1991-03

A stochastic approach to path planning in the Weighted-Region Problem

Kindl, Mark Richard

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

DISSERTATION

**A STOCHASTIC APPROACH
TO PATH PLANNING
IN THE WEIGHTED-REGION PROBLEM**

by

Mark Richard Kindl

March 1991

Dissertation Supervisor: Man-Tak Shing

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

1b. RESTRICTIVE MARKINGS

2a SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE

3. DISTRIBUTION/AVAILABILITY OF REPORT
Approved for public release;
distribution is unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S)

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION
Naval Postgraduate School

6b. OFFICE SYMBOL *(if applicable)*
37

7a. NAME OF MONITORING ORGANIZATION
Naval Postgraduate School

6c. ADDRESS *(City, State, and ZIP Code)*
Monterey, CA 93943-5000

7b. ADDRESS *(City, State, and ZIP Code)*
Monterey, CA 93943-5000

8a. NAME OF FUNDING/SPONSORING ORGANIZATION

8b. OFFICE SYMBOL *(if applicable)*

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS *(City, State, and ZIP Code)*

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION N

11. TITLE *(Include Security Classification)*
A STOCHASTIC APPROACH TO PATH-PLANNING IN THE WEIGHTED-REGION PROBLEM

12. PERSONAL AUTHOR(S)
Kindl, Mark Richard

13a. TYPE OF REPORT
PhD Dissertation

13b. TIME COVERED
From To

14. DATE OF REPORT *(Year, Month, Day)*
March 1991

15. PAGE COUNT
270

16. SUPPLEMENTARY NOTATION
The views expressed in this dissertation are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
path planning, route planning, shortest path, optimization, spatial reasoning, Snell's Law, simulated annealing, weighted-region problem, path annealing, A* search

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Planning efficient long-range movement is a fundamental requirement of most military operations. Intelligent mobile autonomou vehicles designed for battlefield support missions must have this capability. We propose an efficient heuristic algorithm for planning nea optimal high-level routes through complex terrain maps, modeled by the Weighted-Region Problem (WRP). The algorithm is driven by ou adaptation of the combinatorial optimization technique called *simulated annealing*. The WRP provides a cartographically powerfu representation for planar maps. Terrain features are modeled by polygonal homogeneous-cost regions. A cost coefficient assigned to eac region indicates the relative *cost per unit distance* for movement in that region by a point agent on the ground. Region cost coefficients ar assumed to be invariant with direction of movement. Given a start and a goal point, a solution (not necessarily optimal) is a set of piecewis linear connected segments, spanning from start to goal. The cost of a solution path is the sum of the weighted lengths of all segments in th path, where the weighted length of each segment is the product of its Euclidean length and the cost coefficient of the region it crosses. Ideall we seek the least cost path. However, as problem instances approach the actual complexity of the battlefield, faster solutions become mor desirable than absolute optimality. We introduce heuristics designed to reduce the search space independently of start and goal location, thu allowing map preprocessing. We use other heuristics to improve the efficiency of local cost function optimization as well as the annealin process itself.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL
Dr. Man-Tak Shing

22b. TELEPHONE *(Include Area Code)*
(408) 646-2634

22c. OFFICE SYMBOL
Code CS/Sh

Approved for public release; distribution is unlimited

A STOCHASTIC APPROACH TO PATH-PLANNING IN THE WEIGHTED-REGION PROBLEM

by

Mark Richard Kindl
Major, United States Army
B.S., United States Military Academy, 1974
M.S., Naval Postgraduate School, 1983

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE
from the

NAVAL POSTGRADUATE SCHOOL
March 1991

ABSTRACT

Planning efficient long-range movement is a fundamental requirement of most military operations. Intelligent mobile autonomous vehicles designed for battlefield support missions must have this capability. We propose an efficient heuristic algorithm for planning near-optimal high-level routes through complex terrain maps, modeled by the Weighted-Region Problem (WRP). The algorithm is driven by our adaptation of the combinatorial optimization technique called *simulated annealing*. The WRP provides a cartographically powerful representation for planar maps. Terrain features are modeled by polygonal homogeneous-cost regions. A cost coefficient assigned to each region indicates the relative *cost per unit distance* for movement in that region by a point agent on the ground. Region cost coefficients are assumed to be invariant with direction of movement. Given a start and a goal point, a solution (not necessarily optimal) is a set of piecewise-linear connected segments, spanning from start to goal. The cost of a solution path is the sum of the weighted lengths of all segments in the path, where the weighted length of each segment is the product of its Euclidean length and the cost coefficient of the region it crosses. Ideally, we seek the least cost path. However, as problem instances approach the actual complexity of the battlefield, faster solutions become more desirable than absolute optimality. We introduce heuristics designed to reduce the search space independently of start and goal location, thus, allowing map preprocessing. We use other heuristics to improve the efficiency of local cost function optimization as well as the annealing process itself.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I wish to convey my sincere appreciation to my primary dissertation advisor and Ph.D. committee chairman, Professor Man-Tak Shing. His many ideas, personal time, untiring interest, and constant encouragement were instrumental to the completion of this dissertation as well as my entire degree program.

I would also like to thank Professor Neil Rowe. Although only a member of my Ph.D. committee, he also contributed his time and ideas on a weekly basis, and was an additional source of encouragement.

Although he was not a member of my Ph.D. committee, Professor Timothy Shimeall assisted me on many occasions with his programming expertise. I thank him for both his help and also for the encouragement he provided.

The entire Computer Science Department technical staff provided timely software and hardware support during the development, debugging, and testing of the prototypes in this work. In particular, I would like to acknowledge the outstanding assistance I received (almost daily) from both Susan Whalen and Rosalie Johnson.

In every endeavor there are persons who contribute indirectly, and yet significantly. Shirley Oliveira is one such person. As our honorary "Ph.D. Student Den-Mother" she has made the most difficult of times a little easier for all of the Ph.D. students by her encouragement, willingness to listen, and administrative support.

Finally, but most importantly, I wish to thank my wife, Andrea, and my children, Jennifer and Lance, for their unselfish patience and constant understanding throughout the last three and a half years. Their support has been invaluable to my success in this degree program.

I. INTRODUCTION

A. OVERVIEW

Planning efficient long-range movement using map data is fundamental to most military operations. Intelligent mobile autonomous vehicles designed for battlefield support missions must also do this. Within the last decade, successes in the development and practical use of robotic and automation equipment have created a demand for more efficiency and capability in autonomous machines [Iyen89]. Furthermore, U.S. Department of Defense interest in autonomous land vehicles [Keir88, Keir84], automated battle management systems [Wilb89], and digital terrain analysis [Digi88, Clev84], has significantly augmented this demand. Inspired by these requirements, this dissertation presents our design and implementation of an efficient algorithm for high-level route planning through complex terrain. Our algorithm, which we refer to as *path annealing*, finds solutions to what has been called the *weighted-region problem* [Mitc90].

Most previous approaches to the weighted region problem have used systematic search methods. Such techniques may extract huge overheads for agenda maintenance. When problem instances become large and complex, even heuristics can be ineffective in reducing the combinatorics. In contrast, path annealing employs a form of adaptive local search, which samples the solution space and eliminates the need to maintain expensive agendas. As the name suggests, path annealing is based on the probabilistic search technique known as *simulated annealing*, which was first explored independently by [Kirk83] and [Cern85]. In adapting this technique to solve the weighted region problem, we have also designed many heuristics based on annealing, spatial reasoning, and geometry. These enhance performance and also tend to compensate for the weaknesses inherent in local search. Empirical studies indicate that compared to A* search on relatively large problem instances, path annealing has significantly lower execution times, while sacrificing very little solution accuracy. Although, our algorithm cannot guarantee a globally shortest path, our empirical study indicates that it will find it quite often.

B. DEFINITION OF THE WEIGHTED-REGION PROBLEM (WRP)

The weighted region problem (WRP) uses a mathematically simple, yet cartographically powerful problem representation. Input to the WRP is a two-dimensional planar map, which models terrain features as convex polygonal homogeneous-cost regions. We say that a WRP *map* is a connected planar straight-line graph which partitions two-dimensional space into *cost regions*. Each cost region is assigned a *cost coefficient*, which is a weighting value indicating the relative *cost per unit distance* for movement through that region by a *point agent*. Cost regions are assumed to be isotropic, so that their respective cost coefficients are invariant with respect to direction of movement. Without loss of generality, cost coefficients are integers taken from the set $\{w, w+1, w+2, ..., W, +\infty\}$, where w and W are the minimum and maximum weights respectively. If necessary, rational number coefficients can always be handled by rescaling [Mitc90]. A *weighted-region problem instance* is a WRP map specified in terms of boundary edges and cost coefficients, and designated *start* and *goal* points corresponding to the current location and projected destination of the agent.

A solution (not necessarily optimal) is a set of piecewise linear connected line segments, which spans from start point S to goal point G (see Figure 1.1). Denote this path as P_{SG}. The interior points of each segment of P_{SG} lie either entirely within a single cost region, or entirely on a single cost region boundary. Segment end points lie on region boundary edges or vertices. The *total cost* of P_{SG} is the sum of the weighted lengths of all segments of the path, where the weighted length of segment i is the product of its Euclidean length, d_i, and the cost coefficient, μ_i, of the region through which it passes:

$$\left\| P^{*}\ SG \right\| = \sum_{i} (\mu_i d_i) \qquad \text{(Eq 1.1)}$$

A path segment which coincides with all or part of a boundary edge incurs the lesser cost of the two regions separated. An optimal solution, P_{SG}^{*}, is the minimization of Eq 1.1, a globally least cost path from S to G. Measured in time units, it represents the fastest route from start to goal.

We make several additional assumptions regarding the input map. We assume that all weighted regions are convex. Even if this is not the case, it is always possible to insert additional boundary edges between existing vertices to make all regions convex. Such edges will become boundaries between regions of equal cost. We will refer to these as *phantom edges* because their sole purpose is to ensure convexity. *Border edges* and *border vertices* define the outer limits of the map, and bound the only non-convex region. The current implementation of path annealing requires that linear regions (as used in [Rowe90a] and [Mitc90]) must be modeled by regions of positive (non-zero) area. However, this restriction is not essential to the algorithm itself, and can be removed by a few data structure modifications.

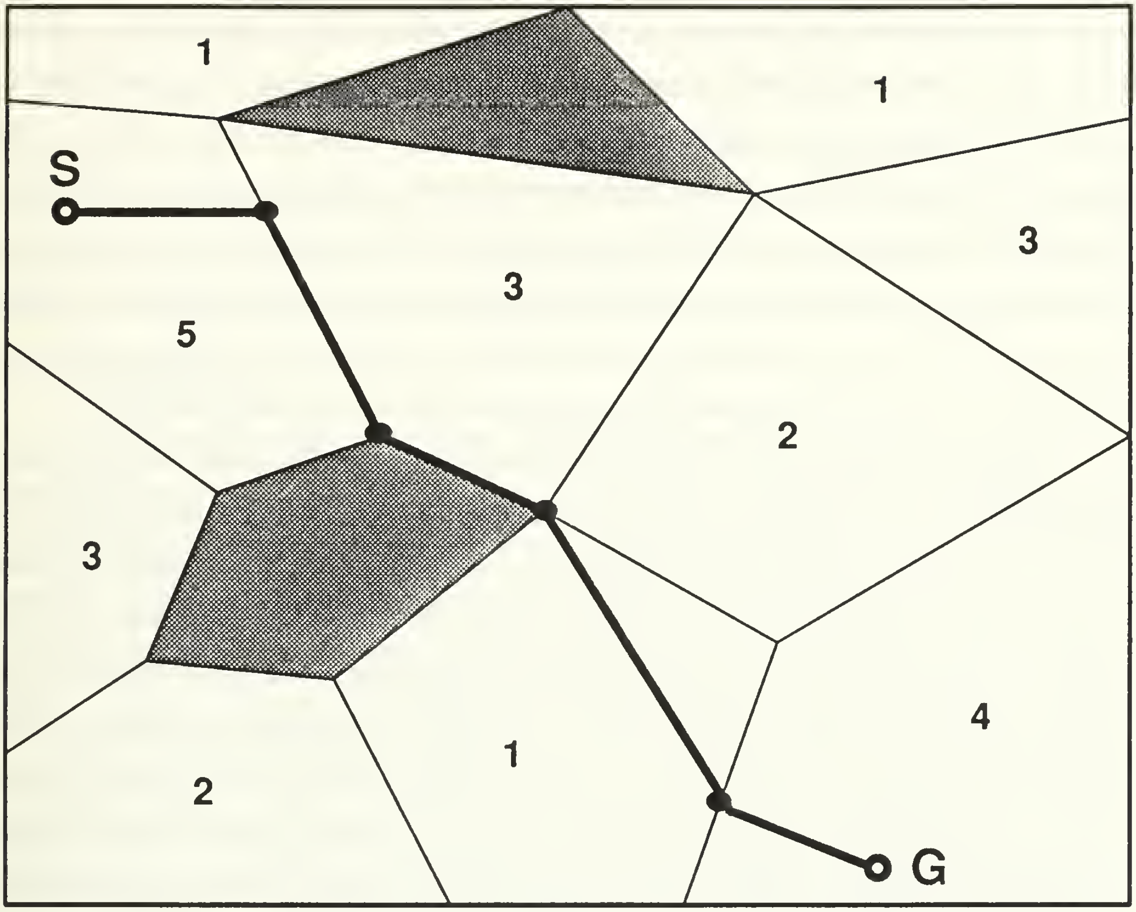


Figure 1.1 Weighted-Region Problem Instance and One Possible Solution (not necessarily optimal)

C. WHY A STOCHASTIC APPROACH?

Algorithms based on systematic search methods apply some deterministic ordering to nodes in a state space search. The usual reason for this is to guarantee search completeness, which may be the only known way to guarantee the optimality of a solution. Such algorithms must track a large number of partial solutions using an *agenda*. However, this does not imply that ordered search is necessarily exhaustive. Heuristics and bounds may provide sufficient cause to prune partial solutions from further consideration. Nevertheless, such search organization sometimes carries a hefty price in the form of agenda maintenance. Larger, more complicated problem instances will generally require more effort of these deterministic algorithms, likely forcing unacceptably worst case performance.

Rather than incur the overhead for organized search, we propose to reduce the effort by using stochastic search. Path annealing is our customization of simulated annealing [Kirk83], a probabilistic combinatorial optimization technique which finds near-optimal solutions to problems having very large and complex search spaces. The annealing approach to the WRP may seem strange for several reasons. First, the WRP is polynomially solvable [Mitc90], yet, annealing's primary domain of application is generally considered to be the class of NP-Hard problems. Second, simulated annealing has often been criticized for its large running times which are sometimes necessary to ensure good solutions. Finally, for some problems, annealing has made a relatively poor showing against good problem-specific heuristics [Naha85]. Although annealing may not appear to be the correct tool for the WRP, our empirical studies tend to indicate otherwise.

Annealing experiments with the Traveling Salesman Problem exhibit a property which is similar to a human approach to route planning. Consider planning a long-range automobile trip using a road atlas. Most people form a solution hierarchically, descending top-down through levels of abstraction from interstate expressways to state highways to local roads. Two basic forms of knowledge contribute to this natural tendency. The first is general knowledge of speeds and traffic patterns associated with the major classes of roads. The second is specific experience-based and analogical knowledge about road quality and traffic patterns in local areas of the map. To solve the Traveling Salesman Problem, simulated annealing forms a solution in this same top-down progression [Kirk83]. Starting with an initial feasible solution, the algorithm samples the search space by rapidly generating and testing random solutions. In the beginning the process finds many of the best gross high-level solutions. The effects of random samples are gradually constrained so that these gross high-level solutions tend to freeze into the evolving detailed solution. Later stages of the process generate and test smaller local changes, gradually refining the lower-level details of a final near-optimal solution.

Hard optimization problems with complex solution spaces usually have a large number of local minima with costs very near to that of the optimal solution(s) [Kirk83]. The WRP solution space, characterized by multiple local minima, appears to be well-suited for a stochastic approach, especially in larger problem instances. A high resolution requirement for terrain modeling could easily produce a problem instance which would be arbitrarily complex in terms of vertices, boundary edges, regions, and their shapes and orientations. Consider the case of an input map which is an approximate model of the real world. The sensitivity of true optimal paths to very small changes in geometry and cost coefficient magnitude suggests that the effort associated with finding a true optimal path is rarely worth the result. Simulated annealing proceeds under this assumption, and finds a near-best solution without the expense of systematically proving its worth.

New approaches to problems often suggest additional constraints, heuristics, or bounds which might not have been possible with earlier methods. Experience with annealing indicates that tailored heuristics can increase its efficiency [Rome84b, Whit84, Gree84, Naha85, Adam85]. We have borrowed several methods of improvement from previous work. In addition, we have developed several of our own heuristics and constraint mechanisms which are specific to the WRP and path annealing. Some of our heuristics can also support other path planning algorithms. Our stochastic approach to the WRP attempts to merge simple intelligence with practical engineering by coupling simple heuristics to non-exhaustive, controlled sampling.

D. APPLICATIONS

If we consider the WRP in a military planning context, then its significance to automated route planning becomes more apparent. The problem is generic enough to model a variety of spatially oriented applications. For a given map, regions and associated cost coefficients can represent any unit of measure provided that within each region, cost remains isotropic, homogeneous, and varies linearly with planar distance. Examples of cost measure include ease of movement through terrain, cover and concealment, exposure to enemy positions, fuel or energy consumption, lethality of enemy weapons, or areas of contamination [Wilb89, Rowe90d]. Multiple weighted-region maps could be merged by overlaying one atop another and performing a combining operation on the cost coefficients of intersecting areas. The composite map would have additional new vertices and regions with revised boundary edges resulting from boundary-edge intersections and region overlaps. A route planning algorithm operating on such a map would simultaneously optimize over several mission-critical parameters. For example, by using cost coefficients obtained from weighting the the relative importance of each cost parameter, an optimal solution on such a composite map might represent the path of combined minimum travel time, least exposure to the enemy, and maximum survivability [Dent84]. In general, the complexity of route-planning on merged maps grows with the number of boundary edge intersections and cost region overlaps. The need for fast solutions assumes increasing importance as the scope of a problem instance approaches the reality and complexity of battlefield mission planning.

For a given problem instance, each execution of a stochastic algorithm can result in a different solution. Whether executed in a parallel processing environment or given ample time to conduct multiple executions serially, the best outcomes can be selected. A collection of different solutions from several processors can provide insights into terrain overlays that might otherwise go unnoticed. For example, high-speed avenues of approach, key choke points, and major by passing routes become more apparent as the military planner correlates multiple solutions.

Solutions generated by a nondeterministic approach such as path annealing are appealing from the military commander's perspective for several reasons. A globally optimal path returned by a deterministic algorithm is predictable and may be very undesirable in an offensive tactical plan. An algorithm which only finds a globally optimal path for a given problem instance will convey little surprise to an enemy, since he can do likewise with a deterministic algorithm! On the other hand, a stochastic algorithm has a reasonably good probability of discovering an optimal solution, but one of many near-optimal paths may be more likely. Since such paths are not always spatially close to the optimal, routes found by the stochastic algorithm are not always predictable.

If route planning must be conducted under time-variable constraints, then nondeterministic path annealing has a definite advantage over the systematic approaches. Systematic search usually must run to completion or near-completion in order to obtain a complete solution. In contrast, an annealing algorithm approaches the optimal solution, i.e. its theoretical asymptotic limit, as time of execution approaches infinity [Rome84a]. This means that the quality of a solution can improve with time. The starting point of annealing is a feasible solution. Therefore, even when time constraints are uncertain, it would be possible to abort path annealing prematurely to return the best solution currently known. The more time allotted to the algorithm, the more opportunities exist for solution improvement.

As we have said, an important advantage provided by path annealing is its ability to achieve dramatic reductions in average computation time necessary to find good solutions, particularly as problem instances become larger and more complex. This is because the annealing algorithm only samples the solution space, and is guided probabilistically by the sampling results. The random nature of the sampling process means that overhead for maintenance of partial solutions in an ordered agenda is unnecessary. In general, a predefined set of parameters constituting an *annealing schedule* determines the expected running time of an annealing algorithm. Figures 5.33 and 5.34 in Chapter V (page 125) show time of execution as a function of problem size for 66 different weighted-region problem instances for A* search and path annealing respectively. This figure illustrates our claim that path annealing can require significantly less average running time than a typical systematic search algorithm. In a majority of these 66 cases the total weighted path cost of the path annealing solution is the same (or nearly the same) as the systematic search solution cost.

E. ORGANIZATION OF CHAPTERS

We begin in Chapter II by discussing basic concepts and terms relevant to this and previous research. The chapter summarizes past approaches to solving the WRP and closely related problems, emphasizing strengths and weaknesses. We also discuss simulated annealing and some examples of its uses. In Chapter III we present the development of path annealing. This chapter is organized around the components of a basic annealing algorithm. While only a few enhancements are discussed in Chapter III, Chapter IV emphasizes performance improvements to path annealing that are achieved through heuristics and bounds. Some of these are applicable to other approaches to the problem as well. In order to test and evaluate path annealing, we must have a valid means of comparison and control. Chapter V begins with a discussion of the wavefront propagation algorithm. We then describe another simple control algorithm which is uniquely different from previous algorithms for the WRP. We refer to it as *uniform-discrete-point global A** (UDPGA*) search. We compare the performance of UDPGA* to wavefront propagation and show that UDPGA* is a faster, more accurate systematic search algorithm. We then directly compare path annealing with UDPGA* for a variety of maps. The remainder of Chapter V is a summary of these test results and analysis. Chapter VI summarizes our conclusions and suggests extensions to this research.

II. RELEVANT CONCEPTS AND PREVIOUS RESEARCH

A. FUNDAMENTAL CONCEPTS

1. Problem Representation

a. High-Level vs. Low-Level Path Planning

The work reported in this dissertation addresses only *high-level* aspects of path planning. In high-level route planning we assume that the moving agent is a point which has no physical dimension. Thus, there is no consideration of restrictions to movement due to the agent's width or height. This is a reasonable assumption when the map scale is of relatively low resolution as compared to the size of a human or a small vehicle. We have tested our algorithms on a 1000 meter grid square extracted from a standard 1:50000 scale military map [Defe74]. One unit of distance in our test maps is equivalent to approximately 10 meters distance on the ground. For a small vehicular agent, this is probably close to the maximum resolution before height and width constraints become active.

A high-level solution provides a general route for cross-country movement. The agent may still require a *low-level* path planning system to navigate the path. The low-level planner accumulates local sensory input and applies spatial reasoning to refine and safely follow the route as necessary for actual movement. Our work does not concern planning at this level.

The goodness of a high-level solution depends to a large degree on how well the map represents the realities of the terrain. The fact that WRP maps approximate curved features with straight lines is one source of representation error. Another source can result from cost coefficient assignment which is inconsistent with the actual cost per unit distance for movement on the ground. The mechanical aspects of our algorithm design assume that input maps are perfect representations of the terrain. However, of necessity, the representation of curves by straight lines normally approaches some finite precision (though possibly very small). Therefore, maps will usually differ from reality to some degree. Our design takes a pragmatic approach to optimization. There is little reason to expend large amounts of time searching exhaustively for a solution whose optimality may be questionable. In larger, more complex problem instances, there may be many near-optimal routes from which to select.

b. *Graph Search vs. Spatial Reasoning Approaches*

There are two general approaches to route finding: *graph search* and *spatial reasoning*. The graph-search approach represents the search space as a network of nodes and arcs through which a route must be found. The major advantage here is the ability to apply graph theory to obtain solutions. The very nature of many well-defined search problems dictates this discrete approach. Path constraints are the usual reason. For example, in city route planning for an automobile, clearly, there is no reason to consider off-road movement. In contrast, a spatial-reasoning approach considers the significance of shape, size, and orientation of features of the environment. The geometry of space and its effect on the cost of movement becomes more important.

It appears that a graph-search approach alone cannot sufficiently capture the spatial relationships inherent in a weighted-region problem (WRP) instance. Most of the previous work in the WRP makes heavy use of spatial reasoning to develop constraints and bounds on the locations and costs of optimal solution paths. Searching outward systematically from the start point, these constraints are continually tightened until only a few routes remain that can satisfy conditions for global optimality. Then, either a graph search or cost comparison selects the optimum.

Path annealing combines graph search and spatial reasoning in a more balanced mix. Whereas most of the previous algorithms focused on spatial reasoning to build a global solution, path annealing uses a fast, low-resolution graph search to get an initial solution. From this solution the algorithm probabilistically explores local spatial relationships among neighboring paths. Local graph search and the application of spatial constraints determine local optimal path-costs necessary to develop these relationships. The cumulative effect of all of this localized activity gradually determines a refined global solution.

c. *Discrete vs. Continuous Representation*

Optimization problems may be classified into two general categories – continuous and discrete [Papa82]. The discrete problems are also called *combinatorial optimization problems*. The solution techniques for each category are usually quite different and depend upon the form of the input. For example, *calculus* and *binary search* are two typical methods of continuous-variable optimization. Which to apply depends upon whether a continuous function is input as an equation or as raw data. On the other hand, standard graph theory provides discrete optimization techniques for finding shortest paths in networks of weighted arcs.

The WRP has both discrete and continuous characteristics. Two-dimensional Euclidean space imposes an uncountably infinite set of feasible paths between any pair of points, implying that continuous optimization might be applicable. However, the need to manage combinatorial complexity encourages a discrete flavor in problem representation. A finite set of boundary edges and vertices partitions the continuous space of weighted region maps. The geometric simplicity of the resulting polygons provides a natural way to manage the complexity and continuity of the WRP. We can divide a problem instance into a finite set of continuous *path subspaces* whose boundaries are defined by the discrete edges and vertices. Figure 2.1 depicts one such path subspace as defined by [Rich87].

The convexity theorem of [Rowe90b, Mitc90] lends further support for such a discrete representation. Simply stated, it concludes that for any path subspace with common start point and which crosses the same ordered sequence of boundary edges, weighted cost is a convex function of the parameters necessary to specify a path. The path subspace in Figure 2.1 satisfies the conditions of this theorem. The parameters necessary to specify each path are the coordinates of the start point and the boundary crossings. The theorem implies that there must exist within this subspace a unique path with minimum weighted cost. A continuous-function optimization technique (e.g. binary search) can determine this minimum cost path.

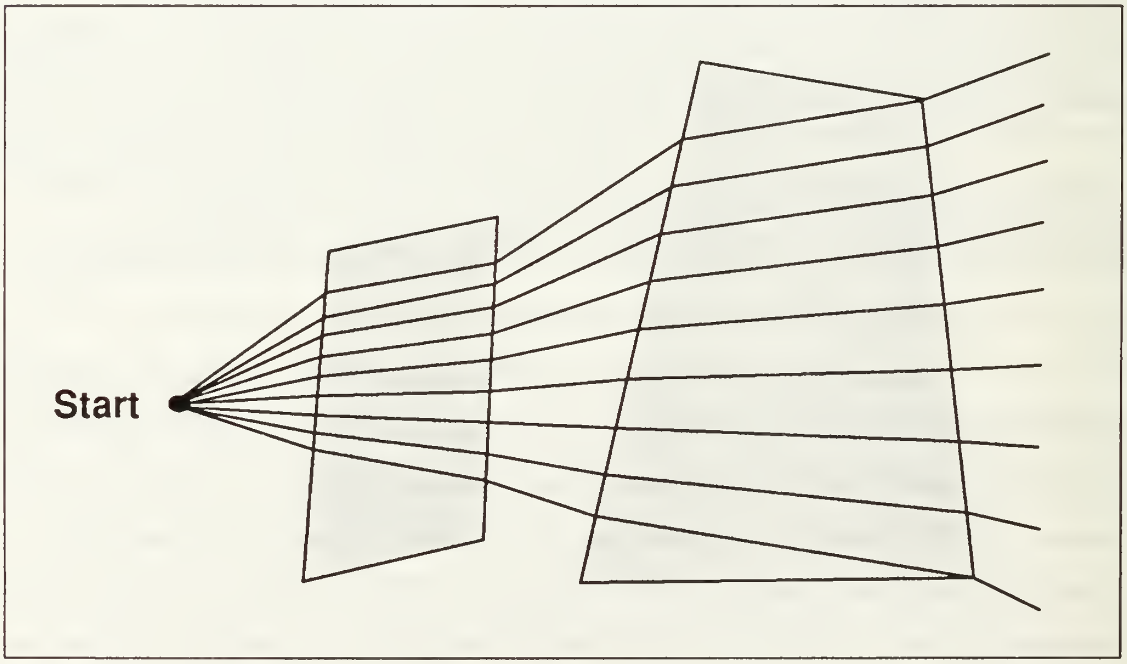


Figure 2.1 Well-Behaved Path Subspace (also called a *wedge*)

We can take advantage of the convexity theorem by allowing boundary edges and vertices to define a discrete set of path-subspaces between start and goal. Each subspace contains a continuous family of paths. By the convexity theorem we can obtain the locally shortest path within each subspace. However, there are other advantages. Discrete data structures are easy to implement and can efficiently support both graph search and spatial reasoning. Another advantage is that imposing a discrete structure on continuous space results in a combinatorial optimization problem. While the search space may be very large, it is nevertheless finite.

2. Sources of Error

a. Terrain Representation

Recall that in Section A.1.a of this chapter we briefly addressed terrain representation error. The solution accuracy for all current WRP solution techniques is dependent upon the accuracy with which straight lines approximate curves. This may have a direct influence on the size of a problem instance. The more straight lines necessary to maintain terrain representation accuracy requirements, the larger will be the number of vertices and edges in the map. The other source of representation error results from the use of homogeneous-cost regions. Of necessity, these regions are bounded areas. Their cost coefficient values are subjective averages, assigned because they best approximate cost per unit movement within that area from a high-level perspective. While errors will always exist, we assume that they will tend to cancel one another or average out over an entire high-level solution path.

We can quantify the effect of cost coefficient assignment error as follows. Let H represent the maximum difference between assigned and true cost coefficient values over all regions of the map. If μ_{max} represents the maximum cost coefficient, then for a path of unweighted length L, $HL\mu_{max}$ is the absolute maximum error in total path-cost induced by cost coefficient errors. This result assumes that the solution path uses regions of weight μ_{max} exclusively. But, the problem objective is to locate the least-cost path. So, the occurrence of the maximum error is very unlikely, because good solutions will use μ_{max} as infrequently as possible. For this reason, average error will be much lower. Also, consider that H may be a positive or negative deviation. Unless cost coefficient assignments are biased high or low, errors will tend to cancel over an entire solution path. The effect on total path-cost may then be minimal.

From a mathematical perspective our algorithm design assumes that input maps are accurate representations of the terrain. However, a probabilistic approach assumes a more practical perspective. Why conduct an extensive, time-consuming systematic search for an optimal solution whose accuracy depends on the accuracy of the map? A stochastic algorithm which performs controlled sampling can be faster, particularly for larger problem instances. Furthermore, we can design such an algorithm to be less sensitive to the accuracy of an input map. A search guided by cost function gradients in the space of complete solutions, can make extensive use of rapid approximations. It can choose to compute more carefully and accurately only when sampled costs indicate such a computation might be fruitful. A systematic search can also be guided by cost gradients. However, most of its time is spent examining partial solutions. Cost gradients of complete solutions contain more knowledge than those of partial solutions.

b. Total Path Cost

A computed solution path cost may not always be precise. Local optimality conditions are complex enough to admit irrational number solutions. As a result, many computed path costs are necessarily approximations. Furthermore, geometric problems like the WRP usually require many line intersections and point location procedures. This increases the likelihood round-off errors due to subtractions between near equal values as happens in computing the intersection of two near-parallel lines. Since solution paths are piecewise linear between such intersection points, then such round-off errors can accumulate, widening the gap between true and computed total path cost. It is often possible to identify and replace such calculations with more rational logic. For example, we can identify a parallel condition to a predetermined precision (six decimal digits for Quintus Prolog Version 2.0 [Quin90]), then explicitly select the point of intersection rather than compute it.

c. Path Location

A computed optimal path location (described by its linear segments) may differ from the true optimal path location. Such errors may be so small that the characteristic behavior (for example, the sequence of boundary edges crossed between start and goal) of both computed and true optimal paths are identical. On the other hand, path location errors can also be very large. Furthermore, large path location errors are sometimes caused by relatively small errors in optimal path cost. The reason is simple. A single WRP instance can have two distinctly different solutions paths whose total weighted costs are equal and globally minimum. The solution path returned by an algorithm will be that for which path cost was most favorable. Thus, if the average error in path cost is large relative to precision, then more solutions have the potential to be selected as optimal due to error. Assuming that only one in this set is the true optimal, then the probability of selecting it is dependent upon the level of noise.

3. Search Techniques

There are many ways to classify search techniques. Furthermore, unions and intersections of these methods form a variety of combinations. We find it most appropriate to discuss search under two general categories – systematic and local. When pertinent, we also mention important subclasses and variants.

a. Systematic Discrete Search

Globally optimal solutions of a WRP instance can only be recognized by demonstrating that no other feasible solution can have lower cost. This requires a search strategy which is both *complete* and *non-repetitious* [Pear84]. Completeness says that no optimal solution can be overlooked. Non-repetition precludes endless reevaluation of locations previously visited (as can happen in cyclic graphs). In general, a systematic search ensures both conditions.

Uniform breadth-first search is one such complete strategy [Barr81]. It expands from the start state in all directions accumulating cost at a uniform rate while maintaining a historical trail which links every child state back to its optimal parent. There is an implicit ordering of state expansion. As it is encountered, each new state is queued on an *agenda*. Thus, a roughly circular *search frontier*, represented by the agenda, begins at the start state and gradually expands outward as the search progresses. Explored states are marked as visited so that they can be identified if revisited. If a partial solution is extended to a state which was reached earlier, then the historical data is updated to point back along the shorter approach path. A search is said to be *uniform* with respect to cost if all partial solution paths are extended by the same cost increment. Uniformity guarantees that the least-cost solution has been discovered as soon as the goal is reached. There are two major drawbacks to uniform breadth-first search. First, it is a *blind* search because it does not use information about the search space or the problem instance to guide exploration. Second, the incremental cost step at which node expansions must occur is a function of the lowest cost coefficient. If the range of cost coefficient values is large, then the rate of expansion through higher-cost regions can be painfully slow.

In some problems execution times can be reduced through *bidirectional* search. In this variant, a search is conducted from both the start and the goal concurrently. If each search by itself is uniform, then when a path from the start reaches a path from the goal, their union is the least-cost solution path. Application of this technique requires *a priori* knowledge of the goal location. This being the case in the WRP, bidirectional uniform search can be applied to shorten the execution time [Rich87].

Branch-and-bound is one search variant which avoids some weaknesses of uniform search. This is considered a form of *ordered search* [Barr81], because there is an explicit ordering of state expansions. Instead of queuing states, branch-and-bound always expands the state whose current accumulated path cost is minimum. This ordering guarantees that an optimal solution has been found when a path to the goal with a cost that is equal to or lower than the minimum cost state currently on the agenda (i.e. the next state to be explored) is discovered. The first complete solution provides an upper bound on the least-cost solution. Intermediate search nodes whose cost equals or exceeds this bound cannot result in an optimal solution, and, therefore, can be pruned from the search. When the bounding mechanism can be activated, and how effective it will be, depends upon how quickly the first solution is found and how close it is to optimal. Using accumulated path-cost, branch-and-bound focuses its attention on paths which tend to have more potential for optimality. However, the measurement of this potential is one-sided because it is relative to the start state only. Unfortunately, when the search graph consists of weighted arcs, then expansion of states will not be uniform with respect to cost. Therefore, branch-and-bound cannot employ bidirectional search to find a least-cost solution.

A search* is a heuristic search variant which extends the concept of branch-and-bound by improving the measurement of optimality potential. Each state placed on the agenda is rated by the value of its *evaluation function* [Barr81]. The evaluation function is the sum of the cost to reach that state from the start plus an estimated cost from that state to the goal. At each stage in the search, the next state selected for expansion is the one which has the minimum evaluation. Furthermore, if the estimate to the goal is a guaranteed underestimate, then the search is called *admissible*. Admissibility guarantees the optimality of the first path reaching the goal. Since A* is more informed it often demonstrates marked improvement over uninformed strategies. Thus, if good cost underestimates are easy to compute, then A* may be preferred over branch-and-bound and uniform breadth-first search.

Clearly, we desire to employ the fastest, most efficient search strategy available. We note that both branch-and-bound and A* search elicit a price for their efficiency. Both require the maintenance of an ordered agenda in the form of a list or heap. Since an agenda contains the costs and locations of all partial solutions, it cannot be truncated before at least one complete solution has been found. Otherwise, information necessary to determine an optimal solution may be lost. Agenda control mechanisms can become unwieldy for large problems. Thus, if a good underestimate of cost from each state to the goal is not easily computed, or is not reasonably close to actual cost, then A* search may require more computing resources than simpler techniques [Rich87]. From a practical standpoint, the best search technique for a given problem ultimately depends upon a number of factors. These include problem domain, representation scheme, solution space, and the instances we expect to encounter.

b. *Local Iterative Search*

One of the most powerful and successful techniques for attacking difficult combinatorial optimization problems is *local search* [Papa82]. In continuous function optimization, local search techniques are referred to as *steepest descent* (minimization) or *hill-climbing* (maximization) [Barr81]. Researchers have applied these searches quite successfully to many NP-hard problems (for example, Traveling Salesman Problem [Lin65]). They are also valuable for some polynomial-time-solvable problems. For example, though linear programming is polynomially solvable, it can be solved efficiently and exactly by the well-known Simplex Algorithm, which is essentially a local search.

Given the set of all feasible solutions, F, local search requires an initial solution, $f_0 \in F$, and a mechanism for generating a *neighborhood*, N, of solutions f_i. The *neighborhood* of a solution f is a set solutions, f_i, which are "close" to f [Papa82]. In general, a solution f_2 is in the neighborhood of f_1, if starting from f_1, we can obtain f_2 by some small predetermined change to f_1. This change (or changes) is usually applied to some representation of the solution f_1 (for example, a simple swapping of two integers representing ordered points in space). How the f_i are generated determines the size and content of N. Beginning from f_0, a local search procedure applies and evaluates perturbations to the current solution in an attempt to find cost-improving configurations. In *iterative improvement* only cost-improving perturbations are accepted and the procedure repeats until no improvement can be found. At this point the current solution is locally optimal with respect to its current neighborhood. The drawback for many problems is that this local minimum may be far from the global minimum, with respect to both cost and physical location.

If the complete cost function of a problem is *convex*, then local search can always find the globally optimal solution. A function $f(x)$ is said to be convex if the following inequality holds for any pair of domain values, x_1 and x_2 [Baza77]:

$$\forall \lambda \in [0,1] \qquad f(\lambda x_1 + (1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda)f(x_2) \qquad \text{(Eq 2.1)}$$

Convexity implies that a local extremum is a global extremum. By successively searching the neighborhood of the improved solutions, a local search procedure eventually enters the neighborhood of the globally optimal solution. The speed of approach depends on the size of the neighborhood, and how quickly its local minimum can be found. In the worst case such a procedure can be very slow. But if the cost function of the solution space is a convex function, then faster techniques such as binary or Golden Ratio search can be used.

Not all problems exhibit the property of convexity. A solution space may consist of multiple local extrema. Taken as a whole, the solution space of the WRP is not convex. However, recall that it can be partitioned into convex subspaces. The cost of the local minimum in each subspace can be found by binary search and compared with each other to determine the global minimum. The difficulty with this is that the number of path subspaces is at least $O(2^v)$ where v is the number of vertices. Either a way must be found to organize these subspaces for systematic search, or a local search must be capable of escaping from local minima.

A significant issue in local search is the configuration of a neighborhood and, consequently, how it is generated. For a given neighborhood structure, when a locally optimal solution is also guaranteed to be globally optimal, then the neighborhood is said to be *exact* [Papa82]. If the functional value of a domain point is better than all other domain points within its neighborhood, then the point is locally optimal. If the neighborhood is exact, then this point is also globally optimal. For a convex function, any non-empty neighborhood is exact. However, for a non-convex function this is usually not true. Consequently, only very large neighborhoods may be exact. Unfortunately, in some non-convex problems the only exact neighborhoods contain all feasible solutions (for example, the Traveling Salesman Problem). In such cases local search will require the same or more work than systematic search. Therefore, from a practical point of view, it is sometimes better to compromise an exact neighborhood for an inexact one and concede the guarantee of global optimality. Using inexact neighborhoods we can also search locally from several initial feasible solutions. This can sometimes be more effective because it tends to broaden search space coverage. However, this does not ensure that the globally optimal solution lies in or can be located from any neighborhood of the initial solutions selected.

Another important issue is local search control, the method by which a neighborhood is searched. One common technique, *steepest descent* examines the entire neighborhood, N, of some solution and selects the locally best solution within N. This best solution has its own corresponding neighborhood, which is searched for *its* local extremum. And the process iteratively repeats itself. A more expedient method, *first-improvement*, searches the neighborhood as follows. Given a solution, neighbors are examined randomly or in some order until the first cost-improving solution is found. This solution is used as the basis for the next neighborhood. These two techniques generally represent the extremes in a spectrum of variations on neighborhood search [Papa82].

The nature of the solution space will likely influence the choice of a specific control procedure. When a neighborhood structure is relatively small and imparts a natural ordering to its solutions, then orderly examination of the neighborhood may be all that is necessary to locate the local optimum. However, for some problems exact neighborhoods may be so large that only partial search is practical. In such cases, random sampling within the neighborhood can be useful and more efficient. As in systematic search, heuristic information can often help to reduce local search. For example, in his work on the Traveling Salesman Problem [Lin65] recognized that it is sometimes possible to identify common features of good solutions. The heuristic is that some of these features are very likely contained in the global optimal solution. To apply this knowledge, a local search should tend to preserve these features in subsequent neighborhoods, thereby reducing the overall search effort.

The steepest descent approach to the WRP would encounter difficulty with discontinuous costs between path subspaces which occur at region vertices. The convexity theorem guarantees path-cost continuity for path subspaces which cross identically ordered sequences of boundary edges. It does not apply when the sequence of edges crossed changes, as occurs if a path subspace includes a vertex joining regions of unequal cost coefficients.

c. Relaxation

Many optimization problems reduce to finding the solution which best satisfies a large set of conflicting constraints, that is conditions required of the solution. A search technique in which constraints are iteratively added, removed, or replaced to improve a solution is called *relaxation*. The term has been applied to both continuous [Char87] and discrete [Rowe88] optimization. We say that a constraint is *active* when it is currently being enforced. Otherwise, it is *relaxed*. We may begin from one solution and activate one or more constraints which require that the original solution be changed to satisfy the new conditions. Another method begins by assuming that all solutions are possible. As constraints are activated, solutions which cannot satisfy them are eliminated. As more constraints become active they may or may not conflict. If they do, it may be necessary to backtrack, relaxing some and activating others. The entire process can be viewed as a state space search where states are solutions satisfying various constraint combinations and/or degrees of restriction.

d. Probabilistic Search

If neighborhoods or problem instances are too large for practical exploration, then a probabilistic approach to search becomes appealing. There are several forms of probabilistic algorithms. A *Monte Carlo* algorithm returns a solution which has some probability of being correct. A *Las Vegas* algorithm always finds a correct solution, but may require multiple executions to do so [Bras88]. Another form of probabilistic algorithm estimates a solution when an exact solution is not possible or is impractical.

Our approach to solving the WRP tends toward probabilistic estimation. We rapidly sample locally optimal solutions randomly by moving through neighborhoods consisting of convex path subspaces. The cost of a subspace is the cost of its locally optimal path. Cumulatively, the sampling covers a broad area of the solution space. Probability focuses local search on the neighborhoods with greater potential for containing an optimal path. Gradually, we reduce the ability of the algorithm to sample across a broad range of path subspaces. This is done by decreasing the probability of movement to cost-increasing solutions. Inevitably, the probability constraints tighten to the point of preventing movement to all but cost-improving solutions. When all cost improvements have been applied, the algorithm halts and delivers its best known solution.

While probabilistic algorithms cannot guarantee globally optimal solutions, the practical advantages can be significant. Probabilistic algorithms are inherently nondeterministic. Two execution sequences on the same problem instance will almost never be identical, although the final solutions can often be the same. Such behavior implies that for a given problem size algorithm performance does not depend on a specific problem instance. In probabilistic algorithms *expected time* of execution replaces average execution time over all possible instances of a particular size. For a given problem size, expected execution time is the same for all problem instances – worst cases included. In this sense, one can argue that probabilistic algorithms can exhibit more predictable performance than deterministic ones.

A typical behavior of many probabilistic algorithms is that the quality of their final solutions can improve over time. This can be a significant advantage when time resources are uncertain. Such algorithms might be terminated prematurely and still return an adequate result. The most obvious time advantage occurs because a probabilistic algorithm merely samples its solution space and uses statistics and heuristics to interpolate and infer knowledge. Often global or near-global optima can be found in significantly less computing time. Thus, a relatively small sacrifice in solution quality can deliver a disproportionately large increase in efficiency.

B. APPROACHES TO THE GENERAL WEIGHTED REGION PROBLEM

1. Grid-Based Wavefront Propagation

Wavefront propagation is an approximating method which employs a modified branch-and-bound search. An orthogonal grid of predetermined resolution overlays a weighted-region map. Each grid cell is assigned a cost coefficient. There are at least two ways to do this. We could use the homogeneous-cost coefficient of the region occupying the greatest area within the cell. We could also take the average cost coefficient weighted by the percentage of its area in the cell. The search begins from the grid cell containing the start point and continues until the cell containing the goal point is reached. A trail of back pointers permits optimal path retrieval. An agenda maintains the wave frontier and associated back pointers. Although simple, the algorithm has three major drawbacks. First, it is a blind search. Thus, without heuristics, it is painfully slow, particularly if the range of cost coefficients is large. Second, even maps with a few regions may require a relatively high resolution grid to accurately represent detail. Each cell of the grid is essentially a node in the search graph. Increasing the resolution of the grid by factor R roughly induces an R^2 increase in the number of search nodes, and, consequently, the size of the problem instance. Third, fixing paths to a grid causes "stair-step" solution paths. Assuming that resolution is fine enough to capture all terrain features modeled by the weighted-region map, an 8-neighbor implementation, may return solutions which are as much as 8% greater than optimal. Nevertheless, digital bias can be reduced [Mitc84], and heuristics can improve efficiency to some degree [Rich87].

2. Snell's Law

More accurate approaches to solving the WRP use Snell's Law as a local optimization criterion. Recall that cost regions are homogeneous. Thus, by the triangle inequality, optimal paths cannot turn at interior points of cost regions. It follows that turning points may occur where cost changes abruptly, that is, at cost region boundary edges.

Snell's Law defines the optimality conditions for such turns. Given a fixed point in each of two adjacent regions, the least cost path between them always satisfies the conditions of Figure 2.2, where θ_1 and θ_2 are region cost coefficients. This relation is analogous to that used in optics to determine how a light ray will bend when it passes between two media with different refraction indexes.

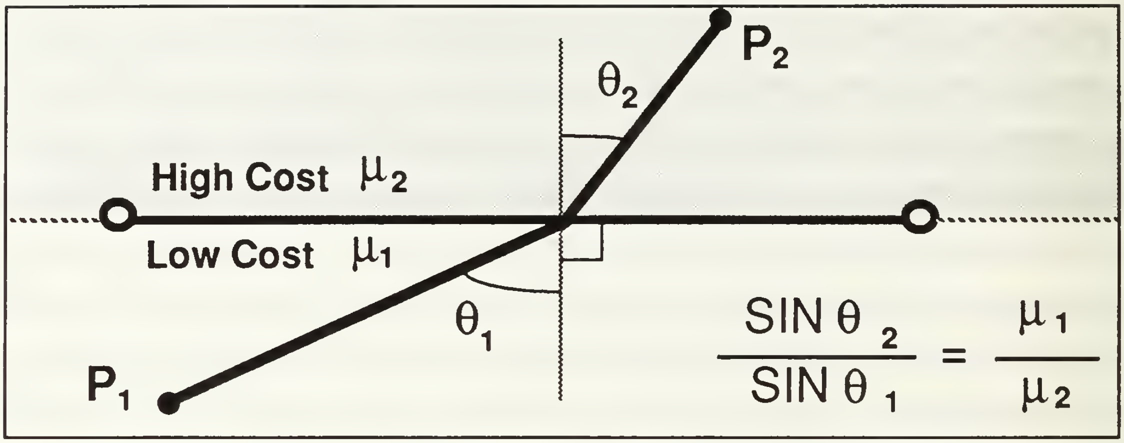


Figure 2.2 Snell's Law Optimality Criterion

The optical analogy is less applicable in the limiting case of reflection. Figure 2.3 illustrates the special case of a crossing episode which occurs at the Snell's Law *critical angle* associated with the higher cost region. An optical light ray reflects back into the medium from which it came when its angle of incidence exceeds the critical angle. Such a path would not be optimal in the WRP. To preserve optimality requires that the path move parallel and infinitesimally close to the boundary edge on the low cost side. Thus, the path incurs the lower cost per unit distance of travel on the edge.

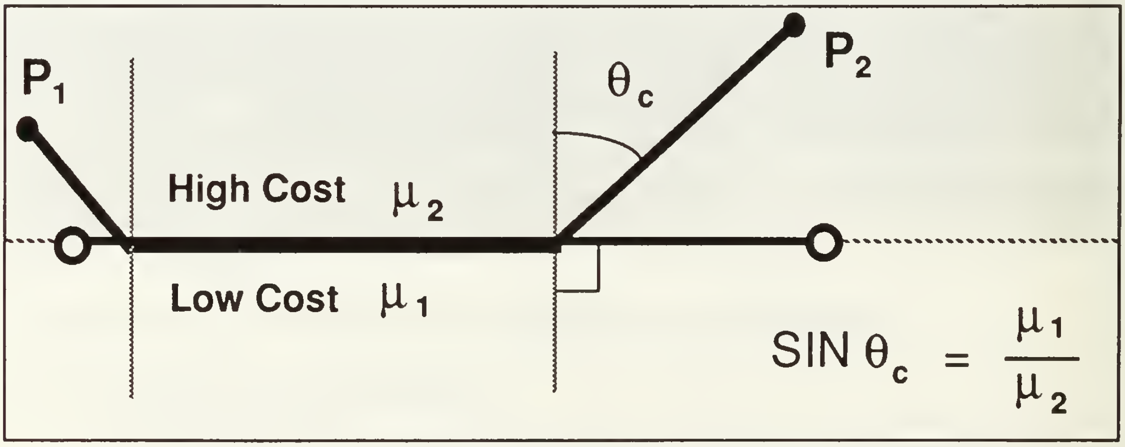


Figure 2.3 Snell's Law Critical Angle Crossing

Previous research suggests that Snell's Law is a powerful concept for the WRP. However, there are also some limitations and difficulties with its use. Snell's Law defines local optimization criteria only. It is a necessary but not sufficient condition for optimality at boundary crossings. Therefore, a globally optimal solution path must obey Snell's Law at every boundary crossing episode (except at vertices). However, a path which satisfies Snell's Law at every crossing episode is not guaranteed to be globally optimal. Depending upon how it is used, the relation may not be convenient to compute. While Snell's Law can compute required angles directly, there is no closed form for computing the coordinates of crossing points. Furthermore, it involves the sine function (or square roots in other forms), a relatively primitive but expensive operation.

3. Continuous Dijkstra Algorithm

The *continuous Dijkstra algorithm* [Mitc90] propagates Snell's Law constraint intervals in all directions from the start. A branch-and-bound procedure searches the graph representing these intervals. This algorithm finds the true optimal path, and establishes a worst case $O(v^7L)$ time complexity and $O(v^3)$ space complexity (where v is the number of region vertices and L is the number of bits used to represent input). While establishing that the WRP is polynomially solvable, the algorithm's current lack of implementation prevents practical comparison. Indeed, its seventh-degree-polynomial-time complexity and its requirement for triangulated cost regions, suggest that it will have relatively large average execution times.

4. Recursive Wedge Decomposition Algorithm

Another algorithm which finds an optimal path to the WRP is the *recursive wedge decomposition algorithm* implemented by [Rowe90b]. The algorithm propagates well-behaved sets of paths (referred to as wedges) from the start point, and recursively expands and refines them to construct a graph of shortest paths. These wedges are similar to the constraint intervals in [Mitc90]. A prototype implementation has demonstrated $O(v^2)$ average time complexity for relatively simple terrain (although worst case is still exponential). The algorithm tends toward its exponential worst case when a large number of wedges overlap the goal [Rich87]. It is a very complicated algorithm which uses trigonometric functions for many heading and visibility computations. Furthermore, its recursive nature can magnify the effects of these computations and increase the amount of agenda maintenance necessary as the size of a problem instance expands. Although, pruning heuristics do help to control these effects. The empirical performance of this algorithm in more complex terrain maps (with a wide range of cost coefficients and many contiguous regions) is unknown.

5. Local, Asynchronous, Iterative, and Parallel Procedures (LAIPP)

The only known work that does not use some type of systematic search is [Smit88]. The Local, Asynchronous, Iterative, and Parallel Procedures (LAIPP) approach is a form of relaxation. It is similar to a neural network simulation executed on a single processor system. A hypothetical processor is assumed at each boundary edge of the map. A boundary edge faces the two regions it borders. Its associated processor communicates with each processor assigned to the other boundary edges of the two regions it faces. Each processor tracks the locations of several path crossing points on its edge. For each point, the processor records what it believes is correct local and global weighted distance data relative to start and goal. The processors iteratively and asynchronously exchange and modify the data. Eventually all processors converge to an agreement on locally and globally optimal path data. At least one serial set of processors from start to goal has recorded identical (within tolerance) lengths for the solution path on which they lie. A systematic search of the discrete graph defined by serial chains of agreeing processors is searched for the shortest such path from start to goal. Although empirical results indicate that the procedure often converges to an exact globally optimal path, it is not guaranteed to do so. Furthermore, many locally minimal paths can be far from optimum. The empirical time complexity is measured as the number of iterations required for convergence to processor agreement. This complexity is slightly more than linear as a function of Euclidean path length. Time complexity is also dependent upon the range of cost coefficients used. LAIPP is an interesting and promising approach which we believe will prove most valuable in a parallel processing environment.

6. Optimal-Path Maps

An interesting approach proposed by [Alex90] attempts to solve a large number of WRP instances at once. For a single goal point location and the geometry of weighted regions, the plane can be partitioned into *behavioral regions*. These are areas in which optimal paths from all other points behave in a similar manner. For example, all optimal paths from start points in a given partition might travel directly to and pass through the same weighted region vertex enroute to the goal. Given enough preprocessing time, complete optimal path maps can be constructed. An efficient storage scheme will enable fast subsequent computing of any optimal path.

Two different approaches are proposed for constructing behavioral regions. The first is a specialized grid-based wavefront propagation algorithm modified to recognize points on behavioral region boundaries. These are recognized as the locations of wavefront convergence, points from which two (or more) distinctly different paths exist which return to the goal at identical costs. This algorithm suffers from the usual inefficiencies and inaccuracies associated with grid-based wavefront propagation.

The second approach is an elegant application of computational geometry. It is based on the observation that in a constant weighted plane (i.e. a single cost coefficient on the entire map) the optimal-path cost function originating from a common goal sweeps the surface of a perfect cone. The introduction of weighted regions perturbs this cost function, and, thus, the shape of this cone. In simple maps, the behavioral region boundaries formed by these perturbations can be computed as straight lines and conic section curves. The algorithm applies complicated parametric equations to find the conic curves. The computations are tedious and expensive, but the result is a complete and concise optimal path map from a given start (goal) to any goal (start) point desired.

To determine behavioral regions more efficiently, [Rowe90e] suggests modifications to the recursive wedge decomposition algorithm of [Rich87]. The algorithm's A* search strategy can be exploited to recursively partition a map into a tree of wedges rooted at the designated start point. A few conditions determine the possible locations of overlapping wedges. Standard polygon-intersection procedures can check these wedges for true overlap, and, if found, establish the shape of conic behavioral region boundaries therein. A prototype of this algorithm is currently under development. While this algorithm builds optimal-path maps more efficiently than those of [Alex90], the time investment is too costly for single start-to-goal paths.

Ideas presented in this dissertation could be used to improve the retrieval of optimal paths from the optimal-path-map storage structure proposed in [Alex90]. An optimal-path map is stored as tree of behavioral regions which are linked back to a common goal point (the tree root) by the vertices and boundary edges through which optimal paths must pass. Given a start point, its associated optimal path to the goal is found by tracing the ordered sequence of edges and vertices which are its ancestors in the tree. This sequence may require some limited optimization to obtain the exact path. Methods presented in this work can perform this optimization efficiently.

7. Minimum-Energy Paths

The work of [Ross89] and [Rowe90c] established a model for finding minimum-energy paths over terrain modeled by disjoint polygonal regions representing constant-slope surfaces. Such a terrain model is sometimes said to represent 2-1/2 dimensions because freedom of movement is restricted to a 3 dimensional surface. Follow-on work [Rowe90e] proposes an integration of both this model and concepts developed in [Rich87]. This research extends the generalized WRP to incorporate anisotropic homogeneous-cost regions.

The proposed unified algorithm solves a WRP in the 2-1/2 dimensional terrain model where region weights are coefficients of friction. An optimal solution in this problem is the minimum energy path from start to goal (or minimum time-travel path under constant power).

The original implementation of the algorithm partitions the map representation into subspaces in which families of paths exist. Our approach to the WRP uses a similar partitioning of the map. However, all of our WRP subspaces are naturally bounded by vertices and boundary edges. The model of [Ross89] requires the additional consideration of directional constraints which may further restrict the shapes and limits of subspaces. These constraints must be examined and correlated in order to form complete path subspaces from start to goal. This additional complexity is absent from the two-dimensional isotropic WRP.

C. SPECIAL CASES OF THE WEIGHTED REGION PROBLEM

Some applications may only require an easily solved special case of the WRP. We briefly consider two algorithms designed to solve instances in which terrain features representation is restricted. The algorithms are similar in that each constructs a form of discrete *visibility* graph from spatial constraints. The concept of visibility occurs often in movement planning. While its meaning may vary with problem context, *visibility* generally implies that at least one unobstructed-straight-line path exists between two points on a map. As the following special cases indicate, visibility analysis can often derive many powerful constraints on optimal paths.

1. Infinity, Zero-cost Regions, and Roads

The algorithm in [Mitc87] solves a special case WRP in which we restrict terrain features to four types: background, zero-cost regions, roads, and obstacles. As usual, obstacles are non-traversable areas of infinite cost. Zero-cost regions are areas in which travel incurs negligible cost regardless of distance (essentially, movement at infinite speed). Cost per unit distance in background terrain is taken as one. Although fixed, this value can be arbitrary. Roads are strictly linear features whose cost per unit distance is less than background cost. Given a start and goal, the algorithm computes a special visibility graph of all possible optimal path locations. These locations are determined through visibility analysis between vertices, and by applying the Snell's Law local optimality criterion. The resulting graph is provably finite, and can be searched using standard graph search techniques. The algorithm has worst case $O(n^2 \log n)$ time complexity, where n is the number of feature vertices. This is a significant improvement over the seventh-degree-polynomial time complexity of the continuous Dijkstra algorithm.

Our approach to the generalized WRP makes an important assumption related to visibility about the input map. The fact that all regions are convex (or can be made so with additional boundary edges) means that all edges bounding a given region are completely visible to one another without further calculation. This removes the need for visibility analysis between end points of path segments.

2. Roads, Rivers, and Rocks Algorithm

Additional work by [Rowe90a] extends the foregoing ideas to *rivers*, linear features with a higher cost than background terrain. While the work does not consider zero-cost regions, it is a valuable extension because roads, rivers, and obstacles can model a great number of commonly encountered terrain situations. Proof is given that optimal paths can only cross rivers at single points and, unlike roads, will never travel on them. Since the width of a linear feature is zero, then each river crossing episode incurs a fixed value, k, which is added directly to total path cost. The *RRR* (Rocks, Rivers, Roads) program is a Prolog implementation of the algorithm. RRR develops a special visibility graph similar to that in [Mitc90]. As a result, worst case time complexity remains $O(n^2 \log n)$. However, RRR also uses several spatial heuristics along with A* search to improve average case performance. The so-called *shortcut meta-heuristic* [Rowe90a] is proposed as the unifying concept for all optimal path constraints. Related to the optimality principle, the Shortcut Meta-Heuristic simply implies that a turning point on an optimal path can never be improved by shortcutting the turn. In our own work, we employ this concept to generalize optimal path constraints between regional boundary edges. Such constraints were first applied in limited form as heuristics by [Rich87].

D. SIMULATED ANNEALING

Simulated annealing is a search technique based upon the Metropolis Algorithm [Metr53], which simulates a complex system of particles (molecules) in a heat bath. Recognizing concepts similar to optimization, [Kirk83] and [Cern85] independently developed simulated annealing. Since then, many researchers have used it to obtain solutions to a variety of combinatorial optimization problems.

1. The Basic Annealing Algorithm

In its simplest form, simulated annealing requires the five basic components described in Figure 2.4. Starting from the initial solution and its evaluated cost, a *move generator* randomly perturbs this solution to obtain a new neighboring solution. The cost function of the new solution is computed and compared to the cost of the current solution. The new solution always becomes the current solution if the change in cost, ΔC, is an improvement. Otherwise, the new solution becomes the current with probability $P=\exp(-\Delta C/T)$. Figure 2.5 shows the form of the algorithm we use in this work [Pres88].

1. Solution space definition
2. S_0 - Initial solution taken from the solution space
3. Move generator to randomly generate small perturbations in solutions, thereby inducing a neighborhood structure to the search space
4. $C(S_i)$ - Cost function which can be evaluated for each solution
5. Annealing schedule to provide algorithm control:

T_0	Initial value of T
T_f	Freezing value of T
R	Reduction factor for T (typically $0.70 \leq R \leq 0.99$)
L	Maximum number of attempted moves at T
L_s	Maximum number of accepted moves at T

Figure 2.4 Basic Components of Simulated Annealing

```
initialize CURRENT starting solution
initialize T := $T_0$        /* $T_0$ starting temperature */
while ($T > T_f$) do       /* until T reaches $T_f$ freezing value */
    for i := 1 to L          /* for L attempted transitions */
        perturb CURRENT to generate NEW solution
        $\Delta C$ := cost(NEW) – cost(CURRENT)
        if ($\Delta C < 0$)
            then CURRENT := NEW
        else CURRENT := NEW with $P = \exp(-\Delta C/T)$
    end for
    T := R * T           /* decrement T by reduction factor R */
end while
return CURRENT solution
```

Figure 2.5 Standard Simulated Annealing Algorithm

The control temperature, T, is a value in the same units as the cost function. It regulates the probability distribution that defines the acceptance criteria of solutions with degrading costs. At each temperature, T, the procedure attempts up to L moves of which up to L_s acceptances are permitted. Then, the temperature is reduced by a factor R, and both L and L_s are reset. The resulting behavior is a downward-biased random walk through the solution space, with some ability to escape local minima. Referring to Figure 2.6, it is easy to see that gradually decreasing control temperature changes the exponential probability distribution. tightening the acceptance criteria against larger cost increases. The value of T for which no cost increases are reasonably expected and no more improvements can be found is called the *freezing temperature*, T_f. At this stage the algorithm returns the most current solution, and halts. As an added advantage, the algorithm can also maintain the best known solution and return this if it is better than the final current solution.

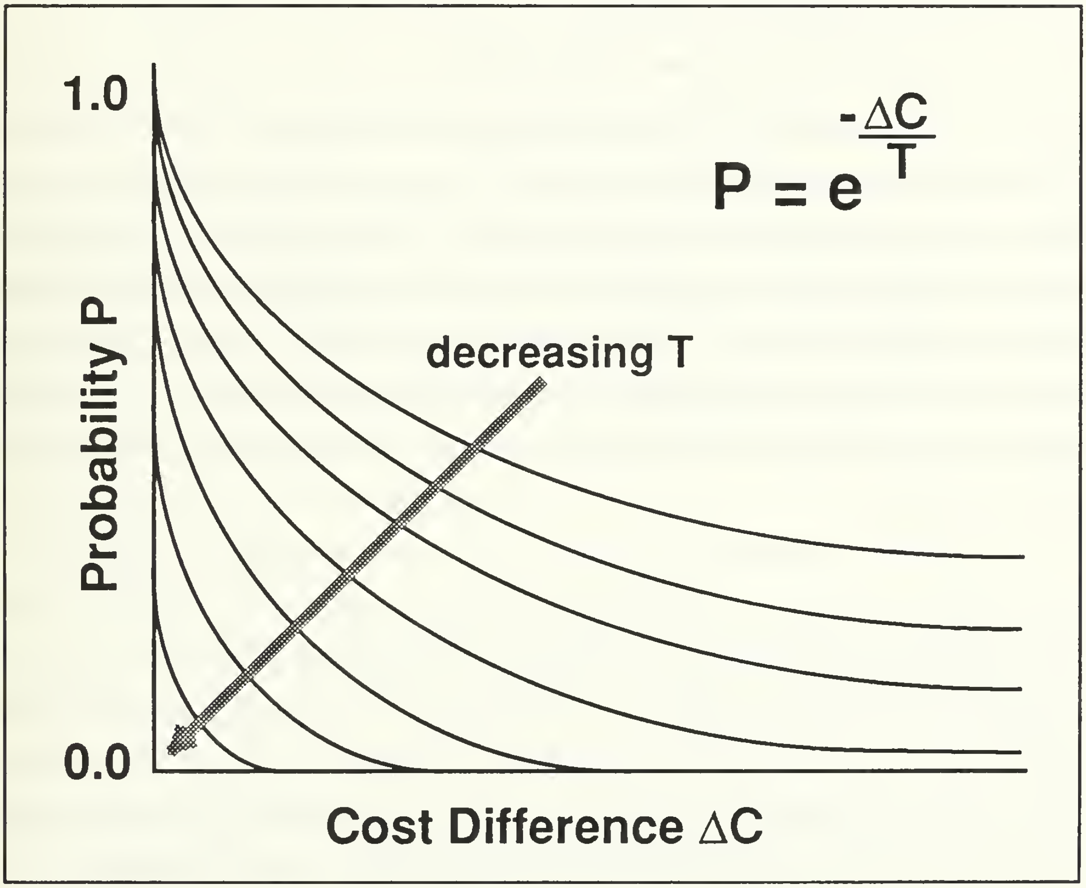


Figure 2.6 Probability of Acceptance vs Cost Difference ΔC

To enhance the performance of the basic annealing algorithm, researchers have developed and tested a variety of improvement techniques and heuristics. All of these methods are concerned with one or more of the basic annealing components. Since the focus of our work is in solving the WRP using annealing as a tool, we defer discussion of enhancements to Chapter IV. To the best of our knowledge, the work of this dissertation is the first (and only) investigation of a simulated annealing algorithm to solve the WRP.

In the sections which follow, we review some of the previous annealing work applied to problems with a path-planning flavor bearing some resemblance to the WRP. Our objective is to identify and discuss similarities and differences between these problems and the WRP as they relate to annealing. For annealing work on other problems and applications, [Coll88] gives an extensive annotated bibliography of simulated annealing theory and applications.

2. Routing Applications of Annealing

a. The Traveling Salesman Problem (TSP)

The classical Traveling Salesman Problem (TSP) is arguably one of the most investigated problems on record. Because it is NP-Complete, large instances of the TSP are likely to be intractable and require approximating algorithms to estimate the optimal solution. As a result, many researchers have explored annealing solutions to it. In addition to the original work of [Kirk83] and [Cern85], [Coll88] lists 40 references to annealing work on the TSP. Most of this work uses the TSP as a vehicle to investigate properties and enhancements to the annealing algorithm itself. While both TSP and WRP are essentially routing problems in two-dimensional Euclidean space, there exist significant differences which impact on the annealing approach.

The TSP admits a fairly straight forward annealing approach. In its classic form, a TSP instance consists of a finite set of discrete points in a two-dimensional plane. The optimal solution is a piecewise linear least-cost path which connects every discrete point with a straight-line path segment. The entire path forms a closed cycle. Thus, every feasible solution is well-defined, includes every input point once, and has a cost which is almost precisely determinable. These three aspects of the TSP make it an excellent annealing application. Solutions are nothing more than a fixed-length ordered set of points. A solution cost is the sum of Euclidean distances between ordered points. The move generator for the annealing algorithm is simple and efficient. To generate a new feasible solution, permute the point ordering of the current solution.

In contrast, an annealing approach to the WRP is more difficult. A problem instance is a map of regions, associated cost coefficients, and designated start and goal points. The number of feasible solutions between start and goal is uncountably infinite. Fortunately, the convexity of the Snell's Law optimality criterion can partition the solution space into a finite number of discrete path subspaces (recall the wedges in [Rich87]). Even so, computation of a single cost for each subset of paths can be expensive and highly dependent upon desired precision [Mitc90]. WRP feasible solutions usually consist of some subset of problem input; generation of new feasible solutions is not as simple as permuting a fixed-length list. Finally, recall that annealing requires an initial solution to begin. While a TSP initial solution is any ordering of the input, such is not the case for the WRP. A straight line path from start to goal is not guaranteed to be feasible in the presence of infinite cost regions. Thus, we must also resort to a more sophisticated method to initiate annealing for the WRP.

An annealing-based algorithm to solve a more general variation of the TSP is presented in [Adam85]. The problem is to plan an ordered cyclic route from a given starting point to a subset of predetermined locations on a two-dimensional map. The objective cost function is considerably more complex than simple Euclidean distance. Each location i has a corresponding value V_i which measures its worth to the overall plan. Subgoal values V_i can be dependent upon time and interdependent upon order of arrival. The objective function also employs the conditional probability, $P[G_{ij}|S]$, of reaching a subgoal i, given that the plan j is pursued and given the current state S of the autonomous vehicle (e.g. fuel supply, weapons status) and its working environment. Constraints of minimum acceptable probabilities and minimum time required to reach particular subgoals may also factor into the cost of a plan. The algorithm searches for a plan which maximizes overall utility defined as:

$$U_j = \sum_{ij} (P[G_{ij}| S] V_{ij}) \qquad \text{(Eq 2.2)}$$

One similarity between this problem and the WRP concerns cost function evaluation. Recall that in annealing, cost function comparisons between solutions drive and guide the process. Clearly, exact evaluation of each feasible solution is not reasonable in the algorithm of [Adam85]. Instead, the authors adopt a Monte Carlo method to approximate the cost of each solution, which significantly increases algorithm efficiency. A similar time-saving technique suggested by [Tove88] is the *surrogate function swindle*. If a cost function is difficult to evaluate, substitute one which is easier and approximates the original. In the WRP, Snell's Law enables us to compute locally optimal path costs. However the nature of Snell's Law makes the procedures for doing so relatively expensive. Consequently, we approximate their costs as often as possible, reserving the more precise computations only when absolutely necessary.

b. Wire Routing and Chip Placement

The design of very-large-scale integrated circuit boards requires the optimal placement of solid state chips and associated wire connections. Simulated annealing has been successfully used in the chip placement problem [Grov86, Sech86], a direct descendent of the NP-complete bin packing problem. Global wire routing is a path optimization problem somewhat closer in similarity to the WRP. While the problem context is a two-dimensional plane, wire routing solutions are usually drawn from a discrete space of predetermined routes. Ordinarily, an optimal arrangement of components predetermines all terminal points for connecting wires. An optimal routing of wires required between all terminal pairs minimizes total distance and the number of bends and overlaps [Vech83]. In global wire routing, the routes are normally confined to the orthogonal channels between components, so that the Manhattan distance metric applies and the solution space is discrete. Also, there is usually some upper bound constraint on the overlap capacity of each channel (i.e. the maximum allowable number of tracks per channel).

Like the Traveling Salesman Problem, the global wire routing problem lacks the continuous space aspect that complicates the WRP. However, wire routing exhibits a more complicated objective function. The objective function of [Vech83] attempts to model all constraints simultaneously. Annealing is conducted in two stages. The first permits only L-shaped paths (single bend) between terminal points. The second stage begins at the temperature from the last state of stage one, and includes both L and Z-shaped paths (two bends). Staging the optimization in two shifts reduces overall computing time yet achieves results comparable to those obtained when both path types are sampled throughout the entire annealing process. Similarly, [Sech86] employs a two-stage approach. However, the first stage is a non-annealing algorithm which selects wiring configurations that minimize only total distance. The second stage uses annealing to minimize total wire overlap by searching through the choices made by the first stage. Again, the result is savings of time.

A multi-stage approach is often applied to difficult problems. The strategy attempts to identify high-level characteristics common to most optimal or near-optimal solutions (for example, general shapes, common placements, etc.) that can later guide development and optimization of a more detailed solution. This strategy is formalized in [Whit84] by relating the scaling of moves to annealing temperature. At higher temperatures, larger random moves through the solution space are more efficient than smaller moves. Detailed solution improvements are discovered at lower temperatures, when smaller moves are most efficient. In this sense, simulated annealing develops a solution in a top-down manner. Evidently, because of this similarity, multi-staging seems to be a good general-purpose annealing heuristic.

Closely related to multiple stages is concept of the *target prejudice swindle* suggested by [Tove88]. This applies when general characteristics or properties of optimal solutions are known. The annealing process often saves time if the the move generator can be biased to generate neighbors with optimal solution characteristics. For example, in the WRP we know that optimal solutions never have cycles. Therefore, we should avoid generating such paths. In the extreme sense, this is what [Vech83] has done by biasing his first stage exclusively toward L-shaped paths. Another simple way to get the effect of the target prejudice swindle heuristic is to begin with a reasonably good initial solution. In the [Sech86] algorithm, the non-annealing first stage selects a large number of good starting solutions. This effectively provides the second stage (annealing) with a boost. Thus, stage two does not waste time examining poor solutions just to identify better ones.

