1-1-2003

Trajectory planning in the presence of risk regions

Michael Allan Sherwood
University of Nevada, Las Vegas

TRAJECTORY PLANNING IN THE PRESENCE OF RISK REGIONS

by

Michael Allan Sherwood

Bachelor of Science
University of Nevada, Las Vegas
1998

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
December 2003

UNLV
UNIVERSITY OF NEVADA LAS VEGAS

Thesis Approval

The Graduate College
University of Nevada, Las Vegas

November 19, 20 03

The Thesis prepared by

Michael Allan Sherwood

Entitled

Trajectory Planning in the Presence of Risk Regions

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative

ABSTRACT

Trajectory Planning in the Presence of Risk Regions

by

Michael Allan Sherwood

Dr. Laxmi P. Gewali, Examination Committee Chair
Professor of Computer Science
University of Nevada, Las Vegas

We consider the problem of developing fast algorithms for computing short collision-free paths for aerial vehicles in the presence of obstacles and enemy radar installations. When aerial vehicles are deployed in such regions, it is critical to compute admissible paths having reduced exposure to threats. The generalized version of this problem is known to be NP-hard [4]. We consider simplified versions in two dimensions. One of the specific problems we address is to adjust a given k-legged trajectory to reduce exposure to threats. We also propose an algorithm to compute a k-legged risk-reduced path using a stage graph which runs in $O(m^2p)$ time where m is the number of vertices per turn region and p is the number of radar and obstacle polygon edges. In addition, we describe methods for obtaining multiple risk-reduced paths and node disjoint paths from the stage graph. We also discuss the implementation of the proposed algorithms.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank Dr. Laxmi Gewali for the tireless guidance, encouragement, and assistance he provided while serving as my examination committee chair.

I would also like to thank Dr. Thomas Nartker for his wonderful pedagogy and ability to weave the threads of knowledge, Dr. Wolfgang Bein for engendering in me a quest for fast non-optimal results, and Dr. Diane Pyper Smith for teaching me how to see farther than I ever have before and yet survive the shock of the experience. I am indebted to all three for graciously agreeing to serve on my examination committee.

Finally, I would like to thank Bonn Corporation for the inspiration for this thesis and for their continued support of research at UNLV.

CHAPTER 1

INTRODUCTION

We consider the problem of computing risk-reduced paths in a two-dimensional plane containing obstacles, free regions, and risk regions. Obstacles are the regions where traversal is forbidden, and risk regions are the regions where a certain risk factor is associated with traversal. There is no risk when traversal is performed in free regions. In computational geometry and operations research literature, free regions, risk regions, and obstacles are also known as 0-regions, 1-regions, and ∞-regions, respectively [8]. In real world situations, these problems arise when planning risk-reduced paths for moving aerial vehicles in Euclidean space containing enemy radar installations. The regions visible to radar sources can be considered as risk regions (or 1-region) where risk is measured by associating a certain finite risk per unit length of the path. The regions where traversal is forbidden are ∞-regions. Obstacles such as mountains or political boundaries that must be avoided by aerial vehicles are examples of ∞-regions. The obstacle-free regions which are not visible to enemy radar sources are free regions or 0-regions.

Computation of collision-free paths connecting given points, in the presence of obstacles, is an important problem in robotics and computational geometry with applications in motion planning, geographic information systems, VLSI design, and defense systems [1, 5, 8, 13]. Efficient algorithms are known to solve this problem in two

dimensions and the problem becomes very hard in three dimensions. In fact, the problem of computing shortest paths in three dimensions is known to be intractable [8]. One of the key reasons why the computation of shortest collision-free paths in three dimensions is hard is the fact that the shortest path can touch polyhedral obstacles at any point in the interior of an edge. In order to develop an approximation algorithm in three dimensions, an edge subdivision approach was reported in [12].

Planning risk-reduced paths in the presence of obstacles and threats can be viewed as a version of the weighted region problem [7, 9]. In a weighted region problem, we are required to find a shortest or least cost path connecting a source point s and a target point t in two-dimensional space consisting of k types of regions. A weight w_i is associated with *type-i* region. The weight w_i represents the cost of traversal per unit length in *type-i* region. Weighted region problems occur in real world applications where we are required to plan a path for robotic vehicles in a surface consisting of roads, sand, water, obstacles, etc. Finding a shortest path in a weighted two-dimensional region is a rather difficult problem. Mitchell and Papadimitriou [10] presented an algorithm that computes a path between two points in a weighted planar subdivision which is at most $(1 + \varepsilon)$ times the shortest path. The execution time of this algorithm is $O(n^8L)$ where n is the number of vertices in the weighted planar region and L represents the bit complexity of the problem instance. An $O(n^3k)$ algorithm for this problem was later reported independently in [9] and [10]. These results are of theoretical interest and are difficult for practical implementation. A more practical approximation algorithm for computing shortest-paths in weighted regions is based on the idea of introducing Steiner vertices on the edges of regions [12]. A theoretical framework based on edge subdivision was first proposed by

Papadimitriou [12]. It took more than 10 years to actually evaluate the performance of the edge-subdivision method for determining an approximate solution for the weighted region problem [9].

Computation of the shortest path in metrics other than the standard Euclidean metric has also been explored. Chen, Klenk, and Tu [5] have developed an algorithm for performing rectilinear shortest path queries in weighted regions. This algorithm reports rectilinear shortest paths in $O(\log\ n)$ time after building a data structure in a preprocessing step that requires $O(n\ \log^{3/2} n)$ time and $O(n\ \log\ n)$ space.

Eppstein has reported an algorithm for finding the k shortest paths in a digraph in $O(m + n\ \log\ n + k)$ time for n vertices and m edges in [6] where he also lists prior literature and applications on the topic.

Wang and Agarwal detail approximation algorithms for shortest paths that are subject to curvature constraints [14]. Boroujerdi and Uhlmann presented an algorithm for computing least cost paths under turn angle constraints with performance similar to Dijkstra's algorithm [2].

In this paper we consider the problem of computing a risk-reduced path connecting source point s and target point t that must pass through exactly k leg points in the presence of threat regions created by enemy radar installations. The goal is to compute a risk-reduced path whose leg-points are constrained to stay within specified regions. Such a path is referred to as a k-legged risk-reduced path. In Chapter 2, we first consider preliminaries related to path planning in two dimensions in the presence of threats and obstacles. Next we describe an algorithm for constructing a k-legged risk-reduced path. The algorithm is developed for a simple model of weighted regions where only three

types of regions are distinguished: (i) free regions, (ii) risk regions, and (iii) obstacles. The proposed algorithm runs in $O(m^2p)$ time where m is the number of vertices per turn region and p is the number of radar and obstacle polygon edges. We also consider issues related to implementation of k-legged risk-reduced path construction. In mission planning, alternate routes are desirable to add variability that reduces the ability of the enemy to predict the ultimate approach of the vehicle. In Chapter 3, we develop an algorithm for constructing multiple risk-reduced k-legged paths. In Chapter 4, we present an implementation of our algorithms and related experimental results. We give concluding remarks in Chapter 5 and also discuss possible extensions and variations of risk-reduced path construction problems.

CHAPTER 2

ALGORITHM FOR RISK-REDUCED SHORTEST PATH

One of the most important issues in path planning is the modeling of obstacles, free regions, and risk regions. The obstacles and risk regions can be modeled by simple polygons. The number of vertices used for modeling obstacles and risk space has a direct effect on the running time of the resulting path planning algorithm—the smaller the number of vertices, the faster the execution time. However, representing an obstacle with a very small number of vertices may reduce the quality of the generated solution. It is therefore very important to appropriately approximate obstacles and risk regions so that the complexity of the representation is not large and at the same time the quality of the generated solution is within an acceptable range for practical application. Approximation of a polygon with a large number of vertices with a polygon with fewer vertices has been investigated by researchers in the computational geometry community [1, 11]. These techniques could be useful for simplifying the complexity of the representation in path planning applications.

For determining risk regions induced by the presence of enemy radar installations, it is useful to apply the notion of weighted regions. Consider four radar sources with indicated angular and distance range as shown in Figure 1. This figure shows that four types of weighted regions are formed by the presence of radar sources as indicated. It may be noted that 0-weighted regions are those regions in the background that are not

visible to any radar sources. The figure does not show obstacles. If obstacles are included, then they become the ∞-weighted regions. Furthermore, obstacles can create a sequence of radar beams from a single radar source.

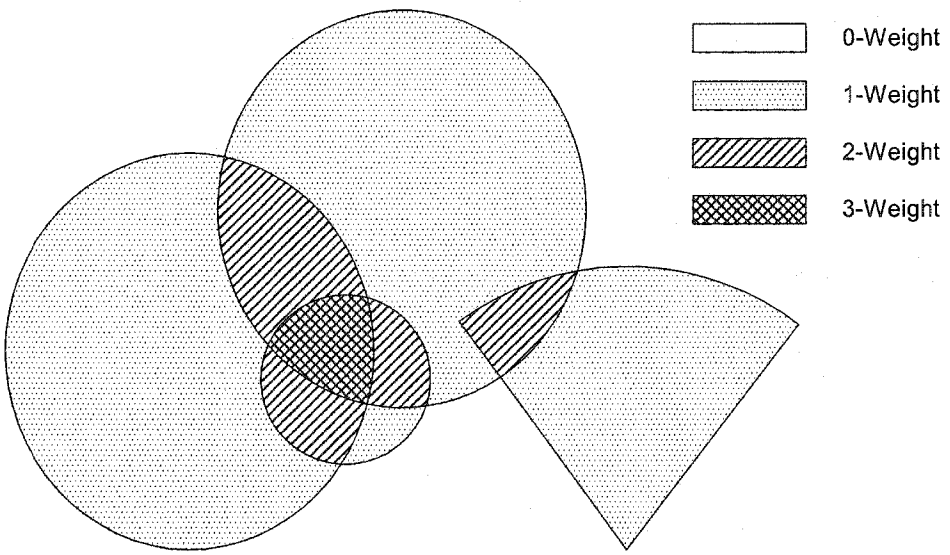

Figure 1. Formation of Weighted Regions

As indicated in the introduction section, the problem of computing shortest paths in a general weighted region is very difficult and the implementation of the available algorithms may not be acceptable in situations where solutions obtained late are useless.

Simplified Formulation of Weighted Regions

One way of reducing the execution time of path planning problems in weighted regions is to make simplifications in the modeling of the regions. In this regard, it is possible to achieve substantial simplification by considering all regions visible to one or more radar sources as risk regions by assigning certain average risk per unit length. With

this simplification, the general weighted region becomes 0/1/∞-weighted regions. Risk-reduced paths in 0/1/∞-weighted regions can be captured by constructing critical graphs [5]. The critical graph is constructed by making use of simple local optimality criteria.

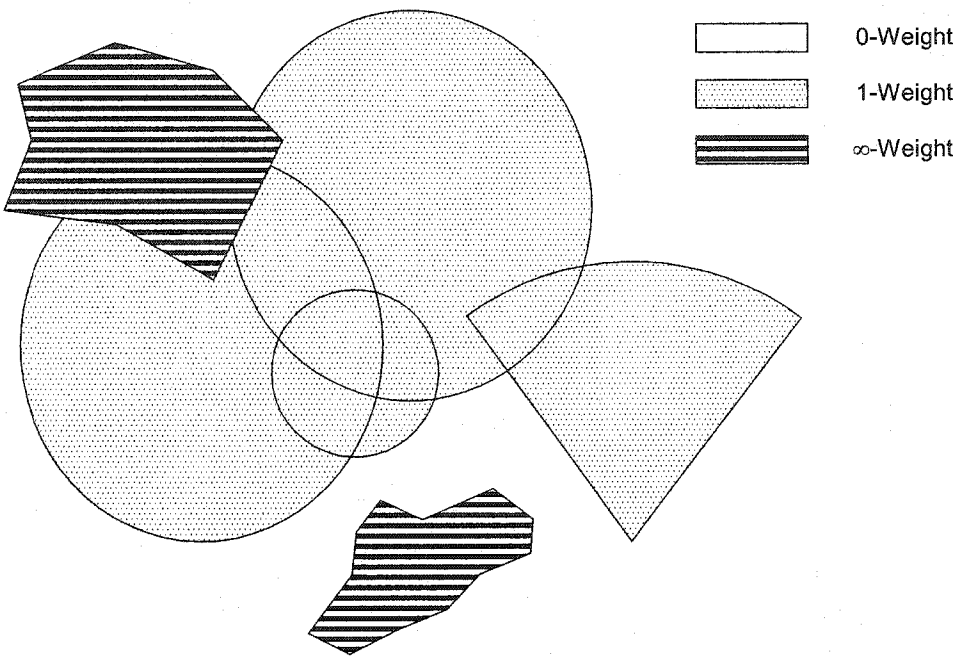

Figure 2. Formation of Simplified Weighted Regions

An edge formed by connecting two vertices in 0/1/∞-regions and lying entirely in a 1-region is called visibility edge. Similarly, an edge starting from a vertex and perpendicular to one of the edges of the 0-regions and lying entirely in 1-regions is called an orthogonal edge. A graph obtained by adding visibility edges and orthogonal edges to 0/1/∞-weighted regions is called the critical graph. It may be noted here that the critical graph also contains the edges of the boundaries of 0-regions and ∞-regions. It is known that the shortest path in 0/1/∞-weighted regions is contained in the critical graph [7].

Furthermore, the critical graph can be constructed in $O(n \log n + k)$ time, where n is the number of vertices in all regions and k the size of the visibility graph [1, 11] induced by 0-regions and ∞-regions. The shortest path itself can be computed by using Dijkstra's algorithm on the critical graph.

Consider a collection of p radar sources in a two-dimensional plane. We want to construct a path connecting a start point s and a target point t having vertices on k leg regions $r_1, r_2, \ldots, r_k$ such that exposure of the path to radar sources is as small as possible. The problem can be formally stated as follows:

K-legged Risk-Reduced Path Problem (KRPP)

Given: A collection of p radar sources, q obstacles, a sequence of k leg regions, a start point s, and a target point t.

Question: Find a $k + 1$ segment path connecting s and t with minimum exposure to radar sites. The path must have exactly one vertex in each leg region.

A radar source has a specific angular range and distance range. For the vast majority of radars, the angular range or azimuth coverage is 360 degrees. A circular or wedge-shaped polygon of certain angular range and radius can be conveniently used as a simplified model of a radar-exposed region. Figure 3 shows a path connecting s and t with exactly one leg-vertex in each leg-region. In this figure, radar-exposed regions are approximated by circular polygons.

Each leg region contains vertices that can be used as a leg vertex of the path. A brute force method of computing a risk-reduced path is to construct all paths connecting s and t and having exactly one vertex in each leg region and report the one with the least weight

as the required risk-reduced path. However, this approach is not practical for 10 or more leg-regions since the number of possible feasible paths grows exponentially with the number of leg regions.

The use of a critical graph can yield a risk-reduced path, but the path obtained by using critical graphs need not have exactly $k + 1$ segments. In fact, the number of segments in the risk-reduced path obtained by using a critical graph can be arbitrarily larger or arbitrarily smaller than $k + 1$.

KRPP Algorithm Development

In this section we now propose an efficient algorithm for solving the k-legged risk-reduced path problem (KRPP). In the rest of the section, unless stated otherwise, we simply use the term "shortest k-legged path" to indicate the risk-reduced k-legged path. The overall algorithm can be described in three distinct stages. In the first stage, the input scenario is examined to construct a much smaller region called a strip polygon, which is guaranteed to contain the shortest k-legged path. In the second stage, intersection points are identified between all potential leg edges and the boundaries of radar polygons and obstacles contained within the strip polygon. In the third stage, the intersection points are sorted in preparation for the calculation of segment weights. Next, a weighted stage graph is constructed by associating weights to leg edges—the weight of a leg edge is the risk associated with it. In the final stage, the shortest k-legged path is determined by invoking Dijkstra's shortest path algorithm on the stage graph.

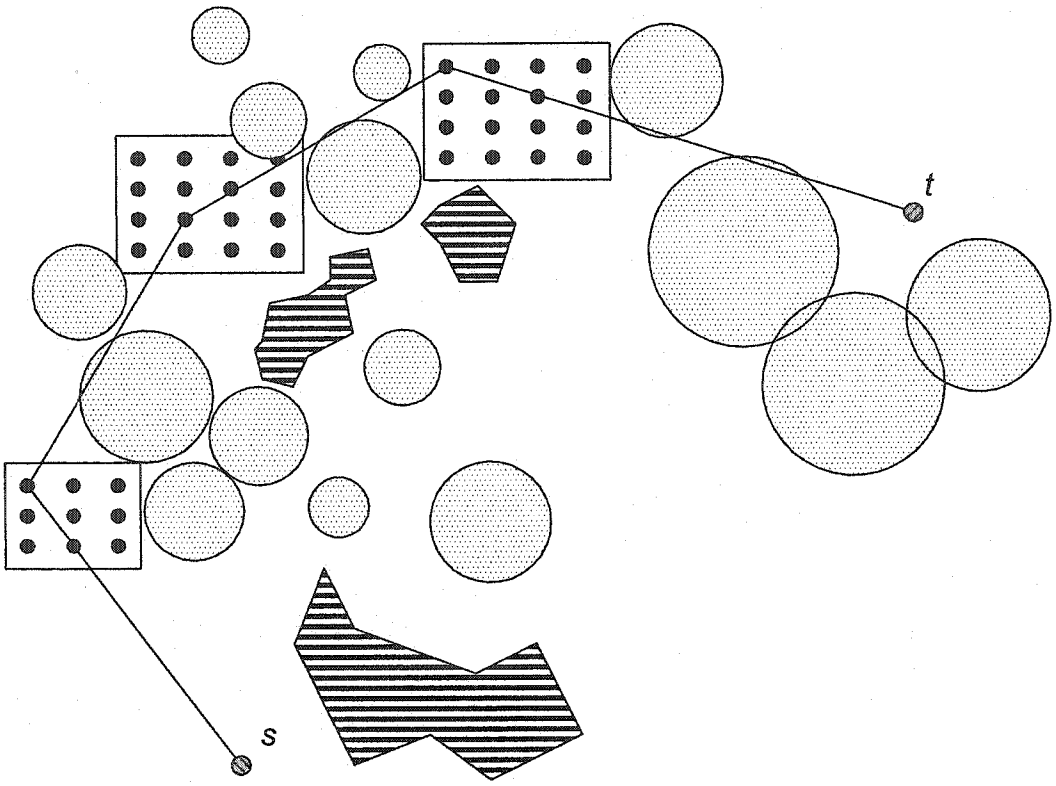

Figure 3. *K*-Legged Risk-Reduced Path

Construction of the Strip Polygon

The region that contains the shortest k-legged path can be made much smaller than the whole convex region containing radar polygons, leg regions, source point s, and target point t. Imagine the supporting lines from source point s to the first leg region r_1. Similarly, we can consider supporting lines between two consecutive leg regions r_i and r_{i+1}. These supporting lines together define the boundary of a simple polygon, which we refer to as the strip polygon.

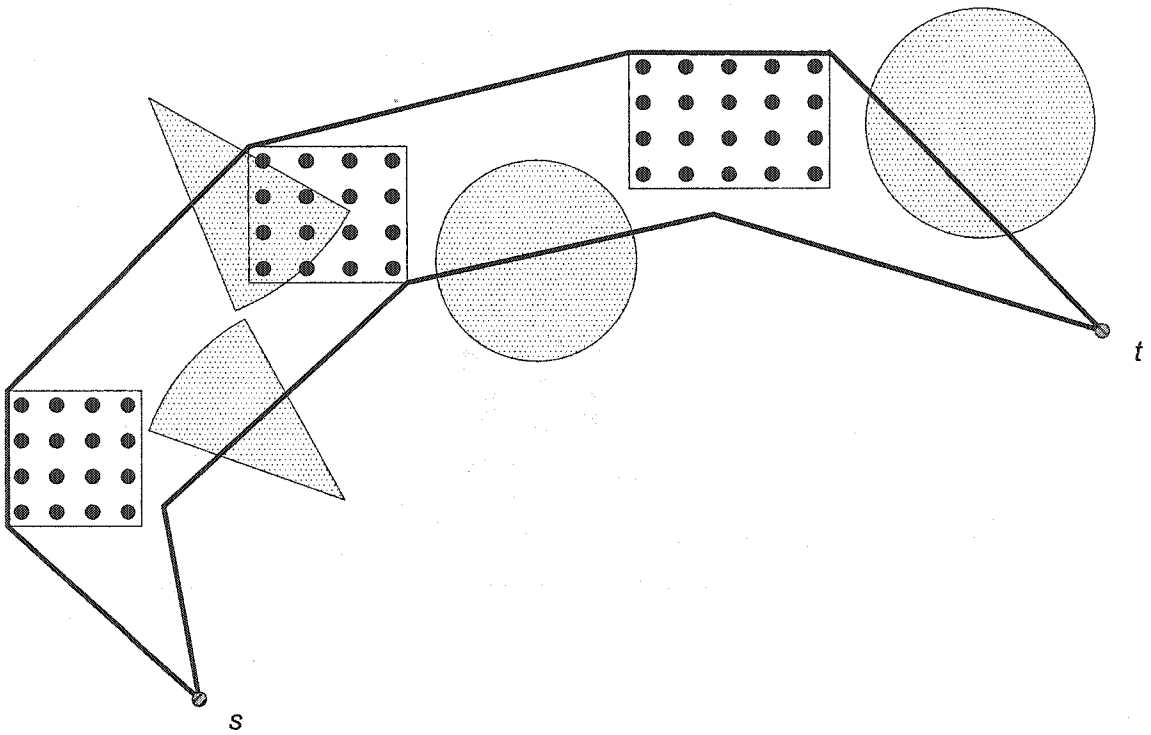

Figure 4. Strip Polygon

Figure 4 shows a strip polygon. The strip polygon can be constructed in a straightforward way in $O(n)$ time, where n is the number of vertices in the input scenario. This is stated in the following lemma:

Lemma 1: Given a collection of radar polygons and leg regions, the induced strip polygon can be computed in $O(n)$ time.

Computation of Intersection Points

Leg edges are formed by appropriately connecting s, t, and the vertices of leg regions $r_1, r_2, \ldots, r_k$. We can consider s and t as leg regions r_0 and r_{k+1} containing one vertex each. A leg edge e is formed by connecting vertices of consecutive leg regions in the sequence $r_0, r_1, \ldots, r_{k+1}$. To compute the points of intersection between leg edges and radar

polygons, we can sweep the strip polygon by a vertical line by using the plane sweep technique from computational geometry [1, 11]. The plane sweep technique is one of the most efficient methods known for determining intersection points between line segments. The time complexity of this algorithm is $O(n \log n + I \log n)$, where I is the number of intersections [1] and n is the number of line segments. At first glance, this algorithm appears to be ideal for this application, but close examination of the value of I, which unfortunately will include the intersections between leg edges, shows this not to be the case. In each turn region there are up to m vertices. Connecting these vertices in all possible ways between consecutive turn regions yields m^2 leg edges and potentially $O(m^4)$ leg edge intersections per stage. None of these intersections come in to play in the succeeding steps of the KRPP algorithm and must be discarded. The number of intersections between leg edges and radar and obstacle polygons could theoretically be km^2p where k is the number of stages and p is the total number of edges comprising the radar and obstacle polygons. Our empirical analysis has shown the number of this type of intersection to be approximately m^2 for the entire graph since it is unlikely that a configuration will repeatedly transit all radar polygon edges from stage to stage. Hence, if we were to use plane sweep for intersection detection, the time complexity of this step of the KRPP algorithm would be $O(km^4 \log m)$ based on the dominant term of this method's running time.

A time complexity of $O(km^4 \log m)$ would severely hobble any implementation, so instead of using plane sweep, we focus on ways to optimize the brute force method of intersection detection. Using a brute force approach, we could test every leg edge against every edge of the radar and obstacle polygons to determine the intersection points in

$O(km^2p)$ time where k is the number of stages, m is the maximum number of vertices per stage, and p is the total number of edges comprising the radar and obstacle polygons. If while constructing the radar and obstacle polygons, we note the extents of the polygon edges, we can construct a bounding box for each polygon. If we also construct a stage polygon using the supporting vertices for consecutive turn regions, we can exclude from consideration of intersection detection any radar and obstacle polygons whose bounding boxes neither are contained within nor intersect with the stage polygon as shown in Figure 5.

