PATH PLANNING FOR SENSING MULTIPLE TARGETS FROM AN AIRCRAFT

by

Jason K. Howlett

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Mechanical Engineering

Brigham Young University

April 2003

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **APR 2003**	2. REPORT TYPE	3. DATES COVERED **00-00-2003 to 00-00-2003**

4. TITLE AND SUBTITLE **Path Planning for Sensing Multiple Targets from an Aircraft**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Brigham Young University,Department of Mechanical Engineering,Provo,UT,84602**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**137**	

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Jason K. Howlett

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Dr. Tim W. McLain, Chair
Date	Dr. Craig C. Smith
Date	Dr. Randal W. Beard
Date	Dr. Michael A. Goodrich

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Jason K. Howlett in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date	Dr. Tim W. McLain Chair, Graduate Committee

Accepted for the Department

Brent L. Adams
Graduate Coordinator

Accepted for the College

Douglas M. Chabries
Dean, College of Engineering and Technology

ABSTRACT

PATH PLANNING FOR SENSING MULTIPLE TARGETS FROM AN AIRCRAFT

Jason K. Howlett

Department of Mechanical Engineering

Master of Science

To generate an assignment of tasks that best utilizes a team's resources, it is necessary to know the costs incurred by a team member for doing those tasks. In a cooperative search and destroy scenario, tasks generally require that the vehicle's sensor pass over specific known target points, which, to produce the associated costs, requires calculating the path the vehicle will take to sense the various targets. When the targets are far apart, the path-planning problem is trivial. For targets that are closely spaced, however, the problem is much more difficult, and thus is needed the ability to plan paths that sense multiple, closely-spaced targets.

Traditional path-planning methods are not well suited for generating paths that sense multiple, closely-spaced targets. Traditional methods focus on connecting some starting point and ending point with a feasible, minimum length path segment. Because an end point must be specified, these methods require too much information about how the path should accomplish its objectives, and hence the complexity of the associated problem is too great for real-time path-planning applications.

This thesis introduces the discrete-step path tree, and several methods for finding paths from the tree that accomplish the desired objectives, as solutions to the multiple-target sensing problem. Two of these methods use potential fields to guide the movement of the vehicle through the path tree. However, these methods are problematic and do not produce very good paths. Augmenting the potential-field methods by randomly branching to different parts of the path tree improves the path-length performance, but still not to completely satisfactory levels. The final two methods are based on the Learning Real-Time A* heuristic search, and provide the best running time and path-length performance. The Non-Improving Learning Real-Time A* (NILRTA*) tree-search algorithm provides the ability to plan near-optimal paths for sensing multiple, closely-spaced targets in sufficiently short periods of time. The NILRTA* algorithm utilizes the full sensing capabilities of the vehicle, and in fact, learns how the vehicle should move so as to accomplish its objectives. Furthermore, the flexibility of the NILRTA* path-planning algorithm allows it to be applied to a variety of path-planning problems and vehicle/sensor configurations.

This thesis shows that finding the optimal solution to the path-planning problem is difficult, and that some path-length performance must be sacrificed to allow the generation of good paths in a sufficiently short amount of time. In general the following maxim holds: if a shorter path is needed, then the time must be spent to find it.

ACKNOWLEDGMENTS

I express my gratitude to all who have contributed to the success of this work. First and foremost I thank my sweet wife, Kristen, whose enduring patience, support, and love has encouraged and sustained. I thank the members of my committee, Dr. Tim McLain, Dr. Randy Beard, Dr. Michael Goodrich, and Dr. Craig Smith, for their guidance, help, new ideas, and encouragement. I am grateful to my parents, Kevin and Sandra Howlett, for their support and encouragement throughout this work and throughout my college career. I also express gratitude to the AFOSR for the research grant that made this work possible. To all who have contributed to my success in this endeavor, I thank you.

Contents

Acknowledgments **vii**

List of Figures **x**

1 Introduction **1**

1.1 LOCAAS Scenario 3

1.2 Previous Work 6

1.2.1 Network Flow Assignment 6

1.2.2 Path-Planning Algorithm 7

1.2.3 AFRL Simulation Results 8

1.3 Complexity 9

1.4 Problem Statement 11

1.5 Requirements 12

1.6 Objective 12

1.7 Related Work 13

1.8 Contributions 16

1.9 Path Classes 17

1.10 Outline 17

2 Dubins-Based Path-Planning Methods **19**

2.1 Constrained-End Path Class 19

2.2 Optimization Based 21

2.2.1 Global Search 22

2.2.2 Bracketing Search 23

2.2.3 Reduced Variable Optimization 24

2.2.4 Optimization Routines 26
2.3 Linkage Analogy 27
2.4 Analytical Solutions 31
2.5 Summary 32

3 Discrete-Step Path-Planning Methods 33

3.1 Preliminaries 33
3.1.1 Notation 34
3.1.2 Dimensionless Parameters 34
3.1.3 Sensor Detection 35
3.1.4 A Special Case 36
3.2 Discrete Step Path Class 36
3.2.1 Path Tree 39
3.3 Potential-Field Based Path-Planning Methods 42
3.3.1 Single-Source Potential-Field Algorithm 43
3.3.2 Multi-Source Potential-Field Algorithm 44
3.4 Random Branching 47
3.5 LRTA* Tree Search 48
3.5.1 Initial Heuristic 51
3.5.2 Complexity 53
3.5.3 Algorithm Details 55
3.5.4 Terminating Conditions 57
3.5.5 Implementation Concerns 58
3.5.6 Flexibility 59
3.6 Summary 59

4 Testing and Comparison 63

4.1 General Testing and Analysis Procedures 64
4.1.1 General Testing 64
4.1.2 Statistical Analysis 65
4.2 LRTA* Tree Search Validation 68

4.2.1 Step Size 68

4.2.2 Running Time vs. Tree Depth 73

4.2.3 NILRTA* Running Limit 75

4.3 Algorithm Comparisons 76

4.3.1 More Than Three Targets 79

4.4 Performance on Different Computing Platforms 83

4.5 Test Conclusions 84

5 Conclusions **89**

5.1 Summary of Results 89

5.2 Future Work 90

A Code Listing **93**

Bibliography **123**

List of Figures

1.1 LOCAAS specifications 3
1.2 Target state diagram. 4
1.3 Path examples for targets that are close together and far apart 8
1.4 Bad simulation example 10
1.5 Good simulation example 10
1.6 Schematic of the three-target inspection problem. 13
2.1 Good and bad Dubins path strings 20
2.2 Good and bad constrained-end paths 21
2.3 Optimize path using final headings 22
2.4 Example bracketing search 24
2.5 Example bracketing boundary search 25
2.6 Good and bad path-length functions 26
2.7 General four-bar turn-straight-turn linkage 28
2.8 Four-bar turn-straight-turn linkage with sensor offset 30
2.9 Path length surfaces 32
3.1 Notation and geometry 34
3.2 Sensor geometry 35
3.3 Discrete-step path special case 37
3.4 Relative end point locations of the turn and straight primitives are close together 38
3.5 Ability of the discrete-step paths to track a nominal path. 39
3.6 Ability of the discrete-step paths to track a nominal path. 40
3.7 Discrete-step paths learn how to best track the nominal path 40
3.8 Path tree 41

3.9 Path tree geometry 42
3.10 LRTA* tree structure 43
3.11 Sample path from the single-source potential field algorithm 45
3.12 Weighting function for potential-field function 46
3.13 Sample path from the multi-source potential-field algorithm 47
3.14 Sample path from the single-source potential-field with branching algorithm 49
3.15 Sample path from the multi-source potential-field with branching algorithm 50
3.16 Sample path from the LRTA* algorithm 52
3.17 Sample path from the LRTA* algorithm 53
3.18 Two-step look-back LRTA* tree structure 54
3.19 Sample path from the LRTA* algorithm 58
3.20 Sample path from the NILRTA* algorithm 59
4.1 Histogram of a bimodal distribution 66
4.2 Short and long path examples 66
4.3 Bimodal probability density function with 99% confidence interval. . 67
4.4 LRTA* path comparison to brute-force global search 69
4.5 Path comparison using full sensor capabilities 69
4.6 LRTA* and NILRTA* distance comparison 70
4.7 Extrapolation of average path lengths for step sizes less than 0.177 radians 72
4.8 LRTA* running time comparison 74
4.9 LRTA* and NILRTA* running time versus tree depth on Jaguar . . . 75
4.10 Mean running time and path length vs. run limit for the NILRTA* algorithm 77
4.11 Comparison of LRTA* and NILRTA* with 100 iteration run limit . . 78
4.12 Running time comparison of all the algorithms 79
4.13 Distance comparison of all the algorithms 80
4.14 Running time comparison with different numbers of targets. 81

4.15 Distance comparison with different numbers of targets. 81
4.16 Example eight-target path 82
4.17 LRTA* distance comparison on Tomservo 83
4.18 LRTA* distance comparison on Robot 4 84
4.19 LRTA* time comparison on Tomservo 85
4.20 LRTA* time comparison on Robot 4 85
4.21 LRTA* time comparison on Robot 4 86
4.22 LRTA* time comparison on Robot 4 86

Chapter 1

Introduction

The success of any group or team depends largely on the group's ability to cooperate among its members. Successful teamwork requires a group to work together, utilizing the strengths of the individual members, and effectively assigning the required tasks. These principles also apply to a team of autonomous robots[1]: without cooperation, the robots cannot effectively and efficiently complete the tasks that they, as a team, are assigned to do. Thus arises the problem of getting a group of independent, autonomous robots to cooperatively work together to efficiently execute and complete their assigned duties.

Cooperative control of a team of autonomous robots has many important applications. Search and rescue operations would benefit from a cooperating team of robots assisting the search effort. A team of robots can replace a team of humans in dangerous tasks, such as hazardous waste containment and clean up. Cooperative teams have important roles in military operations including wide coverage surveillance or defense, resource transportation, and search and destroy missions. The last of these, cooperative search and destroy, as defined by the LOCAAS (**Lo**w **C**ost **A**utonomous **A**ttack **S**ystem) scenario described below, is the motivating problem for this thesis.

An essential component to the cooperative-control process for vehicle systems is that of path planning. To produce an effective assignment of tasks, the group must know the associated costs for doing those tasks. These costs may include the travel distance, fuel consumption, and exposure to enemy threats, all of which depend upon

[1]The term robot refers to any autonomous agent including land, sea, and air vehicles.

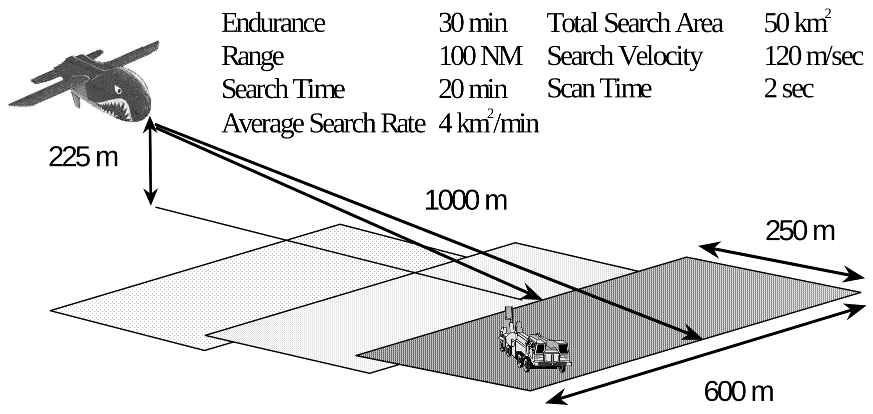
the path taken to accomplish the task. Traditional path-planning approaches require both starting and ending configurations for the path to be specified, which requires that the vehicle knows where it is going. In other words, an assignment must be made before a path can be generated. This paradoxical relationship leads to great difficulty in solving the cooperative-control problem: a path is needed to produce a cooperative assignment, but an assignment is needed to produce a path. Traditional path-planning methods can certainly be used, but the required iterative nature of the resulting cooperative-assignment process is computationally prohibitive.

Regardless of the path-planning method used, there are some fundamental requirements that the method should meet. First, the resulting path must allow the vehicle to effectively and efficiently carry out the desired task or set of tasks. This requires that the path is as short as practically possible, maintains the dynamic constraints of the vehicle, and provides accurate time-over-target (TOT) values. The last of these, time-over-target, is essential when developing cooperation among a team of agents where the completion of tasks is time dependent; in other words, an agent can not perform its assigned task until all other prerequisite tasks are completed. The second fundamental requirement is that the computation time be bounded and reasonably small. For a single vehicle assignment, the number of paths to compute will be few. When trying to produce a cooperative team assignment, however, many hundreds or thousands of path computations may be required. Certainly this speed requirement depends on the particular application: if the world is rapidly changing, a fast path-planning algorithm is needed. Conversely, in a slowing evolving world, speed is not as critical.

Since path planning is fundamental to the task-assignment process, this work focuses on path planning as motivated by the LOCAAS scenario described in Section 1.1. More specifically, this work develops a path-planning technique that utilizes the full capabilities of the vehicle and its sensor to sense a group of closely-spaced targets.

1.1 LOCAAS Scenario

The LOCAAS munition is a thirty-one inch long disposable airframe equipped with a micro-turbojet engine, a multi-mode warhead, and a LADAR[2] vision system. LOCAAS vehicles are deployed in groups of eight from either a manned jet fighter or an Unmanned Combat Air Vehicle (UCAV). Each has approximately thirty minutes of flying time before running out of fuel, at which point they must attack a previously identified target, or self-destruct. Figure 1.1 shows the geometry of the LADAR's viewing footprint, as well as some general specifications for the LOCAAS vehicle. As seen in the figure, the LADAR system's viewing footprint is a considerable distance in front of the LOCAAS vehicle, and this sensor standoff must be taken into account when planning a path to view a target or target area.



Endurance	30 min	Total Search Area	50 km^2
Range	100 NM	Search Velocity	120 m/sec
Search Time	20 min	Scan Time	2 sec
Average Search Rate	4 km^2/min		

Figure 1.1: Specifications and diagram of the LOCAAS vehicle's search capabilities and configuration.

This work uses one possible concept of operations for the LOCAAS scenario, which is described as follows. Targets in the LOCAAS scenario exist in one of six states: Not Detected, Detected but not Classified, Classified but not Attacked, Attacked but not Killed, Killed but not Verified, and Verified. In order for targets

[2]**LA**ser **D**etection **A**nd **R**anging is similar to radar, but uses light waves instead of radio waves.

to move through these states, the LOCAAS team must perform four different tasks. These tasks are Search, Classification, Attack, and Battle Damage Assessment (BDA). The success of each of these tasks is measured by a probabilistic indicator, which determines if the target moves to the next state, or if the previous task must be repeated. There are two important probabilities in this process: the probability of classification, P_c, and the probability of being killed, P_k. The states, along with the necessary tasks and probability levels, are illustrated in Figure 1.2. The team's goal, during their limited lifespan, is to move as many potential targets as possible through these states.

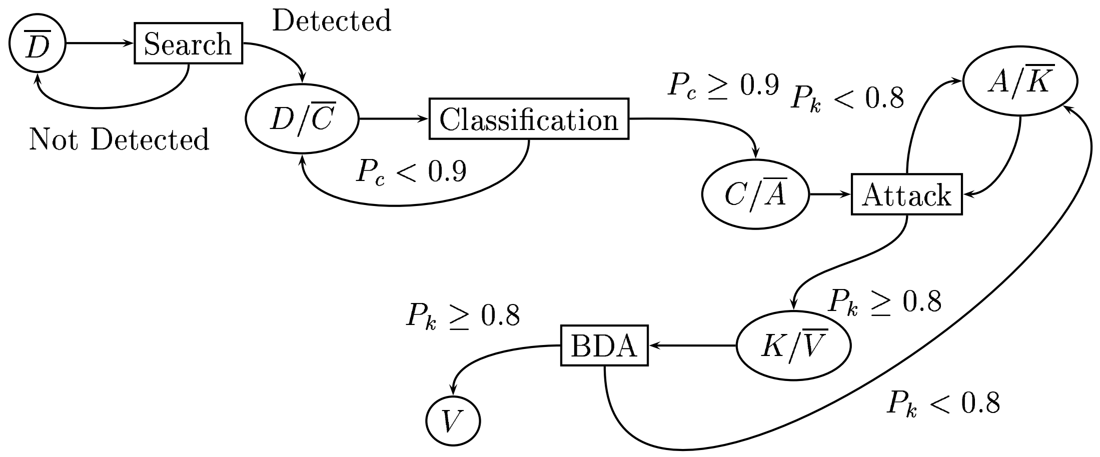


Figure 1.2: Target state diagram.

The LOCAAS vehicles are deployed to, and tasked to search in, an area where enemy targets are suspected to be, especially anti-aircraft weapons. After release from the delivery vehicle, the team cooperatively searches the designated area for enemy targets. Cooperative search itself is a difficult problem, and is addressed to some extent in [1].

When an object is detected, the team gets their first look at a potential target. An initial classification is performed to determine if the object is a possible target, or if it is more likely a non-hostile entity, like a school bus. If this initial classification suggests the object may be a target, a second look at the object is needed to confirm the initial results. This second look, or classification, requires that the sensor footprint of one of the team members pass over the object from a direction that is different than the initial viewing direction, and that will provide as much useful information as possible. The information from these two views is combined to form a composite sketch of the object, which is compared against a library of known target profiles. The best matching profile indicates the target type (see [2]).

Since the classification requires a vehicle's sensor to pass over the target, the classifying vehicle must start its approach to the target at some standoff distance from the target. This means that the vehicle nearest the object may not be the best choice for performing the classification, and therefore, the team must decide which of its members is in the best position to perform the classification at the lowest cost. If the object is determined to indeed be a target, the vehicle performing the classification has the option of continuing to the target and attacking it. By doing so, however, the team loses a member, and therefore the team must consider if the target is worth attacking now, or if it should be set aside for later attack in hopes of finding a more valuable target in the interim.

The LOCAAS vehicle's multi-mode warhead provides flexibility in the types of targets that may be attacked and the deployment method used. Depending on its type, a target may need to be attacked by more than one vehicle to ensure it is destroyed. When planning the attack, the team must consider the capabilities of the vehicles, the requirements of the target, and the possibility that the target may not be destroyed by the initial attack, thus requiring the expenditure of additional resources. Typically, the team will delay attacking targets until the later part of their lifespan, unless the target is considered valuable enough to attack now.

After an attack, a BDA must be performed by a surviving team member to determine if the target was actually destroyed. Generally it is sufficient for a vehicle's sensor footprint to pass over the target from any direction and look for smoke, holes, and other indicators that the target has been damaged or destroyed. If the target remains, then another attack will be initiated, otherwise the target is marked as destroyed and dropped from the team's target list.

The LOCAAS scenario as described thus far assumes that all targets are stationary. If the targets are moving, the team must also incorporate tracking into their list of activities, which, needless to say, makes the problem much more difficult. The above presentation has provided only a glimpse of the difficulty of the LOCAAS scenario and the issues that must be addressed when developing a cooperative strategy for a LOCAAS team.

1.2 Previous Work

The LOCAAS scenario described above has been the focus of work performed at the Air Force Research Laboratory's Air Vehicles Directorate (AFRL-VA) during the past two years [2, 3, 4, 5, 6, 7, 8, 9]. An assignment process, based on network flow optimization, and a path planner, based on Dubins paths, have been developed and implemented in a Simulink simulation of the LOCAAS scenario. Results from the simulation show that both the assignment process and the path planner have certain limitations that can result in unfavorable, and sometimes completely unacceptable, behavior of the LOCAAS team.

1.2.1 Network Flow Assignment

In [8, 10] an assignment technique is presented based upon solving a network-flow optimization problem to allocate tasks. The path costs for each vehicle-task combination in their current states are the link values in the network. Whenever the state of a target changes (i.e. is found, is attacked, and so on), the task assignment is recomputed by optimizing the flow of resources through the network. This approach is advantageous because the solution is calculated very quickly. However, the network flow approach as currently constituted only produces a one-to-one assignment of vehicles to targets. In other words, the assignment considers only what the vehicles are doing now, and gives no concern to what the vehicles will be doing after they complete their current task. Therefore many vehicles may be assigned to do what one or two could do if the assignment process considered more than just the currently needed tasks. While the assignment is optimal at the moment it is made, the myopic view of future events means that optimality is not maintained, and in fact, the resulting behavior over time is inefficient and undesirable.

1.2.2 Path-Planning Algorithm

The path-planning portion of the AFRL simulation calculates the Dubins path between an initial position and heading to a known final position and heading (constant altitude is assumed in all cases). In [11], Dubins shows that the shortest, curvature-constrained path between two points with some initial and final headings will consist of three segments: either turn-straight-turn or turn-turn-turn, where turns are made at the maximum turning rate of the vehicle. Dubins paths are easy to compute, and, theoretically, a path always exists that meets the required conditions. The disadvantage of this approach is that ending conditions for the path must be specified. This means that if the final heading to a target does not matter, a heading must first be chosen that will produce the shortest Dubins path. Fortunately finding the final heading that produces the minimum length path for the single target attack and BDA cases is straightforward [5, 6]. Also, in general there are four possible Dubins paths between configurations, but only one of them has minimum length. Therefore, four paths must be computed to find the single optimal path.

While the above path-planning approach works well for a single known target, it does not perform well if a path is needed that visits a group of known closely-spaced targets. When specifying the ending configuration for a path segment, the initial conditions for the next segment are automatically being specified. These conditions, while good for the current path segment, may adversely influence the remainder of the path, and thus to develop a minimum-length multi-target path, the spatial coupling between the targets must be considered during the path-planning process. The effect of this spatial coupling is inversely proportional to the relative distances of the targets: the farther apart the targets are, the less significant the coupling. When targets are very close together the coupling is significant, and if ignored, the paths produced by

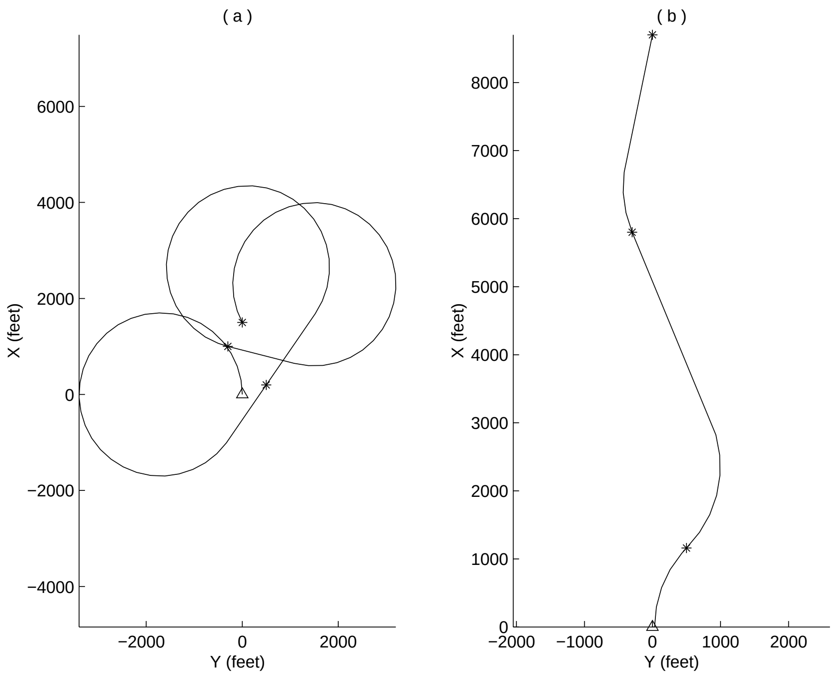


Figure 1.3: The spatial coupling of targets must be considered when planning paths for closely spaced targets, as in (a), but can be ignored when targets are far apart, as in (b).

traditional methods are often poor. The example path shown in Figure 1.3 (a) is longer than it needs be because the spatial coupling of the closely-spaced targets is ignored. In Figure 1.3 (b), however, the targets are far apart and the spatial coupling may be safely ignored by the path-planning process. For further discussion of spatial coupling, see Section 2.1.

In general, a group of targets is considered close together if they can be circumscribed by a circle with a radius of twice the vehicle's turning radius. Therefore, a path-planning method is needed that considers the location of all the targets, and then uses the full capability of the vehicle and its sensor to sense those targets.

1.2.3 AFRL Simulation Results

Figures 1.4 and 1.5 show two example runs from the AFRL Simulink simulation. In Figure 1.4, there are many loops and long excursions. A single vehicle is tasked with all the BDAs, but only one at a time. Therefore, the vehicle performs a

BDA and begins returning to where it left off its search. It is then tasked to another BDA and has to turn around. This happens for the third BDA as well. There are also significant gaps left in the search pattern by the vehicles that are destroyed in the attacks.

Figure 1.5 shows a simulation that was hand-tuned to produce the desired results for illustrative purposes. Notice that a single vehicle also performs all three BDA tasks, but this time it knows before hand that it will be doing so, and is thus able to plan its path accordingly. The results in this simulation are far more favorable than those in Figure 1.4. Also, the other vehicles fill in gaps left in the search pattern by destroyed vehicles. The challenge then, is to develop an assignment process that automates this capability. Multiple-target path planning is a necessary part of such a process, and thus the ability to plan efficient paths that sense multiple, closely-spaced targets is necessary.

1.3 Complexity

The coordination and cooperation of the team of autonomous LOCAAS vehicles is an extremely complex problem [12]. The task assignment process must account for target and vehicle states, mission timings, and unexpected events. Each of these are coupled in space and time, meaning that good local solutions may have adverse global effects in the future. In other words, doing task A now may prevent doing tasks B and C in the future, and so on.

To help appreciate the complexity of the LOCAAS problem, consider the results presented in [6]. For the scenario where there are four LOCAAS vehicles and three detected targets, each needing classification, attack, and BDA, there are 6,512,905 possible assignments. Repeatedly calculating, evaluating, and selecting from among all these assignments is not possible in the finite lifespan of the LOCAAS vehicle.

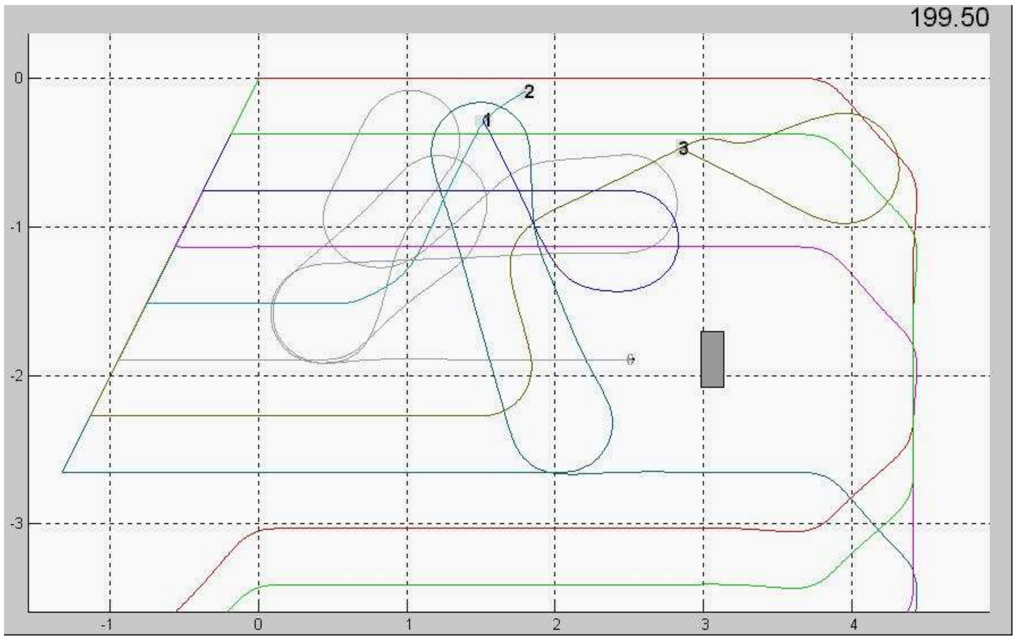


Figure 1.4: Example LOCAAS simulation with long, inefficient paths and gaps in the search pattern.

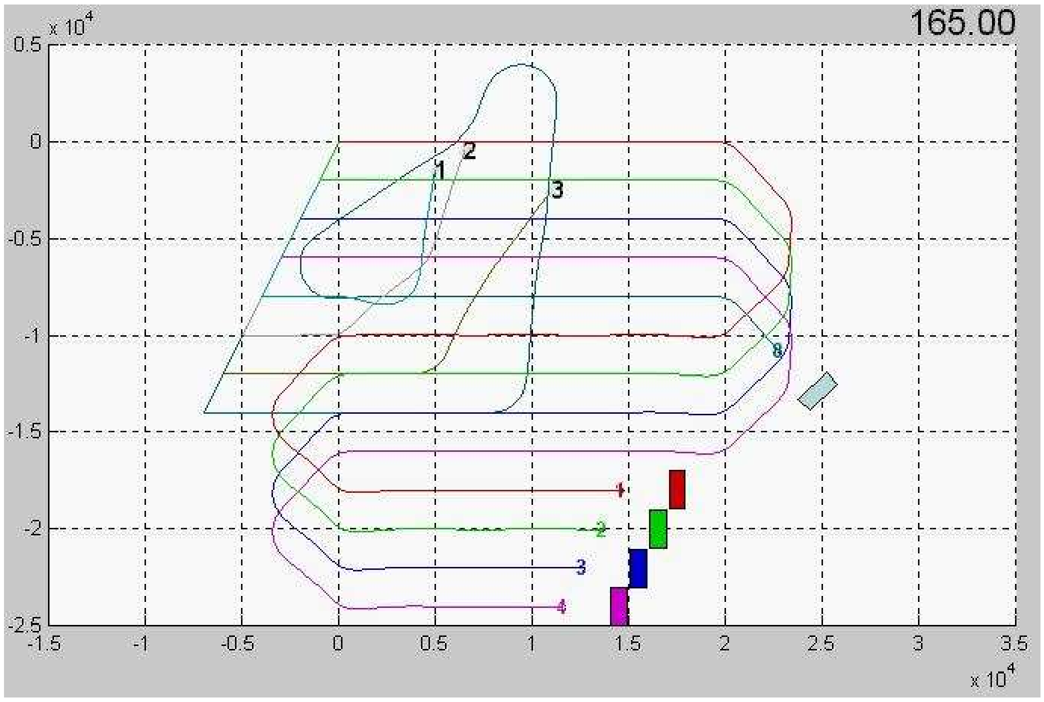


Figure 1.5: Hand-tuned simulation where a single vehicle efficiently performs multiple BDAs and no gaps are left in the search pattern.

