University of Nevada, Reno

Dynamic Path Planning and Replanning for Mobile Robot Team Using RRT*

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

by

Devin M Connell

Dr. Hung M. La - Thesis Advisor
May 2017

Abstract

It is necessary for a mobile robot to be able to efficiently plan a path from its starting or current location to a desired goal location. This is a trivial task when the environment is static. However, the operational environment of the robot is rarely static, and it often has many moving obstacles. The robot may encounter one, or many, of these unknown and unpredictable moving obstacles. The robot will need to decide how to proceed when one of these obstacles is obstructing it's path. A method of dynamic replanning using RRT* is presented. The robot will modify it's current plan when an unknown random moving obstacle obstructs the path. In multi-robot scenarios it is important to efficiently develop path planning solutions. A method of node sharing is presented to quickly develop path plans for a multi-robot team. Various experimental results show the effectiveness of the proposed methods.

Acknowledgments

I would like to thank my advisor, Dr. Hung La, who provided wisdom, inspiration and most of all patience to keep me focused.

To my committee members, Dr. Monica Nicolescu and Dr. Pradeep Menezes, thank you for taking time away from your busy schedules to review this thesis and provide your input.

To my employer, Sierra Nevada Corporation, thank you for being flexible and supportive of my goals.

Finally, I would like to thank my girlfriend, Danella, who provided encouragement when I was struggling, and got me back on track when I was distracted.

Table of Contents

List of Tables

List of Figures

Chapter 1

Thesis Introduction, Contribution, and Organization

1.1 Introduction

Path planning has been one of the most researched problems in the area of robotics. The primary goal of any path planning algorithm is to provide a collision free path from a start state to an end state within the configuration space of the robot. Probabilistic planning algorithms, such as the Probabilistic Roadmap Method (PRM) [1] and the Rapidly-exploring Random Tree (RRT) [2], provide a quick solution at the expense of optimality. Since its introduction the RRT algorithm has been one of the most popular probabilistic planning algorithms. The RRT is a fast, simple algorithm that incrementally generates a tree in the configuration space until the goal is found.

The RRT has a significant limitation in finding an asymptotically optimal path, and has been shown to never converge to an asymptotically optimal solution [3] [4]. There is extensive research on the subject of improving the performance of the

RRT. Simple improvements such as the Bi-Directional RRT and the Rapidly-exploring Random Forest (RRF) improve the search coverage and speed at which a single-query solution is found. The Anytime RRT [5] provides a significant improvement in cost-based planning. The RRT* algorithm provides a significant improvement in the optimality of the RRT and has been shown to provide an asymptotically sub-optimal solution [3].

Since the introduction of the RRT* algorithm, research has expanded to discover new ways to improve upon the algorithm. Research includes adding heuristics [6] [7] or bounds [8] to the algorithm in order to maintain the convergence of the algorithm but reduce the execution time. Additional research attempts to guide the algorithm through intelligent sampling [9], or guided sampling through an artificial potential field [10].

In many scenarios the operational environment is rarely static. The path from a single query plan will often be obstructed during execution. For that reason the topic of replanning is very important to robotic path planning. It is not feasible to discard an entire search tree and start over. One method is to store waypoints and regrow trees called the ERRT [11]. Another method (DRRT) is to place the root of the tree at the goal location, so that only a small number of branches may be lost or invalidated when replanning [12]. The Multipartite RRT maintains a set of subtrees that may be pruned and reconnected, along with previous states to guide regrowth. It is essentially a combination of DRRT and ERRT [13]. More recently the RRT* algorithm has been incorporated into replanning. RRT$^{\mathrm{X}}$ is an algorithm that uses RRT* to continuously update the path during execution [14]. The RRT$^{\mathrm{X}}$ is able to compensate for instantaneous changes in the static environment which is outside the scope of this work.

Research on multi-robot teams has increased in recent years. These multi-robot teams are accomplishing tasks from cooperative sensing [15, 16], formation control [17–21] and target tracking and observation [22]. In a multi-robot scenario it may be important for each robot on the team to quickly develop a path plan. If path plans are found quickly then more time may be spent executing tasks, rather than planning to execute tasks.

1.2 Motivation

The motivation of this thesis is to study robotic path planning in a complex 2-Dimensional (2-D) environment with unknown random obstacles. This environment is intended to be similar to real world scenarios. A static or predictable environment is not representative to the real world. Often, the real world is unpredictable and full of randomly moving obstacles. A robot moving within the real world will need to be able to quickly develop a plan and modify that plan to avoid colliding with the obstacles. Ideally the plan to avoid obstacles would maintain the optimality of a single query path plan. Often more than one robot will be moving throughout the environment. This work also studies a scenario where multiple robots share path planning information. Though many algorithms exists; these scenarios are studied using two popular robot path planning algorithms, the RRT and RRT*.

1.3 Contribution

The contribution of this work contains three parts. The first part is the comparison of the RRT versus RRT* algorithms in a complex static 2D environment. The second part is the method using the RRT* algorithm for replanning in a dynamic environment with random, unpredictable moving obstacles. Finally, the third part is the use of

RRT* for multiple robots to share information about the environment to efficiently develop a path plan for each robot.

1.4 Organization

The organization of this thesis is as follows: Chapter 2 will provide background information on path planning and previous work on the RRT and RRT* algorithms. Chapter 3 will present the environment setup, analysis and comparison results between the RRT and RRT* algorithms in the complex static environment. Chapter 4 will present the analysis and results of replanning using the RRT* algorithm. Chapter 5 presents the analysis and results of using RRT* with multiple robots. Finally Chapter 6 will present the conclusions and future work.

Chapter 2

Background

2.1 Path Planning and Configuration Spaces

The objective of robotic path, or motion, planning is to find a collision free path from a starting configuration to an ending configuration. Path planning algorithms operate on the configuration space of the robot. The configuration space Q is the set of all possible configurations of the robot. A configuration defines the state of the robot. For a 2D planar robot that can rotote a configuration would be the robot's position and orientation. The configuration space will include both the obstacle space and the free space. The obstacle space is the set of configurations of the robot where a collision would occur in the real world. The free space is all configurations that are not in the obstacle space. When defining the configuration space for the robot the simplest method is to derive a transformation such that the robot's configuration is a single point within the space. This transform is also applied to the obstacles in the space, the obstacles become regions in the configuration space based on the transformation applied to the robot.

When the robot moves from one configuration to another there is a cost associated with that movement. In a simple 2-Dimensional configuration space for a planar robot, the cost to go from one location to another location can be measured in Euclidean distance. The cost becomes more complex when the robot has more degrees of freedom. For a 3-D rigid body that can rotate, the cost cannot be the Euclidean distance. The cost to go must also account for the swept volume the robot occupies when rotating about an axis. Cost is what determines the quality of a path in the configuration space. If a path has very high cost, then the robot may be making too many unnecessary movements in order to reach the goal location.

2.2 Decomposition and Roadmap Methods

There are many methods for path planning in simple configuration spaces. These simple methods are built upon manipulation and simplification of the configuration space. Once the configuration space has been simplified a robot may easily plan from the start configuration to the goal configuration. One of the simplest ways to represent the configuration space is an Occupancy Grid. The configuration space is separated into cells, if a cell has an obstacle within it then the cell is considered part of the obstacle space, otherwise it is part of the free space. The robot will move from a start location to a goal location by traversing through the free cells as shown in Fig. 2.1

Another decomposition method is to use a quad-tree to subdivide the free space. If an object is within a grid, but does not cover the entire grid square, then the grid is divided into four smaller grids until the object fills a subdivided grid square. A sample quad-tree decomposition is shown below in Fig. 2.2. This method allows the configuration space to be divided exactly on the borders of the obstacles. Grid

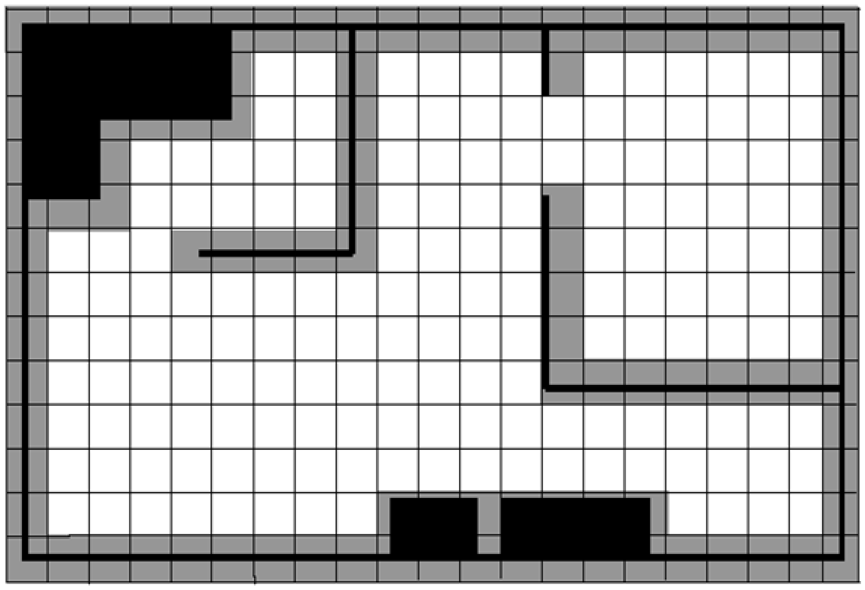


Figure 2.1: The Cell Decomposition Method.

squares marked as obstacles in the quad-tree decomposition will fill an entire grid square, unlike the Occupancy grid where even a small corner of an obstacle within a grid square will cause the entire square to be marked as an obstacle.

The Trapezoidal Decomposition, shown in Fig. 2.3, is another method of subdividing the free space of the configuration space. The decomposition involves sweeping a line through the configuration space. When the line encounters a vertex, whether an obstacle vertex or configuration space boundary vertex, a line is drawn from that vertex to any boundary, obstacle or spatial, in both directions. This decomposition is exact and creates regions whose edges coincide with the obstacle, and spatial boundary, edges and vertices.

Another style of path planning are the roadmap methods. These methods use points in the configuration space to form a graph, or roadmap, for the robot to follow. There are three important properties of roadmaps that must be followed: Accessible, Connectable and Departable. First, the roadmap must be accessible. For any starting location within the configuration space there must be a collision free path

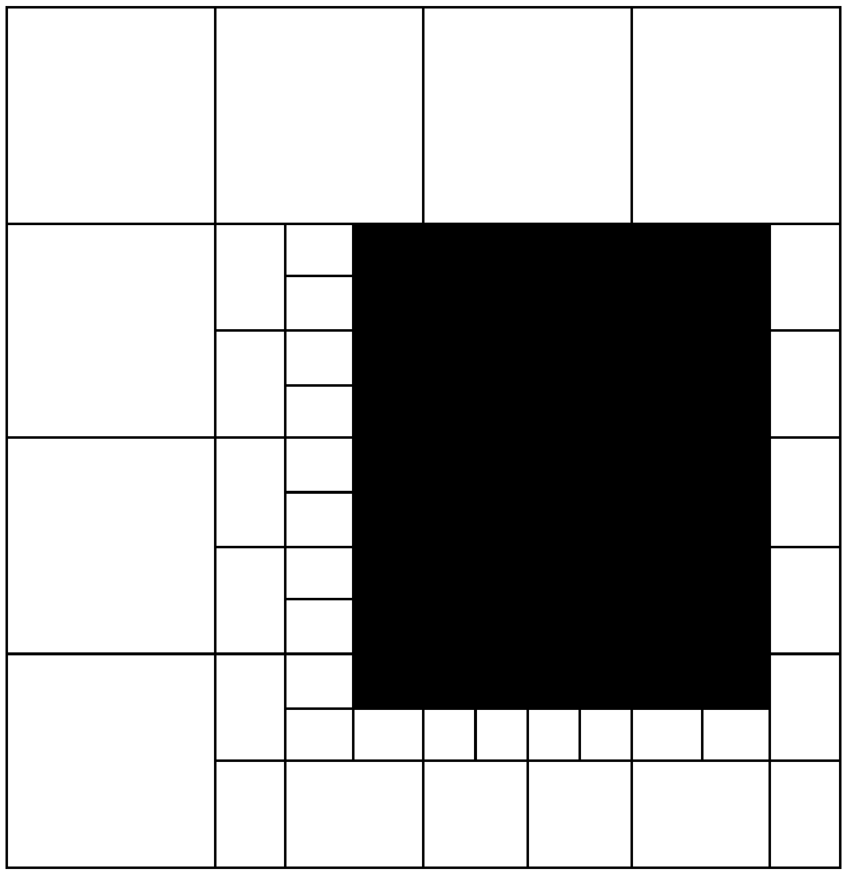


Figure 2.2: A Quadtree Decomposition method.

to the nearest vertex on the roadmap. Second, the roadmap must be connectable. For any vertex on the roadmap there must be a collision free path to any other vertex on the roadmap. Finally, the roadmap must be departable. For any goal location there must be a collision free path from the nearest vertex on the roadmap to the goal location. The visibility graph is a simple roadmap method where the vertices of the obstacles in the configuration space become vertices in the roadmap. The edges are formed by straight lines from each vertex to every other visible vertex, shown in Fig. 2.4.

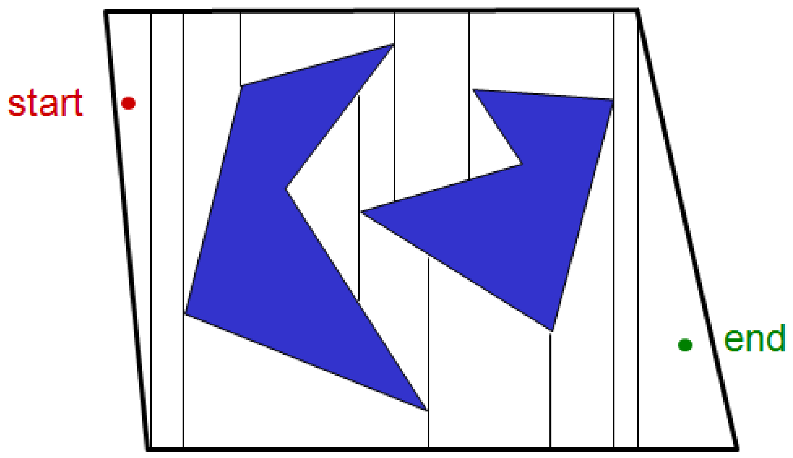


Figure 2.3: The Trapezoidal Decomposition of a simple configuration space.

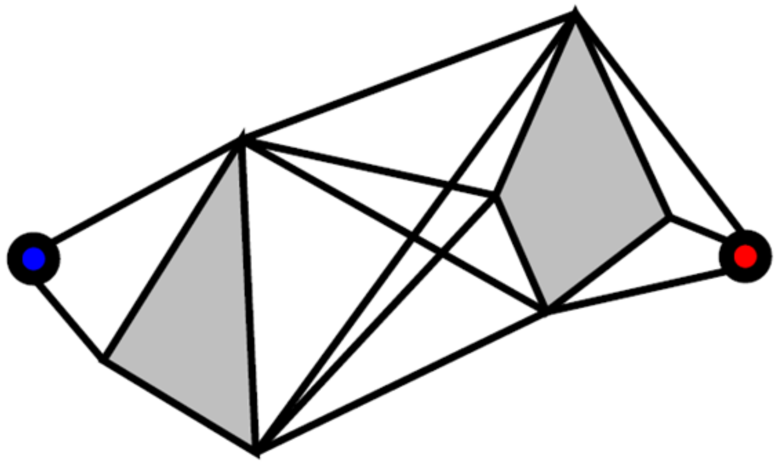


Figure 2.4: A roadmap created using the Visibility Graph.

The start and goal locations are then added to the roadmap and edges are added by straight lines to these locations from each visible vertex. One of the drawbacks with this method is the inferior edges that are created. Edges will be created that intersect an interior obstacle vertex rather than extend to the exterior obstacle vertices. These edges can be removed to create an optimal visibility graph. Fig. 2.5 shows how these interior edges may be removed.

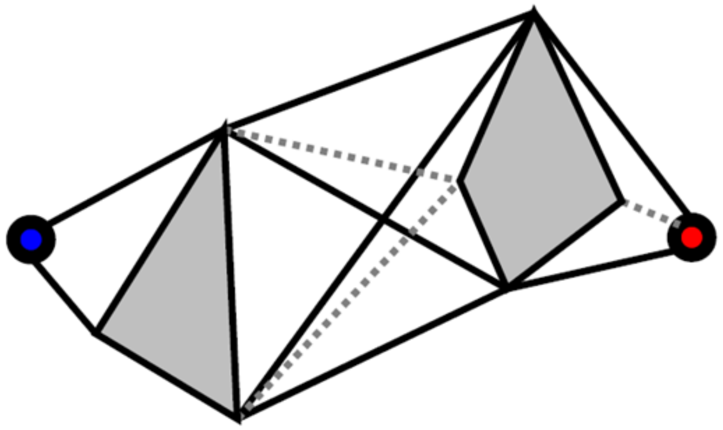


Figure 2.5: A Visibility Graph where the dashed lines represent redundant interior edges to be removed.

A Voronoi Diagram is another effective way to create a roadmap in the configuration space. The Voronoi Diagram creates edges that are equidistant to a point, or many points, in a region. Within the configuration space the diagram is created around the obstacle vertices and edges. The vertices of the roadmap are the intersection points of the diagram and the edges of the roadmap follow the Voronoi diagram edges between the obstacles. Fig. 2.6 is an example of the Generalized Voronoi Diagram.

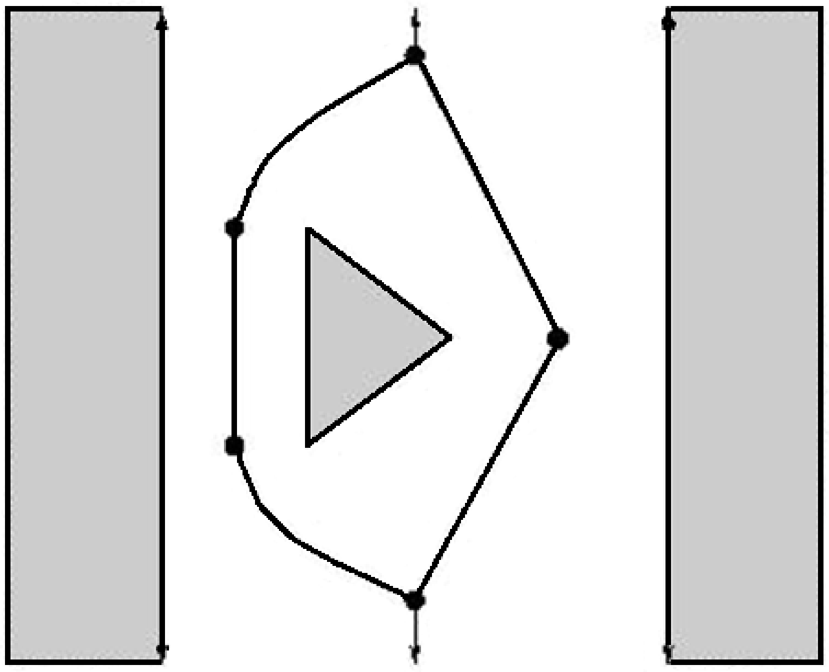


Figure 2.6: A Generalized Voronoi Diagram.

Using the Trapezoidal decomposition method a simple roadmap can be created. The centroid of each region will be a vertex in the roadmap. This method is shown in Fig. 2.7. Additional vertices are added at the midpoints of each edge of the region. The edges of the roadmap are the lines that connect each centroid to the edge points of the region.

These methods are easy to implement in simple, 2-Dimensional, configuration

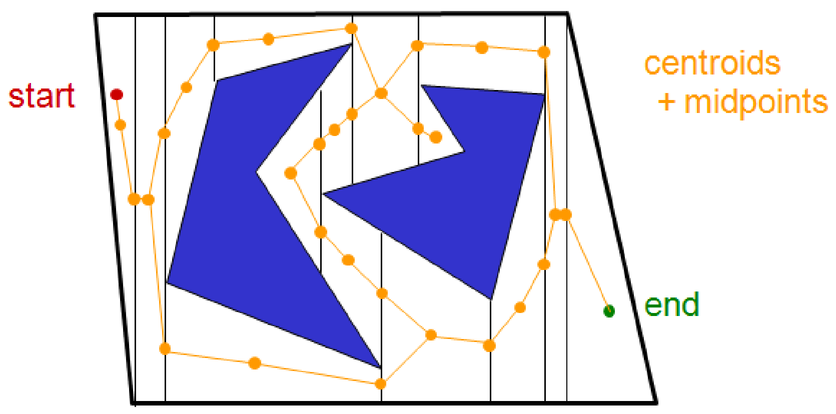


Figure 2.7: Roadmap construction using the Trapezoidal Decomposition of the configuration space.

spaces. Each of these methods becomes prohibitive in more complex, higher dimension, configuration spaces. For example, in a simple 3D configuration space containing a single cube obstacle with the start and goal locations on opposite sides of the cube, the visibility graph would produce edges and vertices on the corners and edges of the obstacle. However, that method will not produce the optimal path, except in rare circumstances. In order to produce the optimal path the visibility graph would need to create a graph edge from the start point to the edge of the obstacle itself. There are an infinite number of possibilities regarding where to select the vertex along the edge of the obstacle. It is impossible to generate a visibility graph to determine the optimal path around the cube.