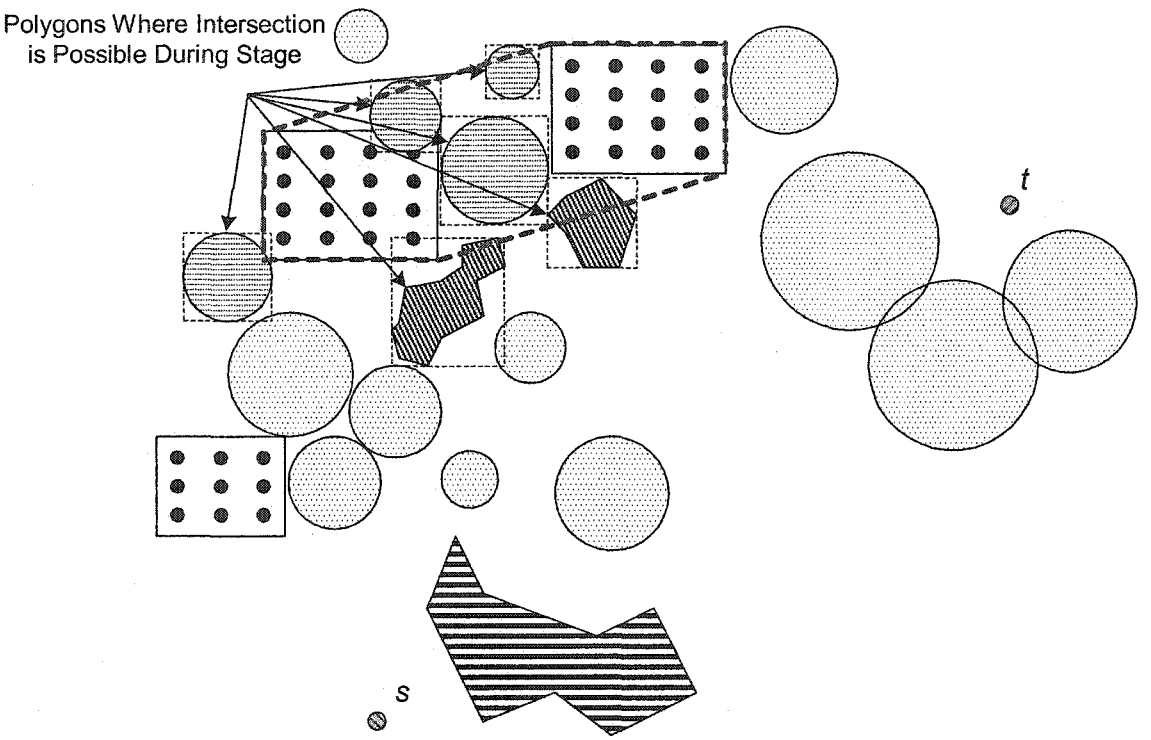

Figure 5. Stage Polygon and Potential Intersection Regions

By short-circuiting the intersection detection requirement for polygon edges that cannot possibly lie within the stage polygon, we are able to offset the multiplicative effect of the stage count k. Thus, in most cases, all points of intersection between leg edges and radar polygons can be found in $O(m^2p)$ time.

Organizing Intersection Points

When intersection points are computed, some additional information is associated with the record of intersection points. In particular, references to the corresponding leg edge and radar polygon are maintained in the record of intersection points. With this additional information in the record of each intersection point, the set of intersection points can be sorted in the priority of (i) coordinates of the source of the leg edge, (ii) coordinates of the destination of the leg edge, and (iii) the distance of the intersection point from the source of the corresponding leg edge.

Figure 6 shows a visualization of this process. Vertices s_1 and s_2 are contained in one leg region and vertices d_1 and d_2 are in the subsequent leg region. The smaller dots on the edges of the radar coverage regions are the intersection points between the leg edges and the risk regions. The numbers adjacent to the intersection points represent the final ascending sort order for this collection of intersection points.

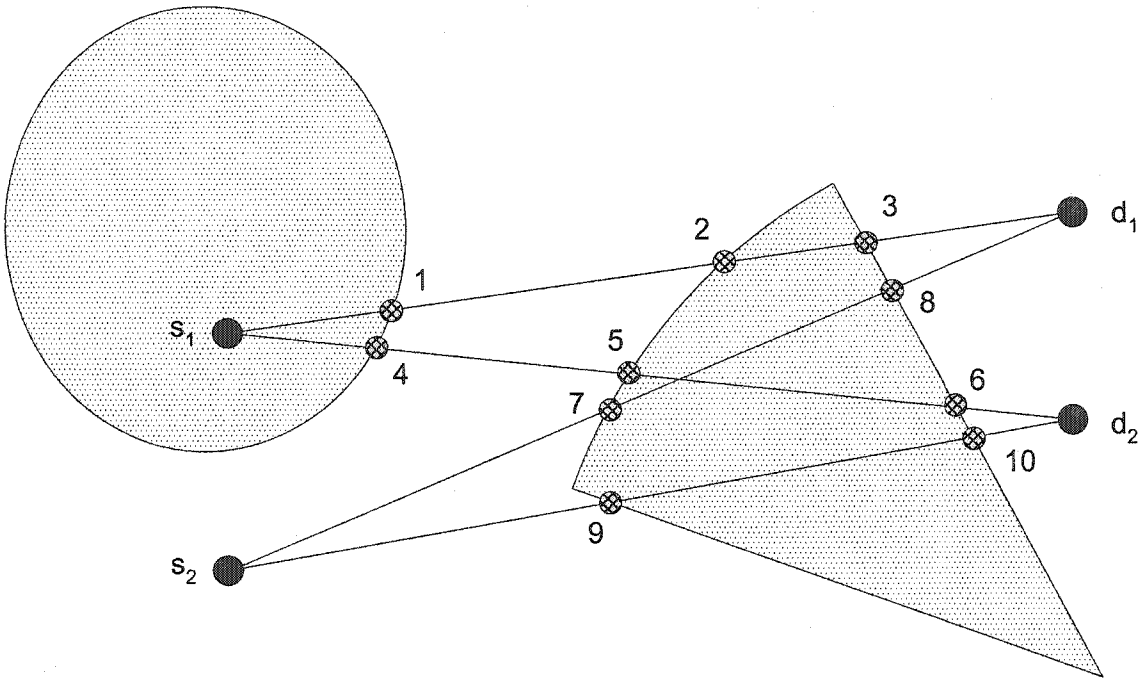

Figure 6. Organizing Intersection Points

Remark: The final outcome of the sorting of intersection points using the above three priorities is the generation of the sequence of intersection points ordered by leg edges.

Assigning Weights to Leg Edges

The *accumulated segment weight* algorithm computes the exposure for a straight-line path from start point s through a collection of free and risk regions given: t the distance to the segment end point, the region intersection points for the path, and the initial in/out state of the start point relative to the risk regions. The region intersection points, numbered from $i = 0$ to $n - 1$, are structures comprised of the region number and q_i, the radius from the start point.

The algorithm initially counts the number of risk regions enclosing the start point based on the values of the in/out states and stores the sum in variable k. Next, radius p,

used in the weight calculation, and the accumulated segment weight w, are both set to zero. The algorithm then iterates through the intersection points. For each intersection point the in/out state of the corresponding region is toggled to the alternate position. For transitions from out to in, k is incremented by 1, and for transitions from in to out, k is decremented by 1. If the resulting value of k is 0, q_i - p is added to w. If the resulting value of k is 1 after an increment, p is set to q_i. After the last intersection point is processed, if the value of k is greater than 0, t - p is added to w.

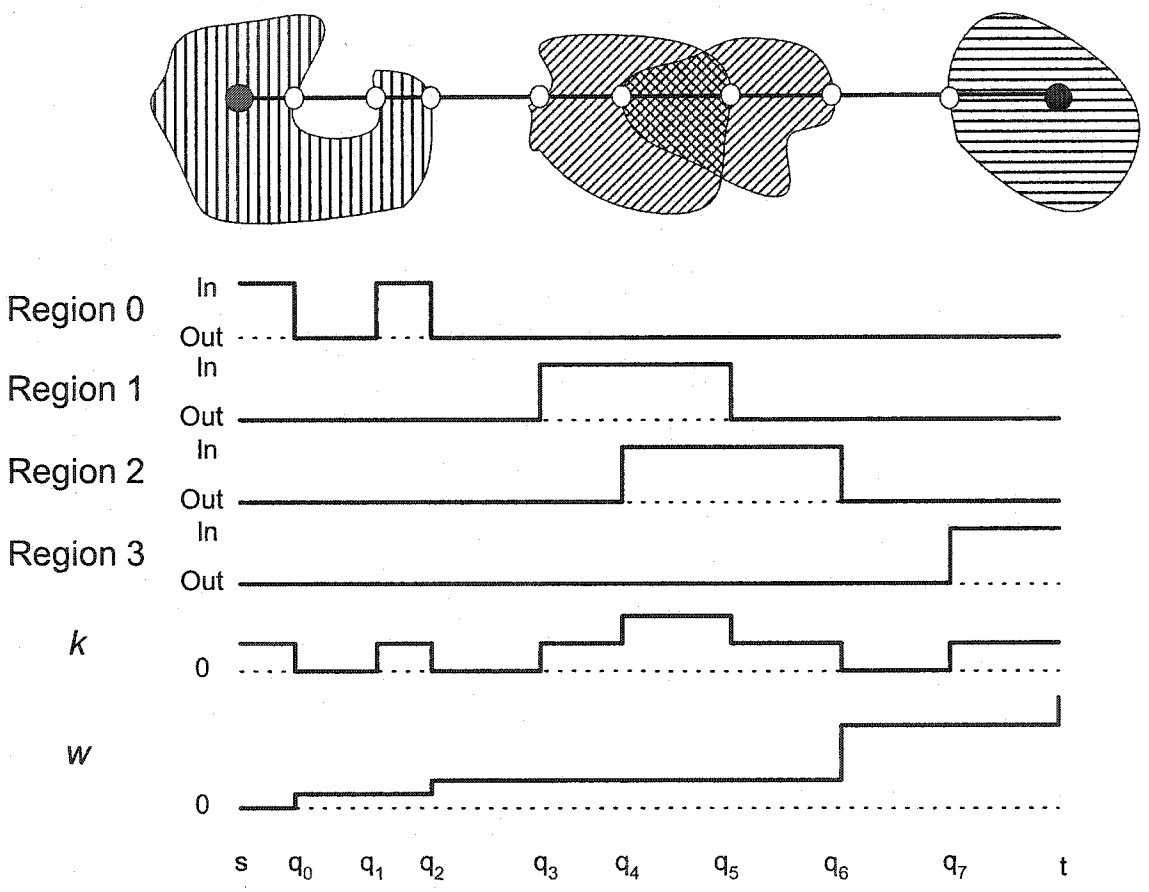

Figure 7. Accumulation of Segment Weights

A slight modification of the algorithm allows for a penalty or discount factor to be applied to the weight when traversing through the intersection of multiple risk regions. For the modified algorithm an n-dimensional cost matrix similar to Table 1 is provided which contains the penalty or discount factor for combinations of n unique region types.

Table 1. Risk Cost Matrix

Region A	Region B	Region C	Cost Factor
X			1
	X		1
		X	1
X	X		1
X		X	2
	X	X	3
X	X	X	5

In this algorithm, the distance q_i - p is calculated on each increment or decrement of k other than an increment from 0 to 1 and multiplied by the cost factor corresponding to the current region state combinations. The resulting value is then added to w. Next, the on/off state for the region is then toggled, and the value of p is set to q_i (on any increment or decrement of k). After the last intersection point is processed, if the value of k is greater than 0, distance t - p multiplied by the cost factor is added to w.

It may be noted that when a curvature factor is incorporated that corresponds to each turn, the turn is either feasible or not feasible. If it is not feasible, the assigned weight is set to infinity.

Construction of the Stage Graph

The graph $G(V, E)$ whose vertices are the vertices of leg regions and whose edges are the set of leg edges is the stage graph. Formally, $V = \{v \mid v$ is a vertex of a leg region$\}$, and $E = \{(v_1, v_2) \mid v_1$ and v_2 are the vertices of consecutive leg regions in the sequence r_0, $r_1, \ldots, r_{k+1}\}$. What remains is the computation of the weights of the edges of the stage graph. The weight of a leg edge e_i is the length of the portions of e_i that lie in radar polygons. To compute the weight of a leg edge e_i, we examine the sequence of intersection points along the leg edge and accumulate the weight in incremental steps. With one scan of the sequence, the total length of the portions of the edge lying in the radar polygon can be computed in linear time.

Computation of Shortest Path

After constructing the weighted stage graph all that remains is to compute the shortest k-legged path. In a directed graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, we can use Dijkstra's algorithm to compute the shortest path connecting two vertices in $O(|V|^2)$ time [15], where $|V|$ is the number of vertices in the graph. Therefore, for our implementation where km is the number of vertices, the shortest k-legged path can be found in $O(k^2m^2)$ time.

K-Legged Path Algorithm

A formal sketch of the algorithm is given as the k-legged path algorithm:

Input: Collection of radar polygons and leg regions.

Output: Shortest k-legged path.

Step 1: Compute the strip polygon by processing radar-polygons and leg regions.

Step 2: Determine all intersection points by using plane sweep.

Step 3: Sort intersection points in the priority order of (i) source vertex ID, (ii) destination vertex ID, and (iii) distance from the source vertex. Let *SL* denote the resulting sorted list of intersection points.

Step 4: Construct the weighted stage graph by scanning the list *SL*.

Step 5: Report shortest k-legged path by using Dijkstra's shortest path algorithm on the stage graph.

Time Complexity Analysis

The strip polygon can be constructed in $O(km)$ time where k is the number of stages and m is the maximum number of vertices per stage. Intersection detection takes $O(m^2p)$ time where p is the number of radar and obstacle polygon edges. Since the number of intersections is approximately m^2, step 3 takes $O(m^2 \log m)$. The number of edges connecting the vertices of consecutive leg-regions is $O(m^2)$. Hence, the total time for step 4 is $O(km^2)$. Since the stage graph has km^2 edges and km vertices, step 5 takes $O(k^2m^2)$. Hence, step 2 is the dominant step and the time complexity of the algorithm is $O(m^2p)$. We therefore conclude our presentation with the following theorem.

Theorem 1: A k-legged risk-reduced path can be computed in $O(m^2p)$ time where m is the number of vertices per turn region and p is the number of radar and obstacle polygon edges.

CHAPTER 3

RISK-REDUCED MULTIPLE PATHS

For trajectory planning of aerial vehicles for reconnaissance missions, it is highly desirable to have multiple risk-reduced collision-free paths. In this chapter, we present approximation algorithms for constructing multiple collision-free k-legged risk-reduced paths.

Algorithm Development

To develop efficient algorithms for constructing multiple risk-reduced collision-free k-legged paths, we start with the k-legged risk-reduced path algorithm described in the last chapter. This algorithm computes shortest path in the stage graph by the direct application of Dijkstra's algorithm [11]. In a directed graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, we can use Dijkstra's algorithm to compute the shortest path connecting two vertices in $O(|V|^2)$ time [15], where $|V|$ is the number of vertices in the graph. While the same approach may be used in a k-stage graph, the structure of the stage graph lends itself to optimization and in this chapter we will use this structure to develop a faster algorithm. As indicated in the last chapter, the path in a k-stage graph must visit exactly one vertex in each leg region $r_1, r_2, r_3, \ldots r_k$. As before, we will refer to the start vertex as s and the target vertex as t. In addition, we will denote vertex j in region r_i as $v_{i,j}$.

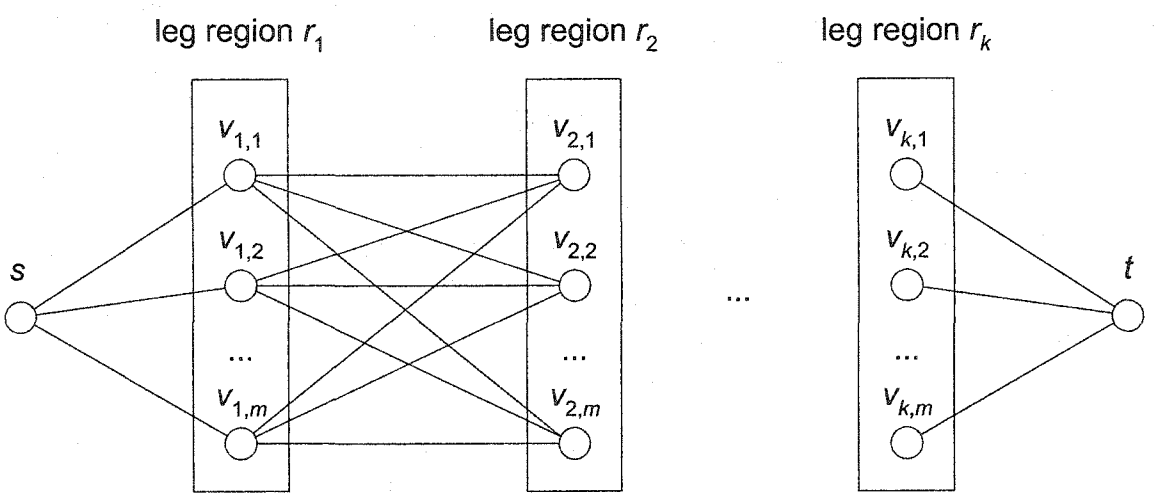

Figure 8. *K*-Legged Stage Graph

Figure 8 illustrates the simplified stage graph and notations. For clarity of presentation, we assume without loss of generality that the number of vertices in each leg region is the same and is equal to m. If not, we can introduce extra vertices in leg regions containing fewer than m vertices to make the number of vertices equal throughout. We then introduce edges between the extra vertices and the vertices of the other leg regions with edge weights equal to infinity as shown in Figure 9. This weight assignment ensures that the shortest route will never go through any of the extra vertices.

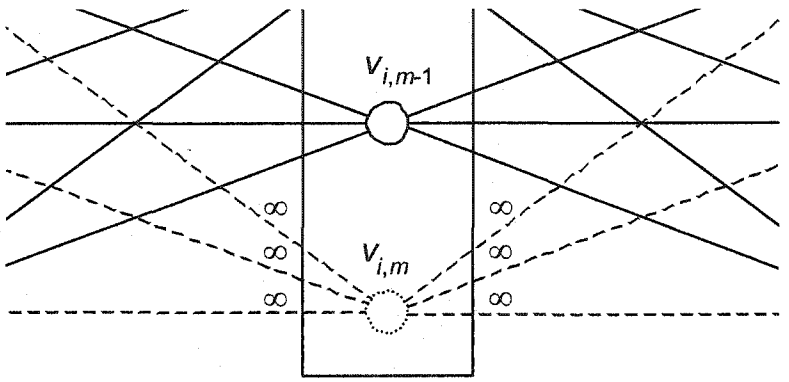

Figure 9. Leg Region with Extra Vertex and Edges

Let $W(v_{i,j})$ denote the weight of the shortest path from source vertex s to vertex j in region r_i and let $w(v_{i,j}, v_{i+1,h})$ denote the weight of the edge connecting vertex $v_{i,j}$ to vertex $v_{i+1,h}$. Suppose we know the shortest path from source vertex s to all vertices $v_{i,1}$, $v_{i,2}$, …, $v_{i,m}$ in region r_i as shown in Figure 10. Observe that the weight of the shortest path from s to $v_{i+1,h}$ is equal to:

$$min(W(v_{i,1}) + w(v_{i,1}, v_{i+1,h}), W(v_{i,2}) + w(v_{i,2}, v_{i+1,h}), \ldots, W(v_{i,m}) + w(v_{i,m}, v_{i+1,h}))$$

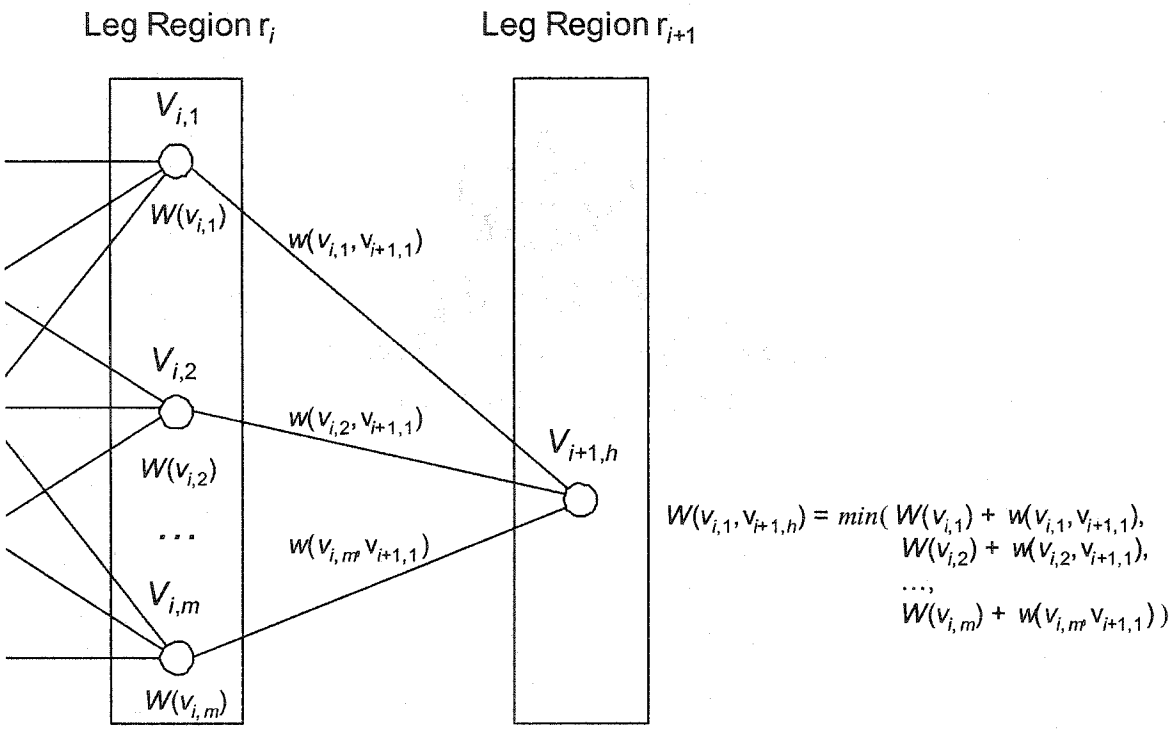

Figure 10. Computing the Weight of the Shortest Path to Vertex $v_{i+1,h}$

Lemma 2: If we know the weights of the shortest paths from source vertex s to all vertices in leg region r_i, then we can compute the weights of the shortest paths from s to all vertices in leg region r_{i+1} in $O(m^2)$ time where m is the number of vertices per leg region.

Proof: We know the weights $W(v_{i,1})$, $W(v_{i,2})$, ... , $W(v_{i,m})$ of the shortest paths from start vertex s to vertices $v_{i,1}$, $v_{i,2}$, ..., $v_{i,m}$. Note that the shortest path from source point s to any vertex in region r_{i+1} must pass through exactly one vertex in region r_i. Hence, the weight of the shortest path from source vertex s to vertex $v_{i+1,j}$ in region r_{i+1} can be written as:

$$W(v_{i+1,j}) = min(W(v_{i,1})+w(v_{i,1}, v_{i+1,j}), W(v_{i,2})+w(v_{i,2}, v_{i+1,j}), \ldots, W(v_{i,m})+w(v_{i,m}, v_{i+1,j}))$$

By using the above relation, the weights of shortest paths from s to all vertices in region r_{i+1} can be found by scanning the weights of edges between regions r_i and r_{i+1} and adding them to the weights of the shortest paths from s to vertices in r_i. The time to compute weights $W(v_{i+1,1})$, $W(v_{i+1,2})$, ... , $W(v_{i+1,m})$ is bounded by the number of edges connecting vertices in r_i to vertices in r_{i+1}. There are m^2 edges connecting vertices in r_i to vertices in r_{i+1}; hence the total time to compute all weights in a region is $O(m^2)$.

By using the above lemma repeatedly between consecutive stages in the stage graph, starting from s and ending at t, the length of the shortest k-legged path can be computed in $O(km^2)$ time. When nodes at region r_i are processed, the weights of the shortest paths from source vertex s to all vertices up to and including vertices in region r_i are known. In other words computation can proceed from left to right without the need of examining nodes to the left of r_i. Furthermore, no nodes in regions to the right of r_i need to be examined for computing shortest paths to nodes in r_i. It is because of these properties that the resulting algorithm is fast. The following algorithm is a high level sketch of this process.