1.4 Problem Statement

Due to the complexity of the full LOCAAS problem, some simplification is required to make this work feasible. Since path planning is fundamental to the task-assignment process, this work's focus is on path planning as motivated by the LOCAAS scenario. More specifically, this work develops the ability to plan near minimum length paths that sense a group of known closely-spaced targets. (The targets have been previously detected and thus their positions are known.)

The specifications of the vehicle in the path-planning problem follow those for the LOCAAS vehicle, namely that the vehicle has constant velocity and altitude, and a turning radius of 1700 feet. The primary difference is that the sensor views an area directly beneath the vehicle, and is gimbaled, meaning the sensor operates normally whether the vehicle is turning or going straight. Also, the number of targets in the path-planning problem is limited to three. Three targets provide an interesting problem of study without greatly increasing the complexity of the problem.

The reason for the above specifications is to place some limit on the complexity of the problem for the purposes of developing the path-planning methods. The path planner developed in this thesis, however, is extensible to any sensor-footprint geometry and any number of targets.

Results from both the AFRL simulation and the author's preliminary work suggest that finding the globally optimal solution for the LOCAAS problem is not feasible. The same applies to the path-planning problem and, therefore, we must be willing to accept solutions that are sub-optimal but satisfactory. The gain in finding the globally optimal path is not significant, i.e. there is a set of paths near the optimum for which the benefit of trying to further improve them is not worth the additional work. In fact, finding the global optimum is not feasible because the path-planning problem as stated is too complex. Therefore, the best we can hope to do is make relative comparisons of different path-planning methods, and perform statistical testing to get an indication of a particular method's performance. Mean path lengths and running times, along with a 99.7% confidence interval, are used as performance indicators for the tests. These indicators, while not absolute, provide statistical bounds on the performance of the path-planning algorithms.

1.5 Requirements

From the above discussion we can further refine the path-planning method requirements that were presented earlier. First, the planner must utilize the full capability of the vehicle's sensor, accounting for the fact that the vehicle must not fly directly over the target to sense it. Second, the path must be as short as possible while maintaining the dynamic constraints of the vehicle and sensing all the targets. Third, the method must be computationally fast, thus feasibly allowing the generation of many paths during the assignment process. And fourth, the path-planning method must require as little information as possible. The ideal method will simply take a list of targets and produce the best path that senses them all.

The computational speed requirement is difficult to quantify because the necessary speed will depend on how the path-planning method is integrated into the larger task assignment process. By meeting the fourth requirement, the path-planning method should actually aid in the task assignment process, which allows more time to be spent on path planning.

1.6 Objective

The objective of this thesis is to develop an algorithm that computes a near minimum length, curvature-constrained path that allows the aircraft's sensor footprint to pass over all the known targets in a closely-spaced target grouping (see Figure 1.6). The algorithm must be able to compute the desired path in a sufficiently short period of time. It is difficult to quantify this requirement since the algorithm's running time depends on the computing platform, the programming language, and the number of targets in the problem. Since it is difficult to find the optimal answer to this path-planning problem, there is a trade off that must be made between computational tractability and path length optimality. Hence, the resulting paths need only be reasonable approximations to the optimal path.

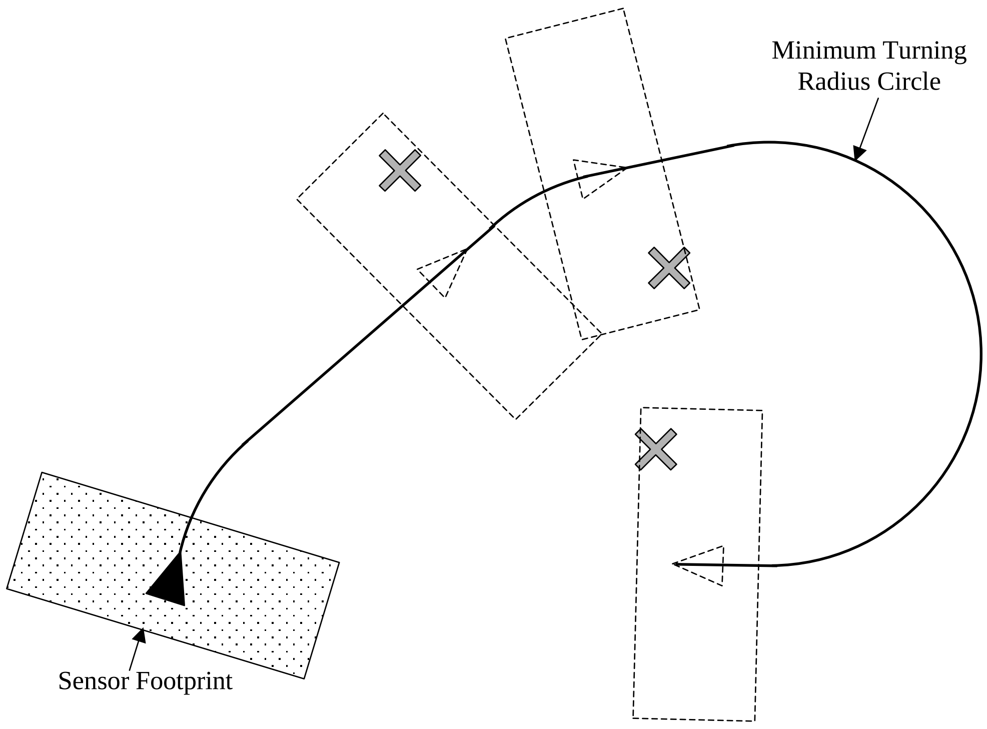


Figure 1.6: Schematic of the three-target inspection problem.

1.7 Related Work

Path planning for mobile robots is a broad area of research and there has been a lot of work performed in this area. This work draws ideas from several different types of path-planning approaches, which are briefly summarized below.

In [13], Yang and Kapila use the concept of Dubins paths and vector calculus to pose LOCAAS path planning as a parameter-optimization problem. Their algorithm maintains the vehicle's dynamic constraints, as well as considers various tactical constraints. The resulting path has minimum length, but, as with other path-planning approaches, the order in which the targets are to be visited must be specified prior to calculating the path.

A related problem to the LOCAAS scenario is the coordinated-rendezvous problem. In the rendezvous problem, several vehicles must plan paths such that they all arrive at their assigned targets at the same time while minimizing the vehicle's exposure to threats. One approach to this problem is to use a Voronoi graph to find an initial straight-line path, and then augment the straight-line path with turns to make smooth transitions from one segment to the next.

Judd developed a path-planning approach to the coordinated-rendezvous problem using polynomial or spline basis functions to add smooth transitions between the straight segments of the Voronoi graph [14]. These approaches both suffer from poor conditioning of the basis functions for optimization. While Judd's work applies to the rendezvous path-planning problem, his results provide insight into the value of using splines for path planning in general. More specifically, splines could be used to generate a flexible path, which could be optimized using the spline control points as variables. This approach is attractive because the splines provide flexibility and, since the splines are coupled together, the ability to globally alter the path with local control points. However, the computation required to optimize over the control points is large, and the optimum spline-based path is not guaranteed to be the global optimum.

An alternative to cubic splines would be NURBS (**N**on-**U**niform **R**ational **B**-**S**plines). NURBS introduce additional controlling parameters, which provide much more flexibility and local control over the path than do cubic splines. However, the complexity of the resulting optimization problem also increases, thus making NURBS infeasible for real-time path planning.

In [15], Anderson develops a differential solution for augmenting the Voronoi path with smooth transitions. This solution has the ability to keep the length of the augmented path the same as the length of the initial Voronoi path. Any straight-line path planner may use this approach to add turns and make the path flyable.

McLain and Beard present a potential-force based method for path planning in the rendezvous problem in [16]. The path is treated as a chain made of discrete segments which are repulsed by the threats. The segments are also repulsed by each other to smooth the path and make it flyable. If the path length needs to be increased, it is simple to add additional links to the chain. Unfortunately, this approach is too slow for real-time path planning. However, the ideas incorporated by this method have motivated some of the path-planning approaches presented in this thesis.

In [17] Frazzoli, Dahleh, and Feron develop a real-time randomized path-planning algorithm that maintains the dynamic constraints of the vehicle. The algorithm, however, considers the path to only one target. In the case of the LOCAAS scenario, this has already been solved. The random path-planning idea, however, may be adapted for use in planning a path that visits multiple targets.

An interesting area of path-planning research is that of randomized probabilistic search, also known as probability road mapping (PRM) [18, 19, 20, 21, 22, 23]. PRM randomly selects configurations from the configuration space, and plans local paths to those configurations. After randomly searching for some time, a road map has been constructed which is then searched for the shortest path to the goal. The underlying idea to PRM is that the the probability of finding the optimal path will converge to one as the time spent building the road map goes to infinity. While probability road mapping does not directly apply, random searching or branching ideas are useful for solving the LOCAAS path-planning problem.

Path-planning techniques for differentially-flat systems are presented in [24, 25]. These approaches require terminating conditions to be specified. A Dubins-path based path-planning method is presented in [26], but also requires that a terminating configuration be specified.

Naturally occurring potential fields have motivated the research of potential-field based path-planning methods [27, 28, 29]. These methods can produce good results, but are inherently problematic [30], and therefore not entirely successful in solving the path-planning problem. Nonetheless, potential-field approaches have been successfully applied to the LOCAAS path-planning problem.

The real-time heuristic search presented in [31] has been the basis for many path planning and other intelligent search algorithms [32, 33, 34]. There has been a great deal of effort expended in increasing the speed and efficiency of the real-time search [35, 36, 37]. See [34] for a comprehensive list of references concerning real-time search and the Learning Real-Time A* (LRTA*) algorithm.

In many problems it is difficult to find the optimal solution and therefore we must be satisfied with solutions that are sufficiently close to the optimal. The set of such solutions has been coined as the satisficing set. The meaning of 'sufficiently close' depends upon the problem and the metric used to determine the distance from the optimal. The idea of satisficing has been applied to single and multi-agent assignment problems as well as path-planning problems [38, 39]. Satisficing applies to this work only in as much as that the path-planning solutions need only approximate the optimal solution.

Path-planning techniques using Mixed Integer Linear Programming (MILP) have been developed in [40, 41, 42]. MILP methods are capable of producing collision-free paths between starting and ending states, but require specialized software for solving the MILP optimization problem.

1.8 Contributions

This thesis contributes to the broad base of current path-planning methodologies for mobile robots. Furthermore, this thesis shows that traditional path-planning methods are not well suited for planning paths for sensing multiple, closely-spaced targets from an aircraft. In general, these path-planning methods are not flexible in the types of problems they can handle, and require too much input as to how the path should accomplish its objectives. Good paths may be found using these methods, but the required computation time is prohibitive for any type of real-time path planning. This thesis overcomes these challenges by providing the ability to

- Produce satisfactory paths that sense multiple targets,
- Utilize the full sensing capability of the vehicle,
- Extend the path planner to a variety of path-planning problems, and
- Consider multiple-target assignments during the assignment process.

The path planner developed in this thesis utilizes a novel application of the Learning Real-Time A* search to generate satisfactory paths for sensing multiple targets. The path planner

- Requires only a list of targets as input,
- Provides the order and times in which the targets are visited,
- Extends to any sensor-footprint geometry, and
- Extends to additional constraints and goals for the path.

The path planner also provides motivation for future research to extend and apply the path planner to a variety of path-planning and assignment problems. The LOCAAS scenario is one such problem, but there are other problems in which the path planner may be used.

1.9 Path Classes

This thesis makes use of the notion of classes of paths. A class of paths are all those paths that are generated in a similar manner. An example path class is the straight-line path class, where all paths in the class are created by connecting a number of points with straight lines.

1.10 Outline

This thesis proceeds as follows. Chapter 2 presents various multiple-target path-planning techniques based on Dubins paths. These methods produce acceptable paths, but do not utilize the full sensor capability and require a significant amount of information be supplied to the path-planning algorithm. Chapter 3 introduces a class of paths that are constructed by assembling primitive turn and straight path segments to create a desirable path. Several path-planning algorithms are presented, including two potential-field algorithms, two potential-field algorithms augmented with random searching, and two algorithms based on the LRTA* search. Testing and validation of these algorithms follows in Chapter 4. Chapter 5 presents conclusions and directions for future research. A complete listing of the C++ code implementing the LRTA* algorithm is included in Appendix A.

Chapter 2

Dubins-Based Path-Planning Methods

This chapter introduces the constrained-end class of paths, which are based on assembling Dubins paths between the known targets. Several methods are presented that use optimization techniques to find the minimum-length path in this class. Because of discontinuities in the path-length function, these optimization approaches are unsuccessful at always finding the shortest path. Following these methods, a novel idea is presented that reparameterizes the Dubins path in hopes of creating a smoother path-length function that is more favorable toward optimization. This reparameterization successfully utilizes the vehicle's sensor capabilities while retaining the simplicity of the constrained-end path class. These path-planning methods are useful for generating paths through a series of multiple targets, such as in planning a multiple-target attack run. These methods are also useful when the targets are located far apart and require no special planning to sense them all.

2.1 Constrained-End Path Class

The constrained-end class of paths are based on creating Dubins path segments from an initial configuration to some target configuration, and then stringing these segments together to produce a flyable path that visits multiple targets. Generating a path from this class requires that the ending configurations for each segment be specified. Since the end configuration of a segment is the start of the next segment, the individual pieces of the path are closely coupled together. The top path in Figure 2.1 shows a path where the individual segments between starting and ending configurations have minimum length, but the path as a whole is not optimal.

Compare this to the bottom path where segment one is not of minimum length, but because the ending configuration for the segment was carefully chosen, the overall path has minimum length. By relaxing the optimality of segment one and giving consideration to the effect that it has on the entire path, the solution is improved. In both cases segments two and three have minimum length given their individual initial configurations. This example clearly demonstrates the coupling between the segments in a path, and the importance of considering the effect on the overall path when generating those segments.

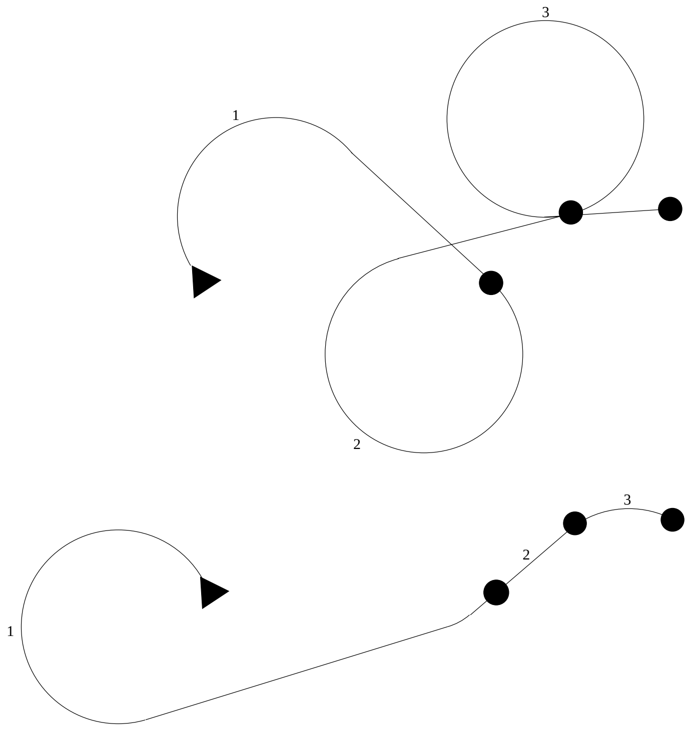


Figure 2.1: Two possible Dubins path strings for three targets. While the path segments between the targets in the top path have minimal length, the overall path does not. In the bottom path, segment one is not of minimal length, but the overall path is.

Because an end configuration must be specified, this class of paths cannot fully utilize the vehicle and its sensor without greatly increasing the complexity of the problem. Since the vehicle need not fly directly over the target to sense it, fully utilizing the vehicle's capability requires not only selecting a heading into the target, but also the sensor offset from the target. The sensor offset is the target's perpendicular distance from the vehicle's line of travel when the target enters the sensor footprint, as shown in Figure 2.2.

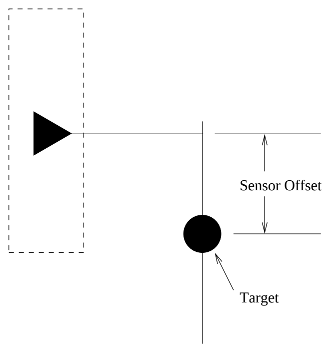


Figure 2.2: To fully utilize the vehicle's sensor, the sensor offset must be considered when selecting an ending configuration.

2.2 Optimization Based

By constraining the end configurations to be at the targets, the headings into the targets are the only remaining free variables. Specifying these final headings for all the end configurations directly determines the shape and length of the path, which suggests we could optimize over the final headings for each path segment to find the global optimum (see Figure 2.3). Fortunately the optimal heading into the n^{th} target is determined directly using the optimal attack heading method mentioned in Section 1.2.2. Thus the problem is to choose a set of headings, $\Psi = \{\,\psi_i : 0 \leq \psi_i \leq 2\pi\,\}\ \forall\ i = 1, 2, \ldots, n-1$, such that the resulting path is optimal. The following sections present several approaches to solving this optimization problem.

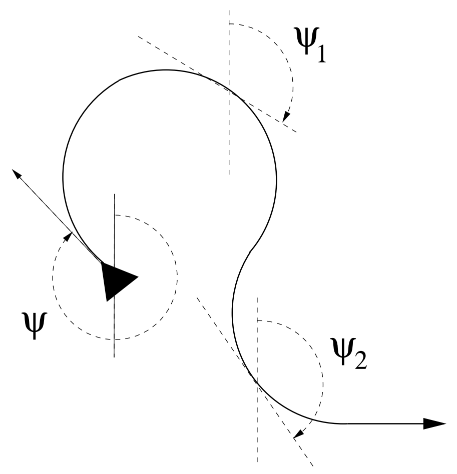


Figure 2.3: The final headings into targets provide a set of parameters to optimize for a global path.

2.2.1 Global Search

The simplest approach to minimizing the path length is a brute-force global search over all the possible values in Ψ. This requires calculating paths for all the possible combinations of the variables ψ_i. To make the problem tractable, we select p discrete values for ψ_i, where the value of p determines the resolution of the search. For any value of p and a set of n targets to be visited in some order, there will be p^{n-1} possible paths to evaluate. Obviously this number becomes very large as p or n increases. Also, for n targets, there are $n!$ possible orderings in which the targets are visited that must be evaluated to determine the ordering that gives the minimum-length path. If we are also finding the optimal paths for all of the m vehicles in our group so an assignment may be made, then we would need to do the above m times. The total number of paths, T, to be computed then is $T = mn!\,p^{n-1}$.

As an example, we will assume that $p = 360$, giving us a one-degree resolution, that there are $n = 3$ targets, and that there are $m = 8$ vehicles in our group. This gives us 129,600 path lengths to compute for each target ordering. For a single vehicle to find the optimal target ordering requires 777,600 path computations. Finding paths for all the vehicles in our group requires 6,220,800 paths be computed. Assuming that a single path takes .001 seconds to compute, it would require approximately 104 minutes to complete the above exercise. Obviously the global search does not meet the requirement of being computationally efficient. There are, fortunately, some ways to speed up the search.

2.2.2 Bracketing Search

One method of speeding up the global search is to perform a coarse global search, select the minimum path length as the center of a heading interval, then perform a more refined search in this interval, as shown in Figure 2.4. We call this the bracketing search. In each search iteration, the interval is recentered at the current minimum, the interval size decreased, and the step size reduced. The new interval is searched using the reduced step size, and the process repeated until the search converges to the minimum. While this bracketing search converges much faster than the global search, it will get stuck in local minima if the global minimum ever lies outside of the current interval, as is also shown in Figure 2.4. If the path-length function for the given problem is reasonably smooth then the algorithm will converge to the global minimum. However, path-length functions may be very discontinuous, which makes finding the global optimum difficult. Some examples of path-length functions are provided in Section 2.2.3.

A variation of the bracketing search is to decrease the interval size by one-half at each step and then only evaluate the path length at the lower and upper bounds of the interval. The minimum of these three points becomes the new starting point, and the process continues until the change in the objective function is within some tolerance. We call this method the bracketing boundary search, and an example is provided in Figure 2.5. The advantage with this method is that the number of path length evaluations may be fewer than for the bracketing search. Another approach is to fit a parabola to the three points with the minimum of the parabola being the next guess. In some cases this approach can converge slowly, with evaluation points actually moving away from the minimum before slowly converging.

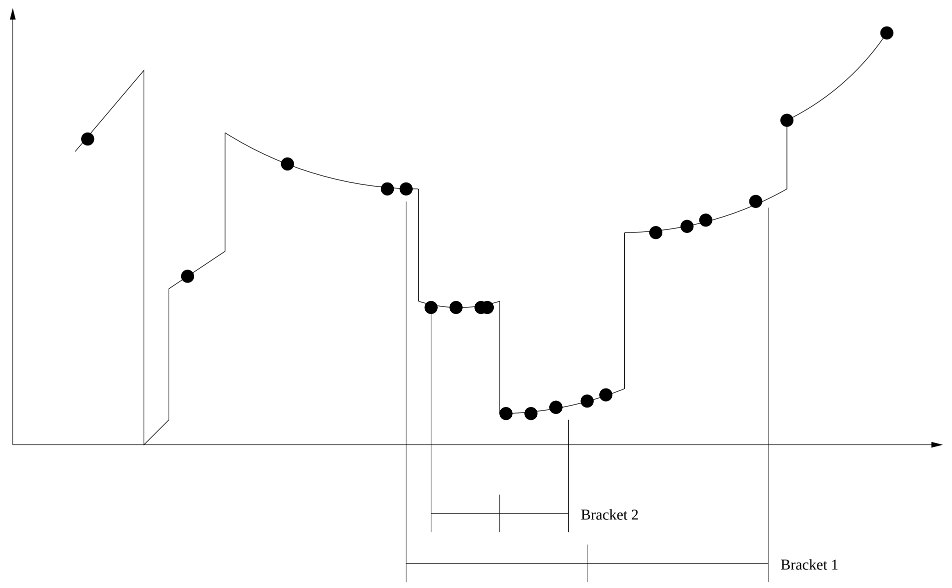


Figure 2.4: Example of bracketing-search minimization where the dots indicate the path-length evaluations made by the search. The entire interval is searched with a large step size and the minimum value from this search is chosen as the center of the next interval. The interval size and step size are decreased and the process repeated until the search converges to the minimum. The global optimum is missed if it ever lies outside of the current interval.

2.2.3 Reduced Variable Optimization

The most significant way to speed up the global search is to reduce the number of free variables. Since the individual segments in a multiple-target path are strongly coupled to one another, it is conceivable that a satisfactory path may be found using only one or two heading variables. Headings into subsequent targets are calculated using the optimal attack heading method.

If we let the heading into the first target be the only free variable, the complexity of the problem is greatly reduced. The drawback is that the effect of the first heading only extends down the path for two or three segments, meaning this method is only valid for three or maybe four targets. Also, the resulting path has no guarantee of having minimum length. Using the same problem specifications given in Section 2.2.1, the best target orderings and path length for all the vehicles are found with only 17,280 path calculations. However, this requires approximately

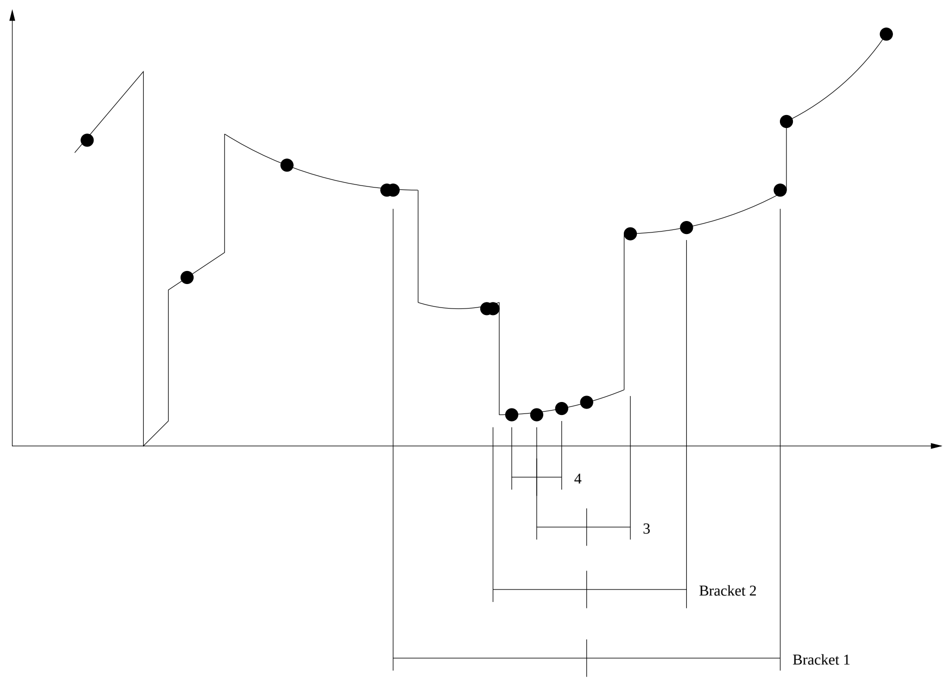
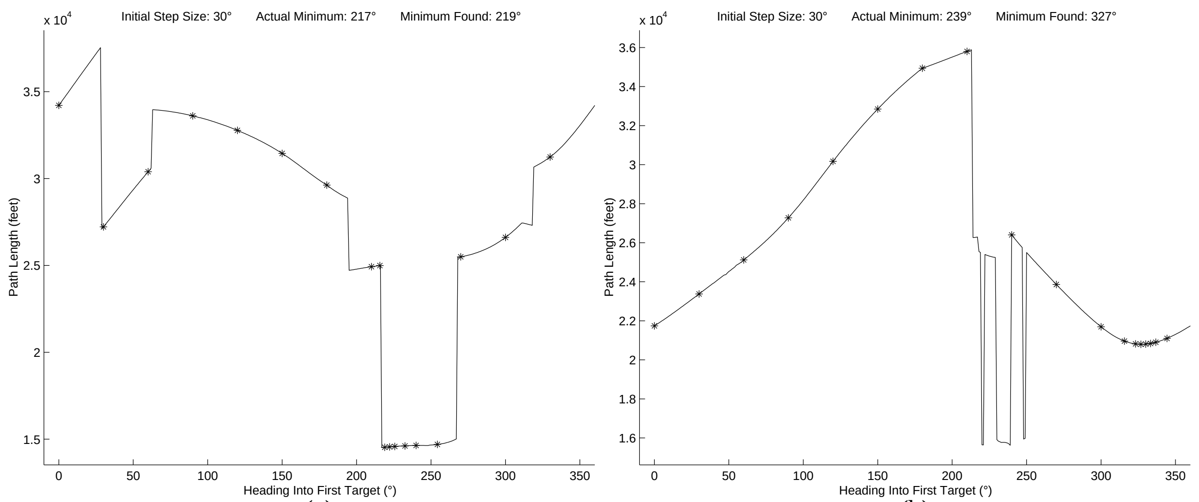


Figure 2.5: Example of bracketing-boundary search minimization. The entire interval is searched with a large step size and the minimum value from this search is chosen as the center of the next interval. The interval size is decreased by one-half and the path length evaluated at the lower and upper bounds of the interval. The minimum of these three points becomes the center of the new interval. This search can also miss the global optimum.

eighteen seconds to compute, which is still too long. We may again speed up the path-planning process by applying the bracketing search to the single-variable problem. Figure 2.6 shows path-length functions for two different three-target problems and the evaluation points made by the bracketing search. For most problems, the global minimum is found as seen in Figure 2.6 (a). However, there are cases such as in Figure 2.6 (b) where the search skips over the global minimum and settles into a non-optimal local minimum.

The bracketing search with one free variable produces good results for most problems, but the global optimum is not always found. The ability to find the global minimum is affected by the step size of the initial search, and the reduction rate of the search interval in each iteration. Smaller initial step sizes increase the chance of finding the global minimum, but with greater computational expense. A modification



(a) (b)

Figure 2.6: Path length vs. heading into the first target for two different three target problems. For most problems, as in (a) the global minimum is found. For some problems, however, the global minimum is skipped over as in (b).

to this method that may help increase performance is to use pseudo-derivative data calculated after each search step to help direct the search. Given the discontinuous nature of the objective functions, however, this approach may not produce favorable results either. Therefore, to increase the feasibility of using optimization in the path-planning process, a method is needed to smooth the path-length function, making the function more favorable toward optimization.

2.2.4 Optimization Routines

Along with the search methods described above, any optimization technique may be applied to the constrained-end path-planning process. The Matlab functions `fmincon` (constrained minimization) and `fminunc` (unconstrained minimization) handle multiple and single variable optimization problems, and both can optionally utilize the function derivative information if available. Unfortunately, these routines also have difficulty finding the global minimum of discontinuous path-length functions. In fact, the same basic problem exists for all gradient-based optimization approaches to finding the minimum-length constrained-end path: the path-length functions can be too discontinuous for these optimization approaches to work. While nonlinear optimization works most of the time, it is not guaranteed to work all the time. For any type of optimization approach to be feasible, the path-length functions need to be smoothed by some means so as to eliminate the narrow global optimums that are so easy to miss.

