IMPLEMENTATION OF A LOCAL PATH PLANNING ALGORITHM FOR UNMANNED AERIAL VEHICLES

By

NICHOLAS WILLIAM ROZELL

Bachelor of Science in Aerospace Engineering
Oklahoma State University
Stillwater, OK
2019

Bachelor of Science in Mechanical Engineering
Oklahoma State University
Stillwater, OK
2019

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 2021

IMPLEMENTATION OF A LOCAL
PATH PLANNING ALGORITHM FOR
UNMANNED AERIAL VEHICLES

Thesis Approved:

Dr. Jamey D. Jacob

Thesis Advisor

Dr. He Bai

Dr. Imraan A. Faruque

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Jamey Jacob for allowing me to pursue this research. I would also like to thank my committee Dr. He Bai and Dr. Imraan Faruque for guiding me in the proper direction of study. I would also like to thank the Excelsior Lab for supporting my research efforts. I would also like to thank Rakshit Allamraju for helping me with understanding and setup of the networking of the aircrafts and the ROS environment. I would also like to thank Levi Ross for helping me with understanding the inner operations of the Pixhawk and Ardupilot code as well as his MatLab code for gathering data sets from the Pixhawk. I would also like to thank Dr. Ben Loh for his insight and piloting of the Eagle multi-rotor aircraft. I would also like to thank Allan Burba and Kyle Hickman for piloting both the Eagle and Nano-Talon during some of the flight tests. I would also like to thank my friends, family, and especially my parents for their understanding and support in these trying times. This work is supported in part by the National Science Foundation under Grant No. 1539070, *Collaboration Leading Operational UAS Development for Meteorology and Atmospheric Physics (CLOUD-MAP)* and the NASA under the University Leadership Initiative under Grant Number 80NSSC20M0162 (WINDMAP). Additional support provided by the OSU Unmanned Systems Research Institute and the Oklahoma State University Tier 1 Initiative (Drones for Good).

Acknowledgments reflect the views of the author and are not endorsed by committee members or Oklahoma State University.

Name: NICHOLAS WILLIAM ROZELL

Date of Degree: JULY, 2021

Title of Study: IMPLEMENTATION OF A LOCAL PATH PLANNING ALGORITHM FOR UNMANNED AERIAL VEHICLES

Major Field: MECHANICAL AND AEROSPACE ENGINEERING

Abstract:

As the presence of Unmanned Aircraft Systems (UASs) become more prominent today and in the future. They are used in a variety of ways to solve solutions for a variety of tasks. UASs that are battery-powered typically have a flight time of no more than 30 minutes. Some tasks make take the drone beyond visual line of sight (BVLOS). The approach taken within this paper is allocating a secondary flight computer onboard the UAS to calculate paths while the primary computer controls the aircraft and follows the path being generated. With a proper map of the environment and use of a path planning algorithm the safety of the aircraft can be increased in missions that are BVLOS. This thesis will cover the concepts of path planning algorithms and the development of a modified version of a popular path planning algorithm. Show simulations of comparison with other variations of path planning algorithms and software in the loop (SITL) simulations on a fixed-wing aircraft. It will also show this algorithm's results when implemented in flight tests onboard a fixed-wing and multi-rotor UAS.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

CHAPTER I

INTRODUCTION

1.1 Motivation

As UAVs continually and increasingly play a role in ways that people have solved problems of inspection, research, and search and rescue. These missions are operated manually by one certified to fly UAVs known as a remote pilot. Adding a layer of autonomy can reduce the operational workload and increase the capability of the UAV during its mission, like searching an area in a Beyond Visual Line of Sight (BVLOS) environment. This can be done by making the UAV more intelligent by providing it the tools needed to quickly navigate these environments and planning paths around zones that have obstacles in them. These being defined in the provided map. This all being implemented in a low-cost system. That uses a planning algorithm that has a low computation time and is utilized on a microcomputer. That is also capable of being integrated on a UAV.

1.2 Background

1.2.1 Graph Theory

Gross et al.[15] defines a graph as a mathematical structure consisting of two sets V and E, and is denoted by the $G = (V, E)$. The elements of the set V are defined as vertices or nodes. The elements of the set E are defined as edges or lines. Vertices and edges can be designated their own attributes i.e. color, cost, etc. And a graph can take place within a plane or in 3D space. Edges are used to connect vertices or to themselves, known as a

self-loop. A group of vertices that are connected is known as a neighborhood. Doubly, a vertex connected to another is known as a neighbor or an adjacent vertex. A degree is a definition for a vertex counting the number of incident edges of a vertex. And lastly, a path is defined as a walk with no repeated edges and no repeating vertices. And a walk is defined as an alternating sequence of vertices and edges that represents a continuous traversal from one vertex to another.

Gross et al.[15] discuss some graph types like a simple graph, digraph, multigraph, weighted graph, etc. Here the digraph and weighted graph types of concern in this thesis. The digraph, or directed graph, is described as a graph that's edges are directed. Directed edges indicate a direction of a connection between two vertices. With a directed edge, one endpoint is labeled as the tail and the other is labeled as the head. The weighted graph simply applies weight or costs to vertices and edges within the graph. These costs can be defined as various amounts of things, like, the distance between vertices, or time to reach a vertex, etc. There is a more extensive catalog of definitions and terminologies that describe a graph. There is also a diverse family of graphs that have different descriptions and uses. However, those graphs and definitions are beyond the scope of this paper. A random but complex graph can be seen in figure 1.

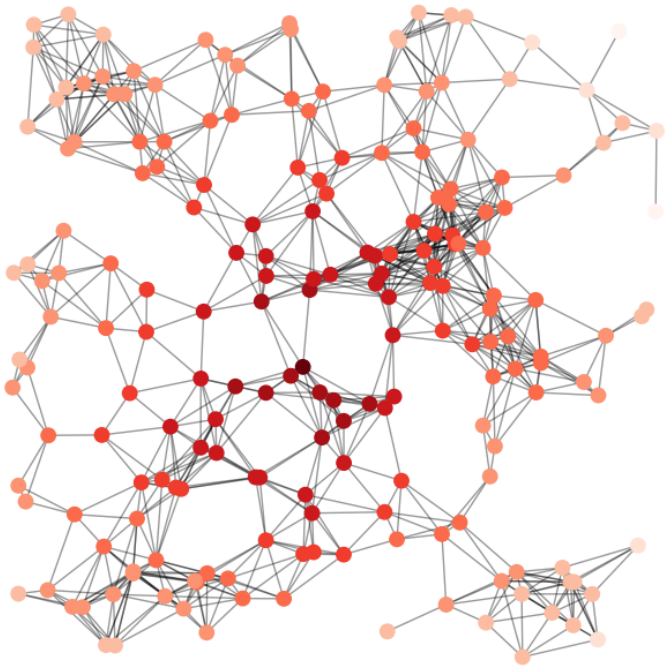

Figure 1: A Random Geometric Graph[9]

Gross et al.[15] talks about operations that can be applied to graphs and other ways to represent a graph. These operations include the adding and deleting of vertices as well as edges, the union of graphs, or a vertex and a graph or, even taking the Cartesian product of two graphs. A way of representing a graph is by using an adjacency matrix, adjacency list, or indecency matrix. The adjacency matrix is the focused method here. This a powerful tool as it brings linear algebra to graph theory. An adjacency matrix is a matrix whose rows and columns are indexed by identical orderings of a graph's vertices. This allows for a quick computational look-up of neighboring vertices. There is also an incidence matrix that similarly uses the vertices and edges as an adjacency matrix. Both matrix representations can be extended to represent digraphs.

Gross et al.[15] also discusses the definitions of what is known as a Tree. Trees are known to be graphs with no cycles. A Cycle is a closed path with at least one edge, and a closed path is said to be a sequence of nonrepeating vertices whose initial and final vertices are

the same. Now there are a variety of tree classifications that are discussed, these include; an ordered tree, rooted tree, binary tree, polytrees, etc. However, the type of tree that is of concern with this paper is the rooted tree. A rooted tree is a tree whose edges are all directed outward from the root, otherwise known as leafward. With a rooted tree, vertices are designated special definitions. A root is a vertex where the tree starts. A parent is a vertex that precedes another vertex this is also defined as a predecessor. Likewise, a child is a vertex that succeeds another vertex and is also defined as a successor. A single vertex can have multiple children, where this group is known as siblings. However, a single vertex can only have one parent. Another vertex definition for rooted trees is what is called a leaf. A leaf is a vertex that has no children.

There are many problems that a graph or tree is used. A few of these problems are known as the Shortest-Path Problem. This kind of problem can be defined as finding the shortest path when traversing a graph from one vertex to another. The minimum weight spanning tree problem is another of these problems. It describes a way of connecting vertices in a graph using the minimal cost possible. And finally, the traveling salesman problem, most likely the most popular of the problems discussed. The traveling salesman problem is defined as finding the minimum and best cost between cities in a map for the salesman to travel efficiently. These problems can be solved using planning algorithms.

1.2.2 Planning Algorithms

Lavelle[23] discusses the meaning of the term planning. Planning in robotics was originally concerned with navigating from one point to another. This can be expanded upon with the piano's mover problem. Which is described as how to move a piano from one room to another in a house without hitting anything. However, the fields of artificial intelligence and control theory have also encompassed different aspects of the topic of planning and have expanded their scopes to share a common ground. Where, artificial intelligence, planning is meant as a search of a sequence of operations or actions that transform an initial state to a

goal state. This is also extended to decision-theoretic ideas like Markov decision processes. For control theory, there is an interest in designing algorithms that find feasible open-loop trajectories for nonlinear systems.

Lavelle emphasizes the focus on algorithm issues relating to planning. With robotics, the focus is designing algorithms that generate ideal motions by processing complicated geometric models. Artificial intelligence focuses on decision-theoretic models to compute appropriate actions. And, for control theory, it is on algorithms that compute feasible trajectories for systems, with some coverage on feedback and optimality.

Souissi et al.[36] categorizes planning algorithms into four different levels. The first level discusses the planning problems and splits them into three different types. The first type is holonomic, which considers that all degrees of freedom of a robot is controllable, allowing it to move any which way regardless of constraints. Nonholonomic problems are the second type, which includes a robot's constraints. Some strategies of nonholonomic platforms are called Dubins curves or Reed Sheppard curves. These create smooth paths for robots to follow. And, lastly, Kinodynamic problems designate two types of constraints, kinematic and dynamic. The second level splits algorithms into two different execution strategies. One where the environment is modeled and a path is found and the other where the path is found without modeling the environment. Level three splits algorithms into on-line and off-line. The difference between the two is that on-line algorithms do without an environment modeling step, whereas the off-line includes this step. However, this is not always the case, as the environment modeling can be done in a pre-processing step for on-line algorithms. And, finally, the fourth level, dived planning into deterministic and probabilistic algorithms. Where deterministic methods produce the same results given the same initial conditions. And, probabilistic methods suggest that as computing time reaches infinity, the probability of finding an exact solution reaches one.

Cormen et al.[6] describes an algorithm as a well-defined computational procedure that takes some value, or set of values, as an input and produces some value or values as an

output. Therefore, an algorithm is a sequence of computational steps that transform the input into an output. Graph algorithms are one of the sets of algorithms discussed.

Grid-Based Algorithms

Grid-based algorithms use methods previously mentioned to solve the problems mentioned as well. A popular way of representing the graphs for these type of algorithms are through grids. Figure 2 shows the grid used for these algorithms. Greyed tiles are used to represent obstacles. One of the most basic search algorithms that use this type of graph is called breadth-first search (BFS). Lavelle[23] and Cromen et al.[6] both discuss this method. BFS is a deterministic method that utilizes a First-In-First-Out (FIFO) queue, sorting function. This allows for a uniform search across the graph. This laid out a ground-level algorithm that others have used to create more robust search algorithms.

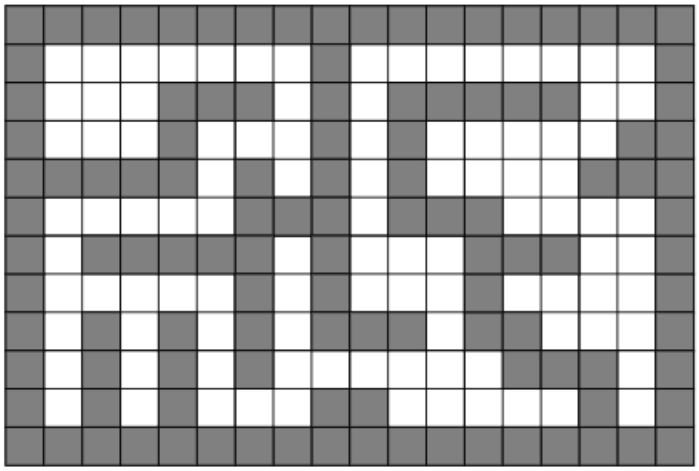

Figure 2: A graph designed as a grid.

In the 1950s, a computer scientist by the name of Edsger W. Dijkstra created a path searching algorithm, known as Dijkstra's Algorithm. It utilizes a similar graph as BFS. However, with Dijkstra's algorithm, a weighted graph and a priority queue are used instead. These are coupled together to provide a solution of a cost minimal path[10]. Lavelle discusses the term cost-to-come to designate this function. What it does is sum up the cost of all

possible paths and is done to produce a path that has the least cumulative cost[23].

A* pronounced as "a star" created by Hart et al.[16], incorporates a heuristic estimate to Dijkstra's algorithm. The approach was used to reduce the number of vertices searched through the graph. This was done by making vertices that are closer to the goal have a higher priority. Lavelle calls this function the cost-to-goal. He states that the A* search algorithm works the same as Dijkstra's algorithm, but the priority queue couples the cost-to-come and cost-to-goal functions to develop a solution[23].

D* pronounced as "d star" or dynamic A* algorithm, designed by Anthony Stentz[37], builds from A*. D* is different as it is dynamic meaning it is capable of modifying cost parameters during the problem-solving process. This allows it to replan a path whenever a change in the map happens. Others have utilized this algorithm and improved upon it as D* utilizes a large amount of memory. Some of these methods include D*-Lite[20], FieldD*[3], and others are mentioned in Souissi et al. survey[36].

Grid-based algorithms do well in making complete and deterministic paths in graphs that have a path between a start and goal location. They are also capable of hierarchical planning, which refines the search areas over time to create a more optimal path[36]. However, they do come with disadvantages. A major disadvantage is the execution time[36]. The algorithms are only ensured up to the grid resolution and grid points grow exponentially with the dimensionality of the state space. This increases the memory usage of the algorithms as well as the running time for them[19]. And it also difficult to find a heuristic that is both efficient to evaluate and provide good search guidance[23].