2.3 Probabilistic Planning

Early path planning algorithms such as Djikstra's algorithm and A* use exhaustive or heuristic based search methods to find the optimal path in the configuration space of the robot. These are complete algorithms that are guaranteed to find an optimal path from a start configuration to the goal configuration, if a path exists. These methods are prohibitive in higher dimension configuration spaces. A complete algorithm for finding the optimal path from a start configuration to a goal configuration in three dimensions is NP-Hard. To overcome this difficulty certain concessions had to be made on the algorithm. The first concession is the implicit definition of the configuration space, instead of a complete representation. Under this definition the only information provided about a specific configuration of the robot is whether it is in the free space or in the obstacle space. The second concession is the completeness of the algorithm. A complete algorithm, such as A* and Djikstra's algorithm, guarantees to return the optimal path or no path if none exists. This concession led to probabilistic algorithms. An algorithm is considered probabilistically complete if given enough time the algorithm will eventually find the solution. If no solution exists, the algorithm will not return and provide no information.

2.4 The RRT

The Rapidly-exploring Random Tree (RRT) is a probabilistic path planning algorithm that is designed to return a solution very fast. This algorithm builds a tree (T = (E, V)) in the configuration space where the vertices, V, are randomly selected collision free configurations within the configuration space of the robot. Each new vertex creates only one edge, E, from itself to the nearest node, or parent node, within the tree. Fig. 2.8 shows the RRT operate in a small simple 2-D configuration space. The starting location is in the lower left corner and the goal location is the upper right corner. Each red line in the figure represents an edge in the tree, and each asterisk symbol is a vertex within the tree.

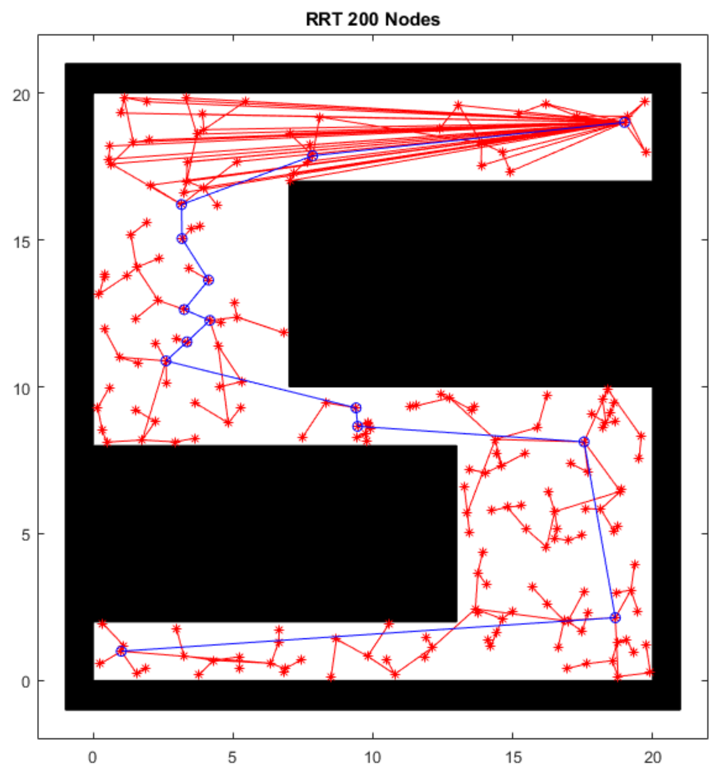


Figure 2.8: A sample RRT showing the structure of the tree. The red lines represent edges in the tree. Each node is marked by an asterisk. The best path found is marked by the blue line.

The tree is initialized with one vertex at the starting location of the robot. Lines 1-2 of Algorithm 1 initialize the tree and insert the starting location into the tree. The tree is formed by selecting random, collision free, samples in the configuration space then attempting to grow the tree toward that location. The algorithm continues to sample and add new vertices and edges to the tree until the maximum allowed number of vertices is, or the goal is found depending on termination conditions of the implementation. The blue line in Fig. 2.8 shows the best path found by the algorithm when the maximum number of nodes was reached. Line 4 in Algorithm 1 selects the random, collision free sample in the configuration space, q_{rand}. Line 5 finds the nearest neighbor already in the tree from which to grow. Line 6 is the $Extend()$ function, described in Algorithm 2, which determines if a new edge (branch) can be added to the tree from the nearest node, q_{near}, to the new random node, q_{rand}. The extension length Δq is a pre-determined value called the growth factor. The growth factor can have a significant effect on the expansion of the tree. If the growth factor is small then the tree will have many short branches. If the growth factor is large, or non-existant, the tree can have long branches. There is a trade off in specifying the growth factor. If the growth factor is too small then it will take many more nodes to explore the configuration space and find the solution. If the growth factor is too large, or non-existant, extending the tree will fail more often, causing the algorithm to resample a new location. When attempting to grow the tree the new branch is checked for collisions using the local planner. Line 1 of Algorithm 2 is a simplification of the local planner. The local planner determines if the edge connecting q_{near} to q_{rand} is collision free. If this extension is collision free then a new vertex is created at the end of the extension, q_{new}, and is added to the set of vertices, Line 3 in Algorithm 2. The parent of this new vertex will be the nearest node in the tree. The edge connecting q_{near} to q_{rand}, is added to the set of edges on Line 4 of Algorithm 2.

Algorithm 1: $T = (V, E) \leftarrow$RRT(q_{init})

```
1 $T \leftarrow$InitializeTree()
2 $T \leftarrow$InsertNode($\emptyset$, $q_{init}$, $T$)
3 for $k \leftarrow 1$ to $N$ do
4     $q_{rand} \leftarrow$RandomSample($k$)
5     $q_{near} \leftarrow$NearestNeighbor($q_{rand}$,T)
6     $q_{new} \leftarrow$Extend($q_{rand}$, $q_{near}$, $\Delta$q)
7 end
```

Algorithm 2: $q_{new} \leftarrow$Extend(q_{rand}, q_{near}, Δq)

```
1 $q_{new} = ||q_{near} - q_{rand}|| + \Delta q$
2 if ObstacleFree($q_{new}$) then
3     $V = V \cup \{q_{new}\}$
4     $E = E \cup \{q_{near}, q_{new}\}$
5 end
6 return $q_{new}$
```

When the RRT is selecting a random sample occasionally the goal location is selected as the sample. This will force the tree to try to connect to the goal from the node closest to the goal. Selecting the goal node as the new sample is based on a pre-determined probability. It is possible for the tree to have found a node very close to the goal but not connect to the goal due to this probability. Another option is to attempt to connect to the goal after each successfully inserted node. Using this method the algorithm tries to complete the path as quickly as possible. This method is used in the algorithm shown in Fig. 2.8, and throughout all experiments. This will also increase calls to the local planner to attempt to add a branch to the tree.

Additional calls to the local planner will increase execution time.

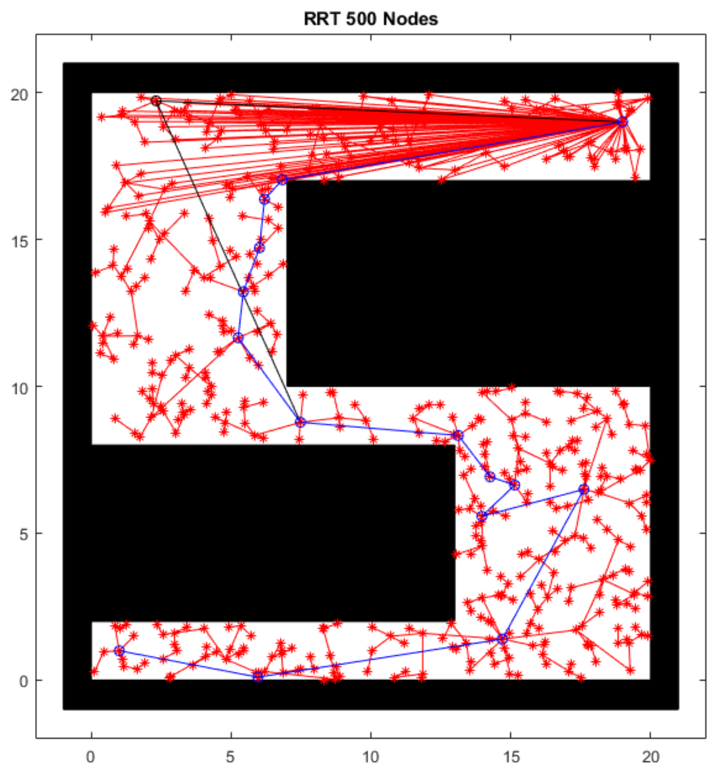


Figure 2.9: A Sample RRT. The blue line represents the best path found and the black line represents the first path found by the RRT.

The RRT explores the configuration space and finds a solution very fast. One of the drawbacks to the RRT is that the algorithm will never converge to an asymptotically sub-optimal solution. The Probabilistic Roadmap Method (PRM), which was introduced two years prior to the RRT, is an asymptotically sub-optimal algorithm. An algorithm is asymptotically sub-optimal when it improves the solution as the execution of the algorithm progresses toward the true optimal solution. Probabilistic algorithms are considered asymptotically sub-optimal because as the algorithm improves the solution, the probability that the algorithm will continue to improve approaches zero. The RRT can randomly find a better solution, it will never deliberately improve upon a solution it has already found. Fig. 2.9 shows how the RRT can improve but does so randomly. In order to improve on the solution in a meaningful way, a method of selecting a better path must be built into the algorithm. The RRT* (RRT-Star) is a modification to the RRT that improves the solution and is able to produce an asymptotically sub-optimal solution.

2.5 RRT*

The RRT* algorithm was first introduced in 2010 and provides a significant improvement in the quality of the paths discovered in the configuration space of the robot. The quality of the path is determined by the cost associated with moving from the start location to the end location. Quality is not determined by the speed at which any path is found. While RRT* does produce higher quality paths, the algorithm does have a longer execution time. The longer execution time of RRT* is due to the algorithm making many additional calls to the local planner in order to continuously improve the discovered paths. RRT* operates in a very similar way as RRT. The algorithm builds a tree using random samples from the configuration space of the robot and connects new samples to the tree as they are discovered. There are two primary differences between RRT and RRT*. The first difference is in the method that new edges are added to the tree. The second difference is an added step to change the tree in order to reduce path cost, using the newly added vertex. Each of these differences contributes to the improvement of discovered paths over time and is reason RRT* will converge to an asymptotically sub-optimal solution. Fig. 2.10 shows a sample RRT* in the same simple environment as the RRT above. The space is similarly explored, however the structure of the tree is very different.

When a random vertex is added to the tree, the RRT will select the nearest

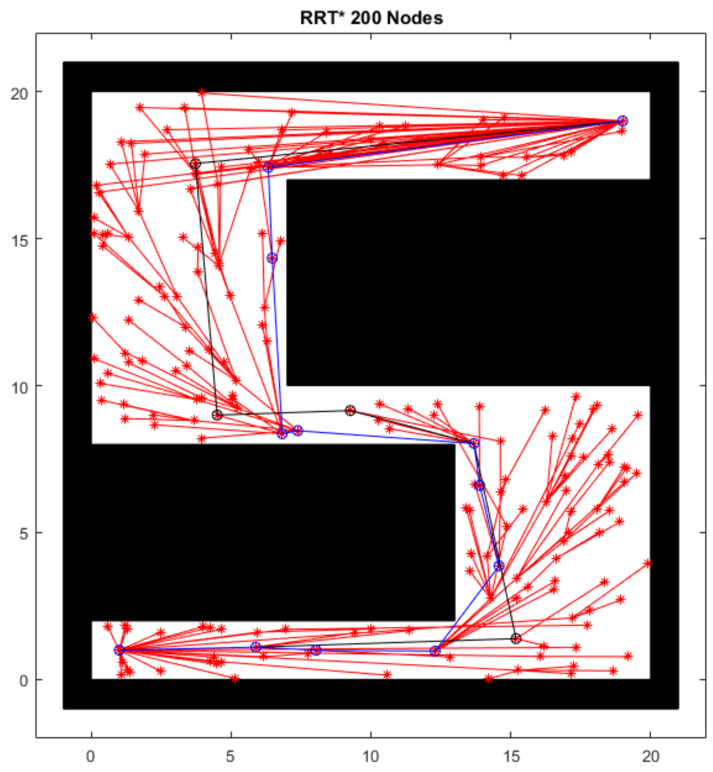


Figure 2.10: A Sample RRT*. The blue line represents the best path found by RRT*. The black line represents the first path found by RRT*.

neighbor as the parent for this new vertex and edge. RRT* will select the best neighbor as the parent for the new vertex. Selecting the best neighbor requires extra steps before adding a vertex to the tree. While finding the nearest neighbor, RRT* considers all the nodes within a neighborhood of the random sample. RRT* will then examine the cost associated with connecting to each of these nodes. The node yielding the lowest cost to reach the random sample will be selected as the parent, and the vertex and edge are added accordingly.

Algorithm 3: $T = (V, E) \leftarrow$RRT*(q_{init})

```
1 $T \leftarrow$InitializeTree()
2 $T \leftarrow$InsertNode($\emptyset$, $q_{init}$, $T$)
3 for $k \leftarrow 1$ to $N$ do
4     $q_{rand} \leftarrow$RandomSample($k$)
5     $q_{nearest} \leftarrow$NearestNeighbor($q_{rand}$, $Q_{near}$,$T$)
6     $q_{min} \leftarrow$ChooseParent($q_{rand}$, $Q_{near}$, $q_{nearest}$, $\Delta$q)
7     $T \leftarrow$InsertNode($q_{min}$, $q_{rand}$, $T$)
8     $T \leftarrow$Rewire($T$, $Q_{near}$, $q_{min}$, $q_{rand}$)
9 end
```

The RRT* algorithm begins in the same way as the RRT. The tree is initialized and the starting location is added to the tree, see Algorithm 3 lines 1-2. RRT* also selects random samples in the same way as RRT. On line 5 of Algorithm 3 the algorithm selects the nearest neighbor to the random sample. The function also selects the set of nodes, Q_{near}, in the tree that are in the neighborhood of the random sample q_{rand}. Line 6 of Algorithm 3 is the first major difference between RRT* and the RRT. Instead of selecting the nearest neighbor as the parent of the random sample, the $ChooseParent()$ function will select the best parent from the neighborhood of nodes.

Algorithm 4: q_{min} ←ChooseParent(q_{rand}, Q_{near}, $q_{nearest}$, Δq)

```
1  $q_{min} \leftarrow q_{nearest}$
2  $c_{min}$ ←Cost($q_{nearest}$) + c($q_{rand}$)
3  for $q_{near} \in Q_{near}$ do
4      $q_{path}$ ←Steer($q_{near}$, $q_{rand}$, Δq)
5      if ObstacleFree($q_{path}$) then
6          $c_{new}$ ←Cost($q_{near}$) + c($q_{path}$)
7          if $c_{new} < c_{min}$ then
8              $c_{min} \leftarrow c_{new}$
9              $q_{min} \leftarrow q_{near}$
10         end
11     end
12 end
13 return $q_{min}$
```

Algorithm 4 describes the *ChooseParent*() function. This function must keep track of which node has the lowest total cost for reaching q_{rand}. At line 1 of Algorithm 4 the nearest neighbor, $q_{nearest}$, is considered the minimum cost neighbor, or q_{min}. On line 2 the cost associated with reaching the new random sample q_{rand} by using $q_{nearest}$ as the parent is stored as the current best cost, or c_{min}. The algorithm then searches the set of nodes in the neighborhood of q_{rand}. The *Steer*() function on line 4 of Algorithm 4 is a variation of the *Extend*() function from the RRT. The *Steer*() function will extend the tree but will not add the node and edge to the tree. Instead the *Steer*() function will return a path from the nearby node, q_{near} to the new random node q_{rand}. If this returned path is obstacle free then a cost comparison is done on the new path, designated q_{path}. Line 6 of Algorithm 4 stores the new cost associated with going from the current nearby node q_{near} to q_{rand}. If this cost is lower than the current minimum cost, then the nearby node becomes the best neighbor, q_{min} and this cost becomes the best cost c_{min} (lines 7-9 of Algorithm 4). If the cost is not lower than the current minimum cost, then the next node in the neighborhood is selected. When all nearby nodes have been examined the function returns the best neighbor. The new random node is inserted into the tree using the best neighbor from the $ChooseParent()$ function as the parent node in the tree. The next step is the second major difference between the RRT* and the RRT algorithms. Line 8 of Algorithm 3 calls the $Rewire()$ function.

Algorithm 5: $T \leftarrow$Rewire(T, Q_{near}, q_{min}, q_{rand})

```
1 for $q_{near} \in Q_{near}$ do
2     $q_{path} \leftarrow$Steer($q_{rand}$, $q_{near}$)
3     if $ObstacleFree(q_{path})$ and $Cost(q_{rand}) + c(q_{path}) < Cost(q_{near})$ then
4         $T \leftarrow$ReConnect($q_{rand}$, $q_{near}$, $T$)
5     end
6 end
7 return $T$
```

The $Rewire()$ function, described in Algorithm 5, changes the tree structure based on the newly inserted node q_{rand}. This function again uses the nearby neighborhood of nodes, Q_{near}, as candidates for rewiring. The $Rewire()$ function again uses the $Steer()$ function to get the path, except this time the path will start from the new node, q_{rand} and go to the nearby node q_{near}. If this path is obstacle free and the total cost of this path is lower than the current cost to reach q_{near} (line 3 of Algorithm 5).

Then the new node q_{rand} is a better parent than the current parent of q_{near}. The tree is then rewired to remove the edge associated with the current parent of q_{near}, and add an edge to make q_{rand} the parent of q_{near}. This is done using the $ReConnect()$ function on line 4 of Algorithm 5.

The functions $ChooseParent()$ and $Rewire()$ change the structure of the search tree when compared to the RRT algorithm. The tree generated by the RRT has branches that move in all directions. The tree generated by the RRT* algorithm rarely has branches that move back in the direction of the parent. The $ChooseParent()$ function ensures edges are created and always moving away from the start location. The $Rewire()$ function changes the internal structure of the tree to ensure internal vertices do not add unnecessary steps on any discovered path. Fig. 2.11, below, is a small tree intended to show how the branches of the tree only move away from the parent node without child nodes that return in the direction of the grandparent node. The $ChooseParent()$ and $Rewire()$ functions guarantee the paths discovered will be asymptotically sub-optimal because these functions are always minimizing the costs to reach each node within the tree.

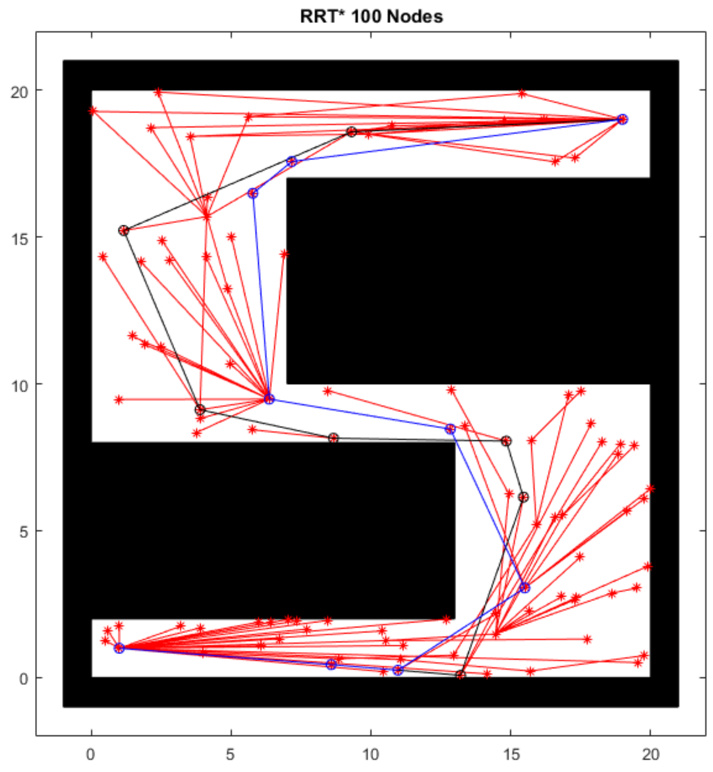


Figure 2.11: A sample RRT*. The blue line represents the best path found by RRT*. The black line represents the first path found by RRT*.