2.3 Linkage Analogy

The path-length function may be smoothed by parameterizing the path length differently, thus making optimization of constrained-end paths a feasible solution to the path-planning problem. The approaches presented above all used the final headings into the targets as the free variables in an optimization problem. By reparameterizing the path and generating a different set of free variables, the resulting path-length function may be more favorable toward optimization.

One reparameterization method is to treat each Dubins path segment as a four-bar mechanical linkage, with each link defining a portion of the turn-straight-turn path, as shown in Figure 2.7. The linkage is defined in the vehicle's body frame with the majority of the linkage constrained by the relative position of the target to the vehicle, and the geometry of the turn-straight-turn path. The linkage pin joints are located at P_0, which is the vehicle, P_1, which is the end of the first turn, P_c, which is the center of the second turn circle, and at the target, P_t. The second turn is defined by connecting the points P_3 and P_4 with an arc of the circle centered at P_c. Accordingly, the length of the line segments $\overline{P_2P_c}$ and $\overline{P_3P_c}$ is the turning radius, R_t. As indicated in the figure, the size of link three is determined by the required sensor standoff distance, s, where $s \geq 0$.

The two variables that control the linkage are the angle of link one, ϕ_1, and the length of the straight segment, l_2. Since the goal is to find the shortest path, l_2 is generally zero, and is only increased in length as needed to make the linkage complete. Looking at Figure 2.7, if l_2 were set to zero, the linkage would still be complete, but the path would approach the target from a different heading. By making l_2 always zero, we give up some control over the path, but we are left with ϕ_1 as the only remaining free variable, thus simplifying the associated optimization problem. It is also important to note that the linkage shown in the figure produces only one of two possible paths for connecting the vehicle and target. In this example the second turn is to the left, but the second turn may instead be to the right, which also results in a feasible path. Therefore, both paths must be computed to find the best path for a given problem. This choosing of paths, however, causes discontinuities in the path-length function since, for a given problem, a small change in ϕ_1 may cause a switch from turning left to turning right, and hence a significant increase or decrease in path length.

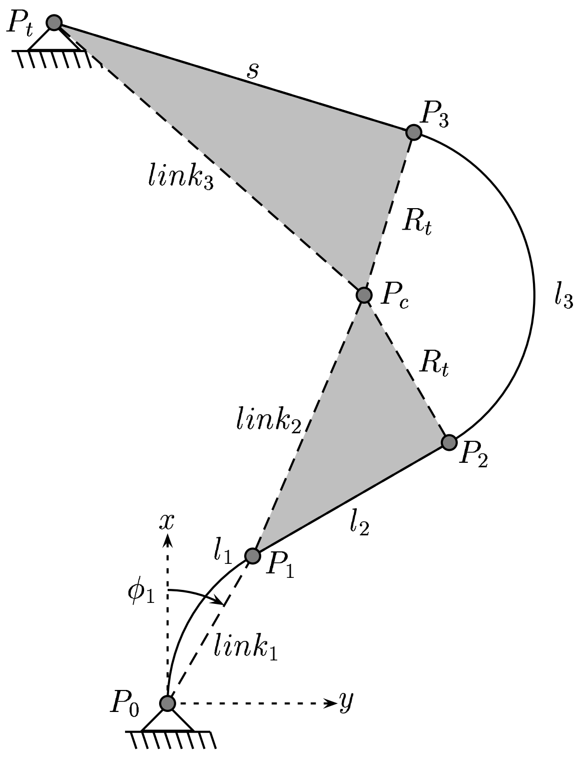


Figure 2.7: The general four-bar turn-straight-turn linkage is defined by the relative location of the target and vehicle, and the geometry of the turn-straight-turn path.

Because the links in a linkage can have an arbitrary geometry, a sensor offset, d, may be added to link three, as shown Figure 2.8. Doing so allows the path planner to use the vehicle's full sensor capability. This flexibility, however, comes at a cost since another free variable, d, is added to the linkage, thus increasing the complexity of the associated optimization problem. In general, however, the sensor offset may safely be set at plus or minus its maximum value [6] and still produce reasonable results. Also, since only the sensor needs to pass over the target, and not the vehicle, the path is not constrained to pass through the target. As shown in the figure, point P_4 is the point in the path by which the vehicle has sensed the target and can now proceed to its next objective.

As with many mechanical linkages, there are certain problematic configurations that can occur with this four-bar linkage. If the distance from the second turn center to the target is less than $d_{opt} = \sqrt{s^2 + (r - d)^2}$, the standoff and offset requirements cannot be met and the linkage breaks. This problem is overcome by increasing l_2 until the turn center is moved to a location that satisfies the d_{opt} condition, or by using the other turn direction if it is feasible. This 'fix', however, introduces discontinuity into the path-length function.

The primary advantage of the four-bar linkage parameterization is the simplicity of its geometry, and hence the speed of calculating the path. The geometry of the linkage is defined by the following set of equations, where $0 \leq \phi_1 \leq 2\pi$, $P_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix}$, $V(\theta) = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix}$, $V^{\perp}(\theta) = \begin{bmatrix} \sin(\theta) \\ -\cos(\theta) \end{bmatrix}$, $U(\theta) = \begin{bmatrix} \sin(\theta) \\ \cos(\theta) \end{bmatrix}$, and $U^{\perp}(\theta) = \begin{bmatrix} \cos(\theta) \\ -\sin(\theta) \end{bmatrix}$:

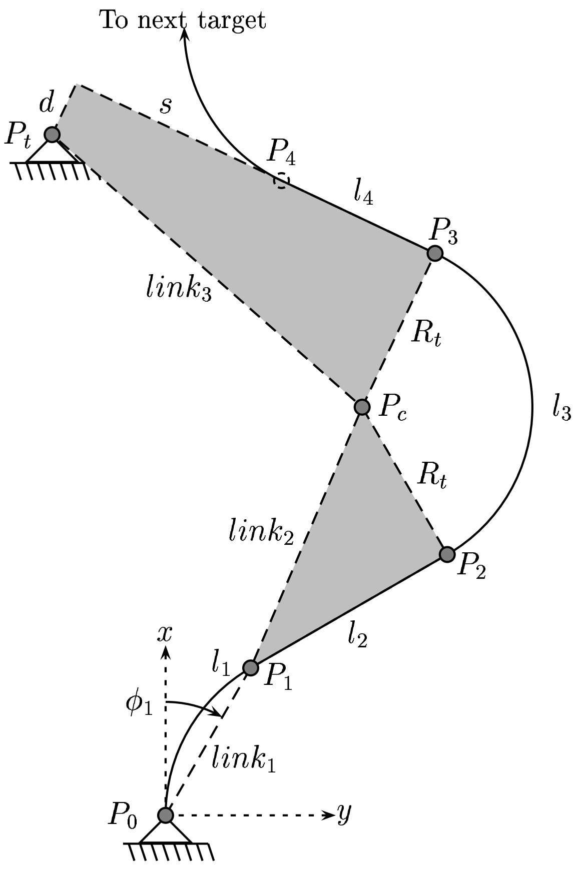


Figure 2.8: The four bar turn-straight-turn linkage with the sensor offset incorporated into link three. Since only the sensor needs to go to the target, the vehicle is free to proceed to its next objective at point P_4.

$$\begin{aligned}
D &= \begin{cases} 1 & \text{if second turn is to right} \\ -1 & \text{if second turn is to left} \end{cases} \\
P_1 &= \operatorname{signum}(\phi_1) r \begin{bmatrix} 1-\cos(2\phi_1) \\ \sin(2\phi_1) \end{bmatrix} \\
P_2 &= P_1 + l_2 U(2\phi_1) \\
P_c &= P_2 + D\, r U^{\perp}(2\phi_1) \\
\delta &= -D \sin^{-1} \frac{r-d}{\|P_t - P_c\|} + \tan^{-1} \frac{y_c - y_t}{x_c - x_t} \\
P_{ti} &= P_t + sV(\delta) \\
P_3 &= P_c + D\, r V^{\perp}(\delta) \\
P_4 &= P_{ti} + D\, d V^{\perp}(\delta)\,.
\end{aligned}$$

The total length of the path is calculated by summing the lengths of the turns and straights defined by the linkage:

$$
\begin{aligned}
l_1 &= \begin{cases} 2r\phi_1 & : \; 0 \le \phi_1 \le \pi \\ r(4\pi - 2\phi_1) & : \; \pi < \phi_1 \le 2\pi \end{cases} \\
l_3 &= r\left[\left[\frac{\pi}{2} + D\left(2\phi_1 - \tan^{-1}\frac{y_3 - y_c}{x_3 - x_c}\right)\right] \bmod 2\pi\right] \\
l_4 &= \|P_4 - p_3\| \\
l_{total} &= l_1 + l_2 + l_3 + l_4 \, .
\end{aligned}
$$

Testing of the linkage shows that for a *single* target, the path-length function is favorable to optimization, but this is obviously not very useful. When two segments are connected together, however, the combined discontinuities of the path segments produce a ragged path-length function, with the discontinuity becoming worse as the targets get closer together. Figure 2.9 shows the path-length functions for two different two-target scenarios, where ϕ_1 for the two segments are the free variables. In (a) the targets are far apart, which results in a fairly smooth surface that is favorable to optimization. The targets in (b) are close together, and, as can be seen, produces a very rugged surface. Unfortunately the linkage reparameterization failed to smooth the path-length function as hoped due to the combined discontinuity of switching turn directions at the second turn, and the lengthening of l_2 to make the linkage complete. This method, however, shows that a path segment may be defined by very simple geometry while achieving the ability to utilize the vehicle's sensing capabilities.

2.4 Analytical Solutions

An attempt was made to analytically solve for the minimum-length constrained-end path using Lagrange multipliers and constrained minimization. The problem quickly became large and complex, and it was apparent that there would not be one unique solution, i.e. there would be one solution for each possible turning direction. This would require solving multiple problems and then picking the shortest path.

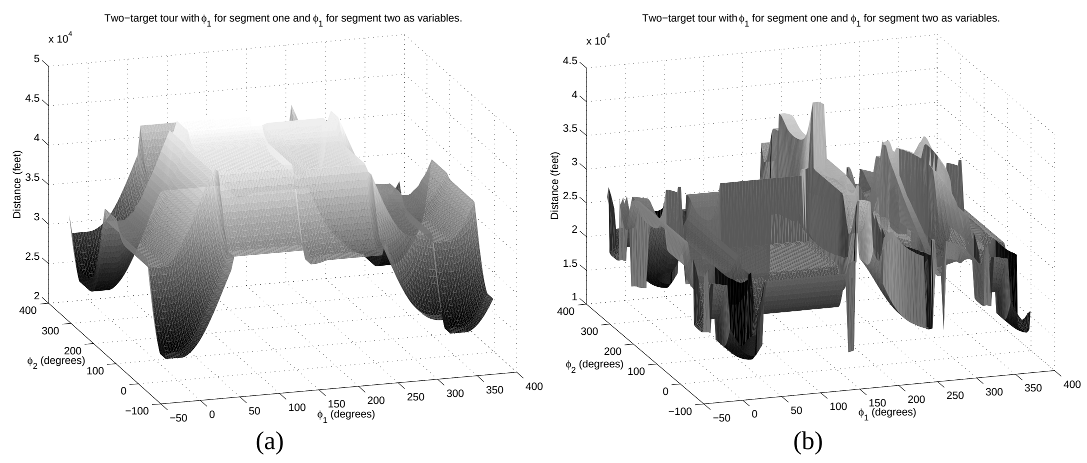


Figure 2.9: Surface plots of path length for two-target tours, varying ϕ_1 for both segments. The targets are far apart in (a) and close together in (b).

2.5 Summary

This chapter presented the constrained-end class of paths and several methods for finding the minimum-length multiple-target path from this class. Optimization is applied to the problem using the final headings into the targets as the free variables. The optimization problem is simplified by using only the first heading as the free variable, but this approach also suffers from discontinuous path lengths. An attempt was made to smooth the path-length function by reparameterizing the path based on a four-bar linkage. This simplified the calculation of the path and utilized the full sensing capability of the vehicle, but failed to smooth the path-length function. In general, the constrained-end class of paths are not well suited for solving the path-planning problem.

Chapter 3

Discrete-Step Path-Planning Methods

This chapter introduces the discrete-step class of paths which are based on building the path one discrete step at a time. This allows specific constraints and goals to be imposed on the path without directly specifying how the path should accomplish those things. Several methods are presented for determining the direction the path should move at each step. The first two methods are based on potential-field theory and treat the targets as attractive forces acting on the vehicle. The next two approaches augment the potential-field methods with random searching to refine and improve the path. The final method applies the Learning Real-Time A* search in a novel way to learn the shortest discrete-step path that senses all the targets. The search running time is determined by the terminating conditions for the search, and a modified LRTA* search is presented that terminates when the path length has not improved after some number of search iterations. The LRTA* method provides a foundation that is extensible to handle a variety of path-planning problems and sensor configurations.

3.1 Preliminaries

Before proceeding with the presentation of the discrete-step path class and the associated path-planning algorithms, we need to introduce the notation, the dimensionless parameters, the sensor detection algorithm, and the special case that can exist in the path-planning problem for the discrete-step path class.

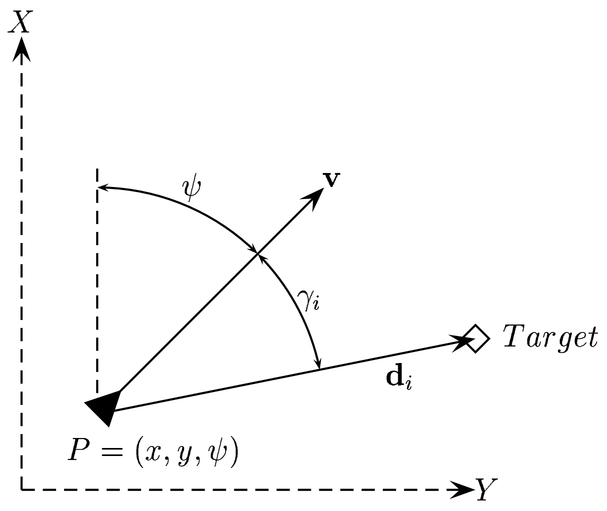


Figure 3.1: Notation and geometry for the discreet path planning problem.

3.1.1 Notation

The notation used throughout the remainder of this work is as follows (see Figure 3.1). The current configuration of the vehicle is given by the triplet $P = (x, y, \psi)$, where x and y represent the inertial position, and ψ is the heading of the vehicle measured from the North. The vector from the vehicle to the i^{th} target, is denoted as $\mathbf{d}_i$. The angle between the vehicle's heading vector, $\mathbf{v}$, and $\mathbf{d}_i$ is denoted by γ_i.

3.1.2 Dimensionless Parameters

To generalize the path-planning algorithms presented below, distance values are normalized by the turning radius to provide dimensionless ratios. Fundamental to the discrete-step paths is the size of the discrete path segments, denoted as dS. Normalizing by the turning radius gives

$$d\psi = \frac{dS}{R_t}$$

where $d\psi$, in radians, is the magnitude of the maximum change in the vehicle's heading at each step. The term step size is used interchangeably to refer to the absolute step

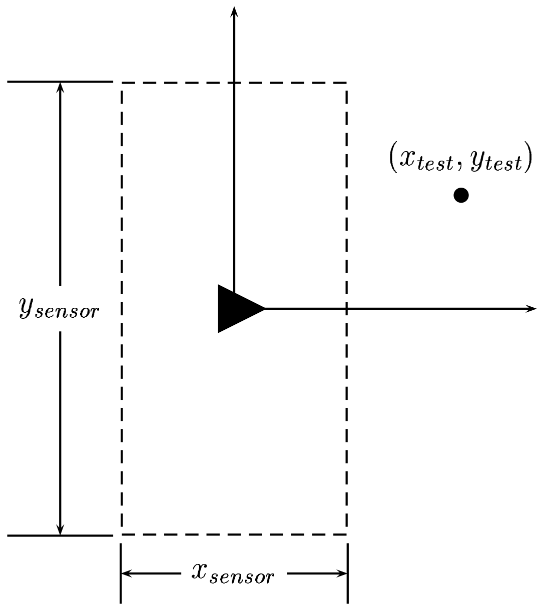


Figure 3.2: Sensor geometry, and the position of targets being tested, are relative to the vehicle.

size, dS, and the normalized step size, $d\psi$. Total path length, P, is normalized by

$$P_n = \frac{P}{R_t}$$

where P_n has units of turning radius, R_t.

3.1.3 Sensor Detection

Targets are sensed by the vehicle whenever they are inside the vehicle's sensor footprint. Determining whether a target is being sensed is independent of any path-planning algorithm, thus any sensor shape may be substituted into the algorithms presented below. For our purposes, the sensor footprint is rectangular and located directly beneath the vehicle as shown in Figure 3.2. The dimensions of the footprint follow the specifications of the LOCAAS vehicle with $x_{sensor} = 1.18R_t$ and $y_{sensor} = 0.48R_t$.

Determining if a target is inside the footprint is straightforward. We transform the target point into the vehicle's local coordinate frame by rotating $\mathbf{d}_i$ by ψ using the transformation

$$\begin{bmatrix} x_{test} \\ y_{test} \end{bmatrix} = \begin{bmatrix} sin(\psi) & cos(\psi) \\ -cos(\psi) & sin(\psi) \end{bmatrix} \mathbf{d}_i.$$

If $|x_{test}| \leq x_{sensor}$ and $|y_{test}| \leq y_{sensor}$, where x_{sensor} and y_{sensor} are the respective sensor dimensions, then the target is inside the sensor footprint and therefore detected.

3.1.4 A Special Case

Depending on the geometry of the sensor footprint, there may arise a special case that must be considered during the path-planning process. If the center of the desired turning circle is too close to the target, the sensor will never pass over the target because the turn radius is greater than half the width of the footprint. The dashed turn circle in Figure 3.3 illustrates this case. The smaller circle indicates the area that will not be sensed if the vehicle uses the dashed turn circle. Fortunately the solution is easy: simply turn the other direction until the desired turning center is sufficiently far from the target. The footprint is then able to pass over the target as the vehicle makes its turn. Notice that the target now lies outside of the region that can not be sensed by the solid turn circle. The decision to turn the other direction instead of flying straight for some distance is supported by the Dubins path idea. Of course, if the sensor is able to see all the way to the turning center, then this special case does not apply.

3.2 Discrete Step Path Class

To develop a successful path-planning algorithm, we need a class of paths that are not constrained by end points or headings. The paths in this class must be able to utilize the full capability of the vehicle's sensor to sense targets, while maintaining the dynamic constraints of the vehicle. In generating paths from the constrained-end class of paths presented in Chapter 2, the only goal that may be imposed on the path planner is to get from the initial configuration to the specified final configuration. Path planning with the discrete-step class of paths, however, is driven not by getting from point A to point B, but instead by meeting a set of specific goals, such as sensing

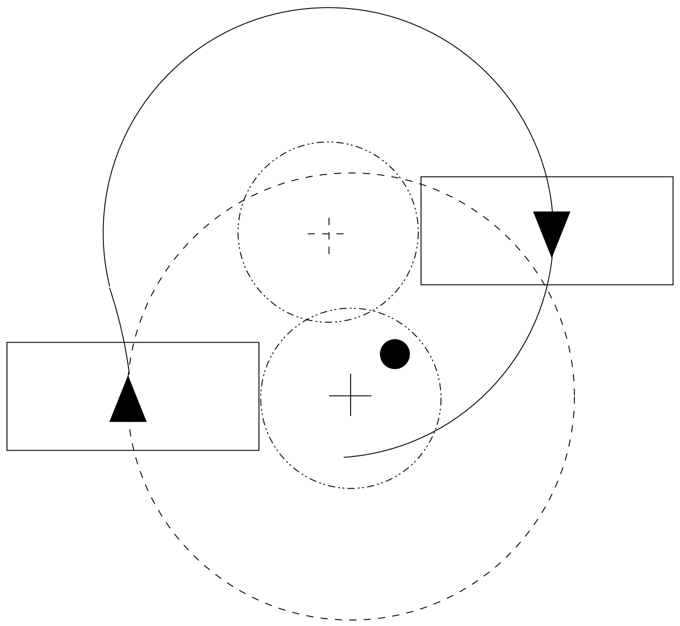


Figure 3.3: Special case where the center of the desired turning circle is too close to the target and the sensor can never pass over the target. The solution is to turn away from the target until the turning center is sufficiently far from the target.

targets. The path planner's only concern is what the path does at each intermediate step, not where the path ends. The discrete-step path class provides the flexibility to impose constraints on, and goals for, the path without needing to specify how the path should meet or accomplish those things.

Paths from the discrete-step class are generated one step at a time, with the path deciding at each step the direction it should go to accomplish its goal, while maintaining its constraints. Each step in the path is a primitive turn or straight segment of some specified step size, dS. Since the step size is constant for each primitive, the overall length of the path is simply the number of steps in the path times dS. In dimensionless terms, the number of steps in the path is given by

$$N = \frac{P_n}{d\psi}$$

where P_n is the normalized path length and $d\psi$ is the normalized step size.

The number of possible directions the path can go at each step is infinite, so, to make using this class of paths feasible, the choices must be limited to a reasonable

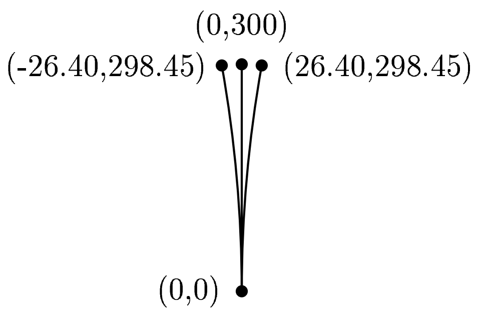


Figure 3.4: For small step sizes, the location of the points are relatively close together, as in this example where $dS = 300$ feet and the turn radius is 1700 feet.

number. Following the Dubins path idea of using turns at the maximum turning rate and straights, we limit the possible directions at each step to three: left turn, right turn, and straight. Using only these three directions is justified by the fact that for small step sizes, the spatial locations of the three points are relatively close, as shown in Figure 3.4, where the primitives are based on $dS = 300$ feet and a turn radius of 1700 feet.

The significant issue with the discrete-step path class is that the direction of the path changes only at the discrete nodes. Figure 3.5 illustrates this problem. The solid line in the figure represents the nominal path, which changes turning directions at $x = 0$. The discrete-step paths are unable to track changes in the nominal path that occur between the discrete points, thus introducing error into the discrete-step paths. In the example, the discrete-step paths with step sizes of .1, .2, .4, and .8 R_t are able to track the nominal path very well, while paths with step sizes of .3, .5, .6, and .7 R_t do not. It is noteworthy that the levels of discretization shown in this example are higher than that typically used in practice.

Changing the nominal path so that it switches directions at $x = -1$, as shown in Figure 3.6, produces totally different results. Here the discrete-step paths with step sizes of .1, .3, and .5 R_t track the nominal path the best. It is difficult to quantify how well a discrete-step path of some step size is able to track a given nominal path, as this ability varies with the nominal path and the step size. Fortunately, however,

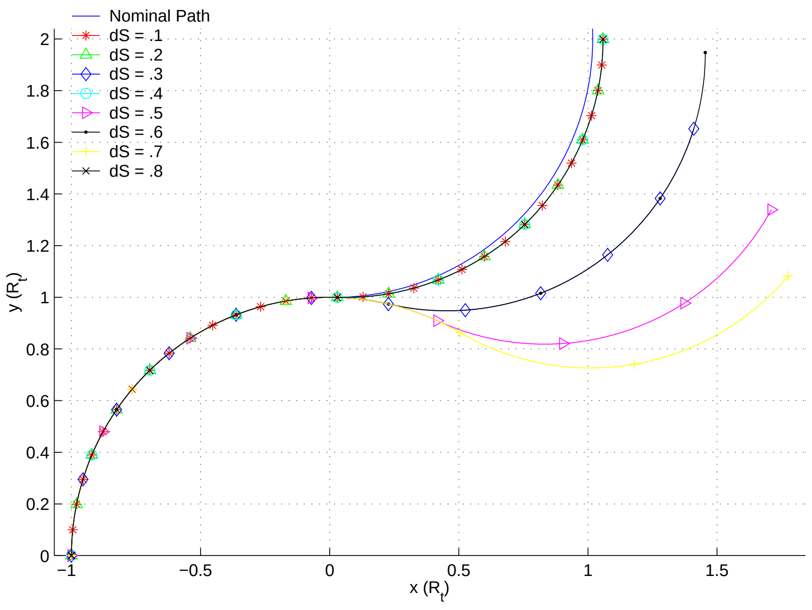


Figure 3.5: Error is introduced in the discrete-step paths when the nominal path changes direction between the points of the discrete-step path. Discrete-step paths with step sizes of .1, .2, .4, and .8 R_t are able to track the nominal path which changes direction at $x = 0$.

the advantage of the discrete-step path class is the ability to learn which path best approximates a given nominal path. Figure 3.7 again shows the nominal path that changes direction at $x = 0$, but in this example the discrete-step paths have learned how and when they should change directions to best match the nominal path. The resulting paths approximate the nominal path fairly well. The primary assumption, then, for the discrete-step path class with any step size, is that there exists a discrete-step path that is able to approximate any nominal path, and hence approximate the global minimum-length path.

3.2.1 Path Tree

Assembling the left turn, right turn, and straight primitives creates a tree of paths as shown in Figure 3.8. Every node in the tree has a parent and three children. The configuration of a child is determined by calculating the change in heading and position relative to the parent configuration as shown in Figure 3.9. The equations for calculating the child configurations, P_l, P_s, P_r, are provided below.

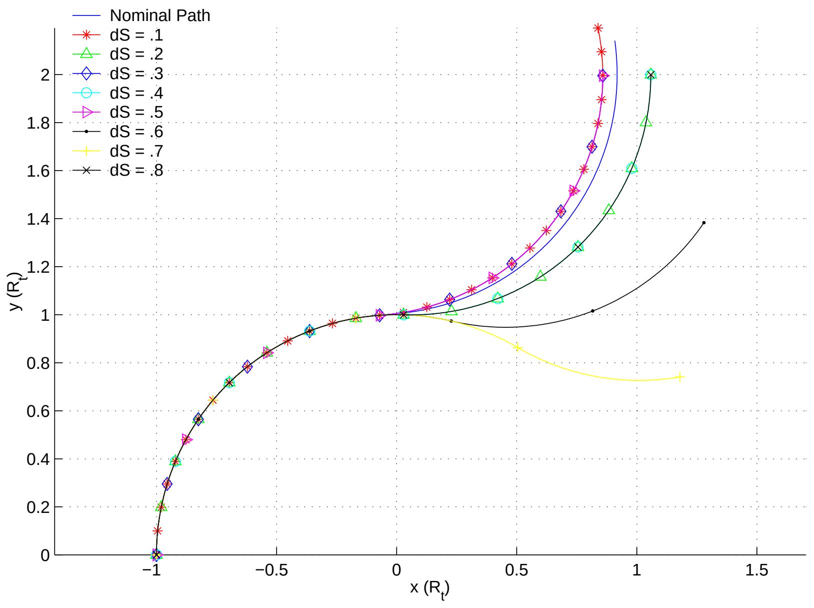


Figure 3.6: The nominal path now changes direction at $x = -1$. The discrete-step paths with step sizes of .1, .3, and .5 R_t best track the nominal path.

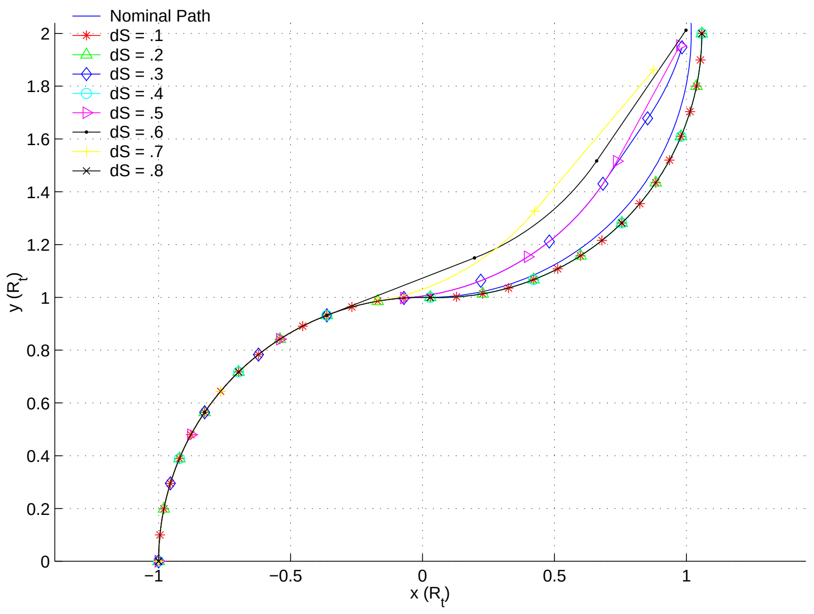


Figure 3.7: The discrete-step paths have learned how and when to change directions to best approximate the nominal path.