Sample-Based Algorithms

Sample-based algorithms are another form of search used to solve the planning problem. These don't use the grid-like structure that the previous methods used. Instead, samples are randomly chosen in the graph to expand to. This allows the searching to be independent of the geometric models like the grid-based algorithms. With sample-based algorithms, more

definitions come with defining aspects of the algorithms that are defined under its umbrella. Algorithm completeness is described as given any input a solution exists within a finite time. For sample-based completeness is not achieved, so weaker notions are tolerated. Denseness is important meaning that samples come arbitrarily close to any configuration as the number of iterations reaches infinity. A deterministic approach that samples densely is defined as resolution complete. This means that if a solution exists it will find one in finite time, however, if a solution does not exist the algorithm will run forever. Another term used is probabilistic completeness, which means that with enough vertices, the probability that an existing solution reaches one. Then there is the rate of convergence albeit, this is difficult to establish[23].

The rapidly-exploring random tree (RRT) algorithm is a popular method of search in the sample-based algorithms set. This method was developed by Lavelle et al.[22]. RRT starts by placing a vertex at the chosen starting location. It then utilizes an operation that goes by the name extend. This can be visualized in figure 3. What this does is, selects a random vertex in the graph and runs a nearest neighbors search function to find the nearest vertex. There is a variety of nearest neighbor search methods found that utilize different techniques. Once the nearest neighbor is found, the randomly selected vertex is then saturated down to a defined distance away from its nearest neighbor. This is known as steering and uses a function called steer. A subroutine is then used to check if the now new saturated vertex conflicts with an obstacle. This checks to see if the new vertex is inside an obstacle or the connecting edge with its nearest neighbor intersects an obstacle. If it is obstacle free then the vertex is added to the graph. The edge that connects the new vertex and its nearest neighbor is, added as well. The pseudocode can be seen in algorithm 1. A figure of a result can be viewed in figure 4.

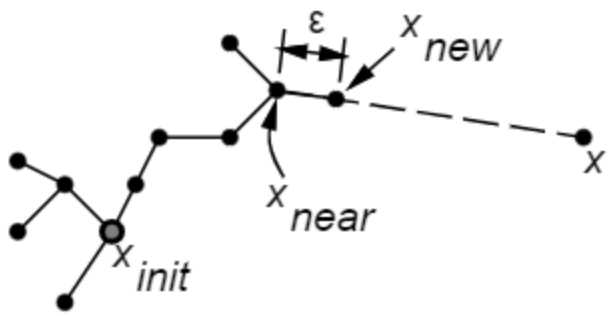

Figure 3: Extend Operation[22]

Algorithm 1: RRT

$V \leftarrow \{x_{init}\}$; $E \leftarrow \emptyset$;
for $i = 1, ..., n$ **do**
    $x_{rand} \leftarrow$ **SampleFree**;
    $x_{nearest} \leftarrow$ **Nearest**$(G = (V, E), x_{rand})$;
    $x_{new} \leftarrow$ **Steer**$(x_{nearest}, x_{rand})$;
    if ObstacleFree$(x_{nearest}, x_{new})$ **then**
        $V \leftarrow V \cup \{x_{new}\}$; $E \leftarrow E \cup \{(x_{nearest}, x_{new})\}$;
return $G = (V, E)$;

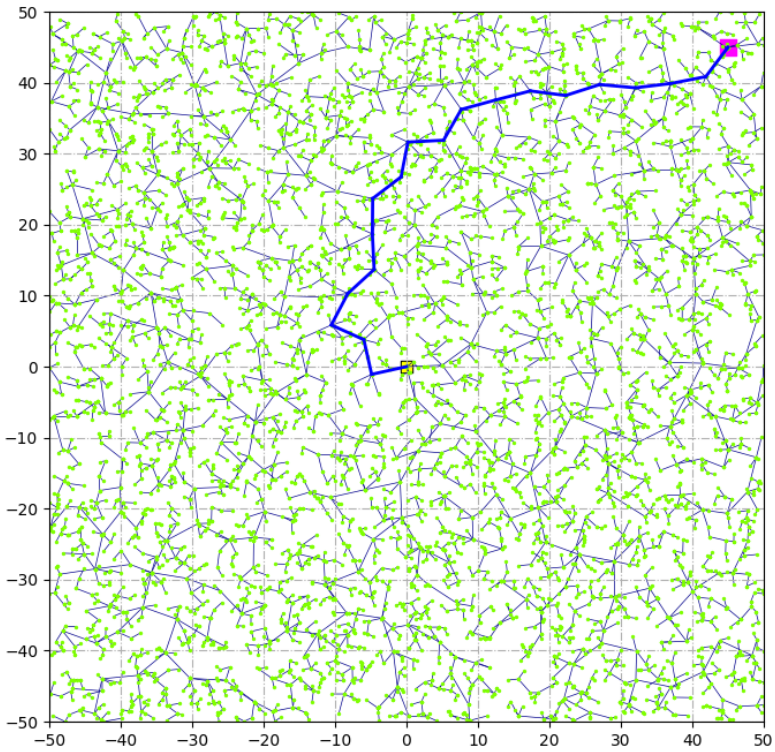

Figure 4: RRT search after 5000 nodes

RRT* pronounced "RRT star" is a more optimized version of RRT. Karaman et al.[19] discusses the changes made to RRT to develop RRT*. It utilizes the same routines used in RRT. However, a certain nearest neighbor method is used. And, two subroutines are added to the algorithm. The nearest neighbors are found using a distance r defined in equation 1.2.1.

$$r(\text{card}(V)) = \min\{\gamma_{RRT^*}(\log(\text{card}(V))/\text{card}(V))^{1/d}, \eta\} \tag{1.2.1}$$

The value r shrinks as the number of vertices increases within the graph. $card(V)$ is essentially shorthand for the cardinality of V (vertices). Cardinality is defined as the number of elements in a set. The variable γ_{RRT*} is defined in equation 1.2.2.

$$\gamma_{RRT^*} = 2(1 + 1/d)^{1/d}(\mu(\mathcal{X}_{free})/\zeta_d)^{1/d} \tag{1.2.2}$$

d is the dimension of the graph $\mathcal{X}$. $\mu(\mathcal{X}_{free})$ is the Lebesgue measure also referred to as the area or volume depending on the dimension. ζ_d is the volume of the unit ball in the d-dimensional Euclidean space, and η is the local distance that is defined by the user to saturate distances between new vertices and the tree's nearest vertex. The two subroutines that are added are the find parent function and the rewire neighbor function. The find parent routine uses the r variable to find nearby vertices to connect to and chooses the one with the least cost. The rewire neighbors routine also uses r to find nearby vertices that have a potential better cost connection. The authors suggest that RRT* is asymptotically optimal. This means that there is a high likelihood of finding the optimal path in the graph. Algorithm 2 shows the pseudocode for the RRT* algorithm, and figure 5 shows the result after an RRT* search has been completed.

Many other variants have come from RRT over the years. These include RRT-Connect an algorithm that utilizes two search trees, one at the start location and one at the goal location[22, 21]. RRT$^{\#}$ and algorithm that builds from RRT* and uses a routine to incorporate replanning to ensure that an optimal path is calculated as RRT* does not guarantee an optimal path[1]. RRT^{X} builds off of both RRT* and RRT$^{\#}$. The algorithm incorporates dynamic obstacles by modifying many steps of RRT* while keeping the graph consistent. Meaning a solution will always be present. It also utilizes two sets of neighbors in the tree. These sets are split up into categories that keep edges short and initial neighbors are not removed to keep the algorithm asymptotically optimal and problematically complete. A subroutine to keep the consistency of the graph is used to rewire neighbors that are considered inconsistent. When an obstacle appears inside the graph, a cost-to-goal increase is applied leafward through the tree. The costs increases start at edges that are made invalid due to the appearance of obstacles[29].

Algorithm 2: RRT*

$V \leftarrow \{x_{init}\}$; $E \leftarrow \emptyset$;
for $i = 1, ..., n$ **do**
  $x_{rand} \leftarrow$ **SampleFree**;
  $x_{nearest} \leftarrow$ **Nearest**$(G = (V, E), x_{rand})$;
  $x_{new} \leftarrow$ **Steer**$(x_{nearest}, x_{rand})$;
  if ObstacleFree$(x_{nearest}, x_{new})$ **then**
    $X_{near} \leftarrow$ **Near**$(G = (V, E), x_{new}, \min\{\gamma_{RRT*}(\log(\text{card}(V))/\text{card}(V))^{1/d}, \eta\}$;
    $V \leftarrow V \cup \{x_{new}\}$;
    $x_{min} \leftarrow x_{nearest}$; $c_{min} \leftarrow Cost(x_{nearest}) + c($**Line**$(x_{nearest}, x_{new}))$;
    for $x_{near} \in X_{near}$ **do**
      if CollisionFree$(x_{near}, x_{new}) \wedge$ **Cost**$(x_{near}) + c($**Line**$(x_{near}, x_{new})) < c_{min}$ **then**
        $x_{min} \leftarrow x_{near}$; $c_{min} \leftarrow$ **Cost**$(x_{near}) + c($**Line**$(x_{near}, x_{new}))$;
    $E \leftarrow E \cup \{(x_{nearest}, x_{new})\}$;
    for $x_{near} \in X_{near}$ **do**
      if CollisionFree$(x_{new}, x_{near}) \wedge$ **Cost**$(x_{new} + c($**Line**$(x_{new}, x_{near})) <$ **Cost**(x_{near}) **then**
        $x_{parent} \leftarrow$ **Parent**(x_{near});
      $E \leftarrow (E \setminus \{(x_{parent}, x_{near})\}) \cup \{(x_{new}, x_{near})\}$;
return $G = (V, E)$;

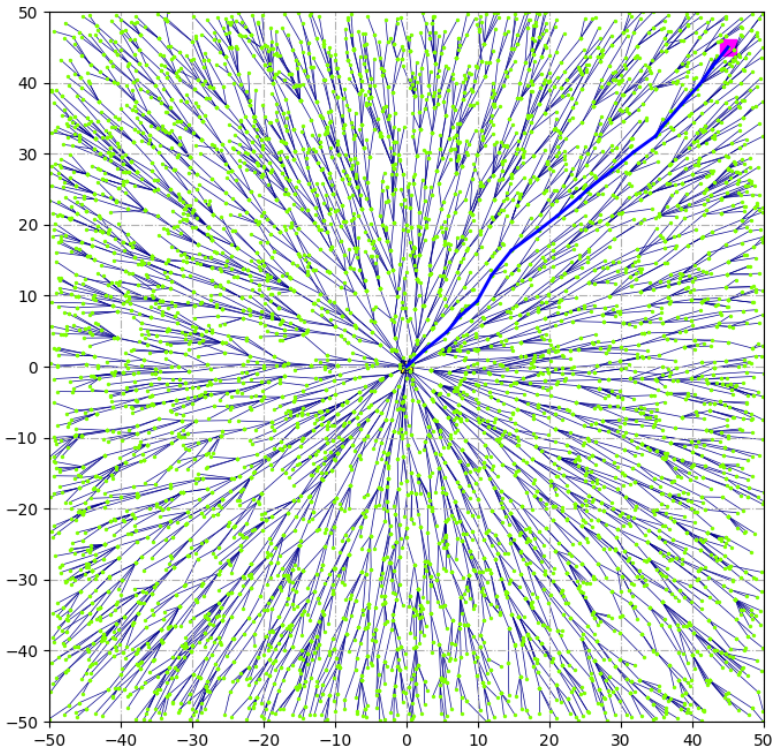

Figure 5: RRT* search after 5000 nodes

Sample-based algorithms have the advantage of being well suited for use in robotic motion planning applications[22]. They can search every corner of the configuration space[23]. They also have the capability of being fast and producing optimal solutions[19, 36]. With RRT using metaheuristics is the efficiency of the obtained solution. Is also one of the fastest methods of path planning and in real-time environments is considered unbeatable[36]. Though these do come with disadvantages. One is the difficulty of reproducing the same solution since they are stochastic. There consists of a failure to stop execution and report when no possible solution exists[36]. Depending on the graph size and distance between vertices searching globally can take a while and depending on the number of obstacles in the graph. This can be very computational intensive[23].

1.3 Objectives

The main goal of this thesis is to implement a path planning algorithm that can be used on-board an Unmanned Aerial Vehicle (UAV) for navigation. The objectives of this thesis are as follows. To develop and modify an optimal path planning algorithm for use on UAVs. That has a reduced computation time and can create close to an optimal path. While allowing for the capability of continuous planning paths to multiple waypoints and avoiding obstacles that are presented and defined in the map. Be able to plan paths online. That is, given a predefined map, the path can be generated while during a UAV's flight. And lastly, to implement and test the algorithm on both a fixed-wing and multi-rotor platform. By using a microcomputer that can be easily integrated on the chosen UAV. And to test the capabilities of the algorithm on that microcomputer and platform.

1.4 Outline

The outline of this thesis are as follows. Chapter I discusses the objectives and motivations of this thesis and the background of underlying theory and popular path planning algorithms. Chapter II addresses the literature review of papers that display a change in the algorithms that are catered to UAVs by others in academia. Chapter III details the approach that this thesis takes as well as expected outcomes of what the changes will do and considers components of software and hardware alike, for implementation into UAVs. Chapter IV displays environment layouts and setups taken into consideration for simulations and flight tests. Chapter V presents the results of simulations and flight tests discussed in Chapter IV. And lastly, Chapter VI goes into the outcomes and examines the potentials of future work.

CHAPTER II

LITERATURE REVIEW

Many others have pursued the use of path planning be it grid-based or sample-based for UAV navigation. Micheal Otte the creator of the RRT^X algorithm used simulations to show its use with a Dubins model same with Lavelle and the basic RRT algorithm[22]. The robot moved at a defined speed and had a predefined minimum turning radius. Distance calculations between two points used the heading and coordinate of the robot. A constant variable determined the cost trade-off difference between the two. A Dubins trajectory was calculated using the two points to calculate the geodesic distance. Depending on the constant cost variable it would guarantee the Dubins trajectory is larger than the geodesic distance so that it would be an admissible heuristic[29].

Yang et al. created a variation called Spline-Based RRT or SRRT, slightly differing from RRT. An extra step in the routine was added that checks if a new node satisfies the feasibility condition arising from differential constraints. The tree is only allowed to grow in a region defined by an upper and lower bound [41]. Figure 6 shows how the paths are generated. Lee and Shim performed simulations of their algorithm. Their first focus was on the algorithm generating obstacle-free paths and second the validity of the dynamic information that is gathered by the algorithm by comparing it to a nonlinear 6-DOF simulation.

