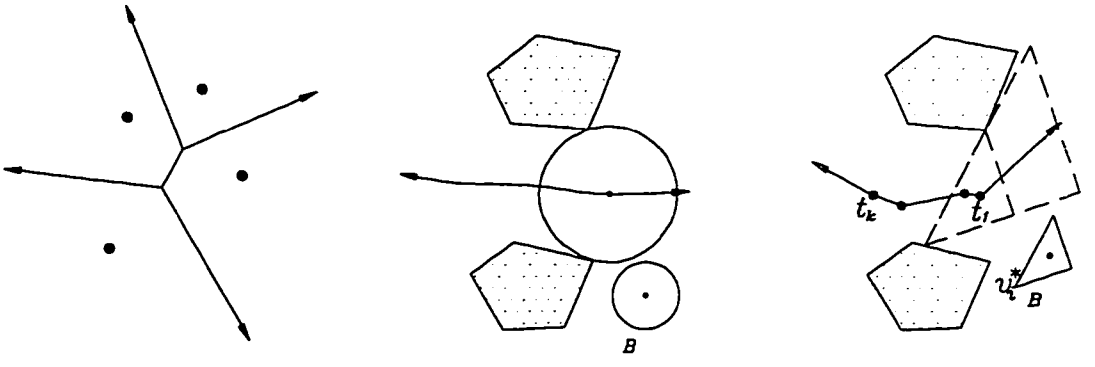
A.2 Definitions

A.2.1 *B*-Voronoi diagram

Let B be a convex robot, and let O be a reference point inside B. If O lies at the origin, the position of B is called *standard position*, and it is denoted by B_o. If B is scaled by a factor λ when it is at the standard position, it is denoted by λB_o.

In [52], the *B-distance* from a point p to a point q is defined by

$$d_B(p, q) = inf\{\lambda: q \in p + \lambda B_o\}.$$



(a) obstacles are points (b) moving object is a disc (c) moving object is a polygon

Figure A.1. Voronoi Diagrams

Informally, $d_B(p, q)$ is the smallest scaling factor λ such that when the reference point O of B is on point p, λB_o just touches q. Similarly the B-distance from a point p to an obstacle S_i is defined as

$$d_B(p, S_i) = inf\{\lambda: p + \lambda B_O \cap S_i \neq \varnothing\},$$

so there exists a point $y \in S_i$ such that when $\lambda = d_B(p, S_i)$ and the reference point O is on p, λB_o touches S_i. Point y is called the *B-closest* point on S_i to B. The set of all points whose B-distance to S_i is less than or equal to the B-distance to S_j for $i \neq j$ is defined as

$$H(i, j) = \{p \in E^2 : d_B(p, S_i) \leq d_B(p, S_j)\}.$$

The *B-Voronoi cell* with respect to S_i is defined as

$$C_B(S_i) = \bigcap_{i \neq j} H(i, j).$$

Thus, the *B*-Voronoi diagram is defined to be the set of points which belong to more than one *B*-Voronoi cell.

In order to avoid degenerate configurations, we assume the obstacles and *B* are in *general position* [52], i.e., we assume that no boundary edge of *B* is parallel to the boundary edge of any obstacle or to a line joining a pair of boundary corners of these obstacles. This assumption prevents *B*-Voronoi segments from degenerating into general two-dimensional regions. An example is of a degeneracy shown in Figure A.2. Edge *e* of polygon *B* is parallel to *ab*. The dotted triangles are some λB_o that touch the two obstacles simultaneously and the shaded area is a degenerate two-dimensional area of the *B*-Voronoi diagram.

As long as the scaling factor λ is greater than one, we can move the robot along the *B*-Voronoi diagram without collision, and it can be used to plan high-clearance motion for any object that is *similar* to *B*. In other words, if there are two similar robots with different sizes, only one *B*-Voronoi diagram needs to be computed to do the motion planning for both robots.

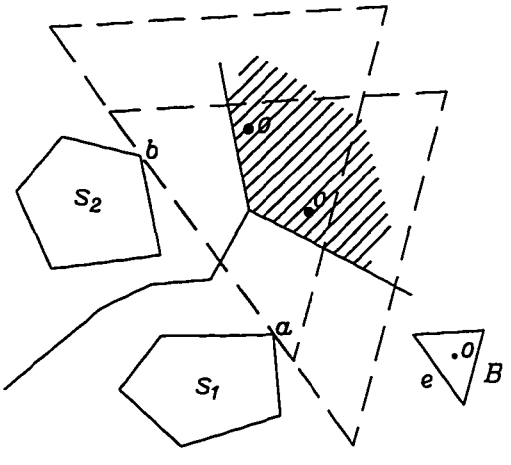


Figure A.2. 2D B-Voronoi Diagram

A.2.2 Data structures

Edges in the objects are represented by vectors. The edge vectors for obstacles S_1 and S_2 are listed in *counterclockwise* order, while the edges for the robot B are ordered in *clockwise* order. Suppose the two outer supporting lines of S_1 and S_2 are sp_1 and sp_2. The two supporting lines and the two obstacles form a closed region. The list of edge vectors which belong to S_1 (resp. S_2) in the closed region is called C_1 (resp. C_2) (see Fig. A.3 (a)).

