SEARCH PROBLEMS IN MISSION PLANNING AND NAVIGATION OF AUTONOMOUS AIRCRAFT

by

James A. Krozel

School of Aeronautics
Purdue University
West Lafayette, Indiana 47907
May 1988

A Final Report

Submitted to NASA-Ames Research Center
Moffet Field, California
Under Cooperative Agreement
NCC 2-367

Principal Investigator
Dominick Andrisani
Period of Performance
July 1, 1985 - May 31, 1987

SEARCH PROBLEMS IN MISSION PLANNING AND NAVIGATION OF AUTONOMOUS AIRCRAFT

A Thesis

Submitted to the Faculty

of

Purdue University

by

James A. Krozel

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Aeronautics and Astronautics

May 1988

dedicated to Sandi

ACKNOWLEDGMENTS

This research was conducted at the NASA Ames Research Center, Moffett Field, CA under a cooperative program with Purdue University, W. Lafayette, IN (NASA Grant NCC 2-357). The author wishes to thank Dr. J. V. Lebacqz, Chief of the Flight Dynamics and Control Branch at NASA Ames, for his coordinating efforts with this research. Also, the author greatly appreciates the assistance of Dr. P. Cheeseman, Research Scientist of the Research Institute for Advanced Computer Science, and his guidance in the field of artificial intelligence applications and programming techniques.

This investigation using artificial intelligence techniques in these aeronautical engineering problems was pursued due to the inspiration of Dr. D. Andrisani II of Purdue University. Dr. Andrisani's guidance and motivation in the aeronautical engineering and artificial intelligence fields has greatly effected my academic success since the beginning of my college education.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Appendix Figure — Page

LIST OF SYMBOLS

Symbol	
A,B,...	Arbitrary nodes for a search graph.
$A, B, \ldots$	Arbitrary cities for a search graph.
A^*	An admissible search algorithm.
AFWAL	Air Force Wright Aeronautical Laboratories.
AHS	Autonomous Helicopter System.
AI	Artificial Intelligence.
C	The Traveling Salesman Problem cost matrix.
$C(i, n, m)$	The cost between node n at stage i and node m at stage $(i+1)$ for a uniform grid search.
$C(n, m)$	The cost between node n and node m of an arbitrary search graph.
c_{ij}	An element of the cost matrix C.
$c_{\min}$	The minimum cost element in the cost matrix C.
$\overline{c}_{\min_{COL}}$	The vector of column minimums of the cost matrix C, sorted in increasing order.
$\overline{c}_{\min_{COL}}(i)$	The i th element of $\overline{c}_{\min_{COL}}$.
$\overline{c}_{\min_{ROW}}$	The vector of row minimums of the cost matrix C, sorted in increasing order.
$\overline{c}_{\min_{ROW}}(i)$	The i th element of $\overline{c}_{\min_{ROW}}$.
$\overline{c}_{\min_{ROW/COL}}$	A heuristic based on $\overline{c}_{\min_{ROW}}$ and $\overline{c}_{\min_{COL}}$.
d	Depth of a node in a search tree.
$d(n)$	Depth of the node n in a search tree.

Symbol	
DARPA	Defense Advanced Research Projects Agency.
DMA	Digital Map Agency.
δ	The closest distance from one obstacle to another obstacle.
f	An estimate of the evaluation function: $f(n) = g(n) + h(n)$.
f^*	The evaluation function: $f^*(n) = g^*(n) + h^*(n)$.
F	The finish node of a search graph.
g	An estimate of the cost function: an estimate of the minimal cost path from the start node **S** to node n.
g^*	The cost function: the minimal cost path from the start node **S** to node n.
$g(i, n)$	The minimum cost from node n at stage i to the final node **F** in a uniform grid search.
G	An arbitrary search graph.
GPS	Global Positioning System.
h	An estimate of the heuristic function: the cost of a minimal cost path from node n to the goal node **F**.
h^*	The heuristic function: the cost of a minimal cost path from node n to the goal node **F**.
I,II,...	Stages of a dynamic programming search.
k	An arbitrary constant.
m, n	Arbitrary nodes of a search graph.
M	A set of successors of a node in a graph.
N	The arbitrary size of the N-city Traveling Salesman Problem.
O_x	The state operator that modifies a state by adding the city x to a tour.
$O(\,)$	Algorithmic time complexity.
p_i	An arbitrary point in the Euclidean plane.
R_N	The ratio of the number of nodes investigated by the A^* algorithm to the number of nodes investigated by Dijkstra's algorithm for a given search problem.

Symbol	
R_P	The ratio of the number of pointers investigated by the A^* algorithm to the number of pointers investigated by Dijkstra's algorithm for a given search problem.
RAV	Robotic Air Vehicle.
RPV	Remotely Piloted Vehicle.
s_i, s_j, s_k	Verticies of a Delaunay triangle.
S	A set of points.
S	The start node of a search graph.
S	A set of cities.
$\bar{S}$	The compliment of the set of cities S.
$SUCC(i, n)$	The successor node at stage $(i+1)$ to the node n at stage i.
T	A node of interest in the Voronoi diagram search graph.
$V(p_i)$	The Voronoi region associated with the point p_i.
X	An assignment matrix.
x_{ij}	An element of the assignment matrix X.
$[x]$	The largest integer no greater than x.
$\varnothing$	The empty set.
$\cup$	The union operator for two sets.
$\cap$	The intersection operator for two sets.

ABSTRACT

Krozel, James A., M.S.A.A., Purdue University. May 1988. Search Problems in Mission Planning and Navigation of Autonomous Aircraft. Advisory Committee Chairman: Prof. Dominick Andrisani II.

An architecture for the control of an autonomous aircraft is presented. The architecture is a hierarchical system representing an anthropomorphic breakdown of the control problem into planner, navigator, and pilot systems. The planner system determines high level global plans from overall mission objectives. This abstract mission planning is investigated by focusing on the Traveling Salesman Problem with variations on local and global constraints. Tree search techniques are applied including the breadth first, depth first, and best first algorithms. The minimum column and row entries for the Traveling Salesman Problem cost matrix provides a powerful heuristic to guide these search techniques. Mission planning subgoals are directed from the planner to the navigator for planning routes in mountainous terrain with threats. Terrain/threat information is abstracted into a graph of possible paths for which graph searches are performed. It is shown that paths can be well represented by a search graph based on the Voronoi diagram of points representing the vertices of mountain boundaries. A comparison of Dijkstra's dynamic programming algorithm and the A^* graph search algorithm from artificial intelligence/operations research is performed for several navigation path planning examples. These examples illustrate paths that minimize a combination of distance and exposure to threats. Finally, the pilot system synthesizes the flight trajectory by creating the control commands to fly the aircraft.

CHAPTER 1

INTRODUCTION

An autonomous aircraft, a pilotless aircraft with no remote operator, is a fascinating concept. The desire for autonomous aircraft stems from the possibilities for their use in military and civilian applications. Applications exploit the salient feature that there is no pilot to endanger in the mission of an autonomous aircraft. Without the need of an accompanying pilot, design constraints are dramatically changed: pilot housing and interfacing are unnecessary as is the consideration of the g-factor limitation, which would produce pilot blackout. A vehicle can be designed to achieve higher speeds, higher maneuverability, and reduced target signature. These benefits suggest several military applications such as deception, reconnaissance, electronic intelligence, or any mission where the risk of losing a pilot is too great. Examples of such missions include deceptive attacks prior to piloted aircraft attacks, visual citings of tank and troop movements, electronic intelligence gathering over or near battlefields, surface-to-air missile battery suppression, and missions in chemical warfare environments. Civilian applications include surveillance, searching over large areas, and investigating hazardous environments. Examples of such missions include patrolling illegal alien activity over national borders, searching for downed aircraft, and hurricane weather research.

One possible direction toward developing an autonomous aircraft would be to start with existing remotely piloted vehicles (RPVs) and extend their capabilities to an autonomous state. The Defense Advanced Research Projects Agency (DARPA) is currently conducting research in low-cost RPVs which must stay airborne longer than current drones [21]. DARPA seeks to endow their RPVs with the ability to alter their mission plans independently of ground controller interaction. This could evolve into an autonomous aircraft requiring no remote pilot for navigation. The remote controller or pilot may play a lesser role: as a supervisor to the actions an autonomous aircraft performs on its own, or possibly as a supervisor to several autonomous aircraft.

Although the development of a smarter RPV may lead into the development of an autonomous aircraft based on the RPV mission objectives, there is also interest in

developing a vehicle which is a general purpose autonomous aircraft. Progress in computer architectures for parallel and symbolic processing has evolved to the point where prototype autonomous air vehicles are now being developed.

The objective of a Robotic Air Vehicle (RAV) program of DARPA/AFWAL (Air Force Wright Aeronautical Laboratories) is to design, implement, and demonstrate an expert system for piloting robotic air vehicles [5,38,39]. The approach used for this research is to combine passive terrain following and navigation with an artificial intelligence expert system "pilot." U.S. Air Force fighter pilots provide the expertise for the expert system, and U.S. Air Force standards for evaluating pilot performance establish the validation measure for the system.

Researchers at the Georgia Tech Research Institute have been developing the Autonomous Helicopter System (AHS) through the use of artificial intelligence technology [16,17]. The AHS is a simulation system integrating the tasks of vision, planning, and control. The AHS uses a knowledge-based terrain navigation system that analyzes digital terrain maps, performs scene interpretation to generate path routes, maintains a record of its position through scene matching, and validates its plans confirming predetermined mission goals.

Artificial intelligence (AI) techniques are playing an important role in the development of autonomous vehicles. The use of AI techniques is not, however, applicable in all areas of the development. With respect to the control of an autonomous vehicle, the use of AI techniques is most applicable to higher level control tasks. The role of AI in the control system of an autonomous aircraft will be discussed further in the next section.

AI in the Control Loop

Aircraft operation involves the guidance and control of a complex physical system. In general, this task requires controlling systems which have varying characteristic time constants. Naturally, the systems with the slowest time constants appear in the outermost loops of a control system, commanding inner loop systems with faster time constants.

For aircraft, guidance is the action of determining the course and speed, relative to some reference frame, to be followed by the vehicle. Guidance tasks are performed in mission planning loops, navigation loops, and flight path loops. These outer loops require the inner loop control of lateral and longitudinal modes of the airframe. These loop closures are shown in Figure 1.1. †

† All figures and tables appear at the back of the chapter in which they are first cited.

The inner loop control systems for an aircraft tend to control or change the fastest modes of motion of the vehicle. Mathematical models describe the physical dynamics of the aircraft, which typically exhibit dynamic modes with time constants of the order of 0.1 second, sometimes quicker. Fast numerical algorithms, such as recursive digital flight control laws, legislate the control commands. The feedback control is reactive — the control involves correcting for errors between the plant state and the desired state.

In contrast to the inner loop characteristics, the characteristics of the outer loops include slower time constants, and involve planning and reasoning. Flight path commands are less frequent; time constants between 1 second and 60 seconds may be expected. Planning prevails in the outermost loops, where reasoning about mission objectives, flight maneuvers, and alternative flight paths is necessary — all within the performance capabilities of the aircraft. Whereas algorithmic reasoning is often unsuitable, symbolic processing and expert systems are applicable. At times, in mission planning there is a need to reason with incomplete and often inexact data. Thus, heuristic solution techniques common to the artificial intelligence field may be useful in solving planning problems in guidance.

At this point, one can see that AI techniques may be useful in the navigation and mission planning loops. The next sections will explain in more detail how a control structure for an autonomous aircraft can be constructed and where some AI techniques could be useful in solving some related problems.

A Hierarchical Autonomous Control System

An anthropomorphic breakdown of the control problem for an autonomous aircraft results in a composition of planner, navigator, and pilot systems. As shown in Figure 1.2, a hierarchical structure for controlling an aircraft emerges. The hierarchy is arranged in order of decreasing time constants of respective systems. The planner system determines high level global plans from overall mission objectives. A broad view of the world is invoked. For example, the planner may use a map labeled with strategic locations, mountain ranges, danger zones, and other general information to determine an order of flight destinations that should be followed for a mission. The planner commands the navigator to perform intermediate level planning based on a more detailed view of the world. For example, the navigator may use Digital Map Agency (DMA) map data and detailed threat information to plan a flight through mountainous terrain. The navigator specifies waypoints for a path to be generated by the pilot system. Finally, the pilot system synthesizes the actual aircraft trajectory

sending commands to generate surface deflections and engine adjustments that result in airframe forces and moments. The pilot system uses a narrow, but detailed view of the world. For example, the pilot system might be a terrain following/terrain avoidance system which uses DMA map data, forward-looking radar data, and radar altimeter data in the immediate look-ahead region of flight. One should note that although the control system is composed of planner, navigator, and pilot systems, these systems are *not* designed to replace or model actual human mission planners, navigators, or pilots. The mission planner, navigator, and pilot systems are simply designed to perform the tasks described above.

The planner, navigator, and pilot systems work in parallel, sending goals and constraints to the next lower system and status reports back to higher systems. The general flow of control for any of these systems is shown in Figure 1.3 (suggested by [11]). Whereas the diagram is the same for each system, the level of abstraction is different. For instance, the maps and information used for planning the global mission are different from that of navigation planning, so the expected world state and observed world state components are at different levels of abstraction for the planner and navigator.

Each control system is linked to a knowledge base, which stores global information and information about the expected and observed world states. Global information may include facts about the aircrafts performance capabilities, information about other airplanes abilities, weather patterns, etc. The expected world state may be composed of a model from map data and terrain feature data, while the observed world state may be composed of a model based on results of the state assessment system. The state assessment system coordinates sensor readings and performs sensor interpretation, possibly using external data, e.g. Global Positioning System (GPS) data, in order to establish an observed world state.

Additional components of each system include a comparator component, an emergency response component, a diagnoser, and a command generator. The comparator analyzes the expected world state and the observed world state and either reports to the command generator, the emergency response component, or the diagnoser. If the differences in world state require simple adjustments, then the command generator combines this information with the nominal plans to form goals and constraints for the next hierarchical system. If the differences indicate a recognized anomaly, then the emergency response component produces an emergency response to the command generator. If the differences indicate an unrecognized anomaly, then the diagnoser must reason about the anomaly and derive modified goals and constraints for replanning.

Thesis Organization

This thesis focuses on the planning and navigation systems of the hierarchical structure presented above. Mission planning is treated only to review some tree search techniques related to the planner system. Greater emphasis is given to the problem of navigation in mountainous terrain, which is representative of the navigator system. No problem representative of the pilot system is presented.

The planner determines high level global plans based on mission objectives, as described previously. A mission plan may be composed of, say, a sequence of destinations that satisfies some local and global constraints. It is proposed that the Traveling Salesman Problem is typical of such a mission planning problem. The Traveling Salesman Problem is stated in Chapter 2, and a survey of solution techniques is presented in Chapter 3. Variations of the Traveling Salesman Problem are investigated and presented in Chapter 4.

After a sequence of destinations is generated, flight paths between these locations are generated by the navigator system. One of the more difficult scenarios for a mission plan would require a flight that progresses through mountainous terrain at low altitude. Chapter 2 gives a problem statement for a navigation problem in mountainous terrain with threats. Chapter 3 surveys approaches to navigation path planning. Finally, a solution technique is presented in Chapter 5 through Chapter 7.

Conclusions from the investigation of the Traveling Salesman Problem and the navigation problem will be presented in Chapter 8, and recommendations for further related research is stated in Chapter 9.

The appendices of this thesis provide for further reading on topics related to the content of this thesis, but not significant enough to include as individual chapters. Appendix A presents the simulated annealing algorithm and its application to the Traveling Salesman Problem. Appendix B presents the construction of Delaunay Triangulations and Voronoi Diagrams, which is the basis for the navigation solution technique presented in Chapter 5 through Chapter 7. Appendix C presents proofs of the feasibility of Voronoi diagram search graphs, which is cited in Chapter 5. Finally, Appendix D states some properties of the A^* algorithm, which is a general graph search algorithm used in the navigation solution technique.

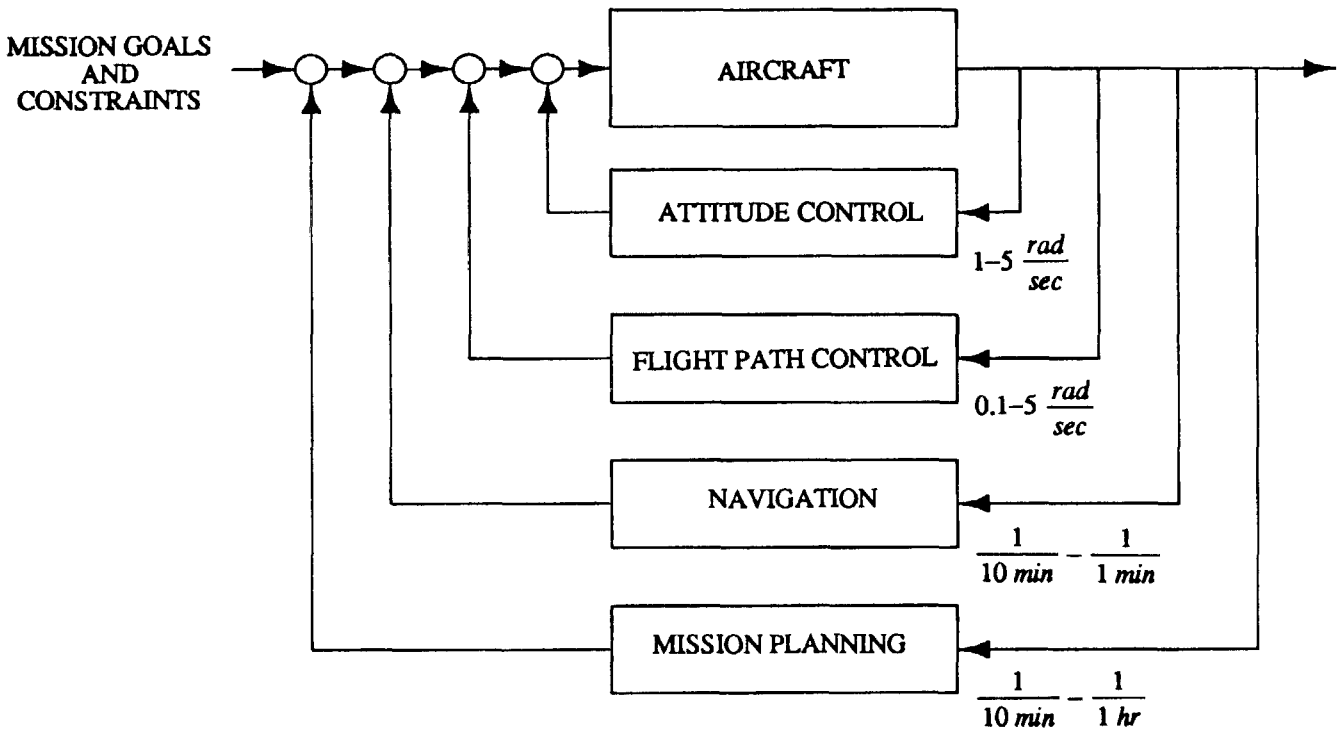


Figure 1.1. Flight path guidance and control loops for a typical fighter aircraft.

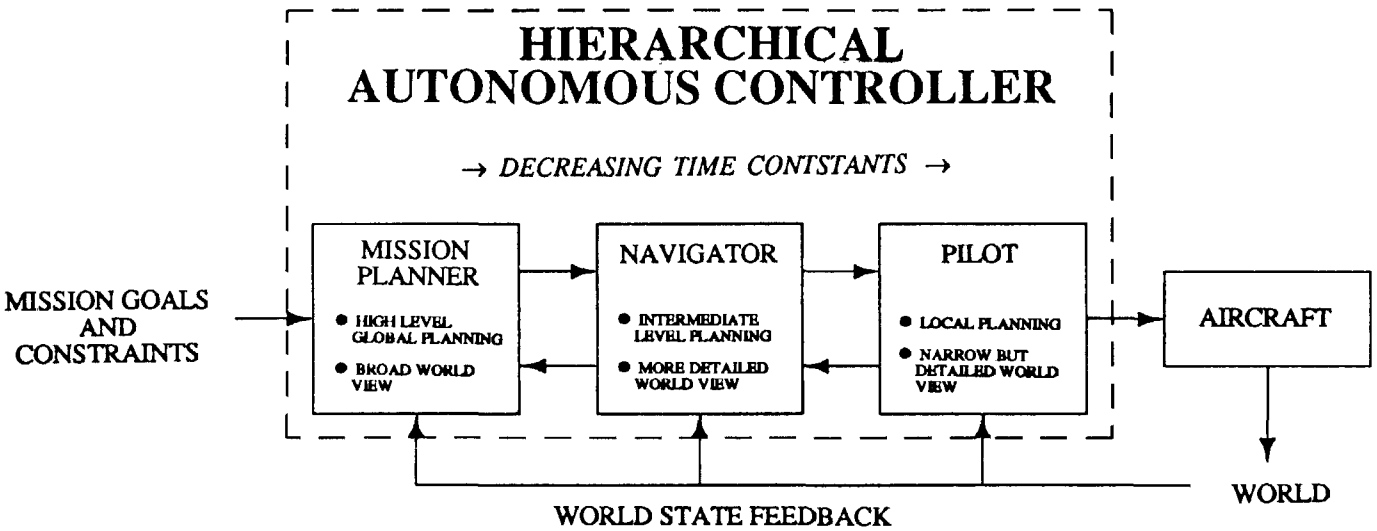


Figure 1.2. A hierarchical control structure for an autonomous aircraft.

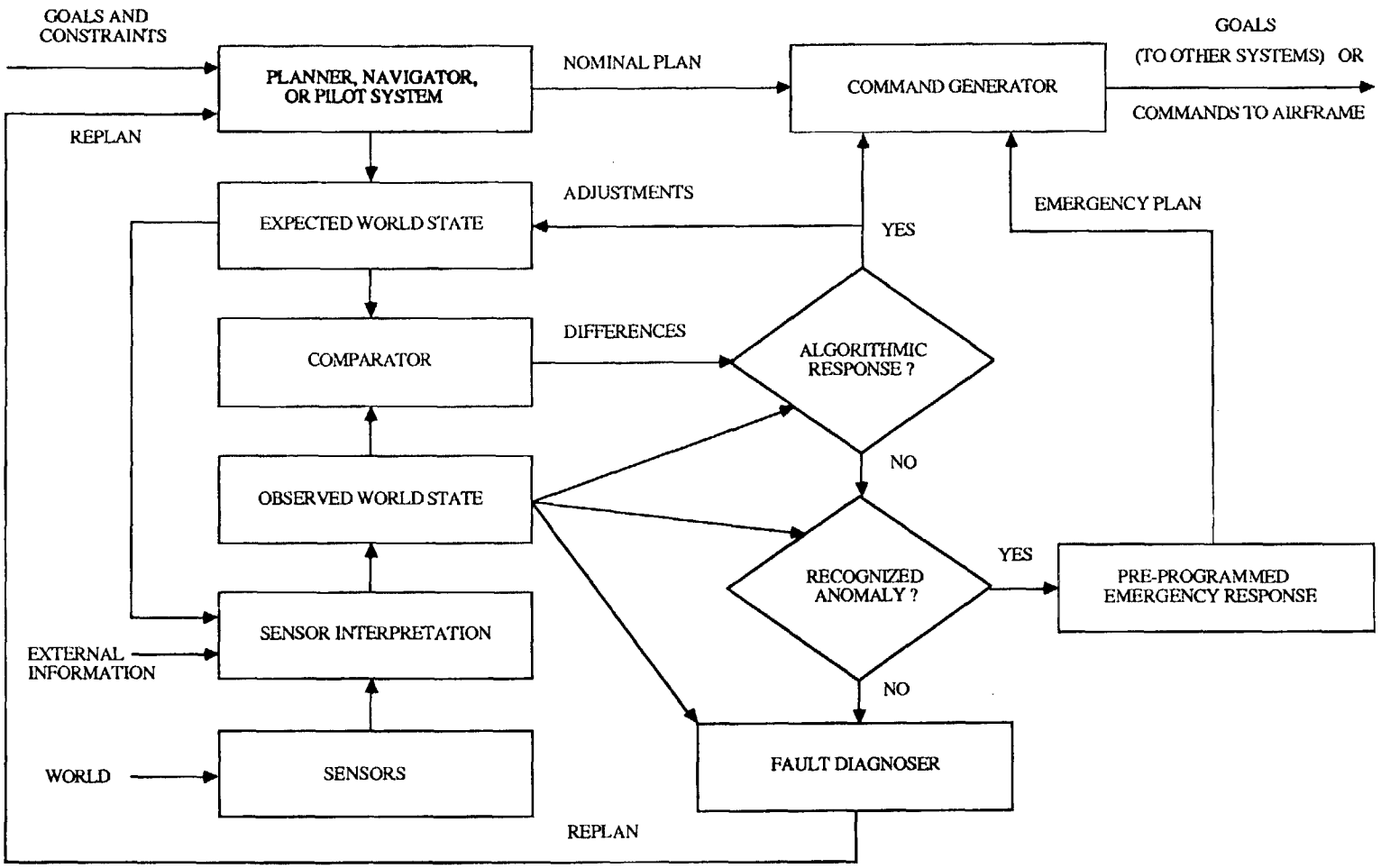


Figure 1.3. Flow of control with anomaly handling for a hierarchical control system (planner, navigator, or pilot).

CHAPTER 2

PROBLEM STATEMENTS

In this chapter a concise definition of the mission planning and navigation problems will be stated. First, general descriptions of these problems are given so that the terminology needed to understand these problems is introduced. Then, concise problem statements follow.

The Mission Planning Problem

The mission planning problem consists of planning *events* that compose a mission in such a way as to optimize some constraints associated with the plan. Consider a sequence of events to be a *state*. The solution criteria for a plan involves satisfying possible local and global constraints based on the state. For example, a *local constraint* may restrict the ordering of a particular set of events, such as, event A must not follow event B in a mission plan. A *global constraint* may require that the entire state meet some criteria, say, the mission plan is not acceptable if any event is left out of the plan. Finally, a particular state that satisfies the local and global constraints of the solution criteria is called a *solution state* or a *goal state*. Of course, more than one solution state may exist depending on the problem. Using these terms, the mission planning problem may be stated in general:

> Arrange a sequence of events for a solution state such that local and global constraints are satisfied.

This problem statement is general enough to encompass a wide variety of problems. This thesis will focus on a problem that is from this class of problems but more narrow in its stated application area. Hopefully, the ability to transfer the results of the particular problem back to similar problems of this class will not be difficult.

From the class of problems stated above, the particular problem that will be treated in this thesis is the Traveling Salesman Problem. In its simplest form, the

$N-$city Traveling Salesman Problem is stated as:

> A salesman, starting at one city, wishes to visit each of $N-1$ other cities once and only once and return to the start. In what order should the salesman visit the cities to minimize the total cost of visiting all the cities?

This problem will be solved along with variations of this problem where local and global constraints will be introduced.

The Navigation Path Planning Problem

One statement of the navigation problem is to find a path from an initial start location to a final destination location through mountainous terrain while optimizing parameters associated with the trajectory. These parameters may include altitude, speed, time, exposure to threats, and fuel used. A navigation path solution is composed of a sequence of waypoints describing a path. At each waypoint the altitude, heading, and velocity of the vehicle is specified.

The particular problem solved will be a problem much more constrained than the general statement above. The navigation problem will be to find a path from a start location to a finish location on a mountainous terrain map minimizing the path length and the exposure to threats. The problem considered is constrained to a constant altitude. The mountains are described by a terrain contour map and the threats are stationary, ground based, and are described by danger regions on the contour map. For any segment of a trajectory that crosses a threat area, it is assumed that a cost associated with danger can be assigned to that segment.

CHAPTER 3

A SURVEY OF MISSION PLANNING AND NAVIGATION APPROACHES

This chapter surveys several solution techniques for mission planning and navigation path planning. The specific mission planning problem treated is the Traveling Salesman Problem. The navigation path planning approaches surveyed include methods for planning the path of an object amongst polygon obstacles and methods for mobile robot terrain navigation.

The Traveling Salesman Problem: A Mission Planning Problem

The Traveling Salesman Problem, as stated in the preceding problem statement, minimizes the total cost of visiting all the cities in a tour. For surveys of Traveling Salesman Problem solution techniques and related problems see [4,18,32,55]. In general, the cost metric may be related to distance, time, direction, airfare, or other notions of cost. However, for the purpose of this background discussion, the cost will be associated with the distance traveled. (All figures with cities are understood to have a cost proportional to the Euclidean distance between the city locations as presented.)

The cost data for the $N-$city Traveling Salesman Problem is contained within an $N \times N$ cost matrix C, where c_{ij} is the cost to travel from city i to city j. Since the cost depends only on distance and, in particular, not on direction, the cost matrix C will be symmetric, i.e. $c_{ij}=c_{ji}$. The diagonal elements c_{ii} are assigned an infinite value to maintain that they are not to be considered in a tour. Although the Euclidean distance cost metric does not allow an asymmetric cost matrix, an asymmetric cost matrix can result in other cost metrics. For instance, if the cost is airfare, the cost matrix may be asymmetric because the cost to travel from city i to city j may not be the cost to travel from city j to city i. The Traveling Salesman Problem with an asymmetric cost matrix will not be discussed here, but can be reviewed in [32].

The Traveling Salesman Problem is a special case of a broader problem known as the assignment problem [49], a standard combinatorial optimization problem. The assignment problem is to choose N elements from an $N \times N$ cost matrix C, one from each row and column, so that the sum of the chosen elements is a minimum. To formulate the optimization problem, an assignment matrix X is introduced: let $x_{ij}=1$ if the element with corresponding cost c_{ij} is chosen as an assignment, otherwise let $x_{ij}=0$. Note that the assignment matrix is a permutation matrix. The total cost C_0 to be minimized is:

$$C_0 = \sum_{i=1}^{N} \sum_{j=1}^{N} c_{ij} x_{ij}.$$

Now the equalities

$$\sum_{i=1}^{N} x_{ij} = 1 \quad j = 1,2,...,N \text{ and}$$

$$\sum_{j=1}^{N} x_{ij} = 1 \quad i = 1,2,...,N$$

must be satisfied since only one element is selected from any particular column and row respectively.

The Traveling Salesman Problem restricts the assignment problem in the following manner. Now $x_{ij}=1$ indicates that the salesman travels from city i to city j. The Traveling Salesman Problem restricts the assignment problem by requiring that no subtours be allowed. Mathematically, this requires X to be a cyclic permutation matrix †. Alternatively, we can define $(S, \bar{S})$ to be a nontrivial partitioning of the integers $1,2,...,N$: that is, $S \cap \bar{S} = \varnothing$ and $S \cup \bar{S} = \{1,2,...,N\}$. If we let $(S, \bar{S})$ represent the set of all N cities, then the no subtour restriction is represented by:

$$\sum_{i \in S,\ j \in \bar{S}} x_{ij} \geq 1.$$

---

† A *cyclic permutation matrix* is a permutation matrix with elements describing a cycle for a graph. For the assignment matrix X, we associate the assignment x_{ij} with the arc from node i to node j in a graph. Consider the assignment matrices:

$$X_1 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad X_2 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

Assignment matrix X_1 is a cyclic permutation matrix because it forms a cycle linking node 1 to node 4 to node 3 to node 2 back to node 1, however, the assignment matrix X_2 is *not* a cyclic permutation matrix (although it is a permutation matrix) because it forms two cycles, one from node 1 to node 2 back to node 1 and another from node 3 to node 4 back to node 3.

There are 2^{N-1} inequality constraints imposed by this restriction.

Stated another way, the Traveling Salesman Problem is the problem of finding the Hamiltonian cycle [1] of shortest length for the graph defined with arc lengths corresponding to the cost matrix entries. This formulation is posed in terms of graph theory terminology, from which several heuristics have been developed. In particular, the minimum spanning tree [1] of a graph and the Eulerian cycle [1] have been useful.

The Traveling Salesman Problem is easy to formulate, yet it is difficult to solve. The mathematical statement provided above formulates the Traveling Salesman Problem into an integer linear programming problem [32,49]. However, because the number of inequality constraints (2^{N-1}) grows exponentially, linear programming is considered an intractable solution technique when N is large. Another technique, dynamic programming [22,49], is also not feasible for large N, since the recursive solution of dynamic programming imposes extremely large storage space requirements on a computer.

Branch-and-bound algorithms [32] can also be used to solve the Traveling Salesman Problem [36,49]. These algorithms subdivide the problem according to potential subsets of tours. For example, one subset of tours may consider only those tours which include a particular edge between two cities, i.e., the set of tours that *must* include a visit from city i to city j. The other subset of tours considers only those tours which do *not* include a particular edge between two cities, i.e., the set of tours that does *not* allow a visit from city i to city j. Bounds are established for the tour costs of these subsets, with the hope that some subsets can be eliminated from the search based on the bounds. Subsets are divided into subproblems until they can be solved easily. An optimal solution is obtained, provided that the branching does not become intolerable for storage. Finding optimal solutions can be expedited by a judicious selection of which subsets to explore first. For large problems, this becomes necessary to keep the problem manageable within computer limitations. Chapter 4 will give examples of search techniques based on this principle.

What should one attribute as the underlying difficulty with solving the Traveling Salesman Problem? One can see that the number of possible tours is large. There are $(N-1)!$ possible tours, and there may be more than one optimal tour. Simply exploring all possible paths to find the optimal tour is not feasible. Additionally, it has been shown that the Traveling Salesman Problem is in the class of $NP-$complete problems (nondeterministic polynomial time complete), for which it is conjectured that *no* polynomial algorithm can solve [15,32]. Finding an optimal solution for a problem that is $NP-$complete would in the worst case require an exponential amount of time, thus, $NP-$complete problems are considered to be inherently intractable from a

computational point of view [15].

Because of the complexity of the Traveling Salesman Problem, approximate solution techniques — rather than optimal solution techniques — must be considered. Approximate solution techniques, called tour composite procedures, invoke tour construction and tour improvement procedures. Tour construction procedures generate approximately optimal tours given the cost matrix C. Tour improvement procedures start with a feasible tour and systematically modify it using a sequence of city interchanges. Some tour construction procedures include the arbitrary insertion procedure, Christofides' procedure, Clarke and Wright Savings procedure, convex hull procedure, greatest angle insertion procedure, nearest neighbor procedure, ratio times difference insertion procedure, and several minimum spanning tree procedures, including the depth first traversal procedure, nearest insertion procedure, nearest merger procedure, nearest addition procedure, and cheapest insertion procedure. Some tour improvement procedures include the $k-opt$ and $Or-opt$ edge exchange procedures, the simulated annealing procedure, a neural-network procedure, and genetic search procedures. Detailed descriptions of these procedures are given in [18,19,24,32]. A few examples of these procedures follow.

An example of a simple heuristic for a tour construction is the nearest neighbor procedure [4]. The procedure is to build a tour from the start city to the nearest (or least costly) neighbor city, then from this city to its nearest neighbor city (that has not already been visited), and so on, until finally the last city is connected back to the start city. This rule for generating a route does not generally give the shortest path, since the final trip from the last city back to the start city can often be quite costly. An example is given in Figure 3.1.

Another example of a heuristic for a tour construction procedure utilizes the minimum spanning tree of the graph defined by the cost matrix C. A spanning tree of a graph of N cities is a tree with $N-1$ edges connecting all the vertices. The minimum spanning tree is the spanning tree of minimum total length. Finding the minimum spanning tree can be done quite efficiently by a number of methods [1]. A depth first traversal of the minimum spanning tree gives a good tour, however, this tour may include some cities more than once. Nevertheless, given the depth first traversal, if a city has been visited, then that city can be skipped and the tour can proceed to the next city (not already visited) as indicated by the depth first traversal. A tour generated by this minimum spanning tree procedure is shown in Figure 3.2. In addition to offering a good initial route, the minimum spanning tree also provides a lower and upper bound on the optimal tour length. The optimal tour length for the Traveling Salesman Problem must be strictly greater than the length of the minimum spanning tree, and no more than twice the length of the minimum spanning tree [32].

An example of a tour improvement procedure is one that uses an edge exchange heuristic. Given a tour for the Traveling Salesman Problem, the tour can be modified by removing k edges from the tour, then replacing them with k edges to form a new tour. Such a modification is called a *k–change*. Figure 3.3 illustrates a *2–change* modification of a tour. If it is not possible to improve a tour via a *k–change*, then the tour is termed *k–optimal* or *k–opt*. The *2–opt* and *3–opt* heuristics were first introduced by Lin [35]. A procedure may start with an initial tour chosen randomly from the set of all possible tours, or an initial tour generated using a tour construction procedure. Then, the *k–change* heuristic is repeatedly invoked until the tour is *k–optimal*. Usually, only a local optimum solution is reached. The *2–opt* exchange procedure will generally terminate at an inferior local optimum in comparison to the *3–opt* exchange procedure. Although *k–opt* exchange procedures ($k>3$) will generally terminate with even better local optimum, their complexity makes them less tractable. A simple composite procedure performs a *2–opt* and thereupon a *3–opt* procedure. This gives good results and runs relatively fast computationally. More elaborate procedures [55] use sophisticated methods for determining what level of *k–change* to use at any given stage of a search.

Simulated annealing [29,40] is another tour improvement procedure that can be applied to the Traveling Salesman Problem [28,29,52]. A crystal system provides a good example for explaining the simulated annealing concept. When one desires to form a crystal structure, one starts by heating the system to some high temperature where the system is in a liquid state. At this stage the system is in a high energy state. By slowly cooling the system, the system settles into a solid crystal structure, which is a minimum energy state. This process does not, however, guarantee that the resulting crystal will be a "perfect crystal" — a global minimum energy state. If the temperature scheduling is too fast, impurities may form in the crystal structure. Unfortunately, when impurities form at high temperatures, they cannot be removed at any lower temperature. Nonetheless, if a suitably slow temperature schedule is used, then a crystal structure with relatively few impurities should result. The simulated annealing optimization procedure for general combinatorial optimization problems attempts to simulate the annealing of a physical structure like this crystal system. In this technique, the states of the optimization problem are generalized to states of a physical system, the objective function of the optimization problem is generalized to the energy of the physical system, and a control parameter of the optimization problem is generalized to the temperature of the physical system. Near optimal solutions are sought by allowing the system to anneal from a high temperature to a low temperature. Appendix A explains in greater detail the simulated annealing procedure and its application to the Traveling Salesman Problem. Figure 3.4 illustrates the results of applying simulated annealing to a 22-city Traveling Salesman Problem.

Navigation Path Planning

A pilot often avoids mountains and creates masking by flying through the valleys of mountainous terrain. In the course of navigating through natural terrain, mountains can be thought of as obstacles. The task of planning an aircraft route through mountainous terrain is similar to the task of planning a path for a robot manipulator arm through a work environment of obstacles. Also, aircraft path planning is similar to mobile robot and autonomous land vehicle path planning. Because of the similarity, there may be attributes of solution techniques in the mobile robot and robot manipulator arm problems which are useful in solving the path planning problem for aircraft.

Several methods for path planning in robotics are now reviewed. Methods for planning amongst polygon obstacles and methods for terrain navigation planning are considered. The additional complications of inexact or unclear maps are not addressed.

The configuration space approach for path planning proposed by Lozano-Perez [37] deals directly with the free space rather than the space occupied by obstacles. The process shrinks the manipulated object down to a single source point and expands the obstacle regions according to the object shape. The vertices of the expanded obstacle regions constitute a search graph. All vertices that can be joined without intersecting an expanded obstacle are connected to form a visibility graph, which is searched for a path between start and finish locations. Figure 3.5 shows the search graph generated by the configuration space approach for a square object amongst polygon obstacles. The dashed search arcs correspond to the expanded obstacle shapes. The location of the obstacle at the start **S** and finish **F** are indicated with the object shown at the start location. The path generated is typically close to the obstacles, which doesn't allow for large uncertainties in the obstacle representation or the object position. If obstacle shapes are expanded to compensate for these errors, then it is possible that some paths between obstacles may be excluded needlessly.

In an attempt to create paths that stay in the middle of corridors between obstacles rather than along the edges of obstacles, Brooks [6] developed a generalized cone method. All pairs of obstacle edges are examined, and those that have free space between them compose the sides of a generalized cone. Figure 3.6 shows a generalized cone between two polygon obstacles with the axis of the cone shown as a dashed line. The union of generalized cones constitutes free space. Intersecting cone axes determine nodes for a connectivity graph, and arcs between nodes are paths along the generalized cone axes. The connectivity graph is used to search for paths between start and finish locations. Figure 3.7 shows the search graph created from

generalized cone axes for an object (not shown) amongst polygon obstacles. Dashed lines indicate the search graph. In this method, the clearance of the object must be checked at the bottlenecks of the generalized cones while searching the graph.

In a mixed representation, Kuan, et al. [30] use generalized cones and convex polygons to represent free space. First, all the polygon obstacles must be decomposed into convex obstacles to create a uniform obstacle representation. Next, a topological neighborhood graph is constructed to identify which shapes constitute distinct obstacles. The dual of this neighborhood graph is a connectivity graph depicting free space. The arcs of the connectivity graph represent channel regions between neighboring obstacles, and the nodes represent passage regions where those channels meet. The channels may be thought of as streets and the passage regions as street intersections. At each arc of the connectivity graph, a generalized cone is used to represent the channel, and at each node of the connectivity graph, convex polygons are used to represent the passage region. The intersection of generalized cone axes and convex polygon segments are used as nodes of a search graph representing free space paths. Figure 3.8 illustrates the partitioning of the free space into channels and passage regions. Figure 3.9 shows a search graph generated with the mixed representation of free space. The start node **S** and finish node **F** are connected to the nodes in their respective channel and passage regions in order to complete the search graph. The graph is then used to search for paths between start and finish points.

Potential field approaches are another technique which carry over from manipulator arm problems [2,27]. In this method, the obstacles are represented as groupings of charged particles that repel the manipulated object while the finish location is modeled with a charge that would attract the object. The manipulator arm moves toward the finish location following the gradient of the potential field — additional heuristics are often needed to maintain the progress towards the finish and to avoid box canyons which typically occur.

A simple approach to establishing a search graph is to use a regular grid method ([26,43] applies it to DARPA's Autonomous Land Vehicle). Free space is represented using an evenly spaced grid of points. Each grid point connects with four or eight neighbors to form a graph. Two simple graphs generated from the same grid points are shown in Figure 3.10 and Figure 3.11. Grid points are superimposed on the polygon obstacle map, and those that fall on or within obstacle regions are removed. Graph arcs are then connected in order to form a search graph. An alternative to removing grid points would be to assign infinite costs to the arcs that connect grid points that fall on or within obstacle regions. The start node **S** and finish node **F** can be connected to the closest grid point, as shown in Figure 3.12. Alternatively, the grid can be constructed so that the start and finish locations are grid points — thereby

influencing the size and orientation of the grid with their relative locations, as illustrated by Figure 3.13.

The path relaxation method of Thorpe [59] combines a regular grid graph and aspects of the potential field method in a two-step process. First, one seeks a solution using a regular grid search. Then in a relaxation step, the nodes of the regular grid solution are perturbed into nearby positions. Each node varies in a motion restricted perpendicular to the line between preceding and following nodes. In a sense, the nodes are positioned by a potential field established as a cost function for the problem.

Methods which rely on search graphs developed from a quadtree representation [23,25,44] partition free space into square regions. Free space is recursively decomposed using quadtrees. Initially, the largest possible squares are fitted to partition the obstacle region. These partitions will either be full, partially full, or free of obstacles. The partially full partitions are further subdivided into four smaller square partitions, and each resulting square is appropriately identified as full, partially full, or free of obstacles. This process of subdividing partially full squares repeats until the square size becomes sufficiently small to well represent the obstacle boundaries. Figure 3.14 shows a free space partitioning with quadtrees. A search graph for path planning can be constructed by connecting nodes located at the center points of adjacent squares. The start node **S** and finish node **F** can be connected to the node in the center of their respective squares. An example of a search graph established using the quadtree representation for free space is shown in Figure 3.15. This method has the salient advantage of representing large open spaces with a coarse grid and more congested (with obstacles) regions with finer grids. The quadtree representation is easily formed from a binary array or raster representation of obstacle data. Furthermore, free space data processed into a hierarchical quadtree representation saves substantial computational time during a search.

Finally, Voronoi diagrams also assist various path planning problems. O'Dunlaing and Yap [48] plan the motion of a disc amongst polygon obstacles, O'Dunlaing, Sharir, and Yap [46,47] for the motion of a ladder, and Canny [7,8] for the motion of a polygon object. A review of Voronoi diagrams and their use in motion planning is given in [56]. In these methods, the Voronoi diagram of the obstacle polygon line segments are used to create a diagram of straight and parabolic arcs connected to form a search diagram for paths. Except in the vicinity of the initial and final locations of the object, the search is constrained to the Voronoi diagram. Thus, a search for a path in a two dimensional space of obstacles is reduced to a search on a one dimensional manifold, the Voronoi diagram. Figure 3.16 shows the Voronoi diagram of a set of polygons in an application of moving a triangular object

amongst a set of polygon obstacles. In three dimensions, Canny [7,8] applies the Voronoi diagram for polyhedra to solve the Piano-Movers Problem (moving a polyhedral object amongst polyhedral obstacles). The Voronoi diagram of three dimensional space is a two dimensional manifold made up of planar and quadratic surfaces. This is a set of points equidistant from two or more obstacle faces, edges, or vertices. Paths can be found by searching the Voronoi diagram, however, Canny uses a simplified Voronoi diagram to find paths. Simplified Voronoi diagrams, introduced by Canny and Donald [7,10], are based on a measure of distance which is not a true metric, resulting in a lower algebraic complexity for the search space compared to the true Voronoi diagram.

In this thesis, the Voronoi diagram of a set of points — rather than a set of line segments — provides a search graph for path planning. The polygon vertex points of the mountain boundaries, or obstacles, create a Voronoi diagram from which one constructs a search graph for path planning. The Voronoi diagram is modified by eliminating certain edges so that no edge in the final search graph will cross over an obstacle boundary. Consequently, the lengths of different paths in the graph could be meaningfully compared. Although the paths at this planning level could be traversed, the purpose of the comparison is to decide on a reasonably good path to consider for further investigation in the path planning hierarchy. Whereas the paths generated by the Voronoi diagram of a set of line segments may be closely followed in the path planning applications mentioned above, the paths generated by the modified Voronoi diagram search graphs in this thesis are *not* considered as the paths to be followed. The hierarchical role of path planning using the modified Voronoi diagram presented in this thesis will become apparent in further chapters.