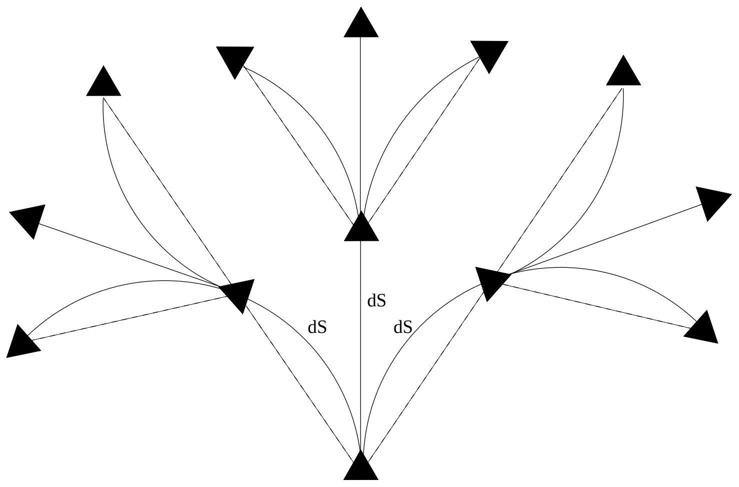


Figure 3.8: Primitive turn and straight path segments, all of equal length, dS, are assembled to form a tree of flyable paths. The tree is then searched for the best path that accomplishes the desired objectives. The choice of dS directly effects the number of nodes in the tree, and therefore the complexity of the tree search.

$$\begin{aligned}
d\psi &= \frac{dS}{R_t} \\
c &= \sqrt{2R_t^2(1 - cos(d\psi))} \\
P_l &= P_0 + \begin{bmatrix} c\cos(\psi_0 + 0.5d\psi) \\ c\sin(\psi_0 + 0.5d\psi) \\ d\psi \end{bmatrix} \\
P_s &= P_0 + \begin{bmatrix} dS\cos(\psi_0) \\ dS\sin(\psi_0) \\ 0 \end{bmatrix} \\
P_r &= P_0 + \begin{bmatrix} c\cos(\psi_0 - 0.5d\psi) \\ c\sin(\psi_0 - 0.5d\psi) \\ -d\psi \end{bmatrix}
\end{aligned}$$

From a mathematical standpoint, the tree is of the form shown in Figure 3.10. Each branch of the tree is terminated with either having sensed all the targets, or

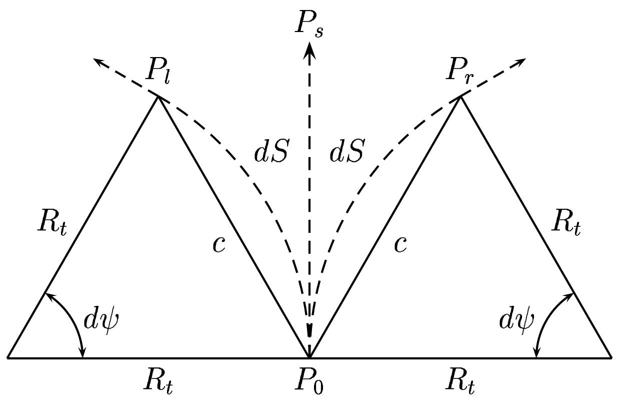


Figure 3.9: Geometry of the path tree

having become too long as compared to the current best path. The goal of the path-planning algorithms presented below is to find the shortest branch of the tree that accomplishes the desired objectives. In other words, the goal of the algorithm is to learn the best path that the vehicle should follow through the path tree such that the sensor passes over all the targets.

Having introduced the discrete-step path class and the necessary preliminary information, we are now ready to proceed with the development of the various path-planning algorithms. The potential-field path-planning methods are introduced first, followed by the potential-field with random-branching algorithms. Finally, the Learning Real-Time A* and Non-Improving Learning Real-Time A* tree-search algorithms are developed.

3.3 Potential-Field Based Path-Planning Methods

In nature, the paths of moving objects are influenced by external forces, or potentials. The deflection of electrons by charged plates to form a television image is one example. On a more macroscopic scale, the orbits of planets, comets, and other heavenly bodies are governed by the gravitational potentials of their surrounding neighbors. Similarly, the targets in the path-planning problem may be treated as potential sources to which the moving vehicle is attracted and/or repelled. By examining these forces, the vehicle is able to choose to which child node in the tree

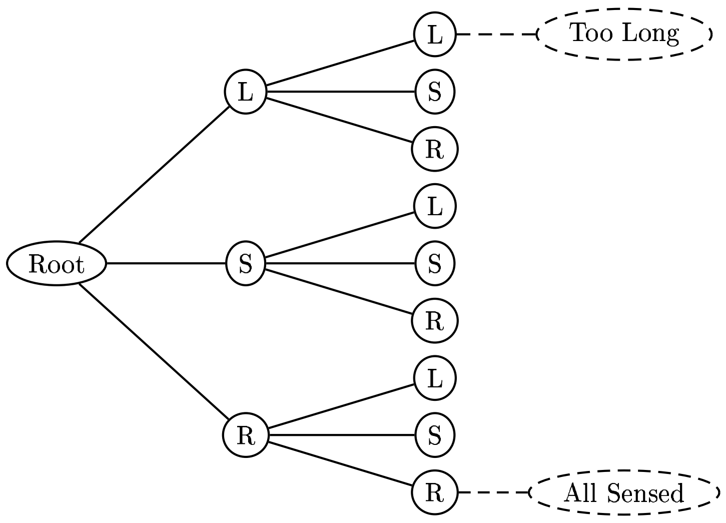


Figure 3.10: Tree structure of the LRTA* path planning algorithm.

it should move. The application of potential fields to path-planning problems is well explored [16, 27, 28, 29]. However, previous work has shown that proving potential-field methods behave as desired is difficult [30]. There is nearly always a pathological scenario where the resulting path is poor or unacceptable.

3.3.1 Single-Source Potential-Field Algorithm

In the single-source potential-field algorithm, the moving vehicle is acted upon by only one potential source at a time. In other words, the vehicle picks a target and then moves toward that target until the target has been sensed. The manner in which the targets are selected is arbitrary, but each selection scheme will produce different results. The motivation for the single-source algorithm is to develop a method that always provides a flyable path that senses the targets, so, to this end, the target nearest to the vehicle's current node in the path tree is selected as the source. The result is a greedy algorithm that is only concerned with getting to the nearest target, while giving no consideration to what happens after the target has been sensed. The primary advantage of this method, however, is that it is able to quickly compute a feasible path that accomplishes the desired objectives.

```
Algorithm 1: Single-Source Potential Field
Input: Initial configuration $P_0 = (x_0, y_0, \psi_0)$
Output: Path
SingleSource($P_0$)
(1)   while !$allSensed$
(2)      if !children
(3)         create children
(4)      $j$ = number of children
(5)      $P_t \leftarrow$ nearest target
(6)      $P \leftarrow$ child $k = \arg\min_j \|P_t - P_j\|$
(7)   return $P$
```

At each node in the tree the vehicle determines which target is closest, and then moves to the child node that is nearest to that target. Once the nearest target is sensed, the vehicle moves toward the next nearest, and so on. The selection of the next node must consider the special case presented in Section 3.1.4, and, if the desired turning center is too close to the target, move in the opposite direction. Since the vehicle always chooses the child node that is nearest to the target, the path will converge to the target. If the target is behind the vehicle, it is possible that the distance to the target may increase as the vehicle turns toward the target, but once the target is in front of the vehicle, the distance to the target is decreased at every step until the vehicle reaches the target. The resulting path has no guarantee of being optimal, but it is guaranteed to be flyable and to sense all the targets. The outline of the single-source potential-field algorithm is presented in Algorithm 1.

Figure 3.11 shows the path generated by the single-source potential-field algorithm for a test scenario with three targets. The same test scenario is used in path examples for the subsequent path-planning algorithms.

3.3.2 Multi-Source Potential-Field Algorithm

The single-source potential-field method described above may be modified to produce more desirable paths. The trade off, however, is a loss of guaranteed convergence, and, like the single-source algorithm, the resulting path has no guarantee of being optimal. The main change from the single-source approach is the use of a potential-field function for each target. The combination of these potential fields

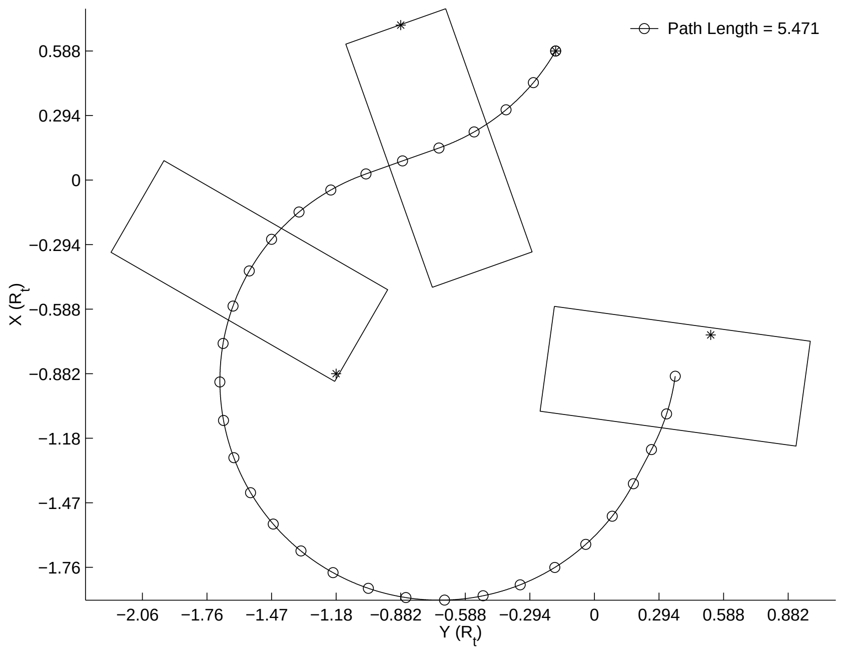


Figure 3.11: Sample path generated by the single-source potential field algorithm.

at any given configuration determine the direction in which the vehicle moves. The advantage of including potentials for all the targets is that the vehicle is able to make a decision that is beneficial for sensing all of the targets, and not just one.

The difficulty with the multi-source method is developing a potential-forcing function for the targets. There are many possible functions, but none can easily be shown to guarantee convergence of the algorithm. We can, however, develop a few basic characteristics of the forcing function to aid in a function selection. First, the potential must increase as the vehicle moves away from a target. Second, for two targets that are equidistant from the vehicle, with one target in front of the vehicle and one target behind the vehicle, the target in front must exert a stronger pull on the vehicle than the target behind. This results in the vehicle wanting to move toward the easily reachable targets in front of it, instead of trying to turn around to get to targets behind the vehicle. Third, the potential force must always be positive.

The most simplistic potential-forcing function is to use the vector from the current configuration to the target as the force (vector $\mathbf{d}_i$ in Figure 3.1). The magnitude of this vector is always positive and increases as the vehicle moves away from

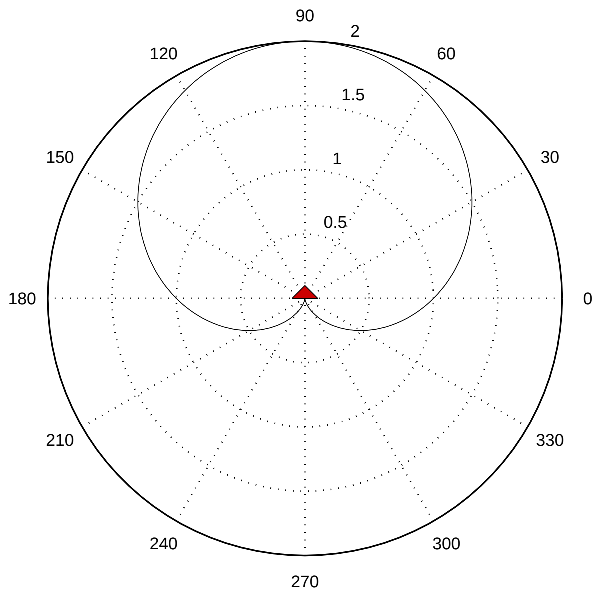


Figure 3.12: The forces from targets are weighted by the function $w(\gamma_i) = (1+\cos(\gamma_i))$ so that the vehicle favors going to the targets in front of it. The effect of a target directly behind the vehicle is zero.

the target. This function, however, does not favor targets that are in front of the vehicle. This is fixed by multiplying the vector, $\mathbf{d}_i$, by a weighting function of the form $w(\gamma_i) = (1 + \cos(\gamma_i))$. As shown in Figure 3.12, the weighting for targets behind the vehicle is nearly zero, and forces for targets in front of the vehicle are magnified. The resultant of these forces, $\mathbf{F} = \sum_i w(\gamma_i)\mathbf{d_i}$, determines the direction the vehicle should move. Figure 3.13 shows the path generated by the multi-source potential-field algorithm for the test case.

The forcing function $\mathbf{F} = \sum_i w(\gamma_i)\mathbf{d_i}$ meets the three requirements stated above, but there exists degenerate cases that cause the potential-field algorithm to not converge. If the targets and vehicle are ever collinear, with all the targets behind the vehicle, then the resultant force is zero and the vehicle's trajectory is not changed. Therefore the vehicle continues on its current heading off to infinity. Also, if the vehicle is equidistant from two targets and flying along the perpendicular bisector of the line connecting the two targets, then $\mathbf{F}$ is always zero. While these special cases can be overcome by never allowing the resultant force to be zero, this is simply a

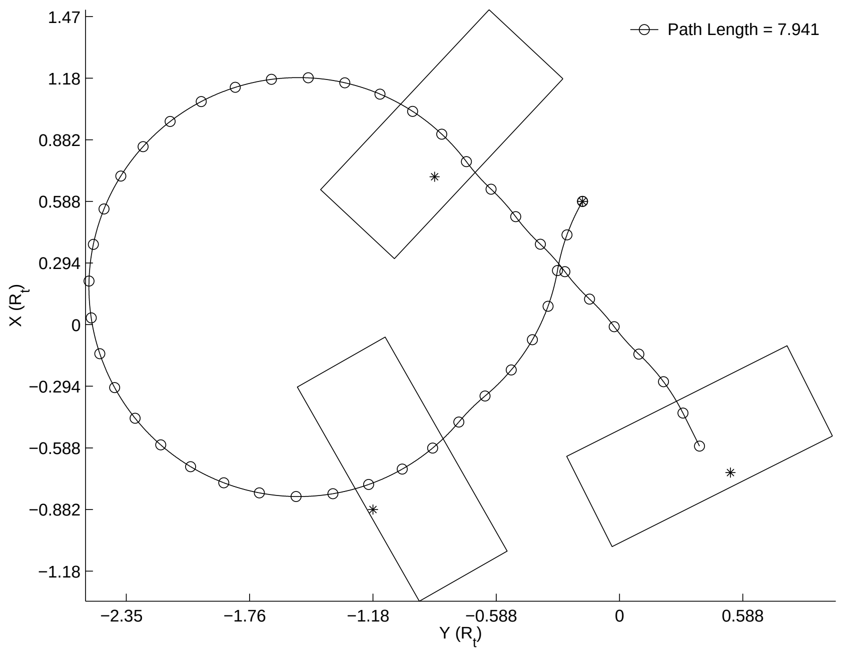


Figure 3.13: Sample path generated by the multi-source potential-field algorithm.

band-aid on a larger problem. Proving that any forcing function will always converge is a difficult problem, and it follows that proving a particular forcing function always produces satisfactory paths is even more challenging. Because of these problems, the potential-field approach by itself is not well suited for solving the path-planning problem.

3.4 Random Branching

A recent focus of path-planning research has been on probability road mapping where a road map is randomly generated and then searched to find the shortest path to some goal configuration. In theory, if the road map generation phase continues indefinitely, the probability of finding the optimal path to the goal becomes unity. PRM does not directly apply to our path-planning problem because there is not a specific goal state, and the road map, or path tree, already exists and is well defined. The goal of our path-planning problem is not to find the path to a specific goal configuration, but to instead find which branches in the path tree accomplish the goal of sensing all the targets. There may be many branches that do so and they must be identified so that the shortest path may be chosen. Random searching, then, may be used to explore the tree and look for these branches. In theory, if the search continued indefinitely, the probability of finding the optimal branch would also converge to unity.

Instead of searching blindly through the path tree, the paths generated by the potential-field methods presented above may be used as a starting point for the random search. At each node of the initial path, the decision is made whether to branch from that node or not. If the decision is made to branch, then a number of random points are selected from the configuration space. The vehicle then plans local paths to each of those points using the single-source potential-field algorithm. Each of these random points essentially become a new starting configuration from which the algorithm begins again: first with the single-source potential-field algorithm, and then random branching along the new path segment. When the targets are all sensed, or the branch becomes too long, the search down that branch is terminated. The search moves back to the last branching node and continues on from there. The outline for the random-branching algorithm is presented in Algorithm 2. The algorithm refers to an overloaded `SingleSource` function which takes as inputs a starting point and a goal point, and returns the path between those two points. The single-source potential-field algorithm may be replaced with the multi-source potential-field algorithm to create the multi-source potential-field with branching algorithm. Example paths from both algorithms are shown in Figures 3.14 and 3.15, respectively.

3.5 LRTA* Tree Search

As stated above, our path-planning problem is to find the branches of the path tree that accomplish the desired objectives, and then select the shortest branch from that set. Since the path tree exists and is well defined, we may apply a Learning Real-Time A* search to the tree to learn the optimal branch. The LRTA* algorithm is well established and has traditionally been applied to path-planning problems in grid based worlds [34]. In general, however, the LRTA* algorithm may be applied to any type of world: grid, tree, directed graph, or other. The LRTA* algorithm is chosen

Algorithm 2: Single-Source Potential-Field with Random-Branching
Input: Initial configuration $P_0 = (x_0, y_0, \psi_0)$
Output: Best Path

```
RandomBranch($P_0$)
(1)   $Path \leftarrow$ SingleSource($P_0$)
(2)   if $length(Path) < bestLength$
(3)       $bestLength = length(Path)$
(4)       $bestPath = Path$
(5)   foreach point in $Path$
(6)       $P \leftarrow$ current point
(7)       if $rand > p_{branch}$
(8)           Generate $n$ random points, $R$
(9)           for $i = 1$ to $n$
(10)              $Path \leftarrow$ SingleSource($P, R_i$)
(11)              $bestPath \leftarrow$ RandomBranch(Path(end))
(12)  return $bestPath$
```

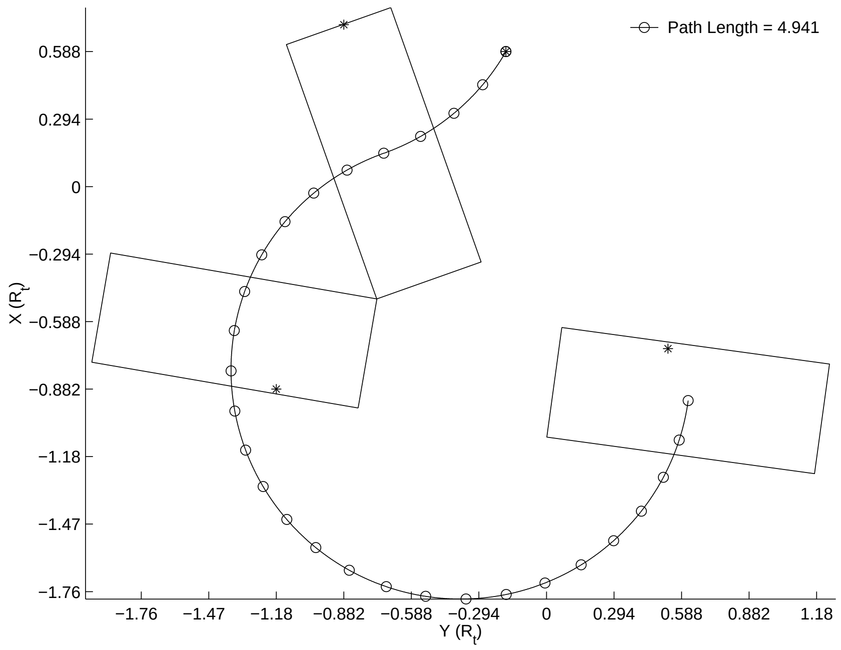


Figure 3.14: Sample path generated by the single-source potential-field with branching algorithm.

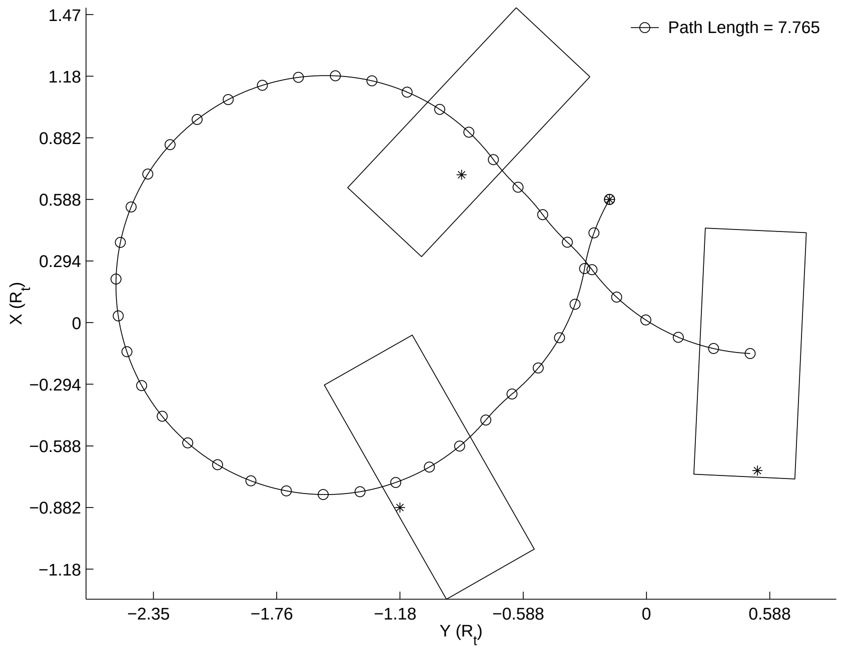


Figure 3.15: Sample path generated by the multi-source potential-field with branching algorithm.

over the faster A* algorithm because, as shown in Chapter 4, the limiting factor on the performance of the path-planning algorithm is the memory space required to store the nodes. Therefore we sacrifice some running time to reduce the spatial complexity of the search by using the LRTA* algorithm.

The LRTA* algorithm itself is simple and elegant and proceeds as follows. Each node, i, has a heuristic estimate, h_i, of the cost to get from itself to the goal. Each node also has a set of m neighbors, N. At each step of the search, the current node, i, calculates $f_j = k_{ij} + h_j \; \forall \; j = 1, \ldots, m$. The value f_j is the heuristic estimate for the j^{th} neighbor plus the cost, k_{ij}, of getting from node i to neighbor j. In other words, f_j is the estimated path length if a move were to be made to neighbor j. Node i updates its heuristic value with $h_i = \min_j f_j$, and then moves to the corresponding neighbor. This continues until the goal is reached, at which point the search begins again from the initial node. At each step of the search, the heuristic value for the current node is updated with a better estimate of the distance to the goal. After some number of iterations, the updated heuristics will converge to the actual path lengths, at which point the search has learned the minimal discrete-step path to the goal. In other words, when $h^* - h = 0$, where h^* is the actual path length, then the search has found the optimal path.

The specific advantage of the LRTA* algorithm for the path-planning problem is that no goal configuration or goal node need be specified. The search is able to learn which branches of the tree accomplish the desired objectives, and then select the shortest path. More specifically, given a set of targets, the LRTA* search learns the optimal movement of the vehicle through the path tree such that the sensor passes over all the targets.

3.5.1 Initial Heuristic

The primary requirement of the LRTA* search is that the heuristic estimates must always underestimate the true path length. In other words, $h_i \leq h_i^*$, where h_i^* is the true path length. The reason for this requirement is simple. If a node on the optimal path overestimates its cost, the search may never move to that node and hence never find the optimal path. Therefore all heuristics must conservatively estimate the true path length. Such heuristics are termed as admissible heuristics [34], and the calculation of the initial values for theses heuristics is essential to the performance of the LRTA* search. It is permissible to let all the initial heuristics be zero, but the search requires a very long time to learn the optimal path. The initial heuristics should estimate the path length as high as possible while still guaranteeing that they are admissible. For the path tree search problem, the heuristic value for a node is the estimate of the distance that must be traveled to sense the remaining targets.

The most simplistic method is to use the distance to the furthest target as the initial heuristic. As shown in the left of Figure 3.16, however, the vehicle can swing the sensor around to the target by making a turn, and thus the vehicle need not go completely to the target. Since it is impossible to determine when this can be done, we assume that the best the vehicle can do is rotate to align the sensor with the line connecting the vehicle and target, and then slide down that line until the target is

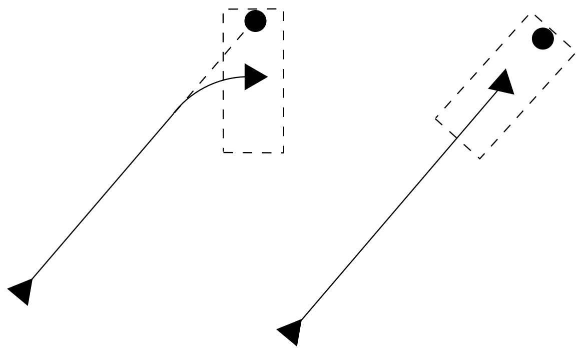


Figure 3.16: Sample path generated by the LRTA* algorithm.

sensed. This approximation is illustrated in the right of the figure. Therefore, the heuristic value is

$$h = \max_i \|\mathbf{d}_i\| - .5y_{sensor}$$

These heuristic values are guaranteed to be admissible, but are not necessarily very close to the actual path length, and therefore the search may converge slowly.

An alternative approach is to find the target closest to the vehicle, and then find the target furthest from this first target as is illustrated in Figure 3.17. To guarantee that the heuristic is admissible, one-half the sensor width is subtracted from the distance to the nearest target, and the full sensor width is subtracted from the second distance. The result are the distances a and b illustrated in the figure. The discrete nature of the path tree must also be accounted for, and so, to be conservative, the step size is subtracted from distance a and twice the step size is subtracted from distance b. The resulting heuristic is

$$h = \min_i \|\mathbf{d}_i\| + \max_j \|\mathbf{t}_j\| - 1.5y_{sensor} - 3dS$$

where $\mathbf{t}_j$ are the vectors from the nearest target to the other targets. These heuristics are guaranteed to be admissible and provide better path-length estimates than the first method. This method is used in the LRTA* algorithm presented below. This method could be expanded to consider more than just the two targets, but then the order in which the targets are visited must be considered as well.

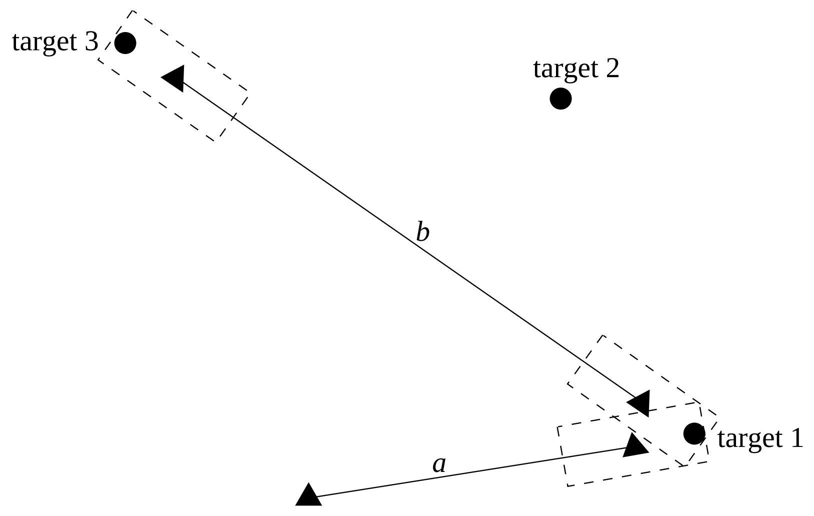


Figure 3.17: Sample path generated by the LRTA* algorithm.

3.5.2 Complexity

A significant problem with searching the path tree is the size of the tree. As the step size decreases, the size of the tree increases very quickly, which makes exploring the tree a lengthy process. A tree of the type presented in Section 3.2 has a size of 3^n, where n is the number of levels in the tree. For a tree with 30 levels there are approximately 2.1×10^{14} nodes. Hopefully the tree search will not need to explore all the nodes, but there will be a large number that must be evaluated, meaning the search will be slow, and require a lot of memory space to store the visited nodes.