We find that staging concepts are also applicable to our annealing-based algorithm for the WRP. Our current implementation finds one good initial starting solution from which to begin annealing. However, several initial starting solutions could improve the performance of the overall algorithm. Chapter III discusses how an initial solution is found. In Chapter IV, we suggest how to obtain multiple starting solutions might be obtained. Chapter IV also describes heuristics for preconditioning weighted region maps to improve path annealing performance.

c. *Robot Arm Movement*

The problem of efficiently and safely moving the end effector of a robot arm through three-dimensional space is another path-planning problem to which simulated annealing has been applied. The algorithm of [Lind87] uses an octree representation of space in which to plan movement for a manipulator of 5 or 6 degrees. Simple problem instances are handled by a deterministic algorithm. However, simulated annealing tackles the complex cases. Here again, we see a two-stage approach. However, in contrast to previously discussed applications, it is the first stage of this algorithm that uses annealing, not the second. The first stage finds solutions for movement of the first three links closest to the manipulator base, corresponding to coarse movement. The solutions are near-optimal in the sense that they minimize the link movements (torque and translation) required to translate the last three links to the target position in space. The second stage deterministically locates refined solutions for the links closest to the wrist.

Another example of an annealing-based algorithm for manipulator arm movement planning is found in [Carr90]. This work reports that reasonably good solutions are obtained from relatively simple annealing schedules. We have also found such to be the case in our approach to the WRP. Although we performed some very limited testing with more elaborate annealing schedules, the best results came from the simplest ones.

III. PATH ANNEALING DESIGN

A. INTRODUCTION

Path annealing is a probabilistic algorithm which combines a fast, crude systematic search with a non-exhaustive local search to find near-optimal solutions for the weighted-region problem. This chapter presents the basic design concepts for the path annealing algorithm.

In this chapter we introduce and define many terms and concepts specific to path annealing. Some of these concepts refer to supporting theorems which are presented only informally (with intuitive explanations as required for immediate understanding). A more formal statement of the theorems and associated proofs is contained in Appendix A.

Since we desired to implement a prototype of path annealing quickly, we chose to use Quintus Prolog Release 2.5 [Quin90] as our base language. There were many good reasons for this decision. Prolog code for several standard algorithms used in our work (such as A* search) was available through other related research projects. Generic Prolog is very high-level and provides automatic backtracking. Furthermore, Quintus Prolog provides automatic hashing, compilation, a foreign language interface, and graphic display capabilities. All of these facilities were used to develop, debug, and optimize the prototype. Appendix B is a complete listing of all source code.

B. STATE SPACE REPRESENTATION

1. Window Sequences

For a WRP instance define the *solution space* as the collection of all conceivable piecewise linear paths between start and goal, whose turning points (between linear segments) occur at regional boundary edges or vertices. We omit from this collection all paths with turning points which do not lie on regional boundaries because such paths can never be optimal. Furthermore, it is always possible to improve the cost of such a path by shortcutting any turning points which occur in region interiors. Note that we do *not* exclude cyclic paths, even though such paths cannot be optimal. WRP solution space contains an uncountably infinite number of solutions. The convexity theorem does not apply to the solution space as a whole unless all paths between start and goal must cross the same sequence of edges in exactly the same order (a possible, but very unlikely problem instance). Most problem instances contain many path-cost discontinuities. In order to manage such complexity we desire to represent this space in some discrete form.

Recall from Section II.A.1.c that the primary reasons for representing the WRP search space discretely are: (1) to utilize the cost function convexity theorem for paths which cross the same sequences of consecutive boundary edges; and (2) to make the problem suitable for combinatorial optimization. We assign a unique integer identifier to each region boundary edge. Furthermore, we assign identifiers *s* and *g* to designated start and goal points respectively. We now define the concept of *window sequence* (WS). This concept was originally adopted by [Ross89] to describe a family of paths which cross the same sequence of boundary edges and vertices and which must obey the same set of movement constraints. Our definition of window sequence is similar, but not identical. Consider the continuous, infinite set of all paths between the start and goal points which cross the same ordered sequence of edges. We can uniquely label this family of paths using an ordered list of edge identifiers. The first and last elements of this list are always *s* and *g* corresponding to start and goal respectively. Such an ordered set is called a *window sequence.* Figure 3.1 is an example of a window sequence (WS). Duplicate edge identifiers are permitted within a legitimate WS. This simply means that a family of paths must cross the same edge multiple times. Also, note that a WS must include every edge identifier that its associated paths cross. Identifiers may not be skipped in a valid WS list.

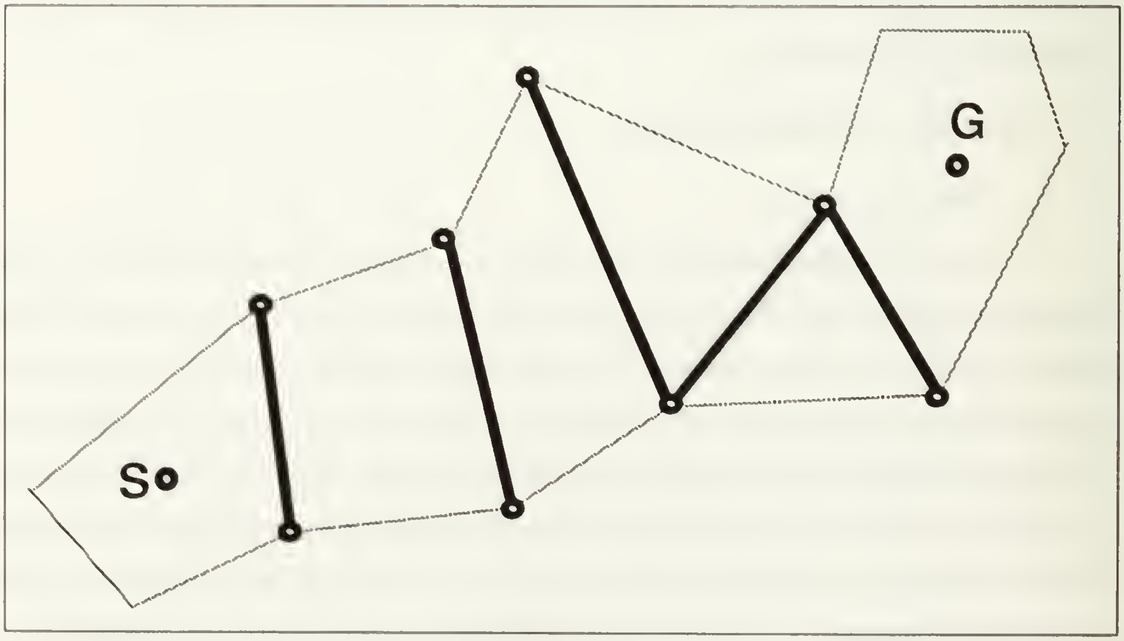


Figure 3.1 Window Sequence

Note that a WS is not the same as a wedge [Rich87] as in Figure 2.1. A wedge has a start but no terminating point, and consists of only Snell's Law paths defined between extreme left and right bounding paths. In contrast, a WS has both start and terminating points and consists of *all* paths (regardless of Snell's Law) which cross the given sequence of defining edges. Nevertheless, the convexity theorem of [Rowe90b] and [Mitc90] still applies to WS because the theorem places no restriction on edge length and the path-cost function remains the same. Therefore, a unique locally optimal path must exist within each WS, and a standard continuous function optimization method can be used to locate it.

Window sequences partition the WRP solution space into possibly overlapping classes of paths. We can identify a number of desirable properties which emphasize the advantages of a window sequence representation:

- All WS's are discretely represented by ordered lists of edge identifiers.
- Assuming a bordered map, the total number of WS's is finite.
- Every path in the WRP solution space is contained in some WS.
- Some WS contains the globally optimal path.
- Consecutive turning points between path segments are always visible.

Though finite, the number of states (i.e. window sequences) in this space is very large. However, if we can find the locally optimal path of any WS, then we have a means of comparing WS's. By searching the solution space of WS's we can use these costs to determine optimal or near-optimal paths.

Recall from Section I.B that we assume all regions in a WRP map are convex (with the exception of the non-traversable border region). The reason is that a convex-region map subsumes the definition of edge visibility. Consecutive edges in a WS are always completely visible to one another, and so, visibility computations are unnecessary. This improves the efficiency of determining the locally optimal path in a WS.

2. Primitive Data Structures and Operations

In order to form a high-level representation of the solution space we preprocess a WRP map into several primitive data structures designed to facilitate rapid navigation through the space at lower levels. Our input maps are just a list of boundary edges with defining vertices and associated cost coefficients to either side. It is assumed that this list represents a complete set of straight-line segments in a planar graph. Henceforth, we refer to this graph as a *weighted-region map.* The input description of each edge in the map conforms to a standard convention. Boundary edge, E, is defined by the set $\{v_1, v_2, \mu_1, \mu_2\}$, where v_1 and v_2 are the coordinates of its terminating vertices, and μ_1 and μ_2 are the cost coefficients of the regions it bounds. An agent traveling along this edge from v_1 to v_2 will always see μ_1 on his left and μ_2 on his right, when μ_1 <

μ_2. However, if $\mu_1 = \mu_2$, then the x coordinates of the vertices are related as $v_1(x) < v_2(x)$. If $\mu_1 = \mu_2$ and $v_1(x) = v_2(x)$, then $v_1(y) < v_2(y)$.

Map preprocessing creates and assigns a unique integer identifier to each edge, and stores it together with defining vertices, associated weights, linear equation form, midpoint, length, bounded regions, and directional data. This provides extremely efficient access since Quintus Prolog [Quin90] hashes predicates on the first atomic argument, in this case the edge identifier. Thus, given a WS of edge identifiers, its vertices, regions, and related data are all immediately accessible.

In addition to the edges themselves, there are two special structures which store knowledge about spatial relationships between edges. We create an *edge link* for each pair of edges which end in the same vertex and are adjacent around the same common vertex. For each such edge-pair, the edge link records appropriate edge identifiers, the common vertex, and the cost coefficient of the common region bounded by the pair. We also create an *edge dual-graph* which is a high-level data structure representing all WS's. Because it is high-level, we defer discussion of this data structure to the next section.

Since we desire quick access among edges, vertices, and regions, preprocessing creates several data structures which are essentially inverted files indexed on these entities. For each vertex in the map, a *vertex edge-list* is maintained. This data structure stores a clockwise ordered list of edges incident around each vertex. Two similar data structures are constructed for regions. A *region edge-list* holds the associated cost coefficient and the ordered list boundary edges for each convex region. A *region vertex-list* stores the ordered list of vertices around each region. As will be explained later, obstacles are treated as though they were special oversized vertices. Similar to a vertex edge-list, an *obstacle edge-list* stores an ordered list of the incident edges around each obstacle.

The geometric nature of the WRP requires a large number of floating point operations. Unfortunately, Prolog was not designed for efficient number crunching. Furthermore, Quintus Prolog Release 2.5 floating point operations are accurate to only 6 decimal digits of precision. To compensate we implemented many primitive routines in C, (see Appendix B). particularly floating point operations. These are called as Prolog predicates through the foreign language interface. Common routines implemented in C include: weighted and unweighted distance computations, random number generation, linear interpolation, routines to implement a heap data structure, Golden Ratio and binary search routines, and annealing schedule control.

A few primitive geometric routines are implemented in Prolog. However, we have intentionally written these procedures to execute simple arithmetic operations only, avoiding square roots and trigonometry whenever possible. We have also compressed the mathematics in these routines to as few operations as possible. Line intersections are taken by solving simultaneous linear equation forms. Perpendicular projections are done similarly. For efficiency and accuracy, special cases (e.g. intersections of parallel or coincident lines) are intercepted by pattern matching of coordinates and line equation coefficients. Then, instead of permitting calculations which might result in great loss of precision, we attempt to solve them through spatial reasoning about local geometry. Directional heading computations require a single arctangent operation at one stage, and otherwise they involve only arithmetic operations. Again, special cases are intercepted and handled separately by spatial reasoning.

C. INITIAL SOLUTION

Annealing can begin from any solution in the space. However, empirical evidence suggests that reasonably good initial solutions can often provide better final results [Naha86, John89]. The obvious straight-line path from start to goal is not always feasible when obstacle regions are present. While there are good algorithms for constructing and searching a visibility graph, this is more work than is necessary. To initialize path annealing, we conduct a limited search through a special representation of the WRP input map.

1. Edge Dual-Graph

A connected graph embeddable in the Cartesian plane without intersecting arcs and whose vertices are minimum degree two is known as a *planar graph*. Restricting the arcs of a planar graph to straight lines results in a *planar straight-line graph* (PSLG) [Prep87]. Our WRP input maps are PSLG's. We further restrict these maps to consist of convex regions only. For all such structures, graph theory defines the notion of a *dual graph*. Regions in the planar graph become vertices in its dual graph; whereas vertices in the planar graph become regions in its dual graph. In the dual graph arcs connect vertices corresponding to adjacent regions in the original graph. Thus, a one-to-one correspondence exists between arcs in the planar graph and arcs in its dual. While this kind of dual graph is often a valuable problem transformation (e.g. Min/Max Cut Problem), it is not that useful for our purposes. Instead, we construct a dual representation for a WRP map that differs from the standard region dual-graph described above.

We define a dual representation of a WRP map called the *edge dual-graph*. The discussion which follows refers to two distinct representations of a problem instance – the weighted region input map and its corresponding edge dual-graph. In order to avoid confusion, we will use the terms *vertices, edges, regions,* and *map* to describe a weighted region map. We will use the terms *nodes, arcs,* and *graph* to describe the edge dual-graph. The edge dual-graph is essentially an adjacency list representing the spatial structure of the map at a relatively high level of abstraction. To create this graph, we assign a node to the midpoint of each map edge which does not bound an obstacle (or the border). Special nodes are assigned to the start and goal points. In each non-obstacle region, we add arcs to connect all nodes representing the edges which bound the same region. The fact that all regions are convex guarantees that all such arcs cannot intersect obstacles or other regions. Thus, each region contains a clique consisting of all nodes representing its non-obstacle boundary edges. The regions containing the start and goal points will have additional arcs connecting the start or goal node to the clique. Figure 3.2 illustrates a simple WRP map with its associated edge dual-graph.

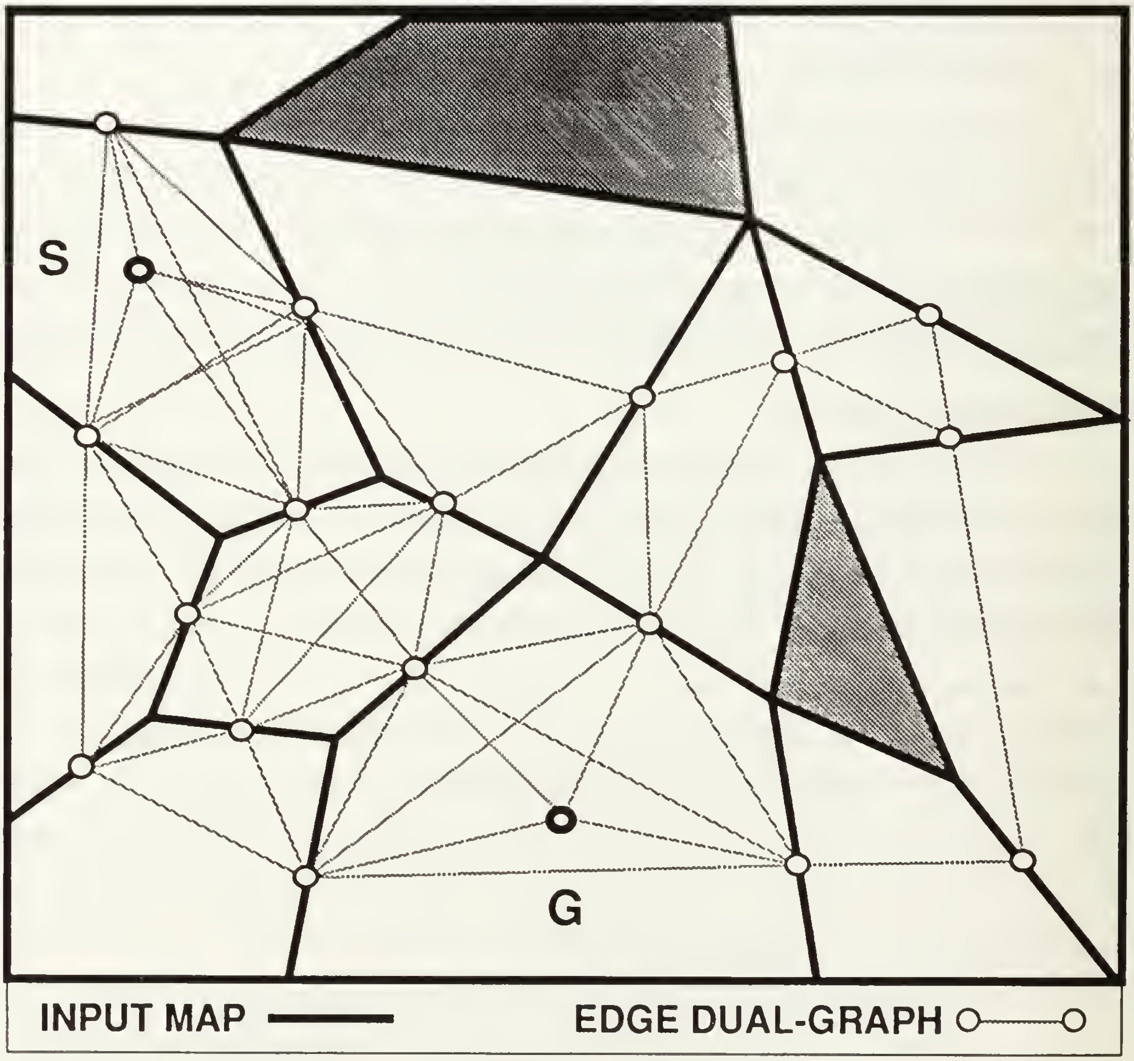


Figure 3.2 Weighted-Region Problem Map and Edge Dual-Graph

2. Search for an Initial Solution

We obtain the initial solution by conducting an A* search of the edge dual-graph. The cost function is the weighted Euclidean distance. The estimated cost to the goal is straight-line distance to the goal weighted by the map's lowest cost coefficient, μ^*. The is an underestimate to the goal, because it is weighted by the optimal cost. Therefore, the search is admissible, and returns the shortest weighted midpoint path, and the corresponding WS in which it resides. We refer to this path as the *optimal midpoint path.* Note that we should *not* expect this midpoint path to be globally optimal; nor should we expect that its corresponding WS necessarily contains a globally optimal path. However, the optimal midpoint path does represent a reasonably good initial solution.

Simulated annealing does not require the optimal midpoint path from which to begin. We could begin with any feasible solution regardless of its cost. So, we could overestimate distance to the goal, thereby inducing a greater depth-first component to the search [Pear84]. Such an evaluation function usually finds a solution faster. The resulting midpoint path might be of higher cost, but nevertheless, will be feasible since the edge dual-graph does not pass through infinite cost regions. However, there are two advantages to initializing with admissible A* search. First, the resulting midpoint path is a reasonably good starting solution. It is an indication of a high-speed avenue with a higher potential for containing (though not necessarily) the optimal or a near-optimal path. Much empirical research indicates that simulated annealing is faster when provided with reasonably good initial solutions [John89, Naha86, Bono84]. Second, the total cost of this path provides an upper bound on the cost of the globally optimal path. Since we want the least upper bound obtainable, then it is desirable to find the best midpoint path.

D. COST FUNCTION EVALUATION

In the main phase of annealing, each newly generated WS must be evaluated to find the cost of its locally optimal path. In many annealing applications the cost of a sample solution is easily evaluated. But in path annealing, cost function evaluation is difficult and may require a relatively large amount of computing time. Thus, streamlining this procedure is a high priority.

Recall that the convexity proof implies each WS contains a unique shortest path [Rowe90b, Mitc90]. It is possible to find this path (within some precision) using Snell's-Law ray-tracing and a standard iterative search technique such as *false position* [Rich87]. By assuming that each edge in the WS is an infinite length line, we can treat a WS similar to Richbourg's wedge. However, the left and right boundaries defining a wedge are Snell's Law paths themselves. In contrast, the left and right boundaries of a WS are lines through the terminating vertices of the edges which define the sequence. As a result, a WS may have no Snell's Law paths within it at all. We find that locally optimal path for many WS's is constrained by, and therefore passes through one or more of the vertices of the defining edges. An iterative ray-tracing approach to WS optimization can expend much computing time backtracking to false starts caused by the ray failing to intersect edges within the WS. In the next section, we describe WS optimization techniques which take more advantage of the nature of WS's.

1. Single-Path Relaxation (SPR)

Because Snell's Law determines the local optimality criteria for path segments that cross boundary edges, it is possible to find the shortest path constrained to a WS without iteratively computing total path-cost. Given an arbitrary WS from start to goal, one can locate the optimal path constrained to the defining edges using the *coordinate descent* method proposed by [Smit88] (also suggested by [Mitc90]). Beginning with a path through the midpoints of the edges, consider any set of three consecutive points P_1, P_2, and P_3. Fix the outside pair, P_1 and P_3, and use *Golden Ratio* search [Press88] to adjust P_2 to the point on the middle edge at which the weighted distance of path P_1-P_2-P_3 is minimized. The point P_2 will be located on the middle edge such that Snell's Law is satisfied relative to P_1 and P_2. Applied to any set of three consecutive points along a path through a WS, this procedure cannot increase total path-cost. Because of this, we may apply the procedure iteratively from start to goal. Beginning with the first set of three consecutive points which includes the start point as P_1, adjust P_2 to its optimal location, and slide to the next set of three points. This process continues to the goal point, then reverses direction moving back to the start. Passes continue in alternating directions until one complete pass in either direction is made through all points, during which no point moves from its previous position more than some precision ε, referred to as *relaxation tolerance*. The procedure converges on the locally optimal path because total weighted path length is a monotonically decreasing function of the sequence of adjustments [Smit88]. It is also possible to pass through the WS continuously in the same direction (such as start to goal). However, we have found that moving in alternating directions tends to cause faster convergence.

Simulated annealing cannot afford expensive primitive operations. Smith's coordinate descent method avoids the direct use of Snell's Law and expensive trigonometric functions, which is good. Also, the technique makes good use of the bounding vertices of the edges. However, simple testing of coordinate descent using Golden Ratio search suggests that Smith's method, though accurate, is too slow to be of practical value in path annealing, since long WS's can require a very large number of iterations. Furthermore, wide window sequences with long edges can have very slow convergence to the locally optimal path. Therefore, we investigated faster alternatives.

Direct determination of a starting direction for a ray-trace which must cross several boundary edges to hit a known target point is very difficult. Combining the system of equations representing the Snell's Law criterion at each boundary crossing, results in a very high-degree-polynomial equation, whose solution yields the starting and target-point coordinates [Mitc90]. However, if only a single boundary edge separates the starting point and the target, then we only need to determine the point of optimal crossing on the boundary edge between start and target. Snell's Law has no closed form to compute this point directly. This is one reason that [Smit88] resorts to Golden Ratio search. Nonetheless, if the range of region cost coefficient values is known, then we can directly determine optimal crossing points with a very fast table-lookup procedure. We employ this procedure in the coordinate descent technique described earlier by replacing Golden Ratio search with a constant-time Snell's-Law-table lookup. We refer to our method of coordinate descent using table-lookup as *single-path relaxation* (SPR).

The problem of finding a Snell's Law crossing point on a boundary edge between two fixed points requires a three-dimensional table. Recall from Chapter I that by rescaling rational cost coefficients [Mitc90], it is always possible to establish integer cost coefficients for a WRP. To construct an appropriate Snell's-Law lookup table, it is necessary to determine the expected working range of cost coefficient ratios *a priori*. By *working range*, we mean those encountered most often in maps of interest. Our implemented table uses integer ratios. It is not necessary for the set to include every ratio to be encountered; out-of-bounds conditions can always trigger a more expensive Golden Ratio search. However, our objective is to severely reduce the number of times a lookup must resort to iterative search.

Given fixed approach points P_1 and P_2, we normalize the geometry of a typical *crossing episode* on a standard rectangle as shown in Figure 3.3. Note that we orient this rectangle such that the crossing edge is always parallel to the x axis. Four parameters characterize every crossing episode geometrically similar to this rectangle:

$\rho = \mu_2/\mu_1$ (ratio of cost coefficients) (Eq 3.1)

$r = x_t/y_t$ (inverse of the slope of line P_1P_2) (Eq 3.2)

$y' = y_0/y_t$ (ratio of y coordinates of P_0 and P_2) (Eq 3.3)

$x' = x_0/y_t$ (ratio of x coordinate of P_0 to y coordinate of P_2) (Eq 3.4)

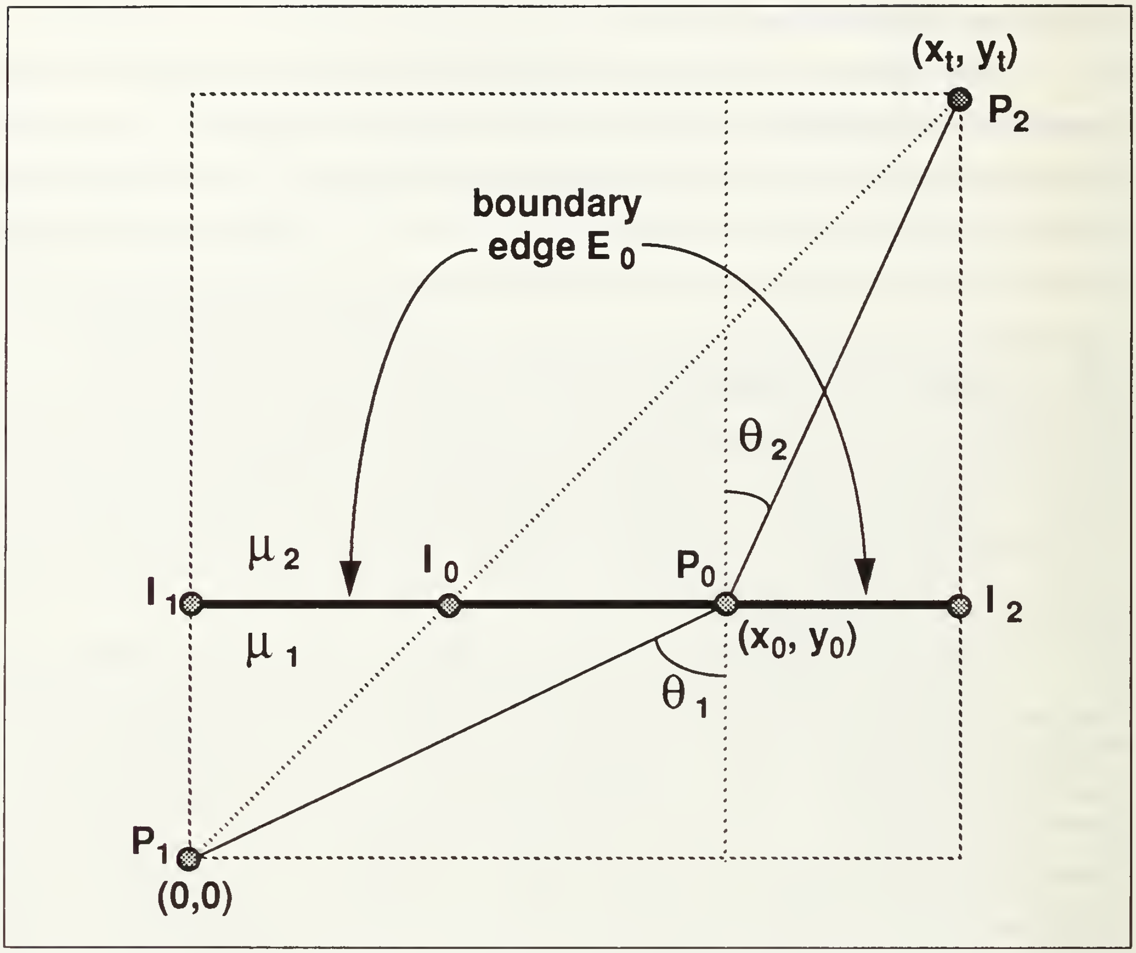


Figure 3.3 Crossing Episode Geometry

The mathematical basis of the table is derived as follows. For optimal crossing point P_0, Figure 3.3 obeys Snell's Law:

$$\mathrm{Sin}(\theta_1) / \mathrm{Sin}(\theta_2) = \mu_2 / \mu_1 = \rho \qquad \text{(Eq 3.5)}$$

Letting $\rho = \mu_2/\mu_1$ and applying trigonometric identities we derive the equivalent expression:

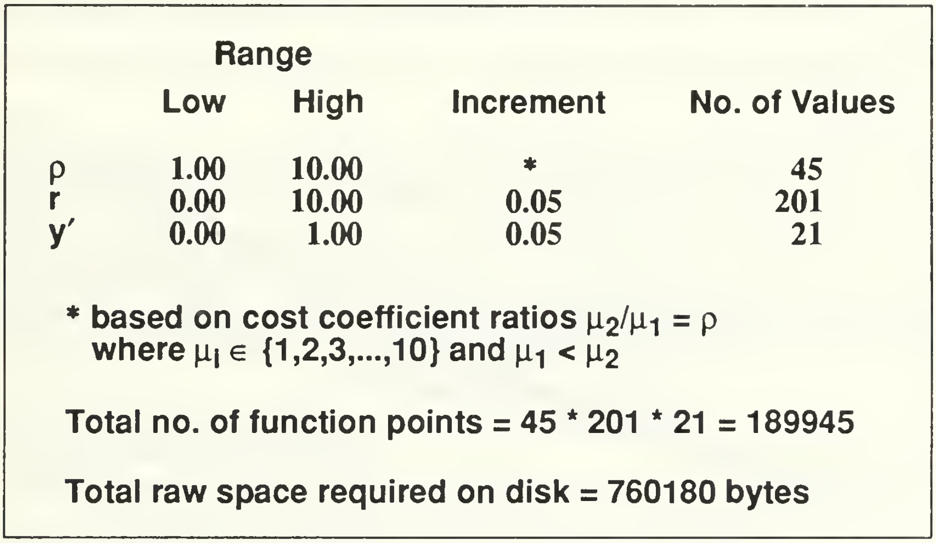
$$\rho = \mathrm{Cos}(\mathrm{ArcTan}(y_0/x_0)) / \mathrm{Cos}(\mathrm{ArcTan}((y_t-y_0)/(x_t-x_0))) \qquad \text{(Eq 3.6)}$$

Multiplying each arctangent argument by $(1/y_t)/(1/y_t)$ and substituting yields a parametric equation in four variables, where ρ, r, and y' determine x':

$$\rho = \mathrm{Cos}(\mathrm{ArcTan}(y'/x')) / \mathrm{Cos}(\mathrm{ArcTan}((1-y')/(r-x'))) \qquad \text{(Eq 3.7)}$$

where $x' = x_0/y_t$, $\rho = \mu_1/\mu_2$, $r = x_t/y_t$, and $y' = y_0/y_t$.

For any given crossing episode, the values of ρ, r, and y' are known. Golden Ratio search precomputes the table values of x' which satisfy Eq 3.7 for discrete values of the other parameters. Figure 3.4 tabulates the parameter ranges and key characteristics of our prototype Snell's-Law lookup table.



	Range			
	Low	High	Increment	No. of Values
ρ	1.00	10.00	*	45
r	0.00	10.00	0.05	201
y′	0.00	1.00	0.05	21

* based on cost coefficient ratios $\mu_2/\mu_1 = \rho$ where $\mu_i \in \{1,2,3,...,10\}$ and $\mu_1 < \mu_2$

Total no. of function points = 45 * 201 * 21 = 189945

Total raw space required on disk = 760180 bytes

Figure 3.4 Characteristics of Snell's-Law Lookup Table

Recall that all region weights, μ_i, are drawn from the integer set $\{1,2,..,10,\infty\}$. We use this set to indirectly index the proper value of ρ in the lookup table. Note that it is always possible to orient the geometry of an episode such that $\rho \leq 1$ is true. Therefore, we employ a two-dimensional triangular array of 45 entries, indexed by integer μ values in numerator-denominator pairs. The array holds the indices to the exact values of x' in the lookup table.

Consider the generalized crossing episode in Figure 3.3. From fixed points P_1 and P_2 we can easily determine intersections I_0, I_1, and I_2, and associated distances relative to the edge, E_0. With this data the lookup procedure computes the parameters r and y'. The parameter, ρ, is exact and indexes the appropriate plane in the table. Using r and y', the indices are created to bracket the appropriate value for x' within a two-dimensional rectangle. Simple linear interpolation determines a value for x'. This value gives the location of the optimal crossing site on E_0 relative to both y_t, and its left vertex. Since $x' = x_0/y_t$, then the location of the optimal point relative to x_t is $x_0 = x'y_t$.

The location of the optimal crossing point on E_0 assumes an infinitely extended boundary edge. However, all boundary edges are actually finite length segments. Therefore, the optimal crossing point may not always lie between the vertices which define the boundary edge. In such cases, the optimal crossing point will be the closest vertex. We refer to these occurrences as *constrained crossings*, because the optimal crossing must violate Snell's Law (as little as possible) to satisfy restrictions of boundary edge length. A constrained crossing is illustrated in Figure 3.5.

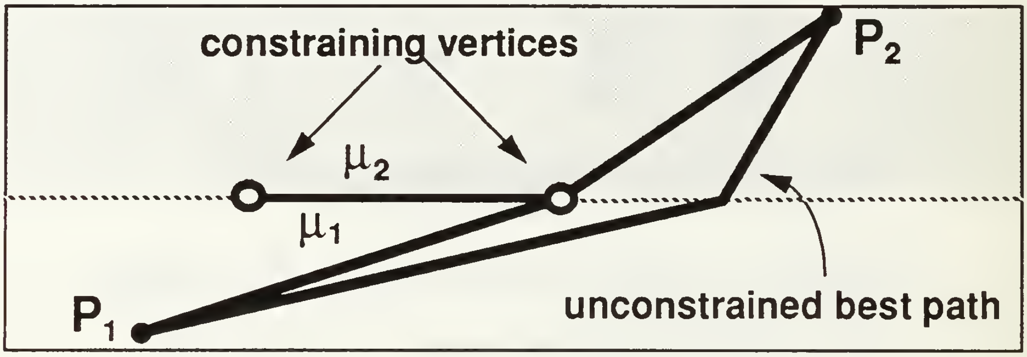


Figure 3.5 Constrained Optimal Crossing

By always orienting the crossing edge E_0, fixed points P_1 and P_2, and cost coefficients μ_1 and μ_2 as shown in Figure 3.3, we can detect many constrained crossings before applying the complete table-lookup procedure. Note that in Figure 3.3 an unconstrained optimal crossing can only occur between intersection points I_0 and I_2. If both vertices of the crossing edge lie outside of the interval defined by I_0 and I_2, then a constrained crossing is certain, and table-lookup is unnecessary. The vertex closest to the I_0-I_2 interval is the optimal crossing site. Otherwise, if a table-lookup is still necessary, the location of the optimal crossing point returned by the lookup is compared to the location of edge vertices to determine if a constrained crossing is optimal.

In summary, the work to lookup x' requires three intersections, three distance calculations (to determine x_t, y_t, y_0), and one linear interpolation in two dimensions. Thus, a single lookup can be done in constant time. On rare occasions, if the geometry of a crossing episode causes the parameters to fall outside the bounds of the table, then the lookup procedure defaults to Golden Ratio search. The major steps of the optimal-crossing-point lookup algorithm are outlined in Figure 3.6.

```
Input:
  fixed points $P_1$, $P_2$
  crossing edge $E_0$ (defined by vertices $V_1$, $V_2$)
  cost coefficients $\mu_1$, $\mu_2$ where $\mu_1 < \mu_2$
Output:
  Snell's Law crossing point $P_0$
Optimal Crossing Point Algorithm:
  Assume that $E_0$ extends infinitely in both directions and compute
  straight-line intersection $I_0$ of $E_0$ and line $P_1P_2$.
  IF $E_0$ is a phantom edge
     THEN $I_0$ is $P_0$, check for constraining vertices and return
  ELSE compute perpendicular intersection $I_2$ from $P_2$ on $E_0$
  IF $V_2$ is between $I_2$ and $V_1$
     THEN $V_2$ is $P_0$.
  ELSE IF $V_1$ is between $V_2$ and $I_0$
     THEN $V_1$ is $P_0$, return $V_1$ as $P_0$
  ELSE compute perpendicular intersection I1 from P1 to $E_0$
     CALL C routine snell($P_1$,$P_2$,$I_1$,$I_0$,$I_2$,$\mu_2$,$\mu_1$). Snell uses $I_1$,$I_2$ to
      determine if table lookup can compute $P_0$. If not, then $I_0$,$I_2$ are
      used as starting points for Golden Ratio search to find $P_0$.
     Check for constrained Snell's-Law vertex crossing and return.
```

Figure 3.6 Optimal-Crossing-Point Table-Lookup Algorithm

To compare the table lookup procedure to pure Golden Ratio search, we generated random crossing episodes across a single boundary edge of 100 map units in length. For each episode one random point was selected from each square region on either side of the crossing edge. For the two regions, unequal pairs of weights, μ_1 and μ_2, were randomly assigned from the set $\{1,2,...,10\}$. Timing results of the tests indicate that the table lookup is one order of magnitude (10 times) faster than pure Golden Ratio search. These results include time needed by the table lookup procedure to perform an occasional Golden Ratio search when random out-of-bounds conditions occur. To compute the accuracy of the lookup routine, we assume that Golden Ratio search finds the exact optimal point and, consequently, the true minimum weighted distance. Let C_{GR} be the total weighted cost of a crossing episode as computed by Golden Ratio search. Let the corresponding table-lookup value be C_{TL}. We compute relative error in weighted distance as

$$|C_{TL}-C_{GR}|/C_{GR} \qquad \text{(Eq 3.8)}$$

For 100,000 random samples the mean relative error was 3.83×10^{-7} with standard deviation 1.73×10^{-6}. The maximum relative error is on the order of 10^{-4}, and occurs in less than 0.01% of the sample. We should expect some error, since our two-dimensional linear interpolation is an approximation of a curved surface by a flat plane. Figure 3.7 is a plot of the discrete points in the table for $\rho = 1.1$. Although slight, the curvature of this surface is most noticeable at the lower values of r and y'. The squaring and square root operations of weighted distance soften the effect of interpolation error on x'. Nonetheless, we can rely on about 5 decimal digits of accuracy for the optimal weighted distance of a crossing episode. Note that there are only 6 decimal digits of precision provided by our implementation language, Quintus Prolog [Quin90]. Depending upon the length of a WS (i.e. how many crossing episodes), errors can accumulate. However, because mean relative error is essentially zero, we may assume that errors accumulated over long WS's will usually cancel one another. We conclude that the Snell's-Law-table lookup method provides SPR with a reasonably accurate and efficient means for computing optimal crossing points on boundary edges.

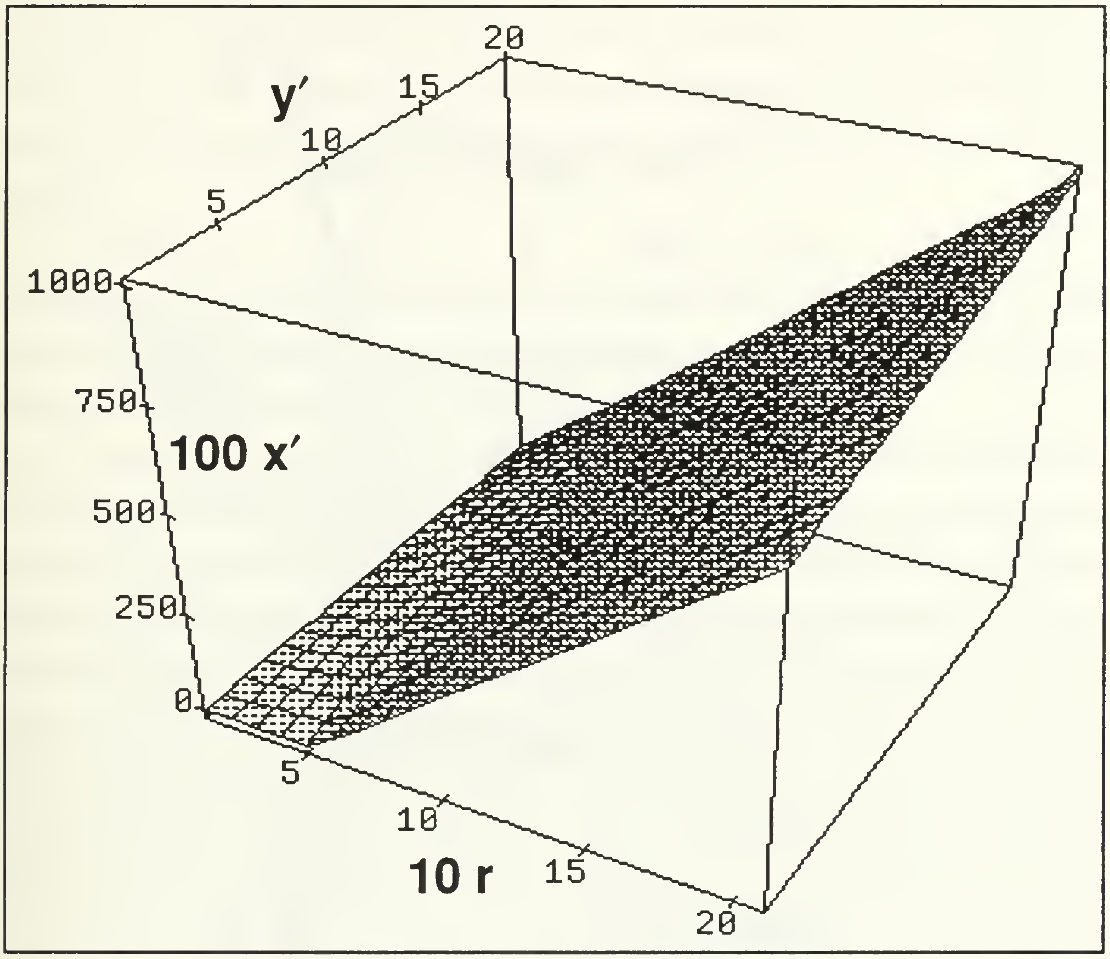


Figure 3.7 Surface Plot of Snell's-Law Table ($x' = f(y', r)$ for ρ=1.1)

2. Uniform-Discrete-Point (UDP) Approximation

While local path optimization with SPR is fast, the procedure can sometimes run into worst case situations. In practice we have found that even with a relatively slack relaxation tolerance, SPR may sometimes converge at an unacceptably slow rate. This happens if the midpoint path (starting condition) is far from optimal and has few sharp bends. In such situations the improvement of each crossing-point adjustment may be slight. Therefore, a large number of iterations may be necessary to optimize the path. Figure 3.8 illustrates a typical worst case for SPR. In this example the rate of SPR convergence decreases rapidly as the solution approaches optimum.

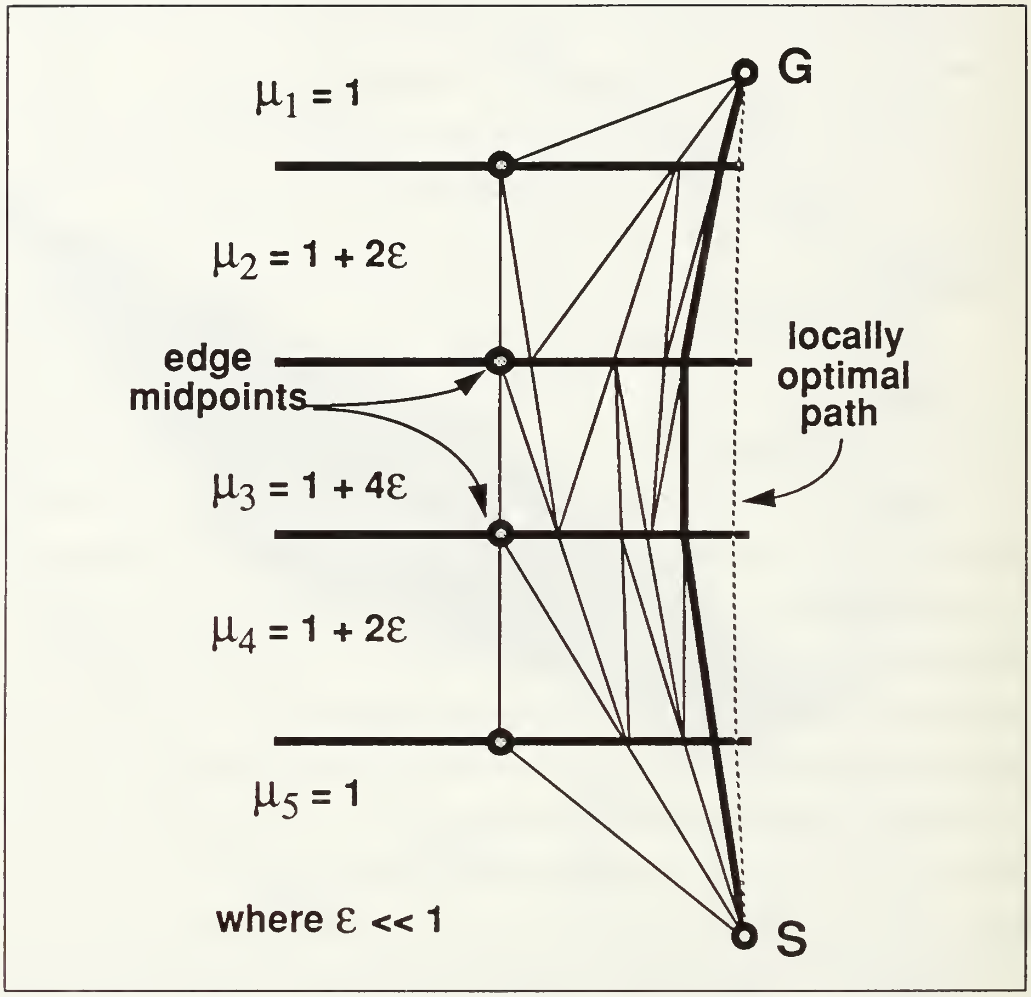


Figure 3.8 Worst Case Convergence for Single-Path Relaxation (SPR) (dark path marks progress from midpoints after four passes)

Usually at high annealing temperatures a large number of the WS's sampled will have locally optimal path-costs which are very large. Therefore, accurate WS cost evaluation is only necessary when the locally optimal path might be globally optimal. Both theoretical and empirical evidence validate the use of approximations to the actual cost function [Adam85, Tove88]. To capitalize on this idea, we implement a very simple approximation to pre-evaluate the cost function of a WS. We refer to the method as *uniform-discrete-point* (UDP) search. It is essentially Dijkstra's algorithm restricted to a WS. UDP is used to rapidly approximate the locally optimal path through a WS, and, if necessary, to provide a starting path for SPR.

Uniformly spaced points, called *edge-points*, are specified on each edge in the WS. From the start point, all paths constrained to pass through this set of points are searched. The *discrete interval*, δ, defines the upper bound for the separation distance of any pair of adjacent edge-points. Given a value for δ, all edge-points can be predetermined and stored for all boundary edges. For a given edge of length, l, the number of required edge-points is

$$m = \lceil (l/\delta) + 1 \rceil. \qquad \text{(Eq 3.9)}$$

The m points (which include terminating vertices) are then evenly spaced across the edge. For any edge with non-zero length, there must be a minimum of two edge-points – the terminating vertices. Note that the resulting structure can be viewed as a higher resolution edge dual-graph.

This idea is similar to one used by [Papa85] to efficiently approximate the optimal path through a field of polyhedral obstacles. To each obstacle edge, he assigns discrete points spaced exponentially out in either direction from the closest point on the edge (the perpendicular intersection from a previous fixed reference point). Exponential spacing is essentially a device which holds the subsequent path search to polynomial time. However, the polyhedrons in this three-dimensional obstacle-avoidance problem are all obstacles (i.e. travel is not permitted within them). Therefore, optimal paths tend to avoid their vertices, and settle into closest approach tangents across polyhedron edges. So, in a sense, exponential spacing also represents point samples from a *normal* probability distribution centered at locations most likely to be on the optimal path. In the WRP this notion is countered by the added dependence of optimal paths upon relative region weights as well as direction. The relative positions and weights of regions can often override the natural tendency for very straight optimal paths. Therefore, it appears that uniform point-spacing is more appropriate for the WRP.

To approximate the optimal cost of a window sequence (WS), the uniform-discrete-point (UDP) procedure works as follows. Beginning from the start point (or inversely the goal), Dijkstra's Algorithm proceeds through the WS toward the goal. Back pointers are recorded at each point on an edge to indicate the direction of the shortest path back to the start. When the goal point is reached, the shortest path is retrieved through back pointers. This search is fast because it is restricted to a WS. Consequently, the search always proceeds in order of WS edges from the start point. The time complexity of the algorithm is $O(nm^2)$, where n is the maximum number of edges in any WS and m is the maximum number of edge-points assigned to any edge in the map. Note that m is a function of δ and the longest edge in the map.

3. Combining Techniques

In our implementation, to find the locally optimal path in a WS, we employ uniform-discrete-point (UDP) approximation and single-path relaxation (SPR) in a two-stage combination. For a given WS, the UDP solution path is a more refined estimate of the local optimum than the midpoint path. Thus, UDP can hasten SPR convergence by finding a starting path which is usually closer to the true local optimum than the midpoint path. In stage one, UDP rapidly selects the locally shortest path constrained to discrete δ-spaced edge-points in the WS. In stage two, SPR begins from the path returned by UDP through these edge-points. Only a few passes by SPR will suffice for convergence to the true local optimum.

Both UDP search and SPR can be tuned to a desired accuracy by adjusting the discrete interval, δ, and the relaxation tolerance, ε (see Section III.D.1), respectively. We have found that setting δ equal to the average length over all edges tends to balance the assignment of edge-points uniformly across the map. We assume that the length of the shortest edge is closely related to the intended map resolution and required solution accuracy. Using mean edge-length to set δ strikes a compromise between accuracy and speed over all problem instances associated with a given map. The value of δ might also be computed dynamically based upon the lengths of boundary edges between and in the vicinity of start and goal locations. As we will discuss in Chapter IV, our particular choice of δ provides another opportunity to improve performance through heuristics.

4. Reentrant Window Sequences

A window sequence (WS) which contains at least one reentrant path is called a *reentrant window sequence*. Recall from Chapter II that a reentrant path exits a high-cost region by striking a boundary edge at the Snell's-Law critical angle, travels along the edge incurring the cost of the low-cost region, and reenters the high-cost region at the critical angle (see Figure 2.3). Reentrant WS's present no special problem for UDP approximation, since this method is not Snell's-Law based. However, we note that such WS's are distinctly different from those which cannot contain reentrant paths. A WS which contains reentrant paths should list each reflective edge twice in succession to indicate that there are two distinct turning points on it – the point of exit from and the point of reentry into the highest region it bounds. UDP approximation simply assumes that for each edge identifier in the WS there must be a corresponding optimal path turning point. Given a WS and its edge-points, the algorithm determines an optimal edge-point on each edge by running Dijkstra's algorithm. For each edge listed twice in succession, weighted distances between all pairings of its edge-points are computed using the optimal weight of this edge. The pair returned as optimal on this edge is the pair along the shortest weighted back-pointer path retrieved after the goal point is reached. If a reentrant path is, indeed, locally optimal within the WS, then the pair returned with the optimal list will usually be those edge-points which least violate Snell's Law. However, in some cases a reentrant path cannot be locally optimal for the WS (i.e. when it is always cheaper for a path to travel directly through the region instead of reflecting). This happens most often when a reentrant WS is forced onto a boundary edge from its low-cost side. Figure 3.9 illustrates such a situation. A reentrant path against edge E_r between approach points p_1 and p_2 is not an advantage over the direct path p_1 to p_2. In this case the pair of entry/exit points returned corresponds to a single point. The resulting path segments can always be shortcut at less cost. However, we expect this because a reentrant WS intentionally forces the cost evaluation mechanism to find the best path which hits the reentrant edge whether or not this path represents cost improvement over the original direct route.

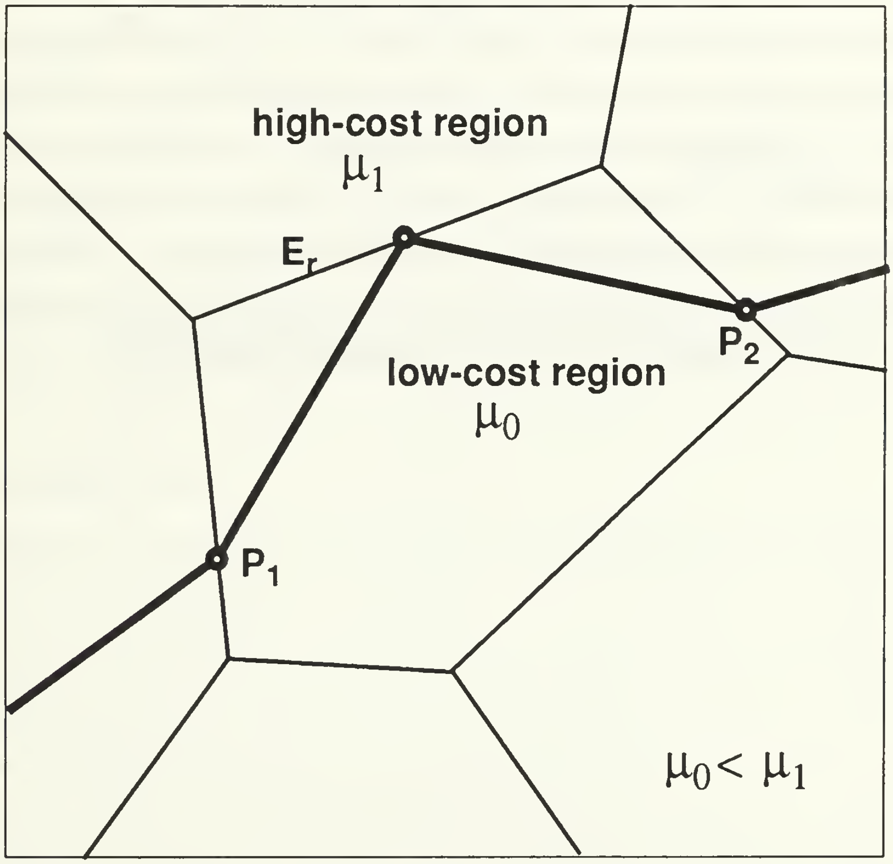


Figure 3.9 Uniform-Discrete-Point (UDP) Approximation Is Forced to Reenter a Low-Cost Region from the Low-Cost Side of a Boundary Edge

Unlike UDP, SPR must recognize each reentrant path location within the WS because it must adjust entry and exit points to the Snell's-Law critical angle, instead of executing a table lookup. This angle is measured from the approach points, i.e. the edge-crossing episodes just prior to and after the reflective edge. When a reentrant path is encountered during a single-path relaxation pass, critical-angle reflection points are determined using Snell's Law from the most current approach points. The stopping condition for SPR, i.e. one complete pass through the WS without any point adjustment greater than tolerance ε, never depends on reflection point adjustments, only on crossing point adjustments. This is because reflection points are predetermined by the adjustments of their corresponding approach points. Therefore, it is sufficient to monitor only the adjustments of crossing points (i.e. non-reflection points) in order to detect convergence to the locally optimal path.

Recall from Chapter I that *phantom edges* are edges which which bound regions with equal cost coefficients. They are used to ensure that all regions are convex. When a reentrant path is forced against either a phantom edge or a higher-cost edge, it is known *a priori* that the reentering path cannot be locally optimal. In such cases, SPR ignores the reentrant path route and optimizes the straight-line path between crossing episodes immediately before and after the points of reflection (i.e. approach points), effectively bypassing the reflective edge. This situation is illustrated in Figure 3.10. The WS still lists the reflective edge twice. The reason for this is that there may still exist an edge within the adjacent region bounded by the reflective edge on which reflection may still be advantageous. This will be discussed more in the next section.

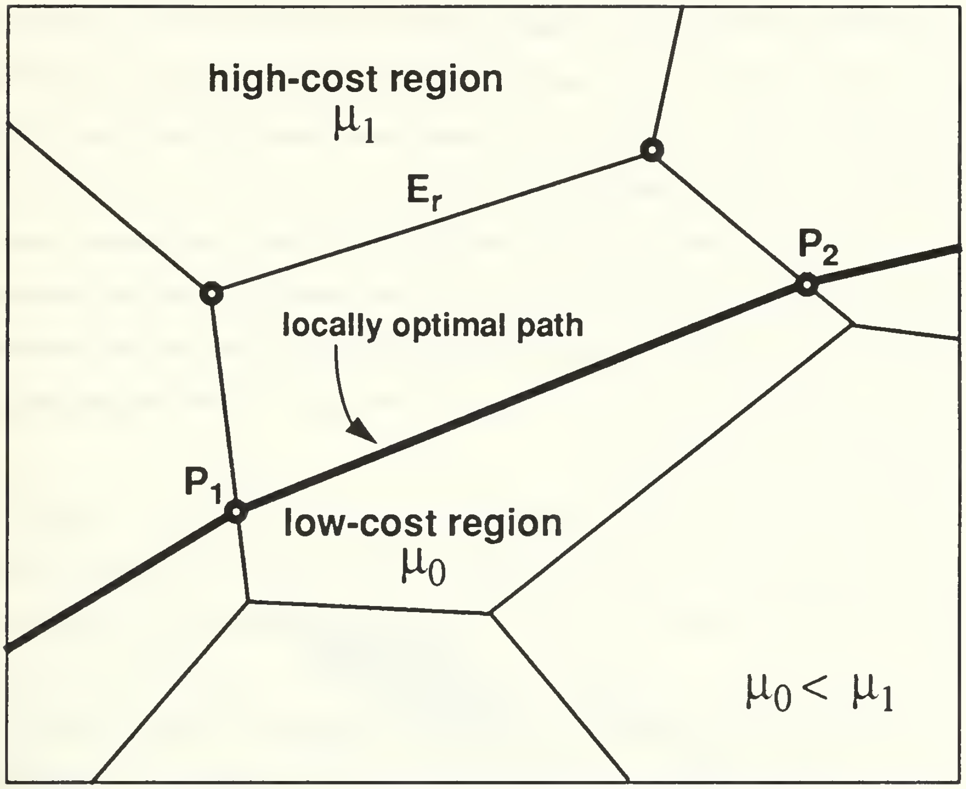


Figure 3.10 Single-Path Relaxation (SPR) Ignores Non-Advantageous Reentrant Path When Forced to Reenter a Low-Cost Region

E. MOVE OPERATIONS

Simulated annealing samples many different solutions by randomly applying move operators to explore the solution space. In path annealing, the *move generator* works in the space of WS's. Recall that we represent each WS uniquely as an ordered list of edge identifiers. A new WS can be created from another by little more than random number generation, data retrieval, and list processing. The move generator perturbs the current WS by rerouting it through neighboring edges and regions. Repetition of this process enables random search. Annealing does not normally search the entire solution space. Nevertheless, all feasible solutions should have some chance of being sampled. Our prototype employs two very simple move operators. These operators require data structure support in addition to the edge dual-graph (previously discussed). In this section, we introduce these move operators and discuss their sampling capabilities.

1. Vertex Rotation

Consider an arbitrary window sequence, WS_1, between designated start and goal. One of the simplest ways to obtain a new window sequence, WS_2, is to shift WS_1 across a vertex which defines one of its edges. We call this operation *vertex rotation*. The operation is relatively efficient because vertex edge-lists (see Section III.B.2) are readily available in sorted order.

Given WS_1, path annealing generates a neighboring WS_2 as follows. Generate a random integer to select one vertex of one edge in WS_1. This vertex, called the *rotation vertex*, defines the site of intended displacement. If the rotation vertex is a border vertex then backtrack and randomly select another vertex. Let E be the vertex edge-list for the rotation vertex. Let E' be those edges in E which are also members of WS_1. Then, WS_2 is defined by the set operations

$$WS_2 = (WS_1 - E') \cup (E - E') = (WS_1 - E') \cup \overline{E}' \qquad \text{(Eq 3.10)}$$

Eq 3.10 simply implies that we replace the edges in E currently in WS_1 with the edges in E which are not in WS_1. The result is WS_2. Of course, order in each set is significant, and list reversals must be performed as necessary.

The vertex rotation operator has an important side-effect. Given the rotation vertex V_R, the current WS is scanned for the first and last occurrences of edges in the vertex edge-list of V_R. All edges between these two edges inclusive will be removed. Thus, if random perturbations have formed a WS with an open loop which crosses two edges incident to V_R, then a rotation across V_R will cause the WS to cross itself at the open base of this loop, and the set operations of Eq 3.10 will remove it. By the Optimality Principle, a globally optimal path cannot cross itself. It follows that any WS which crosses itself cannot contain the optimal path. Therefore, a significant advantage of vertex rotation is its automatic removal of WS's which contain cyclic paths.

2. Obstacle Jumping

The presence of obstacles necessitates special consideration. Recall that the edge dual-graph which represents the feasible solution space does not contain arcs through obstacle regions. However, we can view obstacles as enlarged vertices with non-zero area. Thus, to enable obstacle rotation or *obstacle jumping*, we employ a special data structure similar to a vertex edge-list. An obstacle edge-list (see Section III.B.2) records the clockwise ordered list of incident edges around an obstacle. If an obstacle vertex is selected as the displacement site, the WS shifts completely across the obstacle by using the operation of Eq 3.6, where E is now the appropriate obstacle edge-list.

It is also possible to jump some obstacles by a sequence of normal interior vertex rotations. A jump of this type occurs as the result of pinching a looping path which has circled the obstacle from both the clockwise and counterclockwise directions. Figure 3.11 illustrates a path within a WS with the potential for jumping obstacle region *A*. Closing the base of the loop by rotating across vertex V_R will remove the detour around obstacle region *A*, replacing it with the dotted path segment and its associated WS. Not all obstacles can be jumped in this manner. For such a jump to occur enough interior vertices and edges must surround the obstacle so that a WS can be perturbed completely around it in either direction. Obviously, obstacles which connect directly to the map border cannot be jumped this way (e.g. the obstacle region *B* in Figure 3.11). For such cases, explicit obstacle rotation is necessary.

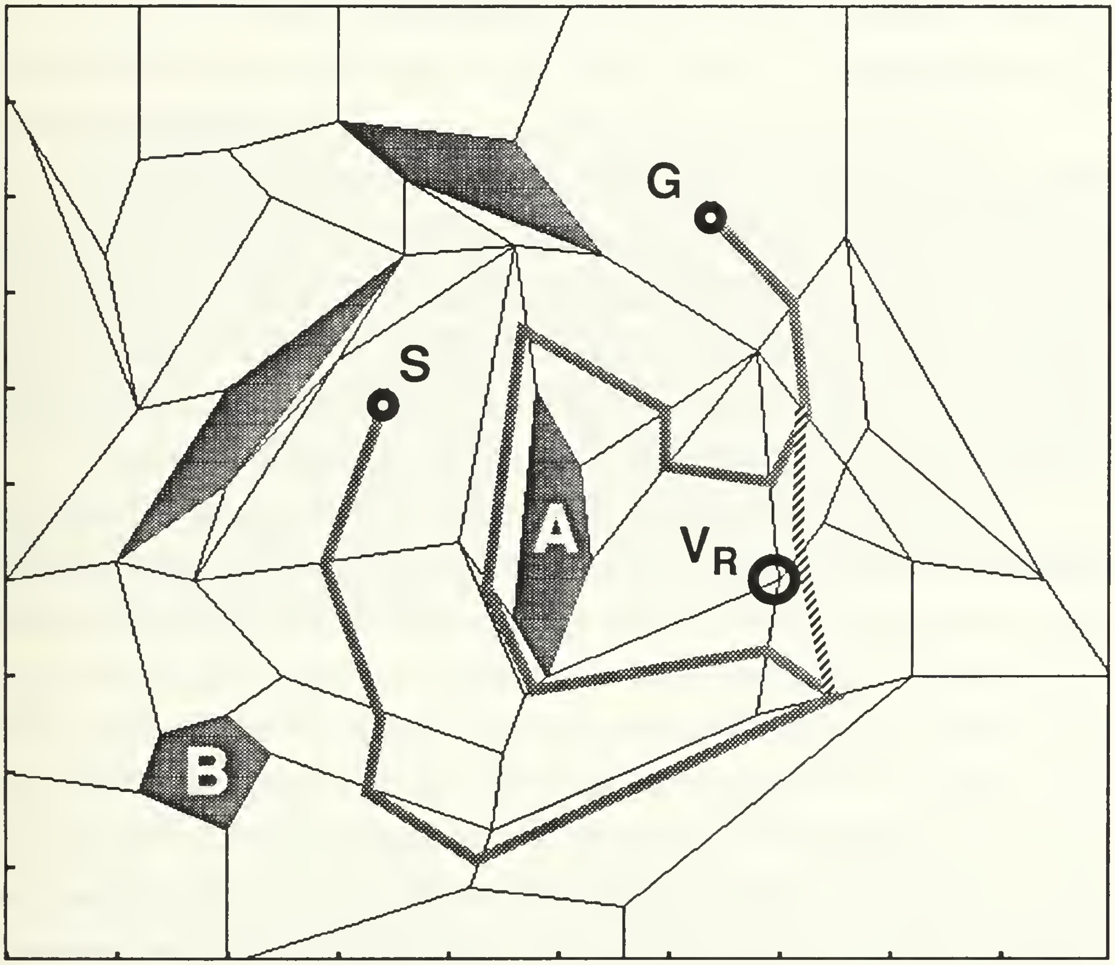


Figure 3.11 A Path with the Potential to Jump Obstacle A on a Single Vertex Rotation at V_R

While obstacles and interior vertices are logically equivalent with respect to the rotation operator, we ensure that they are distinguishable. In particular problem instances it may be more efficient to reduce or temporarily prevent the capability to jump obstacles. This control is necessary because the move generator can have a natural bias toward selection and jumping of large obstacles with many vertices. This is especially true when the current WS includes several edges incident to the same obstacle, as is often the case immediately after jumping it. In such situations, given an equal probability of selecting any edge in the WS from which to obtain the rotation vertex, that vertex with the most incident edges in the WS will have the greatest likelihood of being selected. This bias can waste valuable sampling time with useless repetition as the move generator attempts to jump back and forth across a single large obstacle. As well, there are good reasons for ignoring this bias and allowing the annealing process to make its own decisions. We defer a more complete discussion of these reasons and methods of obstacle control to Chapter IV.

3. Reachability

Recall that the move operators should guarantee *reachability*; that is, that all feasible solutions which could be optimal are accessible to the annealing process with a non-zero probability. This is to ensure that optimal solution cannot be overlooked intentionally. Do not confuse this term with *completeness*, which describes a seemingly similar guarantee offered by systematic search algorithms. The difference is in the probability of locating a globally optimal solution. An algorithm which offers completeness will find the optimal solution with probability one; the probability of not finding it is zero. However, reachability means that both the probability of finding the optimal solution as well as not finding it are non-zero.

A move generator based only upon the rotation operators (vertex and obstacle) can only guarantee reachability for a subset of those WS's with the potential to contain a globally optimal path. We refer to this subset as the class of *proper window sequences*. The class of proper window sequences is all WS's which do not enter the same cost region more than once. This definition implies that no proper WS can cross any boundary edge more than once, otherwise it necessarily reenters some region. In contrast, an *improper window sequence* may enter the same cost region multiple times. We have proven in Theorem 3.3 (see Appendix A) that all proper WS's are reachable from any initial proper WS in a convex weighted region map. Thus, if only proper WS's could contain globally optimal paths then our move generator would guarantee reachability. Unfortunately, in some problem instances an improper WS may contain the optimal path. The next section discusses these improper WS's and the additional move operator required to reach them.

4. Reentrant Installation

Although an optimal path cannot cross, it can cross the same edge more than once, and therefore, can also reenter a region. Recall from Chapter II that an optimal reentrant path will obey Snell's Law by striking a boundary edge at its critical angle and traveling along the edge just inside the lower cost region. Since we disallow linear regions (e.g. roads), such a path may exit the edge in only one of two ways. It may travel to a terminating vertex (Figure 3.12), where Snell's Law does not apply. Or, it may reenter the region at the Snell's-Law critical angle (Figure 3.13). In either case, the path can still be globally optimal. The former case involving the terminating vertex is contained within a proper WS. Since this path does not reenter the region, then it must cross a different edge when it reaches the vertex. Such paths are contained within proper WS's which can be reached by the rotation operators (Theorem 3.3 in Appendix A). However, the reentrant path in the latter case must reside within an improper WS. The reason is that the path reenters a region, and thus, by definition (see Section 3 above) can only be within an improper WS. The rotation operator alone cannot reach improper WS's.

A simple argument shows why this is true. Suppose that some arbitrary sequence of rotations applied to a proper WS results in an improper WS. Without loss of generality, we may assume that at some point in the sequence a rotation operation transformed proper window sequence WS_1 into improper window sequence WS_2. This would require that the rotation insert at least one edge identifier into WS_1 more than once to create WS_2. But this contradicts the definition of a rotation since the only edges inserted are those which are not already in the window sequence. Therefore, our assumption that a proper WS was transformed into an improper WS is false. It follows that a rotation can never create an improper WS from a proper WS. As a consequence, the rotation operator alone can never create a WS with a reentrant path. We need an additional operator to reach improper reentrant WS's.

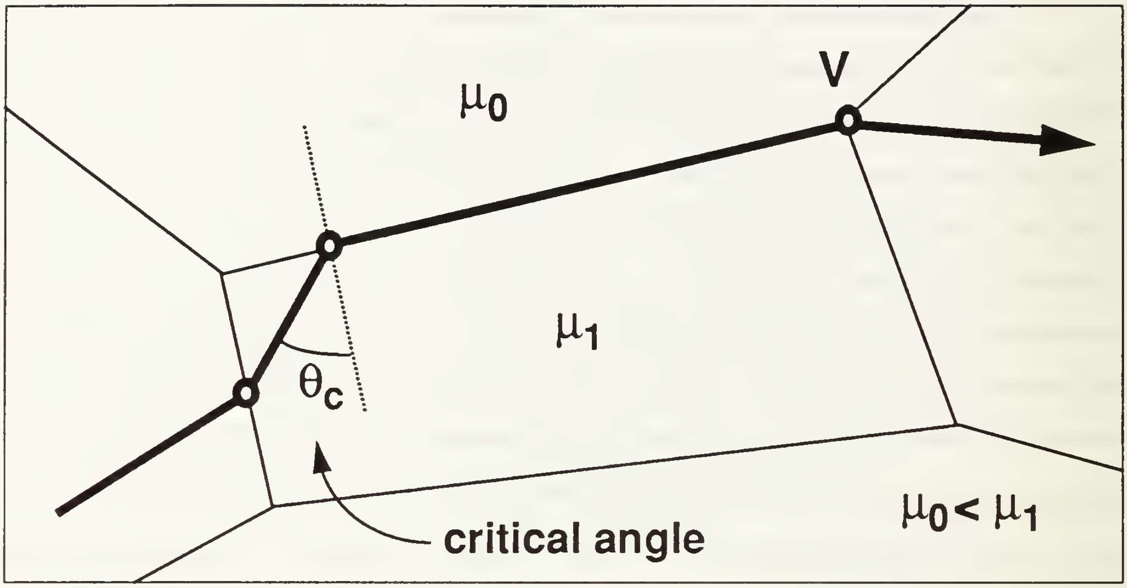


Figure 3.12 Critical-Angle Reflection Path (non-reentrant)

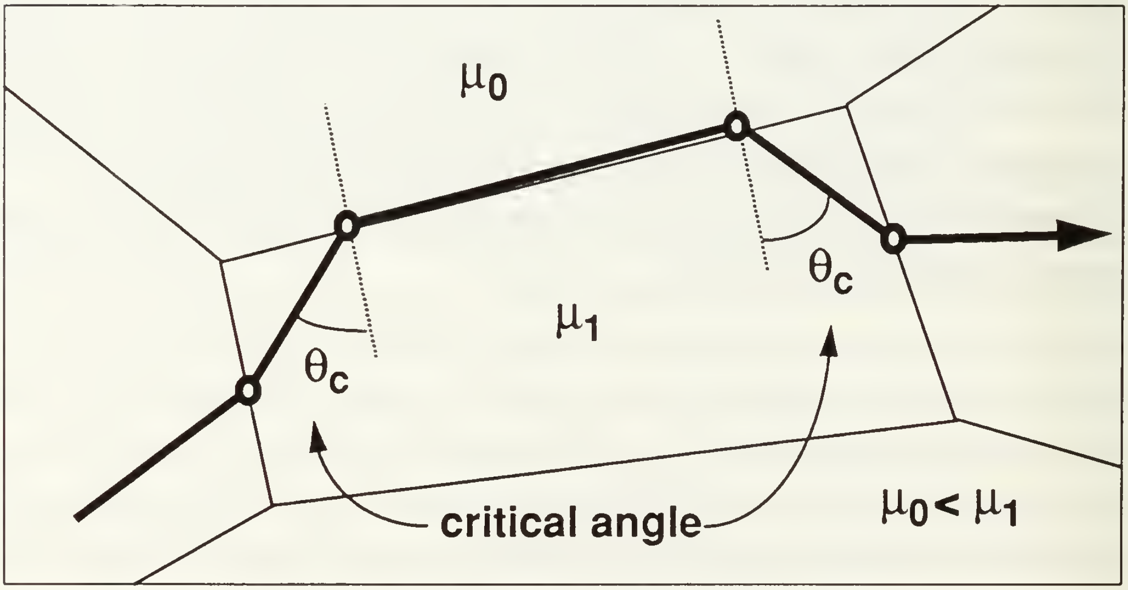


Figure 3.13 Critical-Angle Reentrant Path

The *reentrant installation* operator perturbs a window sequence (WS) by forcing it to reflect on one edge. This so-called *reflective edge* is selected randomly. The selection process is similar to that for a rotation vertex. The reflective edge is randomly chosen from the set of all crossable edges that bound regions through which the current WS passes. Edges which have crossing episodes within the current WS cannot be reflective edges. Transformation to the new WS is a simple double insertion of the reflective edge identifier. In Figure 3.14, window sequence {s,1,2,3,g} becomes {s,1,4,4,2,3,g} after reentrant installation on reflective edge 4. Note that the new WS is improper because by crossing edge 4 twice to reflect, the WS enters the same cost region twice. However, the evaluation of WS {s,1,4,4,2,3,g} will discover if a reentrant path exists which improves cost by using two points on edge 4.