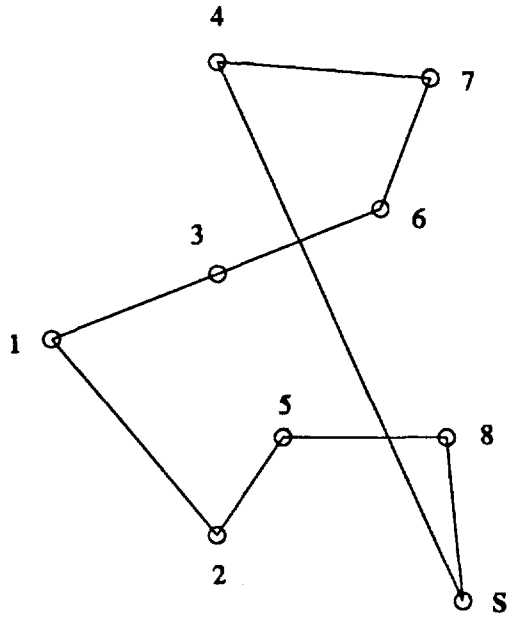


Figure 3.1. In this Traveling Salesman Problem, the nearest neighbor heuristic generates a tour with a final edge from node 4 to the start node S which is quite costly. The final edge constitutes 30% of the total tour cost. The final tour is (S 8 5 2 1 3 6 7 4 S).

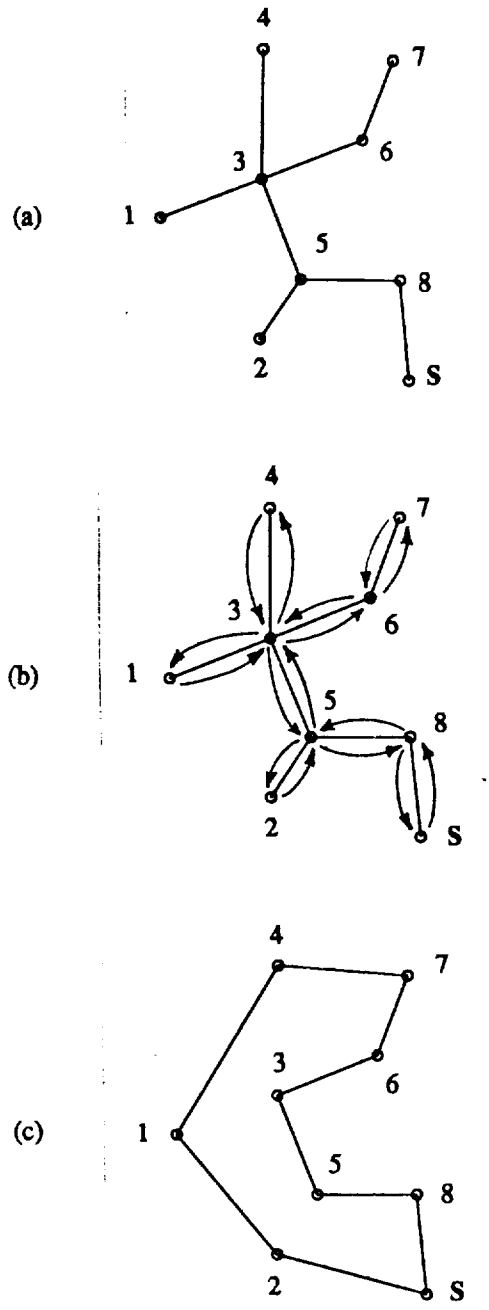


Figure 3.2. The minimum spanning tree can be used to generate a tour for the Traveling Salesman Problem. (a) Construct the minimum spanning tree, (b) perform a depth first traversal of the minimum spanning tree, and (c) modify the depth first traversal by skipping over nodes that are revisited. The final tour is (S 8 5 3 6 7 4 1 2 S).

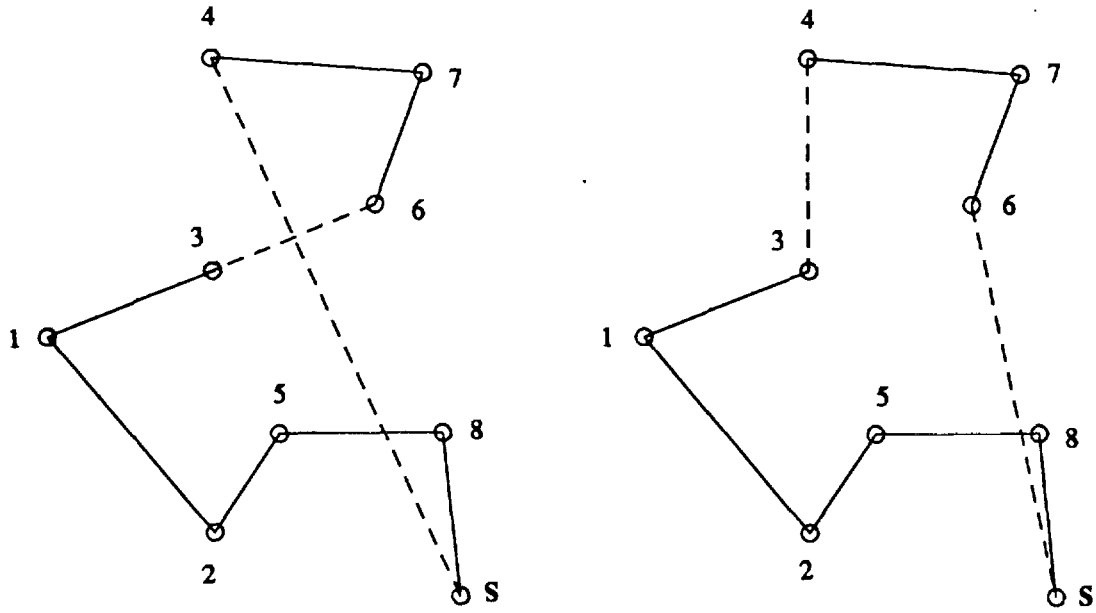


Figure 3.3. The tour (S 8 5 2 1 3 6 7 4 S) is modified using a 2–*change* to form the tour (S 8 5 2 1 3 4 7 6 S). This 2–*change* decreases the total tour cost by 7.3%. The modified edges are shown in dashed lines.

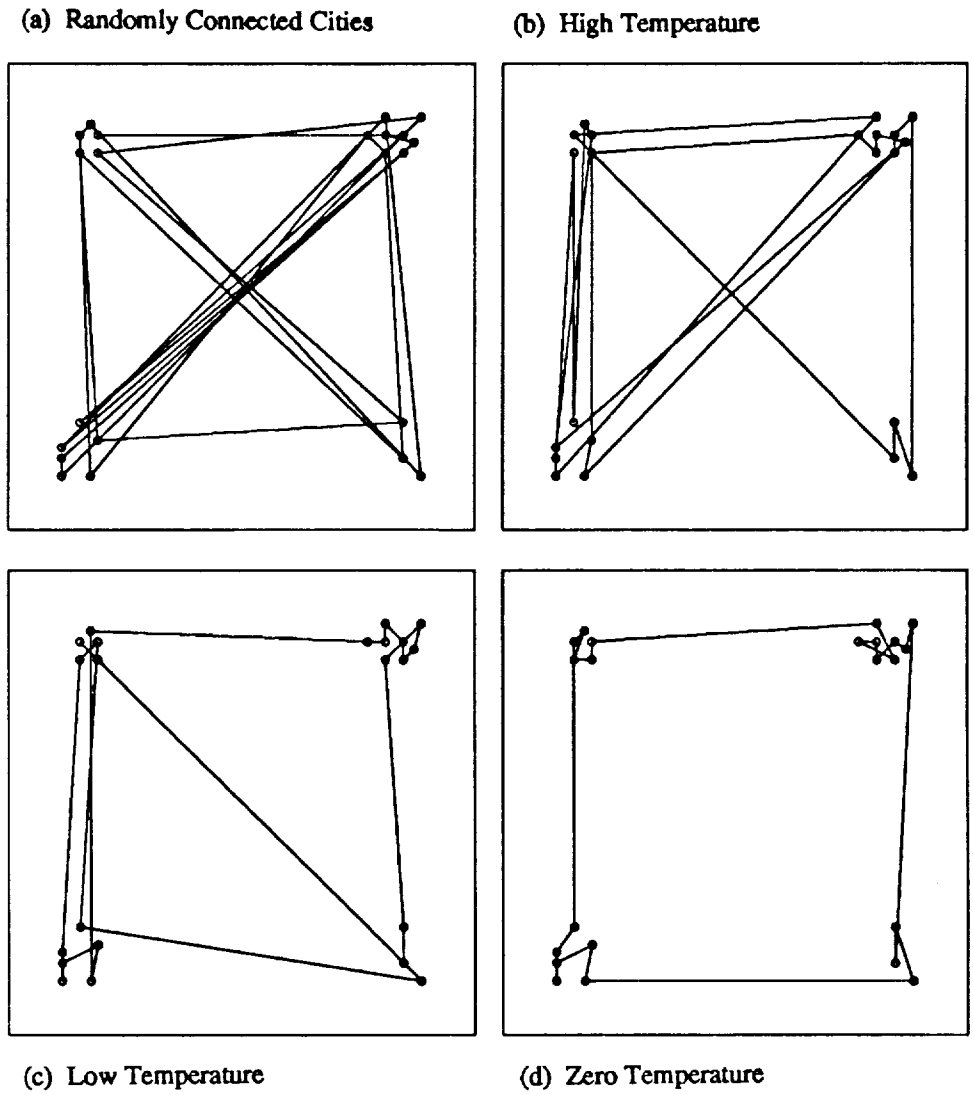


Figure 3.4. Simulated annealing results for a 22-city Traveling Salesman Problem. Initially, the cities are randomly connected (a). As the temperature parameter is lowered from higher temperatures (b) to lower temperatures (c), costly tour edges are removed. The final tour (d) shows a path that visits each cluster once, visiting all the cities within the cluster and then proceeding to visit another cluster.

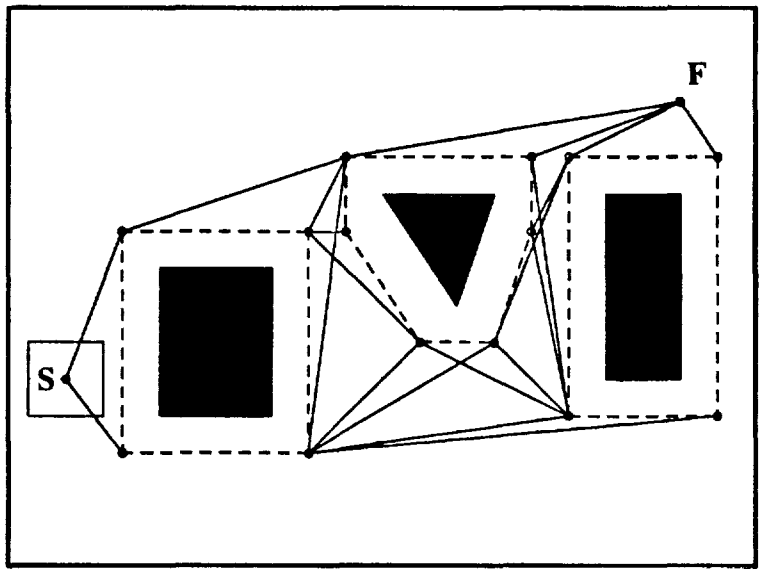


Figure 3.5. A search graph generated with the configuration space approach for path planning.

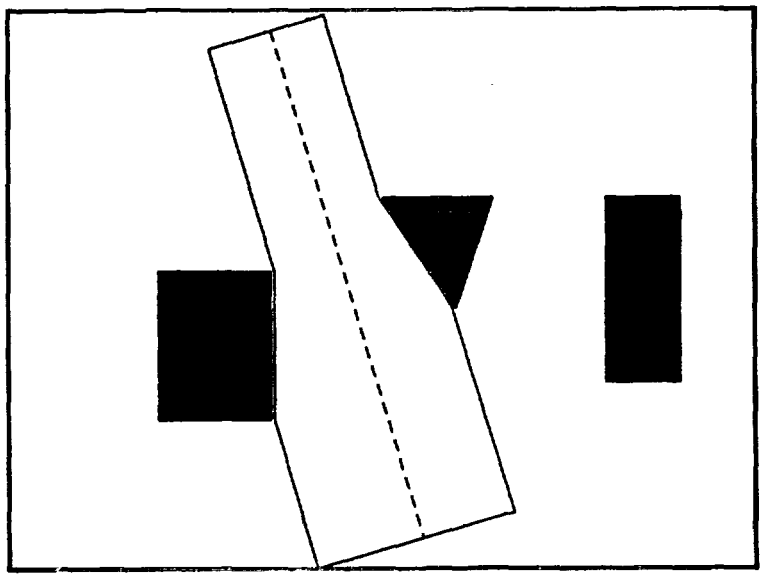


Figure 3.6. A generalized cone.

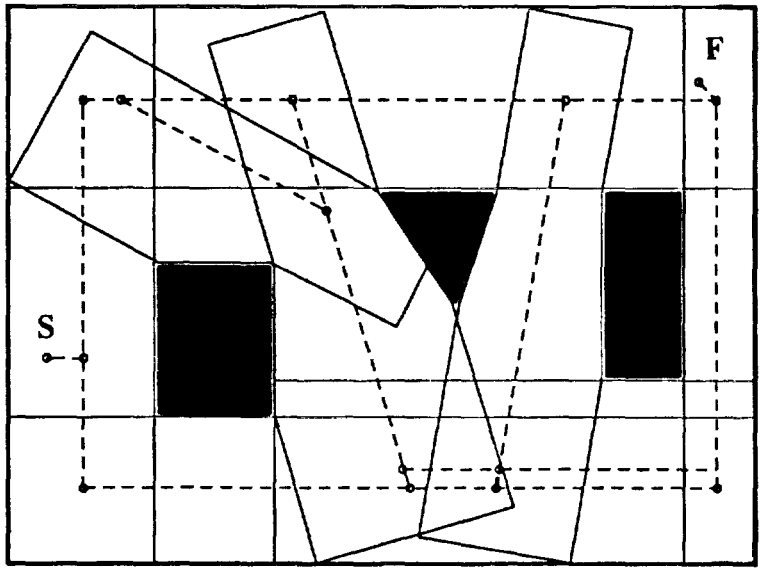


Figure 3.7. A search graph generated with generalized cones.

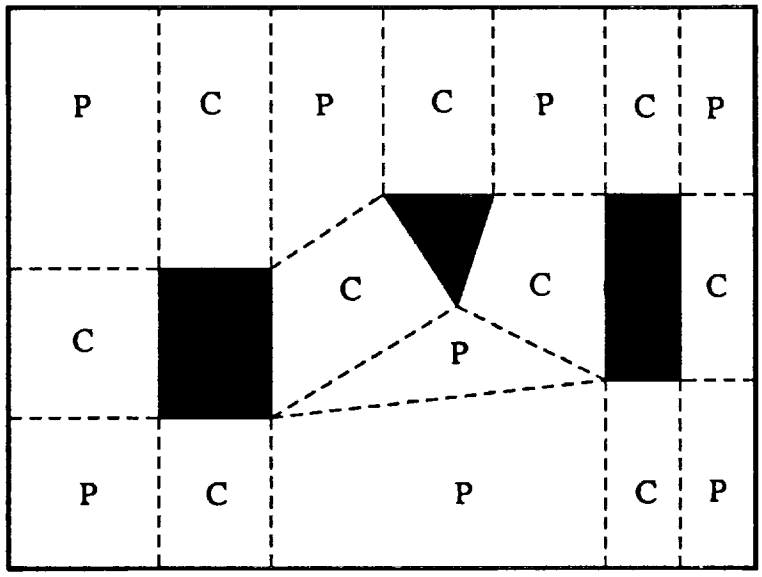


Figure 3.8. Free space between polygon obstacles partitioned into channel regions, labeled with a C, and passage regions, labeled with a P.

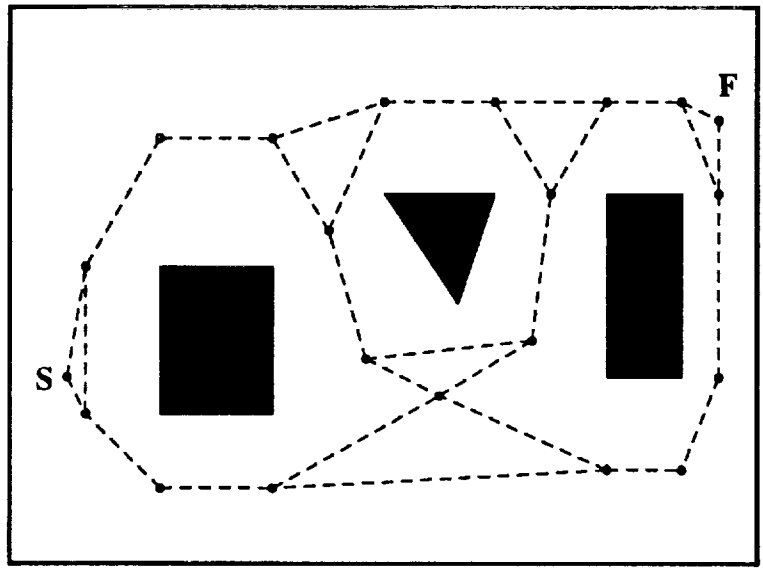


Figure 3.9. A search graph generated with a mixed representation of free space.

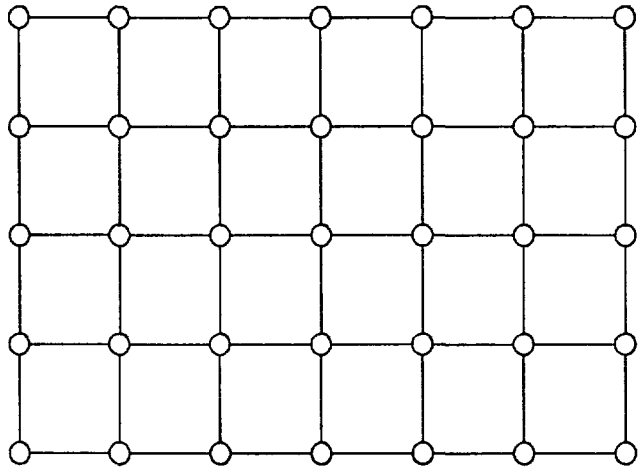


Figure 3.10. A graph connecting four neighboring grid points.

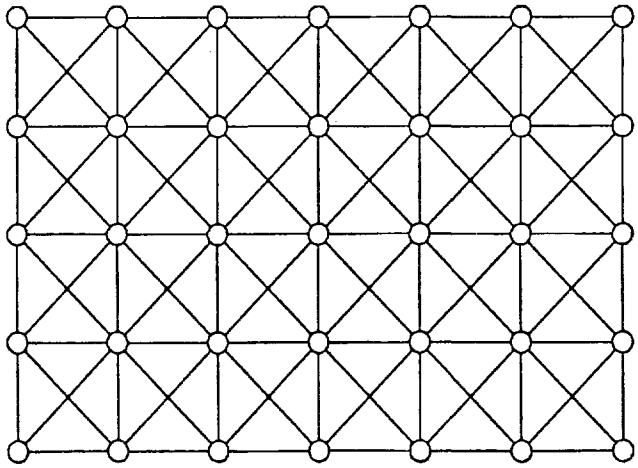


Figure 3.11. A graph connecting eight neighboring grid points.

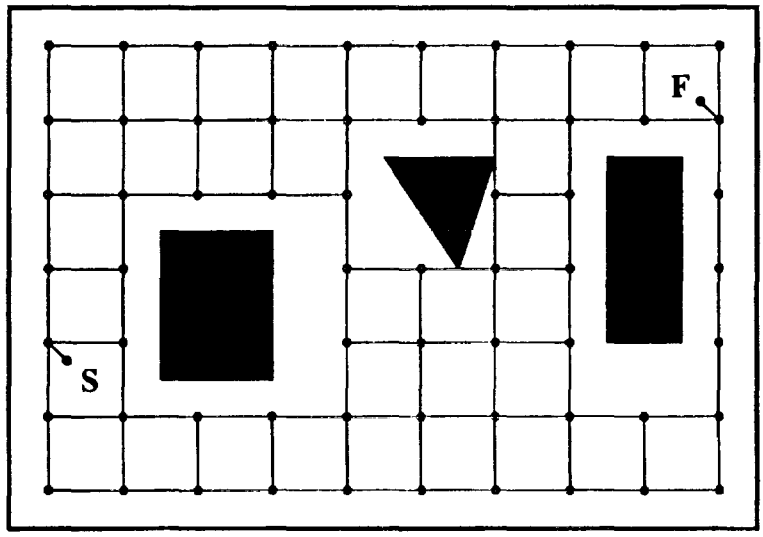


Figure 3.12. A search graph generated from a simple grid connecting four neighboring grid points. The start and finish locations are connected to the closest grid points.

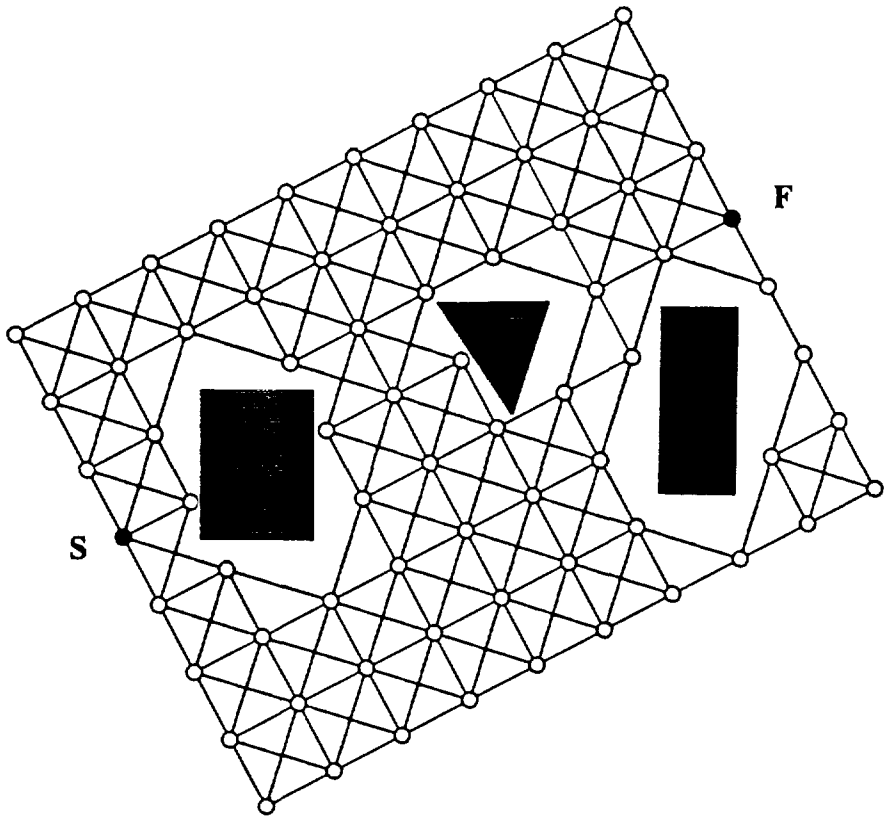


Figure 3.13. A search graph generated from a simple grid connecting eight neighboring grid points. The start and finish locations are forced to be grid points.

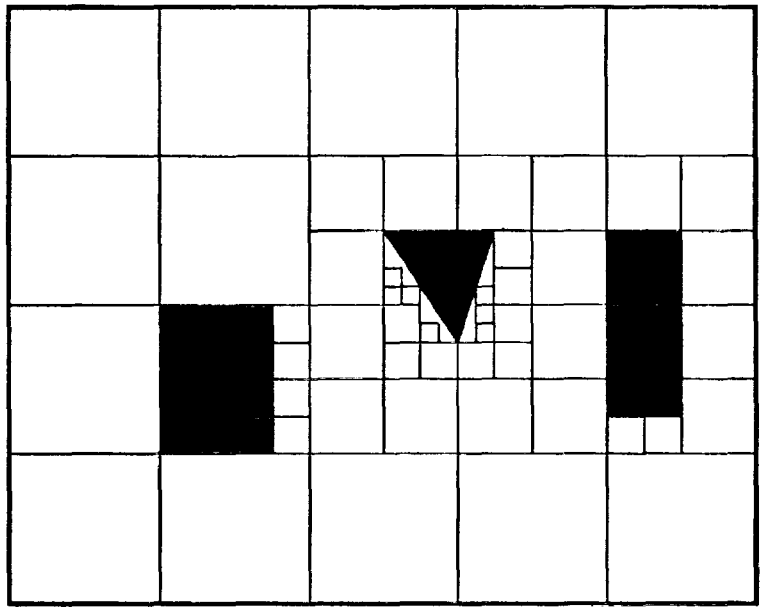


Figure 3.14. Free space between polygon obstacles partitioned using quadtrees.

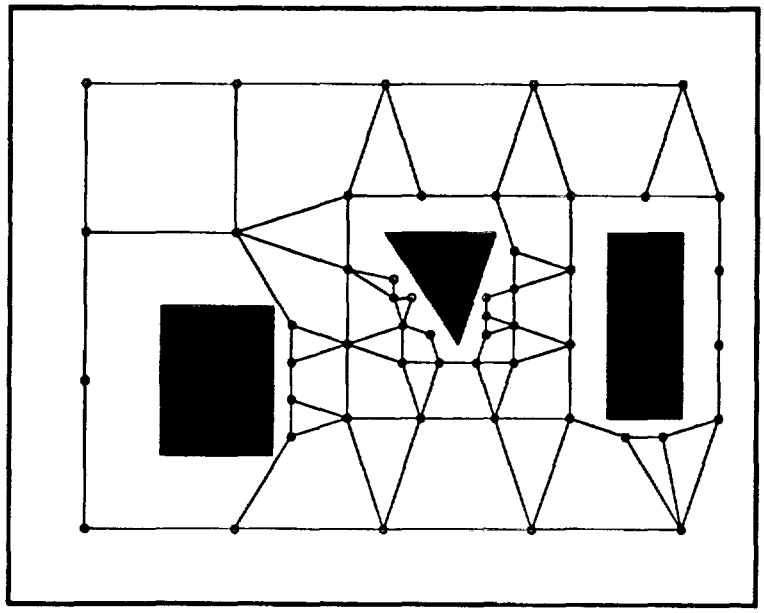


Figure 3.15. A search graph generated using quadtrees.

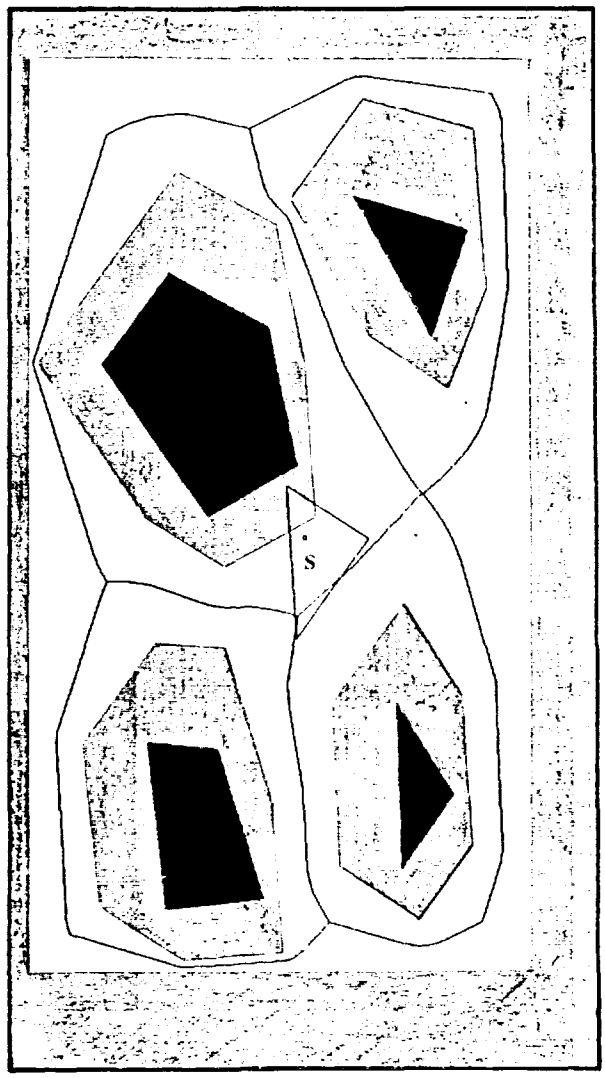


Figure 3.16. The Voronoi diagram is used to plan the motion of a triangular object amongst polygon obstacles. The triangular object is initially at point **S**. Shaded polygon regions around obstacles represent the enlarged obstacles — planning is then simplified to moving a point obstacle amongst these enlarged obstacles. The Voronoi diagram of the enlarged obstacles establishes the search graph for motion planning. This figure is taken from [10] with permission from the authors [9].

CHAPTER 4

TREE SEARCH SOLUTIONS FOR VARIATIONS OF THE TRAVELING SALESMAN PROBLEM

This chapter investigates tree search techniques for the Traveling Salesman Problem. Initially, the terminology and complexity of the search space is discussed. Then, an 11-city Traveling Salesman Problem is introduced, for which all the search problems in this chapter are based. Uninformed search techniques are applied, then heuristic search techniques. Next, the basic Traveling Salesman Problem is varied so that local and global constraints are added. These variations make the problem applicable to more complex mission planning problems.

The Search State Space

In general, a sequence of events constitutes a mission plan. For the Traveling Salesman Problem, an *event* is to visit a city, e.g., visit city A or visit city B. A sequence of events is called a *state,* representing a portion of a tour. A state is suitably represented as a list of cities, e.g., the state $(A\ B\ C)$ represents the portion of a tour for a visit from city A followed by a visit to city B followed by a visit to city C. The *state space* is the set of all states. In order to create one state from another, a *state operator* is invoked. A state operator modifies a state by adding another city, not already in the current state, to the end of a tour. It also maintains the cost of a state. For example, the state operator O_C modifies the state $(A\ B)$ with cost $g=c_{AB}$, adding the visit to city C, to create the new state $(A\ B\ C)$ with cost $g=c_{AB}+c_{BC}$. Since the salesman travels from the final city back to the start city, the state operator modifies the state by adding the start city to the state only after all the cities are included in the tour.

When solving the Traveling Salesman Problem using tree search techniques, the entire state space is represented by a state space tree. The *state space tree* represents all the states as *nodes* and connects all the states with *arcs*. Each arc represents the

appropriate state operator that must be invoked to create the state. Figure 4.1 presents a state space tree for a 4-city Traveling Salesman Problem. The state (S) is the *root node* or the *start node* for the tree. If a state (node) is modified to create a new state (node) with the state operator, then the new state (node) is called the *successor* of the previous state (node). Creating all the successor nodes of a particular node in a tree is called *expanding* the node. A state (node) that has no successor is called a *leaf node*.

The state space tree exhibits some notable qualities. First, the exponential growth of the tree is apparent. Let N denote the number of cities in the problem. At depth 0 the start city composes the root node. At this depth the state operator creates $N-1$ new states at depth 1. At depth 1 the state operator creates $N-2$ new states at each node, and at depth d ($d<N-1$) the state operator creates $N-d-1$ new states at each node. The last invocation of the state operator adds the start city to the end of the tour, represented by a single successor for all states at depth $N-1$. Notice that there are $(N-1)!$ leaf nodes of the state space tree. These nodes represent possible solutions to the problem, since these states satisfy the *global constraint* that all the cities are visited and the tour finishes at the start city. Variations of this problem (investigated later) will allow for *local constraints* that specify that a particular city must be visited prior to another city. A particular state that satisfies the local and global constraints of the criteria is called the *solution state* or the *goal state*. Of course more than one solution state may exist.

The size of the state space is characterized by the number of states or the number of arcs. The state space tree has 1 node at depth 0, $\prod_{k=1}^{d}(N-k)$ nodes at depth d ($0<d<N$), and $(N-1)!$ nodes at depth N. For every node at depth d ($d\neq 0$), there is one arc connecting it to some node at depth $d-1$ (the root node is the exception). Thus, there is only one less arc than the number of nodes, and either the number of nodes or the number of arcs can characterize the state space. The total number of states (or nodes) in the search tree is $1+(N-1)!+\sum_{d=1}^{N-1}\prod_{k=1}^{d}(N-k)$ and the total number of arcs in the search tree is $(N-1)!+\sum_{d=1}^{N-1}\prod_{k=1}^{d}(N-k)$. Using these equations, the size of the search tree can be computed before the search is started. If the size of the tree is determined to be too large for computer storage, then the search may have to rely on efficient pruning of the tree to be able to arrive at a solution to the problem.

In tree searches, the state operator creates (or builds) the state space tree. If the state operator can create a search tree to find a solution without creating the entire search tree, then the sections of the tree that are *not* created are considered *pruned*

sections of the tree. If a node at depth d is pruned from the search tree, then none of its successor nodes at a depth greater than depth d are created by the state operator, and a subtree of size $N-d$ is eliminated from the search. Some of the search techniques investigated in this chapter will show how pruning can be performed while still computing optimal solutions.

Computing the number of nodes or arcs used in the search tree establishes a means for comparing the search efforts of different search techniques. In this chapter, the number of nodes created in the search tree will be used to compare search efforts of different search techniques.

An 11-City Traveling Salesman Problem

Consider the problem where a salesman must visit the cities Atlanta (ATL), Boston (BOS), Chicago (CHI), Dallas (DFW), Denver (DEN), Detroit (DTT), Los Angeles (LAX), Minneapolis (MSP), New Orleans (MSY), Pheonix (PHX), and Seattle (SEA). The salesman must start at Detroit and visit all the cities once before returning home, minimizing the cost to travel to all the cities. To complete this problem statement, a table of intercity costs is given which specifies the cost to travel from any city to any other city. Table 4.1 gives the intercity costs for the 11-city Traveling Salesman Problem example (from [13]). These intercity costs establish the cost matrix C as discussed in Chapter 3.

The search tree for the 11-city Traveling Salesman Problem is large, yet the tree can be searched by several techniques without computational difficulties. The 11-city search tree has a total of 13,492,901 nodes with 3,628,800 leaf nodes (or solution nodes). This search tree size is not unmanageable for computer storage, and even an exhaustive search can be performed, however, for the purpose of comparing many search techniques, this represents a reasonable size problem.

Uninformed Search Techniques for the Traveling Salesman Problem

Exhaustive search is a simple example of an uninformed search technique. An exhaustive search considers all the leaf nodes of the state space tree and compares their costs to arrive at the minimal cost tour. This search can be established by an exhaustive breadth first search [51], as illustrated in Figure 4.2, or a exhaustive depth first search [51], as illustrated in Figure 4.3. The breadth first search establishes the search tree by progressively creating the tree in layers of equal depth. The depth first

search establishes the search tree by creating all the nodes of the leftmost branch first, then creating the next possible solution the problem (leaf node), etc. While these techniques provide optimal results and are easy to program, these techniques are inefficient in computational cost. For large problems, the entire search tree cannot be stored in computer memory, and thus, exhaustive search is ineffective.

For the 11-city problem, an exhaustive search creates all 3,628,800 possible solutions to the problem, incurring the cost of creating 13,492,901 nodes for the search tree. The optimal solution is found as (DTT CHI MSP DEN SEA LAX PHX DFW MSY ATL BOS DTT) with a tour cost of 2220. The results of applying exhaustive search to the 11-city problem are presented in Table 4.2.

Another uninformed search technique can be performed with a depth first search with pruning. Recall, a depth first search establishes the tree by creating all the nodes of the leftmost branch of the state space tree first. After creating the first possible solution, a current best cost is known. If another solution is created with a better cost, then it becomes the current best solution. This current best cost is maintained for the purpose of pruning. When the search is investigating a branch of the state space tree and at that node the cost exceeds the current best cost, then the rest of the tree beyond that node is pruned. After all the possible solutions are considered, the optimal solution is the current best solution. The use of pruning makes this method more efficient than simple enumeration.

For the 11-city problem, a depth first search with pruning creates 1,455,016 nodes of the state space tree. This represents only 10.8% of the nodes of a full state space tree. The optimal solution is found as (DTT CHI MSP DEN SEA LAX PHX DFW MSY ATL BOS DTT) with a tour cost of 2220. The results of applying a depth first search to the 11-city problem are presented in Table 4.3. Notice that some nodes are pruned at depth 5. Although a reduced search effort is realized through the pruning of the depth first search, no heuristics are used. The following section considers informed search techniques that utilize heuristic information about the state space to further reduce search effort.

Informed Search Techniques for the Traveling Salesman Problem

When a human plans a tour around the United States, he often uses heuristics to aid him in achieving a plan. For example, when flying from the midwest to the west, it is cheaper to use a flight which stops over in Denver or St. Louis, rather than to fly direct. If time is also a factor, then a stop at Denver may be avoided if it is snowing there. Humans utilize heuristics to avoid the complications of considering many

possible alternatives. When using tree search techniques, the reasoning procedure is not necessarily the same as that of a human, but both the human and the computer can benefit by using heuristics to reduce the amount of search effort.

Informed search techniques use heuristics to prune the state space tree. With certain heuristics, pruning will not effect the optimality of the search. If a search algorithm terminates finding an optimal path from the start node to a goal node whenever a path from the start node to a goal node exists, then the search algorithm is termed *admissible*. In the tree searches that follow, only heuristics that lead to admissible searches are applied.

Informed search techniques utilize an evaluation function for the purpose of pruning. At each node, it is desired to have an estimate for the cost of any of the solution states which lie below the current node in the tree. The evaluation function $f(n)$, for a node n, takes the form:

$$f(n) = g(n) + h(n).$$

The term $g(n)$ is the cost of the state at node n. The term $h(n)$ is the heuristic function which estimates the optimal cost from node n to a solution state. The evaluation function estimates at node n the cost of a solution state. If this estimate is conservative, i.e., underestimates the actual cost, then it can be used for pruning while maintaining an admissible search. That is, if $f(n)$ exceeds the current best cost, then the states below node n can be pruned from the state space tree without effecting the optimality of the search. For example, the depth first search with pruning is an admissible search because the hueristic $h(n)=0$ is an underestimate of the cost of going from node n to a solution state.

The minimum element c_{min} of the cost matrix C provides a simple heuristic for an admissible search. Since all flights cost at least c_{min}, then one can estimate for any state at depth d of the state space tree that it must cost at least an additional $(N-d)c_{min}$ to reach a solution state. Thus, $h(n)=(N-d)c_{min}$ is the heuristic function. This is certainly an underestimate of the actual cost of completing a tour; an admissible search is guaranteed. Notice that the c_{min} heuristic is a less conservative estimate of the cost from node n to a solution state when compared to no hueristic $h(n)=0$.

For the 11-city problem, a depth first search with the heuristic function $h(n)=(N-d)c_{min}$ creates 356,141 nodes of the state space tree. This represents 2.6% of the nodes of a full state space tree. Table 4.4 shows the results of this search technique, which indicates that the optimal solution is found. Notice that the pruning of this search occurs first at depth 4, which is one depth level before the same search with no heuristic function.

A more powerful heuristic than the $c_{\min}$ heuristic can be formed using the minimum elements of all the rows and columns of the cost matrix C. Using $c_{\min}$ to estimate the cost of all of the remaining arcs in a tree is conservative, since in actuality, if the flight costing $c_{\min}$ were used, it could only be used once. Next, consider that since each city is visited only once, then one can utilize the minimum cost of *each column* or *each row* of the cost matrix for a heuristic. The reasoning is as follows. If one city remains to be added to the tour, then it must cost at least $c_{\min}$. If two cities remain, then $c_{\min}$ should be used to estimate one flight, but the other flight cost must come from some other column (row) of the cost table. Thus, the minimum cost element not in the column (row) of $c_{\min}$ should be used to estimate this other cost. In general, estimates could be made with the minimum elements of each column (row) of the cost matrix. For these N minimums, the lowest cost should be used to estimate the flight represented at depth N of the search tree, the second lowest cost for the arc at depth $N-1$, the third lowest cost for the arc at depth $N-2$, and so on. For example, consider the heuristic function $h_1(n)$ that involves a vector of column minimums. Let $\bar{c}_{\min_{COL}}$ be the vector of column minimums, sorted in increasing order. For the 11-city problem, $\bar{c}_{\min_{COL}}$ = (100 120 130 140 150 150 170 200 200 270). Let $\bar{c}_{\min_{COL}}(i)$ be the ith element of this vector. The heuristic function for a node n at depth $d(n)$ is $h_1(n) = \sum_{i=1}^{N-d(n)} \bar{c}_{\min_{COL}}(i)$. Next, one notes that this heuristic could be improved by using information imbedded in the rows of the cost matrix. The sorted vectors $\bar{c}_{\min_{COL}}$ established using the columns of C and $\bar{c}_{\min_{ROW}}$ established using the rows for C establish the two heuristic functions $h_1(n) = \sum_{i=1}^{N-d(n)} \bar{c}_{\min_{COL}}(i)$ and $h_2(n) = \sum_{i=1}^{N-d(n)} \bar{c}_{\min_{ROW}}(i)$. Let the $\bar{c}_{\min_{ROW/COL}}$ heuristic be defined as $h(n) = \max[h_1(n), h_2(n)]$. This provides a heuristic leading to an admissible search.

For the 11-city problem, a depth first search with the $\bar{c}_{\min_{ROW/COL}}$ heuristic creates 195,827 nodes of the state space tree. This represents 1.5% of the nodes of a full state space tree. Table 4.5 shows the results of this search technique, which indicates that the optimal solution is found. Notice that the pruning of this search occurs first at depth 3, which is one depth level before the depth first search with the $c_{\min}$ heuristic and two depth levels before the depth first search with no heuristic function. Notice also that out of the 3,628,800 possible solutions (leaf nodes) of the state space tree, only 368 leaf nodes were created.

The choice of a search strategy can also effect the pruning of a search tree. Up to this time, only the depth first search technique has been employed for the above

heuristics. While this helped identify that the $\bar{c}_{\min_{ROW/COL}}$ heuristic is a powerful heuristic, it does not necessarily indicate that the most effective search has been performed. With the depth first search, the heuristic function aids pruning of the search tree, but it does not help guide the search toward branches of the search tree which are most promising to contain the optimal solution. The order of node expansion in a depth first search (or a breadth first search) is determined by the structure of the search, as described previously and in Figure 4.2 and Figure 4.3. An alternative search strategy is one that uses the evaluation function f to determine which nodes should be expanded. The best first search technique [51] expands nodes in the search tree based on which node has the minimum evaluation function value. The best first search starts by expanding the root node to create all the states at depth 1. Next, the state with the best (minimum) evaluation function value is expanded. The search continues to expand nodes that have the best evaluation function value, no matter where these nodes appear in the partially developed search tree. Thus, the best first search creates the tree based on which node at any stage of the search has the most promise of being part of the optimal solution. In fact, when the best first search selects a leaf node for expansion, then that leaf node must be the optimal solution to the problem. This is easily seen because first, a leaf node is a solution to the problem, second, the evaluation function value for a leaf node is also the actual cost g of this solution, and third, if this leaf node has the lowest evaluation function value, then no other leaf node in the tree can have a lower evaluation function value (or actual cost g).

For the 11-city problem, a best first search with the $\bar{c}_{\min_{ROW/COL}}$ heuristic function creates 27,075 nodes of the state space tree. This represents 0.2% of the nodes of a full state space tree. Table 4.6 shows the results of this search technique, which indicates that the optimal solution is found. Notice that the pruning of this search occurs first at depth 2, which is before any of the previous search techniques employed. Also note that only one leaf node, the optimal solution, is created, although 3,628,800 leaf nodes exist.

A final comparison of the search techniques applied to the 11-city Traveling Salesman Problem shows the computational benefits of informed search techniques. All the searches are admissible, and thus they all arrive at the same optimal tour (DTT CHI MSP DEN SEA LAX PHX DFW MSY ATL BOS DTT) with a tour cost of 2220. However, the differences between the searches employed should be compared in terms of some cost metric; the notion of heuristic power is useful. *Heuristic power* is a measure of the amount of pruning induced by a heuristic function. This can be expressed as the percentage of nodes pruned by an informed search technique compared to the total number of nodes in a full state space tree. Figure 4.4 shows a

comparison in terms of heuristic power for the search techniques applied to the 11-city problem. Based on heuristic power, the most effective search technique employed was the best first search technique with the $\bar{c}_{\min_{ROW/COL}}$ heuristic function. The effect of implementing an informed search technique with a good heuristic for pruning is apparent; the optimal solution can be found with far fewer nodes searched.

Informed Search Techniques for Variations of the Traveling Salesman Problem

The basic Traveling Salesman Problem is now varied so that local and global constraints are added. Two variations are presented. The first problem does not require much search effort, but is presented to introduce local and global constraints. The second problem is much more difficult, and demonstrates how more complex mission planning problems may be solved.

Consider the 11-city Traveling Salesman Problem with a budget limit and a city order constraint. A planner is interested in *any* tour that is less than $3000 provided that the home city is Detroit (DTT) and Boston (BOS) is visited before Los Angeles (LAX). The $3000 budget limit constraint is an example of a global constraint, and the city order constraint is an example of a local constraint. To solve this problem, it is most applicable to use a depth first search rather than a best first search because only one solution node is sought out of many possible solutions — there are actually 781 tours less than $3000 that obey this city order constraint. The breadth first search should not even be considered since it creates *all* the nodes below depth N before creating a single solution node at depth N. The depth first search should allow for more solution nodes to be investigated sooner than the best first search.

The budget limit and city order constraints can both be used for pruning. Whenever a node has an evaluation function value that exceeds the budget limit, the rest of the tree beyond this node can be pruned. Additional pruning can be done based on the city order constraint. When the state operator adds LAX to a tour that does not contain BOS, then the rest of the tree beyond this node can be pruned. These cases of pruning can easily be seen to lead to an admissible search because the optimal tour must comply with the budget limit and the city order constraints.

