

2009

Intelligent path planning with evolutionary computation

Adam Christopher Parry
University of Dayton

Follow this and additional works at: https://ecommons.udayton.edu/graduate_theses

Recommended Citation

Parry, Adam Christopher, "Intelligent path planning with evolutionary computation" (2009). *Graduate Theses and Dissertations*. 4847.
https://ecommons.udayton.edu/graduate_theses/4847

This Thesis is brought to you for free and open access by the Theses and Dissertations at eCommons. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of eCommons. For more information, please contact mschlangen1@udayton.edu, ecommons@udayton.edu.

**INTELLIGENT PATH PLANNING WITH EVOLUTIONARY
COMPUTATION**

A Thesis

Submitted to

The School of Engineering of the

UNIVERSITY OF DAYTON

In Partial Fulfillment of the Requirements for

The Degree

Master of Science in Electrical and Computer Engineering

by

Adam Christopher Parry

UNIVERSITY OF DAYTON

Dayton, Ohio

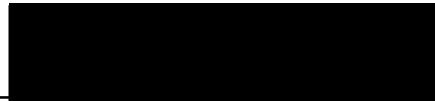
August 2009

INTELLIGENT PATH PLANNING WITH EVOLUTIONARY COMPUTATION

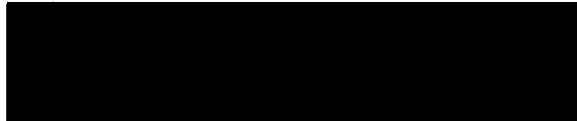
APPROVED BY:



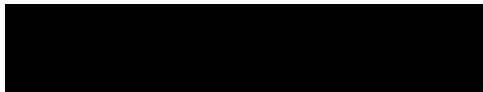
Raúl Ordóñez, Ph.D.
Advisor Committee Chairman
Associate Professor, Electrical and
Computer Engineering



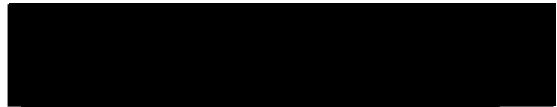
John Loomis, Ph.D.
Committee Member
Associate Professor, Electrical and
Computer Engineering



Daniel Repperger, Ph.D.
Committee Member
Adjunct Professor, Electrical and
Computer Engineering



Malcolm W. Daniels, Ph.D.
Associate Dean
School of Engineering



Joseph E. Saliba, Ph.D., P.E.
Dean
School of Engineering

© Copyright by

Adam Christopher Parry

All rights reserved

2009

ABSTRACT

INTELLIGENT PATH PLANNING WITH EVOLUTIONARY COMPUTATION

Name: Parry, Adam Christopher
University of Dayton

Advisor: Dr. Raúl Ordóñez

Intelligent, autonomous path planning has been the focus of much research over the years. Traditionally, path planning methods can be broken down into two categories: deliberate and reactive. Deliberate path planning methods are generally used as offline mission planners or online mission re-planners. Reactive methods, by their nature, operate completely online, leaving little or no distinction between the planning and execution of a path. Both of these methods offer advantages and disadvantages. The goal of this research is to combine the deliberate and reactive planning methods into a single hybrid method that can make use of knowledge about the scenario while responding to unanticipated events (such as the detection of a new obstacle). The deliberate method was implemented using an evolutionary algorithm and a simple reactive obstacle avoidance scheme was developed. The overall results were mixed, with good performance shown in some scenarios and poor performance in others. Causes of poor performance and potential solutions are discussed.

For my parents for teaching me to think on my own and for my wife for her constant support.

TABLE OF CONTENTS

	Page
Abstract	iii
Dedication	iv
List of Figures	vii
List of Tables	xv
 CHAPTERS:	
I. Introduction and Literature Review	1
1.1 Introduction	1
1.2 Literature Review	2
1.3 Current Work	5
II. Evolutionary Algorithms for Static Path Planning Scenarios	7
2.1 Overall Algorithm Description	7
2.1.1 Solution Representation	8
2.1.2 Fitness Evaluation	10
2.1.3 Mutation Operators	13
2.1.4 Selection Operators	16
2.2 Scenario Descriptions	16
2.3 Simulation Results	21
2.3.1 Scenario 1	21
2.3.2 Scenario 2	26
2.3.3 Scenario 3	31
2.3.4 Scenario 4	34
2.3.5 Scenario 5	37
2.3.6 Scenario 6	40
2.3.7 Scenario 7	43
2.3.8 Scenario 8	45
2.3.9 Discussion and Batch Results	48

III. Evolutionary Algorithms for Dynamic Path Planning Scenarios	62
3.1 Reactive Architecture Design	62
3.1.1 Vehicle Controller	62
3.1.2 Reactive Planner	69
3.2 Algorithm Description	71
3.3 Scenario Descriptions	72
3.4 Simulation Results	77
3.4.1 Scenario1	78
3.4.2 Scenario2	80
3.4.3 Scenario3	82
3.4.4 Scenario4	84
3.4.5 Scenario5	86
3.4.6 Scenario6	88
3.4.7 Scenario7	90
3.4.8 Scenario8	92
3.4.9 Discussion and Batch Results	94
IV. Conclusions	96
4.1 Discussion of Results	96
4.2 Comparison with Previous Work	96
4.3 Future Work	97
Appendices:	
A. Additional Static Scenario Results	99
B. Additional Dynamic Scenario Results	113
C. Code Listings	125
Bibliography	143

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
2.1 Obstacle Intersection Example	12
2.2 Path Mutation Example	15
2.3 Scenario 1	17
2.4 Scenario 2	17
2.5 Scenario 3	18
2.6 Scenario 4	18
2.7 Scenario 5	19
2.8 Scenario 6	19
2.9 Scenario 7	20
2.10 Scenario 8	20
2.11 Scenario 1 Best Path - Single Run	21
2.12 Scenario 1 Path Population (Generation 0) - Single Run	22
2.13 Scenario 1 Path Population (Generation 50) - Single Run	22
2.14 Scenario 1 Path Population (Generation 100) - Single Run	23
2.15 Scenario 1 Path Population (Generation 150) - Single Run	23

2.16 Scenario 1 Path Population (Generation 199) - Single Run	24
2.17 Scenario 1 Best Fitness - Single Run	24
2.18 Scenario 1 Average Fitness 1 - Single Run	25
2.19 Scenario 1 Average Fitness 1 - Single Run	25
2.20 Scenario 2 Best Path - Single Run	26
2.21 Scenario 2 Path Population (Generation 0) - Single Run	27
2.22 Scenario 2 Path Population (Generation 50) - Single Run	27
2.23 Scenario 2 Path Population (Generation 100) - Single Run	28
2.24 Scenario 2 Path Population (Generation 150) - Single Run	28
2.25 Scenario 2 Path Population (Generation 199) - Single Run	29
2.26 Scenario 2 Best Fitness - Single Run	29
2.27 Scenario 2 Average Fitness 1 - Single Run	30
2.28 Scenario 2 Average Fitness 2 - Single Run	30
2.29 Scenario 3 Best Path - Single Run	31
2.30 Scenario 3 Best Fitness - Single Run	32
2.31 Scenario 3 Average Fitness 1 - Single Run	32
2.32 Scenario 3 Average Fitness 2 - Single Run	33
2.33 Scenario 4 Best Path - Single Run	34
2.34 Scenario 4 Best Fitness 1 - Single Run	35
2.35 Scenario 4 Best Fitness 2 - Single Run	35

2.36	Scenario 4 Average Fitness 1 - Single Run	36
2.37	Scenario 4 Average Fitness 2 - Single Run	36
2.38	Scenario 5 Best Path - Single Run	37
2.39	Scenario 5 Best Fitness - Single Run	38
2.40	Scenario 5 Average Fitness 1 - Single Run	38
2.41	Scenario 5 Average Fitness 2 - Single Run	39
2.42	Scenario 6 Best Path - Single Run	40
2.43	Scenario 6 Best Fitness - Single Run	41
2.44	Scenario 6 Average Fitness 1 - Single Run	41
2.45	Scenario 6 Average Fitness 2 - Single Run	42
2.46	Scenario 7 Best Path - Single Run	43
2.47	Scenario 7 Best Fitness - Single Run	44
2.48	Scenario 7 Average Fitness - Single Run	44
2.49	Scenario 8 Best Path - Single Run	45
2.50	Scenario 8 Best Fitness - Single Run	46
2.51	Scenario 8 Average Fitness 1 - Single Run	47
2.52	Scenario 8 Average Fitness 2 - Single Run	48
2.53	Scenario 1 Fitness Histogram	51
2.54	Scenario 1 Best Path - Batch Runs	51
2.55	Scenario 1 Worst Path - Batch Runs	52

2.56	Scenario 2 Best Path - Batch Runs	52
2.57	Scenario 2 Worst Path - Batch Runs	53
2.58	Scenario 3 Best Path - Batch Runs	53
2.59	Scenario 3 Worst Path - Batch Runs	54
2.60	Scenario 4 Best Path - Batch Runs	54
2.61	Scenario 4 Worst Path - Batch Runs	55
2.62	Scenario 5 Best Path - Batch Runs	56
2.63	Scenario 5 Worst Path - Batch Runs	57
2.64	Scenario 6 Best Path - Batch Runs	57
2.65	Scenario 6 Worst Path - Batch Runs	58
2.66	Scenario 7 Best Path - Batch Runs	59
2.67	Scenario 7 Worst Path - Batch Runs	60
2.68	Scenario 8 Best Path - Batch Runs	60
2.69	Scenario 8 Worst Path - Batch Runs	61
3.1	Obstacle Regions for Turn Direction Computation	64
3.2	Heading Regions for Turn Direction Computation	65
3.3	Vehicle Controller Flow Diagram	68
3.4	Evolutionary Planner Flow Diagram	70
3.5	Reactive Scenario 1	72
3.6	Reactive Scenario 2	73

3.7	Reactive Scenario 3	73
3.8	Reactive Scenario 4	74
3.9	Reactive Scenario 5	74
3.10	Reactive Scenario 6	75
3.11	Reactive Scenario 7	75
3.12	Reactive Scenario 8	76
3.13	Reactive Scenario 1 - Best Path	78
3.14	Reactive Scenario 1 - Worst Path	79
3.15	Reactive Scenario 2 - Best Path	80
3.16	Reactive Scenario 2 - Worst Path	81
3.17	Reactive Scenario 3 - Best Path	82
3.18	Reactive Scenario 3 - Worst Path	83
3.19	Reactive Scenario 4 - Best Path	84
3.20	Reactive Scenario 4 - Worst Path	85
3.21	Reactive Scenario 5 - Best Path	86
3.22	Reactive Scenario 5 - Worst Path	87
3.23	Reactive Scenario 6 - Best Path	88
3.24	Reactive Scenario 6 - Worst Path	89
3.25	Reactive Scenario 7 - Best Path	90
3.26	Reactive Scenario 7 - Worst Path	91

3.27 Reactive Scenario 8 - Best Path	92
3.28 Reactive Scenario 8 - Worst Path	93
A.1 Scenario 1 Path - 1	99
A.2 Scenario 1 Path - 2	100
A.3 Scenario 2 Path - 1	100
A.4 Scenario 2 Path - 2	101
A.5 Scenario 2 Fitness Histogram	102
A.6 Scenario 3 Path - 1	102
A.7 Scenario 3 Path - 2	103
A.8 Scenario 3 Fitness Histogram	103
A.9 Scenario 4 Path - 1	104
A.10 Scenario 4 Path - 2	104
A.11 Scenario 4 Fitness Histogram - No Obstacles	105
A.12 Scenario 5 Path - 1	105
A.13 Scenario 5 Path - 2	106
A.14 Scenario 5 Fitness Histogram	107
A.15 Scenario 6 Path - 1	108
A.16 Scenario 6 Path - 2	108
A.17 Scenario 6 Fitness Histogram	109
A.18 Scenario 7 Path - 1	109

A.19 Scenario 7 Path - 2	110
A.20 Scenario 7 Fitness Histogram	110
A.21 Scenario 8 Path - 1	111
A.22 Scenario 8 Path - 2	111
A.23 Scenario 8 Fitness Histogram	112
B.1 Reactive Scenario 1 Path - 1	113
B.2 Reactive Scenario 1 Path - 2	114
B.3 Reactive Scenario 2 Path - 1	114
B.4 Reactive Scenario 2 Path - 2	115
B.5 Reactive Scenario 3 Path - 1	116
B.6 Reactive Scenario 3 Path - 2	116
B.7 Reactive Scenario 4 Path - 1	117
B.8 Reactive Scenario 4 Path - 2	118
B.9 Reactive Scenario 5 Path - 1	119
B.10 Reactive Scenario 5 Path - 2	119
B.11 Reactive Scenario 6 Path - 1	120
B.12 Reactive Scenario 6 Path - 2	121
B.13 Reactive Scenario 7 Path - 1	122
B.14 Reactive Scenario 7 Path - 2	122
B.15 Reactive Scenario 8 Path - 1	123

B.16 Reactive Scenario 8 Path - 2	124
--	------------

LIST OF TABLES

<u>Table</u>	<u>Page</u>
2.1 Static EA Parameters	7
2.2 Static Scenario Fitness Component Weights	10
2.3 Static Scenarios - Batch Simulation Results	50
3.1 Heading Regions	66
3.2 Turn Direction Rules	67
3.3 Dynamic Scenarios - Batch Simulation Results	94

CHAPTER I

Introduction and Literature Review

1.1 Introduction

Owing to the myriad of military and civilian applications, autonomous path planning for unmanned air vehicles has been a topic of much research for many years. Despite the wide usage of unmanned vehicles, autonomous operations have been somewhat limited. Though advances have been made in recent years, researchers still strive to add more autonomy to these systems to make them more robust and to alleviate operator workload. The challenge to develop unmanned vehicles capable of mimicking the intelligent decision making capabilities of human beings still remains. One sub-problem in the overall problem of autonomous vehicle operations is the problem of autonomous path planning. A simple but useful path planning scenario involved planning a path between two points in an environment cluttered with obstacles.

Most contemporary path planners can be placed in one of two categories: deliberate and reactive. Deliberate path planners assume that either perfect knowledge of the environment is available or the locations of obstacles in the environment can be described by some a priori statistical distributions. The main advantage of deliberate methods is that, given enough time to compute a solution, they can often find the globally optimal solution (given perfect knowledge of the scenario). However, the time needed to compute a solution remains a significant factor in the use of deliberate algorithms

as online mission re-planners. This is especially true in the case of path planning for air vehicles, where kinematic constraints prevent the vehicle from stopping while a solution is being computed. These methods must be computationally efficient enough to provide a solution to the vehicle in a reasonable amount of time. In this case, reasonable is defined mostly by the scenario in question. A highly dynamic scenario may require a solution in seconds whereas a static or slowly changing scenario may allow for a slower response time from the path planner.

Reactive path planners, however, are designed specifically for scenarios in which there is little or no information available beforehand. Reactive planners are designed to be used online with information discovered by the vehicle during the execution of its mission. In this way, there is little or no distinction between the planning and execution of a path. The advantage of reactive methods is the relaxation of the requirement of some global knowledge about the scenario. Although this allows the vehicle to respond to unanticipated events in the mission (such as the detection of a new obstacle), this feature also prevents reactive methods from using any information that may be available. It is for this reason that a hybrid method may be more appropriate.

1.2 Literature Review

Reactive planning is actually a misnomer. Reactive methods don't plan a vehicle trajectory in the same sense that deliberate methods do. Reactive methods rely on locally sensed information about the environment rather than global knowledge. One of the most popular reactive methods uses artificial potential functions to create virtual forces that repel a vehicle from obstacles and attract a vehicle to the goal location. The vehicle is guided through the environment through sensing the local gradient of the potential field and following the direction of steepest descent. One problem with these methods is the possibility of becoming trapped in local minima. This can be especially

significant when the environment contains concave obstacles. One way to get around this problem is to define a potential function with a single minimum (known as a navigation function) ([41]).

Biologically inspired methods are also popular reactive methods. Ant colonies have inspired reactive methods based on the concept of digital pheromones ([33], [17], [31]). Algorithms employing this concept envision unmanned vehicles as ants moving through an environment and depositing pheromones in areas that have been searched. Different kinds of pheromones can be used to distinguish between areas of interest and areas that have already been searched. The digital pheromones essentially create a potential field over an environment that can be sensed by different vehicles. An advantage of these methods is that the distributed nature of the algorithm implies that the communication bandwidth required can be greatly reduced. This is really an advantage of any decentralized approach. Additionally, there is no single point of failure in the system (unlike with centralized systems).

In this work, the reactive nature of the algorithm is much simpler than in the methods described above. The vehicle simply senses obstacles in the environment and makes a hard turn (limited only by the kinematic constraints of the vehicle) to avoid the obstacle. This simple method is used to demonstrate the possibility of incorporating reactive thinking into deliberate methods. Also, the algorithm is designed such that more sophisticated reactive methods can be employed in cases where the vehicle loses communication with the centralized planner. However, that is beyond the scope of this work.

An excellent definition of the path planning problem under the deliberate philosophy can be found in [7]. The author describes the problem as "searching forward in time and space to find a trajectory which, when followed, will ideally allow the vehicle to carry out it's designated mission at a specified level of performance." In essence the objective is simply to find a path from an initial

state to a goal state that avoids all obstacles in between. Some techniques used to achieve this make use of graph theory, as in [28] and [22]. Essentially the environment is represented as a graph. Waypoints in the environment to be visited by the unmanned vehicle are encoded as nodes on the graph, connected by vertices. Path planning methods that utilize graph theory typically divide the path generation problem into two steps: the generation of an initial path connecting the start and goal states, and the "smoothing" of the initial path into a feasible path that is flyable by an air vehicle subject to some dynamic constraints. In [22], for example, the initial path is smoothed using splines.

There has also been work done to describe the path planning problem as a Mixed-Integer Linear Programming (MILP) problem. For example, [40] offers a solution to the problem of optimizing the behavior of a fleet of unmanned vehicles in terms of task allocation and trajectory generation. In this case the problem is encoded as a set of vectors and matrices that represent vehicle capabilities and state, the locations of waypoints to be visited and the locations of no-fly-zones. Constraint equations are formed which account for the aircraft dynamics and any time constraints that may be placed on waypoints (i.e. waypoint 1 must be visited before waypoint 2). The cost function is defined as a combination of the overall mission completion time and the mission completion times for the individual vehicles. An approximate method, which partially decouples the task assignment problem from the trajectory generation problem, is also developed to increase the computational efficiency of the algorithm. The authors note that the combined task assignment and trajectory generation optimization problem has significant computational costs, though it is capable of finding the global optimum. Several other MILP methods were found which incorporate additional timing and low observability constraints ([1], [10], [20]).

For the path planning problem, it would be useful to design an algorithm which produces a feasible solution quickly and then uses whatever time is available to refine that solution in a way that

brings it closer to the global optimum. Evolutionary computation provides an interesting framework for achieving this. Evolutionary algorithms have been widely used to solve many different optimization problems and have proven their value in solving problems in which finding the globally optimal solution may not be as important as finding a good solution in a reasonable amount of time. In [46], the incorporation of knowledge specific to the path planning problem into evolutionary algorithms is discussed. Typically, this means developing a representation for the path planning problem along with evolutionary operators such as fitness evaluation, mutation and selection. Currently there has been much work done in developing different operators for this problem and evaluating their efficacy ([47], [48], [29], [8]). Additionally, it has been demonstrated that evolutionary algorithms are extensible to more complicated path planning scenarios that may include planning for multiple vehicles and tracking moving targets ([19], [36], [35], [7]) It has also been shown that the nature of the evolutionary algorithm is such that it can be implemented in a distributed fashion ([21]).

Another advantage of evolutionary algorithms is that they can be employed both offline and online with few, if any, changes to the algorithm. In general, most literature deals with the offline use of these algorithms. However, [47] does develop an online evolutionary algorithm to deal with environments involving known and unknown obstacles. In this work, the evolutionary algorithm runs and continuously updates the best path for the vehicle to follow after some number of generations. When an unknown obstacle is detected, it is added to the environment map and paths are re-evaluated. After the detection of a new obstacle, the best feasible path is executed by the vehicle.

1.3 Current Work

The current work is broken into two phases. Phase one considers using evolutionary algorithms to plan paths in static scenarios. A suitable encoding for the problem is defined along with a fitness

function and operators for mutation and selection of potential paths. Eight scenarios are defined for testing. In the static scenarios, it is assumed that all obstacles are perfectly known. The second phase of this work defines a reactive architecture in which the path planner and the vehicle exist in separate threads and communicate with each other; with the vehicle sending state updates and path requests when necessary and the planner responding with a path. The reactive solution presented makes several improvements on what is current in the literature.

First, the addition of a separate reactive controller allows the evolutionary algorithm to compute a better solution while the vehicle follows the avoidance path provided. In this way, the time for re-planning is inherently limited but the evolutionary algorithm will use any time available to come up with a solution. Also, current reactive algorithms in the literature assume that when a previously unknown obstacle is detected by the vehicle's sensor, the complete extents of the obstacle are perfectly known. This is likely to be unrealistic given the size of the obstacle relative to the sensor footprint. In this work, when an obstacle is detected it is not assumed that the obstacle is completely known. In fact this new information is never used by the evolutionary planner in creating paths. This may be overly conservative and could be responsible for the poor performance in some scenarios. Methods for improving the current algorithm will be discussed in detail.

CHAPTER II

Evolutionary Algorithms for Static Path Planning Scenarios

This chapter describes the development of an evolutionary algorithm for path planning in static scenarios. These scenarios are characterized by a starting position, a goal position and a set of obstacles in the environment which is completely known. There are several steps involved in encoding the path planning problem in an evolutionary algorithm. First, the representation for a potential solution to the problem must be determined. At that point, the criteria for judging the goodness of a path must be chosen and a fitness function defined. Finally, traditional evolutionary operators for mutation and selection are developed.

2.1 Overall Algorithm Description

Table 2.1 gives several important parameters used by the static evolutionary algorithm.

Table 2.1: Static EA Parameters

Parameter	Value
Number of paths in population	10
Max number of segments in a path	30
Number of generations	200

The algorithm begins by creating a population of randomly generated paths. Then the path population is mutated by applying the mutation operator to each path in the population. The mutated path list is combined with the original path list and the fitness of each solution is evaluated. At that point the selection operator is used to select the 10 best solutions from the full population of parent and offspring paths. The selected paths are taken as the starting population for the next generation and the process continues for 200 generations. The following sections describe each component of the evolutionary process in greater detail.

2.1.1 Solution Representation

There are several possible representations that can be used to encode the a path through an environment. Traditionally, the most common representations are lists of waypoints or sequences of maneuvers. For this work, a path will be represented as a series of commands to a vehicle each of which consists of a commanded heading, a commanded speed and a time for a given path segment. Currently, a maximum number of path segments in a path is defined (for the static case this is 30). The headings of the path are constrained so that each path segment has a heading that is feasible based on the previous segment heading. Equations 2.1 and 2.2 are used to ensure the feasibility of the path in terms of the heading angles. It should be noted that because the goal location is fixed, the final path heading is also fixed given all previous path headings. This means that the last path heading is not free to evolve. For this reason, this path representation omits the final path heading required to reach the goal. In some instances this may result in a final infeasible turn to the goal. For the static case this will be accepted given that one objective of the fitness function is to minimize the length of this final path segment. However, for the reactive case, additional steps will be taken to ensure that the path is completely flyable.

$$R_{min} = \frac{V^2}{g \tan \phi} \quad (2.1)$$

$$\Delta\psi_{max} = \frac{V}{R_{min}} \Delta t \quad (2.2)$$

In equation 2.1, ϕ is the aircraft bank angle. It is assumed that the aircraft banks at an angle of 45.0° as it turns to each path segment. Also, Δt in equation 2.2 is the time that it takes to turn between two consecutive path segments. For the purposes of this study we assume that this is one second. This assumption may be unrealistic but it must be made due to the fact that we lack a higher fidelity vehicle model that incorporates the dynamics of a vehicle tracking a commanded heading. Integrating such a model into the algorithm is beyond the scope of the current work.

In addition to the heading constraint, the vehicle speed and segment time are constrained as well. These constraints are shown in 2.3 and 2.4.

$$8.0 \frac{m}{s} \leq V \leq 15.0 \frac{m}{s} \quad (2.3)$$

$$t \geq 10.0 seconds \quad (2.4)$$

The vehicle constraint ensures that the vehicle exhibits realistic kinematics. A path segment which violates the time constraint is removed from the path (i.e. the time is set to zero). This is done to bias the algorithm towards removing unnecessary short path segments in favor of extending other path segments.

2.1.2 Fitness Evaluation

The fitness function used to judge the quality of the potential solutions has three components: the distance to the goal, the overall path length and the intersection between the path and any obstacle in the environment. The fitness components are defined below.

$$f_1 = \sqrt{(p_{x,N-1} - p_{x,N})^2 + (p_{y,N-1} - p_{y,N})^2} \quad (2.5)$$

$$f_2 = \sum_{i=1}^{N-1} \sqrt{(p_{x,i} - p_{x,i+1})^2 + (p_{y,i} - p_{y,i+1})^2} \quad (2.6)$$

$$f_3 = \begin{cases} 0, & \text{if no intersections} \\ 1, & \text{if intersections} \end{cases} \quad (2.7)$$

The simplest way to combine these objectives is to use a weighted sum. So, the overall fitness function is defined in 2.8.

$$f = \sum_{i=1}^N w_i f_i \quad (2.8)$$

The weights used in the static scenarios are given in Table 2.2

Table 2.2: Static Scenario Fitness Component Weights

Component	Symbol	Value
Distance to goal	w_1	10.0
Path length	w_2	15.0
Obstacle intersection	w_3	10^8

These weights were determined through much experimentation, as is usually done with evolutionary algorithms.

Obstacle Intersection Testing

An obstacle intersection method was implemented in the following manner. First, a bounding rectangle is drawn around each path segment. The width of the rectangle can be envisioned as some minimum safe distance that the vehicle must be from the obstacles. Once the bounding rectangles have been created, the Separating Axis Theorem (SAT) is used to test for intersection between each bounding rectangle and each obstacle. The SAT is commonly used in graphics applications as a simple and efficient collision detection mechanism between two polygons. In principle, the SAT relies on the fact that if two obstacles are not intersection, there should be some separating axis that divides them. It can be proven that this separating axis must be parallel to at least one edge of the polygons being tested. The method works by creating potential separating axes by taking an axis perpendicular to the normal vector from each edge of the polygon and projecting each polygon onto it. If the projections are intersecting, then the polygons are intersecting along this axis. This does not mean, however, that the polygons intersect on all possible axes. As a result, every potential separating axis must be tested. If just one axis is found on which the polygons do not intersect, the polygons are said to be non-intersecting. Figure 2.1 shows a potential path with it's bounding rectangles.

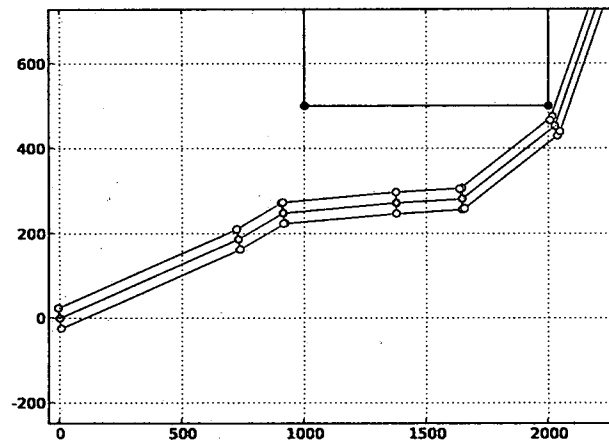


Figure 2.1: Obstacle Intersection Example

2.1.3 Mutation Operators

Naturally, the mutation operators used rely heavily on the encoding of the solution. In this instance, because the solution is essentially a collection of real-valued parameters, a gaussian mutation operator was developed that adds gaussian random variables to each heading, speed and time in the path.

$$\psi_{new} = \psi_{old} + \mathcal{N}(0, \sigma_\psi) \quad (2.9)$$

$$v_{new} = v_{old} + \mathcal{N}(0, \sigma_v) \quad (2.10)$$

$$t_{new} = t_{old} + \mathcal{N}(0, \sigma_t) \quad (2.11)$$

The standard deviations of these random variables are often the subject of much tuning and experimentation in the evolutionary computation community. These parameters could be defined as constants in the algorithm or they could be allowed to evolve online as the algorithm executes. In this implementation, the standard deviation is the same for the headings, speeds and times and it is computed online. Reference [3] gives a method for adjusting the mutation standard deviation known as the Rechenberg 1/5 Rule. This rule commonly used is based on calculated optimal convergence rates for evolutionary algorithms. Basically, the rule states that one out of every five mutations should increase the fitness of the solution. If the calculated ratio of successful mutations to all mutations (p_s) is greater than 0.2, the standard deviation should be increased. The standard deviation should be decreased if p_s is less than 0.2. The mutation standard deviation is updated every 20 generations and the update equation is given in equation 2.12.

$$\sigma_{new} = \begin{cases} \frac{\sigma_{old}}{c}, & \text{if } p_s > 0.2 \\ \sigma_{old}c, & \text{if } p_s < 0.2 \\ \sigma_{old}, & \text{if } p_s = 0.2 \end{cases} \quad (2.12)$$

The standard deviation is initialized to 5.0 for all paths. Figure 2.2 shows an example of a path before and after mutation. It can be seen that the mutation operator does not produce overly large changes to a path, which would be detrimental for the convergence of the algorithm.

In addition to the mutation operator described above, a directed mutation is also used in the event of an obstacle intersection. This mutation changes the heading of the path segment immediately preceeding the segment which intersects with an obstacle. This does not guarantee that the obstacle is avoided. In particular, if the heading difference between the intersecting segment and the segment prior to it is already constrained to the maximum heading change, then no increase in heading is possible. One advantage of the representation chosen is that changing one segment heading propagates the change down the entire path. This makes it possible to induce large changes in the path when necessary.