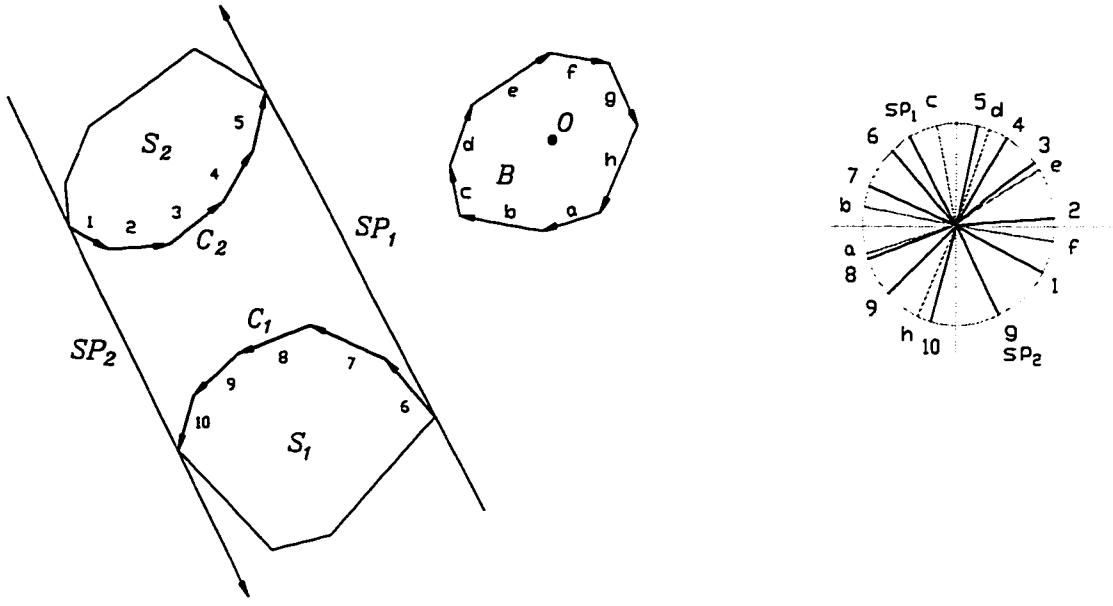
The direction of sp_1 (resp. sp_2) is the same as that of the ray shooting from the start point of C_1 (resp. C_2) to the end point of C_2 (resp. C_1). The two outer supporting lines can be found in time proportional to the total number of vertices of S_1 and S_2 [68]. We will suppose sp_1 is closer to B than sp_2.

A.2.3 Preliminary observations

Observation A.2.1: When λB_o touches S_1 and S_2 simultaneously, the two contact points on S_1 and S_2 lie on C_1 and C_2 respectively.

Leven and Sharir [52] have shown that the turning points correspond to configurations where one vertex of B touches one vertex of S_1 or S_2. In Figure A.4 we can see that the scaling factor for moving B from the position where v_2 touches vertex a until it touches vertex b is a continuous decreasing linear function. Similarly, if B is continuously enlarged and moved from the configuration where v_2 touches vertex b until it touches vertex c, the scaling factor is another linear function.

The turning point occurs at the intersection of the two lines, i.e. when vertex v_2 of λB_o touches vertex b of the obstacle. We take this as an observation.



(a) input Vectors (b) sorted vectors

Figure A.3. Vectors in the objects

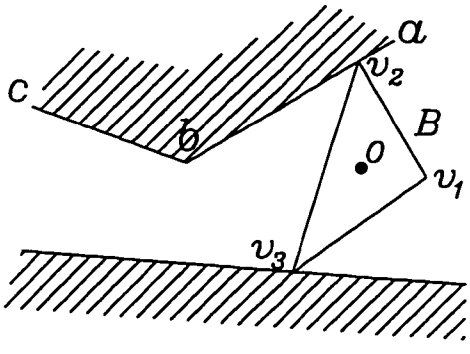


Figure A.4. Turning Point

Observation A.2.2: When one vertex of λB_o touches a vertex of S_1 or S_2, there is a turning point of the B-Voronoi diagram.

Two objects are said to have a *VV contact* if one vertex of an object touches one vertex of another object. An *EV contact* is one where an edge of an object touches a vertex of another object. Similarly, a *VE contact* is one where a vertex of an object touches an edge of another object.

Lemma A.2.1: If λB_o is on the right-hand side of sp_1 and moved along the B-Voronoi diagram from infinity towards the first turning point t_1, the same two consecutive edges of B, say e^*_i and e^*_{i+1}, simultaneously and constantly touch S_1 and S_2 (see Fig. A.1 (c)).

Proof: We know that the turning point t_1 corresponds to a VV contact between one vertex of B and one vertex of S_1 or S_2. Suppose edges e_i and e_j of λB_o touch S_1 and S_2 and are not consecutive when λB_o moves along the B-Voronoi diagram from some point p (a point lying between infinity and t_1) to t_1. Therefore, there exists one edge e_k of λB_o, lying between e_i and e_j, that is between S_1 and S_2, so that if λB_o is moved from infinity to p, a certain scaling of e_k will touch S_1 or S_2 before e_i or e_j touches S_1 or S_2. Edge e_k makes a VV contact before λB_o arrives t_1, so t_1 is not the first turning point, which contradicts the assumption. □

In Lemma A.2.1, the vertex v^*_i between e^*_i and e^*_{i+1} is referred to as a *blocking vertex.*

Lemma A.2.2: If λB_o is on the right-hand side of sp_2 and moved along the B-Voronoi diagram from infinity towards the last turning point t_k, the same two consecutive edges of B, say e^*_j and e^*_{j+1}, simultaneously and constantly touch S_1 and S_2 (see Fig. A.1 (c)).

Proof: Similar to the proof of Lemma A.2.1.

Observation A.2.3: From Lemma A.2.1 it is easy to see that when the reference point O of λB_o lies between t_1 and infinity, sp_1 cuts a triangle from polygon λB_o and the edge vectors of the triangle are sp_1, e^*_i, and e^*_{i+1}. (see Fig. A.5)

A.3 Finding the Turning Points

The segments of the B-Voronoi diagram can be classified into three groups. The first one consists of the ray shooting from the first turning point to infinity. The second group contains the segments between the first and the last turning points. The third group consists of the ray shooting from the last turning point to infinity.

A.3.1 Finding the first turning point

We denote by v_λ the vertex of λB_o corresponding to vertex $v \in B$. The first step to find the B-Voronoi diagram is to identify the blocking vertex v^*_i. The following Lemma gives us an easy way to find vertex v^*_i. Let e^*_i and e^*_{i+1} be the two edges adjacent to v^*_i.

Lemma A.3.1: The two consecutive vectors e^*_i and e^*_{i+1} are the pair of consecutive edge vectors in B with the property that the slope of e^*_i is less than the slope of sp_1 and the slope of e^*_{i+1} is greater than the slope of sp_1.

Proof: From Lemma A.2.1 and Observation A.2.3, we know that vector sp_1 is the tangent vector of v^*_i, so the slope of sp_1 is between the slope of e^*_i and e^*_{i+1}. □

From Lemma A.3.1, we can see that vertex v^*_i can be found in time linear in the number of edges in B.

Observation A.3.1: Suppose v^*_i is the blocking vertex between S_1 and S_2. Then there are three

cases (see Fig. A.5). Case 1: $v^*_{i\lambda}$ does not touch S_1 nor S_2. Case 2: $v^*_{i\lambda}$ touches S_1. Case 3: $v^*_{i\lambda}$ touches S_2.