For the 11-city problem with the $3000 budget limit constraint and the BOS before LAX city order constraint, a depth first search with the $\bar{c}_{\min_{ROW/COL}}$ heuristic function creates only 70 nodes of the state space tree. Table 4.7 shows the results of this search technique, which indicates that a solution is (DTT ATL BOS CHI DFW

DEN LAX PHX SEA MSP MSY DTT) with a tour cost of 2930. The optimal tour that meets the budget limit and city order constraints is (DTT CHI DEN DFW MSY ATL BOS LAX PHX SEA MSP DTT) with a tour cost of 2510. Essentially, this problem is not very difficult to solve. The budget limit allows for a solution to be found without a difficult search. The next problem presented will be more constrained and will require a search that is more complex than the ones performed so far.

Consider a relaxed 11-city Traveling Salesman Problem with a budget limit and city order constraints. A relaxed 11-city Traveling Salesman Problem allows for solutions that do not contain all 11 cities, provided all the local and global constraints are met. A planner is interested in a tour that contains as many of the 11 cities as possible, is less than $2000 in cost, has the home city of Detroit (DTT), visits Boston (BOS) before Los Angeles (LAX), and includes at least Boston (BOS), Chicago (CHI), Dallas (DFW), Denver (DEN), Detroit (DTT), and Los Angeles (LAX). The budget limit and requirement to include BOS, CHI, DFW, DEN, DTT, and LAX are examples of global constraints, and the city order constraint is an example of a local constraint.

The state space of this relaxed 11-city Traveling Salesman Problem is larger than the state space of the basic Traveling Salesman Problem. For the basic Traveling Salesman Problem, the state operator adds the start city to the end of a tour only after all 11 cities have been visited. In the relaxed 11-city problem, the state operator can add the start city to the end of a tour after any number of cities have been visited. Of course, after the start city is added to the end of a tour, the tour is considered complete, and further cities cannot be added. Because the state operator can complete a tour consisting of any number of cities, the state space tree has solution nodes at depth 2 though depth N. Figure 4.5 presents a state space tree for a relaxed 4-city Traveling Salesman Problem. The relaxed 11-city problem has 19,728,201 nodes and 9,864,100 solution nodes (leaf nodes), compared to the 13,492,901 nodes and 3,628,800 solution nodes (leaf nodes) of the basic Traveling Salesman Problem.

One does not know a priori if the budget limit will allow for a full 11-city tour. Indeed, from the previous search results, one notes that the $2000 budget limit is insufficient to allow for any full 11-city tour. However, this information is not known at the beginning of the search, so it cannot be anticipated that a full 11-city tour *cannot* be realized. The search technique must proceed to look for the best 11-city tour until it discovers that an 11-city tour is not possible. Also, a solution node investigated at depth d is the optimal solution only if it meets all the local and global constraints, and if no solution node at a depth greater than depth d meets these constraints. Naturally, it follows that the best first search is the most appropriate search

strategy. A depth first search and breadth first search are less appropriate because they do not direct the search toward the optimal solution.

Pruning for the relaxed $N-$city Traveling Salesman Problem must account for the possibility of tours with less than N cities. The $\bar{c}_{\min_{ROW/COL}}$ heuristic function estimates at any depth the cost of a full $N-$city tour. However, when all the nodes of a search tree have an evaluation function value that exceeds the budget limit, one can conclude that the full $N-$city solution is not possible. In this case, the heuristic function should be changed to estimate the cost of a $(N-1)-$city tour. This is easily implemented with the $\bar{c}_{\min_{ROW/COL}}$ heuristic. Define k to be the $k-$city tour searched for in the relaxed Traveling Salesman Problem (initially, $k=N$). The $\bar{c}_{\min_{ROW/COL}}$ heuristic, defined as $h(n)=\max[h_1(n),h_2(n)]$, can be modified to allow for $k-$city tours by letting $h_1(n)=\sum_{i=1}^{k-d(n)} \bar{c}_{\min_{COL}}(i)$ and $h_2(n)=\sum_{i=1}^{k-d(n)} \bar{c}_{\min_{ROW}}(i)$. Finally, additional pruning can be done by using the city order constraint as described in the previous search.

The relaxed $N-$city Traveling Salesman Problem has the salient feature that the best $k-$city tour $(2 \le k \le N)$ can easily be retained during the search for the optimal tour. This may be useful for large N. That is, if computation time or storage becomes excessive during a search, the current best solution to the problem can be immediately retrieved. This is not the case with the search of the basic Traveling Salesman Problem, since only the leaf nodes of the search tree represent tours that return back to the start city — if the search is stopped before a leaf node is created, then a complete tour cannot be immediately retrieved.

For the relaxed 11-city problem with the $2000 budget limit, a best first search with the $\bar{c}_{\min_{ROW/COL}}$ heuristic function and pruning based on the city order constraint creates 17,660 nodes of the state space tree. This represents only 0.09% of the nodes of a full state space tree. Table 4.8 shows the results of this search, which indicates that the optimal solution is (DTT BOS LAX PHX DEN DFW MSY ATL CHI MSP DTT) with a tour cost of 1910. Table 4.9 shows intermediate results. The first tour created that meets all the local and global constraints is (DTT CHI BOS LAX PHX DEN DFW DTT) with a cost of 1530, which is found after creating 6168 nodes of the search tree. This tour includes only 7 cities so the search proceeds to look for solutions which include more cities. Note that the search determines that an 11-city tour is not possible after creating 17,660 nodes. Then, a 10-city tour is sought, however, at this stage of the search the best 10-city tour has already been found. Thus, the optimal solution is found and the search is complete. This problem shows how local and global constraints can be added to the basic Traveling Salesman Problem. The resulting problem is representative of a complex mission planning problem.

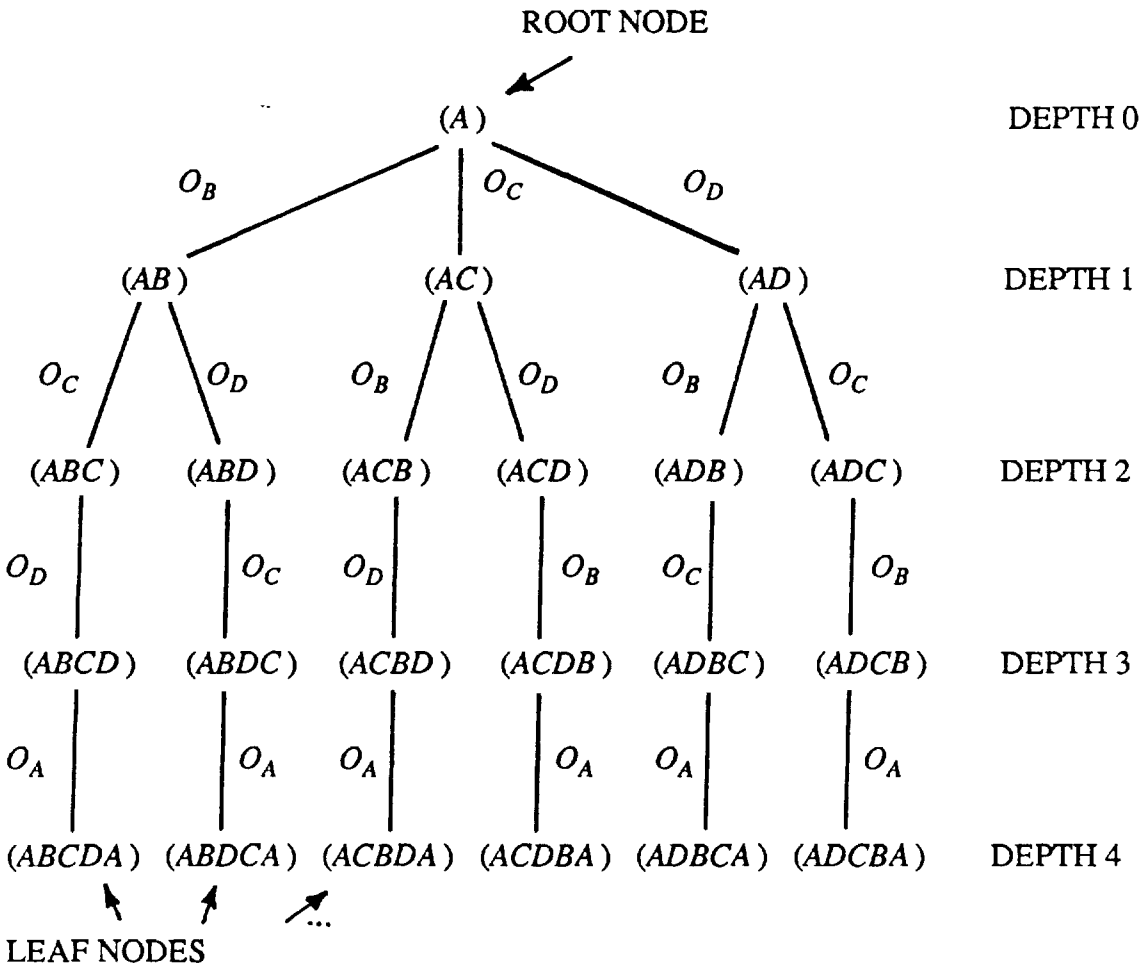


Figure 4.1. A state space tree for a 4-city Traveling Salesman Problem. City A, city B, city C, and city D compose a tour, with city A as the start city. States are shown in parentheses, and state operator decisions are shown on the arcs, e.g., O_B denotes the decision to add city B as the next city on the tour. The root node, leaf nodes, and depths are also labeled.

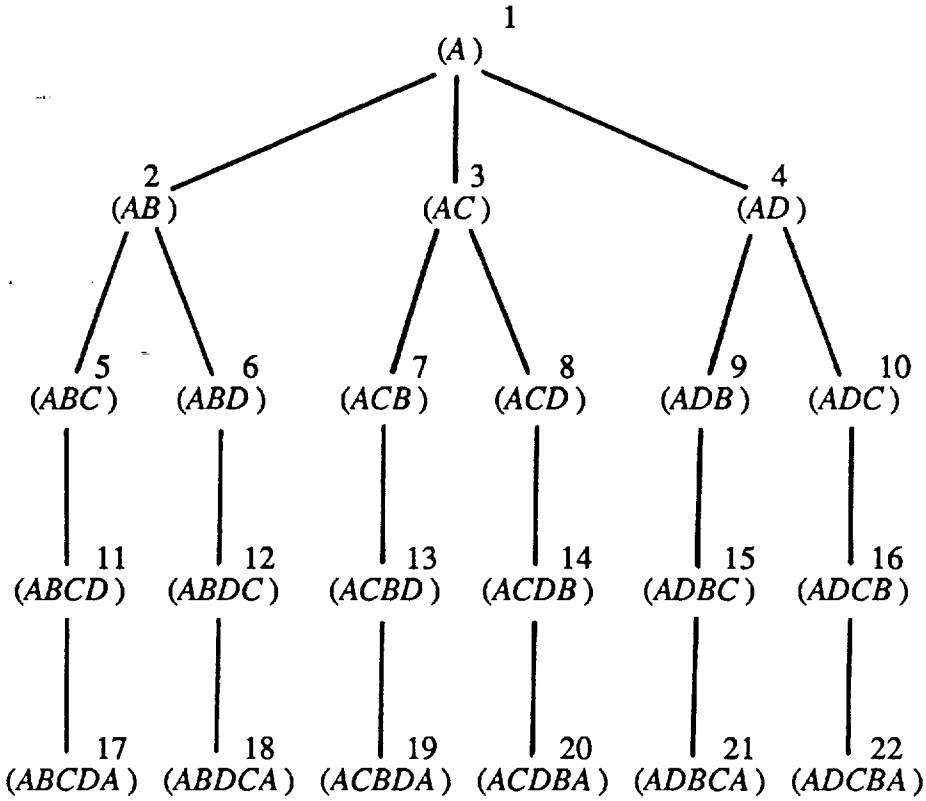


Figure 4.2. The general behavior of an exhaustive breadth first search. Numbers beside nodes indicate the ordering of node expansion for a search of a 4-city Traveling Salesman Problem state space tree.

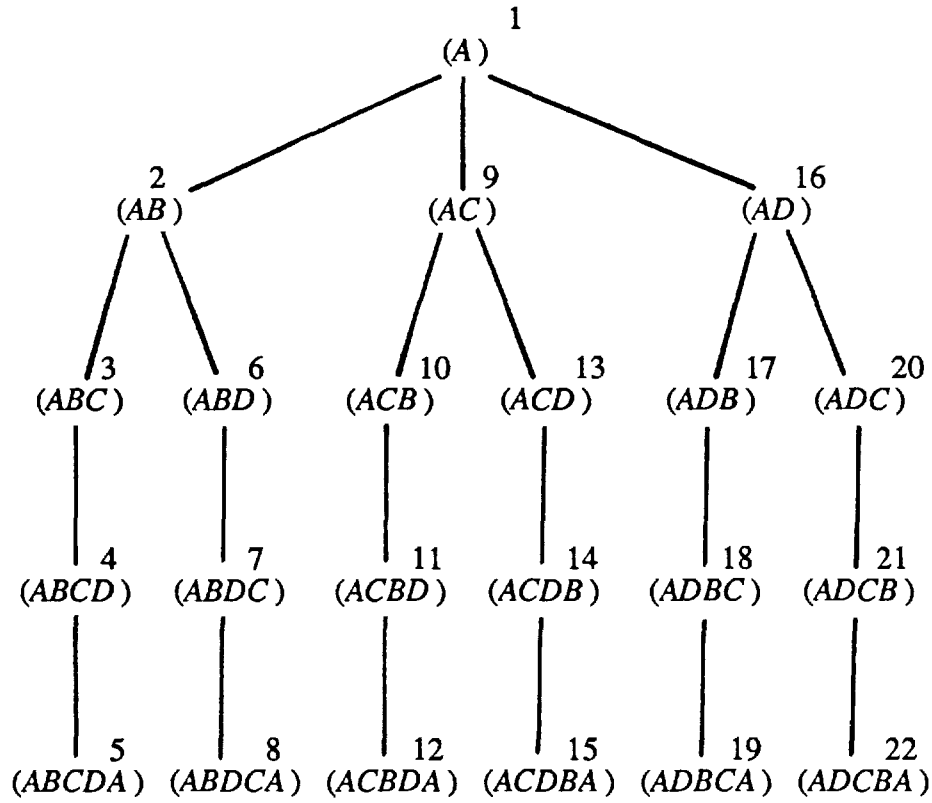


Figure 4.3. The general behavior of an exhaustive depth first search. Numbers beside nodes indicate the ordering of node expansion for a search of a 4-city Traveling Salesman Problem state space tree.

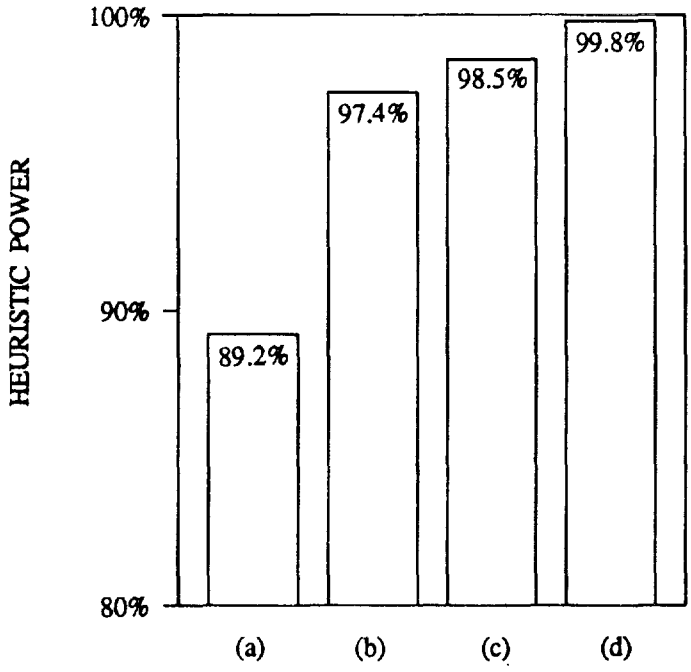


Figure 4.4. A comparison of Traveling Salesman Problem search results. The four search techniques are (a) depth first search with pruning, (b) depth first search with the $c_{\min}$ heuristic, (c) depth first search with the $\overline{c}_{\min_{ROW/COL}}$ heuristic, and (d) best first search with the $\overline{c}_{\min_{ROW/COL}}$ heuristic.

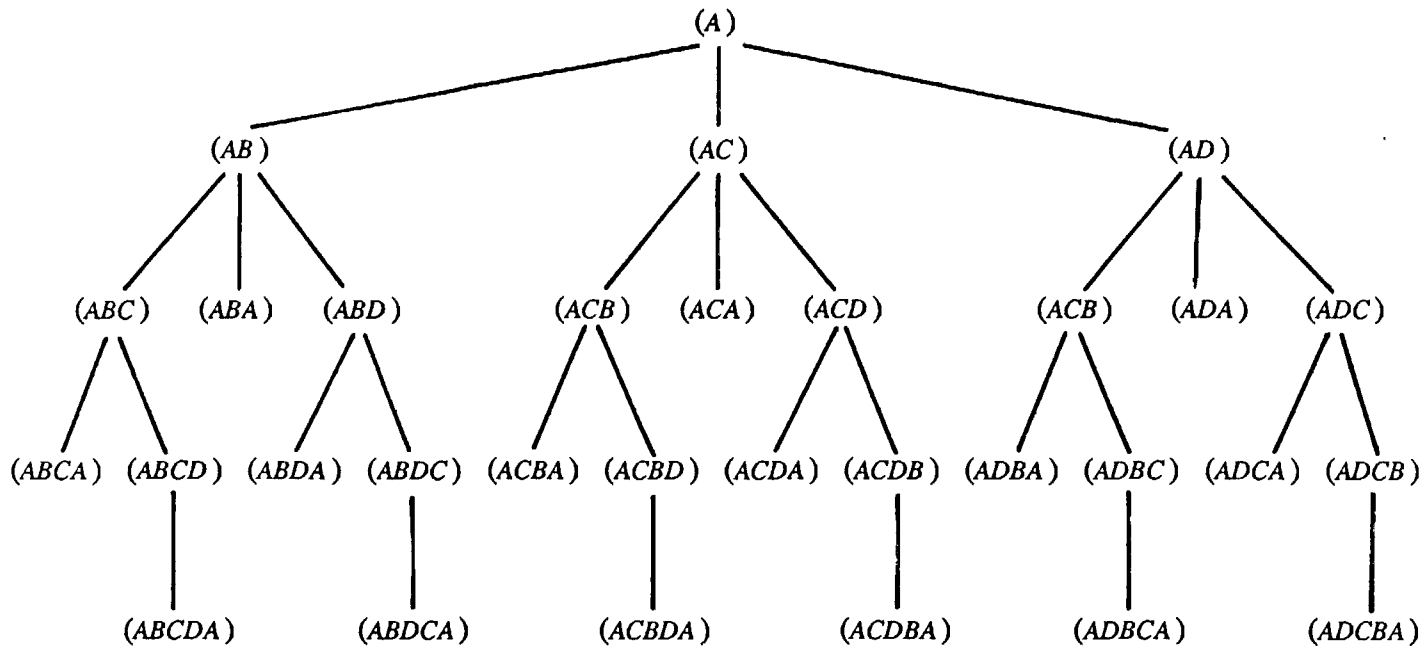


Figure 4.5. A state space tree for a relaxed 4-city Traveling Salesman Problem. City A, city B, city C, and city D may be used to compose a tour of any number of cities up to 4 cities. City A is the start city.

Table 4.1. Intercity costs for an 11-city Traveling Salesman Problem.

INTERCITY AIRLINE FARES											
From To	ATL	BOS	CHI	DFW	DEN	DTT	LAX	MSP	MSY	PHY	SEA
ATL		320	220	250	330	220	600	310	150	420	550
BOS	270		290	410	460	230	780	310	360	580	620
CHI	190	250		230	260	100	640	130	240	380	450
DFW	220	490	270		200	330	400	250	150	250	430
DEN	390	550	300	230		370	290	240	300	190	340
DTT	190	200	120	280	310		600	170	270	440	490
LAX	500	320	270	340	250	510		290	430	140	270
MSP	260	370	140	290	210	200	340		290	340	370
MSY	170	430	280	170	350	310	520	340		410	630
PHX	490	690	450	300	220	520	150	410	350		360
SEA	660	750	530	520	280	590	320	440	530	310	

Table 4.2. Uninformed exhaustive breadth first search results.

SEARCH RESULTS

Search Strategy: Exhaustive Breadth First Search
Heuristic: None
Optimal Tour: (DTT CHI MSP DEN SEA LAX PHX DFW MSY ATL BOS DTT)
Optimal Tour Cost: 2220

Local Search State Space Analysis:

	Nodes Created	Nodes in Full State Space Tree	Percent
Depth 0	1	1	100%
Depth 1	10	10	100%
Depth 2	90	90	100%
Depth 3	720	720	100%
Depth 4	5040	5040	100%
Depth 5	30240	30240	100%
Depth 6	151200	151200	100%
Depth 7	604800	604800	100%
Depth 8	1814400	1814400	100%
Depth 9	3628800	3628800	100%
Depth 10	3628800	3628800	100%
Depth 11	3628800	3628800	100%

First Pruning: not applicable

Global Search State Space Analysis:

Total Number of Nodes Created : 13492901
Number of Nodes in a Full State Space Tree : 13492901
Percent of Nodes Created : 100%

Table 4.3. Uninformed depth first search with pruning results.

SEARCH RESULTS

Search Strategy: Depth First Search with Pruning
Heuristic: None
Optimal Tour: (DTT CHI MSP DEN SEA LAX PHX DFW MSY ATL BOS DTT)
Optimal Tour Cost: 2220

Local Search State Space Analysis:

	Nodes Created	Nodes in Full State Space Tree	Percent
Depth 0	1	1	100%
Depth 1	10	10	100%
Depth 2	90	90	100%
Depth 3	720	720	100%
Depth 4	5040	5040	100%
Depth 5	30012	30240	99.2%
Depth 6	138945	151200	91.9%
Depth 7	401396	604800	66.4%
Depth 8	554211	1814400	15.3%
Depth 9	286700	3628800	7.9%
Depth 10	36535	3628800	1.0%
Depth 11	1356	3628800	0.04%

First Pruning: Depth 5

Global Search State Space Analysis:

Total Number of Nodes Created : 1455016
Number of Nodes in a Full State Space Tree : 13492901
Percent of Nodes Created : 10.8%

Table 4.4. Informed depth first search results with c_{min} heuristic.

SEARCH RESULTS

Search Strategy: Depth First Search
Heuristic: c_{min} Heuristic
Optimal Tour: (DTT CHI MSP DEN SEA LAX PHX DFW MSY ATL BOS DTT)
Optimal Tour Cost: 2220

Local Search State Space Analysis:

	Nodes Created	Nodes in Full State Space Tree	Percent
Depth 0	1	1	100%
Depth 1	10	10	100%
Depth 2	90	90	100%
Depth 3	720	720	100%
Depth 4	4578	5040	90.8%
Depth 5	20856	30240	69.0%
Depth 6	61355	151200	40.6%
Depth 7	108792	604800	18.0%
Depth 8	105465	1814400	5.8%
Depth 9	47216	3628800	1.3%
Depth 10	6690	3628800	0.18%
Depth 11	368	3628800	0.01%

First Pruning: Depth 4

Global Search State Space Analysis:

Total Number of Nodes Created : 356141
Number of Nodes in a Full State Space Tree : 13492901
Percent of Nodes Created : 2.6%

Table 4.5. Informed depth first search results with $\bar{c}_{\min_{ROW/COL}}$ heuristic.

SEARCH RESULTS

Search Strategy: Depth First Search
Heuristic: $\bar{c}_{\min_{ROW/COL}}$ Heuristic
Optimal Tour: (DTT CHI MSP DEN SEA LAX PHX DFW MSY ATL BOS DTT)
Optimal Tour Cost: 2220

Local Search State Space Analysis:

	Nodes Created	Nodes in Full State Space Tree	Percent
Depth 0	1	1	100%
Depth 1	10	10	100%
Depth 2	90	90	100%
Depth 3	584	720	81.1%
Depth 4	3108	5040	61.7%
Depth 5	11952	30240	39.5%
Depth 6	32230	151200	21.3%
Depth 7	55252	604800	9.1%
Depth 8	57498	1814400	3.2%
Depth 9	29272	3628800	0.81%
Depth 10	5462	3628800	0.15%
Depth 11	368	3628800	0.01%

First Pruning: Depth 3

Global Search State Space Analysis:

Total Number of Nodes Created : 195826
Number of Nodes in a Full State Space Tree : 13492901
Percent of Nodes Created : 1.45%

Table 4.6. Informed best first search results with $\bar{c}_{\min_{ROW/COL}}$ heuristic.

SEARCH RESULTS

Search Strategy: Best First Search
Heuristic: $\bar{c}_{\min_{ROW/COL}}$ Heuristic
Optimal Tour: (DTT CHI MSP DEN SEA LAX PHX DFW MSY ATL BOS DTT)
Optimal Tour Cost: 2220

Local Search State Space Analysis:

	Nodes Created	Nodes in Full State Space Tree	Percent
Depth 0	1	1	100%
Depth 1	10	10	100%
Depth 2	71	90	78.9%
Depth 3	423	720	58.7%
Depth 4	1757	5040	34.9%
Depth 5	4825	30240	16.0%
Depth 6	7858	151200	5.2%
Depth 7	7560	604800	1.3%
Depth 8	3767	1814400	0.21%
Depth 9	794	3628800	0.02%
Depth 10	8	3628800	0.00%
Depth 11	1	3628800	0.00%

First Pruning: Depth 2

Global Search State Space Analysis:

Total Number of Nodes Created : 27075
Number of Nodes in a Full State Space Tree : 13492901
Percent of Nodes Created : 0.20%

Table 4.7. Search results for a variation of the Traveling Salesman Problem.

SEARCH RESULTS

Search Strategy: Depth First Search
Heuristic: $\bar{c}_{\min_{ROW/COL}}$ Heuristic
Tour: (DTT ATL BOS CHI DFW DEN LAX PHX SEA MSP MSY DTT)
Tour Cost: 2930

Local Search State Space Analysis:

	Nodes Created	Nodes in Full State Space Tree	Percent
Depth 0	1	1	100%
Depth 1	1	10	10.0%
Depth 2	1	90	1.1%
Depth 3	1	720	0.1%
Depth 4	1	5040	0.9%
Depth 5	1	30240	0.00%
Depth 6	1	151200	0.00%
Depth 7	3	604800	0.00%
Depth 8	9	1814400	0.00%
Depth 9	17	3628800	0.00%
Depth 10	17	3628800	0.00%
Depth 11	17	3628800	0.00%

First Pruning: Depth 1

Global Search State Space Analysis:

Total Number of Nodes Created : 70
Number of Nodes in a Full State Space Tree : 13492901
Percent of Nodes Created : 0.00005%

Table 4.8. Search results for a relaxed Traveling Salesman Problem.

SEARCH RESULTS

Search Strategy: Best First Search
Heuristic: $\bar{c}_{\min_{ROW/COL}}$ Heuristic
Optimal Tour: (DTT BOS LAX PHX DEN DFW MSY ATL CHI MSP DTT)
Optimal Tour Cost: 1910

Local Search State Space Analysis:

	Nodes Created	Nodes in Full State Space Tree	Percent
Depth 0	1	1	100%
Depth 1	10	10	100%
Depth 2	70	100	70.0%
Depth 3	423	810	52.2%
Depth 4	1616	5760	28.1%
Depth 5	3787	35280	10.7%
Depth 6	5352	181440	2.9%
Depth 7	4055	756000	0.54%
Depth 8	1924	2419200	0.08%
Depth 9	408	5443200	0.01%
Depth 10	14	7257600	0.00%
Depth 11	0	3628800	0.00%

First Pruning: Depth 2

Global Search State Space Analysis:

Total Number of Nodes Created : 17660
Number of Nodes in a Full State Space Tree : 19728201
Percent of Nodes Created : 0.09%

Table 4.9. Intermediate search results for a relaxed Traveling Salesman Problem.

INTERMEDIATE SEARCH RESULTS		
Nodes Created	Current Best Tour	Cost
6168	(DTT CHI BOS LAX PHX DEN DFW DTT)	1530
6911	(DTT CHI MSP BOS LAX PHX DEN DFW DTT)	1690
9164	(DTT CHI BOS LAX PHX DEN DFW MSY ATL DTT)	1760
11592	(DTT CHI MSP BOS LAX PHX DEN DFW MSY ATL DTT)	1920
12912	(DTT BOS LAX PHX DEN DFW MSY ATL CHI MSP DTT)	1910
17660†	(DTT BOS LAX PHX DEN DFW MSY ATL CHI MSP DTT)	1910

† State when 11-city problem is determined not possible.

CHAPTER 5

VORONOI DIAGRAM SEARCH GRAPHS FOR MODELING PATHS IN MOUNTAINOUS TERRAIN

This chapter investigates the problem of generating search graphs of the free space around polygon obstacles. This approach can be used to determine paths through mountainous terrain or paths for nap of the earth flight. First, a simple grid graph is presented for comparison with the Voronoi diagram graphs that follow. Three methods of generating search graphs from Voronoi diagrams are presented: the centroid point method, the circle rule method, and the contour vertex point method. Several terrain examples are discussed in order to illustrate the salient features of these terrain modeling techniques.

The Terrain/Threat Environment

In order to characterize the environment through which the vehicle will travel, a search graph is constructed. Since the navigation problem, as defined in this thesis, is restricted to a constant altitude flight through mountainous terrain with stationary threats, the search space could be represented as a contour map depicting mountain regions — these will be thought of as obstacle boundaries. Threats can be modeled on the contour map as regions of increased risk, as illustrated in Figure 5.1.

The obstacles on the contour map are modified to become polygons through a process involving hysteresis filtering of obstacle data to smooth jagged boundaries and a polygonization process to represent the obstacles as polygons. Many techniques have been developed to produce polygons with as few sides as necessary, including methods which totally enclose the original region and methods which minimize the error between the polygon and the obstacle data [17,41,42,50].

A terrain path planning search graph depicts possible paths through the terrain/threat search space. A good search graph includes at least one path between all neighboring obstacles. These paths are represented within a graph of nodes

connected by arcs, where the arcs represent possible paths to be flown. The search environment can be thought of as separated into two spaces, the *free space* and the *obstacle space*. The free space, through which the vehicle is free to move, is unoccupied by obstacles, and the obstacle space is occupied by obstacles. Nodes and arcs of a search graph do not necessarily have to be in the free space, however, if arc lines cross obstacle boundaries, then they cannot be followed on a flight trajectory. Paths along these arcs must be eliminated during a search procedure, possibly by assigning infinite costs to traveling these arcs. A better method of modeling the polygon obstacle search space is to model only the free space with a *feasible* search graph. A feasible search graph includes only nodes and arcs that are in the free space.

Frequently, the planning objective is to minimize the path length and threat exposure cost for a trajectory. For this purpose, information on path length and threat cost should be represented by the search graph. The path length can be represented by simply labeling graph arcs with the distance between nodes. Threat information is combined with the path length arc costs, assuming that for any segment of a trajectory that crosses a threat region, a cost value can be assigned to the segment. This cost would represent the increased risk incurred by flying over the segment. Path length and threat costs are weighted and summed according to their relative importance, and a final cost is then assigned to each graph arc.

Finally, when modeling the terrain/threat environment, one should note that the threat regions are *not* modeled as obstacles. Threat regions *can* be flown through, while mountains are obstacles that *cannot* be flown through. There is always the possibility that an aircraft is flying in an extremely threatening environment where barriers of threats must be penetrated. If threat regions were modeled as obstacles, there would not exist the option to fly through a barrier of threats, rather than to fly around them.

Graphs From Simple Grids

Consider the graph search space generated from a simple square lattice, as presented in Chapter 3. Two types of grid connections are typical, connecting four or eight neighboring grid points. Figure 5.2 illustrates the application of a four neighbor grid for the graph of the terrain in Figure 5.1. This search graph is created for comparison with the Voronoi diagram search graphs that will follow.

It will not be discussed how the sizing of the grid should be established. Certainly, the grid size is an important parameter since using a fine grid would require a large quantity of solutions to be considered by a search algorithm, and a coarse grid

may not represent all possible paths through the terrain. The smallest obstacle considered in the terrain model and the proximity of obstacles influence the size of the grid. Difficulties in grid sizing has motivated the development of techniques that have varying grid sizes. The quadtree technique discussed in Chapter 3 is an example of a technique with a varying grid size.

The search graph constructed from a square lattice has a few notable qualities. First, establishing the grid graph is made simpler (computationally) by the uniform structure of the graph. If fine detail is needed for a path solution, a finer grid can be chosen. However, if the computational work in searching a graph becomes excessive, it is difficult to generalize the grid to a more coarse grid and maintain a good representation of free space. The simple grid point method can fail to represent some paths between obstacles when a coarse grid is used.

Graphs from Voronoi Diagrams †

The use of Voronoi diagrams for establishing a search graph will now be discussed. First, the definition of a Voronoi diagram is given to introduce this geometric construct. Then, through a series of examples, the use of Voronoi diagrams for terrain search graphs will be presented.

A *Voronoi diagram* for a set of N points p_i, $1 \le i \le N$, in the Euclidean plane is a partitioning of the plane into N polygonal regions, one region associated with each point p_i. A point p_i is referred to as a *Delaunay point.* Figure 5.3 shows the Voronoi diagram for a set of points. The *Voronoi region* $V(p_i)$ associated with point p_i is the locus of points closer to p_i than to any of the other $N-1$ points. The *Voronoi edge* separating $V(p_i)$ from $V(p_j)$ is composed of the points equidistant from p_i and p_j. Note that not all Voronoi edges are bounded; some extend to infinity. The intersection of Voronoi edges occur at vertices called *Voronoi points.* Appendix B gives details of the construction of Delaunay triangulations (used in the construction of Voronoi diagrams) and Voronoi diagrams.

Voronoi diagrams provide useful information for solving a number of problems involving the proximity of N points in the plane. The geometric structure of the

† All Delaunay triangulations and Voronoi diagrams in this thesis assume that no four points are cocircular and that no three points are colinear. These assumptions allow for a simple procedure for the generation of these geometric constructs, however, these assumptions are not necessary in general — Guibas and Stolfoi [20] present a procedure that allows for these anomalies.

diagram allows for efficient solutions to problems involving the k nearest and farthest neighbors, the two closest points, the smallest circle enclosing the set, and the Euclidean minimum spanning tree [57]. The use of Voronoi diagrams for defining neighboring points is also useful for navigation path planning. Consider the Voronoi edge to separate two neighboring Delaunay points. Using the terminology of Appendix B, two Delaunay points are *neighbors* if a Delaunay edge connects them in the Delaunay triangulation. With this definition, the following example should introduce the fundamentals for generating search graphs using Voronoi diagrams.

A simple example using Voronoi diagrams for establishing a search graph involves search paths with maximum avoidance of point obstacles. The problem is to find a path from a start location **S** to a finish location **F** in a plane while maximizing the distance from nearest point obstacles and minimizing the distance traveled. The maximization of the distance from point obstacles shall be considered of much greater importance than minimizing the path length. However, specific weights are not given since this problem is intended to be a pedagogical example introducing Voronoi diagrams and not a numerical exercise.

From the definition of a Voronoi edge, we see that the objective of maximizing the distance from the two nearest point obstacles is achieved by following the paths on Voronoi edges. Since the start and finish locations most likely will not be on a Voronoi edge, then a path must be constructed from these locations to a Voronoi edge to start and finish a search. A convenient way to do this is to construct a line segment from the start (or finish) location to the closest Voronoi edge. The start point, finish point, and Voronoi points of the Voronoi diagram are used as nodes of a search graph with the Voronoi edges as arcs. Costs can be assigned to the arcs based on the distance between the nodes. A graph search could be performed to find a path from the start node to the finish node, while minimizing the total arc cost. Any Voronoi edge that extends infinitely could be assigned an infinite length cost and will consequently be eliminated from appearing as part of a minimum length path solution. Figure 5.4 shows an example of a search graph for an arbitrary set of points.

The Centroid Method

The previous example for maximum avoidance of point obstacles may be used to model the terrain for the navigation path planning problem. Each obstacle in the plane could be represented by one point at its centroid. The Voronoi edges that extend infinitely could be bounded by a region that encloses the entire terrain map, creating path search arcs around the boundary of the terrain map. Figure 5.5

illustrates a search graph for the terrain map of Figure 5.1. Defining nodes and arcs for the start and finish locations as mentioned above, a search can be performed using Voronoi edges and these boundary arc additions.

This method would not do a good job in representing clear paths to follow because some of the Voronoi edges may intersect obstacles. For example, Figure 5.6 illustrates a Voronoi diagram which clearly has a Voronoi edge that crosses an obstacle. This is due to the fact that the obstacles are of varying shapes and sizes, and the Voronoi edges are only representing points equidistant from the center points within obstacles. Indeed, the only space that this method would model correctly would be the case of nonoverlapping circular obstacles, all of the same radius. This would effectively be solving the path problem of flying at low altitude in an environment of telephone poles and street lamps. This is of limited practical use.

The graph developed from the Voronoi diagram has some notable qualities. Very few nodes are used to create a graph that attempts to represent all possible routes around obstacles. Although lines may cross over obstacle boundaries, the resulting graph is an attempt to create a single path between neighboring obstacles. Thus, the topological structure of free space is well represented. Only one path between neighboring obstacles exists, consequently, when searching the free space for a path, the only decision to be made at each node is which passage way to proceed through next. With the grid search graph, many possible paths through each passage way between neighboring obstacles may exist. Nonetheless, the centroid method is hardly useful when having to model complex terrain. As illustrated in Figure 5.6, it is quite possible to have a Voronoi edge cross over an obstacle boundary even with fairly simple obstacle shapes. However, further modifications to the way a Voronoi diagram is used to model obstacles will mitigate these problems. These methods are discussed next.

The Circle Rule Method

The previous method using single point locations at the center of obstacles lacks the information of where the boundaries of obstacles are located. The boundaries of obstacles are not modeled and, consequently, it is quite possible for a Voronoi edge to intersect an obstacle boundary. The next logical step for improving this model would be to incorporate information on shapes of obstacles. This can be done using multiple Delaunay points to model the obstacles and performing some modifications to the Voronoi diagram for edges that are associated with Delaunay points common to the same obstacle.

This method entails modeling obstacle boundaries with well placed Delaunay points within obstacle boundaries. For any Delaunay point within an obstacle, imagine a construction circle around it. All Delaunay points will have a construction circle around them with the same radius. To model an obstacle, the union of the interior regions enclosed by overlapping construction circles must completely enclose the obstacle polygon boundary (but not necessarily the entire obstacle interior). Construction circles from neighboring obstacles must not overlap. Developing guidelines for the placement of Delaunay points within obstacle boundaries using the circle rule method will not be discussed. Figure 5.7 shows two obstacles, one modeled with a single Delaunay point and another modeled with several Delaunay points. The construction construction circles are shown in dashed arcs.

After Delaunay points are placed within all the obstacles, the Voronoi diagram is constructed. The Voronoi edges that extend infinitely could be bounded by a region that encloses the entire terrain map, creating path search arcs that allow for paths around the boundary of the terrain map. Figure 5.8 shows the Delaunay points and the Voronoi diagram for the terrain of Figure 5.1. The next step is to remove those Voronoi edges that are defined by two neighboring Delaunay points modeling the same obstacle. Figure 5.9 shows the final search graph.

In comparison to the grid graph for the polygon obstacles of Figure 5.1, the resultant Voronoi graph of Figure 5.9 models the free space in fewer nodes. Again, as mentioned for the centroid point method, the graph gives paths that topologically represents all possible routes around the obstacles with only one path between neighboring obstacles. The circle rule method does a good job modeling complex terrain environments. A typical terrain environment that is difficult to model for some methods is a box canyon. As seen in Figure 5.10 and Figure 5.11, the box canyon can be modeled well with the circle rule method. The graph shows that no path leads into the box canyon. Using multiple Delaunay points to model obstacles requires a more complex Voronoi diagram to be constructed, however, the resulting search graph will no longer have Voronoi edges that cross over obstacle regions, provided the circle overlap condition is met. The justification for this is discussed next.

The motivation for construction circles around the Delaunay points is to allow for boundary information of obstacles. In the modified Voronoi diagram graph, a Voronoi edge passes between neighboring Delaunay points, provided that these Delaunay points are *not* from the same object. Note that when neighboring Deluanay points are from the same object the construction circles intersect, and when they are from neighboring obstacles they do not. Consequently, a Voronoi edge that passes between two Delaunay points from the same obstacle may pass over the two construction circles for these points, while the Voronoi edge that passes between two

Delaunay points from neighboring obstacles may will *not* cross over either of the construction circles, but will pass exactly between them. Only the Voronoi edges that are formed from Delaunay points from the same obstacle may cross over the construction circles and the obstacle boundary for that obstacle. Since the construction circles for an obstacle completely cover the obstacle boundary and the corresponding Voronoi edges are removed from the resultant Voronoi diagram, then all the Voronoi edges that cross over obstacle boundaries are removed. All the remaining Voronoi edges lie completely in free space, since they do not cross over any construction circles. The resultant Voronoi diagram graph is a feasible search graph (Appendix C provides a more detailed proof).

The Contour Vertex Point Method

The method of modeling obstacles with multiple Delaunay points requires the choice of a circle radius parameter and the judicious placement of Delaunay points in order to get a good representation of free space. This work can be eliminated by using existing points on the obstacle contours for establishing Delaunay point locations. The polygon obstacle vertices can conveniently be used as Delaunay point locations to model the obstacles. Intuitively, one can see that the vertices of obstacles are the points that should be avoided when searching for a path through polygon obstacles.

The procedure is similar to that of the circle rule method. First, a Voronoi diagram is constructed for the set of Delaunay points describing the obstacle vertices. The Voronoi edges that extend infinitely could be bounded by a region that encloses the entire terrain map, creating path search arcs around the boundary of the terrain map. Figure 5.12 illustrates the complete Voronoi diagram search graph using the vertices of the obstacles in Figure 5.1. Voronoi edges that are defined by two neighboring Delaunay points of the same obstacle are removed. Start and finish search nodes can be added to the search graph by connecting these point locations to the closest Voronoi edge. The resulting search graph maximizes the distance from the closest obstacle vertices, as shown in Figure 5.13.

In comparison with the previous methods for developing search graphs (cf. Figure 5.2, Figure 5.5, Figure 5.9, and Figure 5.13), the vertex point method gives a good representation for free space using a fairly small amount of nodes. The number of Delaunay points typically exceeds the number of Delaunay points needed using the circle method, but the vertex point method does not require additional analysis to determine where to locate Delaunay points. However, the vertex point method does require polygonization of obstacles. One should note that although the vertex point

method typically creates a search graph with more search nodes, many of these nodes do not increase the complexity of the search. That is, many of the Voronoi points representing search nodes connect only two Voronoi edges in the search graph, and when a search proceeds to expand such a node, there is only one direction to proceed. In contrast, when a Voronoi point connects three Voronoi edges in the search graph, a comparison must be made to determine which of the two Voronoi edges should be traversed after arriving at the Voronoi point from the third Voronoi edge.

Like the circle rule method, the vertex point method guarantees that the resultant Voronoi diagram graph will depict only feasible paths, paths that are completely in the free space. In the case of the vertex point method, feasible paths are guaranteed based on a parameter of the polygonization:

> Let the closest distance from any vertex of an obstacle to a neighboring obstacle be defined as δ. If obstacle polygonization is performed using polygon sides no greater in length than δ, then the resultant Voronoi search graph will depict only feasible paths.

Appendix C provides a proof of the feasibility of search graphs generated with the contour vertex point method.

The combination of using few nodes to represent all possible paths through the passage ways of mountains and depicting only feasible paths makes the vertex point method a useful method of modeling free space. The Voronoi diagram graph provides a simple graph that can be used for deciding which passage ways to proceed through while progressing from a start to a finish location. Later, after the passage ways have been chosen by a search algorithm, a search for a finer solution in the passage regions could be performed. This will be discussed in more detail later for the navigation path planning examples in Chapter 7.

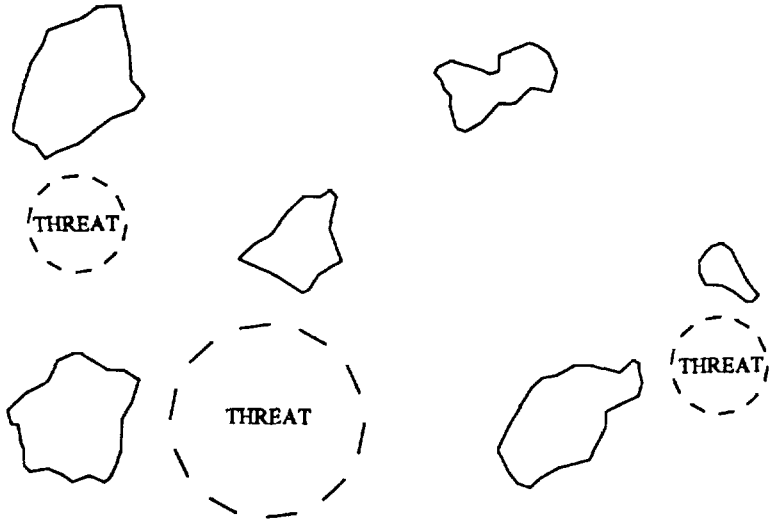


Figure 5.1. A map with obstacle and threat regions.