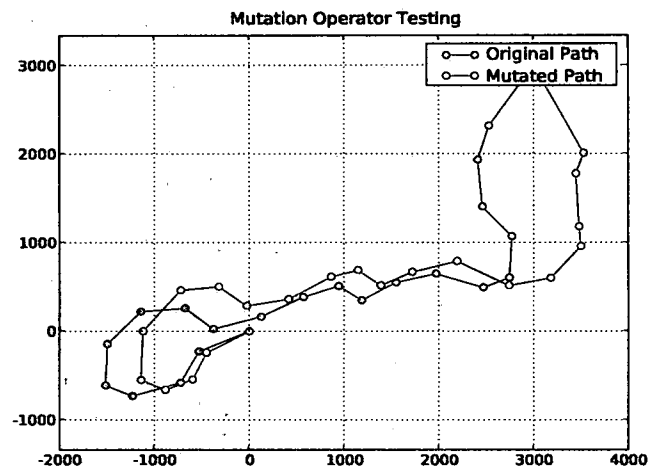


Figure 2.2: Path Mutation Example

2.1.4 Selection Operators

Two different selection mechanisms were developed for the evolutionary algorithm. In both cases, after the current path population is mutated, the new paths are added to the path population and the selection operator acts on the full path list with both the parent and offspring paths. The first selection operator is a simple elitist selection mechanism that promotes the best paths to the next generation. Although simple to implement the disadvantage of elitist selection is that it can promote premature convergence to a local minimum. To address this, a second selection operator was developed. The second, employing a tournament selection mechanism, is not guaranteed to promote the best path to the next generation. This selection method works by taking each member of the population and selecting q other members for a competition. For each member in the tournament, the fitness value of the original path is compared with each competitor. Each time that the fitness of the original path is found to be greater than the fitness of a competitor, the score of the original path is incremented by one. At the end of the selection process the paths with the highest scores are promoted to the next generation. The tournament selection operator is employed for the static evolutionary algorithm.

2.2 Scenario Descriptions

Eight total static scenarios were defined for testing the evolutionary path planning algorithm. Four scenarios were created by hand to test the algorithms effectiveness at dealing with concave obstacles and obstacles separated by narrow paths. Four additional scenarios were generated with randomly placed obstacles throughout the environment. The scenarios are shown in Figures 2.3 through 2.10. The initial and starting positions are indicated with red diamonds

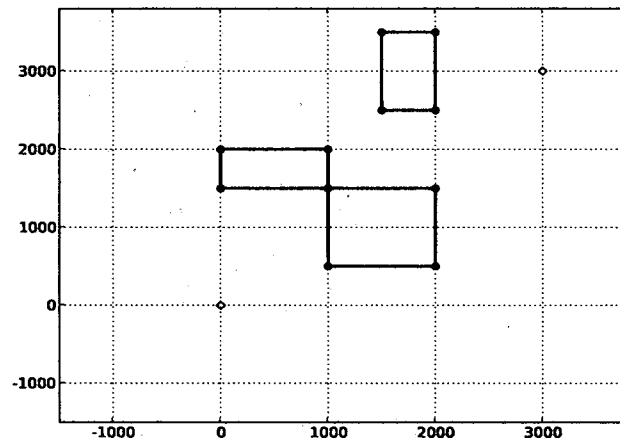


Figure 2.3: Scenario 1

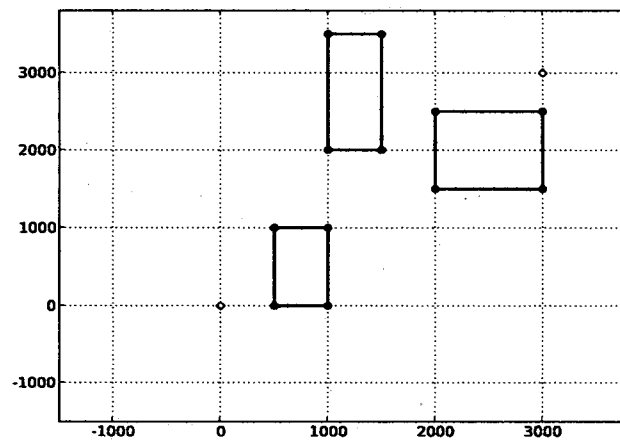


Figure 2.4: Scenario 2

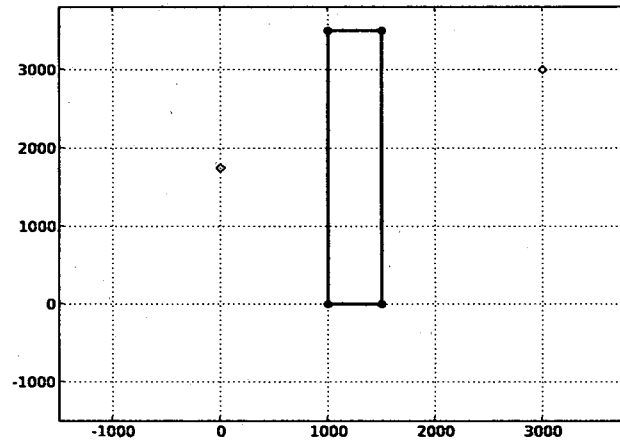


Figure 2.5: Scenario 3

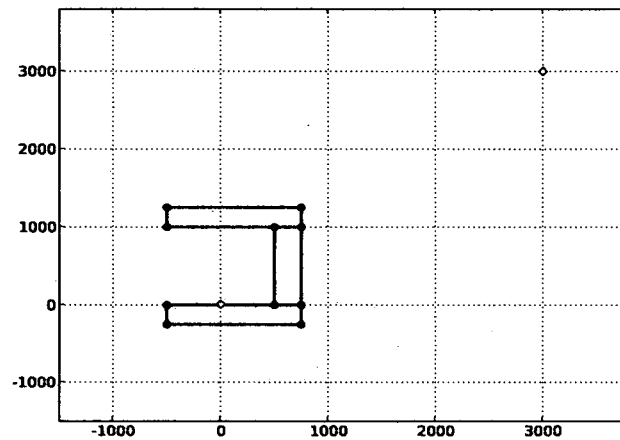


Figure 2.6: Scenario 4

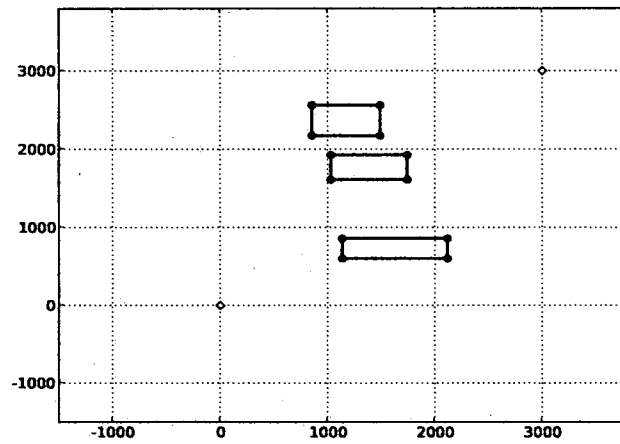


Figure 2.7: Scenario 5

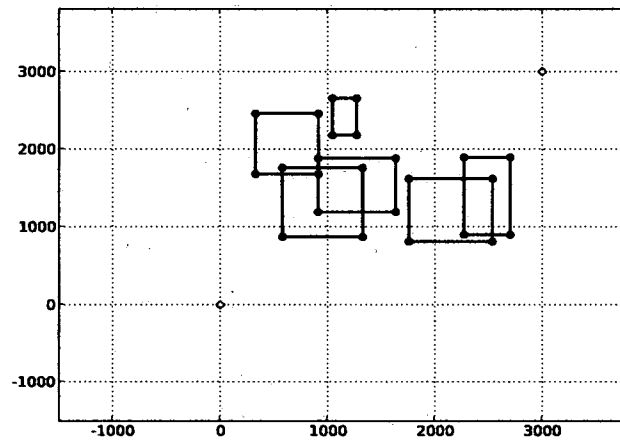


Figure 2.8: Scenario 6

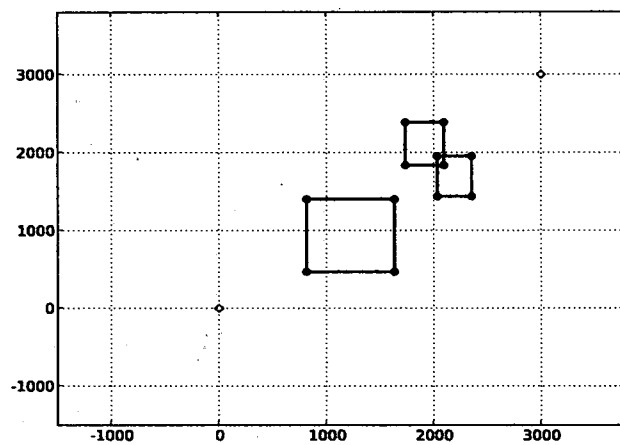


Figure 2.9: Scenario 7

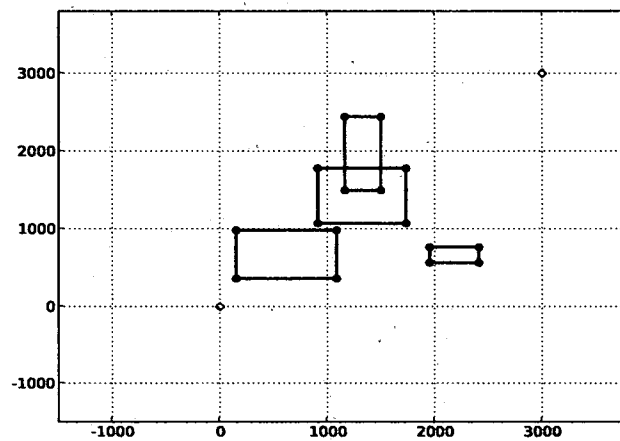


Figure 2.10: Scenario 8

2.3 Simulation Results

The following sections give the results of running this evolutionary algorithm on each of the scenarios described above. For each scenario, the best path produced after 200 generations is shown along with the best fitness and average fitness of the population through time. Additionally, for the first two scenarios, the evolution of the entire population is shown at different points throughout the search process. After this, the algorithm was run 1000 times for each scenario to gather information regarding the distribution of fitness values produced by the algorithm. Histograms of the fitness values are shown along with results of these batch simulations. Additional plots can be found in Appendix A.