Chapter 3

Static Environment Experiments

3.1 The Environment

The environment for all of the experiments is a complex 2-D environment that will also serve as the configuration space for the robot. The environment is complex due to the number of obstacles. The environment also contains several narrow passages, which can be difficult for sampling based planners to overcome. There are also several sub-optimal paths where the algorithm may get stuck. For each experiment the path cost is measured in Euclidean distance. The environment is also intended to mimic a potential real world situation where there will be streets or sidewalks and open areas such as parks and plazas. Fig. 3.1 shows the environment used. The green circle in the upper left corner is the goal location and the blue circle near the bottom center is the start location.

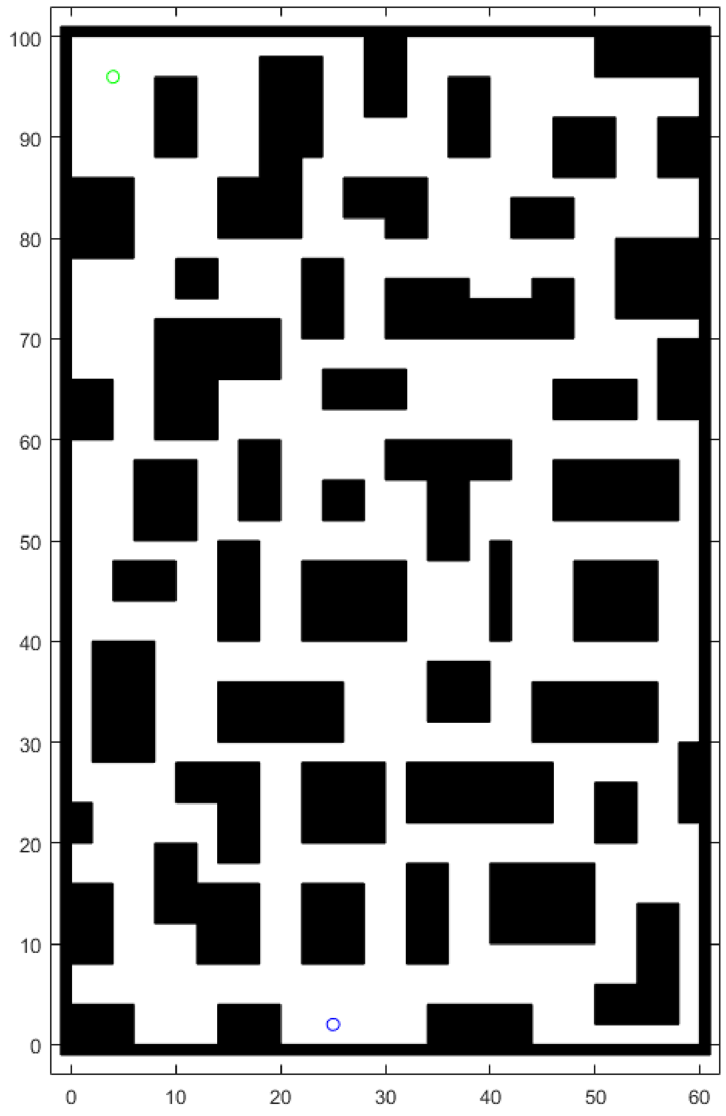


Figure 3.1: The environment and configuration space for the robot.

3.2 RRT Experiments

The static environment experiments begin with the RRT. The implementation of the RRT does not use a growth factor, instead the RRT will attempt to connect directly to the random sample. This means the RRT will aggressively explore the configuration space. The RRT is also goal oriented, meaning after every new node is inserted into the tree the algorithm will attempt to connect to the goal from the new node. These two properties of the algorithm do increase execution time, however the trade off is rapid and expansive space exploration. Since the environment has some areas with long straight regions of obstacle free space; the absence of a growth factor allows the algorithm to create branches reaching up to the full length of these regions.

The first experiment with the RRT was conducted using a maximum of 1000 nodes. Fig. 3.2 shows the result of this experiment. The tree has explored most of the space and would connect to a goal location anywhere within the configuration space. This indicates good coverage by the algorithm. The nodes within the tree are emphasized in Fig. 3.2 to show their coverage. It is also easier to see where branches overlap or where long winding branches make their way around the obstacles. In this experiment the RRT found the goal location quickly, but was unable to find a better path than the first path found. This path has a length of 115.973 units.

The second experiment with the RRT used a maximum of 2000 nodes. This tree has explored all of the area and reached around every obstacle. Fig. 3.3 shows the result of the experiment. In this experiment the RRT was able to slightly improve upon the first path it found. Even with the total nodes double from the first experiment the RRT struggles to find better solutions. This experiment also has a longer path length than the first experiment, 138.729 versus 115.973. There are several instances where the paths backtrack, this is from the randomness of the RRT.

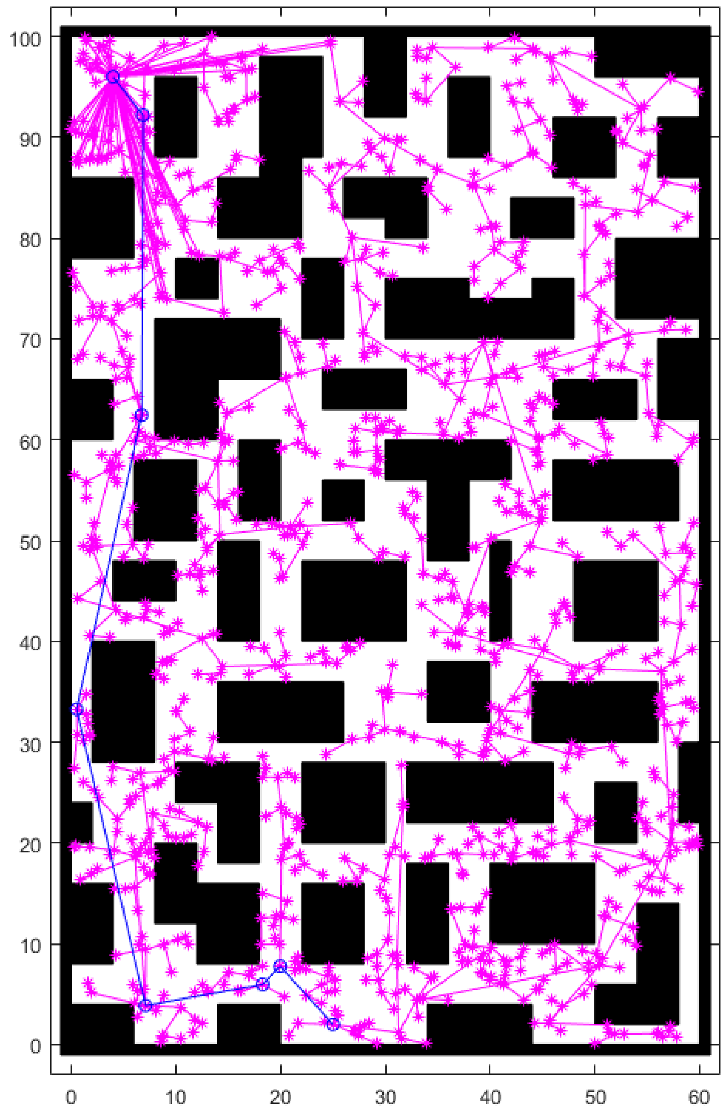


Figure 3.2: RRT with 1000 nodes.

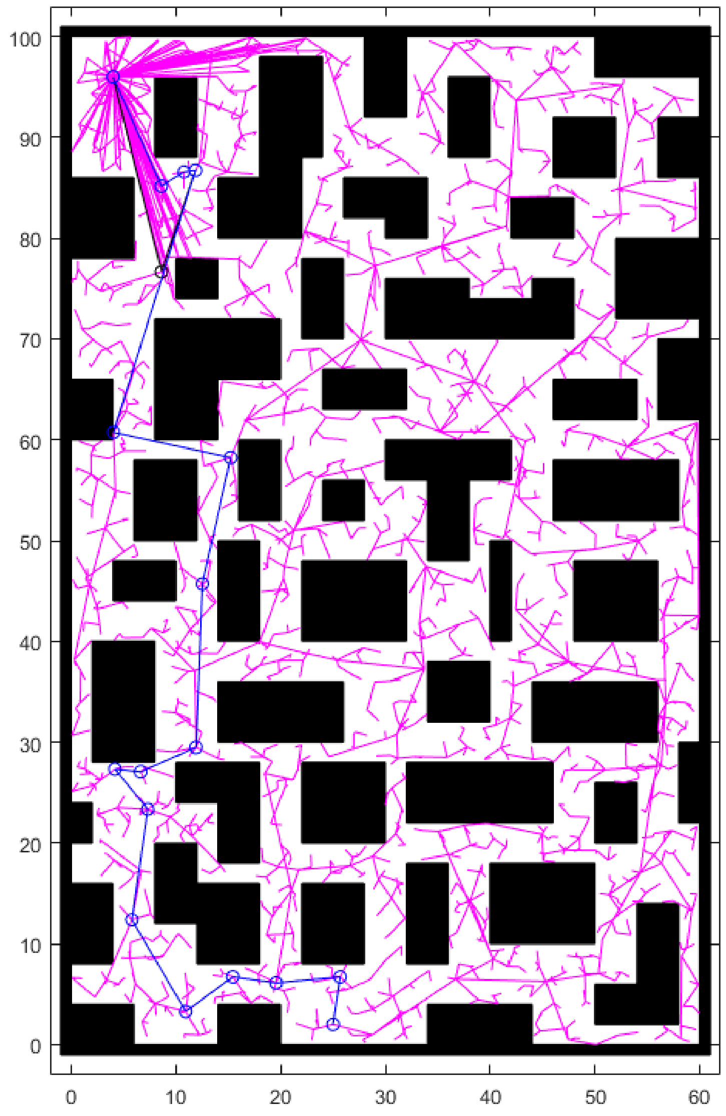


Figure 3.3: RRT with 2000 nodes.

The third experiment using the RRT has a maximum of 5000 nodes. The tree explored the entire configuration space and has branches around all of the obstacles, similar to the second experiment. However, this time the RRT found a good path on it's first try and did not find any improvement to the path as the algorithm progressed. Fig. 3.4 shows the result of the experiment. This time the path length was 117.278 units.

The fourth experiment using the RRT has a maximum of 10000 nodes. The tree is dense as expected, however it is very inefficient. Fig. 3.5 shows the result of this experiment. The tree has branches that reach around obstacles from the wrong direction and block a more optimal path. The best path found in this experiment was 166.883, significantly higher cost than the RRT with a maximum of just 1000 nodes.

The final experiment using the RRT has a maximum of 20000 nodes. Once again the tree is very dense, it would be difficult to find a point that is not within 1 unit of a node. The paths found by the algorithm are very inefficient. The tree generated this time is not better than any of the other experiments. The best path generated was 135.450. Fig. 3.6 shows the results of the experiment.

Each of the RRT experiments contains an inefficient tree for reaching the goal. While there are cases where the RRT has a decent path cost, there are often inefficient branches. In the cases where the algorithm found a better path than the first; the improvement was very close to the goal. In each experiment the RRT would find a solution very quickly, within the first few hundred nodes. Since the additional nodes provide very little improvement over the first path, these extra nodes are a waste of time.

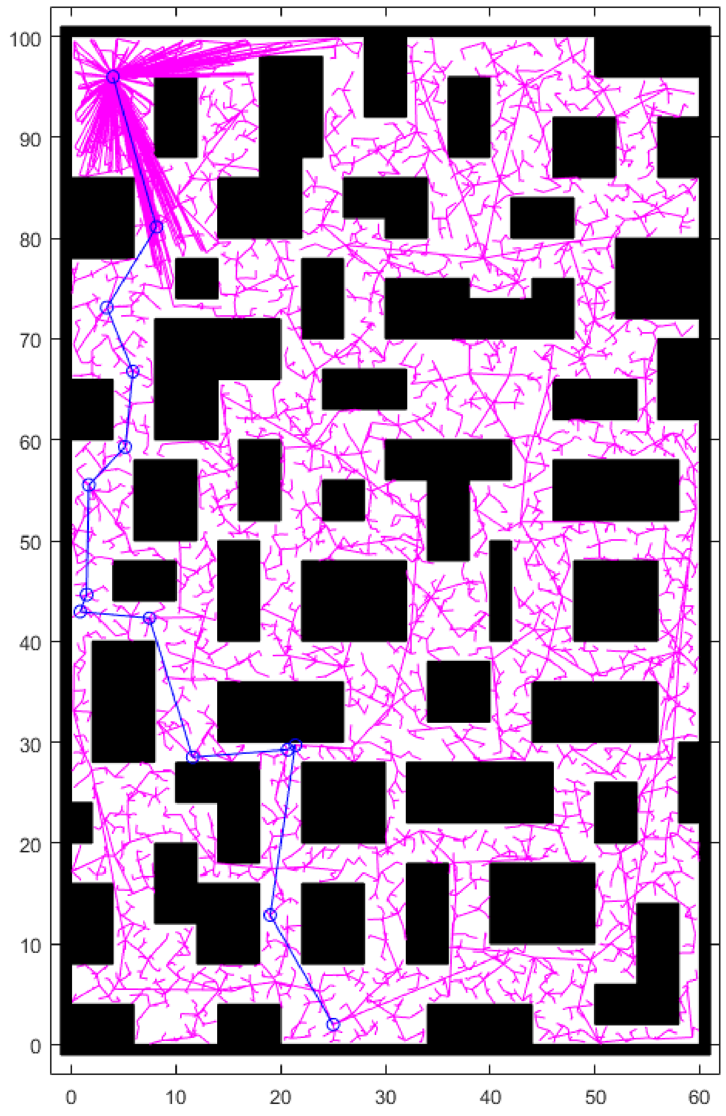


Figure 3.4: RRT with 5000 nodes.

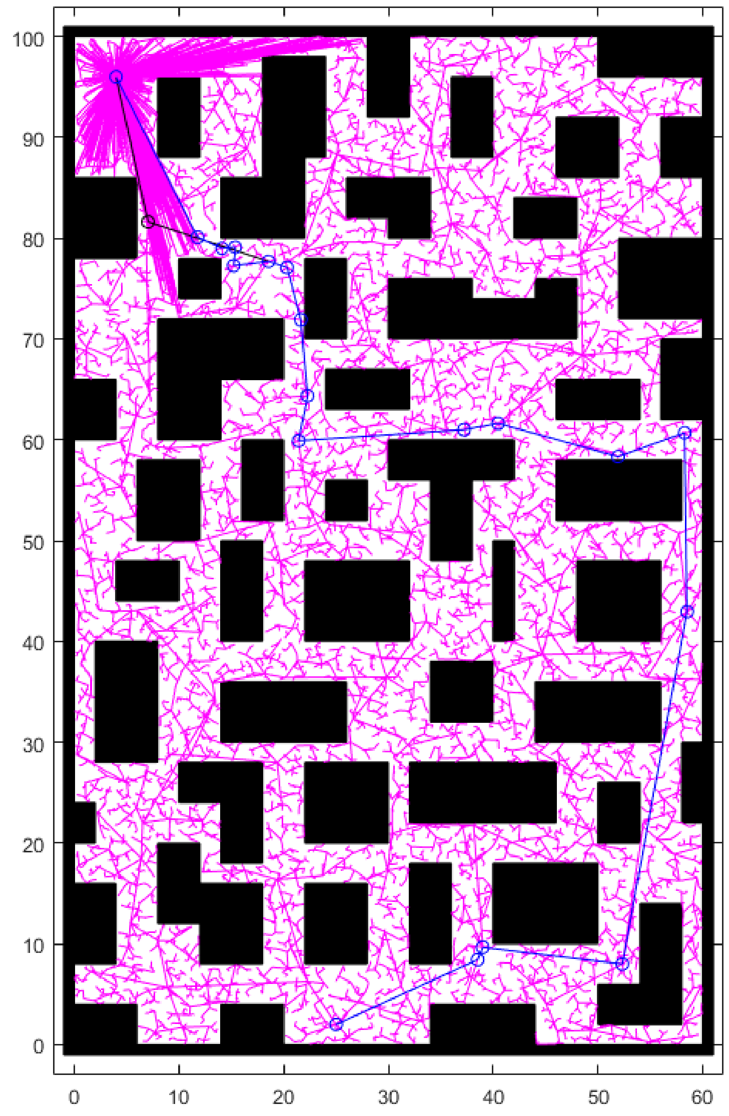


Figure 3.5: RRT with 10000 nodes.

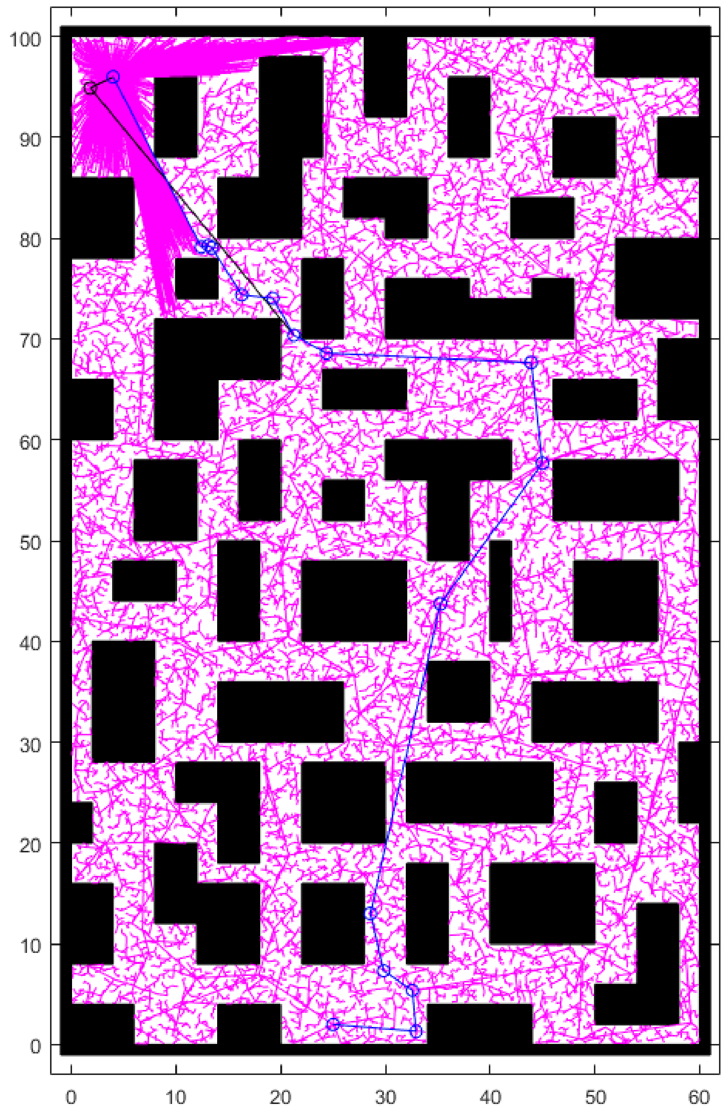


Figure 3.6: RRT with 20000 nodes.

3.3 RRT* Experiments

The RRT* experiments were conducted in the same way as the RRT experiments. There is not a growth factor for extending the tree and the tree is goal oriented, it will attempt to connect to the goal after each node is added to the tree. As with the RRT experiments there are expected to be long branches. These branches are often inefficient, the RRT* $Rewire()$ function will remove these long inefficient branches as the algorithm executes in favor of shorter, lower cost branches. The algorithm also has a maximum number of nearest neighbors, this is configurable to the algorithm. This implementation of RRT* has a maximum number of nearest neighbors equal to 1% of the total number of nodes. Having the number of nearest neighbors as a function of the total number of nodes maintains consistency throughout the experiments. If a fixed number is used it may bias the results based on the total number of nodes. If the number is too small, then too few neighbors may be examined when there are many nodes nearby. If the number is too large then too many neighbors will be examined when the total number of nodes is small.

The first experiment with RRT* is similar to the first with the RRT. Fig. 3.7 shows the results. The best path found has a cost of 110.210. This tree also has 1000 nodes and the nodes are emphasized to show the shape of the tree. In the RRT* tree the branches tend to move out from a central node within an area of the configuration space. This is from the selection of the best parent from the nearby nodes. There are two paths plotted in Fig. 3.7. The black line is the first path found by the algorithm and the blue line is the best path found.