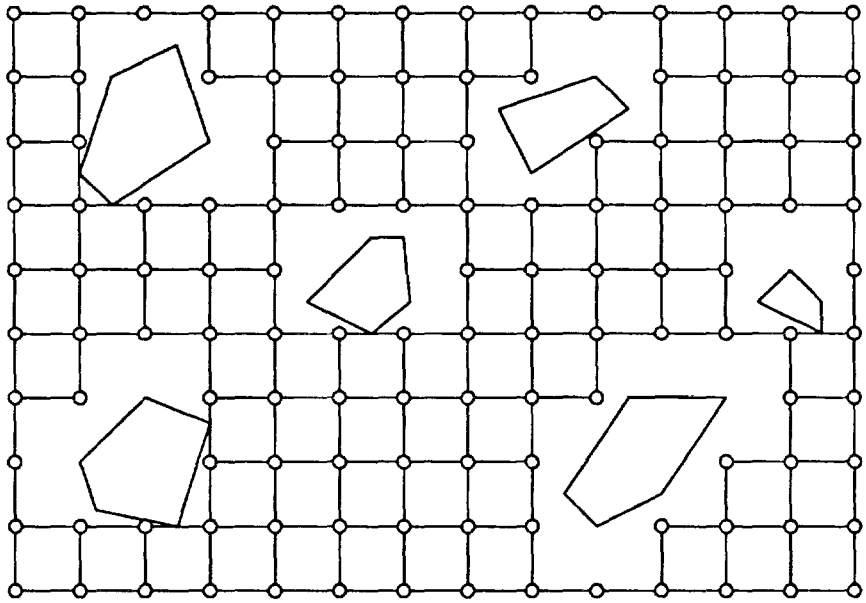


Figure 5.2. A graph search space created with a simple grid.

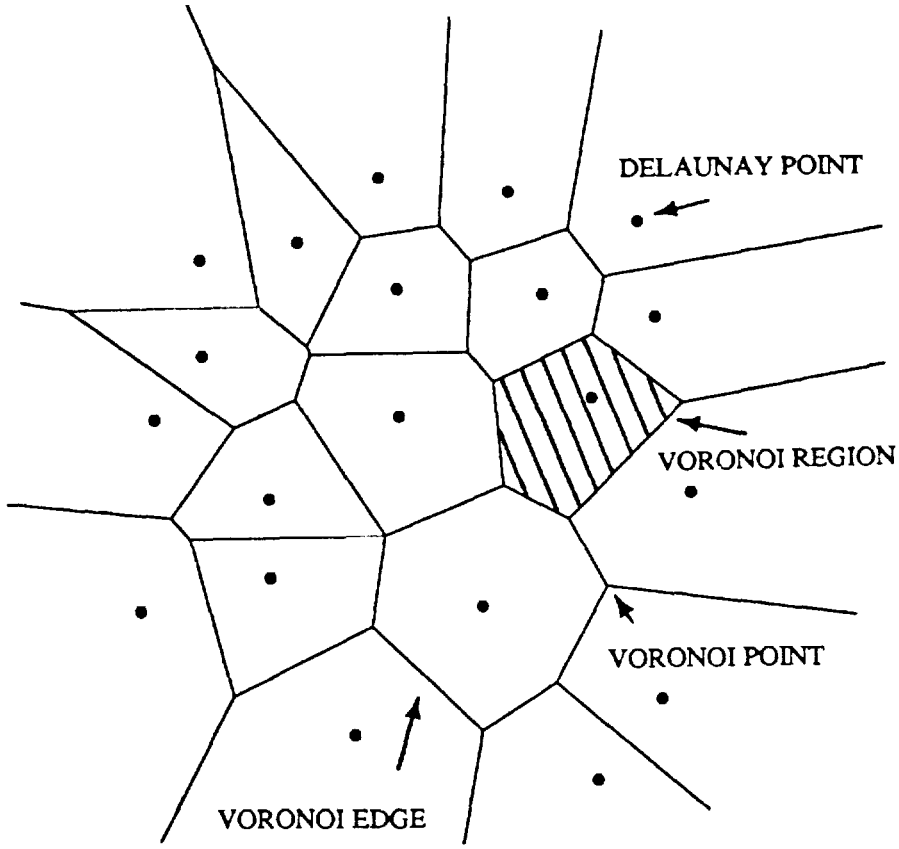


Figure 5.3. The Voronoi diagram for a set of 20 points.

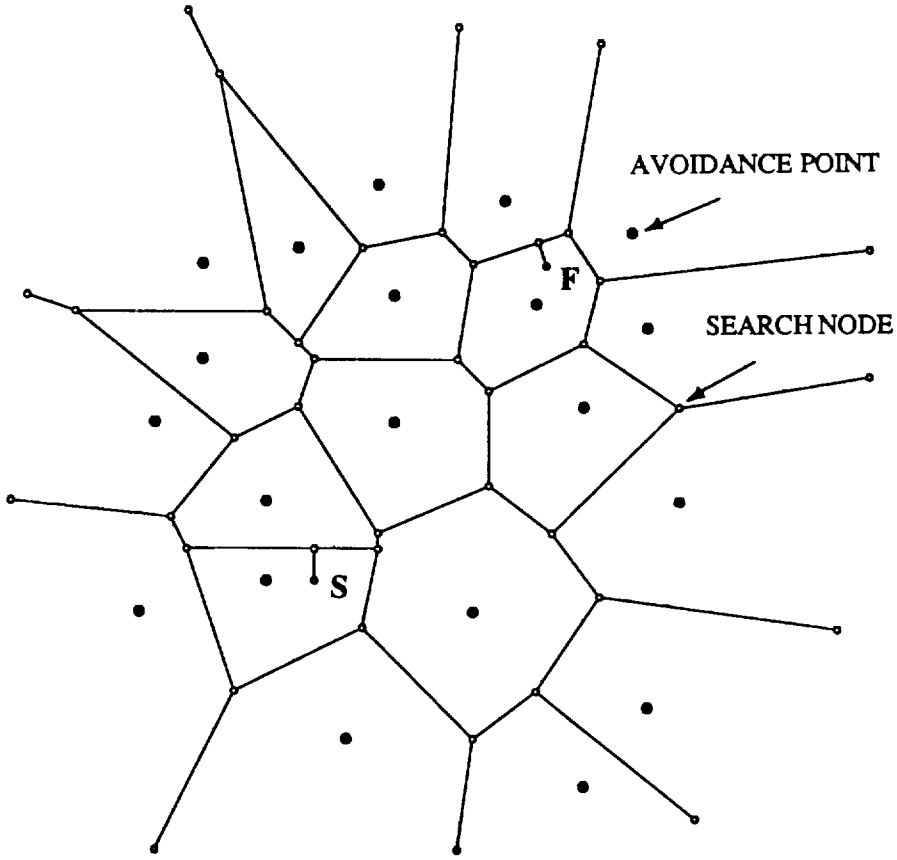


Figure 5.4. A graph search space for point obstacles.

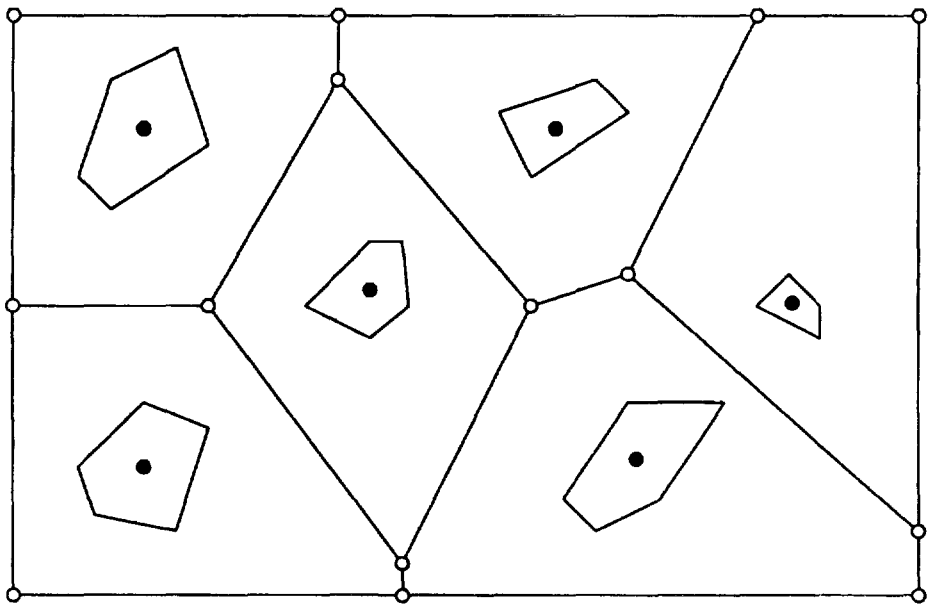


Figure 5.5. A graph search space for polygon obstacles modeled with the centroid method.

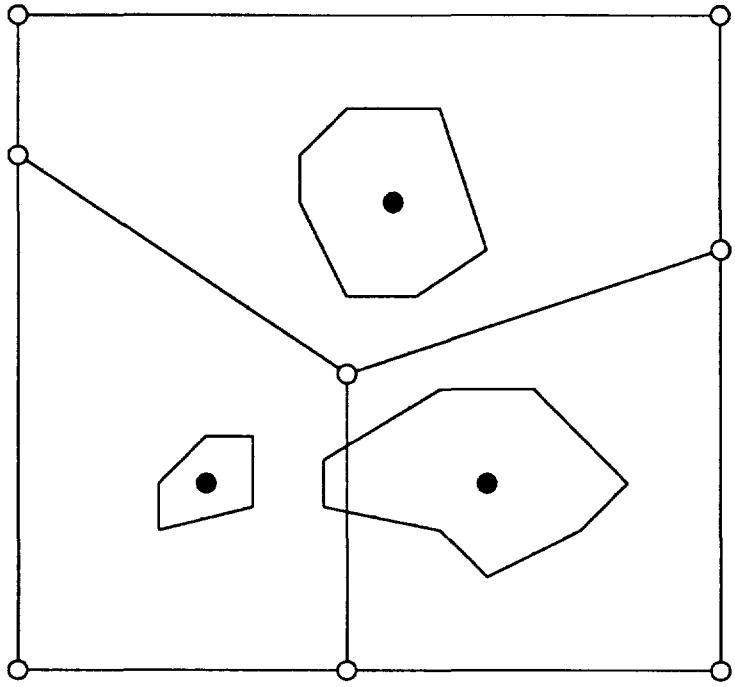


Figure 5.6. An example of a Voronoi edge crossing an obstacle boundary.

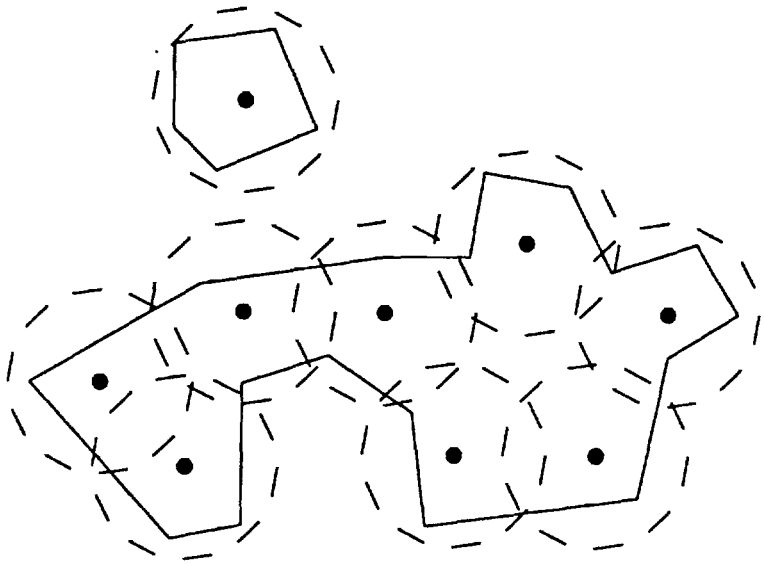


Figure 5.7. Two obstacles modeled with the circle rule.

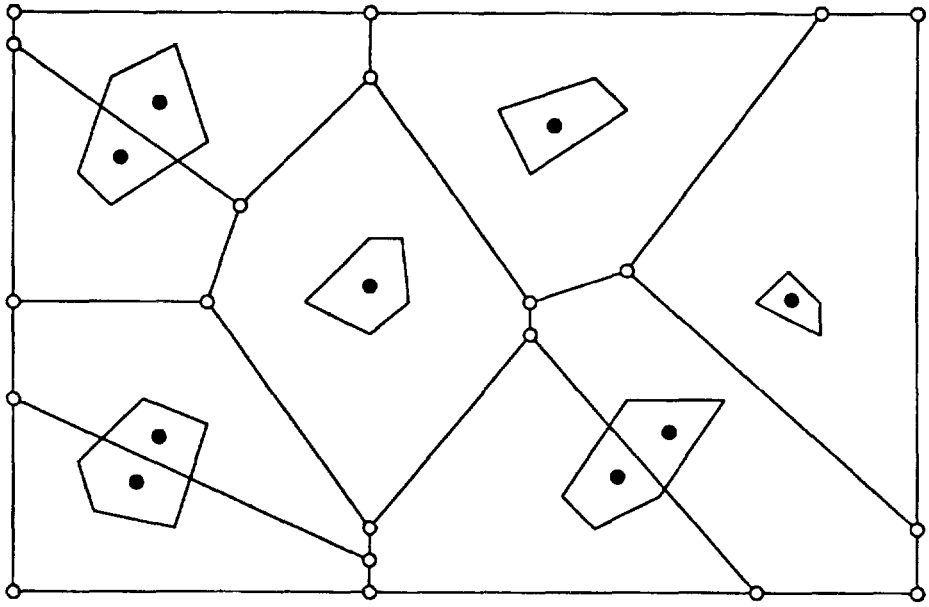


Figure 5.8. The complete Voronoi diagram graph for polygon obstacles modeled with the circle rule.

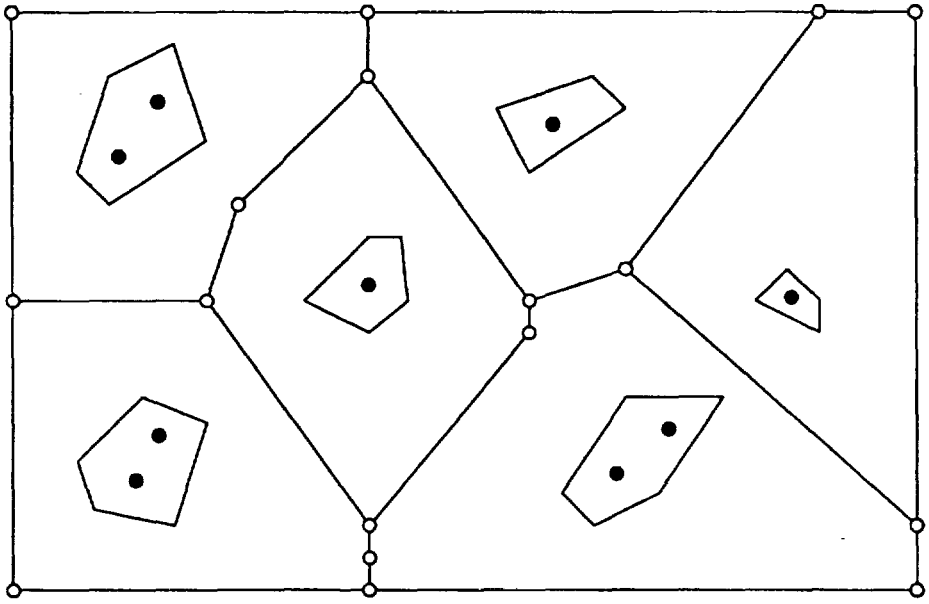


Figure 5.9. The modified Voronoi diagram graph for polygon obstacles modeled with the circle rule.

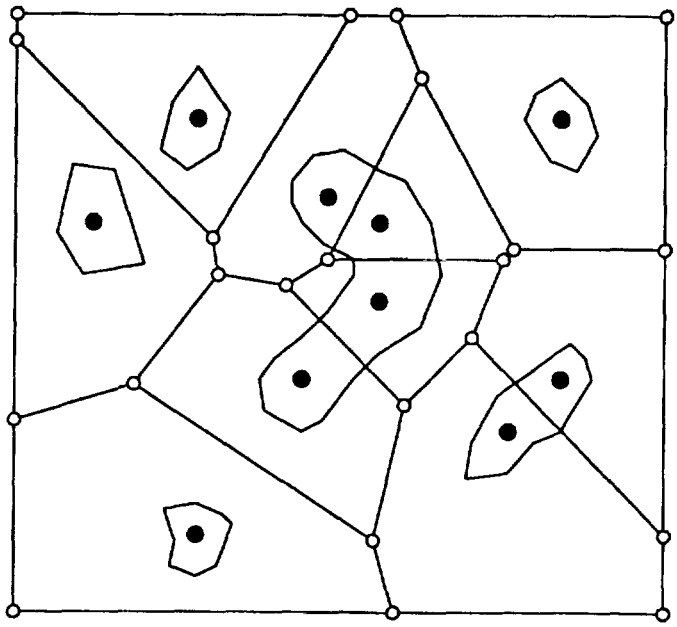


Figure 5.10. The complete Voronoi diagram graph for a box canyon modeled with the circle rule.

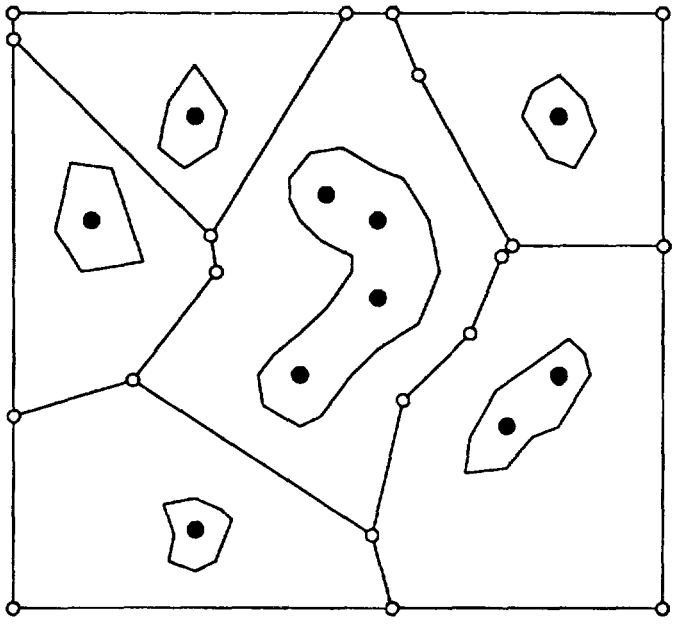


Figure 5.11. The modified Voronoi diagram graph for a box canyon modeled with the circle rule.

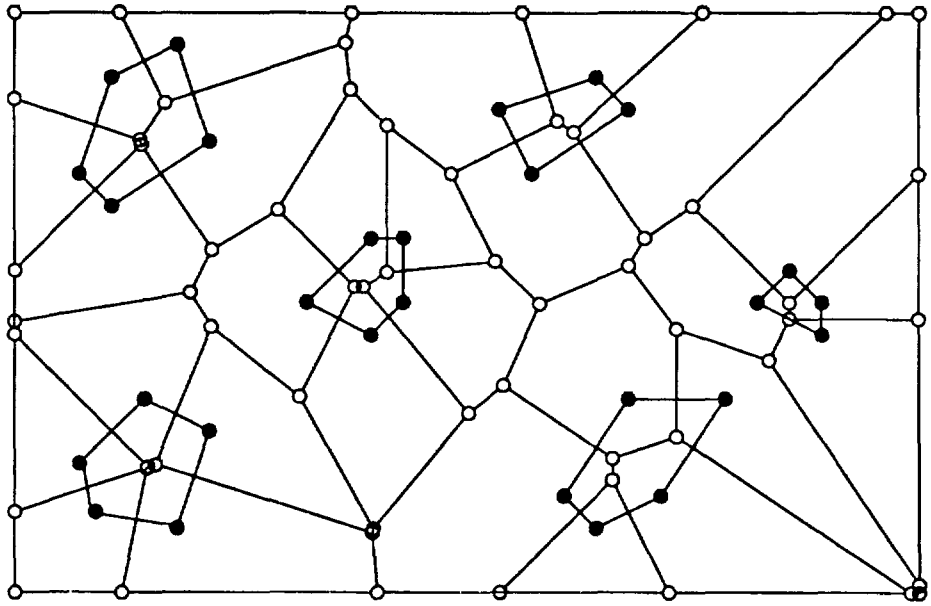


Figure 5.12. The complete Voronoi diagram graph for polygon obstacles modeled with Delaunay points at the vertices of the obstacles.

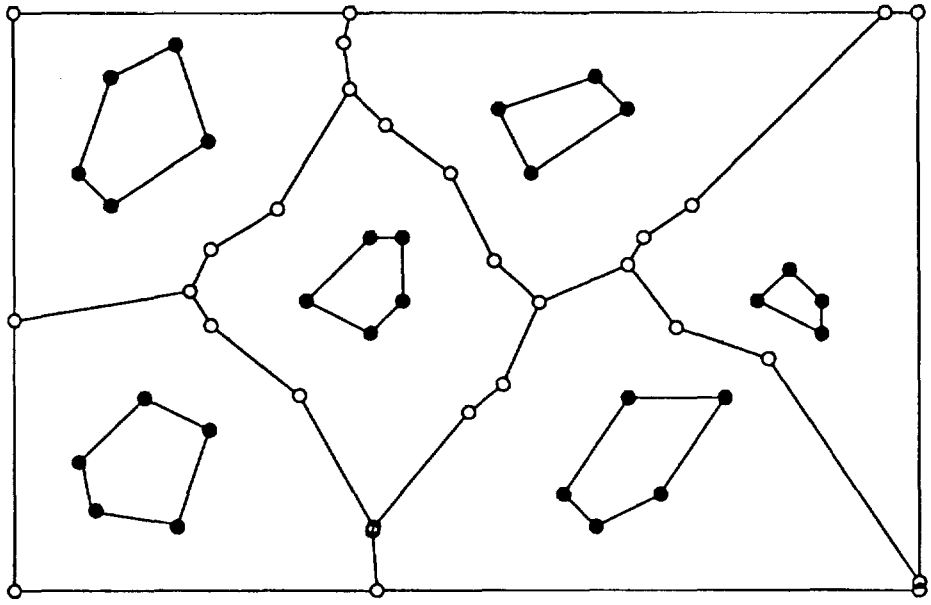


Figure 5.13. The modified Voronoi diagram graph for polygon obstacles modeled with Delaunay points at the vertices of the obstacles.

CHAPTER 6

GRAPH SEARCH TECHNIQUES FOR NAVIGATION PATH PLANNING

This chapter presents some graph search techniques suitable for the grid lattice graphs and the Voronoi diagram graphs presented in Chapter 5. The dynamic programming algorithm, Dijkstra's algorithm, and the A^* algorithm are presented and illustrated with examples.

The Dynamic Programming Solution Technique

Dynamic programming was first introduced by Richard Bellman [3] as a computational method for solving optimization problems. One set of application of dynamic programming is sequential decision process problems. Path planning is such an application since one must select a sequence of graph arcs to traverse in order to achieve an optimal path based on the cost associated with those arcs. The dynamic programming technique will now be introduced with an example of a grid search.

Consider the directed search graph constructed from the uniform grid of Figure 6.1 (hereafter the directed arcs will be omitted and arrows on arcs will correspond to pointers indicating intermediate results). The objective is to find a path from the start node S to the goal node F on the graph, constrained to proceed only from left to right, which minimizes the sum of the arc costs. An arc cost represents the non-negative cost to travel from one grid point to another. The optimal direction to proceed is the decision made at each node.

Dynamic programming segments a problem in stages based on one variable that progresses monotonically. This is typically the time variable (where applicable). However, the grid search problem is not stated to progress in stages of time, but in terms of grid arcs which lead to the goal node. To direct the grid search, the path is restricted to progress from the left to right, from S to F. This restriction guides the search and prevents cycling within the graph network.

The path problem progresses in *stages*. At each stage a *decision* is made. Figure 6.2 shows the six stages where decisions are made. At stage I, one can decide to go from node S to node A or node B, each of which is located at stage II. Similarly, at node A, one can decide to proceed to node C or node D, each of which is located at stage III. Any path from **S** to **F** is called a *policy*. An example of a policy would be the path (SADGJMF) or (SBDHLNF). Any connected section of a policy is called a *subpolicy*, e.g., sections (SAD), (SBDHL), (LNF), and (GJM).

At the heart of the dynamic programming technique is Bellman's principle of optimality [3]:

> An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

More simply stated for the context of this path planning problem:

An optimal policy must contain only optimal subpolicies.

The justification of this is quite simple for the path problem. Suppose that an extracted subpolicy of the optimal policy was not optimal. Then, there exists another subpolicy that could be substituted for this extracted subpolicy that would improve the optimal policy. This constitutes a deduction that violates the hypothesis that the original policy was an optimal policy.

The principle of optimality makes it possible to retain the optimal solution with fewer calculations than brute-force enumeration. In general, the dynamic programming algorithm converts a problem into smaller subproblems. Subproblems are solved sequentially, and the corresponding optimal solutions are combined to yield the overall optimal solution. The amount of work to solve a problem with dynamic programming increases linearly with the number of subproblems. A brute-force enumeration procedure would have an exponential growth with the number of subproblems. Thus, the computational work of dynamic programming is far less than brute-force enumeration procedures.

In order to represent the problem mathematically, it is convenient to define the following terms:

$C(i, n, m)$ = the cost between node n at stage i and node m at stage $(i+1)$,

$g(i, n)$ = the minimum cost from node n at stage i to the final node **F**.

A fundamental approach of dynamic programming is to solve the problem backwards (the implications of solving the problem forward will be discussed later). The first problem solved is at stage VI. The optimal policy from node M to node **F** is (MF) with a cost of 3, and the optimal policy from node N to node **F** is (NF) with a cost of 5: written as g (VI,M)=3, and g (VI,N)=5. A pointer can be used to maintain

this optimal subpolicy result on the graph. The pointer indicates which direction to proceed at a node. The next step is to solve the optimal policy problem from stage V. From node J to **F** the optimal policy is (JMF) with a cost of 7. From node K to node **F** the optimal policy is (KMF) with a cost of 7. From node L to node **F** the optimal policy is (LNF) with a cost of 9. These results are found as follows:

$$g(\mathrm{V}, \mathrm{J}) = \min_{m=\mathrm{M}} [g(\mathrm{VI}, m) + \mathrm{C}(\mathrm{V}, \mathrm{J}, m)] = 7 \quad \text{with } m=\mathrm{M},$$

$$g(\mathrm{V}, \mathrm{K}) = \min_{m=\mathrm{M,N}} [g(\mathrm{VI}, m) + \mathrm{C}(\mathrm{V}, \mathrm{K}, m)] = 7 \quad \text{with } m=\mathrm{M},$$

$$g(\mathrm{V}, \mathrm{L}) = \min_{m=\mathrm{N}} [g(\mathrm{VI}, m) + \mathrm{C}(\mathrm{V}, \mathrm{L}, m)] = 9 \quad \text{with } m=\mathrm{N}.$$

This procedure is followed backward to stage I to find the optimal policy from **S** to **F**. The general procedure is the following:

$$g(i, n) = \min_{m=SUCC(i,n)} [\, g(i+1, m) + \mathrm{C}(i, n, m) \,],$$

where $SUCC(i, n)$ is a successor at stage $(i+1)$ to the node n at stage i. For this example the successor to a node n is either the node to the upper right or lower right of node n. The optimal policy for this example is (SACGJMF). This policy is shown in a dashed line in Figure 6.3. The values of g are shown in parentheses above each node, and subpolicy pointers are also labeled. Note that is is possible to have two pointers (thus two optimal subpolicies) for one node, such is the case shown for node E.

The solution graph has some auxiliary information. Not only is the optimal policy for a path from **S** to **F** known, but the optimal policy from any point in the graph to **F** is also known. This assists error corrections since the new optimal policy is instantly known if the state drifts from one node to a different one. This is the benefit of the backward direction solution. If the dynamic programming solution had been performed in the forward direction, the auxiliary information would be different. The pointers for the solution graph would indicate the optimal policy for going *from* **S** to any node on the graph — rather than the optimal policy from any node on the graph *to* **F**. Nonetheless, both solution procedures will find the same optimal solution for the path from **S** to **F**. Figure 6.4 shows the optimal policy for the graph when solved in the forward direction. Choosing the direction of progression for the dynamic programming solution is usually just a decision based on what auxiliary information is more useful.

The dynamic programming algorithm presented here is well suited for searching grid lattice graphs. However, Voronoi diagram graphs are less uniform in terms of

node positions. For Voronoi diagram graphs, one needs a method for searching an arbitrary graph.

Dijkstra's Dynamic Programming Algorithm

In the previous section a grid search illustrated the general procedure of dynamic programming. The following example extends the technique as applied to arbitrary graphs.

Consider an arbitrary undirected graph as shown in Figure 6.5. The problem is to find a path from **S** to **F** on the graph such that the sum of the arc costs is a minimum. As before, the arc costs are non-negative and represent the cost to travel from one node to another, and the decisions made are simply decisions of which direction to proceed at each node.

In contrast with the previous example, the arbitrary graph does not necessarily have a natural progression from left to right. However, the problem could still be formulated to progress from **S** to **F**. To proceed, successions of nodes are investigated while keeping track of previous investigations so that cycling does not occur. The progression will become apparent as the example is explained.

Some notation established in the previous example will be used, but will be altered in the following manner:

$C(n, m)$ = the cost between node n and node m,

$g(n)$ = the minimum cost from the start node S to the node n.

In addition to these functions, the node successor operator will be used. Simply stated, a successor of a node n is any node connected to the node n in the graph. For example, the successors of node S in Figure 6.5 are nodes A,B, and D.

An algorithm which implements dynamic programming for arbitrary graphs is called Dijkstra's dynamic programming algorithm [14], stated in Figure 6.6. The algorithm invokes a forward search from **S** to **F** and stops when the optimal policy is found. The procedure fails if the node **S** or **F** is never found (the algorithm does not assume that these nodes exist). The search often does *not* encompass the entire graph, consequently the auxiliary information of the search is not found. If auxiliary information is important to the application, the procedure can easily be modified to run in either the forward or backward manner, and to run to completion, so that all the nodes are investigated.

Using Dijkstra's dynamic programming algorithm, the example search would proceed as follows. The problem is solved forward starting with node **S** where

g(S)=0. In the first loop of the algorithm, the node S is expanded creating the list of successors M={A,B,D}. S is put on CLOSED. Since the nodes A, B, and D are neither on OPEN nor CLOSED, pointers are directed from these nodes to the start node S, and the costs g(A), g(B), and g(D) are recorded. Nodes A, B, and D are put on OPEN. Figure 6.7 shows these costs labeled above these nodes in parentheses. Next, node A is selected for expansion because this node minimizes the cost g:

$$g(\mathrm{A}) < g(\mathrm{D}) < g(\mathrm{B}).$$

Expanding node A creates the list of successors M={S,B,E}. Node A in put on CLOSED. Node E is neither on OPEN nor CLOSED so a pointer is directed from node E to node A and node E is put on OPEN. Next, node B is already on OPEN, so a decision has to be made to redirect its pointer from S to A. The pointer is redirected because:

$$g(\mathrm{A}) + C(\mathrm{A},\mathrm{B}) < g(\mathrm{B}).$$

The cost g(B)=3 is established because of the pointer modification. Node S is ignored since it is CLOSED.

The algorithm proceeds to expand nodes which minimize the cost function g. The algorithm does not stop, though, when the node F is found (put on OPEN). This is because there may be some other policy that includes node F and has a lower value for g(F) (this is where pointer modifications change g(F)). Finally, when g(F) is the minimum cost value for all OPEN nodes, then no other policy can be created within the graph that has a solution better than the optimal policy currently indicated from S to F. No subpolicy exists starting with a node on OPEN which has a cost lower than g(F), thus the current policy with the cost g(F) must be optimal. The final solution graph is shown in Figure 6.8, with the optimal policy (SABCF) shown as a dashed line.

A shortcoming of Dijkstra's dynamic programming algorithm is that it lacks a mechanism to direct the search toward the finish node F. That is, the technique will explore as many nodes that lead towards the solution as it does that lead away. This is shown in Figure 6.9 with a forward search through a simple grid having unit arc costs. The square nodes are CLOSED while the circled nodes are OPEN. Only the pointers associated with the optimal solutions are shown. Notice that there are many nodes to the left of S that are explored, even though the finish node is to the right. The next section will focus on an algorithm that will be more informed on where to look for the solution during the search.

A General Graph Searching Procedure

Dijkstra's dynamic programming algorithm described in the previous section is a typical uninformed search. Although uninformed search methods provide a technique for solving the path planning problem, they are inefficient if they expand too many nodes before a solution is found: computational time and storage may become excessive. An alternative algorithm which overcomes these deficiences is a heuristic search technique.

Heuristic search techniques use problem dependent information to reduce the number of nodes investigated. Essentially, the heuristic search technique expands nodes based on which node is the most promising to be on the optimal solution path. In order to establish the promise of a node, the evaluation function is introduced.

The *evaluation function* $f(n)$ evaluates the promise that node n is on the optimal solution path. Define the evaluation function to be the estimate of the sum of the cost of the minimal cost path from **S** to n plus the cost of the minimal cost path from n to **F**. Stated in this manner, $f(n)$ is the minimum cost path constrained to go through node n. The evaluation function takes the form:

$$f(n) = g(n) + h(n).$$

The term $g(n)$ is an estimate of the optimal cost of the path from node **S** to node n. A good estimate for $g(n)$ is the cost of the path indicated by summing the arc costs from node **S** to node n directed by the current pointers in the search graph. The term $h(n)$ is called the *heuristic function*. The heuristic function estimates the optimal cost from node n to node **F**.

The evaluation function guides the general graph searching algorithm [45], as shown in Figure 6.10. This algorithm is very similar to Dijkstra's algorithm, except the evaluation function determines the order for which the nodes will be expanded for the general searching algorithm. Notice that Dijkstra's algorithm results from using $h(n)=0$ so that $f(n)=g(n)$ in the general graph searching algorithm.

Some important properties of the GRAPHSEARCH algorithm arise when the heuristic function $h(n)$ is a lower bound to a function $h^*(n)$ which is the *actual* cost of a minimal cost path from node n to node **F** (see Appendix D and [45]). When $h(n)$ is a lower bound to $h^*(n)$, the GRAPHSEARCH algorithm is termed the A^* algorithm. If there exists a path from the start node **S** to the goal node **F** in the search graph, then the A^* algorithm will terminate finding the optimal path from the start node **S** to the goal node **F**. This is termed the *admissibility* property of the A^* algorithm.

Consider the square lattice path planning problem of Figure 6.9 where the minimum distance path is sought. The arcs of the graph each have unit cost representing the distance between neighboring nodes. The evaluation function $f(n) = g(n) + h(n)$ can be formed using the sum of the arc costs from the start node **S** to node n as $g(n)$, and the Euclidean distance from node n to the finish node **F** as the heuristic function $h(n)$. Note that $h(n)$ is a lower bound to $h^*(n)$ so the GRAPHSEARCH algorithm becomes an A^* algorithm and the optimal path will be found.

The search begins by expanding the start node **S**. Figure 6.11 shows the first stage of the search and the values of $f(n)$, $g(n)$, and $h(n)$ for each node. The start node is also labeled with the number 1 to indicate that it is the first node expanded. The algorithm proceeds to expand nodes as explained in the section on Dijkstra's algorithm except on the basis of minimal evaluation function value $f(n)$. Figure 6.12 shows the state of the search when the optimal solutions are found (more than one optimal path exists). The values of $f(n)$, $g(n)$, and $h(n)$ are shown for each node and the order of node expansion is indicated within each node. The order of node expansion shows how the search proceeds toward to finish node **F**. In comparison to the solution shown in Figure 6.9 for Dijkstra's algorithm, the solution from the A^* algorithm expanded far fewer nodes, as shown in Figure 6.13. For this example of the square lattice grid with the Euclidean distance heuristic, it can be shown that the A^* algorithm will expand no more than 18% of the nodes expanded by Dijkstra's algorithm [51].

The A^* algorithm is developed from the same structure of Dijkstra's algorithm; both are applicable to arbitrary search graphs such as the Voronoi diagram graphs presented in Chapter 5. These algorithms will be applied to an example in navigation path planning in the following chapter.

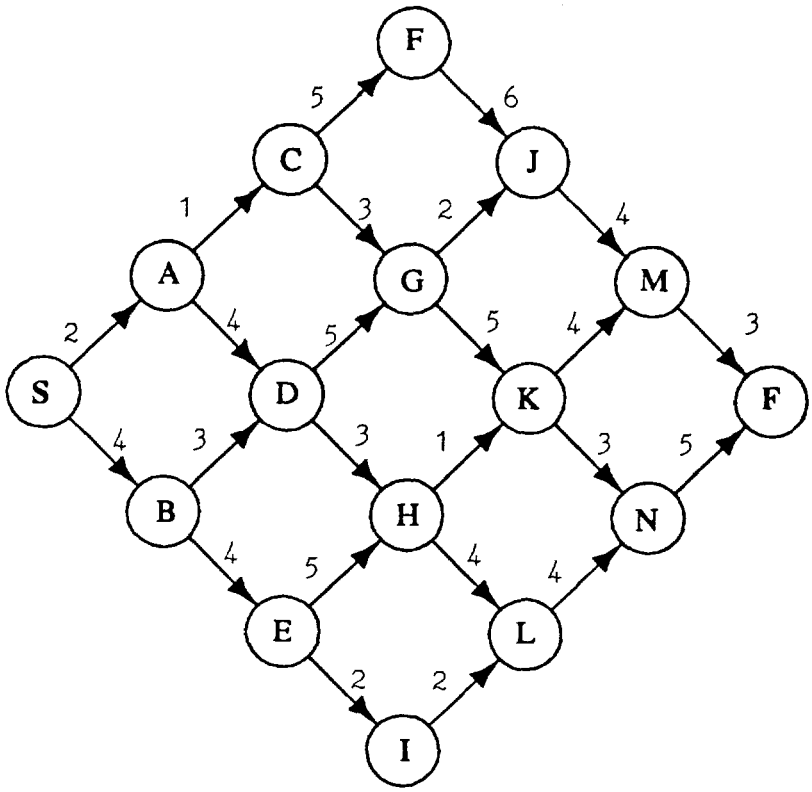


Figure 6.1. A simple grid search graph.

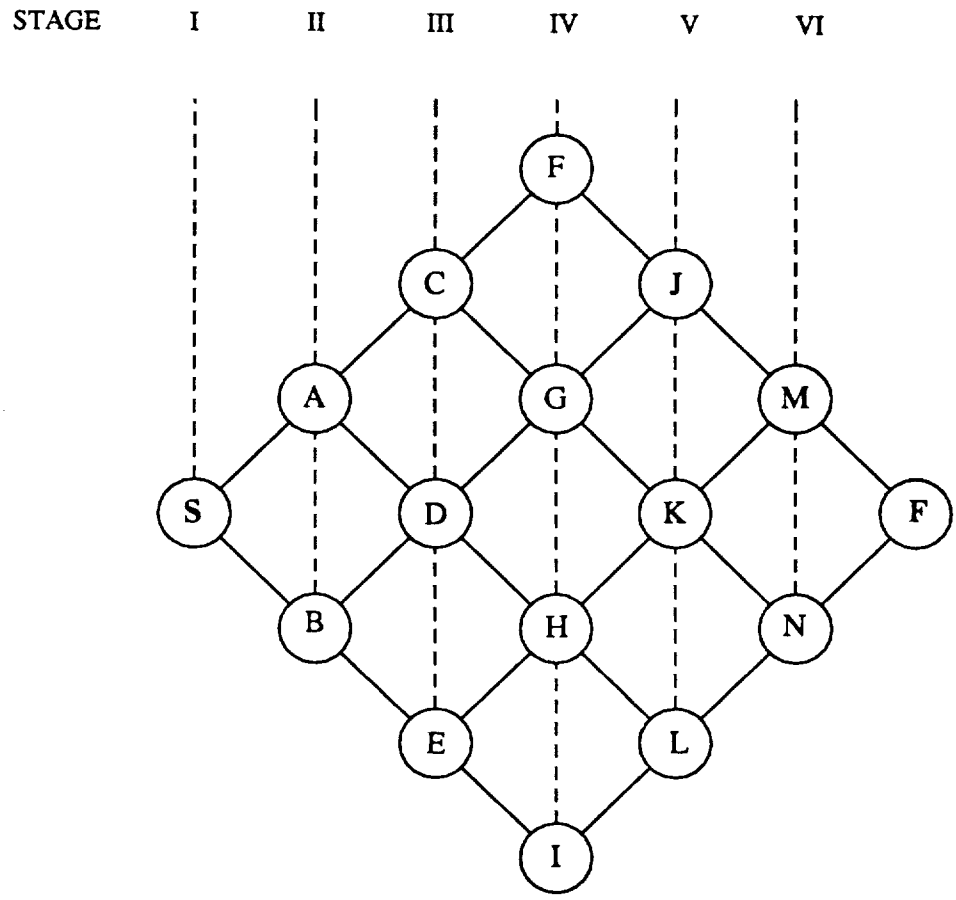


Figure 6.2. A simple grid search graph with stages marked.

C-2

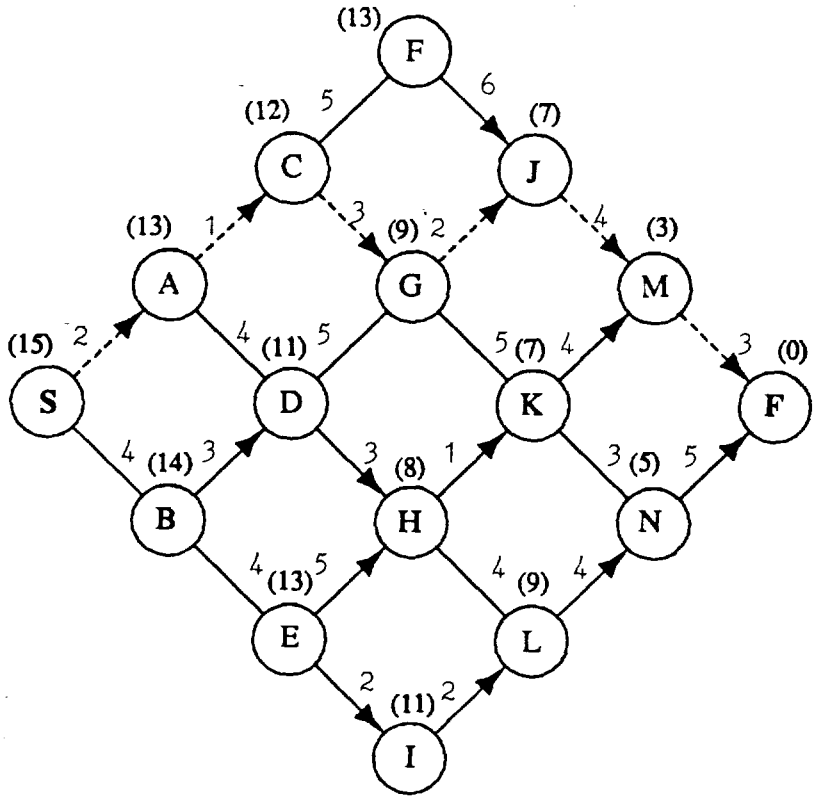


Figure 6.3. The backward solution graph with the optimal solution (dashed line).

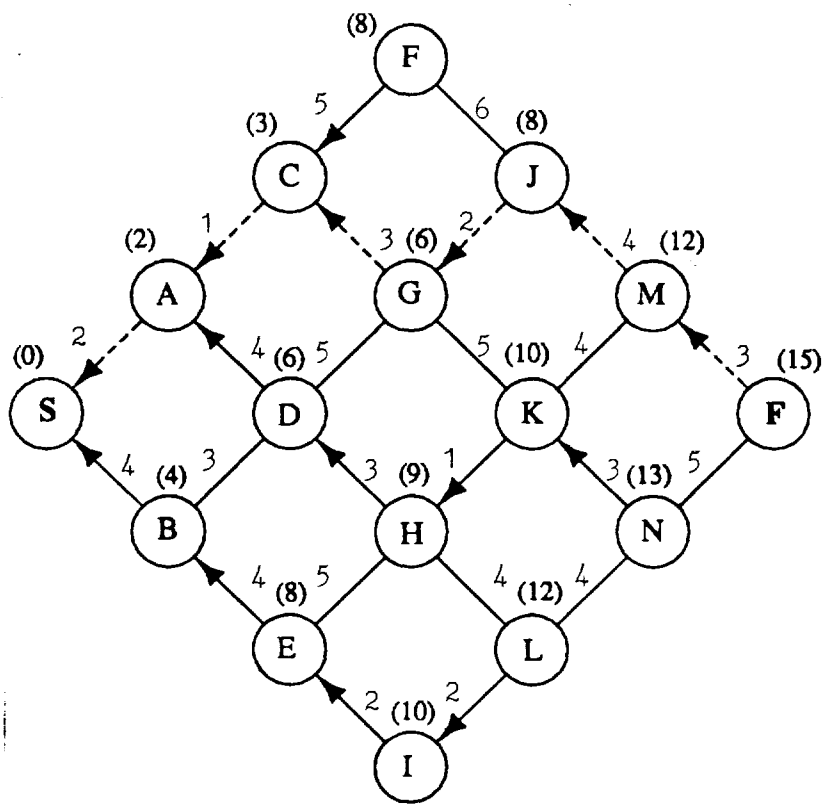


Figure 6.4. The forward solution graph with the optimal solution (dashed line).

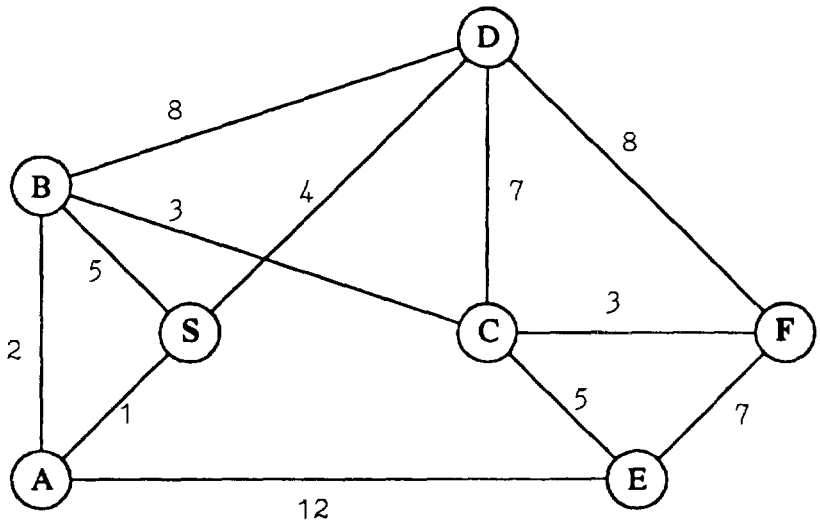


Figure 6.5. An arbitrary search graph.