2.3.1 Scenario 1

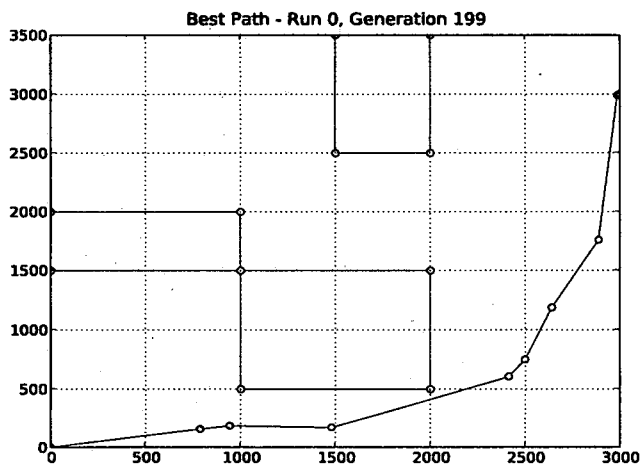


Figure 2.11: Scenario 1 Best Path - Single Run

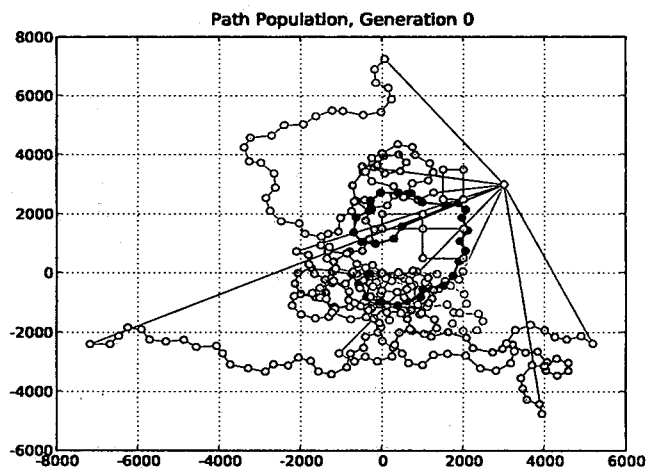


Figure 2.12: Scenario 1 Path Population (Generation 0) - Single Run

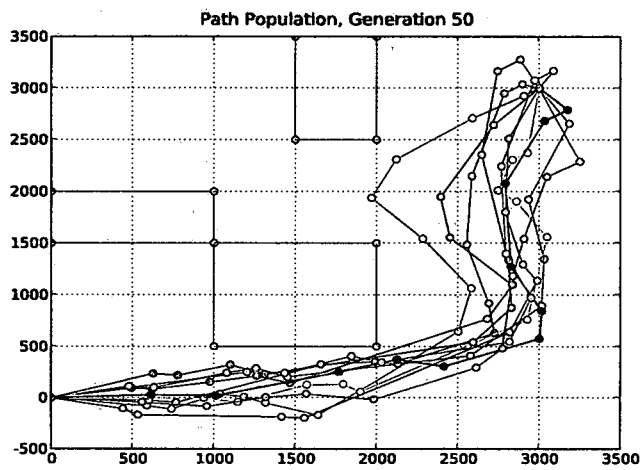


Figure 2.13: Scenario 1 Path Population (Generation 50) - Single Run

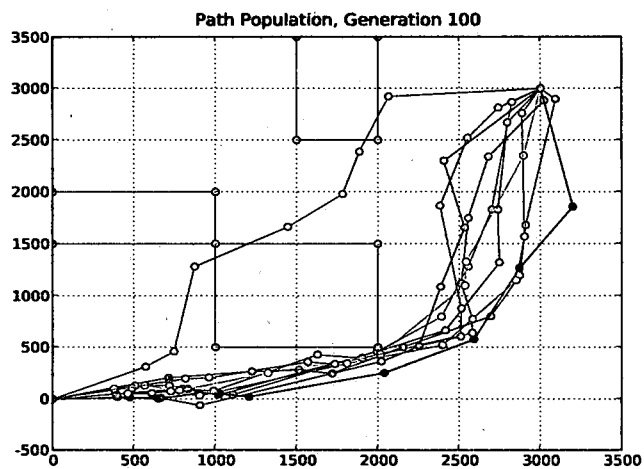


Figure 2.14: Scenario 1 Path Population (Generation 100) - Single Run

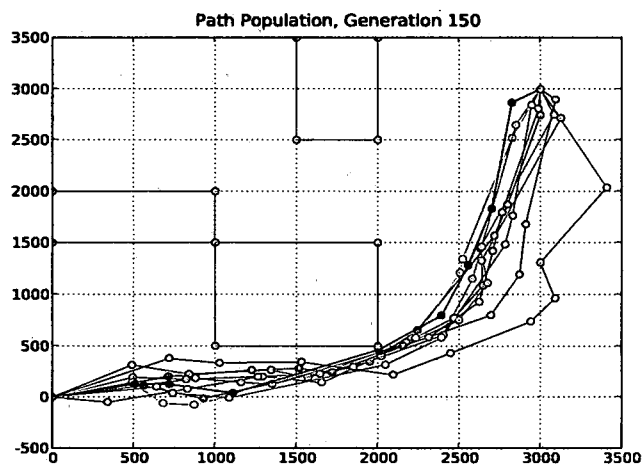


Figure 2.15: Scenario 1 Path Population (Generation 150) - Single Run

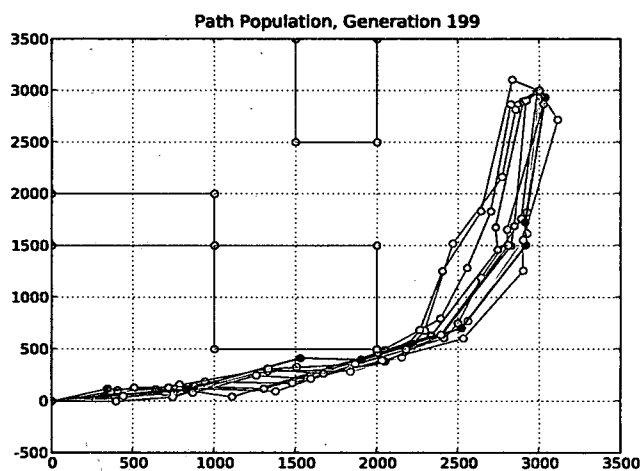


Figure 2.16: Scenario 1 Path Population (Generation 199) - Single Run

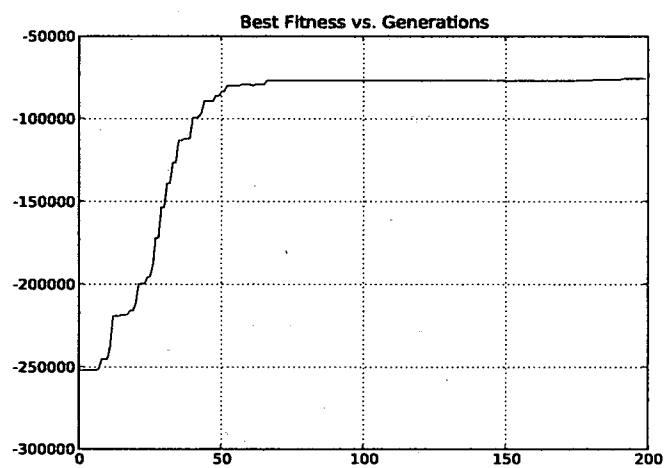


Figure 2.17: Scenario 1 Best Fitness - Single Run

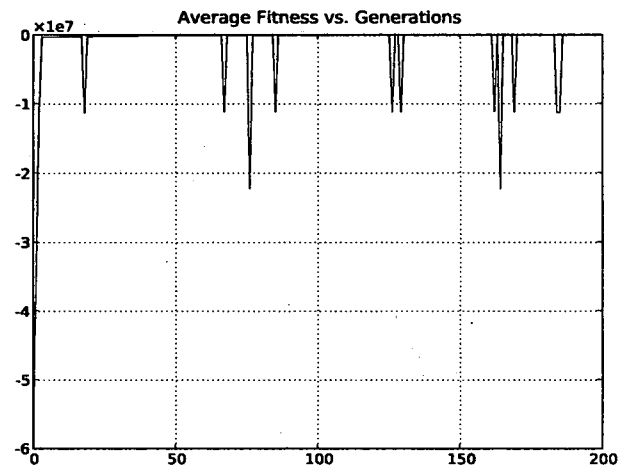


Figure 2.18: Scenario 1 Average Fitness 1 - Single Run

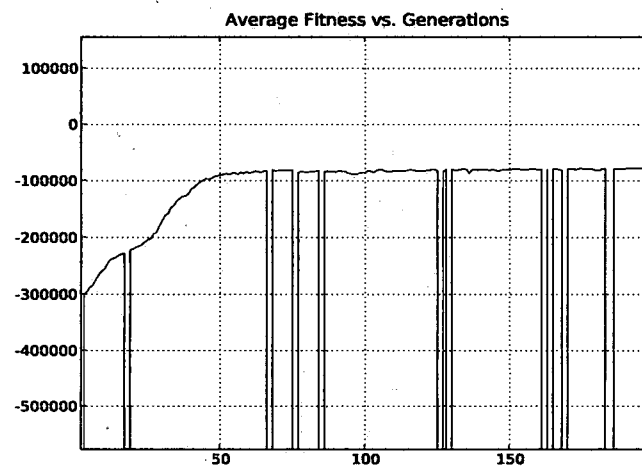


Figure 2.19: Scenario 1 Average Fitness 1 - Single Run

2.3.2 Scenario 2

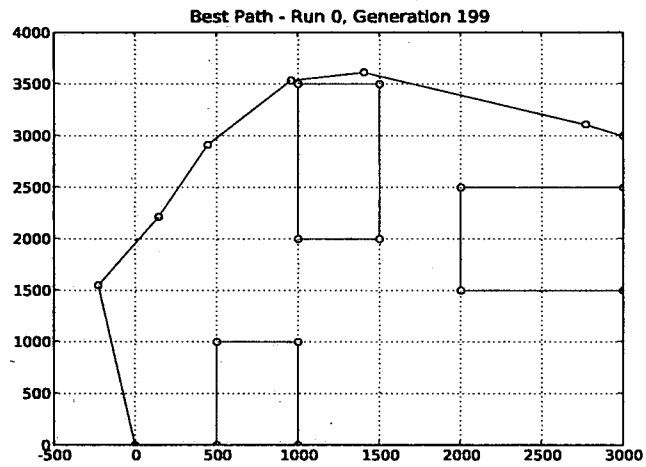


Figure 2.20: Scenario 2 Best Path - Single Run

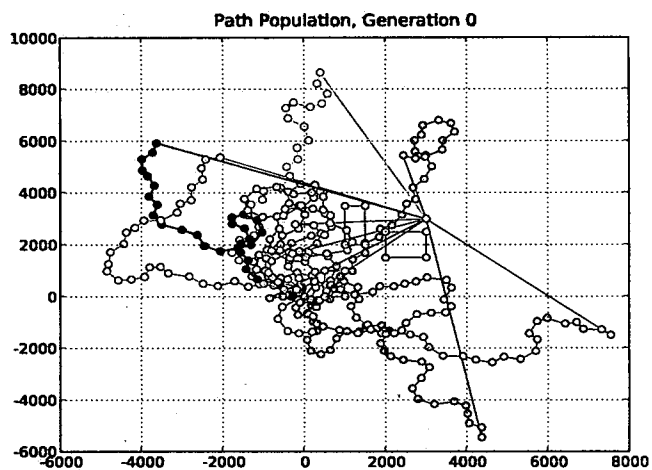


Figure 2.21: Scenario 2 Path Population (Generation 0) - Single Run

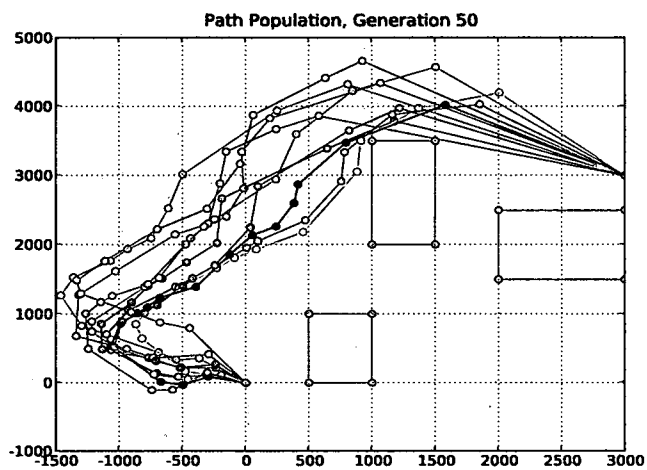


Figure 2.22: Scenario 2 Path Population (Generation 50) - Single Run

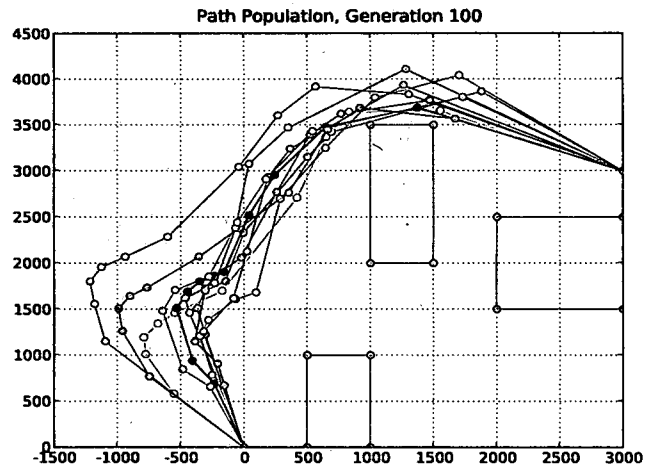


Figure 2.23: Scenario 2 Path Population (Generation 100) - Single Run

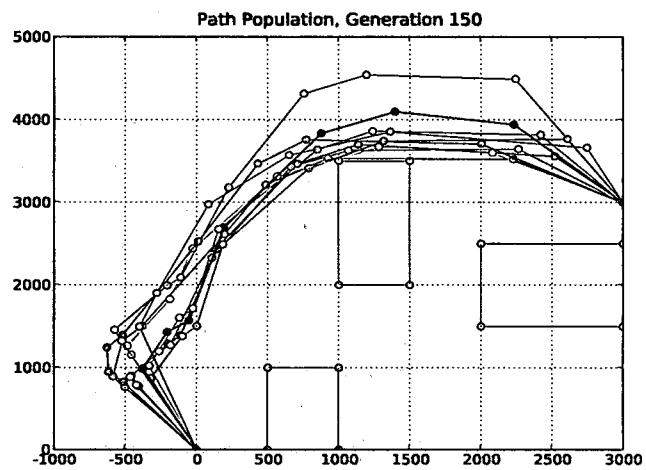


Figure 2.24: Scenario 2 Path Population (Generation 150) - Single Run

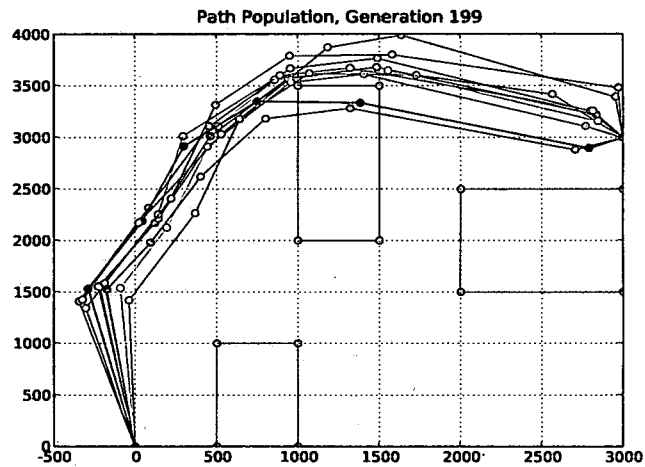


Figure 2.25: Scenario 2 Path Population (Generation 199) - Single Run

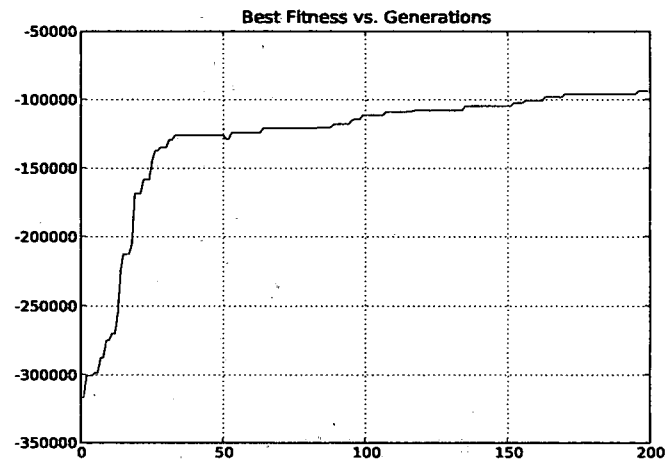


Figure 2.26: Scenario 2 Best Fitness - Single Run

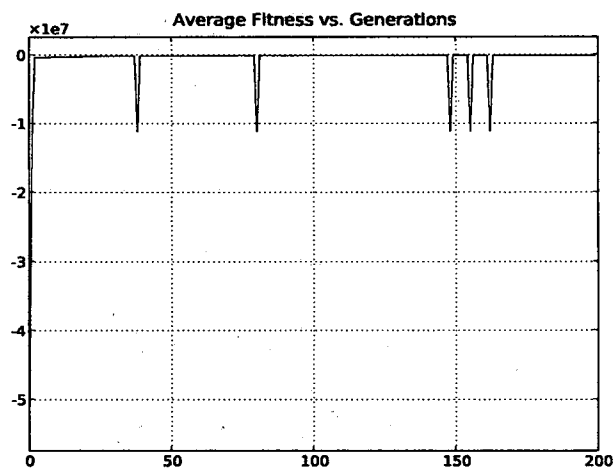


Figure 2.27: Scenario 2 Average Fitness 1 - Single Run

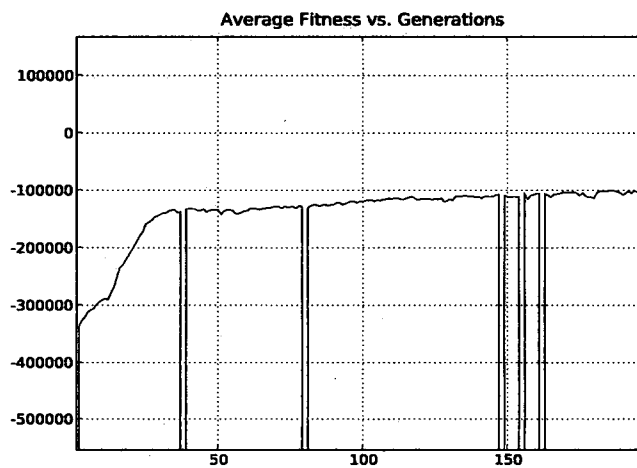


Figure 2.28: Scenario 2 Average Fitness 2 - Single Run

2.3.3 Scenario 3

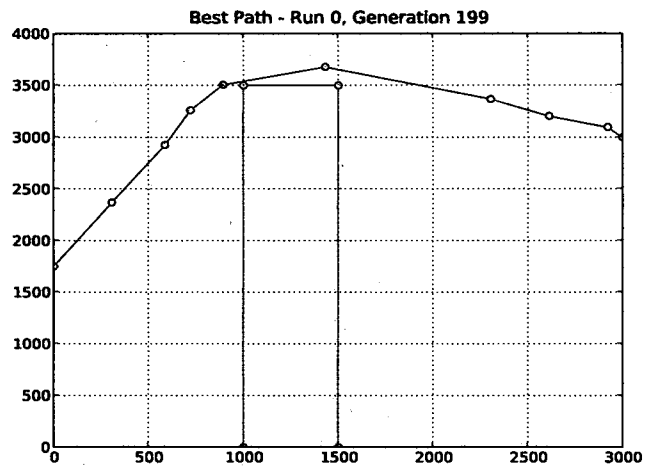


Figure 2.29: Scenario 3 Best Path - Single Run

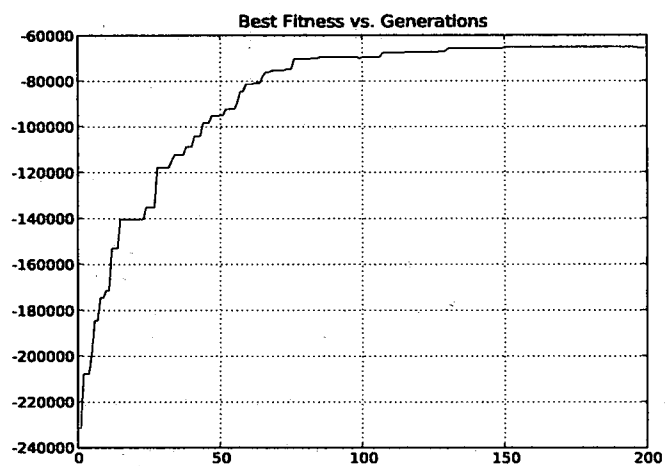


Figure 2.30: Scenario 3 Best Fitness - Single Run

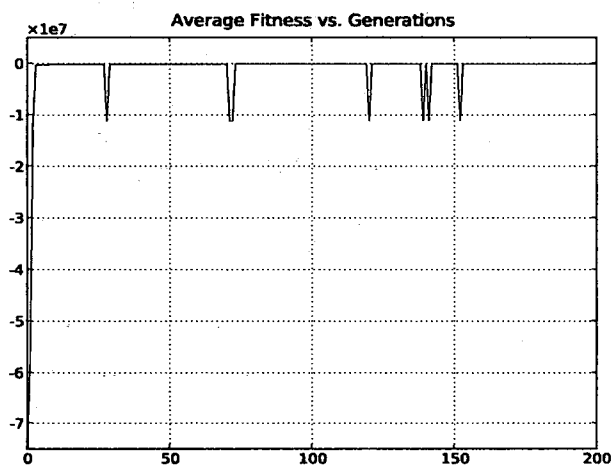


Figure 2.31: Scenario 3 Average Fitness 1 - Single Run

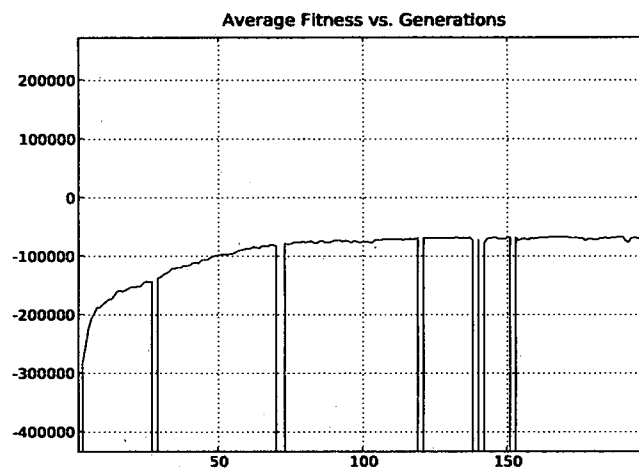


Figure 2.32: Scenario 3 Average Fitness 2 - Single Run

2.3.4 Scenario 4

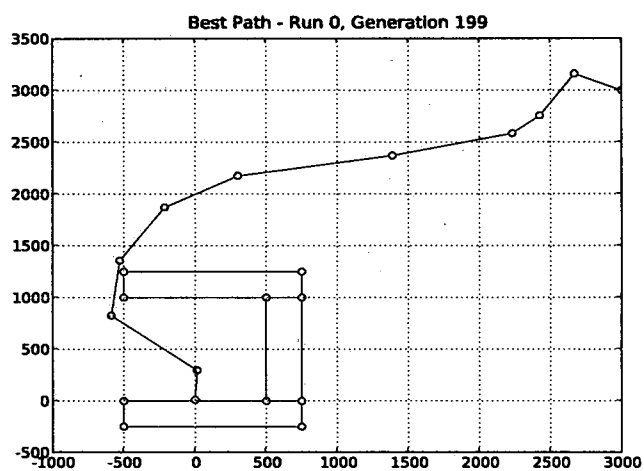


Figure 2.33: Scenario 4 Best Path - Single Run

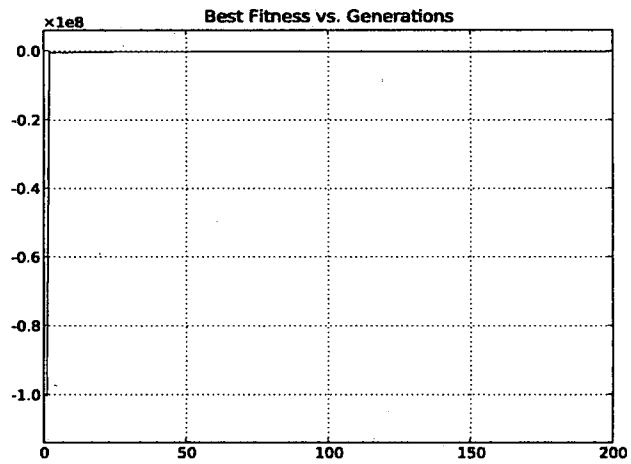


Figure 2.34: Scenario 4 Best Fitness 1 - Single Run

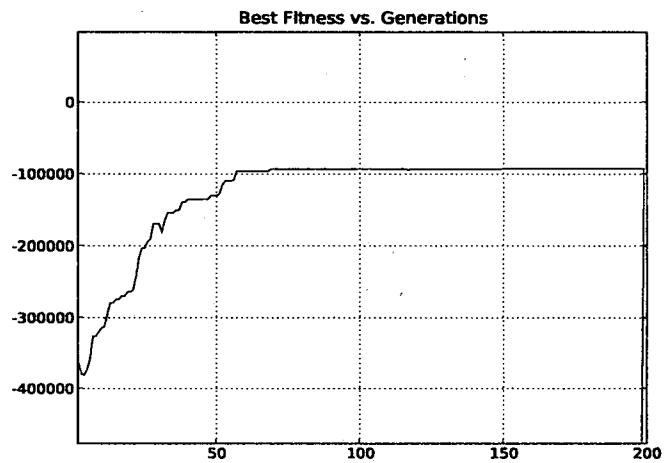


Figure 2.35: Scenario 4 Best Fitness 2 - Single Run

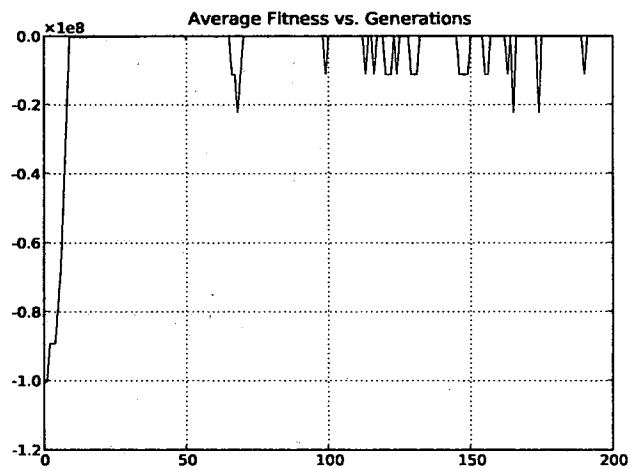


Figure 2.36: Scenario 4 Average Fitness 1 - Single Run

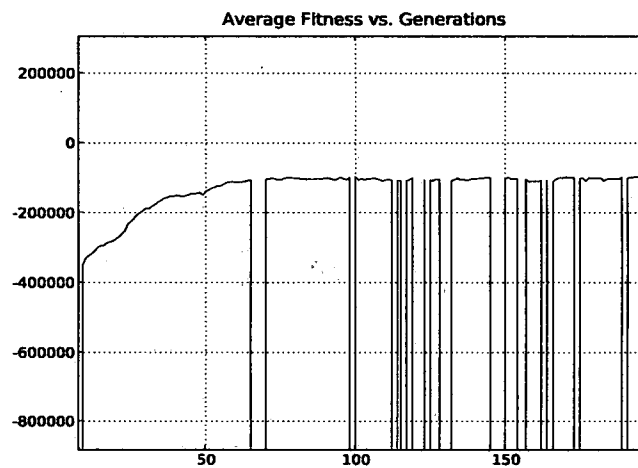


Figure 2.37: Scenario 4 Average Fitness 2 - Single Run

2.3.5 Scenario 5

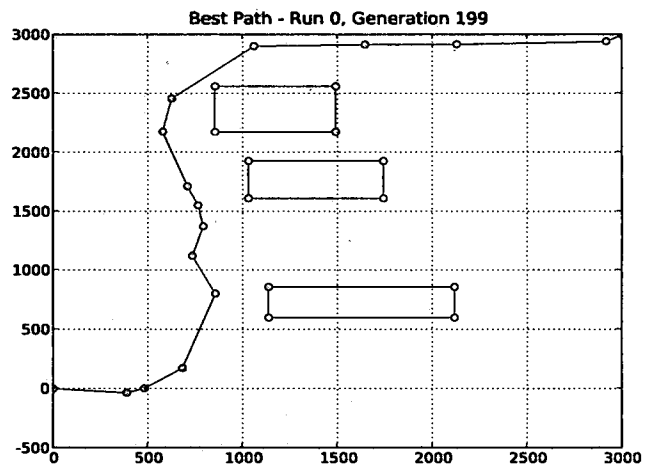


Figure 2.38: Scenario 5 Best Path - Single Run

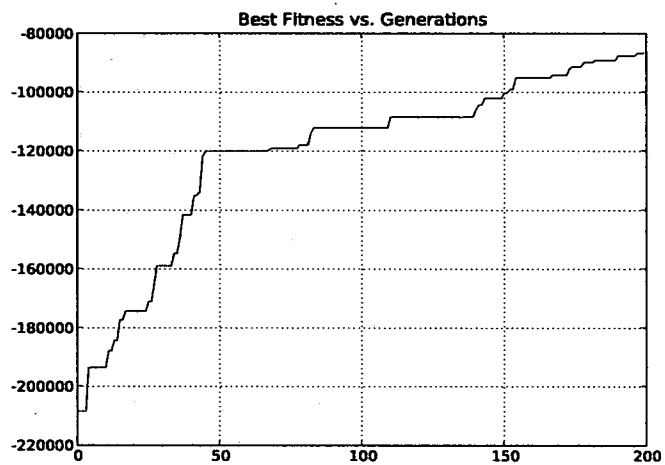


Figure 2.39: Scenario 5 Best Fitness - Single Run

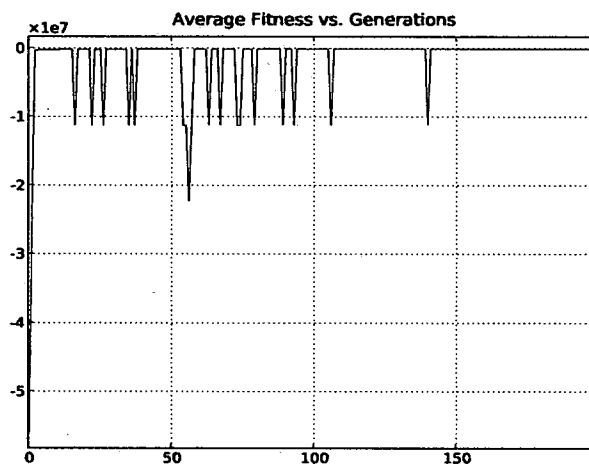


Figure 2.40: Scenario 5 Average Fitness 1 - Single Run

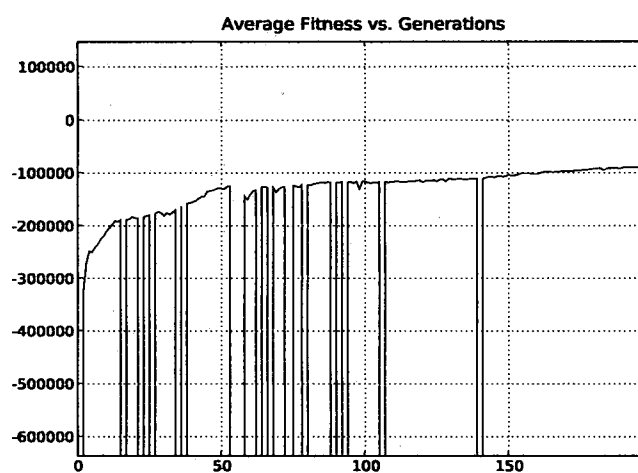


Figure 2.41: Scenario 5 Average Fitness 2 - Single Run

2.3.6 Scenario 6

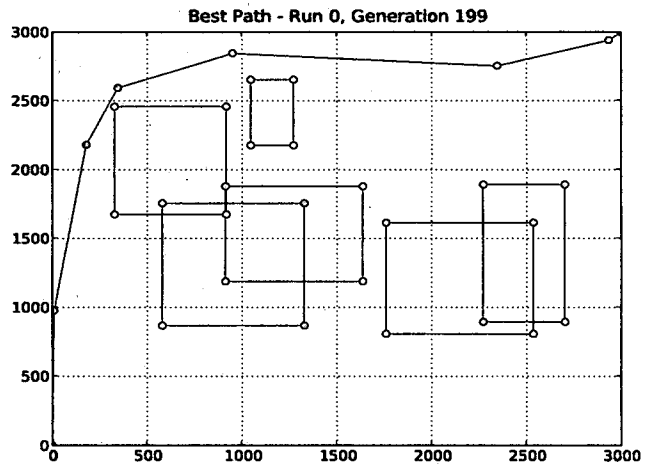


Figure 2.42: Scenario 6 Best Path - Single Run

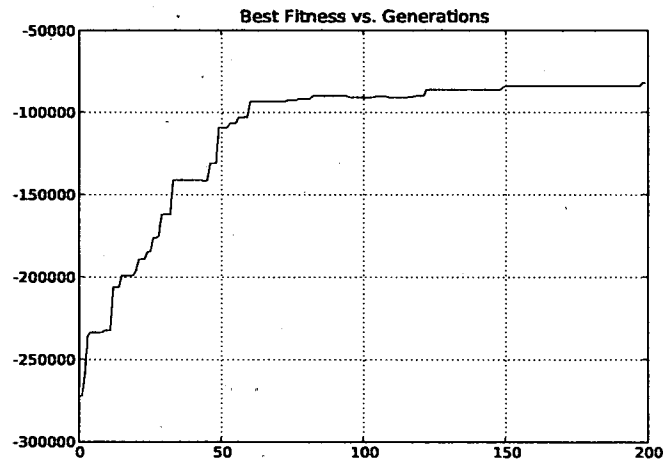


Figure 2.43: Scenario 6 Best Fitness - Single Run

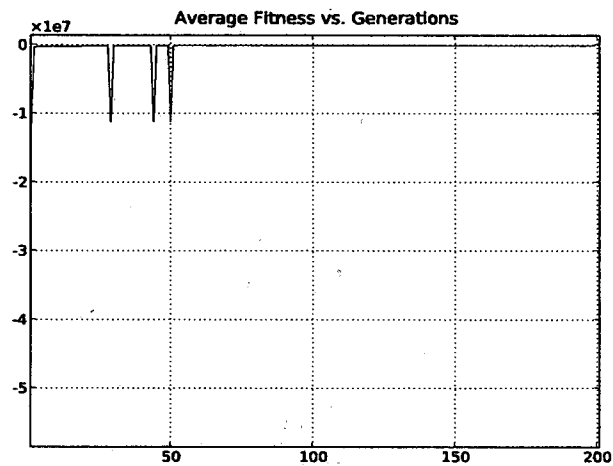


Figure 2.44: Scenario 6 Average Fitness 1 - Single Run

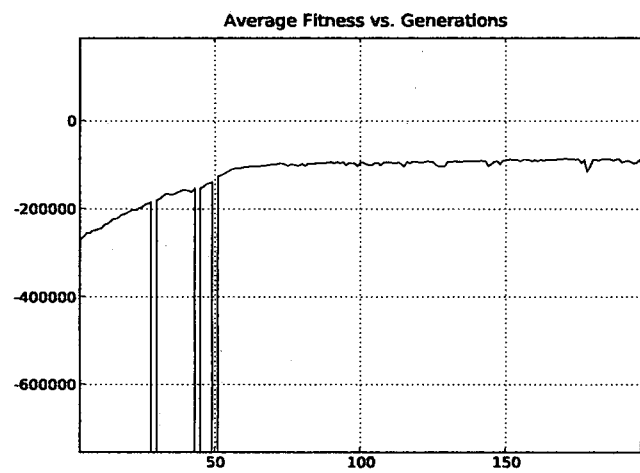


Figure 2.45: Scenario 6 Average Fitness 2 - Single Run

2.3.7 Scenario 7

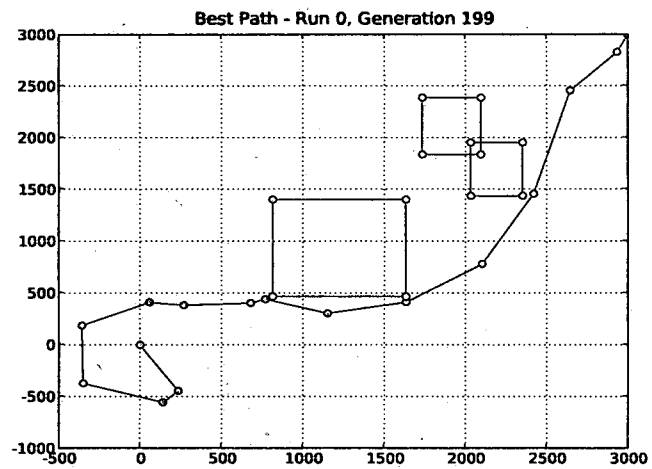


Figure 2.46: Scenario 7 Best Path - Single Run

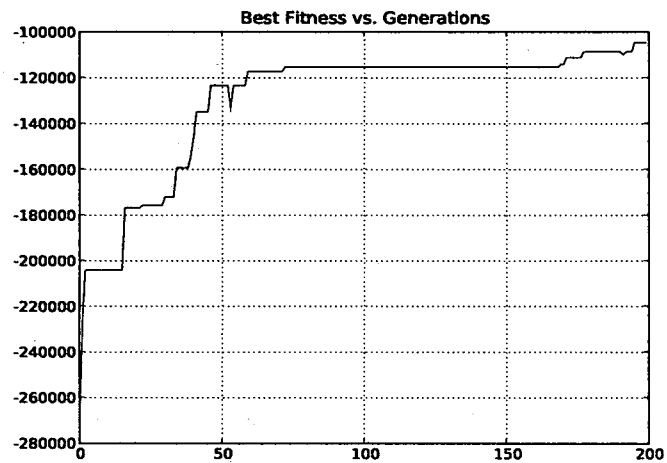


Figure 2.47: Scenario 7 Best Fitness - Single Run

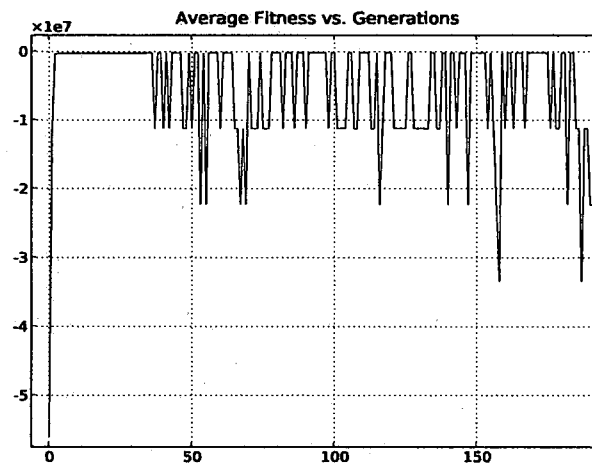


Figure 2.48: Scenario 7 Average Fitness - Single Run

2.3.8 Scenario 8

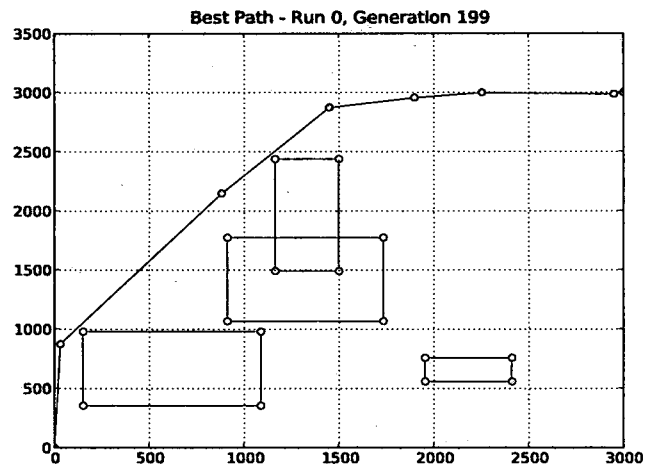


Figure 2.49: Scenario 8 Best Path - Single Run

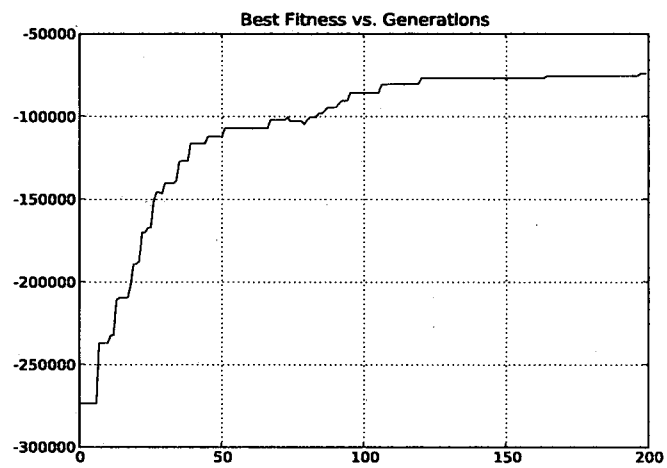


Figure 2.50: Scenario 8 Best Fitness - Single Run

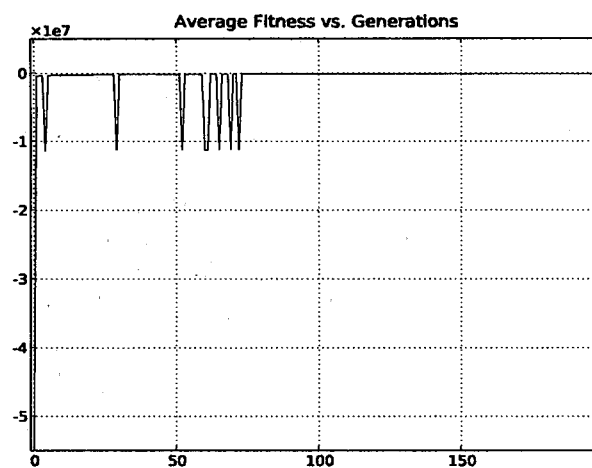


Figure 2.51: Scenario 8 Average Fitness 1 - Single Run

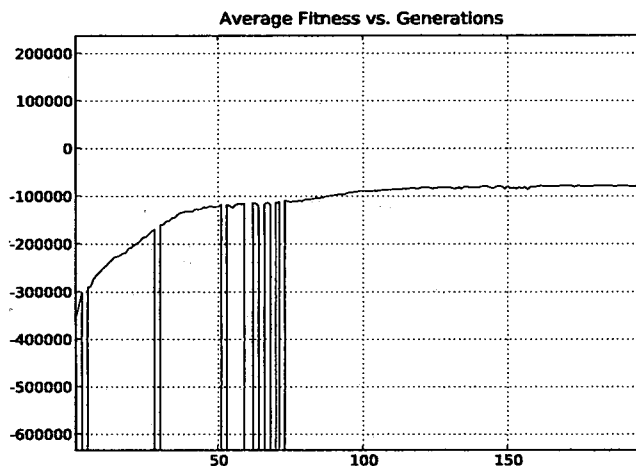


Figure 2.52: Scenario 8 Average Fitness 2 - Single Run

2.3.9 Discussion and Batch Results

Several things should be noted about the single run results above. First, although the best fitness value in the population seems to settle to a steady state value at times (as in Figure 2.30, this is not always the case. In some instances (Figures 2.26 and 2.39) the algorithm terminates before a steady state is reached. Our goal, however, was not to find the globally optimal solution but to find good solutions quickly. It can be seen from the plots that there is generally a sharp increase in the average fitness of the population early on. This is explained by the fact that the algorithm has found solutions that do not intersect with obstacles. Given the importance of avoiding known obstacles, this is a desirable feature. After this initial jump in average fitness, the algorithm tries to optimize the path length and distance to goal metrics.

Also of importance is the fact that both the average and best fitnesses can decrease over short time periods. This is primarily a result of the selection and mutation mechanisms. As stated before, the selection mechanism is not guaranteed to pass the best individual on to the next generation. Any decrease in the best fitness of the population is an indication that the best individual found at one time is not present in the population later. If we had used an elitist selection algorithm, no decrease in best fitness would be observed. The decreases in average fitness are believed to be due primarily to the mutation operator chosen. Because the standard deviation of the mutation operator varies over time, this can cause delays in the convergence of the algorithm. The purpose of this is to allow the algorithm the possibility to escape local optima.

The results of the batch runs are shown in Table 2.3. All values have been normalized with respect to the minimum fitness and negated. Therefore, in each case the minimum fitness is -1 and the best fitness possible is zero. From these data it can be seen that with respect to obstacle intersection, the algorithm performs well. Only in Scenario 4 does the algorithm ever fail to find a collision free path, and then only twice. The standard deviation for all scenarios is shown as well. In general, the standard deviation is near 10% for all scenarios. It should be noted that the high maximum fitness value for Scenario 4 is due primarily to the fact that the worst path in this scenario intersected an obstacle.

To illustrate the diversity of path fitnesses produced by the algorithm, histograms were created of the fitness values for each static scenario. Figure 2.53 shows this histogram for Scenario 1.

This plot shows that there is a grouping of fitness values around the maximum of $-7.2906E4$ but there is also an additional group centered approximately around -95000. These data are presented, in part, to address what seems to be a deficiency in the established literature. Results presented generally consist of typical paths produced by the algorithm without any indication of how likely

Table 2.3: Static Scenarios - Batch Simulation Results

Scenario	Intersections	Min. Fitness	Max. Fitness	Avg. Fitness	Std. Deviation
1	0	-1.0	-0.4151	-0.5326	0.1130
2	0	-1.0	-0.3856	-0.5423	0.0928
3	0	-1.0	-0.3177	-0.4123	0.1016
4	2	-1.0	-0.0008	-0.0029	0.0446
5	0	-1.0	-0.4050	-0.5252	0.0963
6	0	-1.0	-0.4465	-0.5348	0.0966
7	0	-1.0	-0.4204	-0.4830	0.0845
8	0	-1.0	-0.3737	-0.4565	0.0909

the particular algorithm is to produce that path. These data presented here show that the algorithm generally produces a large range of possible paths for a given scenario. In addition to the tabulated data, the best and worst paths for each scenario were collected as well. These paths are shown in the following figures.