Stage-Step Shortest Path Algorithm

Input: Weighted Stage Graph $G(V,E)$

Output: Weight and description of the shortest k-legged path connecting s and t.

Step 1: Compute the path weights from s to each vertex in region r_1.

for (j=1; j<=m; j+1) {

$W(v_{1,j}) = w(s, v_{1,j})$;

}

Step 2: Compute the path weight to each vertex in all subsequent regions.

for (i=1; i<k; i+1) {

for (j=1; j<=m; j+1) {

$W(v_{i+1,j}) = min(W(v_{i,1})+w(v_{i,1}, v_{i+1,j}), W(v_{i,2})+w(v_{i,2}, v_{i+1,j}), \ldots, W(v_{i,m})+w(v_{i,m}, v_{i+1,j}))$;

}

}

Step 3: Compute the path weight to t.

$W(t) = min(W(v_{i,1})+w(v_{i,1}, t), W(v_{i,2})+w(v_{i,2}, t), W(v_{i,1})+w(v_{i,3}, t), \ldots, W(v_{i,m})+w(v_{i,m}, t))$;

If a twin edge list is used to maintain the graph, it is important to select the weight of the reflected edge adjacent to vertex $v_{i+1,j}$ for the incident edge is the backward directed edge and it will have a weight of infinity.

Extracting the Shortest Path from Source Vertex s to Vertex $v_{i,j}$.

When calculating $W(v_{i,j})$ for each vertex $v_{i,j}$, we note the index g of the previous vertex $v_{i-1,g}$ on the shortest path and store it with $v_{i,j}$. We can then extract the shortest path by recursively calling a routine that outputs the previous vertex $v_{i-1,g}$. A sketch of the algorithm is as follows:

```
ShortestPath(v_{i,j}) {
    if((v_{i,j}) != s) {
        g = index of previous vertex on shortest path;
        ShortestPath(v_{i-1,g});
    }
    output v_{i,j}
}
```

Construction of Second Shortest Path

A pair of paths connecting s and t is obviously the simplest example of multiple paths. It is therefore useful to consider the construction of the second shortest path in the stage graph. The second shortest path and the first shortest path could be completely disjoint in their interior or could share some edges. It is critical to note that the first shortest path and the second shortest path must have at least one edge not common between them; otherwise both paths will be identical. Hence, if we execute the shortest path algorithm on the graph by removing an "appropriate" edge of the shortest path then the resulting path will be the second shortest path. But it is not clear how to identify the appropriate edge. So we try all edges of the first shortest path one by one. The algorithm based on this approach is enumerated below.

Edge Elimination Second Shortest Path Algorithm

Step 1: Run the stage-step algorithm on the stage graph to determine the first shortest path p_1.

Step 2: Let $e_0, e_1, ..., e_k$ denote the edges of p_1.

Step 3: Successively delete one edge at a time (e_0, e_1, ..., e_k) from p_1 and run Dijkstra's algorithm again on the resulting graph to create a pool of potential second shortest paths. Note: replace the previously deleted edge before deleting a new edge.

Step 4: The shortest path from the pool of potential second shortest paths is the second shortest path p_2.

A straightforward analysis of the edge elimination second shortest path algorithm reveals that the second shortest path can be computed in $O(k^2m^2)$ time, if the weighted graph is available.

Lemma 3: The edge elimination second shortest path algorithm constructs the second shortest path correctly.

Proof: Let $p_1 = e_0, e_1, ..., e_k$ denote the first shortest path. Let p_2 denote the second shortest path. By the definition of the second shortest path, p_2 cannot contain all edges of p. Hence, p_2 contains at least one edge not present in p_1; denote such an edge by e_q.

Let G_i, $0 \leq i \leq k$ denote the graph obtained by removing edge e_i from G. The edge elimination algorithm finds the shortest paths in graphs G_0, G_1, ..., G_k. We prove the lemma by showing that p_2 is present in one of G_i, $0 \leq i \leq k$.

Case 1: p_1 and p_2 are edge disjoint. In this case p_2 is contained in all of G_i, $0 \leq i \leq k$ and hence the edge elimination second shortest path algorithm finds it.

Case 2: p_1 and p_2 are not edge disjoint. Since e_q is not present in p_1, p_2 is present in G_q and the edge elimination second shortest path algorithm finds it.

Approximation Algorithms for Constructing m Shortest Paths

We now consider the construction of m shortest paths in the stage graph. We could try to generalize the approach used in the construction of the second shortest path to construct the m^{th} shortest path. For computing the second shortest path, the algorithm searches for the solutions in k different graphs. If we use this technique to construct the m^{th} shortest path, the number of candidate graphs grows exponential in m and the time complexity of the resulting algorithm becomes exceedingly high. We therefore consider the development of approximation algorithms for the construction m shortest paths in the stage graph. We propose two approximation algorithms for constructing such paths.

Rank-Ordering Approach

The first approximation algorithm we propose is based connecting nodes of consecutive regions in term of their distances from the source vertex s. Observe that the step-stage algorithm constructs shortest paths from s to all nodes. We sort the vertices in each region in non-decreasing order of their distance from source point s. If a node x of a region is in the q^{th} position in the sorted list, then we define its rank as q. The paths are constructed by connecting nodes of same ranks in consecutive regions. A high-level sketch of the algorithm is given below.

Rank-Ordered Approximation Algorithm

Input: k-stage graph $G(V,E)$

Output: m risk-reduced paths

Step 1: Apply the step-stage algorithm to determine shortest paths to all vertices from source vertex s to all other vertices.

Step 2: Assign ranks to vertices in each stage by sorting them in the order of their distance from the source vertex s.

Step 3: Construct m paths by connecting nodes of same rank in consecutive leg regions.

Time Complexity Analysis

Given the stage graph, the time for step 1 is $O(km^2)$. Sorting the k stages in step 2 can be performed in $O(km \log m)$ time. Construction of the paths in step 3 requires $O(km)$ time. Thus, we can easily see that the time complexity of the above rank-ordered approximation algorithm is $(km^2 + km \log m)$. One added advantage of this algorithm is that the constructed paths are disjoint.

Path Elimination Approach

The second approximation algorithm we propose is based on forcing the $(i+1)^{th}$ risk-reduced path to not take any edges used by the paths constructed so far. The first path is constructed by using the stage-step algorithm. To construct the second shortest path, the edges used in the first shortest path are assigned weights equal to infinity and the stage-step algorithm is executed in the modified stage graph. Other paths are constructed in similar manner. A high-level sketch of this algorithm is as follows.

Path Elimination Approximation Algorithm

Input: k-stage graph $G(V,E)$

Output: m risk-reduced paths

Step 1: Apply the stage-step algorithm on $G(V,E)$ to determine the shortest path connecting s to t. Let *previous* be this path.

Output *previous* as a member of the solution.

Step 2: For (int i=2; i<m; i++) {

Modify $G(V,E)$ by assigning weights of infinity to all edges in *previous*.

Apply the stage step algorithm on $G(V,E)$ to determine the shortest path connecting s to t. Let *previous* be this path.

Output *previous* as a member of the solution.

}

Remark: If the shortest path is of length infinity in the above algorithm then the target is unreachable.

A simple analysis shows that the time complexity of the path elimination approximation algorithm is $O(km^3)$, since we must repeat the $O(km^2)$ step-stage algorithm for each of the m vertices per turn region.

Why Computing the Second and Subsequent Shortest Paths is Difficult

The second shortest path is difficult to compute in a straightforward way because none of the information gathered in determining the shortest path can be used to determine the second shortest path. One might assume that the second shortest path can be determined by selecting the second shortest path from the ordered results of Step 3 in

the stage-step algorithm. This greedy method fails because the second shortest path into a particular vertex of a leg may be shorter than the shortest paths to all other vertices in that leg.

A Method for Computing the m Shortest Paths in a k-Legged Stage Graph

As we have shown, computing the second and subsequent shortest paths in a k-legged stage graph is difficult because we only capture the shortest path to any particular vertex. By modifying our data structure to retain the m shortest paths into a vertex, we are then able to compute the m shortest paths in the graph. Let $W_q(v_{i,j})$ denote the weight of the q^{th} shortest path from source vertex s to vertex j in region r_i.

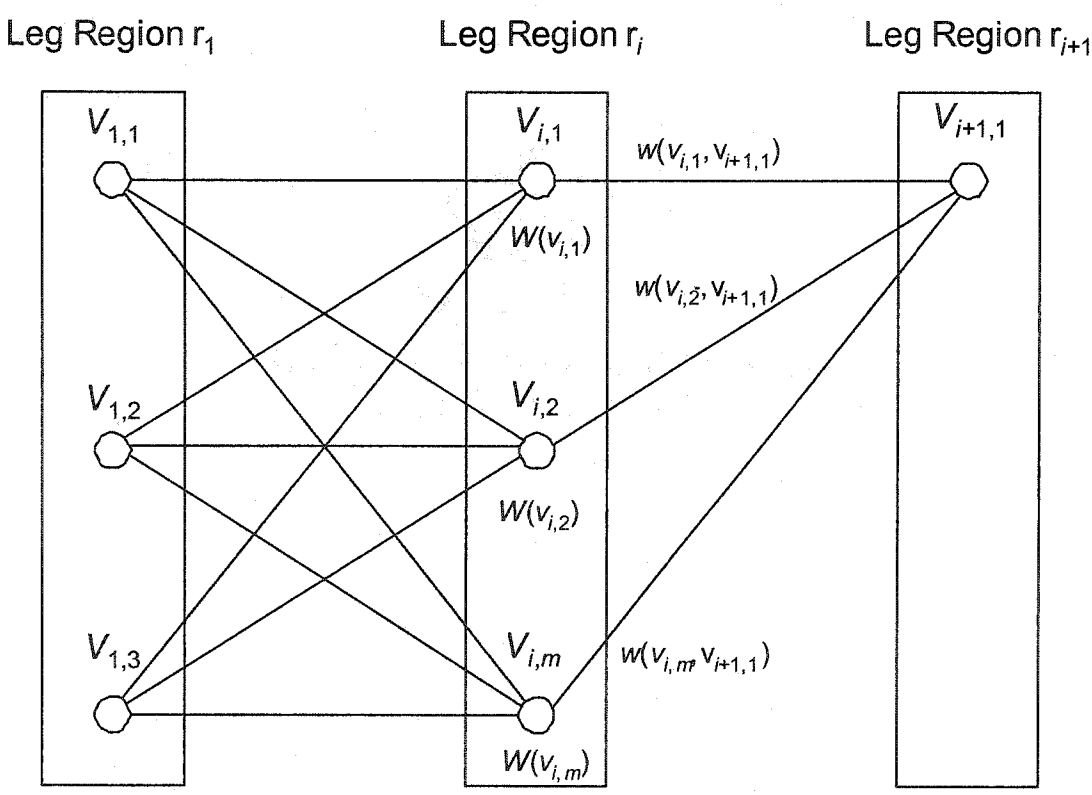

Figure 11. Computing the Shortest Path to a Vertex

In Figure 11 we can observe that the length of the shortest path to a vertex is the minimum of the sums of the shortest path weights to the prior vertices plus the weights of the corresponding connecting edges.

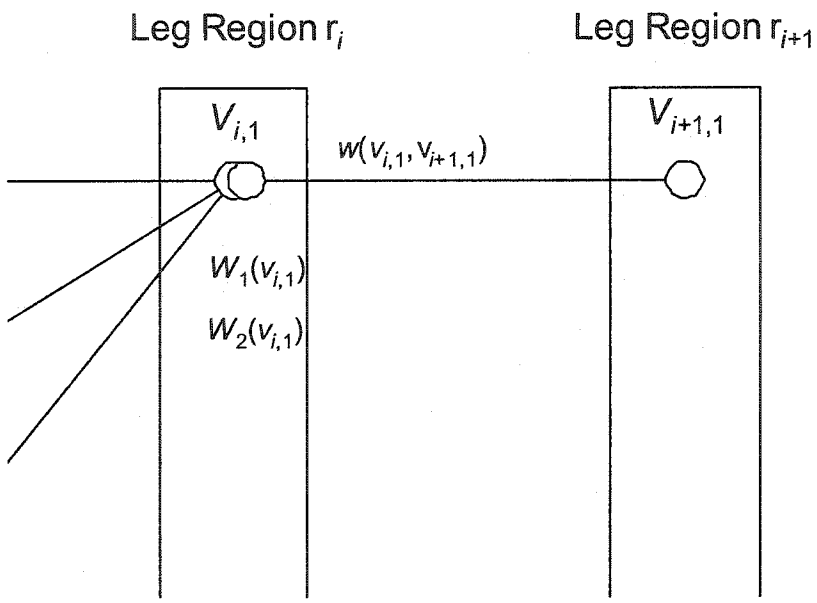

Figure 12. Two Weights per Vertex

Imagine for a moment that each vertex in r_i is really two copositioned vertices as shown in Figure 12 with corresponding edges connecting to $v_{i+1,1}$. Let the shortest path weights to the new vertices vary while the weights of the new edges connecting to vertex $v_{i+1,1}$ remain the same. The weights to the new vertices correspond to the second shortest path weights to the original vertex. Our new graph results in $2m$ possible choices for the shortest path to vertex $v_{i+1,1}$. It is of interest to note that while in the original graph, the second shortest path to vertex $v_{i+1,1}$ might not have been one of the m possible choices, in the new graph, the second shortest path must be contained in the $2m$ possible choices. We can reason that this is the case for if there exists an alternate path p' to vertex $v_{i+1,1}$ from a

vertex in r_i that is not one of the paths in our graph, then it must include the third or longer shortest paths to a vertex in this leg region. Let the weight to the second shortest path be x and the weight to this alternate path be y so that $x < y$. Note that p' must include the same edge e from r_i to r_{i+1} as in one of the paths in our graph. The weight of p' therefore is $y + w(e)$, while the weight through the same vertex taking the second shortest path to the vertex would be $x + w(e)$. Since $x < y$, then $x + w(e) < y + w(e)$ and $x + w(e) < p'$ which contradicts the assertion that p' is the second shortest path to vertex $v_{i+1,1}$. Hence, the $2m$ possible choices must include the second shortest path.

Adding an additional vertex representing the third shortest path to each vertex in r_i would result in $3m$ possible choices for the shortest path to vertex $v_{i+1,1}$ and these $3m$ possible choices must include the three shortest paths to vertex $v_{i+1,1}$. We can continue adding vertices as shown in Figure 13 until there are m vertices per vertex in r_i for a total of m^2 possible shortest paths to vertex $v_{i+1,1}$ which must contain the m shortest paths to vertex $v_{i+1,1}$.

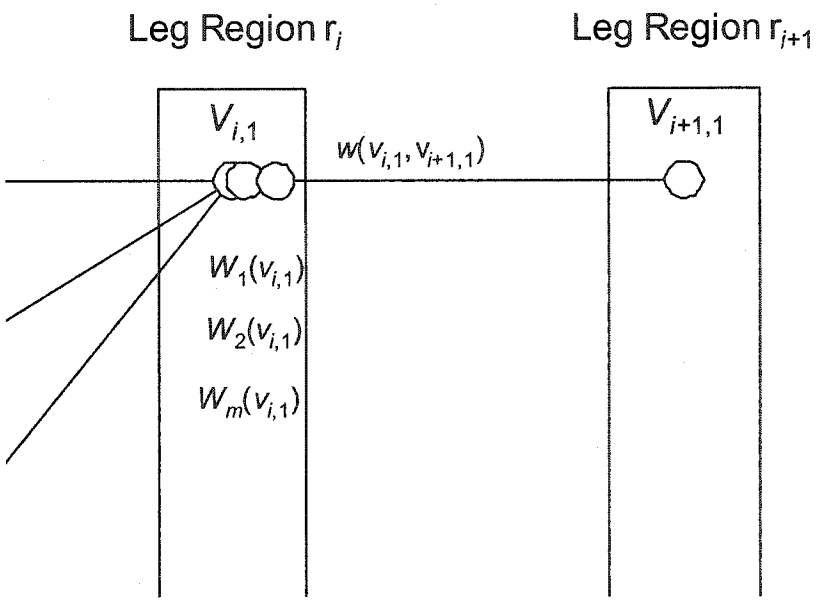

Figure 13. *m* Weights per Vertex

Suppose we know the *m* shortest paths from source vertex *s* to all vertices $v_{i,1}$, $v_{i,2}$, …, $v_{i,m}$ in region r_i. Observe that the weight of the q^{th} shortest path from *s* to $v_{i+1,h}$ is can be written as:

$W_q(v_{i+1,h}) =$

$min_q((W_1(v_{i,1}) + w(v_{i,1}, v_{i+1,h}), W_2(v_{i,1}) + w(v_{i,1}, v_{i+1,h}), ..., W_m(v_{i,1}) + w(v_{i,1}, v_{i+1,h})),$

$(W_1(v_{i,2}) + w(v_{i,2}, v_{i+1,h}), W_2(v_{i,2}) + w(v_{i,2}, v_{i+1,h}), ..., W_m(v_{i,2}) + w(v_{i,2}, v_{i+1,h})), ...,$

$(W_1(v_{i,m}) + w(v_{i,m}, v_{i+1,h}), W_2(v_{i,m}) + w(v_{i,m}, v_{i+1,h}), ..., W_m(v_{i,m}) + w(v_{i,m}, v_{i+1,h})))$

By using the above relation, the weights of the *m* shortest paths from *s* to all vertices in region r_{i+1} can be found by scanning the weights of edges between regions r_i and r_{i+1} and adding them to the weights of the *m* shortest paths from *s* to vertices in r_i. The time to compute weights $(W_1(v_{i+1,1}), W_2(v_{i+1,1}), \ldots, W_m(v_{i+1,1}))$, $(W_1(v_{i+1,2}), W_2(v_{i+1,2}), \ldots, W_m(v_{i+1,2}))$, …, $(W_1(v_{i+1,m}), W_2(v_{i+1,m}), \ldots, W_m(v_{i+1,m}))$ is bounded by the time to sort the number of edges connecting vertices in r_i to vertices in r_{i+1} multiplied by the number of

weights per vertex. There are m^2 edges connecting vertices in r_i to vertices in r_{i+1} and m weights per vertex; hence the total time to compute and sort all weights in a region is $O(m^3 \log m)$. If we were not to limit the number of shortest paths maintained per vertex to m, the time and space required to calculate the complete set of shortest paths would increase exponentially per stage.

By repeating these calculations for consecutive stages in the stage graph, starting from s and ending at t, the length of the m shortest k-legged paths can be computed in $O(km^3 \log m)$ time. This is stated in the following theorem:

Theorem 2: Given a stage graph, the m shortest k-legged paths can be computed in $O(km^3 \log m)$ time.

In our implementation, rather than adding extra vertices, we maintain with each vertex $v_{i,j}$ a pseudovertex array of size m. Each element in the ordered array contains the weight, previous vertex, and previous pseudovertex element index of the q^{th} shortest path from source vertex s to vertex $v_{i,j}$. This array allows us to extract the shortest path from source vertex s to $v_{i,j}$. The following algorithm is a modified version of the stage-step algorithm and computes the m shortest paths.

Stage-Step Algorithm for m Shortest Paths

Input: Weighted Stage Graph $G(V,E)$

Output: Weight and description of the shortest k-legged path connecting s and t.

Step 1: Compute the path weights from s to each vertex in region r_1.

for (j=1; j<=m; j+1) {

$W_1(v_{1,j}) = w(s, v_{1,j})$;

}

Step 2: Compute the m smallest path weights to each vertex in all subsequent regions.

for (i=1; i<k; i+1) {

  for (j=1; j<=m; j+1) {

    $paths[1...m^2]$ = *ordered*($W_1(v_{i,1}) + w(v_{i,1}, v_{i+1,h})$, $W_2(v_{i,1}) + w(v_{i,1}, v_{i+1,h})$, ...,

      $W_m(v_{i,1}) + w(v_{i,1}, v_{i+1,h})$, $W_1(v_{i,2}) + w(v_{i,2}, v_{i+1,h})$, $W_2(v_{i,2}) + w(v_{i,2}, v_{i+1,h})$, ...,

      $W_m(v_{i,2}) + w(v_{i,2}, v_{i+1,h})$, ..., $W_1(v_{i,m}) + w(v_{i,m}, v_{i+1,h})$, $W_2(v_{i,m}) + w(v_{i,m}, v_{i+1,h})$, ...,

      $W_m(v_{i,m}) + w(v_{i,m}, v_{i+1,h})$);

    for (q=1; q<=m; q+1) {

      $W_q(v_{i+1,j})$ = $paths[q]$;

    }

  }

}

Step 3: Compute the m smallest path weights to t.

$paths[1...m^2]$ = *ordered*($W_1(v_{k,1}) + w(v_{k,1}, t)$, $W_2(v_{k,1}) + w(v_{k,1}, t)$, ..., $W_m(v_{k,1}) + w(v_{k,1}, t)$,

  $W_1(v_{k,2}) + w(v_{i,2}, t)$, $W_2(v_{k,2}) + w(v_k, t)$, ..., $W_m(v_{k,2}) + w(v_k, t)$, ..., $W_1(v_{k,m}) + w(v_{k,m}, t)$,

  $W_2(v_{k,m}) + w(v_{k,m}, t)$, ..., $W_m(v_{k,m}) + w(v_{k,m}, t)$);

for (q=1; q<=m; q+1) {

  $W_q(t)$ = $paths[q]$;

}

We presented an $O(km^2)$ time algorithm for computing the shortest path connecting two vertices in a stage graph. In addition, we demonstrated how the algorithm may be modified to capture the m shortest paths in the stage graph in $O(km^3 \log m)$ time at the

expense of $O(km^2)$ more space. We also note that the algorithm can be implemented without using any complicated data structure. In fact it uses only arrays.

CHAPTER 4

IMPLEMENTATION AND RELATED EXPERIMENTAL RESULTS

We have developed three implementations to prototype and test the operation and performance of our algorithms. The first is a trajectory planner application that allows the user to interactively construct leg regions, regions under the exposure of radars, the start point, and the target point for the *k*-legged route.

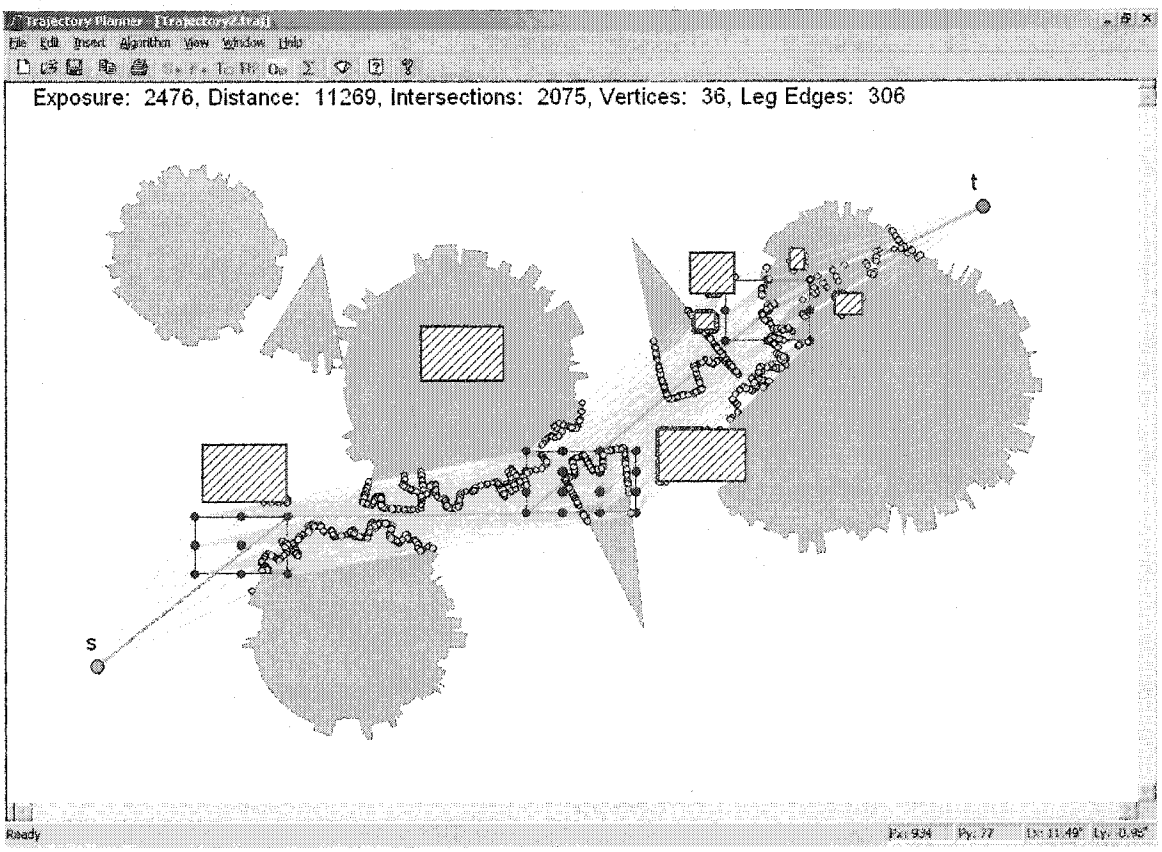

Figure 14. Snapshot of Trajectory Planner Prototype Output

The implementation displays the constructed stage graph and the risk-reduced *k*-legged route. A snapshot of the output produced by the program is shown in Figure 14.