Procedure DIJKSTRA

1. Given a search graph G put the start node **S** on a list called OPEN. If **S** does not exist, then exit with failure. Establish the cost g (S)=0.

2. Create a list called CLOSED that is initially empty.

3. LOOP: if OPEN is empty, exit with failure.

4. Select the node n on OPEN that minimizes the cost g, remove it from OPEN, and put it on CLOSED.

5. If n is the goal node **F**, exit successfully with the solution obtained by tracing a path along the pointers from **S** to **F** in G. (Pointers are established in step 7.)

6. Expand node n, generating the set M of its successors.

7. For each member m of M that was not already on OPEN or CLOSED, establish a pointer from m to n. Add m to OPEN with the cost

$$g(m) = g(n) + C(n, m).$$

For each member m of M that was already on OPEN, decide whether or not to change its cost $g(m)$ and redirect its pointer from n based on

$$g(n) + C(n, m) < g(m),$$

or if

$$g(n) + C(n, m) = g(m),$$

then establish two pointers.

8. Go LOOP.

Figure 6.6. Dijkstra's dynamic programming algorithm.

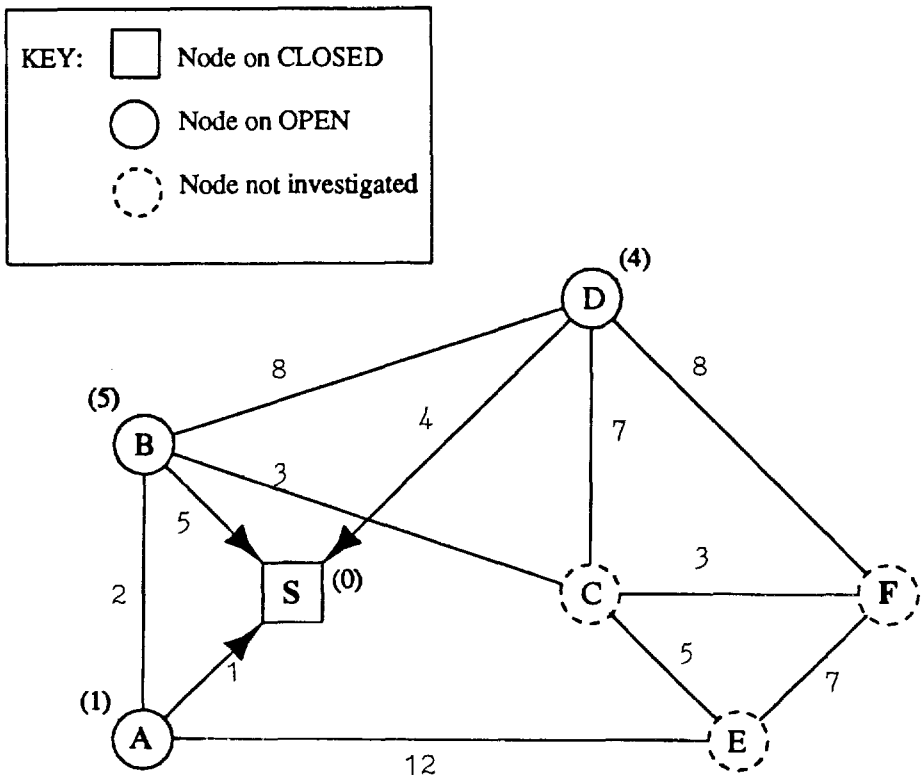


Figure 6.7. The first stage of the search of an arbitrary graph.

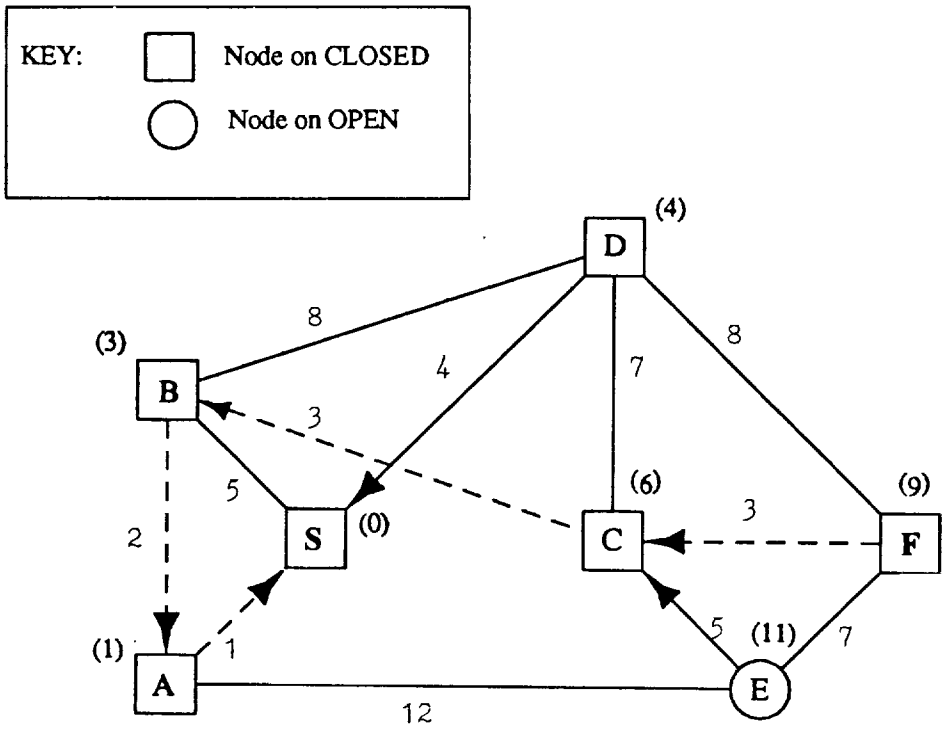


Figure 6.8. The backward solution graph when the optimal policy is found.

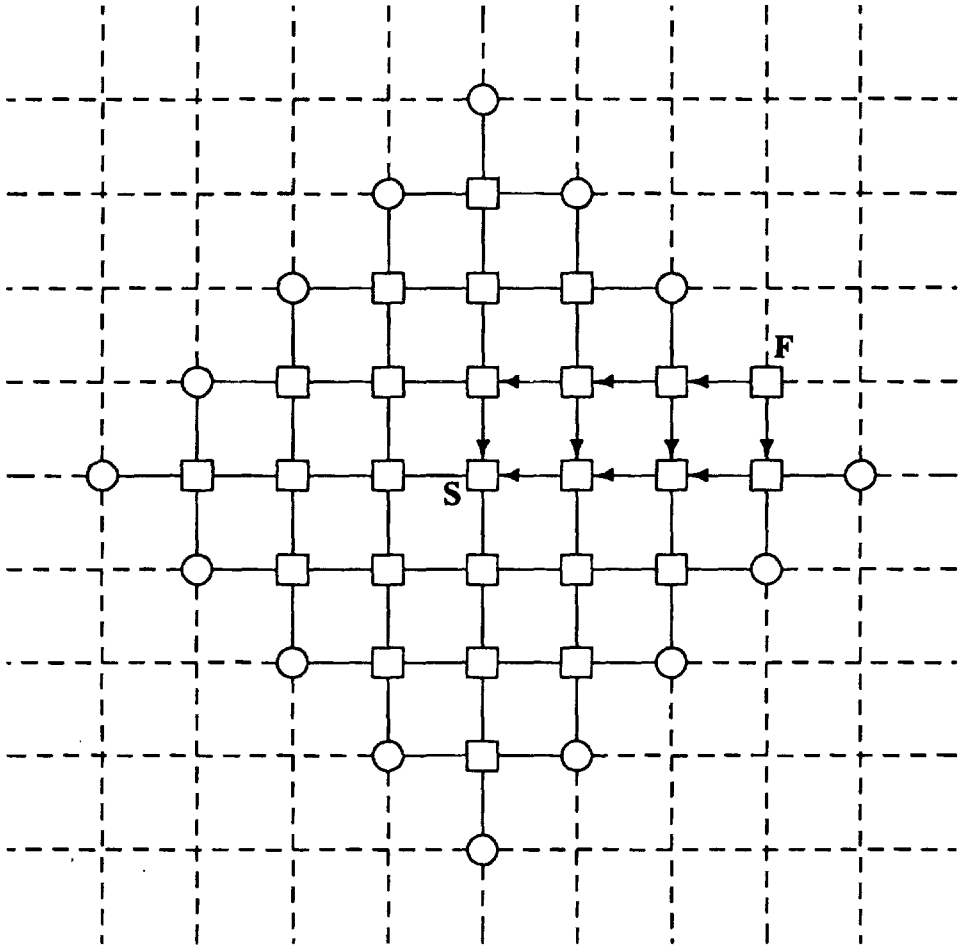


Figure 6.9. An example of an uninformed search.

Procedure GRAPHSEARCH

1. Given a search graph G put the start node **S** on a list called OPEN. If **S** does not exist, then exit with failure. Establish the value f (**S**).

2. Create a list called CLOSED that is initially empty.

3. LOOP: if OPEN is empty, exit with failure.

4. Select the first node on OPEN, remove it from OPEN, and put it on CLOSED. Call this node n.

5. If n is the goal node **F** exit successfully with the solution obtained by tracing a path along the pointers from **S** to **F** in G. (Pointers are established in step 7.)

6. Expand node n, generating the set M of its successors in G.

7. For each member m of M that was not already on OPEN or CLOSED, establish a pointer from n to m. Add m to OPEN with the value

$$f(m) = g(n) + C(n, m) + h(m).$$

For each member m of M that was already on OPEN, decide whether or not to change the value of $f(m)$ and redirect its pointer based on

$$g(n) + C(n, m) < g(m),$$

or if

$$g(n) + C(n, m) = g(m),$$

then establish two pointers.

8. Reorder the list OPEN according to heuristic merit.

9. Go LOOP.

Figure 6.10. A general graph searching procedure.

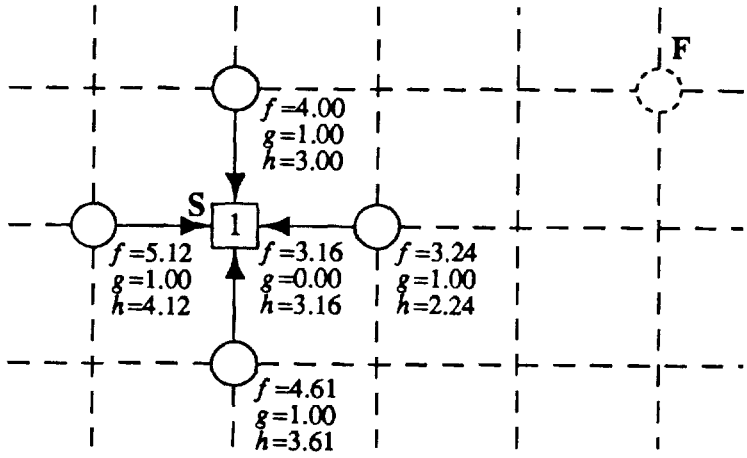


Figure 6.11. First stage of the square grid search.

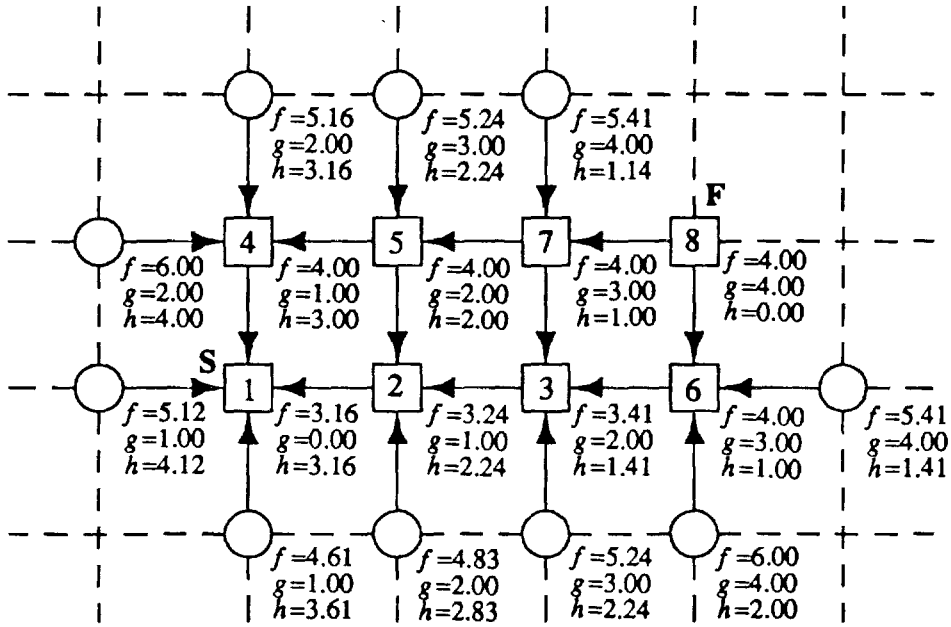


Figure 6.12. Final solution of the square grid search.

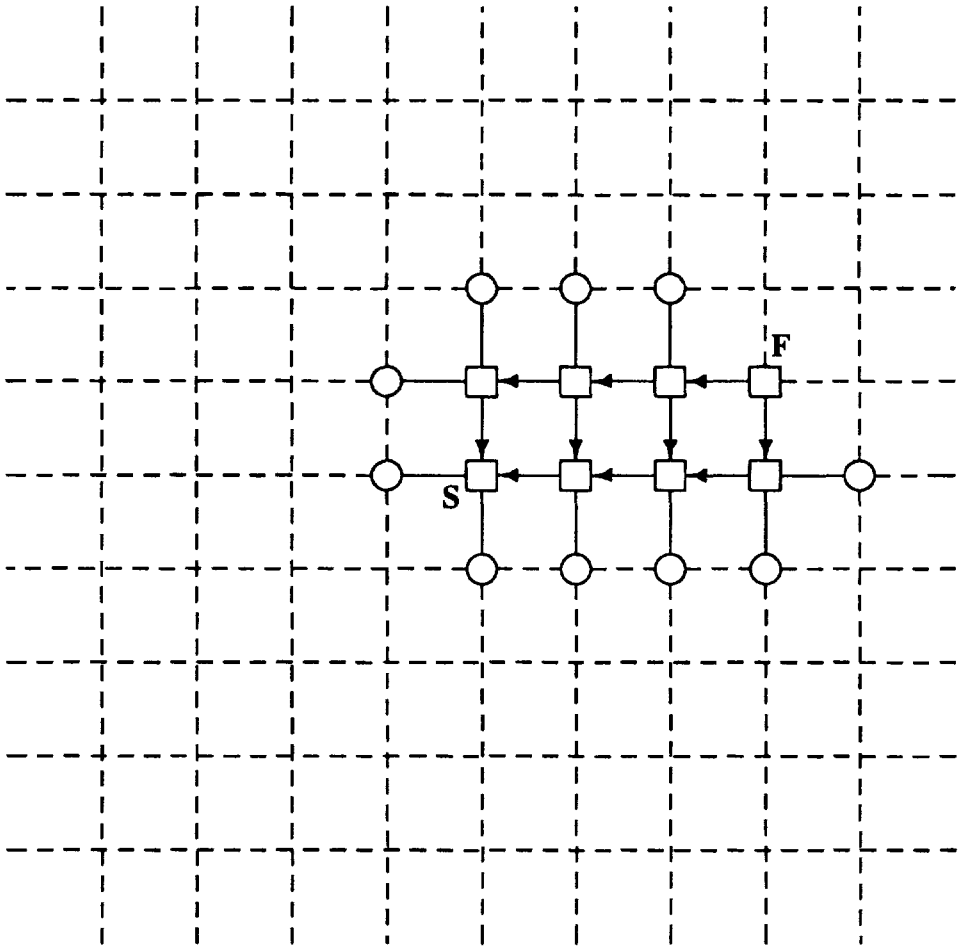


Figure 6.13. An example of an informed search with the A^* algorithm.

CHAPTER 7

NAVIGATION PATH PLANNING EXAMPLES

The following navigation path planning examples use Dijkstra's dynamic programming algorithm and the A^* algorithm as search procedures for Voronoi diagram graphs. Search results will be presented and compared both numerically and graphically.

A Planning Scenario †

It has been determined by a high level global planning system that a path through mountainous terrain with threats is needed to complete a mission plan. A low altitude flight in mountainous terrain is planned so that the aircraft has high masking and minimal exposure of electromagnetic emissions. It is desired to find a constant altitude path from some start location **S** to a finish location **F**. The path should have minimal exposure to threats, should not cross mountain contours at the chosen constant altitude, and should stay within or on the map boundaries.

The Terrain Search Graph

The first step in the modeling of the terrain is to polygonize the terrain contour map. It is important that the polygonization process guarantees that a mountain boundary is contained in the interior of the polygon. This insures that if search graph paths do not cross over polygon edges, then a vehicle traversing these search graph paths will not hit any mountains. The search process will find a path for moving a point aircraft around polygon mountains. This assumes that the dimensions of the

† This path planning scenario may apply to a helicopter or an aircraft. Furthermore, instead of mountainous terrain, a planning scenario could also include planning nap of the earth trajectories.

aircraft are negligible in comparison to the distances between neighboring mountains. If the aircraft dimensions are not negligible, then the polygon mountains can be expanded to account for the aircraft's wing span, or some other characteristic dimension.

The polygons of the terrain map define a set of vertices suitable for the contour vertex point method of generating a search graph. Consider the set of 24 polygons from Figure 7.1 which represent mountains at a constant altitude. These polygons generate the Voronoi search graph shown in Figure 7.2. Graph arcs from map boundaries eliminate any arcs from infinite Voronoi edges. Also, the start and finish nodes do not fall on the Voronoi diagram, so these nodes are added to the graph by connecting them to the closest Voronoi point nodes. Notice that the Voronoi search graph is a feasible search graph allowing for paths between all mountains (see Appendix C for a proof of the feasibility of search graphs generated with the contour vertex point method).

The next step is to assign costs to the arcs of the search graph. The cost is composed of the length of the segment between search nodes plus an additional positive cost if the Voronoi edge crosses over a threat region. The additional threat cost takes into account the relative importance in weight between distance traveled and threat exposure. The cost assignment establishes a positive cost to each arc in the graph so that the monotone restriction for the A^* algorithm is established (see Appendix D). The weighting of these costs will be discussed in the examples where threats are part of the environment.

The Search Algorithms

Two methods for performing the search are used: Dijkstra's dynamic programming algorithm and the A^* algorithm. The cost function $g(n)$ is the total arc cost from the start node **S** to the current node n. Because the monotone restriction is satisfied, the value established for $g(n)$ when node n is chosen for expansion in Dijkstra's algorithm and the A^* algorithm is the optimal value for the path from the start node **S** to the node n. Dijkstra's algorithm expands nodes on OPEN that have the least value of $g(n)$, as described in Chapter 6. The A^* algorithm combines the cost function $g(n)$ with a heuristic function $h(n)$ to arrive at the evaluation function $f(n)$ which determines the node that is expanded next in the search. The heuristic function used in the following examples is the Euclidean distance from the node n to the finish node **F**. This is an admissible heuristic, and thus, the results of Dijkstra's algorithm and the A^* algorithm will be the same optimal result.

While both Dijkstra's algorithm and the A^* algorithm guarantee finding an optimal path, they differ in the amount of work they perform in searching the graph. Comparing parameters associated with Dijkstra's algorithm and the A^* algorithm shows a difference in search effort. The following parameters can be recorded for comparison: the number of nodes on OPEN when the optimal solution is found, the number of nodes on CLOSED when the optimal solution is found, the total number of nodes investigated (equal to the sum of the nodes on OPEN and the nodes on CLOSED), the number of nodes uninvestigated, and the number of pointers assigned to the graph arcs during the search. Other useful comparison quantities are the ratio of the total number of nodes investigated by the A^* algorithm to the total number of nodes investigated by Dijkstra's algorithm, denoted R_N, and the ratio of the number of pointers assigned to graph arcs by the A^* algorithm to the number of pointers assigned to graph arcs by Dijkstra's algorithm, denoted R_P. Since the A^* algorithm is more informed than Dijkstra's algorithm (which is uninformed), then the R_N ratio will *not* exceed unity. Also, since both algorithms may reassign a different number of pointers during a search, then the R_P ratio *may* exceed unity. Nonetheless, for the purpose of comparison, as R_N and R_P approach unity, the amount of effort for a given search using the A^* algorithm approaches the effort using Dijkstra's algorithm.

The comparison of these algorithms shown through numerical quantities can be enhanced by a graphical comparison. The search arcs for which pointers are assigned during a search will be shown for each of the search algorithms. From these graphs, additional information on which part of the search space was not investigated and where the OPEN search graph arcs are located can provide further comparison information not indicated by the numerical parameters.

Example 1: A Minimum Distance Path

The Voronoi diagram search graph of Figure 7.2 is searched to find the minimum distance path from the start node **S** to the finish node **F** with no threats in the environment. The optimal path plan is shown in Figure 7.3. This optimal path is found by both Dijkstra's dynamic programming algorithm and the A^* algorithm.

For comparison, the numerical results of the searches are shown in Table 7.1. The A^* algorithm investigates 33% of the total nodes in the search graph, while Dijkstra's algorithm investigates 79%. It is clear from all of the parameters, especially R_N=.41 and R_P=.37, that the A^* algorithm uses less effort in finding the optimal solution. This is the result of the heuristic used which guides the A^* solution towards the finish node **F**.

A graphical presentation of the search efforts of Dijkstra's algorithm and the A^* algorithm is shown in Figure 7.4 and Figure 7.5. One can see how the A^* algorithm is guided to the final node **F** by the Euclidean distance hueristic. Dijkstra's algorithm searches many more nodes further away from the finish node, including many of the boundary arcs. In fact, if the boundary arcs were removed and more mountains were introduced around this map, Dijkstra's algorithm would probably include still more arcs in the solution, while the A^* algorithm would remain the same as shown.

Example 2: A Terrain Environment with Threats

In this example, the Voronoi diagram search graph for the terrain/threat environment of Figure 7.6 is searched to find a path from the start node **S** to the finish node **F**. The dashed circular regions indicate threat regions. A solution path should have minimal exposure to threats. To account for this, arcs of the search graph are assigned a threat cost in addition to the length cost. For this example, the cost for traversing an arc that is in a threat region is ten times the cost for traversing an arc of the same length in a non-threat region. This level of danger is indicated in parenthesis as (10×) in Figure 7.6.

Using the search graph of Figure 7.2 and the above convention for assigning arc costs, the optimal path plan is shown in Figure 7.7. In general, the solution path is the minimum distance path clear of threats. Initially, the path leads toward a threat region, but turns away before entering the threat region. This is not the effect of the search algorithm making a last minute adjustment due to the threat, rather, the Voronoi diagram search graph has a node at this location which allows for two path choices: to proceed through the threat region or to turn away. If the mountain polygonization were performed differently, or if the threat region were larger, then the Voronoi diagram search graph may have represented this branching within the threat region, since there is no threat location information used in the construction of the Voronoi diagram search graph. For this example a better path plan may be found *near* the solution presented in Figure 7.7, possibly with more clearance of the threat region. Arriving at such a solution will be discussed later.

For comparison, the numerical results of the searches are shown in Table 7.2. Once again, it is clear from all of the parameters that the heuristic of the A^* algorithm allows for less effort in finding the optimal solution. However, compared to the previous example, the difference in search effort is less pronounced, as indicated from R_N=.72 and R_P=.69 . This can be explained by the lack of a heuristic for threat information. Such a hueristic would be difficult to establish, and would probably be

inadmissible. Neither of the search algorithms have any heuristic guidance informing them which paths may lead toward threat regions, and the result is that both algorithms search paths that lead directly into threat regions.

A graphical presentation of the search efforts of Dijkstra's algorithm and the A^* algorithm is shown in Figure 7.8 and Figure 7.9. From these two figures, one can see how the A^* algorithm is guided to the final node **F** by the hueristic. It can also be seen that the heuristic keeps the search of the A^* algorithm away from the boundary arcs. Once again, Dijkstra's algorithm searches many more nodes further away from the finish node, including many of the boundary arcs.

In comparison to the previous example that had no threats, more nodes are investigated by the A^* algorithm when threats are included in the search graph, but according to the numerical results, the number of nodes investigated by Dijkstra's algorithm remains about the same. Comparing Figure 7.4 and Figure 7.8 indicates that Dijkstra's algorithm expands different nodes for the graph with threats, even though the number of nodes investigated is about the same as the case of the environment free of threats. The fact that Dijkstra's algorithm investigates about the same number of nodes should not be typical of all path planning examples. One would expect that more nodes would be searched by both algorithms when more threats are introduced.

Example 3: A Terrain Environment with a Barrier of Threats

In this example, the Voronoi diagram search graph for the terrain/threat environment of Figure 7.10 is searched to find a path from the start node **S**, to the finish node **F**. The dashed circular regions indicate threat regions, which create a barrier of threats between the start and finish nodes. In the threat regions, the arcs of the search graph are assigned a threat cost in addition to the length cost, which is either three times the cost or ten times the cost for traversing an arc of the same length in a non-threat region. These levels of danger are indicated in parenthesis as (3×) and (10×) in Figure 7.10. In addition to these threat regions, a constraint is added so that the solution path does not use the boundary arcs to pass around the barrier of threats. This is established by assigning a cost of 100 times the length cost for using an arc on the boundary.

Using the search graph of Figure 7.2 and the above convention for assigning arc costs, the optimal path plan is shown in Figure 7.11. Since no path free of threats exists, the solution path must penetrate the barrier of threats in the least dangerous

region. The optimal path found by both Dijkstra's dynamic programming algorithm and the A^* algorithm is a path that penetrates two low weighted (3×) threat regions. A comparison of the search efforts follows.

For comparison, the numerical results of the searches are shown in Table 7.3. R_N=.94 and R_P=.94 indicate that both algorithms exert approximately equal effort in searching for the optimal solution. The A^* algorithm searched 75% of the nodes in the search graph and Dijkstra's algorithm searched 79%. While the heuristic of the A^* algorithm allows for slightly less effort, the lack of a heuristic involving the threat information makes it search as many arcs as Dijkstra's algorithm. The lack of a heuristic for threat information in a high threat environment leads to additional work in the search for a solution.

The above comparison indicates that the two algorithms perform quite equally in high threat environments. This is actually misleading. Figure 7.12 and Figure 7.13 show that almost the entire search space, with the exception of the boundary arcs, was searched by both algorithms. If the search graph were not limited to within the search boundary, then as seen by the previous examples, the heuristic for path length would once again guide the A^* algorithm search from staying too far from the region around the finish node.

The important parameter to note for this example is the total number of nodes investigated and the total number of pointers used in each of the searches. This should be compared to the previous search example where fewer threats were present. The comparison shows that more nodes had to be investigated and more pointers were used when searching a more complex threat environment. (Also consider that the boundary arcs of this example were heavily weighted to keep them from being chosen for a path solution.) In this example, when searching for a low cost path free of threats, the search runs into many paths that enter threat regions. The search proceeds to search around threat regions for a path free of threats. However, no threat-free path exists, so in the process of trying to look for a low cost path, many search arcs away from the finish node are investigated. Because the search was limited to within the boundary arcs, neither algorithm was allowed to stray away from the immediate vicinity of the start and finish nodes. As noted previously, if a larger map was used with more mountains, the A^* algorithm would show less nodes investigated, namely, nodes far away from the start and finish nodes.

This example also brings about a discussion on the boundary limits of a search graph. Reviewing the optimal path as presented in Figure 7.11, one may conclude that the initial part of the solution path is not a good solution, as illustrated in Figure 7.14. Three paths, path A, path B, and path C, depict paths from the start location S to an intermediate location T. Referring back to Figure 7.2, only path A and path C

are depicted in the Voronoi diagram search graph. Consequently, the optimal solution includes path A which is shorter than path C. However, path B, which is not represented by the Voronoi diagram search graph, is actually a shorter path! The reason why there is not a good representation of a path similar to path B, rather than path C, is that this path is near the boundary limits. The Voronoi edges that form the arcs that constitute path C are from points that are on or very near to the convex hull of the set of Delaunay points that are used to construct the Voronoi diagram. In the Voronoi diagram, Voronoi edges associated with Delaunay points on the convex hull extend infinitely. Boundary arcs are used to bound these infinite length Voronoi edges and to allow for paths that lead around the outsides of the mountains. However, path C, which is created by boundary arcs and Voronoi edges associated with points near the convex hull, is not representative a good path from the start location S to the intermediate location T. In terms of modeling good paths around mountains, this suggests that the Voronoi diagram search graph does a poor job in modeling paths near map boundary limits. Qualitatively analyzing the Voronoi diagram search graph of Figure 7.2 supports this statement. Even with this attribute, the Voronoi diagram search graph can still be used to represent good paths around mountains provided that the optimal path is not close to the convex hull described above. In terms of the terrain model, this means that the Voronoi diagram search graph should be constructed using mountains that are on all sides of the start and finish locations, so that the start and finish locations are not near the convex hull of the vertices of the mountains.

Additional Optimization Parameters

The above examples were primarily concerned about minimizing a cost that was a combination of length and threat costs. Additional optimization parameters can be weighted and added to the arc costs as appropriate for the problem. However, as shown by the above examples, when additional parameters are introduced and no corresponding heuristic can be formulated, the search efforts of both Dijkstra's algorithm and the A^* algorithm are increased.

Solutions Near the Voronoi Diagram Optimal Path

The method for establishing a path through the terrain environment using Voronoi diagram search graphs gives a good plan that determines which passageways to proceed through from the start point to the finish point. The next step in path

planning may be to consider solutions near the Voronoi diagram optimal path.

One possible procedure for searching for path plans near the Voronoi diagram optimal path is to search a grid search graph in the region around the Voronoi diagram optimal path. Figure 7.15 shows a region around the Voronoi diagram optimal path for the path from S to F. A fine grid may be used within this region to search for further path plans. The spacing for such a grid should be sufficient to generate a final path plan that defines waypoints at intervals within the expected range of the pilot system.

Another possible procedure for searching for path plans near the Voronoi diagram optimal path is to use an iterative improvement procedure. Since the Voronoi points are unequally spaced, additional points would have to be placed along the solution path at evenly spaced intervals. Then, each node would be allowed to vary in a motion restricted perpendicular to the line between the preceding and following nodes. An example of this approach is shown in Figure 7.16 for the path from S to F in the environment free of threats. Such a procedure would resemble Thorpe's [59] relaxation step in the path relaxation method of path planning (see Chapter 3).

The search performed at this level should result in a sequence of waypoints that are directed to a pilot system for the synthesis of the actual flight. The more detailed searches suggested above should not be of such fine detail that they create a discrete representation of the actual path trajectory, rather, sufficient spacing between grid points should be used to assign waypoints compatible with the input expectations of the pilot system. Furthermore, since the search performed will be at a finer level than the general Voronoi diagram search, further information may be included in the cost function. For example, the cost function may account for the aircrafts performance capabilities or limitations. Large changes in heading angle may be punished, as well as paths that come too close to mountain boundaries.

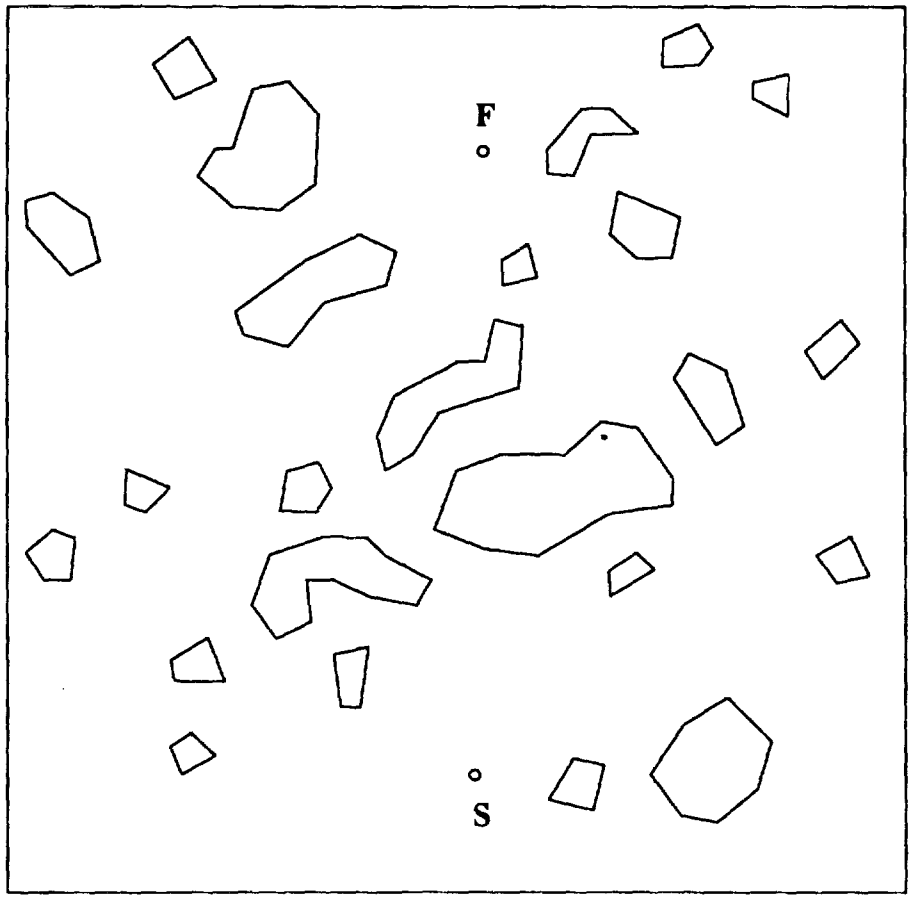


Figure 7.1. A terrain environment free of threats.

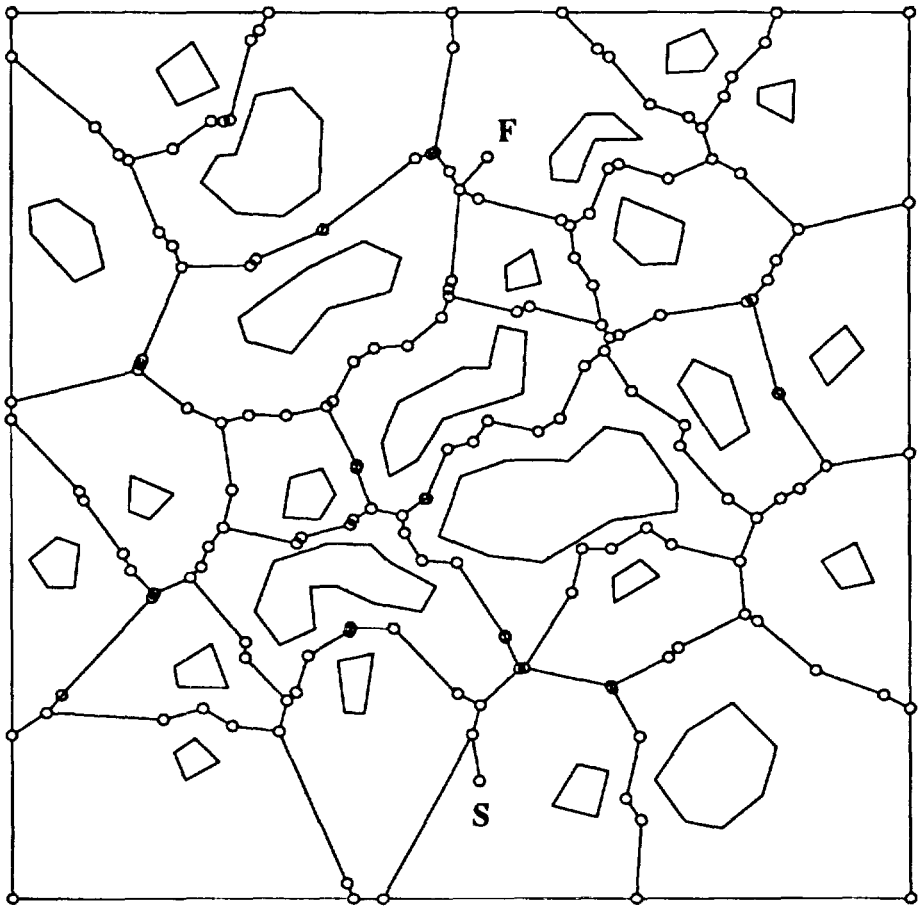


Figure 7.2. A Voronoi search graph. The polygon vertices for 24 mountains define the Delaunay points for the vertex point method that generates this graph.

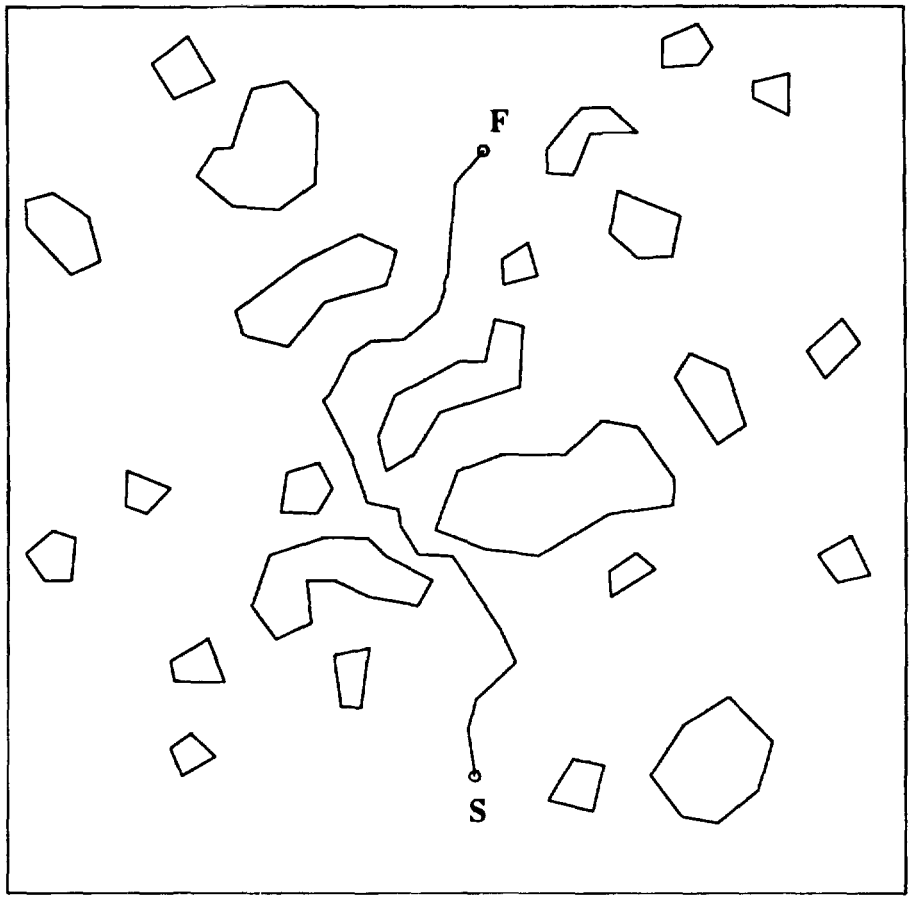


Figure 7.3. The shortest distance path for the Voronoi search graph.

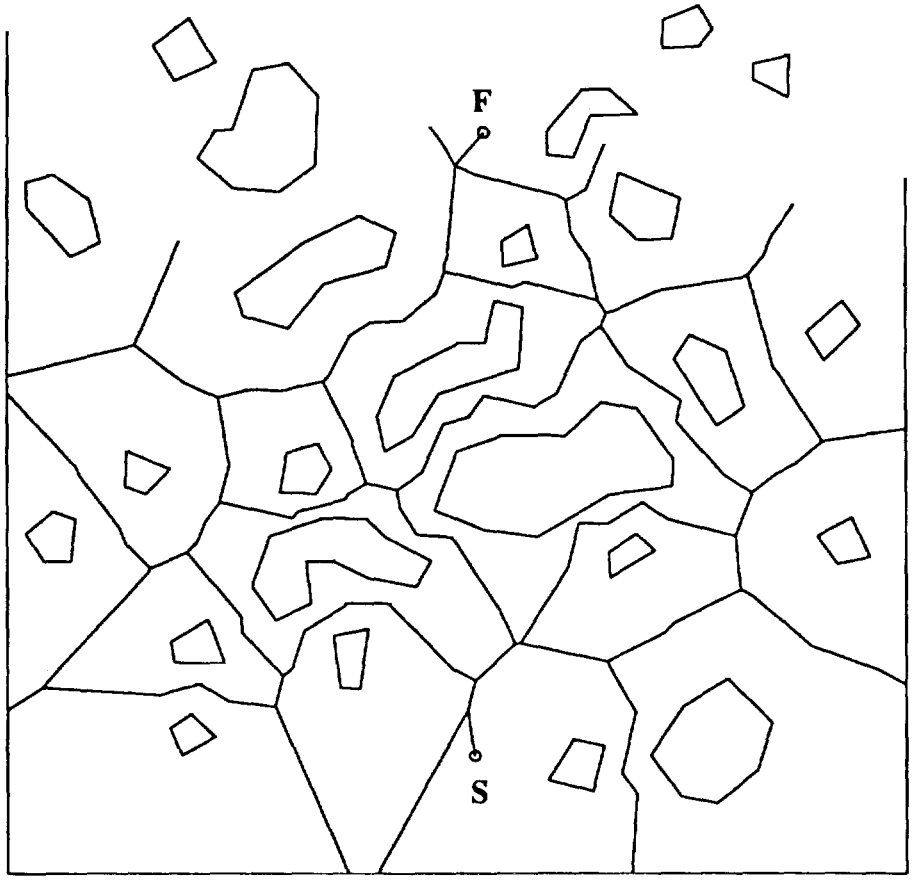


Figure 7.4. The Voronoi search graph arcs searched by Dijkstra's algorithm for the shortest distance path.

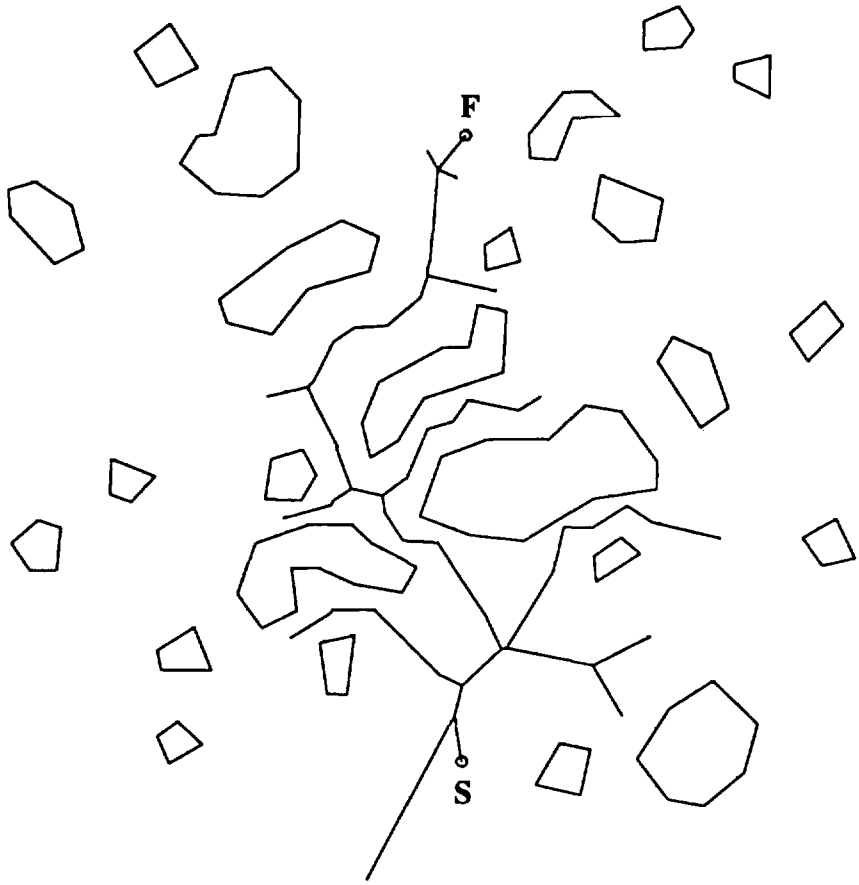


Figure 7.5. The Voronoi search graph arcs searched by the A^* algorithm for the shortest distance path.

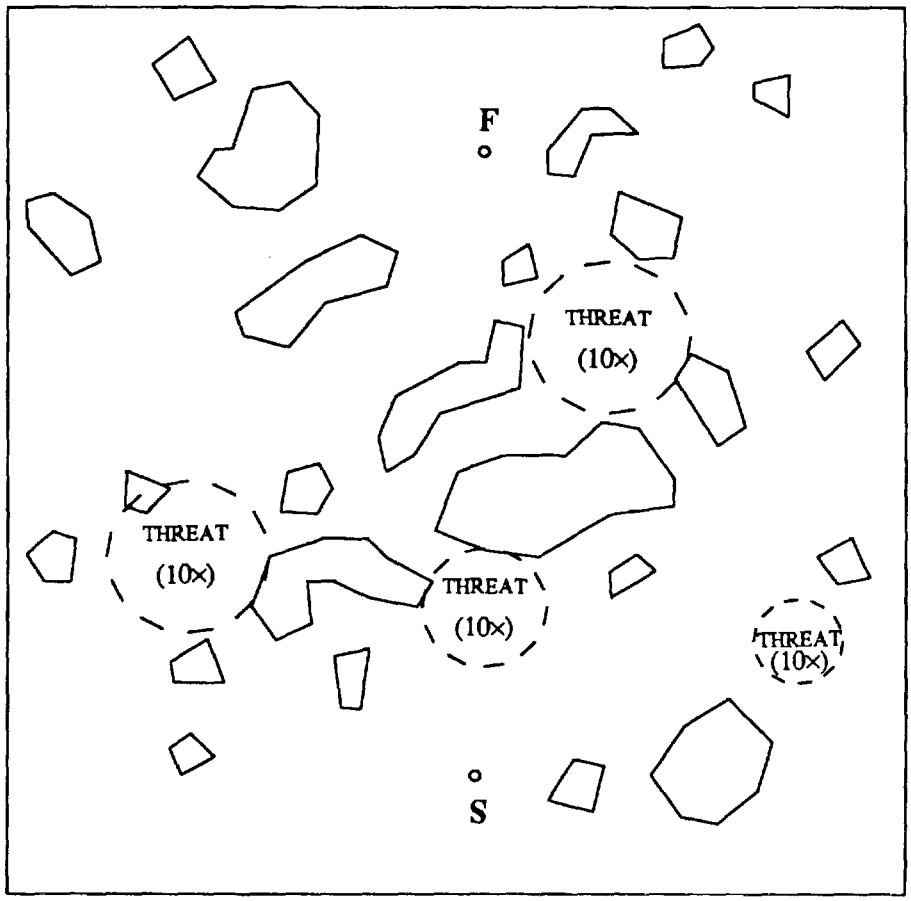


Figure 7.6. A terrain environment with threats.