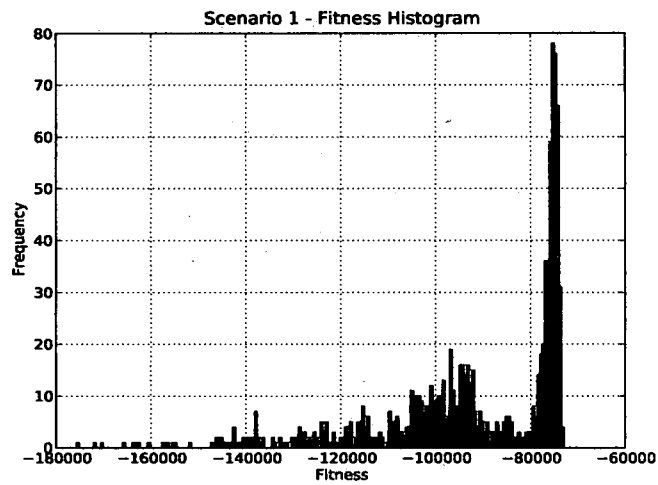


Figure 2.53: Scenario 1 Fitness Histogram

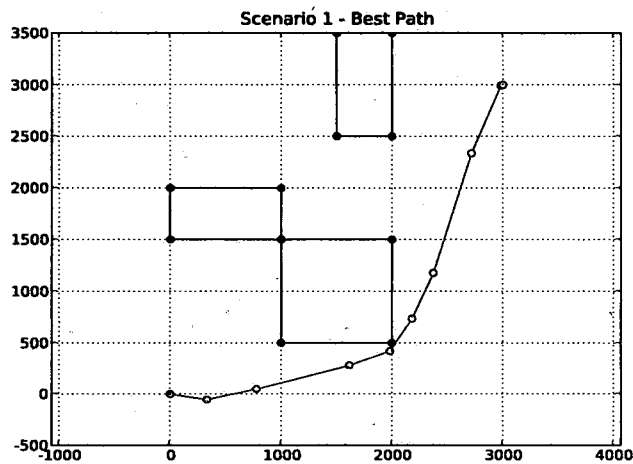


Figure 2.54: Scenario 1 Best Path - Batch Runs

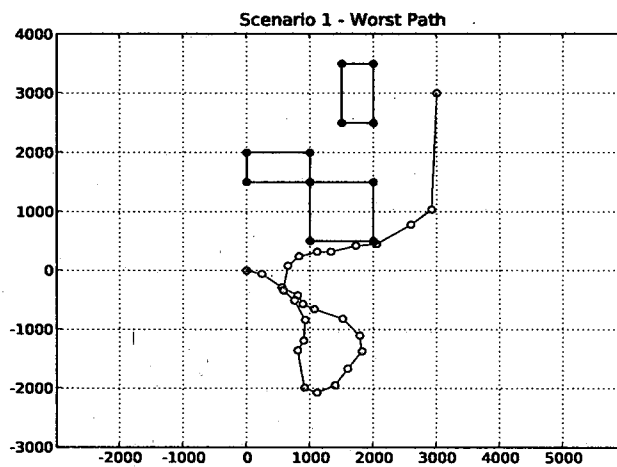


Figure 2.55: Scenario 1 Worst Path - Batch Runs

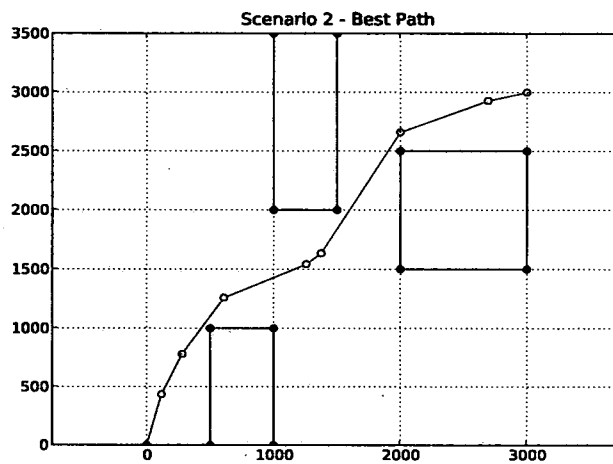


Figure 2.56: Scenario 2 Best Path - Batch Runs

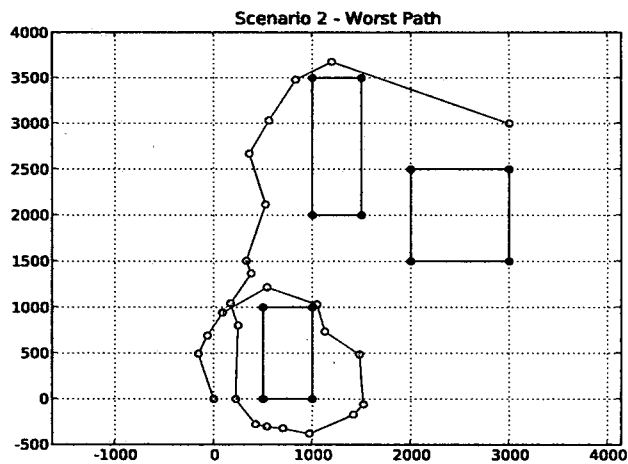


Figure 2.57: Scenario 2 Worst Path - Batch Runs

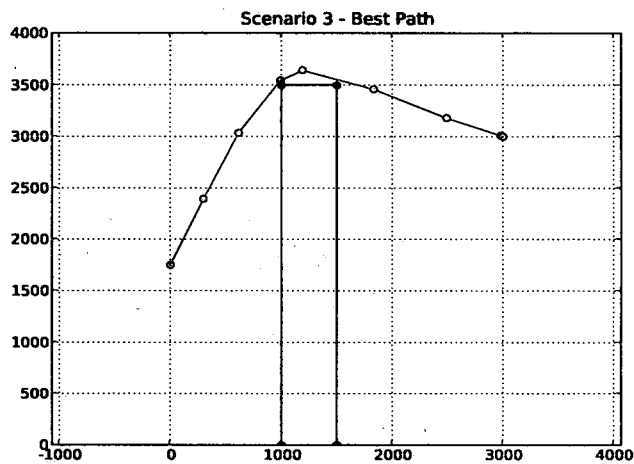


Figure 2.58: Scenario 3 Best Path - Batch Runs

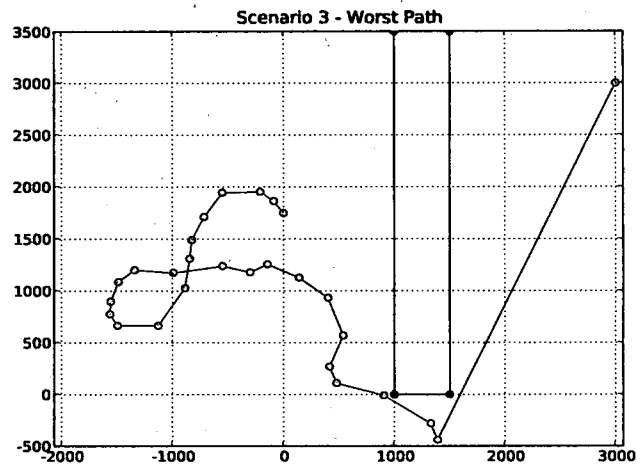


Figure 2.59: Scenario 3 Worst Path - Batch Runs

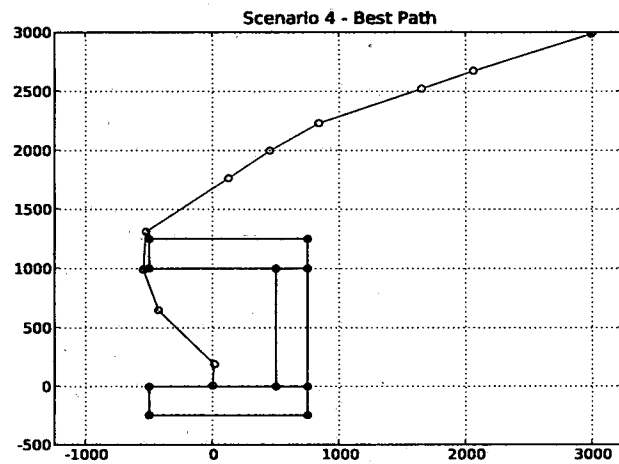


Figure 2.60: Scenario 4 Best Path - Batch Runs

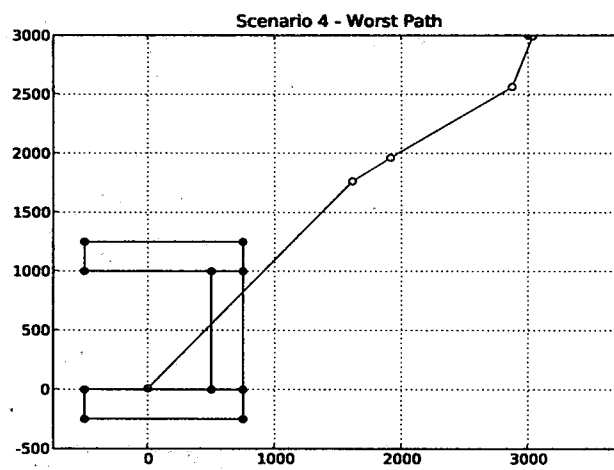


Figure 2.61: Scenario 4 Worst Path - Batch Runs

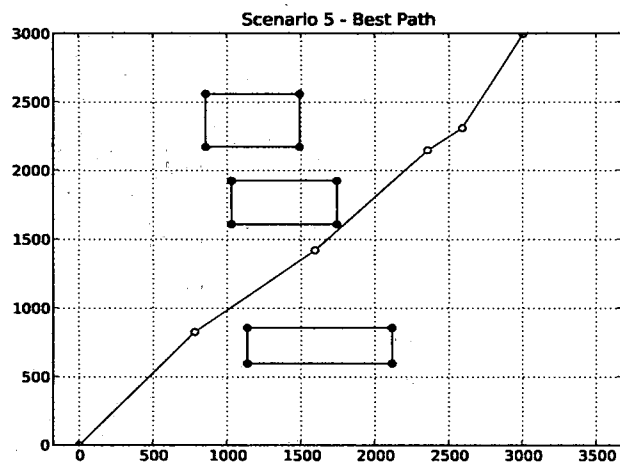


Figure 2.62: Scenario 5 Best Path - Batch Runs

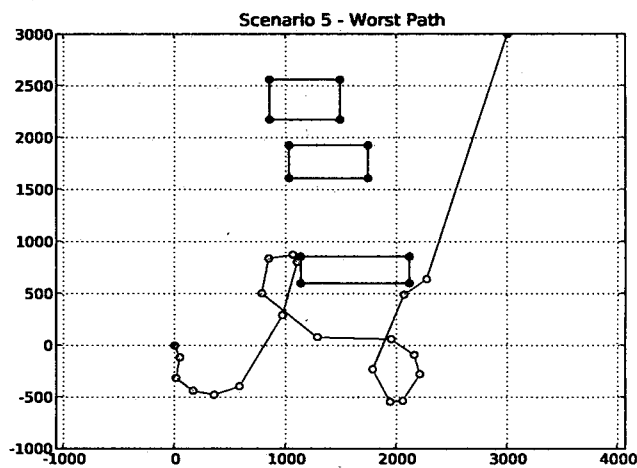


Figure 2.63: Scenario 5 Worst Path - Batch Runs

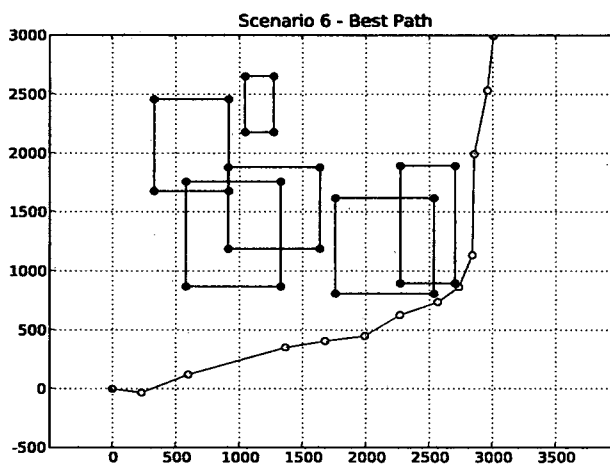


Figure 2.64: Scenario 6 Best Path - Batch Runs

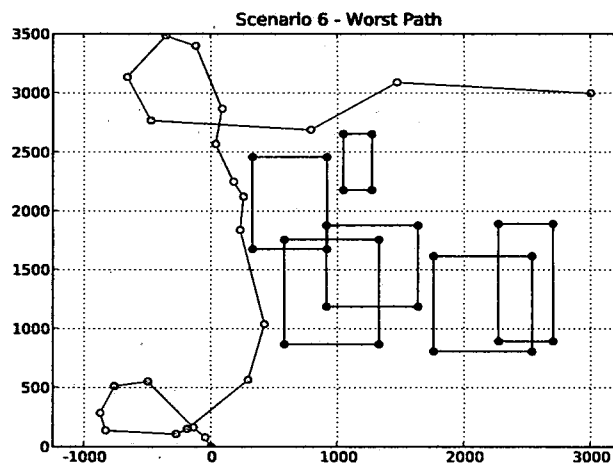


Figure 2.65: Scenario 6 Worst Path - Batch Runs

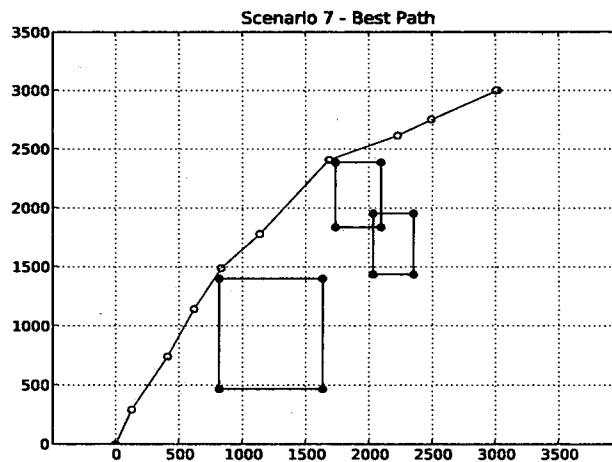


Figure 2.66: Scenario 7 Best Path - Batch Runs

From these plots it can be seen that, although the vast majority of the paths avoid all obstacles, the worst paths are generally unnecessarily long and circuitous. This is a result of the random nature of the algorithm. Given the nature of evolutionary algorithms, it is generally impossible to gain insight into why the algorithm produces some result. Evolutionary algorithms involve complex interactions between all of the components involved: the fitness function along with operators for mutation and selection. This lack of traceability is one disadvantage inherent in evolutionary algorithms. Despite this, the algorithm usually accomplishes the most important objective: avoiding all obstacles.

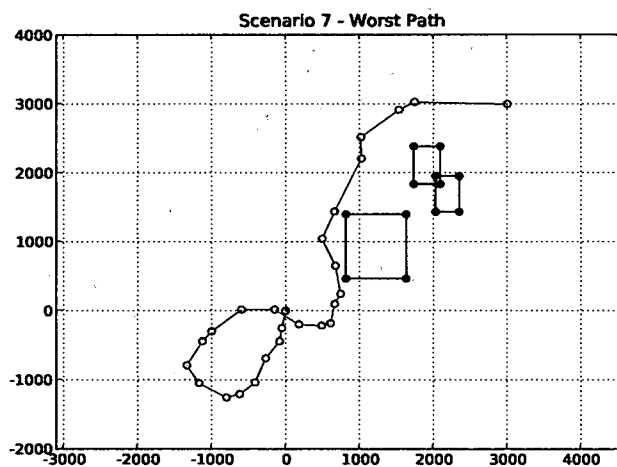


Figure 2.67: Scenario 7 Worst Path - Batch Runs

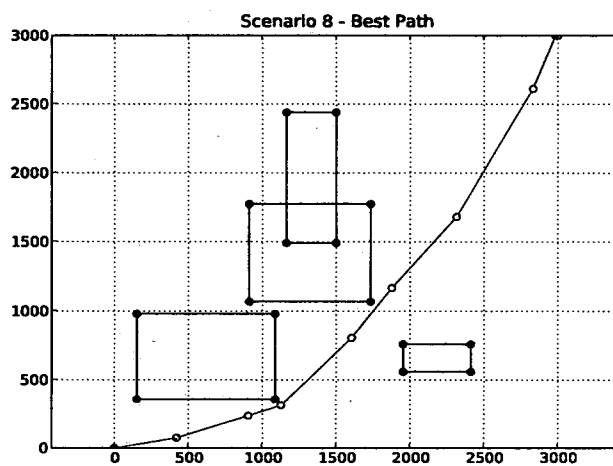


Figure 2.68: Scenario 8 Best Path - Batch Runs

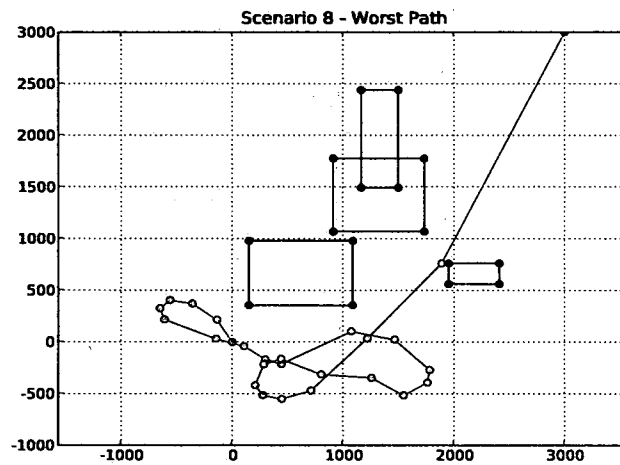


Figure 2.69: Scenario 8 Worst Path - Batch Runs

CHAPTER III

Evolutionary Algorithms for Dynamic Path Planning Scenarios

This chapter describes the development of the reactive planner, which has two components: an evolutionary planner and a reactive obstacle avoidance planner. Additionally, a simple kinematic vehicle model is developed to model the execution of this path. The reactive planner and the vehicle controller run in separate threads and use queues to communicate with each other. The vehicle controller sends state updates and path requests to the planner when necessary. Also, a message is sent to the reactive planner when the vehicle has reached the goal. The reactive planner takes information from the vehicle and computes a path based on the vehicle's state. The computed path is then sent to the vehicle for execution. The following sections describe the vehicle controller and the reactive planner in greater detail.

3.1 Reactive Architecture Design

3.1.1 Vehicle Controller

The purpose of the vehicle controller is to model the execution of the paths produced by the reactive planner. The vehicle model is a simple point mass kinematic model. The position update equations are shown in Equation 3.1.

$$\begin{aligned}x_{n+1} &= x_n + v\Delta t \sin \psi \\y_{n+1} &= y_n + v\Delta t \cos \psi\end{aligned}\tag{3.1}$$

In the position update equations, Δt is the frame time of the simulation (0.005 seconds). The vehicle begins by requesting an initial path from the reactive planner. When it receives the path it begins execution. The vehicle sensor footprint is modelled as a square with the vehicle at the center. The maximum range for the vehicle sensor (defined as the half diagonal of the square) is 250 meters. The same polygon intersection method described in the previous chapter is used to test for intersection between the sensor footprint and all obstacles. To prevent unrealistic turns while avoiding an obstacle, the vehicle is prevented from responding to a detected obstacle intersection for ten seconds of flight time. This ensures that once an avoidance path is given to the vehicle, it follows this path for the minimum path segment time (10 seconds). This assumption may be overly conservative and can result in vehicle-obstacle collisions in situations where the vehicle does not have enough time to avoid an obstacle. This is particularly problematic in scenarios with concave obstacles. It should be noted that this problem exists in the real world as well. There is, in general, no guarantee that a vehicle will be able to avoid a previously unknown obstacle. More sophisticated sensor modelling could improve the performance of the algorithm in this area.

When the sensor detects an obstacle, the vehicle also computes a turn direction based on it's heading and location relative to the obstacle. In order to determine a desirable turn direction, a rule-based system is developed. The area around the obstacle is divided into eight regions and the area around the vehicle is divided into nine regions based on the vehicle heading. Figures 3.1 and 3.2 show these regions.

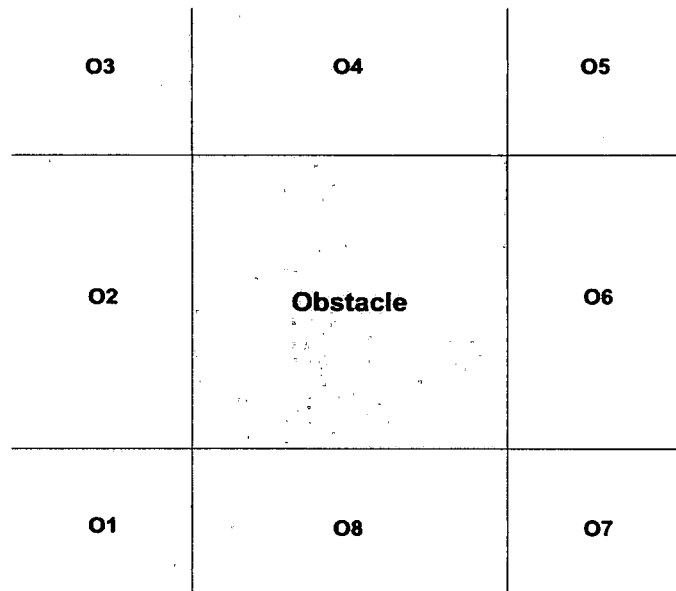


Figure 3.1: Obstacle Regions for Turn Direction Computation

Table 3.1 show the heading angles that define the heading regions and Table 3.2 shows the turn direction rules.

With these turn direction rules, the obstacle intersection method also returns a recommended turn direction based on the vehicle position and heading relative to the obstacle. It should be noted here that the reactive algorithm does require global knowledge of all obstacles for sensing purposes. However, this information is not used for the path planner. Moreover, when a previously unknown

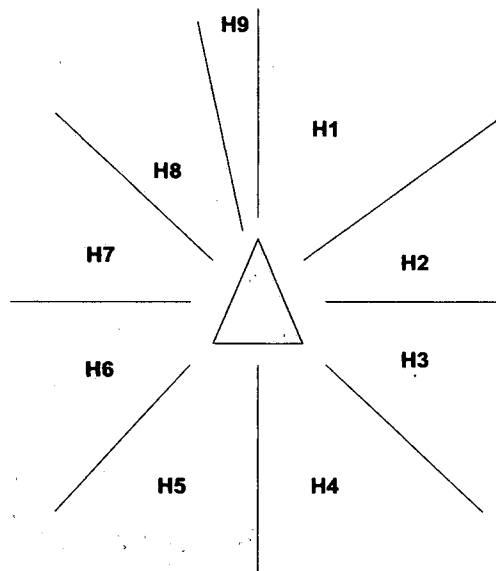


Figure 3.2: Heading Regions for Turn Direction Computation

obstacle is detected in the environment, it is not assumed that the planner knows the full extents of this obstacle. In fact, even after detection, the planner still has no information about this obstacle. This is in contrast current work in which it is assumed that once an obstacle is detected, it is completely known ([47]). Given the relative size of the sensor footprint compared to the obstacle, it may not be reasonable to make this assumption. However, the current approach may be overly conservative. Even if the vehicle does not know the exact location of the obstacle, it is likely reasonable to

Table 3.1: Heading Regions

H_1	$0.0^\circ \leq \psi < 45.0^\circ$
H_2	$45.0^\circ \leq \psi < 90.0^\circ$
H_3	$90.0^\circ \leq \psi < 135.0^\circ$
H_4	$135.0^\circ \leq \psi < 180.0^\circ$
H_5	$-180.0^\circ \leq \psi < -135.0^\circ$
H_6	$-135.0^\circ \leq \psi < -90.0^\circ$
H_7	$-90.0^\circ \leq \psi < -45.0^\circ$
H_8	$-45.0^\circ \leq \psi < -15.0^\circ$
H_9	$-15.0^\circ \leq \psi < 0.0^\circ$

assume that the planner knows roughly that there is an obstacle to the right of the vehicle or above it.

Figure 3.3 shows a flow diagram that describes the logic that governs path requests in the vehicle controller. A path will be requested based on the occurrence of three different events: 1) at the start of the simulation 2) when an obstacle is detected and 3) when an avoidance path is completed. Each time a path is requested the vehicle also sends a state update to the planner to ensure that planner plans a path based on the most current state information.

Table 3.2: Turn Direction Rules

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8	H_9
O_1	Turn Left	Turn Right	No Turn	No Turn	No Turn	No Turn	No Turn	No Turn	Turn Left
O_2	Turn Left	Turn Left	Turn Right	Turn Right	No Turn	No Turn	No Turn	No Turn	Turn Left
O_3	No Turn	No Turn	Turn Left	Turn Right	No Turn	No Turn	No Turn	No Turn	No Turn
O_4	No Turn	No Turn	Turn Left	Turn Left	Turn Right	Turn Right	No Turn	No Turn	No Turn
O_5	No Turn	No Turn	No Turn	No Turn	Turn Right	Turn Left	No Turn	No Turn	No Turn
O_6	No Turn	No Turn	No Turn	No Turn	Turn Left	Turn Left	Turn Right	Turn Right	Turn Right
O_7	No Turn	No Turn	No Turn	No Turn	No Turn	No Turn	Turn Left	Turn Right	Turn Right
O_8	Turn Right	Turn Right	No Turn	No Turn	No Turn	No Turn	Turn Left	Turn Left	Turn Left

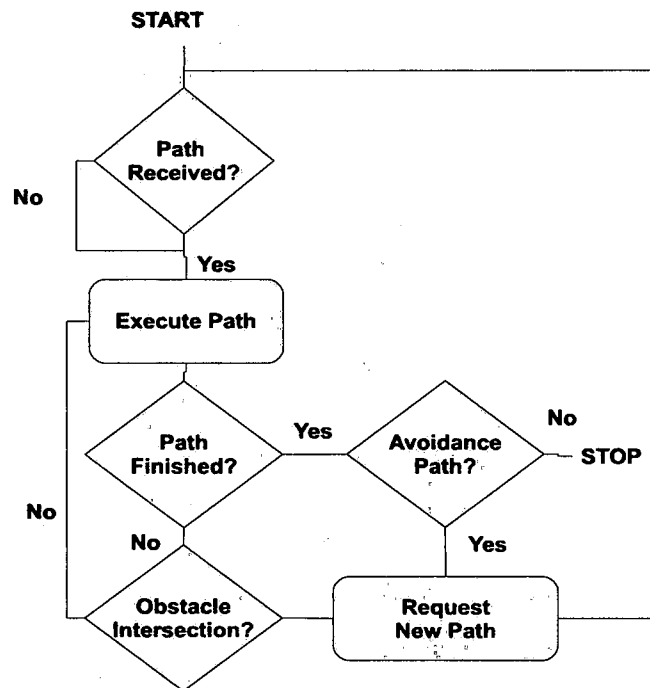


Figure 3.3: Vehicle Controller Flow Diagram

For the purposes of the simulation, the execution of the path is considered to be complete when the vehicle comes within ten meters of the goal position. When this happens, the vehicle alerts the planner that execution of the path is complete and both threads exit. The next section describes the development of the reactive planner.

3.1.2 Reactive Planner

The reactive planner begins by running an evolutionary algorithm for 200 generations to come up with an initial plan for the vehicle. Details of this evolutionary algorithm that differ from the static algorithm described in the previous chapter are discussed in the next section. After the vehicle begins executing the initial path, the planner waits for a path request and a state update from the vehicle before it responds. If the planner receives a path request and the vehicle state indicates that the vehicle has intersected an obstacle and that the turning logic has produced a recommended turn, the planner responds with an avoidance path. In this case, the avoidance path is computed as a single segment path that represents either a left or right turn for the maximum heading change possible. This avoidance path is flown by the vehicle for ten seconds of flight time at a speed of ten meters per second. Avoidance paths are given to the vehicle until it is moving away from the obstacle.

Once the planner has sent an avoidance path to the vehicle, the evolutionary re-planning phase begins. Using the predicted state of the vehicle at the end of the avoidance path, the evolutionary re-planner creates a new population of solutions and evolves them using the same general process as described for the static algorithm. Figure 3.4 gives a flow diagram that shows the logical process behind the reactive planner.

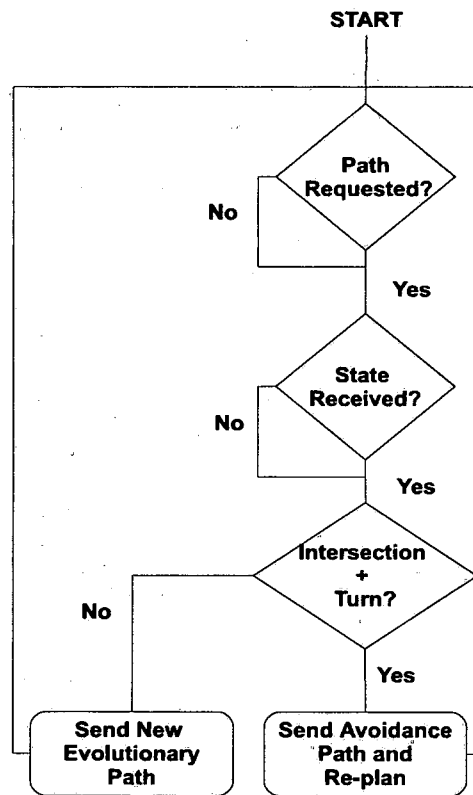


Figure 3.4: Evolutionary Planner Flow Diagram

3.2 Algorithm Description

The evolutionary algorithm used in the reactive planner is similar to the static evolutionary algorithm in many respects. However, there are some differences that should be noted. First, for the initial path produced, the maximum number of segments has been reduced to 20 from 30. This was done to speed up the algorithm given that obstacle intersection test that must occur for each path segment is the most computationally intensive part of the process.

Additionally, the elitist selection operator was used rather than the tournament selection operator. This change was made due to the fact that while the algorithm is running online, we would like to ensure that the best path found is always passed on to the next generation. Given the limited time available for re-planning, this algorithm is biased towards converging quickly rather than exploring the search space more fully. The initialization operator has also been modified to respect the vehicle heading constraints when re-planning paths. The re-planned paths are also further constrained to contain only three path segments. This further increases the speed of the algorithm.

Finally, an additional method was created to address the problem with the last heading in the path. As described in the previous chapter, the last heading in the path is not free to mutate given that the goal location is fixed. As a result, the last path segment would sometimes be infeasible. For the dynamic scenario, an algorithm was developed that would create some number of path segments to append to the evolutionary path that would model a constant turn to the goal heading. Given an arbitrary vehicle state, the algorithm will append headings to the evolutionary path that turns the vehicle towards the goal until it is pointed directly at the goal. Each of these path segments added have a speed of eight meters per second. Because we are modelling a constant turn, the segment time is only one tenth of a second. The vehicle turns in seven degree increments until the turn

towards the goal is less than seven degrees. In this way, the path is guaranteed to be feasible for the reactive case.

3.3 Scenario Descriptions

The scenarios chosen to test the reactive algorithm differ from the static scenarios in just one way. Most scenarios contain at least one obstacle that is designated as unknown to the vehicle. The only exception to this is scenario three, in which there is only a single known obstacle. This scenario was included to test the reactive planner in a situation where all obstacles are known. For the other manually generated scenarios, one obstacle was selected to be unknown. For the randomly generated scenarios, obstacles were randomly set as known or unknown. Figures 3.5 through 3.6 show the reactive test scenarios with the unknown obstacle drawn in red.