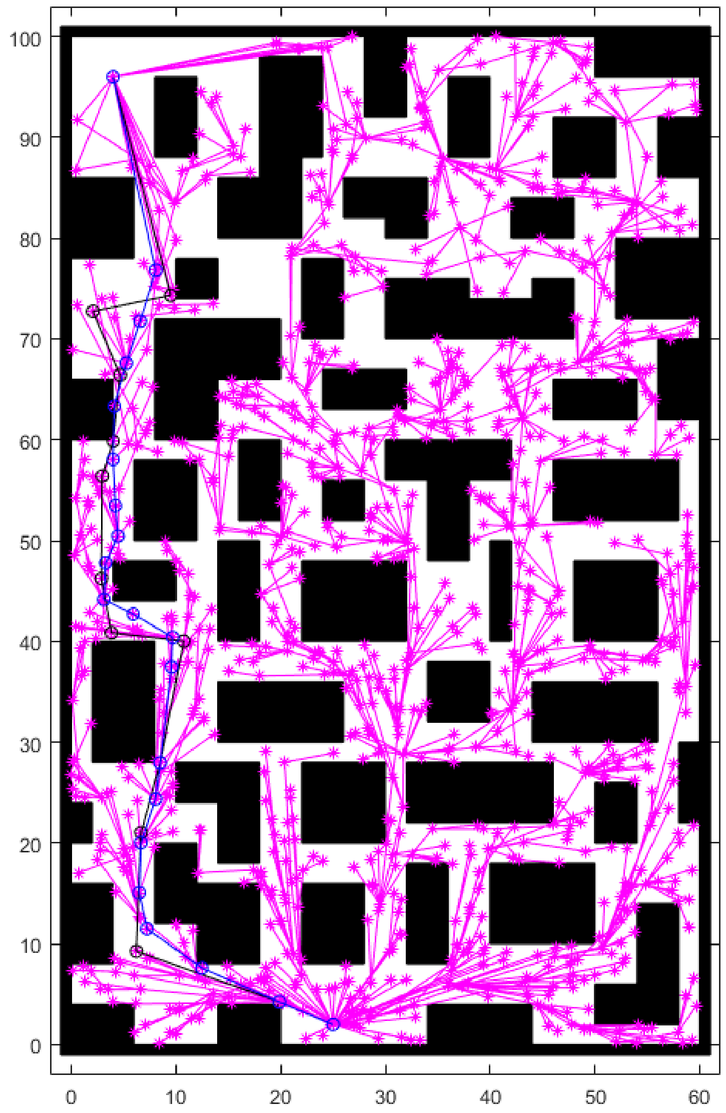


Figure 3.7: RRT* with 1000 nodes.

The second experiment with RRT* contains 2000 nodes, similar to the second experiment with the RRT. Fig. 3.8 shows the results of this experiment. This tree also shows how branches are always spreading out from a central location. In this simulation the tree found a path through the more open spaces of the configuration space. The optimal path found in this experiment is clearly an improvement over the first path found. However, the algorithm was not able to find the true optimal path. The best path found in this experiment has a cost of 105.571

The third experiment with RRT* contains 5000 nodes. Fig. 3.9 shows the results of this experiment. The path found in this experiment is better than the solutions found in the previous two. The cost of the best path found was 103.960. This tree again did not find the optimal path in the configuration space. The expected trend is a small but steady improvement to the path as more nodes are added to the tree.

The fourth experiment with RRT* contains 10000 nodes. Fig. 3.10 shows the tree and optimal path generated in this experiment. This tree finds the route for the optimal path and generated a path along that route. This tree contains double the number of nodes as the previous experiment. However, the improvement is not a large step. The cost of the best path in this tree is 103.040, an improvement of less than 1 unit. Doubling the number of nodes from 1000 to 2000 provided a larger improvement in the optimal path length than doubling the nodes from 5000 to 10000.

The last experiment with RRT* contains 20000 nodes. Fig. 3.11 shows the results of this experiment. As in the previous experiment this tree finds the route for the optimal path. However, the path cost improvement over the previous experiment is minimal. The cost of the best path in this experiment is 102.920.

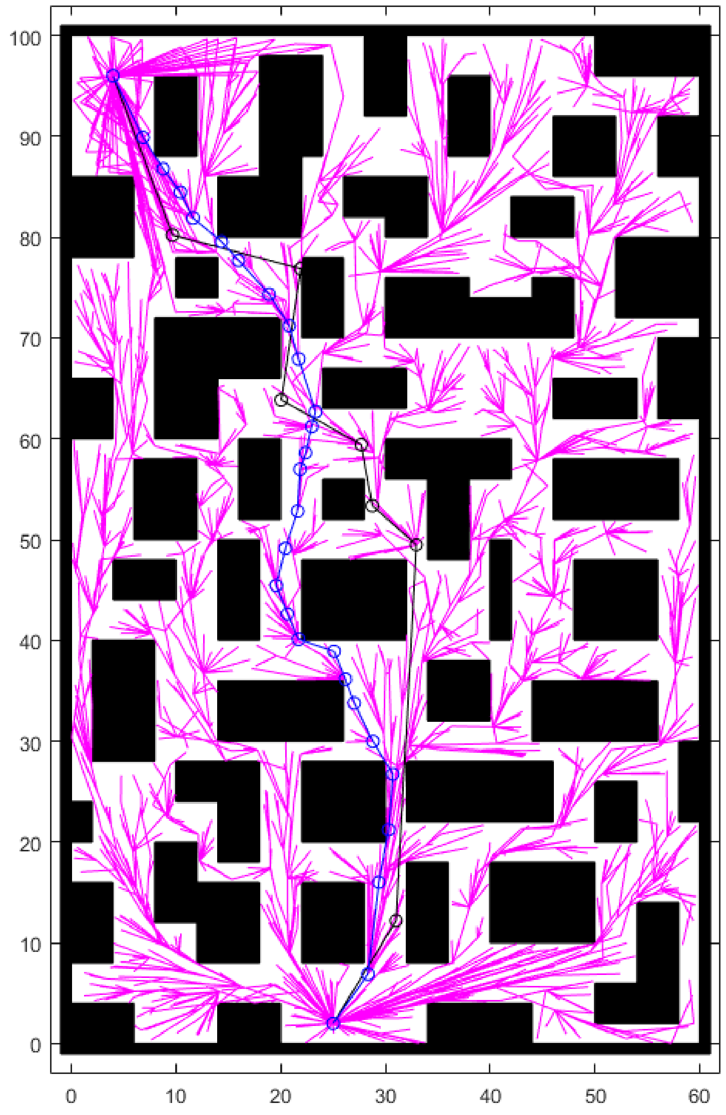


Figure 3.8: RRT* with 2000 nodes.

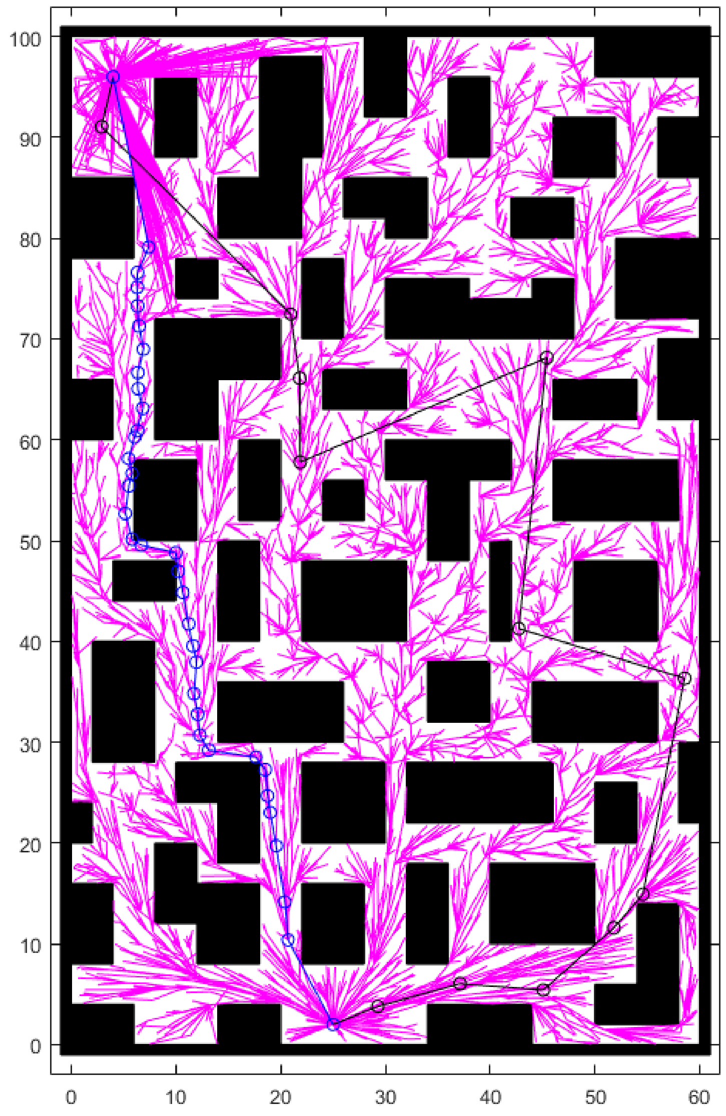


Figure 3.9: RRT* with 5000 nodes.

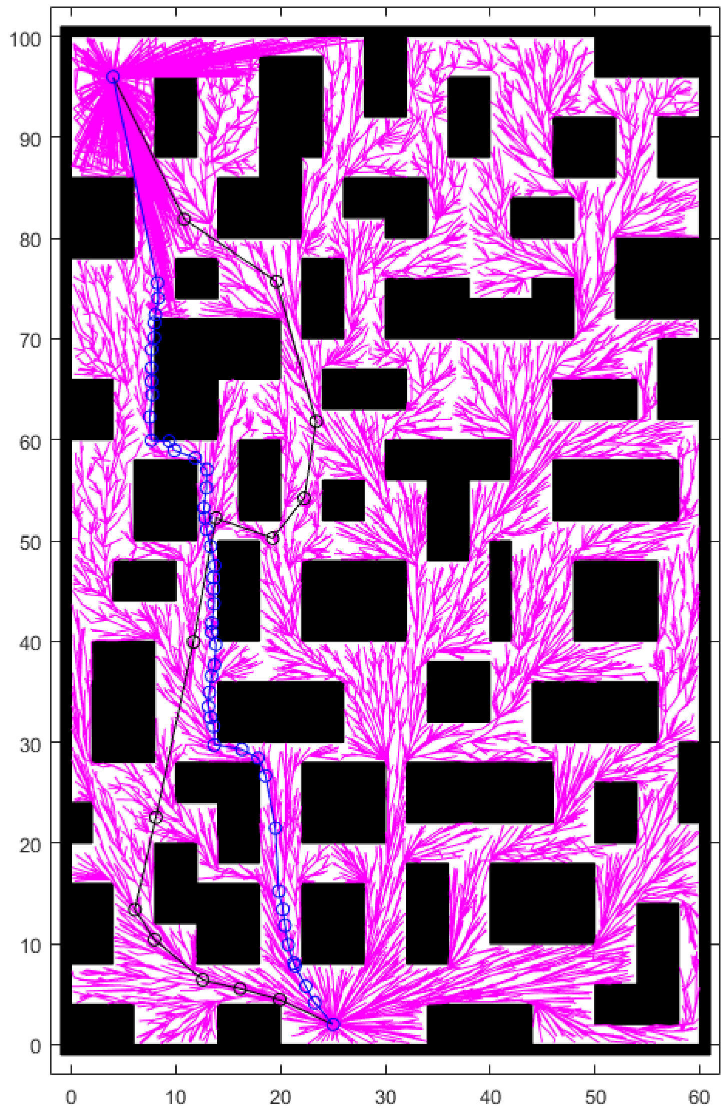


Figure 3.10: RRT* with 10000 nodes.

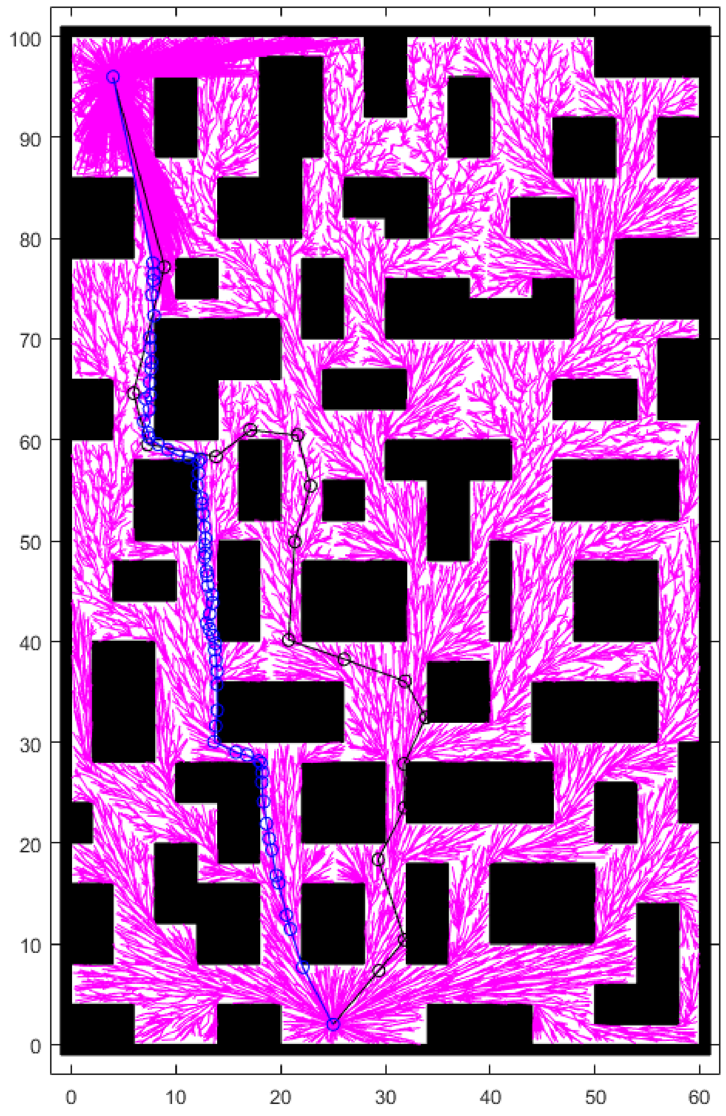


Figure 3.11: RRT* with 20000 nodes.

3.4 Comparison Results

The above experiments show one simulation with a search tree of a particular size. To be sure the optimal path numbers were consistent; both RRT and RRT* were run 100 times using a search tree size of 5000 nodes. During each run the search trees will find many different paths and a best, or optimal, path. For each simulation run the path costs are averaged and stored, similarly the optimal path cost for each run is stored. Fig. 3.12 shows the average path cost over the 100 simulation runs.

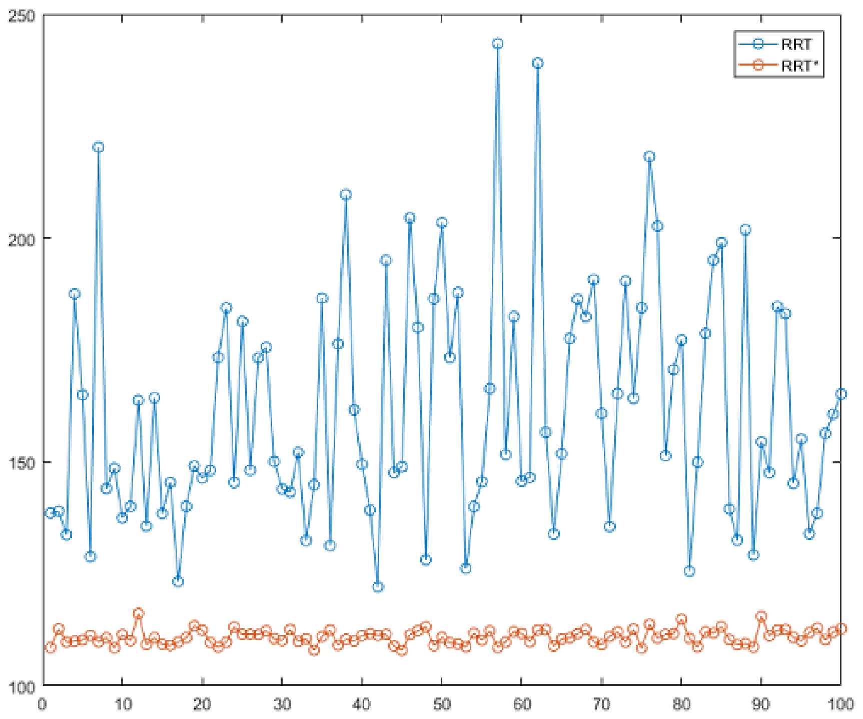


Figure 3.12: Average Path Cost for 100 RRT and RRT* simulations.

The average path cost for the RRT* algorithm is very consistent, the average path cost for the RRT is not. When examining the optimal path cost for each of the 100 simulations, shown in Fig. 3.13, the average path cost is very similar. This is good for the RRT* algorithm, it is not for the RRT algorithm. These two figures show how the RRT* algorithm will converge the discovered paths towards an optimal solution. These figures also prove the randomness of the RRT. There is no convergence with the RRT paths. The average path cost is just as inconsistent as the cost of the best path.

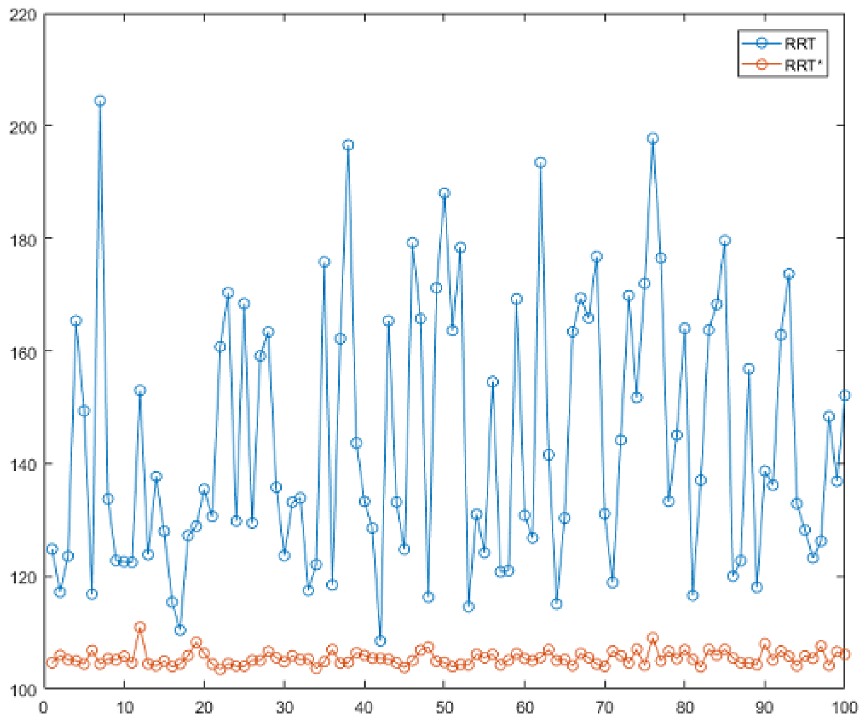


Figure 3.13: Optimal Path Cost for 100 RRT and RRT* simulations.

The experiments conducted demonstrate how the RRT* algorithm will converge to an asymptotically sub-optimal solution, whereas the RRT algorithm will not. The trade-off in reaching this asymptotically sub-optimal solution is the number of nodes needed and the execution time to find the solution. Fig. 3.14 shows the execution time required for each experiment. The blue line is the execution time of the RRT* algorithm and the red line is the execution time of the RRT algorithm. The execution time of RRT is nearly linear, when the number of nodes in the tree is doubled, the execution time is approximately doubled as well. The execution time of RRT* is exponential. The additional execution time comes from the many extra calls to the local planner during the $ChooseParent()$ and $Rewire()$ functions. In the experiments above there is very little improvement in the path when using 10000 nodes or 20000 nodes. But, there is a significant increase in execution time. This is where the trade-off exists in the RRT* algorithm. The path can be improved, but the cost will be significant execution time and resources.