The second implementation shown in Figure 15 calculates intersection points using plane sweep and brute force methods. The application animates the operation of the algorithms and can also benchmark the performance of the algorithms for various datasets.

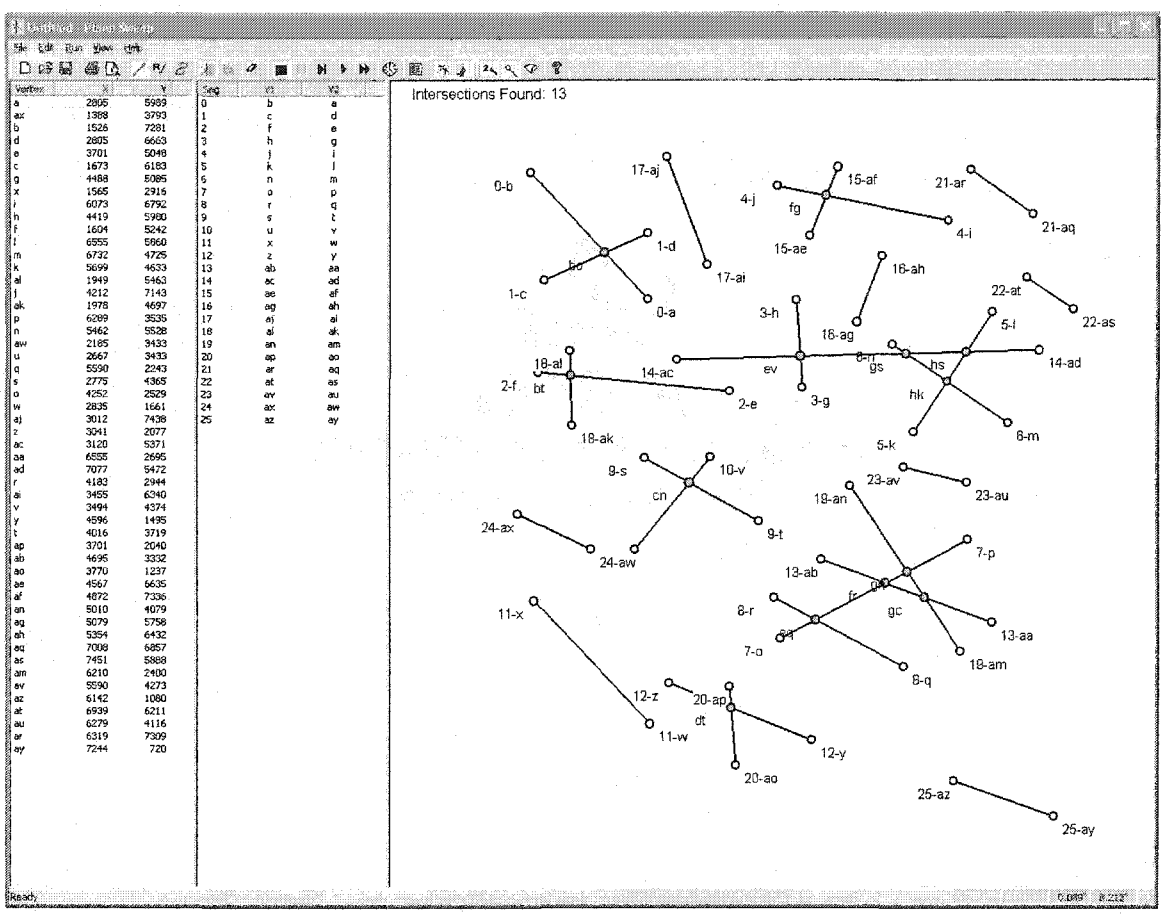

Figure 15. Plane Sweep Program

The third implementation is a multiple path generator application that allows the user to construct a *k*-stage graph with *m* vertices per stage. The leg edges are assigned random

values for weights. The program calculates and displays the m least cost paths. A snapshot of this application is shown in Figure 16.

Unified Modeling Language (UML) class diagrams are provided in the appendix to assist in conceptualizing the implementation of the trajectory planner application.

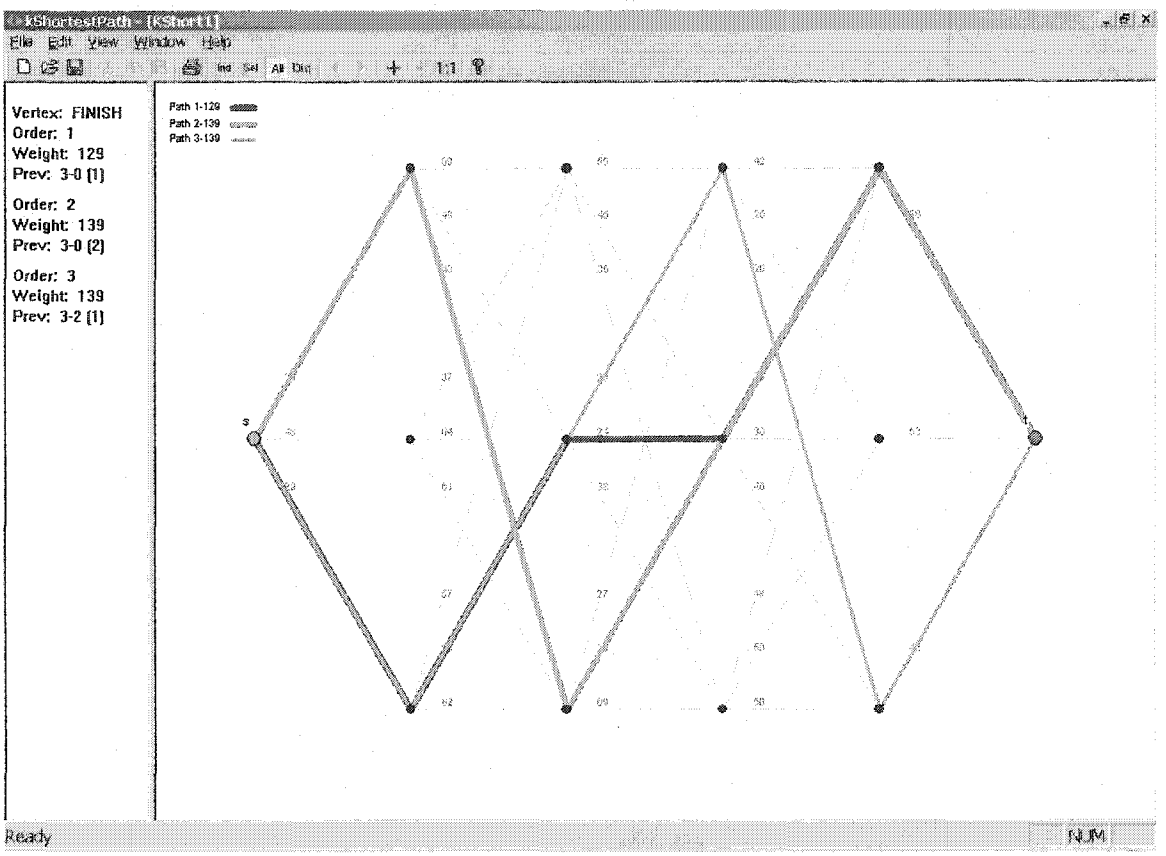

Figure 16. Multiple Risk-Reduced Paths Application

Trajectory Planner Model

For the trajectory planner implementation we start with a very simplified model for testing our algorithms. The simplified model is incorporated in suitable data structures in our implementation. The three main objects in the model are leg regions, obstacles, and

radars. Of course start point s and target point t are simply taken as points represented by a pair of integers. In general terms, leg regions, obstacles, and radars could all be modeled by simple polygons. However, for the purpose of clarity of presentation and for the purpose of simplification for implementation we use a simple polygonal model for radars and obstacles. The leg regions are taken as simple geometric shapes with a few sides. In particular, each leg region is the rectangular region representing the boundary that closely covers the finite set of locations where an aircraft might alter its course.

We are using two submodels for radar coverage regions. In the simplified version, a radar coverage region is represented as a circle or a wedge-shaped sector. The coordinates of the radar site are the center of the circle and the maximum detection range of the radar is the radius of the circle. If the extent of the azimuth scan region of the radar is less than 360 degrees, the start and stop azimuths are the angular coordinates of the bounding radii of the sector. The second submodel for a radar coverage region is a sequence of vertices representing its boundary and allows for a more complex and realistic representation of the threat region.

While the simple model is convenient for implementation and could possibly yield a faster path-planning algorithm, it could compromise the quality of the generated route. The actual coverage area of a radar installation is not a perfect circle or pie shaped wedge, but a more complex shape determined primarily by the line-of-site visibility of an aircraft at a specific altitude. Buildings in the immediate area of the radar antenna and the shape of the terrain in the surrounding area limit the line-of-site of a radar since radio waves cannot penetrate these objects without being greatly attenuated. Figure 17 shows the approximate shape of the maximum detection range of an aircraft flying at 10,000 ft.

elevation by an air surveillance radar located in Las Vegas, Nevada. Las Vegas is situated in a valley surrounded by mountains several thousand feet above the valley floor. At the azimuths of these mountains the radar coverage range is greatly diminished.

Figure 17. Radar Detection Range of Aircraft at 10,000 ft. Elevation

The implementation models the typical changes in radar coverage by randomly varying the outer edges of the risk regions. We believe that these variations will test the

robustness of the algorithm without requiring a separate complex computation of clutter induced range limitations.

The obstacles can be modeled by any simple polygon, and we feel that a polygonal shape of less than a dozen boundary edges is good enough to fairly approximate obstacles in real situations such as the cross section of terrain. In our application, the only obstacle type we implemented is a rectangle, which we feel is sufficient to represent an area where flight is prohibited.

The above components need to be represented in a suitable structure so that the developed trajectory planning algorithm can be executed. All the leg region vertices are combined as a weighted graph in two dimensions. The weights of edges are such that they reflect exposure to radars and obstacles. We also keep track of the Euclidean distance for each edge. Thus our graph can be taken as a Euclidean graph in two dimensions if there is a need to compute paths without considering exposure to threats. When we need to consider paths with exposure to threats then of course we have a non-Euclidian graph.

For a vertex v of each leg region we maintain a pair of lists of adjacent vertices corresponding to incoming and outgoing edges. The implicit edges are assigned direction by setting the distances of the incoming edges to infinity so that travel can only occur from leg region r_i to leg region r_{i+1}.

Computation of Intersection Points

Intersection points between the leg edges and radars may be computed using one of three different methods: segment-circle, segment-polygon, and plane sweep. For the

simple radar model, we represent each radar as a circle or sector. To compute the points where the leg edges intersect with the radars, we use a segment-circle method [3] to calculate the points where the edge line segment intersects with the circle. For radars with radar coverage less than 360 degrees we also need to determine if the intersection points are within the cone of the sector and if the edge segments intersect with the radii that bound the sector.

For the more realistic radar model, we represent each radar as a sequence of vertices that comprise the boundary of the radar coverage region. To compute the intersection points of the leg edges in the graph with the radars, we use a brute force technique to detect if each leg edge intersects with each segment in the radar polygon. If so, we then calculate the intersection point. For each leg edge, we iterate through the collection of radar objects. If the leg edge is neither enclosed by the rectangle bounding the extents of the radar polygon nor intersects with an edge of this rectangle, we are able to assert that the edge cannot intersect with any edge of the radar and bypass testing this radar for intersection. If the alternate condition is true, we iterate through each edge of the radar polygon to detect and possibly calculate any intersection points.

The third method for calculation of intersection points between leg edges and the radar polygons uses the plane sweep technique from computational geometry as described in [1]. While this method also finds intersections between momentarily adjacent leg edges, these are discarded and only intersections between leg edges and edges of the radar polygon are retained.

For all three methods the coordinates of the intersection points are stored so that they may be rendered on the displayed output. The source vertex ID, destination vertex ID,

Euclidean distance of the intersection from the source vertex, and radar ID are stored in a separate data structure so they may be used in the calculation of edge weights.

Setting Edge Weights

The weights of the leg edges are calculated by iterating through the collection of intersection points and accumulating the length of traversal through radar coverage regions as described in Chapter 2. This process requires that we prepare the list of intersection points by sorting them by priority of source vertex ID, destination vertex ID, and distance from source vertex. We define the less than comparison method in our intersection point class based on our priority order and invoke the sorting algorithm built-in to C++ for this initial step.

The weight accumulation algorithm requires us to know the inside/outside relation between each source vertex and each radar polygon. This determination is made by evaluating the relation of new vertex to existing radars or relation of existing vertices to new radar as each vertex or radar is added to the trajectory model. Armed with this a priori information, the algorithm toggles the in/out state in an array of condition variables corresponding to the individual radar polygons as each intersection point for an edge segment is encountered. When the algorithm determines that the intersection point is a transition from a region free from all radar coverage to a radar risk region it notes the entry position. When the state next transitions to a region free from all radar coverage the linear distance between the two points is summed to the weight of the leg edge. If the next intersection point instead corresponds to a different segment, then the distance from

the previously noted entry transition to the destination vertex is summed to the weight of the now previous leg edge.

Calculation of Risk-Reduced Paths

Once the segment weights are set the risk-reduced paths can be calculated. The least cost path is determined by invoking Dijkstra's algorithm [15] on the weighted graph. The determination of the shortest path to each vertex is based primarily on the exposure to threat. Should this value be equal for two competing paths, the path with the shorter Euclidian distance is favored. It may be noted that when a curvature factor is incorporated that corresponds to each turn, the turn is either feasible or not feasible. If it is not feasible, the assigned weight is set to infinity, thereby allowing for a turn-constrained risk-reduced path.

Multiple risk-reduced paths are then generated by using the approximation by path elimination technique described in Chapter 3. The weight of each leg in the shortest path is set to infinity one at a time and Dijkstra's algorithm is invoked in turn on the modified graph. This process results in $k + 1$ alternative routes where k is the number of turn regions and is also greater than zero.

Walk-Through of the Operation of the Program

When the program is first executed the initial window appears with a drop-down menu and toolbar above the blank canvas, and a status bar at the bottom of the screen as shown in Figure 18. At this point, a previously stored program state may be recalled or a new model may be created with the mouse in conjunction with menu and toolbar options.

The status bar at the bottom of the screen contains a message area on the lower left that displays information pertinent to the current state of the program. If the program is not in an insertion mode and the mouse is clicked and dragged across the canvas, the relative distance between the selected point and the cursor is displayed in the message area to serve as a virtual measurement device. To the right of the message area are mouse position fields that report the physical and logical position of the mouse as it is moved around the canvas.

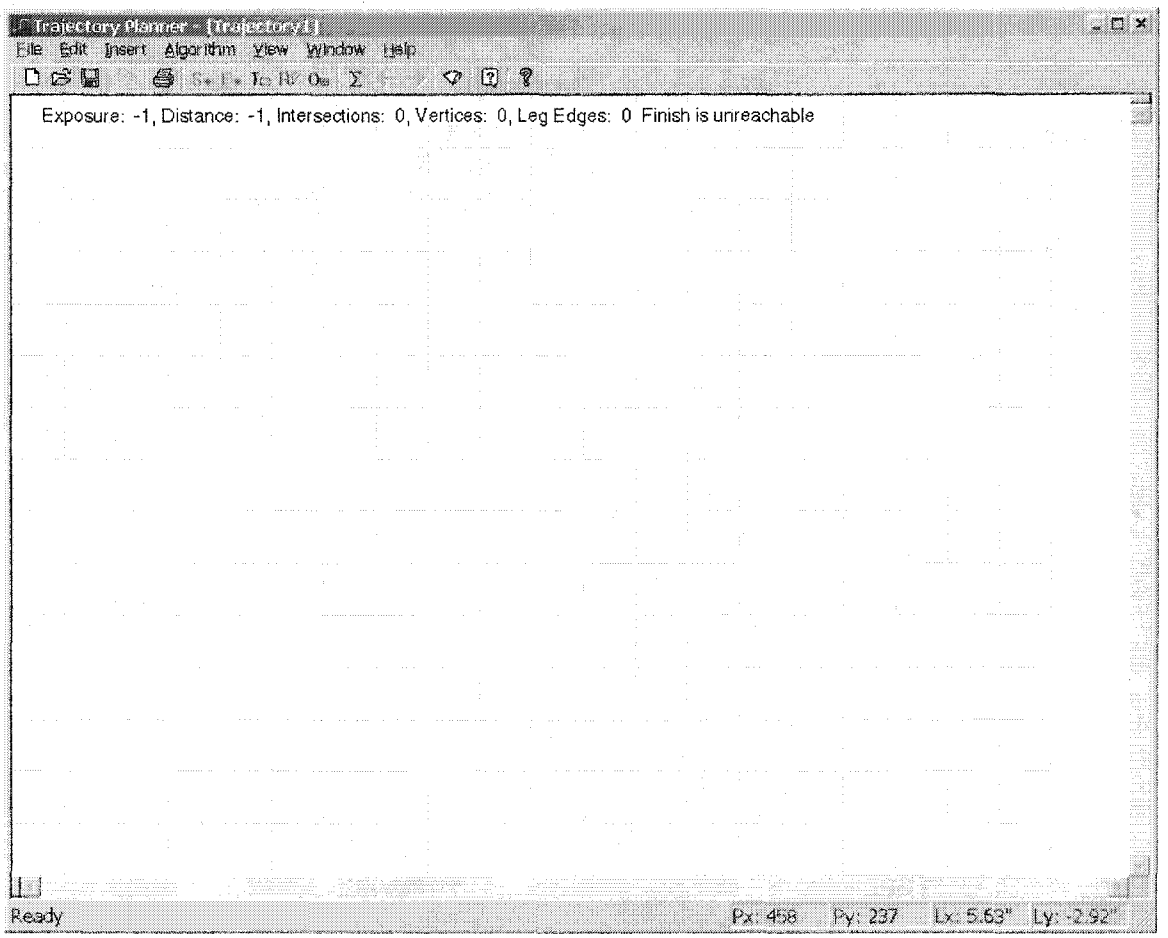

Figure 18. Trajectory Planner Initial Window

At any time the file menu allows the user to save, restore, print, or close the current trajectory model. Multiple models may be open simultaneously and the window menu contains options for selecting how the collection of models is displayed.

The user can add the start point, the target point, a turn region, a radar, or an obstacle in any order by selecting the object to insert from the menu or toolbar. Figure 19 shows the state of the model after insertion of the start and target points.

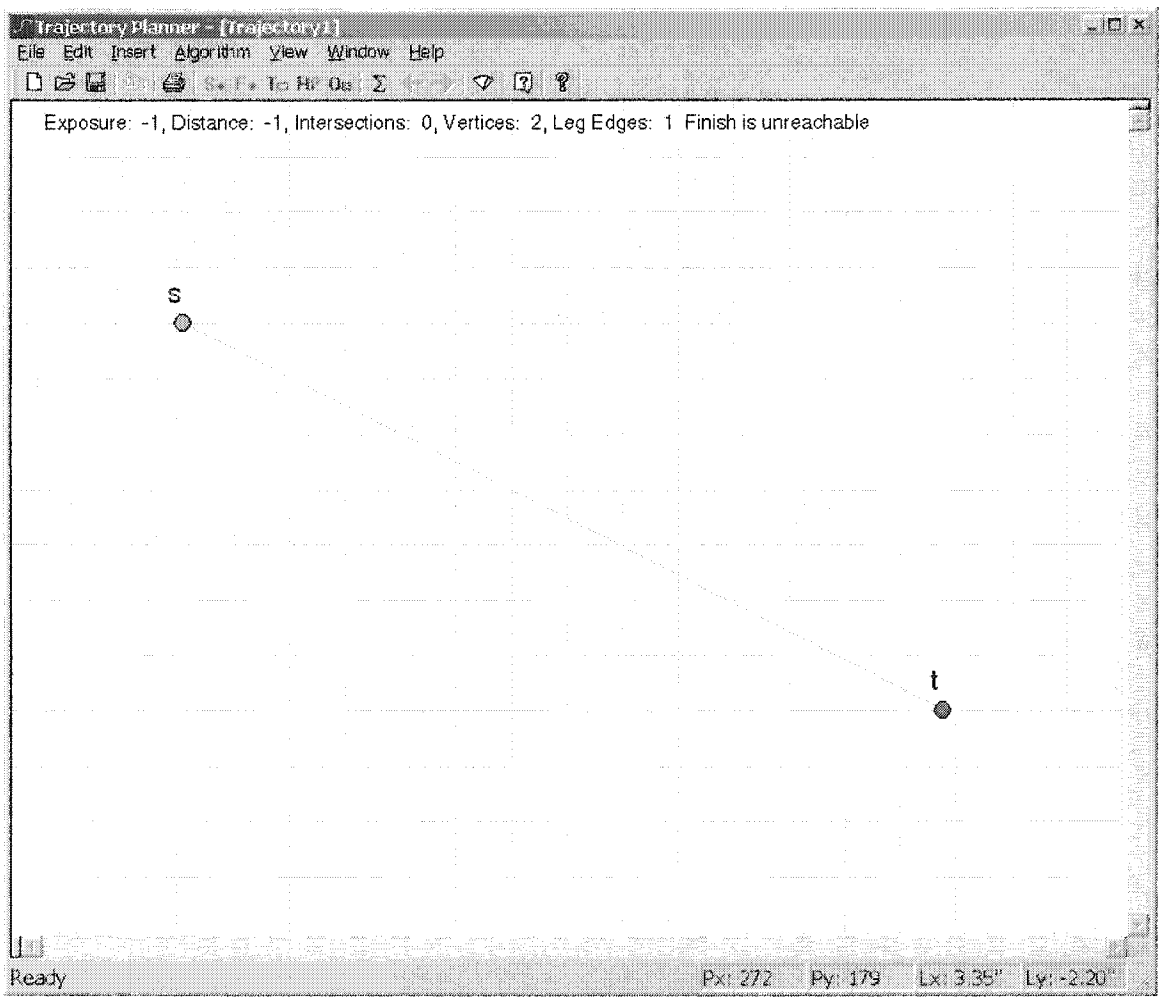

Figure 19. Trajectory Planner Showing Start and Target Points

Turn regions are added by first setting the insertion mode to turn region using the menu or toolbar and then clicking the mouse and dragging from the upper left to lower right corner of the desired rectangular region before releasing the mouse button. The number of vertices per turn region is selectable from one to eighty-one via the insert menu. The order of insertion of each turn region will determine the order of the associated leg edges in the trajectory. Figure 20 shows the display after the insertion of three turn regions.