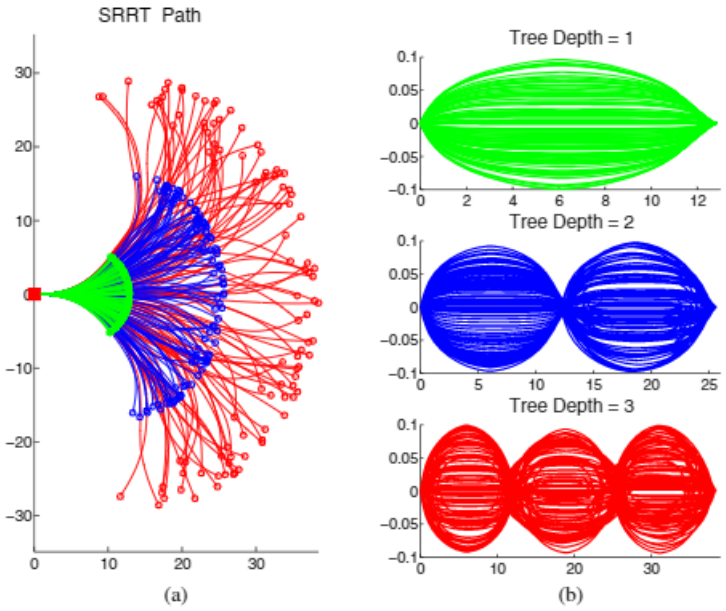

Figure 6: (a) Shows the smooth paths. (b) Shows the curvature of the paths [41].

Multiple experiments by Yang et al were conducted using the SRRT algorithm on a ground mobile vehicle in a cluttered environment. The task of the experiment was to navigate to the goal point while avoiding obstacles in the environment. The vehicle would have no prior information about the obstacles, so a scanning rangefinder was used to sense the obstacles. The authors suggest that SRRT makes several improvements on solutions for planning problems with differential constraints.

Zammit et al.[42] provide a comparison between A* and RRT algorithms when applied to a 3D UAV scenario. They use different 3D scenarios that are confined by a box with a combination of obstacles and their considered performance measures are path length and generation time. The authors suggest three categories for path planning algorithms these are: Graph-Based and Sampled-Based which were discussed earlier in the chapter, and a third category called Interpolation techniques. Interpolation techniques are considered a post path planning stage as it ultimately smooths the path generated by the algorithm. The authors discuss the use of A*, RRT, and Multiple Rapidly-exploring trees (MRRT) with path smoothing, for testing comparisons. MRRT is similar to how RRT-Connect works, but

instead of planting trees at the start and goal. MRRT instead plants trees in predefined or random locations within the graph that are free of obstacles. The experiment took place inside of a uniform cube that's center laid on the origin. After the results, the authors remark that the smoothed path length is shorter for A* than RRT, noting the optimality of the former, where there is a lack in the latter. However, they do say that because RRT is constrained by a step size the A* has a predefined set of points, taking away from the comparison. The RRT has more length reduction in this case as well since A* constructs shorter paths anyways. Another comparison between A* and RRT is settled by not constraining step size. This prolonged the path generation of RRT by quite a bit. And the last few comparisons add MRRT to the mix, showing that MRRT has a longer path generation time compared to the unconstrained RRT. But lower compared to RRT and A* exhibits that is more computationally efficient than the two. However, they say that it is unfair to compare the two since MRRT is unconstrained while the prior algorithms are. The authors conclude that A* can create optimal solutions at low computation where RRT and its variants can perform better in terms of path construction time and length. A final note by the authors, suggests that with the inclusion of a UAV's kinematic and dynamic model, sensor constraints and uncertainties. And with the addition of static and dynamic obstacles can confirm the applicability of A* and RRT in real-life dynamic environments.

Goerzen et al.[14], provide a detailed survey on the use of path planning on UAV guidance, several complexities are discussed. These include the importance of differential constraints, atmospheric turbulence, the uncertainty of the vehicle's state, and the lack of environmental awareness due to sensor capabilities. With this overall knowledge gap, it can be difficult to design a guidance system or choose an algorithm for a vehicle with many different degrees of freedom. The overarching problem that is discussed is the topic of vehicle motion planning. Motion planning is a small derivation from path planning, as path planning only focuses on location. Whereas, motion planning takes into account the orientation of the vehicle to create a solution. Since a UAV operates in three-dimensional space while also having

differential constraints gives it five to twelve degrees of freedom. This is a problem as stated by the authors that no algorithm can create a solution for it yet. However, they gather a variety of algorithms to compare them to establish important algorithm characteristics. These include completeness, optimality, computational complexity, and more. A comparison of many different planning algorithms is made and provided via a table giving the name of the algorithms, completeness, optimality, soundness, and time complexity. The tables can be found in the Appendix.

Goerzen et al.[14], also discusses the use of sample-based algorithms to create trajectory solutions with differential constraints. Most of today's UAV planning problems have to consider dynamics-constrained problems. UAV behavior is not sufficiently well approximated by their kinematics so the equations of motion are necessary for guaranteeing the soundness of the planner. Since approximating the dynamics solely on the kinematic model of the UAV with constraints leads to a conservative model. This class of planning problems is also noted to be monumentally more difficult to solve due to the dependency between time and the state-space by differential constraints. An exact solution is generally not possible even when simply connecting nodes within the graph, exact solutions are solvable in exponential time and polynomial space. If a vehicle is to navigate among obstacles or complex terrain, a form of approximation or heuristic is necessary. Not only for finding a feasible and sub-optimal solution but also to negotiate with hardware capacity. The authors conclude that given the difficulties present, solutions must be chosen, to specifically fit the planning problem's characteristics. Understanding issues and effects of uncertainty and robustness constitutes a fundamental aspect for determining an algorithm's computational efficiency, optimality, and robustness. The authors note that planning methods that have been surveyed don't include discussion on implementation. Simulation results based on idealized vehicles and operational conditions are discussed instead. The authors also note that it is going to be impossible to have a solution that is provably complete or optimal for problems with differential constraints.

Path planning has been used with fixed-wing UAVs in mind, in the recent past. Lee et

al.[24] show this by using RRT-based path planning for a fixed-wing UAV with constraints. The authors use a motion primitive set is used to extend the tree to reflect the dynamic limits of the target aircraft. This is done so that the algorithm can generate dynamically feasible paths and specify approach direction and arrival time constraints. The motion primitive set is based on Bézier curves. This is used to ensure the continuity of the heading angle. The authors say the main reason for using Bézier curves to extend the tree is that a smooth flight path is essential for a fixed-wing aircraft. Since RRT generates jagged and unnatural paths this could cause the UAV to overactuate controls to follow the path. Figure 7 shows the Bézier curve base motion primitive set and figure 8

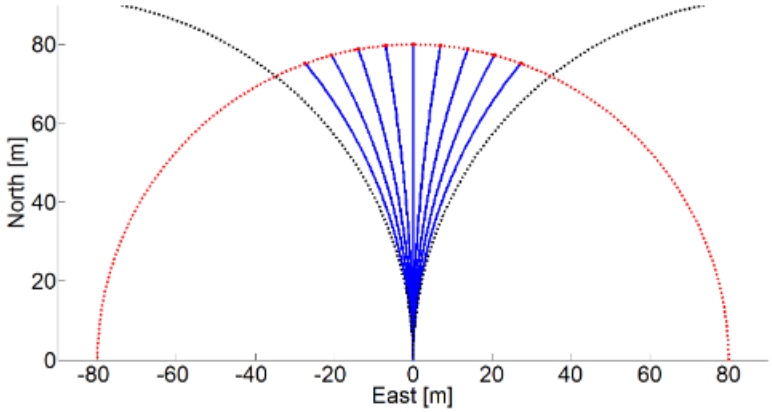

Figure 7: Bézier Curve Based Motion Primitive Set[24]

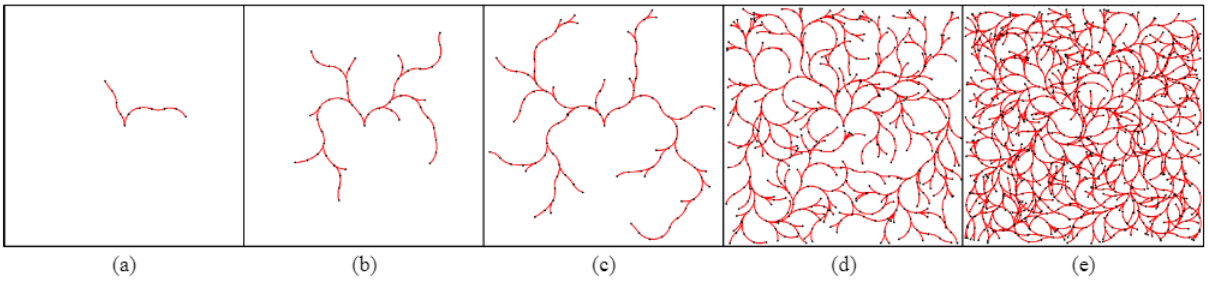

Figure 8: Tree Extension with 10 nodes (a) 50 nodes (b) 100 nodes (c) 500 nodes (d) and 1000 nodes(e)[24]

Lee et al.[24] also performed simulations using their proposed algorithm. The focuses of

their simulations were obstacle-free paths and speed commands with varied initial positions, goal heading angle, and arrival time. Figure 9 shows a path generated, represented in red and green points are where motion primitives are linked.

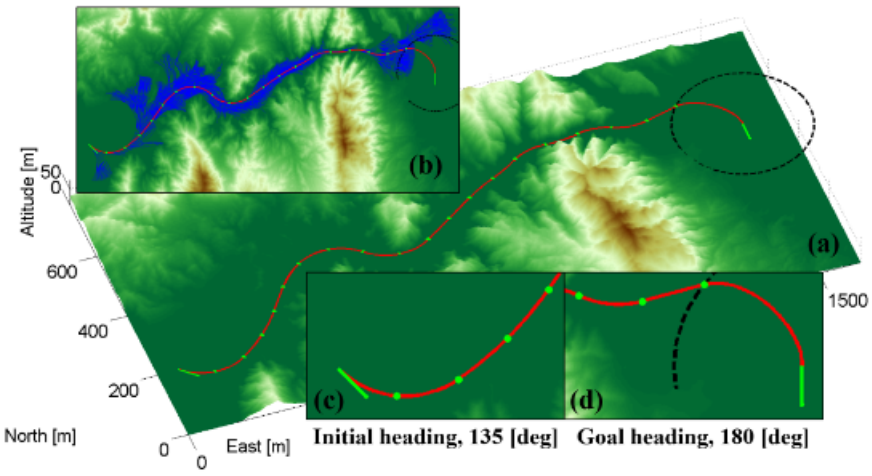

Figure 9: One of the simulation results from Lee's RRT[24]

Meng et al.[27] show a method of transforming the Informed RRT* path planning algorithm from a 2D environment to a 3D environment for UAVs in construction site inspections. 2D Informed RRT* demonstrates a higher convergence rate and final solution quality compared to RRT*. The 2D Informed RRT* uses an ellipsoidal subset of the planning domain, where a 3D Informed RRT* uses an oblique cylinder. This is shown in figure 10. The 3D Informed RRT* solutions are divided into two situations. The first is the initial position is equal with the target position in any dimension this degenerates to 2D path planning. And the path planner generates an ellipsoidal subset of the previous path planning domain. The second where the initial position and the target position are not equal in any dimension, the path planner generates an oblique cylinder subset. The authors suggest an improvement over the general RRT* in search space, path length, calculation time, and the number of branches. The authors demonstrated the use of this algorithm on a real UAV. The experiments took place in a 5×5×5 cubic meter space with cylindrical obstacles attached to autonomous ground vehicles to depict dynamic obstacles. The authors focus the experiments

on the convergence rate and the performance of a global optimizing path.

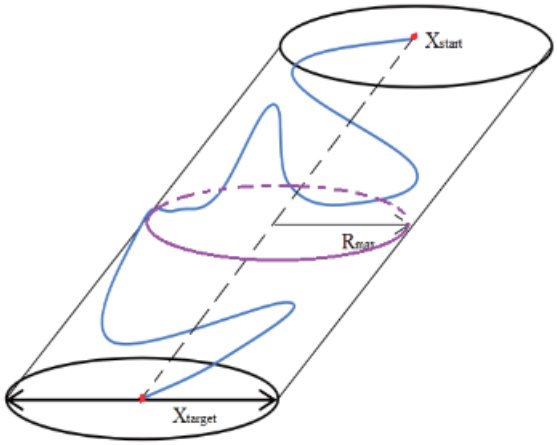

Figure 10: Oblique cylinder space subset for 3D Informed RRT*[27]

Wen et al.[40] used a combination of RRT, a variation of RRT called dynamic domain rapidly-exploring random tree (DDRRT), and RRT* to allow a UAV to plan a path online through a dense obstacle environment. The authors utilize a subgoal routine to generate points to generate paths. Each of these subgoal points leads toward the overall goal point. The RRT algorithm is used to create paths towards the subgoals and is bounded by the sensing horizon of the UAV. DDRRT is used to deal with complex obstacles in the environment as it prioritizes nodes that are not in the proximity of an obstacle. While RRT* optimizes the path toward the subgoal location. Obstacles in the environment are modeled using potential fields. The use of dynamic threats uses the RRT-Connect path planning algorithm to track the UAV. The algorithm used in Monte Carlo simulation to showcase behaviors with online path planning. Figure 11 shows a result of one of these simulations.

Darbari et al.[8] couples together a global path planner with a local path planner. The global path planner uses a discretized 3D occupancy grid of the search space with each cell indicating the risk associated with it. The grid is then sampled randomly using a probabilistic road map (PRM). The samples are clustered into safe regions. The centroid of every safe region is used with the A* algorithm to find the optimal path between them. The

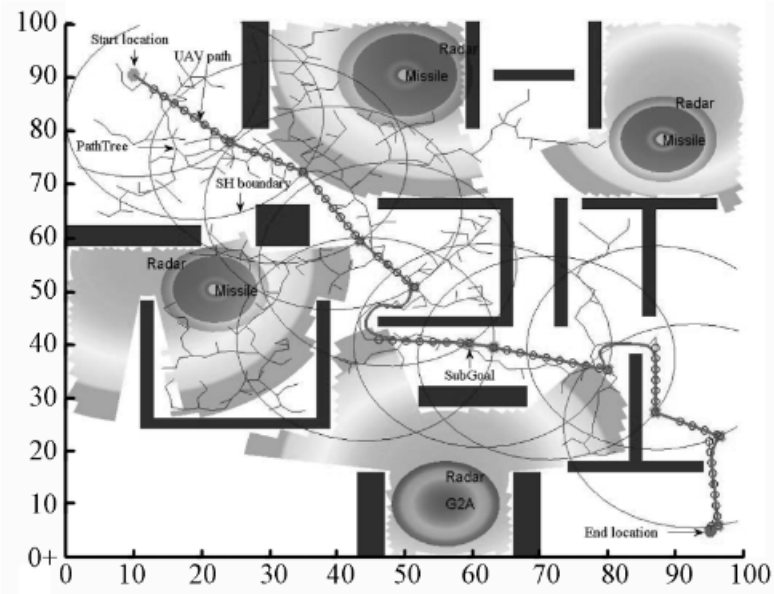

Figure 11: A solution of the path planning algorithm[40]

local planner is used to reduce the computation of continuously updating the global path. It generates motion primitives based on shortest Dubins path between two configurations. Dubins path is modified to steer the UAV clear of obstruction from the path if an obstacle lies on the connecting path between two nodes. The Markov Decision Process is used to evade impeding dynamic obstacles. The authors use a fixed-wing UAV for simulated experiments along with ArduPilot for simulated onboard autopilot and control.