Sort the vectors in C_1, C_2, the vectors in B, and the supporting vectors sp_1 and sp_2, by the slope order. Put the sorted vectors in a unit circle (see Fig. A.3 (b)). The sorted vectors of $sp_1 \cup sp_2 \cup C_1$ (resp. $sp_1 \cup sp_2 \cup C_2$) in the unit circle which are adjacent to e^*_i (resp. e^*_{i+1}) are referred to as r_1 and r_2 (resp. r_3 and r_4). The vertex between r_1 and r_2 is referred to as q_1, and the vertex between r_3 and r_4 is q_2. In other words, e^*_i and e^*_{i+1} are the tangent vectors of q_1 and q_2 respectively. For example, in Figure A.3, edges b and c of polygon B are the two consecutive edges when λB_o touches the two obstacles before the reference point arrives at the first turning point, r_1 and r_2 are edges *7* and *8*, and r_3 and r_4 are edges *5* and sp_1. Therefore, finding q_1 and q_2 takes time linear in the number of edges in C_1 and C_2.

The following algorithm finds the location of $v^*_{i\lambda}$ and its blocking case.

Algorithm Find_The_Block_Case(e^*_i, e^*_{i+1}, q_1, q_2)

Begin

Draw a line, l_1, through q_1 and parallel to e^*_i;

Draw a line, l_2, through q_2 and parallel to e^*_{i+1};

Find the intersection w of the two lines;

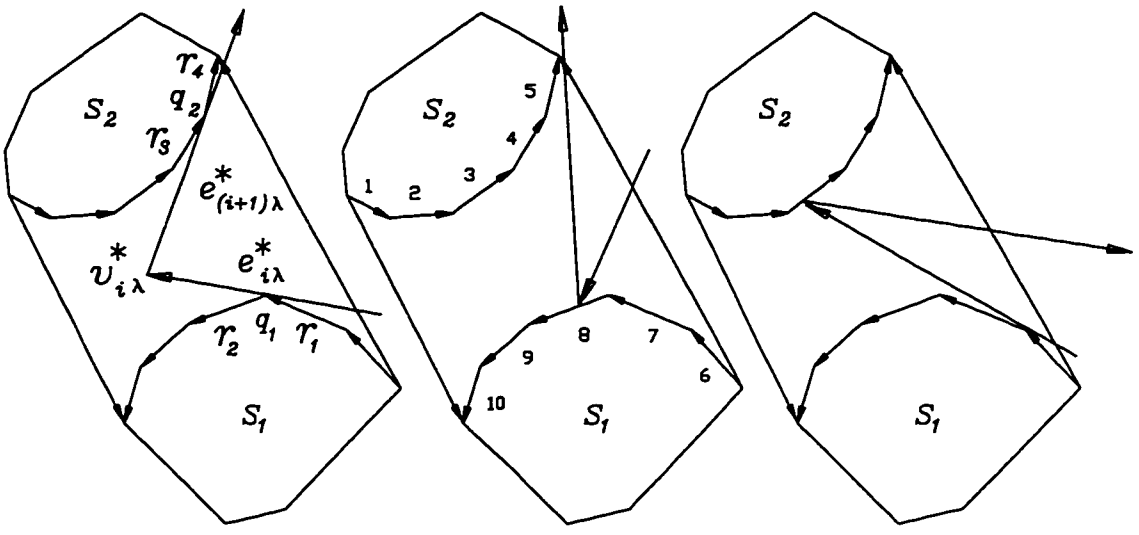
If q_1wq_2 makes a right turn **then**

$v^*_{i\lambda} = w$;

$v^*_{i\lambda}$ is case 1;

Stop;

Else



(a) case1 (b) case2 (c) case3

Figure A.5. Three cases for v^*_i

If there exists one segment, r_i, in C_1 intersects with l_2 at y and $e^*_i \times r_i < 0$ **then**

$v^*_{i\lambda} = y$

$v^*_{i\lambda}$ is case 2;

Else

$v^*_{i\lambda}$ = the intersection of C_2 and l_1;

$v^*_{i\lambda}$ is case 3;

End

Algorithm Find_The_Block_Case takes time linear in the number of edges in C_1 and C_2.

Since the objects are in *general position*, $v^*_{i\lambda}$ will not be coincident with the vertices in S_1 or S_2 when λB_o is on the *B*-Voronoi diagram. In order to find the position of the reference

point O (the first turning point t_1), for case 1 we need to find the ratio $q_1v^*_{i\lambda}/e^*_i$ and $v^*_{i\lambda}q_2/e^*_{i+1}$. The larger ratio will be λ', the scaling factor when B arrives at t_1. The scaling factor λ' for case 2 is $v^*_{i\lambda}q_2/e^*_{i+1}$ and for case 3 it is $q_1v^*_{i\lambda}/e^*_i$. The time for finding λ' is only $O(1)$ for each case. Once λ' is found, the ray shooting from the first turning point to infinity in the B-Voronoi diagram can be determined.

Since any vector in B is parallel to its corresponding vector in λB_o, t_1 is the intersection point of the line, passing through $v^*_{i\lambda}$ and parallel to v^*_iO in B, with another line, passing through the VV contact and parallel to its corresponding segment in B when the scaling factor is λ'. For example, in Figure A.6, t_1 is the intersection of l_2, which is parallel to v^*_iO in B, and of l_1, which passes through $v^*_{(i+1)\lambda}$ and is parallel to $v^*_{(i+1)}O$. We use ray(a, d) to describe a ray whose starting point is a and its direction is d. From Lemma A.2.1, we can see that when the scaling factor is greater than λ', the reference point of λB_o will lie on ray(t_1, $v^*_{i\lambda}t_1$). Thus, the ray shooting from t_1 to infinity in the B-Voronoi diagram is determined by ray(t_1, $v^*_{i\lambda}t_1$). The procedure is described in more detail below.