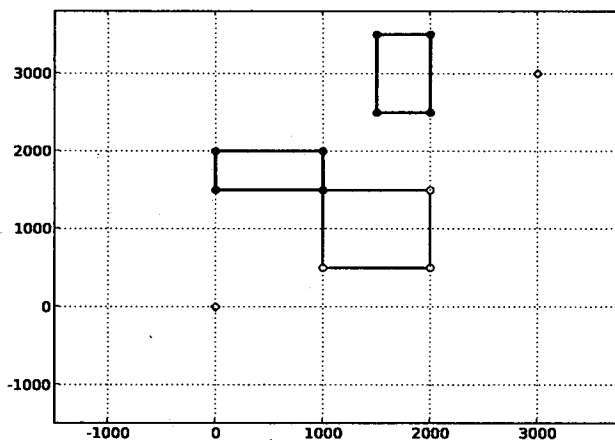


Figure 3.5: Reactive Scenario 1

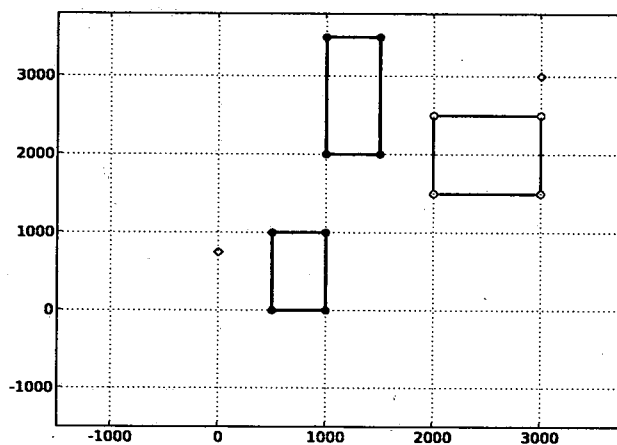


Figure 3.6: Reactive Scenario 2

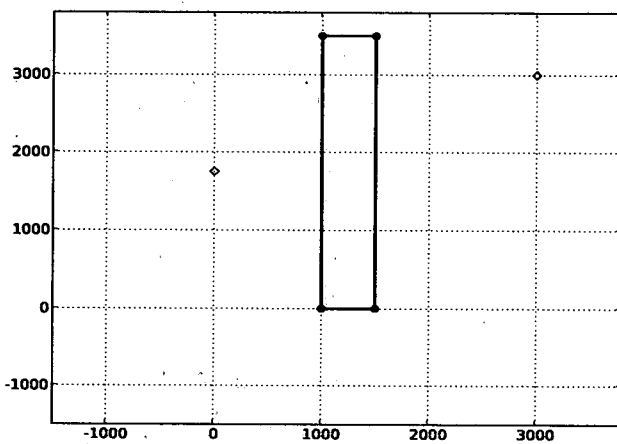


Figure 3.7: Reactive Scenario 3

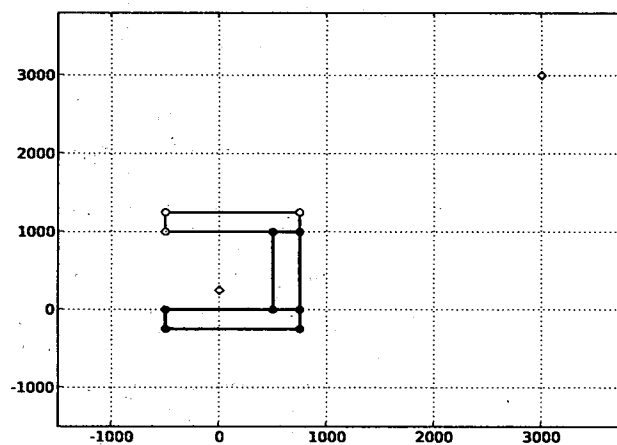


Figure 3.8: Reactive Scenario 4

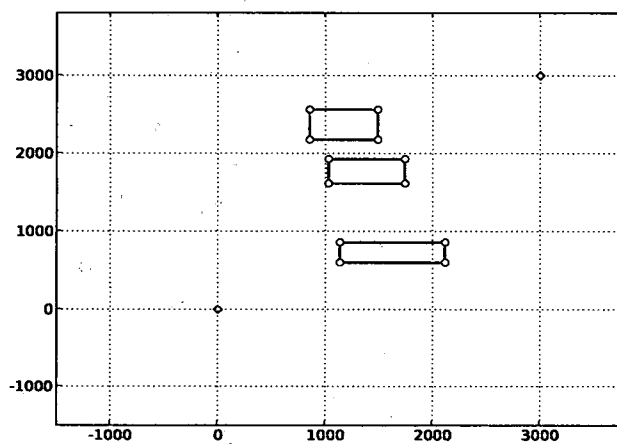


Figure 3.9: Reactive Scenario 5

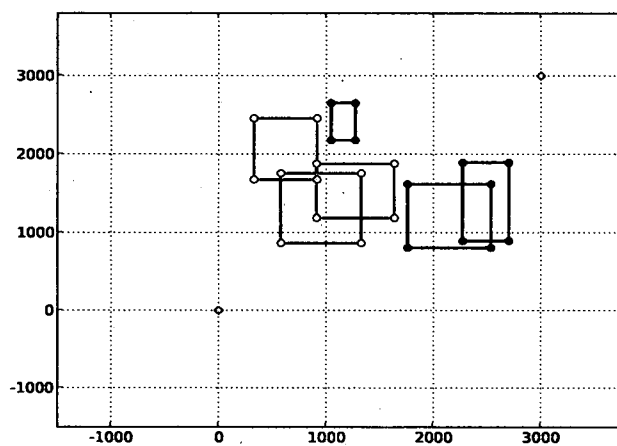


Figure 3.10: Reactive Scenario 6

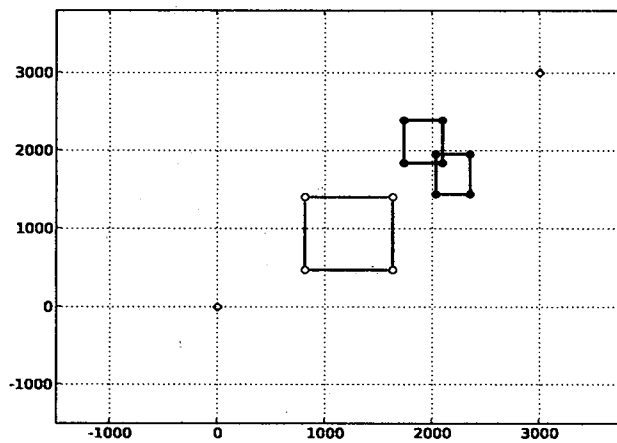


Figure 3.11: Reactive Scenario 7

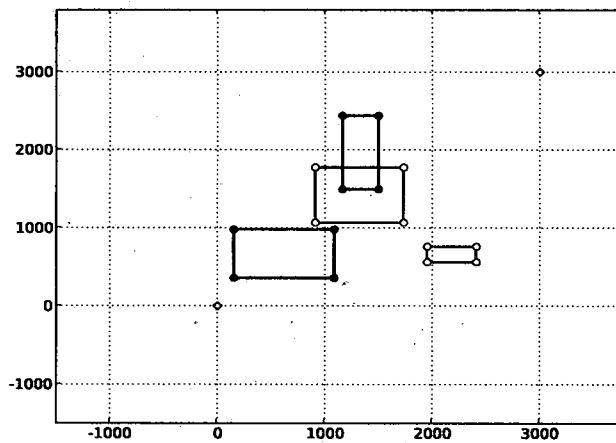


Figure 3.12: Reactive Scenario 8

3.4 Simulation Results

The following sections give results for each scenario. The best and worst path plots are shown along with statistics that describe the overall performance of the algorithm. To compile these statistics, one thousand runs were simulated for each scenario. For the plots shown, the green paths denote the deliberative paths produced by the evolutionary algorithm and the red paths denote the intrusion paths used to avoid obstacles.