James et al.[18] develop an algorithm that generates path trajectories for multiple aircraft flying to multiple destinations while maintaining separation between the aircraft. The steps for this algorithm start with generating several possible future positions for aircraft. Computes the distance of these positions from other aircraft and their final destinations. Choose new position and dynamics for each aircraft based on proximity to their destination and distance from other aircraft. These steps are iterated with time and are repeated until all aircraft have reached their destinations. The authors use both conventional and quantum computing to run the algorithm. Simulations are done with use cases containing a variety of UAVs at an airport, with different UAV corridors and Drone highways.

MathWorks[25] have documentation of a motion planning algorithm that uses RRT for

Fixed-Wing UAV. This method uses 3D Dubins motion primitives to create smooth paths for the UAV from a start pose to a goal pose. The map designed for the simulation is in East-North-Down(ENU) frame. The fixed-wing UAV has aerodynamic constraints of maximum roll angle, flight path angle, and airspeed to have a nonholonomic nature. Figure 12 show the end result of a simulation. Concepts used in this documentation are from Small Unmanned Aircraft: Theory and Practice[2].

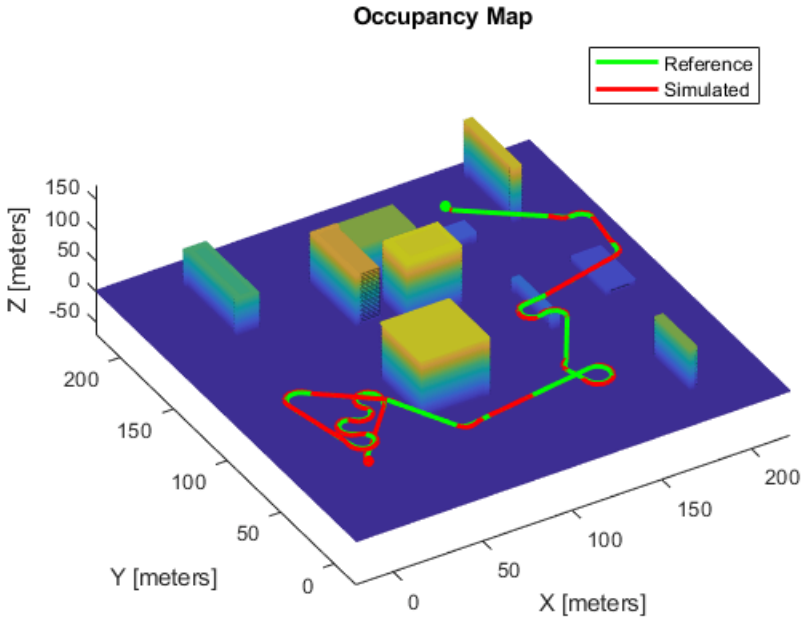

Figure 12: Simulated results of the RRT algorithm in MatLab[25]

Choudhury et al.[4] developed an algorithm that builds from RRT* called RRT*-AR. Where the AR stands for alternate routes and the name for the algorithm being Accelerated Alternate Routes with Replanning. This is used with application with a helicopter when emergency landing situations arise. The RRT*-AR algorithm generates multiple paths to a goal or multiple goals within the graph. Once planning is finished multiple leaf vertices corresponding to candidate solutions are obtained. A subset of these candidates is selected if they satisfy the definition of an AR. This is done with a few new algorithms. Two algorithms worked together in choosing alternate parents. Checking to see if the best parent already

has children the next best parent would possibly be chosen and the same applies when rewiring the tree. The authors wanted to generate alternate routes in real-time, so two algorithms were introduced to speed up the algorithm. One used cost approximation while choosing parents and the other exploited the reachability volume of the vehicle with dynamic constraints. They also implement a method to reuse the search tree by latching the vehicle state onto an existing tree. This algorithm was tested in simulation on a UH-60L Black Hawk.

Divkoti et al.[11] design an algorithm called Regular Chains of Segments Path Planner (RCS). The regular chains of segments are used as sub-paths when moving from one obstacle vertex to another. Each segment has a turning point and a turning angle for an extension from one segment to another. A max angle is defined to not have sharp turns and to generate smooth paths in the graph. To find the shortest paths the authors modify Djikstra's algorithm to process edges with minimum priority. As the algorithm finds shorter paths the minimum priority edge is removed until a path is successfully found between the start point and goal point. Figure 13 shows a comparative result between RCS and the A* algorithm. Through multiple simulations, the authors show that the RCS algorithm generates paths much faster than A* at equivalent path lengths.

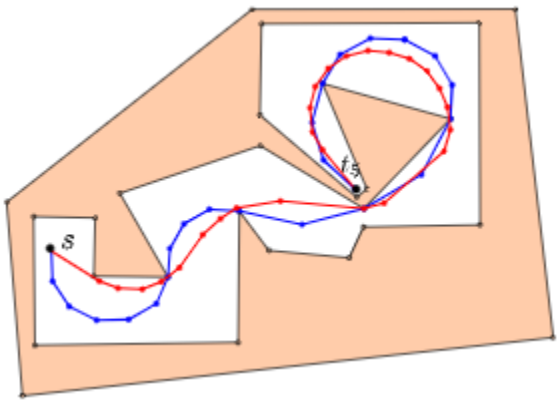

Figure 13: Generated paths for RCS (blue) and A* (red)[11]

Zhang et al.[43] adopt the Improved A* algorithm, sparse A* algorithm (SAS), and

dynamic A* algorithm (D*) to solve a problem on penetration route planning for a stealth UAV. D* like discussed earlier can replan paths in instances when a new obstacle appears in the graph. SAS improves on the A* algorithm by applying constraints on the search to reduce search time, optimize the solution, and improve search efficiency. The improved A* algorithm is a learning real-time A* algorithm (LRTA-Star), this is used for real-time planning in dynamic environments. It updates the costs of each state to the target location. However, due to the nature of the algorithm, it can create broken lines that are hard for a UAV to follow so it is combined with model-based predictive control. This algorithm is simulated with a variety of scenarios to compare paths between the three algorithms.

Mechali et al.[26] develop a variation of the RRT* algorithm called Rectified RRT* and is used with UAVs in a 3D environment. The rectified RRT* algorithm uses a method to smooth and rectify a final path generated by RRT* to reduce the energy consumption during flight. The method uses waypoints from the path with a given distance between two consecutive points, to reduce the path length and save energy. This algorithm was tested in software-in-the-loop, hardware-in-the-loop, and real flight experiments. Where the UAV was outfitted with an NVIDIA Jetson TX2 board, which performed the algorithm. When the path for the UAV is found using RRT* it then flies to the goal location and then hovers when completed.

Newton[28] used an AR.Drone Quadcopter to fly a path planned by RRT. The aircraft is represented as a point to ignore orientation. Virtual Obstacles are presented in the graph. The RRT algorithm is given the UAV's start location and a goal location. Dijkstra's shortest path is used to find the shortest path generated by RRT. The UAV used a script to run the RRT algorithm and when a path was found it would follow it till the end goal was reached.

Cobano et al.[5] used an improved RRT and RRT* path planning algorithm on a fixed-wing UAV for data collection in a Wireless Sensor Network (WSN). The new algorithm used non-uniform random distribution around WSN collection zones to explore them first. A multivariate normal distribution was used to create new samples in the graph this oriented

the tree search in areas of more interest. Additionally sampling was biased toward the goal points when using the RRT* algorithm. The algorithm was simulated for validation and used on a real aircraft for data collection. The algorithm was performed on a auxiliary computer and pushed to the aircraft as a flight plan.

A great many have worked on path planning algorithms from a variety of viewpoints. The works of literature used here focus on applying path planning to UAVs in one way or another. Some created outright new algorithms for the matter while others modified existing algorithms to fit the capability of being used on a UAV. A good portion of these algorithms utilizes the entire graph to create a global path while making minute adjustments called local planning. Most authors also utilize the capability of using simulations to showcase the validity of the algorithm. However, they don't implement them into UAVs in real flights. Those that do utilize auxiliary computers or rather strong and partially expensive boards to generate a path and yet even these algorithms generate a path for the UAV to follow and are applied like a flight plan.

CHAPTER III

METHODOLOGY

3.1 Local Path Planning

To implement a path planning algorithm onboard a UAV using a microcomputer, the algorithm must be lightweight and fast. The approach taken here is to use the popular sample-based algorithm RRT and modify it for use with a UAV's embedded system for capability with autonomous flight. The RRT* algorithm is used as a baseline since it is capable of making more optimal paths compared to the basic RRT. Typically path planning algorithms find a path by searching the entirety of the graph. This is especially the case with the RRT and RRT* algorithms. The parameters of the search tree can be adjusted with edge distance and criteria for completion. Even though RRT is considered a fast path planning algorithm the graph size can be a major contributor to the search time. Adjusting the search parameters can quicken the search time.

To reduce this search time, the path planning algorithm will be reduced from a global search to a local search. The way this is done is through reducing the sample free routine's area to a small circular area around the root node. The radius of the search area is predefined. The area in which a tree can extend is then constrained to a bounded angle, this makes the search area have a conical shape. Figure 14 shows this conical search area in the shaded region. The search tree can have a limited amount of nodes, a defined search time, or any other criteria that are wanted just like a regular RRT. Once the search tree is complete it produces a small path segment.

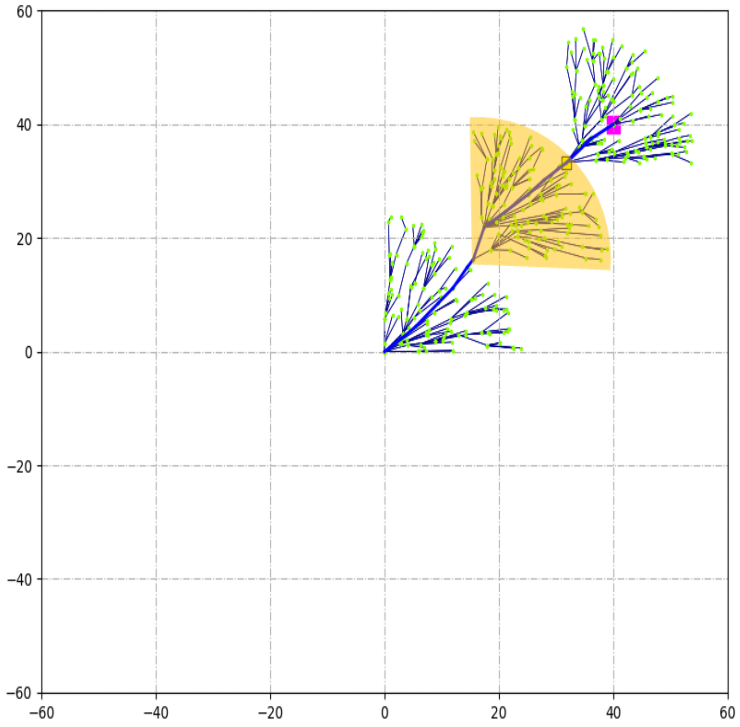

Figure 14: The MiniRRT algorithm

The criteria for choosing a branch to proceed down is based on the leaf nodes for the search tree. Whichever leaf node has the minimal cost is the one chosen. The branch that attaches the leaf to the root is picked and this creates the path segment. Once finished a new search tree is planted at the location of the leaf node. The angle that the conical search area faces depends on the previous tree's path. The last edge of the path is used as the centerline for the search area or the angle based on the horizon with the last two nodes of the path. This will create a smooth transition from tree to tree. This method uses small search trees in number to reach the goal.

Due to the way RRT* searches the local area. Nodes in the tree can be very close together due to the rewiring neighbors routine they can also have jagged paths. So, a B-spline is applied to finished path segments. A B-spline or basic spline is a spline function that has minimal support concerning given parameters, like degree, points or knots, etc. Each knot is equidistant from the other, while the degree influences the curve of the line.

Applying the B-spline to the path smooths it while also creating equidistant points along the path. These points are considered waypoints by the UAV to track.

Using the methods discussed should reduce the search time for paths within the graph. While also making it capable of doing continuous searches to multiple goal points without a reduction in search time compared to other RRT algorithms. When a path is created with RRT the entirety of the tree must be destroyed to create a new path to the next goal. Algorithm 3 details the proposed algorithm line by line. This new proposed algorithm is called MiniRRT.

Algorithm 3: MiniRRT

Input: $C = \{C_{free}, C_{obs}\}, x_{init}, x_{goal}, \delta, k, \psi, n$
Output: $\pi = x_{init}, ..., x_{goal}$
Initialize $G = (V, E)$, $Path = None$;
while $x_{goal} \notin Path$ **do**
    if $card(V) = 0$ **then**
        Initialize **MiniRRT***(G, x_{init}, x_{goal}, δ, k, ψ);
        $V \leftarrow \{x_{init}\}$;
    if $card(V) \leq n$ **then**
        $segment \leftarrow$ **search**();
    else
        $Path \leftarrow Path \cup \{segment\}$;
        $x_{init} \leftarrow$ last node in $Path$;
        clear G;

In the MiniRRT algorithm, the input is the configuration space that includes the free space and obstacle space. The start location and the goal location within the graph are denoted as x_{init} and x_{goal}, respectively. δ the edge saturation constant, this defines the saturation limit between nodes or the edge length. k, the shrinking factor, is used to influence the size of the D-ball radius for nearest neighbor searching. ψ the aircraft's heading, this is to designate the direction of the centerline for search trees. And, n is the maximum number of nodes. The output of the algorithm is a path π from the start location and goal location. The Graph G is initialized with sets V and E, for nodes and edges and the $Path$ is equal to None. A while loop is used to keep the algorithm running until x_{goal} is in the path. The

cardinality of the set V is checked if it is zero. Cardinality refers to the number of elements in a set. If the set of V is zero the MiniRRT search is initialized with the Graph, start location, goal location, δ, k, and ψ. And the start location is added to the set V and acts as the root node. The cardinality of V is checked again to see if it is less than n. If so, the search function is called shown in Algroithm 4 and returns a path segment called segment. If the number of elements in set V is greater than n. Then the segment is added to the $Path$ and the last node in the $Path$ is chosen as the start location for the next search tree. Lastly, Graph G is cleared of any nodes and edges so that the next search tree does not use any nodes or edges of the previous search tree. This in other words means the previous search tree is abandoned.