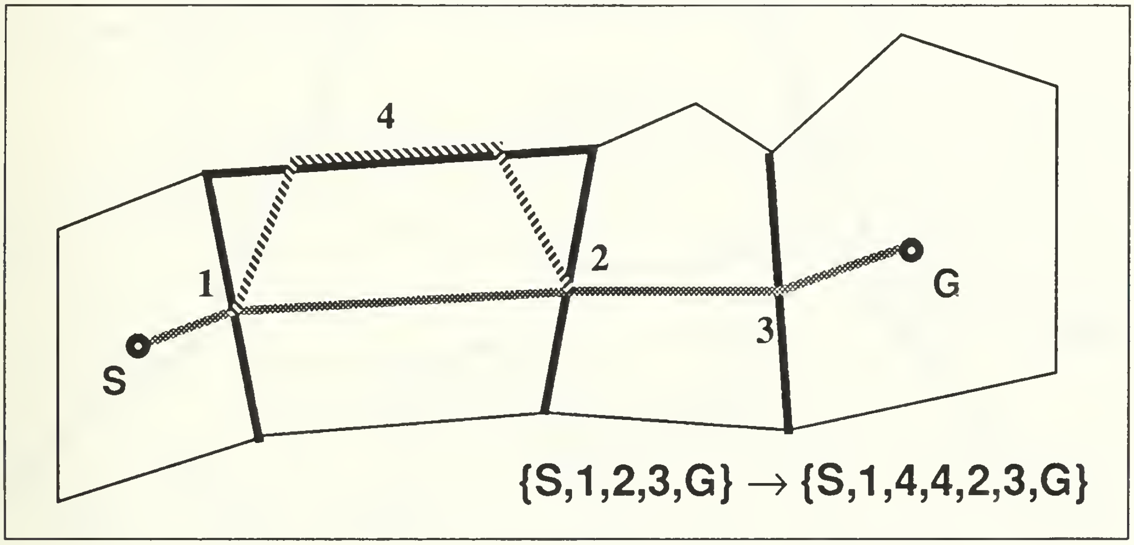


Figure 3.14 Installation of a Reentrant Window Sequence

It is not possible for an optimal path to reenter the same region on both of two adjacent boundary edges. Instead, the path must detour around the common vertex or cut straight across the region between the approach points. This fact is based upon a proof given in [Mitc90] which concludes that between any critical point of entry (exit) and the next critical point of exit (entry) the path must pass through at least one vertex. Because of this, our path annealing prototype will not create WS's with reentrant paths on two consecutive boundary edges.

It is possible for optimal paths to cross a sequence of several boundary edges, reflect and reenter a region at the last edge, and double back through the same sequence of edges in reverse order. Figure 3.15 illustrates the case of an optimal path which must cross several edges to reflect on the last and return. To reach the improper WS containing this path, four reentrant installations must be *stacked* by effectively pushing each through the one preceding it. To reach such special cases requires that the move generator install reentrant paths against an edge on which a reentrant already resides. Therefore, we allow reentrant installations to push (i.e. stack) through one another, ensuring that such special situations are reachable.

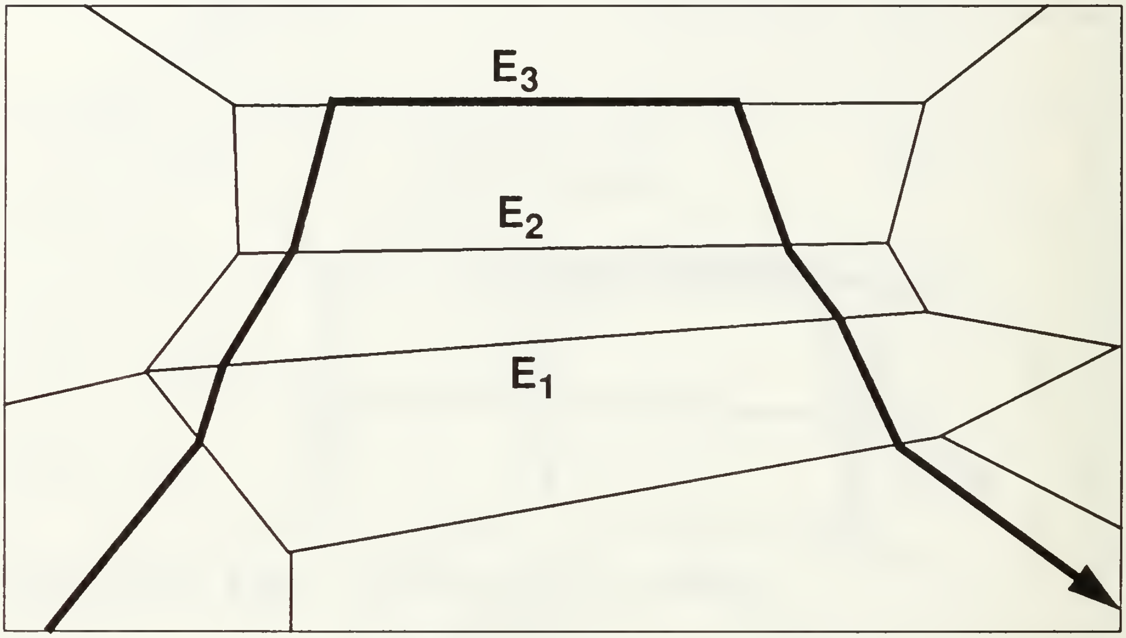


Figure 3.15 Reentrant Path Crosses Edges Multiple Times

Recall that reentrant-path optimal solutions tend to be rare because they require particular cost-region placements and angular alignments [Rich87, Mitc90]. Regardless of this, we give the reentrant installation and rotation operators equal probability of selection, and let the annealing process accept or reject their resulting transitions stochastically. This means that reentrant edges will be frequently installed into the current WS. A newly installed reentrant edge which greatly improves local optimal path cost of the current WS is more likely to be kept within the current WS. This reduces the chance of eliminating good reentrant paths by the vertex rotation.

5. Improving Reachability

The rotation and reentrant operators are capable of reaching all proper WS's and improper WS's which contain at most one reentrant path per edge. However, acyclic improper window sequences (WS) exist that these operators cannot reach. Unfortunately, some of these can contain an optimal path. We can trace the problem to an interaction between the rotation and reentrant operators. Consider the special case of an optimal path in Figure 3.16. Note that the very long edge bounding the top of the narrow cost-region is crossed several times by the optimal path. The vertex rotation operator cannot create such a WS because it automatically removes edges which are already in the WS. The reentrant operator only stacks reentrants on the same edge it cannot install them side-by-side. No combination of these operators can generate this WS from an arbitrary proper WS. Therefore, the rotation and reentrant operators cannot ensure complete reachability for all possible globally optimal paths.

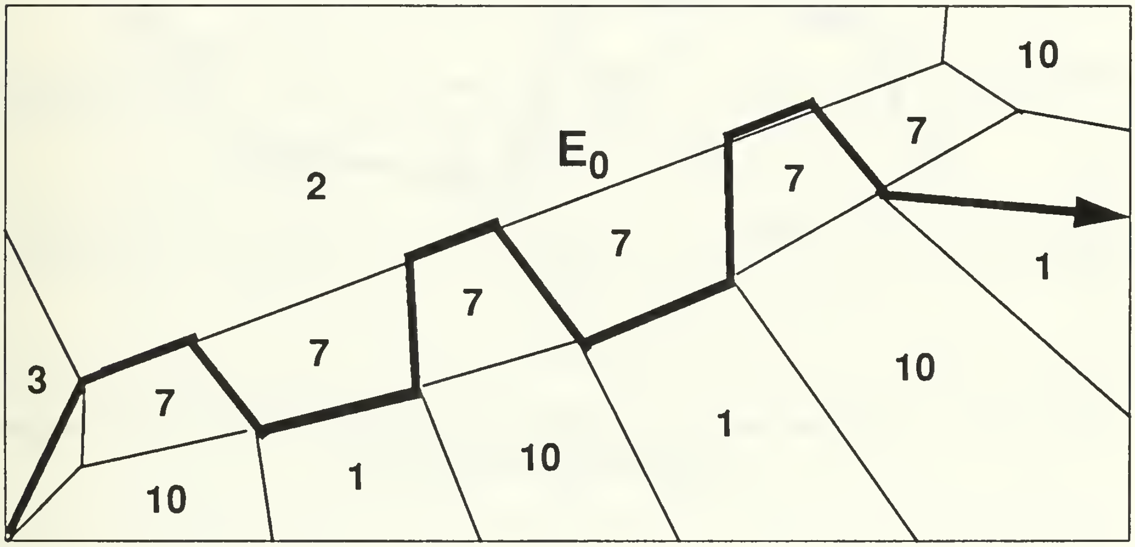


Figure 3.16 Optimal Path Crosses Edge E_0 Multiple Times (integers are cost coefficients)

In the case of Figure 3.16, it is possible to modify the reentrant operator so that reentrants can be installed side-by-side on particularly long edges. This would require that the algorithm recognize boundary edges whose length and geometry could allow multiple reentrant optimal paths. Then, when such an edge were selected for reentrant installation, if a reentrant path already existed on this edge, then an additional decision (possibly random) would have to made to stack the reentrant or install it side-by-side.

One way to avoid operator modification is to fix the source of the problem – relative edge-length. When a long boundary edge is situated among relatively short edges there is an increased likelihood that an optimal path may cross this edge multiple times. It is tempting to believe that complete triangulation of all regions will preclude any optimal path from crossing a single edge more than twice. Unfortunately, this is not true. Consider Figure 3.17 in which two near-degenerate triangular regions are adjacent with cost coefficients as shown. The path from *S* to *G* crosses the long edge (center) three times. Note that by squeezing these triangles closer to degeneracy, we can eventually reach a geometry which ensures that the path shown is optimal for *S* to *G*.

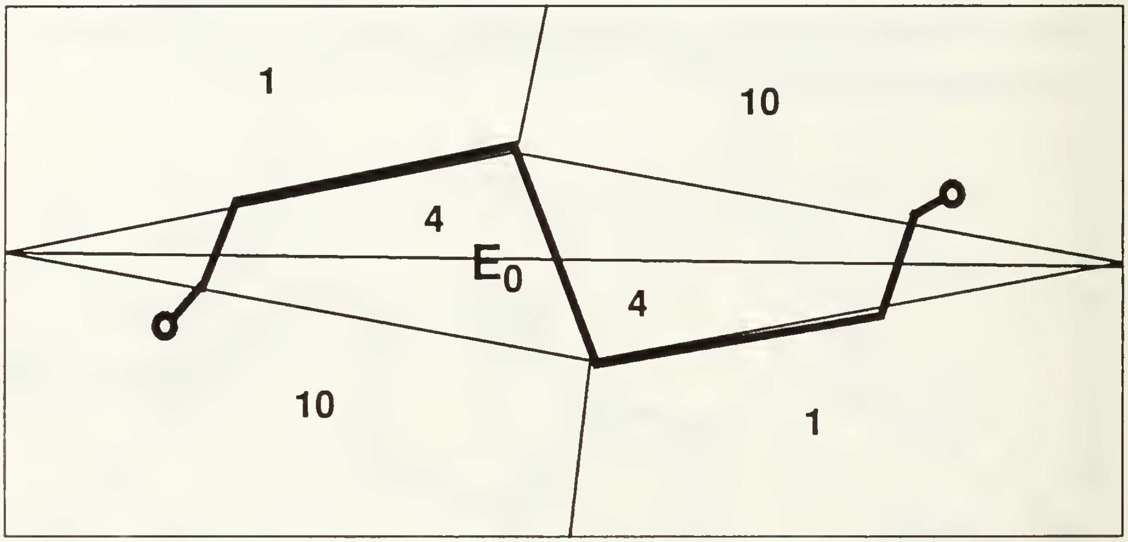


Figure 3.17 Optimal Path Crosses Edge E_0 Multiple Times (triangular regions) (integers are cost coefficients)

From the above simple example we see that the problem concerns resolution disparity in the map. The dimensions that are generally parallel and transverse to the long edge have a relatively large resolution differential. The disproportionately long edge allows ample opportunity for short transverse crossings to greatly improve path efficiency in parallel directions. In Figure 3.17, two path segments that are generally parallel to the long edge become more efficient by connecting them with a very short, and therefore, relatively low-cost transverse segment. We can easily prevent this problem with the addition of one vertex at the center of the long edge and two phantom edges connecting it to existing vertices in either region. This does not change the representation of the map. It simply balances the resolution of the map in a localized area.

Some simple preliminary map analysis can reduce the risk of having optimal paths with multiple edge crossings by identifying and partitioning particularly long edges. This can be done by examining the length of each edge, *e*, relative to its immediate environment. Let *B* represent the set of edges bounding the two regions separated by *e* but not including *e*. Let *V* represent the set of vertices on the two regions. If the length of *e* is much greater than the average length taken over all edges in set *B*, and if the maximum distance over all perpendiculars from points in *V* to *e* is much less than the length of *e*, then *e* should be partitioned with one or more additional vertices connected to existing vertices by phantom edges. Figure 3.18 illustrates how the long edge in Figure 3.17 should be partitioned. A comparison of cost-coefficients should also be made to ensure that the weights of the two regions separated by the edge are large relative to surrounding coefficients. If such is not the case then the possibility of a multiple crossing on the edge is minimal even if it is long.

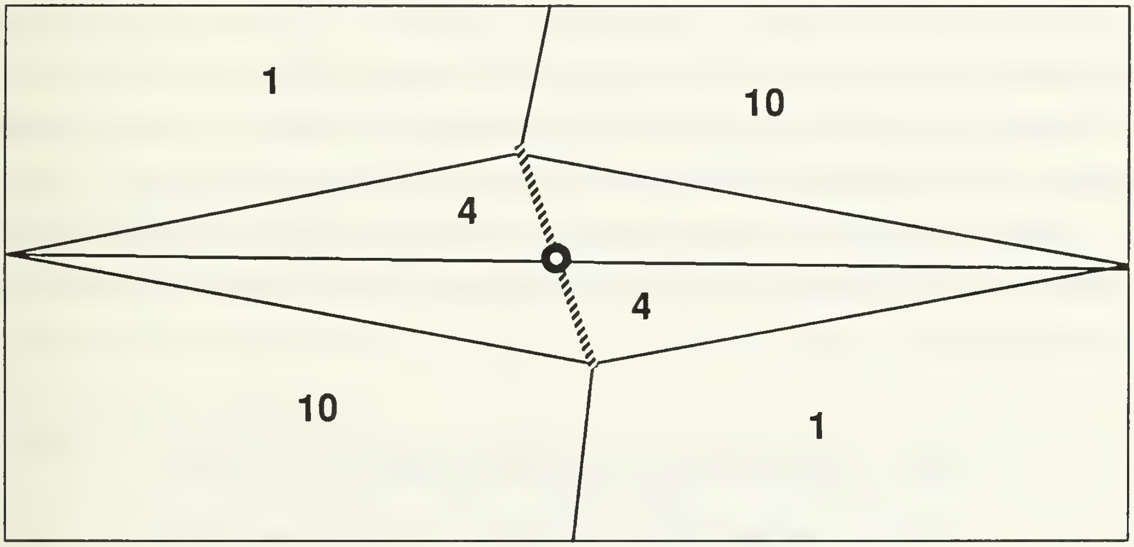


Figure 3.18 Introduction of Additional Edges (dotted) and Vertex (small circle) to Prevent Optimal-Path Multiple Crossings on a Long Edge

We emphasize that the above procedure is heuristic at best. However, a completely thorough analysis is likely to be tantamount to solving many weighted-region problems. Furthermore, the rarity of reentrant-path optimal solutions and solutions which are not contained within proper window sequences (WS) indicates that more detailed analysis is probably not worthwhile. We desire that map resolution be reasonably balanced. In the limiting case, as more vertices and phantom edges are added, the map will approach a uniform grid representation, and the advantages of path annealing may diminish. Therefore, a long edge should be partitioned only in the case that the conditions indicated earlier are extreme.

6. Summary of Move Generator

We summarize the key features of the move generator as follows. Its mechanics are simple and involve only two operators – vertex/obstacle rotations and reentrant installations. It guarantees reachability for all proper window sequences (WS). The perturbations are small and induce a neighborhood structure on the search space. The cumulative effects of the operators are large, particularly at high control temperatures. In the next section we discuss the establishment of the annealing schedule and the control of the temperature.

F. ANNEALING SCHEDULE AND CONTROL

Although many applications of simulated annealing require large amounts of computing time to arrive at reasonably optimal solutions, we have found that path annealing performs reasonably well without the need for such. Our annealing schedule is admittedly crude and uses only the simpler ideas proposed by other researchers. Path annealing efficiency seems to rely more on good WRP bounds and heuristics (to be discussed in Chapter IV) than on a well-tuned annealing schedule. Therefore, we employ the annealing algorithm for its probabilistic control of local search through a subset of the solution space.

While annealing schedules may differ somewhat with particular applications, we use the five control parameters described in Figure 3.19. The following sections discuss the establishment of the particular value for each parameter.

T_0 = starting temperature in weighted distance map units

T_f = freezing temperature in weighted distance map units

R = temperature reduction (cooling) factor; typically $0.70 \leq R \leq 0.99$ where $T_{l+1} = R * T_l$

L = annealing chain length; maximum no. of attempted transitions per temperature T_l

L_s = maximum no. of accepted transitions per temperature T_l; also known as a *cutoff* [John89]

Figure 3.19 Standard Annealing Schedule Parameters

1. Starting Temperature

Recall that annealing temperature, T, controls the shape of the exponential probability distribution used to accept or reject new solution samples. While all cost-decreasing changes are accepted, changes which increase the cost are accepted with probability

$$P(\text{acceptance}) = \text{Exp}(-\Delta C/T) \qquad \text{(Eq 3.11)}$$

where ΔC is the difference between the costs of the new and current solutions (also see Figure 2.6). The *acceptance ratio* over a sample of transitions is estimated by [Aart89] as

$$\chi = (\,(m_1 + (m_2\, \text{Exp}(-\overline{\Delta C}^+/T)\,)\,/\,(m_1+m_2) \qquad \text{(Eq 3.12)}$$

where m_1 is the number of cost-decreasing transitions, m_2 is the number of cost-increasing transitions, and $\overline{\Delta C}^+$ is the mean over the m_2 cost-increasing transitions. To ensure that annealing can sample across the solution space without trapping between high-cost peaks and freezing too early, it is desirable to set T to produce an acceptance ratio close to one. Ratios of .90 to .95 are generally considered close enough [Aart89].

To determine a good starting temperature, T_0 (initial value of T), we have tried an empirical technique suggested by [Aart89]. Randomly generate sample transitions in the solution space without rejections. From the sample values of ΔC compute $\overline{\Delta C}^+$. Eq 3.12 then determines the value of T which produces the required χ. [Aart89] suggests an iterative approach reported to converge rapidly on a value of T.

The conclusions of [John90] suggest that simple application-specific formulas can usually be derived on the basis of desired acceptance rate and a few problem-specific parameters. Thus, another way to determine a starting temperature value is to consider the worst-case transition that can be made by the move generator in the map. For very complex problems, such an approach is not always be reasonably possible. Fortunately, our representation of the WRP by window sequences allows efficient analysis of individual transitions. We desire the value of T which makes the acceptance rate as close to one (1) as possible. In Eq 3.12, let $\chi = 0.95$, and replace $\overline{\Delta C}^+$ with an estimate of the greatest cost-increasing value, $\text{Max}(\Delta C^+)$.

To determine the value of T for an acceptance rate close to one, assume that the generator applies only a single operator to the current WS, then returns a new WS for cost evaluation. Since cost-increasing reentrant installations usually do not result in large cost changes and because they are not necessary to move freely around the solution space, we will consider only rotations. We wish to determine the largest possible cost-increasing transition. This transition will likely be the site of a long edge bounding two regions with a large cost-coefficient differential.

Intuitively, Max(ΔC^+) should depend on an edge with a large weight differential. However, dependence upon the length of that edge may not be as clear. The reason that length is a factor is that a rotation is followed by local optimization to find cost (UDP approximation). If this rotation has just forced the current WS across the large weight differential, then local optimization will also force the locally optimal path to use as little of the high cost region as possible. Following this, a second transition across the other vertex of the same edge gives optimization no choice but to cross through the high-cost region. While it is possible that local optimization can still dampen the cost difference of the second transition, we assume the worst case. Ignoring smaller surrounding effects, the largest change will generally occur along the entire length of the edge. The situation is depicted in Figure 3.20.

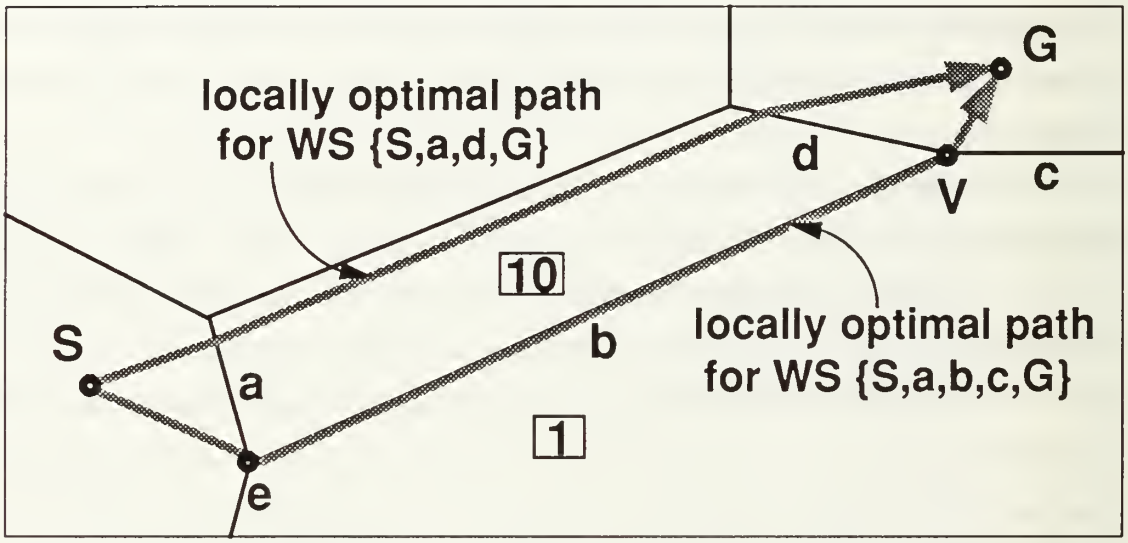


Figure 3.20 Maximum Cost Difference Max(ΔC^+) Occurs for WS Transition {S,a,b,c,G} $\longrightarrow$ {S,a,d,G} (cost coefficients are boxed integers)

Thus, to estimate Max(ΔC^+), we have only to analyze the length and weighted-region-cost differential for every edge in the map according to the following equation:

$$\mathrm{Max}(\Delta C^+) = \mathrm{Max}_{over\ all\ edges\ e}(L_e(m_2 - m_1)) \qquad \text{(Eq 3.13)}$$

If we assume that in a random sample of transitions $m_1 = m_2$, then Eq 3.12 becomes

$$\chi = 1/2\ (\ 1 + \mathrm{Exp}(-\mathrm{Max}(\Delta C^+))\ /\ T\ ) \qquad \text{(Eq 3.14)}$$

solving for T yields

$$T = -\ \mathrm{Max}(\Delta C^+)\ /\ \ln(2\chi - 1) \qquad \text{(Eq 3.15)}$$

In this approach there is no need to conduct empirical testing. One computation during map preprocessing can establish a common starting temperature for all problem instances. The major advantage here is the tendency for this method to be independent of problem instance, depending instead on the particular map. But it is conservative and can increase annealing running times unnecessarily. However, we emphasize that this approach is quite flexible. It is possible to conduct the analysis for Max(ΔC^+) on a subset of map edges confined to the vicinity of start and goal. This variation is more problem-instance dependent while still very efficient.

The presence of obstacles, again, requires special consideration. Small obstacles with few boundary edges will not significantly effect transitions. However, larger obstacles can sometimes hamper lateral movement because they may require acceptance of large cost changes to jump. If we wish to set starting temperature such that all obstacle rotations will be accepted in early annealing, it is necessary to analyze their worst-case cost differences also. Consider a single obstacle rotation. The worst transition tends to occur when the current WS is forced to detour completely around the obstacle from a single edge incident to it. To estimate the value of this worst-case cost difference, we compute cost of traveling on the boundary edges of the obstacle in a closed loop. From this we subtract the shortest weighted length edge of all its boundary edges to account for that portion of the cost incurred by the original path which passed near the obstacle.

2. Stopping Criteria

Freezing temperature, T_f, is the final value of T at which no significant cost-increasing transition has a reasonable probability of acceptance. The reduction of unnecessary search time is as dependent upon T_f as it is on T_0. Once T is low enough, the current WS will trap in a local minimum from which it will likely never escape and in which it can never improve the cost beyond the lowest it has already discovered. At this point we desire that the process terminate soon. However, if it is still the case that $T >> T_f$, then the algorithm will waste a lot of time before it halts.

How is it possible to determine the proper value of T_f when it apparently depends upon *a priori* knowledge of the optimal solution cost and its location? Fortunately, annealing runs exhibit a few similar characteristics. An *annealing curve* is a plot of the mean solution cost vs the logarithm of temperature. Figure 3.21 illustrates a curve which is typical of most annealing runs [Aart89, VanL87, Haje85, Whit84, Kirk83]. $\overline{C}_T$ is the average cost over all solutions sampled at discrete temperature T. Note that the x-axis represents $\log(T)$ in the direction of time. Therefore, $\log(T)$ decreases from left to right. Path-annealing curves tend to be wavier at higher temperatures (left half of the curve) because the starting solutions reside in cost function valleys. Path annealing climbs in and out of many such valleys until the value of *T* is low enough to stabilize (by trapping) it within a localized area. However, at lower temperature path-annealing curves do resemble the tail in Figure 3.21.

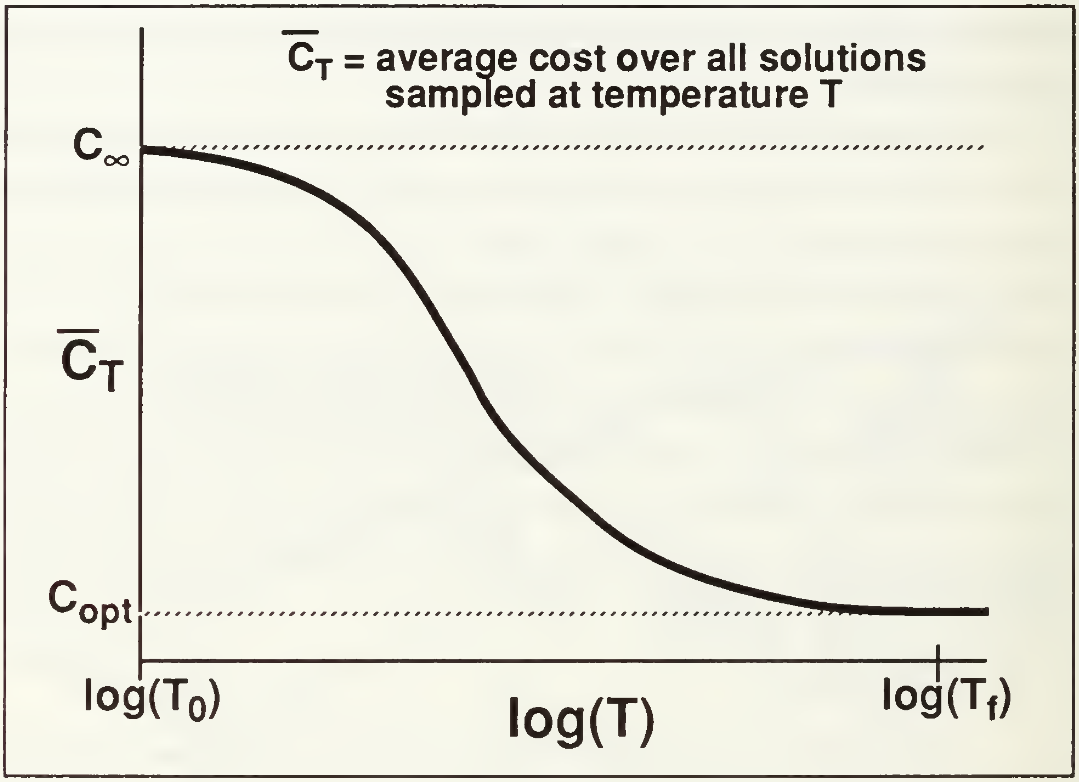


Figure 3.21 Annealing Curve: Average Cost at T vs. Log(T)

One way to set T_f dynamically is to extrapolate the end of the annealing curve by tracking the change in mean cost [Aart89]. This does not work well in path annealing since even with smoothing the mean cost can fluctuate significantly at very low temperatures. The suggestion of [Huan86] is simpler and a much better indicator of the freezing condition in path annealing. For a given *T*, if the cost difference between the largest and smallest accepted transitions is the same (or very nearly so) as the largest accepted ΔC, then apparently all solutions in the neighborhood are of very similar cost. At this point, simulated annealing should default to iterative improvement and halt. To guard against premature freezing, we suggest that the above condition exist for two or three consecutive temperatures.

The above procedure stops the annealing algorithm dynamically. Recall that our prototype actually uses a fixed value for T_f. However, we determined its value from many observations of path annealing using the criteria of [Huan86].

3. Temperature Cooling

There are several sophisticated cooling schemes which have been devised for efficient, adaptive reduction of temperature [Huan86, Lund86, Aart85]. We prefer the more simplistic approach taken in the original work of [Kirk83], also adopted by [John89]. Three fixed parameters control temperature cooling: R, L, and L_s.

The *reduction* or *cooling factor*, R, is a multiplier used to reduce T geometrically according to

$$T_{i+1} = R\,T_i \qquad \text{(typically } 0.70 \le R \le 0.99\text{)} \qquad \text{(Eq 3.16)}$$

This parameter controls ΔT, and thus, the speed at which the algorithm will approach T_f. In accordance with the underlying theory of simulated annealing [Kirk83] it is best to reduce T as gradually as possible with values of R closer to one. In any case, there is a tradeoff between time spent at each temperature value and the size of ΔT. Large ΔT approaches T_f relatively fast, but requires more time at each value of T. On the other hand, smaller ΔT approaches the stopping criteria much slower, but requires less time at each T.

The amount of search time spent at each temperature is directly controlled by L. The *chain length*, L, is the maximum number of transitions attempted by the move generator at each value of T. The best value of L is generally believed to approximate the size of the local search neighborhood [Aart89]. The local search neighborhood is the number of distinct states (i.e. window sequences) reachable in transition one of the move generator. For path annealing the local neighborhood size is a function of the number of edges in a current WS, since the move generator selects transitions on this basis. However, in contrast to the cardinality of a solution set in other problems (e.g. the Traveling Salesman Problem), the number of edges in a WS changes frequently. Also note that WS length varies greatly with the number of bends, waves, and even cycles. Therefore, we suggest the employment of some average or other stable value representative of WS length.

We could use the length of the starting WS found by A* search through midpoint paths. However, this WS is often biased and may not represent the size of most neighborhoods. Thus, for our testing we choose to use a fixed value for all problem instances of a map. A reasonable value for L can be derived from a map by using a sample of straight lines equal in length to the diameter of the map. L is computed by averaging the number of boundary edges crossed by each line over all lines in the sample. This procedure is essentially an estimate of map resolution. Once again, the advantages are simplicity and problem-instance independence with an option to allow adjustments to L for particular problem instances if more efficiency is desired.

Closely associated with L is the maximum number of accepted transitions per temperature, L_s. Recall that a transition attempted by the move generator is accepted if its corresponding solution cost is less than the current solution cost, or with probability P (see Eq 3.11) if more. L_s is referred to as a *cutoff* for L [John89]. The purpose of a cutoff is simply to compensate for overestimating the starting temperature, T_0. We desire a minimum value for T_0 at which the acceptance ratio $\chi = 0.95$ (or very close to one). It is better to begin annealing at an overestimated value of T_0, rather than risk the possibility of trapping too early in a bad local minimum. Assuming T_0 has been overestimated, until T approaches the proper value of T_0, each consecutive sequence of L_s acceptances will cause premature temperature reduction by Eq 3.16. L_s is essentially an annealing-specific heuristic devised by [Kirk83] to reduce search time spent at high temperatures [John89]. There appears to be little published guidance as to how to establish this parameter. Some forms of the annealing algorithm do not use it. In fact, [John89] claims that its effect is often insignificant, and its use is usually unnecessary if good starting temperatures can be found. We should expect some gain in efficiency by using a cutoff because our starting temperatures tend to be higher than necessary. Indeed, path annealing exhibits generally faster execution times with the same performance for $(0.67)L \leq L_s \leq (0.75)L$. Note that use of this control parameter has no effect when $L_s = L$.

G. SUMMARY

This chapter has presented the framework for a simulated-annealing-based algorithm designed to solve large instances of the WRP. We have discussed the theory and practical application of the algorithm in terms of its five essential components:

- state space representation
- initial solution
- cost function evaluation
- move generation
- annealing schedule

We employ simulated annealing as a simple tool for the control of local search. In doing so, many of the specific design concepts (e.g. the annealing schedule) are adapted from previous research. Therefore, we emphasize that the significant contribution of our work is not in annealing, but rather in the novelty and design of our annealing-based approach to solving the WRP. Chapters I and II have already described several of the advantages offered by this algorithm over previous ones.

In Chapter IV we will discuss heuristic enhancements and bounds which are available to path annealing (and some other algorithms) to increase performance and efficiency. We believe this collection of heuristics and bounding techniques is another significant contribution to automated path-planning research and the WRP.

IV. EFFICIENCY ENHANCEMENTS TO PATH ANNEALING

A. INTRODUCTION

In optimization problems it is sometimes possible to predetermine absolute limits on the location or cost of an optimal solution. We refer to such limits as *bounds* rather than *heuristics* since the latter are generally not absolute limits. Both upper and lower bounds can reduce the search effort by restricting the set of solutions or partial solutions which an algorithm must examine. The WRP provides several opportunities to devise bounds by reasoning about geometric conditions in the input maps. When absolute bounds are inadequate to reduce the amount of search required to obtain solutions in reasonable time, heuristic guidance can be employed. Although there is no assurance that heuristic knowledge will always achieve its intended objective, good heuristics can be helpful [Pear84]. The next two sections examine bounds and heuristics designed to increase the performance of path annealing. Some of these also apply to systematic search methods.

B. SEARCH SPACE BOUNDS AND CONSTRAINTS

1. Bounding Ellipse

In their systematic approach to the WRP, [Rich87] and [Rowe90b] implement a *bounding box* which is guaranteed to contain the globally optimal path. The box is actually a simplifying device that circumscribes a *bounding ellipse* whose foci are start and goal points. The bounding ellipse is based on the weighted cost of a feasible path from start, S, to goal, G, and the lowest cost coefficient on the map. For every point, P, on the ellipse, the straight-line path SPG weighted at the lowest cost is equal to the cost of the known feasible path. Figure 4.1 illustrates the ellipse inscribed within its bounding box.

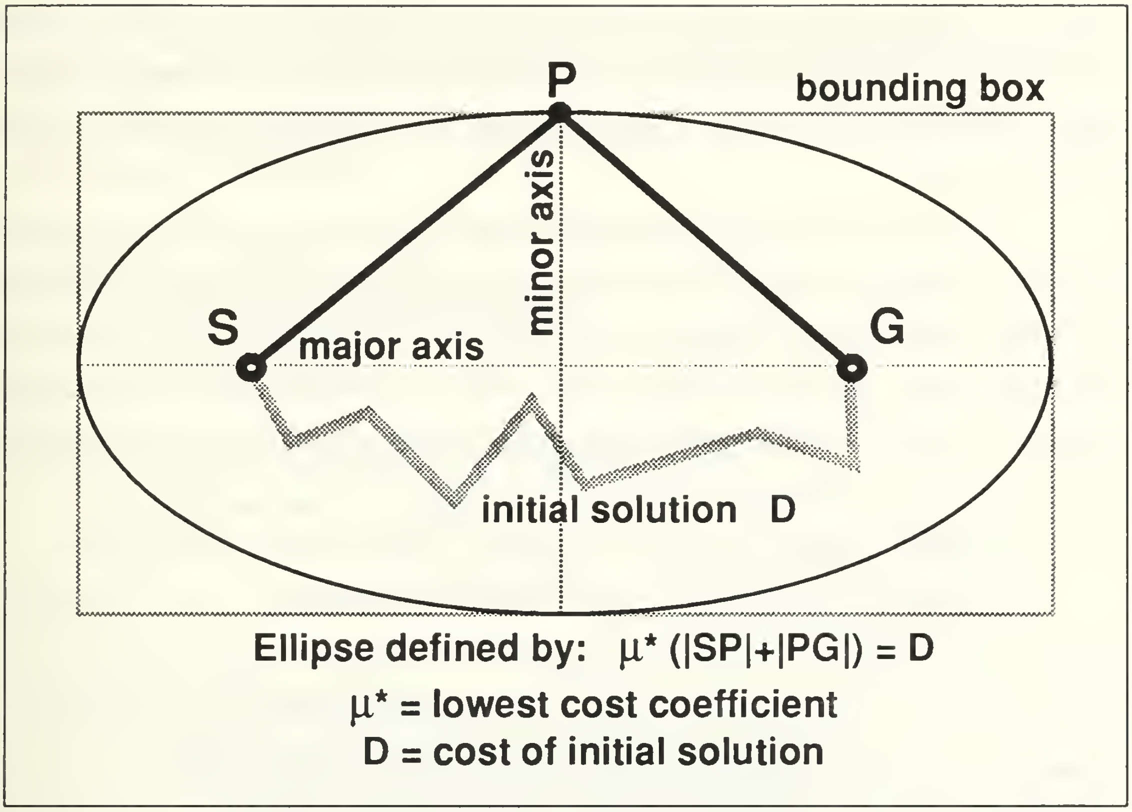

P
bounding box
minor axis
S
major axis
G
initial solution D
Ellipse defined by: μ* (|SP|+|PG|) = D
μ* = lowest cost coefficient
D = cost of initial solution

Figure 4.1 Bounding Ellipse Inscribed within Bounding Box

To compute the box, the implementation of [Rich87] determines the lengths of the major and minor axes of the ellipse. These lengths are functions of the known distance $|SG|$, the lowest cost coefficient and the initial solution cost. Referring to Figure 4.1, let μ^* be the lowest cost coefficient on the map, and let D be the weighted cost of the initial solution. Then the lengths of the major and minor axes are

$$L_{\text{minor}} = \frac{\sqrt{D^2 - |SG|^2}}{\mu^*} \qquad \text{(Eq 4.1)}$$

$$L_{\text{major}} = \frac{D}{\mu^*} \qquad \text{(Eq 4.2)}$$

These lengths, and the heading of the line from S to G, establish the edges of the bounding box. Any path (or partial path in systematic search) which intersects an edge of the bounding box (or ellipse) cannot be a globally optimal because the feasible path used to construct the box must be lower cost. Thus, the search space is absolutely bounded.

In path annealing we implement a true version of the bounding ellipse, which eliminates more area than the box. Path annealing can sometimes benefit greatly by eliminating even a few edges or edge parts from the search space; one less edge translates to combinatorially fewer WS's. Furthermore, cutting a long edge in half results in about one half of the edge-points needed by UDP approximation each time this edge is evaluated in a WS.

The cost of the initial midpoint solution returned by A* search is an upper bound on the cost of the globally optimal path. However, we can usually improve this bound by applying UDP followed by SPR on the WS through which this midpoint solution passes. Let the resulting path-cost be D. The foci of the ellipse are the start, S, and goal, G. By definition, P is a point on the ellipse, if and only if the Euclidean distance of a straight path from S to P to G weighted by μ^*, is equal to D. Thus, a point, Q, inside the ellipse obeys the following inequality:

$$\mu^* (|SQ| + |QG|) < D \qquad \text{(Eq 4.3)}$$

We compute the exact ellipse as follows. Consider the two vertices, V_1 and V_2, which define an edge on the map. Compute the cost of direct paths SV_1G and SV_2G weighted by the lowest cost, μ^*. Let these costs be C_1 and C_2 respectively. In comparing these costs to the ellipse defining cost D, there are three cases to consider, one of which has two subcases. Figure 4.2 outlines each case and the required action. Note that each boundary edge in the map is considered once. Usually only a small portion of the edges will require binary search.

For each boundary edge let C_1 and C_2 be the costs of the SV_1G and SV_2G paths where V_1 and V_2 are the vertices defining the edge.

To construct an exact bounding ellipse from initial solution cost D, there are 3 cases to consider:

```
1.  IF ($C_1 < D$) and ($C_2 < D$)
        THEN edge is inside ellipse
        Action: none

2.  IF ($C_1 < D$ and $C_2 > D$) or ($C_1 > D$ and $C_2 < D$)
        THEN edge intersects ellipse at one point
        Action: binary search between $v_1$ and $v_2$ to locate intersection
                update map database with new vertex for this edge

3.  IF ($C_1 > D$) and ($C_2 > D$)
        THEN edge intersects ellipse at two points
            OR edge is completely outside ellipse
        Action: locate point at which edge would be tangent to ellipse
                let that point be $T$, the virtual tangent point
                compute $S$-$T$-$G$ weighted by lowest cost coefficient $\mu^*$
                let this cost be $C_T$
                There are 2 cases:
                IF ($C_T < D$)
                    THEN edge intersects ellipse at 2 points
                    Action: binary search between $V_1$ and $T$
                            binary search between $T$ and $V_2$
                ELSE IF ($C_T > D$)
                    THEN edge outside ellipse
                    Action: remove edge from map database
```

Figure 4.2 Cases of Bounding Ellipse Construction Algorithm

The third case of Figure 4.2 indicates that it is sometimes necessary to compute a *virtual tangent point* and then binary search for an intersections on both sides of the edge. The virtual tangent point is that point on the edge of consideration at which a smaller ellipse with similar shape would touch the edge. By similar shape is meant that the ratio of the major to minor axis lengths remains constant. Simple geometry and the properties of an ellipse can show that if the virtual tangent point of an edge lies inside the bounding ellipse, then the edge must intersect the ellipse in two locations (once at entry and once at exit). Note that relative to all start and goal point locations, every infinitely extended line (edge) in continuous two dimensional space which does not intersect the segment *SG* between start and goal points, must have a virtual tangent point. If the line intersects segment *SG*, then the virtual tangent is taken to be this point of intersection.

The time required to compute the ellipse by our technique is $O(e \log(M/\varepsilon))$ where *e* is the number of map edges, *M* is length of the longest map edge, and ε is the maximum allowable error in binary search. The ellipse bound is most effective in reducing search space when the initial A* solution is a high speed avenue which uses much of the lowest map weight. If such is not the case, then the ellipse may become fatter than the map itself. Even so, the time expended to check each boundary edge for the existence of an intersection is negligible. Furthermore, it is possible to determine if an initial solution cost can generate a bounding ellipse that lies within the confines of the map border. Given a standard map is 100×100 square distance units, if the lowest cost on the map $\mu^*=1$, then an initial path cost of 281 $(= 100 * \sqrt{2} * 2)$ or more is guaranteed to generate an ellipse larger than the map. In such cases, there will be no intersections. However, for an initial path cost less than 281, ellipse construction is usually worthwhile regardless of the outcome. If many intersections necessitate binary searches, then search space reduction will usually be significant. If not, we lose very little time for intersection existence checks.

2. Ellipse Improvement

Once established, if the regions which determined optimal cost coefficient used to compute the ellipse are no longer contained therein, then a smaller ellipse can be computed on the basis of the new larger optimal cost coefficient within [Rich87]. Path annealing provides still another way to improve the ellipse. Since path annealing always examines complete solutions, the upper bound on the cost of the optimal solution can be reduced whenever a WS is found whose locally optimal path has lower cost than the best known solution. These upper bound improvements provide more opportunities to recompute and shrink the current bounding ellipse. Since some reductions to the upper bound may only be slight, it is better to prearrange a threshold for action on the ellipse. Recomputation of the ellipse should only be initiated when one or more edges will be completely removed from the search space. A new ellipse which only shortens existing boundary edges without eliminating them does not remove window sequences from the search space.

3. Edge-Pair Elimination

The bounding ellipse is a global constraint on the solution space. On the other hand, we have seen that Snell's Law is a local constraint on the way optimal paths cross boundary edges. Snell's Law can determine regions through which globally optimal paths are not allowed to pass. Such regions may be identified independently of start or goal locations, as long as they do not contain either. This is very valuable knowledge since it is independent of the problem instance, and thus, computable as a map preprocessing stage. Such constraints can be used by both systematic and local search algorithms. Although path annealing is satisfied with finding near-optimal paths, local regions which cannot contain optimal paths have little potential for containing near-optimal paths. While unusual problem instances are possible, we focus on the norm.

a. Shortcut Analysis

We generalize and extend a particularly useful constraint that was first proposed in limited form as a search pruning criterion by [Rich87] and [Rowe90b]. What we refer to as *shortcut analysis* determines if any globally optimal path can cross a specified *edge-pair*. An *edge-pair* is a pair of crossable edges which bound a common traversable region, and, therefore, can occur consecutively within a WS. Note that since obstacle edges are not crossable, then they can never be part of an edge-pair. We can prove that for edge-pairs with certain geometric arrangements and cost coefficients, no optimal path can ever enter the region bounded by the pair from interior points of both edges. Note that this does not preclude the possibility of bypassing interior points and enter the region via its vertices. However, a path segment which crosses a region on a pair of its vertices is considered to lie in all WS's that use this path as a left or right defining limit. Search space redundancy at vertex paths means that bypassing path segments will also be contained within other window sequences (WS). Interior crossings of edge-pairs eliminated by shortcut analysis need not be checked for globally optimal paths.

Shortcut analysis considers each edge-pair of the map. Every edge-pair has an associated set of three cost coefficients. These coefficients correspond to the weights of the three distinct regions which are bounded by the edges in the pair. Referring to Figure 4.3, consider the edge-pair E_a-E_b bounding common region R. The set of cost coefficients corresponding to E_a-E_b is $\{\mu_1,\mu_0,\mu_2\}$. If $\mu_1 < \mu_0$ and $\mu_2 < \mu_0$, then we consider R a high-cost region relative to E_a-E_b. In such a case, conditions developed in Theorems 4.1 and 4.2 (Appendix A) dictate how to determine if E_a-E_b can be eliminated as an optimal-path crossing site.

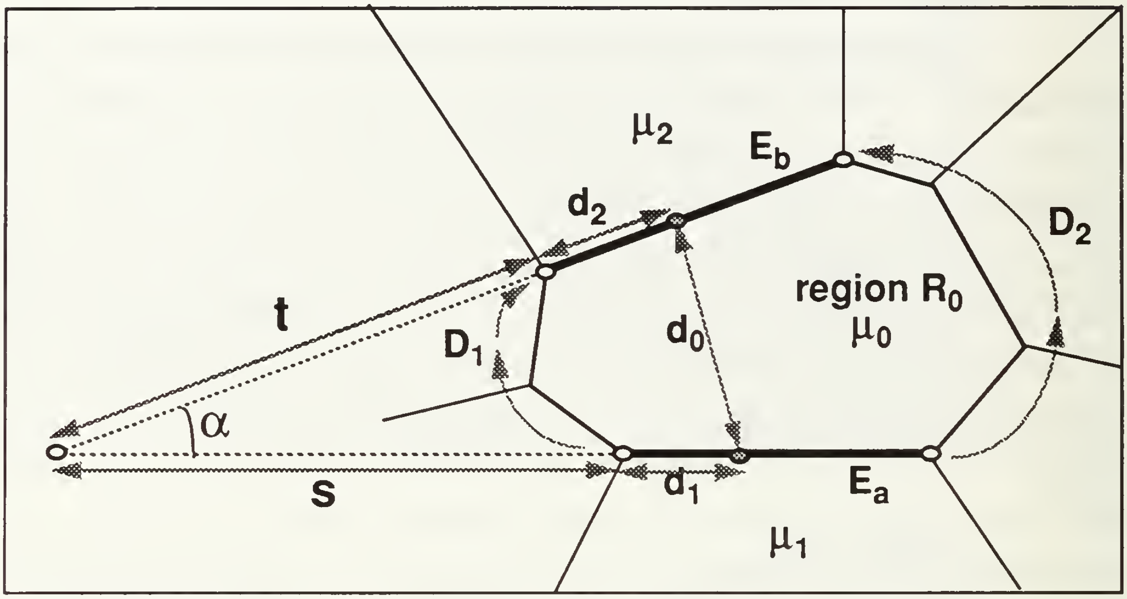


Figure 4.3 Shortcut Analysis on Edge-Pair E_a-E_b

The proofs of Theorems 4.1 and 4.2 permit us to use the following argument. In Figure 4.3 consider the continuous subspace of path segments connecting edge-pair E_a-E_b which cross through region R from interior points on each of E_a and E_b. If the unweighted distance of one such path segment is represented by d_0, then $\mu_0 d_0$ represents its weighted distance. Also connecting each pair of interior points of E_a-E_b are two *perimeter paths*, which detour around R by following along the edges bounding R_0, bypassing its interior points. Let the counterclockwise perimeter path be P_1 and the clockwise path be P_2. The weighted distances of P_1 and P_2 have both a fixed and variable component. The fixed component for each perimeter path is the sum of the weighted lengths of the bounding edges traversed by that path which are not in E_a-E_b. Let D_1 and D_2 represent these fixed weighted-length components for P_1 and P_2 respectively (see Figure 4.3). The variable components are dependent upon the path segment crossing R_0. Let d_1 and d_2 represent the variable-component perimeter-path distances from the crossing points on E_a and E_b. Then $\mu_1 d_1$ and $\mu_2 d_2$ must represent the corresponding weighted distances.

For the moment we will assume that regardless of the crossing location the cost of using P_1 will always be less than the cost of using P_2. To see why it may not be possible for a globally optimal path to cross R using interior points of E_a-E_b, consider the implication of proving that the ratio of the cost of P_1 to the cost of crossing through R is always less than 1:

$$1 > \frac{D_1 + \mu_1 d_1 + \mu_2 d_2}{\mu_0 d_0} \qquad \text{(Eq 4.4)}$$

Using the interior angle α of E_a-E_b, the fixed distances s and t from the projected intersection of E_a and E_b, and the Law of Cosines, Eq 4.4 can be rewritten so that μ_0 is a function of the crossing location on E_a-E_b as:

$$\mu_0 > \frac{D_1 + \mu_1 d_1 + \mu_2 d_2}{\sqrt{(s+d_1)^2 + (t+d_2)^2 - 2(s+d_1)\,(t+d_2)\operatorname{Cos}\alpha}} \qquad \text{(Eq 4.5)}$$

If Eq 4.5 is satisfied for all values of d_1 and d_2 along E_a and E_b respectively (i.e. for all path segments in the subspace), then a path segment through R from interior points of E_a-E_b will always be more expensive than a perimeter path. This function has a maximum value which can be found using partial derivatives with respect to d_1 and d_2. If the true value of μ_0 exceeds this maximum value, then no optimal path can cross through region R using interior points of E_a-E_b because a perimeter path is always cheaper.

Theorems 4.1 and 4.2 in Appendix A cover mathematical details. The constraints provided by these theorems can tag or eliminate from a weighted-region map edge-pairs which cannot have an optimal paths crossing them. Eliminating very many edge-pairs can reduce the search effort required of systematic search algorithms, including the A* search of the edge dual-graph used by path annealing to establish an initial solution. Though crude, the A* search for an initial solution is a major part of path annealing. Therefore, constraints on the size of the edge dual-graph which enhance A* search efficiency will benefit path annealing as a whole. For many problem instances it returns the window sequence very close to or containing the globally optimal path. The sooner annealing can find an optimal or near-optimal path, the more valuable the algorithm becomes in the case of time-constrained execution.

b. Critical Angle Analysis

Sometimes the geometry of an edge-pair is such that we can avoid tedious computations required for shortcut analysis, while still eliminating the edge-pair. This is particularly true when the interior angle α between an edge-pair is close to π radians. *Critical-angle analysis* is a Snell's-Law based technique which uses the critical-angle rays from the four vertices of an edge-pair to determine if any reflective path from either edge can intersect the other. If not, then it follows that no Snell's-Law path can exist between interior points of the edges because the critical angle is the limiting case of Snell's-Law bending on the high-cost side of a boundary edge. Since all optimal paths which cross the interior points of a boundary edge (not at a vertex) must obey Snell's Law, then no optimal path can cross between interior points of the edge-pair. Thus, the edge-pair can be eliminated. This is similar to the concept of *shadowing* in [Ross89]. Shadowing refers to the propagation of constraints which restrict possible movement directions.

A typical situation for critical-angle analysis is depicted in Figure 4.4. Note that $\mu_0 < \mu_1$ and $\mu_0 < \mu_2$, otherwise reflection cannot occur from within region *R*. An algorithm to implement critical-angle analysis begins by computing four rays, each of which emanates from a different vertex in edge-pair E_c-E_d. The direction of each ray must be inward toward the interior angle α and must form a critical angle with respect to the edge from which it emanates. Next, the intersection points P_1, P_2, P_3, and P_4, of the four rays with the opposing edges are computed by assuming that edges E_c and E_d extend infinitely. Finally, check the positions of the intersection points relative to the vertices of the edge segments. If both pairs of intersection points lie completely to one side or the other of their respective line-segment vertices, then no Snell's-Law path can exist for E_c-E_d, and it can be tagged such.

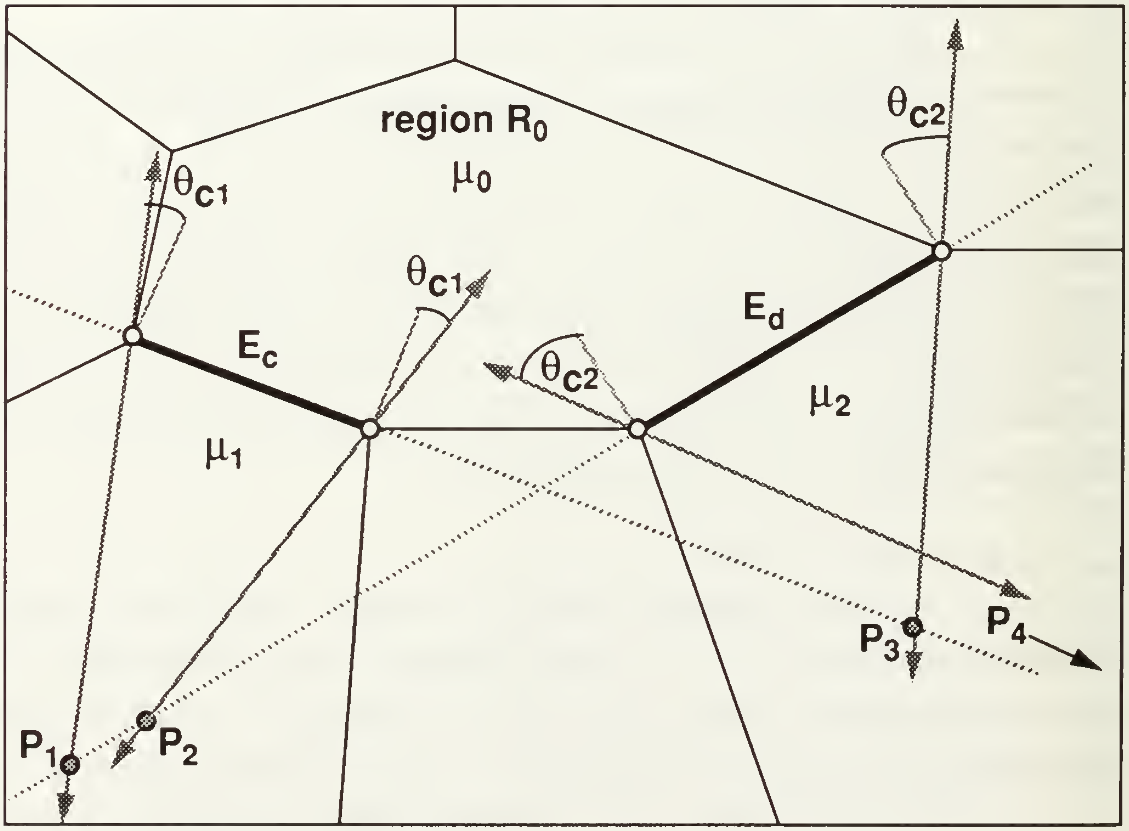


Figure 4.4 Critical-Angle Analysis of Edge-Pair E_c-E_d

c. *Virtual Obstacles*

A given region cost coefficient may be so high relative to its surrounding environment that all of its edge-pairs can be eliminated from the solution space. Such regions are called *virtual obstacles* [Rowe90b]. Virtual obstacles are equivalent to real obstacles, unless the start or goal resides within. It is possible to flag such regions and generate obstacle edge-lists so that they may be jumped during annealing. On the other hand, this may not always be prudent. Recall from Section III.F.1 that large obstacles can hamper the ability of annealing to move freely over it because large cost-increasing transitions may be required to do so. During high-temperature this would be bad for annealing, since the search should explore the solution space across as broad an area as possible. We discuss other ways of handling this situation in the next section on heuristics.

C. HEURISTIC IMPROVEMENTS

We have discussed restriction of the WRP search space using both global and local constraints. These are strong restrictions because they provide an absolute guarantee that no globally optimal solution will be excluded. We now present several *heuristic* techniques which can improve the performance of path annealing. Heuristics are "rules of thumb" which strike a compromise between being simple and making correct choices [Pear84]. Since it is probabilistic and, therefore, cannot guarantee the global optimum, simulated annealing is itself a heuristic approach. Furthermore, for other optimization problems, the annealing technique usually requires larger execution times [Naha85]. However, the heuristic enhancements to be discussed demonstrate the capability to reduce execution times significantly while improving the cost of the final solutions. In combination these heuristics make path annealing very competitive with pure A* search algorithms, particularly in larger problem instances.

1. Rapid Cost Function Evaluation

Recall from Chapter III that to find the locally optimal path within a window sequence is a two step process. The uniform-discrete-point (UDP) technique returns an optimal path constrained to discrete points spaced at an interval δ on each edge in the window sequence. Subsequently, beginning from this approximation, single-path relaxation (SPR) uses a Snell's-Law lookup table to iteratively adjust crossing points until no further improvements can be made. For a single window sequence (WS), the procedure is relatively efficient. However, it becomes less so when we consider that the cardinality of the set of all WS's is at least $O(v2^v)$ where v is the number of interior vertices in the map [Rowe90b]. Thus, we examine heuristic methods designed to increase the efficiency and frequency of cost function evaluation.

a. Relaxation Threshold

In a combinatorial optimization problem the distribution of the number of states having a given value of cost is called the *density of states* [Whit84]. For some large complex problems the density of states resembles the shape of a normal distribution function where the tails represent global extremes – the number of maximum cost states on the right and minimal cost states on the left (see Figure 4.5). Having bounded the search space (either by a bounding ellipse or by the natural border of the map) and using a move generator which tends to remove cyclic WS's, we may expect a WRP instance to have a similar density of states, where each state is a window sequence (WS) and its cost is the cost of its locally optimal path. The stochastic nature of the simulated annealing algorithm requires that we evaluate every state (i.e. WS) encountered in search. However, a relatively low density of optimal and near-optimal solutions suggests that frequent applications of single-path relaxation are unnecessary. Although upward biased, uniform-discrete-point (UDP) approximation is fast and gives a reasonable estimate of the single-path relaxation (SPR) result. For most WS's, UDP approximation suffices to estimate their nearness to a global optimum. Furthermore, that UDP approximation can still effectively guide an annealing-based search is confirmed by other researchers who have observed that exact cost function evaluation is not always necessary [Tove88, Adam85].

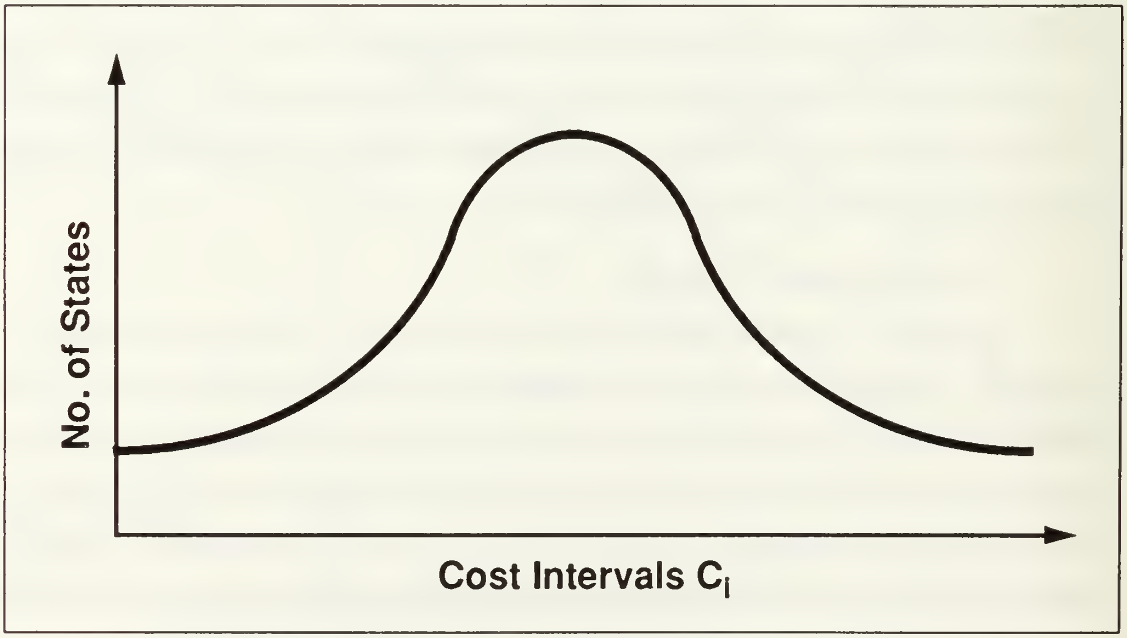


Figure 4.5 Density of States: Number of States in Cost Interval C_i

On the other hand, we do not want path annealing to overlook an optimal or near-optimal solution if it happens to sample one. This could happen since the cost returned by UDP is only an approximation. To control this possibility, we establish the *relaxation threshold*, $C^*+\Delta k$, where C^* is the cost of the *best* known solution (also the current upper bound on the optimal solution cost). The second term, Δk, is a fixed value established at the start of path annealing. The relaxation threshold represents the condition which triggers single-path relaxation: if the UDP cost is less than or equal to $C^*+\Delta k$, then SPR is applied to determine if the refined cost of this WS can improve C^*.

To determine the fixed value of Δk for a given problem instance, we assume a worst case situation for a UDP approximation based upon the discrete spacing of edge-points, δ. Since UDP must consider only paths constrained to cross at edge-points, then it can be forced to return an approximation with path segments which detour in non-optimal directions. For relatively straight WS's, the worst error in UDP approximation generally occurs where the algorithm must select path segments whose directions are perpendicular to the optimal direction. The least flexibility for choice of segment usually occurs in the start and goal-containing regions, because the start and goal are fixed points. Toward the middle of the window sequence more flexibility is available, and more path segments exist which are less perpendicular to the optimal direction, and which tend to self-correct and dampen the effects of errors made at the start and goal. Also, consider the locally optimal path of a window sequence (WS) which has many bends. More bending generally implies that the locally optimal path will cross many end points (vertices) of boundary edges. Since the edge-points used by the UDP approximation always include the vertices of a boundary edge, error is minimal at these locations.

Based on the foregoing discussion, we relate Δk to δ and the cost coefficients of the start and goal-containing regions. Figure 4.6 illustrates a WS in which a locally optimal path (dotted) is approximated by a UDP constrained path (solid). Edge-points are shown as solid circles. We have constructed this figure so that the locations of start and goal points force initial path segments from start and goal in the UDP approximation to be almost perpendicular to the true optimal direction. Under the assumption that this is the worst case UDP approximation for δ spacing and μ_s and μ_g for start and goal-containing regions, then the maximum difference between the UDP and SPR costs through this WS is $0.5\delta(\mu_s+\mu_g)$. Our empirical observations using vertex rotations only (i.e. proper WS's) confirm this calculation. Being conservative, we define

$$\Delta k = 0.5\delta(\mu_s+\mu_g) \qquad \text{(Eq 4.6)}$$

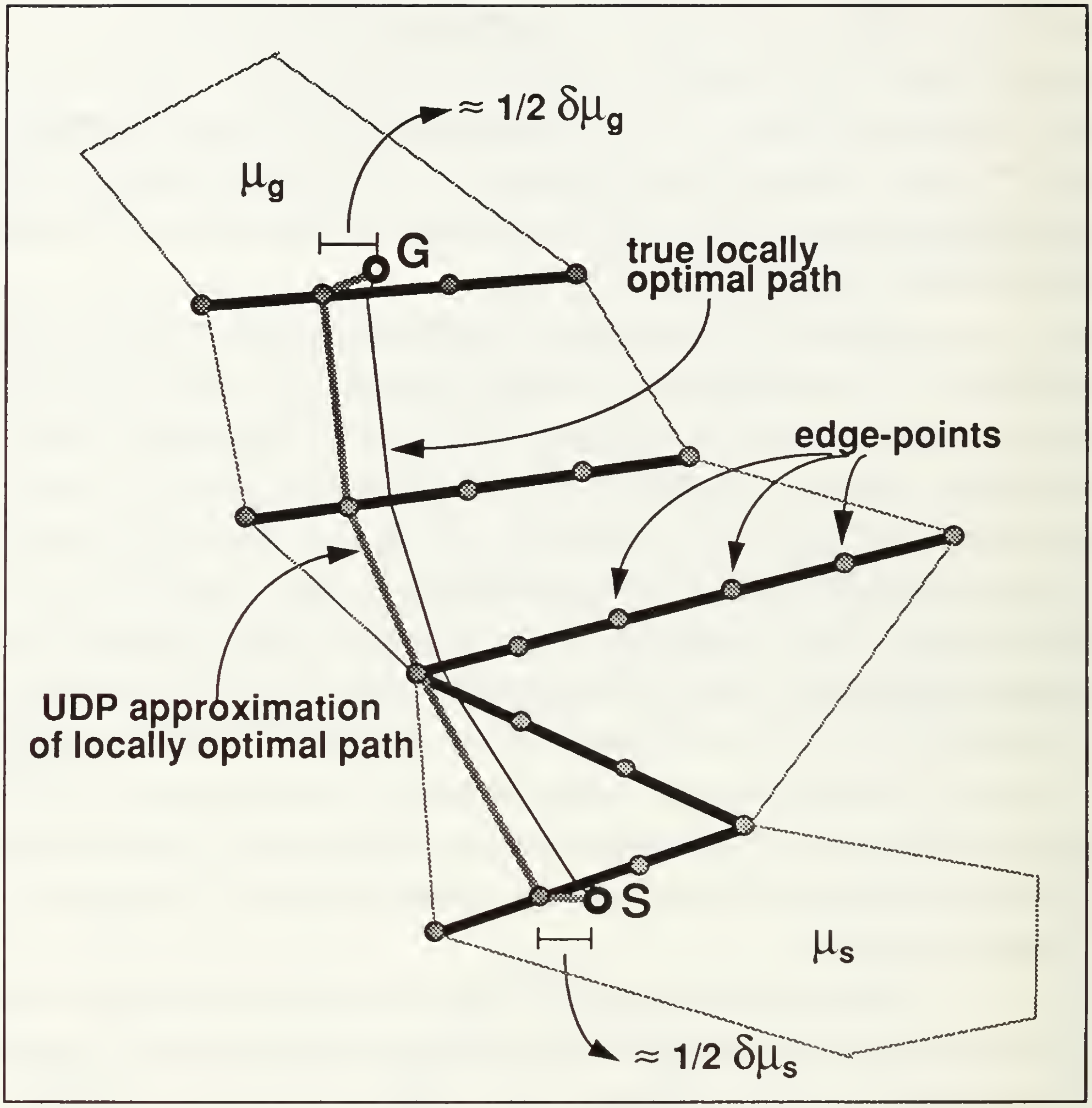

≈ 1/2 $\delta\mu_g$
μ_g
G
true locally optimal path
edge-points
UDP approximation of locally optimal path
S
μ_s
≈ 1/2 $\delta\mu_s$

Figure 4.6 Uniform-Discrete-Point Approximation of a Locally Optimal Path

The reentrant installation operator has a special relation to the relaxation threshold. For WS's with reentrant paths, the cost difference between its UDP approximation and SPR may grossly exceed Δk. This happens when a reentrant-installation move is accepted by the annealing process, but this move does not improve cost. This is because UDP is forced to reflect on edges installed for reentrant paths whether or not they are improvements, while SPR simply ignores such reentrant edges. This does not invalidate our calculation of relaxation threshold, because these reentrant WS's reside in the middle to upper ranges of the density distribution, and therefore never contain optimal paths. However, when a cost-improving reentrant occurs, the cost difference between UDP and SPR for it does not exceed Δk. Therefore, an optimal reentrant path will be recognized if its UDP cost is below the relaxation threshold, $C^{*}+\Delta k$.

The relaxation threshold provides two advantages for path annealing. First, as implied earlier, it indicates when a WS cost is far enough from C^{*} that a refinement with SPR is guaranteed to be useless. Furthermore, as C^{*} is updated, the frequency of SPR application will tend to decrease. This results in a significant time savings at high temperature. As temperature T decreases, SPR may be required more frequently. The second advantage results from the global effect of UDP approximation. It is generally recognized that simulated annealing performs best in a search space with a smooth cost function [John89]. Although UDP approximation raises the relative costs of all WS's, it tends to raise the larger ones less because their bending usually causes locally optimal paths to pass through many vertices. Recall that vertices are always available to UDP as edge-points, and crossings through them will be very accurate. Thus, to the annealing process the search space tends to be globally smoother, except where there is good potential for optimal or near-optimal solutions.

2. Caching Path Costs and Window Sequences

It is possible to stumble upon the globally optimal WS and corresponding path during annealing. So it always makes good sense to maintain a record of the best known path and update it as necessary. At the conclusion of annealing, the final solution returned will be that with the best known cost, C^{*}. While this is not standard annealing, most researchers use this variation (e.g. [Carr90, John89, John90]).

The number of proper WS's in a map is at least $O(2^{v})$ where v is the number of interior vertices. Relative to many of the NP-Hard problems, this is not an exceptionally large search space. Bounds and constraints (e.g. the bounding ellipse) can shrink the space even more. Therefore, path annealing quite often can repeat cost function evaluation on a WS it has already evaluated before. Recall that cost function evaluation is an expensive operation. Therefore, caching path cost for WS's avoids unnecessary uniform-discrete-point approximations. Before any WS is evaluated (with UDP search or SPR), a hash value is computed as a function of the edge identifiers in the WS. This function may include all identifiers in the sequence or some predetermined subsequence. The hash value is used to check the cache of previously stored WS's and their associated costs. If the WS is found in the cache, then its cost is returned without recomputation. Otherwise, WS cost function evaluation proceeds, and the results are cached for future use.

Although we have explored only a few hash functions, we find that better performance results from those which are computationally simple rather than those which represent elaborate attempts to minimize collisions. Our prototype computes a hash value as the sum over all integer edge-identifiers in the WS (except for *s* and *g*). Collisions are handled by linear chaining of linked lists. Limited experiments indicate that use of this hash function results in a 20% to 30% savings in path annealing execution time.

3. Move Generator Heuristics

The generation of transitions generally requires less work than does cost function evaluation. This is a result of the data structure support discussed in Chapter III. However, since move operators are the means by which annealing traverses and samples a large solution space, their design is very important to annealing results. Good operators are apparently difficult to design because problems and applications vary widely [John89]. For example, annealing on the Traveling Salesman Problem has been tested with several different operators, and some perform significantly better than others [Otte89]. In path annealing, we desire that the move generator be capable of causing rapid movement, over a broad area of the search space, and with as little repetition as possible. To enhance these qualities, we introduce several heuristic techniques.

a. Reduction of Repetition

Recall from Chapter III that to maintain transition efficiency, we designed only two very simple move operators: rotation (over a vertex or an obstacle) and reentrant installation. We also implied that the cumulative effects of single applications of these operators over time usually result in great movement across the search space. However, when a single vertex has many edges incident to it, a WS which crosses several of these edges can often waste time rotating back and forth across the vertex. The effect is similar to a physical *standing wave*. In general, the larger the number of edges incident to the vertex, the larger the probability that this vertex will be chosen as the site of rotation (See Figure 4.7). Caching of WS's helps to reduce wasted time, but does not prevent each repetition from counting as an attempted transition, thereby incrementing the annealing chain parameter, *L* (see Section III.F.3). While some repetition is good and allows annealing to return to more promising locations for more thorough search, the standing wave effect represents a stagnation in the sampling effort.