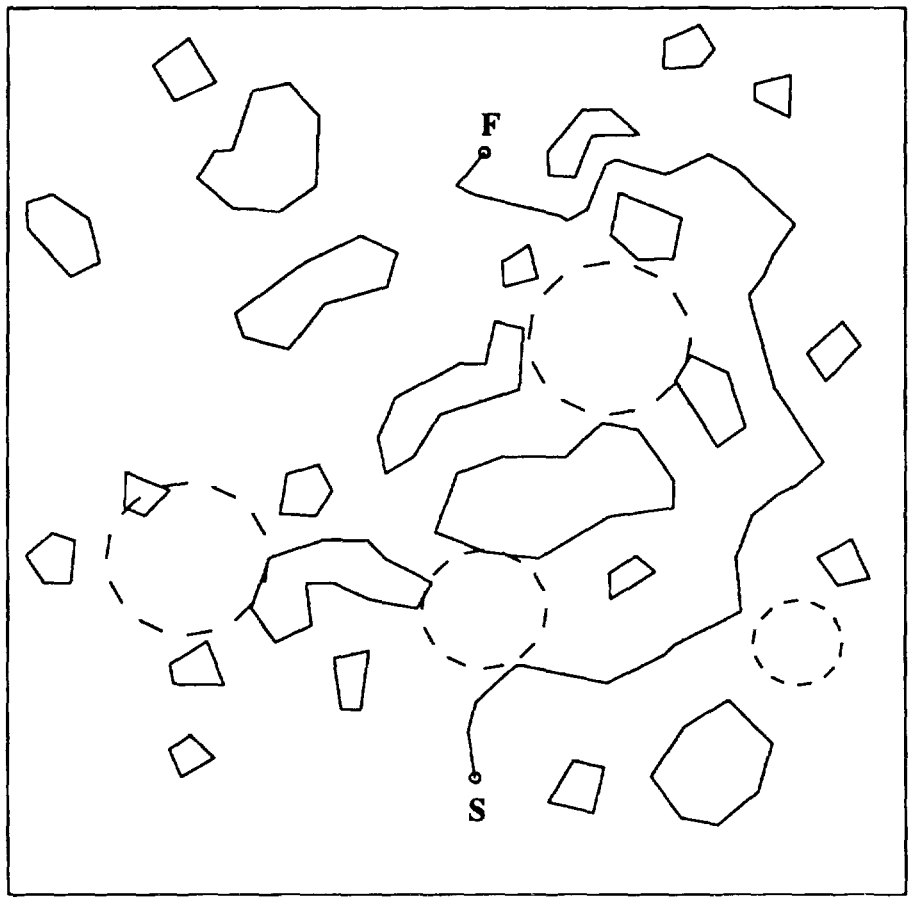


Figure 7.7. The optimal path for the Voronoi search graph with length and threat costs.

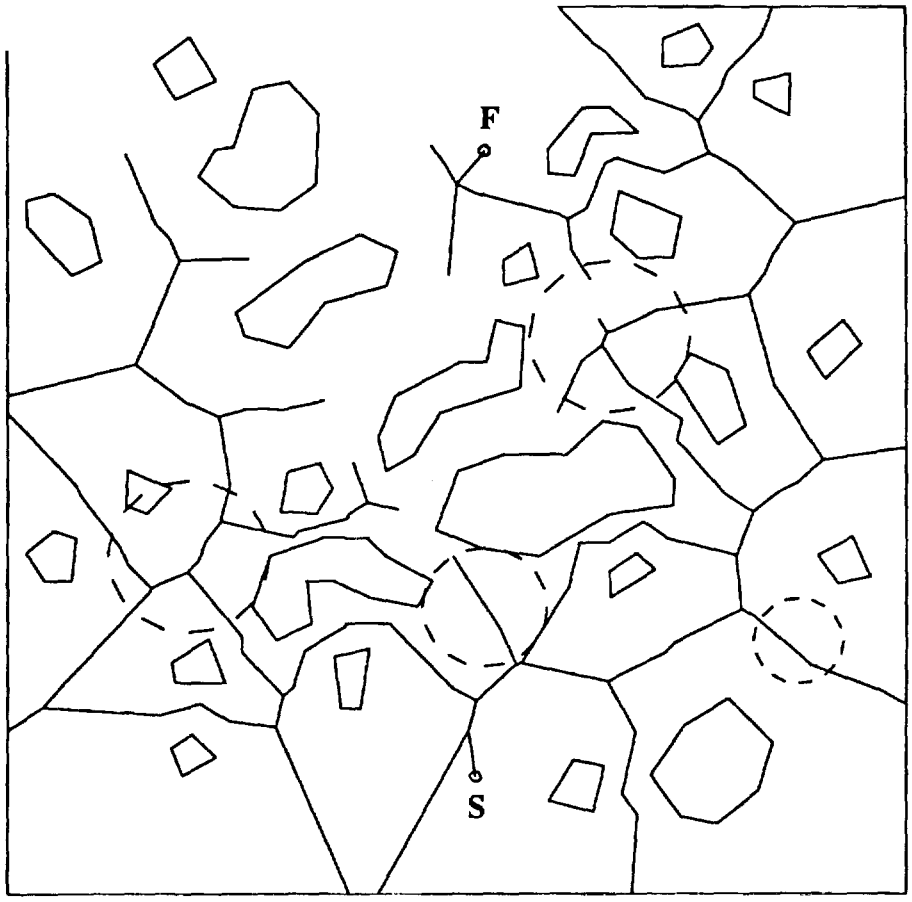


Figure 7.8. The Voronoi search graph arcs searched by Dijkstra's algorithm for the terrain environment with threats.

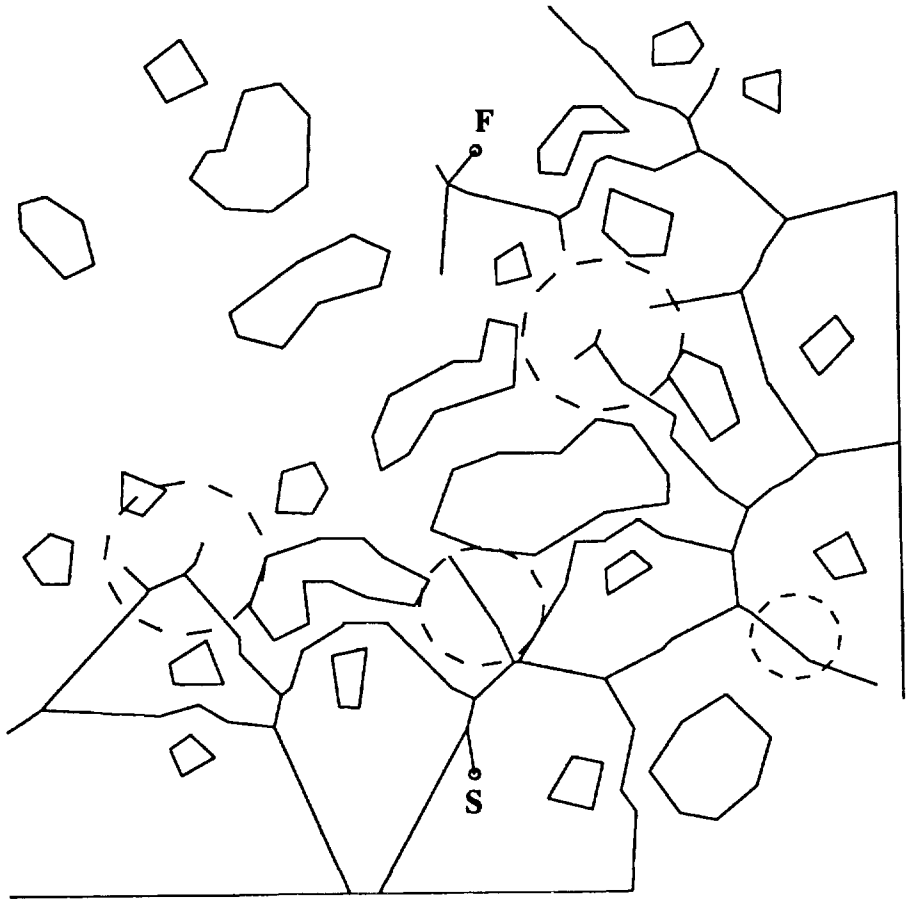


Figure 7.9. The Voronoi search graph arcs searched by the A^* algorithm for the terrain environment with threats.

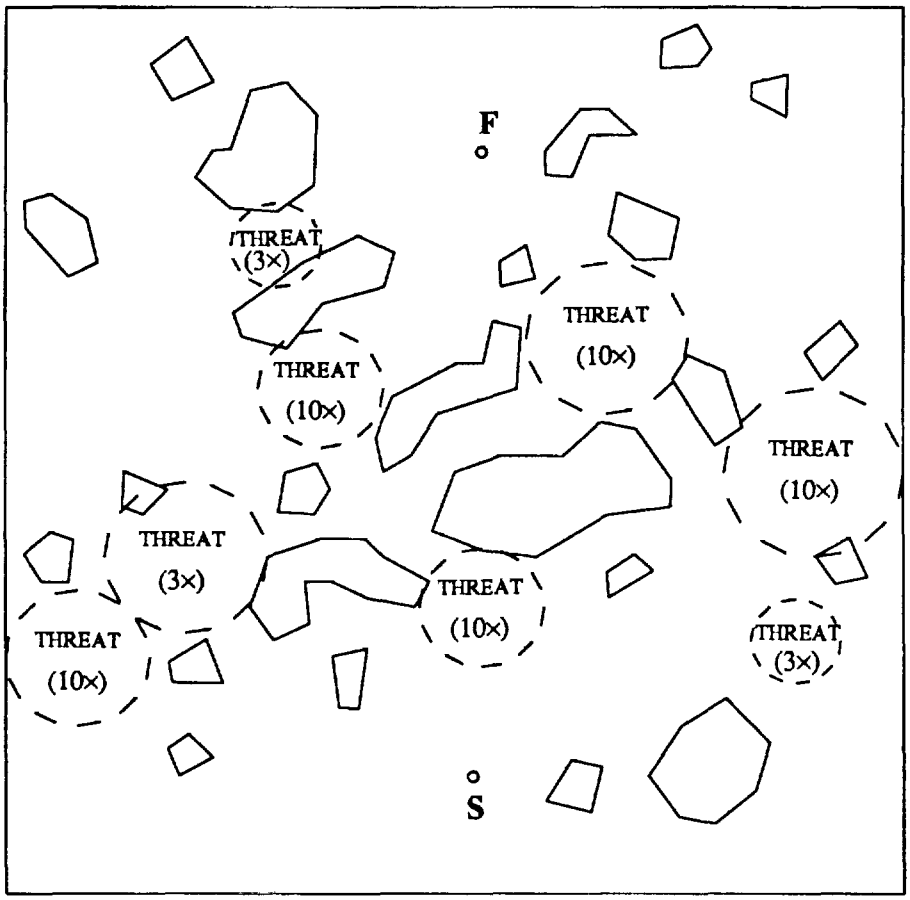


Figure 7.10. A terrain environment with a barrier of threats.

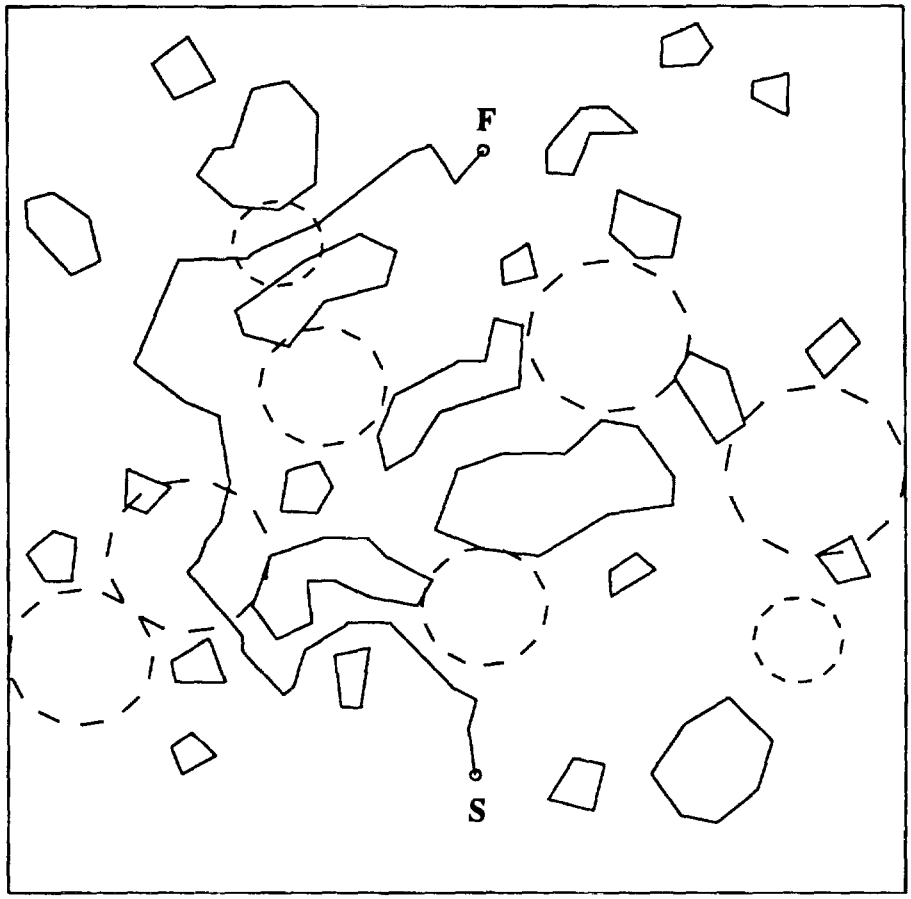


Figure 7.11. The optimal path for the Voronoi search graph with a barrier of threats.

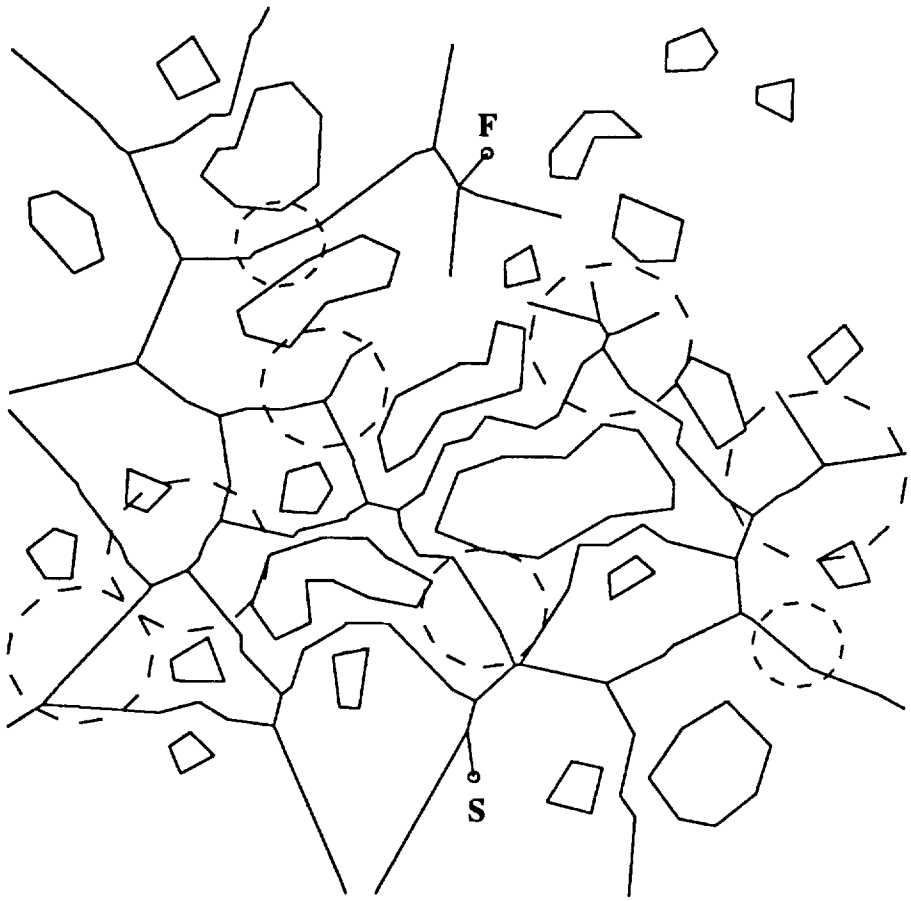


Figure 7.12. The Voronoi search graph arcs searched by Dijkstra's algorithm for the terrain environment with a barrier of threats.

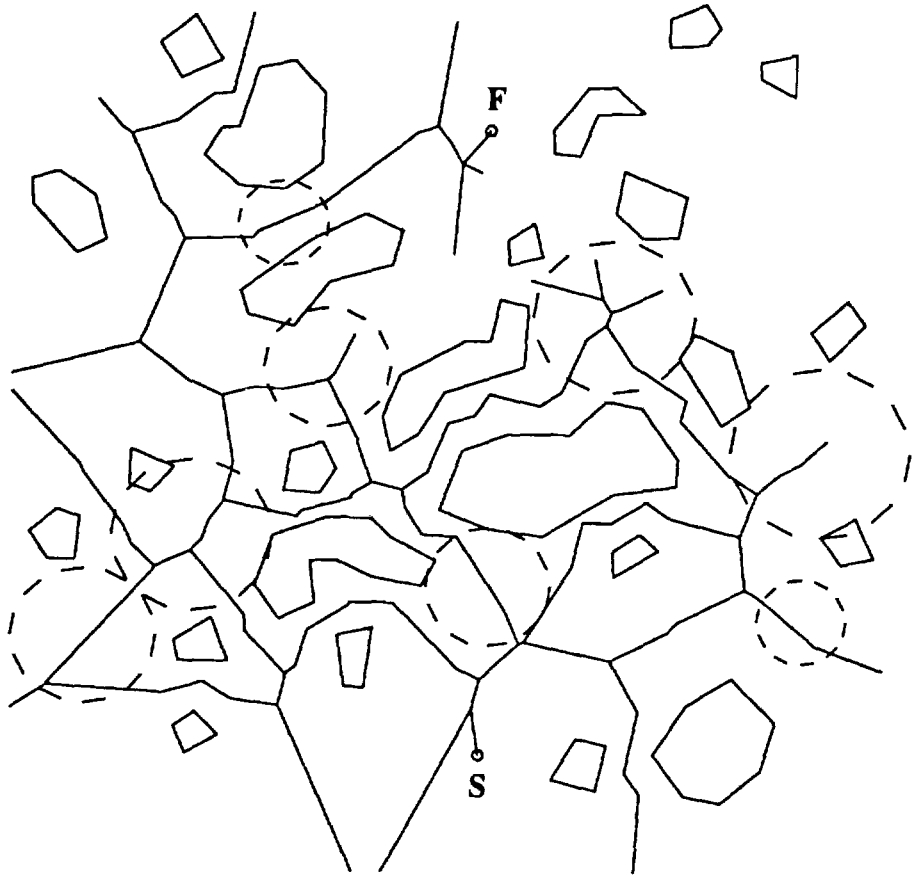


Figure 7.13. The Voronoi search graph arcs searched by the A^* algorithm for the terrain environment with a barrier of threats.

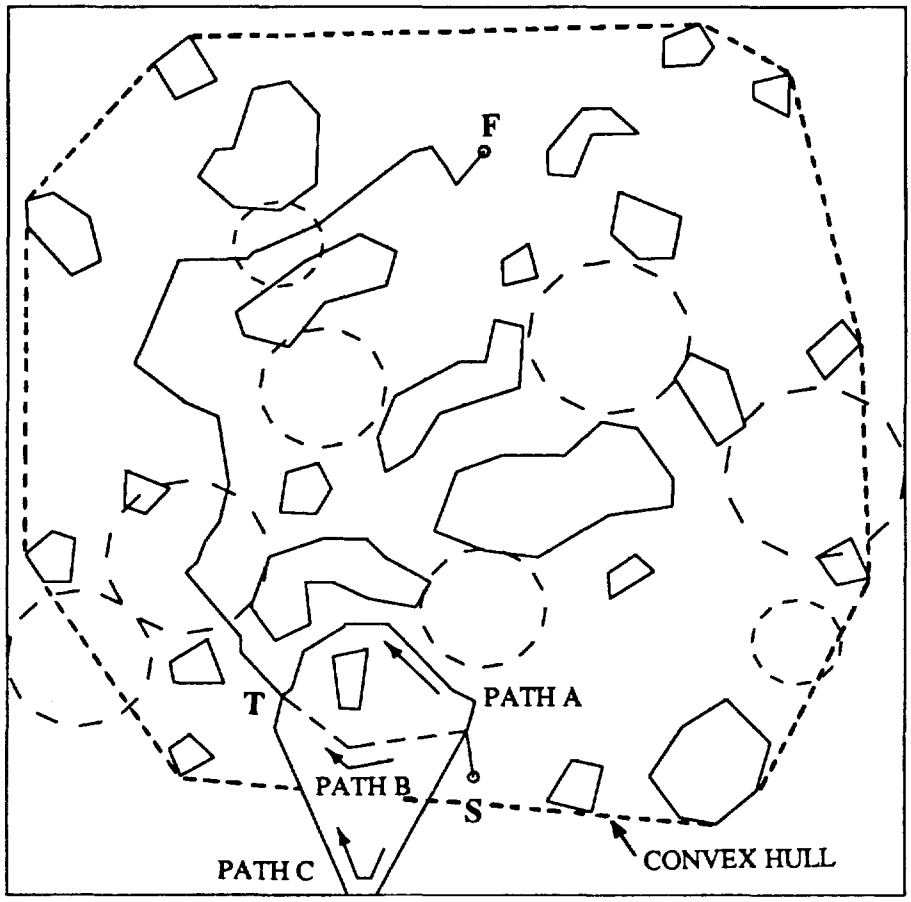


Figure 7.14. Only path A and path C are depicted in the Voronoi diagram search graph. However, path B (dashed lines) is shorter than both path A and path C.

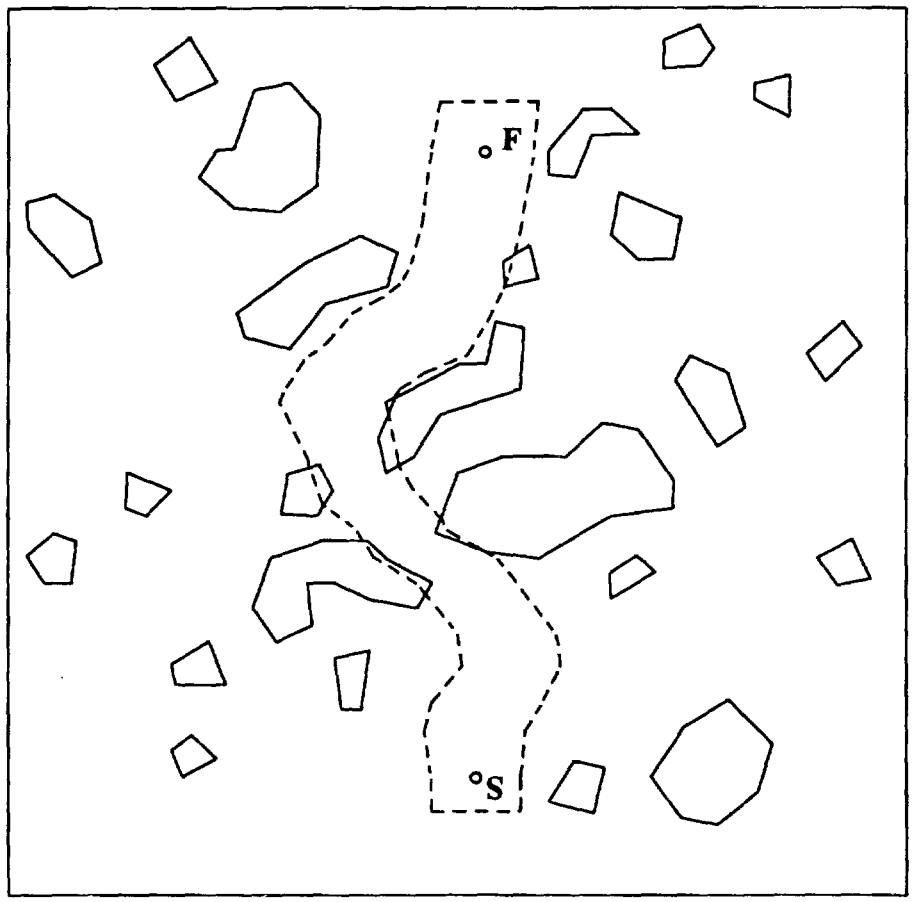


Figure 7.15. A region around the optimal solution from the Voronoi diagram search graph may be considered for further investigation path plans.

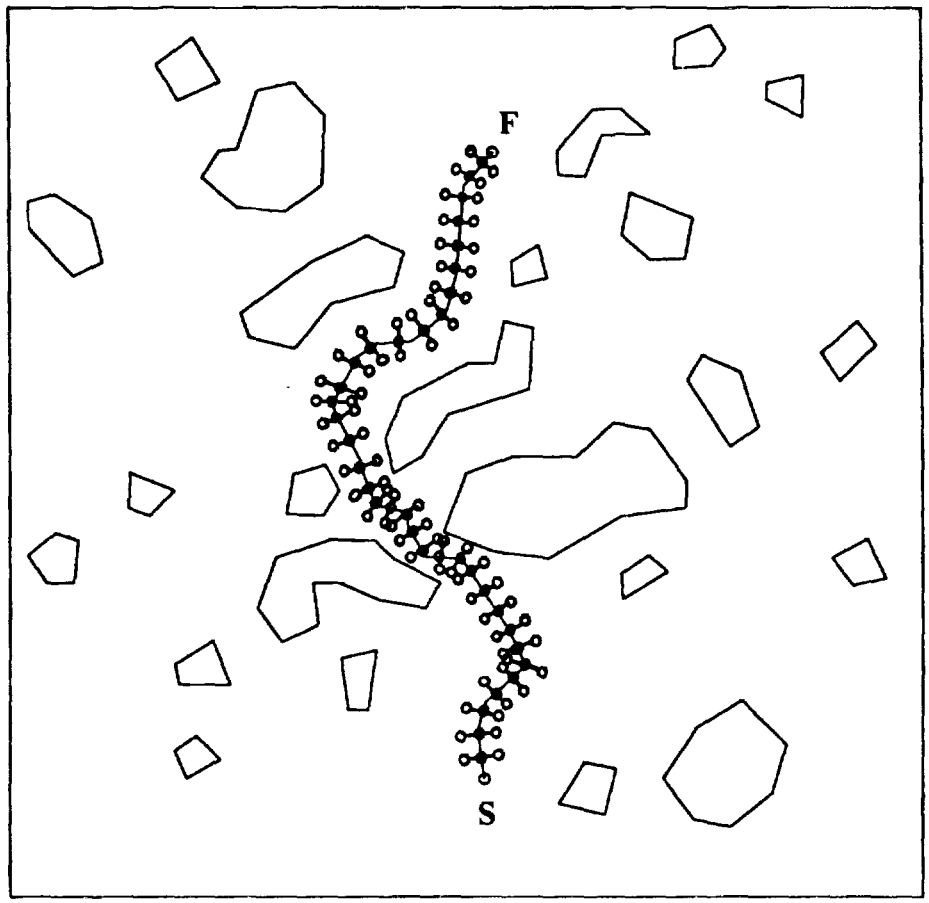


Figure 7.16. Nodes are placed on the optimal solution from the Voronoi diagarm search graph and on lines perpendicular to this solution. An iterative improvement of the solution path allows nodes to vary into new positions.

Table 7.1. Search parameters when the optimal solution is found for the shortest distance path example.

	Dijkstra's Algroithm	A^* Algorithm
Nodes on OPEN	7	11
Nodes on CLOSED	126	44
Nodes investigated †	133 (79%)	55 (33%)
Nodes uninvestigated †	36 (21%)	114 (67%)
Pointers used	148	55

$$R_N = \frac{\textit{Nodes investigated by the } A^* \textit{ algorithm}}{\textit{Nodes investigated by Dijkstra's algorithm}} = .41$$

$$R_P = \frac{\textit{Pointers used by the } A^* \textit{ algorithm}}{\textit{Pointers used by Dijkstra's algorithm}} = .37$$

† Percentages shown indicate the percent of total nodes investigated or uninvestigated. There are 169 nodes in the search graph.

Table 7.2. Search parameters when the optimal solution is found for the example terrain with threats.

	Dijkstra's Algroithm	A^* Algorithm
Nodes on OPEN	13	19
Nodes on CLOSED	121	78
Nodes investigated †	134 (79%)	97 (57%)
Nodes uninvestigated †	35 (21%)	72 (43%)
Pointers used	147	102

$$R_N = \frac{\textit{Nodes investigated by the } A^* \textit{ algorithm}}{\textit{Nodes investigated by Dijkstra's algorithm}} = .72$$

$$R_P = \frac{\textit{Pointers used by the } A^* \textit{ algorithm}}{\textit{Pointers used by Dijkstra's algorithm}} = .69$$

† Percentages shown indicate the percent of total nodes investigated or uninvestigated. There are 169 nodes in the search graph.

Table 7.3. Search parameters when the optimal solution is found for the example terrain with a barrier of threats.

	Dijkstra's Algroithm	A^* Algorithm
Nodes on OPEN	18	21
Nodes on CLOSED	116	105
Nodes investigated †	134 (79%)	126 (75%)
Nodes uninvestigated †	35 (21%)	43 (25%)
Pointers used	139	131

$$R_N = \frac{\textit{Nodes investigated by the } A^* \textit{ algorithm}}{\textit{Nodes investigated by Dijkstra's algorithm}} = .94$$

$$R_P = \frac{\textit{Pointers used by the } A^* \textit{ algorithm}}{\textit{Pointers used by Dijkstra's algorithm}} = .94$$

† Percentages shown indicate the percent of total nodes investigated or uninvestigated. There are 169 nodes in the search graph.

CHAPTER 8

SUMMARY AND CONCLUSIONS

In this thesis the design of an architecture for the control of an autonomous aircraft is presented. The architecture is a hierarchical system representing an anthropomorphic breakdown of the control problem into planner, navigator, and pilot systems. The planner system determines high level plans from overall mission objectives. Mission planning subgoals are directed from the planner to the navigator for intermediate level planning. Finally, the pilot system synthesizes the flight trajectory creating the control commands to fly the aircraft.

The particular navigation problem solved in this thesis is the problem of planning a path for a vehicle flying at constant altitude in mountainous terrain. Mountains (considered as obstacles) from a contour map are first polygonized. The dimensions of the vehicle are considered negligible in comparison to mountain sizes, so the vehicle can be considered as a point. The problem is to construct a path for a point vehicle from a start location to a finish location while avoiding polygon obstacles. Search graphs are constructed to model paths in free space. Three techniques utilizing the Voronoi diagram of points are presented for modeling paths: the centroid point method, the circle rule method, and the contour vertex point method.

The centroid point method models polygon obstacles with a single Delaunay point at the centroid of the polygon. The Voronoi diagram of these Delaunay points is used to construct a search graph. This method is shown to topologically represent free space well, but fails to guarantee a feasible search graph for arbitrary polygon obstacle configurations. The only space that this method models correctly with a feasible search graph is an environment of nonoverlapping circular obstacles, all of the same radius.

The circle rule method models polygon obstacles with multiple Delaunay points, and removes edges from the Voronoi diagram that are formed from two Delaunay points modeling the same obstacle. The search graphs that result have the salient features that they are feasible search graphs, they topologically represent free space well, and are simple to search. The major drawback to this method is that a judicious

selection of Delaunay point locations and construction circle radius parameter must be performed in order to model polygon obstacles well.

The contour vertex point method models polygon obstacles with Delaunay points located at polygon obstacle vertices. The Voronoi diagram of these Delaunay points is used to construct a search graph, removing Voronoi edges that are formed from two Delaunay points modeling the same obstacle. The search graphs that result have the salient features that they are feasible search graphs, they topologically represent the free space well, and are simple to search.

Several navigation path planning examples are presented using the contour vertex point method to model mountainous terrain. Dijkstra's dynamic programming algorithm and the A^* algorithm are used to search the Voronoi diagram search graph. The first example demonstrates how the minimum distance path in the Voronoi diagram search graph can be found. Next, a terrain environment with threats is considered to show how threat information is added to the search graph. The search finds a solution path that minimizes a combination of distance and exposure to threats. The final example involves a terrain environment with a barrier of threats. The search results indicate an optimal path that penetrates the least costly threat regions. For all these example problems, the search efforts of Dijkstra's dynamic programming algorithm and the A^* algorithm are compared. The heuristic applied in the A^* algorithm is the distance from the current position to the finish location. This heuristic reduces the search effort of the A^* algorithm compared to Dijkstra's algorithm for all the search examples. However, the reduced search effort is less pronounced in the examples with threats due to the lack of a heuristic based on estimating threat information.

Additional search problems treated in this thesis are related to mission planning. It is proposed that the Traveling Salesman Problem is typical of the mission planning problem solved by the planner system. An 11-city Traveling Salesman Problem is solved using breadth first, depth first, and best first search techniques. Two heuristics that that lead to admissible searches are presented. Finally, the basic Traveling Salesman Problem is varied by introducing local and global constraints. The resulting problems are representative of complex mission planning problems.

CHAPTER 9

RECOMMENDATIONS

The following recommendations are made for further research related to the content of this thesis.

AI in the Control Loop

This thesis poses the problem of controlling an autonomous aircraft as a problem of controlling a vehicle through the mixture of classical control techniques in the inner loops and AI techniques in the outermost loops. A greater understanding of this general structure is needed. For instance, if an expert system is used to reason about the mountainous terrain in the immediate region of an autonomous aircraft, what kinds of requirements must be imposed on the amount of time and precision of the reasoning for stable control of the aircraft. Will the path plans generated by a navigation path planning system like the one proposed in this thesis command a terrain following/terrain avoidance algorithm properly? Also, can the reasoning in the outmost loops be used to help the innermost loops gain predictive information about what might happen in the immediate future of inner loop controllers.

Voronoi Diagram Search Graphs for Polygon Obstacles

Only feasible search graphs are useful for searching for paths amongst obstacles. The Voronoi diagram of the set of points that define the vertices of obstacles was used in this thesis to create a feasible search graph. However, the true Voronoi diagram of a set of polygons also guarantees feasibility by definition. Bounds on the error between the modified Voronoi diagram from the contour vertex point method and the true Voronoi diagram should be established. Is one diagram better to use than the other? Theoretically, the true Voronoi diagram produces paths maximally clear of polygon obstacles. Furthermore, Canny and Donald [10] introduce the

simplified Voronoi diagram which is a feasible search graph that eventuates to a search of a piecewise linear graph. None of these paths are likely to be followed exactly by an aircraft. Thus, the method with the least complexity should be most useful, provided that the paths generated do not violate the constraints of the dynamics of the vehicle. The modified Voronoi diagram of this thesis, the true Voronoi diagram, and the simplified Voronoi diagram should be compared in terms of complexity and analytically for the application of navigation path planning for aircraft. Bounds on the error between the Euclidean optimal path and the paths generated with any of these methods should be established.

Searching for Paths over Three Dimensional Terrain

The navigation problem solved in this thesis is for flying in mountainous terrain at a constant altitude. A more general navigation problem is to fly over a three dimensional mountainous terrain while avoiding mountains and minimizing parameters, perhaps distance and exposure to threats. The threat environment should also be considered as three dimensional, as well as dynamic.

Methods for modeling paths in three dimensional terrain should be investigated. One possible way to model the free space above three dimensional terrain is to use the Voronoi diagram of polyhedra. However, constructing the Voronoi diagram of the polyhedra that represent mountains is not computationally easy. Another possible way in which a Voronoi diagram can be used to model three dimensional terrain is to discretize the problem by considering Voronoi diagram search graphs for several constant altitudes. The Voronoi diagram of polygons (representing mountains at constant altitude) can be constructed for mountains at altitudes from the lowest flyable altitude to the highest altitude feasible for flying. These search graphs can then be connected by search arcs that "staple" the constant altitude graphs together. Modeling techniques investigated should be analyzed empirically to determine if the paths represent strategic paths for the aircraft path planning application.

REFERENCES

REFERENCES

[1] Aho, A. V., Hopcraft, J. E., and Ullman, J. D., *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, MA, 1984.

[2] Andrews, J. Randolph, *Impedance Control as a Framework for Implementing Obstacle Avoidance in a Manipulator,* Masters Thesis, MIT, 1983.

[3] Bellman, Richard, *Dynamic Programming,* Princeton University Press, Princeton, NJ, 1957.

[4] Bellmore, M., and Nemhauser, G. L., "Traveling Salesman Problem: A Survey," *Operations Research,* Vol. 16, No 3, pp. 538-558, May/June, 1968.

[5] Blair, Jesse and Schricker, Karl E., "Robotic Air Vehicle: A Pilot's Perspective," *IEEE Aerospace and Electronic Systems Society Magazine,* Vol. 2, No. 9, pp. 8-11, Sept., 1987.

[6] Brooks, Rodney A., "Solving the Find-Path Problem by Good Representation of Free Space," *IEEE Trans. on Systems, Man, and Cybernetics,* Vol. SMC-13, No. 3, pp. 190-197, Mar./April, 1983.

[7] Canny, John, *The Complexity of Robot Motion Planning,* Ph.D. Thesis, MIT, May, 1987.

[8] Canny, John, "A Voronoi Method for the Piano-Movers Problem," *IEEE International Conference on Robotics and Automation,* IEEE Computer Society, St. Louis, MO, pp. 530-535, March, 1985.

[9] Canny, John, Dept. of Electrical Engr. and Computer Science, University of California, Berkeley, personal communication, Jan., 1988, and
Donald, Bruce, Computer Science Department, Cornell University, personal communication, Jan., 1988.

[10] Canny, John and Donald, Bruce, "Simplified Voronoi Diagrams," *Proceedings of the Third ACM Symposium on Computational Geometry,* Waterloo, Ontario, June, 1987.

[11] Cheeseman, Dr. Peter, Research Institute for Advanced Computer Science, personal communication, Sept., 1986.

[12] Delaunay, B., "Sur la sphére vide," *Bull. Acad. Science USSR(VII), Class. Sci. Mat. Nat.,* pp. 793-800, 1934.

[13] Deutsch, Owen, "Artificial Intelligence Design Challenge at the 1987 Guidance, Navigation, and Control Conference," *AIAA Journal of Guidance, Control, and*

Dynamics, Vol. 9, No. 5, p. 513, Sept./Oct., 1986.

[14] Dijkstra, E. W., "A Note on Two Problems in Connection with Graph Theory," *Numerische Mathematik,* Vol. 1, pp. 269-271, 1959.

[15] Garey, Michael R. and Johnson, David S., *Computers and Intractability,* W. H. Freeman and Co., San Francisco, CA, 1979.

[16] Gilmore, John F., "The Autonomous Helicopter System", *Applications of Artificial Intelligence,* John F. Gilmore, Editor, Proc. SPIE 485, pp. 146-52, May, 1984.

[17] Gilmore, John F. and Semeco, Antonio C., "Knowledge-Based Approach Toward Developing an Autonomous Helicopter System," *Optical Engineering,* Vol. 25, No. 3, pp. 415-427, March, 1986.

[18] Golden, B., Bodin, L., Doyle, T., and Stewart, W., Jr., "Approximate Traveling Salesman Algorithms," *Operations Research,* Vol. 28, No. 3, pp. 695-711, May/June, 1980.

[19] Grefenstette, John J., editor, *Proceedings of an International Conference on Genetic Algorithms and Their Applications,* Carnegie-Mellon University, Pittsburgh, PA, 1985.

[20] Guibas, Leonidas and Stolfi, Jorge, "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams," *ACM Transactions on Graphics,* Vol. 4, No. 2, pp. 74-123, April, 1985.

[21] Gwynne, Peter, "Remotely Piloted Vehicles Join the Service," *High Technology,* pp. 38-43, Jan., 1987.

[22] Held, M. and Karp, R. M., "A Dynamic Programming Approach to Sequencing Problems," *SIAM Journal of Industrial and Applied Mathematics,* Vol. 10, No. 1, pp. 196-210, March, 1962.

[23] Herman, Martin, "Fast Path Planning in Unstructured, Dynamic, 3-D Worlds," *Applications of Artificial Intelligence III,* John F. Gilmore, Editor, Proc. SPIE 635, pp. 505-512, April, 1986.

[24] Hopfield, J.J. and Tank, D.W., "'Neural' Computation of Decisions in Optimization Problems," *Biological Cybernetics,* Vol. 52, No. 3, pp. 141-152, 1985.

[25] Kambhampati, Subborso and Davis, Larry S., "Multiresolution Path Planning for Mobile Robots," Technical Report CAR-TR-127, CS-TR-1507, Computer Vision Laboratory, Center of Automation Research, Univ. of Maryland, College Park, MD, 20742, May, 1985.

[26] Keirsey, D.M., "ALV Planning and Navigation System," Technical Report HAC REF F8006, Hughes Research Laboratories, Malibu, CA, April, 1987.

[27] Khatib, O., "Real-Time Obstacle Avoidance for Manipulators and Mobile Robots," *IEEE International Conference on Robotics and Automation,* IEEE Computer Society, St. Louis, MO, pp. 500-505, March, 1985.

[28] Kirkpatrick, S., "Optimization by Simulated Annealing: Quantitative Studies," *Journal of Statistical Physics,* Vol. 34, No. 5/6, pp. 975-986, 1984.

[29] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P., "Optimization by Simulated Annealing," *Science,* Vol. 220, No. 4598, pp. 671-678, May, 1983.

[30] Kuan, Darwin, Zamiska, James, and Brooks, Rodney A., "Natural Decomposition of Free Space for Path Planning," *IEEE International Conference on Robotics and Automation,* IEEE Computer Society, St. Louis, MO, pp. 168-173, March, 1985.

[31] Lawler, E. L., Lenstra, J. K., Rinnooy Kan, A. H. G., and Shmoys, D. B., *The Traveling Salesman Problem,* John Wiley and Sons, New York, NY, 1986.

[32] Lawler, E. L. and Wood, D. E., "Branch-and-Bound Methods: A Survey," *Operations Research,* Vol. 14, No 4, pp. 699-719, June/Aug., 1966.

[33] Lee, D. T., *Proximity and Reachability in the Plane,* Ph.D. Thesis, Coordinated Science Lab. Report ACT-12, Univ. of Illinois, Urbana, IL, 1978.

[34] Lee, D. T. and Schachter, B. J. "Two Algorithms for Constructing a Delaunay Triangulation," *International Journal of Computer and Information Sciences,* Vol. 9, No. 3, pp 219-42, June, 1980.

[35] Lin, C., "Computer Solutions of the Traveling Salesman Problem," *Bell Systems Technical Journal,* Vol. 44, pp. 2245-2269, Dec., 1965.

[36] Little, John, Murty, Katta, Sweeney, Dura, and Karel, Caroline, "An Algorithm for the Traveling Salesman Problem," *Operations Research,* Vol. 11, No. 6, pp. 972-989, Nov./Dec., 1963.

[37] Lozano-Perez, Tomas, "Automatic Planning of Manipulator Transfer Movements," *IEEE Transactions of Systems, Man, and Cybernetics,* Vol. SMC-11, pp. 681-89, Oct., 1981.

[38] McNulty, Christa, "Knowledge Engineering for a Piloting Expert System," *Proceedings of the National Aerospace and Electronics Conference (NAECON),* Vol. 4, pp. 1326-1330, Dayton, OH, May, 1987.

[39] McNulty, Christa, Graham, Joyce, and Roewer, Paul, "Robotic Air Vehicle," *Proceedings of Space Operationa Automation and Robotics Conference (SOAR),* Houston, TX, August, 1987.

[40] Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., and Teller, E., "Equation of State Calculations by Fast Computing Machines," *The Journal of Chemical Physics,* Vol. 21, No. 6, pp. 1087-1092, June, 1953.

[41] Meystel, A., "Automated Map Transformation for Unmanned Planning and Navigation," *IEEE Pecora IX Symposium on Remote Sensing,* Sious Falls, SD, pp. 370-374, Oct. 1984.

[42] Meystel, A. and Holeva, L., "Interaction Between Subsystems of Vision and Motion Planning in Unmanned Vehicles with Autonomous Intelligence," *Applications of Artificial Intelligence,* John F. Gilmore, Editor, Proc. SPIE 485, pp. 87-98, May, 1984.

[43] Mitchell, Joseph S. B., "Planning Strategic Paths Through Variable Terrain Data," *Applications of Artificial Intelligence,* John F. Gilmore, Editor, Proc. SPIE 485, pp. 172,179, May, 1984.

[44] Nilsson, Nils J., "A Mobile Automation: An Application of Artificial Intelligence Techniques," *International Jount Conference on Artificial Intelligence,* Washington, DC, pp. 509-520, May 7-9, 1969.

[45] Nilsson, Nils J., *Principles of Artificial Intelligence,* Tioga Publishing Company, Palo Alto, CA, 1980.