3.4.1 Scenario1

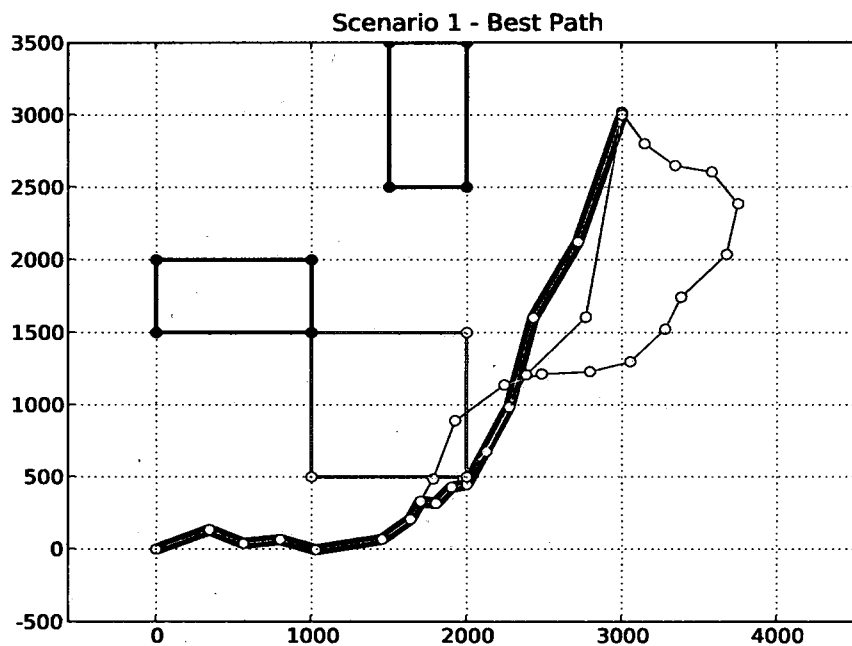


Figure 3.13: Reactive Scenario 1 - Best Path

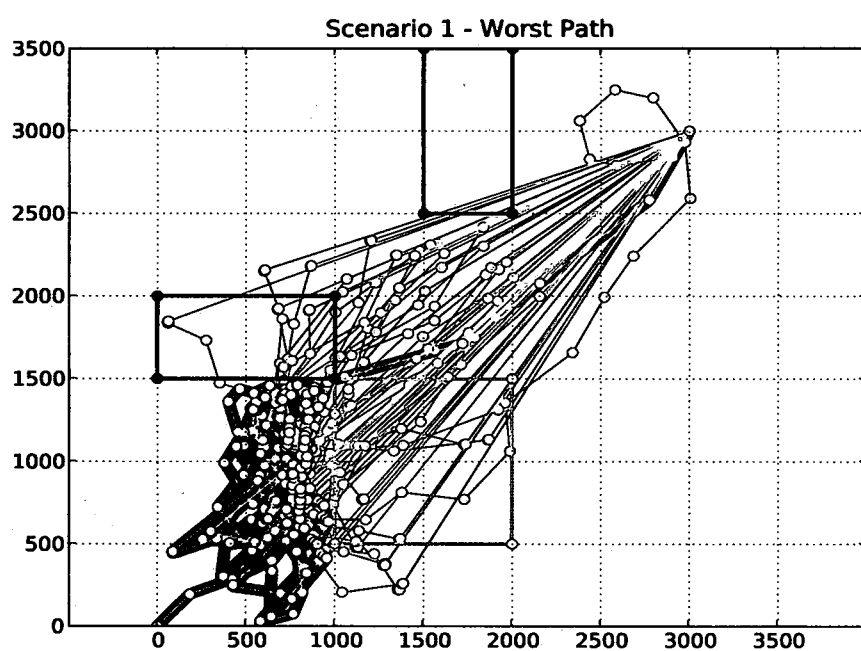


Figure 3.14: Reactive Scenario 1 - Worst Path

3.4.2 Scenario2

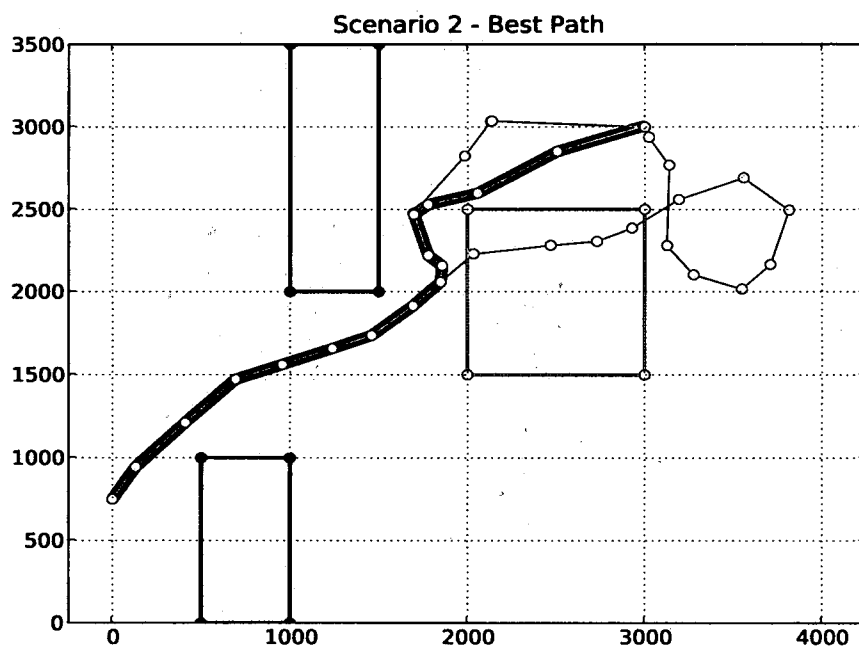


Figure 3.15: Reactive Scenario 2 - Best Path

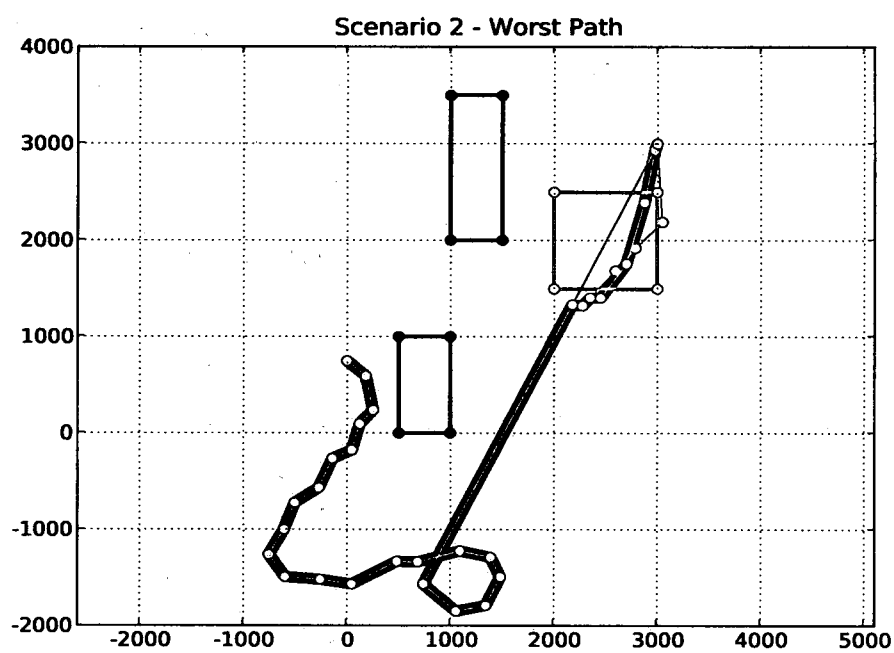


Figure 3.16: Reactive Scenario 2 - Worst Path

3.4.3 Scenario3

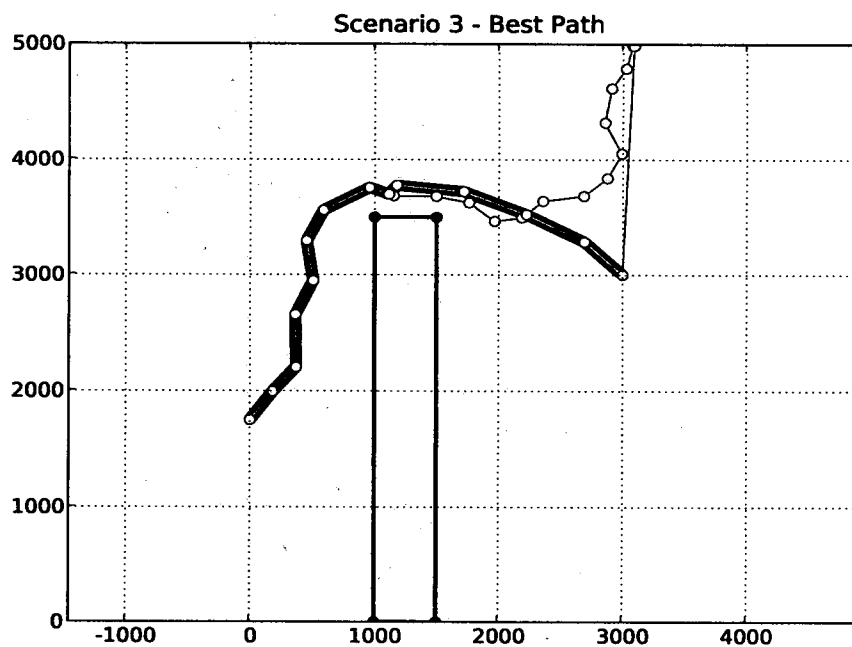


Figure 3.17: Reactive Scenario 3 - Best Path

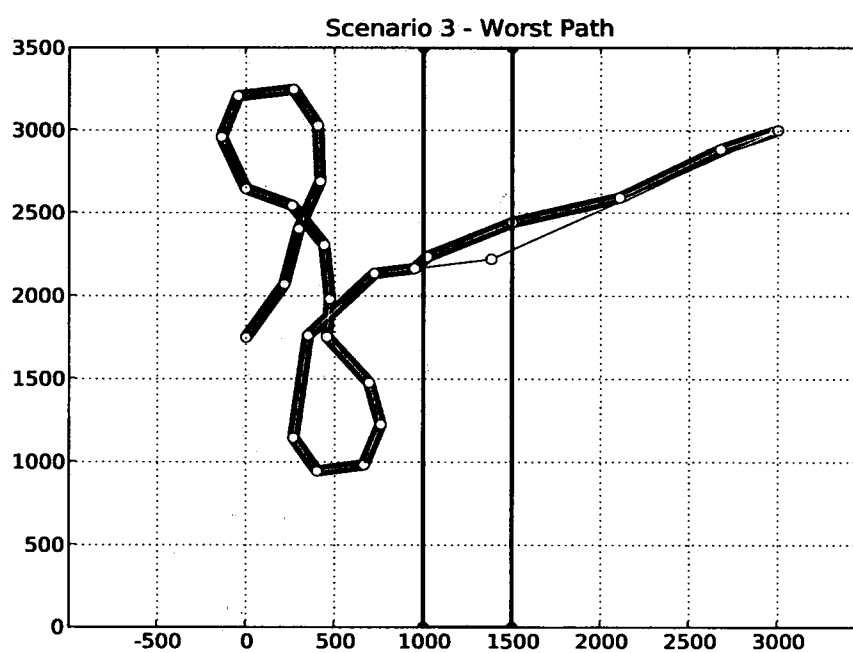


Figure 3.18: Reactive Scenario 3 - Worst Path

3.4.4 Scenario4

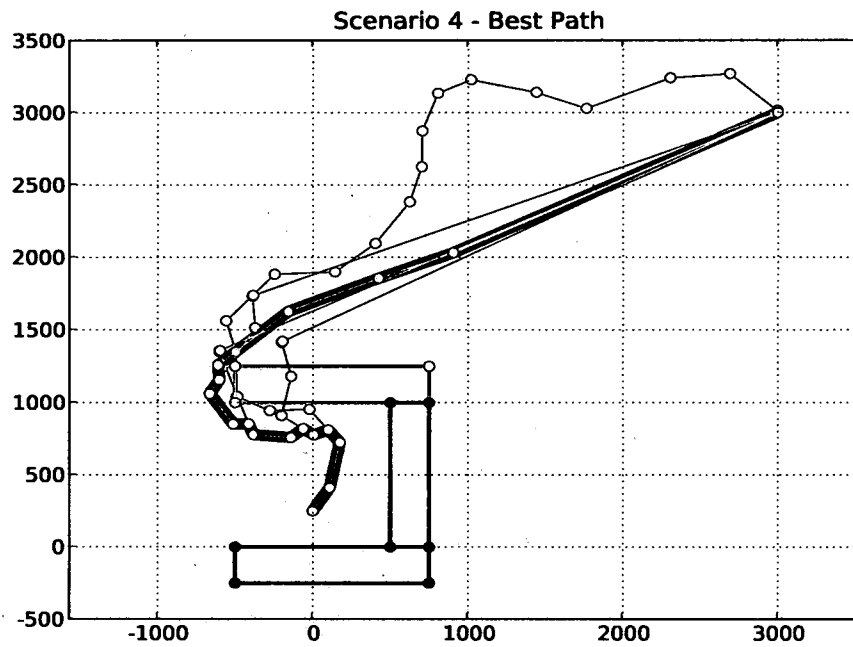


Figure 3.19: Reactive Scenario 4 - Best Path

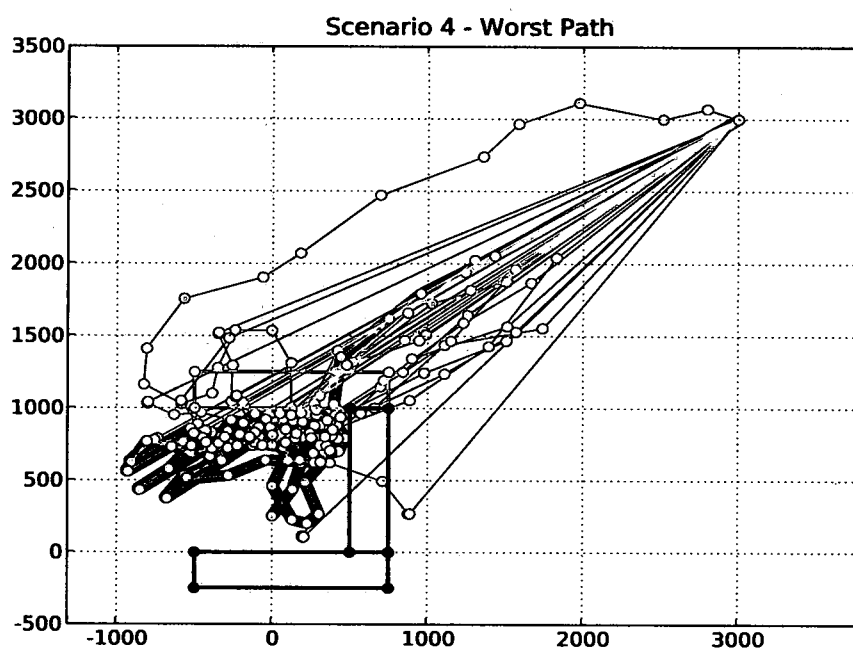


Figure 3.20: Reactive Scenario 4 - Worst Path

3.4.5 Scenario5

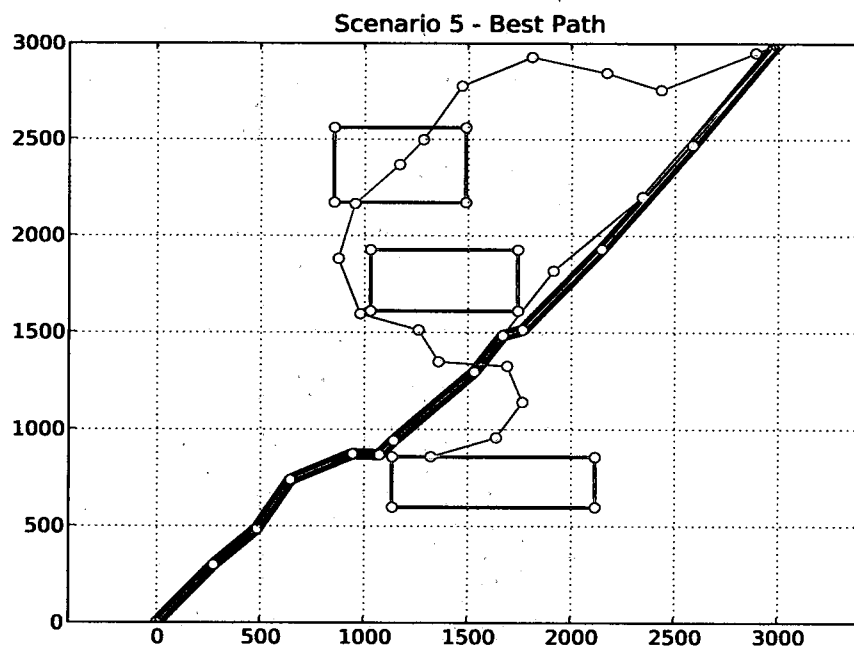


Figure 3.21: Reactive Scenario 5 - Best Path

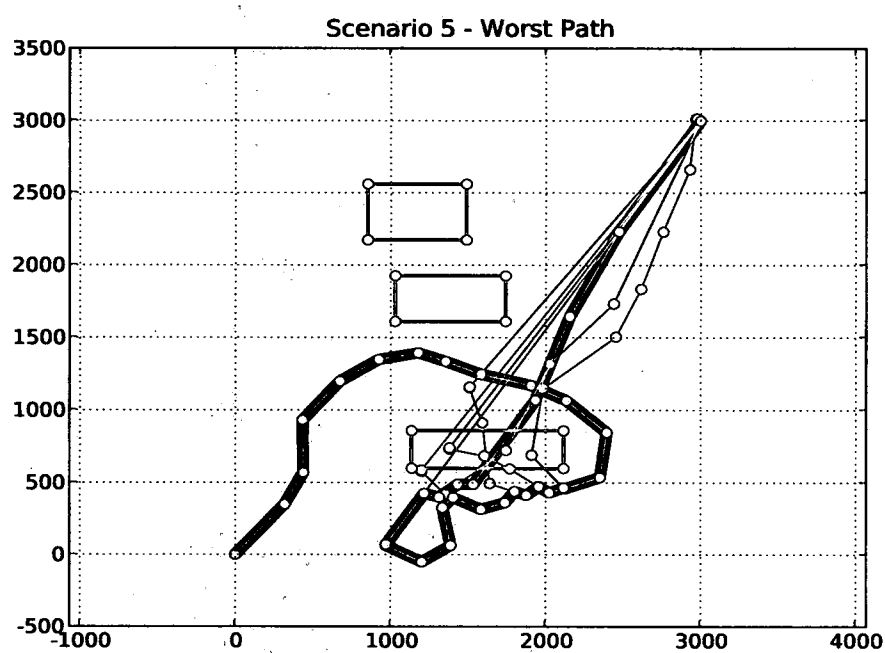


Figure 3.22: Reactive Scenario 5 - Worst Path

3.4.6 Scenario6

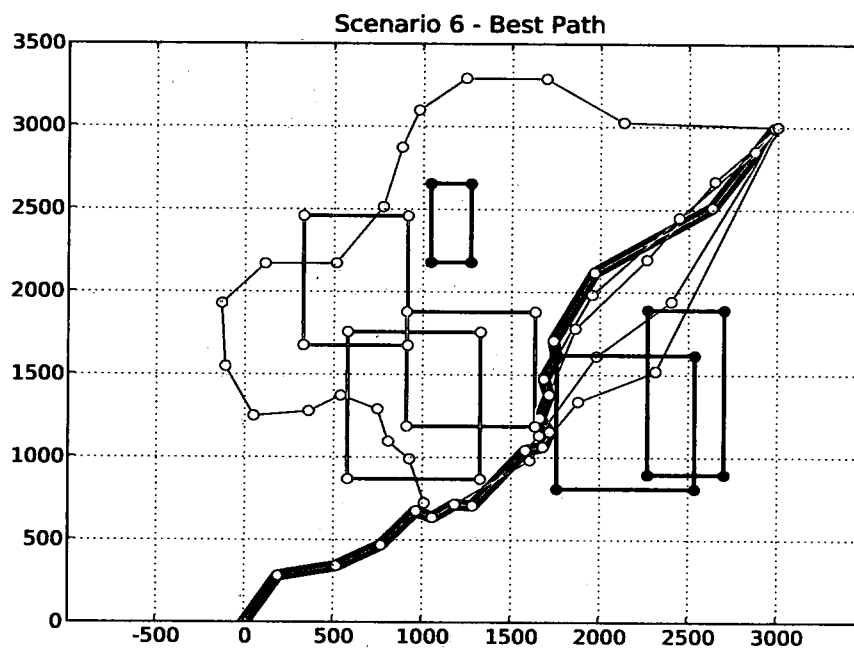


Figure 3.23: Reactive Scenario 6 - Best Path

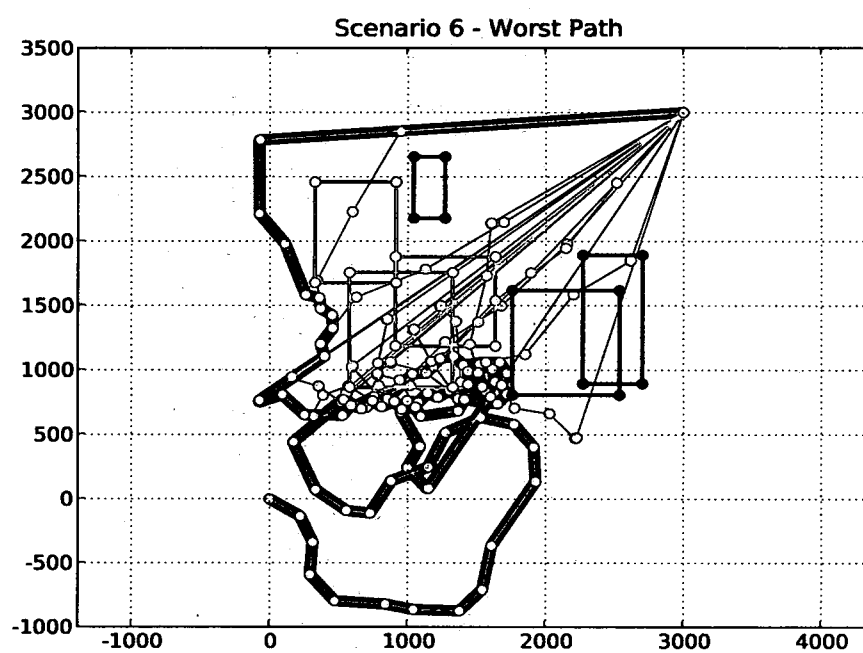


Figure 3.24: Reactive Scenario 6 - Worst Path

3.4.7 Scenario7

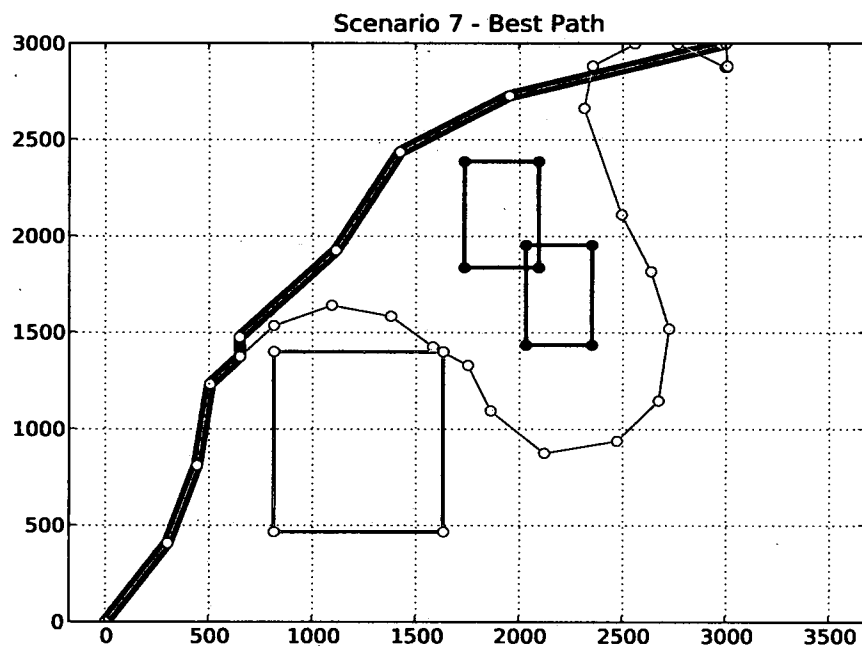


Figure 3.25: Reactive Scenario 7 - Best Path

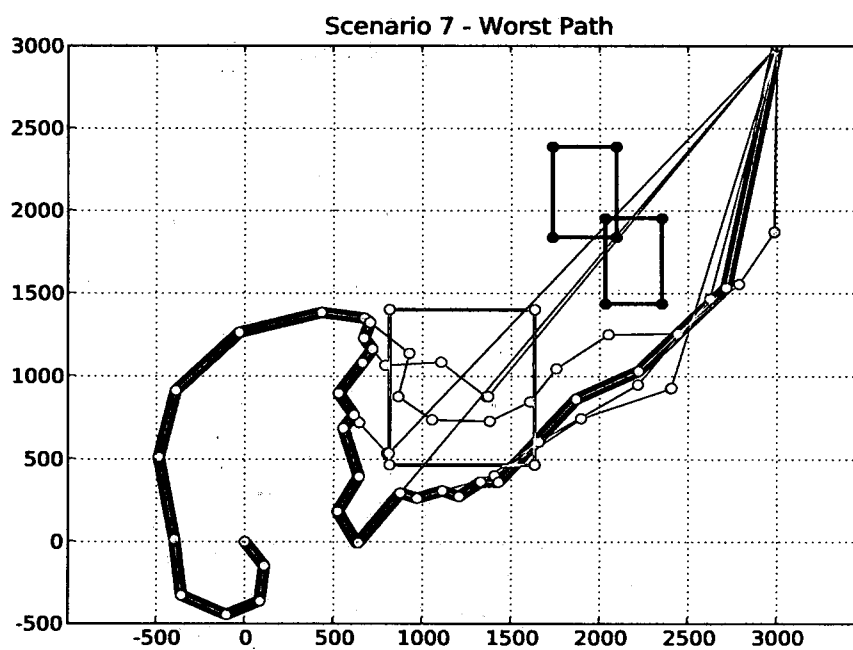


Figure 3.26: Reactive Scenario 7 - Worst Path

3.4.8 Scenario8

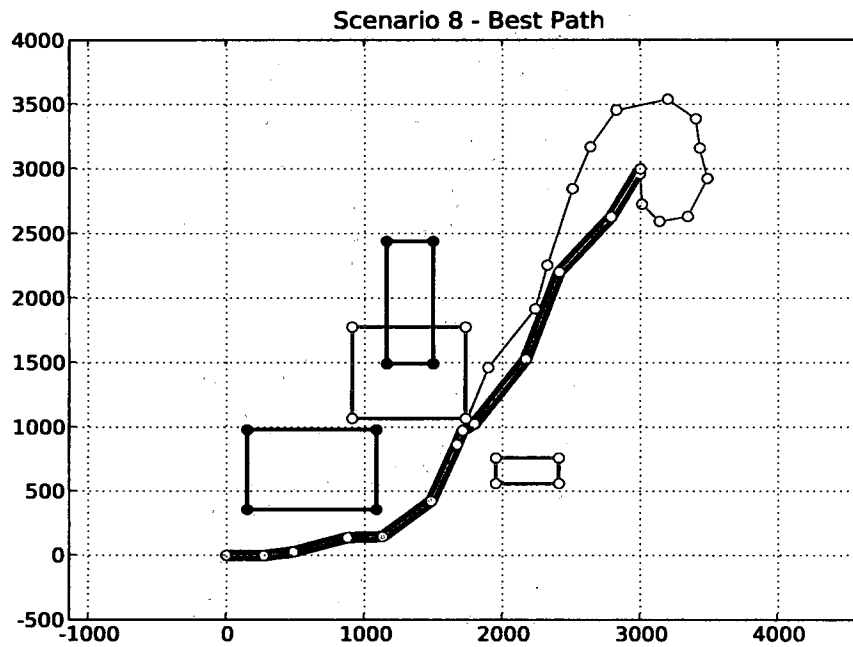


Figure 3.27: Reactive Scenario 8 - Best Path

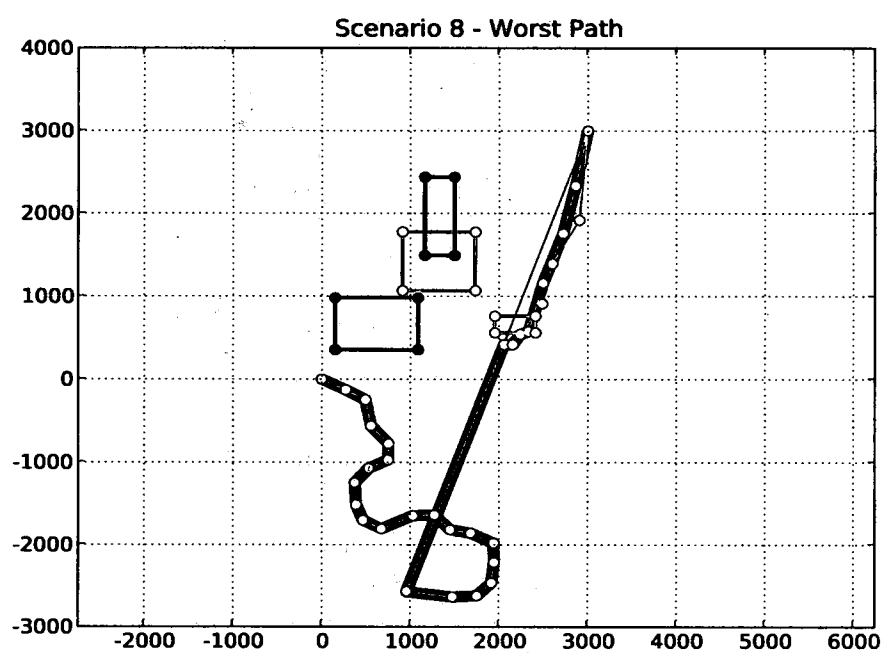


Figure 3.28: Reactive Scenario 8 - Worst Path

Table 3.3: Dynamic Scenarios - Batch Simulation Results

Scenario	Intersections	Min. Fitness	Max. Fitness	Avg. Fitness	Std. Deviation
1	475	-1.0	-0.007	-0.4731	0.4962
2	80	-1.0	-0.006	-0.0810	0.2708
3	5	-1.0	-0.007	-0.0062	0.0704
4	716	-1.0	-0.009	-0.7146	0.4493
5	131	-1.0	-0.006	-0.1317	0.3369
6	318	-1.0	-0.007	-0.3182	0.4644
7	63	-1.0	-0.007	-0.0638	0.2426
8	65	-1.0	-0.007	-0.0659	0.2461

3.4.9 Discussion and Batch Results

For the batch runs, it should be noted that a different notion of obstacle intersection is used to evaluate the fitness of the overall path. Whereas the evolutionary algorithm uses the bounding box approach for determining intersection, for the final reactive paths a point-in-polygon check is performed for each vehicle position to determine whether or not the vehicle actually collided with an obstacle. This is due to the fact that nearly all runs involve a collision between the sensor footprint and an obstacle. In fact, the purpose of these runs is to test the algorithm's response to this event. Table 3.3 shows the results. Again, the data is normalized based on the minimum fitness value so all values are on the range $[-1, 0]$.

Again, it can be seen from the data that the algorithm produces sets of paths with large variance in the fitness values. The exception to this is Scenario 3, in which there is only a single known obstacle. For the reactive scenarios, it is clear that concave obstacles present a very challenging problem for this algorithm. Both scenarios one and four contain a concave obstacle and both show a high percentage of obstacle intersections. It is clear that it is easy for the vehicle to be trapped

inside a concave obstacle. Moreover, in both scenarios one and four the planner only has partial knowledge of the concave obstacle, making the problem that much more difficult. In the other scenarios, however, obstacle intersections are generally less than 20% (with the exception being scenario six which has an intersection rate of 31.8%). In the real world there is no guarantee that a vehicle will be able to avoid an obstacle once detected. Although changes could be made to this algorithm to improve performance, the main contribution is the development of a realistic, multi-threaded simulation that can incorporate known obstacles into a vehicle path planner while adjusting the path to account for unknown obstacles. In addition to this, the re-planning time is inherently limited by the motion of the vehicle. There is no assumption made regarding the amount of time available for planning. Indeed, one of the main benefits of an evolutionary approach is the availability of multiple solutions at any step in the algorithm. The best solution at any point can be extracted and sent to a vehicle for execution. Although the quality of these solutions might be degraded due to the limited re-planning time, the algorithm is able to respond quickly and the reactive controller will still be used to avoid any obstacles.

CHAPTER IV

Conclusions

4.1 Discussion of Results

The static evolutionary algorithm developed was, for the most part, successful at creating good paths that avoided obstacles. Only in two cases did the bounding box surrounding the planned path intersect with obstacles. However, a large variation in fitness values was observed over one thousand trial runs. This indicates that the algorithm does a poor job of locating one optimal solution over time. For our purposes, however, this is acceptable given that the most important metric (obstacle avoidance) was generally satisfied.

The reactive evolutionary algorithm introduces new challenges for the planner. In these simulations, the time available for re-planning becomes of paramount importance. Also, the presence of unknown obstacles that the planner never has any information for makes the problem harder. This is especially true when the unknown obstacle is part of a concave obstacle. In these scenarios the vehicle would often become trapped and the vehicle would often be unable to avoid the obstacle.

4.2 Comparison with Previous Work

Several portions of this work improve on the existing research in some way. First, in [47] when an unknown obstacle is detected it is assumed that it is perfectly known for the planner. In this work,

no such knowledge is assumed for the reactive case. In general, the reactive scenarios involve some combination of known and unknown obstacles. In one scenario, a single known obstacle exists in the environment. This was included to test the reactive algorithm's ability to deal with environments in which all obstacles are known. In this case the algorithm performed well. In scenario five, all obstacles in the scenario are unknown. In this case, the vehicle collides with an obstacle 13% of the time. This represents an improvement over the previous literature in that obstacles which are completely unknown can be accounted for in many situations.

Additionally, this work allows the evolutionary algorithm to operate for several generations before producing a path for the vehicle to follow after an obstacle intersection. The addition of a separate reactive controller that responds to such situations allows this. Although the difference in use of obstacle knowledge in [47] makes comparison difficult, it is believed that this planning period, though limited, is beneficial in terms of finding good paths.

It is believed that algorithms which incorporate both global and local information are capable of providing more robust solutions to the path planning problem. In cases where obstacle information is known, a purely reactive algorithm may unnecessarily encounter obstacles that a deliberate planner would be capable of avoiding. When considering the opposite situation, when no obstacle information is known, purely deliberate planners are completely incapable of finding a good solution. The algorithm developed in this work constitutes a compromise between the two approaches: known information is exploited while new information is accounted for reactively.

4.3 Future Work

Several enhancements could be made to the algorithms described herein that may improve their performance. The performance of any evolutionary algorithm depends greatly on choices made for

the fitness function, mutation and selection operators, the solution representation and the various parameters that control the strength of these operators. Tuning these various parameters could be undertaken as a research project in and of itself and may result in better performance.

Also, the reactive evolutionary algorithm could be improved through the use of speed control. In many instances where an obstacle intrusion occurred, this was due to the fact that the vehicle was moving too fast to respond before the intrusion. The current algorithm has no mechanism for adjusting the speed of path segments while leaving the rest of the path unchanged. If the vehicle could be made to reduce its speed in some circumstances, it may have more time to evade an obstacle. Additionally, the use of rudimentary obstacle knowledge for unknown obstacles in the planner would also improve performance. Often times the evolutionary planner responds with a path that turns the vehicle back into an unknown obstacle it has just sensed. This could be changed by modifying the path initialization function so that it is biased towards producing paths that move away from these obstacles.

Finally there are many other areas that can be looked at for future research. These include: the consideration of moving targets, cooperatively co-evolving paths for multiple vehicles and the inclusion of more complicated tasks like searching an area or along a given route.

APPENDIX A

Additional Static Scenario Results

This appendix includes several additional plots of generated paths for each scenario. Also, for scenarios two through eight, the fitness histogram is given. It should be noted that, for the purposes of the histogram, the two intersection runs from scenario four have been omitted.

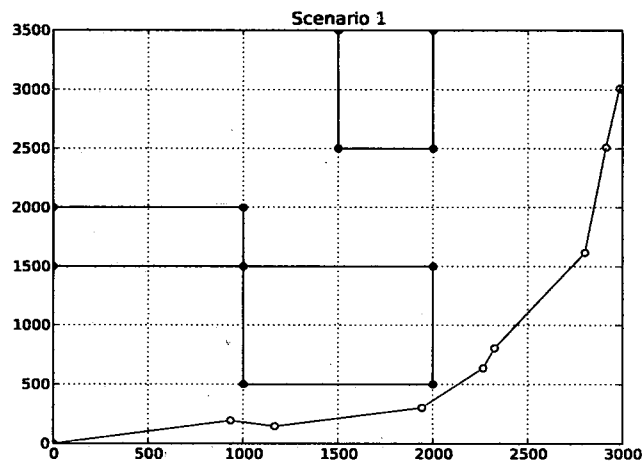


Figure A.1: Scenario 1 Path - 1

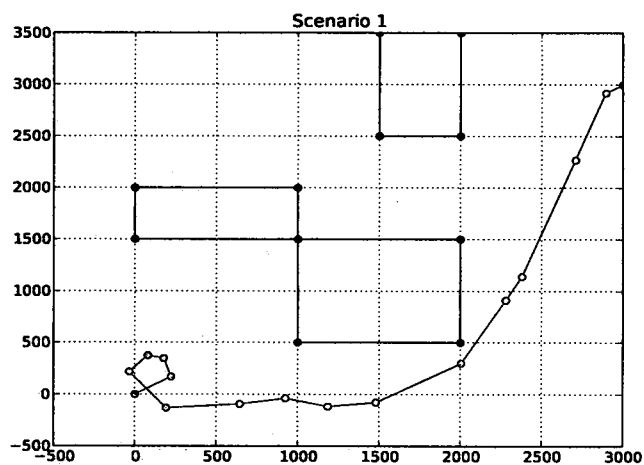


Figure A.2: Scenario 1 Path - 2

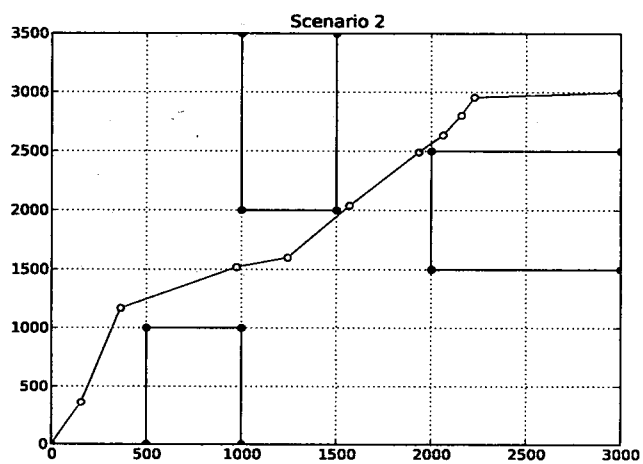


Figure A.3: Scenario 2 Path - 1

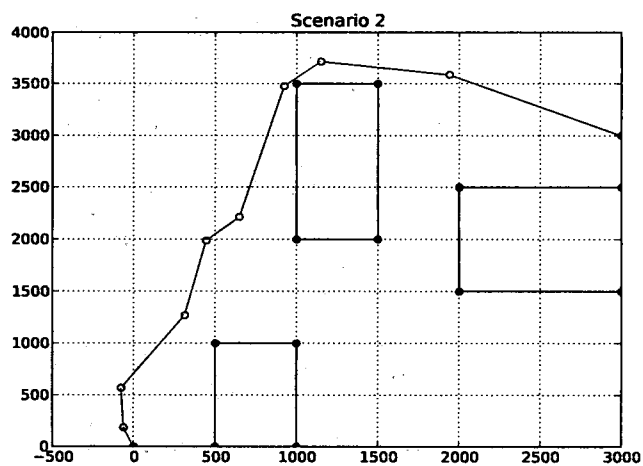


Figure A.4: Scenario 2 Path - 2

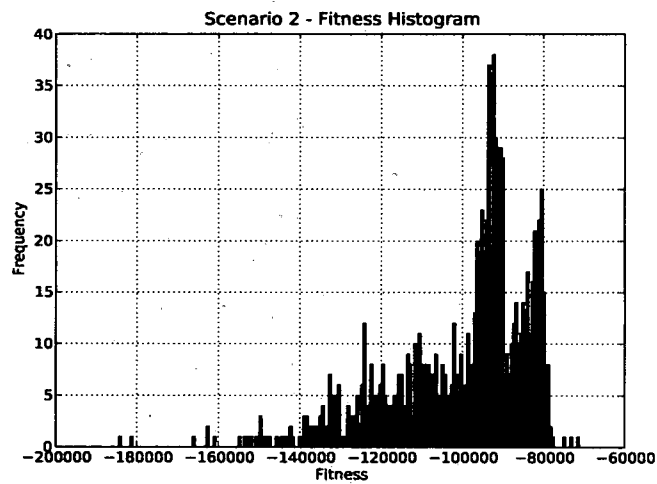


Figure A.5: Scenario 2 Fitness Histogram

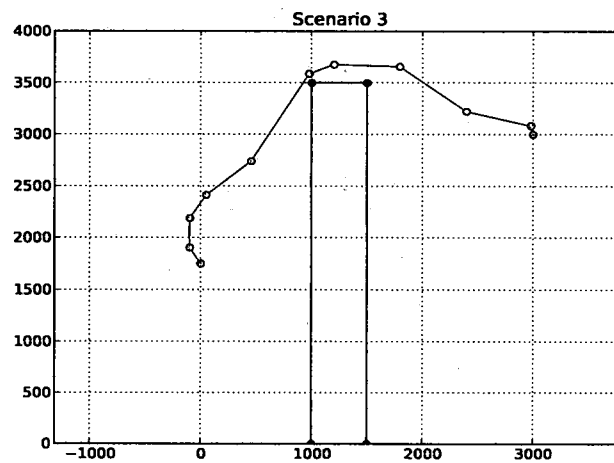


Figure A.6: Scenario 3 Path - 1

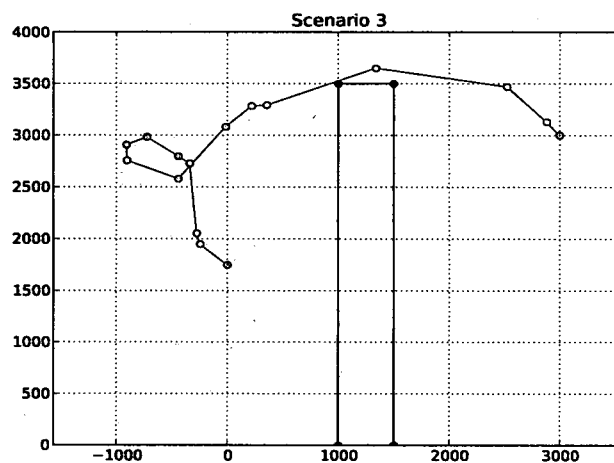


Figure A.7: Scenario 3 Path - 2

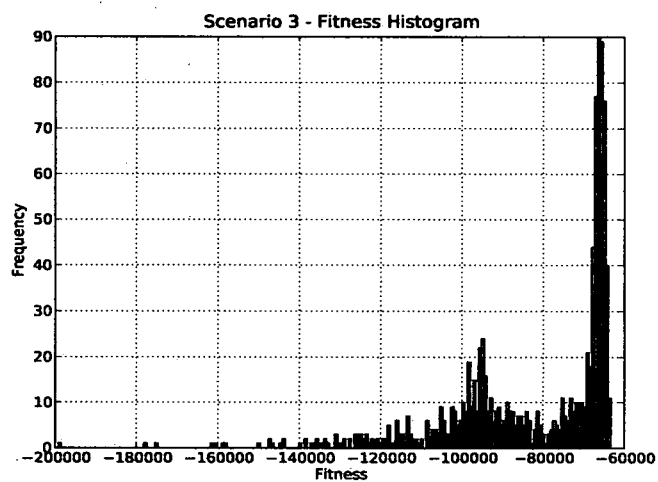


Figure A.8: Scenario 3 Fitness Histogram

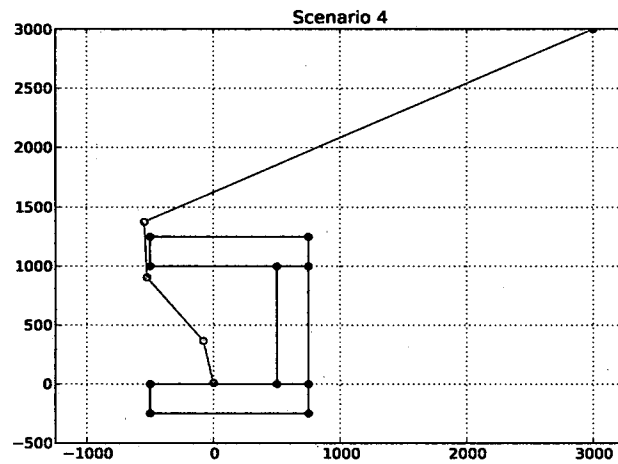


Figure A.9: Scenario 4 Path - 1

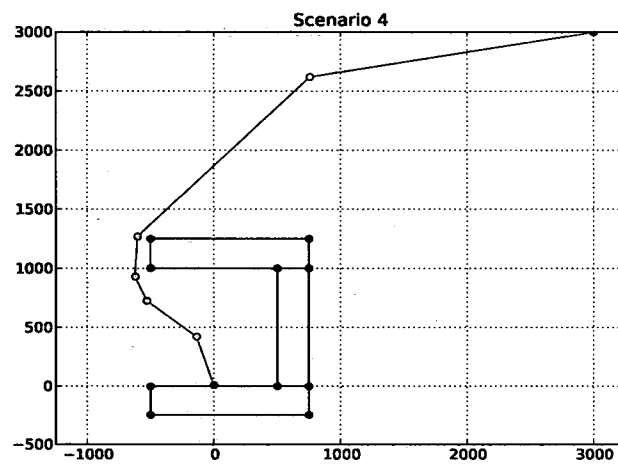


Figure A.10: Scenario 4 Path - 2

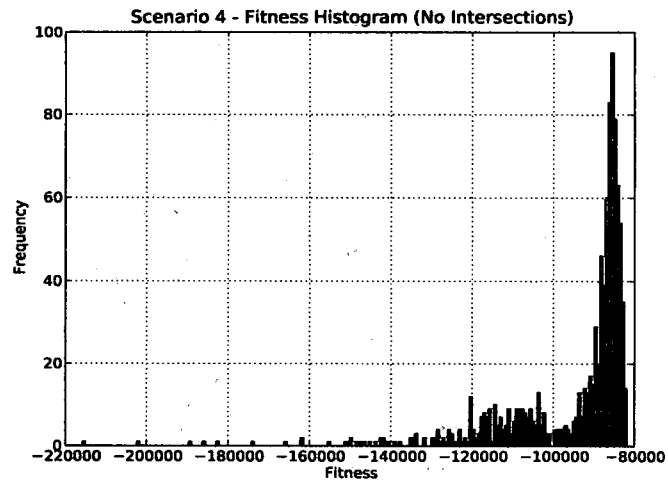


Figure A.11: Scenario 4 Fitness Histogram - No Obstacles

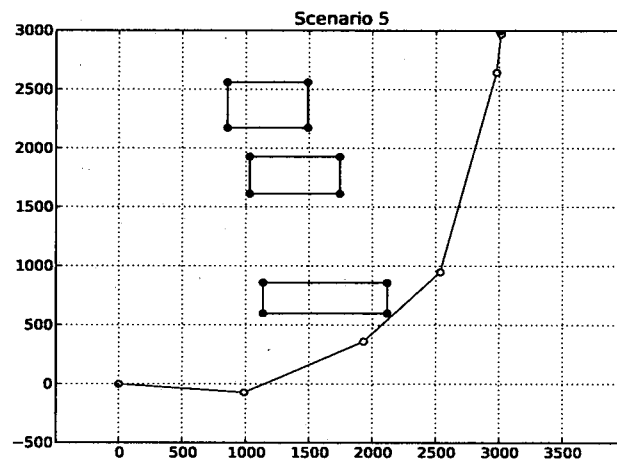


Figure A.12: Scenario 5 Path - 1

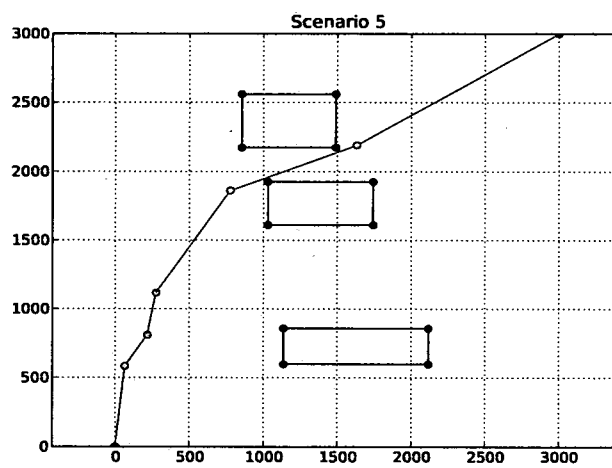


Figure A.13: Scenario 5 Path - 2

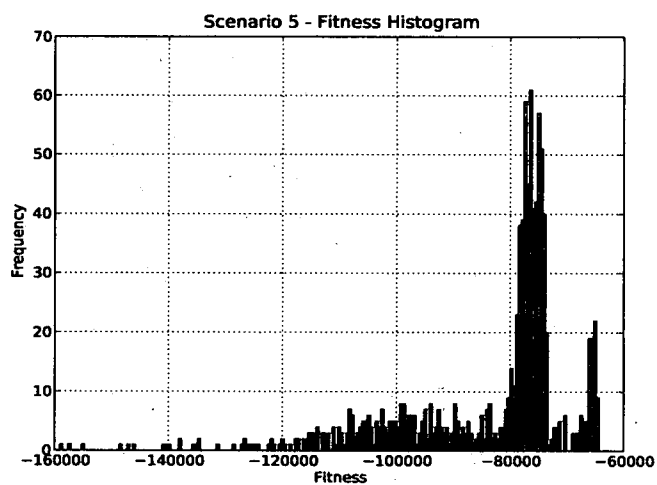


Figure A.14: Scenario 5 Fitness Histogram

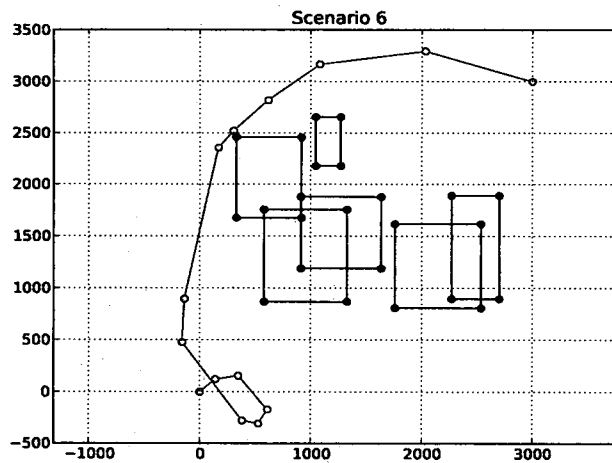


Figure A.15: Scenario 6 Path - 1

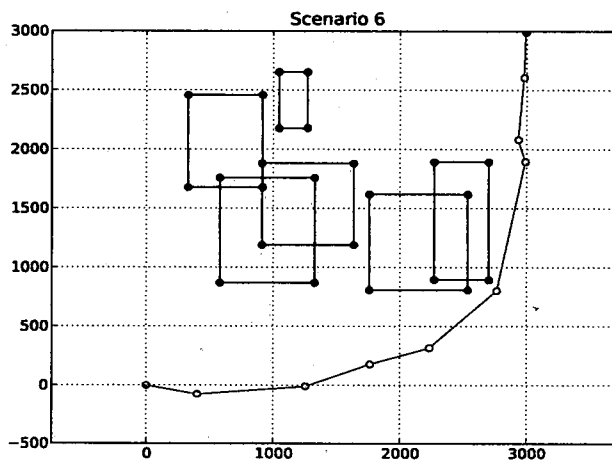


Figure A.16: Scenario 6 Path - 2

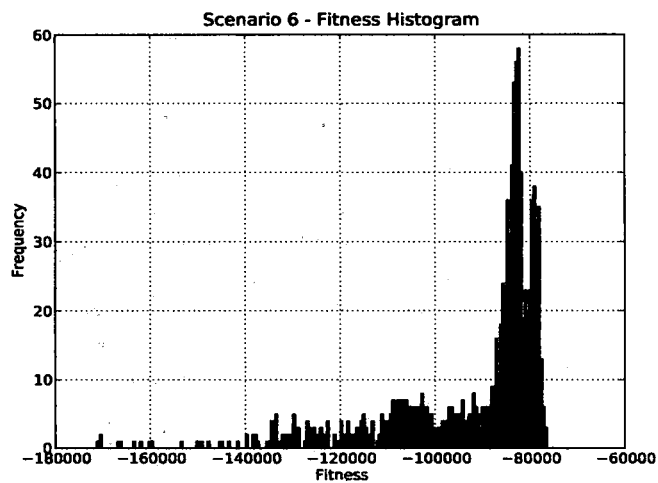


Figure A.17: Scenario 6 Fitness Histogram

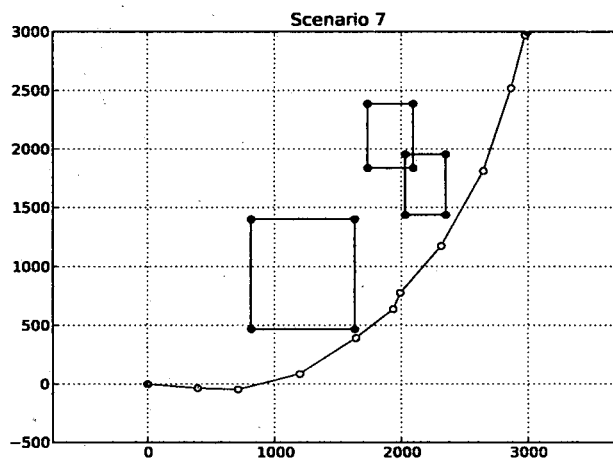


Figure A.18: Scenario 7 Path - 1

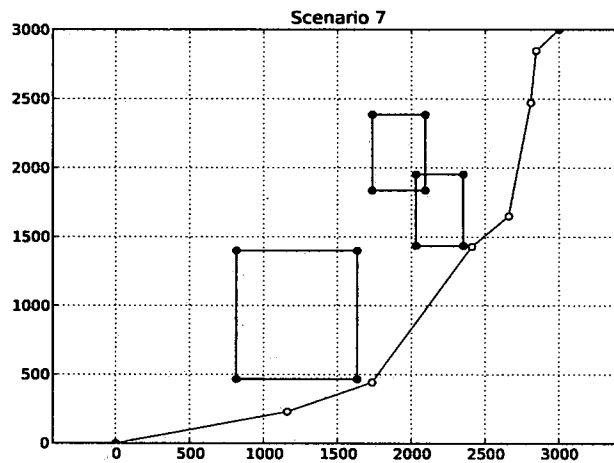


Figure A.19: Scenario 7 Path - 2

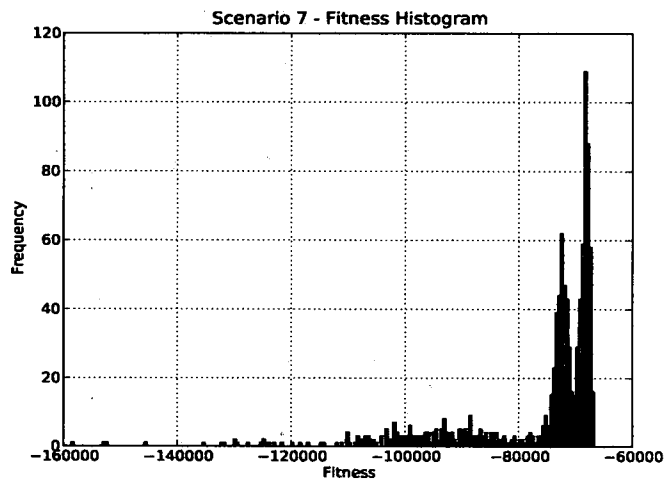


Figure A.20: Scenario 7 Fitness Histogram

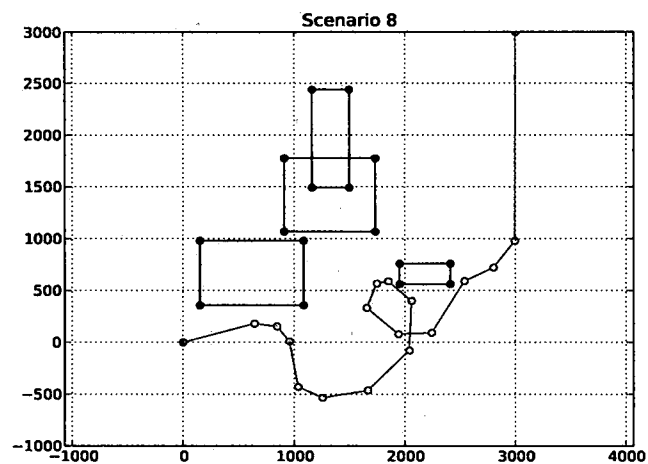


Figure A.21: Scenario 8 Path - 1

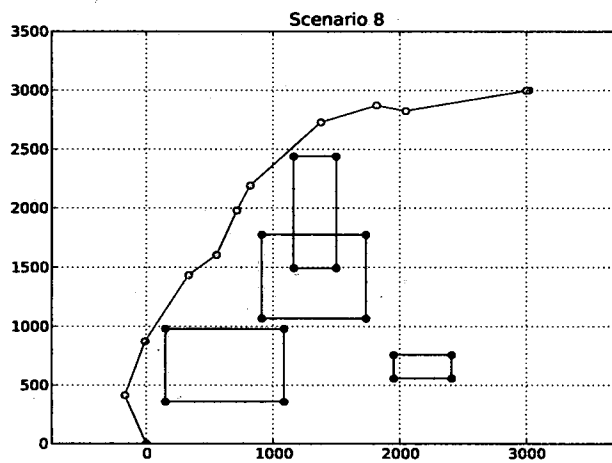


Figure A.22: Scenario 8 Path - 2

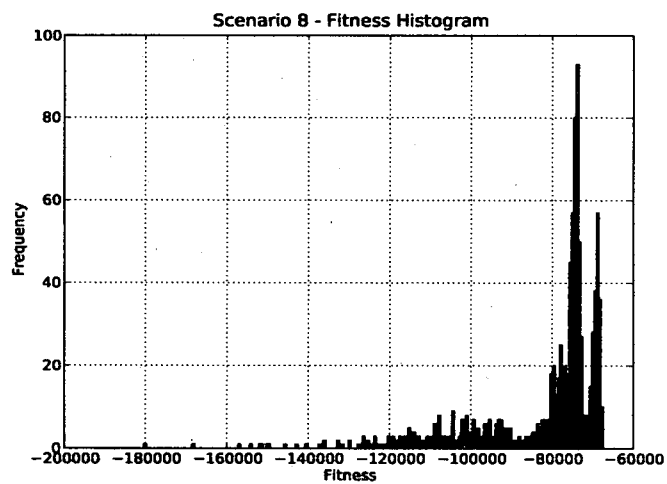


Figure A.23: Scenario 8 Fitness Histogram

APPENDIX B

Additional Dynamic Scenario Results

Additional results from the reactive simulations are shown in this appendix.