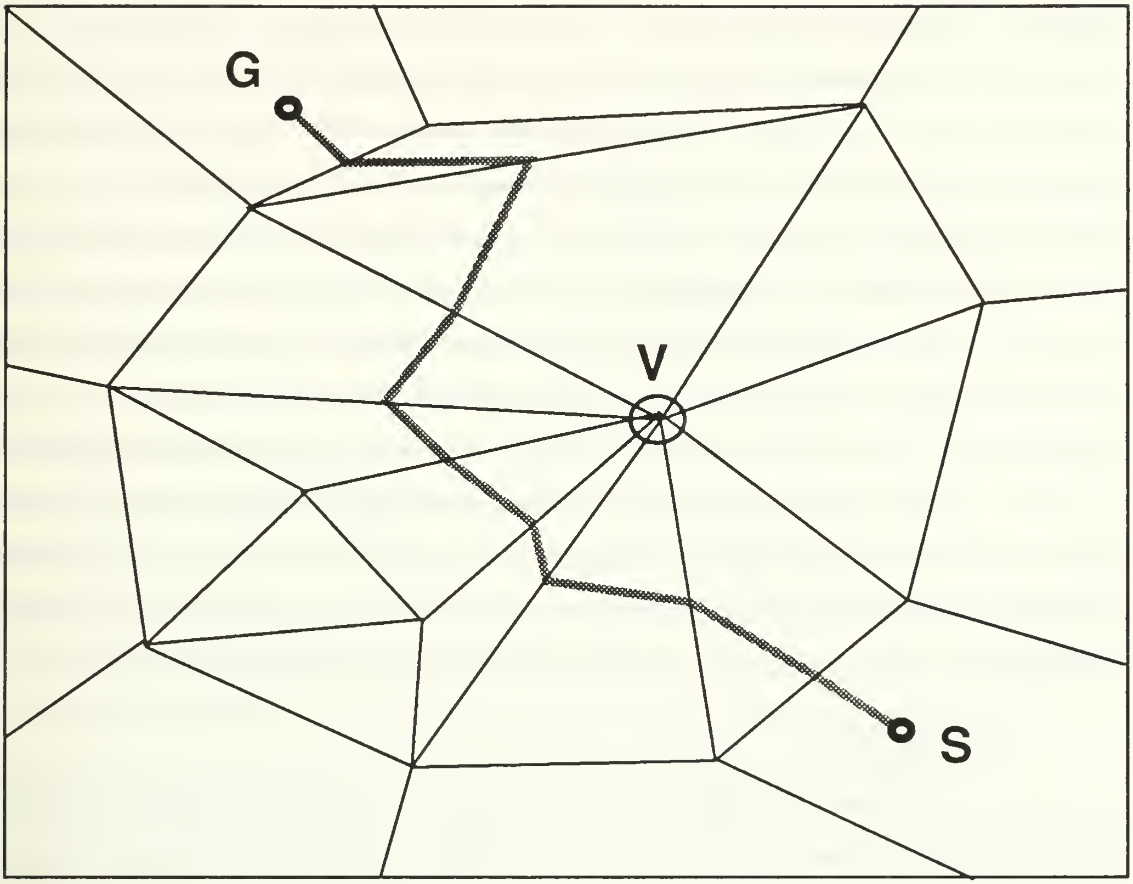


Figure 4.7 Vertex *V* with Many Incident Boundary Edges

To lessen the probability of window-sequence standing waves, for each attempted rotation we induce one additional rotation. This is done by randomly selecting an edge in the new WS that is incident to the original rotation vertex, and forcing rotation across its other vertex. With this technique we have essentially spread the selection probabilities over a wider range of edges, making it more unlikely that a random rotation will undo the transition of its predecessor.

b. Relation of Move Operators to Temperature

The double rotation technique discussed in the last section tends to cause broader movement across the search space than single rotations do. For some problem instances, we find that use of double rotation results in the discovery of lower cost paths (even the globally optimal path) more often. The explanation for this lies in the relation of the magnitude of the attempted transitions to the annealing temperature. [Whit84] indicates that better annealing performance results from matching move size to temperature. High temperature annealing should attempt larger scale moves, while low temperature annealing should attempt smaller scale moves. The intuitive explanation for this is that at high temperature the higher acceptance rates will permit uninhibited sampling of the solution space. At this time, large moves are the most efficient means for moving quickly over a broad area of the space and will be accepted more easily. At lower temperatures it is inefficient to attempt large moves. Since they usually cause great changes in the cost function, many will not likely be accepted, especially if the annealing process has already found a reasonably low cost state. On the other hand, small scale moves can refine such a state efficiently by making local improvements or by climbing out of localized minimums which are not as good as others close by.

Our prototype does not attempt to dynamically adjust or synchronize the move size with temperature. The effects of doubling vertex rotations seem to indicate that introducing multiple rotations at high temperatures could improve path annealing performance. However, the next two sections explore *tunneling* and *cycling* which are alternate methods for scaling up transition size for high temperature.

c. Tunneling

A sequence of neighboring window sequences within the WRP solution space which have similar costs that are relatively low compared to the surrounding environment is called a *high-speed corridor*. A given problem instance may have several high-speed corridors. Furthermore, these corridors may be separated by short or long physical distance, and by high or low weighted regions. In some instances high-speed corridors cross over one another, possibly in multiple locations. Path annealing gradually tends to focus its attention on one or more of these corridors as temperature decreases. The assumption is that during high temperature, the annealing process will have spent enough time roaming around all high-speed corridors to determine which would be best to refine at lower temperatures. Thus, at some low temperature the current WS will have committed to particular corridors from which it will likely not escape.

This is essentially the behavior we desire. However, consider the following question: Have other high-speed corridors been searched as thoroughly as the one in which the process has settled at low temperature? The answer to this question depends upon the starting temperature, T_0, and the annealing chain length, L, the parameter defining the number of moves that will be attempted at each temperature, T. Recall from Chapter III that our method of setting T_0 ensures that the transition acceptance rate will be 95%. This implies that the means for finding virtually all high-speed corridors is available. However, L should be large enough to ensure exploration of such corridors is sufficient for annealing to select and refine the good ones at lower temperatures. In theory the parameter L should be some function of (most often equal to) neighborhood size [Aart89]. However, in practice good values for L apparently vary from one application to another and should probably be set empirically [John89].

Extending annealing chain length, L, or raising starting temperature, T_0, only risks longer execution times with little additional assurance of successfully finding and searching the best high-speed corridors. Instead, we suggest two techniques which take advantage of the fact that the WRP represented as a dual edge-graph has a smaller search space relative to many NP-Hard problems.

The initial WS returned by A* search of the dual edge-graph usually corresponds to the location of a high-speed corridor. As we have said before, this initial WS can, but does not necessarily have to contain the globally optimal solution. Either way, we assume that it has a relatively high likelihood of being in close proximity to the optimal path. Now suppose we determine the *n* shortest midpoint paths and their corresponding window sequences in the dual edge-graph. Under the above proximity assumption, path annealing started from each of these window sequences at a lower value of T_0 will have a higher probability of locating the optimal path. However, we must be careful. The second (and third, etc.) shortest midpoint path within the dual edge-graph may very well reside within the very same high-speed corridor containing the first shortest midpoint path. If so, then we are simply wasting time finding *n* best paths which annealing will probably examine anyway because they are all very close together. Therefore, what we really desire are the *n* most high-speed corridors, which may or may *not* be the *n* shortest midpoint paths. This reasoning is very similar to what [Tove88] refers to as a *neighborhood prejudice swindle*. The basic objective is to reduce computing time and increase the likelihood of finding good solutions by concentrating on the most promising neighborhoods.

To find a second corridor, we can use knowledge within the initial WS. The WRP optimal paths maps generated by [Alex89] reveal that for a given fixed goal point, the vertices and boundary edges of low-cost regions tend to attract many optimal paths. Relatively high-cost regions tend to repel them. On the basis of this observation, we can find a second high-speed corridor which is physically separated from the initial by forcing a second A* search to avoid the edges of regions most responsible for making the initial solution low-cost Therefore, by temporarily removing one or more long, low-cost edges from the original dual edge-graph, a second A* search is driven to find an alternate low-cost midpoint path which is usually in a different corridor. The procedure is equivalent to temporarily blocking a high-speed corridor with one or more large obstacles. If the second WS lies very near the initial WS, then this is strong evidence that this corridor contains the optimal path, since A* search was not able to find a corridor with regions low enough in weight to draw it away from the first.

At least one of the edges chosen for temporary removal from the graph must be within the initial WS. Otherwise, the second A* search will return the same WS. In general, the edge in the initial (or previous) WS which bounds the start or goal-containing region is the one whose removal will maximize the difference in physical location between successive A* search solutions. The reason for this is that removal of such an edge usually forces a large shift in the initial direction of search from the initial region (containing the start or goal). Consequently, the resulting midpoint path may have to use low-cost regions in areas of the map which were not advantageous when leaving the initial region through the removed edge.

Having located several different high-speed corridors, the annealing process may proceed from a lower temperature either sequentially starting from each or concurrently on all high-speed corridors together, returning the best path found over all. Concurrent execution would require that one initial WS be selected as the starting solution. Since there would be *n* independent *current* window sequences, several possibilities exist for concurrency. We could apply random transitions by queuing the current WS's. Or the move generator could randomly select one current WS on which to apply a move operator. We could control random selection of a current WS by annealing for a short interval of time (perhaps random), after which annealing shifts to a WS representing a different corridor. Each time this shift is performed, the algorithm effectively *tunnels* through high-cost peaks in the solution space. One way to trigger tunneling is to treat it as another move operator. It could also be driven by results in the current corridor. For example, tunnel to another WS if the current cost has not improved after *n* consecutive transitions. There are certainly many other possibilities for multiple-corridor execution. Our prototype implements only a single initial A* solution. However, we have observed that a second corridor generated as described above increases the capability to locate optimal paths.

We can also implement tunneling by relying upon high temperature annealing to locate several WS's representative of different high-speed corridors. Later at lower temperatures annealing may tunnel back to these corridors via the WS's. Since we cache each new WS along with its uniform-discrete-point estimated cost, the process can actually tunnel to any WS of its choosing. One problem with this method of tunneling is that it may require very high starting temperatures to ensure broad uniform coverage of the map and discovery of high-speed corridors. Annealing can easily sample WS's which are generally physically close together in one corridor, separated only by a few small cost gradients. However, movement to other corridors may require jumping one or more very high cost peaks. Larger transition size can help to induce greater WS movement (for example, by allowing multiple vertex rotations for each transition attempted by the move generator). But to climb higher cost gradients still requires higher temperatures. We can save time by finding and caching the locations of WS's representing high-speed corridors at the start of path annealing, This avoids the time otherwise required by annealing to climb the upward gradients between these corridors.

Recall that for efficient tunneling we desire a set of WS's which lie in, and thus, represent distinctly different high-speed corridors. Therefore, between WS's, we need some measure of physical similarity or proximity within the search space. Window sequence (WS) costs alone cannot distinguish physical location. One simple measurement is to determine the set of regions entered by each WS and compute the set difference between them. If the resulting set is empty or has very small cardinality, then the WS's probably belong to the same corridor. Otherwise, they are sufficiently distinct. With data structures to support edge to region translation, such a comparison could be relatively efficient.

Our prototype employs a very limited form of tunneling. It records each WS found to contain a path whose cost is lower than the currently best known. The program does not check that these WS's represent distinct corridors. Often they do not; however, if several WS's have accumulated by the time the freezing temperature, T_f, is reached, then the set usually represents more than one corridor. At temperature equal to T_f the program sequentially tunnels to each WS in this set. To each WS it applies pure iterative improvement by attempting L random transitions. At this stage the relaxation threshold is no longer active, and single-path-relaxation (SPR) is applied to each WS. The best path found overall is returned as the solution. Although this technique also depends upon the ability of high temperature transitions to reach major high-speed corridors, it does give some assurance that locations in which good solutions were discovered early have been more thoroughly explored.

4. Faster Initial Solutions

A* search through the edge dual-graph to obtain a starting solution is a major component of path annealing. Although its resolution is crude (midpoints only), this phase of the algorithm generally accounts for roughly 10% of total execution time. In Chapter III we argued that starting annealing from the shortest midpoint path is most appropriate. However, we also mentioned that the primary purpose of A* search is to obtain a *feasible* starting solution. Having one of the best solutions helps, but is not necessary.

Suppose we desire to implement tunneling using A* search to locate five high-speed corridors (first method of tunneling). Assuming that one A* search requires 10% of the total execution time, the total amount of time required to locate five corridors now exceeds 50% of the total path annealing execution time.

This time may sometimes be reduced by reusing partial paths left on the agenda generated by the first A* search. However, if the time required to obtain initial corridors approaches 50% (or more), then we might reconsider whether the time advantages of our stochastic approach are still worthwhile. As Chapter V will demonstrate, such execution times are still less than the times required for a more refined global A* search from start to goal with a resolution capable of discovering the optimal path.

However, with heavier dependence on the annealing phase of the algorithm, we can improve the crude A* search for starting corridors. Recall that admissible A* search computes an underestimate to the goal for each midpoint it reaches. This underestimate is the product of the straight-line distance remaining to the goal and the lowest cost coefficient, μ^*, and ensures that the search returns the shortest (midpoint) path. If we assume that no path exists between start and goal which can travel exclusively in the regions of weight μ^*, then we may relax the value of μ^* by enlarging it. If our assumption is correct for a higher optimal cost coefficient, then the A* search remains admissible and still returns the shortest midpoint path. If it is incorrect, then the A* search will be inadmissable, and the midpoint path may not be the shortest midpoint path. However, we have saved some time in finding an initial solution in exchange for gambling on the ability of annealing to start from a possibly bad solution and still find a good one.

In every problem instance some threshold value for the optimal cost coefficient exist which removes admissibility, and consequently, the shortest-path guarantee. Values higher than this threshold will generally cause A* search to overestimate weighted distance to the goal from more of the midpoints. In turn, the search is driven to the goal more directly and more quickly, because the overestimate increases its depth-first behavior. Using conservative optimal-cost-coefficient overestimates coupled with temporary removal of low-cost boundary edges from the dual edge-graph, is a good way to efficiently obtain an initial set of WS's representing different high-speed corridors.

5. Heuristic Bounds on Search Space

Section B.1 of this chapter described our method for constructing the bounding ellipse of [Rich87] and [Rowe90a] which restricts the WRP search space. Recall that we use the lowest cost coefficient, μ^*, in this calculation. The use of μ^* is conservative and safe, and results in a bounding ellipse which cannot exclude any portion of a globally optimal path. However, more complex maps may have cost coefficients which vary over a wide range of values. For such maps, we indicated that the bounding ellipses for many problem instances are often larger than the map itself. This holds true even when the ellipse is computed on the basis of a globally optimal path. The reason is that for a given arrangement of start and goal, travel through the lowest-cost regions may require detours through high-cost regions which nullify their efficiency. Unfortunately, determining if or when an arrangement of start and goal can efficiently use the lowest-cost regions of a particular geometry appears to require solving the WRP.

To constrain a bounding ellipse to its minimal size requires *a priori* knowledge of the optimal path cost. Assume that for a given problem instance we have such knowledge. Let the cost of the optimal path be C^*. Also, let the total unweighted length of the optimal path be D^*. We define its *average path weight* as

$$\mu^+ = C^* / D^*. \qquad \text{(Eq 4.7)}$$

Average path weight is a rough measure of *path efficiency*. If the globally optimal path has very low average path weight, then the locations of start and goal are apparently favorable to use of many low-cost regions (not necessarily including the lowest cost region) and the optimal path is very efficient. On the other hand, if the optimal path has very high average path weight, then low-cost regions either do not contribute significantly or their efficiency is overshadowed by passage through high-cost regions. With the value of μ^+ we can construct a bounding ellipse which is smaller than that which uses the lowest cost coefficient. Figure 4.8 illustrates what we refer to as a μ^+*-ellipse*.

As implied earlier, we cannot compute a μ^+-ellipse without prior knowledge of the optimal path. Using the A* search through the edge dual-graph we can obtain a least upper bound for C^*. Ideally, we desire a greatest lower bound on μ^+. However, we settle for a heuristic approximation for μ^+. Then, in exchange for a small risk of overlooking the optimal path, we can often achieve a marked performance improvement due to a more restrictive search space.

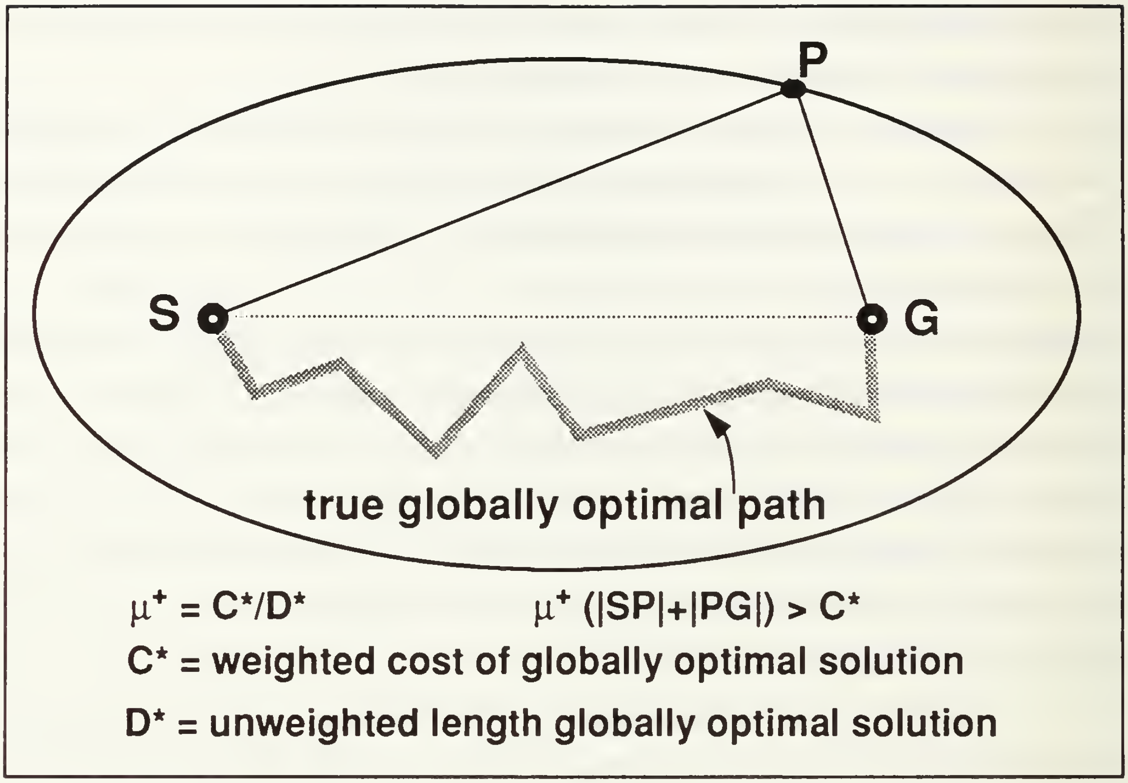

P
S
G
true globally optimal path
μ+ = C*/D*
μ+ (|SP|+|PG|) > C*
C* = weighted cost of globally optimal solution
D* = unweighted length globally optimal solution

Figure 4.8 μ^+-Ellipse

One fast, yet conservative method of approximating μ^+ is to use the initial window sequence (WS) returned by A* search. Although crude (midpoints only), this search has entered most regions of the space in the immediate vicinity of start and goal, so that, intuitively, it provides a reasonable assessment of the distribution of cost coefficients between start and goal. Let the locally optimal path cost within this initial WS be C_0, and let its total unweighted length be D_0, an upper bound on the globally optimal path cost. To establish a conservative estimate for μ^+ we make a simple heuristic assumption about the upper bound for the globally optimal path-cost. Similar to the reasoning used to determine relaxation threshold in Section C.1.a, we compute

$$\Delta K = 0.5(\mu_s+\mu_g)l_{\max} \qquad \text{(Eq 4.8)}$$

where $l_{\max}$ is the length of the longest edge bounding the start and goal-containing regions with cost coefficients μ_s and μ_g respectively. A heuristic least upper bound on the optimal path cost is $C_0-\Delta K$. Figure 4.6 shows why the discrete interval δ is used to determine relaxation threshold. Initial A* search through midpoint paths is much less refined than UDP through edge-points with a spacing of δ. Therefore, the factor 0.5δ in Eq 4.6 is replaced by $0.5l_{max}$ to obtain Eq 4.8. From the above we estimate μ^+ as:

$$\mu^+ \approx (C_0 - \Delta K) / D_0 \qquad \text{(Eq 4.9)}$$

This comes from the observation that optimal paths generally do not stray unreasonably far from the regions between start to goal. In military operations there are often considerations which prohibit movement along paths with such extreme detours (for example, designated areas of operation or the requirement for adequate maneuver space between units). Also note that the procedure is useful for both path annealing and systematic search approaches.

D. SUMMARY

In this chapter we have presented and discussed some techniques which can improve the performance of path annealing. The prototype we have designed employs most but not all of these enhancements. We have concentrated on bounds and heuristics which have shown the most potential to maximize execution speed and probability of location optimal paths.

Throughout the first four chapters we have stressed the performance advantages of path annealing over purely systematic search methods such as A* search. In Chapter V we will present our own version of a global systematic search algorithm which is similar to but faster and more accurate than the wavefront propagation algorithm referenced in Chapter II. It is against this algorithm that we compare path annealing.

We wish to emphasize that path annealing, though primarily a stochastic algorithm, makes important use of A* search. In fact, it is fair to say that path annealing draws a major part of its performance and power from its A* search component. Rather than just another application of simulated annealing, we prefer to describe path annealing as an intelligent marriage of both systematic and local search (both continuous and discrete) under probabilistic control. Chapter V will present empirical results from testing of our path annealing prototype.

V. EMPIRICAL STUDIES

A. TEST PROCEDURES AND MAPS

1. Input Maps

To gain some understanding of how path annealing performs at different levels of input map complexity, we tested the algorithm on four maps. These maps (Figures 5.1, 5.2, 5.3, and 5.4) represent three levels of increasing complexity in terms of resolution and cost coefficient range. Cost coefficients are shown as integers within respective regions, and obstacles appear as gray areas in each map.

Map1 (Figure 5.1) is the least complex, and represents synthetic terrain created and used by [Rich87]. Map2 (Figure 5.2) is of medium complexity and also represents synthetic terrain of our own design. Map3 and Map4 (Figures 5.3 and 5.4) are the most complex test maps. These maps represent our own weighted-region interpretation of a 1000 meter grid square of actual terrain extracted from a standard military map of Fort Hunter-Liggett, California [Defe74]. Using a computer-assisted tool (see Appendix B mapmaker.pl), we constructed these maps on the basis of combined elevation, vegetation, and terrain feature data. Figure 5.5 is a plot of the elevation data used to make Map3 and Map4. Figure 5.6 is a magnified reproduction of this area as it appears on [Defe74]. Figure 5.7 is an aerial photograph of the area taken by aircraft. Map3 and Map4 represent the same section of terrain in different seasons. Map3 represents a wet season during which the reservoir (large obstacle in the left half of Map3) is full. Map4 represents a dry season during which the reservoir is completely dry (the case at the time of this writing). The validity of these maps was also confirmed by a site visit. However, we wish to emphasize that the cost regions assigned were of necessity our own interpretation of combined topographic effects.

In an effort to ensure unbiased testing, we have also created three additional test maps (Figures 5.8, 5.9, and 5.10) with the geometry of Map4, but in which cost coefficients have been randomly scrambled. The topological character of each map is identical to Map4. However, to create Map5, Map6, and Map7 we randomly assigned integer cost coefficients from the sets {1,2,3,...,10}, {1,3,5,7,9}, and {1,4,7} respectively. These maps are intended to test cost functions with varying degrees of smoothness.

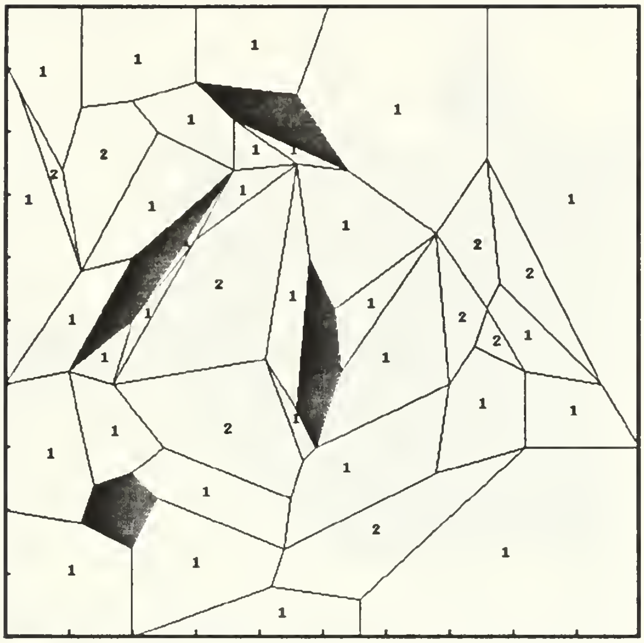


Figure 5.1 Map1

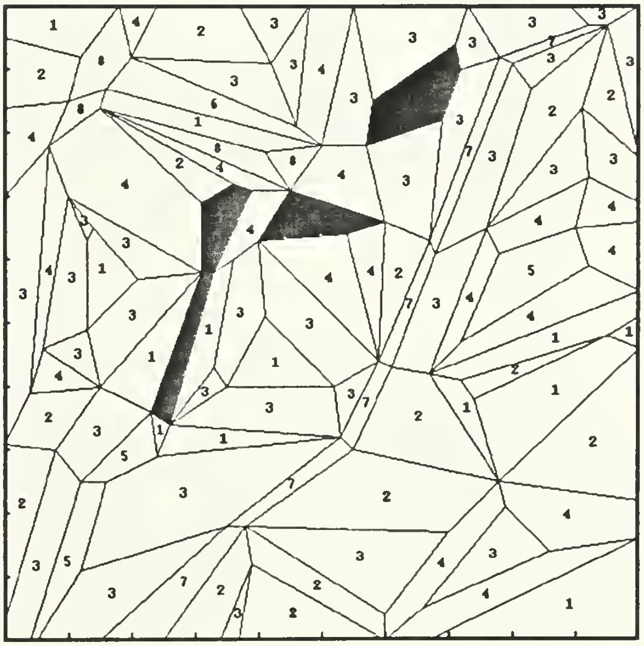


Figure 5.2 Map2

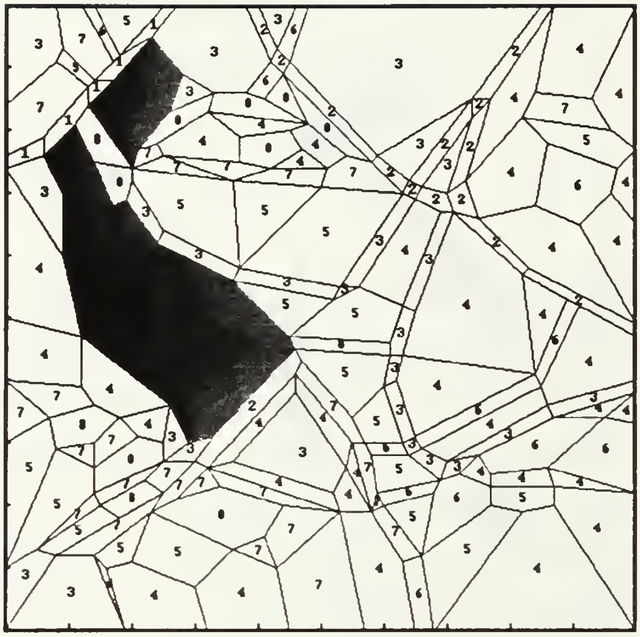


Figure 5.3 Map3

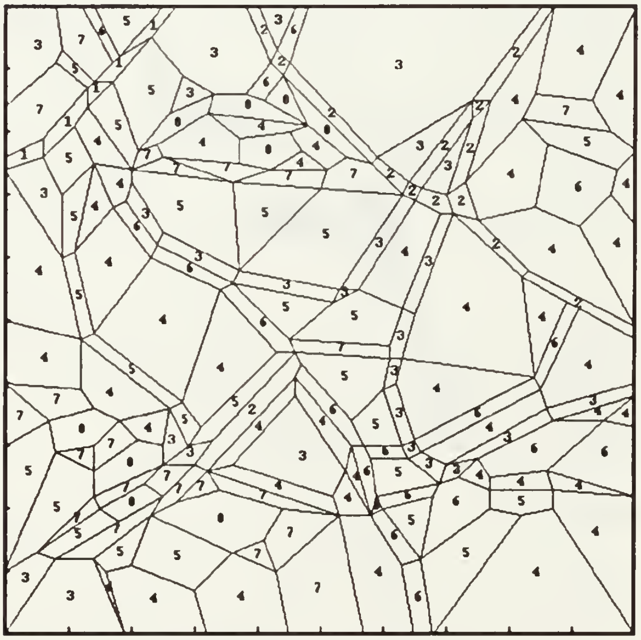


Figure 5.4 Map4

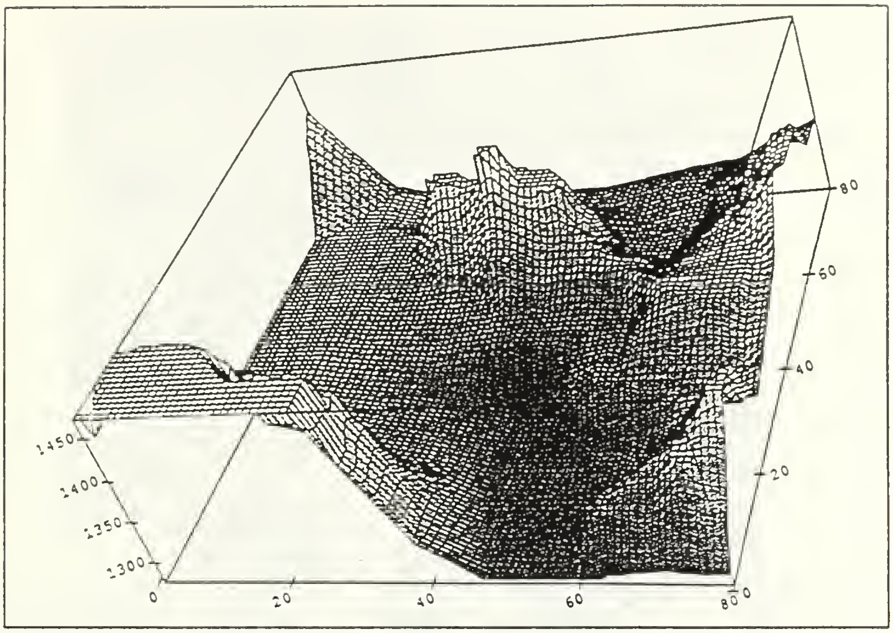


Figure 5.5 Plot of Elevation Data Used to Create Map3 and Map4 (elevation above sea level; tick marks represent 12.5 meter intervals)

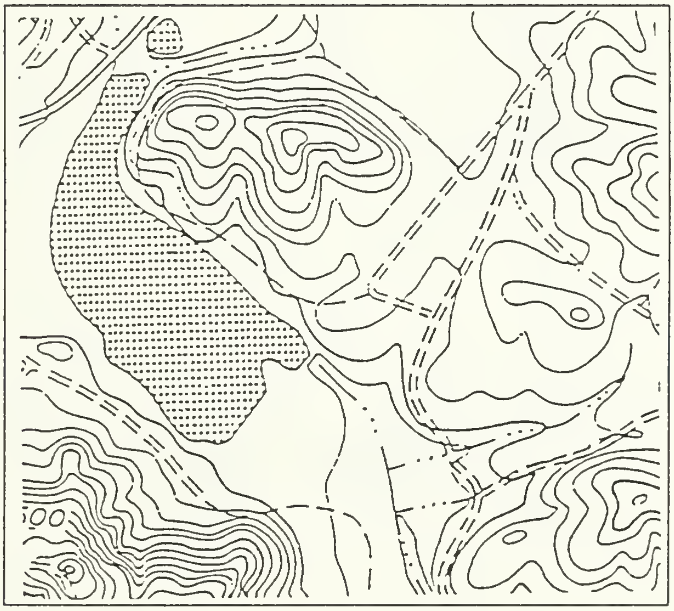


Figure 5.6 Extract of Military Map [Defe74] Used to Create Map3 and Map4

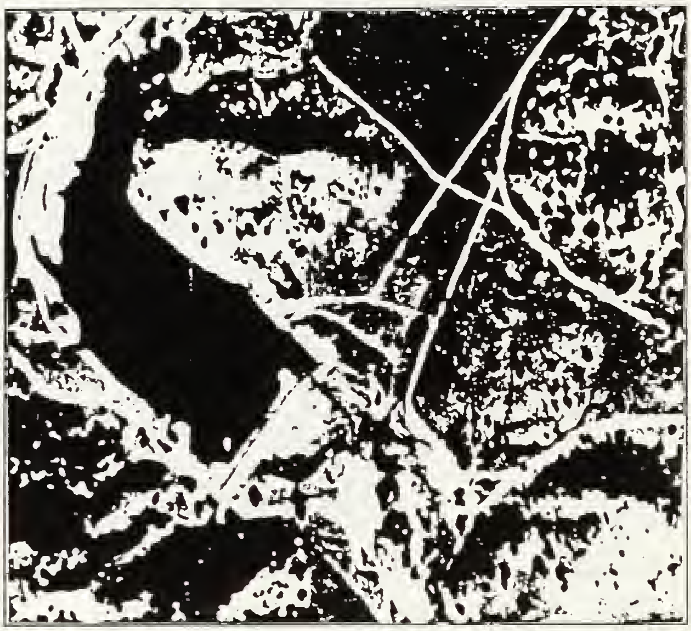


Figure 5.7 Aerial Photograph Used to Create Map3 and Map4

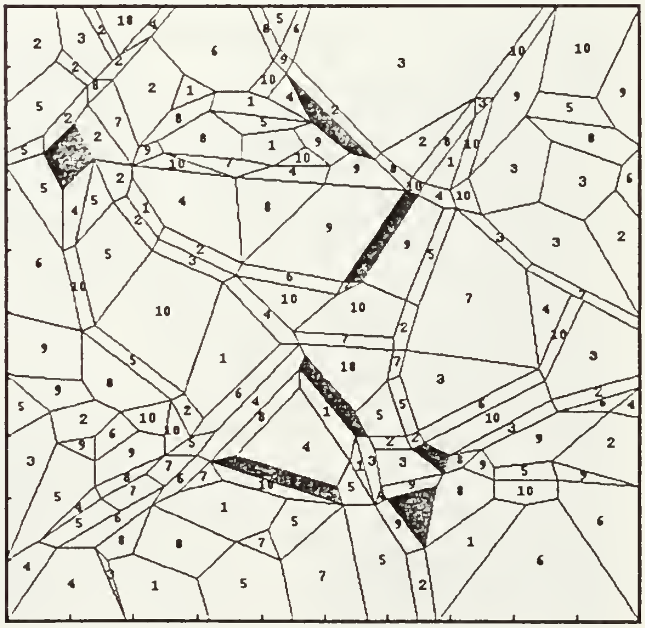


Figure 5.8 Map5

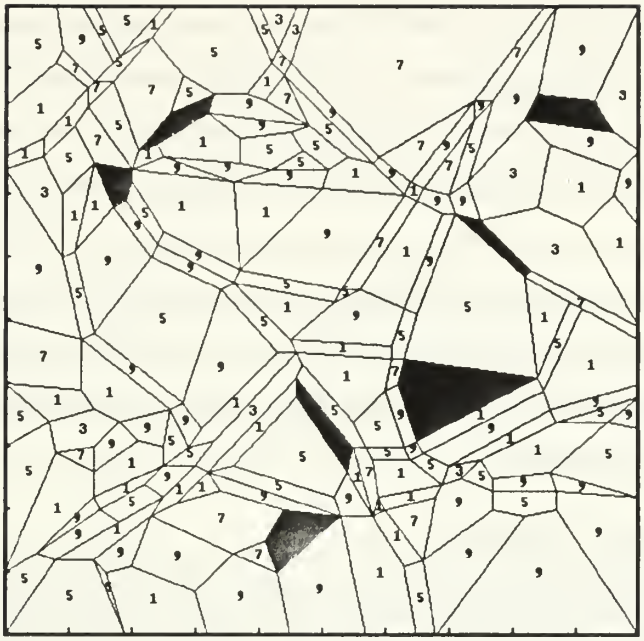


Figure 5.9 Map6

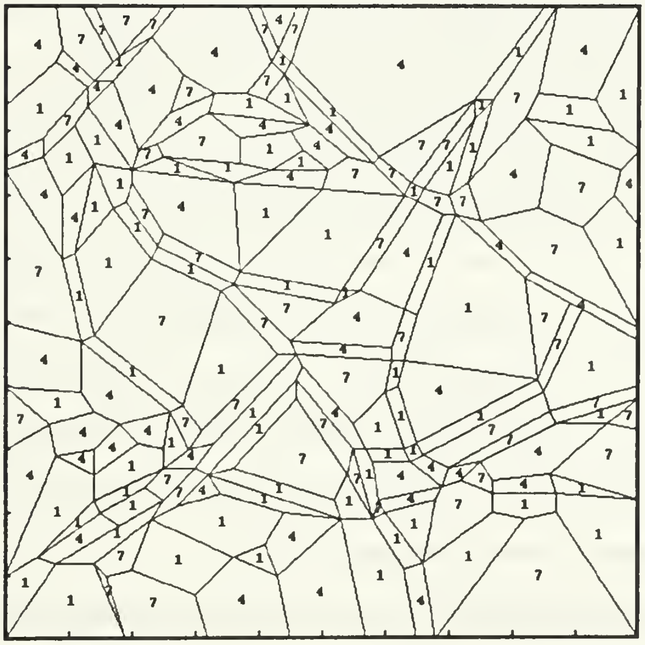


Figure 5.10 Map7

2. Control Algorithm

In Chapter II we discussed previous approaches to solving the weighted-region problem (WRP). To measure the performance of path annealing, we would like to place it in direct competition with these other WRP algorithms. However, most of the existing algorithms have limitations which prevent direct comparison on the test maps we have designed for path annealing. The Continuous Dijkstra Algorithm [Mitc90] has never been implemented. The Recursive Wedge Decomposition prototype of [Rich87] is limited to maps with two distinct cost coefficients and obstacles. Furthermore, it is not designed to handle maps in which vertices have more than two incident boundary edges. A future update to [Rowe90b] and [Alex90] will include a revised prototype to account for these restrictions. However, this prototype will construct optimal path maps instead of solving for specified start/goal pairs.

a. Wavefront Propagation

A limited comparison of path annealing to some of the empirical results of [Rich87] is possible using Map1. Recall that Map1 is taken directly from the work of [Rich87]. We can compare solution quality directly in terms of total path-cost and physical location for identical problem instances. However, a direct comparison of execution times is difficult since different machines and Prolog dialects have been used. Nevertheless, we can compare timing results with [Rich87] indirectly through the grid-based wavefront propagation algorithm. To do so, we have written similar code to perform branch-and-bound search on an 8-neighbor lattice graph representing the homogeneous-cost regions of Map1.

To provide the maximum advantage to the wavefront algorithm, the search graph and all weighted distances are precomputed independently of the wavefront algorithm itself. The cost of each arc in the graph is the product of its Euclidean length and assigned weight. For an 8-neighbor implementation there are only two lengths: R and $R\sqrt{2}$, where R is the orthogonal distance between grid intersection points. In the same manner of [Rich87], we weight each arc in the search graph by the cost coefficient of the region in which its midpoint resides. For our tests, grid resolution corresponds to a 100×100 cell lattice. Relative to test cases in [Rich87], our grid is of medium resolution. [Rich87] tested a low resolution (2:1) 64×64 grid and a high resolution (1:1) 128×128 grid. Thus, we expect solution quality which is generally comparable to [Rich87].

b. Uniform-Discrete-Point Global A Search*

As noted in Chapter II, grid-based wavefront-propagation solution path-cost is generally greater than optimal path-cost by some added percentage due to the "stair-step" effect. [Rich87] demonstrated that this added percentage can vary randomly with resolution. Besides the apparent random nature of the percent-added cost on a solution, the wavefront propagation has other drawbacks. Assuming we know the window sequence in which a solution resides, then the "stair-step" effect could be eliminated using single-path relaxation (SPR) to straighten the path. However, single-path relaxation only removes grid effects from the final solution path. So, path-cost errors resulting from grid bias can cause the systematic wavefront search to overlook the true optimal path. While higher grid resolution can improve accuracy, this also lengthens execution times.

To improve solution quality over grid-based wavefront propagation, we have developed our own simple algorithm, which we also compare against annealing. *Uniform-discrete-point global A* search* is a generalization of the uniform-discrete-point (UDP) method for approximating the locally optimal path in a WS to the whole map. The algorithm bears some resemblance to wavefront propagation but has several important advantages.

The uniform-discrete-point global A* (UDPGA*) search technique is quite simple and adapts naturally to the geometry and orientation of the boundary edges in the terrain map. As with wavefront propagation grid size, UDPGA* resolution is established *a priori* in the form of a discrete interval, δ. This concept is identical to that of the same name discussed in Chapter III Section D.2, so the same time trade-off applies. Along each edge of the map, we locate uniformly spaced discrete points where the spacing is bounded above by the value of δ. The algorithm proceeds in two phases. The first phase is a fast A* search through the dual edge-graph (on midpoints only). This is done in exactly the same manner as the A* search for an initial solution in path annealing. The purpose is to obtain an early upper bound on optimal path cost to construct a bounding ellipse (again, same manner as path annealing). Once the ellipse has bounded the search space, a refined A* search begins on the uniformly δ-spaced points. When the shortest path through these points has been found, it is returned with the WS in which it resides. We apply UDP/SPR once to that WS to find its locally optimal path. This path is returned as the solution.

UDPGA* is an approximating algorithm in the same sense as wavefront propagation. It solves the weighted-region problem (WRP) by systematically searching a discrete graph of some predetermined size. This size is a function of the discrete interval, δ, which sets the resolution of what is essentially a visibility graph. However, UDPGA* has several important advantages over wavefront propagation. First, it does not simulate movement through regions as uniform incremental time steps. Instead, the algorithm takes complete path segments through regions via the weighted arcs of the search graph. It crosses each region in a single step by selecting a path segment connecting a pair of discrete points on boundary edges. Second, UDPGA* makes more effective use of the terrain map geometry, because the discrete points lie on boundary edges. Path segments between these points conform to natural arrangements of edges and vertices in the map. This tends to reduce the error resulting from strict adherence to grid points. Third, UDPGA* actively searches for and can find reentrant path solutions. Finally, UDPGA* makes more effective use of the bounding ellipse than wavefront. UDPGA* can obtain an initial solution to compute the ellipse quickly and exactly because its search graph is oriented on boundary edges instead of a lattice.

3. Validation Criteria

We desire to compare performance in terms of two factors of weighted-region problem solutions: total weighted path-length and total execution time. As we indicated earlier in this chapter, test maps were generated to study path annealing performance in several cost function environments ranging from smooth to hilly. At the same time we wish to vary the size of the problem instance to examine time complexity as a function of problem size. We establish a fixed set of 12 points in the Cartesian plane of 100×100 units square, and we generate 66 distinctly different problem instances by pairing these 12 points as start and goal. Figures 5.11 through 5.13 illustrate the placement of these points. The general location of the points is based on our desire to ensure a wide variety in size and situation of problem instances while minimizing the advantage that either algorithm might gain from a start or goal point that is close to the border. This is an important advantage for both algorithms because it can reduce the search space up to 75%. Large obstacles can also contribute to this. In experiments with grid-based wavefront propagation, [Rich87] refers to the exploitation of this advantage as the *heuristic-selection* strategy. A more specific local adjustment of the points was made to avoid having a start or goal point reside within a real obstacle for any of the maps.

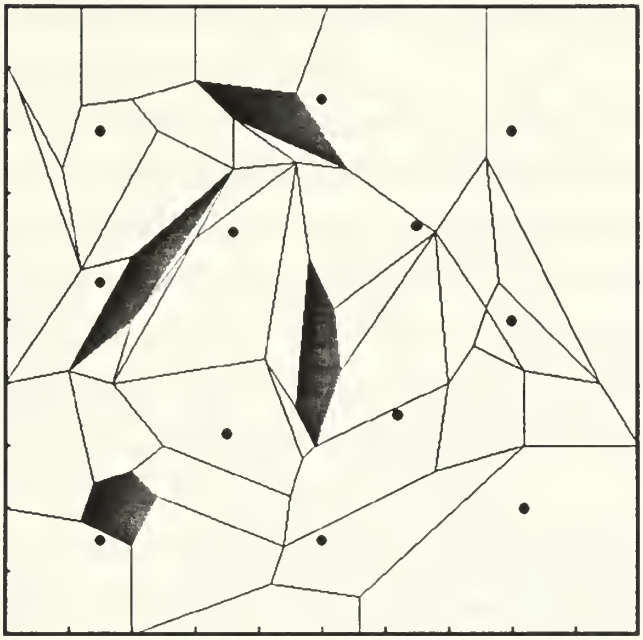


Figure 5.11 Placement of Start/Goal Points (Map1)

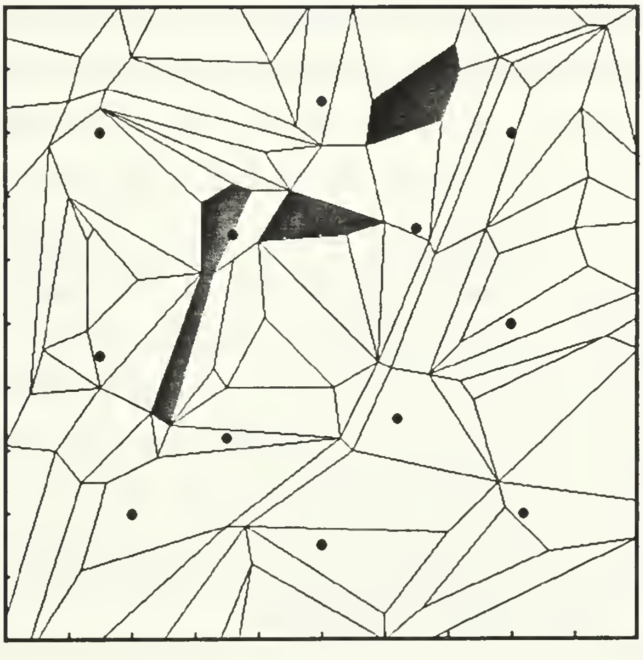


Figure 5.12 Placement of Start/Goal Points (Map2)

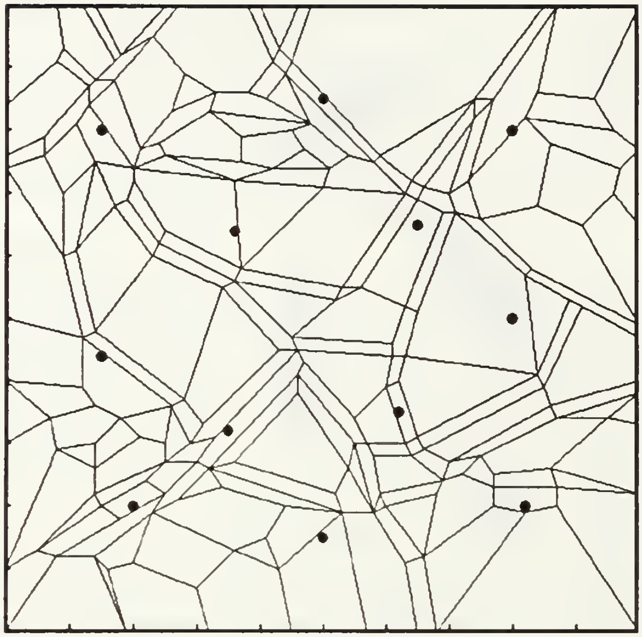


Figure 5.13 Placement of Start/Goal Points (Map3 through Map7)

In order to compare the algorithms we need a consistent measure of the size of a problem instance. The straight distance between start and goal is a rough measure, however it does not account for the resolution of that area of a map in which the search (path annealing or UDPGA*) is conducted. Both [Mitc90] and [Rowe90b] determine problem size as a function of the number of vertices searched. The search performed by our algorithms is oriented on the boundary edges. Thus, we estimate problem size as the total number of boundary edges reached by the initial A* search through midpoints. We count each edge only once regardless of how many times it might be visited during this search. This technique for estimating problem size approximates the number of boundary edges inside the initial bounding ellipse, since we have found that crude A* search generally expands edges in an area roughly equal to the size of this ellipse. The actual bounding ellipse is usually smaller because it clips off portions of longer edges at the limits of the A* search expansion. Note that the Euler equations establish a linear upper bound on the number of vertices that can exist in a connected planar graph of E edges, if the degree of all vertices is three or more [Prep87].

$V \leq 2/3\ E$ if for all vertices, v_i, Degree(v_i) ≤ 3 (Eq 5.10)

Wavefront-propagation problem-size would ordinarily be measured as the number of grid cells expanded in the search. However, in order to have a common measurement upon which to compare all algorithms, we use the number of boundary edges for wavefront as well.

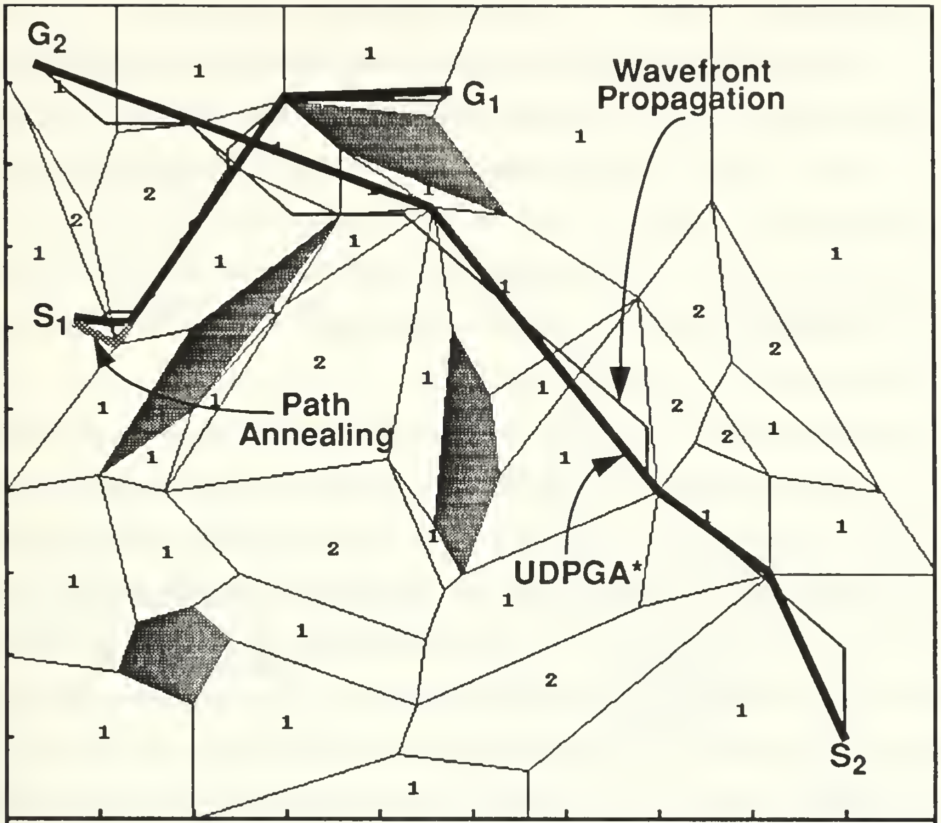
Execution time for each algorithm is computed as total central processing unit (CPU) time expended from the beginning of initial A* search to the return of the final solution. The time required to preprocess the map and create necessary data structures is not included in these times. We have also excluded most of the CPU time required to accumulate and summarize statistical data. Both algorithms are capable of the graphical display of outputs. We found that these facilities were extremely useful in debugging. While maintenance of these displays expends a significant amount of CPU time, such time is not included in the timing statistics. Thus, total CPU time for any given run should be a reasonably reliable indicator of relative search time.

Once we have established a relationship between wavefront propagation and uniform-discrete-point global A* (UDPGA*) solution's quality and timing, we will use UDPGA* search as our gold standard and compare solution costs accordingly. To compare the relative goodness of solutions for a given problem instance we take the ratio of the total weighted path-costs of the path-annealing solution to the UDPGA* solution. Thus, a ratio value of one indicates that both algorithms have achieved similar optimal cost results for a given problem instance. A value greater than one favors the UDPGA* technique. However, as results show, this ratio sometimes dips below one. This happens more often when the value of δ (the discrete interval defining the upper bound on spacing between edge-points) is not small enough to ensure that UDPGA* can find the true optimal solution.

B. ANALYSIS OF TEST RESULTS

1. Testing of Grid-Based Wavefront Propagation vs. UDPGA*

To support the claim that UDPGA* is an effective algorithm for comparing the performance of path annealing, and to relate both algorithms to the work of [Rich87], we present results which compare our prototypes with an implementation of grid-based wavefront propagation on Map1. Figure 5.14 shows the results of our algorithms for wavefront, UDPGA*, and path annealing on two problem instances tested in [Rich87]. The UDPGA* solution paths (shown in figure) are identical to those returned by Richbourg's recursive wedge decomposition algorithm. The path annealing solutions are identical to the UDPGA* solution paths with the exception of two very short segments in the first problem instance (identified in the figure). Our wavefront propagation paths (shown in the figure) are somewhat better than those of [Rich87] (not shown). However, recall that we use a lower resolution than the maximum resolution tested by [Rich87] (see bottom of Figure 5.14). Also, recall that the percent-added cost for wavefront solutions will vary somewhat randomly with resolution. The large difference in execution times between our wavefront algorithm and that of [Rich87] can be attributed to at least two factors. First, each algorithm was executed on different machine architectures with different implementations of Prolog. Therefore, timing results can only be compared in a relative sense. Second, our wavefront implementation does not use the bounding ellipse, whereas the wavefront implementation of [Rich87] uses a precomputed bounding box to reduce the search area. This also accounts for some of the execution time differences between our wavefront algorithm and UDPGA*. Computation of a bounding ellipse or box requires an initial solution which wavefront cannot easily obtain. Nevertheless, even if such a solution were provided at no additional cost, it could not adequately compensate for the execution time disparity. Furthermore, in the more complex maps, the bounding ellipse often circumscribes the entire map, and therefore, does not help either algorithm.



S_1 [8,61] - G_1 [48,89]	Cost	Time (sec)	Resolution
Wavefront	60.5	2131.5	100x100
UDPGA*	57.8	28.0	δ=3
Path Annealing	58.0	7.5	
* Wavefront [Rich87]	62.4	119.1	128x128
* Wedge Decomposition	57.7	51.9	

S_2 [90,10] - G_2 [4,92]	Cost	Time (sec)	Resolution
Wavefront	131.3	6414.2	100x100
UDPGA*	124.9	60.8	δ=3
Path Annealing	124.9	24.3	
* Wavefront [Rich87]	134.0	456.0	128x128
* Wedge Decomposition	124.9	117.0	

* Timing results are from different machine architecture and Prolog implementation

Figure 5.14 Map1: Two Problem Instances Comparing Algorithm Performance

To relate execution times as a function of problem size, we examine the performance of each algorithm on a set of distinctly different problem instances generated by start/goal pairings of the points shown in Figure 5.11. Only 52 of the 66 possible paired cases are tested. The remaining 12 cases are trivial and result in straight-line solutions. In such cases, there is no space to search because the bounding ellipse is degenerate, and therefore, it actually *is* the solution. For the uniform-discrete-point global A* (UDPGA*) algorithm we set the value of the discrete interval as δ=3. This results in a UDPGA* resolution which is usually more than adequate for finding the globally optimal solution. Also, this value of δ is consistent with that used in testing more complex maps.

The first three plots in Figure 5.15a are of running time in CPU seconds vs. problem size in terms of edges for wavefront propagation (100×100 grid), UDPGA* (at δ=3), and path annealing respectively. The last plot in Figure 5.15a is a quadratic least-squares curve fit for points associated with all three algorithms. We use a quadratic fit under the assumption that the area visited by wavefront, and thus execution time, should grow roughly as the circular area of search state expansion around the start point. Use of a bounding ellipse and A* search (as in UDPGA*) will tend to flatten this circular area. However, [Pear84] shows that an admissible A* search still expands an $O(n^2)$ number of states. The quadratic curves in Figure 5.15a show that execution-time growth rate of wavefront is much greater than both UDPGA* and path annealing.

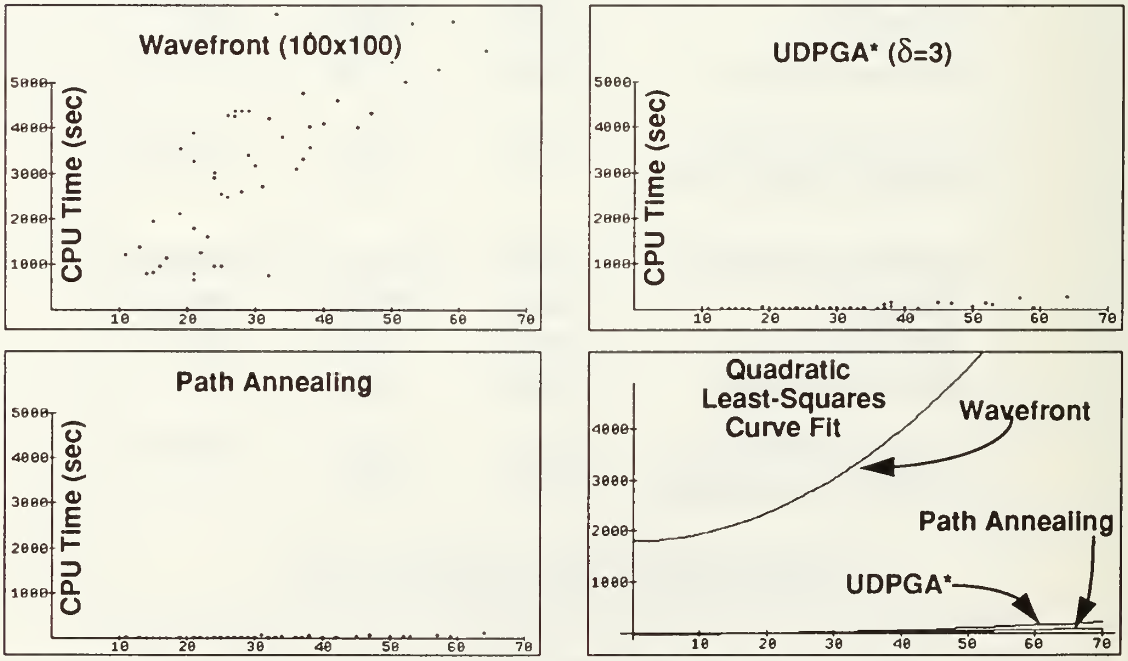


Figure 5.15a Map1 CPU Time vs. Problem Size (in Edges)

We also compared the timing results of wavefront propagation on the same 52 problem instances as before, but at reduced resolution. In this case, we run wavefront propagation on a 50×50 cell lattice. The results are summarized in Figure 5.15b. Note that even after halving the resolution, the execution-time growth rate remains significantly higher than both UDPGA* and path annealing.

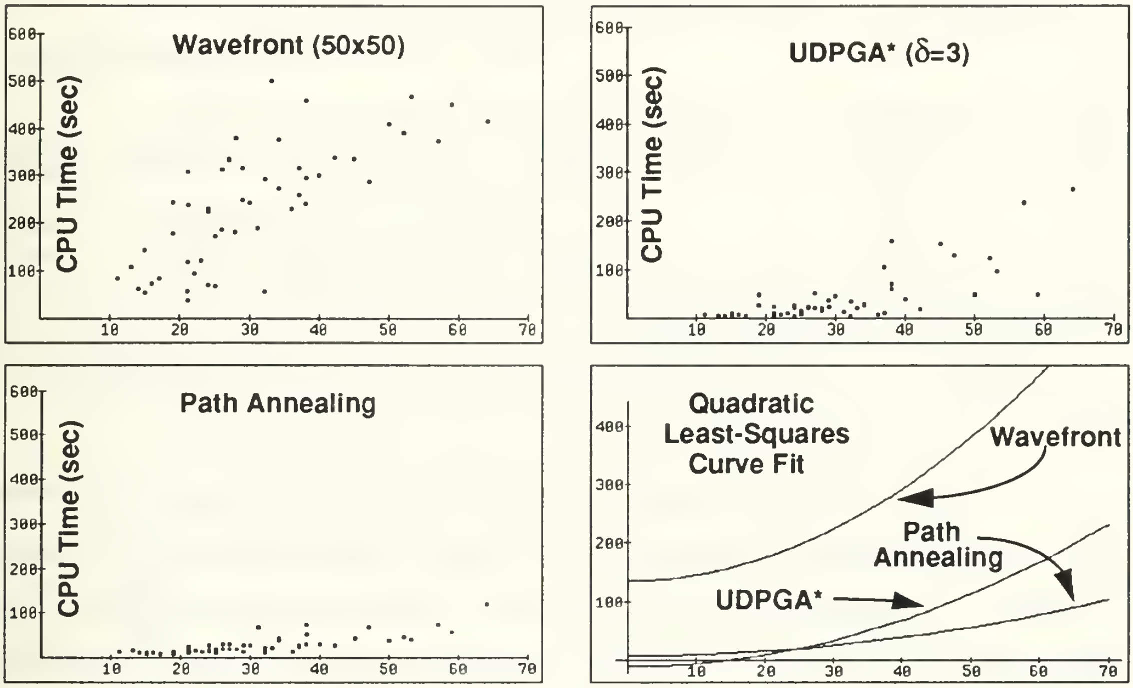


Figure 5.15b Map1 Cost Ratio vs. Problem Size (in Edges)

We examined the relative solution quality of the systematic search algorithms, by dividing the solution path-cost of wavefront propagation by that of UDPGA* for the same problem instance (path annealing will be compared to UDPGA* later). Thus, a value of 1.0 means that both algorithms obtained solutions with identical cost. Figure 5.16 shows cost ratio vs. problem size for the 52 instances. Note that in all cases, UDPGA* obtains a solution which is better than wavefront. This is because the "stair-step" effect of wavefront solution paths results in the percent-added cost discussed earlier. Straightening the wavefront solution paths would generally result in costs close or identical to UDPGA* as the two instances in Figure 5.14 tend to indicate. However, more computation will be required to locate boundary edge intersections and establish Snell's-Law crossings.

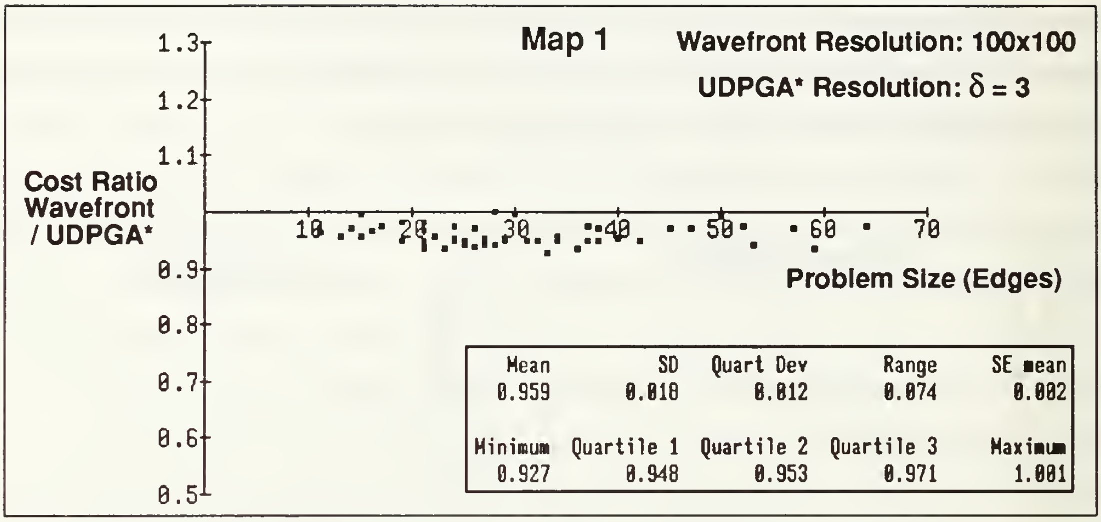


Figure 5.16 Map1 Cost Ratio (Wavefront / UDPGA*) vs. Problem Size

While, our results tend to highlight the disadvantages of wavefront propagation, we emphasize that there are some advantages and additional factors to consider. To its advantage, the wavefront algorithm is relatively easy to implement. If the terrain map is already in digital form, then it is likely well-suited to a wavefront approach. Conversion of digital terrain data into a connected planar map of homogeneous-cost polygons can be a very time-consuming task. On the other hand, we have found that lattice generation (essentially digitizing) of a weighted-region map is also a time-consuming task, depending upon the resolution required to capture an acceptable level of detail. Defense Mapping Agency provides terrain in both digital and planametric (points, lines, areas) forms [Digi88]. However, without data compression, digitized terrain will generally require more raw storage space than planametric format. Furthermore, many map objects which may affect cost per unit distance are not easily coded in digital form. For example, the location of a minefield is usually specified by reference coordinates points and boundary lines.

On the basis of the foregoing results, we can say that the UDPGA* algorithm is generally faster and more accurate than wavefront propagation. Therefore, UDPGA* provides better competition for path annealing than wavefront. The recursive wedge decomposition algorithm is also faster and more accurate than wavefront propagation, though it appears that it is not as time-efficient as UDPGA*. However, wedge decomposition is guaranteed to find the globally optimal path, whereas UDPGA* can overlook it depending upon the value of δ. Note that wedge decomposition has only been tested on relatively simple maps (such as

Map1). To determine time and solution quality relationships conclusively, wedge decomposition should be compared directly to UDPGA* and path annealing on identical machine architectures with identical language implementations. Furthermore, problem instances should vary over a much larger range of sizes, geometric situations, and cost coefficients.

2. Testing of Path Annealing vs. UDPGA*

a. Testing of Small Problem Instances

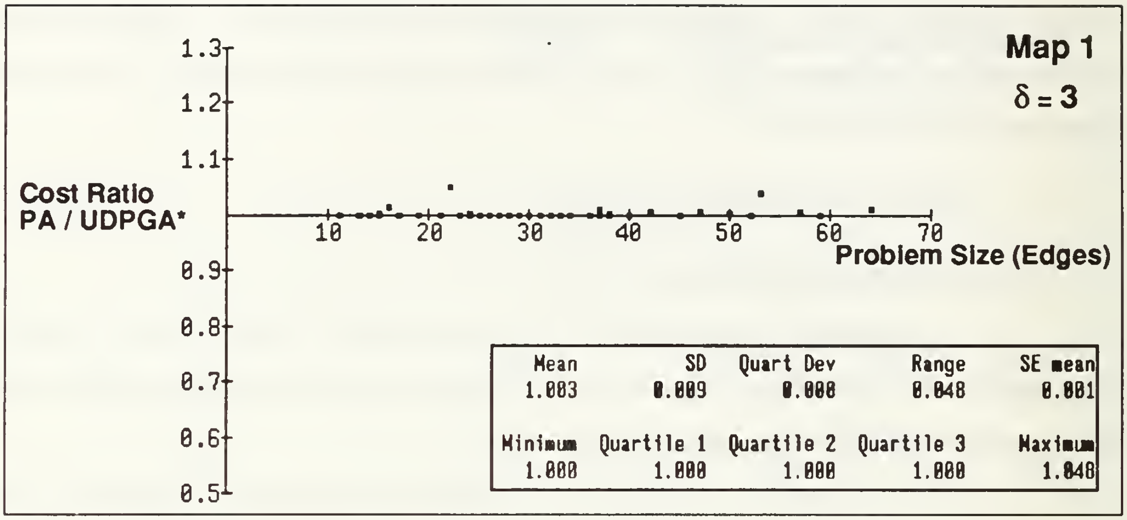
Having demonstrated that uniform-discrete-point global A* (UDPGA*) search is faster and more accurate than wavefront propagation, henceforth, we now focus our attention on path annealing performance using UDPGA* as our gold standard.

In the case of Map1, resolution could have been finer (for example, δ=2 or δ=1). However, we chose δ=3 to be consistent with our studies of the more complex maps. (Even UDPGA* at δ=3 requires over 24 hours of CPU time to solve all 66 cases for the larger maps.)

The path annealing schedule uses fixed value for T_f, and uses the cost of the initial A* solution as T_0.

- T_0 = cost of initial A* solution
- T_f = 1.0
- R = .80
- L = 5
- L_s = 4

First we consider a comparison of the goodness of solutions as measured by the ratio of solution costs. In Figure 5.17 we plot the ratio of solution costs of path annealing to UDPGA* as a function of the size of a problem instance in terms of the number of edges. The inset in this figure indicates some key statistics computed over the 52 cases. Note that the maximum difference from the UDPGA* solution was less than 5%, and that the majority of points lie on the axis (i.e. at a 1:1 ratio).



Mean	SD	Quart Dev	Range	SE mean
1.003	0.009	0.000	0.048	0.001

Minimum	Quartile 1	Quartile 2	Quartile 3	Maximum
1.000	1.000	1.000	1.000	1.048

Figure 5.17 Map1 Cost Ratio (Path Annealing/UDPGA*) vs. Problem Size

A major advantage of path annealing is execution time savings. Although we designed path annealing to be most efficient for larger problem instances, it does surprisingly well even for relatively small instances. Figures 5.18 and 5.19 are plots of central processing unit (CPU) time required as a function of problem size for UDPGA* and path annealing. These plots contain the same 52 problem instances compared to wavefront propagation earlier. However, by not displaying the wavefront data, we are able to expand the CPU time axis to better show the relationship of path annealing to UDPGA*. Even for these smaller instances the plots indicate that path annealing has lower average time complexity. To partially confirm this, we again apply a quadratic least-squares curve fit to each plot. The resultant timing curves are shown in Figure 5.20. A generalization about the quality of path annealing vs. UDPGA* cannot be made on the basis of these problem instances, since they are relatively small.