[46] O'Dunlaing, C., Sharir, M., and Yap, C., "Generalized Voronoi Diagrams for Moving a Ladder: I. Topological Analysis," *Communications on Pure and Applied Mathematics,* Vol. 34, pp. 423-83, 1986.

[47] O'Dunlaing, C., Sharir, M., and Yap, C., "Retraction: A New Approach to Motion-Planning," *Proc. of the 15th Annual ACM Symp. on Theory of Computing,* Boston, MA, pp. 207-220, April, 1983.

[48] O'Dunlaing, C., and Yap, C., "The Voronoi Method for Motion Planning: I. The Case of a Disc," Technical Report 53, New York University, Courant Institute of Mathematical Sciences, March, 1983.

[49] Papadimitriou, Christos H. and Steiglitz, Kenneth, *Combinatorial Optimization,* Prentice Hall, Inc., Englewood Cliffs, NJ, 1982.

[50] Pavlidis, T., "Piecewise Approximation of Plane Curves," *Proceedings of the First Joint Conference of Pattern Recognition,* pp. 396-405, Oct, 1973.

[51] Pearl, Judea, *Heuristics: Intelligent Search Strategies for Computer Problem Solving,* Addison-Wesley Publishing Co., Reading, MA, 1984.

[52] Peot, Mark, Cross, Stephen, and Fausett, Mark, "Application of Simulated Annealing and Genetic Search to a Nonlinear Traveling Salesman Problem," *AIAA 1987 Guidance, Navigation, and Control Conference,* Monterey, CA, pp. 417-421, Aug., 1987.

[53] Preparata, Franco P. and Shamos, Michael Ian, *Computational Geometry: An Introduction,* Springer-Verlag, New York, 1985.

[54] Rogers, C. A., *Packing and Covering,* Cambridge Univ. Press, Cambridge, England, 1964.

[55] Rosenkrantz, D., Stearns, R., and Lewis, P., "Approximate Algorithms for the Traveling Salesperson Problem," *IEEE 15th Annual Symposium of Switching and Automata Theory,* pp. 33-42, Oct. 14-18, 1974.

[56] Schwartz J. and Yap, C. K., *Advances in Robotics,* Lawrence Erlbaum Associates, Hillside, NJ, pp. 187-228, 1986.

[57] Shamos, Michael Ian and Hoey, Dan, "Closest-Point Problems," *Proceedings of the 16th Annual Symposium of Foundations of Computer Science,* Univ. of California, Berkeley, CA., pp. 151-162, Oct., 1975.

[58] Thiessen, A. H., "Precipitation Averages for Large Areas," *Monthly Weather Review,* Vol. 39, pp. 1082-1084, 1911.

[59] Thorpe, Charles F., "Path Relaxation: Path Planning for a Mobile Robot," Technical Report CMU-RI-TR-84-5, The Robotics Institute, Carnegie-Mellon Univ., Pittsburgh, PA, 15213, April, 1984.

[60] White, Steve R., "Concepts of Scale in Simulated Annealing," *IEEE International Conference on Computer Design: VLSI in Computers,* Port Chester, NY, pp. 646-651, Oct., 1984.

APPENDICES

Appendix A: Simulated Annealing and its Application to the Traveling Salesman Problem

Simulated annealing is a statistical optimization technique for solving optimization problems with many variables. The genesis of simulated annealing is from the study of interacting individual molecules using a classical analysis in statistical physics [40]. Metropolis, et al. [40], studied the equations of state for systems of many particles and developed the Metropolis algorithm which provides an efficient simulation of the evolution of a configuration of particles in a random state toward an equilibrium state at a *fixed* temperature. Kirkpatrick et al. [29] developed optimization by simulated annealing using this fixed temperature algorithm. The simulation of annealing of a system of particles is performed by running the Metropolis algorithm at an annealing schedule of temperatures which evolves the system from a random state at a high temperature to an annealed state at a low temperature.

A crystal system provides a good example for explaining the simulated annealing concept. When one desires to form a crystal structure, one starts by heating the system to some high temperature where the system is in a liquid state. At this stage the system is in a high energy state. By slowly cooling the system, the system settles into a solid crystal structure, which is a minimum energy state. This process does not, however, guarantee that the resulting crystal will be a "perfect crystal" — a global minimum energy state. If the temperature scheduling is too fast, impurities may form in the crystal structure. Unfortunately, when impurities form at high temperatures, they cannot be removed at any lower temperature. Nonetheless, if a suitably slow temperature schedule is used, then a crystal structure with relatively few impurities should result.

The simulated annealing optimization procedure for general combinatorial optimization problems attempts to simulate the annealing of a physical structure like this crystal system. In this technique, the states of the optimization problem are generalized to states of a physical system, the objective function of the optimization problem is generalized to the energy of the physical system, and a control parameter of the optimization problem is generalized to the temperature of the physical system. Near optimal solutions are sought by allowing the system to anneal from a high temperature to a low temperature.

The basis of the simulated annealing procedure is Metropolis algorithm. The Metropolis algorithm is essentially an iterative improvement on the state of the system, thus, iterative improvement methods will be reviewed preceding the presentation of the simulated annealing procedure.

Iterative Improvement

Iterative improvement is a common technique used in solving computational optimization problems with heuristic guidance. The procedure starts with an initial system state S_i which obeys all the constraints of the problem. Next, the system is rearranged to an improved state S_j which again must also obey the problem constraints. The state S_j then becomes the starting point for further improvements to the system, and the process is continued until no further improvements can be made or a satisfactory solution is achieved.

The inherent limitation of the iterative improvement process is that it locates only local minima. To advance from a local minimum state to another solution region, more complex improvement alterations may be required. Typically, iterative improvement changes involve the swapping of two state components, or the varying of one of the variables of a multivariable system, however, more complex improvement alterations may be required to escape a local minimum. Such changes may be more difficult to perform under the problem constraints and may require more expert reasoning about the local minimum convergence state. To incorporate expert reasoning knowledge about the problem requires more elaborate programming approaches and requires a domain expert. Applications of iterative improvement should *not* require such "expert system qualities" in order to be effective.

One solution to the local minima problem would be to find many local minima, and keep track of the best solutions found. However, finding repeated local minima may be fruitless for converging to a global minimum for large system. Additional criteria for selecting new initial locations would be needed, however, this may require more effort than the computational worth of such a process. Simulated annealing is introduced as an alternative.

Simulated Annealing

Simulated annealing is just iterative improvement done in a sequence of finite temperature movements governed by the Metropolis criterion for accepting or

rejecting randomly generated trial states S_j. The Metropolis criterion replaces the "improvement only" rule used in the basic iterative improvement process described above. The Metropolis criterion [40] states:

$$\text{if } \Delta E \leq 0, \text{ accept the state } S_j, \text{ or}$$
$$\text{if } \Delta E > 0, \text{ accept the state } S_j \text{ with probability } e^{-\Delta E/kT}$$

where ΔE is the change in energy $\Delta E = E(S_j) - E(S_i)$, k is the Boltzmann constant, and T is the temperature parameter. As the system is altered by accepting and rejecting new states at a fixed temperature, the system tends towards thermal equilibrium at that temperature. Using this probability function has the consequence that the likelihood of the system being in a given state is governed by the Boltzmann distribution for that temperature. That is, in thermal equilibrium, the probability that the system is in the state with energy E is proportional to $e^{-E/kT}$.

The Metropolis criterion allows changes in the state of the system that are at higher energy states, thus allowing for the search to proceed away from a local minimum. Because of the nature of the Boltzmann distribution, a positive change in energy is accepted with greater probability at high temperatures, however, fewer positive changes in energy are accepted at lower temperatures. The probabilistic aspect of accepting a new state S_j is implemented by comparing $e^{-\Delta E/kT}$ to a random number drawn from a uniform distribution in the interval (0,1). If $e^{-\Delta E/kT}$ is less than the random number the new state S_j is accepted, and this new state S_j becomes the current state S_i. This selection of new states is repeated until equilibrium is achieved. Then, the temperature is lowered and the procedure is repeated. Figure A.1 shows a generalized procedure for simulated annealing.

Some of the basic questions that must be dealt with when using the simulated annealing procedure involve selecting the initial temperature and the temperature schedule that are used [28,60]. The initial temperature should be high enough so that no impurities are fixed in the system at the start of the search. Usually, this is not difficult to establish unless the temperature parameter has little physical meaning or it is difficult to interpret. The temperature schedule plays an important role in controlling the search, yet the temperature schedule is not well defined by the theory, and a good annealing schedule is usually highly dependent on the problem domain. In general, though, decreasing the system temperature monotonically by a constant ratio is an efficacious scheme for a temperature schedule [28]. Additionally, once at a temperature, every state should be be altered at least a fixed number of times, or a certain number of attempted changes to the state should be achieved before moving on to the next temperature. Often, a running average of the energy is kept to maintain that the system has sufficiently reached thermal equilibrium before proceeding to a new

temperature.

The Traveling Salesman Problem

Because of the complexity of the Traveling Salesman Problem, it is often useful to consider approximate solution techniques, rather than optimal solution techniques. Simulated annealing is one such approximate solution technique. For large problems, the computation of an optimal solution is considered impractical, simply because of the computer time needed to arrive at the solution. Instead, what is truly desired is a very good solution in a reasonable amount of computation time. For the Traveling Salesman Problem, the solution can be determined to be a good solutions simply by inspection: by viewing the results and noting that the tour does not have any large links included, and that close cities are visited while the salesman is in their vicinity.

Consider the Traveling Salesman Problem where the cities are arranged such that they are grouped together in clusters. One can generalize that a good solution for a tour should proceed from one cluster to another, visiting each city within a cluster while at the cluster. Figure A.2 presents four clusters of cities for a 22-city Traveling Salesman Problem. All the cities are within a 100×100 unit square, initially composing a randomly connected tour. The change of state of the system is performed by considering that two cities chosen randomly from a tour can be interchanged in their position in a tour to form a new tour. When two cities are interchanged, two possibilities can occur. First, if the two cities follow each other in the tour, then three links are broken and three new links are replaced. Second, if the two cities do not immediately follow one another in the tour, then four links between cities are broken, and four new links are replaced. These tour modifications are shown in Figure A.3.

The temperature parameter for the annealing schedule can be thought of as a distance parameter for the selection of tour modifications. For the energy function that is the total tour length, the change in energy is simply the change in the tour length caused by the tour modification. Since a modification to a tour can cause four links to be altered, then the temperature parameter should somehow be related to this length. The initial temperature used in the example is 200, about twice the longest possible link in a tour. It was determined empirically that this temperature was sufficiently high for beginning the annealing procedure. The temperature schedule was controlled through two means: a static iteration count to determine when the system reached thermal equilibrium, and a constant temperature ratio α for reducing the temperature. The system temperature was decreased monotonically by the constant ratio $\alpha = .75$ when the system reached thermal equilibrium. Thus, when thermal

equilibrium was reached at the temperature T, the next temperature in the schedule was set to αT.

The results of applying simulated annealing to the example Traveling Salesman Problem can be judged as reasonably good results. The progress of the solution procedure is illustrated in Figure A.4. The final tour, 489 units long, visits each of the clusters once, visiting all the cities within the cluster and then proceeding to visit another cluster. In general, this is considered a good trait. Thus, this solution is judged, by inspection, to be a reasonably good one.

procedure Simulated_Annealing (X_0,T_0)

COMMENT: Input an initial state X_0 and a sufficiently high starting temperature T_0.

$X \leftarrow X_0$
$T \leftarrow T_0$

COMMENT: Stopping Criterion: Search for a solution until $T \leq 0$ or until the allowable computation time runs out.

while ($T > 0$ or time_limit)

COMMENT: Generate a random state change.
$X' \leftarrow$ Generate_New_State (X)

COMMENT: Calculate the change in energy ΔE.
$\Delta E \leftarrow E(X') - E(X)$

COMMENT: Accept lower energy states unconditionally, and accept higher energy states with probability P.

if ($\Delta E < 0$) then $X \leftarrow X'$
else
$P \leftarrow e^{-\Delta E/kT}$
$R \leftarrow$ Random_Number (0,1)
if ($R < P$) then $X \leftarrow X'$
endif

COMMENT: If there is no significant decrease in energy for many iterations then lower the temperature T.

if (static_iteration_number > iteration_limit)
then $T \leftarrow$ Update(T)

endwhile

end procedure Simulated_Annealing

Figure A.1. The generalized simulated annealing procedure.

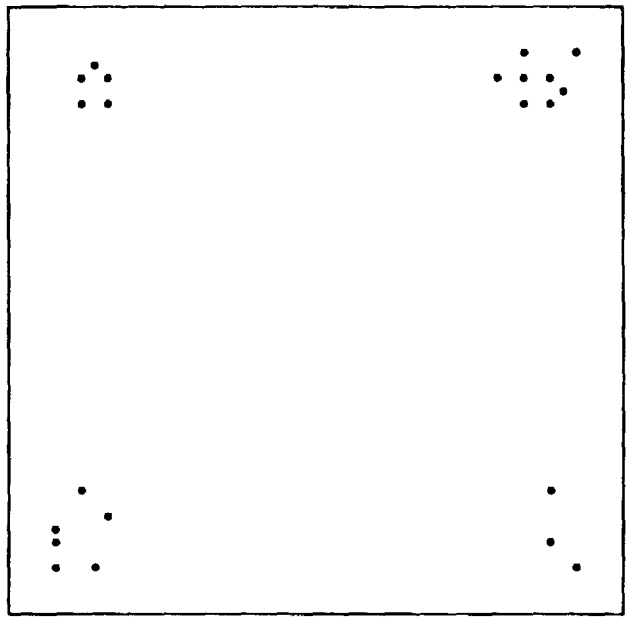


Figure A.2. The cities for a 22-city Traveling Salesman Problem. These cities form four clusters with the cities in close proximity within each cluster.

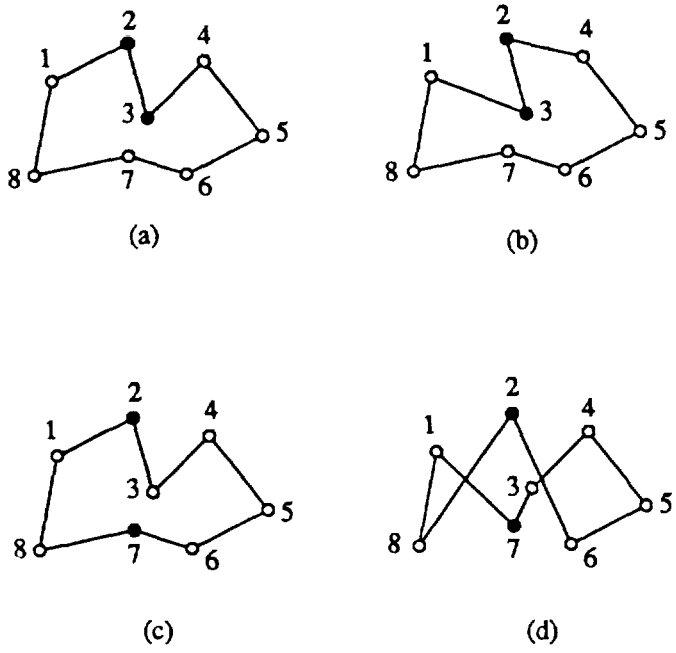


Figure A.3. Tour modifications consist of interchanging two cities in a tour. In the first case, the consecutive cities "2" and "3" are interchanged, thus changing the tour (1 2 3 4 5 6 7 8 1) in (a) to become (1 3 2 4 5 6 7 8 1) in (b). In the second case, the nonconsecutive cities "2" and "7" are interchanged, thus changing the tour (1 2 3 4 5 6 7 8 1) in (c) to become (1 7 3 4 5 6 2 8 1) in (d).

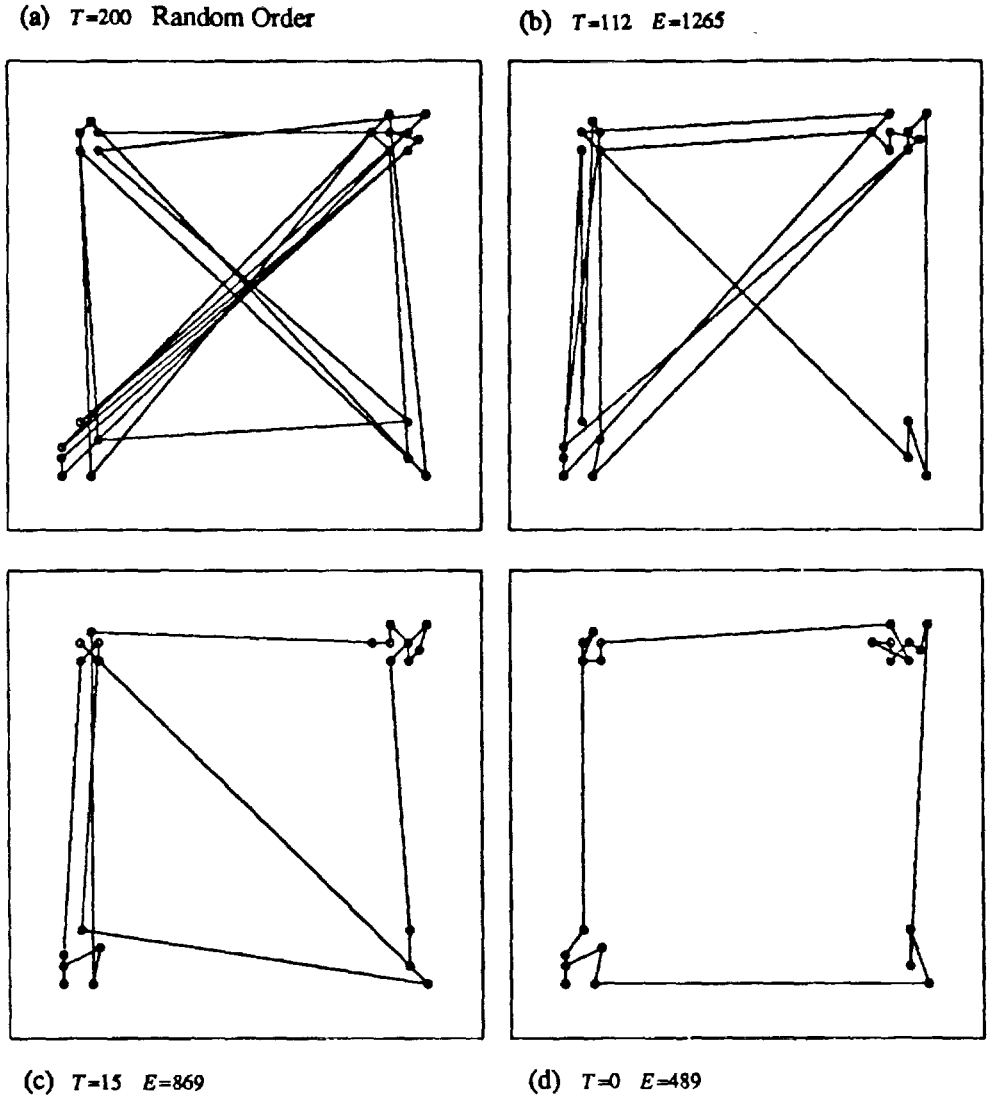


Figure A.4. Simulated annealing results for a 22-city Traveling Salesman Problem with clustered cities. Initially, the cities are randomly connected (a). As the temperature parameter T is lowered from higher temperatures (b) to lower temperatures (c), costly tour edges are removed. The final tour (d) shows a path that visits each cluster once, visiting all the cities within the cluster and then proceeding to visit another cluster. The final tour is judged to be a reasonably good tour by inspection (k=.5, α=.75).

Appendix B: The Construction of Delaunay Triangulations and Voronoi Diagrams

A navigation graph model depicts mountains that are to be avoided and valleys which are flown through. By representing mountains with select points and valley passageways as curves around these points, a search graph can be constructed to aid a decision process for flying though mountainous terrain. Delaunay triangulations [12] and Voronoi diagrams [54] provide the geometric framework for such a model.

Procedures for constructing the Delaunay triangulation and the Voronoi diagram (also known as the Thiessen tessellation [58]) of a set of points in the plane are given henceforth. The accompanying discussion explains the basic concepts necessary for constructing these geometric structures and not the detailed computer code †.

Much of the difficulty in constructing a Voronoi diagram is in determining its topological structure. That is, the vertex locations and edge connections are difficult to maintain while constructing the diagram. The construction of the Voronoi diagram can be simplified by working with its straight-line dual, the Delaunay triangulation of the set of points. For this reason, a discussion of Delaunay triangulations is given first and is followed by an explanation of how a Voronoi diagram can be constructed from a Delaunay triangulation.

The Delaunay Triangulation

A triangulation of a finite set of points S in the Euclidean plane is obtained by joining the points of S by nonintersecting straight line segments so that every region internal to the convex hull of S is a triangle. The convex hull of a set of points is the smallest convex set containing them. Intuitively, the greatest area polygon with edges formed using points in S is the convex hull. Each point in S will either be a vertex of this polygon or be enclosed by this polygon. The Delaunay triangulation of S

† The code presented in this discussion is a pseudo-code not intended to represent any particular language, however, the procedures presented are easily implemented in Ada, C-Language, Lisp, or Pascal — languages that support recursion.

is the triangulation which has the property that the circumcircle of any triangle in the triangulation contains no point of S in its interior. Examples of triangulations are given in Figure B.1 and Figure B.2.

A few properties of the Delaunay triangulation are stated without proof. (Proofs are given in [33]). See Figure B.3 for an example.

> Lemma. Given a set of N points $S = \{s_1, s_2, s_3, \cdots, s_N\}$, any edge between two points s_i and s_j is a *Delaunay edge* of the Delaunay triangulation of S if and only if there exists a point x such that the circle centered at x and passing through s_i and s_j does not contain in its interior any other point in S.
>
> Lemma. (The Circle Criterion) Given a set of N points $S = \{s_1, s_2, s_3, \cdots, s_N\}$, $\Delta s_i s_j s_k$ is a *Delaunay triangle* of the Delaunay triangulation of S if and only if its circumcircle does not contain any other point of S in its interior.

The circle criterion is a fundamental rule for the construction of the Delaunay triangulation. Note that the circle criterion (implicitly) assumes that no four points in the set S are cocircular. Furthermore, a triangulation requires that not all the points in S are colinear.

The algorithm presented for constructing the Delaunay triangulation of a set of points utilizes a divide-and-conquer technique by Lee and Schachter [34]. The algorithm has the following basic structure:

Procedure DELAUNAY_TRIANGULATION (S)
Step 1: Partition S into two subsets S_1 and S_2 of approximate equal size.
Step 2: Construct the Delaunay triangulation for S_1 and S_2 recursively.
Step 3: Merge the Delaunay triangulations for S_1 and S_2 to get the resulting Delaunay triangulation of S.
End procedure DELAUNAY_TRIANGULATION

As indicated by the basic structure, the procedure for calculating the Delaunay triangulation is recursive. First, the set S of N points is sorted in lexicographically ascending order. That is, by increasing x-coordinate or, if the x-coordinates are equal, then by increasing y-coordinate. Duplicate points are removed from the data set. The set S is divided into two subsets S_1 and S_2 where:

$$S_1 = \{s_1, s_2, s_3, \cdots, s_{[N/2]}\} \text{ and}$$

$$S_2 = \{s_{[N/2]+1}, \cdots, s_N\}.$$

Delaunay triangulations of S_1 and S_2 are recursively constructed by repeatedly dividing these subsets into smaller subsets as described above. Recursion terminates when the set S contains either two or three points. (The two point set is not a triangulation. However, it acts as a proper terminating condition for the recursion.) Finally, in backtracking recursive steps, the Delaunay triangulation of the union of two subsets S_1 and S_2 is obtained by merging the individual triangulations of these subsets. The merge process entails finding the upper and lower common tangents of the convex hulls of S_1 and S_2, and zigzagging upward from the lower tangent to the upper tangent making the appropriate insertions for Delaunay edges based on the circle criterion.

Before presenting a more detailed description of the DELAUNAY_TRIANGULATION procedure, some preliminary notation must be adopted. The data structure for representing the triangulation is a doubly linked, circular list. For each point s_i in the triangulation, an ordered adjacency list of points is kept to represent the edges that s_i makes in the triangulation. For example, if s_1 is connected to s_2, s_3, s_4, s_5, and s_6 in a triangulation, then the associated list would be (s_1 s_2 s_3 s_4 s_5 s_6). The counter-clockwise progression of edges is imbedded in this notation. Figure B.4 graphically illustrates this arrangement.

For each point in the configuration there is an associated list, and the entire configuration can be represented as a grouping of all such lists. For the example in Figure B.4, the configuration can be represented as the data structure:

$$\begin{aligned}
&((s_1\ s_2\ s_3\ s_4\ s_5\ s_6) \\
&\ (s_2\ s_3\ s_1\ s_6) \\
&\ (s_3\ s_4\ s_1\ s_2) \\
&\ (s_4\ s_5\ s_1\ s_3) \\
&\ (s_5\ s_6\ s_1\ s_4) \\
&\ (s_6\ s_2\ s_1\ s_5)\ ).
\end{aligned}$$

This list is the configuration list CONFIG_LIST. In addition to the configuration list, a list describing the convex hull of the triangulation is stored. The convex hull of the set of points S, denoted CH(S), is represented as a list of points in counter-clockwise order. For s_i, a point on CH(S), let FIRST(s_i) denote the point immediately counter-clockwise from s_i on the convex hull. For the example in Figure B.4, CH(S) = (s_2 s_3 s_4 s_5 s_6) and s_2 = FIRST(s_6).

In order to locate points relative to a Delaunay edge, the operators PRED(s_i s_j) and SUCC(s_i s_j) are introduced. PRED(s_i s_j) denotes the point which appears immediately clockwise to the edge from s_i to s_j, similarly, SUCC(s_i s_j) denotes the

point immediately counter-clockwise. For the example in Figure B.4, s_6 = PRED(s_1 s_2) and s_3 = SUCC(s_1 s_2). Let LINE_SEG(s_i s_j) denote the line segment directed from s_i to s_j. For determining the location of points relative to line segments (not necessarily Delaunay edges), the simple predicates IS_RIGHT_OF and IS_LEFT_OF are used. For points in a set S, let RM(S) be the rightmost point and let LM(S) be the leftmost point.

A detailed top level Delaunay triangulation procedure will now be presented. The following procedure DELAUNAY_TRIANGULATION uses four other procedures to perform most of the work. Procedures LOWER_COMMON_TANGENT and UPPER_COMMON_TANGENT will return the lower and upper common tangents (LT and UT) for two convex hulls. Procedure MERGE_CONVEX_HULLS will merge two convex hulls into one, and procedure MERGE_TRIANGULATIONS will merge two triangulations into one. These four procedures will be described in more detail later. The procedure for construction of Delaunay triangulations is shown:

```
Procedure DELAUNAY_TRIANGULATION (S CONFIG_LIST)

Comment: CONFIG_LIST is a data structure describing
     the current state of the triangulation configuration.

Comment: Assume that the set S of N points is in
     lexicographically ascending order.

N = number of points in S

If N = 2, then

  CH(S) = the line segment between the points in S
  CONFIG_LIST = the line segment between the points in S
  Return CH(S) and CONFIG_LIST

Else if N = 3, then

  CH(S) = the CCW triangle formed by the points in S
  CONFIG_LIST = the triangle formed by the points in S
  Return CH(S) and CONFIG_LIST

Endif

S1 = first [N/2] points of S
S2 = [N/2]+1 through N points of S
```

Comment: Find the Delaunay triangulation of S_1 and S_2 through recursion.

$CH(S_1)$ = DELAUNAY_TRIANGULATION (S_1 CONFIG_LIST)
$CH(S_2)$ = DELAUNAY_TRIANGULATION (S_2 CONFIG_LIST)

Comment: Find the lower and upper common tangents; merge the convex hulls and triangulations.

LT = LOWER_COMMON_TANGENT ($CH(S_1)$ $CH(S_2)$)
UT = UPPER_COMMON_TANGENT ($CH(S_1)$ $CH(S_2)$)
CH(S) = MERGE_CONVEX_HULLS (LT UT $CH(S_1)$ $CH(S_2)$)
CONFIG_LIST = MERGE_TRIANGULATIONS (LT UT CONFIG_LIST)

Return CH(S) and CONFIG_LIST

End procedure DELAUNAY_TRIANGULATION

The lower common tangent and upper common tangent of two convex hulls are important segments used in the DELAUNAY_TRIANGULATION procedure. These tangent segments are found from two convex hulls. The segments are used in the merge procedures of convex hulls and of triangulations. The lower and upper common tangents of two convex hulls is illustrated in Figure B.5. The procedures for constructing the lower and upper common tangents are now presented:

Procedure LOWER_COMMON_TANGENT ($CH(S_1)$ $CH(S_2)$)

X = $RM(S_1)$
Y = $LM(S_2)$
Z = FIRST(Y)
Z′ = FIRST(X)
Z″ = PRED(X Z′)

```
Loop:  If (Z IS_RIGHT_OF LINE_SEG(X Y))
           TEMP = Z
           Z = SUCC(Z Y)
           Y = TEMP
       Else
           If (Z″ IS_RIGHT_OF LINE_SEG(X Y))
               TEMP = Z″
               Z″ = PRED(Z″ X)
               X = TEMP
           Else
```

```
            Return (X Y)
          Endif
        Endif

        GO Loop

End procedure LOWER_COMMON_TANGENT


Procedure UPPER_COMMON_TANGENT (CH(S1) CH(S2))

X = RM(S1)
Y = LM(S2)
Z' = FIRST(Y)
Z = PRED(Y Z')
Z'' = FIRST(X)

Loop: If (Z' IS_RIGHT_OF LINE_SEG(Y X))
          TEMP = Z'
          Z' = SUCC(Z' X)
          X = TEMP
        Else
          If (Z IS_RIGHT_OF LINE_SEG(Y X))
            TEMP = Z
            Z = PRED(Z Y)
            Y = TEMP
          Else
            Return (X Y)
          Endif
        Endif

    GO Loop

End procedure UPPER_COMMON_TANGENT
```

The MERGE_CONVEX_HULLS procedure for merging two convex hulls will not be shown. This procedure simply breaks the lists representing the two convex hulls at the lower and upper common tangent points, and then connects the two remaining lists that represents the convex hull of the union. For the example in Figure B.5:

$$CH(S_1) = (s_1\, s_5\, s_6\, s_3\, s_2)$$
$$CH(S_2) = (s_7\, s_9\, s_{12}\, s_8)$$
$$CH(S) = (s_1\, s_9\, s_{12}\, s_8\, s_3\, s_2).$$

Finally, the procedure MERGE_TRIANGULATIONS will be described. This procedure takes the lower and upper common tangents of the two convex hulls and merges two triangulations. The circle criterion is used to connect either:

1. the left endpoint of the lower common tangent to a point adjacent to the right endpoint of the lower common tangent, or

2. the right endpoint of the lower common tangent to a point adjacent to the left endpoint of the lower common tangent.

This process determines the next Delaunay edge to be connected to one of the lower common tangent points. The edge is inserted into the triangulation configuration list and the process is repeated with this edge becoming the new starting segment. The process zigzags upward until the inserted edge is the upper common tangent. An example is shown in Figure B.6.

The following procedures are used in the merge procedure. The topological operators INSERT(X Y) and DELETE(X Y) are used to create and destroy edges in the configuration list. INSERT(X Y) will insert X into the adjacency list of Y and Y into the adjacency list of X. DELETE(X Y) will delete X from the adjacency list of Y and Y from the adjacency list of X. The geometric predicate CIRCLE_TEST is the procedure that applies the circle criterion. CIRCLE_TEST(X Y Z W) will test the circumcircle of ΔXYZ and return true if W is not contained in its interior, otherwise it returns false. Note that if CIRCLE_TEST(X Y Z W) is true for all W in S excluding X, Y, and Z, then the triangle ΔXYZ is a Delaunay triangle.

The procedure to merge two triangulations follows:

```
Procedure MERGE_TRIANGULATIONS (LT UT CONFIG_LIST)

Comment: LT is the lower common tangent, UT is the upper common tangent.

L = left endpoint of LT
R = right endpoint of LT

Loop1:  If (LT equals UT) then
            CONFIG_LIST = INSERT(L R)
            Return CONFIG_LIST
        Endif

        A = false
        B = false
        CONFIG_LIST = INSERT(L R)
        R1 = PRED(R L)
```

```
If (R1 IS_LEFT_OF LINE_SEG(L R)) then
  R2 = PRED(R R1)
  If (R2 equals L) then skip Loop2

  Loop2:  If (CIRCLE_TEST(R1 L R R2)) then exit Loop2
          Else
              CONFIG_LIST = DELETE(R R1)
              R1 = R2
              R2 = PRED(R R1)
              If (R2 equals L) exit Loop2
          Endif
  GO Loop2

Else
  A = true
Endif
L1 = SUCC(L R)
If (L1 IS_RIGHT_OF LINE_SEG(R L)) then
  L2 = SUCC(L L1)
  If (L2 equals R) then skip Loop3

  Loop3:  If (CIRCLE_TEST(L R L1 L2)) then
                exit Loop3
          Else
              CONFIG_LIST = DELETE(L L1)
              L1 = L2
              L2 = SUCC(L L1)
              If ( L2 equals R ) exit Loop3
          Endif
  GO Loop3

Else
  B = true
Endif

If (A) then
  L = L1
Else
  If (B) then R = R1
  Else
    If (CIRCLE_TEST(L R R1 L1)) then
     R = R1
    Else
      L = L1
```

```
        Endif
      Endif
    Endif

    BT = LINE_SEG(L R)

  GO Loop1

  End procedure MERGE_TRIANGULATIONS
```

A recursive procedure for finding the Delaunay triangulation for a set of N points has been given. Lee and Schachter [34] show that this procedure has running time $O(N \log N)$, which is asymptotically optimal. Other procedures exist which may be more computationally tractable, to witt, iterative procedures, however, these procedures have running time $O(N^2)$ which is less desirable.

The Voronoi diagram for a set S of N points can be obtained from the straight-line dual of the Delaunay triangulation of S. The following is the Theorem of Delaunay [12].

> Theorem. The straight-line dual of the Voronoi diagram of a set S of points is the Delaunay triangulation of S.

The procedure for finding the Voronoi diagram given the Delaunay triangulation can be performed in linear time $O(N)$. Such a procedure will be presented next.

The Voronoi Diagram

A *Voronoi diagram* for a set S of N points, p_i, $1 \le i \le N$, in the Euclidean plane is a partitioning of the plane into N polygonal regions, one region associated with each point p_i. Figure B.7 shows the Voronoi diagram for a set of points. The *Voronoi region* $V(p_i)$ associated with point p_i is the locus of points closer to p_i than to any of the other $N-1$ points. The *Voronoi edge* separating $V(p_i)$ from $V(p_j)$ is composed of the points equidistant from p_i and p_j. Note that not all Voronoi edges are bounded; some extend infinitely. The intersection of Voronoi edges occur at vertices called *Voronoi points*.

The construction of the Voronoi diagram given the Delaunay triangulation is based on the following Voronoi diagram properties. Assume that no four points in the set S are cocircular. (Proofs are given in [53]).

Theorem. Every vertex of the Voronoi diagram is the common intersection of exactly three edges of the diagram.

Next, consider the points connected by the Delaunay triangulation to be neighboring points.

Theorem. Every nearest neighbor of p_i defines an edge of the Voronoi region $V(p_i)$.

From these two theorems, the Voronoi diagram can be constructed for a set of points by finding all the Voronoi points and connecting them accordingly. Note that each Delaunay triangle has an associated Voronoi point which is the circumcenter of the three points of the Delaunay triangle. Two Voronoi points are connected to each other to form a Voronoi edge if their Delaunay triangles are adjacent. If the Delaunay triangle has an edge that is on the convex hull of the set of points, then the Voronoi edge for that side will extend infinitely. This geometric relationship is shown in Figure B.8.

The procedure for constructing the Voronoi diagram given the Delaunay triangulation is presented next. For each point s_i in the set S, all the Delaunay triangles with a vertex at s_i are processed as follows. A Voronoi point is calculated for each triangle (if not previously calculated). This point is the mutual intersection of the perpendicular bisectors of the edges of a Delaunay triangle. This Voronoi point is connected to the Voronoi point formed for the adjacent triangle counter-clockwise from the current triangle (provided the connection does not already exist). If the current triangle has a Delaunay edge on the convex hull of the set of points, then the Voronoi edge associated with the current Voronoi point is extended infinitely. The process continues in a counter-clockwise fashion until all the Delaunay triangles with a vertex at s_i are processed, as shown in Figure B.9. In this figure, the Voronoi points are shown as dots connected with Voronoi edges in dashed lines. Finally, the process is repeated for all the points, p_i, in the set S.

The procedure VORONOI_DIAGRAM constructs the Voronoi diagram of a set of N points given the Delaunay triangulation. This procedure constructs a data structure V_PT_LIST which stores the Voronoi points and edge connections (this structure is identical to the configuration list structure described previously). The function CALC_V (s_i s_j s_k V_PT_LIST) will calculate the Voronoi point for the Delaunay triangle $\Delta s_i s_j s_k$ and insert this Voronoi point into the V_PT_LIST structure. If this Voronoi point is already in the V_PT_LIST structure, then CALC_V will simply retrieve it. CONNECT (v_i v_j V_PT_LIST) will create a Voronoi edge between Voronoi points v_i and v_j in the V_PT_LIST data structure.

```
Procedure VORONOI_DIAGRAM (CH CONFIG_LIST)

Loop1: For s_i in S

    s_0 = an arbitrary point in the adjacency list of s_i
    s' = s_0
    s'' = SUCC(s_i s')

    If (FIRST(s') equals s_i and FIRST(s_i) equals s'') then
      v_0 = ∞
    Else
      v_0 = CALC_V (s_i s' s'')
    Endif

    V_LAST = v_0

    Loop2: s'' = s'
        s' = SUCC(s')

        If (FIRST(s') equals s_i and FIRST(s_i) equals s'') then
          v = ∞
        Else
          v = CALC_V (s_i s' s'')
        Endif

        V_PT_LIST = CONNECT (v V_LAST V_PT_LIST)

        If (s'' equals s_0) then
          V_PT_LIST = CONNECT (v v_0 V_PT_LIST)
          Exit Loop2
        Endif

        V_LAST = v

        GO Loop2

    GO Loop1

End Procedure VORONOI_DIAGRAM
```

To recapitulate, the procedure to compute the Delaunay triangulation of a set S of N points is an $O(N\log N)$ process, which is asymptotically optimal. The procedure for constructing the Voronoi diagram from the Delaunay triangulation is

$O(N)$. Thus, the combined procedure presented here for constructing the Delaunay triangulation and Voronoi diagram of a set of points is $O(N\log N)$, which is asymptotically optimal.

A final note should be made about the assumptions made for the construction of these geometric structures. The circle criteria is used to determine if a Delaunay edge should be constructed or deleted from the triangulation. This required that no four points could be cocircular. This also lead to the fact that every vertex of the Voronoi diagram is a common intersection of exactly three Voronoi edges. The assumption that no four points are cocircular is *not* essential for the construction of the Delaunay triangulation and Voronoi diagram. However, a more detailed analysis must be executed in the computation of these structures. A simple example is the set of four points that define a rectangle, for which the triangulation could include either of the diagonals of the rectangle. The Delaunay triangulation of the set of points where four or more points are cocircular may not be unique, however, the Voronoi diagram will be unique. The only strict assumption that must be maintained is that all the points must not be colinear. In this case, the problem is a degenerate case and the solution for the Voronoi diagram is trivial. An algorithm that handles degenerate cases is presented by Guibas and Stolfi [20].

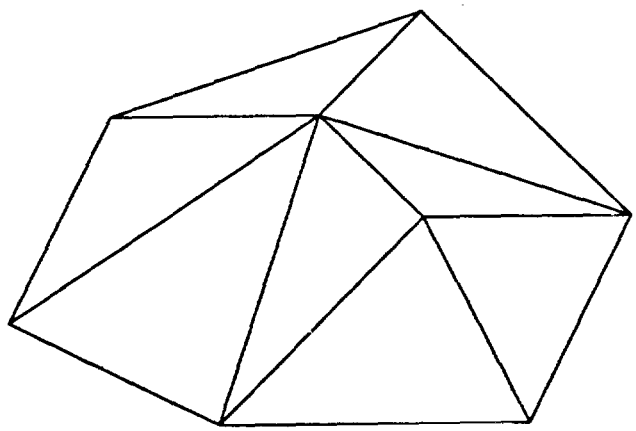


Figure B.1. A triangulation of a set of points.

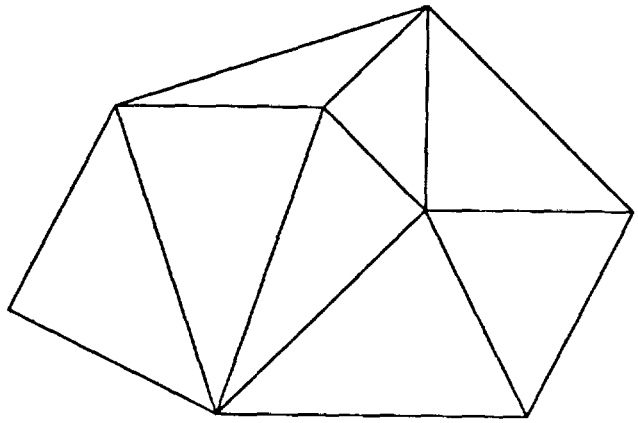


Figure B.2. The Delaunay triangulation of a set of points.

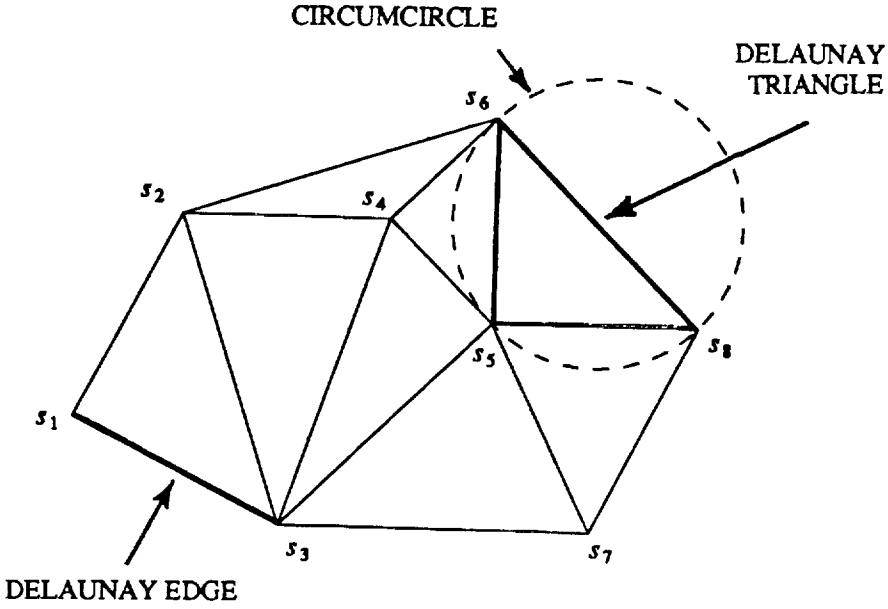


Figure B.3. An illustration of a Delaunay edge, Delaunay triangle, and the circumcircle of the Delaunay triangle.

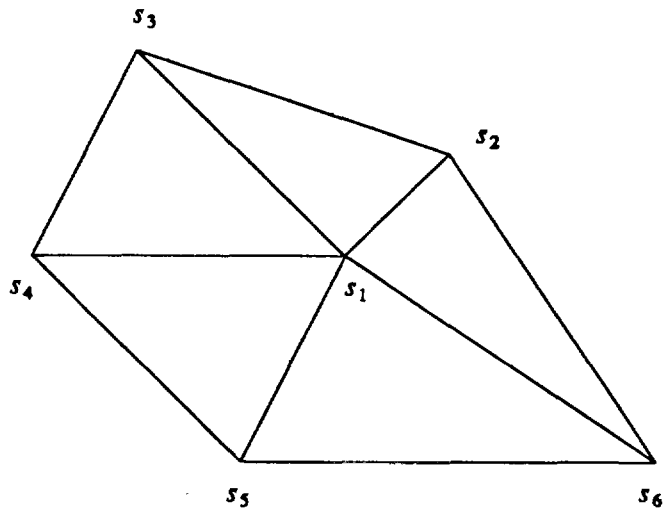


Figure B.4. An example illustrating the linked list $(s_1\, s_2\, s_3\, s_4\, s_5\, s_6)$.

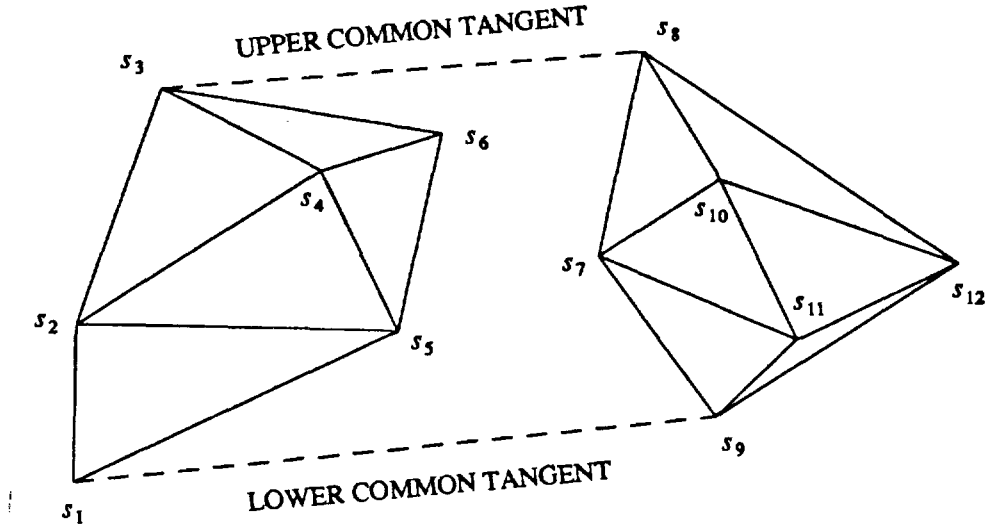


Figure B.5. The lower and upper common tangents of the convex hulls of two triangulations.