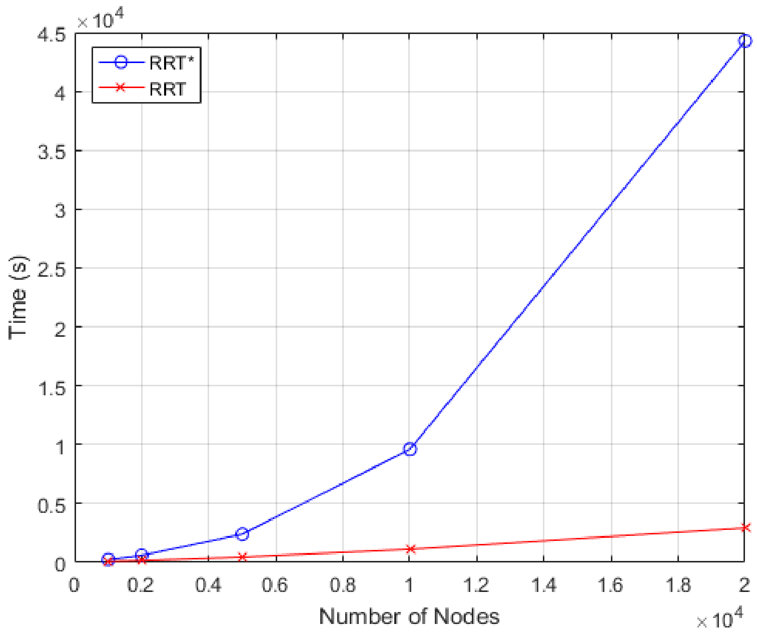


Figure 3.14: Run times of each experiment.

3.5 Summary

This chapter presented the static environment used for all simulations, the experiments and results using the RRT algorithm and the RRT* algorithm. The experiments using the RRT algorithm demonstrate that adding more nodes to the tree does not improve the overal performance of the algorithm. The RRT rarely improves upon the first solution it finds, and when that improvement does occur it is very near the goal node and provides very little improvement to the overall solution. The experiments using the RRT* algorithm demonstrate the algorithm's ability to find an asymptotically sub-optimal solution. As the number of nodes is increased the quality of the solution increases asymptotically toward the optimal path.

Chapter 4

Dynamic Replanning

4.1 Introduction

A real world environment is not static, and often has many moving obstacles. These obstacles are often unpredictable, which makes planning tasks to avoid them difficult. When a moving obstacle is known and is following a known trajectory, the configuration space can be modified to account for this trajectory. When the obstacle is unknown, the robot will need to be able to dynamically determine a course of action in order to avoid a collision. In this chapter a method of dynamic replanning is presented in order to avoid a random obstacle when it is detected by the robot.

4.2 Simulation Environment

For the following simulations the environment remains very similar to the static environment experiments in the previous chapter. The robot is given a configuration space from which to build a tree using RRT* and determine the best path to reach the goal configuration from the start configuration. During the simulation a few random moving obstacles are added to the environment, described in the next section. These obstacles represent a region of the configuration space that would be a collision if the robot were to enter.

4.2.1 Path Execution

After the initial planning process, the robot begins to execute the optimal path found by the search tree. The robot selects the next node along the optimal path and sets a velocity vector to follow the edge to that vertex, see lines 3 and 4 of Algorithm 6. The robot position is updated based on the robot velocity vector. This process is described in Algorithm 7 below. When the vertex is reached the robot changes the velocity vector to move toward the next node, or destination. This process continues until the robot reaches the goal node. If the robot encounters a random moving obstacle that is obstructing the path a replan event occurs. This is described in the following sections.

Algorithm 6: ExecutePath()

```
1  SetObsDestination($numObs$)
2  SetObsVelocities($numObs$)
3  SetRobotDestination()
4  SetRobotVelocity()
5  while $robotLocation != GOAL$ do
6      UpdateObsLocation($numObs$)
7      UpdateRobotLocation()
8      if $Replan$ then
9          DoReplan()
10     end
11 end
```

4.3 Random Moving Obstacles

The random moving obstacles force the robot to dynamically plan around the obstacle using RRT*. In order for the moving obstacles to move about the environment, a graph is created to provide the paths. The edges of the graph provide the paths between the static obstacles, and the vertices of the graph are intersections of the paths. Upon initialization of the simulation the obstacles are placed at random vertices. The vertices are chosen such that the robot will have a chance to move before encountering a random obstacle. When the simulation begins the moving obstacles choose a random adjacent vertex and begins moving toward that vertex, see lines 1 and 2 of Algorithm 6. When the vertex is reached a new random vertex is chosen and the obstacle moves in the new direction, line 6 of Algorithm 6.

4.3.1 Random Obstacle Detection

Robots operating in a real world scenario will have sensors, such as a LIDAR, to detect both static and dynamic obstacles. Sensors are not included in this simulation. Instead a detection range is placed on the robot. The simulation controls whether or not a moving obstacle is within the detection range of the robot. When the positions of both the moving obstacles and the robot are updated by the simulation, the simulation determines if a moving obstacle is within the detection range of the robot (lines 7 and 8 of Algorithm 7). If a moving obstacle is within range the $Steer()$ function is used to determine if any static obstacles are blocking the robot's line of sight to the moving obstacle.

Algorithm 7: UpdateRobotLocation()

```
1  $robotLocation \leftarrow robotLocation + robotVelocity$
2  if $robotLocation == robotDestination$ then
3      $robotDestination$ ←GetNextPathLocation()
4      SetRobotVelocity()
5  end
6  while $obsIndex < numObs$ do
7      $obsDistance$ ←GetDistance($robotLocation$, ObsLocation($obsIndex$))
8      if $obsDistance < robotRange$ then
9          $obs_{path}$ ←Steer($robotLocation$, $obsLocation$)
10         if ObstacleFree($obs_{path}$) then
11             if IsPathBlocked($obsIndex$) then
12                 Replan ←TRUE
13             end
14             SetObsVisible($obsIndex$)
15         end
16     end
17 end
```

The obstacle must be observed for a minimum of two time steps in order to determine the direction the obstacle is moving. Once the direction is observed the robot can determine if the moving obstacle is blocking the path or not, line 11 of Algorithm 7. If the robot decides the path is blocked, the replanning event begins.

4.4 Path Replanning

Path replanning begins with the determination of whether or not the moving obstacle is blocking the path, described in the next section. Algorithm 8, below, lists all the steps executed during the replanning process. The next step is to find the location along the optimal path that is beyond the obstacle. Next, the tree generated by RRT* is modified and expanded in order to find a path around the obstacle. Finally, the best path around the obstacle is chosen and the execution of this sub-path begins. Each of these steps is described in the following sub-sections.

Algorithm 8: $T \leftarrow$DoReplan()

```
1 InvalidateNodes()
2 GetReplanGoalLocation()
3 SetReplanSamplingLimits()
4 Rewire($T$, $Q_{all}$, $NULL$, $q_{robot}$)
5 RRT*($q_{robot}$)
6 SetReplanPath()
```

4.4.1 Path Obstruction

The method for determining if the moving obstacle is blocking the path is a series of trigonometric functions using a direction vector from the robot to the moving obstacle and a comparison between the robot velocity vector and the moving obstacle velocity vector. Since the configuration space is 2-Dimensional, the inverse tangent can be used to find the angles of the vectors. To obtain the direction vector to the moving obstacle the equation is:

$$angle_{direction} = atan((Y_{obs} - Y_{robot}), (X_{obs} - X_{robot})). \tag{4.1}$$

Where (X_{robot}, Y_{robot}) is the position of the robot and (X_{obs}, Y_{obs}) is the position of the obstacle. This will return an angle in degrees over the range $(-180, 180)$. Similarly the angle of the robot's velocity vector can be obtained using the following equation:

$$angle_{V_{robot}} = atan(V_j, V_i). \tag{4.2}$$

Where V_i and V_j are the X and Y components of the robot velocity vector. Using the angles from (4.1) and (4.2) the difference can be taken to see if they are similar. If the absolute value of the difference between the two angles is less than some threshold, then the robot is moving toward the moving obstacle. Note, the angle difference is normalized to be in the range$(-180, 180)$ before the absolute value is taken. This is done for all angle comparisons:

$$|angle_{direction} - angle_{V_{robot}}| < angle_{thresh}. \tag{4.3}$$

If the robot is moving in the direction of the random obstacle the velocity vectors are examined. Substituting the obstacle velocity into (4.2) above the angle of the obstacle velocity can be obtained. Next, the differences between the velocity vectors is found:

$$angle_{V_{diff}} = |angle_{V_{robot}} - angle_{V_{obs}}|. \tag{4.4}$$

There are three possibilities from this point. If the angle difference between the velocity vectors is less than the angle threshold, then the robot and the obstacle are moving in a similar direction. Second, if the angle difference between the velocity vectors is greather than $180 - angle_{thresh}$, then the robot and the obstacle are approximately moving toward each other. Last, if the angle difference falls outside of these ranges the moving obstacle and the robot are moving in different directions.

For the first case: The robot will simply follow the obstacle, until the obstacle changes direction, or the path takes the robot away from the obstacle. The robot will then choose from one of the other conditions. For the second case: The robot quickly activates a replan event to get out of the way. The random obstacle may move out of the way on its own, but there is no way of predicting that will occur. Finally, if the robot and the obstacle are moving in different directions, the robot ignores the obstacle unless it gets too close. This third condition catches the event that the robot moves out from a corner and a random obstacle is detected very close by. This event is best summed up with the following example: When two people approach a hallway intersection they will run into each other if they continue on their current course, it is only when they see each other that they can adjust to avoid a collision.

4.4.2 Establish Replanning Goal Location

The second step during the path replanning event, line 2 in Algorithm 8, is to find a location that will navigate the robot around the random obstacle. The first step in this process is to invalidate any node within the tree that is currently in a collision state with the moving obstacle. The only exception is the goal location of the optimal path. After this step an assumption had to be made to simplify and speed up the rest of the replanning process. The assumption is that the robot is currently following the best path in order to reach the goal location and should return to this path after the moving obstacle is avoided. Using this assumption the nodes along the optimal path are examined. If a node is in a collision state with the obstacle, the nodes immediately following are candidates to be the replan goal location. The node on the current path that is immediately following the node that is closest to the obstacle, without colliding, will be the replan goal location.

4.4.3 Search Tree Modification

The third step in the path replanning process is to modify the original search tree in order to find a way around the moving obstacle. First a node is added to the tree at the robot's current location. Then every node is rewired such that the robot's current location becomes the parent of that node, if there is a collision free path between them. Using the distance to the replan goal node as a metric, a sampling area is established, line 3 in Algorithm 8. New nodes are then sampled within this area and added to the tree using RRT*. Since there are already many nodes in the tree only a small number will need to be added. However, the number of nearest neighbors used during the $ChooseParent()$ function and the $Rewire()$ function is increased. This increase allows each new node to direct the existing tree toward the replan goal location.

4.4.4 Sub-path Selection and Execution

When the search tree modification is complete the best path to the replan goal location is found. Then using the same method as before; the next node along the path is chosen and the velocity vectors are set to move the robot toward that location. When the robot reaches the replan goal location, the execution of the original optimal path resumes. If the robot encounters another moving obstacle and determines the path is obstructed again. The replanning is repeated, however the replan goal location will always be a node on the original optimal path.

4.5 Results

The method described above provides the robot a useful way to avoid a random mobile obstacle while executing an optimal path from one location to another. The simulation results shown below demonstrate the robot's ability to plan a path around the moving obstacle and reach the goal. Using RRT* during the replanning step allows an efficient path to be found to avoid the moving obstacle and continue on the original optimal path. Since the obstacles move randomly, it is possible for the robot to execute the optimal path and never be obstucted by a moving obstacle. Only examples where the robot did encounter these obstacles are shown.

The first simulation example has a search tree containing 2000 nodes. Although the path is suboptimal, the robot will assume the path is the optimal path. Fig. 4.1 shows the search tree and the optimal path found, similar to the figures in the previous chapter. The moving obstacles are placed randomly within the configuration space, at the beginning of the simulation.

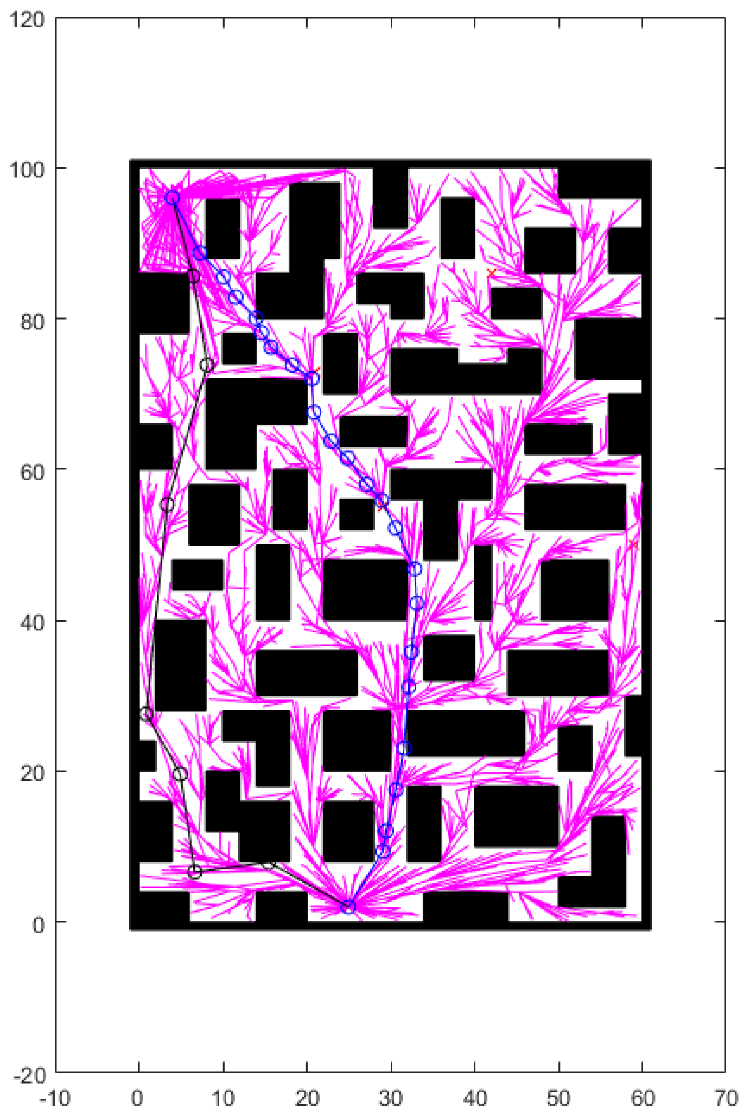


Figure 4.1: Search tree from the first simulation. The blue line is the optimal path found by the search tree. The black line was the first path found.

When executing this path the robot encounters two random obstacles near the center of the configuration space. These obstacles are off the path and moving away when first observed by the robot. However, one obstacle moves back across the path and obstructs the robot. The robot triggers a replanning event at this time. A second obstacle is nearby and can be seen by the robot and must be considered when replanning. Fig. 4.2 is an image from the path execution showing when the robot started a replan event. Following the replanning steps in Algorithm 8, the robot will select a goal location, then modify the search tree to avoid the obstruction.

The next step in the replanning process is to modify the current search tree. This modification changes the original search tree from one that branches out from the starting location, to one that branches out from the robot's current location. Fig. 4.3 shows the full modified search tree. When comparing the original search tree to the modified one, the original branches that lead out from the start location still exist. However, there is now a disconnect from those branches to the branches formed during the replanning modifications. There is an empty region within Fig. 4.3. This region is where the random moving obstacles can be seen by the robot. The abscence of branches within this area shows the robot has considered this space as obstacle space, rather than free space.

When the robot reaches the replanning goal location the original path can resume. In this simulation the robot avoided the obstacle, reached the original planned path and resumed executing that path until it reached the goal. Fig. 4.4 shows the completed path. The magenta line shows the actual path followed by the robot. The sections in blue are from the original path. These sections were skipped due to a replanning event. Additionally, Fig. 4.4 shows the final positions of the random obstacles. Even though one of the obstacles overlaps the path, it was not obstructed when the robot executed that section.

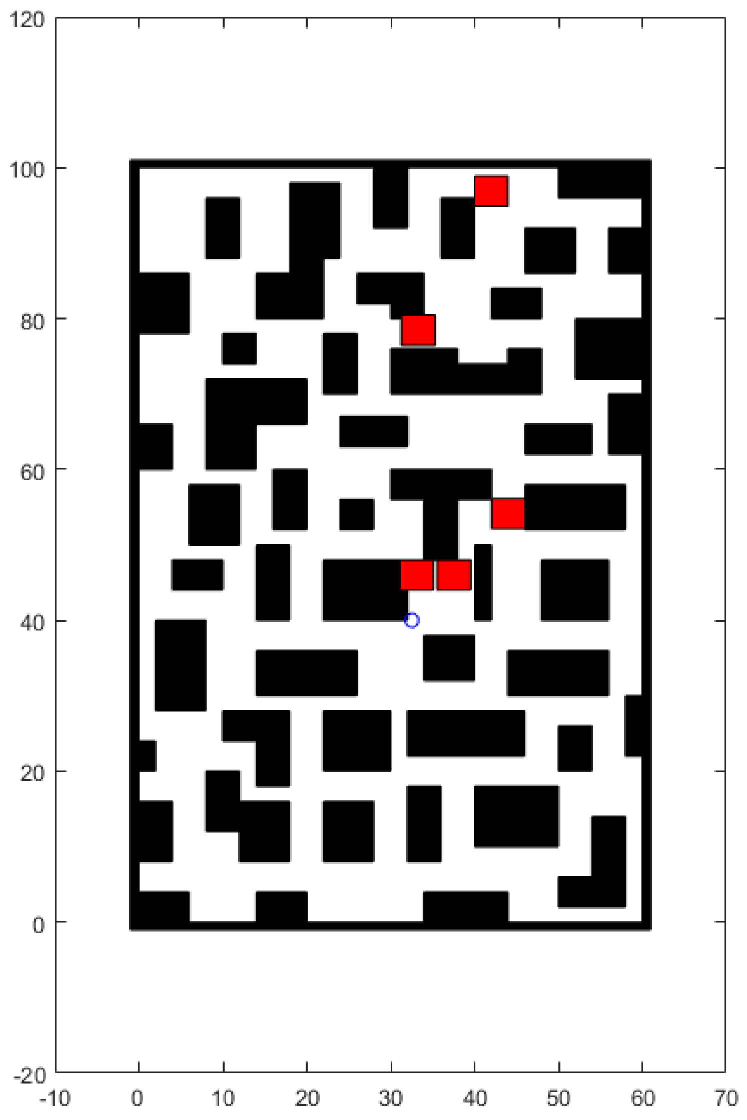


Figure 4.2: A random obstacle, represented by the red square, has blocked the path. The robot must replan to avoid the obstacle.

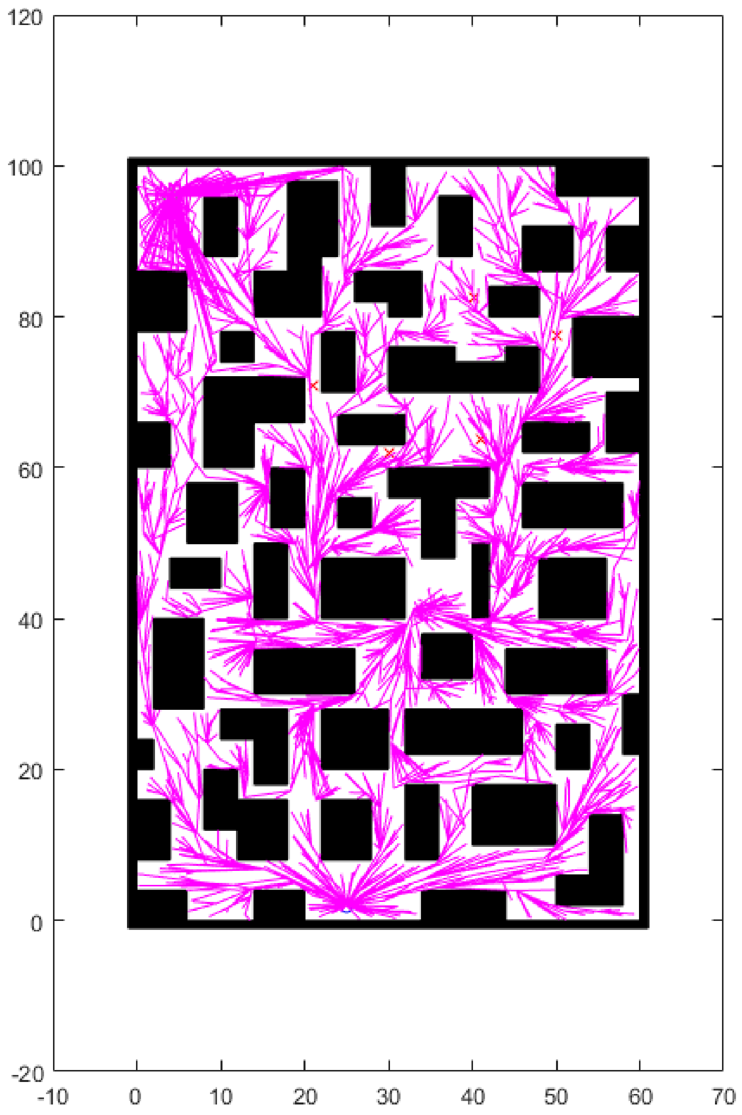


Figure 4.3: The modified search tree.