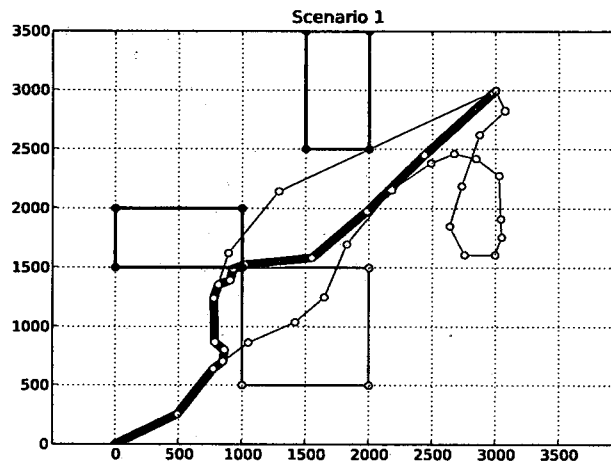


Figure B.1: Reactive Scenario 1 Path - 1

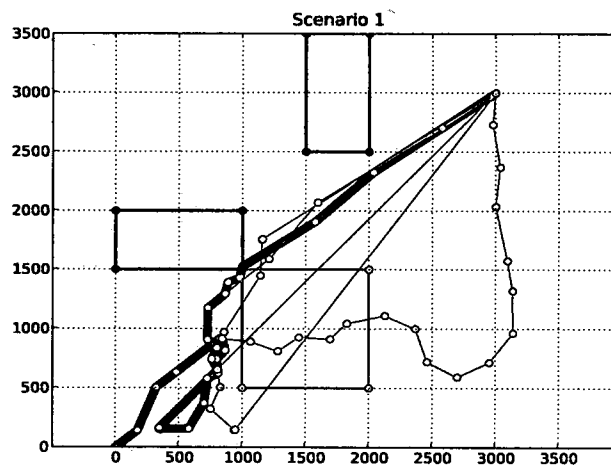


Figure B.2: Reactive Scenario 1 Path - 2

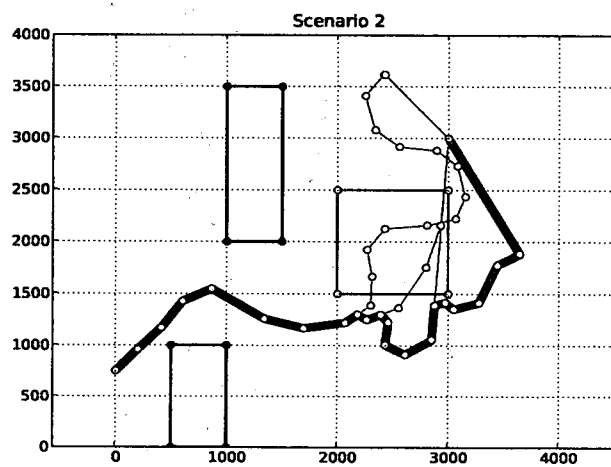


Figure B.3: Reactive Scenario 2 Path - 1

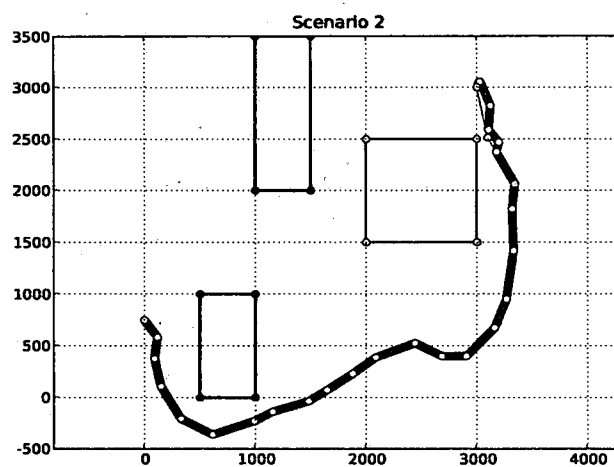


Figure B.4: Reactive Scenario 2 Path - 2

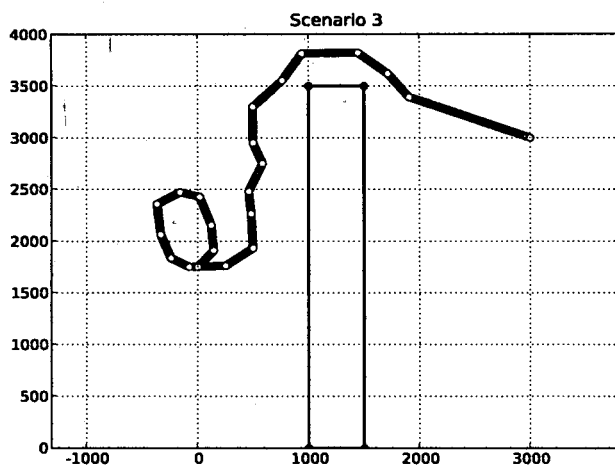


Figure B.5: Reactive Scenario 3 Path - 1

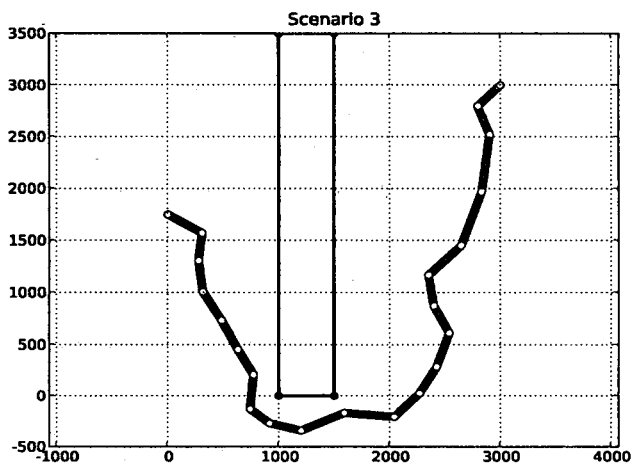


Figure B.6: Reactive Scenario 3 Path - 2

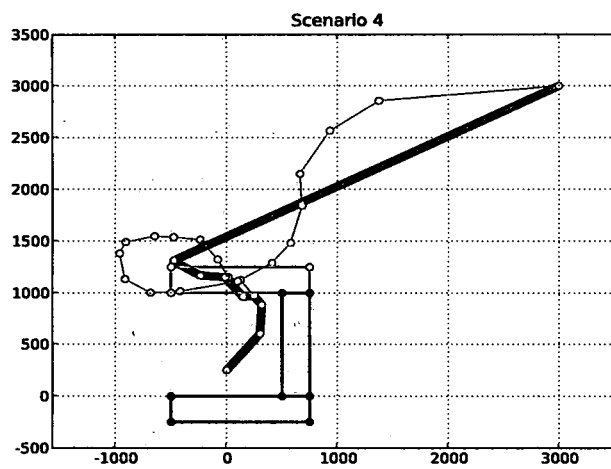


Figure B.7: Reactive Scenario 4 Path - 1

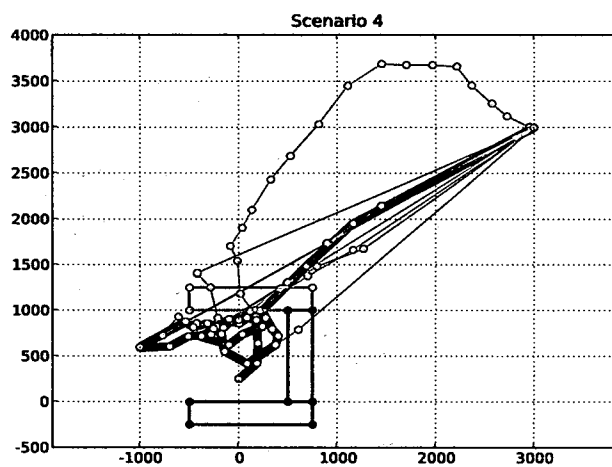


Figure B.8: Reactive Scenario 4 Path - 2

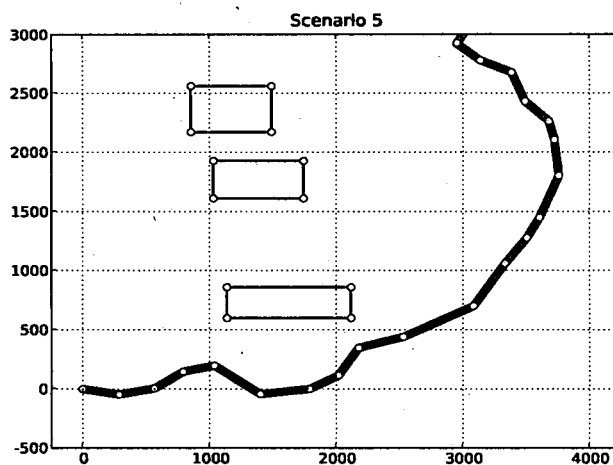


Figure B.9: Reactive Scenario 5 Path - 1

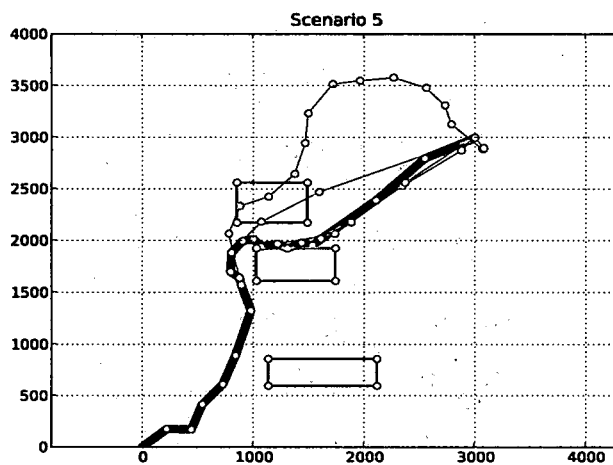


Figure B.10: Reactive Scenario 5 Path - 2

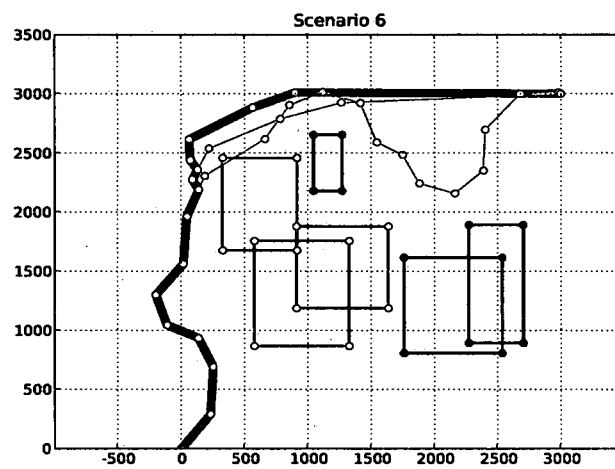


Figure B.11: Reactive Scenario 6 Path - 1

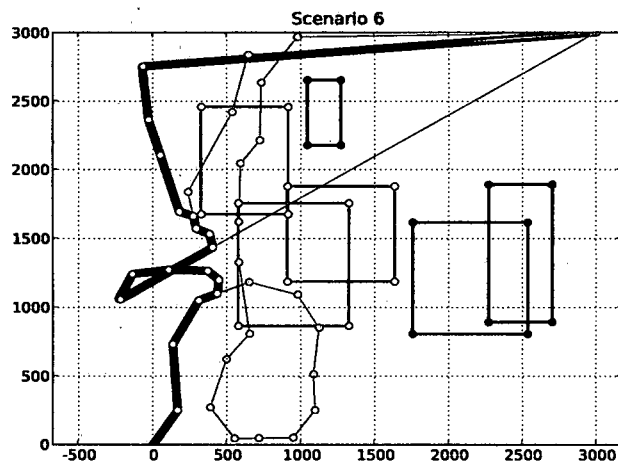


Figure B.12: Reactive Scenario 6 Path - 2

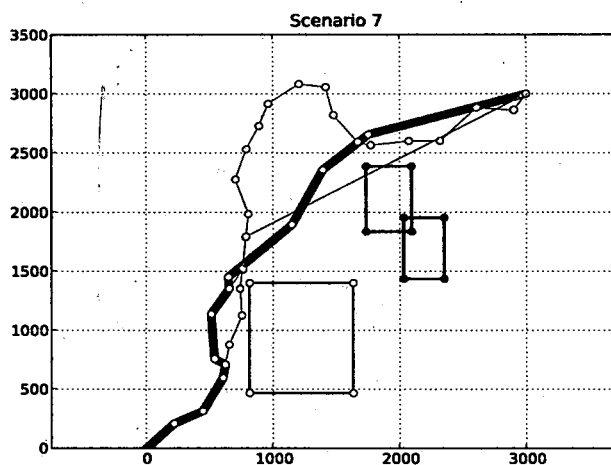


Figure B.13: Reactive Scenario 7 Path - 1

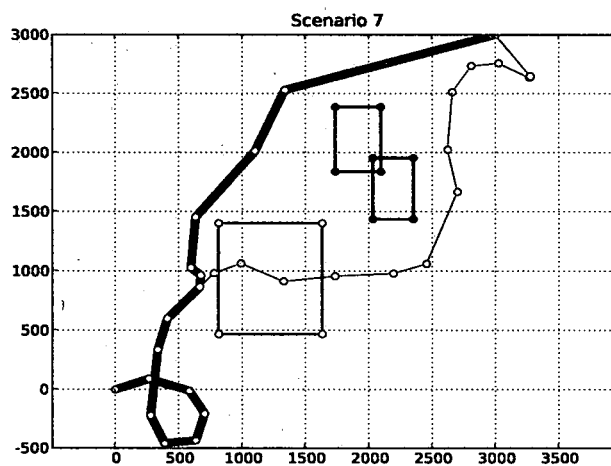


Figure B.14: Reactive Scenario 7 Path - 2

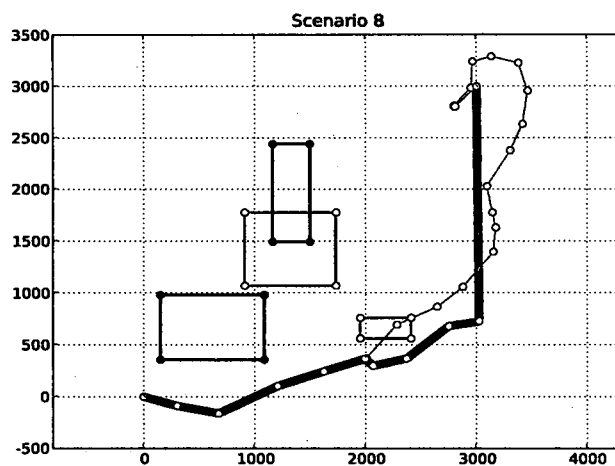


Figure B.15: Reactive Scenario 8 Path - 1

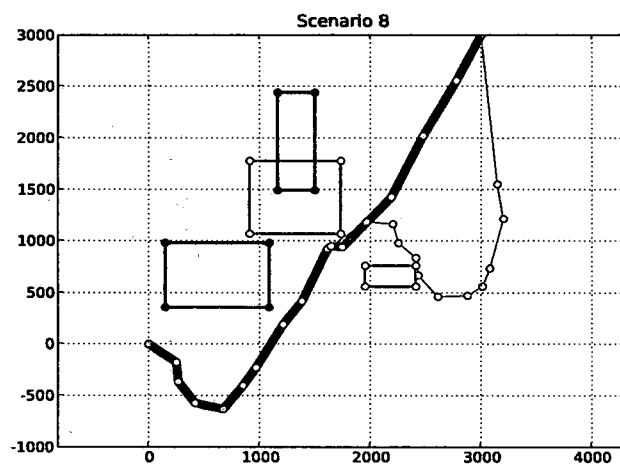


Figure B.16: Reactive Scenario 8 Path - 2

APPENDIX C

Code Listings

```
from __future__ import division

import psyco
psyco.full()

import PathFunctions as pf
from PathRepresentation import Individual
from FitnessEvaluation import PathFitness
from MutationOperators import PathMutation
from SelectionOperators import PathSelection
from CrossoverOperators import PathCrossover
from ObstacleTesting import BoundingBox
from PlanningScenario import PlanningScenario, EvolutionaryParameters
from PathPlots import PathPlots
from UtilityFunctions import Vector, Polygon
from ReactiveTest6 import removePathSegments

import numpy
import math

def removePathSegments(path):
    removedSegments = []
    ctr = 0
    for i in xrange(0, len(path.deltaTimes)):
        if path.deltaTimes[i] == 0.0:
            ctr += 1
            removedSegments.append(i)
    while ctr > 0:
        for i in xrange(0, len(path.deltaTimes)):
            if path.deltaTimes[i] == 0.0:
                ctr = ctr - 1
                del path.deltaTimes[i]
                del path.headings[i]
                del path.speeds[i]
                break
    for j in xrange(0, len(path.intersections)):
        for k in xrange(0, len(removedSegments)):
            if path.intersections[j] > removedSegments[k]:
                path.intersections[j] = path.intersections[j] - 1
```

```

class EvolutionaryProcess():
    def __init__(self, scenario, evparams, showOutput=True):
        self.scenario = scenario
        self.evparams = evparams
        self.showOutput = showOutput

    def evolve(self):
        pathList_xy = []; fullPathList = []; fitness = {}; numgen = 200

        # initialize path population

        pathList = pf.initializePaths(self.evparams.numIndividuals,
                                      self.evparams.numSegments,
                                      self.scenario.initPos,
                                      self.scenario.goalPos)

        if showOutput:
            pp = PathPlots()
            pp.numgen = numgen; pp.plot_gen = 0; pp.paths = pathList;
            pp.obstacles = self.scenario.obstacles; pp.plotPopulation();

        bestfit = []; avgfit = []; allPathLists = {}; bestpaths = {};
        numMutations = 0; numGoodMutations = 0; piaverage = []; pimax = [];

        for gen in range(0,numgen):
            mutatedPathList = []
            if gen == 0:
                ps = 0.2
            for i in range(0,len(pathList)):
                mutatedPathList.append(self.evparams.mutate(pathList[i],
                                                            gen, numgen, ps))

            fullPathList = pathList + mutatedPathList
            for path in fullPathList:
                if path.intersections:
                    self.evparams.directedMutation(path)
            for path in fullPathList:
                path = pf.constrainPath(path)

            for i in range(0,len(fullPathList)):
                fitness[i] = self.evparams.fitEval(fullPathList[i])
            #if gen % 20 == 0:
                #numMutations = 0; numGoodMutations = 0;
            pilist = []
            for i in range(0,len(fitness) - evparams.numIndividuals):
                numMutations += 1
                f_before = fitness[i]
                f_after = fitness[i + evparams.numIndividuals]
                if f_after > f_before: numGoodMutations += 1
                pi = (f_after - f_before) / f_after
                pilist.append(pi*100.0)
                #if pi > 0.5: numGoodMutations += 1
            ps = numGoodMutations / numMutations

```

```

piaverage.append(numpy.average(pilist))
pimax.append(max(pilist))

bestfit.append(max(fitness.values()[0:evparams.numIndividuals-1]))
avgfit.append(numpy.average(fitness.values()[0:evparams.
    numIndividuals-1]))
pathList = self.evparams.select(fullPathList, fitness, q = 3)
#pathList = map(removePathSegments, pathList)
for path in pathList:
    removePathSegments(path)
    path = pf.constrainPath(path)

intent = 0
fullPathList = []

if gen % 50 == 0 and not gen == 0 or gen == numgen-1:
#if gen < 20 and gen != 0 or gen == numgen-1:
    allPathLists[gen] = pathList
    lastfit = []
    for i in range(0, len(pathList)):
        lastfit.append((i, self.evparams.fitEval(pathList[i])))
        fitlist = [x[1] for x in lastfit]
        bestfitlast = max(fitlist)
        for i in range(0, len(lastfit)):
            if lastfit[i][1] == bestfitlast:
                bestind = lastfit[i][0]
        bestpaths[gen] = pathList[bestind]
        self.currentBestPath = pathList[bestind]

if showOutput and gen % 1 == 0: print gen

if showOutput:
    # generate population plots
    for i in allPathLists.iterkeys():
        pp.plot_gen = i; pp.paths = allPathLists[i]; pp.plotPopulation()

    # generate best individual plots
    for i in bestpaths.iterkeys():
        pp.plot_gen = i; pp.plotBestPath(bestpaths[i])

    # generate best and average fitness plot
    pp.plot_gen = gen; pp.bestfitness = bestfit; pp.averagefitness =
        avgfit; pp.plotDebugGraphs()

    # plot average fitness improvement over generations
    generations = range(0, numgen)
    import pylab as p
    f1 = p.figure(); p.plot(generations, piaverage, 'bo-');
    p.grid(True); p.title('Average Fitness Improvement vs. Generations')

    f2 = p.figure(); p.plot(generations, pimax, 'bo-');
    p.grid(True); p.title('Max Fitness Improvement vs. Generations');
    pp.showPlots()

```



```

        #for i in bestpaths.iterkeys():
        #    print bestpaths[i].fitness
        return (bestpaths[numgen-1], bestfit[-1])

if __name__ == "__main__":

    # scenario 1 - basic scenario
    #initPos = (0,0); goalPos = (3000,3000)
    #initialHeading = 90.0; initialSpeed = 10.0;
    #obstaclePoints1 = [Vector(1000,500), Vector(1000,1500),
    #Vector(2000,1500), Vector(2000,500)]
    #obstaclePoints2 = [Vector(0,1500), Vector(0,2000),
    #Vector(1000,2000), Vector(1000,1500)]
    #obstaclePoints3 = [Vector(1500,2500), Vector(1500,3500),
    #Vector(2000,3500), Vector(2000,2500)]
    #obstacles = [Polygon(obstaclePoints1), Polygon(obstaclePoints2),
    #Polygon(obstaclePoints3)]
    #scenario = PlanningScenario(initPos, goalPos, obstacles)

    import cPickle
    scenarioFile = open('C:\\Documents and Settings\\user\\My Documents\\
        MStHesis\\EvPathPlanning\\EvModules\\Data\\ScenarioFiles\\
        randomScenario10.dat','r')
    scenario = cPickle.load(scenarioFile)

    numIndividuals = 10; numSegments = 30; numGenerations = 200;
    pathFit = PathFitness(scenario.obstacles)
    pathMut = PathMutation()
    pathSel = PathSelection(numIndividuals)
    pathCross = PathCrossover()

    evparams = EvolutionaryParameters(numIndividuals, numSegments,
                                        numGenerations,
                                        pathFit.evaluatePathFitness,
                                        pathMut.mutateIndividual,
                                        pathMut.directedMutation,
                                        pathSel.selectPathTournament,
                                        pathCross.crossover)
    #pathSel.selectPathElitist()

    showOutput = True
    evprocess = EvolutionaryProcess(scenario, evparams, showOutput)

    if showOutput:
        evprocess.evolve()

    if not showOutput:
        numRuns = 1000
        results = {}
        for i in range(0, numRuns):
            results[i] = evprocess.evolve()
            print "Run %d Completed..." % i
        results_vals = results.values()

```

```

bestpaths = [el[0] for el in results_vals]
fitnessvals = [el[1] for el in results_vals]

filepath = 'C:\\Documents and Settings\\user\\My Documents\\MSThesis\\
    EvPathPlanning\\EvModules\\Data'
scenarioName = '\\RandomScenario10\\'
scenarioPath = filepath + scenarioName
scenarioFile = scenarioPath + 'bestPaths.dat'
bestPathsFile = open(scenarioFile, 'w')
cPickle.dump(bestpaths, bestPathsFile)

import psyco
psyco.full()

from PathRepresentation import Individual, IndividualWithHeadingFix
from PlanningScenario import PlanningScenario
from SimulationClasses import VehicleState, SimulationParameters
from TurnToStraightLineTest8 import turnToStraightLine
from FootprintCalculation import footprintCalculation
from UtilityFunctions import Vector, Polygon,
normalizeHeadings, minDistanceToObstacle
from PathFunctions import constrainPath
from SimpleObstacleAvoidanceTest import simpleObstacleAvoidance
import math, threading, Queue, time, numpy
from EvModules import PathFunctions, FitnessEvaluation, MutationOperators2,
SelectionOperators, CrossoverOperators
from IntersectionTesting2 import getTurnDirection

def removePathSegments(path):
    original_path = path
    newHeadings = []; newSpeeds = []; newTimes = []; removedSegments = []
    for i in xrange(0, len(original_path.deltaTimes)):
        if not original_path.deltaTimes[i] == 0.0:
            newHeadings.append(original_path.headings[i]);
            newSpeeds.append(original_path.speeds[i]);
            newTimes.append(original_path.deltaTimes[i])
        else:
            removedSegments.append(i)
    intersections = path.intersections
    for j in xrange(0, len(intersections)):
        for k in xrange(0, len(removedSegments)):
            if intersections[j] > removedSegments[k]:
                intersections[j] = intersections[j] - 1
    newPath = Individual(newHeadings, newSpeeds, newTimes,
        original_path.sigma,
        original_path.initPos,
        original_path.goalPos)
    newPath.fitness = original_path.fitness
    newPath.intersections = intersections
    return newPath

class VehicleController(threading.Thread):

```

```

def __init__(self, state, scenario, simParams):
    threading.Thread.__init__(self)
    self.state = state;
    self.scenario = scenario;
    self.simParams = simParams
    self.stateList = []
    self.finalPath = None
    self.maxSensorRange = 250.0
    self.timingData = []
    self.evPathsReceived = []
    self.intrusionPathsReceived = []

def run(self):
    segTime = 0.0; segCounter = 0
    pathFinished = False
    changePath = False
    pathRequest = True
    endPlanning = False
    intrusionState = False
    finalHeadings = [];
    finalSpeeds = [];
    finalTimes = []
    simTime = 0.0
    stateCount = 0
    currentPath = None
    intrusionCounter = 0
    flag = False
    while simTime < self.simParams.stopTime and not flag:
        simTime += self.simParams.timeStep
        stateCount += 1
        try:
            if not pathFinished:
                self.stateList.append(self.state)
                if pathRequest:
                    stateQueue.put(self.state)
        except Queue.Full:
            pass

        try:
            if pathRequest:
                pathRequestQueue.put(pathRequest, block = False)
        except Queue.Full:
            pass

        try:
            if pathRequest:
                currentPath = pathQueue.get()
                if self.state.intrusion:
                    self.intrusionPathsReceived.append(currentPath)
                else:
                    self.evPathsReceived.append(currentPath)
                changePath = True
                pathRequest = False
        except Queue.Empty:

```

```

    pass

# if execution is finished, terminate the planner
try:
    if endPlanning and not flag:
        endQueue.put(endPlanning)
        flag = True
except Queue.Full:
    pass
if pathFinished:
    endPlanning = True

# execution of the current path
#start = time.clock()
if currentPath and not pathFinished:
    if changePath:
        segCounter = 0;
        segTime = 0.0;
        changePath = False
    headings = currentPath.headings;
    speeds = currentPath.speeds;
    deltaTimes = currentPath.deltaTimes
    if segTime > deltaTimes[segCounter]:
        finalHeadings.append(self.state.heading);
        finalSpeeds.append(self.state.speed);
        finalTimes.append(segTime)
        segTime = 0.0; segCounter += 1
        if currentPath.avoidancePath and segCounter == 1:
            pathRequest = True
    else:
        segTime += self.simParams.timeStep
        heading = normalizeHeadings(headings[segCounter],
                                    -180.0, 180.0)

        speed = speeds[segCounter]
        position = (self.state.position[0] +
                    self.state.speed*
                    self.simParams.timeStep*
                    math.sin(math.radians(self.state.heading)),
                    self.state.position[1] +
                    self.state.speed*
                    self.simParams.timeStep*
                    math.cos(math.radians(self.state.heading)))
        (intrusion, turnDirection) = self.testObstacleIntrusion()
        update_rate = round(100.0/(self.simParams.timeStep*10.0))
        if not intrusion:
            intrusionCounter = 0
        else:
            intrusionCounter += 1
        if intrusion and (intrusionCounter % update_rate == 0):
            if not turnDirection == 0:
                pathRequest = True
        self.state = VehicleState(position, heading,
                                   speed, intrusion, turnDirection)
        distToGoal = math.hypot((self.state.position[0] -