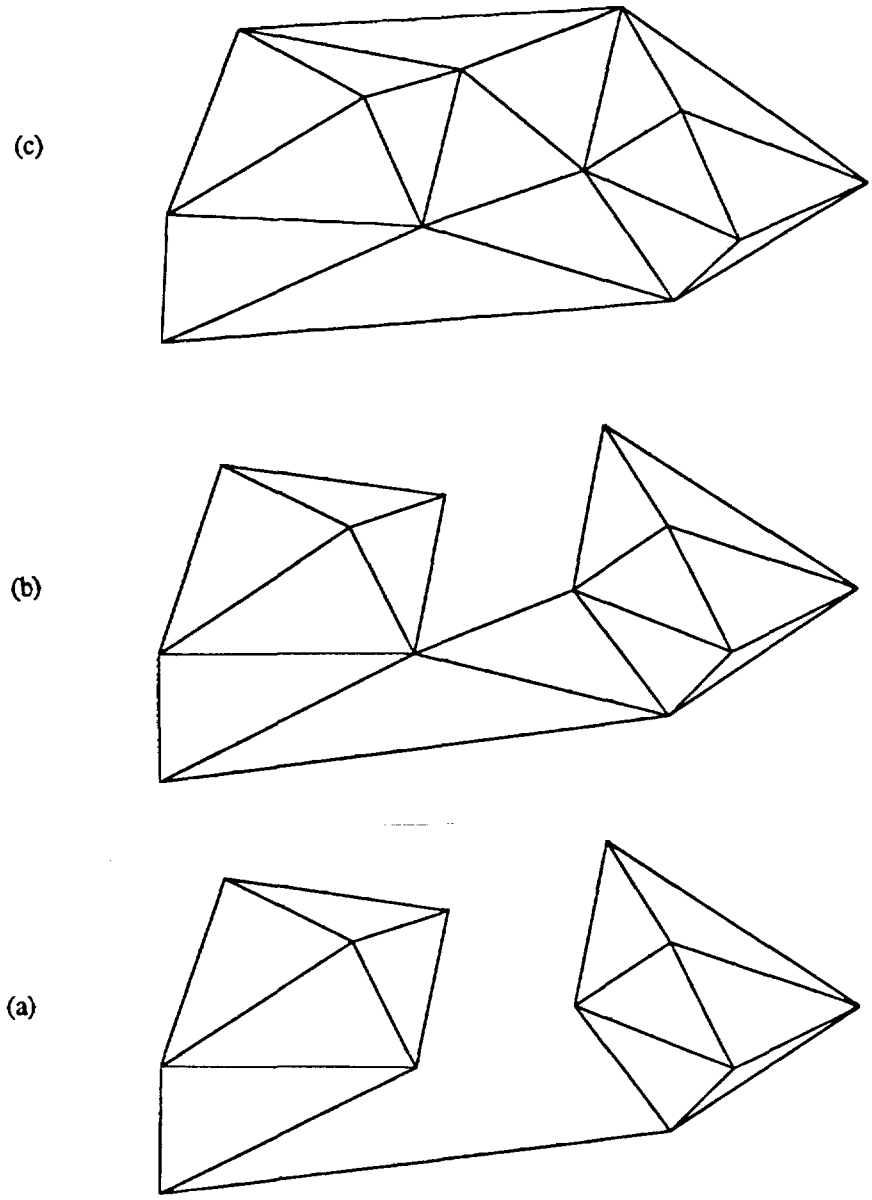


Figure B.6. The triangulation merge procedure starts with the lower common tangent (a), then zigzags upward (b), and ends with the upper common tangent (c).

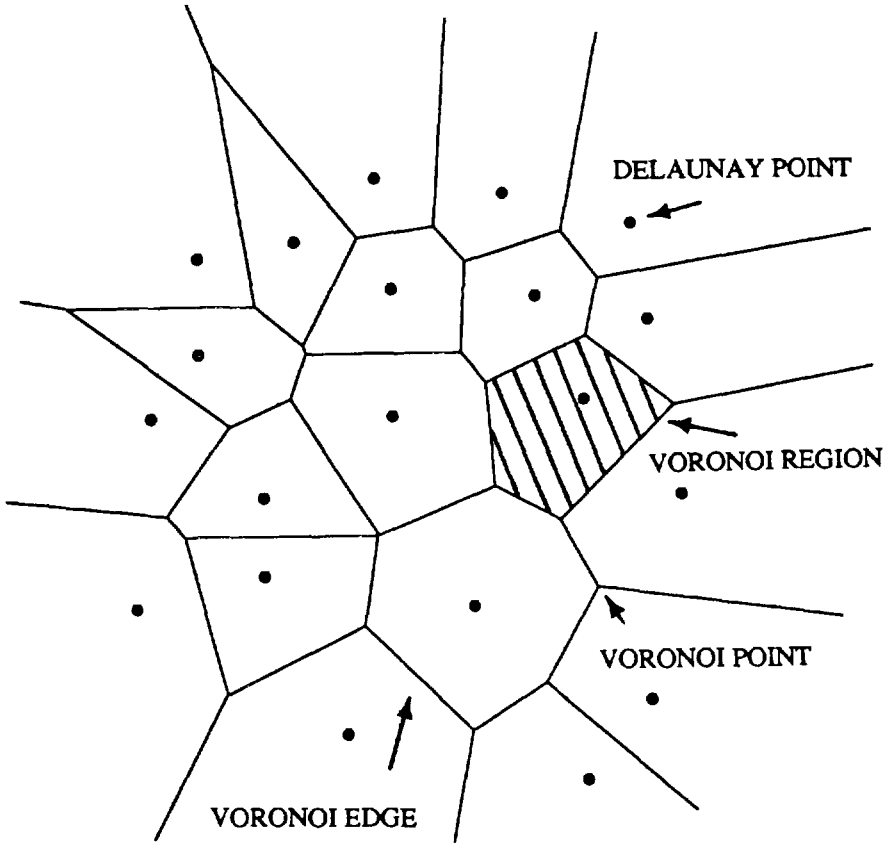


Figure B.7. The Voronoi diagram for a set of 20 points.

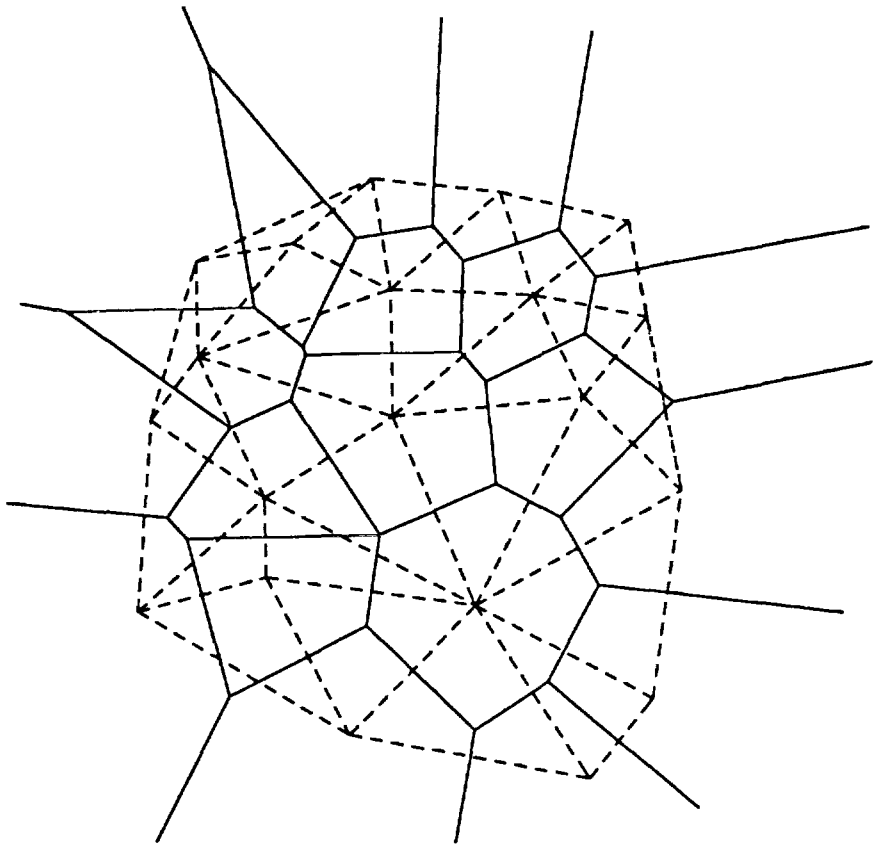


Figure B.8. The Delaunay triangulation (dotted lines) and the Voronoi diagram (solid lines) of a set of 20 points.

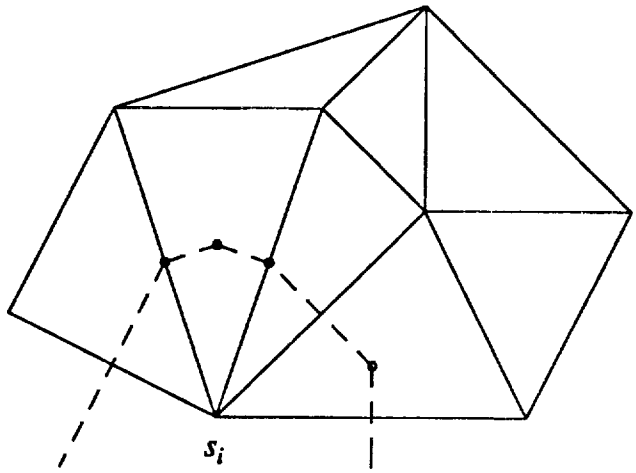


Figure B.9. The Voronoi edges formed when processing the triangles with a vertex at s_i.

Appendix C: Proofs of the Feasibility of Voronoi Diagram Search Graphs

A good method of modeling the polygon obstacle search space is to model only the free space with a feasible search graph. A *feasible* search graph includes only nodes and arcs that are in the free space. This appendix provides proofs of the feasibility of the search graphs generated with the circle rule method and the contour vertex point method.

The Circle Rule Method

The circle rule method entails modeling obstacle boundaries with well placed Delaunay points within obstacle boundaries. All Delaunay points have construction circles around them with the same radius. To model an obstacle, the union of the interior regions enclosed by overlapping construction circles must completely enclose the obstacle polygon boundary. Construction circles from neighboring obstacles must not overlap. After Delaunay points are placed within all the obstacles, the Voronoi diagram is constructed. The Voronoi diagram search graph consists of only those Voronoi edges that are associated with two Delaunay points from different obstacles. Any Voronoi edge that is associated with two Delaunay points from the same obstacle is not part of Voronoi diagram search graph. The proof of feasibility is established by simply showing that no construction circle crosses over any Voronoi edge in the Voronoi diagram search graph. Since construction circles enclose obstacle polygon boundaries, then no Voronoi diagram edge will cross over any obstacle boundary.

Consider the Delaunay triangulation of the set S of N points defining construction circles that completely overlap obstacle boundaries. The Delaunay triangulation and Voronoi diagram for a set of N points can be constructed as discussed in Appendix B. This proof is limited to the restrictions of Appendix B; only consider cases where no three points are colinear and no four points are cocircular. Because the Voronoi diagram is the geometric dual of the Delaunay triangulation, each edge of a Delaunay triangle has an associated edge of the Voronoi diagram, and each Delaunay triangle defines a Voronoi point. Recall from the Circle Criterion that the Voronoi

point is the center of the circumcircle defined by the three vertices of a Delaunay triangle.

Let the points A, B, and C be the Delaunay points defining the *smallest* circumcirle OABC for a Delaunay triangle ΔABC with at least two Delaunay points that do *not* define the same obstacle. Circumcircle OABC has radius r_{ABC}. Let Delaunay points A and B be from different obstacles (point C may or may not be from the same obstacle as point A or point B). There are two cases to consider for a Delaunay triangle ΔABC: Case I, the Voronoi point (center of the circle for the Circle Criterion) and point C are on opposite sides of (or on) the line from A to B, or Case II, the Voronoi point and the point C are on the same side of the line from A to B. Examples of these two cases are shown in Figure C.1. Since we have assumed that the points A and B are from different obstacles, then there must be an associated Voronoi edge $\bar{V}_{AB}$ on the perpendicular bisector of the line segment $\overline{AB}$, as shown in Figure C.2.

We are interested in analyzing the Voronoi edge $\bar{V}_{AB}$. One endpoint of the Voronoi edge is the center of the circumcircle defined by the points A, B, and C. The other Voronoi point defining the Voronoi edge is from the adjacent Delaunay triangle ΔABD, where the two triangles ΔABC and ΔABD share the common edge $\overline{AB}$. An example is shown in Figure C.3 where the Voronoi edge $\bar{V}_{AB}$ is the segment connecting V_{ABC} and V_{ABD}. Because these two triangles are adjacent with the common edge $\overline{AB}$, the points C and D are on opposite sides of $\overline{AB}$. The Voronoi point V_{ABD} is on the ray $\vec{V}$ directed away from point C on the perpendicular bisector of the line segment $\overline{AB}$, as shown in Figure C.4. This is because points C and D are on opposite sides of $\overline{AB}$, and because of the constraint that D cannot be located on or within the circle OABC (this would be a violation of the Circle Criterion). Note that if the edge $\overline{AB}$ is on the convex hull of the set of points S, then there is no adjacent triangle sharing the common edge $\overline{AB}$. In this case, the Voronoi edge extends infinitely.

Can the construction circle for point C intersect the Voronoi edge $\bar{V}_{AB}$? Let the construction circles for the Delaunay points A, B, and C each have the same radius r, where $r < r_{ABC}$. For Case I, it is simple to see that the answer is no. Since $r < r_{ABC}$ and V_{ABC} is on the opposite side of $\overline{AB}$ compared to point C, then the construction circle for point C cannot intersect the Voronoi edge $\bar{V}_{AB}$. The closest the construction circle for point C comes to the Voronoi edge $\bar{V}_{AB}$ is the positive distance $d = r_{ABC} - r$. For Case II, the closest that a construction circle for point C comes to the Voronoi edge $\bar{V}_{AB}$ is no closer than the construction circles for points A and B. Since the construction circles for points A and B do not cross over $\bar{V}_{AB}$, then neither can the construction circle for point C. Figure C.5 illustrates these cases.

Can the construction circle for point D intersect the Voronoi edge $\bar{V}_{AB}$? First, we note that since the circle OABC was chosen to be the smallest Delaunay circumcirle with at least two points from different obstacles, then the construction circle OABD must have a radius greater than r_{ABC}. Now the construction circle for point D has radius r and $r < r_{ABC} < r_{ABD}$. The closest the construction circle for point D can come to the Voronoi edge $\bar{V}_{AB}$ is when point D is arbitrarily close to one of the points A or B. Figure C.6 illustrates this for case I and case II.

Can the construction circle for any point p_i outside of the circumcircles OABC and OABD intersect the Voronoi edge $\bar{V}_{AB}$? It is easily seen that since all construction circles are the same radius r, then no construction circle outside of the circumcircles OABC and OABD can come any closer to the Voronoi edge $\bar{V}_{AB}$ than any construction circle for a point on these circumcircles.

If $\bar{V}_{AB}$ is on the convex hull of the set of points S, then there is no point D (no adjacent Delaunay triangle), and the Voronoi edge extends infinitely. For this case, the argument that the construction circle of point C does not intersect $\bar{V}_{AB}$ remains the same. Considering any construction circle for a point outside the circumcircle OABC must be limited to points outside the circumcircle, but within the convex hull of S. As illustrated by Figure C.7, the construction circle for a point outside of the circumcircle OABC cannot lie any closer than a construction circle on the circumcircle, which cannot intersect $\bar{V}_{AB}$.

Finally, we complete the proof by considering all possible Voronoi edges $\bar{V}_{AB}$. We note that a Voronoi edge $\bar{V}_{AB}$ only exists when at least two points of the Delaunay triangle ΔABC are from different obstacles. Since the arguments above are based on the smallest Delaunay circumcircle of this sort, then any other circumcircle with at least two points from different obstacles must be larger. The above arguments for feasibility can be applied to any circumcircle with at least two points defining different obstacles by allowing the Delaunay triangle ΔABC to be assigned to the smaller of the two circumcircles of two adjacent Delaunay triangles.

The Contour Vertex Point Method

The vertices of obstacle polygons define a set of Delaunay points used for constructing a Voronoi diagram search graph. Voronoi edges that are defined by two neighboring Delaunay points of the same obstacle are removed, creating the resultant Voronoi diagram search graph. Let δ be defined as the shortest distance from any vertex of one polygon obstacle to another polygon obstacle, as illustrated in Figure C.8.

If the polygonization process of the obstacles is performed such that no polygon edge exceeds δ in length, then the resultant Voronoi diagram search graph will be a feasible search graph. That is, the resultant Voronoi diagram search graph will not have any Voronoi edge that crosses over an edge of a polygon obstacle. A proof of this follows.

The Voronoi diagram search graph consists of the Voronoi edges that are associated with two points that are from different obstacles. Any Voronoi edge that is associated with two points of the same obstacle is not part of Voronoi diagram search graph. The proof of feasibility can be shown by simply showing that no obstacle polygon edge can cross over a Voronoi edge in the Voronoi diagram search graph.

Consider the Delaunay triangulation of the set S of N points defining the vertices of the polygon obstacles. Once again, this proof is limited to the restrictions of Appendix B; only consider cases where no three points are colinear and no four points are cocircular. Because the Voronoi diagram is the geometric dual of the Delaunay triangulation, each edge of a Delaunay triangle has an associated edge of the Voronoi diagram, and each Delaunay triangle defines a Voronoi point. Recall from the Circle Criterion that the Voronoi point is the center of the circumcircle defined by the three vertices of a Delaunay triangle.

Let the points A, B, and C be the Delaunay points defining the Delaunay triangle ΔABC, and let A and B be vertices from two different obstacles. There are two cases to consider for a Delaunay triangle ΔABC: Case I, the Voronoi point (center of the circle for the Circle Criterion) and point C are on opposite sides of (or on) the line from A to B, or Case II, the Voronoi point and the point C are on the same side of the line from A to B. Examples of these two cases are shown in Figure C.1.

Since we have assumed that the points A and B are from different obstacles, then there must be an associated Voronoi edge $\bar{V}_{AB}$ on the perpendicular bisector of the line segment $\overline{\text{AB}}$, as shown in Figure C.2. One endpoint of the Voronoi edge is the center of the circumcircle defined by the points A, B, and C. The other Voronoi point defining the Voronoi edge is from the adjacent Delaunay triangle ΔABD, where the two triangles ΔABC and ΔABD share the common edge $\overline{\text{AB}}$. An example is shown in Figure C.3 where the Voronoi edge $\bar{V}_{AB}$ is the segment connecting V_{ABC} and V_{ABD}. Because these two triangles are adjacent with the common edge $\overline{\text{AB}}$, then the points C and D are on opposite sides of $\overline{\text{AB}}$. The Voronoi point V_{ABD} is on the ray directed away from point C on the perpendicular bisector of the line segment $\overline{\text{AB}}$, as shown in Figure C.4. This is because points C and D are on opposite sides of $\overline{\text{AB}}$, and because of the constraint that D cannot be located on or within the circle $\odot$ABC (this would be a violation of the Circle Criterion). Note that if the edge $\overline{\text{AB}}$ is on the convex hull of the set of points S, then there is no adjacent triangle sharing the

common edge $\overline{AB}$. In this case, the Voronoi edge extends infinitely.

Case I and Case II will be treated separately to show that no obstacle edge can cross over the Voronoi edge $\bar{V}_{AB}$.

The Contour Vertex Point Method: Case I

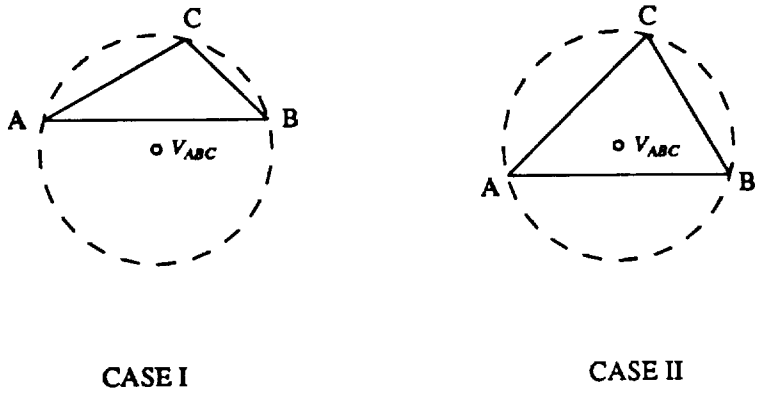
Proving by contradiction, it can be shown that no polygon edge from an obstacle can cross over the Voronoi edge $\bar{V}_{AB}$. Assume that there exist two points p_1 and p_2 from the same obstacle so that the edge from p_1 to p_2 crosses $\bar{V}_{AB}$, as illustrated in Figure C.9. Since the Voronoi point V_{ABD} is defined by the circle containing the points A,B, and D, and since the point D lies on the opposite side of $\overline{AB}$ from the point C, then one can easily see that V_{ABD} is further from the segment $\overline{AB}$ than V_{ABC}, as was shown in Figure C.4. Notice that for Case I, the radius r_{ABD} must be larger than r_{ABC}. Also, because of this, the Voronoi edge $\bar{V}_{AB}$ lies completely within the circle OABD. In order for a line segment to cross $\bar{V}_{AB}$, that segment must cross over the circle OABD, creating a chord to the circle OABD that crosses $\bar{V}_{AB}$. The other constraint for such a segment from p_1 to p_2 is that p_1 and p_2 may not lie on or within the circles OABC or OABD so that the circle criterion is not violated. The shortest possible segment (the limiting case) that meets these constraints is the segment $\overline{AB}$ itself, as shown in Figure C.10. However, by the initial assumption that A and B are from different obstacles, $\overline{AB}$ is of length greater than or equal to δ. Thus, any segment crossing over $\bar{V}_{AB}$ which is formed from two points p_1 and p_2 outside the circles OABC and OABD must be greater in length than δ. Since all segments of an obstacle boundary must be less than or equal to δ in length, then by contradiction, there cannot be any obstacle segment that crosses over the Voronoi edge $\bar{V}_{AB}$. This can also be stated as no Voronoi edge $\bar{V}_{AB}$ crosses over any obstacle edge.

When the segment $\overline{AB}$ is on the convex hull of the set of points, then the Voronoi edge $\bar{V}_{AB}$ is a ray directed away from the edge $\overline{AB}$. In this situation, if the points p_1 and p_2 are *any* two points from the set of obstacle points, then the line connecting them will remain within the convex hull. This line cannot intersect $\bar{V}_{AB}$, which lies outside the convex hull.

The Contour Vertex Point Method: Case II

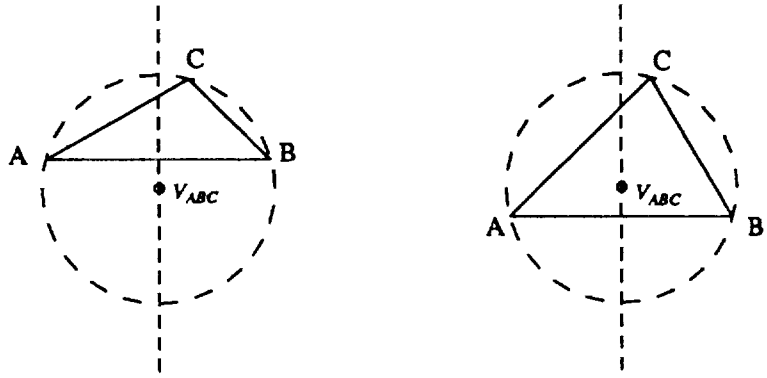
Again proving by contradiction, it can be shown that no polygon edge from an obstacle can cross over the Voronoi edge $\bar{V}_{AB}$. Assume that there exist two points p_1 and p_2 from the same obstacle so that the edge from p_1 to p_2 crosses $\bar{V}_{AB}$, as shown in Figure C.11. For Case II, the point V_{ABC} lies above $\overline{AB}$ while the point V_{ABD} must lie below V_{ABC} as shown in Figure C.4. A particular case is shown in Figure C.3. Next, split $\bar{V}_{AB}$ at the point where $\bar{V}_{AB}$ intersects $\overline{AB}$ to form two segments $\bar{V}_{AB_1}$ and $\bar{V}_{AB_2}$, as shown in Figure C.12. If $\overline{AB}$ does not intersect $\bar{V}_{AB}$, then only the case of $\bar{V}_{AB_2}$ results. Now for $\bar{V}_{AB_1}$ (if it exists), one can show that no segment from p_1 to p_2 can cross $\bar{V}_{AB_1}$ using the same argument as Case I above. For $\bar{V}_{AB_2}$, the argument resides with the fact that the segment $\bar{V}_{AB_1}$ is completely enclosed within OABC. The limiting case is once again the segment $\overline{AB}$. By the initial assumption that A and B are from different obstacles, $\overline{AB}$ is of length greater than or equal to δ. Thus, any segment crossing over $\bar{V}_{AB}$ which is formed by two points p_1 and p_2 outside the circles OABC and OABD must be greater in length than δ. Since all segments of an obstacle boundary must be less than or equal to δ in length, then by contradiction, there cannot be any obstacle segment that crosses over the Voronoi edge $\bar{V}_{AB}$. This can also be stated as no Voronoi edge $\bar{V}_{AB}$ crosses over any obstacle edge.

Finally, we complete the proof by considering all possible edges $\bar{V}_{AB}$. The above arguments for Case I and Case II can be applied to all Voronoi edges $\bar{V}_{AB}$ where the Delaunay points A and B are defined from different obstacles.



CASE I CASE II

Figure C.1. Two cases to consider for a Delaunay triangle. Case I, where V_{ABC} and C are on opposite sides of $\overline{AB}$, and Case II, where V_{ABC} and C are on the same side of $\overline{AB}$. The circumcircle $\odot ABC$ for Delaunay triangle ΔABC is shown in dashed arcs.



CASE I CASE II

Figure C.2. The Voronoi edge $\overline{V}_{AB}$, with one endpoint V_{ABC}, is on the perpendicular bisector (dashed line) of $\overline{AB}$.

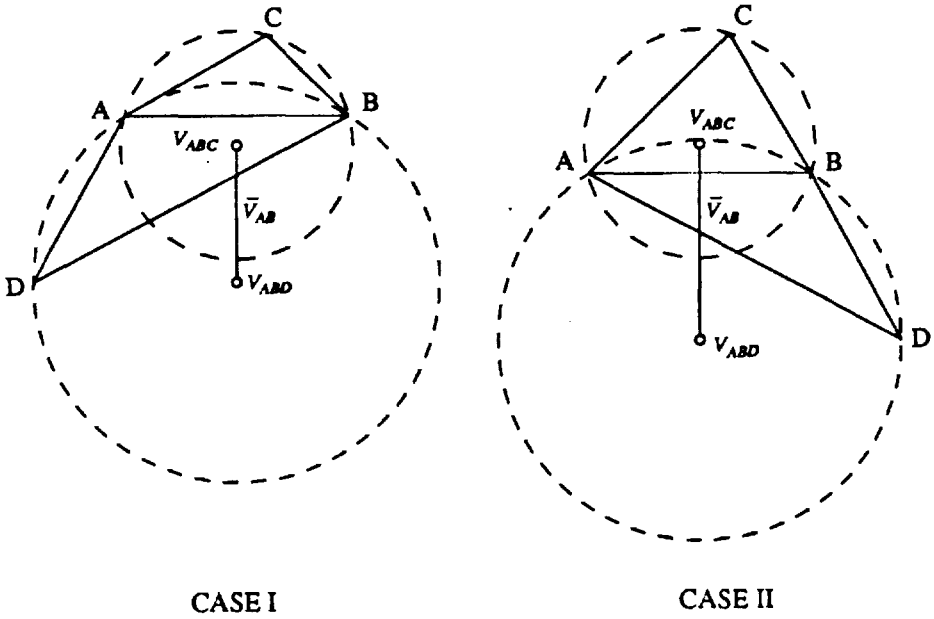


Figure C.3. The Voronoi edge $\bar{V}_{AB}$ is the segment connecting V_{ABC} to V_{ABD}. Note that $\bar{V}_{AB}$ lies on the perpendicular bisector of $\overline{AB}$. Circumcircles for Delaunay triangles ΔABC and ΔABD are shown in dashed arcs.

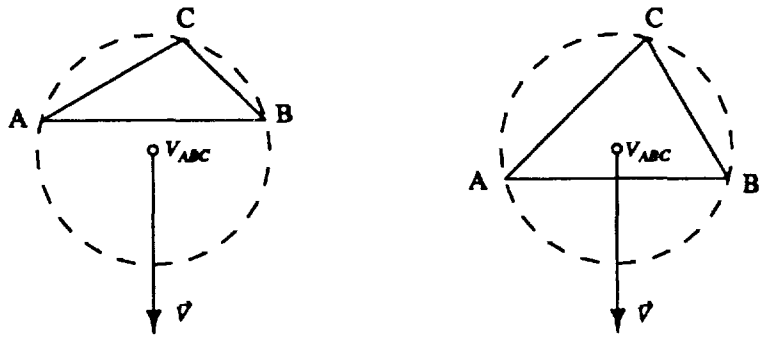
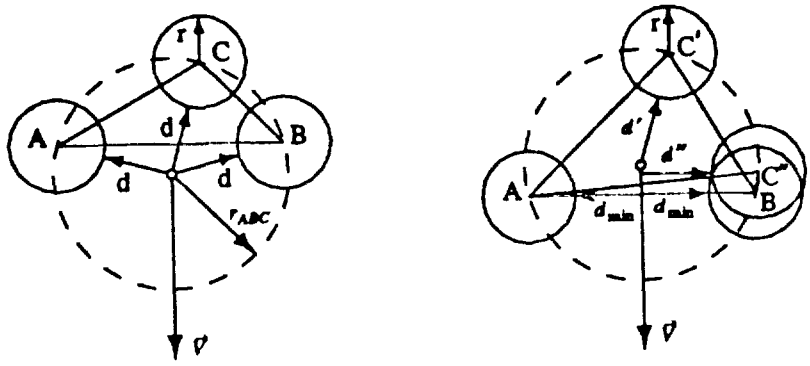


Figure C.4. The Voronoi point V_{ABD} is located on the ray $\vec{V}$, with endpoint V_{ABC}.



CASE I CASE II

Figure C.5 Can the construction circle for point C intersect the Voronoi edge $\bar{V}_{AB}$? For Case I, the construction circle for point C comes no closer than the distance $d = r_{ABC} - r$. For Case II, the closest the construction circle for point C comes to $\bar{V}_{AB}$ is when C is arbitrarily close to point A or point B. The point C′ is at the distance $d' = r_{ABC} - r$ from $\bar{V}_{AB}$, and the point C″ is closer to $\bar{V}_{AB}$ than the distance d', that is, $d'' < d'$, but not closer than the construction circles for points A and B, that is, $d_{min} < d'' < d'$. Construction circles are shown as solid circles. The circumcircle for the Delaunay triangle ΔABC is shown in dashed arcs.

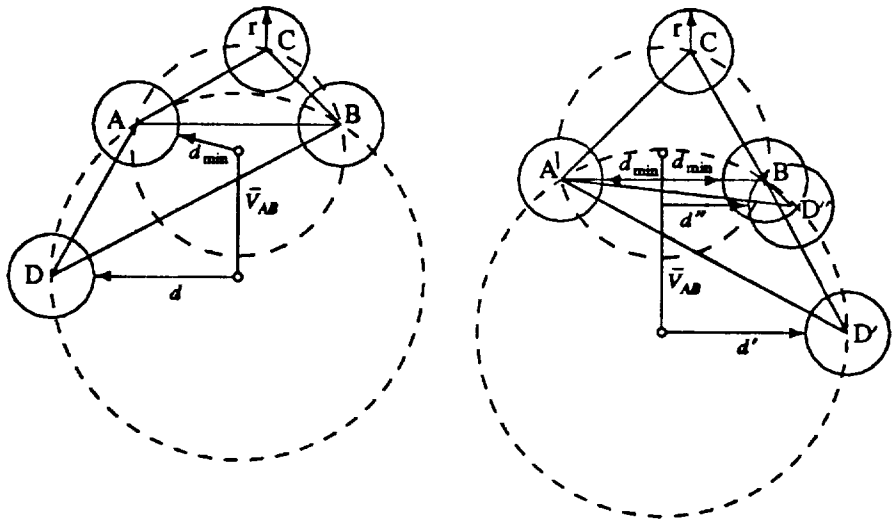


Figure C.6 Can the construction circle for point D intersect the Voronoi edge $\bar{V}_{AB}$? For Case I, the construction circle for point D comes no closer than the distance $d_{\min} = r_{ABC} - r$. For Case II, the closest the construction circle for point D comes to $\bar{V}_{AB}$ is when D is arbitrarily close to point A or point B. This figure shows point D′ which is at the distance $d' = r_{ABD} - r$ from $\bar{V}_{AB}$, and the point D″ which is closer to $\bar{V}_{AB}$ than the distance d', that is, $d'' < d'$, but not closer than the construction circles for points A and B, that is, $d_{\min} < d'' < d'$. Construction circles are shown as solid circles. The circumcircles for the Delaunay triangles ΔABC and ΔABD are shown in dashed arcs.

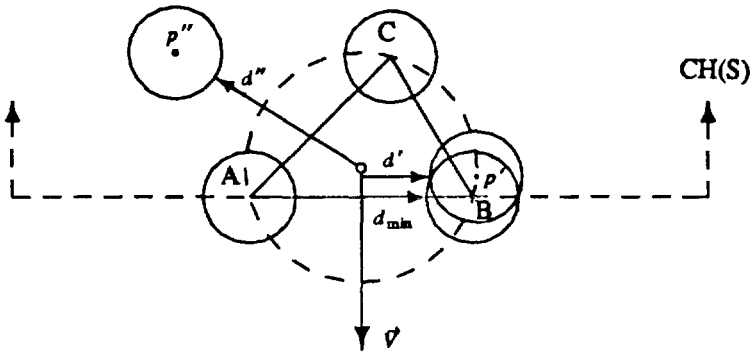


Figure C.7 When $\overline{AB}$ is a segment on the convex hull of the set of points S, then no constuction circle intersects the Voronoi edge $\bar{V}_{AB}$. The closest a construction circle can be to the $\bar{V}_{AB}$ is when the construction circle is arbitrarily close to points A or B, for which the construction circle would still be at least the distance $d_{\min}$ from $\vec{V}$. For example, the point p' is at a distance $d' = r_{ABC} - r$ from $\vec{V}$, and the point p'' is at a distance $d'' \gg d_{\min}$. CH(S) denotes the half plane where the convex hull of S may exist.

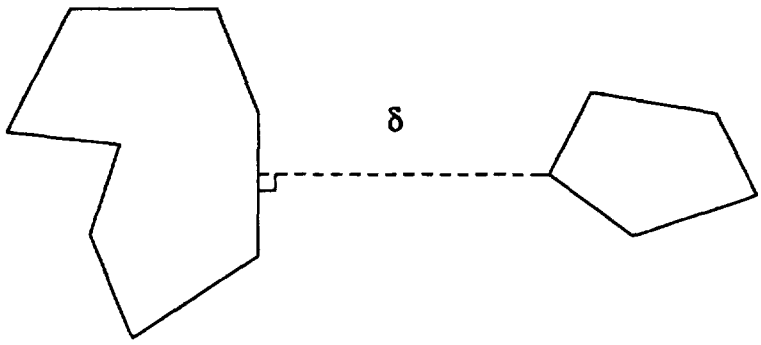


Figure C.8 The shortest distance from any vertex of one polygon obstacle to another polygon obstacle defines δ.

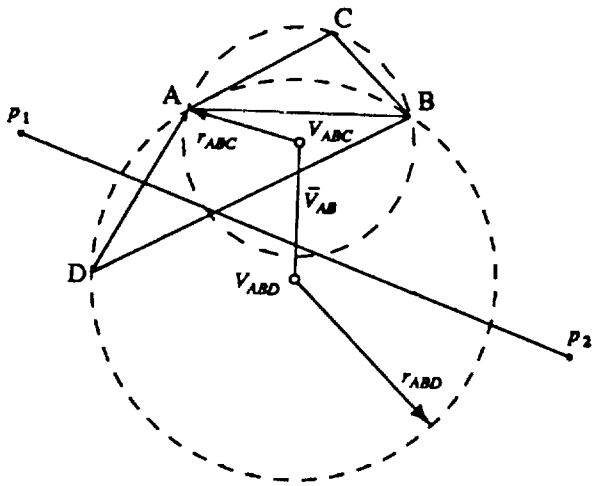


Figure C.9. An arbitrary example for Case I. Assume the segment from p_1 to p_2 crosses over the Voronoi edge $\bar{V}_{AB}$, noting that the points p_1 and p_2 must lie outside the circles OABC and OABD due to the Circle Criterion. Circle OABC has radius r_{ABC} and circle OABD has radius r_{ABD}.

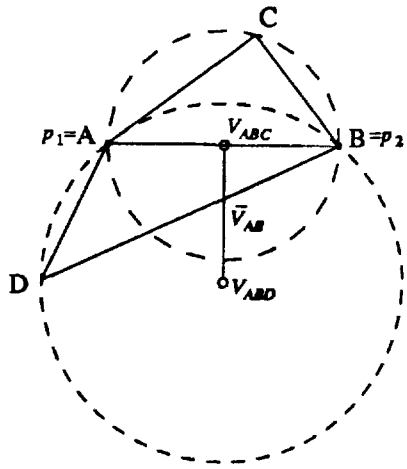


Figure C.10. The limiting case where the segment from p_1 to p_2 is considered to be the segment $\overline{AB}$ which touches the Voronoi edge $\bar{V}_{AB}$.

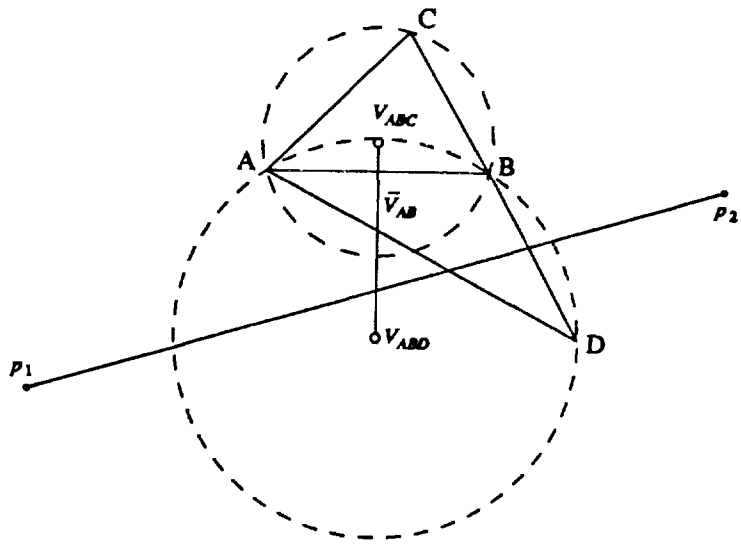


Figure C.11. An arbitrary example for Case II. Assume the segment from p_1 to p_2 crosses over the Voronoi edge $\bar{V}_{AB}$, noting that the points p_1 and p_2 must lie outside the circles OABC and OABD due to the Circle Criterion. Circle OABC has radius r_{ABC} and circle OABD has radius r_{ABD}.

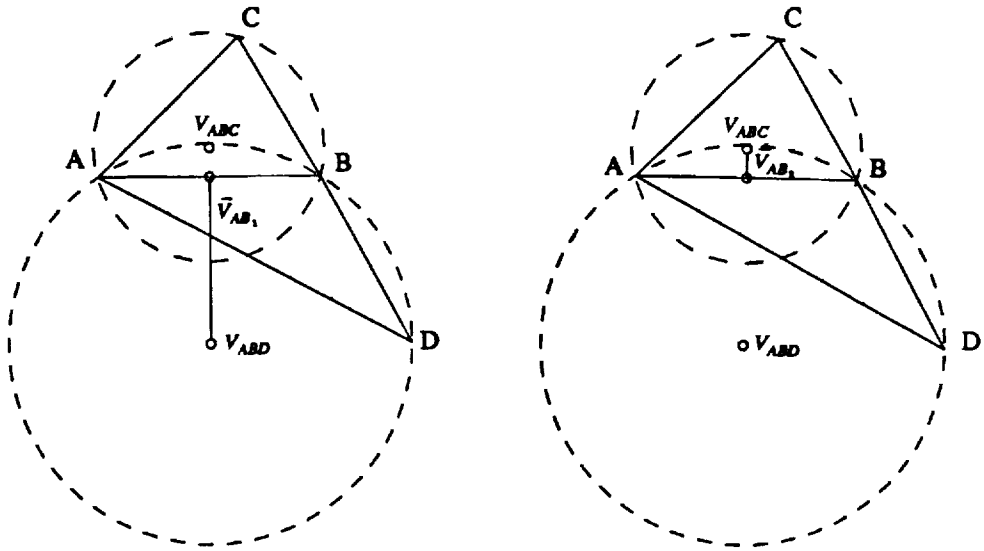


Figure C.12. The Voronoi edge $\overline{V}_{AB}$ is split at the point where $\overline{V}_{AB}$ intersects $\overline{\text{AB}}$ to form two segments $\overline{V}_{AB_1}$ and $\overline{V}_{AB_2}$ as shown.

Appendix D: Properties of the A* Algorithm

This appendix presents the terminology and results of some important properties of the A^* algorithm from the artificial intelligence/operations research fields. The terms used are standard in these fields. A detailed discussion and proofs of the properties of the A^* algorithm are given in [45].

Consider the general graph searching algorithm of Figure D.1. The terms $f(n)$, $g(n)$, and $h(n)$ are defined as follows. The *evaluation function* $f(n)$ estimates the sum of the cost of the minimal cost path from the start node S to node n plus the cost of the minimal cost path from node n to a goal node:

$$f(n) = g(n) + h(n).$$

The term $g(n)$ is called a *cost function,* and the term $h(n)$ is called a *heuristic function.*

Let the function $f^*(n)$ be defined as the sum of the *actual* cost of a minimal cost path from the start node S to node n plus the *actual* cost of a minimal cost path from node n to a goal node:

$$f^*(n) = g^*(n) + h^*(n).$$

With the notation from above, $f(n)$ estimates $f^*(n)$, $g(n)$ estimates $g^*(n)$, and $h(n)$ estimates $h^*(n)$.

Before properties of the general graph searching algorithm can be stated, some additional terms must be defined. The notions of admissibility and the monotone restriction are now stated. If a search algorithm terminates finding an optimal path from the start node S to a goal node whenever a path from the start node S to a goal node exists, then the search algorithm is termed *admissible.* The *monotone restriction* is satisfied by the heuristic function $h(n)$ if for all nodes n and m, with m the successor of n:

$$h(n) \leq h(m) + C(n, m)$$

where $C(n, m)$ is the arc cost between nodes n and m.

The general graph search procedure GRAPHSEARCH in Figure D.1 is termed the A algorithm. If the evaluation function $f(n)$ uses a heuristic function $h(n)$ which is a lower bound on $h^*(n)$, then this general graph search procedure is termed the A^* algorithm. The GRAPHSEARCH (A algorithm) and the A^* algorithm have

some important properties [45]:

Property 1. GRAPHSEARCH always terminates for finite graphs.

Property 2. At any time before A^* terminates, there exists on OPEN a node n that is on an optimal path from node S to a goal node, with $f(n) \leq f^*(S)$.

Property 3. If there is a path from the start node S to a goal node, then A^* terminates.

Property 4. The A^* algorithm is admissible.

Property 5. For any node n selected for expansion by A^*, $f(n) \leq f^*(S)$.

Property 6. If A_1 and A_2 are two versions of A^* such that A_2 is more informed than A_1, then at the termination of their searches on any graph having a path from the start node S to a goal node, every node expanded by A_2 is also expanded by A_1. It follows that A_1 expands at least as many nodes as does A_2.

Property 7. If the monotone restriction is satisfied, then A^* has already found an optimal path to any node it selects for expansion. That is, if A^* selects n for expansion, and if the monotone restriction is satisfied, then $g(n) = g^*(n)$.

Property 8. If the monotone restriction is satisfied, then the value of the evaluation function $f(n)$ of the sequence of nodes expanded by A^* is non-decreasing.

Procedure GRAPHSEARCH

1. Given a search graph G put the start node **S** on a list called OPEN. If **S** does not exist, then exit with failure. Establish the value f (**S**).

2. Create a list called CLOSED that is initially empty.

3. LOOP: if OPEN is empty, exit with failure.

4. Select the first node on OPEN, remove it from OPEN, and put it on CLOSED. Call this node n.

5. If n is the goal node **F** exit successfully with the solution obtained by tracing a path along the pointers from **S** to **F** in G. (Pointers are established in step 7.)

6. Expand node n, generating the set M of its successors in G.

7. For each member m of M that was not already on OPEN or CLOSED, establish a pointer from n to m. Add m to OPEN with the value

$$f(m) = g(n) + C(n, m) + h(m).$$

For each member m of M that was already on OPEN, decide whether or not to change the value of $f(m)$ and redirect its pointer based on

$$g(n) + C(n, m) < g(m),$$

or if

$$g(n) + C(n, m) = g(m),$$

then establish two pointers.

8. Reorder the list OPEN according to heuristic merit.

9. Go LOOP.

Figure D.1. The general graph searching procedure GRAPHSEARCH.

VITA

VITA

James A. Krozel was born to Walter and Irene Krozel on [redacted] in [redacted] He graduated from Niles West High School, Skokie, IL in 1982 and continued his education attending Purdue University, West Lafayette, IN. During his undergraduate program, he achieved degrees in computer science and aeronautical engineering. He received an A.S. in Computer Science on December, 1984 for which he concentrated on course work in numerical methods and computer graphics. On December, 1985 he received a B.S. in Aeronautical Engineering, which was pursued with a major in dynamics and control of aircraft and a minor in propulsion. Jimmy continued to attend Purdue University entered in the Masters program. The M.S. degree was completed in May, 1988 with a major in dynamics and control of aircraft and a minor in artificial intelligence. Jimmy anticipates continuing his education with aspirations of achieving the Ph.D. degree.