Fortunately, the size of the path tree may be decreased by pruning out branches that result in paths that frequently switch directions. For example, if a right turn has been made, there is no point in making a left turn on the next step. Therefore, after a turn has been made, the only choices are to go straight or to turn in the same direction again. This look-back may be extended to consider the last n moves, thus effectively reducing the size and complexity of the tree. An example tree with a two-step look-back is shown in Figure 3.18. This tree structure is still complex, however, with the size at the nth level being given by $e^{0.79524n+0.85865}$. If we follow the above example of thirty levels, the size of the tree will be 5.42×10^{10} nodes, which is a considerable decrease, but still a large number of nodes that may need to be

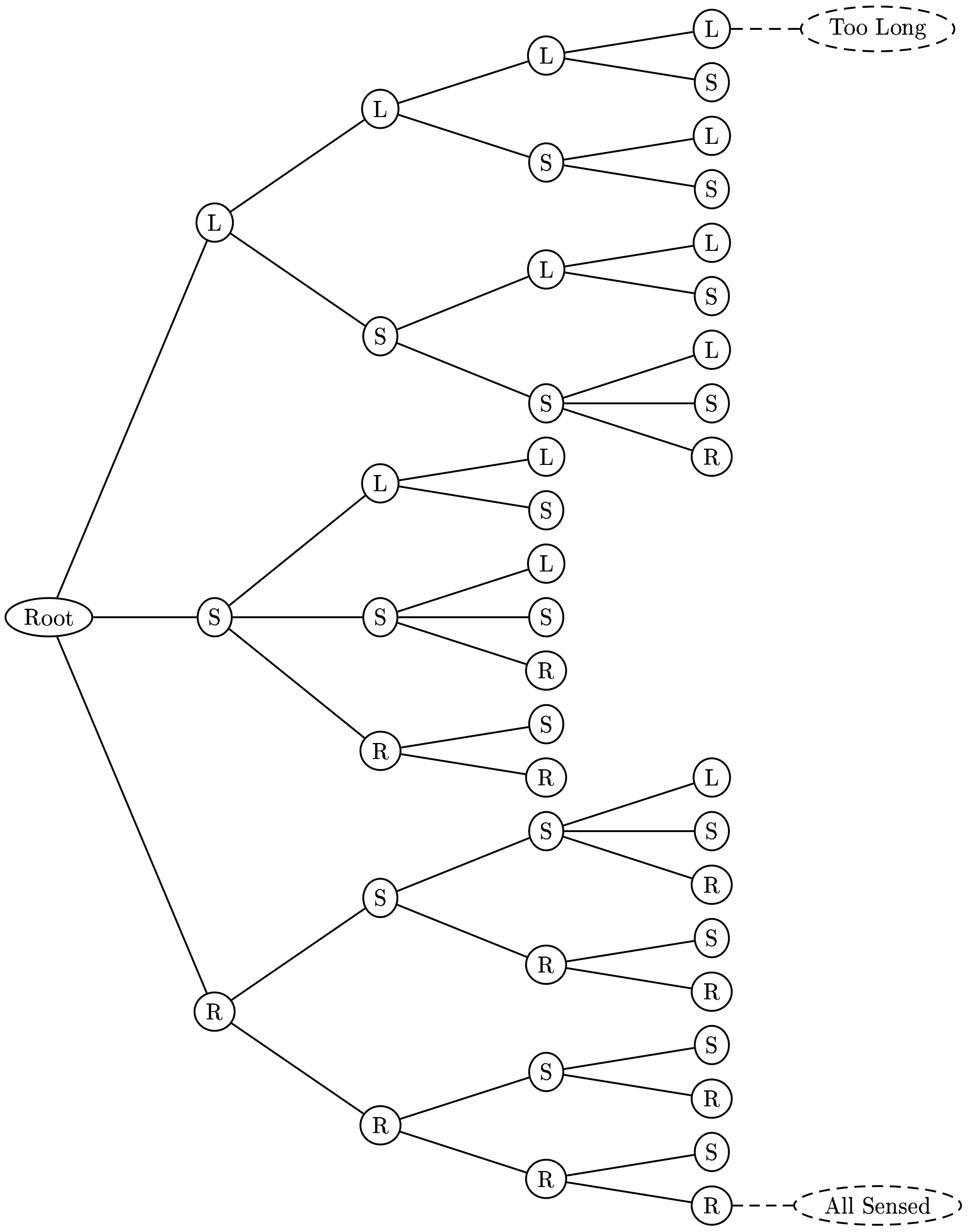


Figure 3.18: Two-step look-back tree structure. The previous two moves limit the possible next moves.

evaluated and stored. Pruning the tree in this manner is particularly helpful when using small step sizes. For larger step sizes, however, it is permissible to allow the turn direction to change between segments, and thus the full path tree structure may be used.

The LRTA* tree search is guaranteed to find the minimum-length path from the class of discrete-step paths. However, the resulting path length is not necessarily the true global optimum. In theory, if the step size in the LRTA* tree search were decreased to some infinitesimally small amount, then the true global optimum would indeed be found. But as we have seen above, the smaller the step size, the larger the tree becomes. The key, then, to making the LRTA* tree search work well, is choosing a step size that is small enough to best approximate the global optimum, but without making the tree too large to search quickly. We show in Chapter 4 that there is a lower bound on the step size that guarantees convergence of the LRTA* tree search in sufficiently short time. We also show that increasing the step size by a small amount has little affect on the average path length, but dramatically reduces the average search running time because the tree size decreases as the step size increases. One other consideration when choosing a step size is that the algorithm only senses targets at the discrete points. Therefore, if the step size is too large, the detection process may miss targets when moving from one node to the next. This may be remedied, however, by adding an algorithm which senses along the path connecting the two nodes instead of sensing targets only at the nodes themselves.

3.5.3 Algorithm Details

Nodes in the path tree contain the vehicle's configuration information and a list of the targets that have been sensed at prior nodes. Nodes are created only as they are needed and, when created, a node copies its parent's information about which targets have been sensed. The node then tests to see if any targets are sensed at its location. The LRTA* tree search proceeds as outlined in Algorithm 3, where Δh_{total} is the total heuristic change for the current run, *count* is the length of the

```
Algorithm 3: LRTA* Tree Search
Input: Set of targets $T$
Output: Path
LRTA($T$)
(1)   while $\Delta h_{total} > 0$
(2)      while $count < bestcount$
(3)         if allSensed
(4)            break
(5)         if !children
(6)            create children
(7)         $m$ = number of children
(8)         for $j = 1$ to $m$
(9)            $f_j = h_j + dS$
(10)        $h \leftarrow \min_j f_j$
(11)        $count = count + 1$
(12)        move to child $k = \arg\min_j f_j$
(13)     if $count < bestcount$
(14)        $bestcount = count$
```

current path, *bestcount* is the length of the best path found thus far, and *allSensed* indicates whether all the targets have been sensed.

Because it may take many search iterations before the heuristic error becomes zero, we need the ability to terminate the path-planning process at any time [43, 44, 45] and still have a feasible path. To guarantee that a feasible path is always available, the LRTA* search is preceded by another path-planning algorithm that will produce a feasible path. The single-source potential-field algorithm explained in Section 3.3.1 is one such candidate. By preceding the LRTA* search with the single-source algorithm, we guarantee that a feasible path is always available at any time the path-planning process is terminated. The length of the LRTA* search phase may then be adjusted as necessary: if more time is available, time can be spent searching for a better path. At the very least, a path will exist that is flyable and senses all the targets.

There are several methods to speed up the tree search and also reduce the number of nodes in the tree. When the currently-explored path becomes longer than the best path, the current knot is terminated by deleting its children and setting its heuristic value to infinity. The search is then restarted at the root node. It is feasible to move to the node's parent and continue the search from there, but doing so results in large portions of the tree being explored without finding a better path. It is more efficient to restart at the root in hopes that the search will explore different parts of the tree and find a better path. A node is also terminated if all of its children have heuristic values of infinity. In this case the node is a dead end, so its children are deleted and its heuristic set to infinity. Doing this helps propagate terminals up the tree, thus speeding the search.

Since the heuristic value of a node is always less than or equal to the actual path length, the heuristic may be used to cull nodes from the tree. If at a given node, $h + count\, dS > bestcount\, dS$, where *count* is the current depth in the tree and *bestcount* is the number of nodes in the currently known best path, then all paths extending from that node are longer than the best path. Therefore the node and all its children are safely terminated.

3.5.4 Terminating Conditions

The method used to terminate the tree search directly affects the running time of the algorithm and the path lengths that are produced. There are three possible terminating conditions: stop when the total heuristic change becomes zero, stop when a maximum running time is reached, or stop when the path length has not improved after some number of iterations. Each of these termination methods has its benefits and disadvantages. Running until the heuristic change is zero guarantees that the optimal path has been found, but requires more running time. Running until a maximum running time is reached guarantees the algorithm terminates by a certain time, but the resulting path may not be optimal. Stopping the search after no improvement for some number of iterations also limits the running time of the algorithm with no guarantee that the resulting path is optimal. It is shown in Chapter 4 that the optimal path is typically found fairly quickly and that the majority of the search time is spent confirming that it is indeed the optimum. It is also shown that terminating after some number of iterations of non-improvement provides a satisfactory trade off of speed and path-length performance. The algorithm with this terminating condition is hereafter referred to as the Non-Improving LRTA*,

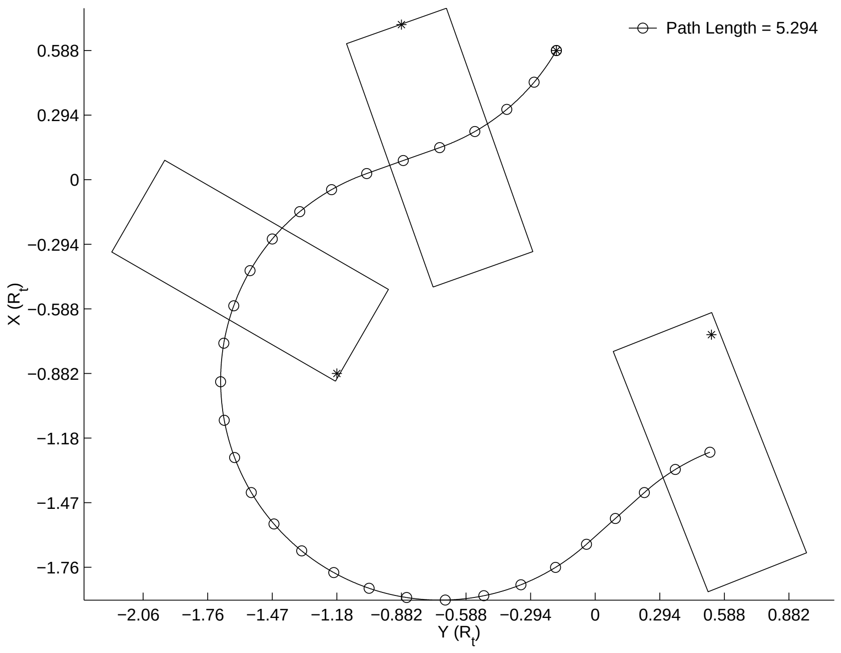


Figure 3.19: Sample path generated by the LRTA* algorithm.

or NILRTA* algorithm. The algorithm that terminates when the heuristic change is zero is simply the LRTA* algorithm.

3.5.5 Implementation Concerns

The primary concern when implementing the LRTA* tree search is the memory required to store the nodes. If the physical memory becomes full, the computer begins using the virtual memory, or hard disk, to store the nodes. Creating and accessing nodes on the hard disk is very slow and the search algorithm essentially grinds to a halt. Therefore lots of physical memory, and good management of that memory, is essential to achieving good performance of the LRTA* search. An implementation of the LRTA* tree-search algorithm in C++ is provided in Appendix A.

One method to help manage the memory is to clean the tree whenever a shorter path is found. Since there is no need to keep nodes that are part of longer paths, these nodes and their children are deleted. Cleaning the tree does require time, and there is no guarantee that the reduction in the node count is worth the time spent, especially if only a few nodes are are deleted from a large tree. Tree cleaning should

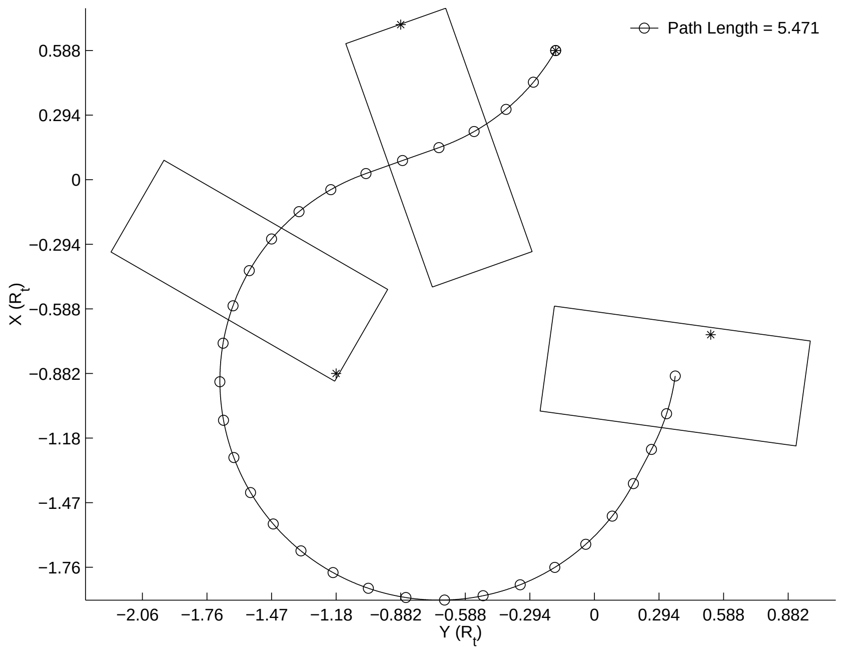


Figure 3.20: Sample path generated by the NILRTA* algorithm.

only be used when there is little memory available and the time trade off is worth the possible memory savings.

3.5.6 Flexibility

The LRTA* tree-search method provides a foundation that is extensible to a number of path-planning problems. The essence of the LRTA* search is learning which path the vehicle should take such that sensor passes over the targets. Adding constraints to the problem is very easy. For example, the direction from which the sensor views the target may be constrained, or a sensor standoff may be incorporated. Adding these constraints simply eliminates branches from the tree. The search is able to learn the branch of the tree that accomplishes the desired objectives, while maintaining the constraints.

3.6 Summary

This chapter presented the discrete-step class of paths. Assembling primitive turns and straights of some specified step size creates a tree of paths, which may be searched for the shortest path that accomplishes the desired objectives. The size of the tree is determined by the step size of the primitives. Smaller step sizes allow better approximations of optimal paths, but increase the size of the tree and thus the complexity of the associated search.

Several algorithms for selecting the optimal path from the tree were also presented. The targets may be treated as sources of potential force which act on the vehicle and influence its movement. In the single-source potential-field algorithm, the target nearest to the vehicle is the only potential source, resulting in a greedy algorithm that is guaranteed to sense all the targets, but not necessarily in an optimal manner. The multi-source potential-field algorithm considers the potential forces from all the targets on the vehicle. This method produces better paths than the single-source algorithm, but is not guaranteed to converge.

To improve the performance of the potential-field algorithms, random branching is added to explore the tree for a better path. The paths generated with the single-source or multi-source potential-field algorithms are used as starting points for the random search. At each node of the initial path, the decision is made to branch or not branch from that node. If branching is to be done, a set of random points are selected and local paths are planned from the current node to those points. If all the targets are not sensed when a random point is reached, the search continues as if the random point is a new root starting location. Using random branching to refine the potential-field paths produces favorable results, but it is difficult to prove that the results are optimal.

The tree may be searched using a LRTA* algorithm, which learns the shortest path in the tree that senses all the targets. The resulting path is guaranteed to converge to the minimum-length discrete-step path as long as the initial heuristic values are admissible. The size of the tree directly relates to the running time of the search. The tree size is reduced by culling out paths that frequently switch direction. The terminating condition for LRTA* search also determines the running time and the path-length performance of the search. A variation of the LRTA* search is to terminate the search when there has been no improvement in the path length for some number of iterations. This algorithm is called the NILRTA* tree search.

Chapter 4

Testing and Comparison

This chapter presents testing and validation for the various path-planning algorithms introduced in Chapter 3. The general testing and analysis procedures are set forth, followed by the testing and validation of the LRTA* and NILRTA* tree searches. The tests show that both algorithms are capable of producing good paths in reasonably short amounts of time. Tests verify that the NILRTA* algorithm is significantly faster than the LRTA* algorithm with only a minor decrease in path-length performance. These tests also show that the LRTA* tree search with a step size of 0.177 radians produces paths that are reasonable approximations to the optimal paths, thus establishing a benchmark to which the other algorithms are compared.

The running time and path-length performance of the potential field, potential field with branching, and NILRTA* algorithms are compared to each other and the LRTA* benchmark. These tests show that the single and multi-source potential field with branching algorithms are viable options to the LRTA* and NILRTA* algorithms. Both random-branching algorithms exhibit reasonable path-length performance and fast running times. The pure potential-field algorithms, while very fast, do not always produce good paths. The various algorithms are also tested on scenarios with one to eight targets, which shows that all of the algorithms are capable of handling any number of targets. Finally, the performance of the LRTA* and NILRTA* algorithms is tested on computing platforms with different processor speeds and memory capacity.

Table 4.1: Specifications for the various computing platforms

Computer	Component	Specifications
Jaguar	Processor	Athlon XP 1500+ (1.33GHz)
	OS	Gentoo Linux
	RAM	512 Mb PC150 @ 133MHz
	Hard Disk	Western Digital 120Gb, 7200 RPM, 8Mb, ATA 100
	Compiler	gcc 3.2 with optimization level 3
Robot 4	Processor	Pentium II 266MHz
	OS	Linux
	RAM	64 Mb PC66
	Hard Disk	6.4Gb
	Complier	gcc 2.95.4 with optimization level 3
Tomservo	Processor	Pentium II 300MHz
	OS	Debian Linux
	RAM	128 Mb PC66
	Hard Disk	Unknown
	Complier	gcc 2.95.4 with optimization level 3

4.1 General Testing and Analysis Procedures

4.1.1 General Testing

Each of the algorithms was tested on a set of 2000 target scenarios in a $2.35R_t$ by $2.35R_t$ world, and include three randomly-selected targets and a random initial position and heading for the vehicle. For heading changes of 0.47 radians and above, the full path tree is used instead of the simplified tree discussed in Section 3.5.2. Running times for and paths lengths produced by the algorithms are used to generate statistical performance measures as explained below. The running time for any algorithm is limited to one minute, after which it is terminated. Data from terminated runs are used in the analysis, but a note indicates the number of runs that did not complete in the required time. The terminating condition for the NILRTA* algorithm is 10,000 iterations of no improvement. All tests are performed on the Athlon computer designated as Jaguar in Table 4.1, unless otherwise noted.

Because of the complexity of the LRTA* tree search, step sizes are no smaller than 0.177 radians. For step sizes of less than 0.177 radians, the tree becomes too complex to search for most problems. It is shown, however, that the performance trend for increasing the heading change suggests that using step sizes below 0.177 radians does not significantly decrease the path lengths produced by the algorithm. Therefore, we assume that the path obtained with $d\psi = 0.177$ radians is a sufficient approximation to the optimal path.

4.1.2 Statistical Analysis

The running time and path length data from the algorithm tests follow a bimodal distribution as shown by the example path-length data in Figure 4.1. Bimodal distributions usually suggest that there are two different operating conditions involved, such as a batch of parts where half were made on a new milling machine and the other half were made on an old, sloppy milling machine. For the path-planning problem, the cause of the two modes is that the target scenarios tend to fall into two categories: those that require short paths, and those that require long paths. In other words, there are some problems where the vehicle can fly mostly straight ahead and sense the targets. In other problems the vehicle must make nearly a full turn to accomplish its tasks. Examples of these two types of paths are shown in Figure 4.2.

Analysis of bimodal distributions is difficult because common statistical characteristics, such as mean and standard deviation, do not directly apply. Typically, bimodal distributions are separated into their two modes, such as separating the batch of parts according to which machine they were made on, and each mode analyzed individually. This cannot be done for the path-planning data because it is difficult to separate problems with medium length paths into either the short or long group.

The bimodal distributions are analyzed by finding the probability density function (pdf) for each data set, and then calculating some confidence level for the data. The pdf is found by scaling and normalizing the frequency function, or histogram, of the data. Since the histogram is a discrete approximation to the pdf, smaller bin sizes in the histogram provide a more accurate estimate of the pdf.

The pdf is calculated by first scaling the abscissa of the histogram, and then normalizing the ordinate. The abscissa is scaled using the z transform

$$z_i = \frac{x_i - \mu}{\sigma}$$

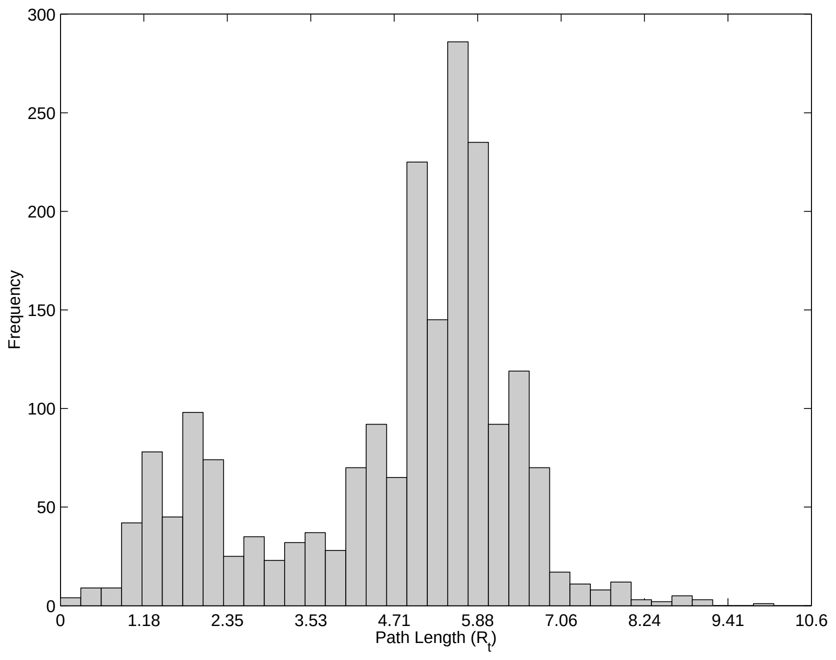


Figure 4.1: The data for the various tests follow a bimodal distribution, which requires specialized analysis to make use of the information.

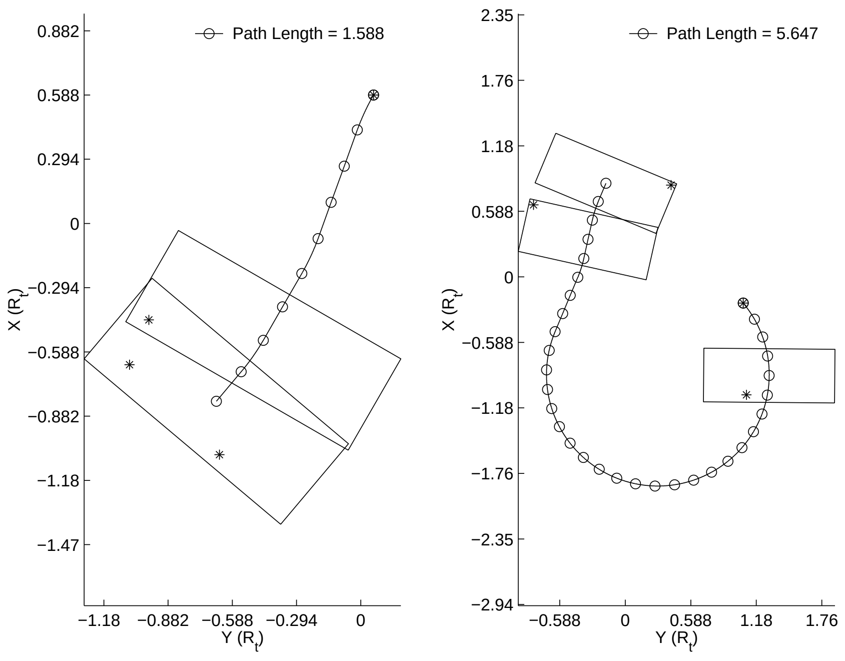


Figure 4.2: Target scenarios tend to require short or long paths

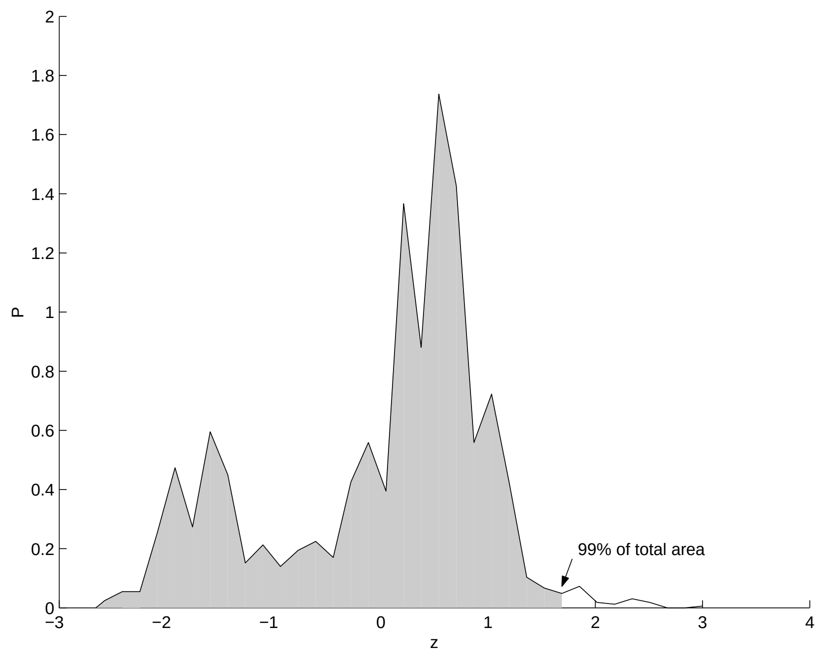


Figure 4.3: A normalized probability density function for the data in Figure 4.1, with the 99% confidence region is shaded

where x_i is the center of the i^{th} histogram bin, and μ and σ are the data's mean and standard deviation, respectively. The frequency is normalized by

$$P_i = \frac{f_i}{N\, dz_i}$$

where N is the total number of data points, f_i is the number of data points in the i^{th} bin, and dz_i is the width of the bin. The result is a discrete function $P_i = f(z_i)$ where

$$\sum_{i=-\infty}^{\infty} P_i\, dz_i = 1$$

as is required for any probability density function. A confidence interval for the resulting pdf is found by solving for the value of a such that

$$\sum_{i=0}^{a} P_i\, dz_i = p$$

where p is the desired confidence probability. Since it makes no sense to have negative running times or path lengths, the lower bound for the confidence interval is zero. Figure 4.3 shows the probability function for the data in Figure 4.1, along with the

99% confidence region. The z-value at which this confidence level occurs corresponds to a path length of $7.62R_t$. Therefore, 99% of the paths in this example have lengths of less than $7.62R_t$.

The algorithm analyses presented below generated histograms with 5,000 bins, which results in a good approximation of the pdf. Using more than 5,000 bins does not significantly improve the accuracy of the calculation. The analyses use a confidence interval of 99.7% to indicate the spread of the data.

4.2 LRTA* Tree Search Validation

To demonstrate the advantages of the LRTA* tree search, a comparison is made between the path generated by the LRTA* algorithm and the path found using the brute-force global search of the constrained-end path class. The sensor footprint for the LRTA* algorithm is narrowed to $0.21R_t$ by $0.029R_t$ to approximate the vehicle passing through the target points, and the global search has a resolution of one degree. Figure 4.4 shows that the LRTA* tree search produces nearly the same optimal path as the brute-force global search, with the paths differing in length by $0.049R_t$. Reseting the sensor to its original dimensions produces the path shown in Figure 4.5. This path is 33%, shorter than the minimum-length constrained-end path, and demonstrates the ability of the LRTA* tree-search algorithm to utilize the full sensor footprint and to learn the order in which to view the targets.

4.2.1 Step Size

The step size of the discrete-step path tree directly influences the running time and path-length performance of the LRTA* tree-search algorithm. It is proposed that increasing the step size significantly reduces the running time of the algorithm with only a moderate increase in the resulting path lengths. These tests use step sizes between 0.177 and 0.706 radians, in 0.059 radian increments.

The path-length performance for the two algorithms is presented in Figure 4.6. Plot (a) shows the mean path length at the various step sizes, and plot (b) shows the mean path length difference between the NILRTA* and LRTA* algorithms. We see

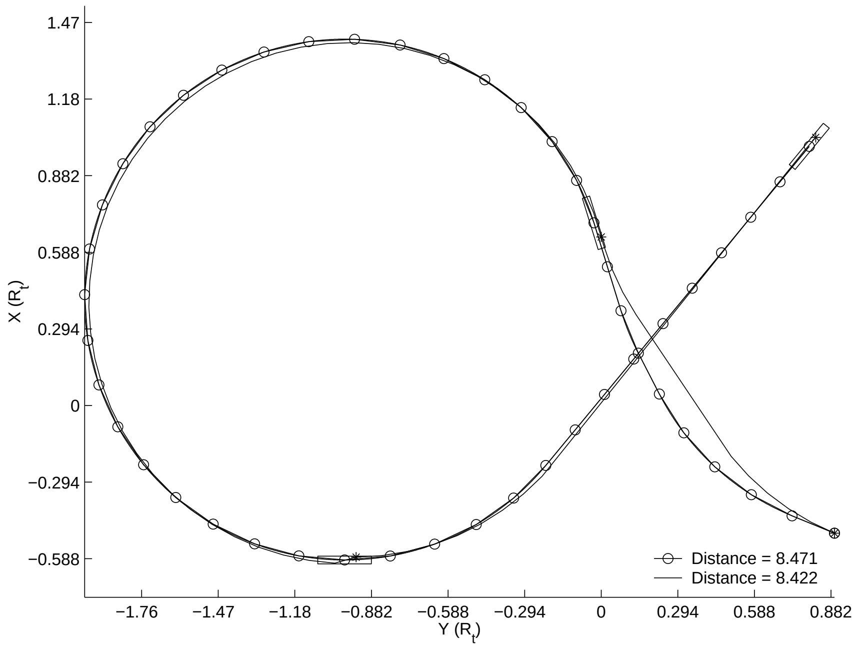


Figure 4.4: The LRTA* algorithm and the discrete path class are capable of producing nearly the same optimal path through a set of targets as the brute-force global search of the constrained end path class.