```

```

        scenario.goalPos[0]),
        (self.state.position[1] -
         self.scenario.goalPos[1]))

    if distToGoal < 25.0:
        pathFinished = True

    self.finalPath = Individual(finalHeadings,
                                finalSpeeds, finalTimes,
                                0.0, scenario.initPos,
                                scenario.goalPos)

def testObstacleIntrusion(self):
    fpPoints = footprintCalculation(self.state.position, self.state.heading,
                                     self.maxSensorRange)
    fpVectors = []
    intersections = []
    for point in fpPoints:
        fpVectors.append(Vector(point[0], point[1]))
    fpPoly = Polygon(fpVectors)
    for obstacle in self.scenario.obstacles:
        if fpPoly.intersects(obstacle):
            turn_direction = getTurnDirection(self.state, obstacle)
            #return (True, turn_direction)
            intersections.append((obstacle, turn_direction))
    if len(intersections) == 1:
        return (True, intersections[0][1])
    elif len(intersections) > 1:
        distances = []
        for i in xrange(0, len(intersections)):
            if not intersections[i][1] == 0:
                distances.append(minDistanceToObstacle(
                    intersections[i][0], self.state))
        if distances:
            minDist = min(distances)
            for i in xrange(0, len(distances)):
                if distances[i] == minDist:
                    return (True, intersections[i][1])
        else:
            return (True, 0)
    return (False, 0)

class ReactiveEvolutionaryPathGenerator(threading.Thread):
    def __init__(self, state, scenario):
        threading.Thread.__init__(self)
        self.state = state; self.scenario = scenario
        self.evStateList = [];
        self.evPathLists = [];
        self.evBestPaths = [];
        self.numPaths = 10;
        self.numSegments = 20;
        self.timingData = [];
        self.fitnessTimingData = [];
        self.intrusionPaths = [];
        self.evPathsWithoutEnd = [];

```

```

self.pathPopulations = []

fitness = FitnessEvaluation.PathFitness(
    self.scenario.known_obstacles)
mutate = MutationOperators2.ReactivePathMutation()
selection = SelectionOperators.PathSelection(
    self.numPaths)

self.fit_func = fitness.evaluatePathFitness
self.mutation_func = mutate.mutateIndividual
self.directed_mutation_func = mutate.directedMutation
self.selection_func = selection.selectPathElitist

def run(self):
    endPlanning = False
    pathList = PathFunctions.initializePaths(self.numPaths,
                                             self.numSegments,
                                             self.state.position,
                                             self.scenario.goalPos)

    pathList = map(removePathSegments, pathList)
    states = [self.state]*len(pathList)

    genCounter = 0
    fitness = {}; bestfit = []; avgfit = []; bestpaths = {}
    for i in xrange(0,200):
        #print i
        mutatedPathList = map(self.mutation_func, pathList, states)
        fullPathList = pathList + mutatedPathList
        for path in fullPathList:
            if path.intersections:
                path = self.directed_mutation_func(path)
            fullPathList = map(removePathSegments, fullPathList)
            fullPathList = map(constrainPath, fullPathList)
            fitness_values = map(self.fit_func, fullPathList)
            fitness_seq = []
            for i in xrange(0,len(fitness_values)):
                fitness_seq.append((i, fitness_values[i]))
            fitness_dict = dict(fitness_seq)
            bestfit_val = (max(fitness_dict.values())[0:self.numPaths-1])
            bestfit.append(bestfit_val)
            avgfit.append(numpy.average(fitness_dict.values())[0:self.numPaths-1]))
        pathList = self.selection_func(fullPathList, fitness_dict)
    self.pathPopulations.append(pathList)
    currentBestPath = pathList[0]
    for i in xrange(1,len(pathList)):
        if currentBestPath.fitness < pathList[i].fitness:
            currentBestPath = pathList[i]

    heading = currentBestPath.headings[-1]; speed = currentBestPath.speeds
    [-1]
    currentBestPath.xy = currentBestPath.getXYPathNoGoal()
    position = currentBestPath.xy[-1]
    deltaY = self.scenario.goalPos[1] - position[1];

```

```

deltaX = self.scenario.goalPos[0] - position[0]
state = VehicleState(position, heading, speed, False, 0)
endPath = turnToStraightLine(state, self.scenario)
currentBestPath.headings = currentBestPath.headings + endPath.headings
currentBestPath.speeds = currentBestPath.speeds + endPath.speeds
currentBestPath.deltaTimes = currentBestPath.deltaTimes + endPath.
    deltaTimes
currentBestPath = removePathSegments(currentBestPath)
path = currentBestPath

requestCtr = 0
replanEvPaths = False
firstTime = False
pathRequested = False
spawnPoint = None
spawnState = None
gotState = False
while not endPlanning:

    try:
        self.state = stateQueue.get(block = False)
        gotState = True
        self.evStateList.append(self.state)
    except Queue.Empty:
        pass

    try:
        pathRequested = pathRequestQueue.get(block = False)
    except Queue.Empty:
        pass

    try:
        endPlanning = endQueue.get(block = False)
    except Queue.Empty:
        pass

    # evolutionary loop
    if replanEvPaths:
        if firstTime:
            pathList = PathFunctions.initializeReactivePaths(
                self.numPaths,
                3, spawnState.position,
                self.scenario.goalPos,
                spawnState)
            states = [spawnState]*len(pathList)
            mutatedPathList = map(self.mutation.func,
                pathList, states)
            fullPathList = pathList + mutatedPathList
            fitness_values = map(self.fit_func,
                fullPathList)

            fitness_seq = []
            for i in xrange(0, len(fitness_values)):
                fitness_seq.append((i, fitness_values[i]))

```

```

fitness_dict = dict(fitness_seq)
bestfit_val = (max(fitness_dict.values()
                  [0: self.numPaths - 1]))
bestfit.append(bestfit_val)
avgfit.append(numpy.average(fitness_dict.values()
                           [0: self.numPaths - 1]))
pathList = self.selection_func(fullPathList,
                              fitness_dict)

currentBestPath = pathList[0]
for i in xrange(1, len(pathList)):
    if currentBestPath.fitness < pathList[i].fitness:
        currentBestPath = pathList[i]
currentBestPath = removePathSegments(currentBestPath)
currentBestPath = constrainPath(currentBestPath)

heading = currentBestPath.headings[-1];
speed = currentBestPath.speeds[-1]
currentBestPath.xy = currentBestPath.getXYPathNoGoal()
position = currentBestPath.xy[-1]
state = VehicleState(position, heading, speed, False, 0)
endPath = turnToStraightLine(state, self.scenario)
currentBestPath.headings = currentBestPath.headings +
endPath.headings
currentBestPath.speeds = currentBestPath.speeds +
endPath.speeds
currentBestPath.deltaTimes = currentBestPath.deltaTimes +
endPath.deltaTimes
firstTime = False

try:
    if pathRequested and gotState:
        gotState = False
        if self.state.intrusion:
            if not self.state.turn_direction == 0:
                avoidancePath = self.shiftPathHeading()
                self.intrusionPaths.append(avoidancePath)
                path = avoidancePath
                spawnPoint = path.getXYPath()[-2]
                path.avoidancePath = True
                spawnState = VehicleState(spawnPoint,
                                           path.headings[0],
                                           path.speeds[0],
                                           False, 0)

                replanEvPaths = True
                firstTime = True
            if self.state.turn_direction == 0:
                self.evBestPaths.append(currentBestPath)
                self.pathPopulations.append(pathList)
                path = currentBestPath
                path.avoidancePath = False
                replanEvPaths = False
                genCounter = 0

```



```

        else:
            self.evBestPaths.append(currentBestPath)
            self.pathPopulations.append(pathList)
            path = currentBestPath
            path.avoidancePath = False
            replanEvPaths = False
            genCounter = 0
            #print path.headings
            pathQueue.put(path)
            pathRequested = False
    except Queue.Full:
        pass

def shiftPathHeading(self):
    #print 'SHIFTING PATH HEADINGS... Turn Direction: %d' % self.state.
    #turn_direction
    numSegments = 1
    newHeadings = [self.state.heading]
    deltaTimes, speeds = [], []
    minR = self.state.speed**2/(9.81*math.tan(math.radians(45.0)))
    deltaPsi_max = (self.state.speed/minR)*1.0
    deltaPsi_max = normalizeHeadings(math.degrees(deltaPsi_max), -180.0,
    180.0)
    for i in range(0,numSegments):
        deltaTimes.append(10.0)
        speeds.append(10.0)
        if self.state.turn_direction == 1:
            newHeadings.append(newHeadings[0] -
            1.0*deltaPsi_max)
        elif self.state.turn_direction == 2:
            newHeadings.append(newHeadings[0] +
            1.0*deltaPsi_max)
    newHeadings.pop(0)
    path2 = Individual(newHeadings, speeds,
        deltaTimes, 0.0,
        self.state.position,
        self.scenario.goalPos

    return path2

if __name__ == "__main__":
    pathQueue = Queue.Queue(1);
    stateQueue = Queue.Queue(1);
    endQueue = Queue.Queue();
    pathRequestQueue = Queue.Queue(1);

    # basic scenario
    #initialPosition = (0,0); goalPosition = (3000,3000)
    #initialHeading = 90.0; initialSpeed = 10.0;
    #obstaclePoints1 = [Vector(1000,500), Vector(1000,1500),
    #Vector(2000,1500), Vector(2000,500)]
    #obstaclePoints2 = [Vector(0,1500), Vector(0,2000),
    #Vector(1000,2000), Vector(1000,1500)]
    #obstaclePoints3 = [Vector(1500,2500), Vector(1500,3500),

```

```

                                #Vector(2000,3500), Vector(2000,2500)]
#known_obstacles = [Polygon(obstaclePoints2),
                    #Polygon(obstaclePoints3)]
#all_obstacles = [Polygon(obstaclePoints1),
                  #Polygon(obstaclePoints2), Polygon(obstaclePoints3)]

scenario = PlanningScenario(initialPosition, goalPosition, all_obstacles,
                             known_obstacles)
import pickle

initialState = VehicleState(scenario.initPos, initialHeading, initialSpeed,
                             False, 0)

simParameters = SimulationParameters(0.005, 20000.0)

numRuns = 400

start = time.clock()
for runCount in xrange(0,numRuns):
    startrun = time.clock()
    # create threads
    vehicleController = VehicleController(initialState, scenario,
                                           simParameters)
    evPathGenerator = ReactiveEvolutionaryPathGenerator(initialState,
                                                         scenario)

    # start threads
    evPathGenerator.start()
    vehicleController.start()

    #evPathGenerator.setDaemon(True)
    # join threads
    evPathGenerator.join()
    vehicleController.join()

    filepath = 'C:\\Documents and Settings\\user\\My Documents\\MSThesis\\
               EvPathPlanning\\EvModules\\Reactive\\Data\\Reactive6\\
               ConcaveObstacleScenario\\'

    filename = 'BestPaths\\bestPaths' + '_Run_' + str(runCount + 600) + '.
               dat'
    vehicleStatesFileName = 'VehicleStates\\vehicleStates' + '_Run_' + str(
        runCount + 600) + '.dat'
    intrusionPathsFileName = 'IntrusionPaths\\intrusionPaths' + '_Run_' +
        str(runCount + 600) + '.dat'
    pathPopulationsFileName = 'PathPopulations\\pathPopulations' + '_Run_' +
        str(runCount + 600) + '.dat'
    finalPathFileName = 'FinalPaths\\finalPath' + '_Run_' + str(runCount +
        600) + '.dat'

    completeBPFilePath = filepath + filename
    completeVSFilePath = filepath + vehicleStatesFileName
    completeIntrusionFilePath = filepath + intrusionPathsFileName
    pathPopulationsFilePath = filepath + pathPopulationsFileName

```

```

finalPathFilePath = filepath + finalPathFileName

bestPathsFile = open(completeBPFilePath, 'w')
pickle.dump(evPathGenerator.evBestPaths, bestPathsFile)

newStateList = []
ctr = 0
for state in vehicleController.stateList:
    ctr += 1
    if ctr % 10 == 0:
        newStateList.append(state)

vehicleStatesFile = open(completeVSFilePath, 'w')
pickle.dump(newStateList, vehicleStatesFile)

intrusionPathsFile = open(completeIntrusionFilePath, 'w')
pickle.dump(evPathGenerator.intrusionPaths, intrusionPathsFile)

pathPopulationsFile = open(pathPopulationsFilePath, 'w')
pickle.dump(evPathGenerator.pathPopulations, pathPopulationsFile)

finalPathFile = open(finalPathFilePath, 'w')
pickle.dump(vehicleController.finalPath, finalPathFile)
endrun = time.clock() - startrun
print 'Finished run: %d in %g minutes' % (runCount, endrun/60)
runTime = time.clock() - start
print '%d runs took %g minutes' % (numRuns, runTime/60)

class Individual:
    def __init__(self, headings, speeds, deltaTimes, sigma, initPos, goalPos):
        self.headings = headings
        self.speeds = speeds
        self.deltaTimes = deltaTimes
        self.sigma = sigma
        self.initPos = initPos
        self.goalPos = goalPos
        self.intersections = []
        self.fitness = -10*12

    def getXYPath(self):
        import math
        path = [(self.headings[i], self.speeds[i])
                 for i in range(0, len(self.headings))]
        path_xy = []
        path_xy.append(self.initPos)
        for i in range(0, len(path)):
            pos = (path_xy[i][0] + path[i][1]*math.sin(
                (math.pi/180.0)*path[i][0])*self.deltaTimes[i],
                path_xy[i][1] + path[i][1]*math.cos(
                (math.pi/180.0)*path[i][0])*self.deltaTimes[i])
            path_xy.append(pos)
        path_xy.append(self.goalPos)
        return path_xy

```

```

from ObstacleTesting import BoundingBox
import math, numpy

class PathFitness():
    def __init__(self, obstacles):
        self.obstacles = obstacles

    def evaluatePathFitness(self, individual, obstacle_list = []):
        path_xy = individual.getXYPath()
        f1 = self.computeDistanceToGoal(path_xy)
        f2 = self.computeTotalPathLength(path_xy)
        intersections = self.computeObstacleIntersection(path_xy)
        if intersections:
            individual.intersections = intersections
            f3 = 1.0
        else:
            f3 = 0.0
        # static weights
        weights = (10, 15, 10**8)
        f = weights[0]*f1 + weights[1]*f2 +
            weights[2]*f3
        individual.fitness = -1*f
        return -1*f

    def computeDistanceToGoal(self, path_xy):
        p1, p2 = path_xy[-1], path_xy[-2]
        return math.sqrt(math.fabs((p1[0] - p2[0])**2 +
            (p1[1] - p2[1])**2))

    def computeTotalPathLength(self, path_xy):
        pathLength = 0
        for i in range(0, len(path_xy)-1):
            p1, p2 = path_xy[i], path_xy[i+1]
            pathLength += math.sqrt(math.fabs((p1[0] - p2[0])**2 +
                (p1[1] - p2[1])**2))
        return pathLength

    # not used currently
    def computePathAngle(self, individual):
        pathAngle = 0.0
        for i in range(0, len(individual.headings)-1):
            pathAngle += individual.headings[i+1] -
                individual.headings[i]
        return pathAngle

    def computeEnergyUsage(self, individual):
        energy = 0.0
        for i in range(0, len(individual.speeds)):
            energy += individual.speeds[i]**2
        return energy

    def computeObstacleIntersection(self, path_xy):
        delta = 25.0 # used for static planner
        #delta = 176.7 # for max sensor range of 250.0m

```

```

poly_list = []
intersections = []
for i in range(0, len(path_xy)-1):
    p1, p2 = path_xy[i], path_xy[i+1]
    if not(p1 == p2):
        br = BoundingBox([p1, p2], delta)
        poly = br.createBoundingBox()
        poly_list.append(poly)
for i in range(0, len(poly_list)):
    for obstacle in self.obstacles:
        intersects = poly_list[i].intersects(obstacle)
        if intersects:
            intersections.append(i)
return intersections

from __future__ import division
import random, math
import UtilityFunctions

class PathMutation():
    def __init__(self):
        pass

    def mutateIndividual(self, individual, gen = 0, numgen = 0, ps = 0):
        from PathRepresentation import Individual
        from PathFunctions import constrainPath

        mu = 0.0;
        individual.sigma = self.findMutationSigmaUsingFeedback(
            individual.sigma,
            ps, gen)

        mutatedInd = Individual([[],[],[]],
                                individual.sigma,
                                individual.initPos,
                                individual.goalPos)
        mutatedInd.speeds = [c + random.gauss(mu,
                                                mutatedInd.sigma)
                             for c in individual.speeds]
        mutatedInd.headings = [c + random.gauss(mu,
                                                  mutatedInd.sigma)
                                for c in individual.headings]
        upperLims = [180.0]*len(mutatedInd.headings);
        lowerLims = [-180.0]*len(mutatedInd.headings)
        mutatedInd.headings = map(UtilityFunctions.normalizeHeadings,
                                   mutatedInd.headings, lowerLims, upperLims)

        mutatedInd.deltaTimes = [c + random.gauss(mu,
                                                    mutatedInd.sigma)
                                  for c in individual.deltaTimes]

        return constrainPath(mutatedInd)

    def findMutationSigma(self, gen, numgen):

```

```

        return 5.0*(1.0 - 0.9*(gen/numgen))

def findMutationSigmaUsingFeedback(self, sigma, ps, gen):
    # 0.817 <= c <= 1.0
    c = 0.83; ps_ideal = 0.2
    if gen % 20 == 0:
        if ps > ps_ideal: return sigma/c
        elif ps < ps_ideal: return sigma*c
    return sigma

def directedMutation(self, path):
    from PathFunctions import constrainPath
    for segment in path.intersections:
        #print 'Segment in Directed Mutation: %d' % segment
        if path.headings:
            path.headings[segment-1] = path.headings[segment-1] - 50.0
    return constrainPath(path)

def directedMutationWithHeadingFix(self, path):
    from PathFunctions import constrainPathWithHeadingFix
    for segment in path.intersections:
        path.headings[segment-1] = path.headings[segment-1] - 50.0
    path.makePathFeasible()
    return constrainPathWithHeadingFix(path)

class PathSelection():
    def __init__(self, numIndividuals):
        self.numIndividuals = numIndividuals

    def selectPathElitist(self, fullPathList, fitness):
        if fullPathList and fitness:
            selectedPaths = []
            fitVals = fitness.values()
            fitVals.sort(reverse=True)
            bestfit = fitVals[0:self.numIndividuals]
            for key in fitness.iterkeys():
                for i in range(0,self.numIndividuals):
                    if fitness[key] == bestfit[i]:
                        selectedPaths.append(fullPathList[key])
            return selectedPaths

    def selectPathTournament(self, fullPathList, fitness, q=3):
        from random import randrange
        from operator import itemgetter
        if fullPathList and fitness:
            selectedPaths = []
            scores = {}
            for path in fullPathList:
                competitors = {}
                for i in range(0,q):
                    idx = randrange(len(fullPathList))
                    competitors[idx] = fullPathList[idx]
                score = 0
                for competitor in competitors.iterkeys():

```

```
        if fitness[fullPathList.index(path)] > fitness[competitor]:  
            score = score + 1  
        scores[fullPathList.index(path)] = score  
s = scores.items()  
s.sort(key = itemgetter(1), reverse=True)  
s = s[0:self.numIndividuals]  
selected_indices = [e[0] for e in s]  
for i in selected_indices:  
    selectedPaths.append(fullPathList[i])  
return selectedPaths
```

BIBLIOGRAPHY

- [1] Mehdi Alighanbari, Yoshiaki Kuwata, and Jonathan How. Coordination and control of multiple uavs with timing constraints and loitering. 2003.
- [2] Erik P. Anderson and Randal W. Beard. Real-time dynamic trajectory smoothing for unmanned air vehicles. *IEEE Transactions on Control Systems Technology*, 13(3), 2005.
- [3] Thomas Bäck, David Fogel, and Zbigniew Michalewicz. *Handbook of Evolutionary Computation*. Oxford University Press, 1997.
- [4] Scott Bortoff. Path planning for unmanned air vehicles. Technical report, Air Force Research Lab, 1999.
- [5] Seth G. Bullock. Co-evolutionary design: Implications for evolutionary robotics. In *Proceedings of the 3rd European Conference On Artificial Life*, 1995.
- [6] Mark Campbell and Jarurat Ousingsawat. On-line estimation and path planning for multiple vehicles in an uncertain environment. In *Proceedings of the AIAA Guidance, Navigation and Control Conference*, 2002.
- [7] Brian Capozzi and Juris Vagners. Evolving (semi)-autonomous vehicles. In *Proceedings of the AIAA Guidance, Navigation and Control Conference*, 2001.
- [8] Brian J. Capozzi. *Evolution-Based Path Planning and Management for Autonomous Vehicles*. PhD thesis, University of Washington.
- [9] Phillip R. Chandler and S. Rasmussen. Uav cooperative path planning. In *Proceedings of the AIAA Guidance, Navigation and Control Conference*, 2000.
- [10] Atif Chaudhry, Kathy Misovec, and Raffaello D'Andrea. Low observability path planning for an unmanned air vehicle using mixed integer linear programming. 2004.
- [11] Jen-Hui Chuang and Narendra Ahuja. Path planning using the newtonian potential. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1991.
- [12] Khac Duc Do and Zhong-Ping Jiang. A global output-feedback controller for simultaneous tracking and stabilization of unicycle-type mobile robots. In *IEEE Transactions on Robotics and Automation*, 2004.

- [13] Atilla Dogan. Probabilistic path planning for uavs. In *Proceedings of the 2nd AIAA Unmanned Unlimited Systems, Technologies and Operations Conference*, 2003.
- [14] Yeonju Eun and Hyochoong Bang. Cooperative control of multiple ucavs for suppression of enemy air defense. In *Proceedings of the 3rd AIAA Unmanned Unlimited Technical Conference*, 2004.
- [15] Mathew Flint, Marios Polycarpou, and Emmanuel Fernandez-Gaucherand. Cooperative path-planning for autonomous vehicles using dynamic programming. In *Proceedings of the 15th Triennial World Congress of the International Federation of Automatic Control*, 2002.
- [16] Emilio Frazzoli, Munther A. Dahleh, and Eric Feron. Real-time motion planning for agile autonomous vehicles. *AIAA Journal of Guidance, Control and Dynamics*, 25(1):116–129, 2002.
- [17] Paolo Gaudiano, Benjamin Shargel, Eric Bonabeau, and Bruce T. Clough. Control of uav swarms: What the bugs can teach us. In *Proceedings of the 2nd AIAA Unmanned Unlimited Systems, Technologies and Operations Conference*, 2003.
- [18] Veysel Gazi and Kevin Passino. Stability analysis of swarms in an environment with an attractant/repellent profile. In *Proceedings of the American Control Conference*, 2002.
- [19] J.D. Han and Mark Campbell. Artificial potential guided evolutionary path plan for target pursuit and obstacle avoidance. In *Proceedings of the AIAA Guidance, Navigation and Control Conference*, 2003.
- [20] Tamer Inanc, Kathy Misovec, and Richard Murray. Nonlinear trajectory generation for unmanned air vehicles with multiple radars. 2004.
- [21] Dong Jia and Juris Vagners. Parallel evolutionary algorithms for uav path planning. In *Proceedings of AIAA's 1st Intelligent Systems Technical Conference*, 2004.
- [22] Kevin B. Judd and Timothy W. McLain. Spline based path planning for unmanned air vehicles. In *Proceedings of the AIAA Guidance, Navigation and Control Conference*, 2001.
- [23] Myungsoo Jun and Raffaello D'Andrea. *Cooperative Control: Models, Applications and Algorithms*, chapter 6, pages 95–111. Springer, 2003.
- [24] Shreecharan Kanchanavally, Raúl Ordóñez, and Jeff Layne. Mobile target tracking by networked uninhabited autonomous vehicles via hospitability maps. In *Proceedings of the 2004 American Control Conference*, 2004.
- [25] Shreecharan Kanchanavally, Chunlei Zhang, and Raúl Ordóñez. Mobile target tracking with communication delays. In *Proceedings of the 2004 IEEE Conference on Decision and Control*, 2004.

- [26] Jusuk Lee, Rosemary Huang, and Andrew Vaughn. Strategies of path planning for a uav to track a ground vehicle. In *Proceedings of the Second Annual Symposium on Autonomous Intelligent Networks and Systems*, 2003.
- [27] Ti-Chung Lee and Kai-Tai Song. Tracking control of unicycle-modeled mobile robots using a saturation feedback controller. *IEEE Transactions on Control Systems Technology*, 9(2), 2001.
- [28] Theju Maddula, Ali Minai, and Marios Polycarpou. Multi-target assignment and path planning for groups of uavs. In *Proceedings of the Conference on Cooperative Control and Optimization*, 2002.
- [29] Zbigniew Michalewicz, Jing Xiao, and Krzysztof Trojanowski. Evolutionary computation: One project, many directions. In *Proceedings of the 9th International Symposium, ISMIS '96*, 1996.
- [30] Reza Olfati-Saber and Richard Murray. Distributed cooperative control of multiple vehicle formations using structural potential functions. In *Proceedings of the 15th Triennial World Congress of the International Federation of Automatic Control*, 2002.
- [31] H. Van Dyke Parunak. Go to the ant: Engineering principles from natural multi-agent systems. *Annals of Operations Research*, (75):69–101, 1997.
- [32] H. Van Dyke Parunak, Sven Brueckner, and James Odell. Swarming coordination of multiple uav's for collaborative sensing. 2003.
- [33] H. Van Dyke Parunak, Michael Purcell, and Robert O'Connell. Digital pheromones for autonomous coordination of swarming uavs. In *Proceedings of AIAA's 1st Technical Conference on Unmanned Aerospace Vehicles*, 2002.
- [34] Olaf Pettersson and Patrick Doherty. Probabilistic roadmap based path planning for an autonomous unmanned aerial vehicle. *American Association for Artificial Intelligence*, 2004.
- [35] Anawat Pongpunwattana and Rolf Rysdyk. Real-time planning for multiple autonomous vehicles in dynamic uncertain environments. *AIAA Journal of Aerospace Computing, Information and Communication*, 1:580–604, 2004.
- [36] Anawat Pongpunwattana, Rolf Rysdyk, and Juris Vagners. Market-based co-evolution planning for multiple autonomous vehicles. In *Proceedings of the 2nd AIAA Unmanned Unlimited Systems, Technologies and Operations Conference*, 2003.
- [37] Krishnakumar Ramamoorthy, John L. Crassidis, and Tarunraj Singh. Potential functions for en-route air traffic management and flight planning. In *Proceedings of the AIAA Guidance, Navigation and Control Conference*, 2004.

- [38] David Rathbun and Brian Capozzi. Evolutionary approaches to path planning through uncertain environments. In *Proceedings of the 1st AIAA Technical Conference and Workshop on Unmanned Aerospace Vehicles*, 2002.
- [39] David Rathbun, Sean Kragelund, and Anawat Pongpunwattana. An evolution based path planning algorithm for autonomous motion of a uav through uncertain environments. In *Proceedings of the 21st Digital Avionics Systems Conference*, 2002.
- [40] Arthur Richards and John Bellingham. Coordination and control of multiple uavs. In *Proceedings of the AIAA Guidance, Navigation and Control Conference*, 2002.
- [41] Elon Rimon and Daniel Koditschek. Exact robot navigation using artificial potential functions. *IEEE Transactions on Robotics and Automation*, 8(5), 1992.
- [42] Juan Carlos Rubio, Juris Vagners, and Rolf Rysdyk. Adaptive path planning for autonomous uav oceanic search missions. In *Proceedings of AIAA's 1st Intelligent Systems Technical Conference*, 2004.
- [43] Jerzy Sasiadek and Ignacy Duleba. 3d local trajectory planner for uav. *Journal of Intelligent and Robotic Systems*, 29(2), 2000.
- [44] Takanori Shibata and Toshio Fukuda. Intelligent motion planning by genetic algorithm with fuzzy critic. In *Proceedings of the 1993 International Symposium on Intelligent Control*, 1993.
- [45] Takanori Shibata and Toshio Fukuda. Intelligent motion planning by genetic algorithm with fuzzy critic. In *Proceedings of the 1993 International Symposium on Intelligent Control*, 2004.
- [46] Krzysztof Trojanowski and Zbigniew Michalewicz. Evolutionary algorithms and the problem-specific knowledge, 1997.
- [47] Jing Xaio, Zbigniew Michalewicz, and Krzysztof Trojanowski. Adaptive evolutionary planner / navigator for mobile robots. *IEEE Transactions on Evolutionary Computation*, 1997.
- [48] Jing Xaio, Zbigniew Michalewicz, and Krzysztof Trojanowski. Adding memory to the evolutionary planner / navigator. In *Proceedings of the IEEE International Conference on Evolutionary Computation*, 1997.
- [49] Hong Iris Yang and Yiyuan J. Zhao. Trajectory planning for autonomous aerospace vehicles amid known obstacles and conflicts. *AIAA Journal of Guidance, Control and Dynamics*, 27(6):997–1008, 2004.
- [50] Chunlei Zhang and Raúl Ordóñez. Decentralized adaptive coordination and control of uninhabited autonomous vehicles via surrogate optimization. In *Proceedings of the American Control Conference*, 2003.

- [51] Chunlei Zhang and Raúl Ordóñez. Decentralized adaptive coordination and control of uninhabited autonomous vehicles via surrogate optimization. In *Proceedings of the 2003 American Control Conference*, 2003.
- [52] Chunlei Zhang and Raúl Ordóñez. Multi-vehicle cooperative search with uncertain prior information. In *Proceedings of the 2004 AIAA Conference on Guidance, Navigation and Control*, 2004.
- [53] Min Zhao, Nirwan Ansari, and Edwin S. H. Hou. Mobile manipulator path planning by a genetic algorithm. In *Proceedings of the 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1992.

R002594547