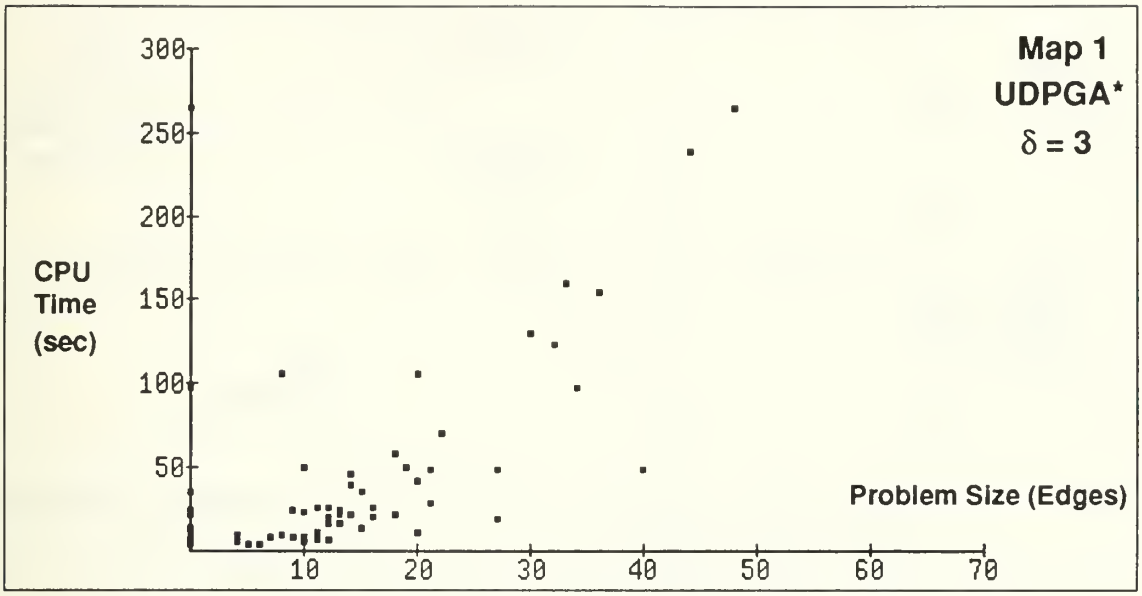


Figure 5.18 Map1 UDPGA* CPU Time Required vs. Problem Size (Edges)

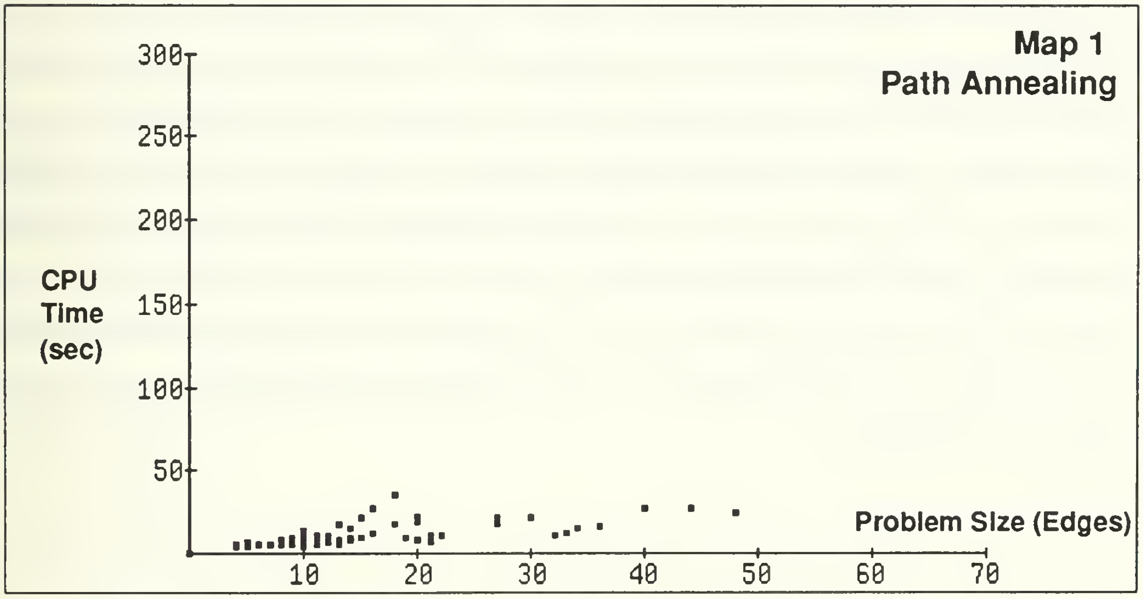


Figure 5.19 Map1 Path Annealing CPU Time Required vs. Problem Size (Edges)

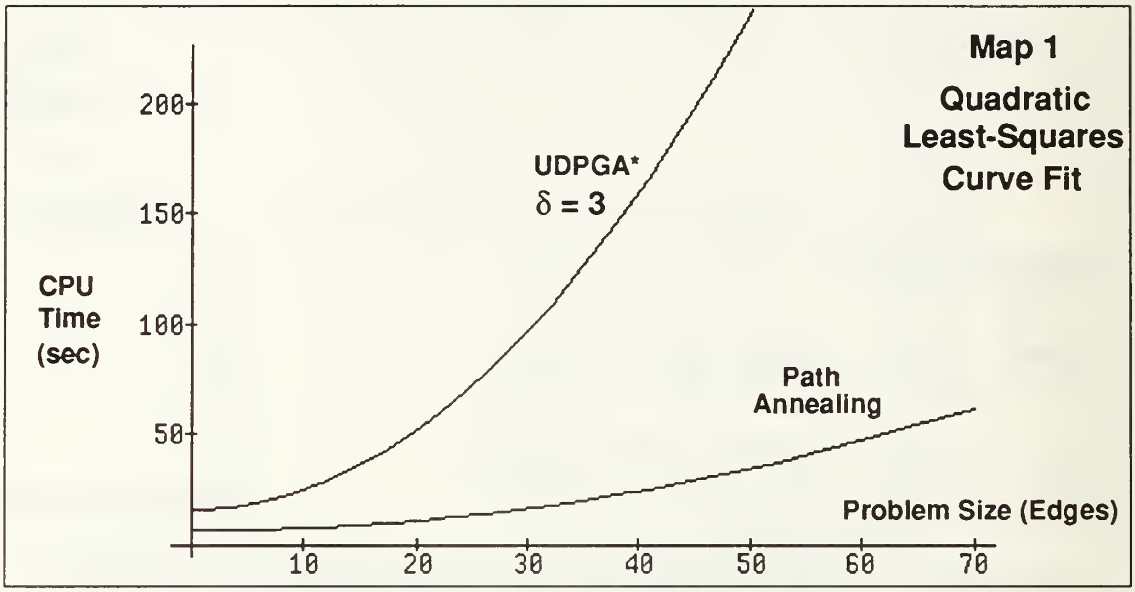


Figure 5.20 Map1 Quadratic Least-Squares Curve Fit: CPU Time vs. Problem Size

As a final comparison between path annealing and UDPGA* for Map1, we plot the solution cost ratio against time ratio for each problem instance in Figure 5.21. This plot provides a rough indication of the relative time-cost trade-offs between the algorithms. If we consider the point (1,1) as the origin, then the four quadrants define classes which categorize test results as good, bad, or in between. A test run which falls in quadrant I (upper-right) indicates bad path annealing performance since both time and cost ratios exceed one. On the other hand, a point which occurs in quadrant III (lower-left) indicates good path annealing performance. Keep in mind that this performance measure is always relative to UDPGA* which does not guarantee optimal solutions either. However, the fact that a large number of test points reside on the time ratio axis (i.e. path annealing cost / UDPGA* cost = 1.0) is a reasonably good indication that many true optimal paths have been discovered.

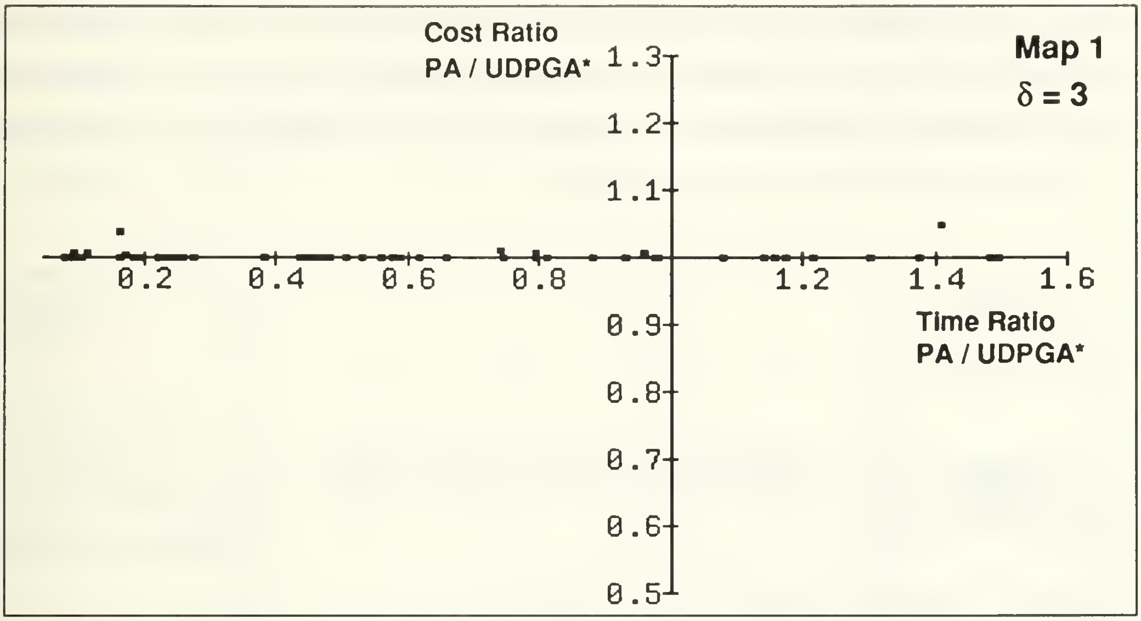


Figure 5.21 Map1 Cost Ratio vs. Time Ratio (Path Annealing/UDPGA*)

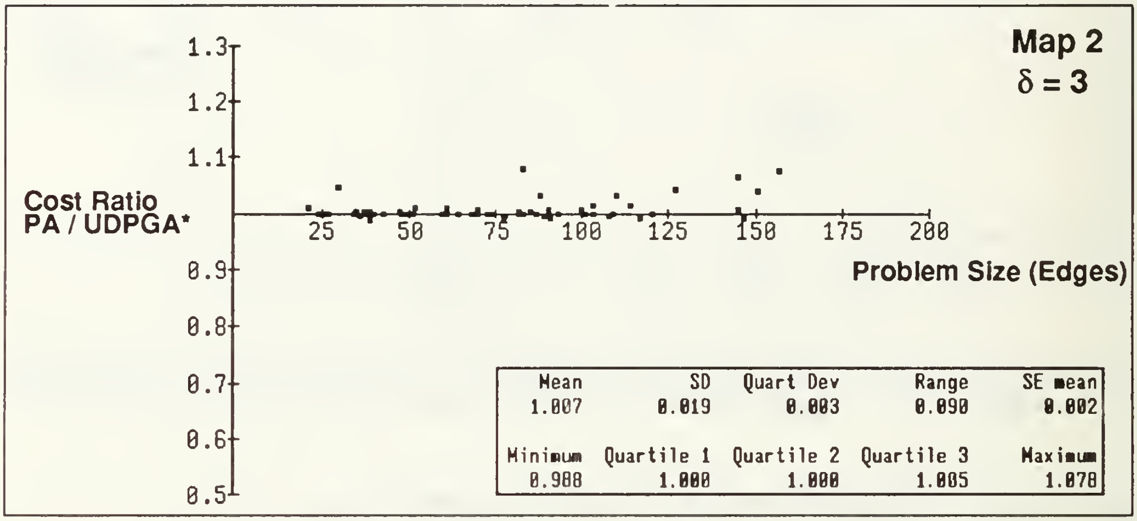
b. Testing of Moderately Complex Problem Instances

For the remaining tests we use the standard set of start/goal points defined earlier in this chapter. Recall that this set generates 66 distinct problem instances. The series of plots introduced in the last section will be the common basis for discussion of results. In this section we examine the results on Map2 which has over twice as many edges and regions as Map1. This map will compare performance of the algorithms at a larger, yet intermediate level of complexity.

In these tests, path annealing solutions are compared to UDPGA* solutions with δ=3 and δ=6. We used the following annealing schedule:

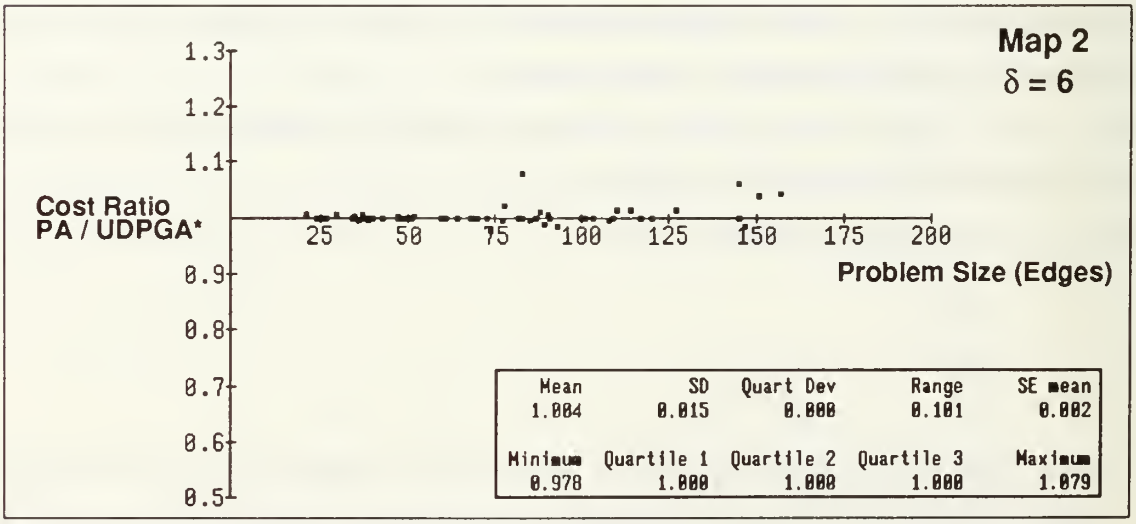
- T_0 = 1.5 × (cost of initial A* solution)
- T_f = 1.0
- R = .90
- L = 15
- L_s = 11

Figures 5.22a and 5.22b display cost ratio vs. problem size for comparison of path annealing against UDPGA* at δ=3 and δ=6 respectively. Although very slight, there is a difference between plots in the general height of the points above the *x*-axis (where cost ratio = 1). This is reflected by the small difference in mean cost ratio between the plots as observed in the inset statistical summaries. We should expect this since UDPGA* resolution is relaxed. Note also that the quartiles (in statistical summary insets of Figures 5.22a and 5.22b) are very small for both point sets, because the majority of the points rest very close to or on the *x*-axis. We attribute tiny deviations to rounding errors accumulated during total path cost computation more so than physical deviation from an optimal path.



Mean	SD	Quart Dev	Range	SE mean
1.007	0.019	0.003	0.090	0.002
Minimum	**Quartile 1**	**Quartile 2**	**Quartile 3**	**Maximum**
0.988	1.000	1.000	1.005	1.078

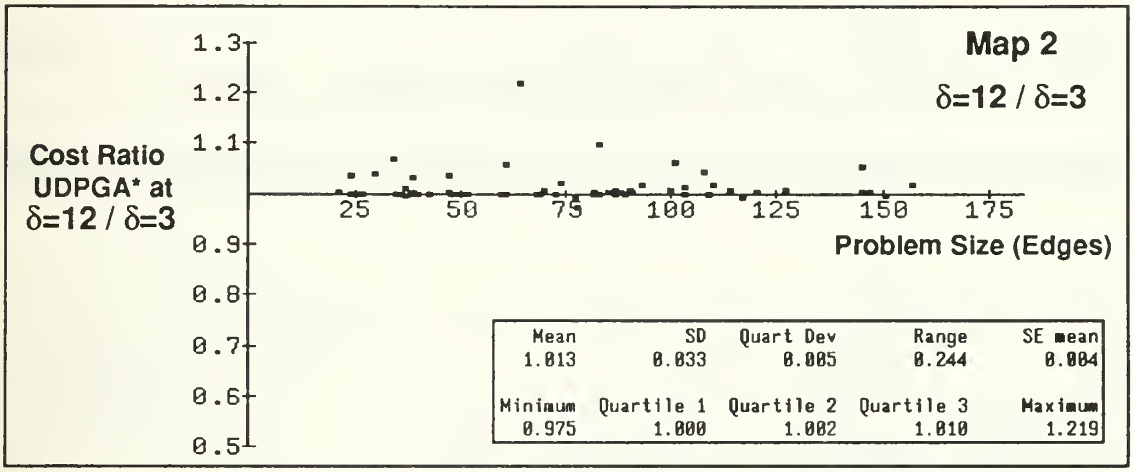
Figure 5.22a Map2 Cost Ratio (Path Annealing /UDPGA*) vs. Problem Size (Edges)



Mean	SD	Quart Dev	Range	SE mean
1.004	0.015	0.000	0.101	0.002
Minimum	**Quartile 1**	**Quartile 2**	**Quartile 3**	**Maximum**
0.978	1.000	1.000	1.000	1.079

Figure 5.22b Map2 Cost Ratio (Path Annealing /UDPGA*) vs. Problem Size (Edges)

The difference in means between δ=3 and δ=6 is so small that the Figures 5.22a and 5.22b do not adequately show how the value of δ can impact on the solution quality of UDPGA*. To see this effect, we executed UDPGA* on the same 66 problem instances in Map2 for δ=12. Figure 5.23 is a plot of the cost ratios for solution cost of UDPGA* at δ=12 over cost at δ=3. This figure shows that solution quality degrades significantly as δ increases.



Mean	SD	Quart Dev	Range	SE mean
1.013	0.033	0.005	0.244	0.004

Minimum	Quartile 1	Quartile 2	Quartile 3	Maximum
0.975	1.000	1.002	1.010	1.219

Figure 5.23 Map2 UDPGA* Cost Ratio (δ=12 / δ=3) vs. Problem Size (in Edges)

The next set of plots (Figures 5.24 through 5.27) show the effects of varying δ (resolution) on CPU time. Recall that δ defines the upper bound on the distance between edge-points. As we expect, the time required for UDPGA* at δ=6 is less than that required at δ=3. Comparison of the quadratic curve fits (Figure 5.28) shows the effect of increasing UDPGA* resolution. The dotted lines indicate that doubling resolution (i.e. halving the value of δ) results in a four-fold increase in the execution time requirement.

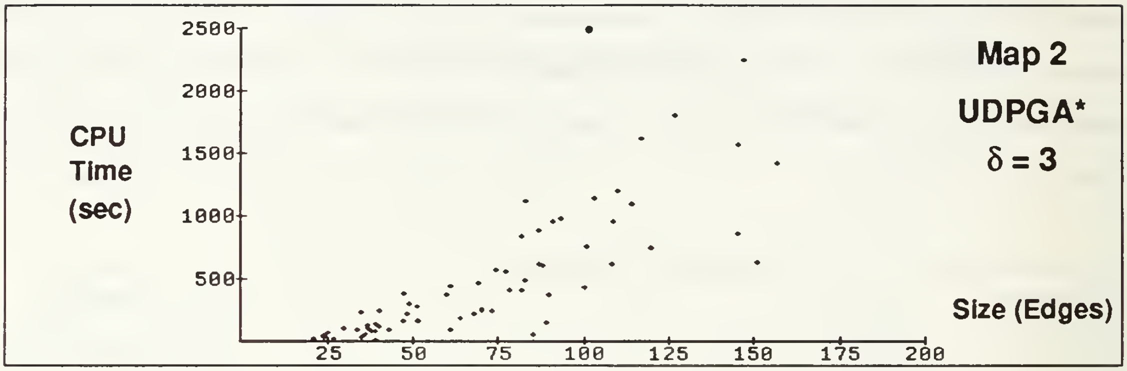


Figure 5.24 Map2 UDPGA* at δ=3: CPU Time Required vs. Problem Size (Edges)

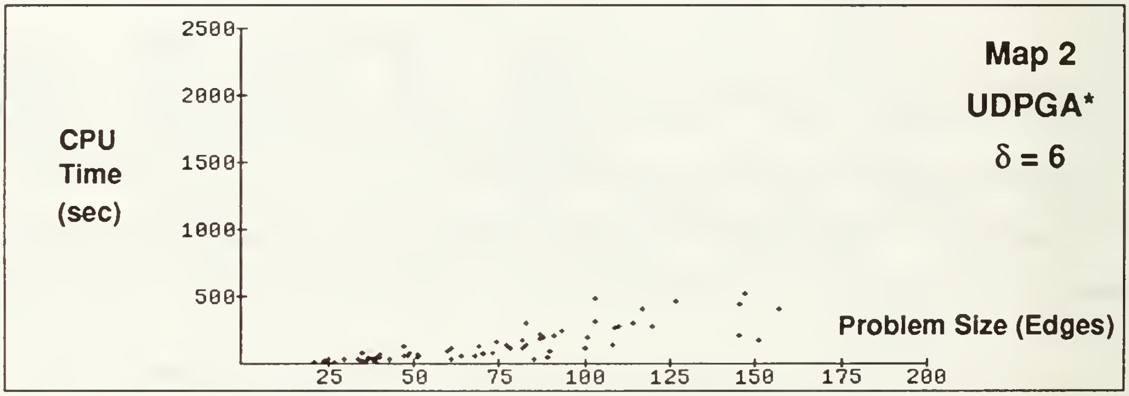


Figure 5.25 Map2 UDPGA* at δ=6: CPU Time Required vs. Problem Size (Edges)

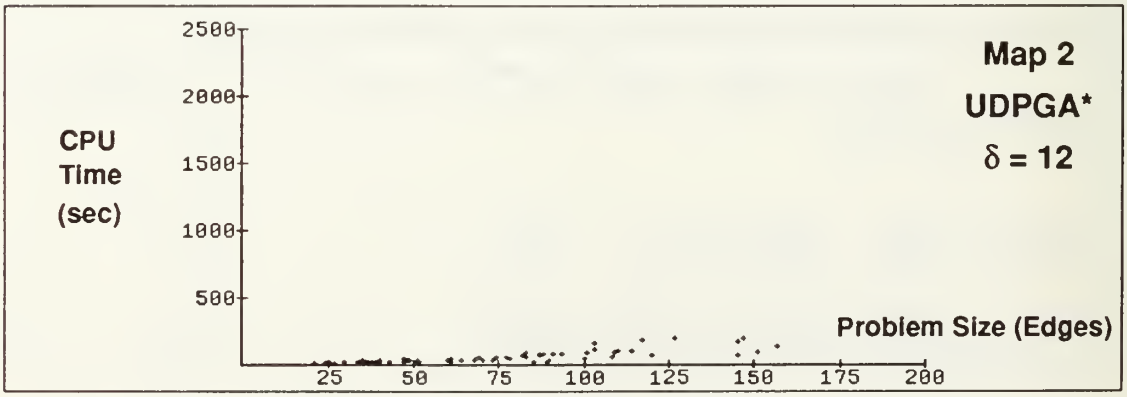


Figure 5.26 Map2 UDPGA* at δ=12: CPU Time Required vs. Problem Size (Edges)

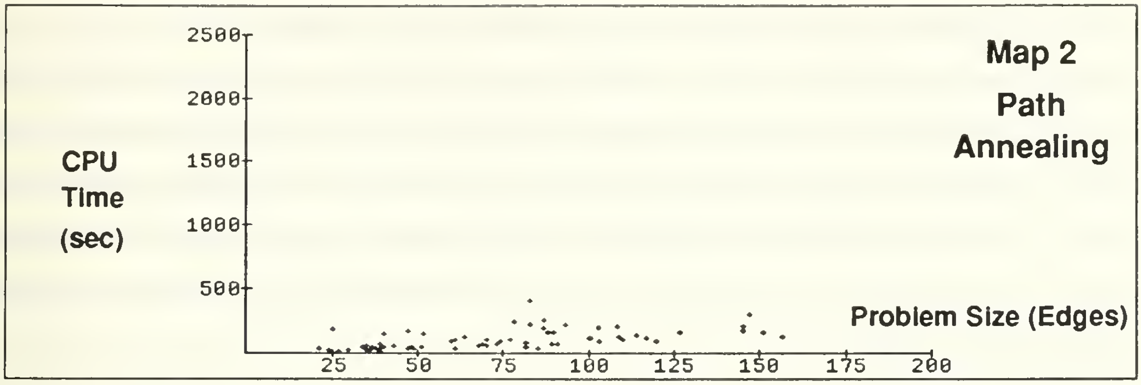


Figure 5.27 Map2 Path Annealing CPU Time Required vs. Problem Size (Edges)

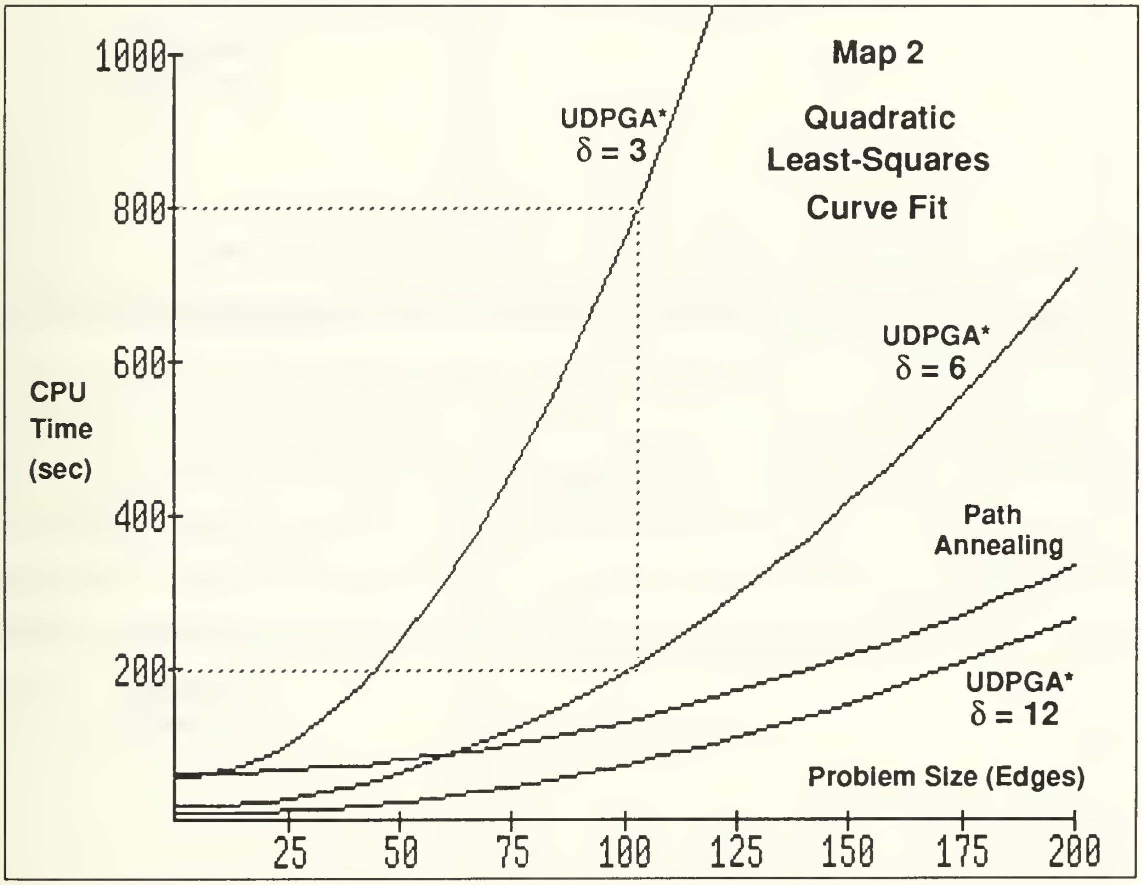


Figure 5.28 Map2 Quadratic Least-Squares Curve Fit for CPU Time vs. Prob Size

Finally, consider Figures 5.29 and 5.30 which are the cost-ratio vs. time-ratio plots when δ=3 and δ=6 respectively. For δ=3 more points reside further to the left, which is consistent with the overall increased time requirement for UDPGA*. However, the cost ratios change very little between δ=3 and δ=6. For path annealing, this cost-ratio stability between UDPGA* resolution changes can be attributed to a high rate of success by initial crude A* search in locating the optimal-path-containing window sequence or one which is close to it. In the following subsections, we shall examine the results of tests for which the initial crude A* search is not quite as successful. This will expose a weakness of path annealing – its dependence on good starting solutions.

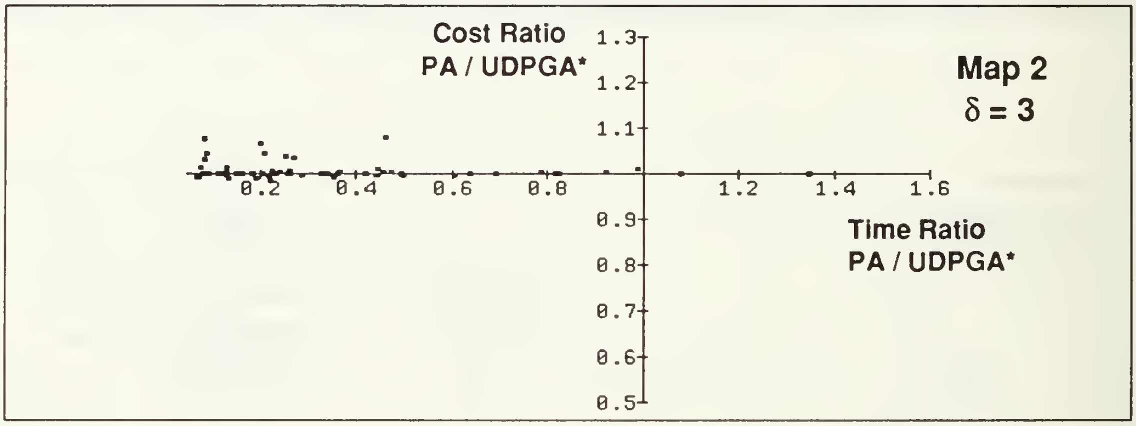


Figure 5.29 Map2 Cost Ratio vs. Time Ratio (Path Annealing/UDPGA*) δ=3

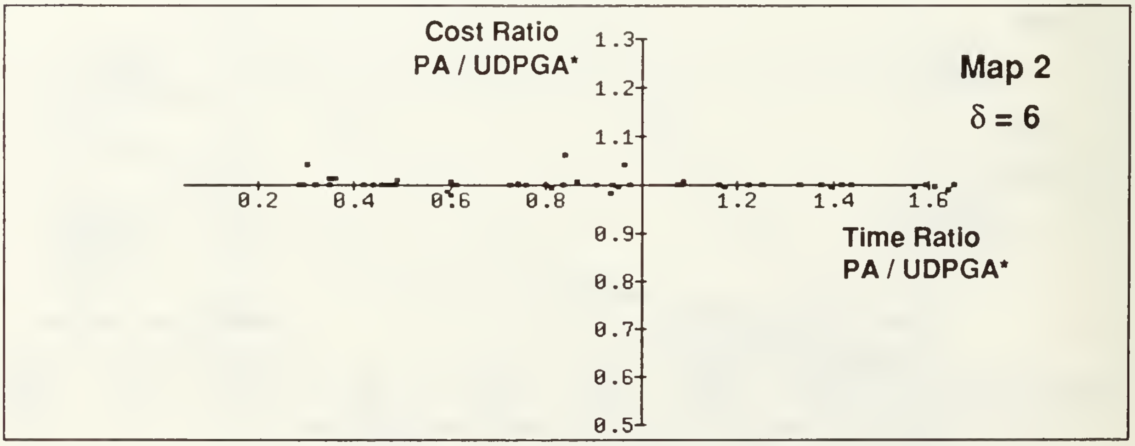


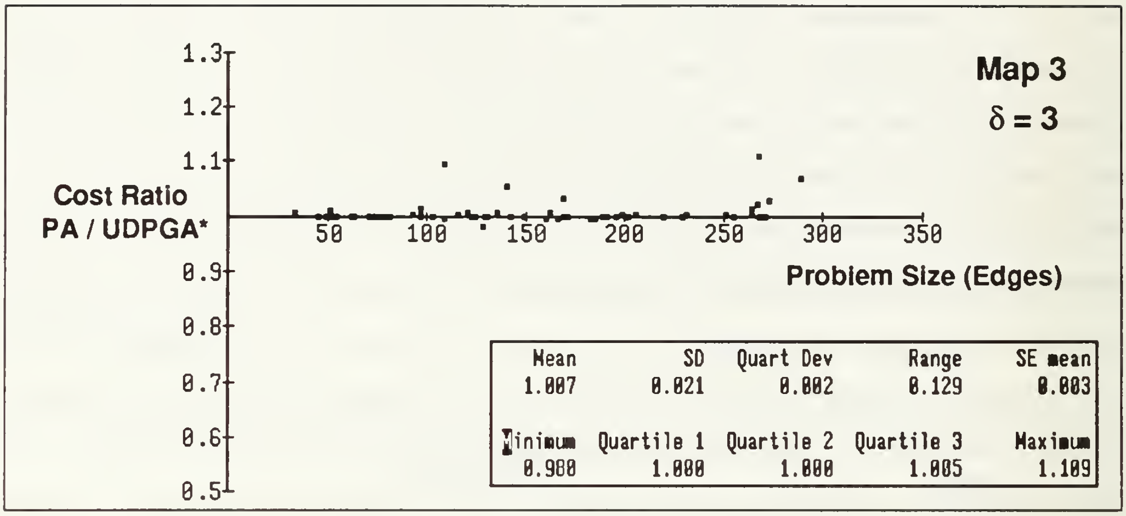
Figure 5.30 Map2 Cost Ratio vs. Time Ratio (Path Annealing/UDPGA*) δ=6

c. *Testing Real Terrain*

In this section we present the results of tests conducted on Map3 and Map4 which model actual terrain. These maps have about three times the number of edges and regions as Map1. It is important to realize that the cost functions in these maps are relatively smooth because they approximate the cost of movement as determined from real elevation and vegetation data. In addition to the large reservoir, there are also cost coefficient differences between the maps which account for seasonal changes in trafficability. However, we have assumed that regional topology remains unchanged from wet to dry season. We have not superimposed tactical data that might induce drastic cost differentials across boundaries, although the elevation data does tend to cause such in a few locations. In these maps we set δ=3 to achieve relatively high quality UDPGA* solutions. The path-annealing schedule is adjusted somewhat to account for longer window sequences which will result from the complexity in these maps. We have extended the annealing chain length and cutoff to 20 and 15 respectively:

- T_0 = 1.5 × (cost of initial A* solution)
- T_f = 1.0
- R = .90
- L = 20
- L_s = 15

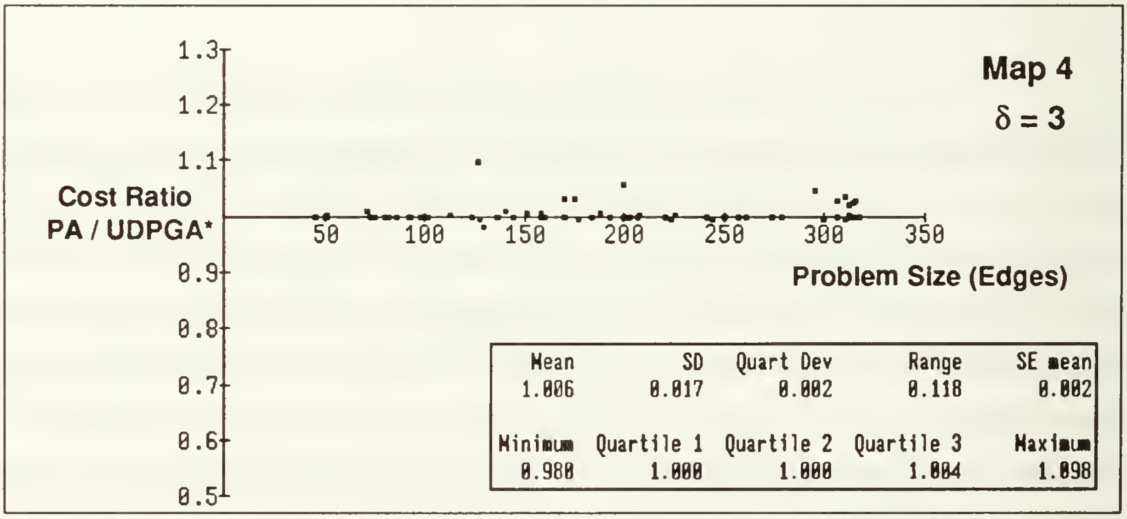
Figures 5.27 and 5.28 are plots of cost ratio as a function of problem size for runs on Map3 and Map4 respectively. In both maps path annealing does reasonably well for such complex maps. It is interesting to note that there does not appear to be any significant decrease in solution quality for path annealing as the size of problem instances increases. Furthermore, if we assume for a moment that δ=3 is a fine enough resolution to ensure that UDPGA* does not overlook an optimal path, then all but a few of the path annealing solutions fall within a 5% margin of error. We attribute this behavior to the success of the crude initial A* search which often finds a window sequence in the vicinity of the optimal path. We emphasize that the success of path annealing is often dependent on the power of this initial crude systematic search.



Mean	SD	Quart Dev	Range	SE mean
1.007	0.021	0.002	0.129	0.003

Minimum	Quartile 1	Quartile 2	Quartile 3	Maximum
0.980	1.000	1.000	1.005	1.109

Figure 5.31 Map3 Cost Ratio (Path Annealing/UDPGA*) vs. Problem Size (Edges)



Mean	SD	Quart Dev	Range	SE mean
1.006	0.017	0.002	0.118	0.002

Minimum	Quartile 1	Quartile 2	Quartile 3	Maximum
0.980	1.000	1.000	1.004	1.098

Figure 5.32 Map4 Cost Ratio (Path Annealing/UDPGA*) vs. Problem Size (Edges)

Algorithm timing comparisons on Map3 and Map4 yield similar results. Therefore, in Figures 5.33 and 5.34 we present only the plots for Map4. The important difference from what we have already viewed is the fact that these time plots provide empirical evidence of average time complexities for very large problem instances (300+ edges) without extrapolation. Another quadratic curve fit (Figure 5.35) shows that path annealing timing requirements rise very slowly and are almost linear as a function of the problem size. The reason for this behavior is that the annealing algorithm is a random sampling mechanism whose annealing schedule predetermines to a large extent its running time. The reason why these timing results actually do vary with problem size is that the general length of window sequences determines how quickly the locally optimal path can be found. Longer WS's require somewhat more time. Larger problem instances will generally have to optimize longer WS's.

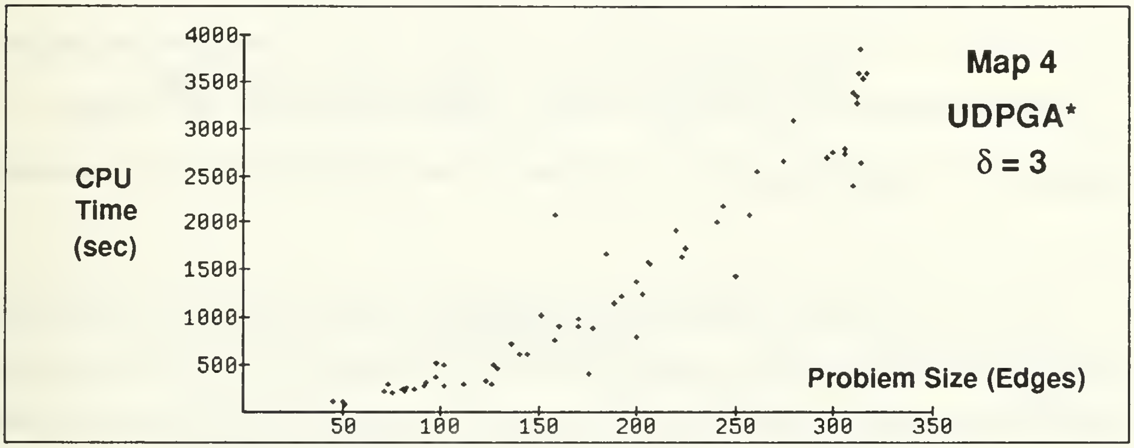


Figure 5.33 Map4 UDPGA* CPU Time Required vs. Problem Size (in Edges)

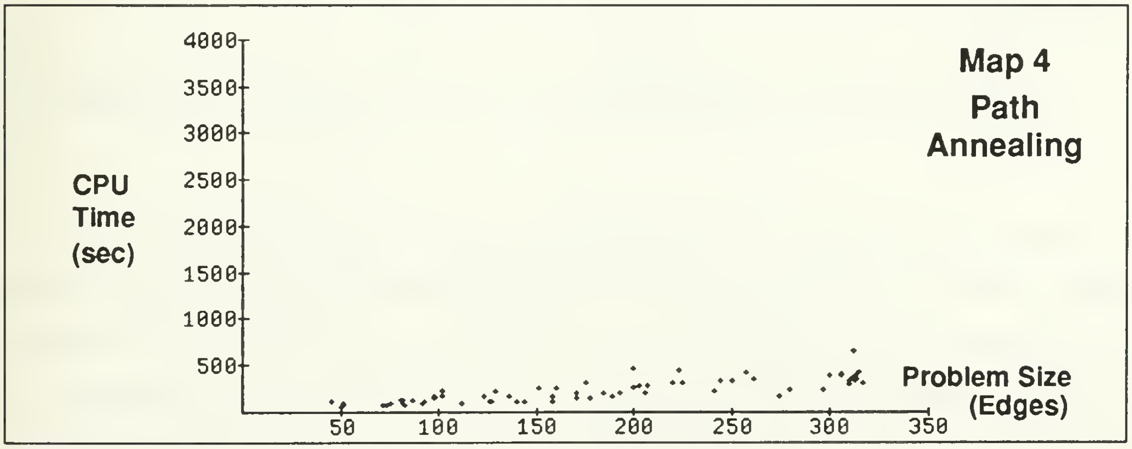


Figure 5.34 Map4 Path Annealing CPU Time Required vs. Problem Size (in Edges)

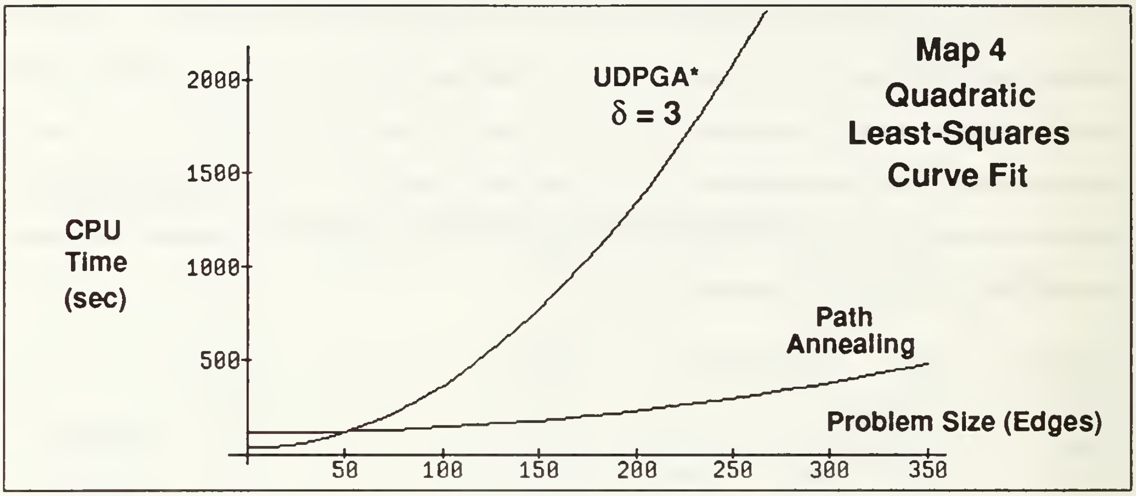


Figure 5.35 Map4 Quadratic Least-Squares Curve Fit: CPU Time vs. Problem Size

The cost-ratio vs. time-ratio plots in Figures 5.36 and 5.37 (for tests on Map3 and Map4 respectively) reflect the reduced CPU time requirement of path annealing. The few points which remain to the right of the cost-ratio axis (time-ratio = 1) correspond to the smallest problem instances. Also, note that almost no path annealing runs significantly improve the optimal cost. As demonstrated in the next section, this is usually a good indication that the UDPGA* solutions are optimal. The reason is that when UDPGA* overlooks true optimal solutions, some path annealing runs usually discover a few better solutions (even optimal) than those returned by UDPGA*. The result will be a few points which dip below the time-ratio axis (cost-ratio = 1).

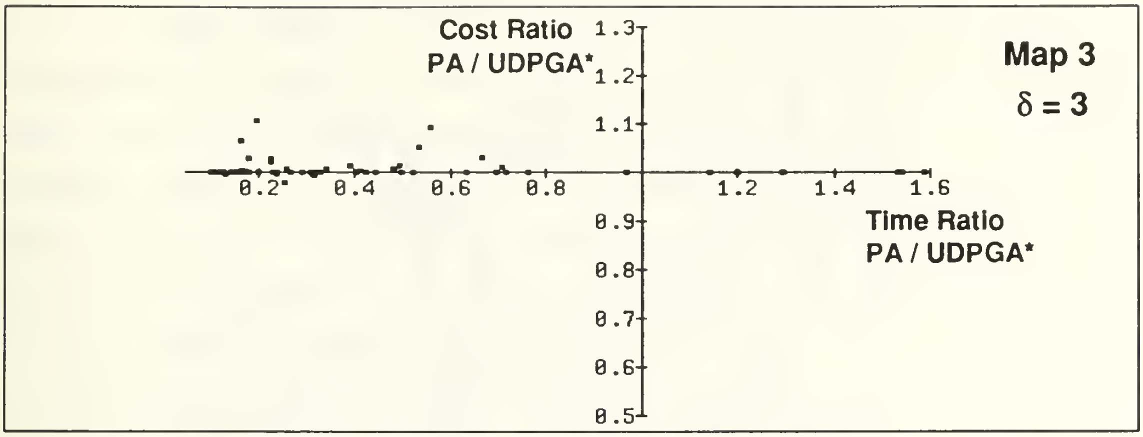


Figure 5.36 Map3 Cost Ratio vs. Time Ratio (Path Annealing/UDPGA*)

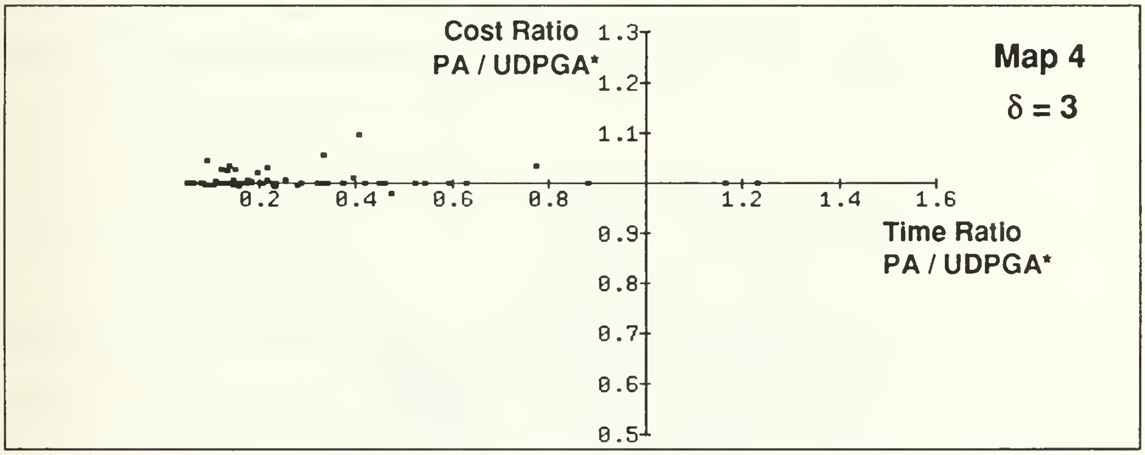


Figure 5.37 Map4 Cost Ratio vs. Time Ratio (Path Annealing/UDPGA*)

As a final comparison, we present the physical locations of the solution paths returned by the algorithms. Figure 5.38 and 5.39 illustrate the test maps with all solution paths drawn between respective start/goal points. We provide these pictures merely as a visual demonstration to show that the location and shape of annealing solution paths compare favorably with those of UDPGA*. The majority of the solution paths are identical. A few solutions have slight physical differences which have little impact on corresponding total path-cost.

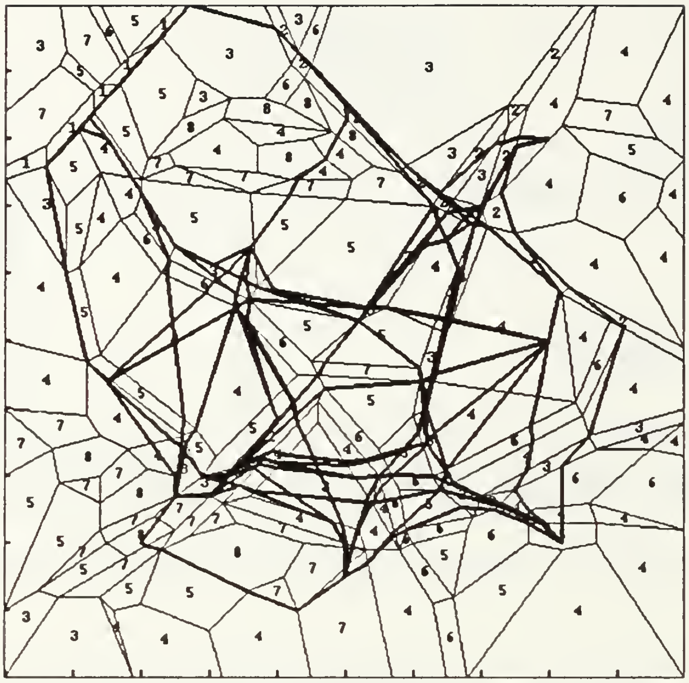


Figure 5.38   Map4 Solution Paths Found by UDPGA*

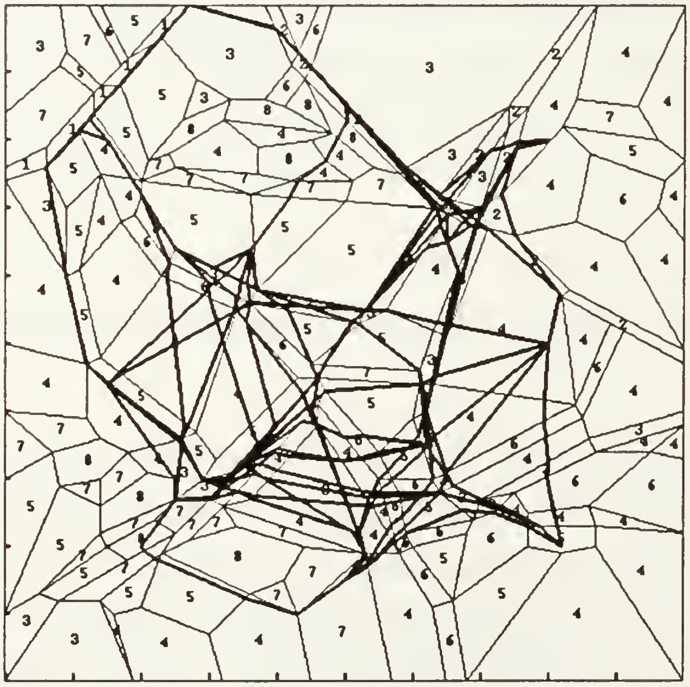


Figure 5.39   Map4 Solution Paths Found by Path Annealing

d. Testing Maps with Highly-Variable Cost Coefficients

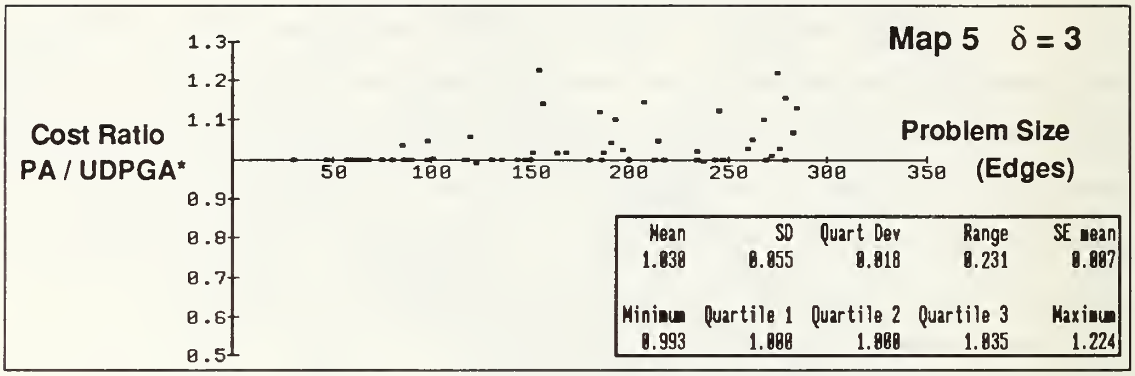
The next series of tests is designed to determine how well path annealing performs on maps with generally hilly cost functions. To some extent, these tests indicate the kind of performance we might expect if we were to model extremely complicated tactical situations with a variety of overlapping and interacting cost coefficients (as discussed in Chapter I). Recall that for Map5, Map6, and Map7 we have randomly assigned region cost coefficients drawn respectively from the three sets: {1,2,3,4,5,6,7,8,9,10}, {1,3,5,7,9}, {1,4,7}. Therefore, we describe the general character of each map as follows:

- Map5 - uniformly choppy; many shallow local minima
- Map6 - choppy but irregular; deeper local minima
- Map7 - high cost peaks and deep valleys; several very deep local minima

For this set of tests we have again chosen $\delta=3$ for the uniform-discrete-point global A* (UDPGA*) search algorithm. We modify the path-annealing schedule to conduct a more thorough local search at each temperature by raising the reduction factor *R* slightly. We also start at a somewhat higher temperature to delay the onset of freezing, and hopefully, to enable the annealing algorithm to climb out of deeper local minima. The new schedule is as follows:

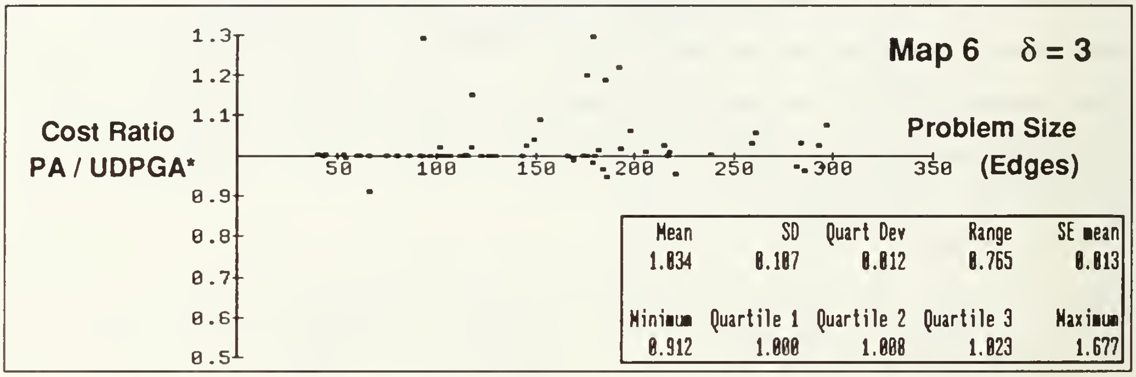
- T_0 = 2.0 × (cost of initial A* solution)
- T_f = 1.0
- R = .92
- L = 20
- L_s = 15

We expect that larger numbers of deeper local minima will severely tax the ability of path annealing algorithm to find good solutions. This conjecture is partially confirmed by the cost-ratio vs. problem size plots in Figures 5.40 through 5.42. While path-annealing solution quality on Map5 is somewhat worse than it was for smoother maps, we can see a noticeable trend going to Map6 and Map7 as more points wander higher above the x-axis. On the other hand, several points below the x-axis indicate that UDPGA* also has difficulty finding true optimal solutions. The fact that more points lie above than below the x-axis is reasonable since we consider UDPGA* our gold standard, and because it is a systematic search which looks everywhere a with finite resolution.



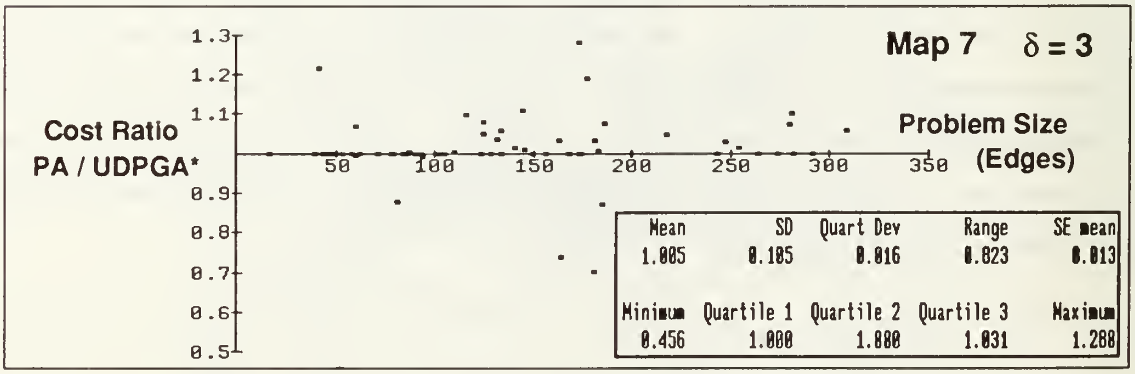
Mean	SD	Quart Dev	Range	SE mean
1.030	0.055	0.018	0.231	0.007
Minimum	**Quartile 1**	**Quartile 2**	**Quartile 3**	**Maximum**
0.993	1.000	1.000	1.035	1.224

Figure 5.40 Map5 Cost Ratio (Path Annealing/UDPGA*) vs. Problem Size (Edges)



Mean	SD	Quart Dev	Range	SE mean
1.034	0.107	0.012	0.765	0.013
Minimum	**Quartile 1**	**Quartile 2**	**Quartile 3**	**Maximum**
0.912	1.000	1.008	1.023	1.677

Figure 5.41 Map6 Cost Ratio (Path Annealing/UDPGA*) vs. Problem Size (Edges)



Mean	SD	Quart Dev	Range	SE mean
1.005	0.105	0.016	0.823	0.013
Minimum	**Quartile 1**	**Quartile 2**	**Quartile 3**	**Maximum**
0.456	1.000	1.000	1.031	1.280

Figure 5.42 Map7 Cost Ratio (Path Annealing/UDPGA*) vs. Problem Size (Edges)

Although the solution cost from path annealing is significantly more than UDPGA*, we emphasize the following two points. First, as mentioned earlier, simulated annealing in general does not perform well on search spaces whose cost functions have many deep local minima. The reason is that such deep valleys more easily trap the annealing algorithm, preventing it from sampling other areas of the space. Second, in these tests we anneal from a single initial solution. If this solution lies within a deep local minimum, then path annealing might not be able to escape. Raising the starting temperature is one way to prevent such behavior. However, we believe that this is not the most effective use of time in a simulated-annealing approach to route-finding. Limited testing indicates that better performance can be achieved for minimal time penalties by using the tunneling approach discussed in Chapter IV. Instead of annealing from a single deterministic initial solution, it is more advantageous to begin from several which are physically scattered around the map. Our concept of crude (boundary-edge midpoint) A* searches can serially find several such paths relatively quickly. However, more efficient methods for doing so in a single sweep should be investigated.

An analysis of the CPU time vs. problem size once again illustrates a significant time advantage for path annealing, even with the annealing schedule modified to search more thoroughly (see Figures 5.43 through 5.51). With regard to the UDPGA* algorithm, we can identify an interesting trend in the point spread going from smooth cost coefficients to very rough. Consider the UDPGA* timing results illustrated in Figures 5.33, 5.43, 5.46, and 5.49, corresponding to Map4 through Map7. We can see more point scatter as the cost coefficients become more varied. There is also a general downward movement of all points, indicating that the average time complexity of UDPGA* improves with rougher cost functions. We can attribute each behavior to an increase in the number of deep local minima, and an indication of bounding constraint effectiveness. Recall that both algorithms conduct crude initial A* search for an initial solution. The UDPGA* uses this solution only to compute the bounding ellipse which can sometimes prune a large amount of search space. If the crude initial A* search locates a deep local minimum then the bounding ellipse will be tight. The spreading of points shows that some bounding ellipses are small enough to dramatically reduce the work of UDPGA*. Path annealing can also be improved with this bound. However, these improvements are reflected more in path-annealing solution optimality and not so much in CPU time requirements, which are more sensitive to the annealing schedule.

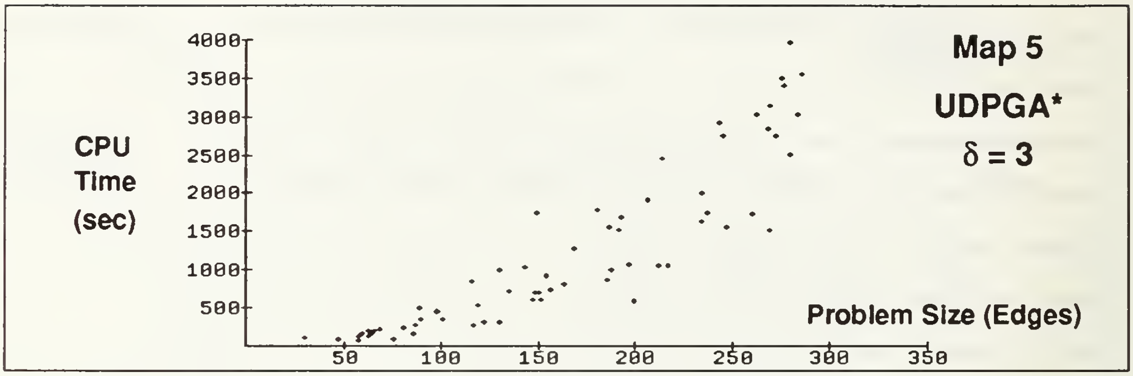


Figure 5.43 Map5 UDPGA* CPU Time Required vs. Problem Size (in Edges)

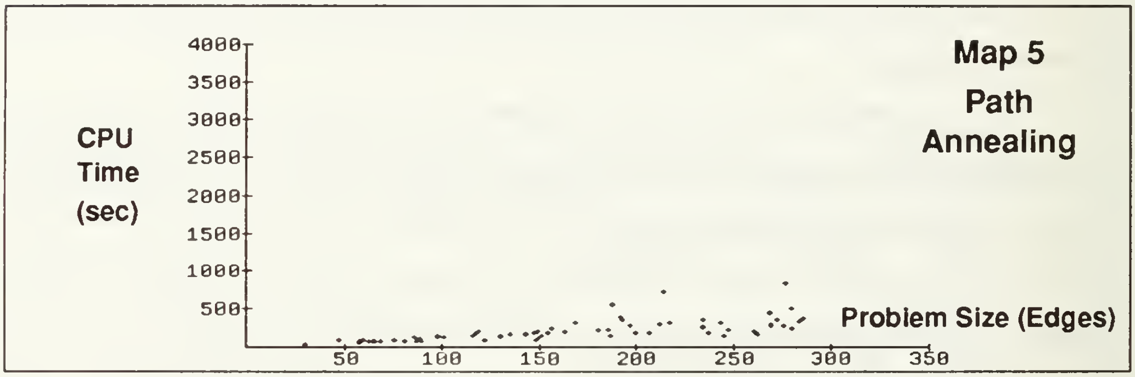


Figure 5.44 Map5 Path Annealing CPU Time Required vs. Problem Size (in Edges)

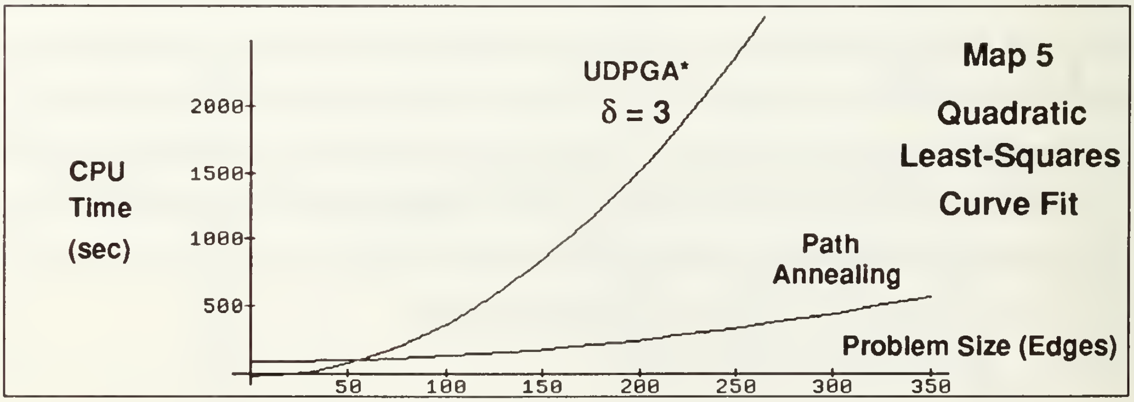


Figure 5.45 Map5 Quadratic Least-Squares Curve Fit: CPU Time vs. Problem Size

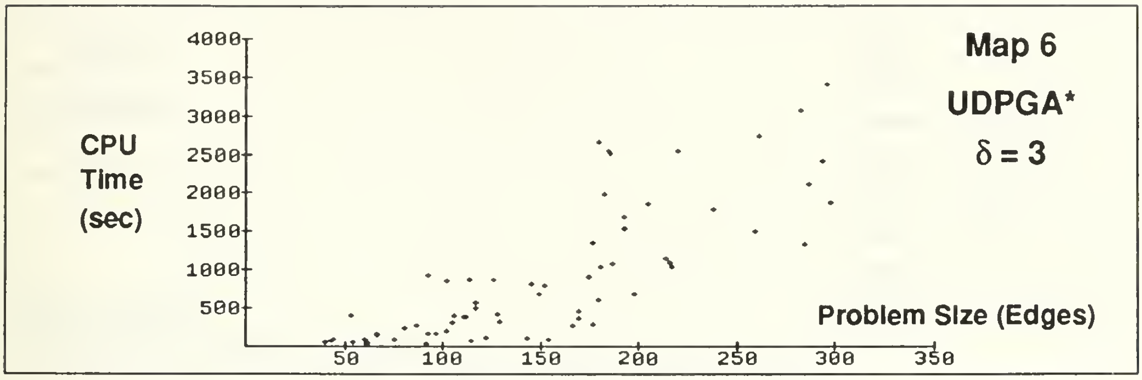


Figure 5.46 Map6 UDPGA* CPU Time Required vs. Problem Size (in Edges)

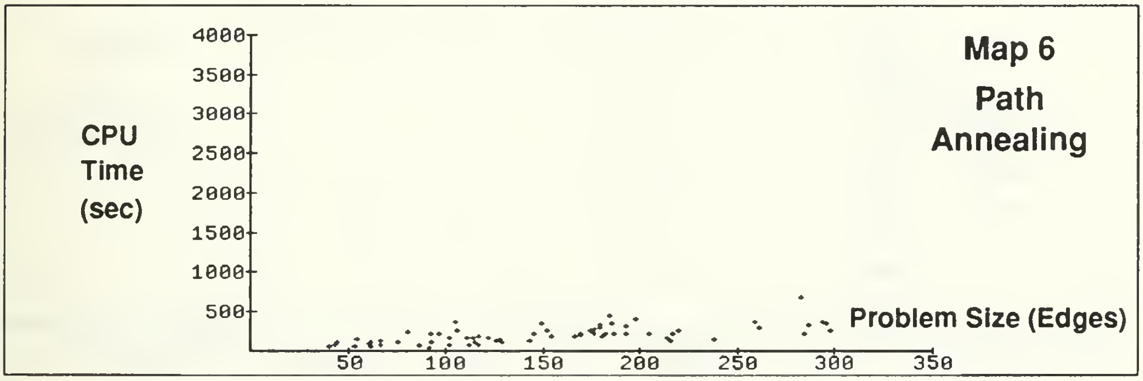


Figure 5.47 Map6 Path Annealing CPU Time Required vs. Problem Size (in Edges)

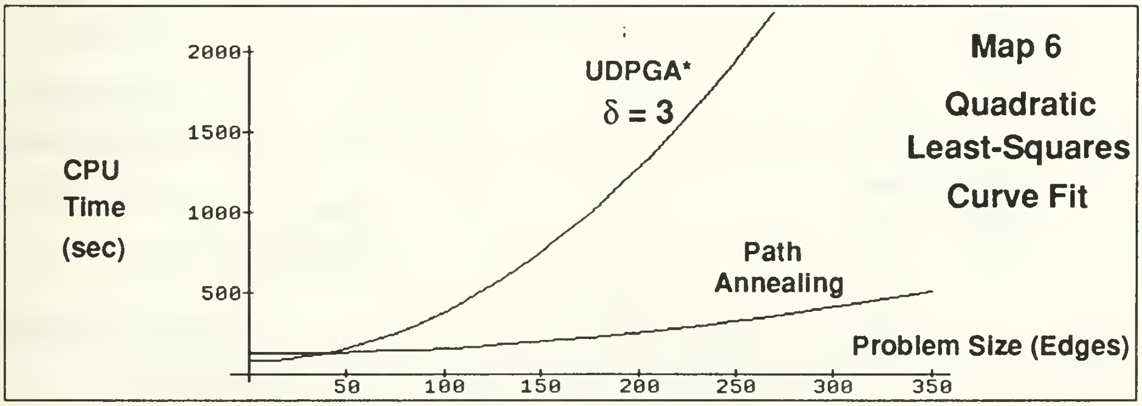


Figure 5.48 Map6 Quadratic Least-Squares Curve Fit: CPU Time vs. Problem Size

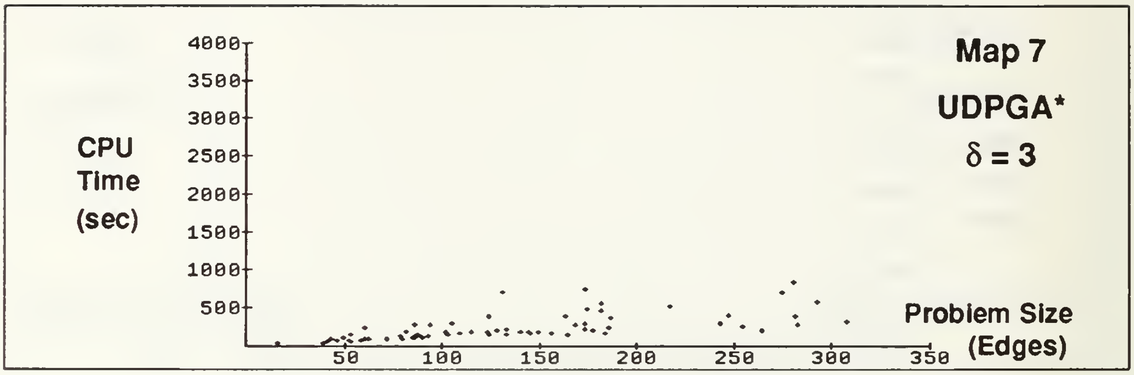


Figure 5.49 Map7 UDPGA* CPU Time Required vs. Problem Size (in Edges)

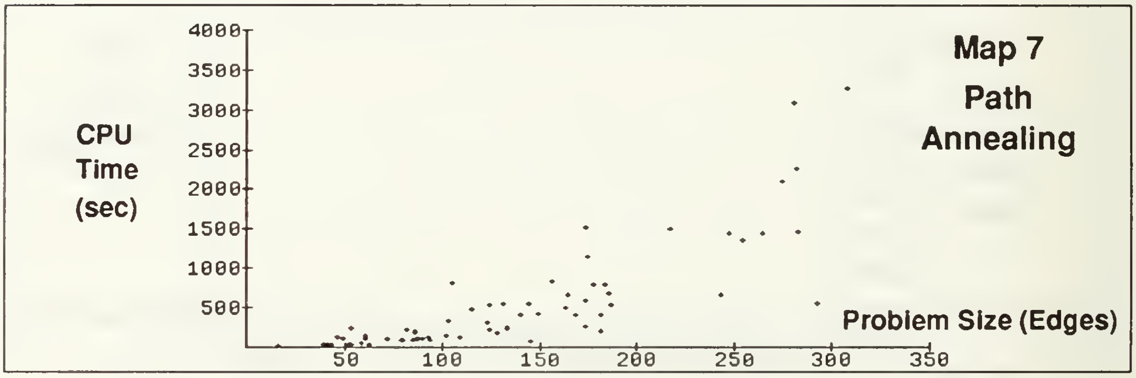


Figure 5.50 Map7 Path Annealing CPU Time Required vs. Problem Size (in Edges)

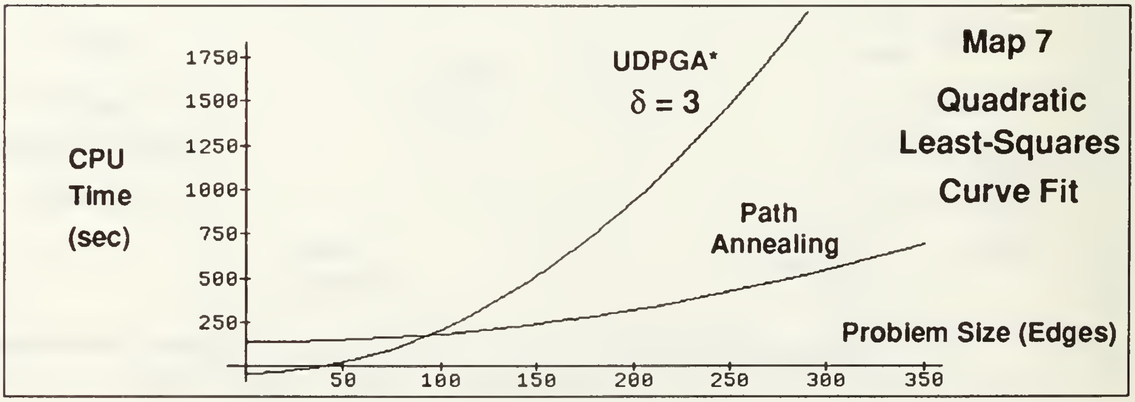


Figure 5.51 Map7 Quadratic Least-Squares Curve Fit: CPU Time vs. Problem Size

The cost-ratio vs. time-ratio plots in Figures 5.52 through 5.54 are consistent with the foregoing discussions of optimality and time. We see a general trend in favor of the UDPGA* algorithm as the number of deep local minima in the cost function increases (from Map5 to Map7). However, note that in Figure 5.54 for Map7 there are several very low points. As implied earlier, these indicate that even UDPGA* is having trouble finding the true globally optimal path. To fix this problem, one must decrease the value of δ to increase resolution. However, recall that the CPU time requirement reciprocates as the square. Yet, in less time we may be able to run a path-annealing schedule set for more thorough local search, and we may also anneal from multiple starting solutions.