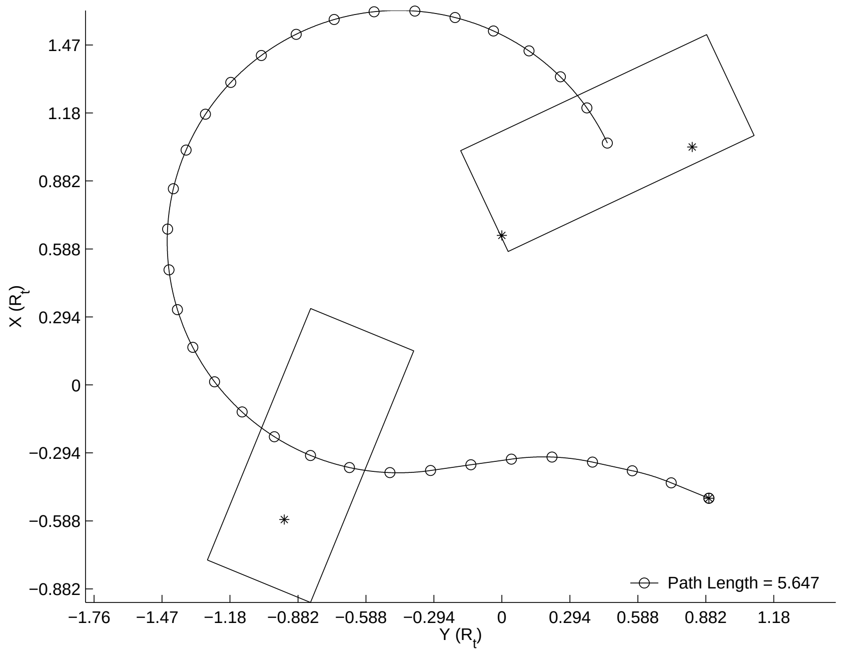


Figure 4.5: The LRTA* algorithm utilizes the vehicle's full sensing capabilities and produces paths far superior to those generated from the constrained end path class.

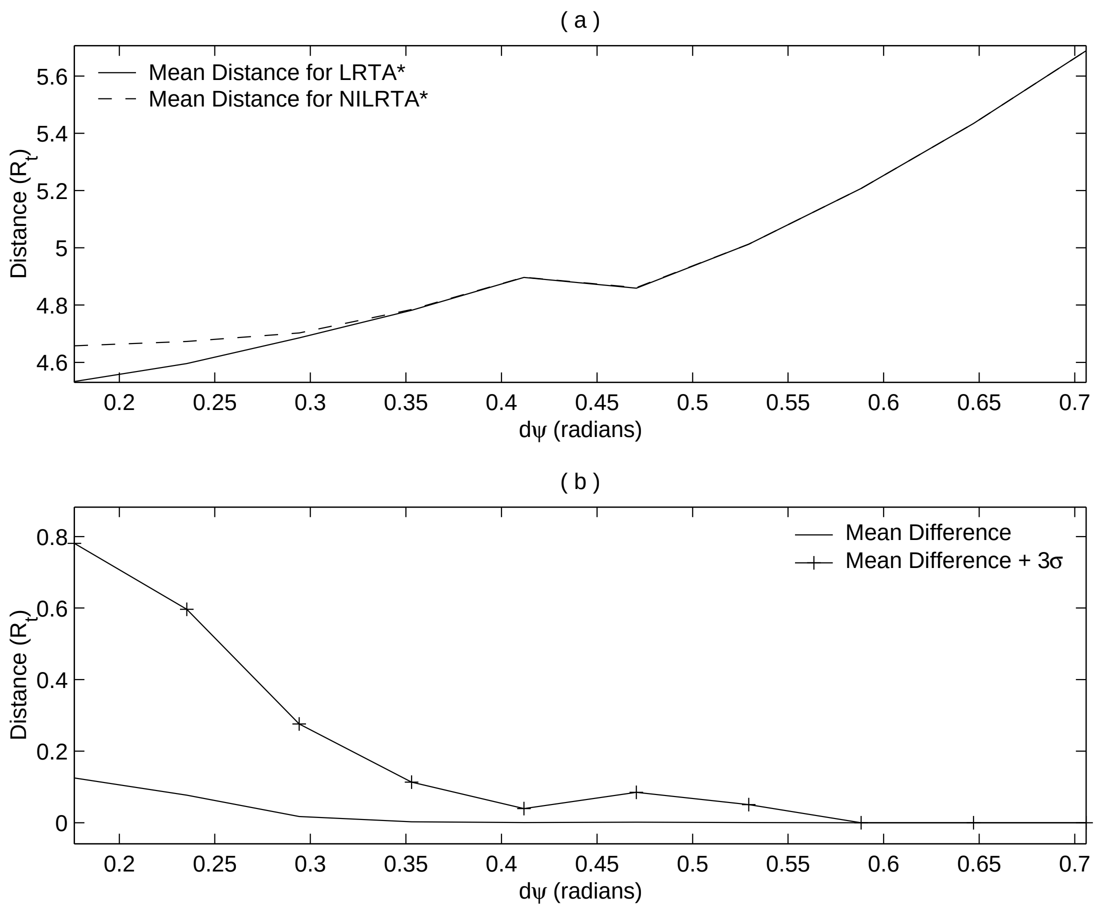


Figure 4.6: Distance comparison between the LRTA* and NILRTA* algorithms. (a) shows that NILRTA* produces average path lengths comparable to LRTA*. The average difference in path length between the two algorithms is shown in (b).

that the average path length increases approximately $1.12R_t$ over the range of step sizes. These results indicate that smaller step sizes are better, however, the increase in the mean path length between 0.177 and 0.412 radians is only about $0.353R_t$. This suggests that step sizes in the range of 0.177 and 0.412 radians produce acceptable results. Also shown in the two plots is that the mean path-length performance of the NILRTA* algorithm is comparable to that of the LRTA* algorithm. Plot (b), however, shows that for a step size of 0.177 radians, the confidence level for the difference in path lengths between the LRTA* and NILRTA* algorithms is over $0.588R_t$. For step sizes of 0.294 radians and above, the NILRTA* algorithm produces paths that are within 6% of the LRTA* paths 99.7% of the time.

At a step size of 0.471 radians, the algorithms switch to searching the full path tree instead of the simplified path tree. The path-length performance improves slightly, but for step sizes above 0.529 radians, the mean path lengths are more than $.588R_t$ greater than the mean path length at $d\psi = 0.177$ radians. This indicates that step sizes of 0.588 radians and greater should not be used for the LRTA* and NILRTA* algorithms. Note that the confidence level indicated in Figure 4.6 (b) is the mean plus three standard deviations. The distance-difference data follow a normal distribution, so the 3σ confidence level is used.

Estimate of Minimum Average Path Length

The trend for the average path lengths at step sizes of 0.177 and 0.235 radians suggests that the average path length at step sizes of 0.118 and 0.059 radians is not significantly better. We validate this assumption by fitting a quadratic curve to the average path-length data at step sizes of 0.177 to 0.412 radians. In theory, for an infinitesimally small step size, the tree search should produce the optimal path. Thus, we constrain the quadratic curve to have a slope of zero at a step size of zero, leading to the curve of the form $f(x) = ax^2 + b$. Using least squares to calculate the coefficients produces the curve shown in Figure 4.7. We see that average path length at a step size of 0.177 radians is only 1.8% longer than the predicted average path length at a

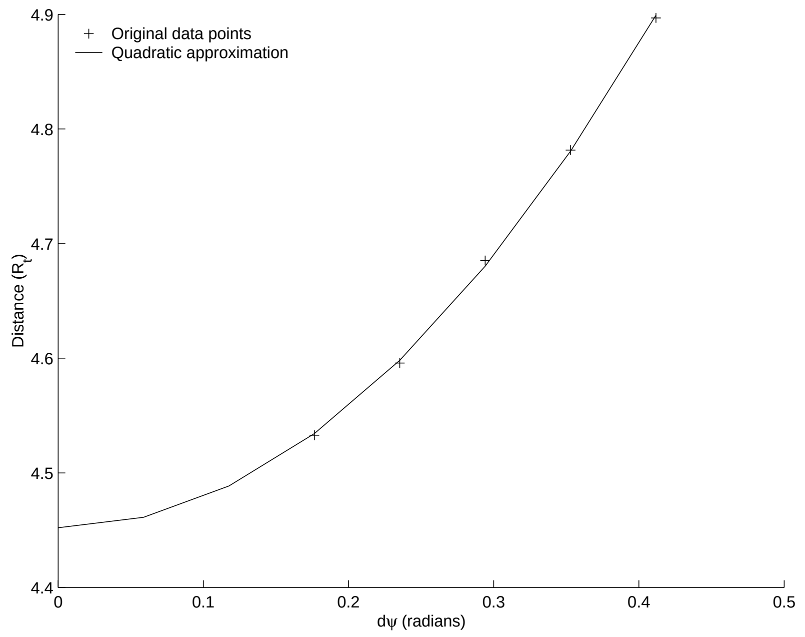


Figure 4.7: Curve fitting the average path-length data shows that a step size of 0.177 radians provides reasonable approximations to the optimal paths.

step size of zero radians. Therefore, the assumption that the paths produced with a step size of 0.177 radians are reasonable approximations to the optimal paths is valid.

Running Times

Figure 4.8 presents the running-time results for the two algorithms. The average total running times for both the LRTA* and NILRTA* algorithms at the various step sizes are shown in plot (a). At $d\psi = 0.177$ radians, the average running time of the LRTA* algorithm is about 18 seconds, and about 0.4 seconds for the NILRTA* algorithm. At a step size of 0.235 radians, the average total time for the LRTA* algorithm drops to about 2 seconds, which is an 89% decrease. The NILRTA* algorithm's running time drops to about 0.3 seconds. As the step size continues to increase, the average running times decrease further, leveling off at about 0.01 seconds. The confidence interval indicates that the spread of times for the LRTA* algorithm is broad for smaller step sizes, with the upper bound at a step size of 0.177 radians being 60 seconds, which is the running-time limit for the algorithm. The bound for the

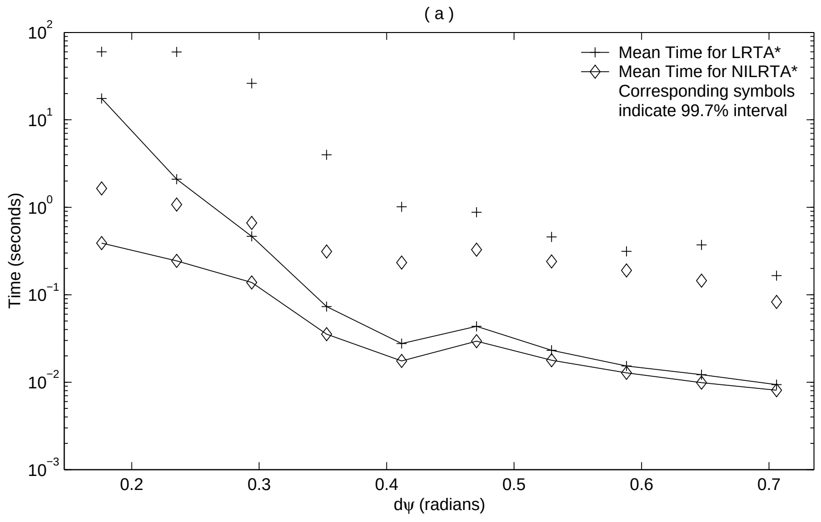
NILRTA* algorithm is about 1.75 seconds, which is significantly better than that for the LRTA* algorithm. For step sizes of 0.471 radians and above, the two algorithms perform nearly the same. The running times increase slightly where the switch to using the full path tree is made, but continues to decrease as the step size increases.

The results shown in plot (b) of Figure 4.8 are the mean running times at which the best paths are found. This means that any additional running time is spent confirming the optimum path. For the LRTA* algorithm, roughly 79% of the total running time is spent confirming the optimal path. The NILRTA* spends about 90% of its total running time processing the 10,000 iterations of no improvement. Thus, if either algorithm is terminated prematurely, the likelihood of having found the best path is high.

The path-length and running-time results show that increasing the step size improves the running time without significantly decreasing the path-length performance. Step sizes of 0.177 and 0.412 radians are acceptable. For step sizes above 0.412 radians, the path-length performance becomes unacceptable. The test results also show that the NILRTA* algorithm runs significantly faster than the LRTA* algorithm, and has comparable path-length performance. Step sizes of 0.235 or 0.294 radians produce the best results for the NILRTA* algorithm. In general, these results show that there is a need to improve the efficiency of the LRTA* search so that both the LRTA* and NILRTA* tree search algorithms may converge to the best path more quickly. Also, using the full path tree for larger step sizes does not significantly improve the path-length performance of the algorithms.

4.2.2 Running Time vs. Tree Depth

Figure 4.9 shows the mean running time of the LRTA* and NILRTA* algorithms versus the depth of the path tree. The LRTA* algorithm can only handle tree depths of about twenty and still have running times of less than 0.5 seconds. On the other hand, the NILRTA* algorithm handles tree depths of about thirty in the same amount of time. These results also show that the significant advantage of



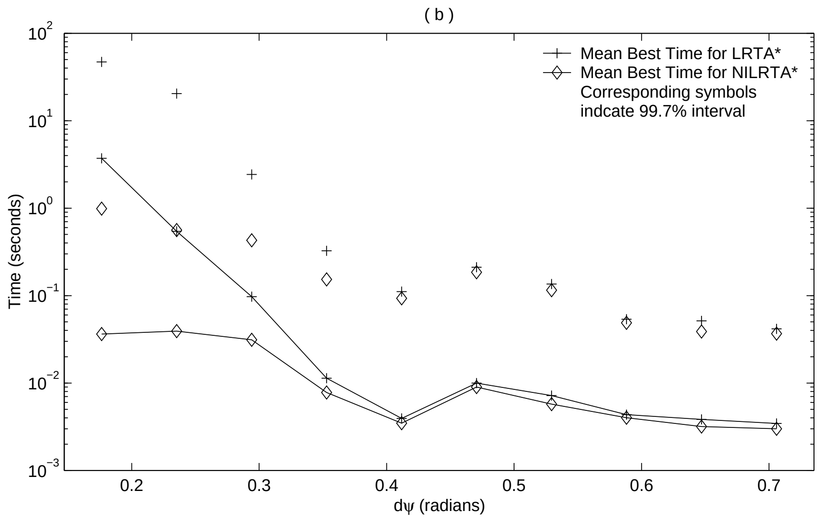


Figure 4.8: LRTA* and NILRTA* running time comparison for increasing step sizes.

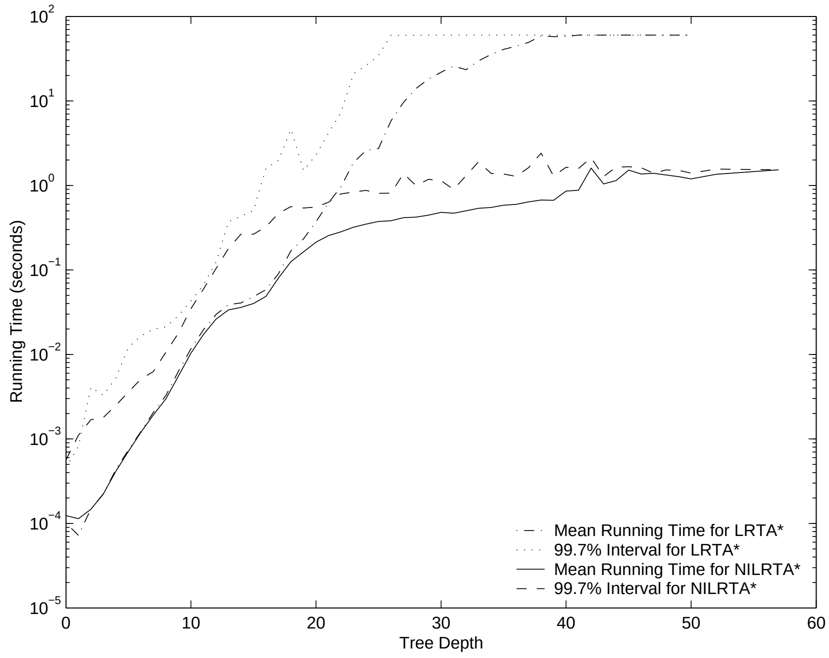


Figure 4.9: Running times of the LRTA* and NILRTA* versus the depth of the path tree on Jaguar.

the NILRTA* algorithm is its performance for tree depths above thirty. The NILRTA*'s running time converges to about 1.5 seconds for tree depths of forty and higher, which suggests that the NILRTA* algorithm could be used for any size path tree. The chances of finding the best path before the run limit is reached decreases, however, as the tree size increases. Therefore the path-length performance of the algorithm tends to degrade with increasing tree sizes.

4.2.3 NILRTA* Running Limit

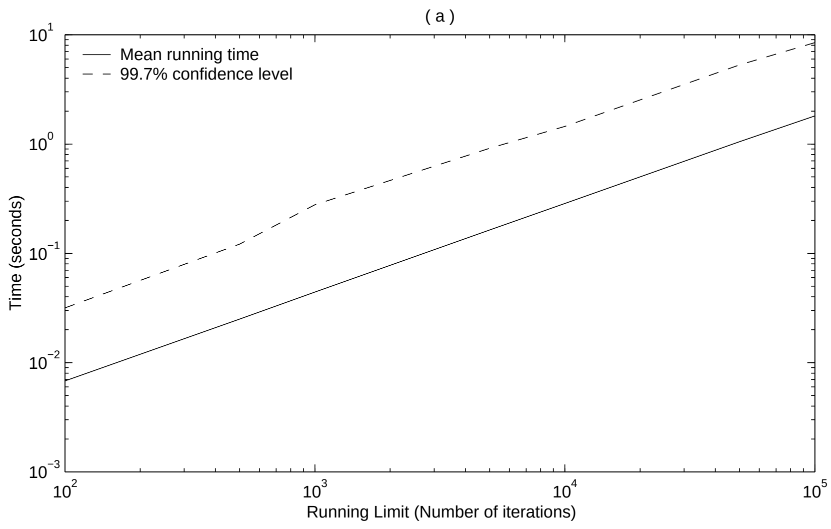
The running time and path-length performance for the NILRTA* algorithm are determined by the number of non-improving iterations that must lapse before the algorithm terminates. The results presented in Figure 4.10 compare the average running time and path length for different run limits at a step size of 0.177 radians. Plot (a) shows that the running time increases with the run limit, and that for run limits between 100 and 1,000 iterations, the algorithm terminates very quickly. At a run limit of 1,000 iterations, the mean running time is about 0.045 seconds. The path-length results in plot (b) show that the NILRTA* algorithm produces better paths with higher run limits, but the improvement in the path length is only about $0.265R_t$, or 5.8%, over the range of run limits. These results suggest that the NILRTA* algorithm with a run limit of 100 iterations may be used to produce satisfactory paths in an average time of less than 0.01 seconds. This assumption was tested by repeating the step-size comparison tests with the NILRTA* algorithm using a run limit of 100 iterations. The running time and path-length results are presented in Figure 4.11. The average running times at all step sizes are significantly faster than those for the LRTA* algorithm. The average path lengths, however, are about $0.235R_t$ longer than those produced by the LRTA* tree search. This indicates that a run limit of 100 iterations may produce acceptable results if short paths are not of primary concern. If shorter paths are necessary, the run limit should be increased, perhaps to 1,000 or 5,000 iterations.

4.3 Algorithm Comparisons

The preceding discussion establishes the LRTA* algorithm with a step size of 0.177 radians as the benchmark for comparison of the other algorithms. This section presents comparison testing of the six path-planning algorithms with a step size of 0.177 radians.

The results presented in Figure 4.12 show that the average running times for the potential-field methods are under one millisecond. Adding branching to the potential-field algorithms increases the running time, but the 99.7% confidence level is still under 0.05 seconds for both algorithms. The LRTA* tree search is the slowest of the algorithms with an average run time of 17.5 seconds. The NILRTA* algorithm on the other hand, has an average running time of 0.39 seconds with a 99.7% confidence level of 1.6 seconds.

The path-length comparison for the algorithms is presented in Figure 4.13. The LRTA* tree search has the an average path length of $4.53R_t$, followed by $4.66R_t$ for the NILRTA* algorithm. The single-source potential-field algorithm performs



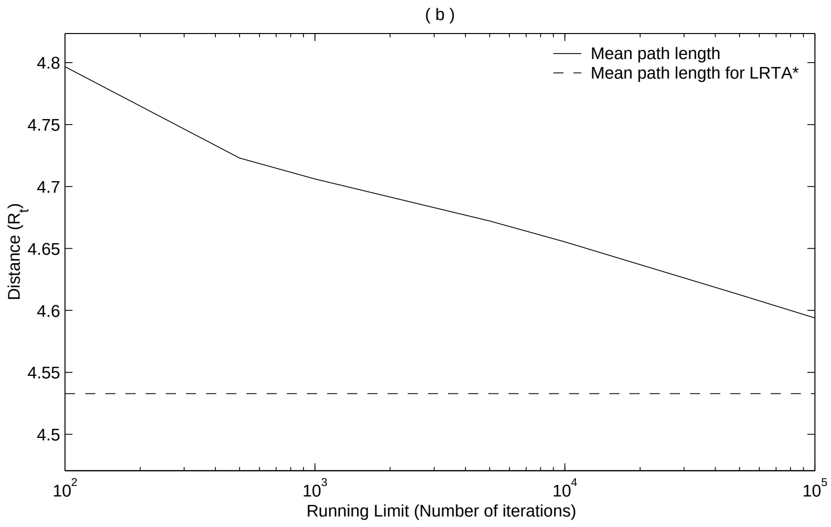


Figure 4.10: (a). Lower run limits significantly increase the speed of the NILRTA* algorithm. (b). Increasing the run limit produces better average path lengths, however, the amount of improvement is only about $0.265 R_t$t.

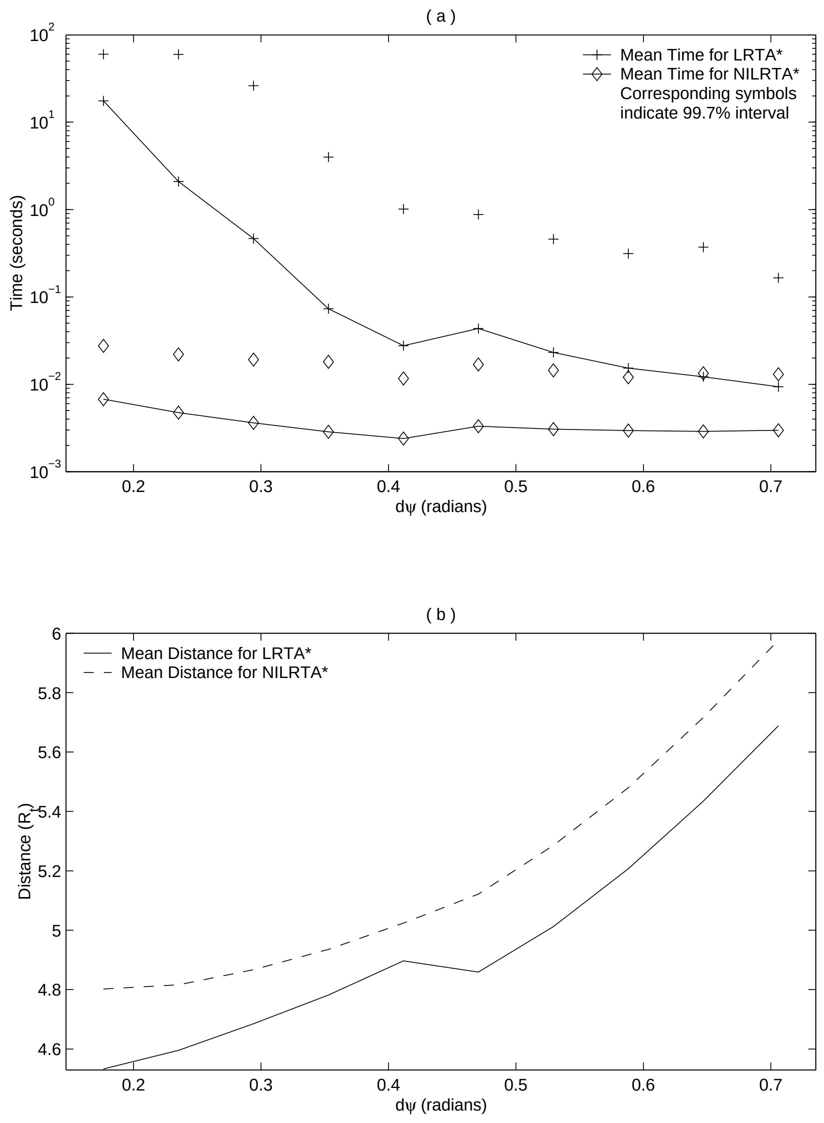


Figure 4.11: Mean running times and path lengths for the LRTA* algorithm, and the NILRTA* algorithm with a run limit of 100 iterations. The step size is 0.177 radians.

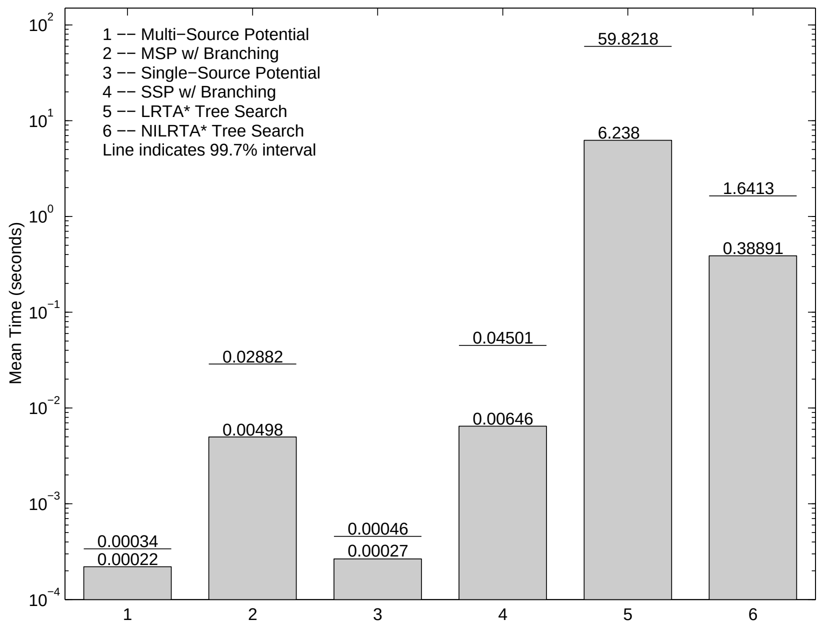


Figure 4.12: Running time comparison of all the algorithms.

the worst, with an average path length of $6.67R_t$, which is an increase of 47% over the LRTA* algorithm . Adding branching to either of the potential-field algorithms significantly increases their performance to $5.34R_t$ (18% over LRTA*) for the single-source with branching algorithm, and $5.00R_t$ (10% over LRTA*) for the multi-source potential-field with branching algorithm.

The results of this comparison suggest that the potential-field algorithms, while not provably optimal, perform rather well, especially when augmented with random branching. The single and multi-source potential-field with branching algorithms are a viable alternative to the LRTA* and NILRTA* algorithms. Both of these branching algorithms are fast and provide fairly good path-length performance. These results also show that there is a definite trade off between speed and path length: if a shorter path is needed, then the time must be spent to get it.

4.3.1 More Than Three Targets

The results presented thus far were all gathered using scenarios with only three targets. Obviously we would like to see how the various algorithms perform for more

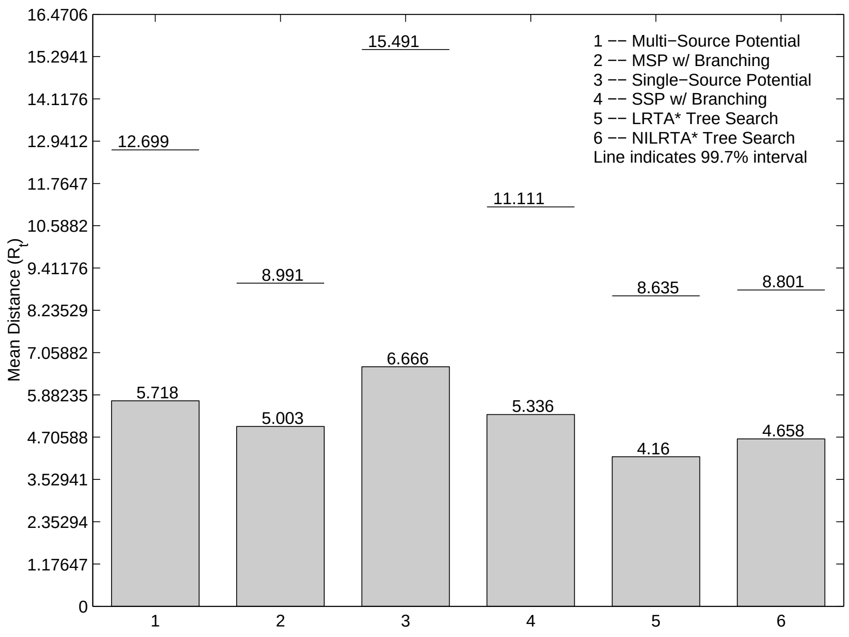


Figure 4.13: Distance comparison for all the algorithms.

or less than three targets. Two-thousand target scenarios are used for each number of targets between one and eight, with a step size for the tests of 0.235 radians.

The running-time results in Figure 4.14 show that the single and multi-source potential-field algorithms, and the single and multi-source potential-field with branching algorithms are fairly insensitive to the number of targets. On the other hand, the running times for both the LRTA* and NILRTA* algorithms increase with the number of targets. The 99.7% confidence interval for the LRTA* search is about fifteen seconds with eight targets, which is not favorable. In fact, the LRTA* algorithm does not do very well for any more than three targets. The running times of the NILRTA* algorithm are about 88% better than those for the LRTA* algorithm, however, with a 99.7% confidence level of approximately 1.75 seconds for eight targets. The speed of the NILRTA* algorithm may be further increased by reducing the run limit for the algorithm. Doing so should decrease the path-length performance of the algorithm by only a small amount, as discussed in the previous section.