Algorithm Find_The_Ray_in_B-Vor(v^*_i, e^*_i, e^*_{i+1}, q_1, q_2)

Output: the first turning point t_1;

the ray shooting from t_1 to infinity;

Begin

Find_The_Block_Case(e^*_i, e^*_{i+1}, q_1, q_2);

If ($v^*_{i\lambda} \in$ blocking case1 and $q_1v^*_{i\lambda}/e^*_i < v^*_{i\lambda}q_2/e^*_{i+1}$) or ($v^*_{i\lambda} \in$ blocking case 2) **then**

$\lambda' = v^*_{i\lambda}q_2/e^*_{i+1}$;

t_1 = Find_The_Turning_Point(q_2, v^*_{i+1}, $v^*_{i\lambda}$, v^*_i);

Return(t_1, ray(t_1, $v^*_{i\lambda}t_1$));

Else

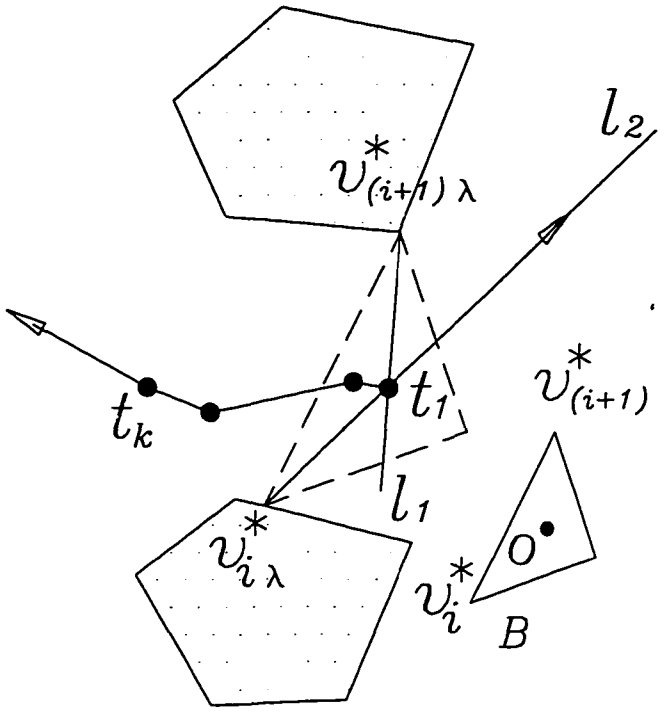


Figure A.6. Find the first turning point

$\lambda' = q_1 v^*_{i\lambda} / e^*_i$;

t_1 = Find_The_Turning_Point(q_1, v^*_{i-1}, $v^*_{i\lambda}$, v^*_i);

Return(t_1, ray(t_1, $v^*_{i\lambda} t_1$));

End

Algorithm Find_The_Turning_Point(z_1, z_2, z_3, z_4)

Begin

Draw a line l_1 from z_1 parallel to $z_2 o$;

Draw a line l_2 from z_3 parallel to $z_4 o$;

Return(the intersection point of l_1 and l_2);

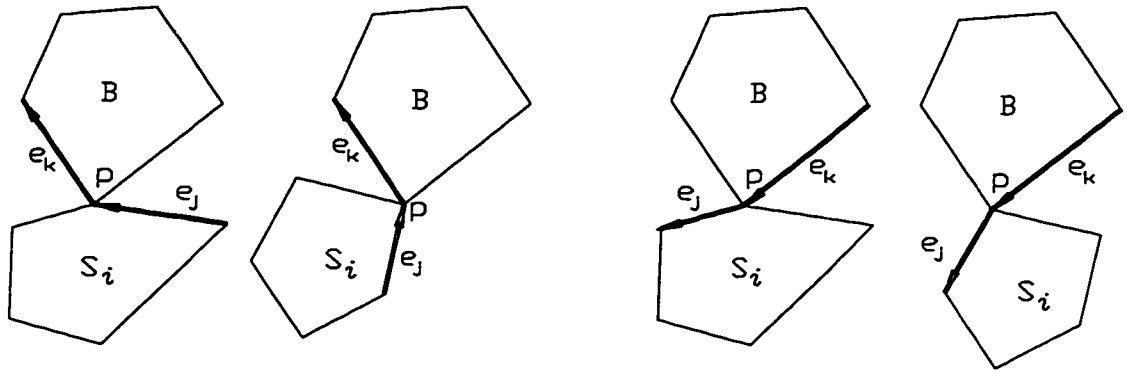
End

Since Algorithm *Find_The _Block_Case* is called in *Algorithm Find_The_Ray_in_B-Vor*, *Algorithm Find_The_Ray_in_B-Vor* also takes time linear in the number of edges in C_1 and C_2 to find the first turning point and the ray shooting from the first turning point to infinity.

A.3.2 Finding the interior turning points

Now we can slide B from t_1 into the passage until we find another two consecutive edges of B that touch S_1 and S_2 simultaneously. In Figure A.3, when B moves from the right-hand side of sp_1 to the right-hand side of sp_2, λB_o traces around S_1 counterclockwise and traces around S_2 clockwise. Since λB_o touches S_1 and S_2 simultaneously, there are two contact points: one between S_1 and λB_o and one between λB_o and S_2. The contact types of the moving object and the obstacles can be VV, VE or EV. Since the objects are in *general position*, the two contacts cannot both simultaneously be VV contacts, and there is no EE contact either. The contact vertex and the contact edge will be called the *tracing vertex* and the *tracing edge* respectively. Except for VV contacts, there are always *two* tracing vertices and *two* tracing edges during the motion. Observation A.3.2 determines the tracing condition during the motion.

Suppose there is a VV contact when B slides around the boundary of polygon S_i clockwise (resp. counterclockwise). Let p be the contact point, and let p_B be the vertex on B that coincides with p. Let p_S be the vertex on S_i that coincides with p. Let e_k be the edge in B just ahead (resp. behind) of p and let e_j be the edge in S_i just behind (resp. ahead) of p (see Fig. A.7). We have the following observation.



(a) B traces S_i in clockwise (b) B traces S_i in counterclockwise

Figure A.7. Tracing vertices and tracing edges

Observation A.3.2: If $e_k \times e_j$ is greater than zero (resp. less than zero), p_B will trace e_j, otherwise p_S will trace e_k after the VV contact.

Since the turning point occurs when there is a VV contact, we need to determine if the tracing vertex between B and S_1 reaches some vertex first or the tracing vertex between B and S_2 reaches some vertex first (see Fig. A.8). The following algorithm finds the two tracing vertices, two tracing edges, and the two vertices that the two tracing vertices will hit. In it, *start(e)* (resp. *end(e)*) denotes the start (resp. end) point of vector e.