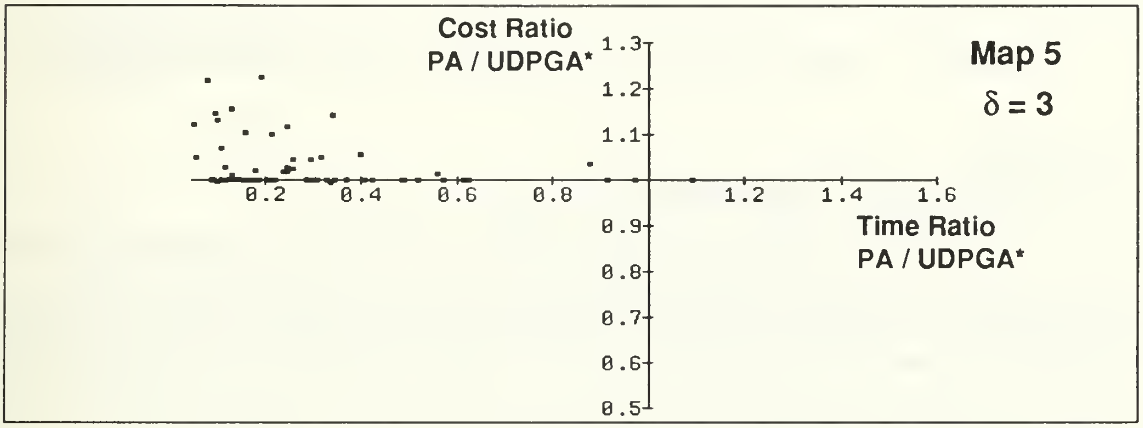


Figure 5.52 Map5 Cost Ratio vs. Time Ratio (Path Annealing/UDPGA*)

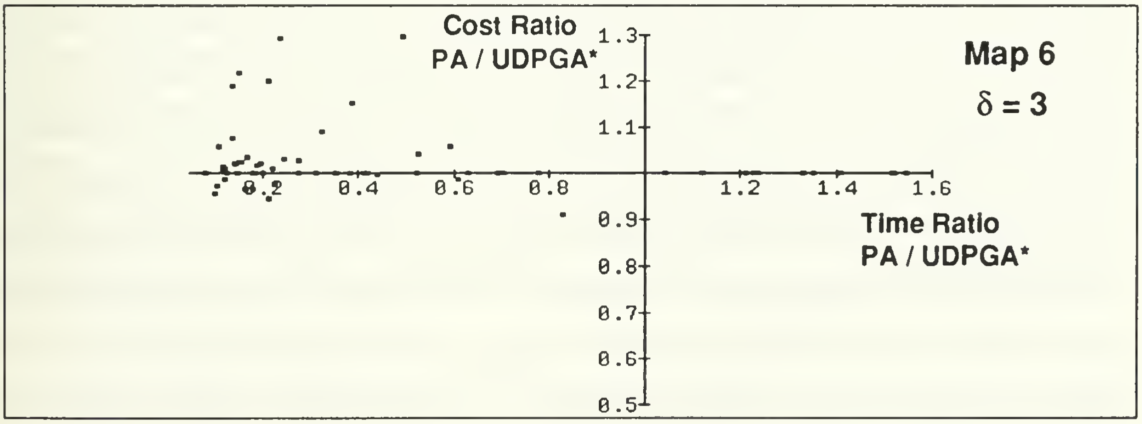


Figure 5.53 Map6 Cost Ratio vs. Time Ratio (Path Annealing/UDPGA*)

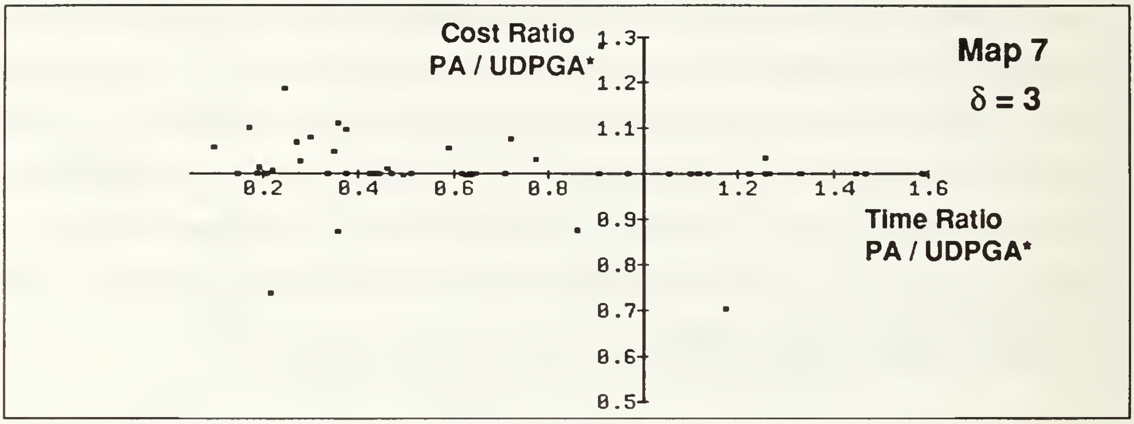


Figure 5.54 Map7 Cost Ratio vs. Time Ratio (Path Annealing/UDPGA*)

e. Testing of a Difficult Problem Instance

As a final test of path annealing behavior, we have selected one particular problem instance for which path annealing appears to overlook the globally optimal path fairly often. This problem instance is on Map6 for start point (15,80) and goal point (80,50). The shortest midpoint path-cost (returned by the initial A* search) is 239.5 weighted units, while UDPGA* (at δ=3) indicates that the optimal solution cost is 186.9 weighted units. Repeated runs have discovered window sequences (WS) with locally optimal path-cost values throughout the range {186.9...239.5}.

We applied path annealing 400 times on this problem instance. The freezing temperature (T_f), reduction factor (R), annealing chain length (L), and cutoff (L_s) values were fixed at 1.0, 0.92, 20, and 15 respectively. We used eight different starting temperatures, and 50 runs were conducted for each starting temperature. These results are summarized in Figure 5.55. This plot indicates the percentage of runs (out of 50) that path annealing returns the optimal path (cost = 186.9) as a function of the starting temperature. The chart shows that for this particular problem instance, it is more advantageous to begin annealing from a lower starting temperature. The reason for this is that the initial A* search returns a midpoint path which represents a high speed corridor containing the global optimum. Higher temperatures allow the annealing process to easily wander out of this corridor. Recall that since Map6 is very hilly, many relatively deep cost-function valleys exist. Once out of the corridor corresponding to the initial solution, decreasing temperature may prevent annealing from returning to it. In such cases, if annealing did not discover the optimal solution while inside this corridor, then it will settle for a non-optimal solution within some other corridor.

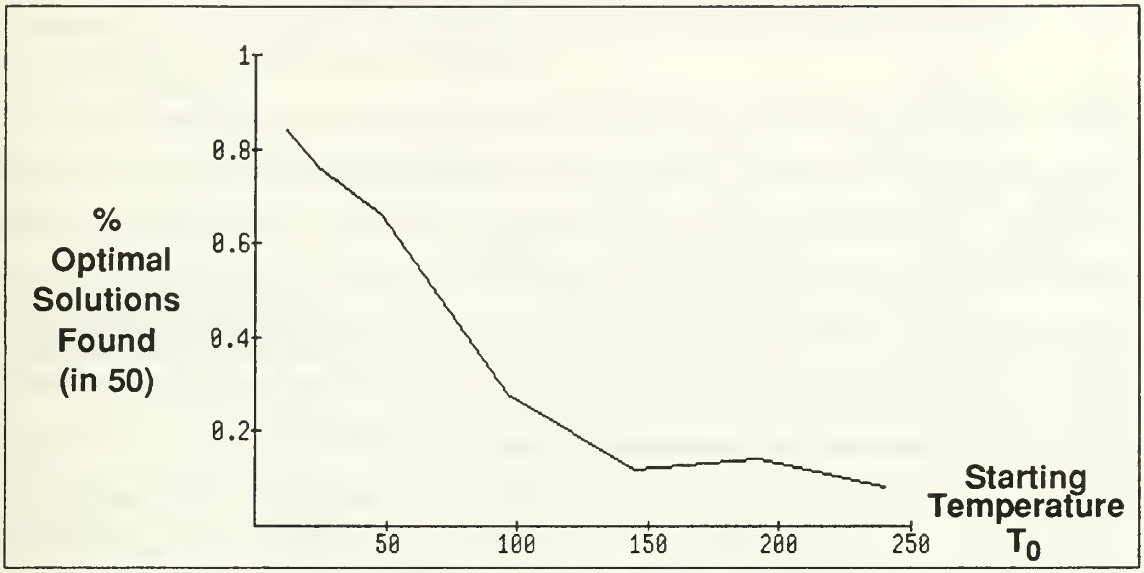


Figure 5.55 Percentage of Optimal Solutions (out of 50) vs. Starting Temperature T_0

This behavior lends support for the tunneling ideas proposed in Chapter IV, Section C.3.c. Recall that the basic concept of tunneling is to locate a set of initial window sequences representing different high speed corridors, and permit annealing to search each corridor more intensively by starting at lower temperature. This method redirects the time necessary to climb out of and move between corridors. Instead, this time can be devoted to more thorough searching of high speed corridors which have the most potential to contain optimal solutions. For this particular problem instance, Figure 5.56 provides an indication of the wide range of solutions on which path annealing can settle. This figure represents a list of all final solutions sorted by cost for over 400 runs of the path annealing algorithm on this problem instance. Note that even though the starting temperatures vary, path annealing found the globally optimal solution (cost = 186.9) in over 33% of these runs, and found better solutions than initial A* (cost = 239.5) in over 75% of the runs.

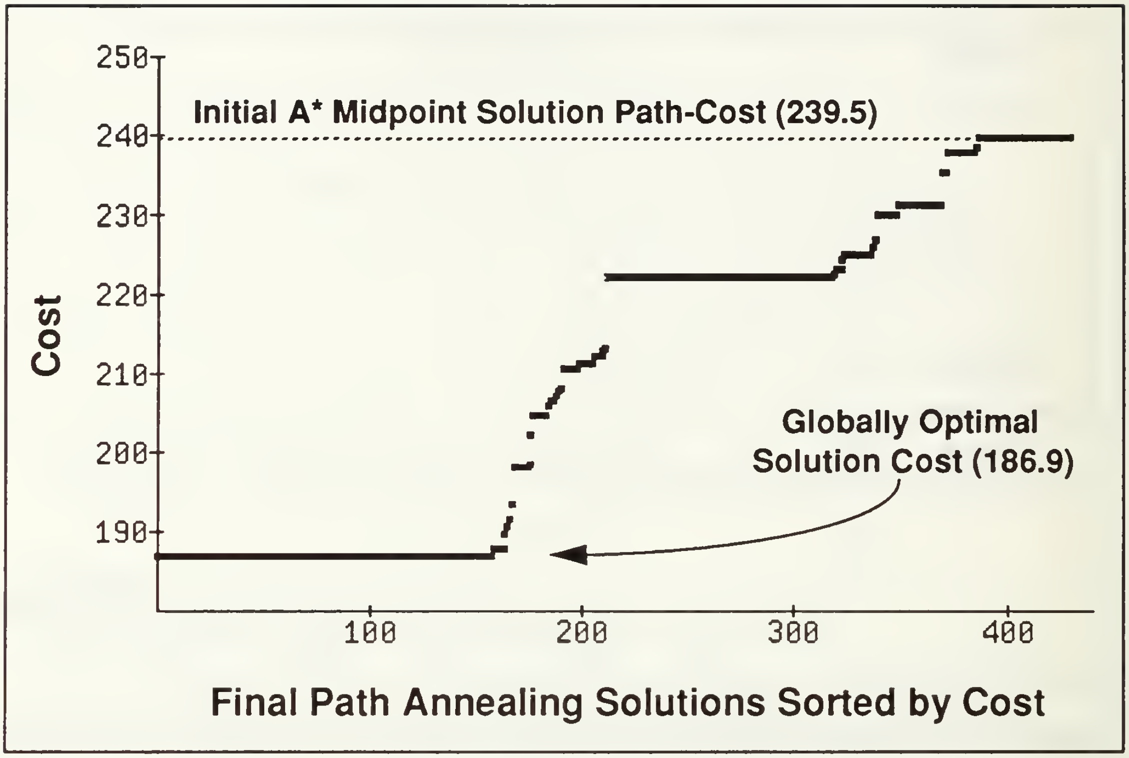


Figure 5.56 Final Path Annealing Solutions Sorted by Cost for Multiple Runs on a Particular Problem Instance (Start [15,80], Goal [80,50], Map6)

VI. CONCLUSIONS

A. SUMMARY OF SIGNIFICANT RESULTS

In this dissertation we have proposed a stochastic algorithm, *path annealing*, to solve the weighted-region problem (WRP), in particular, for large complex problem instances. Based primarily upon the combinatorial optimization technique known as *simulated annealing*, path annealing employs systematic and local search in coordination to obtain good initial solutions that can improve over time. Since the algorithm is a local search of complete solutions, there are more opportunities to establish and update bounds on the globally optimal solution. Furthermore, we have designed additional spatial constraints and heuristic enhancements to reduce search space and increase the performance of path annealing to an extent that it becomes competitive with other approaches to the WRP. We have implemented and tested a prototype of this algorithm on maps with over 300 boundary edges and 100 weighted-regions.

We believe that path annealing is a practical approach to weighted-region path planning in a military setting in which region cost coefficients may model many battlefield parameters. In our review of various path-planning implementations, we have seen evidence that large problems overwhelm agenda mechanisms used by systematic techniques such as A* and uniform cost search. Our approach represents an attempt to focus the strengths of both local search and systematic search to solve a relatively difficult problem in a reasonable amount of time.

Our investigation of current algorithm implementations reveals that grid-based wavefront propagation appears to be the only one capable of handling the complexity of our test maps. Recall that this algorithm's uniform cost search and resulting grid-biased solutions hamper both efficiency and solution accuracy. In order to provide a faster, more accurate algorithm to compete with path annealing, we designed and implemented uniform-discrete-point global A* (UDPGA*) search. It appears that this algorithm may also be significant, since it too is a new approach for solving the weighted-region problem (WRP). It represents a wavefront propagation approach which employs an adaptive grid that conforms to the geometry of the homogeneous-cost regions, and essentially forms a visibility graph. With the addition of the single-path-relaxation technique for finding the locally optimal path in a window sequence, and a few heuristic improvements, UDPGA* search often finds an optimal path with very good accuracy. While empirical execution times are significantly greater than path annealing, they would quite likely be much lower than grid-based wavefront propagation.

B. ALGORITHM STRENGTHS AND WEAKNESSES

1. Strengths

Test results indicate that for non-trivial problem instances path annealing has faster average execution times than uniform-discrete-point global A* (UDPGA*) search. Also, the path-annealing time advantage appears to increase as problem instances become larger in size. Both algorithms use Snell's Law to optimize within the window sequence determined to contain the optimal path. Therefore, compared to grid-base wavefront propagation, path annealing and UDPGA* final solutions can be more accurate, since they avoid the digital bias problem associated with an orthogonal grid. Furthermore, for similar resolutions, we believe that the solutions found by UDPGA* are likely much closer to true optimal than will be those found by wavefront propagation. In this regard, path annealing solutions are also accurate. However, their nearness to true optimal can be both much worse and much better than UDPGA*.

Significant advantages result from the fact that path annealing examines only complete solutions. This gives the algorithm the potential to discover and cache many good routes between a given start and goal. Furthermore, additional opportunities occur to update the upper bound on the optimal solution cost, which controls the size of the bounding ellipse. Solution quality tends to improve over time after the establishment of the initial solution. In time-constrained circumstances, we can abort path annealing prematurely and accept the best known solution. This cannot be done with systematic search techniques because they produce a final solutions by constructing and comparing many smaller partial solutions.

Since path annealing is probabilistic, its results are non-deterministic. Multiple runs of the same problem instance may return different solutions. Even if two runs return identical solutions, the sequence of states processed to reach those solutions are almost never identical. Therefore, every run has the potential to discover other good routes between designated start and goal. This behavior can be exploited by parallel execution on multiple processors, to which path annealing is easily adapted. Recall from Chapter IV that by constraining the A* search in various ways, we can force it to locate several initial solutions corresponding to different high-speed corridors. Since each processor begins from a different initial solution, we are able to increase efficiency by annealing at lower starting temperatures, and, at the same time, we broaden coverage of the search space.

Many spatial constraints and heuristics are available to reduce search space and improve execution times. Some of these are annealing-specific, while others are useful for local search techniques which examine complete solutions. We have also developed shortcutting analysis and critical-angle analysis. These are general spatial constraints which can be used by most other weighted-region problem algorithms.

2. Weaknesses

Apparently no practical WRP algorithm can provide an absolute guarantee of finding a globally optimal path. However, systematic approaches can effectively upper bound the error if the optimum is overlooked. For example, the total cost of an 8-neighbor grid-based wavefront propagation solution can be no more than 1.08 of true optimal solution cost. We emphasize that while execution times can be more practical, the cost of a path annealing solution is bounded above by a function of the initial A* solution cost. This represents a weaker upper bound, since this is restricted to midpoint paths. While improvements can be made by increasing the resolution of the initial A* search, tests on UDPGA* indicate that execution times will rise quite rapidly.

To obtain good results from path annealing requires some general knowledge of the topology and cost function of the map. Input maps should have reasonably balanced resolution, devoid of boundary edges which are unusually long in relation to neighboring edges. Regions should not be extraordinarily narrow. Recall that improper window sequences involving the multiple crossing of such edges and regions may not be reachable by the simple move generator we have designed. A more elaborate move generator might remove this restriction. However, this may also increase the time required to generate the next window sequence.

Path annealing appears to work best on maps with smoother cost functions. This comes as no surprise since simulated annealing reportedly performs best in such search spaces [John90]. Hilly cost functions increase the risk of prematurely freezing the annealing process in a bad local minimum. On the other hand, hilly cost environments appear to require higher resolution of the UDPGA* algorithm in order to obtain solutions nearer to true optimal. In turn, this results in much longer running times. Though the general performance of path annealing in such maps tends to be worse, it can still find a few dramatically better solutions in less time.

Good algorithm performance can be sensitive to the annealing schedule. There are many ways to establish good annealing schedules. We have suggested some primitive methods on the basis of current annealing and WRP research. But, our testing does not attempt to determine how best to establish annealing schedules. Instead, we used our own judgement based upon experience with the test maps. However, in most cases, establishment of an annealing schedule must be done empirically or with some knowledge of the search space.

Finally, even with the combination of simulated annealing, A* search, relaxation, bound construction, and heuristic improvements, the path annealing algorithms still only solves a single problem instance for a specific start/goal pair.

3. Strengths of Uniform-Discrete-Point Global A* (UDPGA*) Search

We have focused on the advantages and disadvantages of path annealing. However, we believe that the uniform-discrete-point global A* search algorithm provides fairly stiff competition for path annealing. We reemphasize that UDPGA* search has a few advantages of its own over grid-based wavefront propagation. Since it employs an adaptive grid similar to a visibility graph, it is capable of more accurate solutions than orthogonal-grid-based wavefront propagation. Although we have not made direct comparisons, we conjecture that UDPGA* would be generally faster than wavefront propagation because it is not uniform-cost search, and therefore does not search in simulated time increments. Furthermore, UDPGA* preserves the ability to adjust resolution through the establishment of edge-points spaced at a discrete upper bound, δ. Finally, unlike wavefront propagation, UDPGA* actively searches and detects reentrant-path-containing solutions.

C. SUGGESTED FUTURE DIRECTIONS

1. Path Annealing Improvements

Our primary focus in this work has been to establish that a simulated-annealing approach to the weighted-region problem (WRP) is both feasible and practical. In designing a working implementation, we took steps to streamline particular aspects of the algorithm, such as cost function evaluation and move generation. In doing so, we designed several methods for extracting performance improvements and we have suggested several others. However, there are surely other techniques for driving these components which might be more efficient or more thorough. In particular, more sophisticated move operators could be developed by relying more heavily on efficient random access data structures (such as arrays). Prolog dictated our use of list-oriented data structures.

The Snell's-Law table lookup could also be improved. Recall from Chapter III that this technique reduces ray-tracing to constant-time table interpolation for a single crossing episode, so that an iterative coordinate descent method (single-path relaxation) can find the locally optimal path. The table could be extended to compute two or more crossing episodes. This might increase the efficiency of single-path relaxation, particularly for longer window sequences.

The spatial nature of the weighted-region problem suggests that several undiscovered geometric heuristics may still exist. In particular, the idea of a bounding ellipse originally suggested by [Rich87] and [Pear84] appears to have potential. We have improved the technique and heuristically extended its usefulness (e.g. μ^+-ellipse). However, there are other interesting possibilities. For example, some form of crude bidirectional search might quickly outline a general elliptical but irregular boundary which contains the a global optimal path.

The determination of good annealing-schedule parameter values is another aspect of the algorithm which may deserve more attention. However, recall that path annealing employs the power of A* search as well as local search. Therefore, we believe that good performance of path annealing may not be as sensitive to the annealing schedule as the need to obtain good general coverage of the map. This might be done by designing a more elaborate move generator. Another way is to find a good set of initial solutions to use as a basis for tunneling at lower annealing temperatures. We have suggested that several crude A* searches could be used to find this set. However, some limited or modified algorithm to find the first *k* shortest paths might do this more efficiently [Yen71].

If parallel processing is available, then a horizontal asynchronous scheme for path annealing would require a set of different starting solutions. Each processor would draw one initial solution out of this set from which to begin annealing at a low starting temperature. Such a parallel scheme could proceed with very little communication overhead. Annealing can also be parallelized in a vertical sense. Move operator and cost function evaluation algorithms could be redesigned for parallel execution. The ideas for local asynchronous iterative parallel procedures (LAIPP) first proposed and implemented by [Smit88] and later implemented for a transputer by [Garc89] have already laid some of that groundwork.

2. Extensions to Other Route Planning Applications

In his work on shortest paths in three-dimensional space, [Akma87] proposed that Monte Carlo algorithms may constitute the only reasonable means of solving intractable path-planning problems associated with higher-dimension space. We share his belief that such algorithms are not always appreciated by the computer science community. Having demonstrated the feasibility of a stochastic approach to the two-dimensional weighted-region problem, a logical extension would be to solve a three-dimensional weighted-volume problem. Analogous to aspects of path annealing, the work of [Lewi88] and [Rowe89] attacks this problem. This implementation finds a sequence of volumes using A* search through volume centroids, then iteratively improves the path in a manner similar to single-path-relaxation. Follow-on work by [Wren90] implements random ray-tracing to improve the A* solution paths. However, neither implementation uses stochastic control.

In practical three-dimensional route planning it is often necessary to consider vehicular constraints. For example, shortest paths for high performance aircraft differ from those of helicopters because of turning radius constraints. To account for vehicular direction, extension to a fourth dimensional may be necessary. Even without weighted volumes, finding optimal paths through fields of obstacles in such search spaces is another problem for which a simulated annealing approach could be valuable.

Another interesting problem for which annealing might provide significant improvements is one of locating minimal energy routes through anisotropic terrain [Ross89]. Ideas in this work have inspired some of our own, particularly those pertaining to definition and representation of the search space. However, we emphasize that the anisotropic modeling of cost regions precludes a simple translation from path annealing. In particular, we suspect that a more sophisticated move generation mechanism would be required.

Although we have said that path annealing integrates and coordinates the operations of several techniques to solve for only a single problem instance, the idea of constructing optimal-path maps [Alex90] probabilistically remains an interesting question. Particular problem instances have optimal solutions which appear to be easily found by path annealing. Given a single goal (start), annealing might be capable of constructing a proposed optimal-path map of these easily found solutions. Call these *skeleton paths*, that is suspected (with relatively high probability) optimal paths to several scattered start (goal) locations. The paths in the skeleton would be assumed optimal until conditions were detected which indicated otherwise. Such conditions might be the discovery of better paths which cross over one or more of the current skeleton paths. These conditions would trigger updates to the skeleton. Over time continued annealing would eventually confirm or correct the skeleton. The algorithm would gradually fill in gaps between skeleton paths and confirm them until the entire map was complete. The process would essentially learn the optimal-path map by sampling, assuming, correcting, and confirming.

D. CONCLUDING REMARKS

In this dissertation we have designed, implemented, and tested an efficient path-planning algorithm to solve the weighted-region problem. This algorithm integrates systematic and local search strategies under stochastic control to obtain solutions which are optimal or near-optimal. We have demonstrated that an intelligent, probabilistic approach to the weighted-region problem can obtain faster average execution times in exchange for the marginal risk of a bad solution. We believe that path annealing represents a successful endeavor, whose further development and extension to other path-planning problems will be a worthwhile effort.

APPENDIX A - THEOREMS

Definitions:

Planar Straight-Line Graph (PSLG) - a graph which can be embedded in Euclidean E^2 space without any of its edges crossing and whose edges are all straight line segments [Prep88]

Monotone PSLG - Consider a PSLG which subdivides Euclidean E^2 space. A subset of E^2 is monotone with respect to the x-axis if the intersection of each region in the subset with any vertical line results in at most one segment. A subdivision of E^2 is monotone (with respect to the x-axis) if all of its regions are monotone [Edel89]. This definition implies that every vertex has degree 2. We relax this definition slightly by allowing a PSLG to be monotone if any orientation of the reference coordinate system admits a monotone graph.

Boundary Edge - finite length line defined by two terminating vertices, separating two homogeneous-cost regions in the map.

Traversable Region or Edge - travel on or within is feasible; cost coefficient $\mu < \infty$

Non-Traversable Region or Edge - travel on or within is infeasible; cost coefficient $\mu = \infty$; e.g. obstacles and edges bounding obstacle regions

Crossable Edge - boundary edge between two traversable regions; note that obstacle boundary edges are traversable but NOT crossable

Border - the outer limits of a finite map beyond which movement is infeasible and cost coefficient $\mu = \infty$

Border Edge - a single unique non-crossable edge; all edges attached to a border vertex also link to the border edge; the degree of the border vertex is always > 1.

Edge Chain - connected sequence of vertices such that all vertices have degree = 2, except the first and last which have degree > 0

Border Vertex - terminates a traversable edge at the border

Obstacle Vertex - terminates at least one non-traversable edge of an infinite cost region

Interior Vertex - vertex which is not an obstacle or border vertex

Closed Region - traversable region; all boundary edges are traversable

Open Region - traversable region; has at least one non-traversable boundary edge or at least one border vertex

Vertex Rotation - move operation defined by shifting window sequence across a neighboring interior vertex, thereby rerouting it through a different sequence of edges

Adjacent Edge-Pair - two edges which bound a common convex region and share a common terminating vertex

Non-Adjacent Edge-Pair - two edges which bound a common convex region but do not share a common vertex

Edge Visibility - edge E_1 and E_2 are visible from one another if and only if for any straight path between interior points of E_1 and E_2, all points on the path belong to the interior of the common region bounded by E_1 and E_2.

Window Sequence (WS) - any ordered set of boundary edges which a piecewise linear path from the designated start must cross to reach the designated goal.

Proper Window Sequence - window sequence which does not cross through any region more than once. This also implies that no edge is crossed more than once.

Improper Window Sequence - window sequence which crosses through at least one region more than once.

Reentrant Window Sequence - improper window sequence which contains at least one reentrant path

Self-Crossing Path - a cyclic path; note that by the Optimality Principle, such paths cannot be globally optimal

Reachable - window sequence WS_2 is reachable from another window sequence WS_1 if and only if there exists a finite sequence of vertex rotations which transforms WS_1 to WS_2

n-Reachable - WS_2 is n-reachable from WS_1 if and only if there exists a sequence of n or fewer vertex rotations which transforms WS_1 to WS_2 (We also say that WS_2 is an n-neighbor of WS_1)

Weighted-Region-Problem Map - a PSLG which satisfies the following constraints:

- All regions are convex polygons.
- All edge-pair interior angles are strictly less than 180 degrees.
- All vertices terminate a minimum of 3 boundary edges.
- All traversable edges are connected.

The following set of theorems are proven for monotone PSLG's. The set of all WRP maps is a more restrictive class of graphs and is a proper subset of the set of all monotone PSLG's. Therefore, these theorems also apply to WRP maps. Without loss of generality, we may assume that the infinite non-traversable border region does not exist, and that edges which would otherwise connect to it extend infinitely. This is not a problem because locally optimal paths cross such edges at Snell's Law. Since start and goal are well-defined terminal points for all feasible solution paths, then the optimal crossing points on edges which are infinite rays cannot possibly exist at infinity.

Theorem 3.1:

Consider the set of all proper window sequences (PWS) which cross a monotone edge chain in a finite PSLG. The edges in the chain partition the set of proper window sequences into disjoint subsets. Each subset is reachable from the other.

Proof of Theorem 3.1:

The proof is trivial since it is always possible to perform successive vertex rotations in either direction over the entire length of an edge chain. Therefore, any PWS which can reach one subset can reach all subsets. QED.

Theorem 3.2:

Given a monotone connected PSLG, it is always possible to remove one monotone edge chain (thereby, reducing the number of regions by one) and still preserve monotonicity and connectedness.

Proof of Theorem 3.2:

Consider an arbitrary monotone connected PSLG. Since monotonicity implies a well-defined ordering of regions with respect to the x-axis, then the bottom-most and top-most regions can always be identified. Without loss of generality, we will remove a chain from the boundary of the bottom-most region, R. Starting from the left ray, scan right and examine each vertex in sequence which is degree > 2. If no such vertex exists, then the graph must consist of a single monotone edge chain separating two regions. Merge the regions by removing this entire chain. Otherwise, find the common chain of edges separating R from its nearest neighbor above (in the monotonic ordering). This chain must be in one of three classes:

- left ray to right intermediate vertex
- left intermediate vertex to right intermediate vertex
- left intermediate vertex to right ray

We can always identify the terminal locations of this chain by selecting the first vertex of degree > 2 with an incident edge (not part of the boundary of R) that exits the vertex on the side of a vertical axis through the vertex which opposes our scan direction. This vertex and the last vertex of degree > 2 (or the left ray if this vertex was the first of degree > 2) are the terminal locations of the chain. Note that the scan may not find a right vertex with incident edge opposing the direction of scan. In this case, the right ray becomes the right terminal. The procedure cannot disconnect the graph because vertices defining the chain to be removed are selected on the basis of incident edges which always have opposing headings with respect to the vertical axis. Therefore, the chains to which they are connected must be the same, otherwise the graph was not monotonic to begin with. This means that there will always exist a second chain which links left and right vertices of the chain to be removed. Therefore, these vertices will still be attached after removal of one chain. It follows that the graph remains connected. Furthermore, since we always merge the bottom-most or top-most regions, the PSLG remains monotone. **QED.**

Theorem 3.3:

If a finite connected PSLG is monotone, and if start and goal lie in different regions, then all PWS's are reachable from any initial PWS.

Proof of Theorem 3.3 (by induction on the number of regions):

(*Base case*) Consider a monotone connected PSLG with only two traversable regions, where start lies in one and goal lies in the other. The theorem holds for this case. To see this observe that for any finite number of vertices in the chain of common edges which separates the two regions, all PWS's are reachable by vertex rotations along this common chain. (*Inductive hypothesis*) Assume that the theorem is true for a finite (*n*)-region PSLG where start and goal lie in different regions. (*Induction*) Consider an (*n+1*)-region map. Arbitrarily select one region, *R*, to merge with either start or goal-containing region, *T*. *R* must satisfy the following conditions:

- *R* does *not* contain the start or goal.
- *R* shares at least one edge with *T*.
- All shared edges form a continuous sequential chain, *C*.
- Removal of *C* does *not* destroy monotonicity.

Theorem 3.2 guarantees that at least one chain exists which satisfies these conditions. Merge *R* and *T* into a single region, *T'*, by removing the entire edge chain common to *R* and *T*. Let the set of edges in this chain be *C*. The resulting map will have *n* regions. By inductive hypothesis all PWS's in this (*n*)-region map are reachable from any initial PWS between start and goal. We claim that all PWS's in the (*n+1*)-region map are reachable from any initial PWS between start and goal. To see this, first consider the set C. Since C is an edge chain, then by Theorem 3.1 any PWS that crosses an edge in C can reach any other PWS that crosses an edge in C. (We will say that such a PWS is C-crossing.) Next, consider all ordered pairings of boundary edges between R and T (one taken from the R boundary, and one taken from the T boundary) where neither is in C. For each such pair, all non-C-crossing PWS's containing this pair in the $(n+1)$-region map must be reachable by inductive hypothesis. For each such pair, all C-crossing PWS's containing this pair in the $(n+1)$-region map correspond to the same PWS's in the (n)-region map where the edge in C has been removed. This one-to-one correspondence between (n)-region and $(n+1)$-region PWS classes means that removal of the edge chain C does not change the total number of PWS's. Since the inductive hypothesis guarantees that all PWS's in the (n)-region map are reachable, then it follows that all PWS's in the $(n+1)$-region map must also be reachable. QED.

Definitions:

G - an arbitrary globally optimal path

R_i - homogeneous cost region i

μ_i - cost coefficient assigned to region i

E_i-E_j - edge-pair consisting of E_i and E_j

p_i - arbitrary interior point on p_i (i.e. not a vertex)

p_j - arbitrary interior point on p_j (i.e. not a vertex)

α - interior angle formed by an edge-pair

P_1 - clockwise perimeter bypass path

P_2 - counterclockwise perimeter bypass path

Theorem 4.1:

In Figure A.1 consider the edge-pair E_1-E_2 which bounds high-cost region R_0 whose cost coefficient is μ_0. Let $f(p_i,p_j)$ be the function which maps all pairs of interior points p_i and p_j between edge-pair E_1-E_2 to the maximum of the two ratios P_1/d and P_2/d, where P_1 and P_2 are the weighted costs of the perimeter paths clockwise and counterclockwise around R_0 from p_i to p_j, and d is the unweighted Euclidean distance between points p_i and p_j. That is, $f(p_i, p_j) = \text{Max}(P_1/d, P_2/d)$. Since the length of each edge in E_1-E_2 is finite, then $f(p_i, p_j)$ must reach some maximum value, μ_t. If $\mu_0 > \mu_t$, then no optimal path can cross through region R_0 from interior points of edge-pair E_1-E_2.

Proof of Theorem 4.1 (by contradiction):

Suppose there exists a globally optimal path G which crosses R_0 from interior points of edge-pair E_1-E_2, and that $\mu_0 > \mu_t$. For simplicity, let $\mu_t = P/d$, where $P = \text{Max}\{P_1, P_2\}$. Then it must be true that $\mu_0 > P/d$. It follows that $\mu_0 d > P$. But this contradicts the Principle of Optimality, that is all subpaths of a globally optimal path must be optimal themselves. If $\mu_0 d > P$, then the perimeter path, P, must be shorter than the segment which lies on the globally optimal path, G. Therefore, G cannot be a globally optimal. But P/d represents the maximum ratio over all crossings of R_0 via edge-pair E_1-E_2. We conclude that no globally optimal path can cross R_0 from edge-pair E_1-E_2 because a shorter perimeter path always exists. QED.

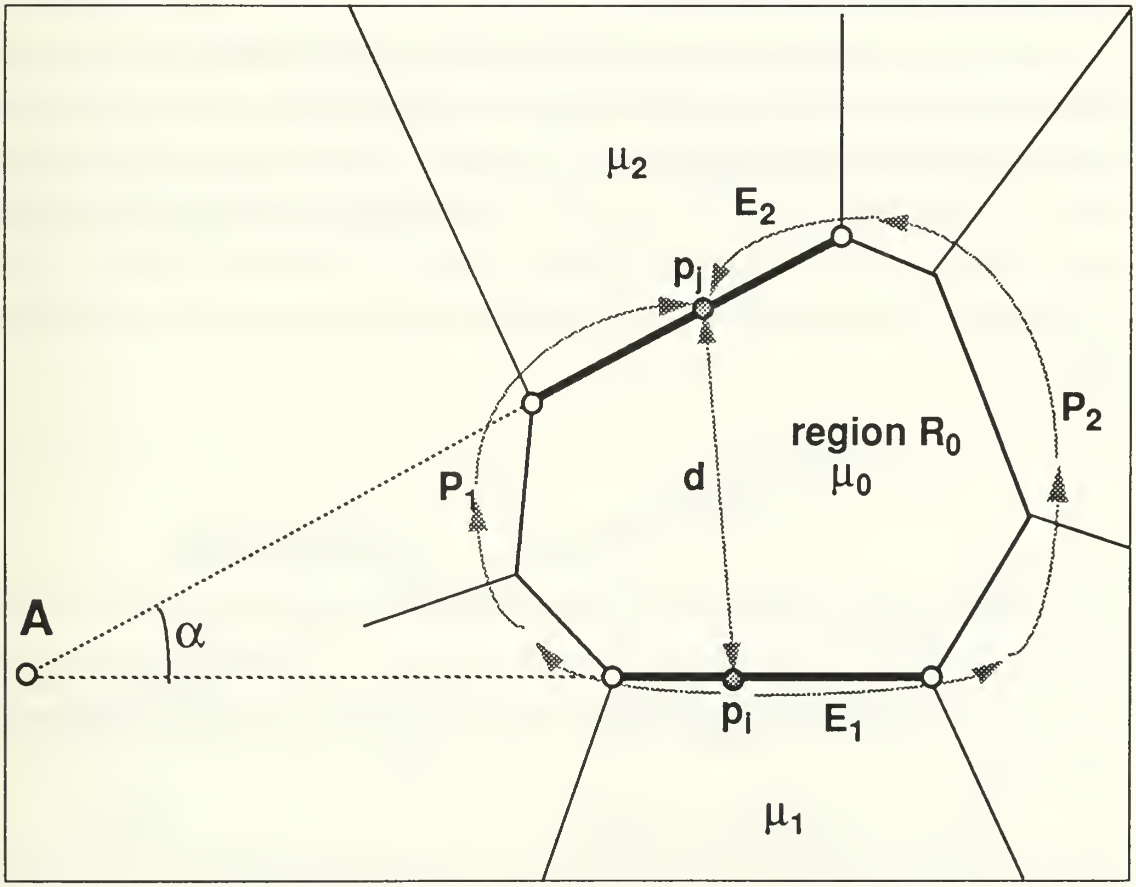


Figure A.1 Edge-Pair E_1-E_2 on Homogeneous-Cost Region R_0

The significance of Theorem 4.1 is that for a given edge-pair on region R_0, if we can determine μ_t, the maximum attainable value of P/d over all interior crossing points, then a simple comparison to μ_0 tells us if the edge-pair can ever be crossed by any globally optimal path, whose start and goal points do not reside within R_0. Since μ_t is the maximum attainable value of P/d over all interior crossing points, then any upper bound on this value which is exceeded by μ_0, the cost coefficient of R_0, can be used to eliminate this edge-pair as an optimal path crossing site.

It is important to note that we need only concern ourselves with analysis of crossings involving a pair of *interior* points on the edge-pair. The reason is that the search space of window sequences is redundant at boundary edge vertices. That is, any path or portion thereof, which passes through a map vertex must be within two or more distinct window sequences. Thus, eliminating edge-pair E_1-E_2 from region R_0 means that the value of μ_0 is too large to permit an optimal crossing between any interior points of E_1-E_2. It does not mean that an optimal path cannot cross over E_1 or E_2 by using a vertex of the other. However, doing so constitutes use of a neighboring edge-pair.

The greatest perimeter bypass cost for any edge-pair on any region is bounded above by one half the total weighted cost of traveling in a closed-loop around the region following its boundary edges. This cost value is easily computed for any region. We will refer to it as P_h. Note that there may exist many pairs of points p_i,p_j on an edge-pair for which $P_1 = P_2 = P_h$. To obtain the tightest upper bound on P/d we must find a pair of interior points for which $P_1 = P_2 = P_h$ and d is a minimum. On the basis of geometry, it is possible to find a subspace of crossings through R_0 between E_1-E_2 which maximizes the ratio P/d. Theorem 4.2 describes how.

Theorem 4.2:

Refer to Figure A.2. If the infinite line extensions of an edge-pair E_1-E_2 bounding region R_0 intersect at a point A at which an interior angle α is formed, and if $\mu_1 < \mu_0$ and $\mu_2 < \mu_0$ are the cost coefficients of E_1 and E_2 to the outside of R_0, then for any crossing of E_1-E_2 from interior points p_i to p_j the maximum value of P/d occurs when

$$s/t = (\mu_1 + \mu_2 \operatorname{Cos}\alpha) / (\mu_2 + \mu_1 \operatorname{Cos}\alpha) \qquad \text{(Eq A.1)}$$

where s is the Euclidean distance from A to p_i along E_1, and t is the Euclidean distance from A to p_j along E_1.

Proof of Theorem 4.2:

We obtain Eq A.1 by using Figure A.3. In this figure the edge-pair is adjacent and so a perimeter bypass path from p_i to p_j has weighted distance:

$$P = \mu_1 s + \mu_2 t \qquad \text{(Eq A.2)}$$

By the Law of Cosines a crossing path from p_i to p_j through R_0 has unweighted Euclidean distance:

$$d = \sqrt{s^2 + t^2 - 2st\text{Cos}\alpha} \qquad \text{(Eq A.3)}$$

The ratio P/d is now a function of s and t:

$$f(s,t) = \frac{\mu_1 s + \mu_2 t}{\sqrt{s^2 + t^2 - 2st\text{Cos}\alpha}} \qquad \text{(Eq A.4)}$$

Taking the partial derivatives of $f(s,t)$ with respect to s and t, setting them equal to zero, and solving for s and t yields:

$$s/t = (\mu_1 + \mu_2 \text{Cos}\alpha)/(\mu_2 + \mu_1 \text{Cos}\alpha) \qquad \text{(Eq A.5)}$$

Since $f(s,t)$ reaches its maximum value when the ratio s/t satisfies Eq A.1, then P/d is also maximum when this condition is satisfied. Although we began with an adjacent edge-pair intersecting at common vertex A, note that Eq A.1 is the maximizing condition for any edge-pair whether adjacent or non-adjacent. The reason is that only Eq A.2 (cost of the perimeter path) changes if the edge-pair is non-adjacent. In this case, the addition of a constant term (fixed cost to travel the perimeter edges between the edge-pair) does not effect the result of taking partial derivatives. QED.

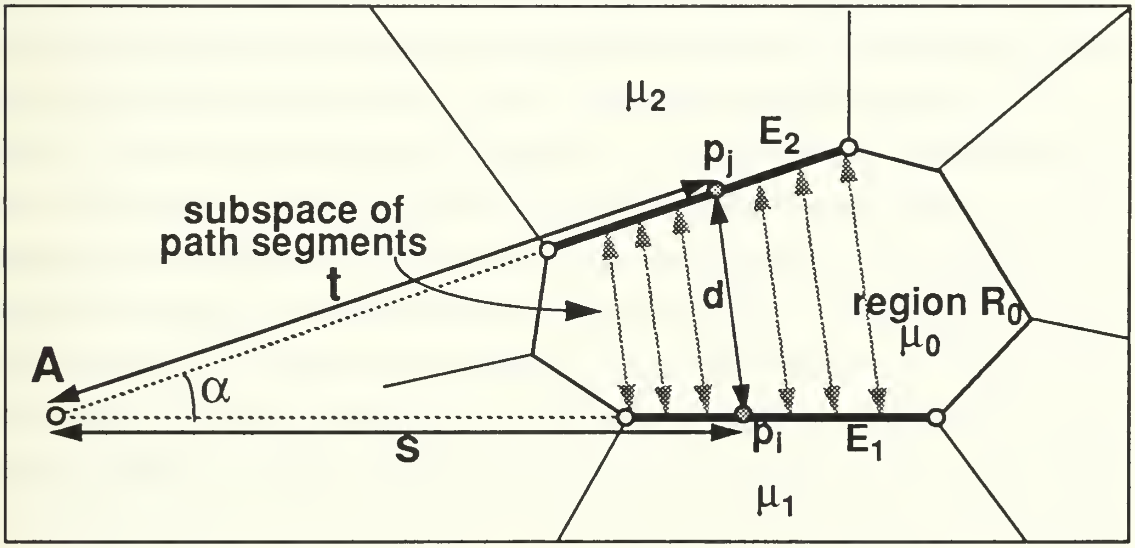


Figure A.2 Non-Adjacent Edge-Pair

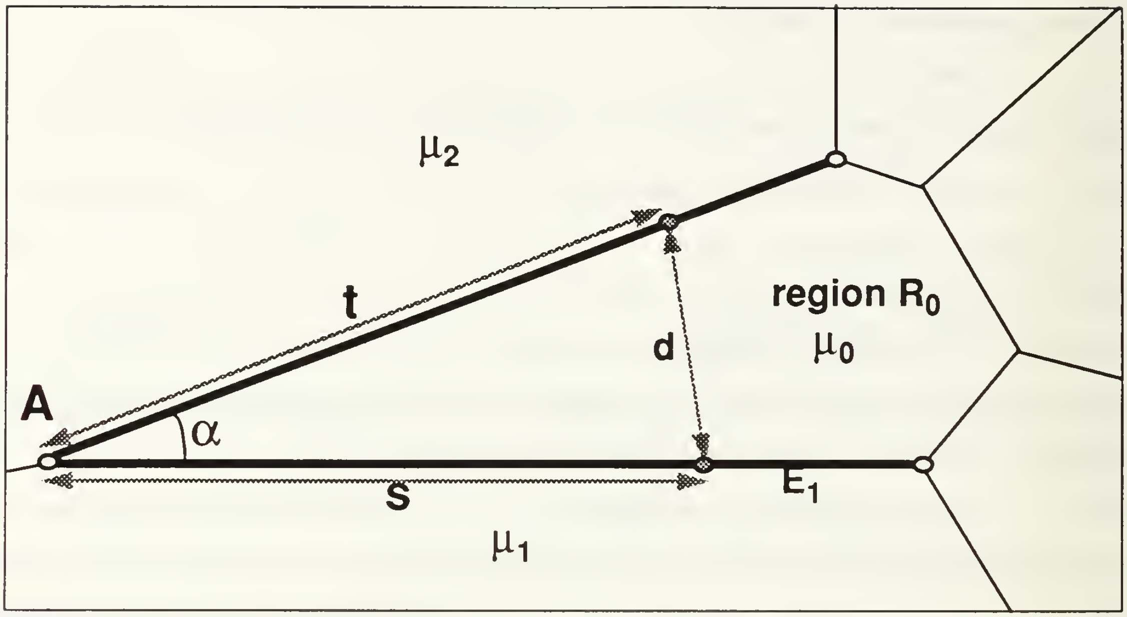


Figure A.3 Adjacent Edge-Pair

Theorem 4.2 tells us that with respect to P/d, the maximum advantage for an edge-pair crossing on interior points occurs in a subspace of segments between the edges. Figure A.2 illustrates such a subspace. The bounds of the subspace are defined by vertices of both edges in the pair. In the case of a parallel edge-pair angle $\alpha=0$. Therefore, Eq A.1 tells us that the maximum advantage occurs at perpendicular crossings.

We can employ the concepts of Theorems 4.1 and 4.2 to find the value of μ_t for a given non-adjacent edge-pair as follows. Using the values of μ_1, μ_2, and angle α compute the ratio for s/t by Eq A.1. Using this ratio we can determine the two segments which bound the subspace of crossings that maximize P/d. A binary search can locate the segment which divides the clockwise and counterclockwise perimeter bypass path-cost exactly in half. The value of $\mu_t = P_h/d_h$ where d_h is the Euclidean distance of the segment at half perimeter. If such a segment does not exist within the subspace, then the subspace boundary segment whose perimeter paths (clockwise and counterclockwise) have the least difference should be used. Let its smaller perimeter path-cost be P_s. Then the value of $\mu_t = P_s/d_s$ where d is the Euclidean distance of the segment bounding the subspace.

For adjacent edge-pairs the computation is more straight forward. There is no need to find the bounding segments of the subspace because one end is bounded by a vertex. So we can use Eq A.1 to substitute for *s* or *t* directly into Eq A.4. This yields the following equation for adjacent edge-pairs:

$$\mu_t = \frac{\sqrt{\mu_1^2 + \mu_2^2 - 2\mu_1\mu_2 \mathrm{Cos}\alpha}}{\mathrm{Sin}\alpha} \qquad \text{(Eq A.6)}$$

Having computed μ_t for the edge-pair in question, we can now compare to μ_0, the cost coefficient between the edges. If $\mu_0 > \mu_t$, then the weight in region R_0 is too great to permit an optimal path to cross on interior points of the edge-pair.