Algorithm 4: search

```
r ← shrinkingBallRadius();
x_rand ← sampleFree();
x_nearest ← nearest(G = (V, E), x_rand, r);
x_new ← steer(x_nearest, x_rand);
if collisionFree(x_nearest, x_new) then
    X_near ← extend(x_new, x_nearest);
if x_new ∈ V then
    rewireNeighbors(x_new, X_near);
connectToGoal(x_new);
return computeTrajectory();
```

In the search routine, the *shrinkingBallRadius* routine is called. This uses the equation 3.1.1 to solve for the minimum nearest neighbor search radius to look for nearest neighbors in the graph. This uses the variable γ_{RRT} to influence the size of the search radius, this equation is shown in equation 3.1.2. This uses the variable X_{free} known as the Lebesgue measure of the graph and, k in the shrinking factor.

$$r = ((\gamma_{RRT}/\zeta_D) * (\log_{10}(\text{card}(V))/\text{card}(V)))^{1/d} \tag{3.1.1}$$

$$\gamma_{RRT} = (2^d(1 + (1/d))(\mathcal{X}_{free}))^k \tag{3.1.2}$$

The next subroutine ran is *sampleFree* which randomly selects a node within the defined search area. The *nearest* subroutine finds the nearest neighbor within the graph closest to x_{rand} within the ball radius r. With *steer* x_{rand} is saturated toward its nearest neighbor $x_{nearest}$ to the defined distance δ. The edge connecting x_{new} and $x_{nearest}$ is then checked for collision with an existing obstacle in the graph. If it is free of collision, the *extend* function is ran which finds the best parent node to extend the search tree to and returns X_{near}. A group of nodes within the ball radius r. The x_{new} node to check if it exists in the set of nodes V. If so, the $rewireNeighbors$ function is called using x_{new} and X_{near}. An attempt to connect to the goal using x_{new} is done using the function $connectToGoal$. Lastly, a path segment is created with the search tree using the subroutine $computeTrajectory$.

Using this path planning algorithm on a proper embedded system on board a UAV should allow for it to autonomously plan a path to a defined goal point in real-time operation. Meaning that the UAV will calculate path segments using the MiniRRT algorithm. This will create waypoints for the UAV to track while it continues to calculate a path toward the goal. This will allow the UAV to do the planning and tracking operations in real time. As the waypoint tracking will be done in the time that waypoints are produced. The waypoints will be generated when paths are completed the operations from the MiniRRT algorithm in the order of seconds.

3.2 Software Selection

A variety of software packages are used to create the proposed algorithm. These are also used for ease of implementation on the board that will be used on the UAV for flight tests.

3.2.1 Ardupilot

ArduPilot[38] is a well-known, reliable, and open-source autopilot system that is developed under the terms of the GNU General Public License version 3 (GPLv3) published by the Free Software Foundation, permitting the software to be changed or modified, and redis-

tributed. Ardupilot can be used on a variety of vehicle types, such as fixed-wing aircraft, rotorcraft, ground vehicles, sailboats, submarines, and more. For this project, it is used as an autonomous controller to manipulate control surfaces and motors on UAVs.

3.2.2 Robot Operating System

ROS[17] stands for Robotic Operating Software and, it is a framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. ROS is created by and, property of Open Source Robotics Foundation, Inc. ROS allows for customized messages to be used within the system. The ROS environment works with both the C++ and Python programming languages to create customized messages that include a variety of parameters, like location, heading, velocity, etc.

3.2.3 Programming Languages

The popular programming language Python[12] developed and maintained by the Python Software Foundation is used for the creation of the path planning algorithm. Using the Python language allows for quick implementation to the systems whenever changes are needed for purpose of debugging, parameter changes, etc. Additionally, using Python opens up the vast library of Python packages. Two packages used in the coding and implementation of the path planning algorithm are NetworkX[9] and Shapely[13].

NetworkX is an open-source package under the 3-clause BSD license, which allows for the redistribution and use in both source and binary forms, with or without modifications following certain guidelines. It is used for the creation, manipulation, and study of the structure, dynamics, and functions, of complex networks. One of these complex networks is data structures of graphs, digraphs, and multigraphs. So the capabilities of adding and removing nodes and edges with other graph functionality are built into the package. To prep NetworkX for use with this type of algorithm it needed to be stripped down as the

NetworkX functions modeled graphs a certain way that did not allow for the methods that were attempting to be implemented. So, much of the overall functionality was reduced down only to the bare minimum of what was needed for use with the algorithm.

Shapely is also a BSD-licensed Python package and it is used for the manipulation and analysis of planar geometric objects. It is based on the GEOS and JTS libraries. This package is mainly used for the modeling of obstacles in the graph as it allows the creation of any shape. Nodes and edges that are created with the NetworkX packages can be used as inputs to create points and lines for Shapely to understand. These can then be used to check for collision avoidance as Shapely has the functionality to check if these points lie within an obstacle's space or if lines intersect obstacles.

The C++ programming language was used to create the guidance and navigation modules for the aircraft. The serial link communication module was as well. These were both implemented onto the embedded systems.

3.3 Hardware Selection

The path planning algorithm will be implemented and tested using the following hardware during flight tests.

3.3.1 Embedded Systems

A popular hardware choice for UAS autopilot is the Pixhawk. The Pixwawk is an open-source platform that uses the Ardupilot software. The Dronecode Foundation is the overseeing authority on the Pixhawk Program and its open standards for hardware specifications and guidelines for drone systems development. It is used for lower-level flight controls on aircraft. Figure 15 shows a Hex Cube Black Flight Controller, previously known as the Pixhawk 2.1, and is manufactured by CubePilot. This is the specific hardware board that will be implemented onto the aircraft. It has a 32-bit ARM Cortex-M4F architecture with 256 kilobytes of memory. It also has a microSD card for high-rate logging.

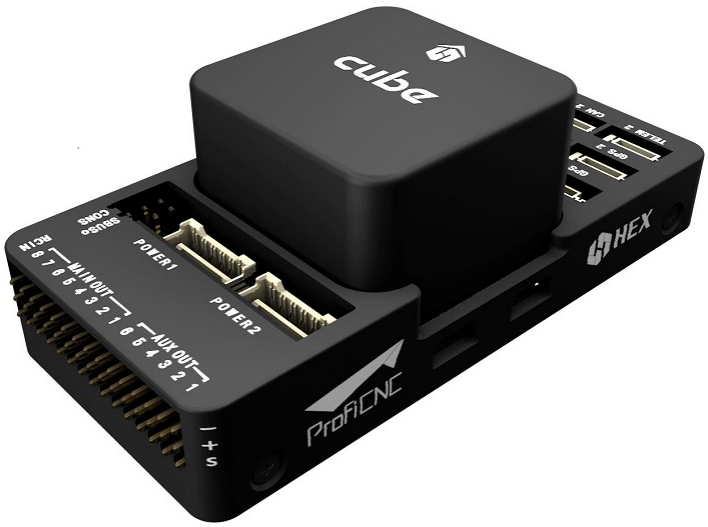

Figure 15: Hex Cube Black[7]

A secondary flight computer is used to operate the proposed path planning algorithm. There are plenty of microcomputer candidates that can do the job and a few are named and compared. Of these, are the Raspberry Pi 4 Model B, Latte Panda, and the BeagleBone Black. The Raspberry Pi has been a popular choice for microcomputers for a range of projects, LattePanda is a Windows 10 enabled development board. And the BeagleBone Black is another development board that is low-cost and uses Linux. Table 1 shows a comparison of CPU speed and RAM Size, as well as the size of the board and listed price. These specs were gathered from each microcomputer's respective web page.

Name	CPU	RAM	Size	Price
Raspberry Pi 4B	Quad Core 64-bit, 1.5GHz	2GB LPDDR4	85 × 56 mm	$35
LattePanda	Quad Core 64-bit, 1.92GHz	2GB DDR3L	88 × 70 mm	$119
BeagleBone Black	ARM 32-bit, 1GHz	512MB DDR3	86 × 48 mm	≈ $60

Table 1: Microcomputer Comparison

When looking for a microcomputer the priority characteristic was the CPU speed. This would allow for quicker computation time and allow for computation-intensive methods to be

an option. RAM is not as important since the type of computation that is happening doesn't require a lot of RAM. Size is important since the microcomputer will have to fit inside the UAS with limited space allowed. However, these microcomputers are of similar size apart from the BeagleBone Black being slightly smaller. The price of each microcomputer is also list, but since this project is for swarm-based planning multiple microcomputers are required. Taking these categories into account the LattePanda seems to be the obvious winner based on CPU speed alone, however it comes at a significant price compared to the other two. Although having many LattePandas that are readily available for the project, they will be used anyways. Figure 16 shows a picture of the LattePanda. A potential microcomputer for future work could be the BeagleBone Blue. It was not considered cause it was not released at the time of decision for using the LattePanda.

Figure 16: Picture of a LattePanda [39]

Hardware used for communication can either be WiFi-based or by Radio. Since UASs will be outside and flown where a WiFi access point is not readily available. Radio communication will be used instead. The specific radio hardware that will be used is called the RFD900x by RFDesign, which can be seen in figure 17 [33].

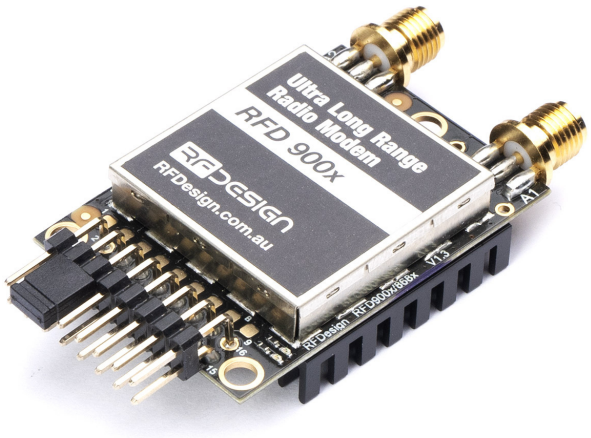

Figure 17: RFD900x Modem[33]

With the RFD900x Modem there comes multiple different communication architectures and node topologies. These include Peer-to-peer, Multi-point networks, and asynchronous mesh. The Peer-to-peer architecture is the default configuration settings on the RFD900x. It allows the user to transmit and received data across a great distance between two nodes. Whenever two nodes come within range of each other and have compatible parameters they will synchronize and communication will begin. Figure 18 shows a diagram from the manual.

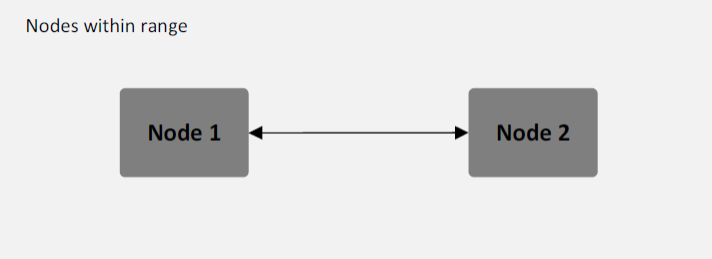

Figure 18: Peer-to-peer network diagram[32]

An asynchronous mesh offers straightforward communication that allows a user to transmit and receive data between two, or more, or all nodes within the network. Each node in the mesh communicates to all other nodes within its radio frequency range. Each node also must have different node id's for the network to work properly. Figure 19 shows an

asynchronous mesh topology.

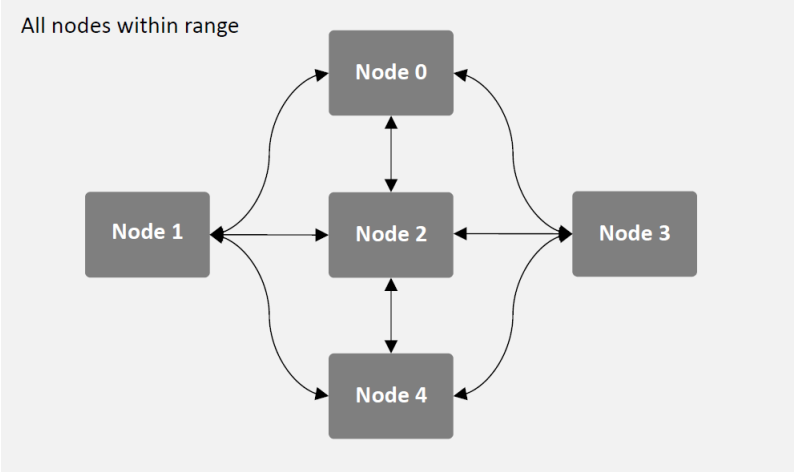

Figure 19: Asynchronous Mesh Topoplogy[30]

With a multipoint network each node withing the network must have a unique ID. A specific node must be defined as the master, it controls the radio time slot allocations. In the default set up, nodes broadcast information to all other nodes. However, it can be set for a node to address another specific node give the Node ID parameter. A multipoint network has the advantage of the ground control station being the master to synchronize timings between nodes so that messages are not dropped compared to the asynchronous mesh network. Figure 20 shows a diagram for a multipoint network.

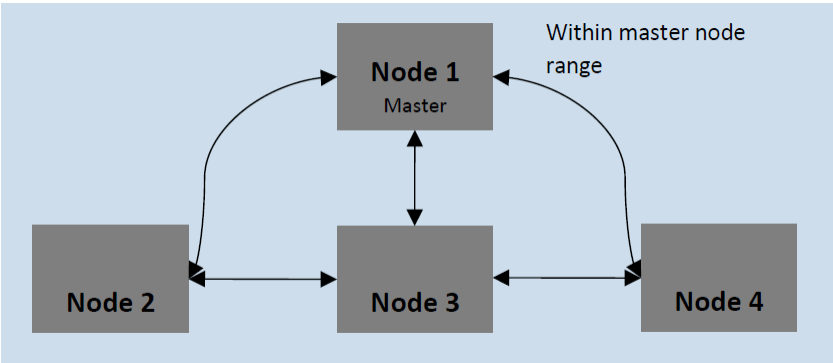

Figure 20: Mulitpoint Network Diagram[31]

With each method discussed the chosen architecture for the experiments will be the

asynchronous mesh as it offers fast straight forward communication that can be used and then later forgotten.

3.3.2 Unmanned Aerial Vehicles

A few unmanned aircraft were investigated for implementation of the proposed path planning algorithm. For fixed-wing aircraft, the Nano Talon EVO and the Mini Skyhunter V2 were looked at. These aircraft can be seen in figure 21 for the Nano Talon and figure 22 for the Mini Skyhunter V2. Both of these aircraft are manufactured by SonicModell. SonicModell is a manufacturing company that specializes in Radio Controlled Foam Electric RC Hobby Aircraft. Both of these models are equally viable for use with the planning algorithm and fly well and have plenty of space to fit all the necessary components inside. However, due to the Nano Talon EVO being readily available it will be the chosen aircraft to fly over the Mini Skyhunter V2. Another type of aircraft will also be used for the implementation of the path planning algorithm. The Eagle is a quadcopter developed by the Unmanned Systems Research Institute Team will be also be used to test the capabilities of the algorithm on a rotorcraft. A picture of it can be seen in figure 23. It has the capability of testing the algorithm in smaller environments, as well as quicker compared to the Nano Talon EVO. Nonetheless, both platforms will be tested with the algorithm implemented on them.