Figure 4.15 presents the path-length performance for the various algorithms. All of the algorithms have average path lengths that are about the same. The spread

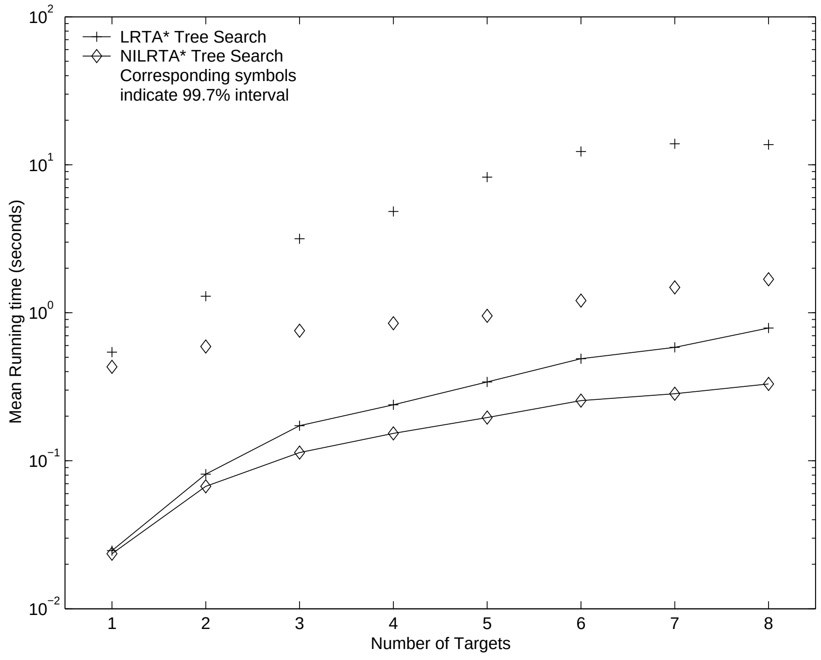


Figure 4.14: Running time comparison with different numbers of targets.

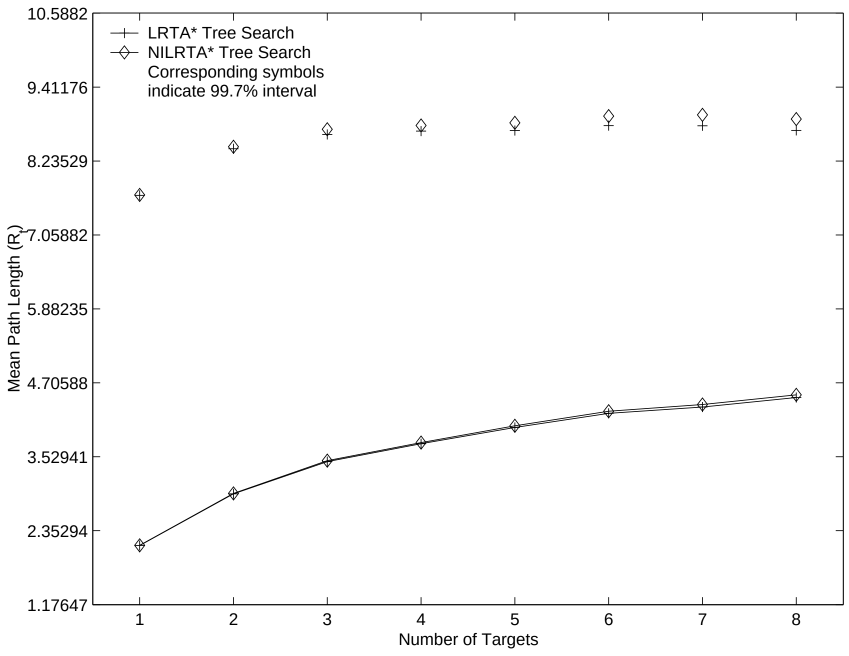


Figure 4.15: Distance comparison with different numbers of targets.

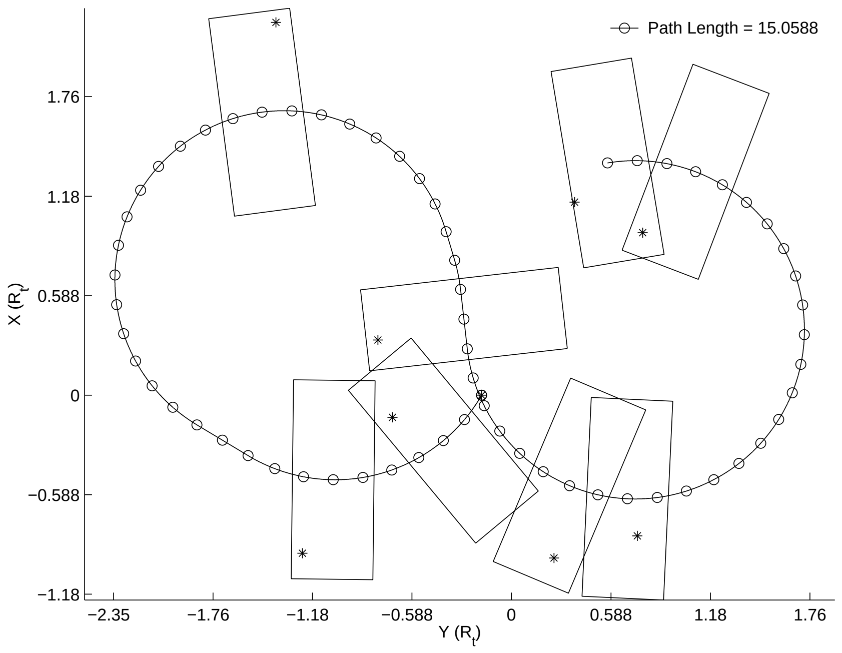


Figure 4.16: Example path produced by the LRTA* algorithm for sensing eight targets.

of the path lengths, however, as indicated by the 99.7% confidence intervals, show that the potential-field algorithms perform the worst, followed by the potential-field with branching algorithms, and then the LRTA* and NILRTA* algorithms. It is interseting that the average path lengths for the LRTA* and NILRTA* algorithms are higher than those for the other four algorithms for numbers of targets greater than five. This is ascribed to the fact that longer paths are required for higher numbers of targets, and that, in some cases, the tree searches termninated before finding the optimal path. The confidence levels for the LRTA* and NILRTA* algorithms, however, are still lower than those for the other algorithms. We conclude from these tests that all of the algorithms are capable of handling any number of targets. The potential-field with branching algorithms and the LRTA* and NILRTA* algorithms produce better path lengths, but with more running time. The results also show that the NILRTA* algorithm is better suited for handling any number of targets than is the LRTA* algorithm. Figure 4.16 shows the path generated by the LRTA* algorithm for an eight-target scenario.

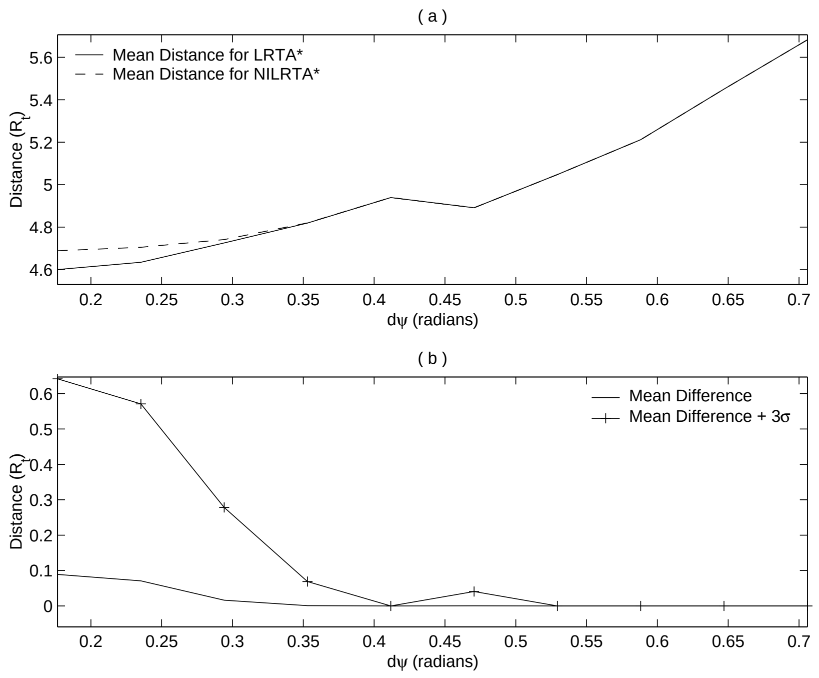


Figure 4.17: LRTA* distance comparison for increasing step sizes on Tomservo.

4.4 Performance on Different Computing Platforms

The test results presented above were all gathered on the high-speed computing platform designated as Jaguar in Table 4.1. Since we are interested in using the LRTA* and NILRTA* path-planning algorithms on any computing platform, tests are also run on the additional computing platforms listed in the table.

Figures 4.17 and 4.18 show the mean path lengths obtained on Tomservo and Robot 4, respectively. The average path lengths are comparable to those obtained on Jaguar. This suggests that the computing platform has little or no effect on the path-length performance of either algorithm.

Figures 4.19 and 4.20 show that the mean running times on the two platforms for both algorithms at any step size is slower than that for Jaguar. To have average running times of less than 0.5 seconds, the LRTA* algorithm needs a step size of 0.353 radians or greater on Tomservo, and a step size of 0.412 radians or greater on Robot 4. The NILRTA* algorithm needs step sizes of 0.294 radians and 0.353 radians on Tomservo and Robot 4, respectively. The average running times versus tree depth

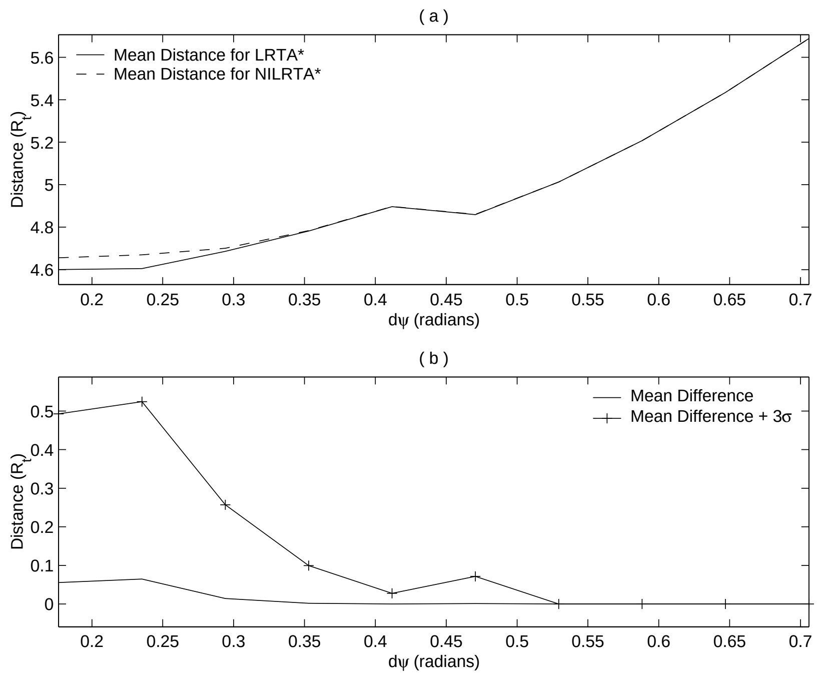


Figure 4.18: LRTA* distance comparison for increasing step sizes on Robot 4.

for the two platforms are presented in Figures 4.21 and 4.22. Again the NILRTA* algorithm shows better performance than the LRTA* algorithm.

The conclusion of these tests, then, is that both the LRTA* and NILRTA* algorithms may be used on slower computers if there is a trade off made between path-length performance and speed. The processor speed and memory capacity of the computer directly affect the performance of the algorithms, and hence more speed and more memory provide better performance. As discussed in Section 4.2.3, however, reducing the run limit for the NILRTA* algorithm should sufficiently increase the algorithm's speed to make it feasible for use on any computing platform.

4.5 Test Conclusions

The testing and validation presented above shows that the LRTA* and NILRTA* algorithms are successful at quickly computing satisfactory paths. The NILRTA* algorithm is much faster than the LRTA* algorithm with only a slight decrease

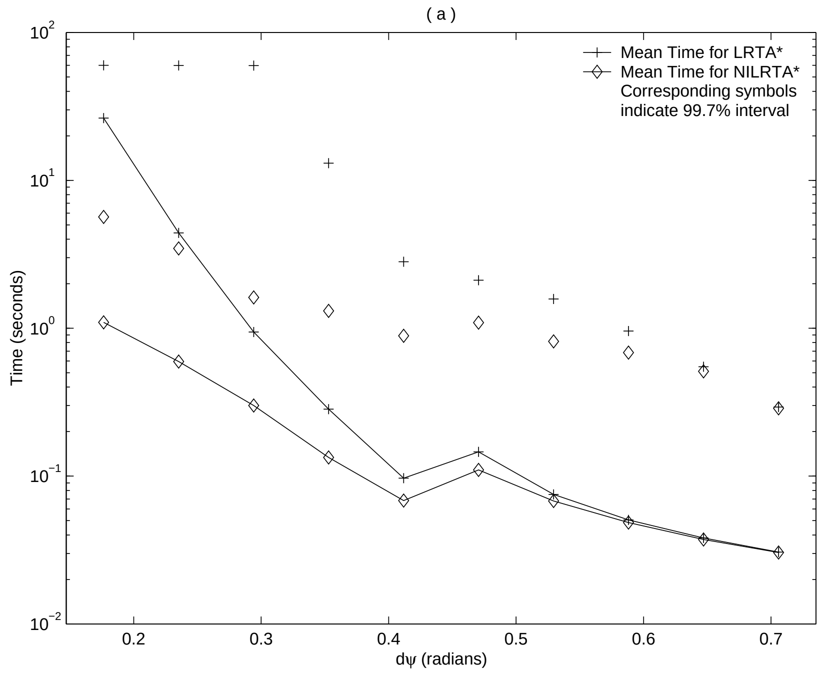


Figure 4.19: LRTA* time comparison for increasing step sizes on Tomservo.

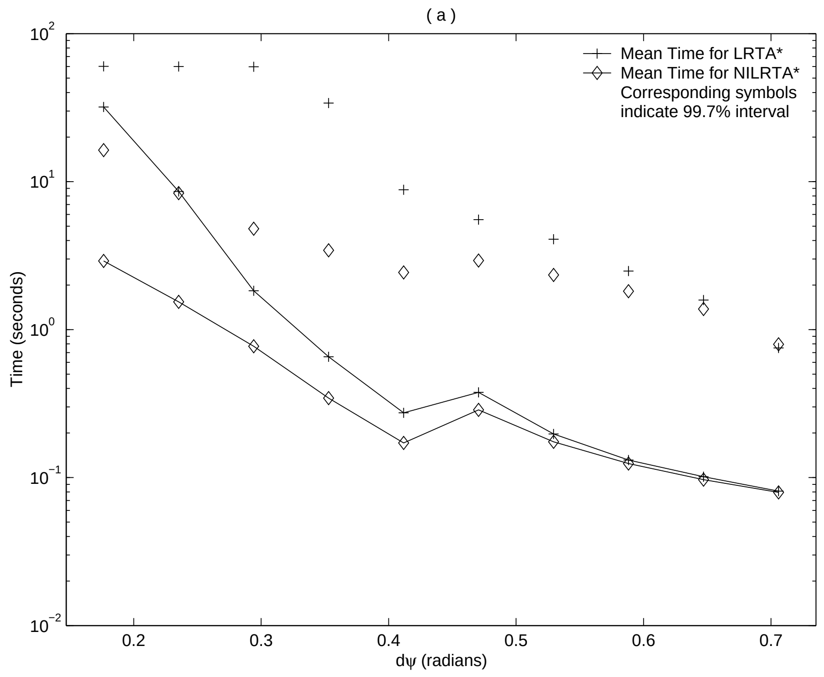


Figure 4.20: LRTA* time comparison for increasing step sizes on Robot 4.

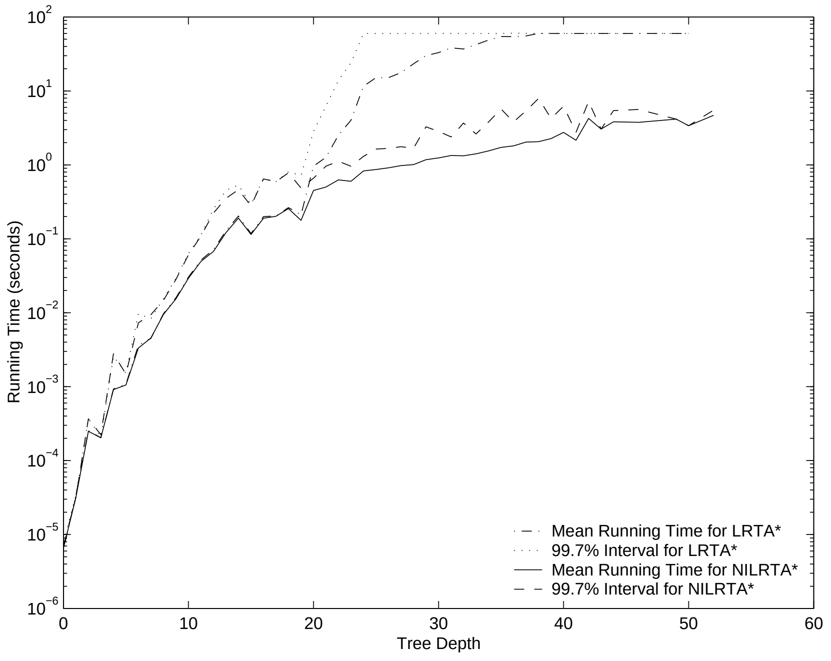


Figure 4.21: LRTA* time comparison for increasing step sizes on Robot 4.

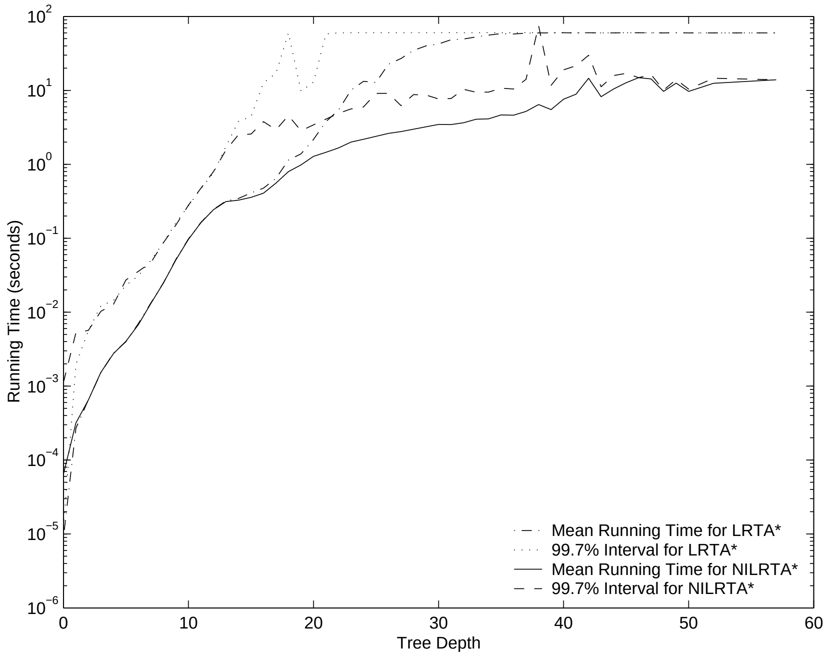


Figure 4.22: LRTA* time comparison for increasing step sizes on Robot 4.

in the path-length performance at smaller step sizes. The speed of the NILRTA* algorithm may be improved further by lowering the run limit for the algorithm. A run limit of 100 iterations has running times of less than 0.03 seconds, but should only be used if finding the shortest path is not an important requirement. If short paths are important, then run limits of 1,000 to 5,000 iterations provide better results.

The potential-field algorithms have fast run times, but do not consistently converge to produce good path lengths. The path-length performance is improved by adding random branching to the algorithms, but doing so approximately triples their running times. The overall performance of the potential-field with random-branching algorithms is acceptable, however, which make these algorithms viable alternatives to the LRTA* and NILRTA* algorithms.

The running-time performance of all the algorithms is affected by the computing platform, but the path-length performance remains nearly the same. The NILRTA* algorithm or either of the potential-field with branching algorithms are viable for use on any computing platform if there is a trade off made between path-length performance and running time.

Chapter 5

Conclusions

This chapter presents the results of this thesis and provides recommendations for future work. The primary focus of this thesis is planning paths that fully utilize the vehicle's sensor to sense a group of closely-spaced targets. To generate an assignment of tasks that best utilizes a team's resources, it is necessary to know the costs incurred by a team member for doing a series of those tasks. For the LOCAAS scenario, tasks generally require that the vehicle's sensor pass over specific target points, which, to produce the associated costs, requires calculating the path the vehicle will take to sense the various targets. Thus the need for the ability to plan paths that sense multiple, closely-spaced targets.

5.1 Summary of Results

This thesis develops the Non-Improving Learning Real-Time A* tree-search algorithm, which provides the ability to plan near-optimal paths for sensing multiple, closely-spaced targets with running times of 0.5 seconds or less. The algorithm utilizes the full sensing capabilities of the vehicle, and in fact, learns how the vehicle should move so as to accomplish its objectives. Furthermore, the flexibility of the NILRTA* path-planning algorithm allows it to be applied to a variety of path-planning problems and vehicle configurations. The Learning Real-Time A* tree-search algorithm possesses the same functionality and flexibility as the NILRTA* algorithm, but produces minimal-length discrete-step paths at the expense of increased running times. The difference in path-length performance between the two algorithms, however, is small, making the NILRTA* algorithm the best solution to the path-planning problem.

The single and multi-source potential-field algorithms are fast, but have poor path-length performance as compared to the NILRTA* algorithm. Neither potential-field methods are provably optimal, and only the single-source potential-field algorithm is shown to always converge. Also, these methods can not directly incorporate different sensor configurations or additional constraints on the path. In general, the potential-field algorithms are useful only for quickly producing some non-optimal path that senses a group of targets.

The single and multi-source with branching algorithms exhibit improved path length performance over the pure potential-field algorithms, but they too are not guaranteed to be optimal, nor are they flexible and extensible. However, either algorithm is a viable solution to the path-planning problem if speed is more important than path-length optimality.

This thesis shows that finding the optimal solution to the path-planning problem is difficult, and that some path-length performance must be sacrificed to allow the generation of good paths in a sufficiently short amount of time. The path-planning methods developed herein range from fast speed and poor paths to slow speed and optimal paths. In general the following maxim holds: if a shorter path is needed, then the time must be spent to find it. Of the six path-planning methods, the NILRTA* tree search algorithm provides a satisfactory trade off between path-length optimality and running time, and is therefore the best solution to the path-planning problem.

5.2 Future Work

This thesis provides a foundation upon which various path planning and assignment algorithms may be developed. There are several directions for future development to incorporate the NILRTA* tree searched into a larger assignment process. More specifically, these directions are to

- Speed up the NILRTA* tree search,
- Incorporate additional constraints and goals, and
- Develop an assignment process that uses the NILRTA* tree-search algorithm.

The speed requirement for the NILRTA* tree search is difficult to quantify as it depends on how the algorithm is incorporated into the larger assignment process. However, methods exist for improving the efficiency of the LRTA* search [35, 36, 37], and these methods should be explored to see if they may be applied to the NILRTA* tree-search algorithm. A large portion of the running time is spent confirming the best path is indeed the optimum, and methods may exist for speeding up this confirmation phase.

The discrete-step path tree and the NILRTA* tree-search algorithm have the flexibility to incorporate additional constraints on, and goals for, the path. The motivating problem for this thesis is sensing a group of closely-spaced targets. The sensor views an area directly beneath the vehicle and there are no constraints on the directions from which the targets are viewed. Since the NILRTA* algorithm learns how the vehicle should move through the path tree such that the sensor passes over the targets, any sensor configuration may be used. Additional constraints may also be added to the path-planning method, such as requiring targets to be sensed from certain directions. Adding constraints of this type is only a matter of detecting those branches of the tree that are invalid, and pruning them out.

The NILRTA* tree-search algorithm may be modified a number of ways to aid in producing an assignment. For example, a vehicle may be given a set of tasks and asked to choose the two of three tasks that they can most easily do. The NILRTA* algorithm finds which branches of the path tree do only two or three tasks and then reports those tasks along with their completion times back to the assignment process. Also, since the NILRTA* algorithm supplies the order in which the vehicle will visit the target, this information may be directly used by the assignment process. In general, the NILRTA* algorithm provides a flexible base for developing cooperative-assignment algorithms.