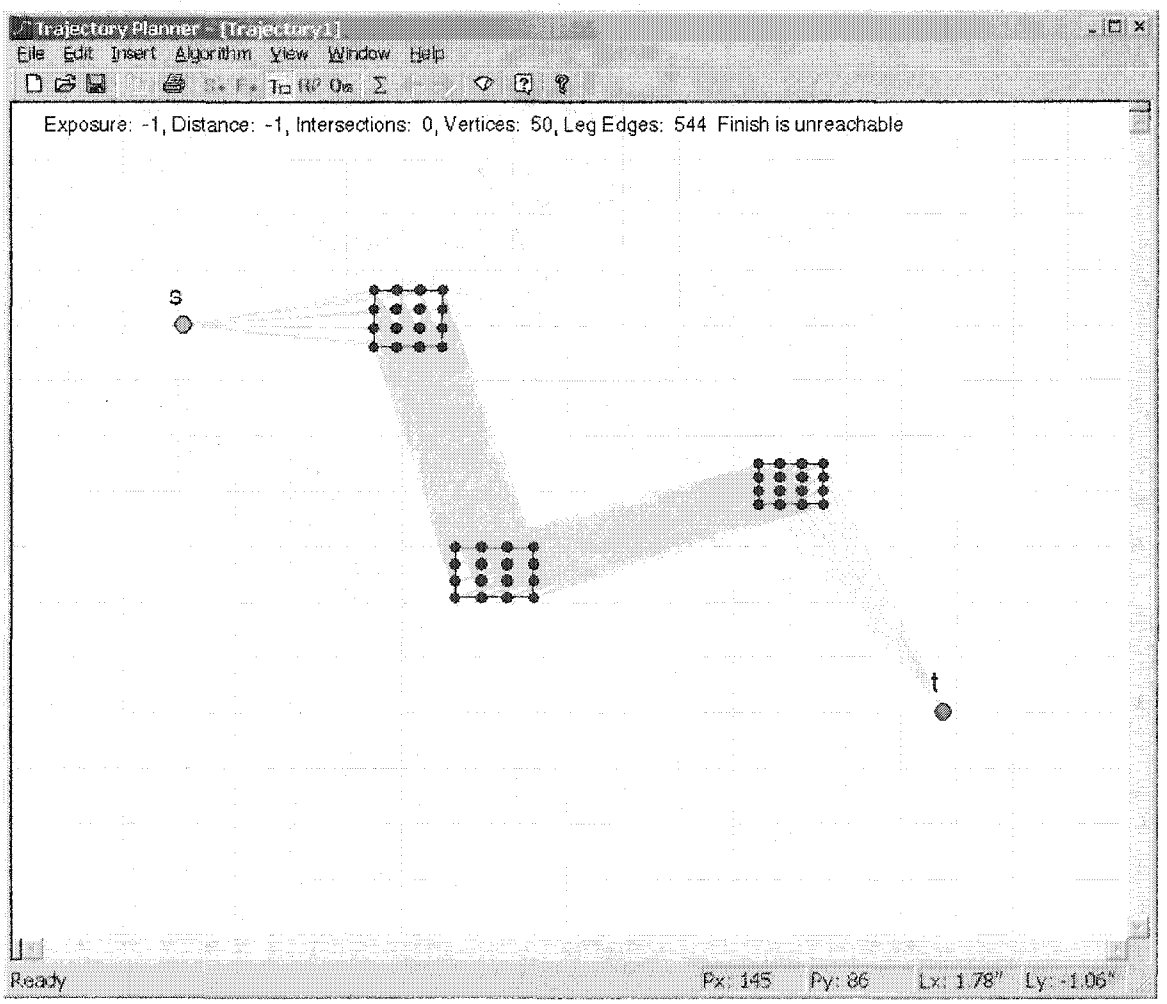

Figure 20. Trajectory Planner After Insertion of Turn Regions

The vertices comprising the turn region are displayed as blue dots and the leg edges are displayed as gray lines. The view menu allows the user to hide the leg edges if desired. The summary report area at the top of the screen displays the number of vertices and leg edges currently in the model.

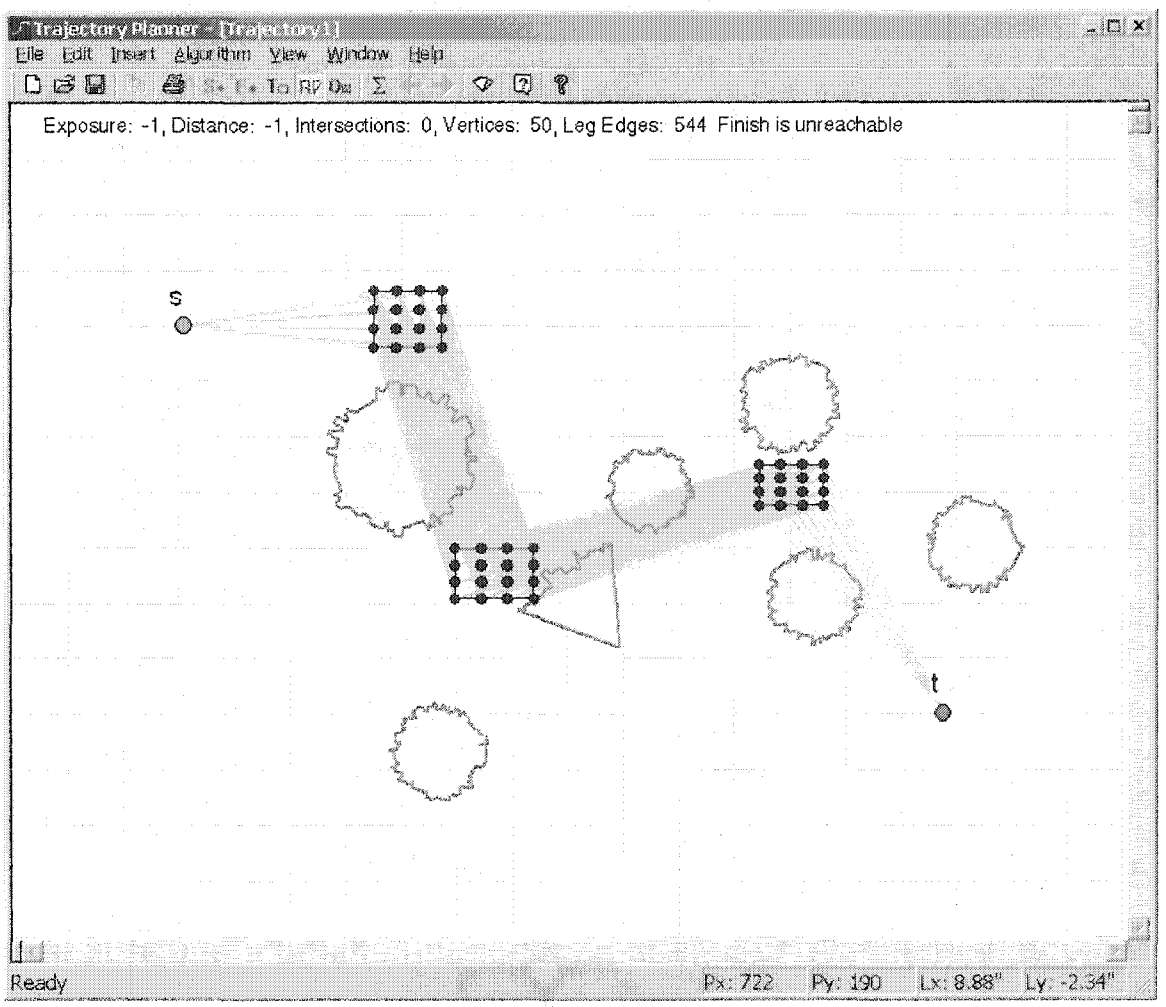

Figure 21. Trajectory Planner After Insertion of Radars

Radar coverage regions are added by first setting the insertion mode to radar using the menu or toolbar and then clicking the mouse and dragging from the center to the maximum detection range of the radar before releasing the mouse button. If the azimuth scan region for the radar is less than 360 degrees, after establishing the range of the radar,

but before releasing the primary mouse button, press and hold the secondary mouse button at the azimuth of the initial bounding radii of the sector, then move the mouse clockwise until the desired azimuth range is achieved and release the mouse to set the second bounding radii of the sector. Figure 21 shows the state of the model after seven radars have been added including one with azimuth coverage of approximately forty-five degrees. The range of the radars varies randomly to simulate uneven coverage caused by clutter-induced line of site limitations. This random variation may be toggled off or on via the insert menu.

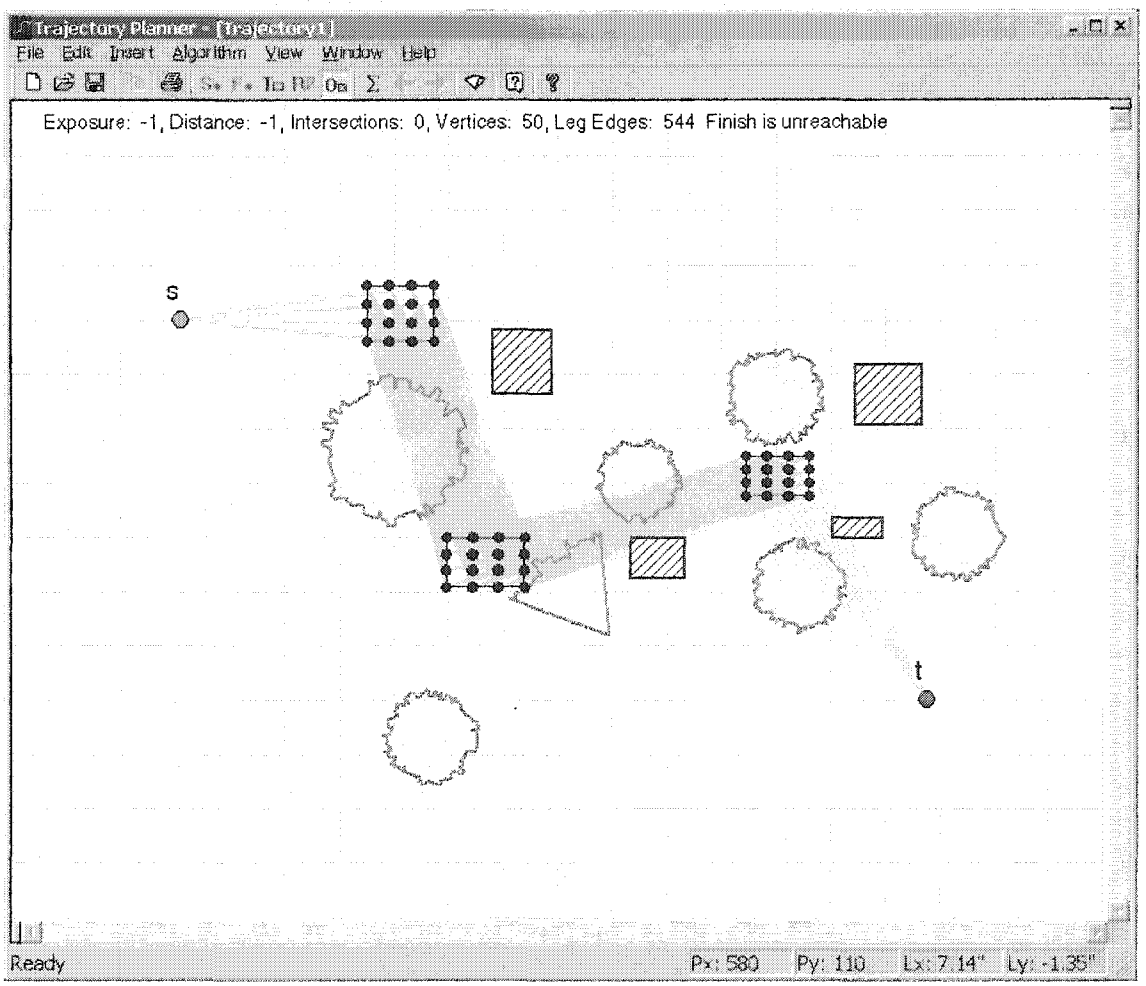

Figure 22. Trajectory Planner After Insertion of Obstacles

Obstacles are added by first setting the insertion mode to obstacle using the menu or toolbar and then clicking the mouse and dragging from the upper left to lower right corner of the desired rectangular region before releasing the mouse button. The resulting areas where flight is forbidden appear as red rectangles with diagonal hatched marks as shown in Figure 22.

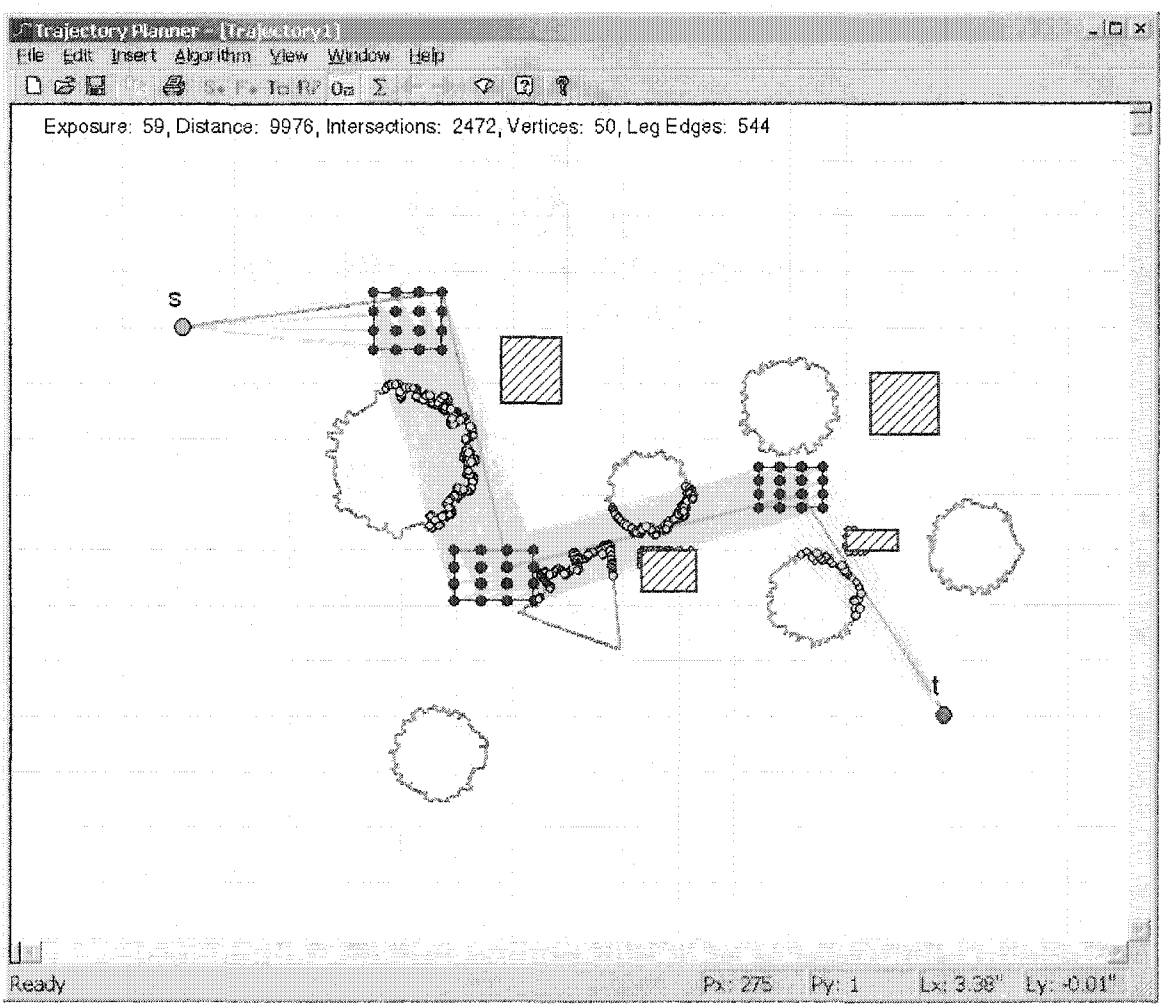

Figure 23. Trajectory Planner Showing Risk-Reduced Path

Once the desired model is constructed, the risk-reduced paths may be calculated by pressing the calculate trajectory button Σ, on the toolbar. The message area of the status bar will display the current state of the program as it progresses through the necessary calculations. Upon completion the reduced risk path is displayed with an orange line as shown in Figure 23.

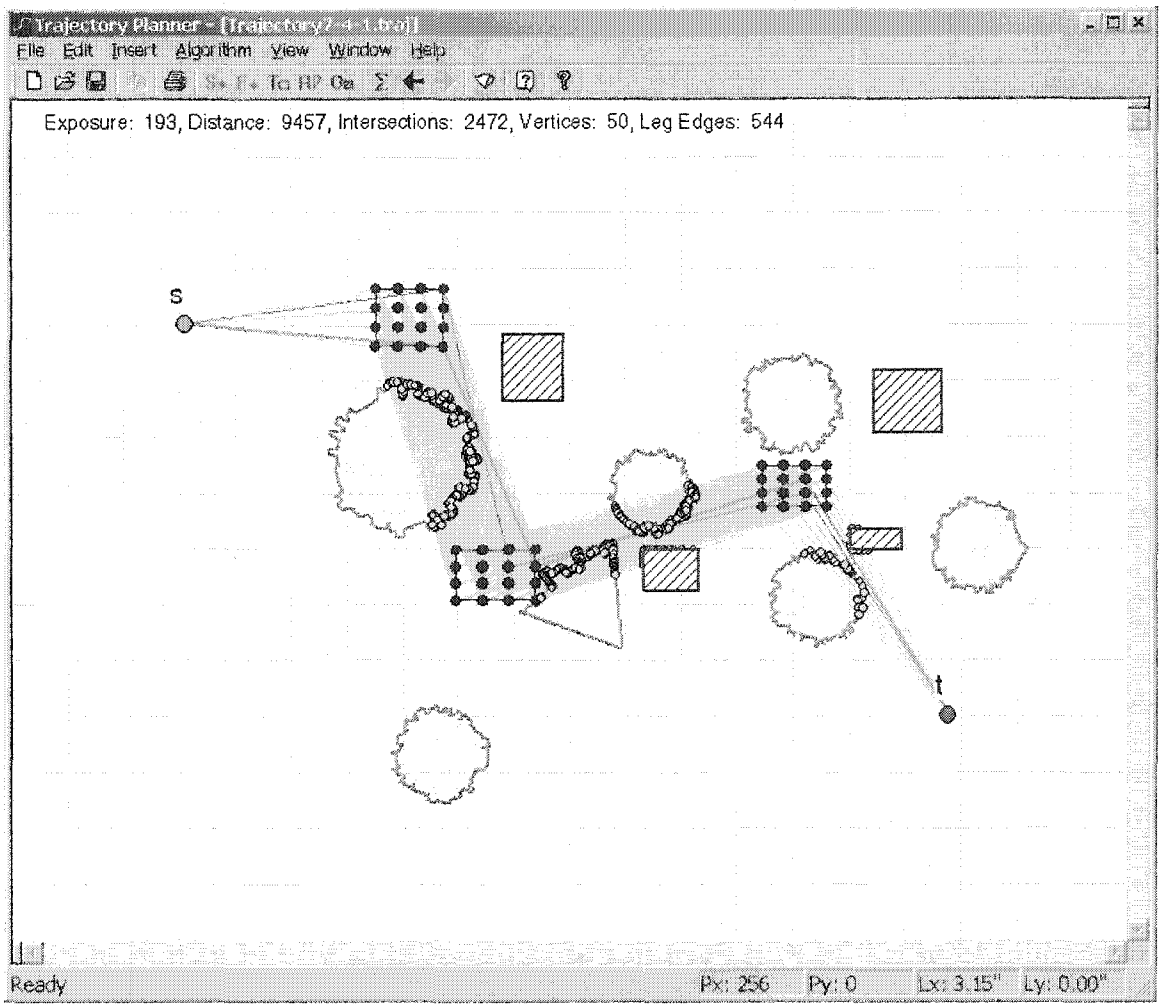

Figure 24. Trajectory Planner Showing Alternate Risk-Reduced Path

Intersection points between leg edges and radar polygons are rendered as small yellow dots. These may be hidden via an option on the view menu. The summary report area at the top of the screen indicates the unit length of exposure to enemy radar and the total distance of the trajectory. The number of intersection points is also displayed.

Alternate paths calculated using the edge elimination technique are selectable via the left and right arrow buttons on the toolbar. One such alternate path is shown in Figure 24. As different paths are selected, the exposure and distance values are updated in the summary report displayed at the top of the canvas.

Experimental Results on Test Input

In any experimental investigation, generation of test data is crucial. If test data can be generated in a meaningful way, then the proposed algorithm can be evaluated for performance. In our case, the complexity of the scene can be very high. The size, shape, and position of radars, obstacles, and turn regions can vary greatly. Generating these objects in a random way is a challenging problem in itself. Our general approach is to create test data where the solution for the optimum path is more or less known and then fire our algorithm and see how close the generated solution is to the expected result. If the generated solution and the known solution are close to each other, then the algorithm is probably working well. One obvious example of a known least risk path is a configuration in which there exists a k-legged route without any exposure to threats. Similarly we could create a data set in which the optimal path goes through a known exposure to threat. We have created several test input configurations and obtained the

solution by executing our algorithm. The paths as constructed by our algorithm are shown in Figure 25 through Figure 27 and are summarized in Table 2.

Table 2. Trajectory Planner Experimental Results

Test Description	Input File	Exposure	Distance	Result
No Exposure	T001	0	8768	Pass
Exposed to Threats	T002	615	7380	Pass
Finish Unreachable	T003	N/A	N/A	Pass

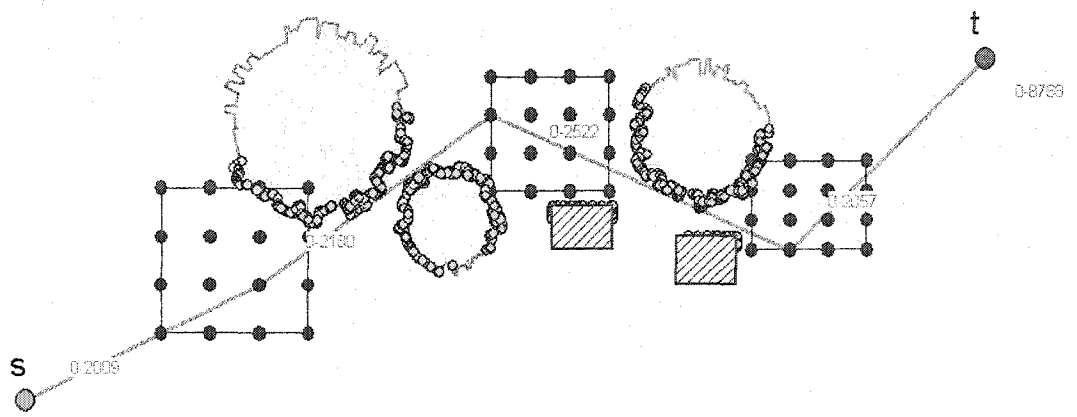

Figure 25. No Exposure Test Result

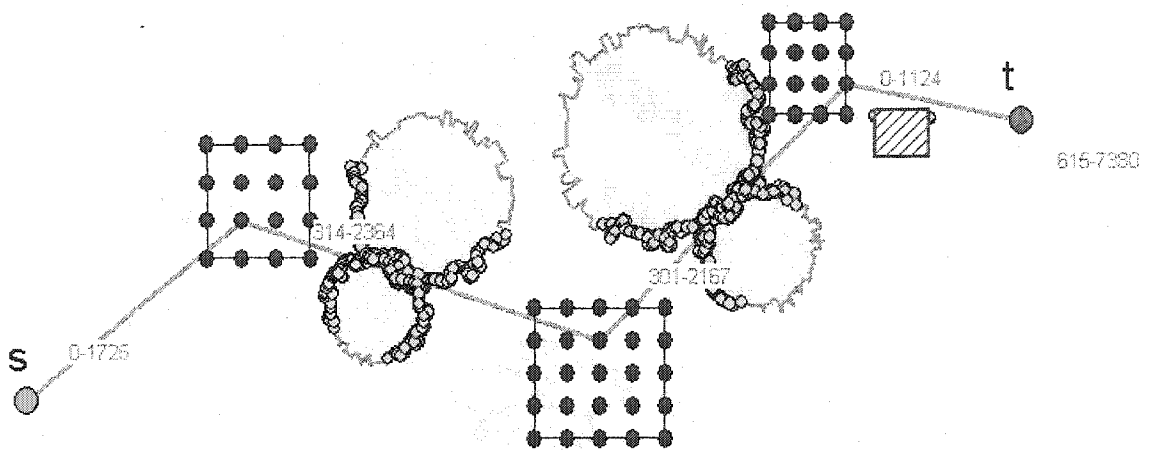

Figure 26. Exposed to Threats Test Result

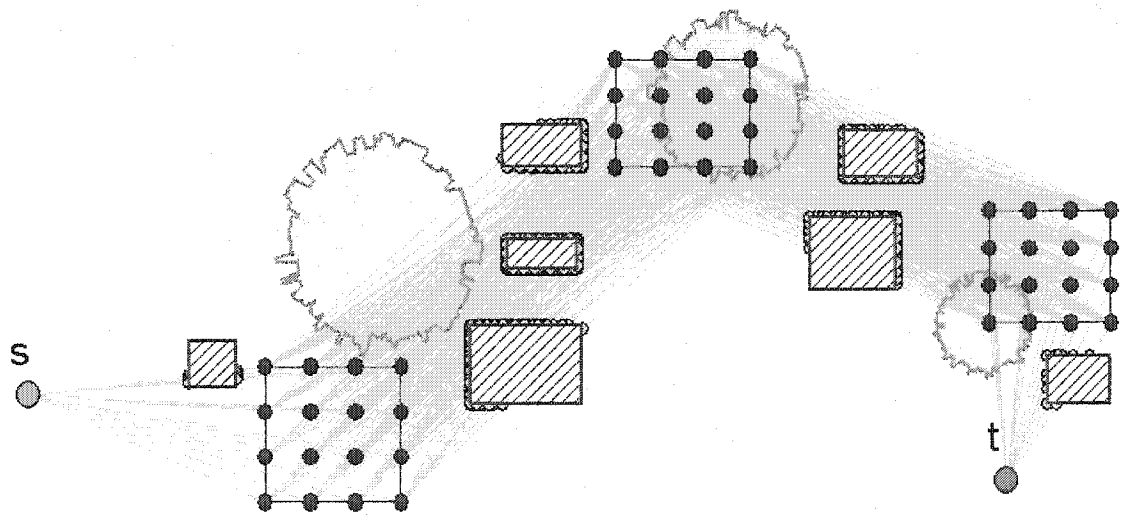

Figure 27. Finish Unreachable Test Result

Possible extensions to this application include deleting and editing turn regions, radars, and obstacles; and incorporation of angle constrained paths.