Algorithm Find_Tracing_Vertices_and_Edges(e_1, e_2, e_3, e_4)

Input: e_1: vector in B coming just behind the latest VV contact point with S_1;

e_2: vector in B coming just ahead of the latest VV contact point with S_2;

e_3: vector in S_1 coming just ahead of the latest VV contact with B;

e_4: vector in S_2 coming just behind the latest VV contact with B;

Output: Trace_V[1]: tracing vertex between B and S_1;

Trace_V[2]: tracing vertex between B and S_2;

Trace_E[1]: tracing edge between B and S_1;

Trace_E[2]: tracing edge between B and S_2;

Next_V[1]: the next vertex that Trace_V[1] will hit;

Next_V[2]: the next vertex that Trace_V[2] will hit;

Begin

If $(e_1 \times e_3 > 0)$ **then**

Trace_V[1] = *start*(e_3);

Trace_E[1] = e_1;

Next_V[1] = *start*(e_1);

Else

Trace_V[1] = *end*(e_1);

Trace_E[1] = e_3;

Next_V[1] = *end*(e_3);

If $(e_2 \times e_4 > 0)$ **then**

Trace_V[2] = *start*(e_2);

Trace_E[2] = e_4;

Next_V[2] = *start*(e_4);

Else

Trace_V[2] = *end*(e_4);

Trace_E[2] = e_2;

Next_V[2] = *end*(e_2);

End

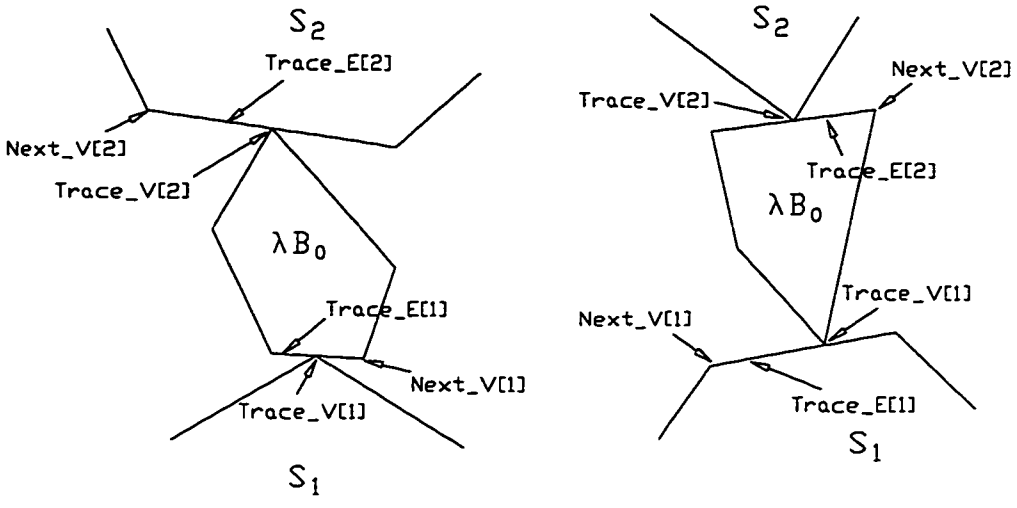


Figure A.8. One tracing vertex is in B, another one is in S

From *Algorithm Find_Tracing_Vertices_and_Edges*, we know that the next VV contact occurs when either Trace_V[1] coincides with Next_V[1] or Trace_V[2] coincides with Next_V[2]. It is now necessary to determine which one happens first.

If the two tracing vertices are all in B or all not in B, we can find the turning point through the line connecting the two tracing vertices because the line has a fixed orientation during the motion. Without loss of generality, suppose the two tracing vertices are all in B and Trace_V[1] hits Next_V[1] first. At this time Trace_V[2] should still be on Trace_E[2]. Since any vector in B is parallel to its corresponding vector in λB_o, we know that if we draw a line L, passing through Next_V[1] and parallel to Trace_V[1]Trace_V[2], L should intersect with Trace_E[2] and the intersection point will be the contact point between B and S_2 when Trace_V[1] coincides with Next_V[1]. If L does not intersect with Trace_E[2], that means Trace_V[2] hits Next_V[2] first. After the intersection point and the VV contact are determined, the turning point can be found by *Algorithm Find_The_Turning_Point.*

The same strategy is applied when one tracing vertex is in B and another one is not in B. L

will be the line passing through Trace_V[1] and parallel to Next_V[1]Trace_V[2]. If L intersects with Trace_E[2], Trace_V[1] will hit Next_V[1] first; otherwise Trace_V[2] will hit Next_V[2] first (see Fig. A.8). The whole procedure for finding all intermediate turning points takes $O(n)$ time.

A.3.3 Finding the last turning point

From Lemma A.2.2, we know that when there are another two consecutive edges in B touching S_1 and S_2 or only one vertex of B blocks in the closed region again, we have found the last turning point. We can therefore follow the same steps described in Section 3.1 to find the last turning point and the ray shooting from it to infinity.

A.4 Conclusion

This Appendix gives an $O(n)$ algorithm to find the exact description for every turning point in the B-Voronoi diagram when there are two obstacles and one homothetic robot. Once all of the turning points are found, and if all of the scaling factors are greater than one, we can move the robot along the path without collision.

REFERENCES

[1] C. Ahrikencheikh and A. A. Seireg, *Optimized-Motion Planning, Theory and Implementation*, John Wiley & Sons, Inc, New York, 1994.

[2] C. Ahrikencheikh, A. A. Seireg, and B. Ravani, "Optimal and Conforming Motion of a Point in a Constrained Plane," *Transactions of the ASME, Journal of Mechanical Design*, pp. 474-479, 1994.

[3] H. Alt and C. K. Yap, "Algorithmic Aspects of Motion Planning: a Tutorial: Part 1," *Algorithms Rev.*, Vol. 1, No. 2, pp. 43-60. 1990

[4] H. Alt and C. K. Yap, "Algorithmic Aspects of Motion Planning: a Tutorial: Part 2," *Algorithms Rev.*, Vol. 1, No. 2, pp. 61-77. 1990.

[5] M. Barbehenn and S. Hutchinson, "Efficient Search and Hierarchical Motion PLanning by Dynamically Maintaining Single-Source Shortest Paths Trees," *IEEE Transactions on Robotics and Automation*, Vol. 11, No. 2, pp. 198-214, Apr. 1995.

[6] J. Barraquand and J-C Latombe, "On Nonholonomic Mobile Robots and Optimal Maneuvering", *Revue d'Intelligence Artificielle*, Vol. 3, No. 2, pp. 77-103, 1989.

[7] J. Barraquand and J-C Latombe. "Nonholonomic Multibody Mobile Robots: Controllability and Motion Planning in the Presence of Obstacles," *Algorithmica*, Vol. 10, pp. 121-155, 1993.