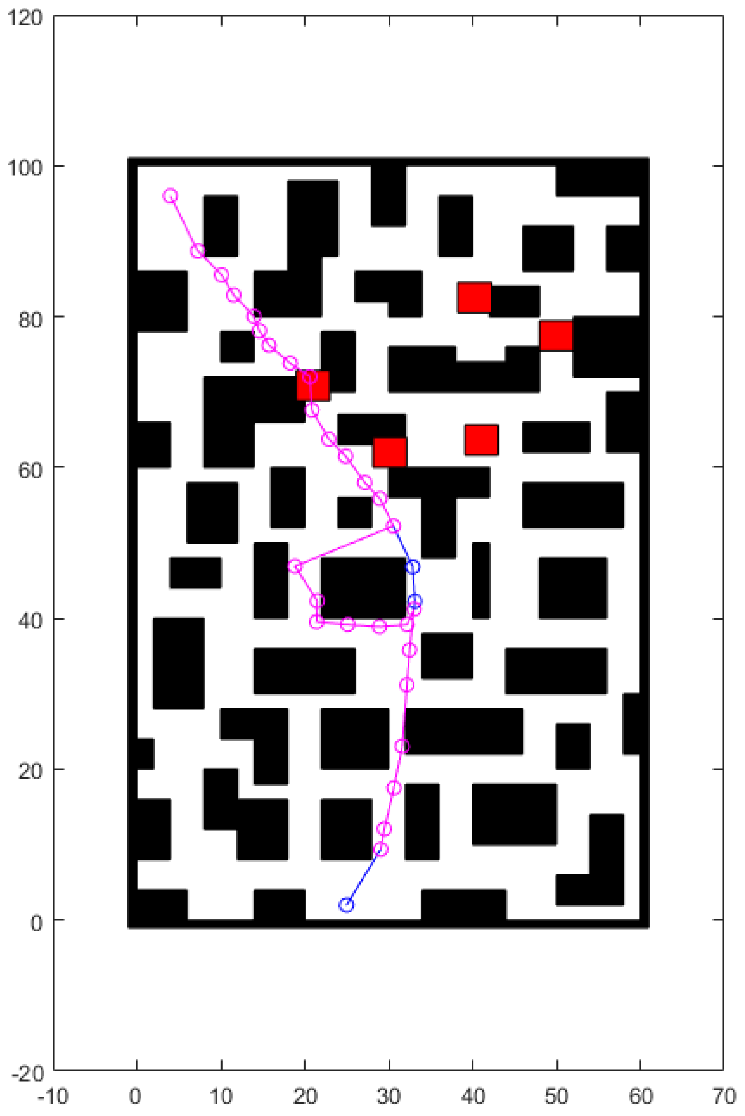


Figure 4.4: The conclusion of the simulation. The magenta line represents the actual path followed by the robot. The blue line in the center are the unexecuted portions of the optimal path.

The second simulation example has a tree similar to the first. There is one minor difference from the first simulation. In the first simulation the moving obstacle initial positions were randomized. In this simulation the random obstacle initial positions were selected in a manner to increase the probability the moving obstacles would block the path. In this simimulation a random obstacle blocked the path three times, causing the robot to replan around the obstacle. Fig. 4.5 shows the initial plan and optimal path found by the robot. Fig. 4.6 shows the configuration space with the random obstacles placed in their initial positions. With the random obstacles placed in these positions the likelihood of one of them to obstruct the robot is very high.

The results of this simulation were very good. The robot was obstructed three times during the execution of the path and successfully planned around the moving obstacle each time. Fig. 4.7 shows the final positions of moving obstacles and the path followed by the robot. The blue sections shown in the figure are portions of the optimal path that were unable to execute due to obstruction by a random moving obstacle. Following the magenta line shows the robot successfully planned and executed a path around those moving obstacles.

The third simulation has a search tree with a maximum of 5000 nodes. The obstacle initial positions were selected in order to increase the probability the moving obstacles would block the path. Fig. 4.8 shows the initial plan and the optimal path found by the robot. The robot encountered a moving obstacle very early in the execution of the path. With the additional nodes in the search tree, the robot is able to quickly find a path around the obstacle. Fig. 4.9 shows the modified search tree after the robot found a path around the obstacle. The robot then executes the path to avoid the obstacle and resumes the original path. Fig. 4.10 shows the final positions of the moving obstacles and the path followed by the robot.

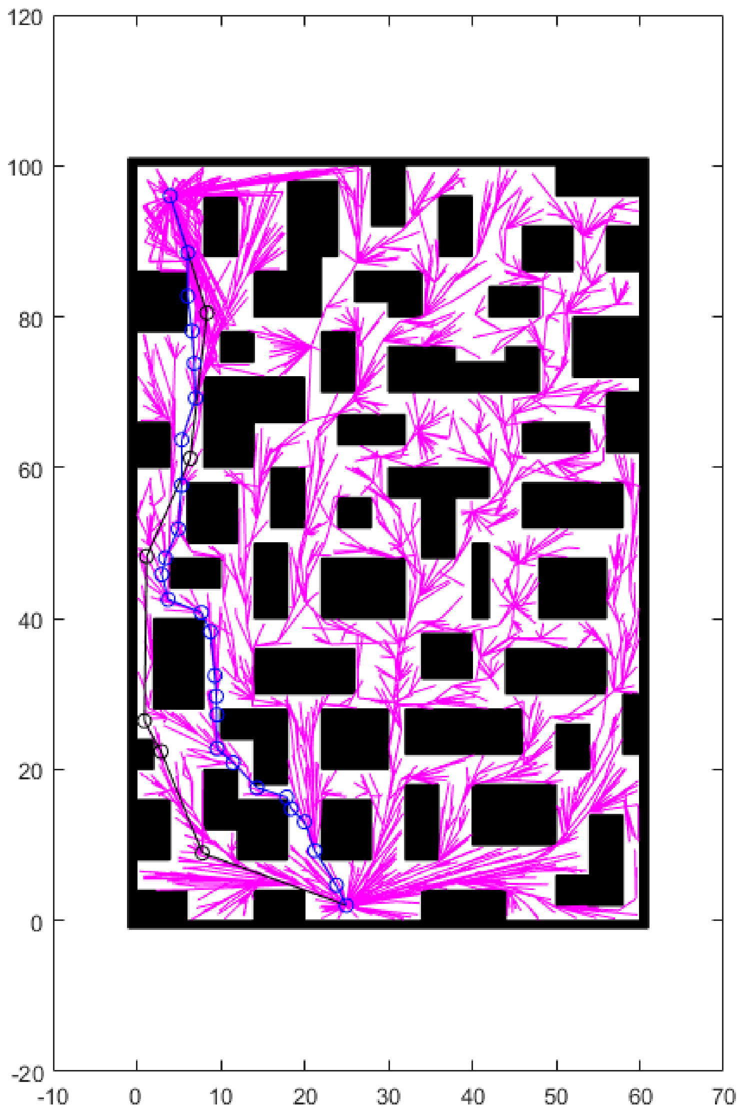


Figure 4.5: Search tree from the second simulation.

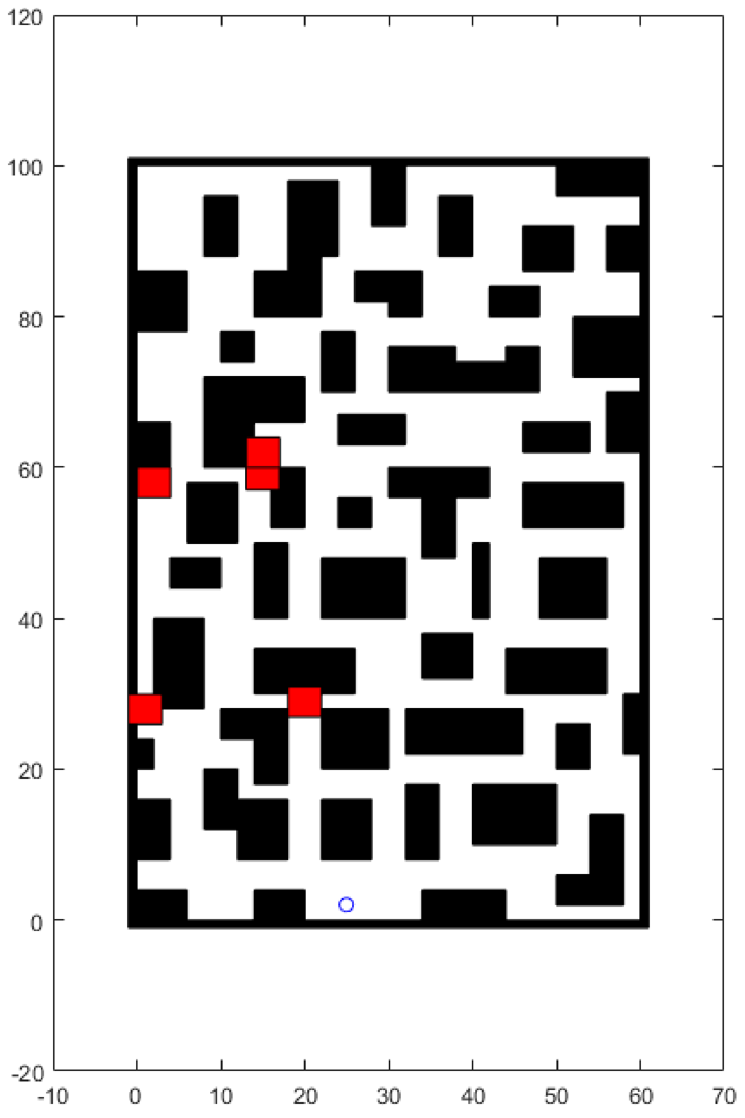


Figure 4.6: The configuration space with the random obstacles, represented by the red squares, in their initial positions.

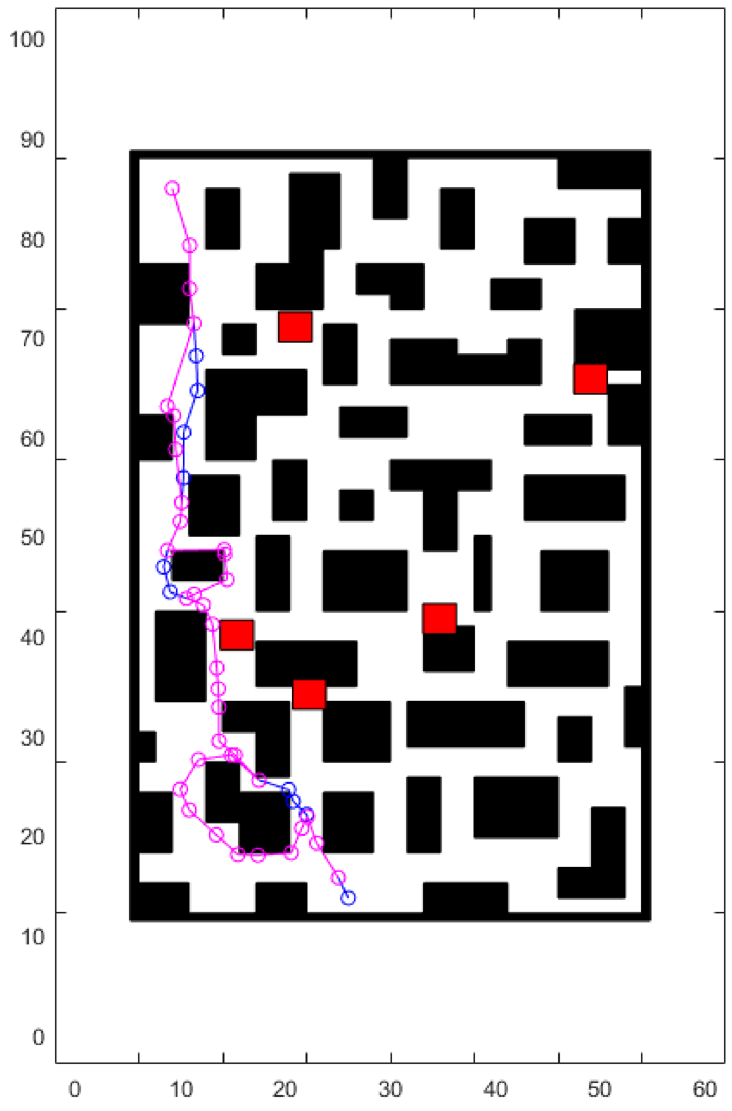


Figure 4.7: The conclusion of the second simulation showing the executed path. The magenta line represents the actual path followed by the robot. The blue line in the center are the unexecuted portions of the optimal path.

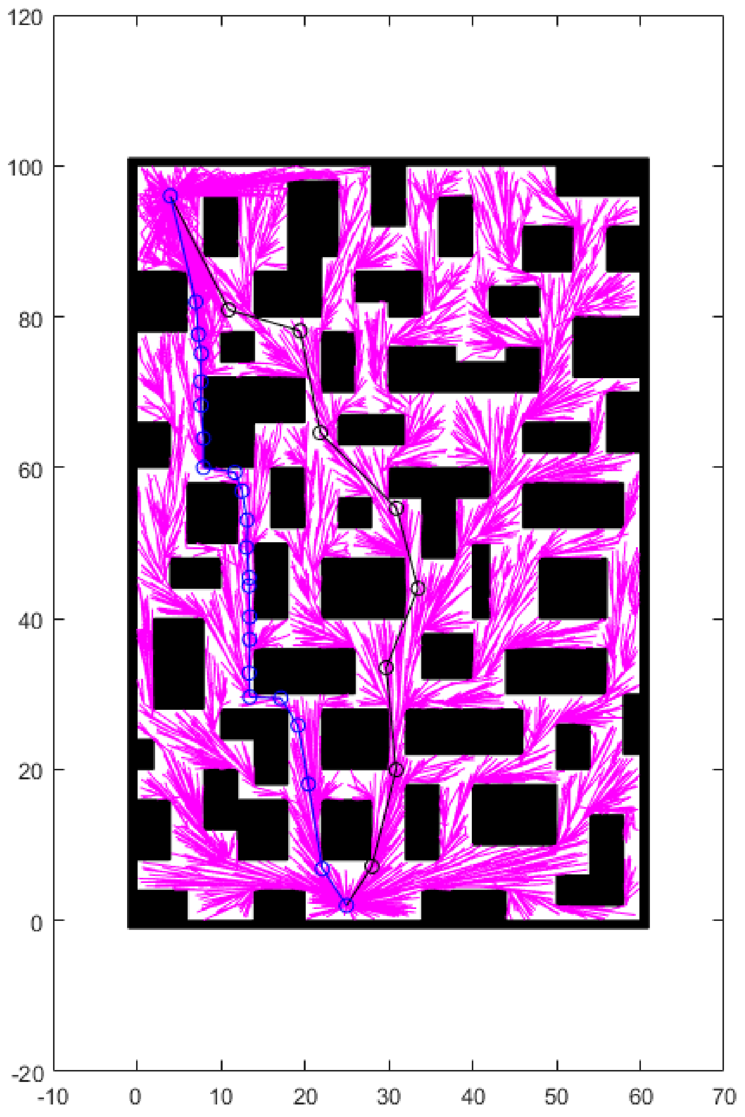


Figure 4.8: Search tree from the third simulation.

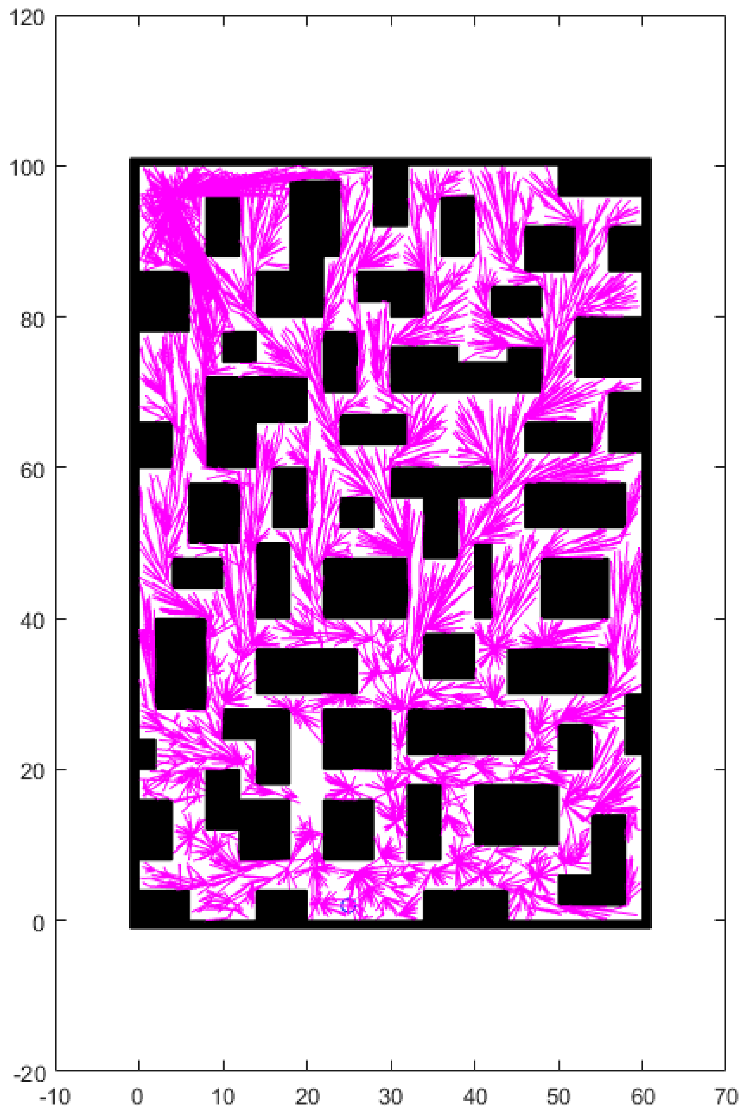


Figure 4.9: The modified search tree from the third simulation.

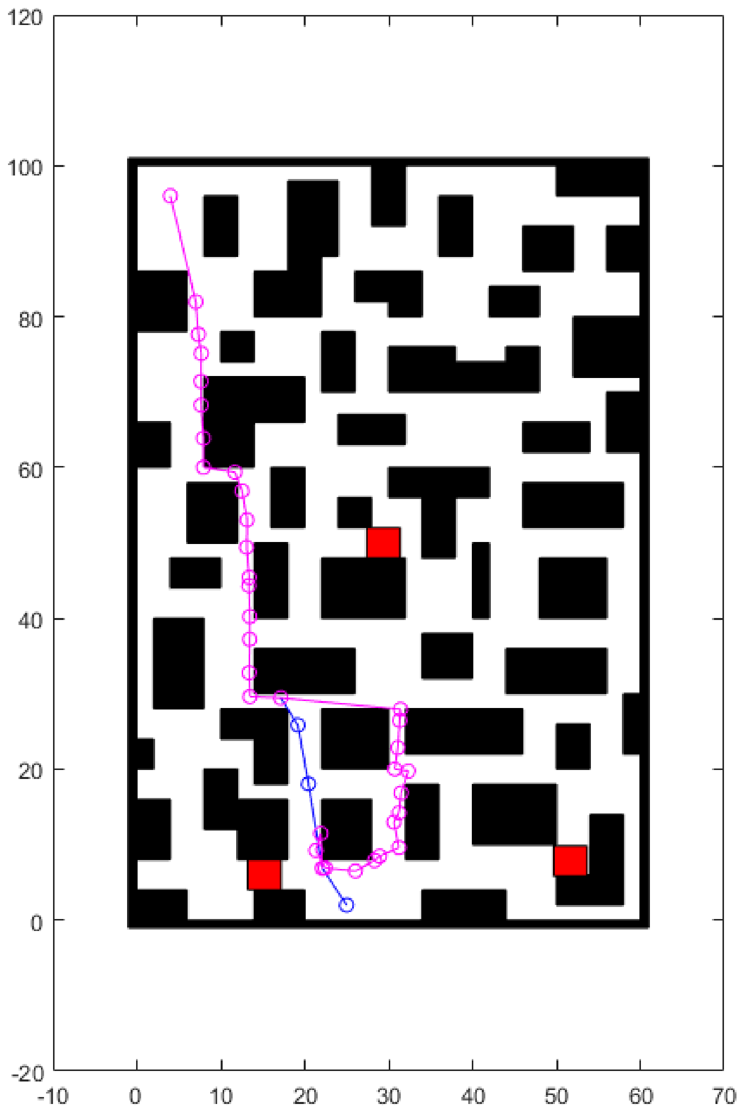


Figure 4.10: The conclusion of the third simulation showing the executed path. The magenta line represents the actual path followed by the robot. The blue lines represent unexecuted portions of the optimal path.