Figure 21: Iso-view of the ZOHD Nano Talon EVO [35]

Figure 22: Iso-view of the SonicModell Mini Skynuhter V2 [34]

Figure 23: The Eagle multi-rotor UAV

CHAPTER IV

EXPERIMENTATION

4.1 Case Study

A comparative case study is done to compare the search time and path cost of the MiniRRT algorithm to RRT and RRT*. A variety of environments are created to test the capabilities of the algorithms. These environments span a range graph sizes from 100×100, 250×250, 500×500, to 1000×1000 unit maps. With different obstacle orientations with a varying number of obstacles. With the varying graph sizes, the obstacles will be scaled properly. The start and goal locations will be placed at the bottom left corner and top right corner respectfully and will be scaled properly with graph size changes as well. The δ parameter will be kept the same through each simulation for each algorithm to keep tree growth the same between them all. The shrinking factor k will be kept constant for each simulation and algorithm. The node count for the MiniRRT trees will be set to a cap of 250 per tree and the total number of nodes used will be the max number of nodes the RRT and RRT* algorithms will be able to search.

4.2 Software-in-the-loop

Using Windows Subsystem for Linux (WSL) running Ubuntu 20.04 LTS allowed for a software simulation on a laptop or desktop. Within Ubuntu 20.04 LTS the ArduPilot API was cloned from ArduPilot's main Github repository. ROS's latest version, Noetic Ninjemys was also installed. This allowed the use of Python ROS Packages within WSL. On the Windows side of the system, Mission Planner was installed for visualization of the aircraft flying the

mission. Mission Planner is Ardupilot Graphical User Interface (GUI) it is a very powerful tool as it can, configure autopilot settings for the vehicle, point-and-click waypoint, fencing, and rally point entry, and downloading of mission log files. Using a communication script written in C++ allows the software to communicate from Ubuntu to Mission Planner through User Datagram Protocol or UDP, for help with visualization.

For the software-in-the-loop (SITL) experiments a few environments were created to test how the algorithm reacts to no obstacles, one obstacle, and many obstacles. Each of the obstacles in the setup were static obstacles. This means that they do not move from the the position they are set at. Obstacles are placed inside the graph at a position that does not conflict with the goal or start positions. The obstacles are created through the Shapley package which manipulates and analyzes planar geometric objects. With Shapely any planar shape can created. The Shapley package has a buffer function to manipulate the size of the obstacle so a range of circle radii can be tested. The graph itself is sized to a 3000 $\times$ 3000 meter square, so it covers a large reasonable plot of land. Since Mission Planner works in the World Geodetic System (WSG84) frame of reference and the graph used the Cartesian System. A function is used to take the path planning algorithm's waypoints transform them to the WSG84 system and pushed to the Pixhawk, the main flight computer, for the aircraft to follow. The data taken from the flight is then changed back for plotting within python using the Matplotlib package.

4.3 Architecture

The system contains a Hex Cube Black Pixhawk with three redundant inertial measurement units (IMUs) and two redundant barometer sensors. The Pixhawk is connected to a here2 Global Position System (GPS) puck for positioning. The Pixhawk interfaces with the LattePanda through a Universal asynchronous receiver transmitter (UART) connection. Within the LattePanda is ROS the Robot Operation System. Two modules are built inside the ROS environment, serial link and guidance and navigation. And the MiniRRT algorithm is placed

alongside these two modules. The guidance and navigation module uses data received from the Pixhawk which is state information and publishes control signals to the actuators and motors on the aircraft. The serial link module receives relevant telemetry data from guidance and navigation and sends it to the ground station via radio modules, the RFD900x modems. These are connected to the LattePanda using a Universal Serial Bus (USB) interface. The modules inside the LattePanda use the ROS environment to exchanged messages with each other. Figure 24 shows the systems layout.

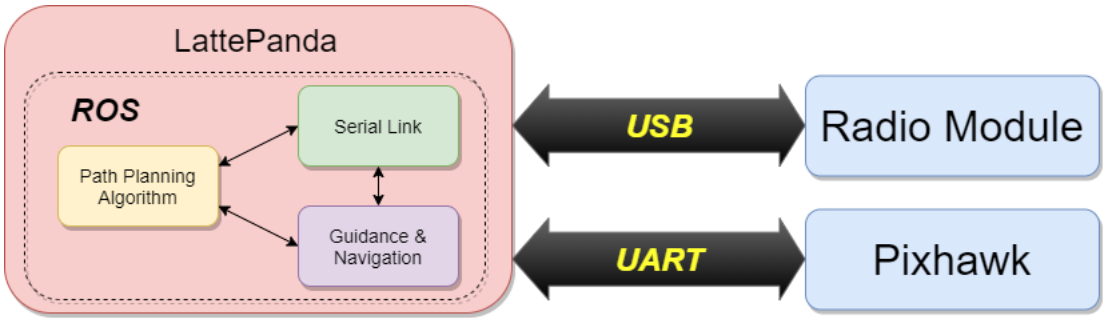

Figure 24: System Diagram

A breakdown on the NanoTalon's setup can be seen in figures 25 and 26. A hole cut out can be seen on the figure containing the side view of the aircraft. This is done because the LattePanda is slightly too large to fit in the belly of the aircraft. Once the LattePanda is secured in the aircraft a piece of foam is used to fill up the hole and taped over. The Hex Cube Black uses a Mini Carrier Board that is mounted on the upper inside of the aircraft's body. And the RFD900x and here2 puck is mounted on top. The battery for the Nano Talon is placed inside the nose of the aircraft.

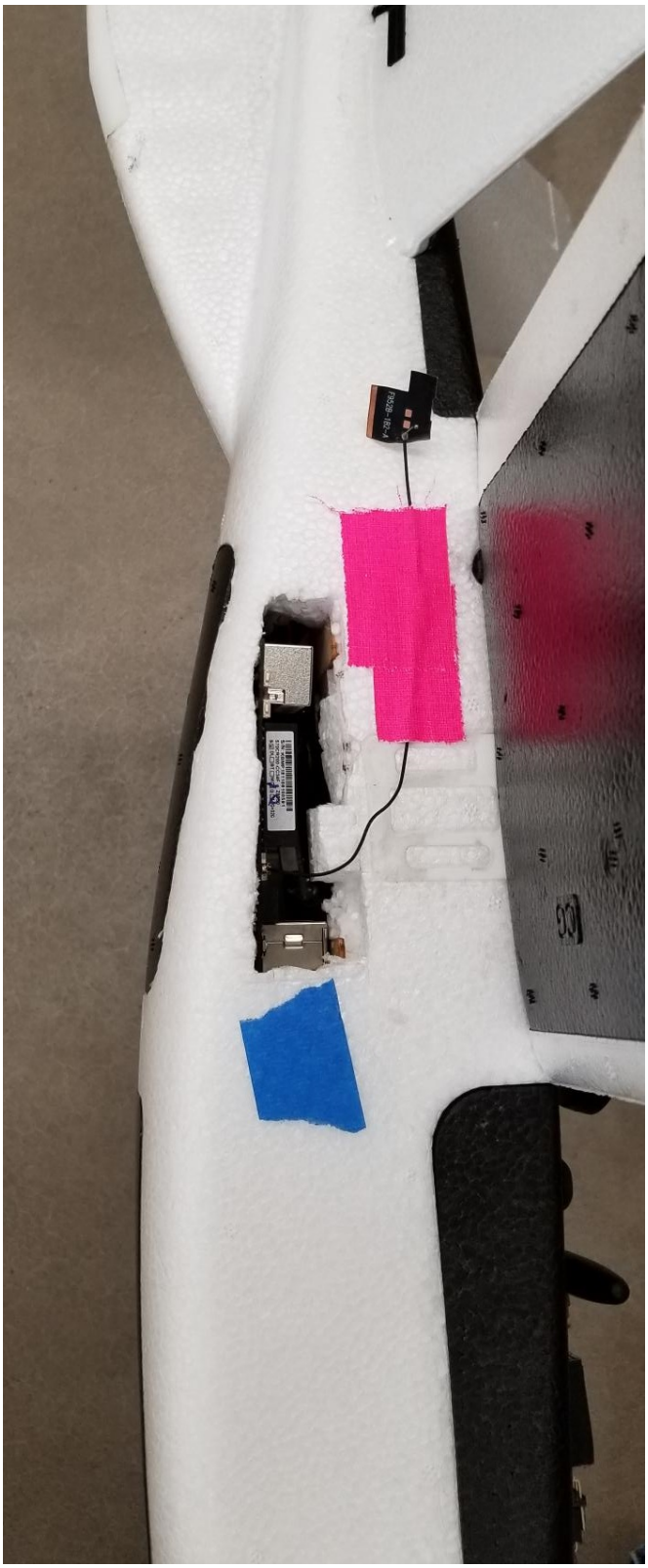

Figure 25: The Side of the Nano Talon

Figure 26: The Bottom of the Nano Talon

The Eagle can be seen in figure 27. The LattePanda is secured to the top plate of the Eagle using two different Velcro straps. The Pixwhak platform uses a Kore Carrier Board that is integrated between the top and bottom plates. The RFD900x modem is Velcroed to the top plate next to the GPS puck. And the battery is strapped in the center of the top plate.

Figure 27: Iso-view of the Eagle Multi-rotor

4.4 Flight Tests

The flight tests for the Nano Talon will have a single virtual obstacle that is placed in the center of the graph. A goal point will be placed within the graph, and it will be a test to see if the aircraft will reach that goal. The test will take place at the OSU Unmanned Aircraft Flight Station.

Figure 28: Nano Talon on the Flight Station's Runway

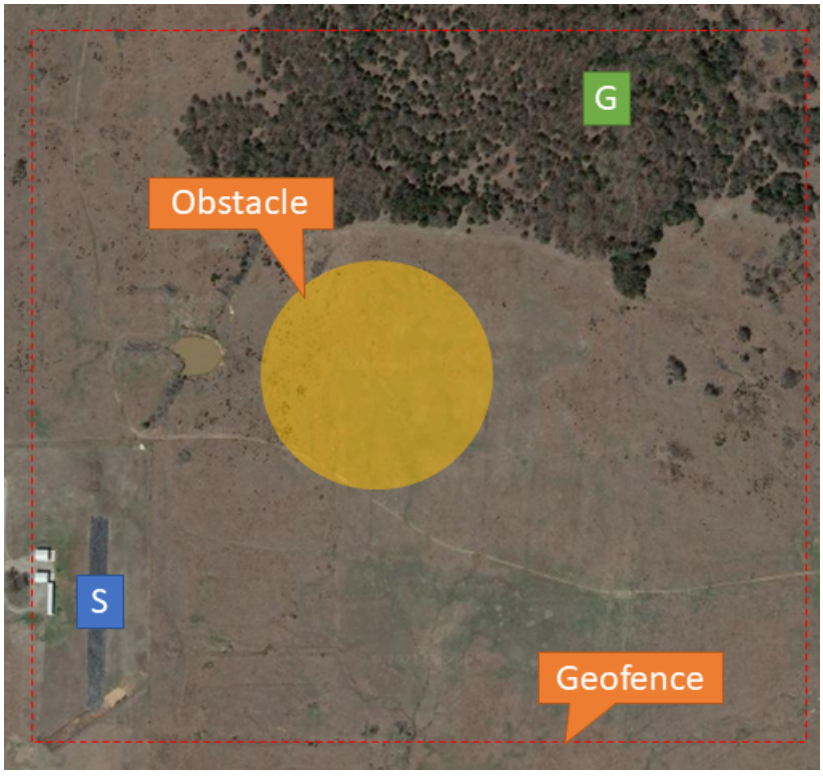

Figure 29: Map Layout of Nano Talon flight test

For the Eagle's live experiments, tests will be conducted at the soccer fields outside of the Excelsior Laboratory. For the soccer field test a goal point will be placed on the opposite corner of the field with an virtual obstacle placed in between them. Figure 30 shows a layout of the experiment. The mission is for the Eagle to travel to the goal point to the opposite corner of the field and come back to the start point all by using the path planning algorithm. Other tests will take place in the same area, but a variety of virtual obstacle orientations will be placed within the graph.

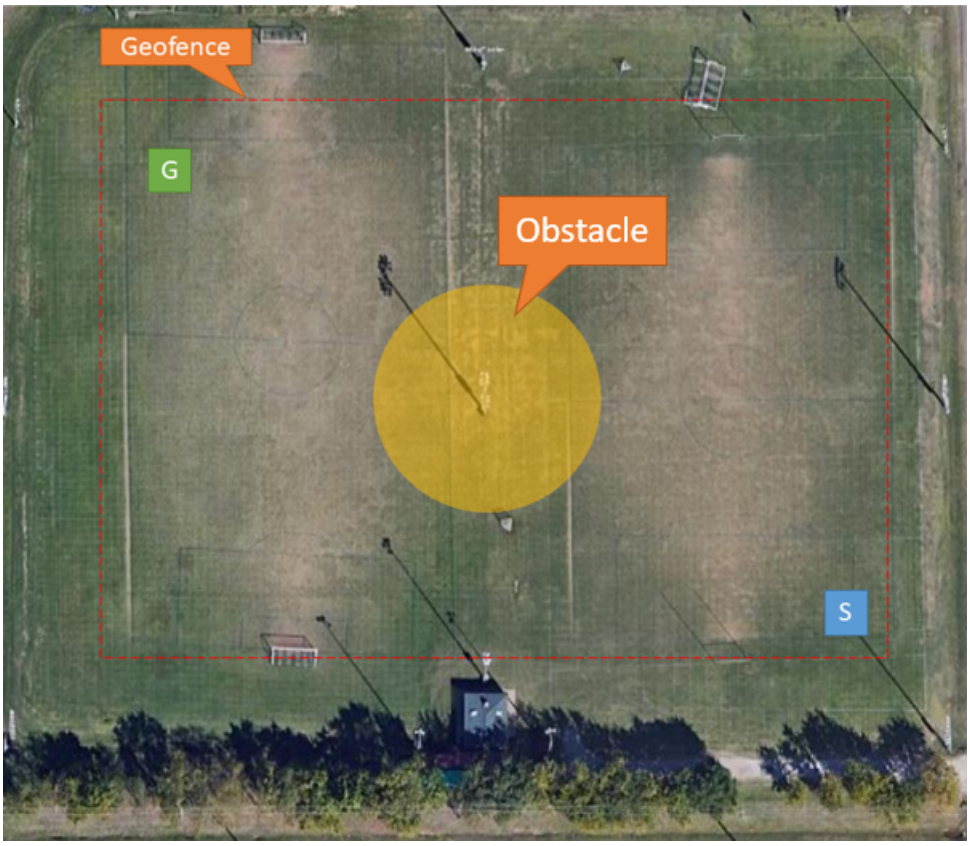

Figure 30: Map Layout of Eagle flight test outside Excelsior

CHAPTER V

RESULTS

The MiniRRT path planning algorithm generates trees in batches so each tree is planted at a point along the preceding tree to generate a path. The algorithm itself can generate different types of trees using different parameters. Each parameter adjusts how the search is done. The number of nodes for a tree can be changed, to quicken the search time of each tree, however this will reduce the optimality of the path segments created with each tree in the overall search. The distance between nodes or the length of edges can be adjusted as well. This will change the the length of the trees branches. The local area prioritized for search can also be altered. Both the height and the width of the trees can be adjusted, by changing the distance and degree at which nodes can be placed. Using these parameters for the path planning algorithm is can adjusted, tuned, and optimized for different types of aircraft to be use with.