Plane Sweep Implementation

The plane sweep application is a multithreaded implementation of De Berg's plane sweep line segment intersection point algorithm [1]. The input to the algorithm is a vector of line segments. These can be imported from a tab-delimited text file or created within the program using the mouse or a random segment generator function. The program also implements a brute force intersection calculation algorithm to establish a baseline for performance comparison. The user interface is highly refined to allow for detailed interactive analysis of the operation of the algorithms. The contents of the primary data structures are displayed to aid in this analysis.

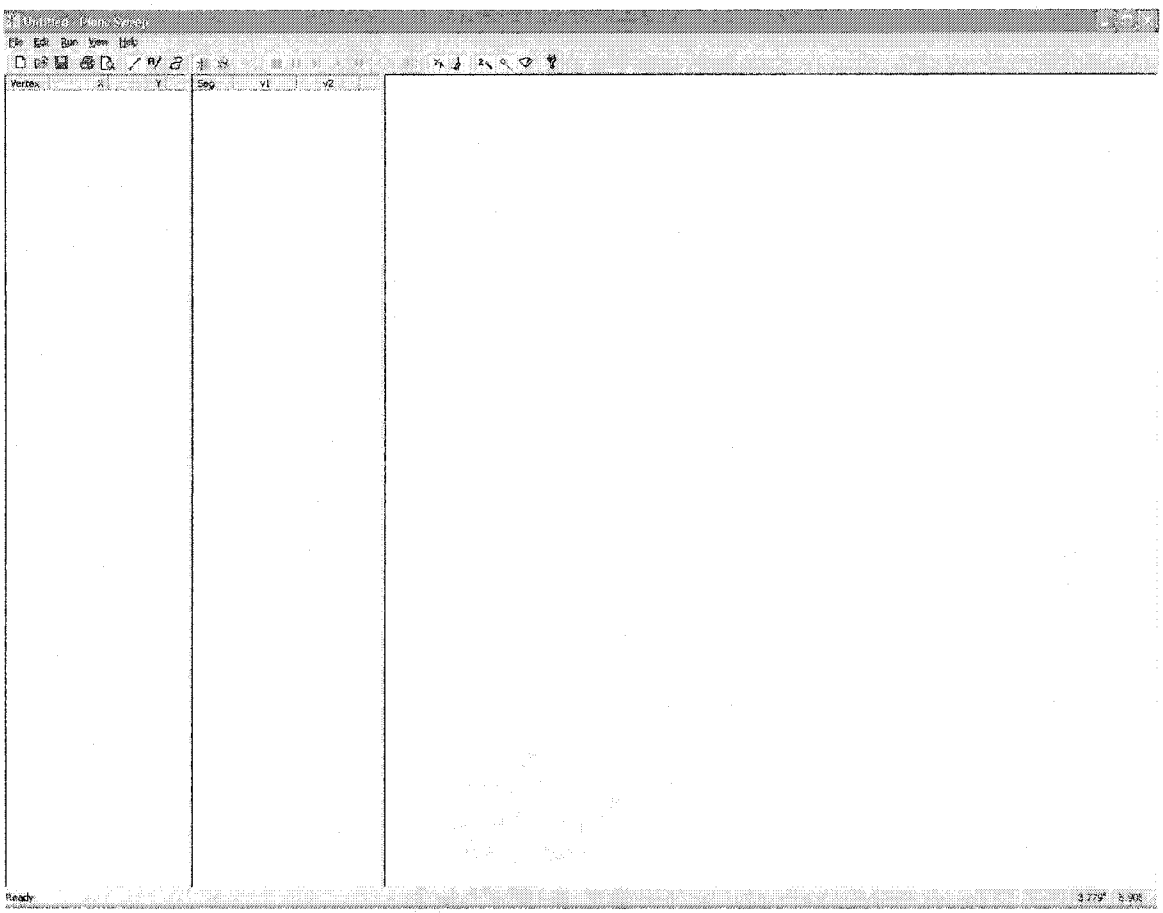

Figure 28. Plane Sweep Initial Display

Walk-Through of the Plane Sweep Application

When the program is first executed the initial window appears with a drop-down menu and toolbar above the blank canvas, and a status bar at the bottom of the screen as shown in Figure 28.

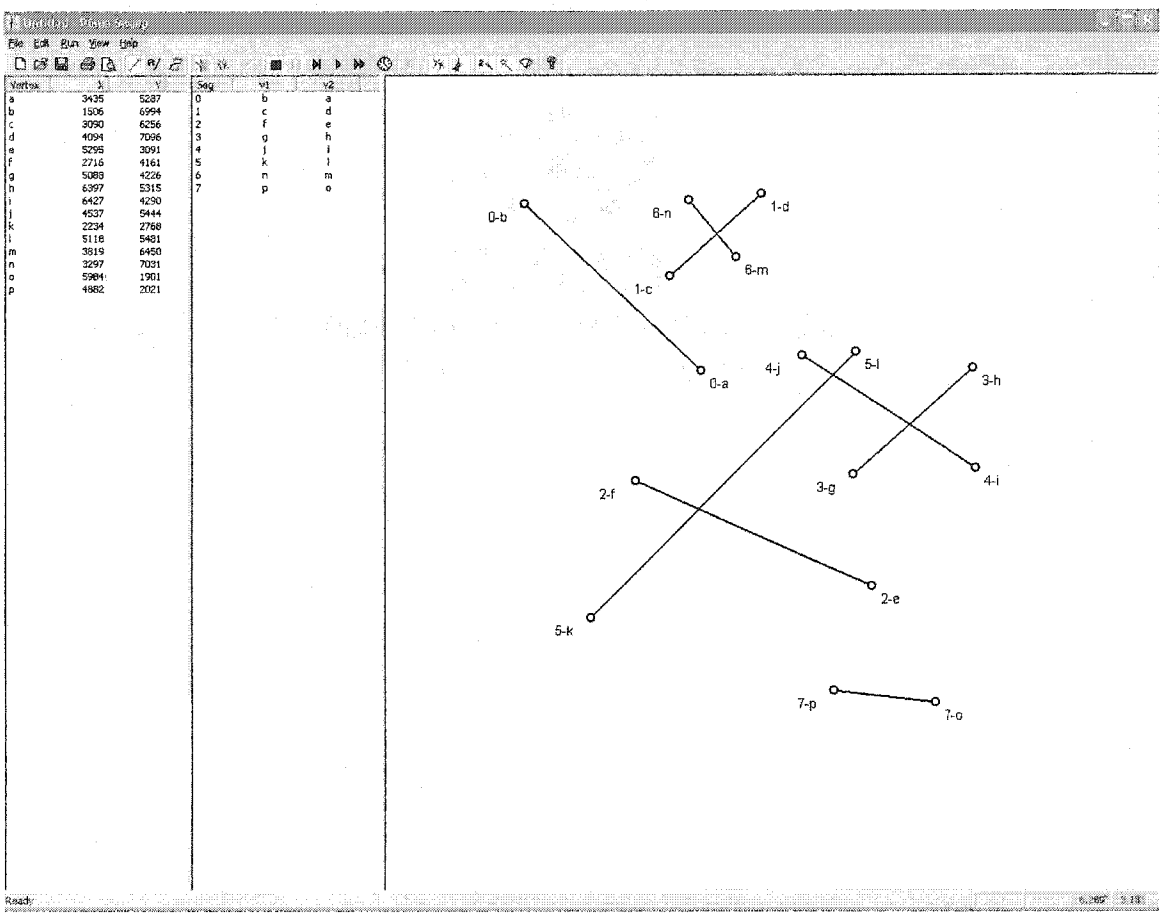

Figure 29. Plane Sweep After Segment Insertion

At this point, a previously stored program state may be recalled or a new set of line segments may be imported from a tab-delimited text file or created with the mouse in conjunction with menu and toolbar options. The status bar at the bottom of the screen contains a message area on the lower left that displays information pertinent to the

current state of the program. To the right of the message area are mouse position fields that report the logical position of the mouse as it is moved around the canvas.

As the user clicks and drags the mouse, line segments are drawn on the canvas as shown in Figure 29. When the mouse button is released the coordinates of the segment endpoints appear in the far left list. Each vertex is labeled alphabetically as it is created. To the right of this list are the numerically labeled segments and the corresponding vertices.

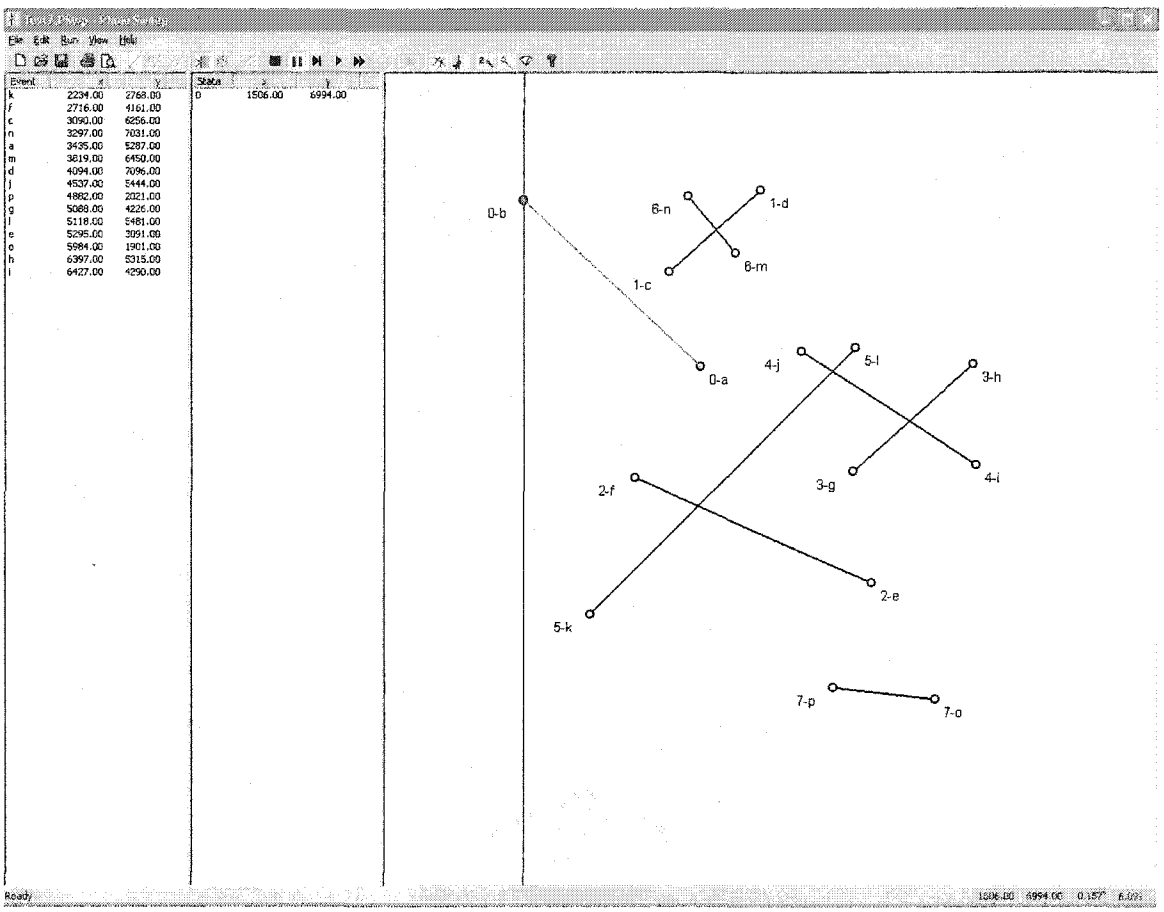

Figure 30. Plane Sweep Animation of Event Point

After the segments are inserted, the user selects either the plane sweep or brute force intersection calculation method from the toolbar or drop-down menu. Execution of the algorithm is controlled via the VCR-style buttons on the toolbar. The buttons allow the user to stop, pause, step, or run the algorithm. There are three versions of the run mode. One version runs at full speed, the second pauses at each step for a user specified delay time of up to 1 second, and the third runs a sequence of full speed timed tests. In addition, the user may optionally animate the operation of the algorithms.

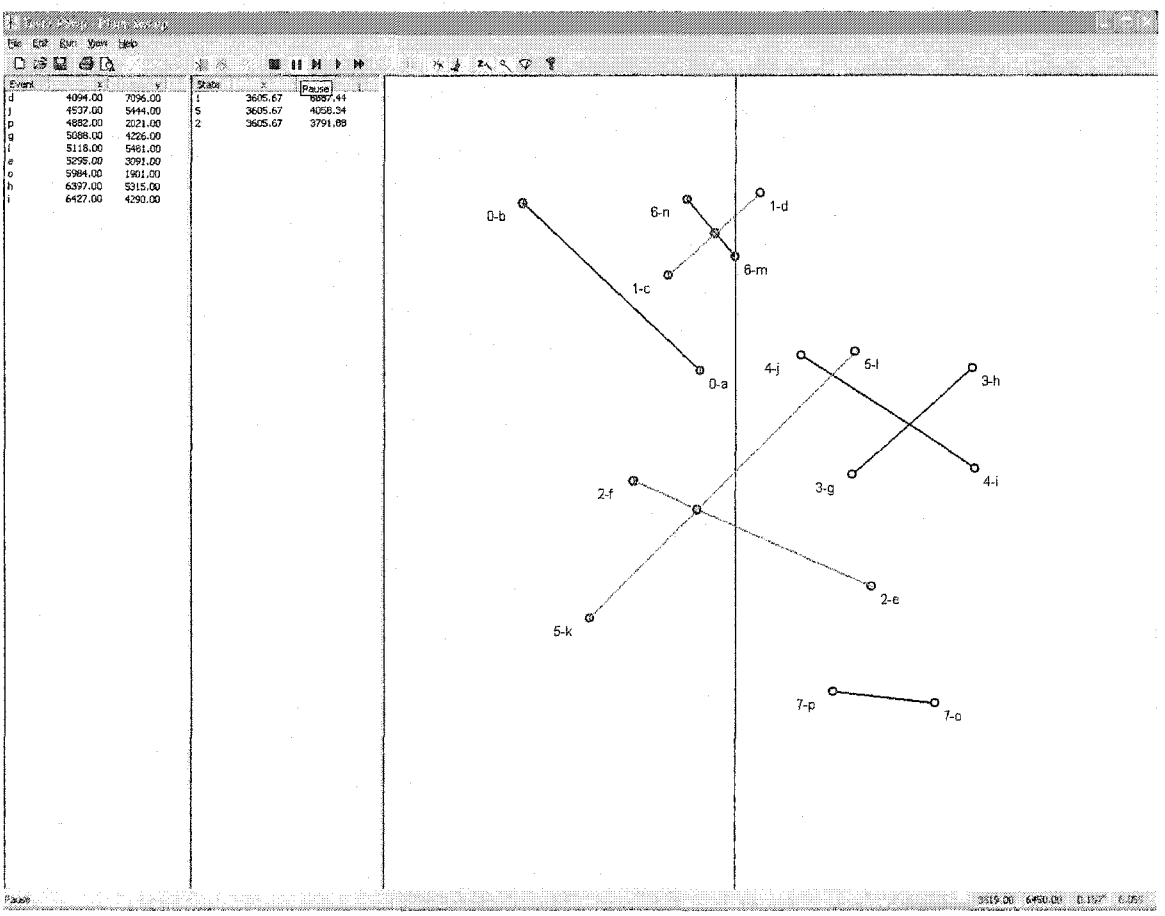

Figure 31. Plane Sweep Showing Intersection Points

In the plane sweep mode with animation enabled, the list of vertices is transformed into a list of event points. The event points are displayed in ascending order based on their x-coordinate values. As the current event point is removed from the event queue, it is displayed in red as shown in Figure 30. A vertical sweep line may also be displayed at the x-coordinate of the event point.

Each segment is colored green as it is added to the status structure and restored to black after it is removed. As intersection points are detected and calculated, they are displayed as blue dots as shown in Figure 31.

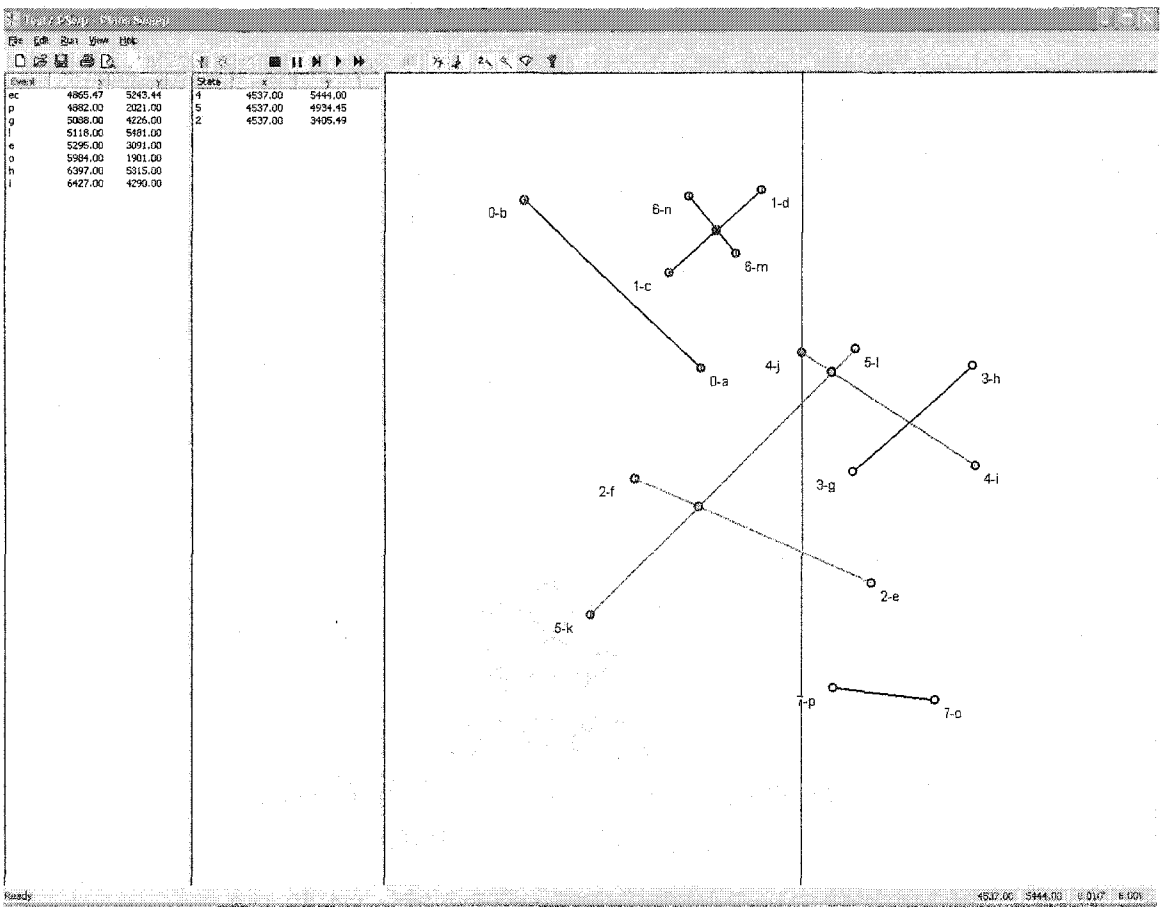

Figure 32. Plane Sweep Showing Potentially Intersecting Line Segments in Red

As a pair of line segments is passed to the intersection detection and calculation function, it is colored red by the program as shown in Figure 32.

Upon completion of the algorithm, the intersection points are displayed as blue dots and the data structure displays are restored to vertices and segments. If desired, the coordinates of the intersection points and the associated line segments can be displayed in a pop-up list as shown in Figure 33.

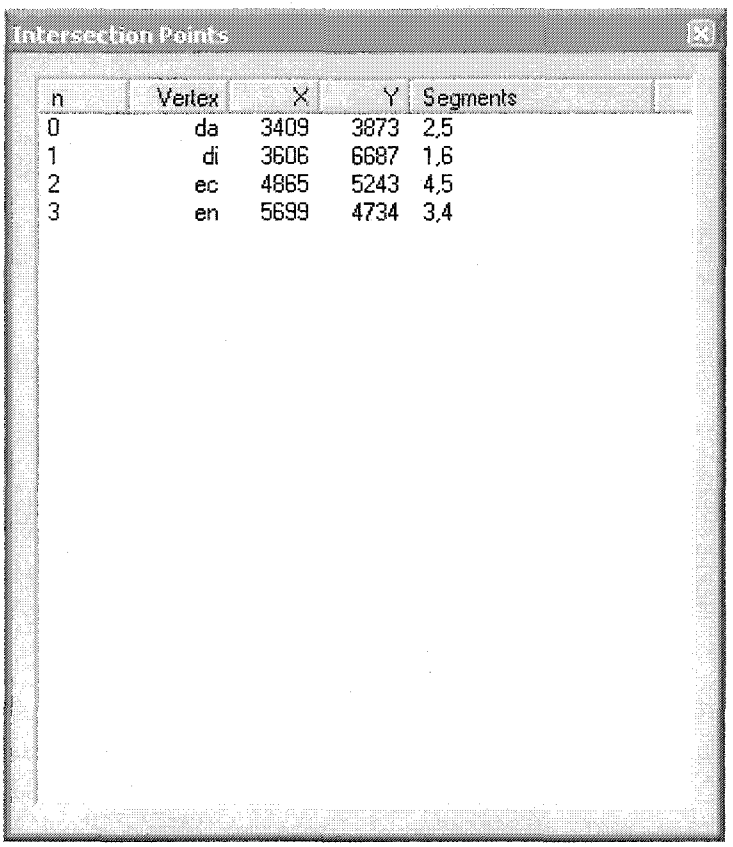

Figure 33. Plane Sweep List of Intersection Points

Multiple Risk-Reduced Path Planner Implementation

The multiple risk-reduced path planner implementation initially displays a graph settings dialog box as shown in Figure 34 that allows the user to set the number of turn regions for the stage graph as well as the number of vertices per region.

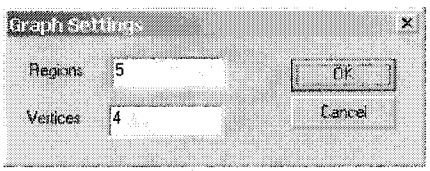

Figure 34. Path Planner Graph Settings Dialog

After the user enters the graph settings and presses the OK button to dismiss the dialog, the program generates the specified stage graph by adding a start point, the vertices of the turn regions, and a finish point. Edge weights are set to random values between 20 and 70. The shortest paths to each vertex are then computed. For the first turn region, the shortest paths to each vertex all originate at the start point. For subsequent turn regions, the program uses the method described in Chapter 3 to determine the weights of the m^2 paths into each vertex from the vertices of the previous turn region. The weights are then sorted, and the m least cost weights are stored in the vector of exposures for each vertex. Upon completion of the computation of the final vertex, the graph is displayed with the least cost path highlighted as shown in Figure 35. Vertices are small blue dots with the exception of the start and finish vertices, which are green and red respectively. Leg edges are light gray lines annotated with the cost of traversal.