[8] J. Bentley and T. Ottmann, "Algorithms for Reporting and Counting Geometric Intersections," *IEEE Transactions on Computers*, Vol. c-28, No. 9, pp. 643- 647, Sep. 1979.

[9] R. Brooks and T. Lozano-Perez, "A Subdivision Algorithm in Configuration Space for Findpath with Rotation," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-15, No. 2, Mar./ Apr., pp. 224-233, 1985.

[10] J. Canny, *The Complexity of Robot Motion Planning*, MIT Press, Cambridge, Massachusetts, 1988.

[11] J. Canny, "Collision Detection for Moving Polyhedra," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-8, No. 2, pp. 200-209, March 1986.

[12] J. Canny and M. Lin, "An Opportunistic Global Path Planner," *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 1554-1559, 1990.

[13] J. Canny, A. Rege, and J. Reif, "An Exact Algorithm for Kinodynamic Planning in the

Plane," *Proceeding of the ACM Symp. on Computational Geometry*, pp. 271-280, 1990.

[14] L. P. Chew and K. Kedem, "A Convex Polygon Among Polygonal Obstacles: Placement and High-Clearance Motion," *Computational Geometry: Theory and Applications*, Vol. 3, pp. 59-89, 1993.

[15] J. Chuang and N. Ahuja, "Path Planning Using the Newtonian Potential," *Proceedings of the IEEE International Conference on Robotics and Automation*, Vol. 1, 1991.

[16] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 1990.

[17] L. Dubins, "On Curve of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents," *Am. J. Math.* Vol. 79, pp. 697-516, 1957.

[18] R. Earnshaw, *Theoretical Foundations of Computer Graphics and CAD*, NATO ASI Series, Springer-Verlag, New York, 1987.

[19] M. Erdmann and T. Lozano-Perez, "On Multiple Moving Objects," *Algorithmica*, Vol. 2, pp. 477-521, 1987.

[20] B. Faverjon, "Obstacle Avoidance Using an Octree in The Configuration Space of a Manipulator," *IEEE Int. Conf. Robot. and Automat.*, Atlanta, pp. 504-512, 1984.

[21] B. Faverjon, "Object Level Programming of Industrial Robot," *IEEE Int. Conf. Robot. and Automat.*, pp. 1406-1412, 1986.

[22] S. Fortune and G. Wilfong, "Planning Constrained Motion," *Proc. ACM Symp. Theory of Compt.*, pp. 445-459, 1988.

[23] T. Fraichard, "Dynamic Trajectory Planning with Dynamic Constraints a 'a State-Time Space' Approach," *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, Japan, pp. 1393-1400, 1993

[24] T. Fraichard and C. Laugier, "Path-Velocity Decomposition Revisited and Applied to Dynamic Trajectory Planning," *IEEE Int. Conf. Robot. and Automat.*, Vol. 2, pp. 40-45, 1993.

[25] W. R. Franklin, V. Akman, and C. Verrilli, "Voronoi Diagrams with Barriers and on Polyhedra for Minimal Path Planning, " *The Visual Computer*, Vol. 1, pp. 133-150, 1985.

[26] L. Guibas, L. Ramshaw, and J. Stolfi, "A Kinetic Framework for Computational Geometry," *IEEE Symp. on FOCS*, pp. 100-111, 1983.

[27] V. Hayward, "Fast Collision Detection Scheme by Recursive Decomposition of a Manipular Workspace," *IEEE Int. Conf. Robot. and Automat.*, pp. 1044-1049, 1986.

[28] M. Held, J. Klosowski, and J. Mitchell, "Evaluation of Collision Detection Methods for Virtual Reality Fly-Throughs," Department of Applied Mathematics and Statistics, State University of New York, Stony Brook.

[29] H. Hirukawa and S. Kitamura, "Collision Avoidance Method for Robot Manipulators Based on the Safety First Algorithm and the Potential Function," *Advanced Robotics*, Vol. 4, No. 1, 1990.

[30] J. E. Hopcroft, J. T. Schwartz, and M. Sharir "On the Complexity of Motion Planning for Multiple Independent Object; PSPACE-Hardness of the "Warehouseman's Problem," *The International Journal of Robotics Research*, Vol. 3, No. 4, pp. 76-88, 1984.

[31] M. Houle, "Computing the Width of a Set," *Proceedings of the ACM Symp. on Computational Geometry*, pp. 1-7, 1985.

[32] Y. K. Hwang and N. Ahuja, "A Potential Field Approach to Path Planning," *IEEE Transactions on Robotics and Automation*, Vol. 8, pp. 23-32, Feb. 1992.

[33] Y. Kanayama and B. I. Hartmen, "Smooth Local Path Planning for Autonomous Vehicles," *Technical Report of the Department of Computer Science at University of California at Santa Barbara*, TRCS88-15, June 1988.

[34] K. Kedem, R. Livne, J. Pach, and M. Sharir, "On the Union of Jordan Regions and Collision-Free Translational Motion Amidst Polygonal Obstacles," *Discrete & Computational Geometry*, pp. 59-71, 1986.

[35] K. Kedem and M. Sharir, "An Efficient Algorithm for Planning Collision-free Translational Motion of a Convex Polygonal Object in 2-dimensional Space Amidst Polygonal Obstacles," *Proceedings of the ACM Symposium on Computational Geometry*, pp. 75- 80, 1985.

[36] K. Kedem and M. Sharir, "An Efficient Motion-Planning Algorithm for a Convex Polygonal Object in Two-Dimensional Polygonal Space," *Discrete & Computational Geometry*, Vol. 5, pp. 43-75, 1990.

[37] O. Khatib, "Real-Time Obstacle Avoidance for Manipulators and Mobile Robot," *International Journal of Robotics Research*, Vol. 5, No. 1, pp. 90-98, 1986.

[38] D. Kirkpatrick, "Efficient Computation of Continuous Skeletons," *Proceedings of the 20th Symp. on Foundations of Computer Science*, pp. 18-27, 1979.

[39] D. Kirkpatrick and J.Snoeyink, "Tentative Prune-and-Search for Computing Vertices," *Proceedings of the 9th Annual Symp. on Computational Geometry*, pp. 133-142, 1993.

[40] M. Kohler and M. Spreng, "Fast Computation of the C-Space of Convex 2D Algebraic Objects," *The International Journal of Robotics Research*, Vol. 14, No. 6, pp. 590-608, December 1995.