4.6 Summary

The replanning method described above is an effective way for a robot to plan a path around a random moving obstacle that has blocked the path. This method reuses as much of the existing search tree as possible in order to minimize the cost to plan around the obstacle. The $Rewire()$ function from the RRT* algorithm facilitates the reuse of the existing tree to quickly find a replanning solution.

Chapter 5

Multi-Robot Path Planning using RRT*

5.1 Introduction

In some scenarios the use of multiple robots will accomplish a task far more efficiently than a single robot. A task may be to obtain sensor readings, or search a building for people. In these scenarios, each robot will have a starting position and a goal position. With each robot operating in the same environment, the ability to share path planning and search tree information is valuable. The following chapter presents a method of information sharing in which nodes within each robot's search tree are shared among nearby robots in order to obtain path planning results more efficiently.

5.2 Node Description

The nodes, or vertices, used in a RRT search tree must maintain more information than position. Each must store which node is the parent within the tree. The node must also maintain it's cost in order to determine an optimal path. When using RRT* the nodes may also manage a list of nodes within it's neighborhood. The list of nearby nodes is used for rewiring in order to reduce cost when new nodes are added to the tree.

5.3 Multi-robot RRT*

The extension of RRT* to be used by multiple robots is at it's core the single robot version of RRT*. The difference is a few extra steps during the construction of the search tree. The search is paused in order to share nodes between other robots in communication range. Algorithm 9, below, describes the method for using RRT* with multiple robots. The primary functions in the algorithm are the same as the RRT* functions descibed in Algorithm 3 in Chapter 2. The algorithm will continue to add nodes until the maximum number is reached, line 3 of Algorithm 9. Adding nodes to the tree using RRT* is broken up into steps. Each robot will add a predetermined number of nodes to their own trees using the same method as RRT*, see lines 4-10 of Algorithm 9. The next step is to share the newly added nodes with each robot in communication range. When this is complete the nodes received from other robots are added to the tree. The total number of nodes is then updated and the process continues until the maximum number of nodes is reached.

Algorithm 9: $T \leftarrow$MultiRobotRRT*()

```
1  $T \leftarrow$InitializeTree()
2  $T \leftarrow$InsertNode($\emptyset$, $q_{init}$, $T$)
3  for $k \leftarrow 1$ to $N$ do
4      for $l \leftarrow 1$ to $M$ do
5          $q_{rand} \leftarrow$RandomSample($k$)
6          $q_{nearest} \leftarrow$NearestNeighbor($q_{rand}$, $Q_{near}$, $T$)
7          $q_{min} \leftarrow$ChooseParent($q_{rand}$, $Q_{near}$, $q_{nearest}$, $\Delta$q)
8          $T \leftarrow$InsertNode($q_{min}$, $q_{rand}$, $T$)
9          $T \leftarrow$Rewire($T$, $Q_{near}$, $q_{min}$, $q_{rand}$)
10     end
11     ShareNodes($T$, $k$, $M$)
12     AddSharedNodes($T$, $Q_{shared}$)
13     $k \leftarrow M + ||Q_{shared}||$
14 end
```

5.4 Node Sharing

Node sharing will only be executed with robots that are in communication range of each other. In the results below the communication is controlled by the simulation. The nodes are copied from one robot to a buffer of nodes to be used by the neighboring robot. These nodes are not immediately inserted into the search tree. The robot will wait until all nodes from neighboring robots are transferred before adding them to the tree.

5.5 Adding Shared Nodes using RRT*

The next step is to add the nodes received from neighboring robots to the search tree. First, the entire buffer of shared nodes is appended to the search tree list of nodes. Each of these nodes does not have a parent node and is given a large initial cost. The large cost is used to determine which nodes have a valid connection to the search tree, and which do not. Next, beginning with the first shared node, each node is added to the tree using RRT*. Algorithm 10 describes the process for connecting the shared nodes to the tree.

Algorithm 10: AddSharedNodes(T, Q_{shared})

```
1 for $q_{node} \in Q_{shared}$ do
2     $q_{nearest} \leftarrow$NearestNeighbor($q_{node}$, $Q_{near}$, $T$)
3     $q_{min} \leftarrow$ChooseParent($q_{node}$, $Q_{near}$, $q_{nearest}$, $\Delta$q)
4     $T \leftarrow$Rewire($T$, $Q_{near}$, $q_{min}$, $q_{node}$)
5 end
```

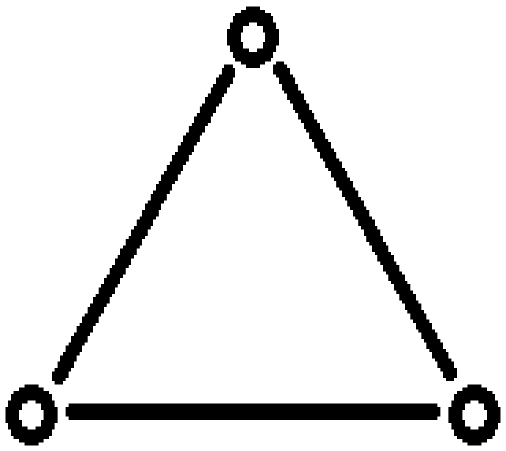
First, the node finds all the nodes within it's neighborhood, line 2 of Algorithm 10. Next, the node selects the best parent using the *ChooseParent*() function, line 3 of Algorithm 10. Once the node is connected to a parent node within the tree the *Rewire*() function is called, line 4 of Algorithm 10. As described above, the *Rewire*() function changes the parent of a nearby node if the cost to reach that node is lower than it's current cost. In this case, there may be several nodes in the neighborhood that are not yet connected to the search tree. *Rewire*() will connect these nodes by giving them a parent and a cost. In some cases a shared node will be rewired and be connected to the best possible parent. When all the shared nodes are added to the robot's search tree the total number of nodes is updated (line 13 of Algorithm 9) and the algorithm continues.

5.6 Results

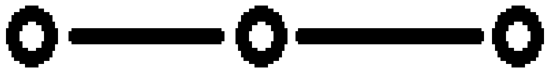
The results for The Multi-Robot Path Planning method described were gathered in the same manner as the static RRT and RRT* comparison results. In the experiments below there is a 3 robot configuration and a 4 robot configuration. There are also two different communication patterns. The first communication pattern is an all-to-all communication pattern where each robot communicates will all other robots. The second communication pattern is restricted to neighboring robots only. For example, in the three robot scenario one robot has two neighbors whereas two robots only have one neighbor. Each robot will build a search tree to satisfy a single-query path plan from it's own start and goal location. In the following scenarios the robots will add 100 nodes to their search tree before sharing those 100 nodes with the other robots.

5.6.1 Scenario 1

In the first multi-robot scenario there are three robots with an all-to-all communication pattern, shown in Fig. 5.1a The robots are allowed a maximum of 1500 nodes in the search tree. Each robot will reach the maximum of 1500 nodes at the same time. Since the robots have perfect communication with each other the nodes within each search tree will be identical. The path costs and parent nodes will be specific to the individual robot. Fig. 5.2 shows the search trees for each robot.

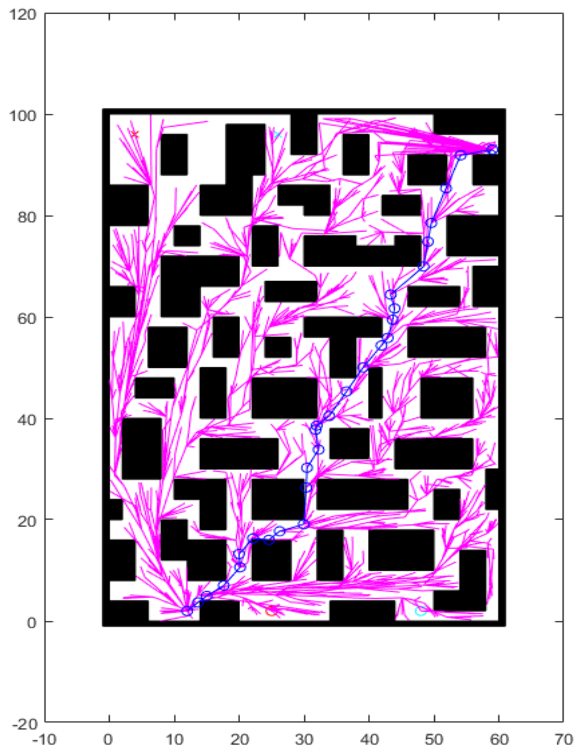


(a) All-to-all communication pattern

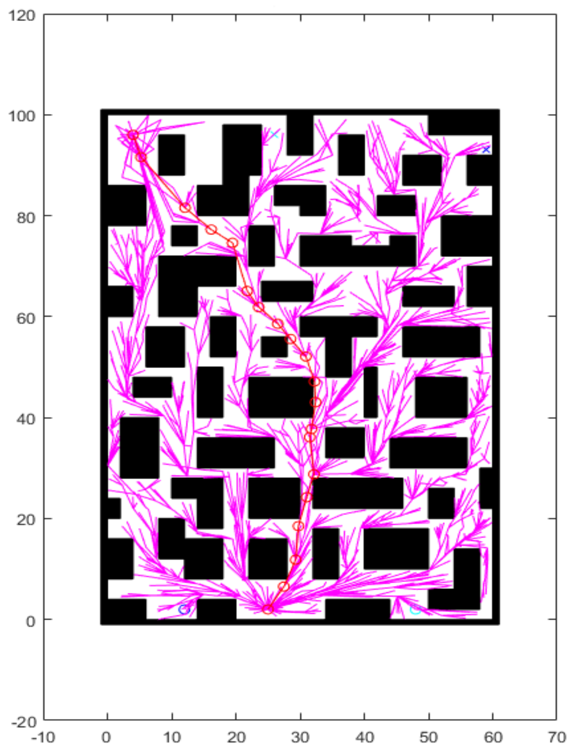


(b) Neighbor only communication pattern

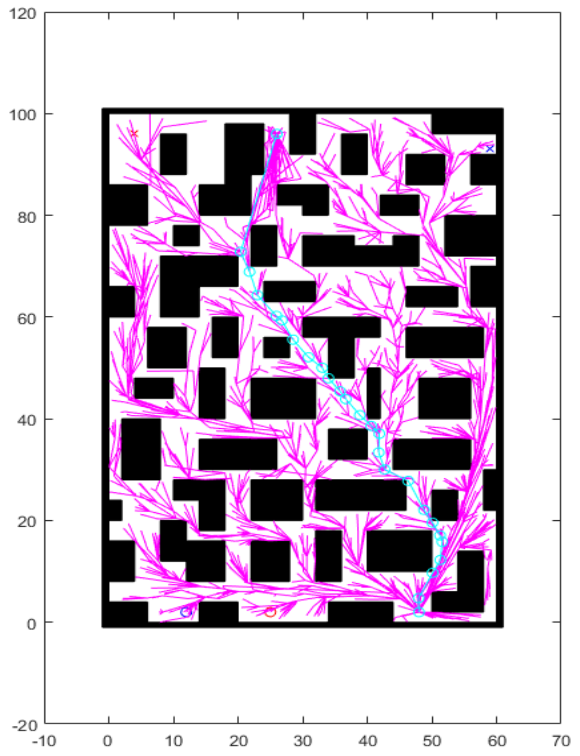
Figure 5.1: 3 Robot Communication Patterns.



(a) Search tree for robot 1



(b) Search tree for robot 2



(c) Search tree for robot 3

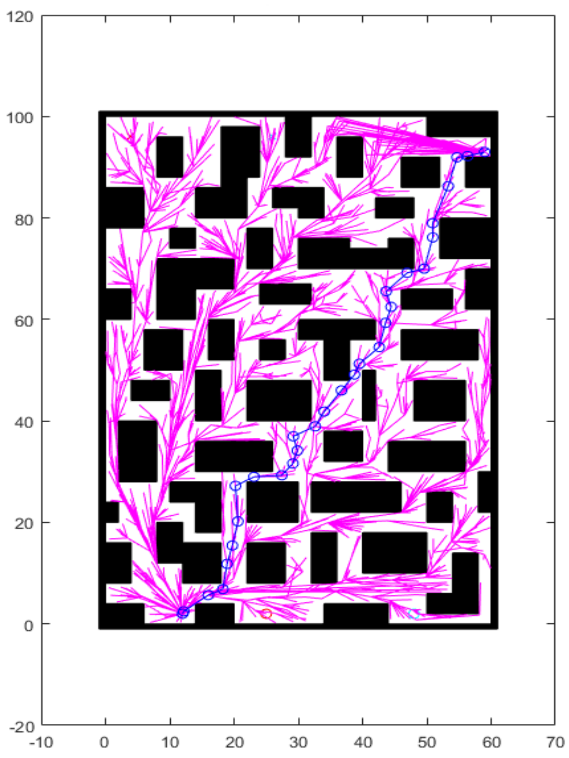
Figure 5.2: Mulit-Robot planning results from the first scenario.

5.6.2 Scenario 2

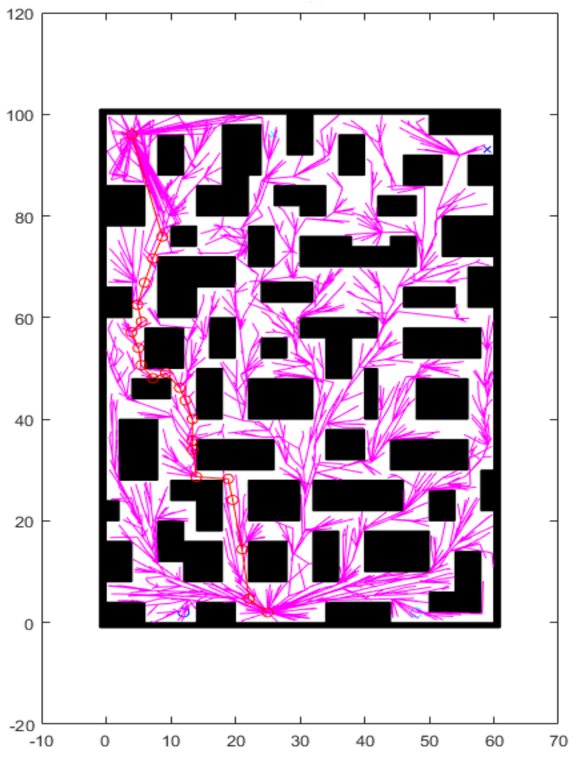
The second mulit-robot scenario has three robots that may only communicate with neighboring robots, shown in Fig. 5.1b. The center robot, or robot 2, in this scenario completes a 1500 node search tree faster than the other two robots because it is receiving nodes from two neighbors instead of just one. In order for the other two robots to reach the maximum of 1500 nodes they must spend extra time adding nodes using RRT* without receiving any new nodes from a neighbor. Fig. 5.3 shows the search tree for each robot. The optimal paths found by each robot are slightly worse than in Scenario 1. Table 5.1 shows the path costs for each three robot scenario. Robot 2 has the same start and goal locations as the static environment experiments in Chapter 3. The optimal path found in both scenario 1 and 2 have a similar cost to the static RRT* experiment using 2000 nodes.

Multi-Robot Path Costs			
	Robot 1	Robot 2	Robot 3
Scenario 1	112.847	105.185	107.233
Scenario 2	116.652	106.911	108.558

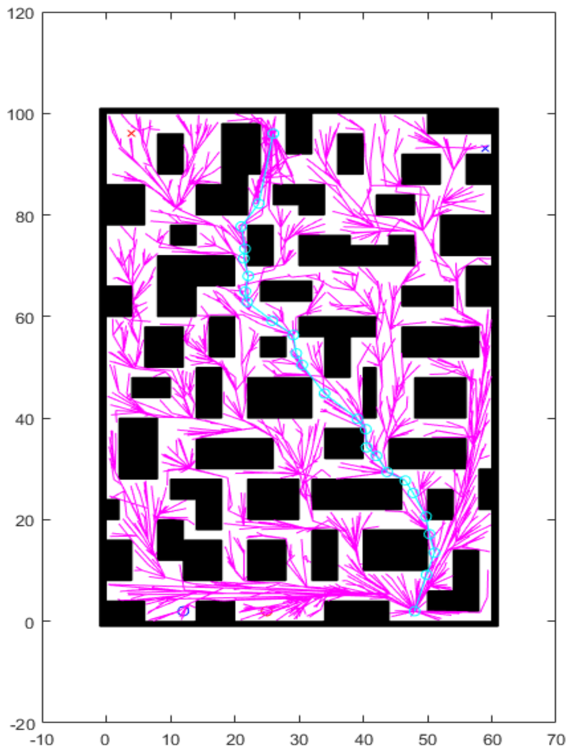
Table 5.1: Path Costs of 3 Robot Scenarios.



(a) Search tree for robot 1

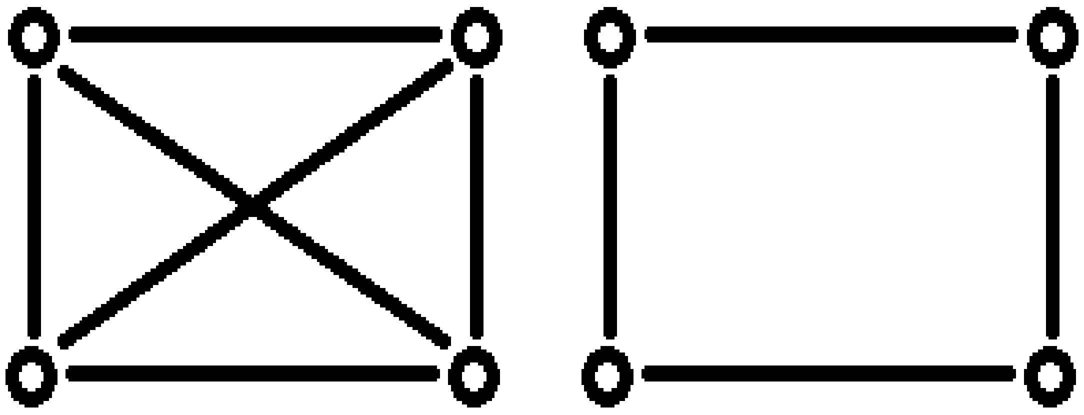


(b) Search tree for robot 2



(c) Search tree for robot 3

Figure 5.3: Mulit-Robot planning results from the second scenario.

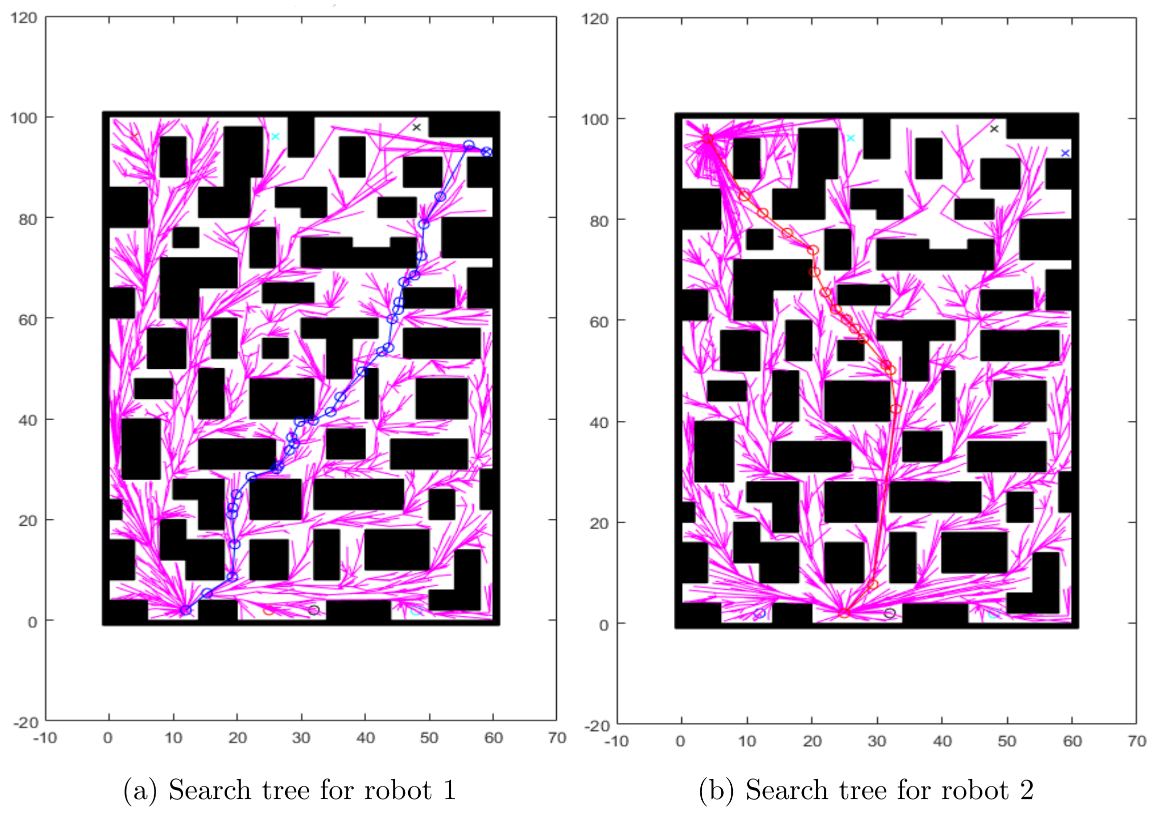


(a) All-to-all communication pattern (b) Neighbor only communication pattern

Figure 5.4: 4 Robot Communication Patterns.