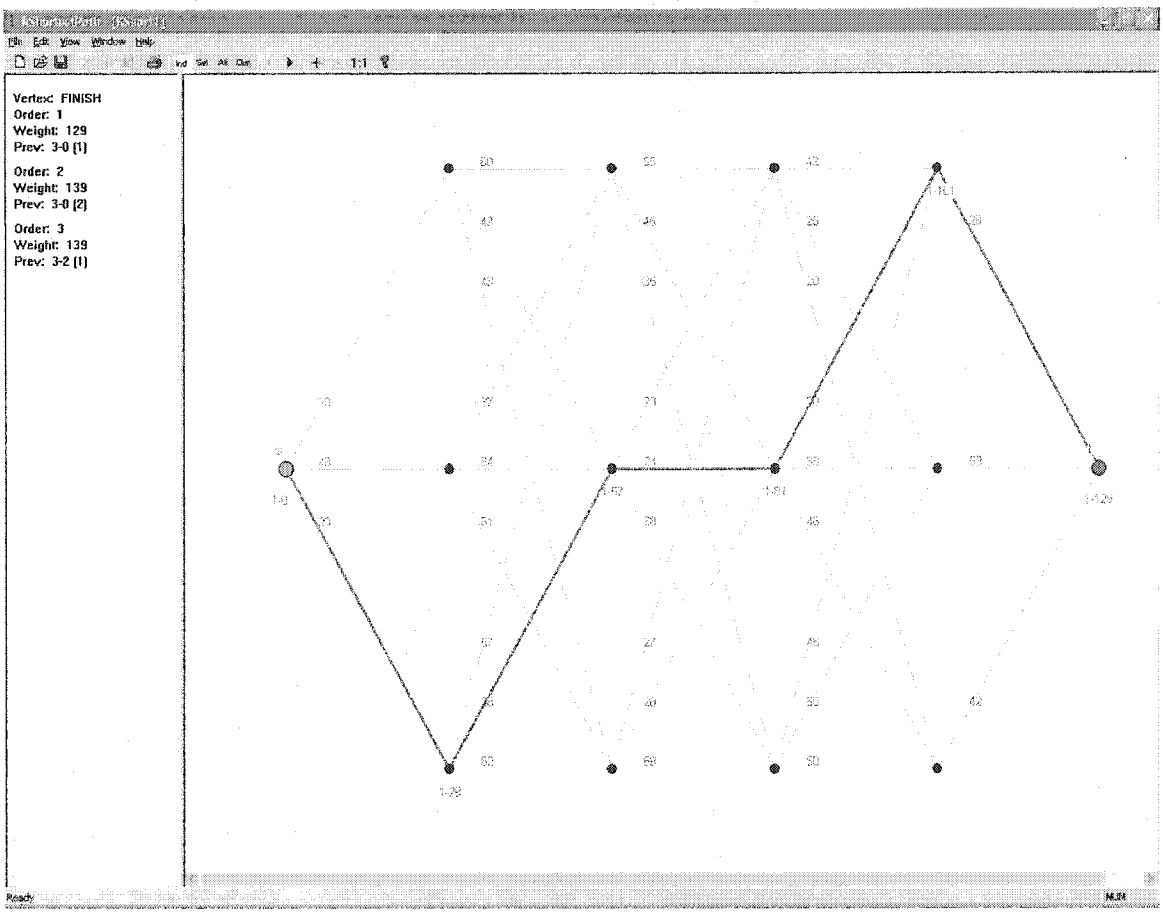

Figure 35. Path Planner Displaying Least Cost Path

The left pane of the main window details the order, weight, and previous point in the path for the *m* least cost paths to the selected vertex. By clicking on a vertex with the mouse, the user can display the relevant information for any vertex in the graph.

On the toolbar are four buttons used to display individual paths, selected paths, all paths, and disjoint paths. When the individual paths button is depressed, arrow buttons to the right allow the user to scroll through alternate paths. As each alternate path is selected, the corresponding leg edges are highlighted. In the selected paths mode, the user is presented with a pop-up list of paths from which to choose. When the all button is depressed all *m* paths are displayed simultaneously as shown in Figure 36.

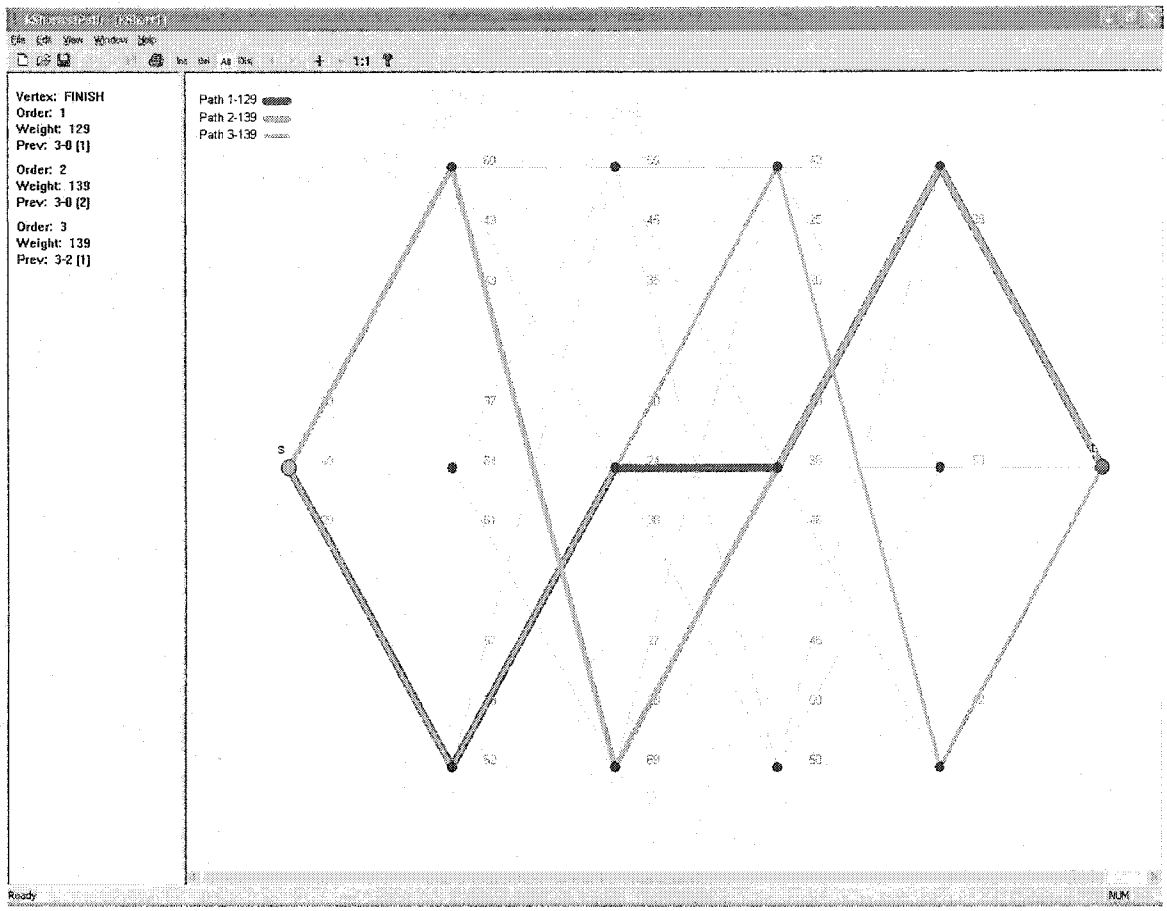

Figure 36. Path Planner Showing All *m* Paths

Each path is displayed in a different color and thickness to allow for discrimination of trajectories for paths that share a common edge. The shortest path is displayed using the thickest line weight. The last option allows the user to display *m* disjoints paths.

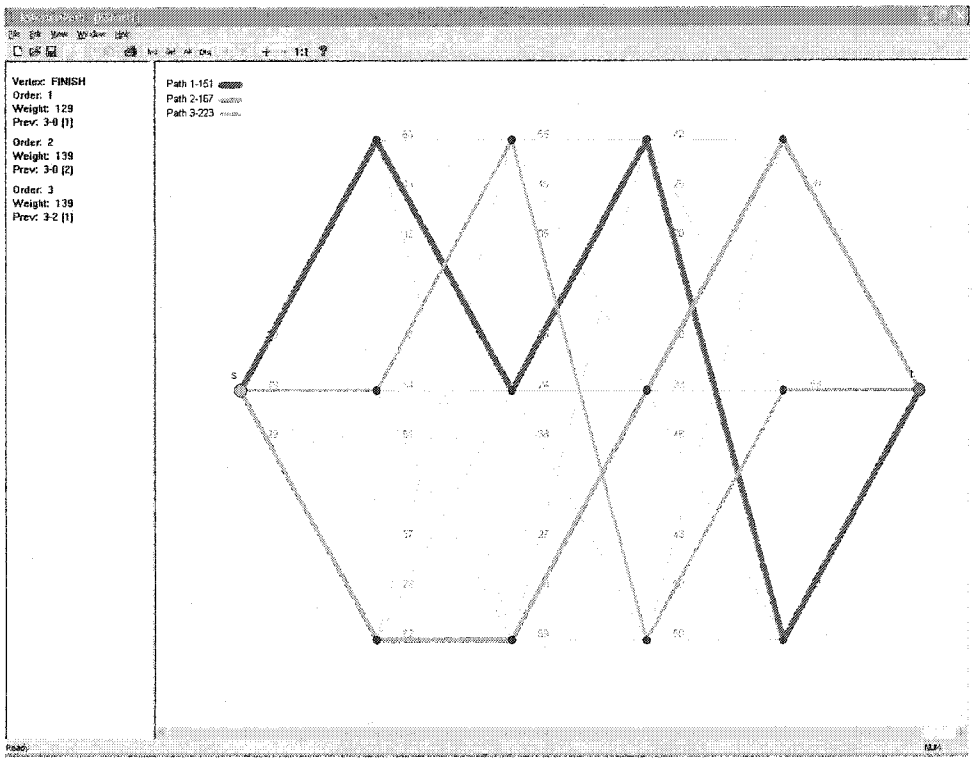

Figure 37. Path Planner Showing Disjoint Paths

The paths are formed using the rank ordering approach described in Chapter 3. Colors and weights are varied in the same manner as the all paths mode.

Difficulties Encountered

During the course of implementation we experienced some pitfalls that might beset any researcher or programmer attempting to implement geometric algorithms. Our intersection detection algorithm identified duplicate intersection points where a leg edge intersected the vertex of a radar polygon since there are two edges associated with each polygon vertex. This in turn caused our accumulated segment weight algorithm to produce incorrect results. Another anomaly encountered was an inexplicable loss of some data points when sorting the list of intersection points using the sort function contained in

the C++ standard template library. When we switched containers from a list to a set, which is implemented as an ordered binary tree, the data structure behaved as expected without any loss of data.

Identifying duplicate intersection points can pose a problem if the resulting points are maintained as floating-point numbers. Equality of floating-point numbers is ascertained by checking to see if the absolute value of the difference between the two numbers is less than the value of the associated FLT_EPSILON or DBL_EPSILON value found in the `float.h` header file.

CHAPTER 5

CONCLUSION AND EXTENSIONS

We presented a brief review of existing path planning algorithms. Next we presented an $O(m^2p)$ algorithm for computing a k-legged risk-reduced path in the presence of enemy radar sources and obstacles as well as algorithms for computing multiple risk-reduced paths. Finally we implemented the algorithms in the Visual C++ programming environment and executed them on several sets of test data.

There are several areas that may benefit from extensions and further exploration. An extension to the KRPP algorithm could be the computation of k-legged risk-reduced paths subject to turn radius constraint. Our trajectory model focuses on weighted regions where there is greater than 50 percent probability of detection from enemy radar for a uniform scatterer. In real situations there could be a more realistic exposure model. It may be necessary to allow for a gradient probability of detection based upon the momentary range of the vehicle to the radar source. Other factors that alter the probability of detection include terrain-induced variations in radar subclutter visibility, the aspect angle of the scatterer to the radar antenna, and the localized impact of electronic counter measures.

The simplification of risk regions in our model does not take into account the number or type of radar systems that can detect the aircraft during its mission. Should the adversary possess an integrated air defense system, identification by multiple sources

may increase the lethality of the enemy's weapons. Radar systems that are designed for tracking and missile guidance might be especially dangerous or may only be dangerous if the vehicle is also within range of a surveillance radar system that would alert the missile system to the presence of a target and cue the tracker with its general direction and range. The level of peril for an aircraft flying within the kill zone of surface to air missile system is typically inversely proportional to its range from the system.

The reduced-risk path generated by our algorithm is a sequence of piece–wise linear segments. If we try to curve the path, the exposure to threat may change. It would be interesting to enhance the algorithm so that the generated path is not necessarily piece-wise linear segments.