[41] K. Kondo, "Collision Avoidance by Free Space Enumeration Using Multiple Search Strategies," *Advanced Robotics*, Vol. 5, No. 4, 1991.

[42] K. Kondo and K. Ohtomi, "Motion Planning in Plant CAD Systems," *ASME Advances in Design Automation*, Vol. DE-32, No. 2, 1991.

[43] T. S. Ku and B. Ravani, "Model Based Rigid Body Guidance in Presence of Non-Convex Geometric Constraints," *ASME Advances in Design Automation*, DE-Vol. 14, pp. 67-79, 1988.

[44] J-C Latombe, *Robot Motion Planning*, Kluweer Academic Publishers, Boston, 1991.

[45] J-C Latombe, *Robot Algorithms, Algorithmic Foundations of Robotics*, K.Goldberg et al. (eds), AK Peters, Wellesey, 1995.

[46] J-P Laumond, "Finding Collision-Free Smooth Trajectories for a Non-Holonomic Mobile Robot," *Proc. Tenth International Joint Conference on Artificial Intelligence*, Milano, Italy, pp. 1120-1123, 1987.

[47] J-P. Laumond, P. E. Jacobs, M. Taix, and R. M. Murray, "A Motion Planner for Nonholonomic Mobile Robots," *IEEE Transactions on Robotics and Automation*, Vol. 10, No. 5, pp. 577- 593, October 1994.

[48] J-P. Laumond, T. Simeon, R. Chatila, and G. Giralt, "Trajectory Planning and Motion Control for Mobile Robot," *Geometry and Robotics*, J.D. Boissonnat and J. P. Laumond, Eds, Lecture Notes in Computer Science, Springer- Verlag, New York, Vol. 391, pp. 133-149, 1989.

[49] J-P Laumond and P. Soueres, "Metric induced by the Shortest Paths for a Car-Like Mobile Robot," *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, Yokoama, pp. 1299-1303, 1993.

[50] D. T. Lee and R. L. Drysdale, "Generalization of Voronoi Diagrams in the Plane," *SIAM J. Comput.*, Vol. 10, No. 1, pp. 73-87, Feb. 1981.

[51] J. Lenarcic and B. Ravani, *Advances in Robot Kinematics and Computational Geometry*, Kluwer Academic Publishers, Dordrecht, 1994.

[52] D. Leven and M. Sharir, "Planning a Purely Translational Motion for a Convex Object in Two-Dimensional Space Using Generalized Voronoi Diagrams", *Discrete & Computational Geometry*, Vol.2, pp. 9-31, 1987.

[53] D. Leven and M. Sharir, "An Efficient and Simple Motion Planning Algorithm for a Ladder Amidst Polygonal Barriers," *Journal of Algorihtms*, Vol. 8, pp. 192-215, 1987.

[54] T. Lozano-Perez, "Spatial Planning: A Configuration Space Approach," *IEEE Transactions On Computer*, Vol. c-32, No. 2, 1983.

[55] A. A. Maciejewski and C. A. Klein, "Obstacle Avoidance for Kinematically Redundant Manipulators in Dynamically Varying Environments," *International Journal of Robotics Research*, Vol. 4, No. 3, 1985.

[56] H. Martinez-Alfaro, Collision-Free Path Planning for Robots Using B-Splines and Simulated Annealing, Ph.D dissertation, Iowa State University, Ames, 1993.

[57] B. Mirtich and J. Canny, "Using Skeletons for Nonholonomic Path Planning Among Obstacles," *Proc. of the IEEE International Conference on Robotics and Automation*, pp. 2533- 2540, 1992.

[58] D. Mount, "Intersection Detection and Separators for Simple Polygons," *8th Annual Computational Geometry*, pp. 303-311, 1992.

[59] P. Moutarlier, B. Mirtich, and J.Canny, "Shortest Paths for a Car-Like Robot to Manifolds in Configuration Space", *The International Journal of Robotics Research*, Vol.15, No. 1, pp. 36-60, February, 1996.

[60] J. R. Munkres, *Topolygy: A First Course*, Prentice-Hall, Englewood Cliffs, NJ, 1975.

[61] J. Nievergelt and F. P. Preparata, "Plane-sweep Algorithms for Intersecting Geometric Figures," *Comm. ACM*, Vol. 25, pp.739- 747, 1982.

[62] C. O'Dunlaing, M. Sharir, and C.K. Yap, "Retraction: a New Approach to Motion-Planning," *Theory Comput.*, pp. 207-220, 1983.

[63] C. O'Dunlaing, M. Sharir, and C. Yap, "Generalized Voronoi Diagrams for a Ladder: II Efficient Construction of the Diagram," *Algorithmica*, Vol.2, pp. 27-59, 1987.

[64] C. O'Dunlaing and C.K. Yap, "A "Retraction" method for planning the motion of a disk," *J. of Algorithms*, Vol. 6, pp. 104-111, 1985.

[65] M. Okutomi and M. Mori, "Decision of Robot Movement by Means of a Potential Field," *JRSJ Advanced Robotics*, Vol. 1, No. 2, pp. 131-141, 1986.

[66] T. Ottman, P. Widmayer, and D. Wood, "A Fast Algorithm for Boolean Mask Operations," *Computer Vision*, Graphics and Image Processing, Vol. 30, pp. 249- 268, 1985.

[67] M. H. Overmars and P. Svestka, "A Probabilistic Learning Approach to Motion Planning," *Technical Report UU-CS-1994-03*, Department of Computer Science, Utrecht University.

[68] F. P. Preparata and M. I. Shamos, *Computational Geometry, an Introduction*, Springer-Verlag, New York, 1985.

[69] F. P. Preparata, *Advances in Computing Research*, Computational Geometry, Vol 1, 1983.

[70] A. Pruski and S. Rohmer, "Multivalue Coding: Application to Autonomous Robot Planning with Rotations," *Proceedings of the IEEE International Conference on Robotics and Automation*, Vol. 1, pp. 694-699, 1991.

[71] N. S. V. Rao, N. Stoltzfus, and S. S. Iyengar, " A "Retraction" Method for Learned Navigation in Unknown Terrains for a Circular Robot", *IEEE Transactions on Robotics and Automation*, Vol. 7, No. 5, pp. 699-707, Oct. 1991.

[72] J. A. Reeds and L. A. Shepp, " Optimal Paths for a Car that Goes both Forwards and Backwards," *Pacific Journal of Mathematics*, Vol. 145, No. 2, pp. 367-393, 1990.