5.1 Case Study

A comparison is done to similar path planning algorithms to show differences between them. Numerous different graph sizes and obstacle orientations were created and repeated with each path planning algorithm. Each test was performed on a laptop that had a Intel Core i7-6820HQ and 16 GB of DDR4-2133 Memory. The simulation results can be seen in tables 2 through 5.

Algorithm	Search Time (seconds)	Cost (distance)	Graph Size (distance)
RRT	1.40	187.47	100 x 100
RRT*	3.69	127.85	100 x 100
MiniRRT	2.00	155.72	100 x 100
RRT	30.26	473.42	250 x 250
RRT*	54.41	325.96	250 x 250
MiniRRT	8.12	359.59	250 x 250
RRT	175.20	872.63	500 x 500
RRT*	286.65	660.04	500 x 500
MiniRRT	20.29	670.70	500 x 500
RRT	655.58	1599.52	1000 x 1000
RRT*	1022.43	1280.32	1000 x 1000
MiniRRT	39.34	1306.07	1000 x 1000

Table 2: Simulation 1 Comparison, No Obstacles

In the first simulation, no obstacles were present in the graph, so the path planning algorithms have a simple environment to generate a path. Each set of three simulations use the same graph size, and from there was incrementally increased to 1000 $\times$ 1000 graph size. From table 2 it can be seen that the RRT* and MiniRRT algorithms have a cost that is close to each other, but the time takes longer for the RRT* than the MiniRRT. The comparisons are the same for each change in the graph size. RRT on the other hand had a quicker search time than RRT* but the cost is greater than both the other algorithms. Plots can be seen in figures 31 and 32 showing the comparison of search time and path cost, respectively. To see the optimality of the path solutions the 500 $\times$ 500 can be seen in figures 33, 34 and 35. The rest of the graphs can be seen in the Appendix.

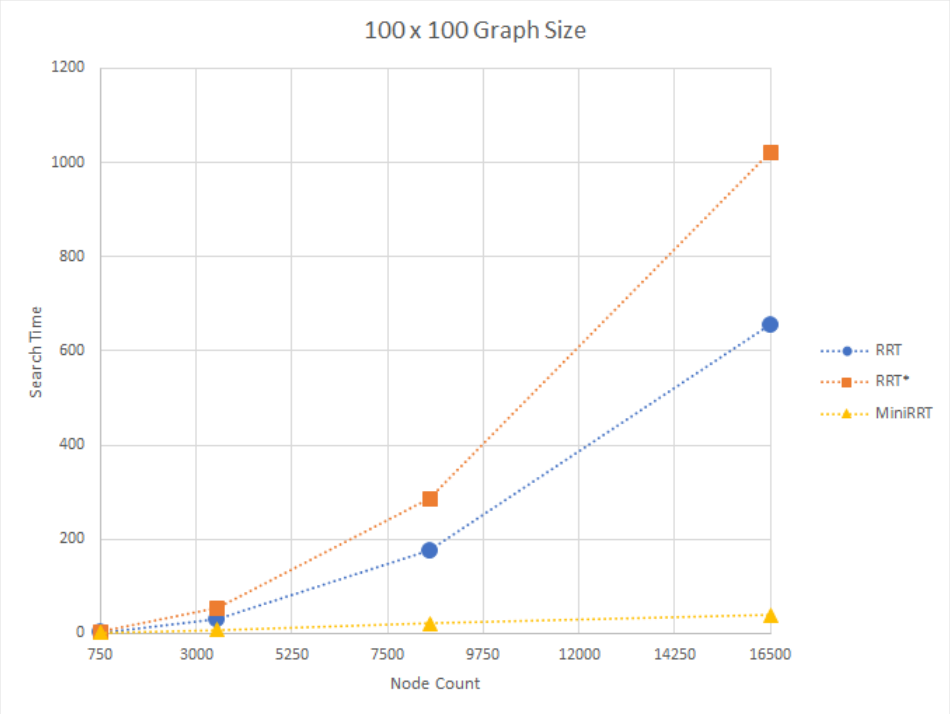

Figure 31: Simulation 1 plot comparison with search time

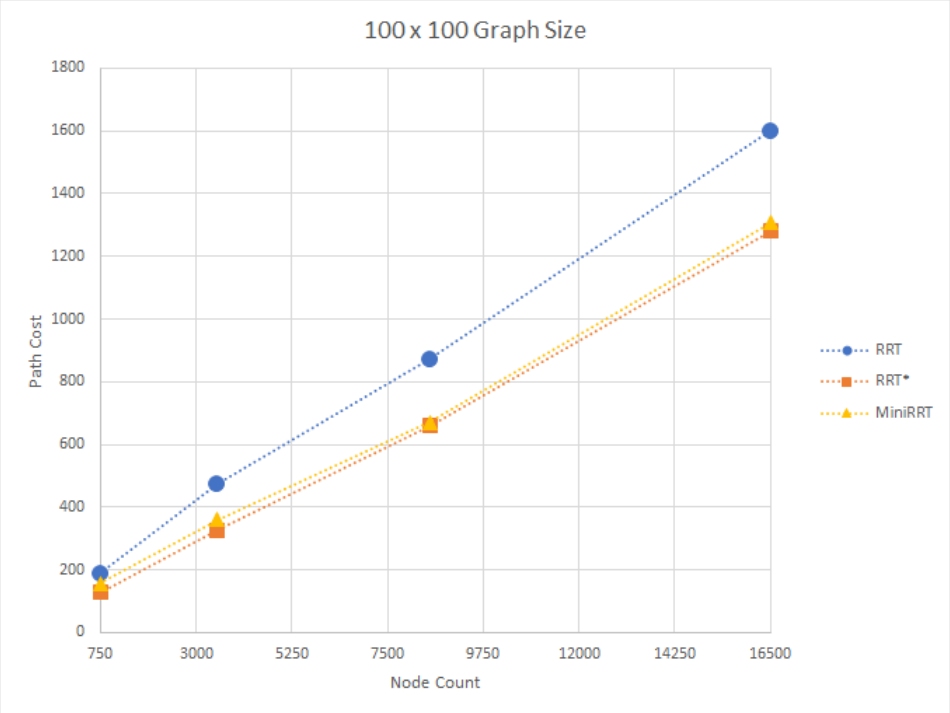

Figure 32: Simulation 1 plot comparison with path cost

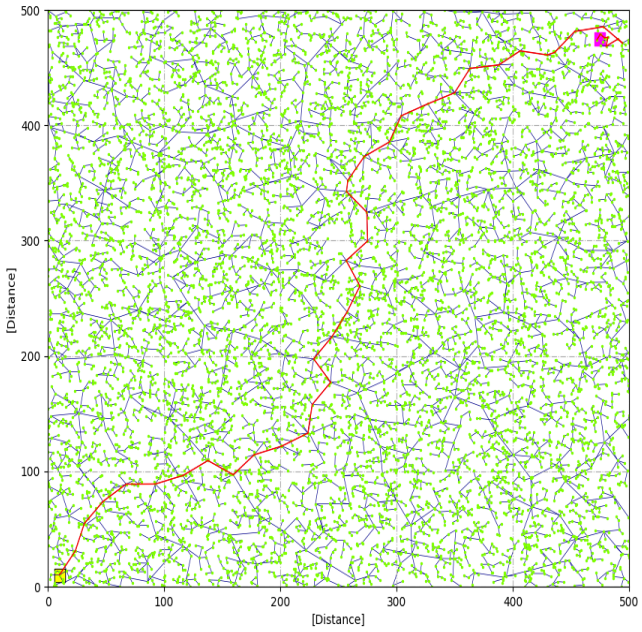

Figure 33: Simulation 1 RRT

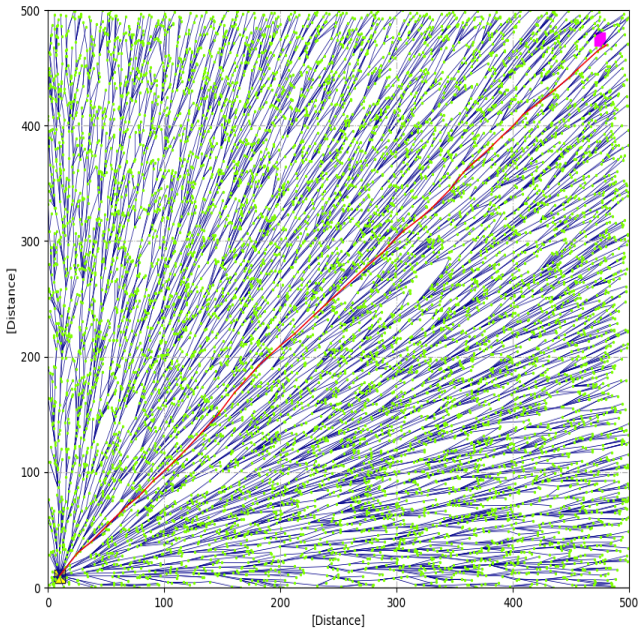

Figure 34: Simulation 1 RRT*

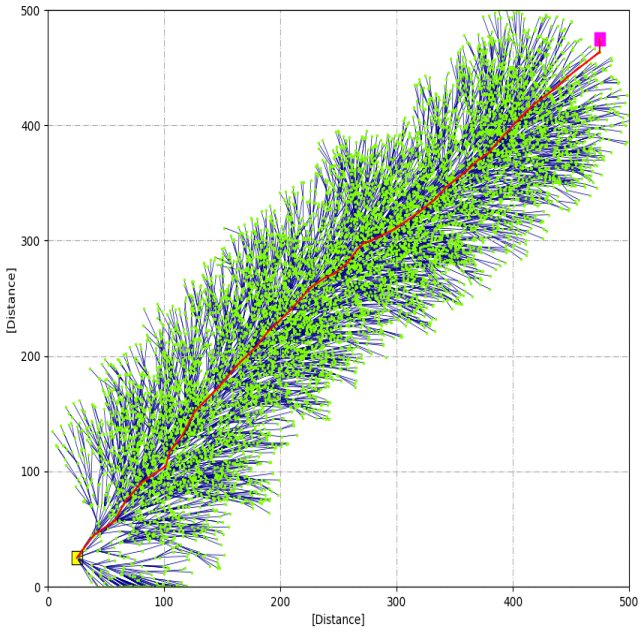

Figure 35: Simulation 1 MiniRRT

In the second comparative simulation a circular object was placed in the center of the graph to make a slightly more complex environment. From the results in table 3 the overall search time and cost of the path solutions has increased due to the obstacle in the graph. Plots for search time and path cost can be seen in figures 36 and 37. The figures 38, 39 and 40 for the 500 $\times$ 500 graphs. The rest of the graphs can be seen in the Appendix.

Algorithm	Search Time (seconds)	Cost (distance)	Graph Size (distance)
RRT	2.925	170.205	100 x 100
RRT*	8.296	133.439	100 x 100
MiniRRT	3.518	158.203	100 x 100
RRT	31.953	440.98	250 x 250
RRT*	90.67	341.729	250 x 250
MiniRRT	10.779	368.003	250 x 250
RRT	197.351	886.529	500 x 500
RRT*	316.713	673.53	500 x 500
MiniRRT	26.251	710.267	500 x 500
RRT	1221.19	1962.162	1000 x 1000
RRT*	1433.748	1385.666	1000 x 1000
MiniRRT	69.741	1444.228	1000 x 1000