Appendix A

Code Listing

```
////////////////////////////////////////////////////////////////
//
//   File: knot.h
//
//   Requires: globals.h
//
//   Description: Provides the class CKnot that is used for the
//     LRTA* search path-planning method.
//
//   History:
//   Created and Debugged      11/1/02    JKH
//
////////////////////////////////////////////////////////////////
#ifndef CLASS_CKNOT_H
#define CLASS_CKNOT_H

#include "globals.h"

static const int CHILD_CNT = 3;

typedef char kType;
static const kType ROOT = -1;
static const kType LEFT = 0;
static const kType RIGHT = 1;
static const kType STRAIGHT = 2;
static const kType PARENT = 3;

class CKnot
{
public:
  CKnot( CKnot* parent,
         kType dir );
  CKnot( float x,
         float y,
         float psi );

  ~CKnot( void );

  CKnot* GreedyPath( void );
  CKnot* LRTA( const int count,
               const int best,
               int& dir,
               unsigned int& delta_h );

  float x( void ) { return m_x; }
  float y( void ) { return m_y; }
  float psi( void ) { return m_psi; }
  kType type( void ) { return m_type; }
  CKnot* parent( void ) {return m_parent; }
  bool Sensed( int i ) { return m_sensed[i]; }
  bool AllSensed( void ) { return m_AllSensed(); }
  unsigned int heuristic( void ) { return m_h; }
  unsigned int heuristic( const unsigned int dist2me,
                          const unsigned int best );

  void CleanTree( const int count,
                  const int best );
  void Write( fstream& file );
  void WriteAll( fstream& file );

private:
  CKnot* m_parent;
  CKnot* m_children[CHILD_CNT];

  float m_x;
  float m_y;
  float m_psi;

  kType m_type;

  unsigned int m_h;

  bool* m_sensed;

#ifdef PLOT_PATH
  bool m_iSense;
#endif
```

```
  void m_Terminate( void );
  int m_ChooseDir( const unsigned int h[4],
                   const int nCount );
  void m_Initialize( void );
  void m_InitializeHeuristic( void );
  void m_CreateChildren( void );
  void m_DeleteChildren( void );
  void m_SenseTargets( void );
  bool m_AllSensed( void );
  int m_GetNearestTarget( void );
};

#endif  //#ifndef CLASS_CKNOT_H



////////////////////////////////////////////////////////////////
//
//   File: knot.cpp
//
//   Requires: knot.h
//
//   Description: Implementation of the class CKnot that is used
//     for the LRTA* search path-planning method.
//
//   History:
//   Created and Debugged      11/1/02    JKH
//
////////////////////////////////////////////////////////////////

#include "knot.h"

float** TARGETS;
int TARGET_CNT;
double STEP;
double DTHETA_MAX;
unsigned long total;

CKnot::CKnot( CKnot* parent,
              kType type )
              : m_parent( parent ),
              m_type(type)
{
  total++;
  switch( m_type )
  {
  case LEFT:
  {
    m_psi = m_parent->psi() + DTHETA_MAX;
    double d = sqrt(2*Rt*Rt*(1-cos(DTHETA_MAX)));
    double angle = m_parent->psi() + DTHETA_MAX/2.0;
    m_x = d*cos(angle) + m_parent->x();
    m_y = d*sin(angle) + m_parent->y();
    break;
  }
  case RIGHT:
  {
    m_psi = m_parent->psi() - DTHETA_MAX;
    double d = sqrt(2*Rt*Rt*(1-cos(DTHETA_MAX)));
    double angle = m_parent->psi() - DTHETA_MAX/2.0;
    m_x = d*cos(angle) + m_parent->x();
    m_y = d*sin(angle) + m_parent->y();
    break;
  }
  default:
    m_psi = m_parent->psi();
    m_x = STEP*cos(m_psi) + m_parent->x();
    m_y = STEP*sin(m_psi) + m_parent->y();
    break;
  }
  m_Initialize();
  m_SenseTargets();
  m_InitializeHeuristic();
}

CKnot::CKnot( float x,
              float y,
              float psi )
              : m_parent(NULL),
              m_x(x),
              m_y(y),
              m_psi(psi),
              m_type(ROOT)
{
  total++;
  m_Initialize();
  m_SenseTargets();
  m_InitializeHeuristic();
}
```

```
CKnot::~CKnot( void )
{
  m_DeleteChildren();
}

////////////////////////////////////////////////////////////////
//
//    PATH FUNCTIONS
//
////////////////////////////////////////////////////////////////

CKnot* CKnot::LRTA( const int count,
                    const int best,
                    int& moveDir,
                    unsigned int& delta_h )
{
  delta_h = 1;

  if( m_h == uiINF )
  {
    return NULL;
  }
  if( count >= best )
  {
    m_Terminate();
    return NULL;
  }

  if( m_children[0] == NULL ) m_CreateChildren();

  unsigned int bestDist = best*static_cast<unsigned int>( STEP );
  unsigned int dist2me = count*static_cast<unsigned int>( STEP );
  unsigned int h[4] = {uiINF, uiINF, uiINF, uiINF};

  int nCount = 0;
  while( m_children[nCount] != NULL && nCount < CHILD_CNT )
  {
    h[nCount] = m_children[nCount]->heuristic(dist2me,bestDist);
    nCount++;
  }

#ifdef USE_PARENT
  if( m_parent )
    h[nCount] = m_parent->heuristic();
  else
    h[nCount] = uiINF;
#else
  nCount--;
#endif

  bool terminate = true;
  for( int i = 0; i <= nCount; i++ )
    terminate = terminate && (h[i] == uiINF);

  if( terminate )
  {
    m_Terminate();
    moveDir = 1;
    return NULL;
  }

  int dir = m_ChooseDir( h, nCount );

  int oldh = m_h;
  m_h = static_cast<unsigned int>(STEP) + h[dir];
  int newh = m_h;

  delta_h = abs( newh - oldh );

#ifdef USE_PARENT
  if( dir == nCount )
  {
    moveDir = -1;
    return m_parent;
  }
  else
#endif
  {
    moveDir = 1;
    return m_children[dir];
  }
}

CKnot* CKnot::GreedyPath( void )
{
  if( m_children[0] == NULL ) m_CreateChildren();

  int target = m_GetNearestTarget();
```

```
float ptX = TARGETS[target][0];
float ptY = TARGETS[target][1];

// Move to the child that is closest to the target.
int index = 0;
int bestIndex = 0;
int worstIndex = 0;

float dist;
float best = fINF);
float worst = 0;
float dx;
float dy;

while( m_children[index] != NULL && index < CHILD_CNT )
{
  dx = ptX - m_children[index]->x();
  dy = ptY - m_children[index]->y();

  dist = norm( dx, dy );
  if( dist < best )
  {
    best = dist;
    bestIndex = index;
  }
  if( dist > worst )
  {
    worst = dist;
    worstIndex = index;
  }
  index++;
}

float turnDir;
switch( m_children[bestIndex]->type() )
{
  case LEFT:
    turnDir = -1;
    break;

  case RIGHT:
    turnDir = 1;
    break;

  default:
      turnDir = 0;
      break;
  }

  // see if the point is inside the desired turning circle. If
  // the point is inside the desired turning circle, then we
  // want to turn the opposite direction.

  float psi = m_children[bestIndex]->psi();
  float xc = m_children[bestIndex]->x() + turnDir*Rt*sin(psi);
  float yc = m_children[bestIndex]->y() - turnDir*Rt*cos(psi);

  if( norm( (xc-ptX), (yc-ptY) ) < (Rt-posYoffset) )
  {
    return m_children[worstIndex];
  }
  else
    return m_children[bestIndex];
}
////////////////////////////////////////////////////////////////
//
//  PUBLIC ACCESS FUNCTIONS
//
////////////////////////////////////////////////////////////////

inline unsigned int CKnot::heuristic( const unsigned int dist2me,
                                      const unsigned int best )
{
  if( (m_h + dist2me-STEP) > best && m_h < uiINF )
  {
    m_Terminate();
  }
  return m_h;
}

////////////////////////////////////////////////////////////////
//
//  PUBLIC UTILITY FUNCTIONS
//
////////////////////////////////////////////////////////////////

void CKnot::CleanTree( const int count,
                       const int best )
{
  if( count > best )
  {
    m_DeleteChildren();
    if( !m_AllSensed() )
      m_h = uiINF;
    return;
  }

  bool terminate = false;
  bool hasChildren = false;
  for( int i = 0; i < CHILD_CNT; i++ )
  {
    if( m_children[i] )
    {
      m_children[i]->CleanTree( count + 1, best );
      terminate = terminate&&(m_children[i]->heuristic() == uiINF);
      hasChildren = true;
    }
  }

  if( terminate && hasChildren )
  {
    m_Terminate();
  }
}

void CKnot::Write( fstream& file )
{
#ifndef PLOT_PATH
  file << m_x << "," << m_y << "," << m_psi << "," << m_h;
#else
  file << "," << m_iSense << endl;
#endif
  file << endl;

  if( m_parent )
    m_parent->Write( file );
  else
    return;
}

void CKnot::WriteAll( fstream& file )
{
  file << m_x << "," << m_y << endl;

  for( int i = 0; i < 3; i++ )
  {
    if( m_children[i] )
      m_children[i]->WriteAll( file );
  }
}

////////////////////////////////////////////////////////////////
//
//  PRIVATE CLASS HELPER FUNCTIONS
//
////////////////////////////////////////////////////////////////

int CKnot::m_ChooseDir(const unsigned int h[4],
                       const int nCount)
{
  unsigned int best = uiINF;
  unsigned int next = uiINF;
  unsigned int last = uiINF;
  int index1 = 0;
  int index2 = 0;
  int index3 = 0;

  for( int i = 0; i <= nCount; i++ )
  {
    if( h[i] <= best )
    {
      last = next;
      index3 = index2;
      next = best;
      index2 = index1;
      best = h[i];
      index1 = i;
    }
  }

  double value = static_cast<double>(rand())/
                 static_cast<double>(RAND_MAX);
  int dir = index1;;

  // Randomly break any tie between the three best.
  if( best == last && best == next )
  {
    if( value < .333333333 )
      dir = index3;
    else if( value > .66666666)
```

```
      dir = index2;
    else
      dir = index1;
  }
  else if( next == best ) // Randomly break any tie
  {
    if( value > .5 )
      dir = index2;
    else
      dir = index1;
  }

  return dir;
}

// Finds distance to the closest target and then the distance to
// the target furthest from the closest.
void CKnot::m_InitializeHeuristic( void )
{
  double dist;
  double best = dINF;
  int bestIndex = -1;

  if( m_AllSensed() )
  {
    m_h = 0;
    return;
  }

  for( int i = 0; i < TARGET_CNT; i++ )
  {
    if( !m_sensed[i] )
    {
      dist = norm(m_x-TARGETS[i][0], m_y-TARGETS[i][1]);

      if( dist < best )
      {
        bestIndex = i;
        best = dist;
      }
    }
  }

  double nextbest = 0;
  int nextIndex = -1;
  for( int i = 0; i < TARGET_CNT; i++ )
  {
    if( !m_sensed[i] )
    {
      dist = norm(TARGETS[bestIndex][0]-TARGETS[i][0],
                  TARGETS[bestIndex][1]-TARGETS[i][1]);

      if( dist > nextbest )
      {
        nextIndex = i;
        nextbest = dist;
      }
    }
  }

  if( nextIndex >= 0 )
  {
    dist = norm(m_x-TARGETS[nextIndex][0],
                m_y-TARGETS[nextIndex][1] );

    if( dist <posYoffset+STEP ) dist = 0;
    else dist -=  posYoffset+STEP;
    if( nextbest < 2*(posYoffset+STEP) ) nextbest = 0;
    else nextbest -= 2*(posYoffset+STEP);

    m_h = static_cast<unsigned int>( dist + nextbest );
  }
  else
  {
    if( best < posYoffset ) best = 0;
    else best -= posYoffset;

    m_h = static_cast<unsigned int>( best );
  }
}

bool CKnot::m_AllSensed( void )
{
  bool b = true;
  for( int i = 0; i < TARGET_CNT; i++ )
    if( !m_sensed[i] )
    {
      b = false;
      break;
    }
  if( b ) m_h = 0;
  return b;
}

void CKnot::m_SenseTargets( void )
{
  for( int j = 0; j < TARGET_CNT; j++ )
  {
    if( !m_sensed[j] )
    {
      double dx = TARGETS[j][0] - m_x;
      double dy = TARGETS[j][1] - m_y;

      // Rotate the vector to the target by the vehicle's heading
      // using a Given's rotation matrix. This puts the target
      // into the vehicle's local coordinate frame.
      double cosPSI = cos(m_psi);
      double sinPSI = sin(m_psi);

      double xtest = (dx*cosPSI + dy*sinPSI);
      double ytest = (dy*cosPSI - dx*sinPSI);

      //Test to see if the target is inside the sensor footprint
      if( xtest <= posXoffset && xtest >= negXoffset &&
         ytest <= posYoffset && ytest >= negYoffset )
      {
        m_sensed[j] = true;

#ifdef PLOT_PATH
        m_iSense = true;
#endif
      }
    }
  }
}

int CKnot::m_GetNearestTarget( void )
{
  double closest = dINF;
  int target = -1;
  for( int j = 0; j < TARGET_CNT; j++ )
  {
    if( !m_sensed[j] )
    {
      double dist2tar = norm( TARGETS[j][0] - m_x,
                              TARGETS[j][1] - m_y );
      if( dist2tar < closest )
      {
        closest = dist2tar;
        target = j;
      }
    }
  }
  return target;
}

void CKnot::m_Initialize( void )
{
#ifdef PLOT_PATH
  m_iSense = false;
#endif

  int i;

  for( i = 0; i < CHILD_CNT; i++ )
    m_children[i] = NULL;

  m_sensed = new bool[TARGET_CNT];
    if( m_parent )
  {
    for( int i = 0; i < TARGET_CNT; i++ )
      m_sensed[i] = m_parent->Sensed(i);
  }
  else
  {
    for( int i = 0; i < TARGET_CNT; i++ )
      m_sensed[i] = false;
  }
}

void CKnot::m_CreateChildren( void )
{
  if( STEP < STEP_TRANS )
  {
    kType parentType;
    if( m_parent )
      parentType = m_parent->type();
    else
      parentType = ROOT;
    bool left = true;
    bool right = true;

      if( m_type == LEFT || parentType == LEFT ) right = false;
      if( m_type == RIGHT || parentType == RIGHT ) left = false;

    int index = 0;
    if( left )
      m_children[index++] = new CKnot( this, LEFT );

    if( right )
      m_children[index++] = new CKnot( this, RIGHT );

    m_children[index++] = new CKnot( this, STRAIGHT );
  }
  else
  {
    m_children[0] = new CKnot( this, LEFT );
    m_children[1] = new CKnot( this, RIGHT );
    m_children[2] = new CKnot( this, STRAIGHT );
  }

}

inline void CKnot::m_Terminate( void )
{
  m_DeleteChildren();
  m_h = uiINF;
}

void CKnot::m_DeleteChildren( void )
{
  for( int i = 0; i < CHILD_CNT; i++ )
  {
    if( m_children[i] )
      delete m_children[i];
    m_children[i] = NULL;
  }
}


////////////////////////////////////////////////////////////////
//
//   File: planner.h
```

```
//
//   Requires: knot.h
//
//   Description: Provides the class CPlanner that implements
//    the LRTA* tree-search path-planning method. The search
//    operates according to the mode and parameters specified
//    by the call to PlanPath. PlanPath returns a pointer to
//    a vector of the path waypoints.
//
//   History:
//   Created and Debugged      11/1/02    JKH
//
////////////////////////////////////////////////////////////////
#include "knot.h"

// Termination modes for the path planner.
typedef int kMode;
static const kMode TIME_LIMIT = 0;
static const kMode DELTA_H_0 = 1;
static const kMode RUN_LIMIT = 2;

class CPlanner
{
  friend class CKnot;

  public:
  CPlanner( double STEP );
  ~CPlanner( void );

  float* PlanPath( const float x0,
                   const float y0,
                   const float psi0,
                   const int targetCnt,
                   float** const targets,
                   const kMode = RUN_LIMIT,
                   const int timeLimit = 60,
                   const int runLimit = 10000 );

  private:
  const double m_STEP;
  CKnot* m_endKnot;

  float* m_MakePoints( const int count );
};
```

```
///////////////////////////////////////////////////////////////
//
//   File: planner.cpp
//
//   Requires: planner.h
//
//   Description: Implements the class CPlanner that implements
//     the LRTA* path-planning method.
//
//   History:
//   Created and Debugged      11/1/02    JKH
//
///////////////////////////////////////////////////////////////
#include "planner.h"

extern float** TARGETS;
extern double STEP;
extern double DTHETA_MAX;
extern unsigned long total;
extern int TARGET_CNT;

extern int FILE_NUM;

CPlanner::CPlanner( double step )
{
  STEP = step;
  DTHETA_MAX = STEP/Rt;

  srand( (unsigned)time( NULL ) );
}

float* CPlanner::PlanPath( const float x0,
                           const float y0,
                           const float psi0,
                           const int targetCnt,
                           float** const targets,
                           const kMode mode,
                           const int timeLimit,
                           const int runLimit )
{
  TARGETS = targets;
  TARGET_CNT = targetCnt;

  CKnot* root = new CKnot( x0, y0, psi0 );
  CKnot* currentKnot = root;
```

```
unsigned int count = 0;
unsigned int best;
int dir = 1;

TIME_REC(fstream tfile( "nitime.txt", ios::out|ios::app ));

TIMING(double time = getTime());

while( !currentKnot->AllSensed() && count < MAX_DEPTH )
{
  currentKnot = currentKnot->GreedyPath();
  count++;
}
m_endKnot = currentKnot;

TERM_MSG(cerr << "All targets Sensed at " << count << endl);

best = count;

int runCount = 0;
int currentRunCnt = 0;

TIME_REC(double bestTime = getTime() - time);
TIME_REC(double cleanTime = 0);

unsigned int totalDeltaH = 0;

bool run = true;
while( run )
{

  runCount++;
  currentRunCnt++;

  count = 0;
  currentKnot = root;
  bool allSensed = false;
  totalDeltaH = 0;
  unsigned int delta_h = 0;

  while( !allSensed && currentKnot != NULL && count < best  )
  {
    currentKnot = currentKnot->LRTA(count, best, dir, delta_h);
    totalDeltaH += delta_h;
```

```
      if( currentKnot) allSensed = currentKnot->AllSensed();
      count += dir;
    }

    if( allSensed )
    {
      if( count < best )
      {
        m_endKnot = currentKnot;
        best = count;
        currentRunCnt = 0;

        TIME_REC(bestTime = getTime() - time);

#ifdef CLEAN_TREE
        TIME_REC(double ctime = getTime());
        root->CleanTree( 0, best );
        TIME_REC(cleanTime += getTime() - ctime);
#endif

      TERM_MSG(cerr << "All targets Sensed at " << best
                    << "  Time = " << getTime() - time
                    << "  Iteration " << runCount << endl);
      }
    }
    switch( mode )
    {
      case TIME_LIMIT:
        run = (totalDeltaH > 0)&&((getTime() - time)<timeLimit);
        break;
      case DELTA_H_0:
        run = totalDeltaH > 0;
        break;
      case RUN_LIMIT:
        run = (totalDeltaH > 0) && (currentRunCnt < runLimit);
        break;
    }
  }  // while( run )


  TIME_REC(
    double totaltime = getTime() - time;
    tfile <<  best*STEP << "," << bestTime << "," << runCount
          << "," << totaltime << "," << totalDeltaH << ","
          << cleanTime << "," << total <<"," << FILE_NUM << endl;
    tfile.close()
    );

#ifdef CLEAN_TREE
  STATS(cerr << "Cleaning time = " << cleanTime << endl);
#endif
  STATS(cerr << "Delta h = " << totalDeltaH << " after "
      << runCount << " iterations." << endl);

  STATS(cerr << "Expanded " << total <<  " knots." << endl);
  STATS(cerr << "Best Length = " << best*STEP << " feet  ("
             << best << " knots)" << endl);

#ifdef PLOT_PATH
  fstream file("lrtaknots.txt", ios::out);
  m_endKnot->Write( file );
#endif
  float* points = m_MakePoints( best );
        delete root;
  return points;
}

float* CPlanner::m_MakePoints( const int count )
{
  float* points = new float[count*2];

  int index = 0;
  CKnot* currentKnot = m_endKnot;
  while( currentKnot )
  {
    points[index] = currentKnot->x();
    points[index+count] = currentKnot->y();

    currentKnot = currentKnot->parent();
  }

  return points;
}


////////////////////////////////////////////////////////////////
//
//   File: globals.h
//
//   Requires: math.h, vector, fstream, iostream, time.h,
//             sys/time.h, limits
//
//   Description: Provides global control parameters, constants,
//      and functions for the classes CKnot and CPlanner.
//
//   History:
//   Created and Debugged      11/1/02    JKH
//
////////////////////////////////////////////////////////////////
#ifndef MY_PRM_GLOBALS_H
#define MY_PRM_GLOBALS_H

////////////////////////////////////////////////////////////////
//
// Global Includes
//
////////////////////////////////////////////////////////////////
#include <math.h>
#include <limits>

#include <vector>
using std::vector;

#include <stdio.h>

#include <fstream>
using std::fstream;
using std::ios;
using std::ios;
using std::endl;

#include <iostream>
using std::cout;
using std::cerr;

#include <sys/time.h>
#include <time.h>


////////////////////////////////////////////////////////////////
//
// Global Control Parameters
//
////////////////////////////////////////////////////////////////
```

```
#define DEBUG(x)// x     //outputs debug messages
#define TIMING(x) x      //times algorithm performance
#define TIME_REC(x) x    //Records running times to file
#define TERM_MSG(x)// x  //Termination messages for the algorithm
#define STATS(x)// x     //Statistical information about  the run

//#define PLOT_PATH      //Writes path points to a file
//#define CLEAN_TREE     //Enable to clean the tree
//#define USE_PARENT     //Enables moving to parent nodes

// Sensor footprint Geometry
//width in feet along the y-axis when angle = 0
static const double sensWidth = 2000;

//depth in feet along the x-axis when angle = 0
static const double sensDepth = 820;
static const double posXoffset = sensDepth/2;     // feet
static const double negXoffset = posXoffset - sensDepth;  // feet
static const double posYoffset = sensWidth/2;     // feet
static const double negYoffset = posYoffset - sensWidth;  // feet

// Minium turning radius of the vehicle in feet
static const double Rt = 1700;

// Step size above which the full path tree is used
static const double STEP_TRANS = 800;

// Maximum search depth (maximum path length) (Steps)
static const unsigned int MAX_DEPTH = 500;


////////////////////////////////////////////////////////////////
//
// Global Constants
//
////////////////////////////////////////////////////////////////

// Big number to represent infinity
static const unsigned int uiINF =
       std::numeric_limits<unsigned int>::max();
static const float fINF = std::numeric_limits<float>::max();
static const double dINF = std::numeric_limits<double>::max();

// pi
static const double PI = 3.14159265358979;
```

```
/////////////////////////////////////////////////////////////////
//
// Global Functions
//
/////////////////////////////////////////////////////////////////

// Linux specific function to return the current seconds
inline double getTime( void )
{
  struct timeval tp;
  gettimeofday( &tp, NULL );
  return( (double)tp.tv_sec + (double)tp.tv_usec*1e-6 );
}

// Returns the 2-norm of the vector [x y]
inline double norm( double x, double y )
{
  return sqrt( x*x + y*y );
}

inline double abs( double x )
{
  if( x < 0 )
    return -x;
  else
    return x;
}

inline unsigned int abs( unsigned int x )
{
  if( x < 0 )
    return -x;
  else
    return x;
}

#endif  //#ifndef GLOBALS_H


/////////////////////////////////////////////////////////////////
//
//   File: main.cpp
//
//   Requires: planner.h
```

```
//
//   Description: Example driver utilizing the CPlanner class
//
//   History:
//   Created and Debugged      11/1/02    JKH
//
////////////////////////////////////////////////////////////////
#include "planner.h"

float** LoadTargets( float& x0,
                     float& y0,
                     float& psi0,
                     int& targetCnt );
int FILE_NUM = 0;

int main( int argc, char* argv[] )
{
  int step = 300;
  int runLimit = 10000;

  if( argc == 3 )
  {
    runLimit = atoi(argv[2]);
    step = atoi(argv[1]);
  }

  if( argc == 2 )
    step = atoi(argv[1]);

  cerr << "Beginning NILRTA...\n";

  float x0;
  float y0;
  float psi0;
  int targetCnt;

  float** targets = LoadTargets( x0, y0, psi0, targetCnt );

  CPlanner planner(step);

  TIMING(double time = getTime());

  planner.PlanPath( x0, y0, psi0, targetCnt, targets, RUN_LIMIT,
                    300, runLimit );
  TIMING(cerr << "Time Lapsed = " << getTime() - time << "\n");
  cerr << "Done...\n\n\n";

  return 0;
}

float**  LoadTargets( float& x0,
                      float& y0,
                      float& psi0,
                      int& targetCnt )
{
  FILE* fp;

  fp = fopen("../targets.txt", "r" );

  fscanf( fp,"%f,%f,%f\n", &x0, &y0, &psi0 );

  fscanf( fp, "%d\n", &targetCnt );

  float** targets = new float*[targetCnt];
  for( int i =0; i < targetCnt; i++ )
    targets[i] = new float[2];

  float x = 0;
  float y = 0;
  for( int i = 0; i < targetCnt; i++ )
  {
    fscanf( fp, "%f,%f\n", &x, &y );
    targets[i][0] = (float) x;
    targets[i][1] = (float) y;
  }

  fscanf( fp, "%d\n", &FILE_NUM );
  fclose(fp);
  return targets;
}
```

The code in the above listing requires the file `targets.txt` as input. Several files are provided as output depending on the parameters specified in the file `globals.h`. The file `lrtaknots.txt` contains the path information for the optimal path, and performance information is provided in the file `ltime.txt`. Formats for these files are as follows:

`targets.txt`:

> *vehicle* x, *vehicle* y, *vehicle* ψ
> *number of targets*
> *target* x_1, *target* y_1
> $\vdots$
> *target* x_n, *target* y_n
> *target set number*

`lrtaknots.txt`:

> *point* x_{end}, *point* y_{end}, *heading*$_{end}$, *heuristic*$_{end}$, *flag*$_{end}$
> $\vdots$
> *point* x_0, *point* y_0, *heading*$_0$, *heuristic*$_0$, *flag*$_0$

Flag is true if a target was sensed at the location, false otherwise.

`ltime.txt`:

> *length, best time, iterations, total time,* Δh, *clean time, total nodes, target set*

Bibliography

[1] Marios M. Polycarpou, Yanli Yang, and Kevin M. Passino, "A cooperative search framework for distributed agents", in *Proc. IEEE Int. Symp. on Intelligent Control*, 2001, pp. 1–6.

[2] Phillip R. Chandler, Meir Pachter, and Steve Rasmussen, "UAV cooperative control", Tech. Rep., AFRL/VACA WPAFB, Dayton, Ohio, 2001.

[3] Darbha Swaroop, "A method of cooperative classification and atack for LOCAAS vehicles", Tech. Rep., AFRL/VACA WPAFB, Dayton, Ohio, 2000.

[4] Darbha Swaroop, "Teaming strategies for a resource allocation and coordination problem in the cooperative control of UAVs", Tech. Rep., AFRL/VACA WPAFB, Dayton, Ohio, 2001.

[5] Jason K. Howlett, "Path planning and cooperative assignment", Tech. Rep., AFRL/VACA WPAFB, Dayton, Ohio, 2001.

[6] Jeffrey M. Fowler, "Coupled task planning for multiple unmanned air vehicles", Tech. Rep., AFRL/VACA WPAFB, Dayton, Ohio, 2001.

[7] Kendall E. Nygard, "Hierarchical cooperative control and resource allocation for UAVs", Tech. Rep., AFRL/VACA WPAFB, Dayton, Ohio, 2000.

[8] Corey Schumacher, Phillip R. Chandler, and Steven R. Rasmussen, "Task allocation for wide area search muniions via network flow optimization", in *Proc. AIAA Guidance, Navigation, and Control Conference*, August 2001.

[9] Phillip R. Chandler and Meir Pachter, "Hierarchical control for autonomous teams", in *Proc. AIAA Guidance, Navigation, and Control Conference*, August 2001.

[10] Kendall E. Nygard, Phillip R. Chandler, and Meir Pachter, "Dynamic network flow optimization models for air vehicle resource allocation", in *Proc. of American Control Conf.*, June 2001, pp. 1853–1856.

[11] L.E. Dubins, "On curves of minimal length with a constraint on average curvature and with prescribed initial and terminal positions and tangents", *American J. of Math*, vol. 79, pp. 497–516, 1957.

[12] Phillip R. Chandler, Meir Pachter, Dharba Swaroop, Jeffrey M. Fowler, Jason K. Howlett, Steven Rasmussen, Corey Schumacher, and Kendall Nygard, "Complexity in UAV cooperative control", in *Proc. of American Control Conf.*, June 2002.

[13] Guang Yang and Vikram Kapila, "Optimal path planning for unmanned air vehicles with kinematic and tactical constraints", Tech. Rep., Department of Mechanical, Aerospace, and Manufacturing Engineering, Polytechnic University, Brooklyn, NY 11201, Feburary 2002.

[14] Kevin B. Judd, "Trajectory planning strategies for unmanned air vehicles", Master's thesis, Brigham Young University, Provo, Utah, 2001.

[15] Erik P. Anderson, "Extremal control and unmanned air vehicle trajectory generation", Master's thesis, Brigham Young University, Provo, Utah, 2002.

[16] Timothy W. McLain and Randal W. Beard, "Trajectory planning for coordinated rendezvous of unmanned air vehicles", in *Proc. AIAA Guidance, Navigation, and Control Conference*, Denver, CO., August 2000.

[17] Emilio Frazzoli, Munther A. Dahleh, and Eric Feron, "Real-time motion planning for agile autonomous vehicles", *Journal of Guidance, Control, and Dynamics*, vol. 25, no. 1, Janurary-Feburary 2002.

[18] Lydia E. Kavraki, Petr Švestka, Jean-Claude Latombe, and Mark H. Overmars, "Probabilistic roadmaps for path planning high-dimensional configuration spaces", *IEEE Trans. on Robotics and Automation*, vol. 12, no. 4, August 1996.

[19] Lydia E. Kavraki, Mihail N. Kolountzakis, and Jean-Claude Latombe, "Analysis of probabilistic roadmaps for path planning", *IEEE Trans. on Robotics and Automation*, vol. 14, no. 1, Feburary 1998.

[20] Robert Bohlin and Lydia E. Kavraki, "Path planning using lazy prm", in *Proc. IEEE Conf. on Robotics and Automation*, San Fancisco, CA, April 2000.

[21] Andrew Ladd and Lydia E. Kavraki, "Generalizing the analysis of prm", in *Proc. IEEE Conf. on Robotics and Automation*, Washington, DC, May 2002.

[22] Petr Švestka and Mark H. Overmars, "Coordinated motion planning for multiple car-like robots using probabilistic roadmaps", in *Proc. IEEE Conf. on Robotics and Automation*, 1995.

[23] Mark H. Overmars and Petr Švestka, "A paradigm for probabilistic path planning", Tech. Rep., Department of Computer Science, Utrecht University, March 1996.

[24] Nadeem Faiz, Sunil K. Agrawal, and Richard M. Murray, "Trajectory planning of differentially flat systems", *Journal of Guidance, Control, and Dynamics*, vol. 24, no. 2, March-April 2001.

[25] Michiel J. van Nieuwstadt and Richard M. Murray, "Real time trajectory generation for differentially flat systems", *Int. J. Robust and Nonlinear Contr.*, vol. 8, no. 11, pp. 995–1020, 1998.

[26] Pankaj K. Agarwal and Hongyan Wang, "Approximation algorithms for curvature-constrained shortest paths", *Siam Journal of Computing*, vol. 30, no. 6, pp. 1739–1772, 2001.

[27] Christopher I. Connolly and Roderic A. Grupen, "The application of harmonic functions to robotics", *Journal of Robotic Systems*, vol. 10, no. 7, pp. 931–946, 1992.

[28] F. Lamiraux and J. Laumond, "On the expected complexity of random path planning", in *Proc. IEEE Conf. on Robotics and Automation*, 1996, pp. 3014–3019.

[29] J. S. Zelek, "Dynamic path planning", Tech. Rep., Centre for Intelligent Machines and Dept. of Electrical Engineering, McGill University, 1995.

[30] Yoram Koren and Johann Borenstein, "Potential field methods and their inherent limitations for mobile robot navigation", in *Proc. IEEE Conf. on Robotics and Automation*, Sacramento, CA., April 1991.

[31] R. E. Korf, "Real-time heurisitic search", *Artificial Intelligence*, vol. 42, no. 2-3, pp. 189–211, 1990.

[32] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh, "Learning to act using real-time dynamic programming", *Artificial Intelligence*, vol. 72, no. 1, pp. 81–138, 1995.

[33] Toru Ishida, "Real-time search for autonomous agents and multiagent systems", *Autonomous Agents and Multi-Agent Systems*, vol. 1, no. 2, pp. 139–167, 1998.

[34] Gerhard Weiss, Ed., *Multiagent Systems*, chapter 4, pp. 165–199, The MIT Press, 2000.

[35] Stefan Edelkamp and Jürgen Eckerle, "New strategies in real-time heuristic search", in *AAAI-97 Workshop on On-Line Search*. 1997, pp. 30–35, AAAI Press.

[36] David Furcy and Sven Koenig, "Speeding up the convergence of real-time search", in *Proc. of the Nat. Conf. on Artifical Intelligence*, 2000, pp. 891–897.

[37] Toru Ishida and Masashi Shimbo, "Improving the learning efficiencies of realtime search", in *Proc. of AAAI-96*, 1996, pp. 305–310.

[38] Wynn C. Stirling, Michael A. Goodrich, and Dennis Packard, "Satisficing equilibria - a non-classical theory of games and decisions".

[39] Michael A. Goodrich, "A satisficing approach to assinging vehicles to targets", Tech. Rep., Department of Computer Science, Brigham Young University, February 2001.

[40] Arthur Richards, Johnathan How, Tom Schouwenaars, and Eric Feron, "Plume avoidance maneuver planning using mixed integer linear programming", in *Proc. AIAA Guidance, Navigation, and Control Conference*, August 2001.

[41] Tom Schouwenaars, Bart De Moor, Eric Feron, and Jonathan How, "Mixed integer programming for multi-vehicle path planning", in *European Control Conference*, 2001.

[42] Arthur Richards, John Bellingham, Michael Tillerson, and Jonathan How, "Coordination and control of multiple UAVs", in *Proc. AIAA Guidance, Navigation, and Control Conference*, 2002.

[43] Shlomo Zilberstein and Stuart J. Russell, "Anytime sensing, planning and action: A practical model for robot control", in *Proc. of the 13th Int. Joint Conf. on Artificial Intelligence*, Chambery France, 1993, pp. 1402–1407.

[44] Eric A. Hansen, Shlomo Zilberstein, and Victor A. Danilchenko, "Anytime heuristic search: First results", Tech. Rep. 97-50, Computer Science Department, University of Massachusetts, 1997.

[45] Shlomo Zilberstein, "Using anytime algorithms in intelligent systems", *AI Magazine*, vol. 17, no. 3, pp. 73–83, 1996.