5.6.3 Scenario 3

The third scenario has four robots with an all-to-all communication pattern, shown in Fig. 5.4a. In this scenario the robot will add 100 nodes to the tree, then add the 300 nodes received from the neighboring robots. The maximum number of nodes for this scenario was increased to 1600 in order to complete the node sharing process. If the maximum was still 1500, the last 100 shared nodes would not be added to the tree. Fig. 5.5 shows the search tree results.


(a) Search tree for robot 1 (b) Search tree for robot 2

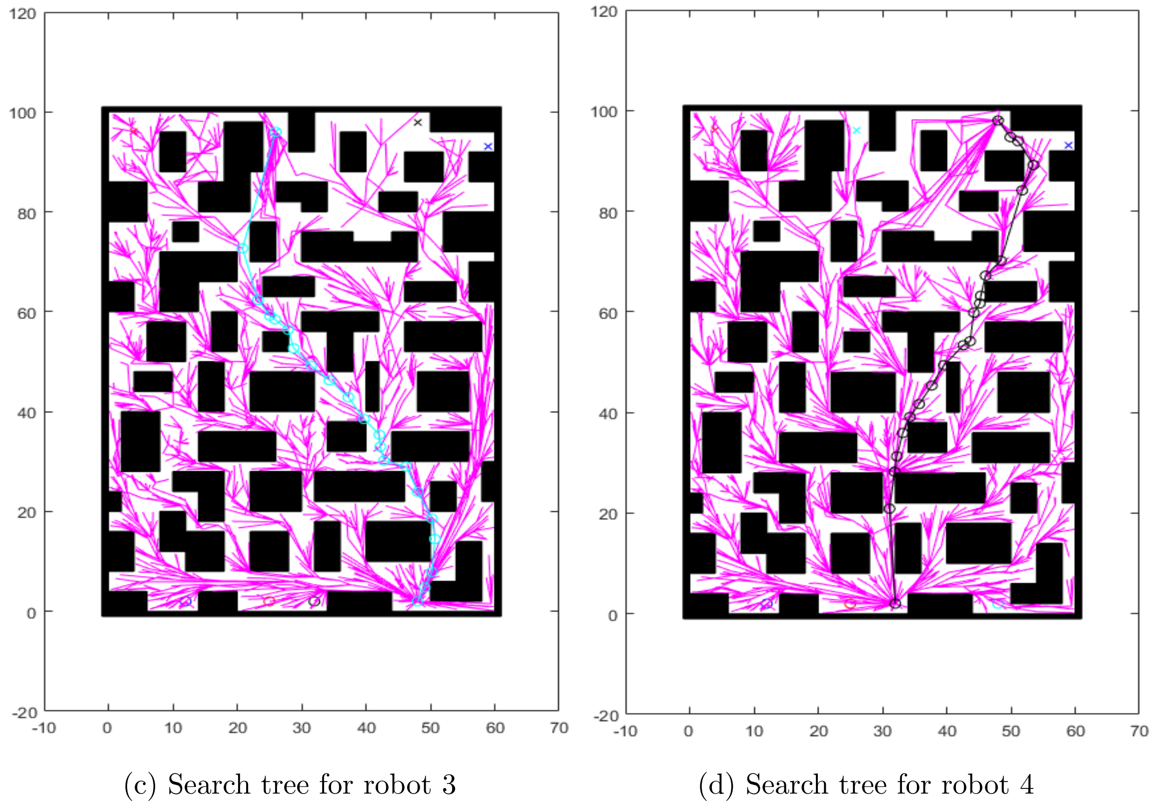

(c) Search tree for robot 3 (d) Search tree for robot 4

Figure 5.5: Mulit-Robot planning results from the third scenario.

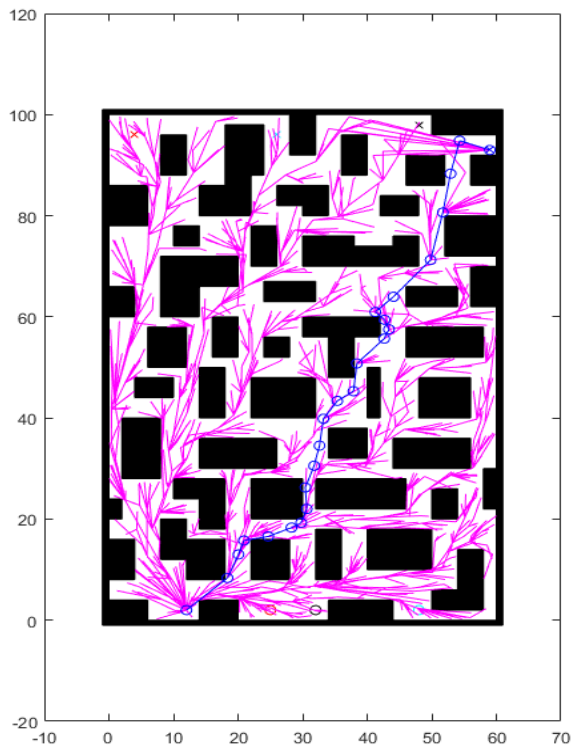
5.6.4 Scenario 4

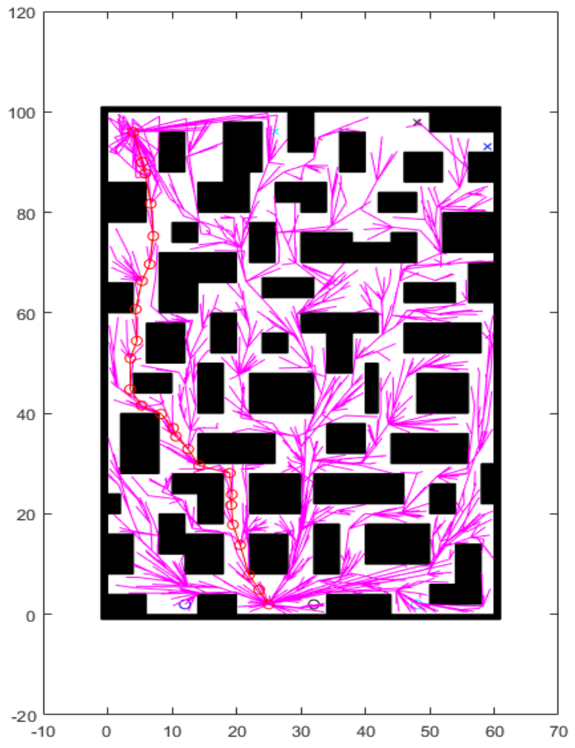
The fourth scenario has 4 robots that are only allowed to communicate with their immediate neighbors, shown in Fig. 5.4b. The communication scheme can be thought of as a square, where each robot occupies a corner. The edges of the square represent the communication links. If the robots numbers are counted clockwise around the square, robot 1 will communicate with robot 2 and robot 4. In this scenario each node sharing step will produce 300 nodes for the tree. The maximum number of nodes allowed is 1500 to complete the node sharing process. Fig. 5.6 shows the search tree results for this scenario.

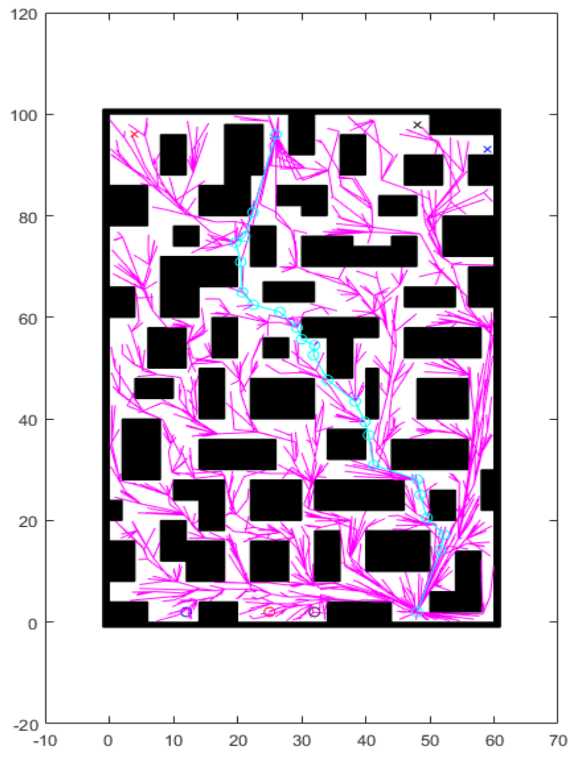
The overall path quality was better in the third scenario. This was expected since each tree contains 100 more nodes than the fourth scenario. Table 5.2 shows the path length results from scenarios 3 and 4. Although the trees in scenario 3 are larger the paths found in scenario 4 are still comparable. Similar to scenarios 1 and 2, the optimal path found for robot 2 is similar, or better than, the RRT* static environment experiments using a search tree of 2000 nodes. Experiments indicate sharing node information from multiple robots yields equivalent or better paths with fewer total nodes in an individual search tree.

Multi-Robot Path Costs				
	Robot 1	Robot 2	Robot 3	Robot 4
Scenario 3	114.254	104.860	106.800	104.864
Scenario 4	117.664	104.877	112.519	113.257

Table 5.2: Path Costs of 4 Robot Scenarios.


(a) Search tree for robot 1


(b) Search tree for robot 2


(c) Search tree for robot 3

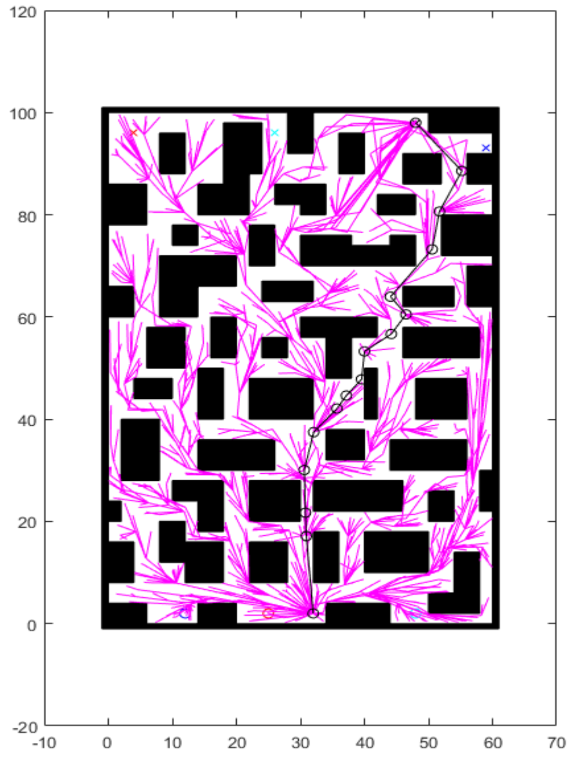

(d) Search tree for robot 4

Figure 5.6: Mulit-Robot planning results from the fourth scenario.

5.6.5 Multi-Robot Timing Results

In addition to comparing the quality of the paths found, the execution time to build the search tree was also investigated. In the scenarios above each individual robot will build a search tree containing the maximum number of nodes. The other option to build these search trees is to have each robot build it's own individual search tree. Table **??** shows the average execution time over 10 runs for each method of building a search tree for three robots. The time to build these search trees in scenario 1, where each robot has perfect communication with the other two robots performed the best. The improvement over building 3 individual search trees is about 10%. Even scenario 2 performed better than building 3 separate search trees. In scenario 2 robot 2 will complete a 1500 node search tree faster than it's neighbors because it is receiving nodes from both neighbors. When robot 2 reaches the maximum of 1500 nodes it will stop discovering new nodes and therefore stop sharing nodes with neighboring robots. The other two robots will need to spend extra time completing their search trees without receiving nodes from their neighbor. The extra time spent completing the search trees is why scenario 2 has a lower time performance than scenario 1.

3 Robot Execution Times	
3 Robot Scenario	Average Time (10 Simulations)
3 Individual RRT* search trees	1274.8 sec
Multi-robot Scenario 1	1144.3 sec
Multi-robot Scenario 2	1207.8 sec

Table 5.3: Execution Times of 3 robot scenarios

The timing results for the 4 robot scenarios are similar to the 3 robot scenarios. Scenarios 3 and 4 execution times are compared to building 4 individual search trees. Table **??** shows the average execution time over 10 runs for these scenarios. The timing for each of the multi-robot node sharing experiments is better than building a search tree individually for each robot. Scenario 3 does have search trees with 100 more nodes than scenario 4. There are 2 other differences between scenario 3 and scenario 4 that influence the execution time. First, each robot in scenario 3 will receive 100 nodes from each neighbor for a total of 300 at each node sharing step. In scenario 4 each robot receives a total of 200 nodes from the neighboring robots. The second difference is in the number of times nodes are shared between robots. In scenario 3 the robots will reach the maximum node limit after 4 iterations of lines 4-13 of Algorithm 9. Scenario 4 will reach the maximum node limit after 5 iterations. Since scenario 3 has a longer execution time, the additional time must be coming from the additional 100 nodes received by each robot during each node sharing step.

4 Robot Execution Times	
4 Robot Scenario	Average Time (10 Simulations)
4 Individual RRT* search trees	1663.8 sec
Multi-robot Scenario 3	1597.9 sec
Multi-robot Scenario 4	1312.3 sec

Table 5.4: Execution Times of 4 robot scenarios

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis presented three topics: First, a comparison the RRT and RRT* algorithms. Second, a method of dynamic replanning using RRT* to avoid unknown and unpredictable moving obstacles. Finally, a method of multi-robot path planning by sharing nodes between robots. The experiments in the first topic confirm the RRT algorithm will not converge to an asymptotically sub-optimal solution regardless of the number of nodes in the search tree. The RRT* static environment experiments also confirm the RRT* algorithm will converge to an asymptotically sub-optimal solution. There is a trade off when using RRT* in the execution time of the algorithm. As more nodes are added to the tree the quality of the path will improve, but the execution time increases exponentially.

The second topic presented is a method of dynamic replanning using RRT* to avoid unknown and unpredictable moving obstacles. The replanning method leverages the *Rewire*() function of RRT* in order to modify and expand the existing search tree to avoid a random obstacle when it is detected. This replanning method is a good first step toward more research in replanning using RRT*.

The last topic presented is a multi-robot path planning method using RRT*. The method presented is broken into three steps. First, each robot will add nodes to it's own search tree. Upon reaching a predetermined number of nodes, each robot will share the newly discovered nodes with each neighboring robot. Finally, each robot will add the nodes received from neighbors to the search tree using RRT*. The results presented indicate that the multi-robot path planning method produces equivalent quality paths in a shorter execution time than building multiple individual search trees using RRT*.

6.2 Future Work

There are several options for future research on the topic of dynamic replanning using RRT*. Future work will research a floating replan goal location to minimize the total remaining cost to the query goal. Minimizing the modifications to the original search tree is another area of improvement. This method has only been implemented in a 2-Dimensional configuration space. The algorithm must be expanded and modified to operate in higher dimension configuration spaces. The simulations up to this point have been with small numbers of moving obstacles, further research is needed to determine how the algorithm performs when there are many random moving obstacles.

Multi-robot path planning also requires additional research. Future work will research increasing the number of robots in the multi-robot team. Additional research is needed to maximize the number of shared nodes while minimizing execution time to add the shared nodes to the tree. Another area of future work is comparing multi-robot results to single robot results when a larger search tree is used by the single robot. Finally, more research is needed using sensors to detect the environment and dynamic communication. Sensors will limit how much of the environment the robot is able to explore and share with neighboring robots. Dynamic communication will represent a realistic communication scheme where robots will move in and out of communication range.

Bibliography

[1] L. E. Kavraki, P. Svestka, J. C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, Aug 1996.

[2] S. M. Lavalle, "Rapidly-exploring random trees: A new tool for path planning," Tech. Rep., 1998.

[3] S. Karaman and E. Frazzoli, "Optimal kinodynamic motion planning using incremental sampling-based methods," in *49th IEEE Conference on Decision and Control (CDC)*, Dec 2010, pp. 7681–7687.

[4] S. Karaman, M. R. Walter, A. Perez, E. Frazzoli, and S. Teller, "Anytime motion planning using the rrt*," in *2011 IEEE International Conference on Robotics and Automation*, May 2011, pp. 1478–1483.

[5] D. Ferguson and A. Stentz, "Anytime rrts," in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct 2006, pp. 5369–5375.

[6] A. Perez, R. Platt, G. Konidaris, L. Kaelbling, and T. Lozano-Perez, "Lqr-rrt*: Optimal sampling-based motion planning with automatically derived extension heuristics," in *2012 IEEE International Conference on Robotics and Automation*, May 2012, pp. 2537–2542.

[7] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "Informed rrt*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sept 2014, pp. 2997–3004.

[8] O. Salzman and D. Halperin, "Asymptotically near-optimal rrt for fast, high-quality, motion planning," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, May 2014, pp. 4680–4685.

[9] F. Islam, J. Nasir, U. Malik, Y. Ayaz, and O. Hasan, "Rrt*-smart: Rapid convergence implementation of rrt*; towards optimal solution," in *2012 IEEE International Conference on Mechatronics and Automation*, Aug 2012, pp. 1651–1656.

[10] A. H. Qureshi, K. F. Iqbal, S. M. Qamar, F. Islam, Y. Ayaz, and N. Muhammad, "Potential guided directional-rrt* for accelerated motion planning in cluttered environments," in *2013 IEEE International Conference on Mechatronics and Automation*, Aug 2013, pp. 519–524.

[11] J. Bruce and M. Veloso, "Real-time randomized path planning for robot navigation," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 3, 2002, pp. 2383–2388 vol.3.

[12] D. Ferguson, N. Kalra, and A. Stentz, "Replanning with rrts," in *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, May 2006, pp. 1243–1248.

[13] M. Zucker, J. Kuffner, and M. Branicky, "Multipartite rrts for rapid replanning in dynamic environments," in *Proceedings 2007 IEEE International Conference on Robotics and Automation*, April 2007, pp. 1603–1609.

[14] M. Otte and E. Frazzoli, "Rrtx: Asymptotically optimal single-query sampling-based motion planning with quick replanning," *The International Journal of Robotics Research*, vol. 35, no. 7, pp. 797–822, 2016.

[15] H. M. La, W. Sheng, and J. Chen, "Cooperative and active sensing in mobile sensor networks for scalar field mapping," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 45, no. 1, pp. 1–12, Jan 2015.

[16] H. M. La and W. Sheng, "Distributed sensor fusion for scalar field mapping using mobile sensor networks," *IEEE Transactions on Cybernetics*, vol. 43, no. 2, pp. 766–778, April 2013.

[17] H. La, T. Nguyen, T. D. Le, and M. Jafari, "Formation control and obstacle avoidance of multiple rectangular agents with limited communication ranges," *IEEE Transactions on Control of Network Systems*, vol. PP, no. 99, pp. 1–1, 2016.

[18] H. M. La, R. S. Lim, W. Sheng, and J. Chen, "Cooperative flocking and learning in multi-robot systems for predator avoidance," in *2013 IEEE International Conference on Cyber Technology in Automation, Control and Intelligent Systems*, May 2013, pp. 337–342.

[19] H. M. La and W. Sheng, "Multi-agent motion control in cluttered and noisy environments," *Journal of Communications*, vol. 8, no. 1, pp. 32–46, 2013.

[20] A. D. Dang, H. M. La, and J. Horn, "Distributed formation control for autonomous robots following desired shapes in noisy environment," in *2016 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, Sept 2016, pp. 285–290.

[21] H. M. La and W. Sheng, “Flocking control of multiple agents in noisy environments,” in *2010 IEEE International Conference on Robotics and Automation*, May 2010, pp. 4964–4969.

[22] ——, “Multi-agent motion control in cluttered and noisy environments,” *Elsevier Journal on Robotics and Autonomous Systems*, vol. 60, no. 7, pp. 996–1009, 2012.