APPENDIX

Trajectory Planner UML Class Diagrams

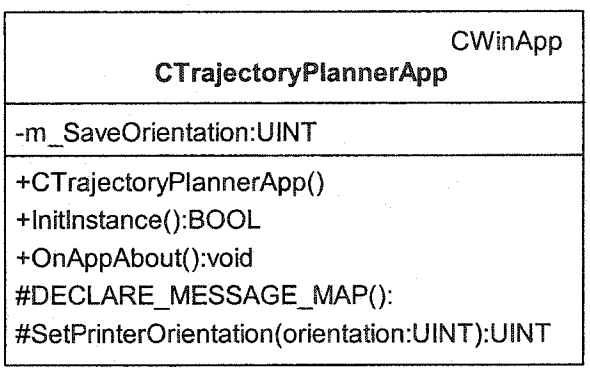

Figure 38. CTrajectoryPlannerApp Class Diagram

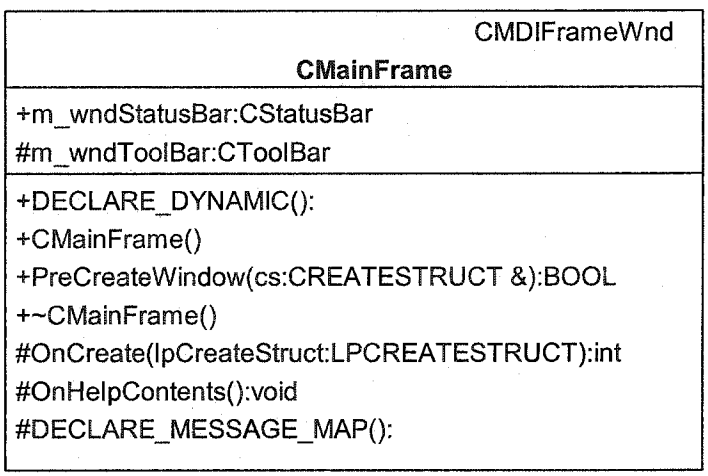

Figure 39. CMainFrame Class Diagram

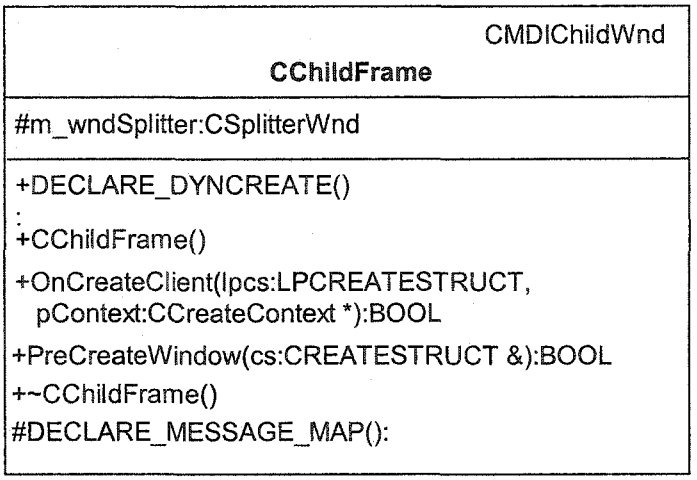

Figure 40. CChildFrame Class Diagram

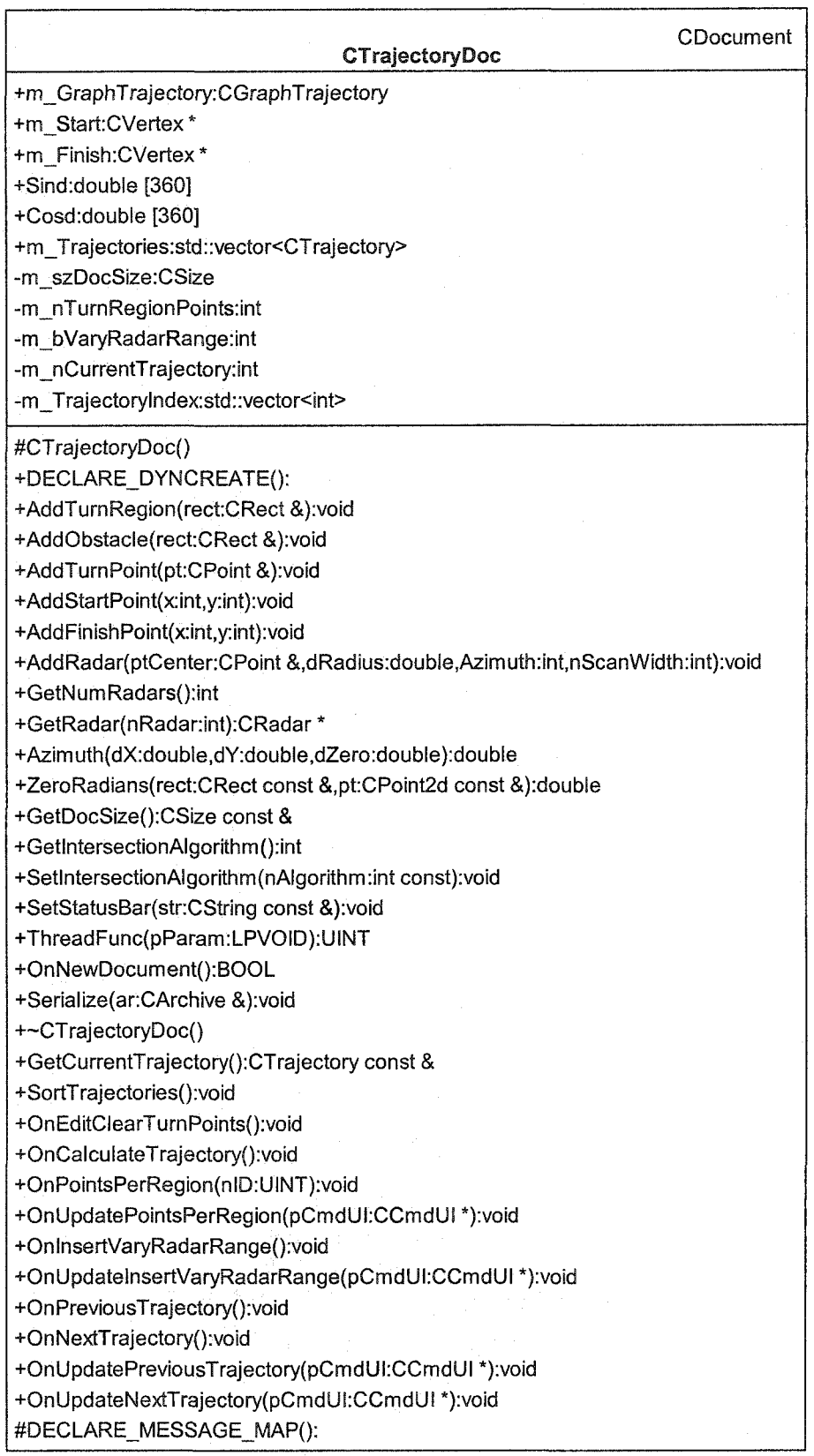

Figure 41. CTrajectoryDoc Class Diagram

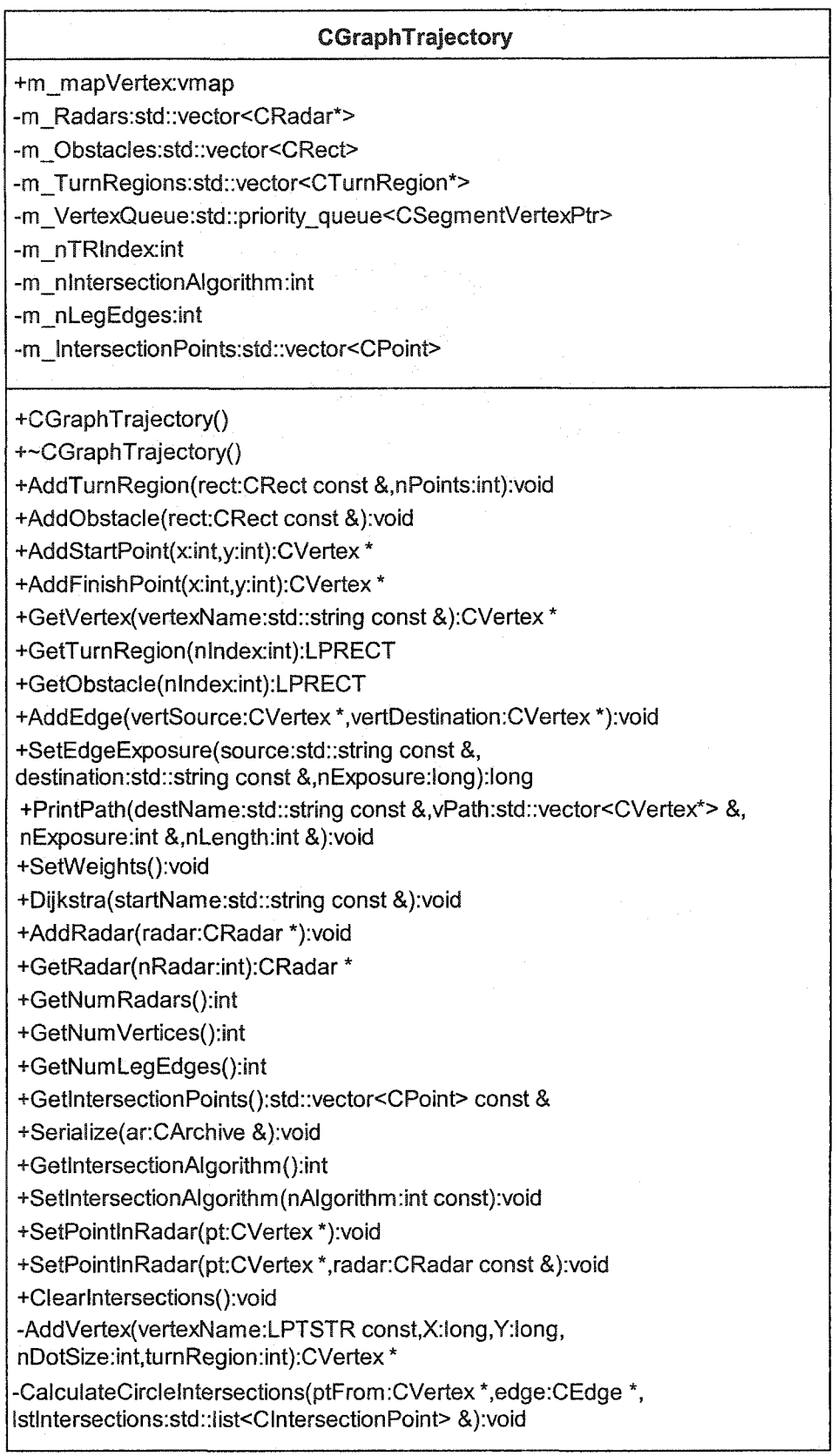

Figure 42. CGraphTrajectory Class Diagram Part 1 of 2

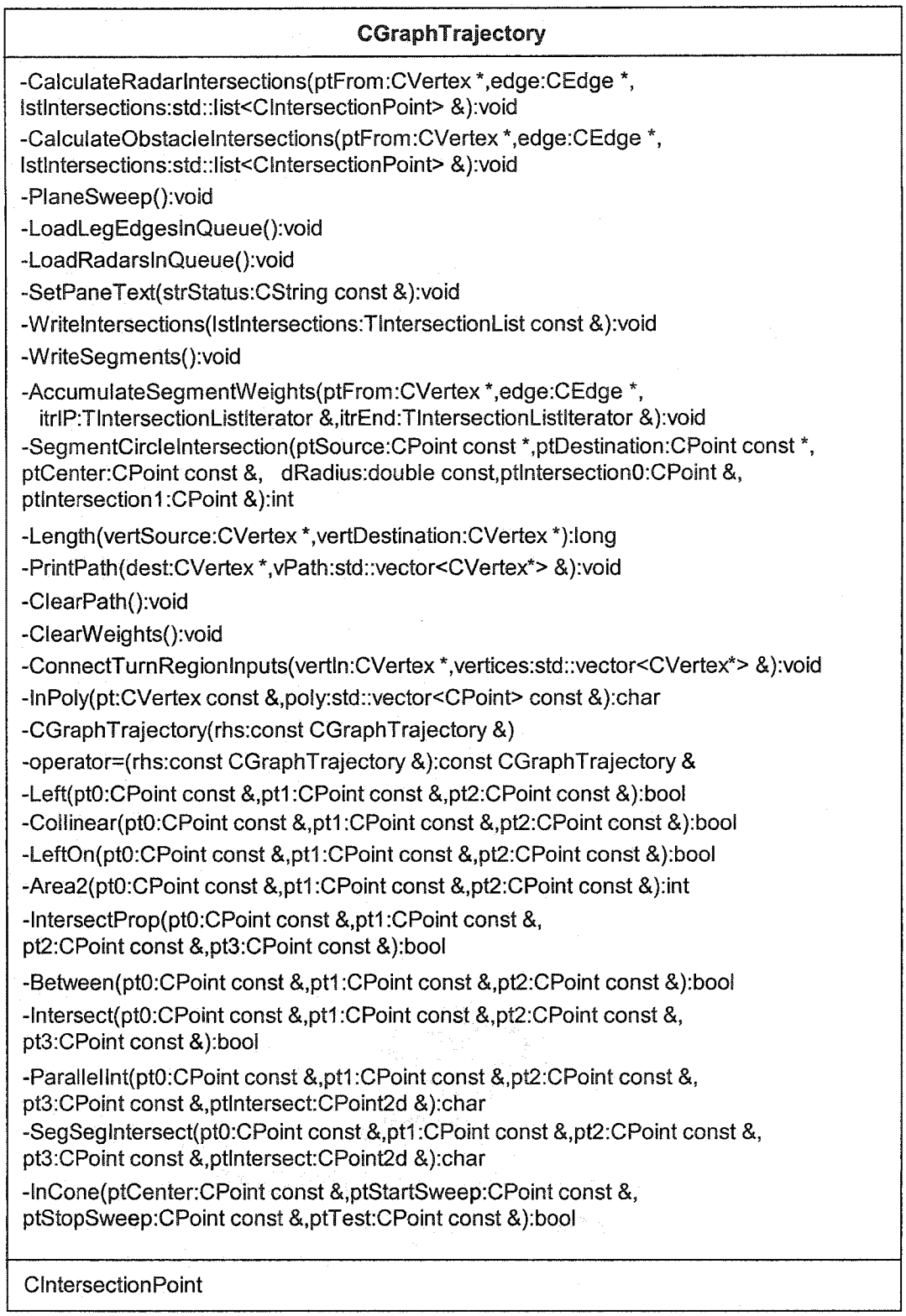

Figure 43. CGraphTrajectory Class Diagram Part 2 of 2

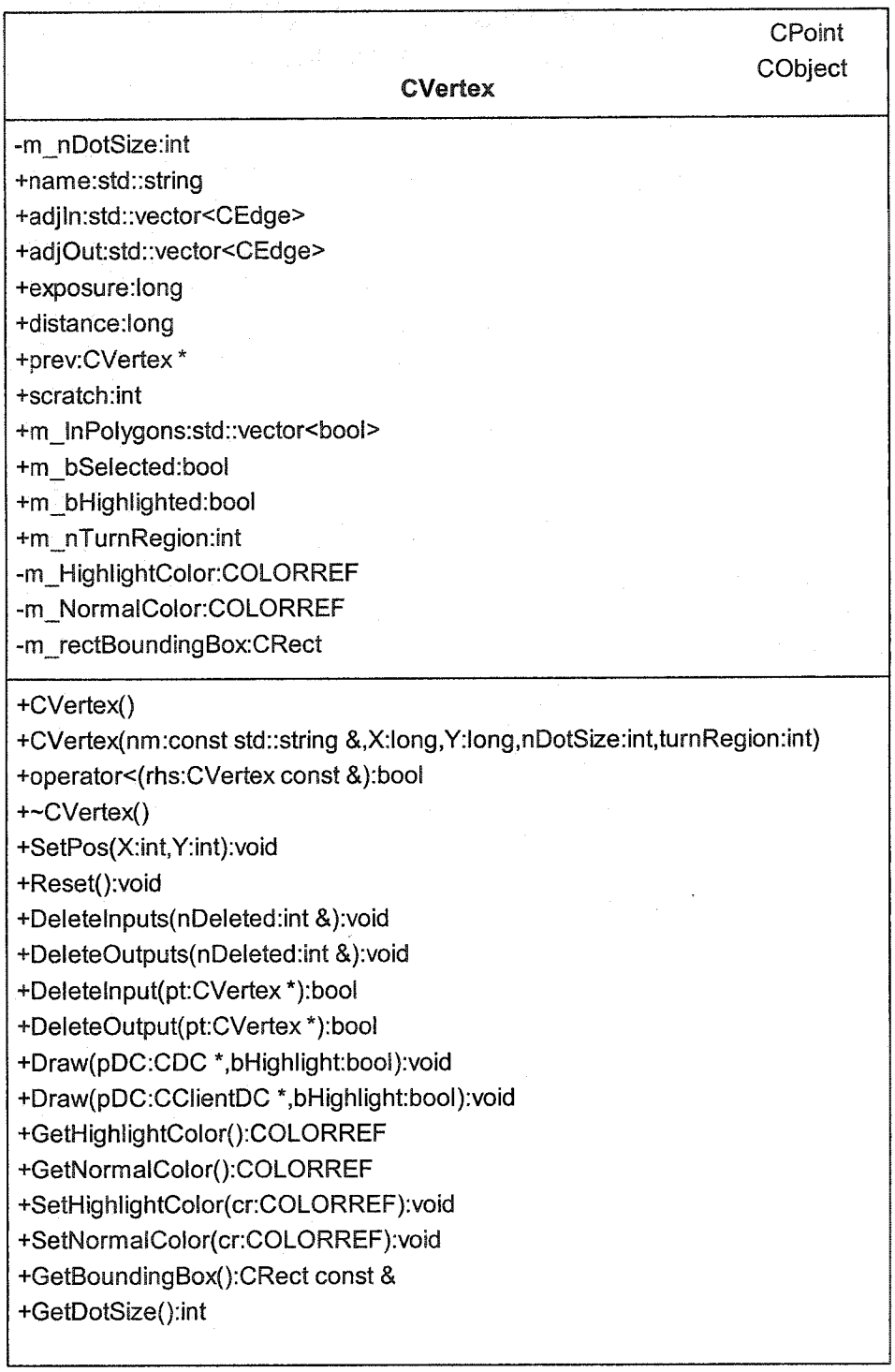

Figure 44. CVertex Class

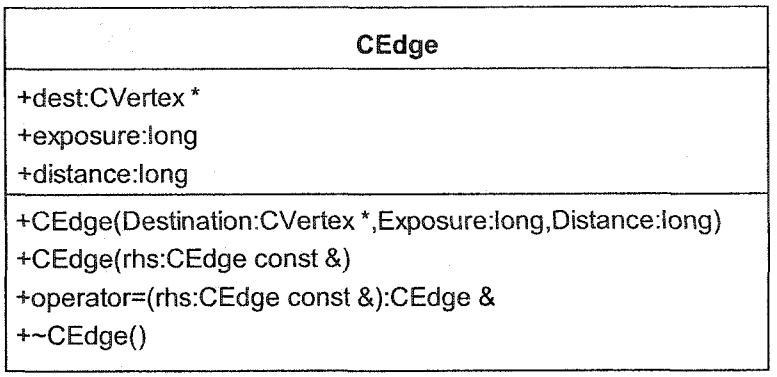

Figure 45. CEdge Class Diagram

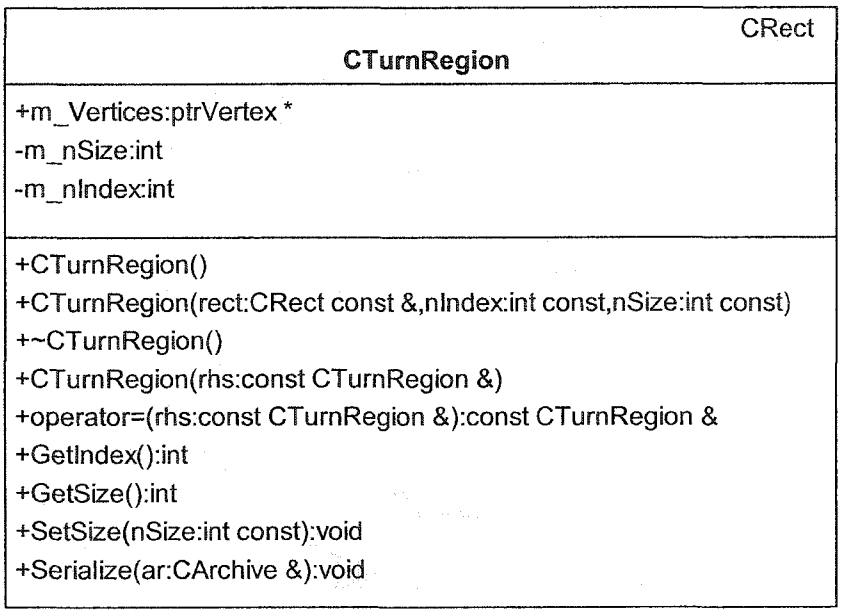

Figure 46. CTurnRegion Class Diagram

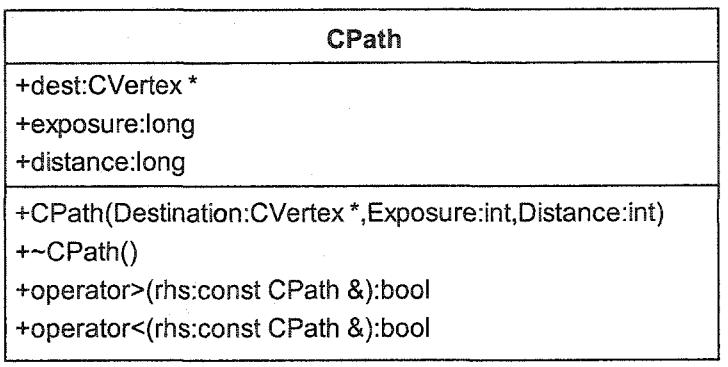

Figure 47. CPath Class Diagram

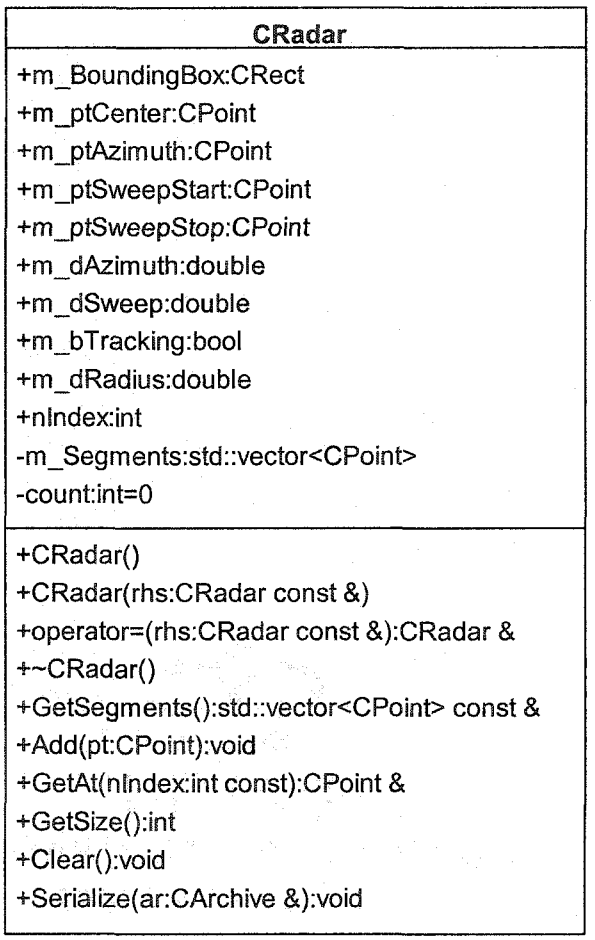

Figure 48. CRadar Class Diagram

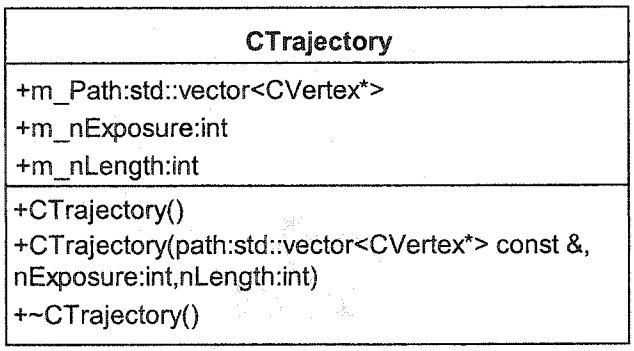

Figure 49. CTrajectory Class Diagram

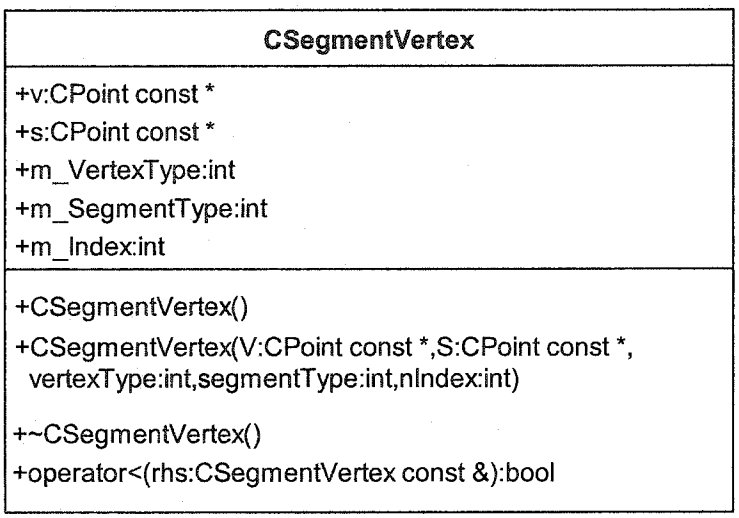

Figure 50. CSegmentVertex Class Diagram

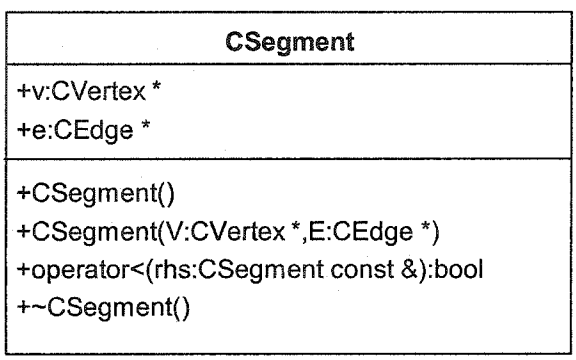

Figure 51. CSegment Class Diagram

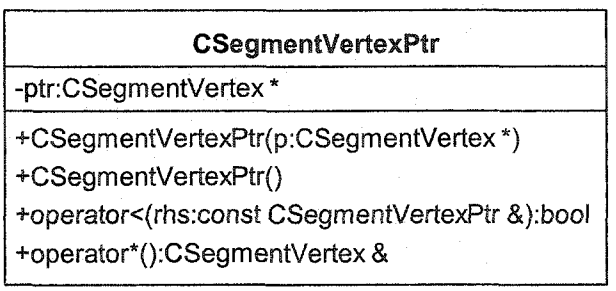

Figure 52. CSegmentVertexPtr Class Diagram

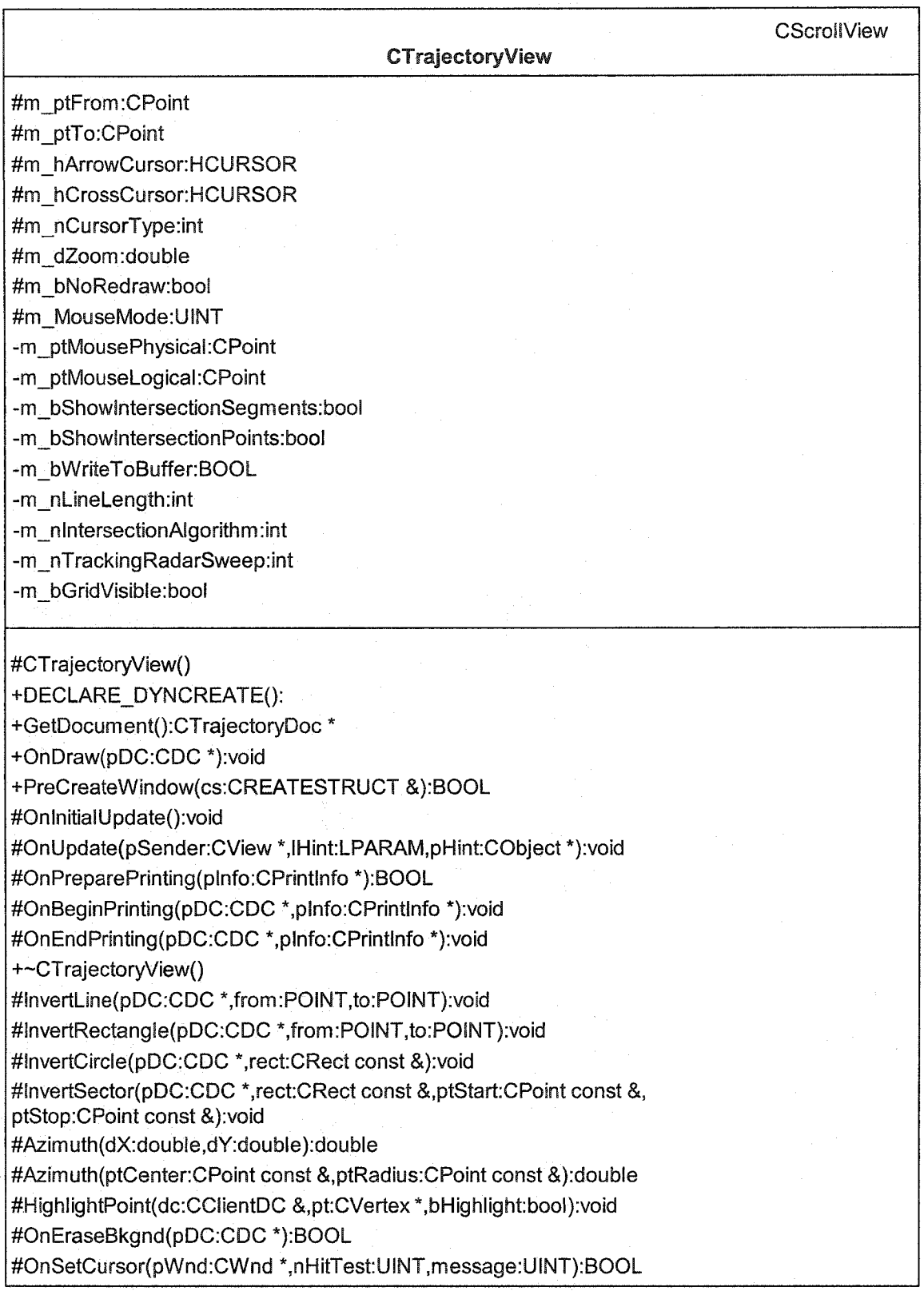

Figure 53. CTrajectoryView Class Diagram Part 1 of 2

Figure 54. CTrajectoryView Class Diagram Part 2 of 2

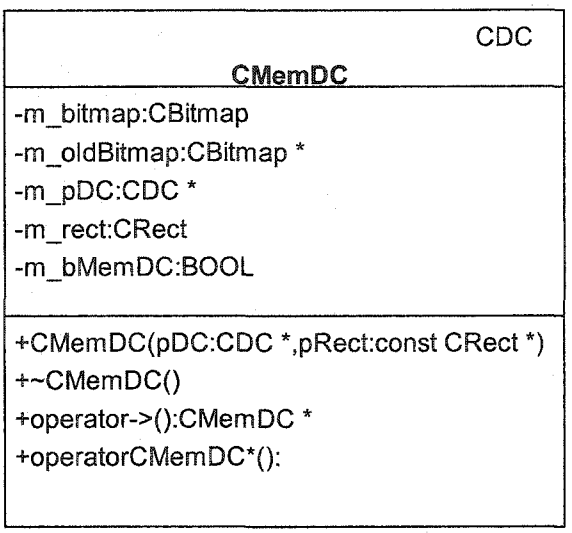

Figure 55. CMemDC Class Diagram

REFERENCES

[1] M. de Berg, M. Kreveld, M. Overmars, and O. Schwarkopf, *Computational Geometry: Algorithms and Applications*, Springer Verlag, Berlin, Germany, 1997.

[2] A. Boroujerdi and J. Uhlmann, An Efficient Algorithm for Computing Least Cost Paths with Turn Constraints, *Information Processing Letters*, Volume 67, Issue 6, Sept. 1998, pp. 317-321.

[3] A. Bowyer and J. Woodwark. *A Programmer's Geometry*. Butterworths, London, England, 1983, pp. 25-26.

[4] J. Canny and J. Reif. New Lower Bound Techniques for Robot Motion Planning Problems, *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, 1987, pp. 49-60.

[5] D. Z. Chen, K. S. Klenk, and H. T. Tu, Shortest Path Queries Among Weighted Obstacles in the Rectilinear Plane, *SIAM Journal on Computing*, 2000 Vol. 29, No. 4, pp. 1223-1246.

[6] D. Eppstein, Finding the k Shortest Paths, *SIAM Journal on Computing*, 1998 Vol. 28, No. 2, pp. 652-673.

[7] L. P. Gewali, A. C. Meng, and J. S. B. Mitchell, Path Planning in $0/1/\infty$ weighted regions with Applications, *ORSA Journal on Computing*, (1990) 2, pp. 253-272.

[8] C. Mata and J. S. B. Mitchell, A New Algorithm for Computing Shortest Paths in Weighted Planar Subdivisions, *Proceedings of the 13th Annual ACM Symposium on Computational Geometry*, 1997, pp. 264-273.

[9] M. Lanthier, A. Maheswari, and J. R. Sack, Approximating Shortest Paths on Weighted Polyhedral Surfaces, *Algorithmica*, (2001) 30, pp. 527-562.

[10] J. S. B. Mitchell and C. H. Papadimitriou, The Weighted Region Problem: Finding Shortest Paths Through a Weighted Planar Subdivision, *Journal of the ACM*, (1991) 38, pp. 18-73.

[11] J. O'Rourke, *Computational Geometry in C*, Cambridge University Press, 1998.

[12] C. H. Papadimitriou, An Algorithm for Shortest Path Motion in Three Dimensions, *Information Processing Letters*, (1985) 20, pp. 259-263.

[13] L. T. Schwartz and Micha Sharir, *Algorithmic Motion Planning in Robotics*, MIT Press, Cambridge, MA, USA, 1991.

[14] H. Wang and P. K. Agarwal, Approximation algorithms for curvature constrained shortest paths, *Proceedings 7th ACM-SIAM Symposium. Discrete Algorithms*, 1996, pp. 409-418.

[15] M.A.Weiss. *Data Structures and Problem Solving Using C++*, Addison-Wesley, Reading, MA, USA, 2000, p. 489-529.

VITA

Graduate College
University of Nevada, Las Vegas

Michael Allan Sherwood

Home Address:
75 Sea Holly Way
Henderson, Nevada 89074

Degrees:
Associate in Applied Science, Electronic Systems Technology, 1989
Community College of the Air Force

Bachelor of Science, Computer Science, 1998
University of Nevada, Las Vegas

Thesis Title: Trajectory Planning in the Presence of Risk Regions

Thesis Examination Committee:
Chairperson, Dr. Laxmi P. Gewali, Ph. D.
Committee Member, Dr. Thomas A. Nartker, Ph. D.
Committee Member, Dr. Wolfgang W. Bein, Ph. D.
Graduate Faculty Representative, Dr. Diane Pyper Smith, Ph. D.