APPENDIX B - PROLOG AND C SOURCE CODE

B.1 PATH ANNEALING

```
/*======================================================================
 SA.PL - simulated annealing search thru the space of window sequences
========================================================================

Annealing schedule notes:

A schedule is defined in terms of the initial A* solution cost.
The predicate annealing_schedule(TC,TF,TR,NT,NS) defines the parameters
to be used in the call to init_anneal(TS,TF,TR,NT,NS).

      TS starting temperature in units of cost
      TC cost multiplier, ensures high temp (TS=TC*init_soln_cost)
      TR temp reduction factor (geometric schedule) (T2 <- TR*T1)
      NT n_limit, total number ATTEMPTED moves at temperature T
      NS ns_limit, total number ACCEPTED moves at temperature T

------------------------------------------------------------------------


:- dynamic
      finished/0,
      annealing_schedule/5,        % annealing_schedule(TC,TF,TR,NT,NS).
      starttemp/1,
      current/3,
      next/1,
      best/4,
      newbest/4,
      ultimate_best/4,
      best_count/1,
      hash_ws/4,
      hits/1,
      hthresh/1,
      deltak/1,
      time/1.

annealing_schedule(1.5,1.0,0.9,20,15).                % default schedule

hthresh(10).                                                   % default

hash_fn([_|L],H) :- sumlist(L,H), !.                          % sum of E

sumlist([_],0) :- !.
sumlist([X|L],SUM) :- sumlist(L,S), SUM is X+S, !.
```

```
prep :-
      compute_deltak,
      w_initparms,                                % time, di, st, deltak
      retractall(finished),
      retractall(starttemp(_)),
      retractall(current(_,_,_)),
      retractall(next(_)),
      retractall(best(_,_,_,_)),
      retractall(newbest(_,_,_,_)),
      retractall(ultimate_best(_,_,_,_)),
      retractall(best_count(_)),
      retractall(hash_ws(_,_,_,_)),
      retractall(hits(_)),
      asserta(best_count(0)),
      asserta(hits(0)),
      w_wsinit,
      randinit(_),
      reset_points,                          % remove epts from last run
      !.

compute_deltak :-
      retractall(deltak(_)),
      start(S,SR),
      goal(G,GR),
      rv(SR,SW,_),
      rv(GR,GW,_),
      di(DI),
      DK is 0.5*DI*(SW+GW),
      asserta(deltak(DK)),
      !.

install_best(COST,WS,PATH) :-
      retract(best_count(N)),
      M is N+1,
      asserta(best_count(M)),
      asserta(best(N,COST,WS,PATH)),
      !.

%-----------------------------------------------------------------------

anneal :-
      prep,
      ts,
      search(WS0,_),                                 % A* thru midpoints
      dijkstra(WS0,DSEED,_),                         % approximate w/UDP
      solve(WS0,DSEED,SEED,COST),                         % refine w/SPR
      avg_path_wt(COST,SEED,APW),
%     asserta(optimal_cost(APW)),                    % heuristic ellipse
      write_list(['Average Weight =',APW]),
      ellipse(COST),                                  % bounding ellipse
      install_best(COST,WS0,SEED),
      asserta(current(COST,WS0,SEED)),
      hash_fn(WS0,H),                               % hash fn for ws
      asserta(hash_ws(H,WS0,SEED,COST)),          % insert in hash table
      updates(COST,WS0),
      annealing_schedule(TC,TF,TR,NT,NS),
      TSTART is TC*COST,
      asserta(starttemp(TSTART)),
      init_anneal(TSTART,TF,TR,NT,NS),             % set annealing parms
      anneal_loop,                                 % search by annealing
      !.

anneal_loop :-
      repeat,
      current(C1,WS1,_),
      perturb_ws(WS1,WS2),
%     perturb_ws(WSA,WS2),                            % for larger moves
%     asserta(next(WS2)),                           % for debugging only
      hash_fn(WS2,H),
      compute_cost(H,WS2,PA2,C2),
      metrop(C1,C2,I),
%     (I=1 -> writeln(C2)),                      % observation tool only
      action(I,C2,WS2,PA2),
%     retract(next(_)),                             % for debugging only
      finished,
      !.

hit_rate :-
      hits(K),
      hthresh(T),
      hit_msg(K,T),
      retract(hits(_)),
      asserta(hits(0)),
      !.
hit_rate :- !.

hit_msg(K,T) :-
      K > T,
      writeln('Rising cache hit rate...return to a previous best'),
      best_count(N),
      return_to_a_previous_best(N),
      !.

return_to_a_previous_best(1) :- replace_current(0), !.
return_to_a_previous_best(N) :-
      irand(N,X),
      replace_current(X),
      !.

replace_current(X) :-
      retract(current(_,_,_)),
      best(X,C,W,P),
      asserta(current(C,W,P)),
      !.

action(X,C2,WS2,PA2) :-
      ( X =:= 0 -> fail
      ; X =:= 1 -> accept(C2,WS2,PA2), hit_rate
      ; X =:= -1 -> frozen(C2,WS2,PA2) ),
      !.

accept(C2,WS2,PA2) :-
      retract(current(_,_,_)),
      asserta(current(C2,WS2,PA2)),
      compare_to_best(C2,WS2,PA2),
      !.

frozen(C2,WS2,PA2) :-
      compare_to_best(C2,WS2,PA2),
      asserta(best(C2,WS2,PA2)),            % check vicinity for optimal
      update_clock(T1),
      write_list(['iterative improvement begins time',T1]),
      iter_improve,
      ultimate_best(N,C,WS,PATH),
      update_clock(T2),
      nl,
      w_showdata(pacost,C),
      w_showpath(PATH),
      write_list(['finish time =',T2]),
      write_list([N,'Optimal Cost =',C]),
      write_list([' WS =',WS]),
      write_list(['Path =',PATH]),
      nl,
      w_clear_ws,                          % reqd by ws display routines
      asserta(finished),
      notrace.

iter_improve :-
      best(_,_,_,_),
      repeat,
      get_top_best(N,C,WS,PA),
      perturb_ws(WS,WS0),
      hash_fn(WS0,H),
      compute_cost(H,WS0,PA0,C0),
      iter_impr(C,C0,X),
      ( X =:= 0 -> fail
      ; X =:= 1 -> retract(best(N,_,_,_)),          % from top of stack
        asserta(best(N,C0,WS0,PA0)),
        fail
      ; X =:= -1 -> retract(best(NN,CC,WW,PP)),
        asserta(newbest(NN,CC,WW,PP)),
        iter_improve ),
      !.
```

```
iter_improve :-
      retract(newbest(N,C,WS,PA)),
      asserta(ultimate_best(N,C,WS,PA)),
      retrieve_best,
      !.

get_top_best(N,C,WS,PA) :- best(N,C,WS,PA), !.

retrieve_best :-
      retract(newbest(N0,C0,WS0,PA0)),
      ultimate_best(N,C,WS,PA),
      ( C0=<C -> retract(ultimate_best(N,C,WS,PA)),
        asserta(ultimate_best(N0,C0,WS0,PA0)) ),
      retrieve_best,
      !.
retrieve_best :- !.

compute_cost(H,WS2,PA,C) :-                              % hash for cost
      hash_ws(H,WS2,PA,C),                                         % hit
      score(hit),
      !.
compute_cost(H,WS2,PA,C) :-                          % must compute cost
      dijkstra(WS2,PA2,C2),                               % approx w/UDP
      best(_,C0,_,_),
      refine(WS2,PA2,C2,C0,PA,C),
      asserta(hash_ws(H,WS2,PA,C)),                              % cache
      score(cache),
      !.

refine(WS2,PA2,C2,C0,PA,C) :-
      deltak(DK),
      C2-C0 < DK,
      solve(WS2,PA2,PA,C),                                % refine w/SPR
      !.
refine(_,PA2,C2,_,PA2,C2) :- !.                  % not worth the trouble

score(X) :-
      retract(hits(I)),
      ( X==hit -> J is I+1
      ; X==cache -> J is I-1 ),
      asserta(hits(J)),
      !.

compare_to_best(C2,WS2,PA2) :-
      best(_,C0,_,_),
      ( C2 < C0 -> install_best(C2,WS2,PA2),       % install best so far
        updates(C2,WS2)
      ; otherwise -> updates(null,WS2) ),
      !.
```

```
updates(C,WS) :-
      update_clock(T),
      w_show_ws(WS),                                   % display new WS2
      ( C==null -> true
      ; otherwise -> best(N,_,_,PA),                  % must be new best
        w_showpath(PA),
        w_showdata(pacost,C),
        write_list([T,sec,N,'New best cost =',C]) ),
      time_expired,
      tm,
      !.

time_expired :-
      time(T),
      T > 2000.0,
      w_clear_ws,
      writeln('Time expired: 2000 sec...execution terminates.'),
      best(N,C,WS,PATH),                                   % best to date
      write_list([N,'Optimal Cost =',C]),
      write_list([' WS =',WS]),
      write_list(['Path =',PATH]),
      asserta(finished),
      !.
time_expired :- !.
```

```
/*======================================================================
 ASTAR.PL - A* SEARCH
 adapted from Rowe, N.C., "Artificial Intelligence Through Prolog,"
 Prentice Hall, 1988, pp. 244-247.
========================================================================

 For an application, define 5 predicates:
      (a) successor(State,Newstate)            (gives state transitions)
      (b) goalreached(State)                (defines when goal achieved)
      (c) eval(State,Evaluation)            (estimates cost to the goal)
      (d) cost(Statelist,Cost)                 (computes cost of a path)
      (e) top-level predicate that initializes things if needed,
          then calls astarsearch with two arguments, the starting
          state and variable which will be the solution path

 Note that "cost" must be nonnegative. The "eval" should be a lower
 bound on cost in order for the first answer found to be guaranteed
 optimal, but the right answer will be reached eventually otherwise.

 These routines are used to quickly obtain a good initial solution
 for annealing. The search is to find the shortest weighted path
 through the edge dual graph (midpoints of boundary edges only).

 At conclusion of processing:
      no. of items in agenda - 1 = no. of unexamined states
      no. of items in usedstate = no. examined states

 Modifications to original code:

 In this version, cost is changed to cost(Statelist,OldCost,NewCost)
 to reduce the work required to compute the cost of a new statelist.
 Also, we have replaced pick_best_path rules with a more efficient
 version which uses Quintus Prolog hashing on the first atomic
 argument. The agenda_index is a sorted list of indices into agenda.
 Sorting is accomplished by insertion (since large heaps are not
 efficient in prolog).

 The original code referenced in [Rowe88, pp. 244-247] states that if
 the evaluation function is ALWAYS a lower bound on the cost function,
 then faster performance results if we delete the first rule of
 agenda_check and usedstate_check, and the entire definitions of
 fix_agenda and replace_front. We have determined that the first
 rule of agenda_check IS REQUIRED to guarantee optimality in a
 general graph (ie. NOT a tree). Therefore, we have NOT excluded it.
 Since our evaluation function is Euclidean distance to the goal at
 lowest cost on the map, then it is a lower bound on cost.
 We have, therefore, modified the original code accordingly.
======================================================================*/
```

```
:- dynamic
      max_e/1,
      agendacount/1,
      agenda_index/1,
      usedstate/2,
      agenda/5.

/*---------------------------------------------------------------------
 successor, goalreached, eval, and cost functions CANNOT have cuts !
---------------------------------------------------------------------*/

successor(E0,E1) :- ( ep(E0,E1,_,_,_) ; ep(E1,E0,_,_,_) ).

goalreached(g).

eval(E0,EVAL) :-% direct Euclidean dist to goal at lowest cost
      goal(GP,_),
      edge_data(E0,_,MP),
      optimal_cost(OC),
      wdistance(MP,GP,OC,EVAL).

cost([s],_,0.0).
cost([E0,E1|L],OLDCOST,COST) :-
      ( ep(E0,E1,_,_,C) ; ep(E1,E0,_,_,C) ),
      COST is OLDCOST+C.

%---------------------------------------------------------------------

search(WS,Cost) :-
      unknown(_,fail),                              % required by astarsearch
      writeln('Begin A* search for initial WS...'),
      astarsearch(s,WS,Cost),
      write_list(['Initial WS =',WS]),
      write_list(['Cost thru midpoints =',Cost]),
      unknown(_,trace),                             % reset after astarsearch
      w_clear_iter,                                                % DISPLAY
      !.
search(_,_) :- writeln_b('ERROR (astar)...A* search failed !'), !.

astarsearch(Start,Goalpathlist,C) :-
      cleandatabase,
      add_state(Start,[],0.0),
      repeatifagenda,
      pick_best_state(I,State,Pathlist,C,D),
      add_successors(I,State,Pathlist,C),
      agenda(I,State,Goalpathlist,C,_),           % soln on top of agenda
      w_show_ws(Goalpathlist),                                     % DISPLAY
      !.

pick_best_state(I,State,Pathlist,C,D) :-
      agenda_index([[I,_]|_]),
      agenda(I,State,Pathlist,C,D),
      !.

add_successors(_,State,Pathlist,_) :- goalreached(State), !.
add_successors(_,State,Pathlist,C) :-
      successor(State,Newstate),
      add_state(Newstate,Pathlist,C),
      fail.
add_successors(I,State,Pathlist,C) :-
      retract_agenda(I,State,Pathlist,C,D),
      asserta(usedstate(State,C)),
      fail.

add_state(Newstate,Pathlist,C) :-
      cost([Newstate|Pathlist],C,Cnew),
      !,
      agenda_check(Newstate,Cnew),
      !,
      usedstate_check(Newstate,Pathlist,Cnew),
      !,
      eval(Newstate,Enew),
      D is Enew + Cnew,
      assert_agenda(Newstate,[Newstate|Pathlist],Cnew,D),
      update_clock(T),
      max_eval(Enew),
      display_newstate(Newstate,Pathlist),            % DISPLAY newstate
      tm,
      !.
add_state(Newstate,Pathlist,C) :-
      not(cost([Newstate|Pathlist],C,_)),
      write_list_b([
                'ERROR (astar)...cost fn failure',[Newstate|Pathlist]]),
      !.
add_state(Newstate,Pathlist) :-
      not(eval(Newstate,Enew)),
      write_list_b([
                'ERROR (astar)...eval fn failure',[Newstate|Pathlist]]),
      !.

display_newstate(E0,[E1|_]) :-                                  % DISPLAY
      edge_data(E0,_,MP0),
      edge_data(E1,_,MP1),
      w_show_line(MP0,MP1,2),
      !.
display_newstate(_,[]) :- !.                           % for start state

agenda_check(S,C) :-
      agenda(I,S,_,C2,_),
      C<C2,
      retract_agenda(I,S,_,_,_),
      !.
```

```
agenda_check(S,_) :- agenda(_,S,_,_,_), !, fail.
agenda_check(_,_).

usedstate_check(S,_,_) :- usedstate(S,_), !, fail.
usedstate_check(_,_,_).

repeatifagenda.
repeatifagenda :- agenda(_,_,_,_,_), repeatifagenda.

cleandatabase :-
      retractall(agenda_index(_)),
      asserta(agenda_index([])),
      retractall(agendacount(_)),
      asserta(agendacount(0)),
      retractall(agenda(_,_,_,_,_)),
      retractall(usedstate(_,_)),
      retractall(max_e(_)),
      asserta(max_e(0)).

/*----------------------------------------------------------------------
 routines to maintain a sorted index into the agenda
----------------------------------------------------------------------*/

assert_agenda(State,Pathlist,C,D) :-
      get_index(I),
      asserta(agenda(I,State,Pathlist,C,D)),
      retract(agenda_index(L1)),
      insert_index([I,D],L1,L2),
      asserta(agenda_index(L2)),
      !.

insert_index(ID,[],[ID]).
insert_index([I,D],[[I0,D0]|L],[[I,D],[I0,D0]|L]) :- D =< D0.
insert_index(ID,[ID0|L1],[ID0|L2]) :- insert_index(ID,L1,L2).

delete_index(I,[],[]) :- write_list_b(['ERROR...delete_index',I]).
delete_index(I,[[I,_]|L],L).
delete_index(I,[ID|L1],[ID|L2]) :- delete_index(I,L1,L2).

retract_agenda(I,State,Pathlist,C,D) :-
      retract(agenda(I,State,Pathlist,C,D)),
      retract(agenda_index(L1)),
      delete_index(I,L1,L2),
      asserta(agenda_index(L2)),
      !.

get_index(I) :-
      retract(agendacount(N)),
      I is N+1,
      asserta(agendacount(I)),
      !.
```

```
/*======================================================================
 ELLIPSE.PL - intersect all edges with bounding ellipse computed from
      weighted length (D) of best path found to date; update edge data
======================================================================*/

:- dynamic
      ecost/1,
      optimal_cost/1.

optimal_cost(1.0).                    % default optimal cost coefficient

%-----------------------------------------------------------------------

ellipse(D) :-
      asserta(ecost(D)),
      w_showdata(elcost,D),
      get_optimal_cost(OW),
      write_list(['optimal_cost =',OW]),
      start(S,_),
      goal(G,_),
      edge(E,[V1,V2],_,_,_),
      total_dist(S,G,V1,OW,D1),
      total_dist(S,G,V2,OW,D2),
      intersect_ellipse(E,D,OW,S,G,V1,D1,V2,D2),
      fail, !.
ellipse(_) :- writeln('Computed bounding ellipse.'), !.

get_optimal_cost(OW) :- optimal_cost(OW), !.         % ONLY the top cost

total_dist(S,G,V0,OW,DT) :-
      wdistance(S,V0,OW,DS),
      wdistance(G,V0,OW,DG),
      DT is DS+DG,
      !.

intersect_ellipse(_,D,_,_,_,_,D1,_,D2) :-         % both vertices inside
      D1 =< D,                                               % V1 inside
      D2 =< D,                                               % V2 inside
      !.
intersect_ellipse(E,D,OW,S,G,V1,D1,V2,D2) :-         % edge crosses once
      D1 < D,                                                % V1 inside
      D2 > D,                                               % V2 outside
      elsearch(D,OW,S,G,V1,V2,D2,NV2,LEN),
      update_db(E,V1,NV2,LEN),
      asserta(ovr(NV2,0)),
      retractall(ve(V2,_)),
      w_point(NV2,4),
      !.
```

```
intersect_ellipse(E,D,OW,S,G,V1,D1,V2,D2) :-          % edge crosses once
      D1 > D,                                                 % V1 outside
      D2 < D,                                                  % V2 inside
      elsearch(D,OW,S,G,V2,V1,D1,NV1,LEN),
      update_db(E,NV1,V2,LEN),
      asserta(ovr(NV1,0)),
      retractall(ve(V1,_)),
      w_point(NV1,4),
      !.
intersect_ellipse(E,D,OW,S,G,V1,D1,V2,D2) :-         % edge crosses twice
      vtan_inside(E,D,S,G,VTAN),
      elsearch(D,OW,S,G,VTAN,V1,D1,NV1,LEN1),
      elsearch(D,OW,S,G,VTAN,V2,D2,NV2,LEN2),
      LEN is LEN1 + LEN2,
      update_db(E,NV1,NV2,LEN),
      asserta(ovr(NV1,0)),
      asserta(ovr(NV2,0)),
      retractall(ve(V1,_)),
      retractall(ve(V2,_)),
      w_point(NV1,4),
      w_point(NV2,4),
      !.
intersect_ellipse(E,_,_,_,_,_,_,_,_) :-             % edge outside ellipse
      remove_from_db(E),
      !.

vtan_inside(E,D,S,G,VTAN) :-
      edge(E,[V1,V2],_,L,_),
      vtangent(E,L,S,G,VTAN),                      % virtual tangent point
      between(V1,VTAN,V2),                     % must be between V1 and V2
      get_optimal_cost(W),
      total_dist(S,G,VTAN,W,DVT),
      DVT =< D,                                   % vtan is inside ellipse
      !.

vtangent(E,L,S,G,VTAN) :-
      perpendicular(S,L,PS),
      perpendicular(G,L,PG),
      distance(S,PS,DPS),                           % perpendicular from s
      distance(G,PG,DPG),                           % perpendicular from g
      PART is DPS/DPG,                % CAUTION no denominator zero check!
      partition(PART,[PS,PG],VTAN),                 % by similar triangles
      !.

update_db(E,V1,V2,LEN) :-
      retract(edge(E,VOLD,W,L,H)),
      asserta(edge(E,[V1,V2],W,L,H)),
      retract(edge_data(E,_,_)),
      midpoint(V1,V2,MP),
      assert(edge_data(E,LEN,MP)),
      obstacle_adjust(E,VOLD,W),
      !.

obstacle_adjust(E,[V1,_],[_,0]) :- ovr(V1,0), !.% already done
obstacle_adjust(E,[V1,_],[_,0]) :-
      ovr(V1,R),
      re(R,_,ELST),
      merge_with_ellipse_bound(ELST),
      !.
obstacle_adjust(_,_,_) :- !.

merge_with_ellipse_bound([]) :- !.
merge_with_ellipse_bound([E|ELST]) :-
      adjust_ovr(E),
      merge_with_ellipse_bound(ELST),
      !.

adjust_ovr(E) :-
      edge(E,[V1,V2],_,_,_),
      retractall(ovr(V1,_)),
      retractall(ovr(V2,_)),
      asserta(ovr(V1,0)),          % region 0 considered outside ellipse
      asserta(ovr(V2,0)),
      !.
adjust_ovr(_) :- !.% in case edge has already been removed

remove_from_db(E) :-
      retract(edge(E,[V1,V2],_,_,_)),
      retractall(edge_data(E,_,_)),
      retractall(er(E,_,_)),
      retractall(ep(E,_,_,_,_)),
      retractall(ep(_,E,_,_,_)),
      retractall(ve(V1,_)),
      retractall(ve(V2,_)),
      retractall(link(E,_,_,_)),
      retractall(link(_,_,_,E)),
      !.
```

```
/*======================================================================
 SPR.PL - solve for local optimal path through ws by relaxation;
      relaxation requires Snell's Law Table loaded by optimal.pl
======================================================================*/

:- dynamic
      iter_limit/1,        % iteration limit (prevents infinite looping)
      stable/0,                           % stability indicator (switch)
      st/1.                                        % stability tolerance

st(0.1).

/*----------------------------------------------------------------------
 NOTE: solve guarantees ws and path are ordered consistently in parallel
----------------------------------------------------------------------*/

solve(WS0,[P0|PLST],OPLST,COST) :-
      ensure_parallel(P0,WS0,WS),
      make_wtlst(WS,WL),                      % create WL - weight list
      compute_iter_limit,
      ipass(WS,WL,[P0|PLST],OPLST),       % iterate thru WS until stable
      which_direction(OPLST,WS,WL,TWL),  % orient TWL = either WL or RWL
      compute_cost(OPLST,TWL,COST),        % solution cost (path length)
      retractall(iter_limit(_)),
      !.

compute_iter_limit :-
      di(DI),
      st(ST),
      F is DI/ST,
      ceiling(F,IL),
      asserta(iter_limit(IL)),
      !.

ensure_parallel(P0,[s|WSg],[s|WSg]) :- start(P0,_), !.
ensure_parallel(P0,[g|WSs],[g|WSs]) :- goal(P0,_), !.
ensure_parallel(_,WS,RWS) :- rev(WS,RWS), !.

make_wtlst([s|WS],[SW|WT]) :-
      start(_,SR),
      rv(SR,SW,_),
      make_wtlst2(WS,WT,SW),
      !.
make_wtlst([g|WS],[GW|WT]) :-
      goal(_,GR),
      rv(GR,GW,_),
      make_wtlst2(WS,WT,GW),
      !.
```

```
make_wtlst2([_],[],_) :- !.                 % start or goal (end of list)
make_wtlst2([E1,E2|WS],[W1|WT],W0) :-
      edge(E1,_,W,_,_),
      next_wt(E1,W0,W,W1,E2),                  % rule contained in udp.pl
      make_wtlst2([E2|WS],WT,W1),
      !.

ipass([E0|WS],WL,[P0|PLST0],OPLST) :-        % first pass, do not skip 1
      w_show_iter([P0|PLST0],1),
      assert(stable),
      pass2(WS,WL,[P0|PLST0],PLST2),
      pass_chk(0,[E0|WS],WL,[P0|PLST2],OPLST),        % change direction
      w_clear_iter,
      !.

pass(I,[E0,E1|WS],[W0|WL],[P0,P1|PLST0],OPLST) :-
      w_show_iter([P0,P1|PLST0],1),
      assert(stable),
      pass2(WS,WL,[P1|PLST0],PLST2),
      pass_chk(I,[E0,E1|WS],[W0|WL],[P0,P1|PLST2],OPLST),
      !.

/*------------------------------------------------------------------------
 NOTE: pass2 ensures proper reflection; no stability check is made
      by this rule since reflections are exact per critical angle
------------------------------------------------------------------------*/

pass2([_],[_],[_,P0],[P0]) :- !.
pass2([E1,E1,E2|WS],[W1,W2|WL],[P0,P1,P2,P3|PLST],[NEWP1,NEWP2|PLST2]):-
      W1 > W2,
      reflect(P0,E1,PR1,PR2),
      reflect(P3,E1,PR3,PR4),
      closest_pair(PR1,PR2,PR3,PR4,TPA,TPB),
      refl_crossover(P0,TPA,P3,TPB,TTPA,TTPB),
      vertex_check(E1,TTPA,TTPB,NEWP1,NEWP2),
      pass2([E2|WS],WL,[P2,P3|PLST],PLST2),
      !.
pass2([E1,E1,E2|WS],[W1,W2|WL],[P0,_,_,P3|PLST],[P0,P0|PLST2]) :-
      pass2([E2|WS],WL,[P0,P3|PLST],PLST2),           % ignore reentrant
      !.
pass2([E1|WS],[W0,W1|WL],[P0,P1,P2|PLST],[NEWP1|PLST2]) :-    % CROSSING
      optimal(P0,P2,[W0,W1],E1,NEWP1),
      stable_check(P0,P1,P2,NEWP1),
      pass2(WS,[W1|WL],[P1,P2|PLST],PLST2),
      !.

pass_chk(I,WS,WL,PLST,OPLST) :-
      not(retract(stable)),                               % if NOT stable,
      iter_limit(IL),
      I < IL,                                       % and < iteration limit
      rev(PLST,RPLST),                              % then reverse PLST,
      rev(WS,RWS),
      rev(WL,RWL),
      J is I + 1,
      pass(J,RWS,RWL,RPLST,OPLST),                 % and continue w/ RWS
      !.
pass_chk(_,_,_,OPLST,OPLST) :-               % else stable, return OPLST
      w_show_iter(OPLST,2),                  % highlight stable solution
      !.

which_direction([P0|OPLST],[s|WSg],WL,WL) :- start(P0,_), !.
which_direction([P0|OPLST],[g|WSs],WL,WL) :- goal(P0,_), !.
which_direction(_,_,WL,RWL) :- rev(WL,RWL), !.         % reverse wt list

compute_cost([_],[],0.0) :- !.
compute_cost([P0,P1|OPLST],[W0|WL],TCOST) :-
      wdistance(P0,P1,W0,C0),
      compute_cost([P1|OPLST],WL,COST),
      TCOST is COST+C0,
      !.

%-----------------------------------------------------------------------

refl_crossover(P0,PA,P3,PB,MP,MP) :-           % if reflection crossover
      sqr_dist(P0,PA,SDA),
      sqr_dist(P0,PB,SDB),
      SDA > SDB,
      midpoint(PA,PB,MP),
      !.
refl_crossover(_,PA,_,PB,PA,PB) :- !.

stable_check(NP1,_,_,NP1) :- !.
stable_check(_,_,NP1,NP1) :- !.
stable_check(_,P1,_,NP1) :-
      not(stabilized(P1,NP1)),
      retract(stable),
      !.
stable_check(_,_,_,_) :- !.

stabilized([X0,Y0],[X1,Y1]) :-
      DX is X1-X0,
      DY is Y1-Y0,
      absolute(DX,AX),
      absolute(DY,AY),
      st(TOL),
      !,                        % DO NOT try another stability tolerance
      AX < TOL,
      AY < TOL.
```

```
vertex_check(E,TPA,TPB,PA,PB) :-
      edge(E,[V1,V2],_,_,_),
      v_chk(V1,V2,TPA,PA),
      v_chk(V1,V2,TPB,PB),
      !.

v_chk(V1,V2,TP,TP) :- between(V1,TP,V2), !.
v_chk(V1,V2,TP,V1) :- between(TP,V1,V2), !.
v_chk(V1,V2,TP,V2) :- between(V1,V2,TP), !.

/*======================================================================
 reflect - computes reflection points, critical angles, etc.
======================================================================*/

:-
      xload('lib/util'),
      xload('lib/line').

reflect(FP,ER,PR1,PR2) :-
      edge(ER,VR,[W1,W2],LR,_),
      crit_angle(W1,W2,THC),
      ray_line_angle(FP,LR,THC,P1,P2),
      endpt_check(P1,VR,PR1),
      endpt_check(P2,VR,PR2),
      !.

crit_angle(W1,W2,CRIT) :-
      W1 =< W2,
      R is W1/W2,
      asin(R,CRIT),
      !.
crit_angle(W1,W2,CRIT) :-
      R is W2/W1,
      asin(R,CRIT),
      !.

/*----------------------------------------------------------------------
 relative_angle - computes small relative angle between lines L1,L2
----------------------------------------------------------------------*/

relative_angle(L,L,GAMMA) :-
      fzero(GAMMA),
      writeln('WARNING (spr)...coincident edges.'),
      !.
relative_angle([A1,B1,_],[A2,B2,_],GAMMA) :-
      M1 is -A1/B1, M2 is -A2/B2,
      DEN is 1.0 + M1*M2,
      NUM is M1-M2,
      rel_ang2(NUM,DEN,GAMMA),
      !.
```

```
rel_ang2(NUM,DEN,+1.57080) :-
      fzero(DEN),
      NUM>=0.0,
      !.
rel_ang2(NUM,DEN,-1.57080) :-
      fzero(DEN),
      NUM< 0.0,
      !.
rel_ang2(NUM,DEN,GAMMA) :-
      TN is NUM/DEN,
      atan(TN,GAMMA),
      !.

fzero(0.0) :- !.
fzero(X) :-
      fabs(X,AX),
      AX < 0.0001,
      !.

/*----------------------------------------------------------------------
 ray_line_angle - computes left,right intersections P1,P2 for a ray
      from fixed point X0,Y0 at angle TH from normal to line [A,B,C]
----------------------------------------------------------------------*/

ray_line_angle(_,_,TH,_,_) :-                                % TH = pi/2
      DIF is TH-1.57080,
      fzero(DIF),
      writeln('WARNING (spr)...no intersect, parallel ray.'),
      !.
ray_line_angle(P,L,TH,I1,I1) :-                               % TH = 0.0
      fzero(TH),
      perpendicular(P,L,I1),
      !.
ray_line_angle([X0,Y0],[A,B,C],TH,[X1,Y1],[X2,Y2]) :-
      THI is 1.57080-TH,                  % convert to angle from normal
      tan(THI,TAN),
      L0 is A*X0 + B*Y0 - C,
      DEN is (A*A + B*B)*TAN,
      E0 is A*Y0 - B*X0,
      E1 is TAN/DEN,
      E2 is L0/DEN,
      R is (C*A - B*E0)*E1,
      S is (C*B + A*E0)*E1,
      T is A*E2,
      U is B*E2,
      X1 is R + U,
      Y1 is S - T,
      X2 is R - U,
      Y2 is S + T,
      !.
```

```
/*---------------------------------------------------------------------
 closest_pair - determines which pair of points on reflection edge
      are true points of reflection (they must be the closest pair).
---------------------------------------------------------------------*/

closest_pair([X1,Y1],[X2,Y2],[X3,Y3],[X4,Y4],[XI,YI],[XO,YO]) :-
      DXI is X3-X2, DXO is X4-X1,
      fabs(DXI,AXI), fabs(DXO,AXO),
      AXI > 0.0005, AXO > 0.0005,
      return_pair(AXI,AXO,
       [X1,Y1],[X2,Y2],[X3,Y3],[X4,Y4],[XI,YI],[XO,YO]),
      !.
closest_pair([X1,Y1],[X2,Y2],[X3,Y3],[X4,Y4],[XI,YI],[XO,YO]) :-
      DYI is Y3-Y2, DYO is Y4-Y1,
      fabs(DYI,AYI), fabs(DYO,AYO),
      AYI > 0.0005, AYO > 0.0005,
      return_pair(AYI,AYO,
       [X1,Y1],[X2,Y2],[X3,Y3],[X4,Y4],[XI,YI],[XO,YO]),
      !.

return_pair(AI,AO,[X1,Y1],_,_,[X4,Y4],[X1,Y1],[X4,Y4]) :- AI >= AO, !.
return_pair( _, _,_,[X2,Y2],[X3,Y3],_,[X2,Y2],[X3,Y3]) :- !.
```

```
/*=======================================================================
 OPTIMAL.PL - compute Snell's Law optimal crossing point by either
      table lookup or golden search (selected by C routine snell)
=======================================================================*/

:-
      xload('lib/util'),
      xload('lib/line'),
      xload('lib/foreign').               % also loads Snell's Law Table

optimal([X1,Y1],[X2,Y2],_,_,OP) :-
      DX is X2-X1, fabs(DX,AX), AX < 0.001,
      DY is Y2-Y1, fabs(DY,AY), AY < 0.001,
      midpoint([X1,Y1],[X2,Y2],OP),
      !.
optimal(P1,P2,WW,E,OP) :-               % WW are weights from weight list
      edge(E,V,W,L,_),                          % W are weights from edge
      line(P1,P2,LP),
      intersection(LP,L,ITMP),               % straight-line intersection
      optimal2(P1,P2,WW,[V,W,L],ITMP,OP),
      !.

optimal2(P1,P2,WW,[V,_,_],coincide,OP) :-          % parallel ==> P1-P2=E
      coincident_intersection(P1,P2,WW,I0),
      endpt_check(I0,V,OP),
      !.
optimal2(_,_,[W0,W0],[V,_,_],I0,OP) :-       % phantom edge I0=OP, unless
      endpt_check(I0,V,OP),                       % outside edge vertices
      !.
optimal2(P1,P2,WW,[V,W,L],I0,OP) :-            % else, setup table lookup
      match_regions(WW,W,P1,P2,PW1,PW2),          % PW* consistent with W
      perpendicular(PW2,L,I2),
      optimal3(PW1,PW2,V,W,L,I0,I2,OP),
      !.

coincident_intersection(P1,P2,[W1,W2],P2) :- W1 < W2, !.
coincident_intersection(P1,P2,[W1,W2],P1) :- W2 < W1, !.
coincident_intersection(P1,P2,[W0,W0],OP) :- midpoint(P1,P2,OP), !.

optimal3(_,_,V,_,_,I0,I2,OP) :-
      vertex_crossing(V,I0,I2,OP),
      !.
optimal3(PW1,PW2,V,W,L,I0,I2,OP) :-
      perpendicular(PW1,L,I1),
      snell(PW1,PW2,W,I1,I0,I2,OP0),
      endpt_check(OP0,V,OP),
      !.
```

```
vertex_crossing([V1,V2],I0,I2,V1) :-      % ensure point between vertices
      between(I0,V1,V2),
      between(I2,V1,V2),
      !.
vertex_crossing([V1,V2],I0,I2,V2) :-
      between(V1,V2,I0),
      between(V1,V2,I2),
      !.

match_regions(W,W,P1,P2,P1,P2).
match_regions(_,_,P1,P2,P2,P1).

endpt_check(I,[V1,V2],V2) :- between(V1,V2,I), !.
endpt_check(I,[V1,V2],V1) :- between(I,V1,V2), !.
endpt_check(I,[ _, _],I) :- !.
```

```
/*======================================================================
 UDP.PL - approximates locally optimal path through a WS using
      Dijkstra's Algorithm through a constrained graph of discrete
      points evenly distributed on each edge with upper bound di;
      used for coarse approximations, and as a preprocessor to
      the more accurate single-path relaxation (SPR) algorithm.
======================================================================*/

:- dynamic
      di/1,                        % di(i).current discrete interval = i
      epts/3,                % epts(e0,di,[p1,..,pn]). edge-points on e0
      pe/2.                         % pe(e0,[[v1,s1,p1],..,[vn,sn,pn]]).
                                                  % edge-point record e0

:- use_module(library(lists),[rev/2]).

di(8).                                       % default discrete interval

/*----------------------------------------------------------------------
 dijkstra - path returned is REVERSED with respect to WS edges
----------------------------------------------------------------------*/

dijkstra([_,_],[SP,GP],COST) :-                       % trivial soln s-g
      start(SP,SR),
      re(SR,SW,_),
      goal(GP,_),
      wdistance(SP,GP,SW,COST),
      !.
dijkstra([E0,E1|L],PATH,COST) :-                    % ws begins E0 = s;g
      start_goal(E0,P0,W0),
      edge(E1,_,W,_,_),
      di(INT),
      epts(E1,INT,PTS1),
      process_routes(W0,[[P0,0,0]],PTS1,PTLST1),
      asserta(pe(E1,PTLST1)),
      next_wt(_,W0,W,WN,_),
      !,
      dijkstra2([E1|L],WN,PATH,COST),
      w_clear_iter,
      !.

%-----------------------------------------------------------------------

dijkstra2([E1,E0],_,[P0|PATH],D) :-                    % ws end E0 = s;g
      pe(E1,PTLST1),
      start_goal(E0,P0,W0),
      process_routes(W0,PTLST1,[P0],[[P0,D,BG]]),
      back_pointer(P0,[[P0,D,BG]],BP),
      w_show_line(P0,BP,4),
      retrieve_min_path(BP,PATH),
      !.
dijkstra2([E1,E2,E3|L],W1,PATH,D) :-                            % mid ws
      pe(E1,PTLST1),
      edge(E2,_,W,_,_),
      di(INT),
      epts(E2,INT,PTS2),
      process_routes(W1,PTLST1,PTS2,PTLST2),
      asserta(pe(E2,PTLST2)),
      next_wt(E2,W1,W,WN,E3),
      !,
      dijkstra2([E2,E3|L],WN,PATH,D),
      !.
dijkstra2(WS,W,_,_) :-                            % catastrophic failure
      current(_,WS0,_),
      retract(next(WS1)),
      writeln('WS ERROR (udp)...dijkstra2 failed !'),
      write_list(['weight: ',W]),
      write_list(['partial: ',WS]),
      write_list(['current: ',WS0]),
      write_list(['next: ',WS1]),
      writeln('Corrective action...continuing from current WS0...'),
      !, fail, !.

start_goal(s,P,W) :- start(P,R), re(R,W,_), !.
start_goal(g,P,W) :- goal(P,R), re(R,W,_), !.

%-----------------------------------------------------------------------

back_pointer(P,[[P,_,BP]|_],BP) :- !.
back_pointer(P,[_|L],BP) :- back_pointer(P,L,BP), !.

retrieve_min_path(SP,[SP]) :- start(SP,_), !.
retrieve_min_path(GP,[GP]) :- goal(GP,_), !.
retrieve_min_path(P,[P|PATH]) :-
      retract(pe(_,PTLST)),
      back_pointer(P,PTLST,BP),
      w_show_line(P,BP,4),
      retrieve_min_path(BP,PATH),
      !.

/*----------------------------------------------------------------------
 process_routes - establishes all routes between an edge-pair;
      W1 = weight of the region separating the edges
      PTLST1 = list of point records [V,S,P] for the "from" edge
      PTS2 = list of points on the "to" edge
      PTLST2 = new list of point-records [V,S,P] for the "to" edge
      min_route finds shortest accumulated distance S and the pointer
      P back to the appropriate "from" edge point-record labeled V.
----------------------------------------------------------------------*/
```

```
process_routes(_,_,[],[]) :- !.
process_routes(W1,PTLST1,[V|L],[[V,S,P]|PTLST2]) :-
      min_route(W1,PTLST1,V,_,[S,P]),
      process_routes(W1,PTLST1,L,PTLST2),
      !.

next_wt(_,W0,[W0,W2],W2,_) :- !.
next_wt(_,W0,[W1,W0],W1,_) :- !.
next_wt(EC,_,chk,WN,EA) :-        % EC choke vertex ==> look ahead at EA
      chr(EC,VC,_),
      er(EA,RR,_),
      next_wt2(RR,VC,WN),
      !.

next_wt2([R1,_],VC,WN) :- rv(R1,WN,VLST), member(VC,VLST), !.
next_wt2([_,R2],VC,WN) :- rv(R2,WN,VLST), member(VC,VLST), !.

/*-----------------------------------------------------------------------
 min_route - updates the point record [D,P] for point V by computing
 weighted distance from each point in list L (points on the last
      edge in the window sequence) and returning the min total weighted
      distance D and the point P from which it was measured to V.
-----------------------------------------------------------------------*/

min_route(_,[],_,[D,P],[D,P]) :- !.
min_route(W,[[V,D,_]|L],V0,[D0,P0],[DF,PF]) :-
      wdistance(V,V0,W,DV),
      w_show_line(V0,V,1),
      DT is D + DV,
      update_point([D0,P0],[DT,V],[D1,P1]),
      min_route(W,L,V0,[D1,P1],[DF,PF]),
      !.

update_point([D,_],[DT,V],[DT,V]) :- ( var(D) ; DT<D ), !.
update_point([D,P],_,[D,P]) :- !.

/*-----------------------------------------------------------------------
 get_pts - uses a fixed interval (di) to space points uniformly across
      each edge; this interval is an upper bound on the point spacing,
      ie. spacings on some edges may be < di (never greater than).
-----------------------------------------------------------------------*/

epts(E,I,PLST) :- gen_epts(E,I,PLST), !.     % non-exist ==> generate pe

gen_epts(E,INT,PLST) :-
      edge(E,V,_,_,_),
      get_pts(E,V,PLST),
      asserta(epts(E,INT,PLST)),
      !.
```

```
reset_points :-                            % for subsequent program runs
      retractall(edge_pt(_,_)),
      retractall(epts(_,_,_)),
      asserta(( epts(E,I,PLST) :- gen_epts(E,I,PLST) )),
      !.

get_pts(_,[V0,V0],[V0]) :- !.                             % choke vertex
get_pts(E,[V1,V2],[NV1|L]) :-                  % fixed interval size INT
      edge_data(E,LENG,_),
      LENG > 0.0,                              % in case V1,V2 too close
      di(INT),
      R_INTS is LENG/INT,
      ceil(R_INTS,INTS),            % di(INT) LUB on dist to next point
      PART is 0.001/INTS,                % pull in slightly from vertex
      partition(PART,[V1,V2],NV1),
%     w_point(NV1,3),                     % show point spacing/coverage
      divisions(INTS,V1,V2,L),
      !.
get_pts(_,[V1,_],[V1]) :- !.             % V1,V2 too close for precision

divisions(INTS,V1,V2,[NV2]) :-
      INTS =< 1,
      PART is 0.999/INTS,                % pull in slightly from vertex
      partition(PART,[V1,V2],NV2),
%     w_point(NV2,3),                     % show point spacing/coverage
      !.
divisions(INTS,V1,V2,[P0|L]) :-
      PART is 1/INTS,
      partition(PART,[V1,V2],P0),
%     w_point(P0,3),                      % show point spacing/coverage
      NX_INTS is INTS-1,
      divisions(NX_INTS,P0,V2,L),
      !.

all_epts :-                                   % generate all edge-points
      edge(E,V,[_,W],_,_),
      W =\= 0,
      get_pts(E,V,L),
      fail,
      !.
all_epts :- !.
```

```
/*======================================================================
 PERTURB.PL - move generator; generates random WS transitions
      (vertex/obstacle rotation and reentrant installation)
======================================================================*/

:- dynamic
      single_jump/0,                       % flag to set single rotation
      lock_path_class/0.                % flag to lock between obstacles

%-----------------------------------------------------------------------

perturb_ws([S,G],WS1) :-                % [s,g] or [g,s] ==> same region
      repeat,                                      % until [S,G] changes
      select_type(TYPE),
      special_sg_move(TYPE,[S,G],WS1),
      !.
perturb_ws(WS0,WS3) :-
      length(WS0,LEN),                                   % compute range
      RANGE is 2*LEN-5,
      repeat,                             % until move from WS0 is found
      select_edge_which(RANGE,EDGE,WHICH),
      sp_nth0(EDGE,WS0,E0,C1,C2),         % NTH edge is E0 between C1,C2
      select_type(TYPE),
      sub_selections(TYPE,WHICH,E0,[C1,C2,LOC],ITEM),
      take_action(TYPE,ITEM,LOC,WS0,WS1),
      cycle_chk(WS1,WS2),
      fix_ws(WS2,WS3),                       % strip out dumb turn edges
%     ws_integrity(WS0,WS,WS1,WS1),                 % for debugging only
      !.

select_type(TYPE) :-
      irand(2,X),
      ( X=:=0 -> TYPE=0                                      % reentrant
      ; otherwise -> TYPE=1 ),                            % vertex cross
      !.

%-----------------------------------------------------------------------

special_sg_move(0,[S,G],[S,E0,E0,G]) :-              % special reentrant
      start(_,R0),
      re(R0,W0,ELST),
      select(ELST,E0),
      not(edge(E0,_,[_,0],_,_)),            % no border or obstacle edge
      !.
special_sg_move(1,[S,G],WS1) :-                   % special vertex cross
      start(_,R0),
      rv(R0,_,VLST),
      select(VLST,V0),
      border(BRD),
      not(ovr(V0,BRD)),                    % cannot jump a border vertex
      ve(V0,VELST),
      re(R0,_,RELST),
      set_intersect(RELST,VELST,EE),
      split_ee(EE,VELST,[S,G],WS1),
      !.

set_intersect([],A,[]).
set_intersect([A|B],C,D) :-
      memberchk(A,C),
      !,
      D=[A|E],
      set_intersect(B,C,E).
set_intersect([A|B],C,D) :- set_intersect(B,C,D).

split_ee([E1,E2],VELST,[S,G],NEWSEQ) :-   % shift E1,E2 to ends of VELST
      shift_list(E1,VELST,[E1,E|LST]),
      ( E==E2 -> append([E|LST],[E1],SUB)
      ; SUB=[E1,E|LST] ),
      irand(2,X),
      ( X=:=0 -> append([s|SUB],[g],NEWSEQ)     % insert SUB between s,g
      ; X=:=1 -> append([g|SUB],[s],NEWSEQ) ),
      !.

%-----------------------------------------------------------------------

select_edge_which(RANGE,EDGE,WHICH) :-
      irand(RANGE,J),
      I is J+2,                                          % NOT at s or g
      EDGE is I//2,                                          % pick edge
      WHICH is I mod 2,                 % lf/rt vertex or fwd/bck region
      !.

sub_selections(0,WHICH,E0,[C1,C2,LOC],ER) :-            % reentrant move
      ( WHICH=:=0 -> er(E0,[R0,_],_)                      % which region
      ; WHICH=:=1 -> er(E0,[_,R0],_) ),
      re(R0,_,ELST),
      ( memberchk(C1,ELST) -> LOC=[C1,E0]
      ; memberchk(C2,ELST) -> LOC=[E0,C2] ),        % failure ==> s or g
      subtract(ELST,LOC,RELST),                       % remove E0 and C*
      select(RELST,ER),                    % select ER from region elist
      !.
sub_selections(0,_,E0,_,V0) :-             % failed reentrant ==> vertex
      irand(2,WHICH),
      sub_selections(1,WHICH,E0,_,V0),
      !.
sub_selections(1,WHICH,E0,_,V0) :-                         % vertex move
      ( WHICH=:=0 -> edge(E0,[V0,_],_,_,_)             % select V1 or V2
      ; WHICH=:=1 -> edge(E0,[_,V0],_,_,_) ),
      !.
```

```
select(L,ITEM) :-                       % select random ITEM from list L
      length(L,LEN),
      irand(LEN,X),
      nth0(X,L,ITEM),
      !.                                           % do NOT select again

sp_nth0(N1,[A,B,C|WS],E0,C1,C2) :-
      ( N1=:=1 -> E0=B, C1=A, C2=C
      ; N0 is N1-1,
       sp_nth0(N0,[B,C|WS],E0,C1,C2) ),
      !.

%-----------------------------------------------------------------------

take_action(0,ER,LOC,WS0,WS1) :-                             % reentrant
      possible_reentrant(ER,LOC,WS0),            % legit reetrant path ?
      reentrant_install(ER,LOC,WS0,WS1),
      !.
take_action(1,V0,_,WS0,WS1) :-                  % single vertex rotation
      single_jump,                               % flag for single jumps
      get_elst(V0,ELST),
      move_ws(WS0,ELST,WS1,_),
      !.
take_action(1,V0,_,WS0,WS1) :-                  % double vertex rotation
      get_elst(V0,ELST),
      move_ws(WS0,ELST,WSA,E),
      second_move(E,V0,WSA,WS1),
      !.

second_move(null,_,WS,WS) :- !.
second_move(E,V0,WSA,WS1) :-
      new_vertex(E,V0,V1),
      get_elst(V1,ELST2),
      move_ws(WSA,ELST2,WS1,_),
      !.

new_vertex(E,V0,V1) :- edge(E,[V0,V1],_,_,_), !.
new_vertex(E,V0,V1) :- edge(E,[V1,V0],_,_,_), !.

%-----------------------------------------------------------------------

possible_reentrant(ER,[E1,E2],WS0) :-
      ( ep(E1,ER,_,W0,_) ; ep(ER,E1,_,W0,_) ),                % exists ?
      ( ep(E2,ER,_,W0,_) ; ep(ER,E2,_,W0,_) ),
      not(consec_reentrants([E1,E1,E2],WS0)),   % consecutive reentrants
      not(consec_reentrants([E1,E2,E2],WS0)),
      !.
```

```
consec_reentrants([A,B,C],[A,B,C,D|LIST]) :- !.
consec_reentrants(SUBLST,[A,B,C|LIST]) :-
      consec_reentrants(SUBLST,[B,C|LIST]),
      !.

reentrant_install(ER,[E1,E2],[E1,E2|WS0],[E1,ER,ER,E2|WS0]) :- !.
reentrant_install(ER,LOC,[E3,E4|WS0],[E3,E4|WS1]) :-
      reentrant_install(ER,LOC,[E4|WS0],[E4|WS1]),
      !.

%-----------------------------------------------------------------------

get_elst(V0,ELST) :-
      ovr(V0,R),                                    % if obstacle vertex
      !,
      not(lock_path_class),                   % NO lock on path subclass
      not(border(R)),                                  % NO border jumps
      not(ovr(V0,0)),                           % NO ellipse bound jumps
      ( chr(E,V0,_), che(E,ELST)                          % choke vertex
      ; oe(R,ELST) ),                          % regular obstacle vertex
      !.
get_elst(V0,ELST) :- ve(V0,ELST), !.% else non-obstacle vertex

%-----------------------------------------------------------------------

move_ws(WS0,ELST,WS1,E) :-
      slice(WS0,ELST,WF,S1,S2,WB),              % extract critical edges
      make_new_seq(ELST,S1,S2,NEWSEQ),     % S1=[EF,E1,R1] S2=[EB,E2,R2]
      append(WF,NEWSEQ,TEMP),
      append(TEMP,WB,WS1),
      ( NEWSEQ = [E]
      | NEWSEQ = [] -> E = null
      | otherwise -> select(NEWSEQ,E) ),
      !.

slice(WS0,ELST,WF,S1,S2,WB) :-
      extract(WS0,ELST,WF,S1),                       % from front of WS0
      rev(WS0,RWS0),
      extract(RWS0,ELST,RWB,S2),                      % from back of WS0
      rev(RWB,WB),
      !.

extract([A,B,B|WS],ELST,[A],[A,B,1]) :- memberchk(B,ELST), !.% reentrant
extract([A,B |WS],ELST,[A],[A,B,0]) :- memberchk(B,ELST), !.  % crossing
extract([A,B|WS],ELST,[A|WF],[EF,E1,R]) :-
      extract([B|WS],ELST,WF,[EF,E1,R]),
      !.

make_new_seq(ELST,[EF,E1,_],[EB,E2,_],SEQ) :-% shifts E1 to front
      shift_list(E1,ELST,[_|OLST]),
      split_list(OLST,E1,E2,L1,L2),
      order_list(EF,EB,E1,E2,L1,L2,SUB),
      back_list(EB,E2,SUB,SEQ),
      !.

back_list(_,_,[],[]) :- !.
back_list(EB,E2,SUB,NEWSEQ) :-
      last(FB,SUB),
      ( continuous(FB,EB) -> NEWSEQ=SUB
      ; cross_continuous(FB,E2,EB) -> append(SUB,[E2],NEWSEQ) ),
      !.

special_conditions(_,_,E,E,_) :- !, fail.
special_conditions(EF,EB,_,_,F) :-
      continuous(EF,EB),
      not(continuous(EF,F)),
      not(continuous(EB,F)),
      !.

order_list(EF,EB,E1,E2,[],[F|_],[]) :-
      special_conditions(EF,EB,E1,E2,F),
      !.
order_list(EF,EB,E1,E2,[F|_],[],[]) :-
      special_conditions(EF,EB,E1,E2,F),
      !.
order_list(EF,EB,E1,E2,L1,L2,SUB) :-
      first_elt(L1,F1),
      first_elt(L2,F2),
      ( continuous(EF,F1) -> SUB=L1
      ; continuous(EF,F2) -> SUB=L2
      ; continuous(E1,F1) -> SUB=[E1|L1]
      ; continuous(E1,F2) -> SUB=[E1|L2]
      ; rev(L1,RL1),
        rev(L2,RL2),
        order_list(EF,EB,E1,E2,RL1,RL2,SUB) ),
      !.
order_list(EF,EB,E1,E2,L1,L2,_) :-
      writeln_b('ERROR...(perturb) order_list failed !'),
      writeln_b([EF,EB,E1,E2,L1,L2]),
      abort.

first_elt([],null).
first_elt([F|_],F).

split_list(OLST,E1,E1,OLST,ROLST) :-                % if E1==E2 (single)
      rev(OLST,ROLST),
      !.
split_list(OLST,_,E2,L1,L2) :- break_list(OLST,E2,L1,L2) ,!.
```

```
break_list([E|OLST],E2,L1,L2) :-
      ( E==E2 -> L1=[], L2=OLST
      ; L1=[E|LL], break_list(OLST,E2,LL,L2) ),
      !.

continuous(E1,E2) :- ( ep(E1,E2,_,_,_) ; ep(E2,E1,_,_,_) ), !.

cross_continuous(E1,E2,E3) :-                                 % crossing
      (ep(E1,E2,R1,_,_) ; ep(E2,E1,R1,_,_)),
      (ep(E2,E3,R3,_,_) ; ep(E3,E2,R3,_,_)),
      not(same(R1,R3)),
      !.

%------------------------------------------------------------------------

cycle_chk(WS1,WS2) :-
      cycle_clear(WS1,WS2),
%     irand(2,X),
%     ( X=:=0 -> all_cycle_clear(WS1,WS2)
%     ; otherwise -> cycle_clear(WS1,WS2) ),
      !.

cycle_clear(WS1,WS2) :-
      triple(WS1,T),
      save_front(T,WS1,FRONT,BACK),
      rev(BACK,RBACK),
      save_front(T,RBACK,RRBACK,_),
      rev(RRBACK,BAK),
      close_gap(FRONT,BAK,T,WS2),
      !.
cycle_clear(WSL,WSL) :- !.

triple([_,_],_) :- fail.
triple([A|L],A) :- double(L,A), !.
triple([A|L],B) :- triple(L,B), !.

double([_],_) :- fail.
double([A|L],A) :- memberchk(A,L), !.
double([A|L],B) :- double(L,B), !.

save_front(T,[T|WS1],[],WS1) :- !.
save_front(T,[E|WS1],[E|WS2],WSB) :- save_front(T,WS1,WS2,WSB), !.

close_gap(FRONT,[E1|BAK],T,WS2) :-
      last(E0,FRONT),
      ( cross_continuous(E0,T,E1) -> append(FRONT,[T,E1|BAK],WS2)
      ; otherwise -> append(FRONT,[E1|BAK],WS2) ),
      !.

%------------------------------------------------------------------------
```

```
all_cycle_clear(WS,WS1) :-        % removes ALL cycles except reentrants
      find_first_dup(WS,E),
      remove_cycle(E,WS,WS1),
      !.
all_cycle_clear(WS,WS) :- !.

find_first_dup([],_) :- fail.
find_first_dup([A,A|B],C) :- not(member(A,B)), find_first_dup(B,C), !.
find_first_dup([A|B],A) :- member(A,B), !.
find_first_dup([A|B],C) :- find_first_dup(B,C), !.

remove_cycle(_,[],[]) :- !.
remove_cycle(E,[E|WST],[E|WS1]) :- remove_cycle2(E,WST,WS1), !.
remove_cycle(E,[E1|WST],[E1|WS1]) :- remove_cycle(E,WST,WS1), !.

remove_cycle2(_,[],[]) :- !.                           % pinch off cycle
remove_cycle2(E,[E|WST],[E|WST]) :- !.
remove_cycle2(E,[E1|WST],WS1) :- remove_cycle2(E,WST,WS1), !.

%-----------------------------------------------------------------------

ws_integrity(WS0,WSC,WS1,[E1,E]) :- continuous(E,E1), !.        % end ws
ws_integrity(WS0,WSC,WS1,[E1,E2,E2,E3|WS]) :-                % reentrant
      (ep(E1,E2,R,_,_) ; ep(E2,E1,R,_,_)),
      (ep(E2,E3,R,_,_) ; ep(E3,E2,R,_,_)),
      ws_integrity(WS0,WSC,WS1,[E2,E3|WS]),
      !.
ws_integrity(WS0,WSC,WS1,[E1,E2,E3|WS]) :-                    % crossing
      cross_continuous(E1,E2,E3),
      ws_integrity(WS0,WSC,WS1,[E2,E3|WS]),
      !.
ws_integrity(WS0,WSC,WS1,[E1,E2,E3|_]) :-
      write_list(['ERROR (perturb)...Invalid WS at',[E1,E2,E3]]),
      start(S,_),
      goal(G,_),
      write_list(['start =',S,' goal =',G]),
      write_list(['WS0 =',WS0]),
      write_list(['WSC =',WSC]),
      write_list(['WS1 =',WS1]),
      writeln('Taking corrective action...returning to WS0.'),
      fail.

fix_ws([S|WS],CW) :- correct_ws(S,[S|WS],CW), !.

correct_ws(E0,[E1],[E1]) :-
      (ep(E1,E0,R,_,_) ; ep(E0,E1,R,_,_)),
      !.
correct_ws(E1,[E2,E2,E3|WS],Z) :-
      (ep(E1,E2,R,W,_) ; ep(E2,E1,R,W,_)),
      (ep(E2,E3,R,W,_) ; ep(E3,E2,R,W,_)),
      ( edge(E2,_,[_,W],_,_) -> EN=E2, Z=[E2,E2|CW]
      ; otherwise -> EN=E1, Z=CW ),                          % strip out
      correct_ws(EN,[E3|WS],CW),
      !.
correct_ws(E1,[E2,E2,E3|WS],CW) :-                    % faulty reentrant
      (ep(E1,E2,R1,_,_) ; ep(E2,E1,R1,_,_)),
      (ep(E2,E3,R2,_,_) ; ep(E3,E2,R2,_,_)),
      correct_ws(E1,[E2,E3|WS],CW),
      !.
correct_ws(E0,[E1,E2|WS],CW) :-                              % remove E1
      (ep(E0,E1,R,_,_) ; ep(E1,E0,R,_,_)),
      (ep(E1,E2,R,_,_) ; ep(E2,E1,R,_,_)),
      correct_ws(E0,[E2|WS],CW),
      !.
correct_ws(_,[E1,E2|WS],[E1|CW]) :-
      correct_ws(E1,[E2|WS],CW),
      !.
correct_ws(_,WS,CW) :-
      writeln('ERROR (perturb)...failed to correct WS !'),
      write_list(['WS =',WS]),
      write_list(['CW =',CW]),
      fail, !.
```

B.2 C INTERFACE AND C CODE

```
/*=========================================================================
 FOREIGN.PL - Prolog to C interface routines
=========================================================================*/

distance([X1,Y1],[X2,Y2],D) :-
distance( X1,Y1, X2,Y2, D), !.

wdistance([X1,Y1],[X2,Y2],W,WD) :-
wdistance( X1,Y1, X2,Y2, W,WD), !.

snell([PX1,PY1],[PX2,PY2],[W1,W2],
      [IX1,IY1],[IX0,IY0],[IX2,IY2],[XOP,YOP]) :-
snellc(PX1,PY1, PX2,PY2, W1,W2,
       IX1,IY1,  IX0,IY0,  IX2,IY2,  XOP,YOP), !.

elsearch(D,OW,[SX,SY],[GX,GY],[V1X,V1Y],[V2X,V2Y],D2,[X,Y],LEN) :-
      elsearch(D,OW, SX,SY, GX,GY, V1X,V1Y, V2X,V2Y, D2, X,Y, LEN), !.

foreign_file('/n/ai9/work/kindl/sa/lib/heap.o', [
      init_heap,
      del_min,
      del_heap,
      ins_heap
      ]).

foreign_file('/n/ai9/work/kindl/sa/lib/snells_law.o', [
      getable,
      distance,
      wdistance,
      snellc,
      elsearch
      ]).

foreign_file('/n/ai9/work/kindl/sa/lib/rnd.o', [
      irand,
      randinit,
      init_anneal,
      metrop,
      iter_impr,
      total_ws
      ]).
```

```
/*----------------------------------------------------------------------
 heap
----------------------------------------------------------------------*/

foreign(init_heap, c, init_heap([-integer])).
foreign(ins_heap, c, ins_heap(+float,[-integer])).
foreign(del_min, c, del_min([-integer])).
foreign(del_heap, c, del_heap(+integer)).

/*----------------------------------------------------------------------
 snellc calls require that getable has been called to load tabledump;
      getable accesses tabledump directly in the current directory
----------------------------------------------------------------------*/

foreign(getable,c,getable([-integer])).
foreign(distance,c,distance(+float,+float,+float,+float,[-float])).
foreign(wdistance,c,
      wdistance(+float,+float,+float,+float,+float,[-float])).
foreign(snellc,c,snellc(+float,+float,+float,+float,+float,+float,
      +float,+float,+float,+float,+float,+float,-float,-float)).
foreign(elsearch,c,elsearch(+float,+float,+float,+float,+float,+float,
      +float,+float,+float,+float,+float,-float,-float,[-float])).

/*----------------------------------------------------------------------
 Random generators must NOT be cut (!);
 may require re-execution by backtrack.
 randinit calls require access to seed5
 (random digits generated by last run).
----------------------------------------------------------------------*/

foreign(irand,c,irand(+integer,[-integer])).
foreign(randinit,c,randinit([-integer])).
foreign(init_anneal,c,
      init_anneal(+float,+float,+float,+integer,+integer)).
foreign(metrop,c,metrop(+float,+float,[-integer])).
foreign(iter_impr,c,iter_impr(+float,+float,[-integer])).
foreign(total_ws,c,total_ws([-integer])).

/*----------------------------------------------------------------------
 INITIALIZATION
----------------------------------------------------------------------*/
:-
      load_foreign_files( [
         '/n/ai9/work/kindl/sa/lib/heap.o',
         '/n/ai9/work/kindl/sa/lib/snells_law.o',
         '/n/ai9/work/kindl/sa/lib/rnd.o'
          ],['-lm'] ),                                   % lm = lib math
      abolish([foreign_file/2,foreign/3]),
      getable(_).                               % load Snell's Law Table
                                % tabledump MUST be in current directory
```

```
/*======================================================================
 GLOBAL.C - C definitions - determine table size and range
======================================================================*/

#include <stdio.h>
#include <math.h>

/*----------------------------------------------------------------------
if P,Y,R are adjusted, then revise PSIZE,YSIZE,RSIZE for ocp[][][];
also, remember to recompile with revised global.c
----------------------------------------------------------------------*/

#define TOLERANCE 0.000001

#define PLO 1
#define PHI 10

#define YLO 0.00
#define YHI 1.00
#define YINC 0.05

#define RLO 0.00
#define RHI 10.00
#define RINC 0.05

#define PSIZE 45
#define YSIZE 21
#define RSIZE 201

float ocp[PSIZE][YSIZE][RSIZE], ***ptbl = 0;
/* NOTE: ptbl=0 ==> table was NOT loaded */

int pidx[PHI][PHI], **px;

int i_lim = PHI-PLO+1;
int j_lim = 1.01 + (YHI-YLO)/YINC;
int k_lim = 1.01 + (RHI-RLO)/RINC;

FILE *fp, *fopen();

/*--------------------------------------------------------------------*/
```

```
/*=======================================================================
 SNELLS_LAW.C - table lookup or golden search for Snell's Law crossing
=======================================================================*/

#include "global.c"

/*-----------------------------------------------------------------------
 GETABLE - initialize table of x values by loading table core dump
-----------------------------------------------------------------------*/

getable(){

      int *pd = &(pidx[0][0]);
      float *oc = &(ocp[0][0][0]);
      float **occ = &oc;
      FILE *infile = fopen("tabledump","r");

read(fileno(infile),pidx,sizeof(pidx));
read(fileno(infile),ocp,sizeof(ocp));

ptbl = &occ;
px = &pd;

return;

}

/*-----------------------------------------------------------------------
 XLOOKUP - linear interpolation into Snell's Law table
      x=f(p,y,r) ==> interpolation of ix=f(i,j,k); table index p is
      obtained indirectly from 2D triangular table indexed by iw1,iw2
-----------------------------------------------------------------------*/

double
xlookup(iw1,iw2,y,r)

      int iw1,iw2;
      double y,r;
{
      int i,j,k;
      double x,jy,kr,rfactor,yfactor,dxr1,dxr2,dx1,dx2;

/* get table index i for appropriate rho = iw2/iw1
 by convention this routine ASSUMES that iw2>iw1 !!! */

i = pidx[iw2-1][iw1-1];
/* compute indices of square interval for interpolation */

jy = (y-YLO)/YINC; kr = (r-RLO)/RINC;
j = (int)jy; k = (int)kr;/* safer way to floor */

/* compute orthogonal distances from lower corner of square */

yfactor = jy-(float)j; rfactor = kr-(float)k;

/* planar interpolation (or linear in 2 dimensions) */

dxr1 = ocp[i][j][k+1] - ocp[i][j][k];
dxr2 = ocp[i][j+1][k+1] - ocp[i][j+1][k];
dx1 = ocp[i][j][k] + dxr1 * rfactor;
dx2 = ocp[i][j+1][k] + dxr2 * rfactor;
x = dx1 + (dx2-dx1) * yfactor;

if (ptbl==0) printf("WARNING...Snell's Law table is NOT loaded!!!\n");

return(x);

}     /* end XLOOKUP */


/*-------------------------------------------------------------------------
 WDISTANCE - weighted Euclidean distance between 2 points
 DISTANCE - UNweighted                                                   "
 WSQRDIST - weighted, squared"
 SQRDIST - UNweighted, squared"
------------------------------------------------------------------------*/

double
wdistance(x1,y1,x2,y2,w)double x1,y1,x2,y2,w;
{ return(w * sqrt( (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1) ) ); }

double
distance(x1,y1,x2,y2)                                  double x1,y1,x2,y2;
{ return( sqrt( (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1) ) ); }

double
sqrdist(x1,y1,x2,y2)                                   double x1,y1,x2,y2;
{ return( (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1) ); }


double
wsqrdist(x1,y1,x2,y2,w)                              double x1,y1,x2,y2,w;
{ return(w*w * ( (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1) ) ); }
```

```
/*------------------------------------------------------------------------
 GOLDEN - golden ratio search for minimum
------------------------------------------------------------------------*/

#define R 0.61803399
#define C (1.0-R)
#define SHFT(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);

golden(px1,py1,px2,py2,w1,w2,vx1,vy1,vx2,vy2,xop,yop)

      float px1,py1,px2,py2,w1,w2,vx1,vy1,vx2,vy2,*xop,*yop;
{
      float ff,f0,f1,f2,f3;
      float dx,x0,x1,x2,x3;
      float dy,y0,y1,y2,y3;

x0 = vx1; y0 = vy1;
x3 = vx2; y3 = vy2;

dx = vx2-vx1; dy = vy2-vy1;

x1 = C*dx+vx1; y1 = C*dy+vy1;
x2 = R*dx+vx1; y2 = R*dy+vy1;

f1 = wdistance(px1,py1,x1,y1,w1) + wdistance(px2,py2,x1,y1,w2);
f2 = wdistance(px1,py1,x2,y2,w1) + wdistance(px2,py2,x2,y2,w2);

while (fabs(x3-x0) > TOLERANCE*(fabs(x1)+fabs(x2))
      || fabs(y3-y0) > TOLERANCE*(fabs(y1)+fabs(y2))) {

 if (f2 < f1) {
   SHFT(x0,x1,x2,R*x1+C*x3);
   SHFT(y0,y1,y2,R*y1+C*y3);
   ff = wdistance(px1,py1,x2,y2,w1) + wdistance(px2,py2,x2,y2,w2);
   SHFT(f0,f1,f2,ff);
 }
 else {
   SHFT(x3,x2,x1,R*x2+C*x0);
   SHFT(y3,y2,y1,R*y2+C*y0);
   ff = wdistance(px1,py1,x1,y1,w1) + wdistance(px2,py2,x1,y1,w2);
   SHFT(f3,f2,f1,ff);
 }
}

*xop = 0.5*(x1+x2);
*yop = 0.5*(y1+y2);

return;

} /* end GOLDEN */
```

```
/*--------------------------------------------------------------------------
 SNELLC - computes Snell's Law crossing point (xop,yop)
--------------------------------------------------------------------------*/

snellc(px1,py1,px2,py2,w1,w2,ix1,iy1,ix0,iy0,ix2,iy2,xop,yop)

      float     px1,py1,px2,py2,w1,w2,ix1,iy1,ix0,iy0,ix2,iy2,*xop,*yop;
{
      int                                                         iw1,iw2;
      double                                             x,xt,y,yt,r,part;

iw1 = nint(w1); iw2 = nint(w2);

if (iw1 == iw2) {/* should NEVER be used, computed in Prolog */
 *xop = ix0;
 *yop = iy0;
 return(-1);                          /* -1 = NO computation performed */
}

xt = distance(ix1,iy1,ix2,iy2);
y = distance(px1,py1,ix1,iy1);
yt = y + distance(px2,py2,ix2,iy2);
y = y/yt;                                      /* compute normalized y */
r = xt/yt;

if (y>0.01 && y<YHI && r>0.01 && r<RHI) {/* check if in table */

 if (iw1 > iw2)                           /* check correct orientation */
 x = 1.0 - xlookup(iw2,iw1,1.0-y,r);
 else
 x = xlookup(iw1,iw2,y,r);
 part = x/r;                 /* optimal partition of edge between i1,i2 */

 *xop = part*(ix2-ix1) + ix1;
 *yop = part*(iy2-iy1) + iy1;

 return(1);                          /* 1 = table lookup was performed */
 }

else {
 golden(px1,py1,px2,py2,w1,w2,ix0,iy0,ix2,iy2,xop,yop);
 return(0);                          /* 0 = golden search was required */
 }

}     /* end SNELLC */
```

```
/*-------------------------------------------------------------------------
 ELSEARCH - binary search for intersection of ellipse and line (edge)
-------------------------------------------------------------------------*/

float
elsearch(bestc,ow,sx,sy,gx,gy,xin,yin,xout,yout,cout, x0, y0)

      float bestc,ow,sx,sy,gx,gy,xin,yin,xout,yout,cout,*x0,*y0;
{
      float x,y,c,x1,y1,tol=0.001;

x1=xin; y1=yin;
c=cout;

while (fabs(c-bestc) > tol) {

 x = 0.5*(xin+xout);
 y = 0.5*(yin+yout);
 c = wdistance(sx,sy,x,y,ow) + wdistance(gx,gy,x,y,ow);

 if (c > bestc) { xout = x; yout = y; }
 else { xin = x; yin = y; }

 }

*x0=x; *y0=y;

return( distance(x1,y1,x,y) );                    /* new length of edge */

} /* END elsearch */
```

```
/*=======================================================================
 RND.C - annealing primitives and random number generation routines
=======================================================================*/

#include <stdio.h>
#include <math.h>
#include "state1.h"                          /* random list of numbers */

#define DIGIT 4
#define MAXINT 0x7FFFFFFF

FILE *fp, *fopen();                         /* file pointers for seed5 */
unsigned seed = 1;
float fmaxint = (float)MAXINT;

/*----------------------------------------------------------------------
 Annealing parameters (global):
 Define change in cost function as: dc = c[i] - c[i+1]
      Then dc > 0.0 implies cost improvement, accept at p = 1.0
      and dc < 0.0 implies cost degradation, accept at p = exp(dc/t)
 Thus, function metrop(dc,t) should operate as follows:
      return( (dc >= 0.0) || (rand01() < exp(dc/t)) ? 1:0 )
----------------------------------------------------------------------*/

float tfactor;                         /* temp reduction factor (eg .9)*/
float temp;                                      /* current temperature*/
float tfrozen;                                   /* current temperature*/

int n_total = 0;                         /* total attempted moves all t*/

/* The following stats are accumulated at each temperature temp*/

int n_limit;                            /* max attempted moves (eg 100)*/
int ns_limit;                            /* max successful moves(eg 10)*/
int n = 0;                                           /* attempted moves*/
int ns = 0;                                         /* successful moves*/

int n_imp = 0;                           /* attempted moves @ dc >= 0.0*/
                                                   /* n_deg = n - n_imp*/

float sum_dc_deg = 0.0;                          /* sum of dc degrading*/
float mean_dc_deg;                      /* <dc degrading> = exp val dc-*/

float sum_c = 0.0;                                          /* sum of c*/
float ss = 0.0;                                           /* sum of c^2*/
float mean;                                   /* <c> = expected value c*/
float sd;                                         /* std deviation of c*/
```

```
/*----------------------------------------------------------------------
 IRAND(K) - generate random integer in range [0..K)
----------------------------------------------------------------------*/

int
irand(k) int k; {

if (k>0) return(random() % k);
else printf("ERROR...attempt to call irand(k) for k<=0\n");
} /* END irand */

/*----------------------------------------------------------------------
 INITSEED() - generate a new seed from the system clock
----------------------------------------------------------------------*/

unsigned
initseed() {

      unsigned t,curtime,loctime,save,oldsave;

/*
MUST call time(0) (ie. arg = NIL pointer) to ensure that time
does NOT store return structure on top of program variable space.
*/

curtime = (unsigned)time(0);        /* no of seconds since 1 Jan 1970 */

if (fp = fopen("seed5","r")) {
 fscanf(fp,"%d",&oldsave);
 fclose(fp);
 }
else
 oldsave = 2311;                           /* some large prime number */

t = (unsigned)exp10((float)DIGIT);
loctime = curtime % t;                           /* last DIGIT digits */
seed = loctime*oldsave + loctime + 13;
save = seed % t;                                 /* last DIGIT digits */
fp = fopen("seed5","w");
fprintf(fp,"%d",save);
fclose(fp);

printf("\ncurtime = %12d\n",curtime);
printf( "loctime = %12d\n",loctime);
printf( "t = %12d\n",t);
printf( "oldsave = %12d\n",oldsave);
printf( "(new)save = %12d\n",save);
printf( "seed = %12d\n\n",seed);

return(seed);
} /* END initseed */
```

```
/*---------------------------------------------------------------------
 RANDINIT() - initialize new random seed and state1 for random()
---------------------------------------------------------------------*/

unsigned
randinit() {

      int n = 256;                /* total no. bytes in state1 = 4*64 */

initseed();                                        /* to change seed */
initstate(seed,state1,n);             /* to initiate state with seed */
setstate(state1);                                   /* to use state1 */

/* srandom(1);             /* generates SAME sequence each execution */
srandom(seed);              /* generates NEW sequence each execution */

return(seed);
} /* END randinit */

/*---------------------------------------------------------------------
 GENSTATE() - generate a new state1
---------------------------------------------------------------------*/

genstate() {

      int i, size = 64;

randinit();
fp = fopen("state1.h","w");
fprintf(fp,"static long state1[64] = {\n");

for (i=1; i<size; i++)
 fprintf(fp,"%15d,\n",irand(1000000000));

fprintf(fp,"%15d };\n",irand(1000000000));
fclose(fp);
} /* END genstate */

/*--------------------------------------------------------------------
 rand01() - uses random() to generate random number on [0,1)
--------------------------------------------------------------------*/

float
rand01()

{ return( (float)random() / fmaxint ); }

/* to extend range returned by rand01, use this call:
 return((float)random() / (float)0x7FFFFFFF * range); */
```

```
/*=====================================================================
 ANNEAL.C - simulated annealing control routines and statistics
=====================================================================*/

/*---------------------------------------------------------------------
 init_anneal() - initialize annealing parameters
---------------------------------------------------------------------*/

init_anneal(ts,t0,tfac,n0,ns0)

      float ts, t0, tfac;
      int n0, ns0;
{

temp = ts;
tfrozen = t0;
tfactor = tfac;
n_limit = n0;
ns_limit = ns0;
n = 0;
ns = 0;
n_total = 0;

printf("\n");
printf(" starting temp = %8.2f\n",ts);
printf(" freezing temp = %8.2f\n",t0);
printf(" temp reduction factor = %8.2f\n",tfac);
printf("max perturbations per temp = %8d\n",n0);
printf(" max successes per temp = %8d\n",ns0);
printf("\n");
printf(" temp n ns dc+ dc- <dc-> <c> sd cum\n");
printf("-------------------------------------------------------------");
printf("\n");
}/* END init_anneal */

/*---------------------------------------------------------------------
 cool() - reduce the temperature (temp) per annealing schedule;
      may be modified for more elaborate cooling schemes
---------------------------------------------------------------------*/

cool() {

      int den;

den = n - n_imp;
if (den == 0) mean_dc_deg = 0.0;
else mean_dc_deg = sum_dc_deg/(float)den;
sum_dc_deg = 0.0;
mean = sum_c/(float)n;
sd = sqrt( (ss/(float)(n-1))-(mean*sum_c/(float)(n-1)) );
sum_c = 0.0;
ss = 0.0;

n_total = n_total + n;

printf("%7.1f %3d %3d %3d %3d %8.1f %7.1f %7.1f %4d\n",
      temp,n,ns,n_imp,n-n_imp,mean_dc_deg,mean,sd,n_total);

temp = tfactor * temp;
n = 0;
n_imp = 0;

if (ns == 0 || temp <= tfrozen) {
 printf(" freezing temp = %8.2f\n",temp);
 printf("total perturbations evaluated = %8d\n",n_total);
 return(0);                              /* HALT - effectively frozen */
 }
else {
 ns = 0;
 return(1);                                     /* continue annealing */
 }
} /* END cool */
/*---------------------------------------------------------------------
 metrop(c1,c2) - returns 1-accept/0-reject or -1 if frozen
      c1 = current cost;                                   c2 = new cost;
---------------------------------------------------------------------*/

metrop(c1,c2) float c1,c2; {

      int determ;                 /* determinant = accept 1 or reject 0 */
      int improve;                              /* 1 if dc non-negative */
      float dc;

dc = c1-c2;
n++;
sum_c = sum_c + c2;
ss = ss + (c2*c2);
improve = (dc > 0.0) ? 1:0;

if (improve) n_imp++;                              /* n_deg = n - n_imp */
else sum_dc_deg = sum_dc_deg + dc;

determ = ( improve || (rand01() < exp(dc/temp)) ) ? 1:0;
ns = ns + determ;

if (n >= n_limit || ns >= ns_limit) {
 if (cool()) return(determ);                    /* continue annealing */
 else { n=0; return(-1); }               /* HALT - effectively frozen */
 }

return(determ);
} /* END metrop */
```

```
/*------------------------------------------------------------------------
 iter_impr(c1,c2) - iterative improvement; returns 1-improves/0-reject
      c1 = current cost; c2 = new cost;
------------------------------------------------------------------------*/

iter_impr(c1,c2) float c1,c2; {

n++;
if (n <= n_limit) return( (c1-c2 >= 0.0) ? 1:0 );
else { n=0; return(-1); }

} /* END iter_impr */

/*------------------------------------------------------------------------
 total_ws() - returns total number of window sequences evaluated
------------------------------------------------------------------------*/

total_ws() { return(n_total); }
```

```
/*=======================================================================
 HEAP.C - routines to operate a heap data structure
=======================================================================*/

#include <math.h>

#define NULL 0
#define MAXHEAP 100000

struct heap_item { int cost, hash; };
struct heap_item heap[MAXHEAP];
int i, hashcount = 0, heap_size = 0;

/*-----------------------------------------------------------------------
 INIT_HEAP
-----------------------------------------------------------------------*/

init_heap() {

heap_size = 0;
hashcount = 0;
for (i=0;i<MAXHEAP;i++) {
 heap[i].cost = -1;
 heap[i].hash = -1;
 };

} /* END INIT_HEAP */

/*-----------------------------------------------------------------------
 SIFT_UP - heap routine
-----------------------------------------------------------------------*/

sift_up(k) int k; {

      int parent;

if (k > heap_size)
 printf("ERROR...Attempted to sift_up an element not in heap!\n");
while (k > 1) {
 parent = floor((double)k/(double)2);
 if (heap[parent].cost > heap[k].cost) {
   interchange(parent,k);
   k = parent;
   }
 else return(NULL);
 };

} /* end SIFT_UP */
```

```
/*------------------------------------------------------------------------
 INS_HEAP - insert into heap
------------------------------------------------------------------------*/

ins_heap(cost) int cost; {

      int hash;

hashcount++;
heap_size++;
hash = hashcount;

if (heap_size >= MAXHEAP)
 printf("ERROR...Over-extended heap!");

heap[heap_size].hash = hash;
heap[heap_size].cost = cost;
if (heap_size > 1) sift_up(heap_size);
return(hash);

} /* END INS_HEAP */

/*------------------------------------------------------------------------
 SIFT_DOWN - heap routine
------------------------------------------------------------------------*/

sift_down(k) int k; {

      int j;

if (k > heap_size+1) {              /* k might be the old last element */
 printf("ERROR...Attempted to sift_down element not in the heap!\n");
 printf("k = %d heap_size = %d\n",k,heap_size);
 }

do {
 j = k;
 if (2*j <= heap_size && heap[2*j].cost < heap[k].cost)
 k = 2*j;
 if (2*j < heap_size && heap[2*j+1].cost < heap[k].cost)
 k = 2*j+1;
 interchange(k,j);
} while (j != k);

} /* end SIFT_DOWN */
```

```
/*----------------------------------------------------------------------
 DEL_MIN - returns top of heap (minimum) and restores heap property
----------------------------------------------------------------------*/

del_min() {

      int root;

if (heap_size > 0) {
 root = heap[1].hash;
 interchange(1,heap_size);
 heap[heap_size].hash = NULL;
 heap[heap_size].cost = NULL;
 heap_size--;
 if (heap_size > 1) sift_down(1);
 return(root);
 }

else {
 printf("ERROR...Attempt to delete from empty heap!\n");
 return(NULL);
 };

} /* end DEL_MIN */

/*--------------------------------------------------------------------
 DEL_HEAP - delete from anywhere in heap; requires linear search
--------------------------------------------------------------------*/

del_heap(k) int k; {

      int i = 1;

if (heap_size > 0) {
 while ( (heap[i].hash != k) && (i < heap_size) ) i++;
 interchange(i,heap_size);
 heap[heap_size].hash = NULL;
 heap[heap_size].cost = NULL;
 heap_size--;                  /* heap[heap_size] is now last element */
 if (heap_size > 1) sift_down(i);
 }

else printf("ERROR...Attempt to delete from empty heap!\n");

} /* end DEL_HEAP */
```

```
/*--------------------------------------------------------------------
 INTERCHANGE - swap elements in heap
--------------------------------------------------------------------*/

interchange(parent,child) int parent,child; {

      struct heap_item hold;

hold = heap[parent];
heap[parent] = heap[child];
heap[child] = hold;

} /* end INTERCHANGE */
```

```
/*======================================================================
 TABLE.C - table generation, read, dump, get routine
======================================================================*/

#include "global.c"

/*----------------------------------------------------------------------
 GENTABLE.C - generate Snell's Law lookup table for crossing episodes
----------------------------------------------------------------------*/

gentable() {

      float v1,v2,r,p0,p,x,y,e1,e2;
      int i,j,k,m,n,i_lim;

fp = fopen("sldata0","w");

i_lim = PHI-PLO+1;
i_lim = (i_lim*i_lim - i_lim)/2;

i = -1;

for (m=PLO-1; m<PHI; m++)
 for (n=PLO-1; n<m; n++) {
   p0 = (float)(m+1)/(float)(n+1);
   i++;
   pidx[m][n] = i;
   fprintf(fp,"%d\n",pidx[m][n]);
     for (j=0; j<j_lim; j++) {
       y = (float)j*YINC+YLO;

       for (k=0; k<k_lim; k++) {
         r = (float)k*RINC+RLO;

         v1 = 0.0; v2 = r;
         e1 = 1.0; e2 = 1.0;

         while (e1>TOLERANCE && e2>TOLERANCE) {

           x = 0.5*(v1+v2);
           p = cos(atan(y/x))/cos(atan((1.0-y)/(r-x)));
           if (p<p0) v1 = x;
           else v2 = x;
           e1 = fabs(p0-p);
           e2 = fabs(v2-v1);

         } /* end while */
         fprintf(fp,"%8.6f\n",x);
         fflush(fp);

         } /* end for */
       } /* end for */
     } /* end for */

fclose(fp);

} /* end GENTABLE */

/*-------------------------------------------------------------------------
 READTABLE.C - read table in ascii format (sldata0)
-----------------------------------------------------------------------*/

readtable() {

      int m,n,i,j,k;

ptbl = (float ***) malloc((unsigned) j_lim*k_lim*sizeof(float**));
px = (int**) malloc((unsigned) (PHI-PLO+1)*sizeof(int*));

fp = fopen("sldata0","r");
printf("Loading Snell's Law table... ");
fflush(stdout);

for (m=PLO-1; m<PHI; m++) {
 for (n=PLO-1; n<m; n++) {
   fscanf(fp,"%d",&pidx[m][n]);
   i = pidx[m][n];
   for (j=0; j<j_lim; j++)
     for (k=0; k<k_lim; k++)
       fscanf(fp,"%f",&ocp[i][j][k]);
   }
 printf("%d ",m+1);
 fflush(stdout);
 }

printf(" ...done.\n");
fflush(stdout);
fclose(fp);
return;
}     /* end READTABLE */
```

```
/*----------------------------------------------------------------------
 DUMPTABLE.C - read 'sldata' in ascii, dump core image to 'tabledump'
----------------------------------------------------------------------*/

dumptable() {

      FILE *outfile = fopen("tabledump0","w");

readtable();                /* requires table in ascii form (sldata0) */

write(fileno(outfile),pidx,sizeof(pidx));
write(fileno(outfile),ocp,sizeof(ocp));
fflush(outfile);
return;

} /* end DUMPTABLE */

/*----------------------------------------------------------------------
 GETABLE.C - reads core dump of table (tabledump)
----------------------------------------------------------------------*/

getable() {

      int *pd = &(pidx[0][0]);
      float *oc = &(ocp[0][0][0]);
      float **occ = &oc;
      FILE *infile = fopen("tabledump0","r");

read(fileno(infile),pidx,sizeof(pidx));
read(fileno(infile),ocp,sizeof(ocp));

ptbl = &occ;
px = &pd;
return;

} /* end GETABLE */

/*--------------------------------------------------------------------*/

main() {

gentable();                               /* generate sldata0 (ascii) */
dumptable();      /* execute readtable() and dump tabledump0 (binary) */
getable();                           /* load to check for correctness */

} /* END main */
```

B.3 UTILITIES AND PRIMITIVES

```
/*========================================================================
 PROLOG.INI - makes custom version of Quintus Prolog
=========================================================================

 Run Quintus Prolog in your root with this file called prolog.ini and
 your regular prolog.ini file renamed to init.pl. After you receive
 the normal Q-Prolog opening message and prompt, enter 'make(name).'
 to save the state in executable form as 'name'. Henceforth, you must
 start Quintus Prolog by entering 'name' in Unix. This will rapidly
 load compiled versions of all libraries requested below, and give you
 the prolog prompt.
-----------------------------------------------------------------------*/

:-
      compile('lib/init'),          % contains root and xload definitions

      use_module(library(math),all), % append,member,memberchk,nonmember
      use_module(library(basics),all),
      use_module(library(lists),[
       last/2,
       rev/2,
       is_list/1,
       nth0/3,
       nth1/3,
       delete/3 ]),

      use_module(library(sets),[subtract/3,subset/2]),
      use_module(library(strings),[concat/3,concat_atom/3]),
      use_module(library(files),[file_exists/1]),
      use_module(library(ctypes),[is_endfile/1]),
      use_module(library(dialog)),
      use_module(library(interpret_messages)),
      use_module(library(messages)),

      xload('lib/util'),
      xload('lib/line'),
      xload('lib/pwin'),
      xload('lib/pre'),
      xload('lib/ellipse'),
      xload('lib/udp'),
      xload('lib/optimal'),
      xload('lib/spr'),
      xload('lib/foreign'),
      xload('lib/start'),
      xload('lib/mouse'),
      xload('lib/stats'),

      nl, writeln('Enter ''make(P).'' to create executable ''P''.').

make(P) :-
      save(P,1),         % remaining statements execute at start-up of P
      nl,
      writeln('Q-Prolog Rel 2.5 (SPARC)...custom vers 5 Nov 90').
make(P) :-
      nl, write('Created custom vers of Q-Prolog: '),
      writeln(P),
      halt.

make_exec(P) :- save(P,1), background.
make_exec(_) :- halt.

background :- file_exists('run.pl'), compile(run).% background
background :- start.                                % interactive
```

```
/*=======================================================================
 INIT.PL - primitive routines
=======================================================================*/

/*-----------------------------------------------------------------------
 Library pointers and file manipulation
-----------------------------------------------------------------------*/

library_directory('/usr/local/quintus/generic/qplib2.5/library').
library_directory('/usr/local/quintus/pw1.1/library').

root('/n/ai9/work/kindl/sa/').

xconsult(F) :- filepath(F,PATH), consult(PATH).

xload(F) :- filepath(F,PATH), ensure_loaded(PATH).

filepath(F,PATH) :-
      root(D),
      name(D,L1), name(F,L2),
      append(L1,L2,L),
      name(PATH,L).

/*-----------------------------------------------------------------------
 Primitives
-----------------------------------------------------------------------*/

not(X) :- \+ X.

% :- unknown(X,fail).            % call to unknown predicate simply fails
% :- unknown(X,trace).                       % (default) trap to debugger

% append([],L,L).
% append([X|L],R,[X|List]) :- append(L,R,List).

sys(STRING) :- unix(system(STRING)).

writeln(X) :- write(X), nl.
writelnln(X) :- write(X), nl, nl.
writeln(S,X) :- write(S,X), nl(S).
writelnln(S,X) :- write(S,X), nl(S), nl(S).

writeln_b(X) :- bold, write(X), nl, alloff, !.

write_list([]) :- nl, !.
write_list([X|L]) :- write(X), write(' '), write_list(L), !.

write_list_b(X) :- bold, write_list(X), alloff, !.
```

```
same(X,X).

cls :- display('^L').                              % clearscreen or page
nt :- notrace.                                          % shorthand code
ns :- nospyall.
l(X) :- listing(X).

dump(P) :-                           % dump listing of predicate to file
      open(P,write,S),
      tell(S),
      listing(P),
      told.

/*----------------------------------------------------------------------
 Enhancements to On-Line Help / Manual
----------------------------------------------------------------------*/

man :- manual.

m :- manual.
m(X) :- manual(X).
m(S,N) :- sect(S,SEC), manual(-(SEC,N)).
m(S,N,P) :- sect(S,SEC), manual(-(-(SEC,N),P)).
m(S,N,P,Q) :- sect(S,SEC), manual(-(-(-(SEC,N),P),Q)).

sect(u,user).
sect(r,ref).
sect(s,sysdep).
sect(l,lib).
sect(p,pw).

h(X) :-
      name(X,L),
      char_list(L,CL),
      append([m],CL,MLIST),
      M =.. MLIST,
      call(M),
      !.
h(_) :-
      writeln('ERROR...Invalid use of h(X)...'),
      writeln(' X must be form of ''sd'', ''sdd'', or ''sddd'''),
      writeln(' s = u(user), r(ref), s(sys), or l(lib)'),
      writeln(' d = digit 0..9 [if d>9, use form m(s,d,d,d)]').

char_list([],[]).
char_list([N|L],[C|CL]) :-
      name(C,[N]),
      char_list(L,CL).
```

```
/*======================================================================
 VT-100 Terminal escape sequences
======================================================================*/

eb :- esc, lbrack.                                % escape, left bracket
clear :- eb, two, uj.                                     % clear screen
home :- eb, uh.                                            % home cursor
reverse :- eb, ques, five, lh.                          % reverse screen
normal :- eb, ques, five, ll.                            % normal screen
smooth :- eb, ques, four, lh.                            % smooth scroll
scr_norm :- eb, ques, four, ll.                          % normal scroll
backup :- esc, um.                                       % reverse index
save_cur :- esc, seven.
rest_cur :- esc, eight.
cur(L,C) :- esc, uy, write(L), write(C).
alloff :- eb, zero, lm.                        % turn off all attributes
bold :- eb, one, lm.                                       % bold attrib
under :- eb, four, lm.                                % underline attrib
blink :- eb, five, lm.                                 % blinking attrib
rvideo :- eb, seven, lm.                          % reverse video attrib

/*----------------------------------------------------------------------
 ASCII display codes (used above)
----------------------------------------------------------------------*/

esc :- put(27).                                               % escape

zero :- put(48).                                    % numerical digits
one :- put(49).
two :- put(50).
three :- put(51).
four :- put(52).
five :- put(53).
six :- put(54).
seven :- put(55).
eight :- put(56).
nine :- put(57).

ques :- put(63).                                                   % ?
lbrack :- put(91).                                                 % [

la :- put( 97).                                      % lowercase alpha
lh :- put(104).
ll :- put(108).
lm :- put(109).

uh :- put(72).                                       % uppercase alpha
uj :- put(74).
um :- put(77).
uy :- put(89).
```

```
/*======================================================================
 UTIL.PL
======================================================================*/

irandom(A,B,IR) :-                 % random integer range A..B inclusive
      RNG is B-A+1,
      irand(RNG,N),
      IR is N+A,
      !.

/*----------------------------------------------------------------------
 timing - CPU sec
----------------------------------------------------------------------*/

timestart :- statistics(runtime,_), !.

timemark(TIME) :-
      statistics(runtime,[_,T]),
%     format('~3d sec CPU time~n',[T]),
      TIME is T/1000,
      !.

ts :- timestart.

tm :- timemark(_).

tm(T) :- timemark(T).

/*----------------------------------------------------------------------
 tolerance - between a pair of points
----------------------------------------------------------------------*/

:- dynamic tolerance/1.

tolerance(0.000001).            % default tolerance, change with asserta

within_tolerance([[X1,Y1],[X2,Y2]]) :-
      abs_diff(X1,X2,DX), abs_diff(Y1,Y2,DY),
      tolerance(Spec),
      !,                          % do NOT get another tolerance Spec */
      DX < Spec, DY < Spec.

abs_diff(X1,X2,AD) :- X1=<X2, AD is X2-X1, !.
abs_diff(X1,X2,AD) :- AD is X1-X2, !.
```

```
/*----------------------------------------------------------------------
 Constants/Conversions
----------------------------------------------------------------------*/

pi(3.14159).
half_pi(1.5708).

rad_deg(R,D) :-
      not(var(R)), var(D),
      D is R*180.0/3.14159,
      !.
rad_deg(R,D) :-
      var(R), not(var(D)),
      R is D*3.14159/180.0,
      !.
rad_deg(_,_) :- writeln('ERROR (util)...rad_deg needs a constant.'), !.

/*----------------------------------------------------------------------
 absolute - prolog type-independent absolute value
----------------------------------------------------------------------*/

absolute(X,X) :- X>=0, !.
absolute(X,AX) :- AX is -X, !.

/*----------------------------------------------------------------------
 miscellaneous routines - common to table lookup and search
----------------------------------------------------------------------*/

between([X1,Y1],[XM,YM],[X2,Y2]) :-
      number_between(X1,XM,X2),
      number_between(Y1,YM,Y2),
      !.

number_between(A,B,C) :- A=<B, B=<C, !.
number_between(A,B,C) :- A>=B, B>=C, !.
number_between(A,B,C) :-
      close_enough(A,B),
      close_enough(B,C),
      !.

close_enough(N1,N2) :-
      D is N1-N2,
      fabs(D,AD),
      AD<0.0001,
      !.

wtd_path(P1,P2,[W1,W2],P,C) :-
      wdistance(P1,P,W1,C1),
      wdistance(P2,P,W2,C2),
      C is C1+C2,
      !.
```

```
/*------------------------------------------------------------------------
 closest - determines which of {P1,P2} closest to ref point PR
------------------------------------------------------------------------*/

closest(PR,P1,P2,PC) :-
      sqr_dist(PR,P1,SD1),
      sqr_dist(PR,P2,SD2),
      nearer(P1,SD1,P2,SD2,PC),
      !.

sqr_dist([X1,Y1],[X2,Y2],SD) :-
      DY is Y2-Y1, DX is X2-X1,
      SD is DX*DX + DY*DY,
      !.

nearer(P1,SD1,_,SD2,P1) :- SD1=<SD2, !.
nearer(_,_,P2,_,P2) :- !.
```

```
/*======================================================================
 LINE.PL - geometric routines
======================================================================*/

:-
      ensure_loaded(library(math)),
      xload(’lib/util’).

intersect(P1,P2,P3,P4,I) :-
      line(P1,P2,ABC),
      line(P3,P4,DEF),
      intersection(ABC,DEF,I),
      !.

line([X,Y],[X,Y],[0,0,0]) :-                  % degenerate line = point
      write_list([’WARNING (line)...degenerate line at’,[X,Y]]),
      !.
line([X,_],[X,_],[1.0,0.0,X]) :- !.     % infinite slope line = vertical
line([_,Y],[_,Y],[0.0,1.0,Y]) :- !.       % zero slope line = horizontal
line([X1,Y1],[X2,Y2],[A,B,C]) :-
      DX is X2-X1,
      line2(DX,[X1,Y1],[X2,Y2],[A,B,C]),
      !.

line2(DX,[X1,_],[X2,_],[1.0,0.0,XM]) :-                  % near vertical
      fabs(DX,ADX),
      ADX < 0.001,
      XM is 0.5*(X1+X2),
      !.
line2(DX,[X1,Y1],[X2,Y2],[A,1.0,C]) :-
      A is (Y1-Y2)/DX,
      C is (Y1*X2-Y2*X1)/DX,
      !.

/*----------------------------------------------------------------------
 INTERSECTION CONDITIONS:
      DEN = B*D-A*E = 0 AND B*F-C*E = 0
      ==> coincident or degenerate lines

      DEN = B*D-A*E = 0 AND B*F-C*E /= 0
      ==> parallel lines
----------------------------------------------------------------------*/

intersection([A,B,C],[D,E,F],I) :-
      DEN is B*D-A*E,
      intersection2(DEN,[A,B,C],[D,E,F],I),
      !.
```

```
intersection2(DEN,[A,B,C],_,coincide) :-
      DEN =:= 0.0,                    % DEN requires explicit zero check
%     write_list(['WARNING (line)...coincident lines L =',[A,B,C]]),
      !.
intersection2(DEN,[A,B,C],[D,E,F],[X,Y]) :-
      X is (B*F-C*E)/DEN,
      Y is (A*F-C*D)/(-DEN),
      !.

midpoint([X1,Y1],[X2,Y2],[MX,MY]) :-
      MX is 0.5*(X1+X2),
      MY is 0.5*(Y1+Y2),
      !.

partition(P,[[X1,Y1],[X2,Y2]],[PX,PY]) :-
      PX is P*(X2-X1) + X1,
      PY is P*(Y2-Y1) + Y1,
      !.

perpendicular([PX,PY],[A,B,_],[PX,PY]) :-
      A =:= 0.0, B =:= 0.0,
      writeln('WARNING (line)...perpendicular to degenerate line'),
      !.
perpendicular([PX,PY],[A,B,C],[X,Y]) :-
      T is B*PX - A*PY,
      D is A*A + B*B,
      X is (A*C + B*T)/D,
      Y is (B*C - A*T)/D,
      !.

/*------------------------------------------------------------------------
 heading - returns bearing 0..359+ for ray from [X1,Y1] thru [X2,Y2]
------------------------------------------------------------------------*/

heading([X,Y],[X,Y],_) :-
      write_list(['WARN (line)...degen line',[X,Y],'has no heading']),
      !, fail.
heading([X1,Y1],[X2,Y2],H) :-
      Y2<Y1, X2<X1,
      heading2([X2,Y2],[X1,Y1],H2),
      H is H2+180.0,
      !.
heading([X1,Y1],[X2,Y2],H) :-
      Y2<Y1,
      heading2([X2,Y2],[X1,Y1],H2),
      H is 180.0-H2,
      !.
heading([X1,Y1],[X2,Y2],H) :-
      X2<X1,
      heading2([X2,Y2],[X1,Y1],H2),
      H is 360.0-H2,
      !.
heading([X1,Y1],[X2,Y2],H) :-
      heading2([X1,Y1],[X2,Y2],H),
      !.

heading2([X1,Y1],[X2,Y2],H) :-
      DX is X1-X2, DY is Y1-Y2,
      fabs(DX,ADX), fabs(DY,ADY),
      ADX>ADY,
      Q is ADY/ADX, atan(Q,AT),
      H is 90.0-(57.29577951 * AT),
      !.
heading2([X1,Y1],[X2,Y2],H) :-
      DX is X1-X2, DY is Y1-Y2,
      fabs(DX,ADX), fabs(DY,ADY),
      Q is ADX/ADY, atan(Q,AT),
      H is 57.29577951 * AT,
      !.

/*------------------------------------------------------------------------
 relative_heading - clockwise difference (0..359+) between 2 headings
 back_heading - used to link edges and find regions
------------------------------------------------------------------------*/

relative_heading(H1,H2,H3) :- H3 is H2-H1, H3>=0, !.
relative_heading(H1,H2,H3) :- H3 is 360-(H1-H2), !.

back_heading(H,HB) :- H < 180.0, HB is H+180.0, !.
back_heading(H,HB) :- HB is H-180.0, !.
```