[73] J. T. Schwartz and M. Sharir, "On The Piano Movers' Problem: I. The Special Case of a Rigid Polygonal Body Moving Amidst Polygonal Barriers," *Commun. Pure Appl. Math.*, Vol. 36, pp. 345-398, 1983.

[74] J. T. Schwartz and M. Sharir, "On The Piano Movers' Problem: II. General Techniques for Computing Topological Properties of Real Algebraic Manifolds," *Adv. Appl. Math.*, Vol. 4, pp. 298-351, 1983.

[75] J. T. Schwartz and M. Sharir, "Algorithmic Motion Planning in Robotics", *Handbook of Theoretical Computer Science*, pp. 393-430, 1990.

[76] J. T. Schwartz, M. Sharir, and J. E. Hopcroft, *Planning, Geometry, and Complexity of Robot Motion*, Ablex Publishing, Norwood, NJ, 1987.

[77] J.T. Schwartz and C. K. Yap, *Advances in Robotics, Algorithmic and Geometric Aspects of Robots*, Lawrence Erlbaum Assoc. Publishers, Hillsdale, NJ, 1987.

[78] M. Sharir, "Efficient Algorithms for Planning Purely Translational Collision-free Motion in Two and Three Dimensions", *Proc. IEEE Symp. on Robotics and Automation*, pp.1326-1331, 1987.

[79] M. Sharir, "Algorithmic Motion Planning in Robotics," *IEEE Computer*, pp. 9-20, 1989.

[80] M. Sharir and S Toledo, "Extremal Polygon Containment Problems," *Computational Geometry, Theory and Application*, Vol. 4, pp. 99-118, 1994.

[81] K. G. Shin and N. D. Mckay, "Minimum-Time Control of Robot Manipulators with Geometric Path Constrains," *IEEE Transactions on Automatic Control*, Vol. AC-30, No. 6, pp. 531-541, June 1985.

[82] T. Simeon, "Planning Collision Free Trajectories by a Configuration Space Approach," *Geometry and Robotics*, J.D. Boissonnat and J. P. Laumond, Eds, Lecture Notes in Computer Science, Springer Verlag, New York, Vol. 391, pp. 116-132, 1989.

[83] P. Soueres, J-Y. Fourquet, and J-P. Laumond, "Region of Accessibility for a Car-Like Robot," *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, Yokohama, Japan, pp. 1304-1309, 1993.

[84] P. Soueres and J.P. Laumond, "Shortest Path Synthesis for a Car-like Robot" in *IEEE Transaction on Automatic Control*, Vol. 41, No. 5, pp. 672-688, May 1996.

[85] M. Spong, F. Lewis, and C. Abdallah, *Robot Control, Dynamics, Motion Planning, and Analysis*, IEEE Press, NJ, 1993.

[86] A. Stappen and M. Overmars, "Motion Planning amidst Fat Obstacles," *Proceedings of the 10th annual Symposium on Computational Geometry*, pp. 31-40, 1994.

[87] A. Stewart, "Local Robustness and its Applications to Polyhedral Intersection," *International Journal of Computational Geometry & Applications*, Vol. 4, No. 1, pp. 87-118, 1994.

[88] P. Svestka and M. H. Overmars, "Motion Planning for Car-Like Robots Using a Probabilistic Learning Approach," *UU-CS-1994-33*, Department of Computer Science, Utrecht University, Netherlands.

[89] O. Takahashi and R.J. Schilling, "Motion Planning in a Plane Using Generalized Voronoi Diagrams", *IEEE Transactions on Robotics and Automation*, Vol. 5, No. 2, pp. 143-150, Apr. 1989.

[90] S. Toledo, "Extremal Polygon Containment Problems," *Proc. of the ACM symp. on Computational Geometry*, pp. 176-185, 1991.

[91] P. Tournassound and O. Jehl, "Motion Planning for a Mobile Robot with a Kinematic Constraint," *Proc. IEEE International Conference on Robotics and Automation*, pp. 1785-1790, 1988.

[92] P. Tournassound, "Motion Planning for a Mobile Robot with a Kinematic Constraint," *Geometry and Robotics*, J.D. Boissonnat and J. P. Laumond, Eds, Lecture Notes in Computer Science, Springer Verlag, New York, Vol. 391, pp. 150-171, 1989.

[93] M. Vendittelli and J.P. Laumond, "Visible positions for a car-like robot amidst obstacles", *2nd Workshop on Algorithmic Foundations of Robotics*, WAFR'96, Toulouse, July 1996.

[94] J. M. Vleugels, J. N. Kok, and M. H. Overmars, "Motion Planning Using a Colored Kohonen Network," Utrecht University, Department of Computer Science, Netherlands, 1993.

[95] J. M. Vleugels and M. H. Overmars, "Approximating Generalized Voronoi Diagrams in Any Dimension," UU-CS-1995-14, Utrecht University, Department of Computer Science, Netherlands, 1995.

[96] C. Wang, "Collision Dection of a Moving Polygon in the Presence of Polygonal Obstacles in the Plane," I*EEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 10, No. 6, pp. 571-580, June 1994.

[97] C. W. Warren, "Global Path Planning Using Artificial Potential Field," *IEEE Journal of Robotics and Automation*, pp. 316-321, 1989.

[98] G. T. Wilfong, "Motion Planning for an Autonomous Vehicle," *Proc. IEEE International Conference on Robotics and Automation*, pp. 529-533, 1988.

[99] C. K. Yap, "Algorithmic Motion Planning," *Advances in Robotics*, Vol. 1: Algorithmic and Geometric Aspects, J. T. Schwartz and C. K. Yap, Eds. Lawrence Erlbaum Assoc., Hillsdale, NJ, 1987.

[100] C.K. Yap, "How to Move a Chair Through a Door," *IEEE J. Robotics Automat.*, Vol. RA-3, No. 3, pp. 172-181, June 1987.

[101] C. K Yap, "An O(n *log* n) Algorithm for the Voronoi Diagram of a Set of Simple Curve Segments," *Discrete & Computational Geometry*, Vol. 2, pp. 365-393, 1987.

[102] D. Zhu and J-C Latombe, "New Heuristic Algorithms for Efficient Hierarchical Path Planning," *IEEE Transactions on Robotics and Automation*, Vol. 7, No. 1, pp. 9-20, Feb. 1991.