B.4 USER INTERFACE

```
/*========================================================================
 START.PL - general start sequence and user interface
========================================================================*/

:-    use_module(library(files),[file_exists/1]),
      use_module(library(ctypes),[is_endfile/1]).

% shorthand(s,start,s0,1).                           % for reference only
% shorthand(g,goal,g0,2).

start :-                                             % interactive session
      cls,
      bold, nl,
      prompt(_,'Enter MAP name: '), read(MAP),
      root(ROOT),
      concat(ROOT,'lib/',LIB),
      concat(LIB,MAP,LIBMAP),
      alloff, nl,
      pre(LIBMAP),
      start_prowindows,                             % contained in pwin.pl
      writeln('Coordinate format is [x,y]. Range [0..100,0..100].'),
      enter_point(s),
      enter_point(g),
      tail_sequence,
      !.

start(N,MAP,S,G) :-                                      % background job
      root(ROOT),
      concat(ROOT,'lib/',LIB),
      concat(LIB,MAP,LIBMAP),
      pre(LIBMAP),
      set_sg(s,S),
      set_sg(g,G),
      show_parms,
      go,
      stats(N),
      !.
start(N,MAP,S,G) :-
      writeln('WARN (start)...failed to start!'),
      write_list(['Prob No.',N]),
      write_list([' Map =',MAP]),
      write_list(['Start =',S]),
      write_list([' Goal =',G]),
      !.
```

```
new :- new(s), new(g), !.

new(X) :-
      nl,
      retractall(ep(X,_,_,_,_)),
      mapname(MAP),
      shorthand(X,XXXX,_,_),
      write_list(['Reinitialize',XXXX,'for',MAP]),
      writeln('Coordinate format [x,y]. Range [0..100,0..100].'),
      enter_point(X),
      tail_sequence,
      !.

enter_point(s) :-
      bold, nl,
      prompt(_,'Enter new START: '), read(P),
      alloff,
      set_sg(s,P),
      w_showsg(s),
      !.
enter_point(g) :-
      bold, nl,
      prompt(_,'Enter new GOAL: '), read(P),
      alloff,
      set_sg(g,P),
      w_showsg(g),
      !.

%------------------------------------------------------------------------

store :- mapname(M), store(M).

store(M) :-
      concat(M,'.pre',MPRE),
      save(MPRE,1),
      reinitial.
store(_).

reinitial :- w_init, tail_sequence.

recall :- mapname(M), recall(M).

recall(M) :-
      concat(M,'.pre',MPRE),
      recall2(MPRE).

recall2(MPRE) :-
      file_exists(MPRE),
      kill_prowindows,
      restore(MPRE),
      !.
```

```
recall2(MPRE) :-
      write_list_b(['WARN...state',MPRE,'cannot be loaded']),
      writeln('Issue ''start.'' command.').

tail_sequence :-
      show_parms,
      nl,
      writeln_b('COMMANDS '),
      writeln('go. begin execution'),
      writeln('new(s). reset start'),
      writeln('new(g). reset goal'),
      writeln('store. save preprocessed map'),
      writeln('recall. recall this map'),
      writeln('gtrace(X). graphics tracing:'),
      writeln(' 1 ALL algorithms'),
      writeln(' d DISCRETE approximations'),
      writeln(' i RELAXATION (iteration)'),
      writeln(' 0 trace OFF'),
      writeln_b(' ').

show_parms :-                                     % order is SIGNIFICANT
      w_showdata(time,0.0),
      di(DI),
      w_showdata(di,DI),
      deltak(DK),
      w_showdata(deltak,DK),
      !.
show_parms :- !.

/*=======================================================================
 SET_SG - given s(S) and g(G), finds RS,RG regions containing them;
 algorithm is based upon fact that a convex region cannot have
 2 consecutive vertices whose heading difference from an
 arbitrary interior point exceeds pi radians (180 deg)
=======================================================================*/

:- dynamic
      hd/2,                              % hd(v,h). heading from s/g to v is h
      start/2,                  % start(p0,r0). start and region containing it
      goal/2.                     % goal(p1,r1). goal and region containing it

set_sg(X,P) :-
      retractall(hd(_,_)),
      find_region(X,P,R),
      shorthand(X,XXXX,_,_),
      OLD =.. [XXXX,_,_],
      retractall(OLD),
      NEW =.. [XXXX,P,R],
      asserta(NEW),
      set_sg_arcs(X),
      !.
```

```
find_region(X,P,R) :-
      rv(R,W,VLST),
      fanout(P,VLST),
      not(same(W,0)),
      shorthand(X,XXXX,_,_),
      write_list([XXXX,P,'in region',R]),
      !.
find_region(_,P,_) :-
      write_list_b(['ERROR (start)...invalid instance...',P,'is']),
      writeln_b(' inside obstacle, outside border, or on vertex.'),
      prowindows_exists,
      !.

fanout(P,[V0|VLST]) :-
      get_heading(P,V0,H0),
      fanfwd([V0,H0],P,V0,H0,VLST),
      !.

get_heading(_,V,H) :- hd(V,H), !.            % return if already computed
get_heading(P,V,H) :- % else
      heading(P,V,H), % compute heading from P to V
      asserta(hd(V,H)), % cache result
      !.

fanfwd([V0,H0],P,V1,H1,[]) :-                          % close the circle
      deltah(H1,H0,DH),
      !,
      DH < 180.0,
      !.
fanfwd(VH,P,V1,H1,[V2|VLST]) :-
      get_heading(P,V2,H2),
      deltah(H1,H2,DH),
      !,
      DH < 180.0,
      fanfwd(VH,P,V2,H2,VLST),
      !.

deltah(H1,H2,DH) :- H2 > H1, DH is H2-H1, !.
deltah(H1,H2,DH) :- DH is H2-H1+360.0, !.

%------------------------------------------------------------------------

set_sg_arcs(X) :-
      retract_old_sg_arcs(X),
      get_p_r(X,P,R),
      re(R,W,ELIST),
      asserta(edge(X,[P,P],[W,W],X,X)),
      asserta(edge_data(X,0.0,P)),
      asserta(er(X,[R,R],[W,W])),
      gen_sg_ep(X,P,R,W,ELIST),
        rv(R,_,VLIST),
        gen_sg_choke_ep(X,P,R,W,VLST),
        straight_line_ep,
        !.
set_sg_arcs(_) :- writeln('set_sg_arcs failed.'), !.

retract_old_sg_arcs(X) :-
        retractall(edge(X,_,_,_,_)),
        retractall(edge_data(X,_,_)),
        retractall(er(X,_,_)),
        retractall(ep(X,_,_,_,_)),
        retractall(ep(_,X,_,_,_)),
        !.

straight_line_ep :-
        start(S,R),
        goal(G,R),
        re(R,W,_),
        wdistance(S,G,W,WD),
        asserta(ep(s,g,R,W,WD)),
        !.
straight_line_ep :- !.

gen_sg_ep(_,_,_,_,[]).
gen_sg_ep(X,P,R,W,[E|ELIST]) :-
        not(edge(E,_,[_,0],_,_)),
        edge_data(E,_,MP),
        wdistance(P,MP,W,WD),
        asserta(ep(X,E,R,W,WD)),
        gen_sg_ep(X,P,R,W,ELIST).
gen_sg_ep(X,P,R,W,[_|ELIST]) :-
        gen_sg_ep(X,P,R,W,ELIST).

gen_sg_choke_ep(_,_,_,_,[]).
gen_sg_choke_ep(X,P,R,W,[V|VLST]) :-
        chr(EC,V,_),
        wdistance(P,V,W,WD),
        asserta(ep(X,EC,R,W,WD)),
        gen_sg_choke_ep(X,P,R,W,VLST).
gen_sg_choke_ep(X,P,R,W,[V|VLST]) :-
        gen_sg_choke_ep(X,P,R,W,VLST).
```

```
/*=======================================================================
 MOUSE.PL - mouse driver for UDPGA* and Path Annealing with ProWindows
=======================================================================*/

:- dynamic mse/1.

mse(s0).                        % designate start point first by default

:-
      use_module(library(dialog)),
      use_module(library(interpret_messages)),
      use_module(library(messages)).

%------------------------------------------------------------------------

pixels(500).% total number of map pixels in x and y directions

map_size(100).% total number of map units in x and y directions

map_item(@xcoord,text_item('X: ',0,none),below,[width: 4]).
map_item(@ycoord,text_item('Y: ',0,none),below,[width: 4]).
map_item(@mstart, button('Start Pt',m_start), below,[]).
map_item(@mgoal, button('Goal Pt',m_goal), below,[]).
map_item(@mgo, button('GO',                            m_go), below,[]).
map_item(@mstore, button('Store',m_store), below,[]).
map_item(@mrecall,button('Recall',m_recall), below,[]).
map_item(@mpost, button('Print',m_post), below,[]).
map_item(@mwinit, button('W_Init',m_w_init), below,[]).
map_item(@mnewmap,button('New Map',m_newmap), below,[]).
map_item(@mkill, button('Kill Mouse',m_kill), below,[]).
map_item(@msusp, button('Suspend',m_suspend),below,[]).
map_item(@mhalt, button('Quit',m_quit), below,[]).

calculate_dim :-
      pixels(P),
      map_size(C),
      G is P/C,
      IG is P//C,
      assert(pm_ratio(G)),
      assert(ipm_ratio(IG)).

exterminate :-
      destroy(@map_dialog),
      destroy(@mstart),
      destroy(@mgoal),
      destroy(@mkill),
      destroy(@mgo),
      destroy(@mstore),
      destroy(@mrecall),
      destroy(@mwinit),
      destroy(@xcoord),
      destroy(@ycoord),
      !.

%-----------------------------------------------------------------------

mouse :-
      exterminate,                          % exterminate old mouse stuff
      calculate_dim,
      send(@pic,smart,off),
      new_dialog(@map_dialog,'Mouse Ctrl',map_item),
      send(@map_dialog,open,point(510,470)),
      mloop.

mloop :-
      send(@pic,cursor_x,cascade(@xcoord,adjustx,0)),
      send(@pic,cursor_y,cascade(@ycoord,adjusty,0)),
      send(@pic,left_up, cascade(@pic,clicked,0)),
      interpret_messages(fail).

%-----------------------------------------------------------------------

clicked(PIC,_) :-
      mse(P0),
      get_ref(@PIC,last_click,PT),
      get(PT,x,XPIX),
      get(PT,y,YPIX),
      convert_pt([XPIX,YPIX],[X,Y]),
      click_sg(P0,[XPIX,YPIX],[X,Y]),
      !.

click_sg(s0,[XPIX,YPIX],[X,Y]) :-
      set_sg(s,[X,Y]),
%     retractall(start(_,_)),
%     asserta(s([X,Y])),
      send(@s0,pen,1),
      send(@s0,center,point(XPIX,YPIX)),
      w_showdata(start,[X,Y]),
      !.
click_sg(g0,[XPIX,YPIX],[X,Y]) :-
      set_sg(g,[X,Y]),
%     retractall(goal(_,_)),
%     asserta(g([X,Y])),
      send(@g0,pen,2),
      send(@g0,center,point(XPIX,YPIX)),
      w_showdata(goal,[X,Y]),
      !.

%-----------------------------------------------------------------------
```

```
m_start(_,_) :- retractall(mse(_)), asserta(mse(s0)), !.
m_goal(_,_) :- retractall(mse(_)), asserta(mse(g0)), !.
m_go(_,_) :- go.
m_store(_,_) :- store.
m_recall(_,_) :- recall.
m_post(_,_) :- w_post.
m_w_init(_,_) :- w_init.
m_kill(_,_) :- exterminate, abort.
m_newmap(_,_) :- start.
m_suspend(_,_) :- writeln('Mouse suspended...hit ^D to return.'), break.
m_quit(_,_) :- halt.

%-----------------------------------------------------------------------

adjustx(XITEM,X) :-                % display transformed x in dialog box
      pm_ratio(R),
      XX is X*10/R,
      floor(XX,RX),
      send(@XITEM,selection,RX).

adjusty(YITEM,Y) :-                % display transformed y in dialog box
      pm_ratio(R),
      map_size(M),
      YY is (10*M)-Y*10/R,
      floor(YY,RY),
      send(@YITEM,selection,RY).

convert_pt([PX,PY],[X,Y]) :-
      pixels(W),
      pm_ratio(G),
      XX is PX/G,
      YY is (W-PY)/G,
      round(XX,X),
      round(YY,Y).

convert_pix([X,Y],[PX,PY]) :-
      pixels(W),
      pm_ratio(G),
      TPX is G*X,
      TPY is W-G*Y,
      round(TPX,PX),
      round(TPY,PY).
```

```
/*======================================================================
 PWIN.PL - ProWindows: windows and tracing of algorithms
======================================================================*/

:- dynamic
      timer/0,        % time display for interactive runs w/o ProWindows
      ws_switch/0,
      wnos/0,                           % switch: display region weights
      rnos/0,                     % switch: display region index numbers
      enos/0,                       % switch: display edge index numbers
      detail_d/0,                  % discrete point (udp) tracing switch
      detail_i/0.                       % iteration (spr) tracing switch

% enos.
% rnos.
wnos.                                                          % default

/*----------------------------------------------------------------------
 graphical tracing ON(1) / OFF(0)                   default = all OFF(0)
----------------------------------------------------------------------*/

:- retractall(detail_d), retractall(detail_i).          % init switches

gtrace :- gtrace(1).

gtrace(1) :- t_set(detail_d), t_set(detail_i).                 % both ON
gtrace(i) :- gtrace(0), t_set(detail_i).                  % iteration ON
gtrace(d) :- gtrace(0), t_set(detail_d).                   % discrete ON
gtrace(0) :-                                                  % both OFF
      retractall(detail_d),
      retractall(detail_i),
      destroy(@iter).

t_set(D) :-
      prowindows_exists,
      retractall(D),
      assert(D),
      warn(0),
      create_iter,
      warn(1).
t_set(_) :-
      writeln('CANNOT trace graphics...ProWindows NOT running.'),
      fail.

%-----------------------------------------------------------------------

create_iter :- new(@iter,figure(0,0,0)), send(@pic,display,@iter).
create_iter.                                            % already exists
```

```
warn(1) :- prowindows_exists, send(@sys,warnings,on).
warn(0) :- prowindows_exists, send(@sys,warnings,off).
warn(_).

start_prowindows :-                        % attempt to start prowindows
      prowindows, nl,
      w_init,
      !.
start_prowindows :-                       % prowindows CANNOT be started
      writeln(user,'NOTE...All screen output will be textual.'), nl,
      !.

w_init :-
      kill_prowindows,
      prowindows,
      mapname(MN),
      grey_bitmap,
      black_bitmap,
      display_map(MN,@pic),
      showtitle,
      w_initlbls,
      reestablish_detail_status,
      create_sg(s0),
      create_sg(g0),
      w_showsg(s),
      w_showsg(g),
      w_initparms,
%     new(@path,path),           % used when erasing and replacing paths
      mouse,
      !.
w_init :- !.                       % ProWindows warns if it cannot start

create_sg(CIR) :-
      shorthand(_,_,CIR,TH),
      new(@CIR,circle),
      send(@CIR,radius,5),
      send(@CIR,pen,TH),
      initial_place(CIR),
      send(@pic,display,@CIR).

initial_place(s0) :- send(@s0,center,point(425,565)).
initial_place(g0) :- send(@g0,center,point(425,585)).

reestablish_detail_status :-
      ( detail_d ; detail_i ),
      new(@iter,figure(0,0,0)),
      send(@pic,display,@iter),
      !.
reestablish_detail_status :- !.
```

```
w_showsg(SG) :-
      prowindows_exists,
      get_p_r(SG,P,_),
      shorthand(SG,STRING,CIR,TH),
      convert_pix(P,[XPIX,YPIX]),
      send(@CIR,pen,TH),
      send(@CIR,center,point(XPIX,YPIX)),
      w_showdata(STRING,P),
      !.
w_showsg(SG) :-
      get_p_r(SG,P,_),
      shorthand(SG,XXXX,_,_),
      write_list([XXXX,'=',P]),
      !.
w_showsg(SG) :-
      shorthand(SG,XXXX,_,_),
      write_list_b(['NO',XXXX,'point established.']),
      !.

get_p_r(s,P,R) :- start(P,R), !.
get_p_r(g,P,R) :- goal(P,R), !.

shorthand(s,start,s0,1).
shorthand(g,goal, g0,2).

%-----------------------------------------------------------------------

w_pin(P) :-                                 % used to highlight a point X,Y
      prowindows_exists,
      ptransform(P,[SX,SY]),
      X1 is SX+12,                                         % diagonal arrow
      Y1 is SY-12,
      new(@P,circle(X1,Y1,5)),
      send(@P,pen,4),
      send(@pic,display,@P),
      new(@L,line(X1,Y1,SX,SY)),
      send(@pic,display,@L),
      !.
w_pin(P) :- write_list(['WARNING...pin point placed on',P]), !.

w_point(P,RADIUS) :-                        % used to highlight a point X,Y
      prowindows_exists,
      ptransform(P,[SX,SY]),
      new(@PT,circle(SX,SY,RADIUS)),
%     send(@PT,fill,@black),                                 % option to fill
      send(@PT,pen,5),                                     % option to thicken
      send(@pic,display,@PT),
%     EXT is RADIUS+2,                        % option to extend crosshairs
%     crosshairs([SX,SY],EXT),                         % option for crosshairs
      !.
w_point(_,_) :- !.
```

```
crosshairs([SX,SY],EXT) :-
      X1 is SX+EXT,                                         % crosshairs
      X2 is SX-EXT,
      Y1 is SY+EXT,
      Y2 is SY-EXT,
      new(@LH,line(X1,SY,X2,SY)),
      new(@LV,line(SX,Y1,SX,Y2)),
      send(@pic,display,@LH),
      send(@pic,display,@LV),
      !.

w_edge(E) :-                                 % used to highlight an edge E
      prowindows_exists,
      edge_data(E,_,MP),
      ptransform(MP,[SX,SY]),
      new(@RING,circle(SX,SY,9)),
      send(@pic,display,@RING),
      !.
w_edge(_) :- !.

w_clear_iter :-
      (detail_d ; detail_i),
      send(@iter,destroy),
      new(@iter,figure(0,0,0)),
      send(@pic,display,@iter),
      !.
w_clear_iter :- !.

w_show_iter(PATH,TH) :-
      detail_i,
      new(@ITER,path),
      append_points(0,@ITER,PATH),
      send(@ITER,pen,TH),
      send(@iter,append,@ITER),
      !.
w_show_iter(_,_) :- !.

w_show_line(P1,P2,TH) :-
      detail_d,
      ptransform(P1,[SX1,SY1]),
      ptransform(P2,[SX2,SY2]),
      new(@L,line(SX1,SY1,SX2,SY2)),
      send(@L,pen,TH),
      send(@iter,append,@L),
      !.
w_show_line(_,_,_) :- !.


w_showpath(PATH) :-
      prowindows_exists,
      destroy(@path),            % used when erasing and replacing paths
      display_path(@path,PATH,@pic,2),
%     display_path(@P,PATH,@pic,3),
      !.
w_showpath(_) :- !.

w_showdata(CAT,DATA) :-
      prowindows_exists,
      code(CAT,_,X,Y),
      spacing(S),
      XD is X+S,
      type_convert(DATA,SDATA),
      destroy(@CAT),
      new(@CAT,text(SDATA,XD,Y,left)),
      send(@CAT,font,font(screen,b,14)),
      send(@pic,display,@CAT),
      !.
w_showdata(time,DATA) :- screen_timer(DATA), !.
w_showdata(pacost,_) :- !.                          % displayed by sa.pl
w_showdata(CAT,DATA) :- write_list([CAT,’=’,DATA]), !.

screen_timer(DATA) :- timer, writeln(DATA), backup, !.
screen_timer(_) :- !.

type_convert(DATA,SDATA) :-
      is_list(DATA),
      concat_atom(DATA,’,’,SDATA),
      !.
type_convert(DATA,SDATA) :- make_string(DATA,SDATA), !.

/*-----------------------------------------------------------------------
 window sequence display routines
-----------------------------------------------------------------------*/

w_wsinit :-
      prowindows_exists,
      w_clear_ws,
      new(@ws2,path),
      !.
w_wsinit :- !.

w_clear_ws :-
      prowindows_exists,
      retractall(ws_switch),
      destroy(@ws1),
      destroy(@ws2),
      !.
w_clear_ws :- !.

destroy(OBJ) :- object(OBJ), send(OBJ,destroy), !.
destroy(_) :- !.
```

```
w_show_ws(L) :-
      prowindows_exists,
      get_midpts(L,LM),
      swap_ws(LM),
      !.
w_show_ws(_) :- !.

get_midpts([],[]) :- !.
get_midpts([E|LE],[M|LM]) :-
      edge_data(E,_,M),
      get_midpts(LE,LM),
      !.

swap_ws(LM) :- retract(ws_switch), w_ws(@ws1,@ws2,LM), !.
swap_ws(LM) :- assert(ws_switch), w_ws(@ws2,@ws1,LM), !.

w_ws(@WS1,@WS2,LM) :-
      new(@WS2,path),
      append_points(0,@WS2,LM),
      send(@WS2,pen,2),
% send(@WS2,smooth,on),                               % smoothing option
      send(@WS1,destroy),
      send(@pic,display,@WS2),
      !.

append_points(_,_,[]) :- !.
append_points(SW,@PATH,[[_,[X,Y]]|L]) :-                % discrete paths
      ptransform([X,Y],[IX,IY]),
      new(@PT,point(IX,IY)),
      send(@PATH,append,@PT),
      turn_mark(SW,[IX,IY]),                   % SW=1, mark turns by '+'
      append_points(SW,@PATH,L),
      !.
append_points(SW,@PATH,[P|L]) :-
      ptransform(P,[IX,IY]),
      new(@PT,point(IX,IY)),
      send(@PATH,append,@PT),
      turn_mark(SW,[IX,IY]),                   % SW=1, mark turns by '+'
      append_points(SW,@PATH,L),
      !.

turn_mark(1,[IX,IY]) :-
      TXTIY is IY+1,                               % correct text center
      new(@TXT,text('+',IX,TXTIY,center)),          % turn marks are '+'
      send(@TXT,font,font(screen,b,12)),
      send(@pic,display,@TXT),
      !.
turn_mark(0,_) :- !.
```

```
/*-----------------------------------------------------------------------
 print routines
-----------------------------------------------------------------------*/

w_post :- prowindows_exists, w_post(ss1).

w_post(PRT) :-
      prowindows_exists,
      send(@pic,postscript,pspicfile),
      send_to(PRT),
%     unix(system('rm pspicfile')),
      concatenate('lpq -P',PRT,QPRT),
      unix(system(QPRT)).
w_post(_).

send_to(ss1) :- unix(system('lpr -Pss1 pspicfile')).
send_to(up1) :- unix(system('lpr -Pup1 pspicfile')).
send_to(up2) :- unix(system('lpr -Pup2 pspicfile')).
send_to(secs) :- unix(system('lpr -Psecs pspicfile')).
send_to(phd) :- unix(system('lpr -Pphd pspicfile')).

/*-----------------------------------------------------------------------
 preparation of map display
-----------------------------------------------------------------------*/

display_map(MAP,PIC) :-
      algorithm_type(ALG),                              % set in make file
      name(ALG,NALG),
      name(' - ',NSPC),
      name(MAP,NMAP),
      append(NALG,NSPC,NLBL1),
      append(NLBL1,NMAP,NLBL2),
      name(LABEL,NLBL2),
      new(PIC,picture(LABEL,size(500,700))),
      send(PIC,horizontal_scrollbar,0),
      send(PIC,vertical_scrollbar,0),
      send(PIC,open,point(620,70)),
      ticks,
      new(@map,figure(0,0,0)),
      send(@map,can_move,off),
      append_edges(@map),
      send(PIC,display,@map),
      fill_obstacles,
      !.

ticks :-
      xticks(100,0),
      yticks(0,100),
      !.
```

```
xticks(X,Y) :-
      X >= 0,
      xtransform(X,TX),
      ytransform(Y,TY),
      new(@TXT,text('|',TX,TY,center)),
      send(@TXT,font,font(cour,b,10)),
      send(@pic,display,@TXT),
      X2 is X-10,
      xticks(X2,Y),
      !.
xticks(_,_) :- !.

yticks(X,Y) :-
      Y >= 0,
      xtransform(X,TX),
      ytransform(Y,TY),
      new(@TXT,text('-',TX,TY,center)),
      send(@TXT,font,font(cour,b,18)),
      send(@pic,display,@TXT),
      Y2 is Y-10,
      yticks(X,Y2),
      !.
yticks(_,_) :- !.

append_edges(@FIG) :-
      edge(E,[V1,V2],[W1,W2],_,_),
%     not(same(W1,W2)),                    % phantom edges not displayed
      not(same(E,s)),
      not(same(E,g)),
      midpoint(V1,V2,MP),
      ptransform(V1,[PX1,PY1]),
      ptransform(V2,[PX2,PY2]),
      ptransform(MP,[PXM,PYM]),
      new(@LIN,line(PX1,PY1,PX2,PY2)),
%     send(@LIN,arrows,first),            % visual check edge consistency
      send(@FIG,append,@LIN),
      enos,
      new(@TXT,text(E,PXM,PYM,center)),
      send(@TXT,font,font(cour,b,10)),
      send(@pic,display,@TXT),
      fail.
append_edges(_) :- ( enos ; region_lbls ), !.

region_lbls :-
      rv(R,W,VLST),
      not(same(W,0)),
      avg_vlst(VLST,V0),
      ptransform(V0,[PX,PY]),
      get_region_lbl(R,W,L),
      new(@RWT,text(L,PX,PY,center)),
      send(@RWT,font,font(cour,b,10)),
      send(@pic,display,@RWT),
      fail.
region_lbls :- !.

get_region_lbl(R,_,R) :- rnos, !.
get_region_lbl(_,W,W) :- wnos, !.

nos(N) :-
      retractall(enos),
      retractall(rnos),
      retractall(wnos),
      ( N == e -> assert(enos)
      | N == r -> assert(rnos)
      | N == w -> assert(wnos) ).

avg_vlst(VLST,[X,Y]) :-
      accum_vlst(VLST,[TX,TY]),
      length(VLST,L),
      X is TX/L,
      Y is TY/L,
      !.

accum_vlst([],[0.0,0.0]) :- !.
accum_vlst([[X,Y]|VLST],[XCM,YCM]) :-
      accum_vlst(VLST,[XSB,YSB]),
      XCM is XSB+X,
      YCM is YSB+Y,
      !.

display_path(@PATH,L,PIC,TH) :-     % L = [[x1,y1],[x2,y2],...,[xn,yn]]
      new(@PATH,path),
      append_points(0,@PATH,L),
%     append_points(1,@PATH,L),
      send(@PATH,pen,TH),         % TH = thickness in pixels (1 default)
      send(PIC,display,@PATH),
      !.

ptransform([X,Y],[SX,SY]) :-
      xtransform(X,SX),
      ytransform(Y,SY),
      !.

xtransform(X,IX) :-
      XX is X*5,
      round(XX,IX),
      !.

ytransform(Y,IY) :-
      YY is (100-Y)*5,
      round(YY,IY),
      !.
```

```
grey_bitmap :-
      new(@PATTERN,bitmap(2,2)),
      send_list(@PATTERN,set,[point(0,0),point(1,1)]),
      new(@grey,bitmap(64,64)),
      send(@grey,replicate,@PATTERN).

black_bitmap :-
      new(@PATTERN,bitmap(2,2)),
      send_list(@PATTERN,set,
                [point(0,0),point(0,1),point(1,0),point(1,1)]),
      new(@black,bitmap(64,64)),
      send(@black,replicate,@PATTERN).

% send_list(@PATTERN,set,[]),               % what white would look like

w_initlbls :-
      code(CAT,LABEL,X,Y),
      new(@LBL,text(LABEL,X,Y,left)),
      new(@CAT,text(' ',0,0,left)),
      send(@LBL,font,font(screen,b,14)),
      send(@pic,display,@LBL),
      fail.
w_initlbls :- !.

decode(TERM,SDATA,CAT,XD,Y) :-
      TERM =.. [CAT,DATA],
      make_string(DATA,SDATA),
      code(CAT,_,X,Y),
      spacing(S),
      XD is X+S,
      !.

make_string(DATA,STRING) :-
      number(DATA),
      number_chars(DATA,CLIST),
      atom_chars(STRING,CLIST),
      !.
make_string(DATA,STRING) :-
      atom(DATA),
      name(DATA,CLIST),
      atom_chars(STRING,CLIST),
      !.
make_string(DATA,_) :-
      var(DATA),
      writeln('ERROR (pwin)...cannot make string from variable'),
      !.
make_string(DATA,_) :-
      write_list(['ERROR (pwin)...unknown data type:',DATA]),
      !.
```

```
concatenate(S1,S2,S3) :-
      name(S1,L1),
      name(S2,L2),
      append(L1,L2,L3),
      name(S3,L3),
      !.

showtitle :-
      mapname(MN),
      algorithm_type(ALG),
      concatenate(ALG,' - ',TITLE1),
      concatenate(TITLE1,MN,TITLE),
      new(@ttl,text(TITLE,250,530,center)),
      send(@ttl,font,font(screen,b,16)),
      send(@pic,display,@ttl),
      new(@datestmp,text(' ',250,660,center)),
      send(@datestmp,font,font(screen,b,14)),
      send(@pic,display,@datestmp),
      !.

spacing(150).

code(pacost, 'best path ...... ', 10,560).
code(elcost, 'ellipse cost ... ', 10,580).
code(di, 'discrete int ... ', 10,600).
code(deltak, 'delta k ........ ', 10,620).

code(start, 'start point .... ', 260,560).
code(goal, 'goal point ..... ', 260,580).
code(blank1, ' ', 260,600).
code(time, 'cpu time used .. ', 260,620).

update_clock(NEWTIME) :-
      tm(ETIME),
      retract(time(TIME)),
      NEWTIME is TIME+ETIME,
      assert(time(NEWTIME)),
      w_showdata(time,NEWTIME),
      tm,
      !.

w_initparms :-
      retractall(time(_)),
      asserta(time(0.0)),
      w_showdata(time,0.0),
      di(DI),
      w_showdata(di,DI),
      deltak(DK),
      w_showdata(deltak,DK),
      !.
w_initparms :- !.
```

```
ds :- datestamp(X), ds2(X), !.

ds2(X) :-
      prowindows_exists,
      send(@datestmp,string,X),
      fail, !.
ds2(X) :-
      algorithm_type(ALG),
      nl,
      writeln('_______________________________________________'),
      write(ALG),                                         % algorithm id
      write(' '),
      writeln(X),                                           % time stamp
      !.

datestamp(X) :-
      sys('date>date'),
      open(date,read,S),
      see(S),
      fill_string(NLIST),
      name(X,NLIST),
      seen,
      sys('rm date'),
      !.

fill_string([C|NLIST]) :-
      get0(C),
      not(is_endfile(C)),
      fill_string(NLIST),
      !.
fill_string([]) :- !.

fill_obstacles :-
      rv(R,0,[V|VLST]),
      not(border(R)),
      append([V|VLST],[V],VL),
      fill_poly(VL),
      fail.
fill_obstacles :- !.

fill_poly(VL) :-
      new(@POLY,path),
      append_points(0,@POLY,VL),
      send(@POLY,fill,@grey),
      send(@pic,display,@POLY),
      !.
```

B.5 UNIFORM-DISCRETE-POINT GLOBAL A* (UDPGA*) SEARCH

```
/*======================================================================
 UDPGA.PL - Uniform-Discrete-Point Global A* (UDPGA*) Search
      Approximates the globally optimal path by using A* search
      on the graph of evenly spaced points on each boundary edge,
      connected by arcs (segments) through common cost regions.
      Parameter di(I) is the upper bound for point to point spacing.
      uses HEAP in C to maintain agenda
======================================================================*/

/*======================================================================
 A* SEARCH
      adapted from Rowe, N.C.,
      "Artificial Intelligence Through Prolog,"
      Prentice Hall, 1988, pp. 244-247.
=======================================================================

 For an application, define 5 predicates:
      (a) successor(State,Newstate)           (gives state transitions)
      (b) goalreached(State)                 (defines when goal achieved)
      (c) eval(State,Evaluation)             (estimates cost to the goal)
      (d) cost(Statelist,Cost)                 (computes cost of a path)
      (e) search: top-level predicate that initializes things,
          then calls astarsearch with two arguments, the starting
          state and variable which will be the solution path

 Note that "cost" must be nonnegative. The "eval" should be a lower
 bound on cost in order for the first answer found to be guaranteed
 optimal, but the right answer will be reached eventually otherwise.

 At conclusion of processing:
      no. of items in agenda - 1 = no. of unexamined states
      no. of items in usedstate = no. examined states

 Modifications to original code:

 In this version, we modified cost to cost(Statelist,OldCost,NewCost)
 to reduce the work required to compute the cost of a new statelist.
 Also, we have replaced pick_best_path rules with a more efficient
 version which maintains heap to find the minimum cost+eval on agenda
 more efficiently. Since heaps are not efficient in Prolog, the heap
 routines have been coded in C.

 The original code referenced in [Rowe88, pp. 244-247] states that if
 the evaluation function is ALWAYS a lower bound on the cost function,
 then faster performance results if we delete the first rule of
 agenda_check and usedstate_check, and the entire definitions of
 fix_agenda and replace_front. We have determined that the first rule
 of agenda_check IS REQUIRED to guarantee optimality in a general
 graph (ie. NOT a tree). Therefore, we have NOT excluded it. Since
 our evaluation function is Euclidean distance to the goal at lowest
 cost on the map, then it is a lower bound on cost.
 We have, therefore, modified the original code accordingly.

========================================================================

:- dynamic
      edge_pt/2,                                     % edge_pt(e,[x,y]).
      usedstate/2,                         % usedstate([e,[x,y]], cost).
      agenda/5,    % agenda(i,[e,[x,y]], [[e,[x,y]],..],cost,eval+cost).

                                   % dynamic: crude and refined versions
      successor/2,
      goalreached/1,
      eval/2,
      cost/3,                             % cost(state,oldcost,newcost).

      time/1,                                                 % time(t).
      di/1,                                         % overrides udp.pl
      deltak/1,            % used for compatibility w/ pwin.pl, start.pl
      solution/4.                         % solution(ws,path,cost,time).

di(8).                                                 % default setting
deltak(0.0).             % for compatibilty (NOT used in this algorithm)

%-----------------------------------------------------------------------

clean_ops :-
      retractall(successor(_,_)),
      retractall(goalreached(_)),
      retractall(eval(_,_)),
      retractall(cost(_,_,_)),
      !.

/*----------------------------------------------------------------------
 Crude search through midpoints in order to compute bounding ellipse
 NOTE: successor, goalreached, cost, and eval fns CANNOT have cuts !
----------------------------------------------------------------------*/

set_crude_ops :-
      clean_ops,
      asserta((
          successor(E0,E1) :-
          ( ep(E0,E1,_,_,_) ; ep(E1,E0,_,_,_) )
          )),
      asserta(( goalreached(g) )),
      asserta((
        eval(E0,EVAL) :-
        goal(GP,_),
        edge_data(E0,_,MP),
        distance(MP,GP,EVAL)
        )),
    asserta((
        cost([E0,E1|_],OLDCOST,COST) :-
        ( ep(E0,E1,_,_,C) ; ep(E1,E0,_,_,C) ),
        COST is C+OLDCOST
        )),
    asserta(( cost([s],_,0.0) )),
    !.

/*-----------------------------------------------------------------------
 Refined search through uniformly spaced (interval di) edge-points
-----------------------------------------------------------------------*/

set_refined_ops :-
    clean_ops,
    asserta((
        successor([E0,P0],[E1,P1]) :-
        ( ep(E0,E1,_,_,_) ; ep(E1,E0,_,_,_)
        ; refl_edge(E0,E1) ),                             % or reflection
        edge_pt(E1,P1),                         % P1 must be a point on E0
        not(same(P0,P1))                      % same points ==> no progress
        )),
    asserta(( goalreached([g,_]) )),
    asserta((
        eval([_,P0],EVAL) :-                % Eucl dist to goal at coef = 1
        goal(GP,_),
        optimal_cost(W),
        wdistance(P0,GP,W,EVAL)
        )),
    asserta(( cost([S0,S1|_],OC,C) :- rcost(S0,S1,OC,C) )),
    asserta(( cost([[s,_]],_,0.0) )),
    !.

refl_edge(E0,E0) :- not(edge(E0,_,[W,W],_,_)).

rcost([E0,P0],[E0,P1],OLDCOST,COST) :-
    edge(E0,_,[W0,_],_,_),
    wdistance(P0,P1,W0,C),
    COST is OLDCOST + C.
rcost([E0,P0],[E1,P1],OLDCOST,COST) :-
    ( ep(E0,E1,_,W,_) ; ep(E1,E0,_,W,_) ),
    wdistance(P0,P1,W,C),
    COST is OLDCOST + C.
```

```
/*----------------------------------------------------------------------
 refined search data structure generation
----------------------------------------------------------------------*/

gen_all_points :-                   % also generates records for s and g
      edge(E,V,[_,W],_,_),
      not(same(W,0)),
      get_pts(E,V,PLST),
      make_records(E,PLST),
      fail, !.
gen_all_points :- !.

make_records(_,[]) :- !.
make_records(E,[[X,Y]|PLST]) :-
      asserta(edge_pt(E,[X,Y])),         % ensures hashing on E edge no.
      make_records(E,PLST),
      !.

/*----------------------------------------------------------------------
 search driver
----------------------------------------------------------------------*/

search :-
      w_initparms,                             % display initial parameters
      unknown(_,fail),                            % required by astarsearch
      reset_points,                  % reset epts and edge-pt for udp.pl
      set_crude_ops,
      writeln('Stage 1: Crude A* search (init solution)...'),
      ts,
      astarsearch(s,WS,_),
      writeln(WS),
      optimize_ws(WS,CrudePath,UCost,SCost),
      write_list(['Crude path costs UDP =',UCost,' SPR =',SCost]),
      update_clock(T1),
      write_list(['time =',T1]),
      w_showpath(CrudePath),
      avg_path_wt(SCost,CrudePath,APW),
%     asserta(optimal_cost(APW)),                    % heuristic ellipse
      write_list(['Average Weight =',APW]),
      tm,
      ellipse(SCost),
      update_clock(T2),
      write_list(['time =',T2]),
      writeln('Stage 2: Refined A* search (UDPGA*)...'),
      set_refined_ops,
      gen_all_points,
      start(SP,_),
      astarsearch([s,SP],StateList,ACost),
      write_list(['A* search path cost =',ACost]),
      separate(StateList,AWS,_),
      writeln(AWS),
      writeln('Check WS for inconsistencies...'),
      fix_ws(AWS,FixedWS),
      writeln(FixedWS),
      optimize_ws(FixedWS,SolutionPath,UOCost,OptCost),
      write_list(['Optimal Path costs UDP =',UOCost]),
      write_list([' SPR =',OptCost]),
      update_clock(T3), nl,
                                              % Cleanup and Housekeeping
      unknown(_,trace),                        % reset after astarsearch
      w_showpath(SolutionPath),
      w_showdata(pacost,OptCost),
      write_list(['UDPGA* Optimal cost =',OptCost]),
      write_list(['Finish time =',T3]),
      asserta(solution(FixedWS,SolutionPath,OptCost,T3)),
      notrace,                                    % in case trace was on
      !.
search :- writeln_b('ERROR (astar)...A* search failed !'), !.

optimize_ws(WS,PATH,C,COST) :-
      writeln('UDP...'),
      dijkstra(WS,P,C),
      writeln('SPR...'),
      solve(WS,P,PATH,COST),
      !.

%-----------------------------------------------------------------------

separate([],[],[]) :- !.
separate([[E,P]|SL],[E|WS],[P|PA]) :- separate(SL,WS,PA), !.

fix_ws([S|WS],CW) :-
      correct_ws(S,WS,WS0),
      remove_bad_reentrant([S|WS0],CW),
      !.

correct_ws(E0,[E1],[E1]) :-                                  % end of ws
      (ep(E1,E0,R,_,_) ; ep(E0,E1,R,_,_)),
      !.
correct_ws(E1,[E1,E1|WS],CW) :-                         % unnecessary E1
      correct_ws(E1,[E1|WS],CW),
      !.
correct_ws(E0,[E1,E2|WS],CW) :-                         % unnecessary E1
      (ep(E0,E1,R,_,_) ; ep(E1,E0,R,_,_)),
      (ep(E1,E2,R,_,_) ; ep(E2,E1,R,_,_)),
      correct_ws(E0,[E2|WS],CW),
      !.
correct_ws(_,[E1,E2|WS],[E1|CW]) :-                       % OK, continue
      correct_ws(E1,[E2|WS],CW),
      !.
correct_ws(E,WS,CW) :-
      writeln_b('ERROR (astar)...failed to fix WS !'),
      writeln_b(E),
      writeln_b(WS),
      writeln_b(CW),
      !.

remove_bad_reentrant([E0],[E0]) :- !.
remove_bad_reentrant([E1,E0],[E1,E0]) :- !.
remove_bad_reentrant([E1,E2,E2,E3|WS],Z) :-
      ( ep(E1,E2,R1,W1,_) ; ep(E2,E1,R1,W1,_) ),
      ( ep(E2,E3,R3,W3,_) ; ep(E3,E2,R3,W3,_) ),
      ( not(same(R1,R3)) -> Z=[E1,E2|CW]                      % continuous
      | edge(E2,_,[_,W3],_,_) -> Z=[E1,E2,E2|CW]                      % OK
      | otherwise -> Z=[E1|CW] ),                                 % remove
      remove_bad_reentrant([E3|WS],CW),
      !.
remove_bad_reentrant([E1|WS],[E1|CW]) :-                    % OK, continue
      remove_bad_reentrant(WS,CW),
      !.

%-------------------------------------------------------------------------

astarsearch(Start,Goalpathlist,C) :-
      cleandatabase,
      add_state(Start,[],0.0),
      repeatifagenda,                                     % iterative loop
      pick_best_state(I,State,Pathlist,C),
      add_successors(I,State,Pathlist,C),
      agenda(I,State,Goalpathlist,C,_),            % soln at top of agenda
      !.

pick_best_state(I,State,Pathlist,C) :-
      del_min(I),                                     % delete top of heap
      agenda(I,State,Pathlist,C,_),                  % retract in add_succ
      !.

% add_successors succeeds only when best state on agenda includes goal

add_successors(_,State,_,_) :- goalreached(State), !.
add_successors(_,State,Pathlist,C) :-               % fail, add successors
      successor(State,Newstate),
      proper(Newstate,Pathlist),
      add_state(Newstate,Pathlist,C),
      fail.
add_successors(I,State,_,C) :-                    % remove cur state, fail
      retract(agenda(I,_,_,_,_)),                   % pick_best (top heap)
      asserta(usedstate(State,C)),
      fail.

proper([E0,PR2],[[E0,PR1],[E1,P1]|_]) :-                    % reflection ?
      !,                                         % fail rule if head match
      er(E0,R0,[W0,_]),
      er(E1,R1,_),
      match_r(R0,R1),
      edge(E0,_,[W0,_],_,_),                 % reflective side of edge ?
      sqr_dist(P1,PR1,D1),
      sqr_dist(P1,PR2,D2),
      D2 > D1,                                 % reasonable reflection ?
      !.
proper(_,_) :- !.   % no match to rule 1 head ==> no reflection, succeed

match_r([R,_],[_,R]) :- !.
match_r([R,_],[R,_]) :- !.
match_r([_,R],[_,R]) :- !.
match_r([_,R],[R,_]) :- !.

add_state(Newstate,Pathlist,C) :-
      cost([Newstate|Pathlist],C,Cnew),
      !,
      agenda_check(Newstate,Cnew),                   % fail if on agenda
      !,
      usedstate_check(Newstate,Pathlist,Cnew),     % fail if seen before
      !,
      eval(Newstate,Enew),
      D is Enew + Cnew,
      ins_heap(D,I),
      asserta(agenda(I,Newstate,[Newstate|Pathlist],Cnew,D)),
      update_clock(T),
      display_newstate(Newstate,Pathlist),
      tm,
      !.
add_state(Newstate,Pathlist,C) :-
      not(cost([Newstate|Pathlist],C,_)),
      write_list_b(['ERROR (astar)...cost fail',[Newstate|Pathlist]]),
      !.
add_state(Newstate,Pathlist,_) :-
      not(eval(Newstate,_)),
      write_list_b(['ERROR (astar)...eval fail',[Newstate|Pathlist]]),
      !.

% FOR DISPLAY DEBUG ON VT-100 WITHOUT PROWINDOWS
% display_newstate([_,P0],[[_,P1]|_]) :- write_list([P1,'-->',P0]).
% display_newstate([s,_],[]).

display_newstate([_,P0],[[_,P1]|_]) :-                 % refined DISPLAY
      w_show_line(P1,P0,2),
      !.
display_newstate([s,_],[]) :- !.                  % ignore refined start
display_newstate(E0,[E1|_]) :-                              % crude DISPLAY
      edge_data(E0,_,MP0),
      edge_data(E1,_,MP1),
      w_show_line(MP0,MP1,2),
      !.
display_newstate(_,[]) :- !.                            % ignore crude start

/*------------------------------------------------------------------------
 agenda_check - check for state S already on agenda;
      linear search for S which has greater cost back to start;
      remove and replace with shorter route;
      linear search to delete from heap;
------------------------------------------------------------------------*/

agenda_check(S,C) :-
      agenda(I,S,_,C2,_),
      C<C2,
      retract(agenda(I,_,_,_,_)),
      del_heap(I),
      !.
agenda_check(S,_) :- agenda(_,S,_,_,_), !, fail.
agenda_check(_,_).% else, if not on agenda, succeed and put on

usedstate_check(S,_,_) :- usedstate(S,_), !, fail.
usedstate_check(_,_,_).

repeatifagenda.                             % repeat while agenda NOT empty
repeatifagenda :- agenda(_,_,_,_,_), repeatifagenda.

cleandatabase :-
      retractall(agenda(_,_,_,_,_)),
      retractall(usedstate(_,_)),
      init_heap(_).
```

B.6 MAP MAKER

```
/*==============================================================================
 MAPMAKER.PL - mouse assisted weighted-region problem map making tool;
      create/edit simple line maps for the weighted region problem;
      capture screen points, convert to map coordinates, and store
      as edge predicates; requires special version of ProWindows
      created with particular libraries from Quintus ProWindows.
==============================================================================*/

library_directory('/usr/local/quintus/generic/qplib2.5/library').
library_directory('/usr/local/quintus/pw1.1/library').

make :- save(mm,1), go.
make :- writeln('Created executable: mm'), halt.

pixels(700).                                % number pixels in one dimension

map_size(100).                            % number map units in one dimension

:- dynamic
      weights/1,
      point/1,
      pm_ratio/1,
      ipm_ratio/1,
      pix_edge/5.

weights([1,1]).                             % initial default weight setting

go :-
      kill_prowindows,
      retractall(pix_edge(_,_,_,_,_)),
      precalc,
      make_map.

kill :- kill_prowindows.

:-
      use_module(library(basics)),
      use_module(library(dialog)),
      use_module(library(interpret_messages)),
      use_module(library(messages)),
      use_module(library(strings)),
      use_module(library(math)).

writeln(X) :- write(X), nl, !.
```

```
midpoint([X1,Y1],[X2,Y2],[XM,YM]) :-
      XM is 0.5*(X1+X2),
      YM is 0.5*(Y1+Y2),
      !.

%-----------------------------------------------------------------------

map_item(@xcoord,text_item('X: ',0,none),below,[width: 4]).
map_item(@ycoord,text_item('Y: ',0,none),below,[width: 4]).
map_item(@new_line,button('New Ln',cascade(@pic,new_line,0)),below,[]).
map_item(@mclear, button('Clear', mmclear), below,[]).
map_item(@wt1,text_item('W1: ',1,cascade(@pic,set1,0)),below,[width:2]).
map_item(@wt2,text_item('W2: ',1,cascade(@pic,set2,0)),below,[width:2]).

map_item(@blank1,label('',0),below,[]).

map_item(@lfile,text_item('Load:',map,cascade(@pic,stay,0)),
      below,[width: 8]).
map_item(@mload, button('Load', mmload), below,[]).

map_item(@blank2,label('',0),below,[]).

map_item(@sfile,text_item('Save:',map,cascade(@pic,stay,0)),
      below,[width: 8]).
map_item(@msave, button('Save', mmsave), below,[]).

map_item(@blank?,label('',0),below,[]).
map_item(@mprint, button('Print', mmprint), below,[]).
map_item(@blank4,label('',0),below,[]).
map_item(@mquit, button('Quit', mmquit), below,[]).

stay(_,_) :-
      send(@sfile,advance,none),
      send(@lfile,advance,none).

set1(_,STRW1) :-
      name(STRW1,L),
      number_chars(W1,L),
      retract(weights([_,W2])),
      assert(weights([W1,W2])),
      send(@wt1,advance,next).

set2(_,STRW2) :-
      name(STRW2,L),
      number_chars(W2,L),
      retract(weights([W1,_])),
      assert(weights([W1,W2])),
      send(@wt2,advance,previous).
```

```
ticks :-
      pixels(W),
      T is W//10,
      W0 is W-T,
      ticks2(W0,T),
      !.

ticks2(X,T) :-
      X >= 1,
      new(@TXT,text('|',X,0,center)),
      send(@TXT,font,font(screen,r,12)),
      send(@pic,display,@TXT),
      X2 is X-T,
      ticks2(X2,T),
      !.
ticks2(_,T) :-
      pixels(W),
      ticks3(W,T),
      !.

ticks3(Y,T) :-
      Y >= 1,
      AY is Y-6,
      new(@TXT,text('-',0,AY,left)),
      send(@TXT,font,font(screen,r,12)),
      send(@pic,display,@TXT),
      Y2 is Y-T,
      ticks3(Y2,T),
      !.
ticks3(_,_) :- !.

precalc :-
      pixels(P),
      map_size(C),
      G is P/C,
      IG is P//C,
      assert(pm_ratio(G)),
      assert(ipm_ratio(IG)).

%-----------------------------------------------------------------------

make_map :-
      pixels(P),
      pm_ratio(G),
      new(@pic,picture(none,size(P,P))),
      send(@pic,horizontal_scrollbar,0),
      send(@pic,vertical_scrollbar,0),
      send(@pic,smart,off),
      round(G,IG),
      send(@pic,grid,size(IG,IG)),
      new_dialog(@map_dialog,none,map_item),
      send(@map_dialog,right,@pic),
      PX is 1000-P,        % screen = 1150x900 pixels; dialog = 150 wide
      send(@map_dialog,open,point(PX,80)),
      send(@map_dialog,label,'MAP MAKER'),
      stay(_,_),
      draw_border,
      ticks,
      new(@status,text_block('',area(430,480,70,20),center)),
      send(@pic,display,@status),
      send(@pic,cursor_x,cascade(@xcoord,adjustx,0)),
      send(@pic,cursor_y,cascade(@ycoord,adjusty,0)),
      send(@pic,moved,cascade(@pic,moved_line,0)),
      send(@pic,left_down,cascade(@pic,clicked,0)),
      send(@pic,middle_click,cascade(@pic,del_line,0)),
      interpret_messages(fail).

new_line(_,_) :- retractall(point(_)).                   % start new line

del_line(_,FIG) :-
      send(FIG,destroy),
      retract(pix_edge(FIG,WTLBL,_,_,_)),
      send(WTLBL,erase).

moved_line(PIC,FIG) :-
      retract(pix_edge(FIG,WTLBL,[X1,Y1],[X2,Y2],W)),
      send(WTLBL,erase),
      get(FIG,reference_point,REF),
      send(FIG,destroy),
      REF =.. [_,XR,YR],
      NX1 is X1+XR, NY1 is Y1+YR,
      NX2 is X2+XR, NY2 is Y2+YR,
      new(@L,line(NX1,NY1,NX2,NY2)),
%     send(@L,arrows,first),                       % option for arrowheads
      new(@FIG2,figure(0,0,0)),
      send(@FIG2,append,@L),
      send(@PIC,display,@FIG2),
      wt_label([NX1,NY1],[NX2,NY2],NWTLBL,W),
      asserta(pix_edge(@FIG2,NWTLBL,[NX1,NY1],[NX2,NY2],W)).

wt_label([X1,Y1],[X2,Y2],@WTLBL,W) :-
      midpoint([X1,Y1],[X2,Y2],[XM,YM]),
      round(XM,IXM),
      round(YM,IYM),
      weights(W),
      place_lbl(W,WSTR),
      new(@WTLBL,text(WSTR,IXM,IYM,center)),
      send(@WTLBL,font,font(cour,b,10)),
      send(@pic,display,@WTLBL),
      !.
```

```
place_lbl([W,W],WSTR) :- number_chars(W,STR), name(WSTR,STR), !.
place_lbl(W,WSTR) :- concat_atom(W,:,WSTR), !.

round_pix(POS,X,Y) :-
      pm_ratio(G),
      round(G,IG),
      get(POS,x,X0), get(POS,y,Y0),
      X1 is X0/G, Y1 is Y0/G,
      round(X1,X2), round(Y1,Y2),
      X is X2*IG, Y is Y2*IG,
      !.

clicked(PIC,@POS) :-
   (retract(point(LASTPT)) ->
      round_pix(@LASTPT,X1,Y1),
      round_pix(@POS,X2,Y2),
      new(@L,line(X1,Y1,X2,Y2)),
      send(@L,arrows,first),
      new(@FIG,figure(0,0,0)),            % mouse cannot find simple line
      send(@FIG,append,@L),                % so, must append to a figure.
      send(@PIC,display,@FIG),
      wt_label([X1,Y1],[X2,Y2],WTLBL,W),
      asserta(pix_edge(@FIG,WTLBL,[X1,Y1],[X2,Y2],W)),
      assert(point(POS))
      ;
      assert(point(POS))
   ).

convert_pt([PX,PY],[X,Y]) :-
      pixels(W),
      pm_ratio(G),
      XX is PX/G,
      YY is (W-PY)/G,
      round(XX,X),
      round(YY,Y).

convert_pix([X,Y],[PX,PY]) :-
      pixels(W),
      pm_ratio(G),
      TPX is G*X,
      TPY is W-G*Y,
      round(TPX,PX),
      round(TPY,PY).

adjustx(XITEM,X) :-                  % display transformed x in dialog box
      send(@status,string,''),                           % clear status line
      pm_ratio(R),
      XX is X*10/R,
      floor(XX,RX),
      send(@XITEM,selection,RX).
```

```
adjusty(YITEM,Y) :-                   % display transformed y in dialog box
      pm_ratio(R),
      map_size(M),
      YY is (10*M)-Y*10/R,
      floor(YY,RY),
      send(@YITEM,selection,RY).

mmsave(_,_) :-
      send(@status,string,'SAVING'),
      get(@sfile,string,FILE),
      send(@sfile,advance,none),
      open(FILE,write,S),
      tell(S),
      pix_to_coord,
%     insert_border,
      told,
      send(@status,string,'READY').

concat(S1,S2,S3) :-
      name(S1,L1),
      name(S2,L2),
      append(L1,L2,L3),
      name(S3,L3).

pix_to_coord :-
      nl,
      writeln(':- dynamic mapname/1, edge/6.'), nl,
      get(@sfile,string,MAP),
      concat('mapname(',MAP,M1),
      concat(M1,').',M2),
      writeln(M2), nl,
      pix_edge(F,W,[PX1,PY1],[PX2,PY2],[W1,W2]),
      convert_pt([PX1,PY1],[CX1,CY1]),
      convert_pt([PX2,PY2],[CX2,CY2]),
      EDGE =.. [edge,CX1,CY1,CX2,CY2,W1,W2],
      portray_clause(EDGE),
      fail.
pix_to_coord.

insert_border :-
      nl, writeln('% border edges'), nl,
      b_edge(X1,Y1,X2,Y2,W1,W2),
      portray_clause(edge(X1,Y1,X2,Y2,W1,W2)),
      fail.
insert_border :- nl.

draw_border :-
      b_edge(X1,Y1,X2,Y2,_,_),
      convert_pix([X1,Y1],[PX1,PY1]),
      convert_pix([X2,Y2],[PX2,PY2]),
      new(@L,line(PX1,PY1,PX2,PY2)),
      send(@pic,display,@L),
      fail.
draw_border.

b_edge( 0, 0,100, 0,1,0).
b_edge(100, 0,100,100,1,0).
b_edge(100,100, 0,100,1,0).
b_edge( 0,100, 0, 0,1,0).

mmprint(_,_) :-
      send(@pic,postscript,picfile),
      unix(system(’lpr -Pss1 picfile’)),
      unix(system(’rm picfile’)),
      unix(system(’lpq -Pss1’)),
      send(@status,string,’READY’).

mmclear(_,_) :-
      retractall(pix_edge(_,_,_,_,_)),
      send(@pic,clear),
      retractall(edge(_,_,_,_,_,_)),
      send(@status,string,’ ’),
      send(@pic,display,@status).

mmload(_,_) :-
      send(@status,string,’LOADING’),
      get(@lfile,string,FILE),
      send(@lfile,advance,none),
      consult(FILE),
      draw_edges,
      abolish(edge/6),
      send(@status,string,’READY’),
      !.

draw_edges :-
      edge0(X1,Y1,X2,Y2,W1,W2),
      retractall(weights(_)),
      asserta(weights([W1,W2])),
      convert_pix([X1,Y1],[PX1,PY1]),
      convert_pix([X2,Y2],[PX2,PY2]),
      new(@L,line(PX1,PY1,PX2,PY2)),
      send(@L,arrows,first),
      new(@FIG,figure(0,0,0)),
      send(@FIG,append,@L),
      send(@pic,display,@FIG),
      wt_label([PX1,PY1],[PX2,PY2],WTLBL,_),
      assert(pix_edge(@FIG,WTLBL,[PX1,PY1],[PX2,PY2],[W1,W2])),
      fail.
draw_edges :- !.

mmquit(_,_) :- send(@map_dialog,destroy), halt.
```

LIST OF REFERENCES

[Aart85] Aarts, E.H.L. and van Laarhoven, P.J.M., "A New Polynomial-Time Cooling Schedule," Proceedings of the International Conference on Computer-Aided Design, pp. 206-208, IEEE, November 1985.

[Aart89] Aarts, E.H.L and Korst, J., *Simulated Annealing and Boltzmann Machines*, John Wiley and Sons, 1989.

[Adam85] Adams, M., Deutsch, O., and Harrison, J., "A Hierarchical Planner for Intelligent Systems," *Proceedings of SPIE Applications of Artificial Intelligence II*, vol. 548, pp. 207-218, International Society for Optical Engineering, Bellingham, WA, 9-11 April 1985.

[Akma87] Akman, V., *Unobstructed Shortest Paths in Polyhedral Environments*, pp. 90-92, Springer-Verlag, 1987.

[Alex90] Alexander, R.S. and Rowe, N.C., "Path Planning by Optimal-Path-Map Construction for Homogeneous-Cost Two-Dimensional Regions," *Proceedings of the IEEE International Conference on Robotics and Automation*, Cincinnati, OH, May 1990.

[Barr81] Barr, A. and Feigenbaum, E.A., *The Handbook of Artificial Intelligence*, vol. 1, Morgan-Kaufman, 1981.

[Baza77] Bazaraa, M.S. and Jarvis, J.J., *Linear Programming and Network Flows*, John Wiley and Sons, Inc., 1977.

[Bono84] Bonomi, E. and Lutton, J., "The N-City Travelling Salesman Problem: Statistical Mechanics and the Metropolis Algorithm," *SIAM Review*, vol. 26, no. 4, pp. 551-568, Society for Industrial and Applied Mathematics, October 1984.

[Bras88] Brassard, G. and Bratley, P., *Algorithmics Theory and Practice*, p. 247, Prentice Hall, 1988.

[Carr90] Carriker, W.F., Khosta, P.F., and Krogh, B.H., "The Use of Simulated Annealing to Solve the Mobile Manipulator Path Planning Problem," *Proceedings of the IEEE International Conference on Robotics and Automation*, vol. 1, pp. 204-209, Cincinnati, OH, May 1990.

[Cern85] Cerny, V., "A Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm," *Journal of Optimization Theory and Application*, vol. 45, pp. 41-45, 1985.

[Char87] Charniak, E. and McDermott, D., *Introduction to Artificial Intelligence*, p. 435, Addison-Wesley Publishing Co., 1987.

[Clev84] Cleva, S.J., "High-Tech Maps," *Army*, pp. 33-38, November, 1984.

[Coll88] Collins, N.E., Eglese, R.W., and Golden, B.L., "Simulated Annealing - An Annotated Bibliography," *American Journal of Mathematical and Management Sciences*, vol. 8, nos. 3 and 4, pp. 209-307, American Sciences Press, Inc., 1988.

[Defe74] Defense Mapping Agency, *Hunter-Liggett Special*, edition 1-DMATC 1:50000, stock no. V795SHUNTERLI, Topograhic Center, Washington, D.C., 1974.

[Dent84] Denton, R.V. and Froeberg, P.L., "Application of Artificial Intelligence in Automated Route Planning," *Proceedings of SPIE Applications of Artificial Intelligence*, vol. 485, pp. 126-132, International Society for Optical Engineering, Arlington, VA, May 1984.

[Digi88] *Digitizing the Future*, 2d ed., DMA stock no. DDIPDIGITALPAC, Defense Mapping Agency, 1988.

[Edel89] Edelsbrunner, H., *Algorithms in Combinatorial Geometry*, pp. 244-245, Springer-Verlag, 1987.

[Garc89] Garcia, I., *Solving the Weighted Region Least Cost Path Problem Using Transputers*, M.S. Thesis, Naval Postgraduate School, Monterey, CA, December 1989.

[Gree84] Greene, J. and Supowitz, K., "Simulated Annealing without Rejected Moves," *International Conference on Computer Design*, pp. 658-663, IEEE, October 1984.

[Grov86] Grover, L.K., "A New Simulated Annealing Algorithm for Standard Cell Placement," *Proceedings of the International Conference on Computer-Aided Design*, pp. 378-380, IEEE, 1986.

[Haje85] Hajek, B., "A Tutorial Survey of Theory and Applications of Simulated Annealing," *Proceedings of the 24th Conference on Design and Control*, pp. 755-760, December 1985.

[Huan86] Huang, M.D., Romeo, F., and Sangiovanni-Vincentelli, A., "An Efficient General Cooling Schedule for Simulated Annealing," *Proceedings of the International Conference on Computer-Aided Design*, pp. 381-384, IEEE, 1986.

[Iyen89] Iyengar, S.S. and Kashyap, R.L., "Autonomous Intelligent Machines," *Computer*, vol. 22, no. 6, pp. 14-15, IEEE, June 1989.

[John89] Johnson, D.S., Aragon, C.R., McGeoch, L.A., and Schevon, C., "Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning," *Operations Research*, vol. 37, no. 6, pp. 865-892, November-December 1989.

[John90] Johnson, D.S., Aragon, C.R., McGeoch, L.A., and Schevon, C., "Optimization by Simulated Annealing: An Experimental Evaluation; Part II, (Graph Coloring and Number Partitioning)," to be published in *Operations Research*, 1990.

[Keir84] Keirsey, D.M., Mitchell, J.S.B., Payton, D.W., and Preyes, E.P., "Multilevel Path Planning for Autonomous Vehicles," *Proceedings of SPIE Applications of Artificial Intelligence*, vol. 485, pp. 133-137, International Society for Optical Engineering, Arlington, VA, May 1984.

[Keir88] Keirsey, D., Payton, D.W., and Rosenblatt, J., "Autonomous Navigation in Cross-Country Terrain," *Proceedings of DARPA Image Understanding Workshop*, pp. 411-416, Defense Advanced Research Projects Agency, 1988.

[Kirk83] Kirkpatrick, S., Gelatt, C.D., and Vecchi, M.P., "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671-680, 13 May 1983.

[Lewi88] Lewis, D.H., *Optimal Three-Dimensional Path Planning Using Visibility Constraints*, M.S. Thesis, Naval Postgraduate School, Monterey, CA, May 1988.

[Lin65] Lin, S., "Computer Solutions of the Traveling Salesman Problem," *Bell System Technical Journal*, vol. 44, no. 10, pp. 2245-2269, Bell Laboratories, December 1965.

[Lind87] Lindsay, C., "Automatic Planning of Safe and Efficient Robot Paths using Octree Representation of Configuration Space," paper presented at the IEEE International Conference on Robotics and Automation, Raleigh, NC, 2 April 1987.

[Lund86] Lundy, M. and Mees, A., "Convergence of an Annealing Algorithm," *Mathematical Programming*, vol. 34, pp. 111-124, North-Holland, 1986.

[Metr53] Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., and Teller, E., "Equation of State Calculations by Fast Computing Machines," *Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087-1092, June 1953.

[Mitc84] Mitchell, J.S.B. and Keirsey, D.M., "Planning Strategic Paths Through Variable Terrain Data," *Proceedings of SPIE Applications of Artificial Intelligence*, vol. 485, pp. 172-179, International Society for Optical Engineering, Arlington, VA, May 1984.

[Mitc87] Mitchell, J.S.B., "Shortest Paths Among Obstacles, Zero-Cost Regions, and Roads," Technical Report No. 764, School of Operations Research and Industrial Engineering, Cornell University, December 1987.

[Mitc90] Mitchell, J.S.B. and Papadimitriou, C.H., "The Weighted Region Problem: Finding Shortest Paths Through a Weighted Planar Subdivision," Technical Report No. 885, School of Operations Research and Industrial Engineering, Cornell University, January 1990, to appear in *Journal of the ACM*.

[Naha85] Nahar, S., Sahni, S., and Shragowitz, E., "Experiments with Simulated Annealing," *Proceedings of the 22rd Design Automation Conference*, pp. 748-752, IEEE, 1985.

[Naha86] Nahar, S., Sahni, S., and Shragowitz, E., "Simulated Annealing and Combinatorial Optimization," *Proceedings of the ACM/IEEE 23rd Design Automation Conference*, pp. 293-299, June 1986.

[Otte89] Otten, R.H.J.M. and van Ginneken, L.P.P.P, *The Annealing Algorithm*, Kluwer Academic Publishers, 1989.

[Papa82] Papadimitriou, C.H. and Steiglitz, K., *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[Papa85] Papadimitriou, C.H., "An Algorithm for Shortest-Path Motion in Three Dimensions," *Information Processing Letters*, vol. 20, p. 259-263, North-Holland, 15 June 1985.

[Pear84] Pearl, J., *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley Publishing Co., Reading, MA, 1984.

[Prep87] Preparata, F.P. and Shamos, M.I., *Computational Geometry: An Introduction*, Springer-Verlag, 1988.

[Pres88] Press, W.H., Flannery, B.P., Teukolsky, S.A., and Wetterling, W.T., *Numerical Recipes in C*, Press Syndicate of the University of Cambridge, 1988.

[Quin90] *Quintus Prolog Reference Manual*, Release 2.5, Quintus Computer Systems, Inc., Mountain View, CA, January 1990.

[Rich87] Richbourg, R.F., *Solving a Class of Spatial Reasoning Problems: Minimal Cost Planning in the Cartesion Plane*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, CA, June 1987.

[Rome84a] Romeo, F. and Sangiovanni-Vincentelli, A., "Probabilistic Hill-Climbing Algorithms: Properties and Applications," *ERL College of Engineering*, University of California, Memorandum No. UCB/ERL M84/34, 13 March 1984.

[Rome84b] Romeo, F., Sangiovanni-Vincentelli, A., and Sechen, C., "Research on Simulated Annealing at Berkeley," *Proceedings of the International Conference on Computer Design*, pp. 652-657, IEEE, October 1984.

[Ross89] Ross, R.S., *Optimal Grid-Free Path Planning Across Arbitrarily-Contoured Terrain With Anisotropic Friction and Gravity Effects*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, CA, June 1989.

[Rowe88] Rowe, N.C., *Artificial Intelligence Through Prolog*, p. 321, Prentice-Hall, Inc., 1988.

[Rowe89] Rowe, N.C. and Lewis, D.H., "Vehicle Path-Planning Using Optics Analogs for Optimizing Visibility and Energy Cost," *NASA Jet Propulsion Laboratory Publication 89-7*, vol. IV, pp. 217-226, NASA Conference on Space Telerobotics, Pasadena CA, January 1989.

[Rowe90a] Rowe, N.C., "Roads, Rivers, and Obstacles: Optimal Two-Dimensional Route Planning Around Linear Features for a Mobile Agent," *International Journal of Robotics Research*, vol. 9, no. 6, December 1990.

[Rowe90b] Rowe, N.C. and Richbourg, R.F., "An Efficient Snell's-Law Method for Optimal-Path Planning Across Multiple Two-Dimensional Irregular Homogeneous-Cost Regions," *International Journal of Robotics Research*, 6 December 1990.

[Rowe90c] Rowe, N.C. and Ross, R.S., "Optimal Grid-Free Path Planning Across Arbitrarily-Contoured Terrain with Anisotropic Friction and Gravity Effects," *IEEE Transactions on Robotics and Automation*, vol. 6, no. 5, pp. 540-553, October 1990.

[Rowe90d] Rowe, N.C., "Plan Fields and Real-World Uncertainty," *AAAI Workshop on Planning in Uncertain, Unpredictable, or Changing Environments*, Stanford, CA, March 1990.

[Rowe90e] Rowe, N.C., "Construction of Optimal-Path Maps for High-Level Path Planning across Weighted Regions," Computer Science Dept., Naval Postgraduate School, in preparation, December 1990.

[Sech86] Sechen, C. and Sangiovanni-Vincentelli, A., "TimberWolf 3.2: A New Standard Cell Placement and Global Routing Package," *Proceedings of the 23rd Design Automation Conference*, pp. 432-439, ACM/IEEE, June 1986.

[Smit88] Smith, T.R., Peng, G., and Gahinet, P., "A Family of Local, Asynchronous, Iterative, and Parallel Procedures for Solving the Weighted Region Least Cost Path Problem," University of California at Santa Barbara, 20 April 1988.

[Tove88] Tovey, C.A., "Simulated Simulated Annealing," *American Journal of Mathematical and Management Sciences*, vol. 8, nos. 3 and 4, pp. 389-407, American Sciences Press, Inc., 1988.

[VanL87] Van Laarhoven, P.J.M and Aarts, E.H.L, *Simulated Annealing: Theory and Applications*, D. Reidel Publishing Co., 1987.

[Vecc83] Vecchi, M.P. and Kirkpatrick, S., "Global Wiring by Simulated Annealing," *IEEE Transactions on Computer-Aided Design*, vol. CAD-2, no. 4, pp. 215-222, October 1983.

[Whit84] White, S., "Concepts of Scale in Simulated Annealing," *Proceedings of the International Conference on Computer Design*, pp. 646-651, IEEE, November 1984.

[Wilb89] Wilber, R., "Expert Systems Aid On-Board Mission Management," *Defense Computing*, vol. 2, no. 1, pp. 27-30, Cardiff Publishing Co., January/February 1989.

[Wren90] Wrenn, L.R. III, *Three-Dimensional Route-Planning for a Cruise Missile for Minimal Detection by Observers*, Technical Report NPS52-90-028, Department of Computer Science, Naval Postgraduate School, Monterey, CA, May 1990.

[Yen71] Yen, J.Y., "Finding the *K* Shortest Loopless Paths in a Network", *Management Science*, vol. 17, pp. 712-716, 1971.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
Cameron Station
Alexandria, VA 22304-6145

2. Library, Code 52 2
Naval Postgraduate School
Monterey, CA 93943-5002

3. Man-Tak Shing, Code CS/Sh 2
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943

4. Neil C. Rowe, Code CS/Rp 2
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943

5. Vincent Lum, Code CS/Lm 1
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943

6. Walter M. Woods, Code OR/Wo 1
Department of Operations Research
Naval Postgraduate School
Monterey, CA 93943

7. Clyde Scandrett, Code MA/Sc 1
Department of Mathematics
Naval Postgraduate School
Monterey, CA 93943

8. Robert E. McGhee, Code CS/Mz 1
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943

9. MAJ(P) Mark R. Kindl 3
3560 Eagle Landing Drive
Lithonia, GA 30058

617-576

Thesis
K42455 Kindl
c.1 A stochastic approach
to path planning in the
Weighted-Region Problem.