Table 3: Simulation 2 Comparison, 1 Obstacle

Figure 36: Simulation 2 plot comparison with search time

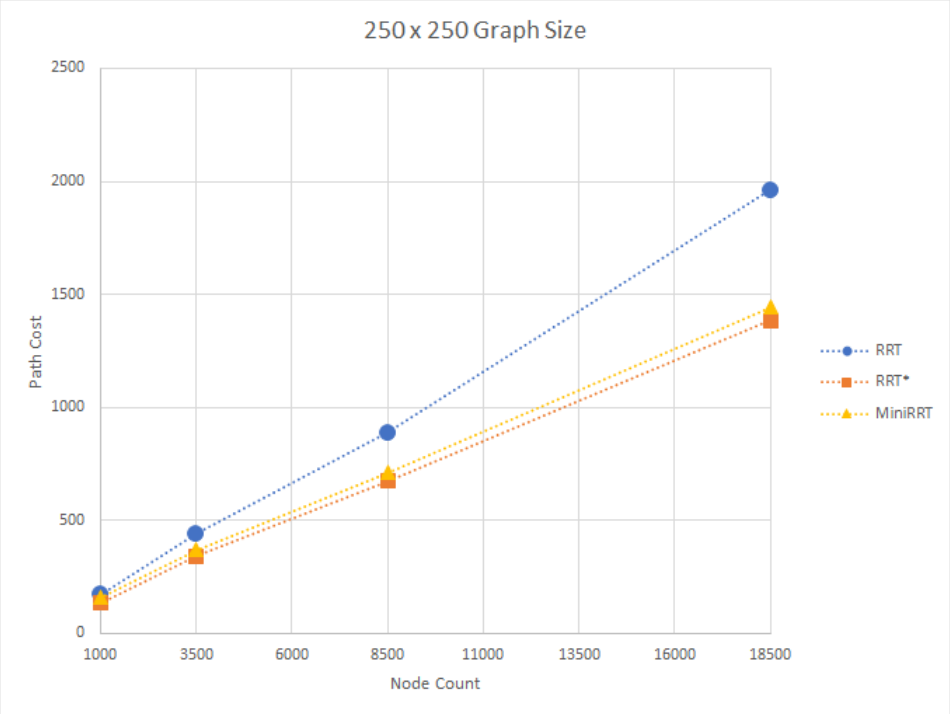

Figure 37: Simulation 2 plot comparison with path cost

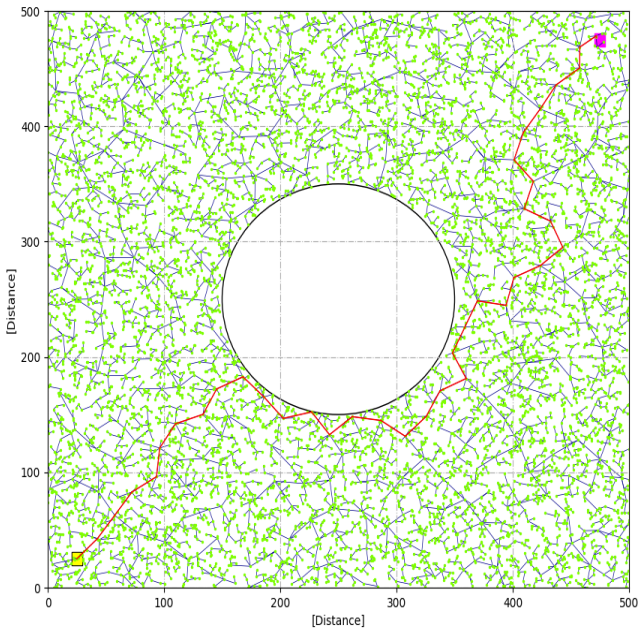

Figure 38: Simulation 2 RRT

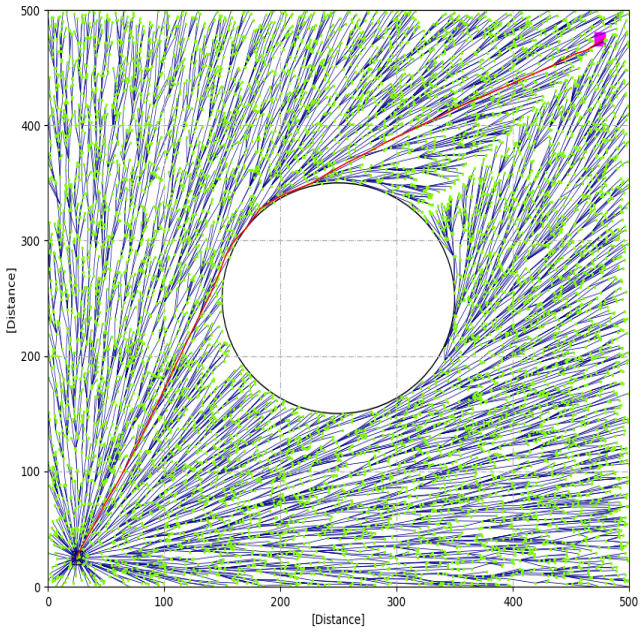

Figure 39: Simulation 2 RRT*

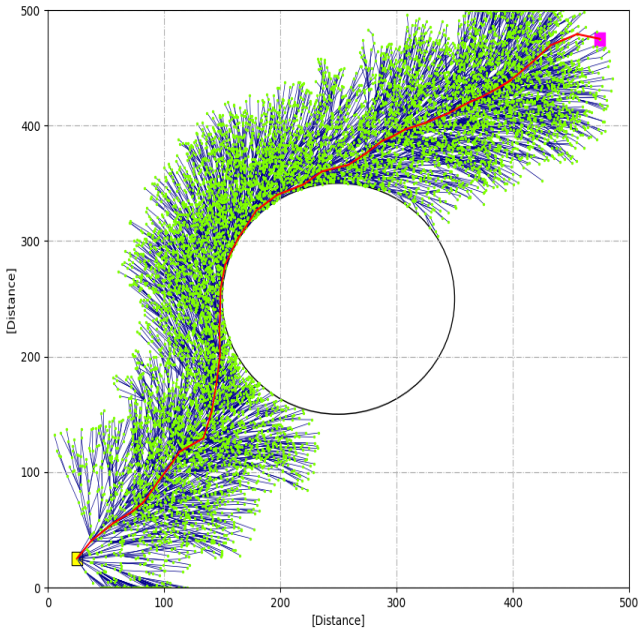

Figure 40: Simulation 2 MiniRRT

In the third simulation, more obstacles are added. This showcase the MiniRRT more greedy aspects as it takes the path along the long obstacle in the center instead of going around like the RRT*. This can also be seen in the costs of both RRT* and MiniRRT as the spread apart from each other. The results of these simulations can be seen in table 4. Plots for search time and path cost can be seen in figures 41 and 42. The figures for each algorithm can seen in figures 43, 44 and 45.

Algorithm	Search Time (seconds)	Cost (distance)	Graph Size (distance)
RRT	0.719	186.992	100 x 100
RRT*	2.093	137.923	100 x 100
MiniRRT	1.988	152.992	100 x 100
RRT	61.863	580.976	250 x 250
RRT*	113.722	343.512	250 x 250
MiniRRT	14.243	375.691	250 x 250
RRT	300.164	870.366	500 x 500
RRT*	457.433	671.566	500 x 500
MiniRRT	32.895	766.967	500 x 500
RRT	1152.624	1841.341	1000 x 1000
RRT*	1827.315	1342.984	1000 x 1000
MiniRRT	65.367	1493.463	1000 x 1000

Table 4: Simulation 3 Comparison, 3 Obstacles

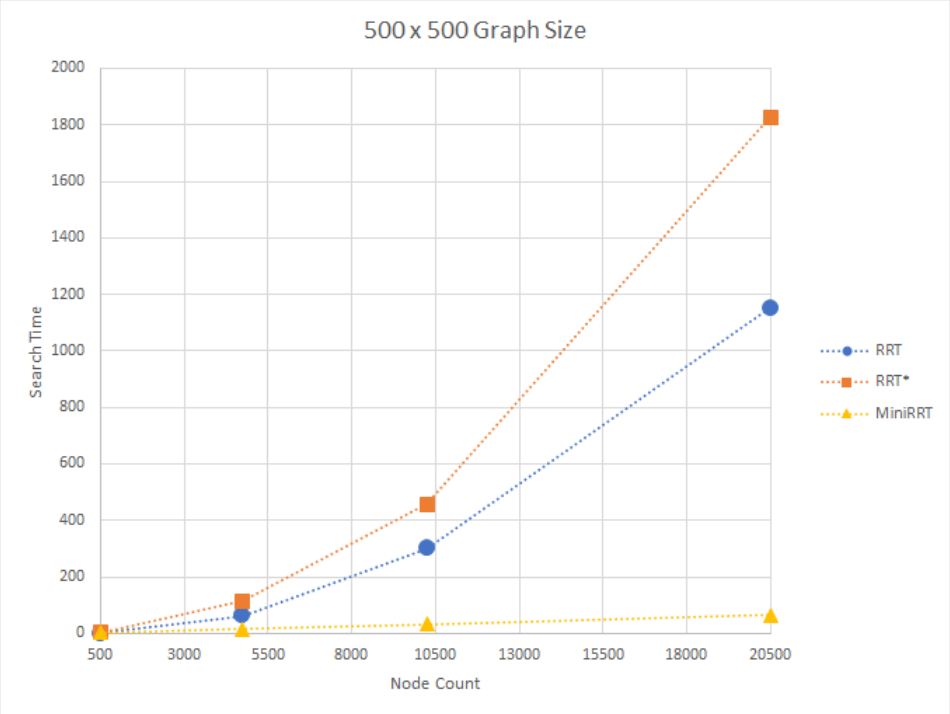

Figure 41: Simulation 3 plot comparison with search time

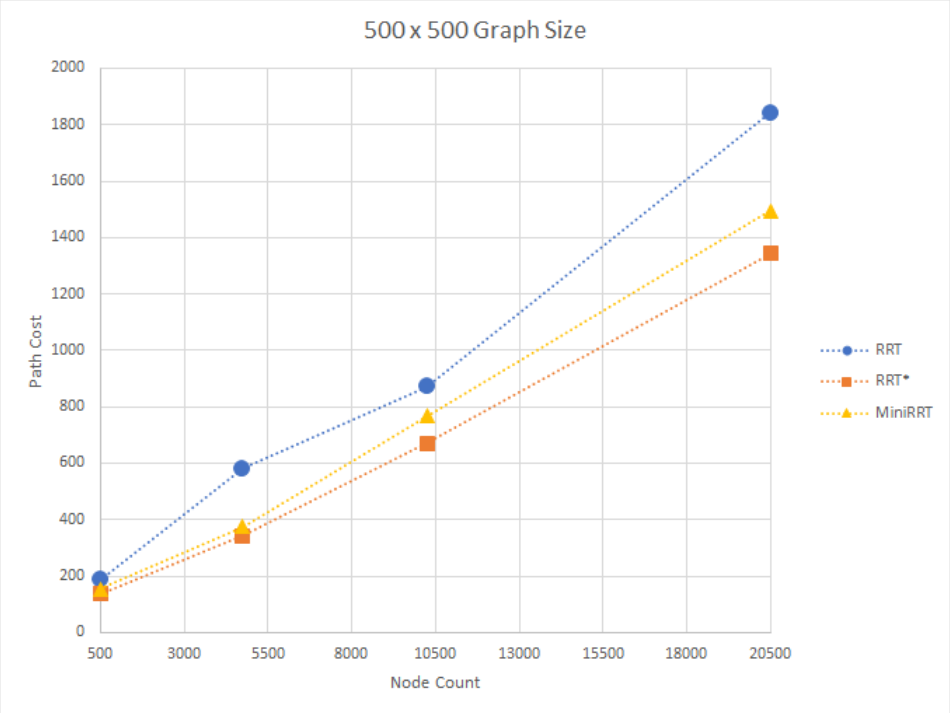

Figure 42: Simulation 3 plot comparison with path cost

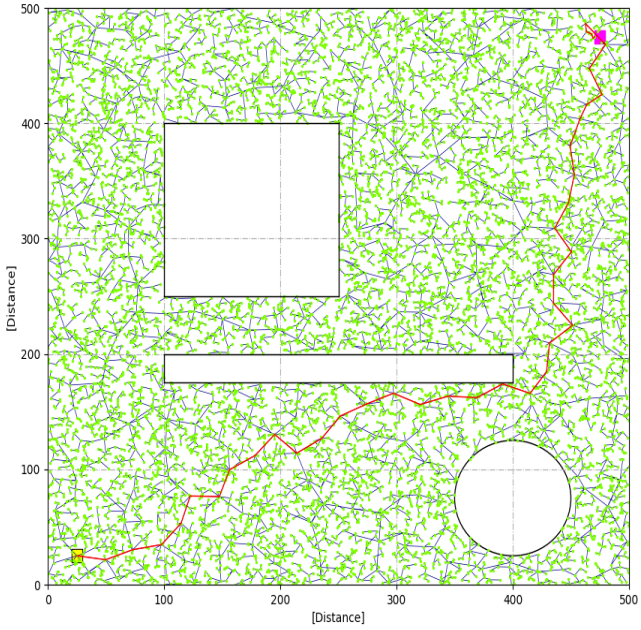

Figure 43: Simulation 3 RRT

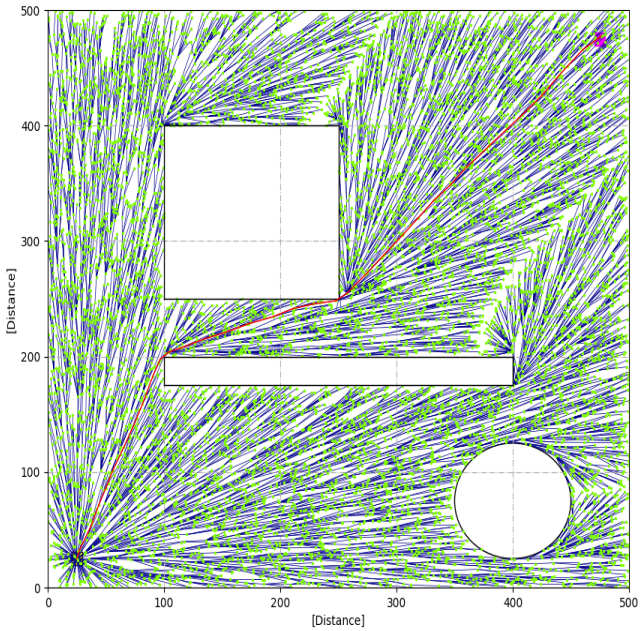

Figure 44: Simulation 3 RRT*

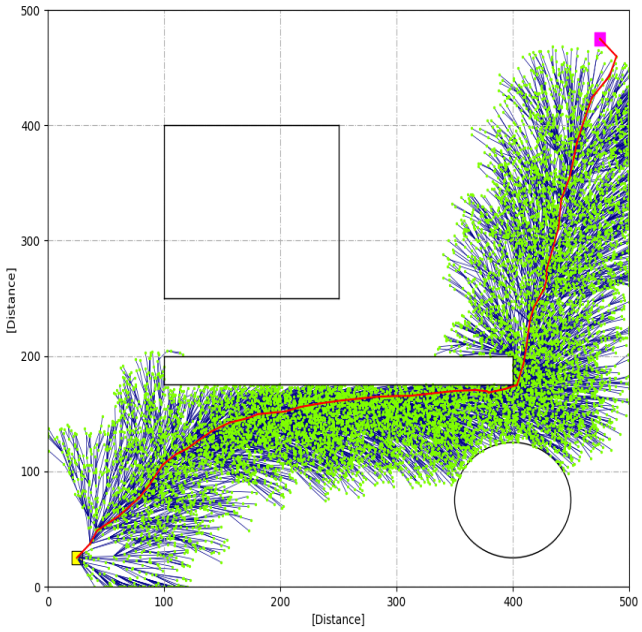

Figure 45: Simulation 3 MiniRRT

The final and fourth comparative simulations added two more obstacles. The trend between each algorithm continues. The results for this simulation can be seen in table 5 and plots for search time and path cost can be seen in figures 46 and 47. Figures showing the search trees can be seen in 48, 49 and 50. This simulation took the longest due to the node count that the MiniRRT had which was used for the other algorithms to create a full solutions. The RRT* algorithm took up to thirty minutes to complete, which is not ideal for UAVs.

Algorithm	Search Time (seconds)	Cost (distance)	Graph Size (distance)
RRT	3.39	196.175	100 x 100
RRT*	9.659	139.101	100 x 100
MiniRRT	5.471	163.602	100 x 100
RRT	52.929	417.228	250 x 250
RRT*	97.674	351.325	250 x 250
MiniRRT	18.593	376.657	250 x 250
RRT	273.358	929.361	500 x 500
RRT*	419.621	692.284	500 x 500
MiniRRT	42.349	734.959	500 x 500
RRT	1522.046	1902.634	1000 x 1000
RRT*	2139.151	1391.988	1000 x 1000
MiniRRT	77.961	1527.014	1000 x 1000

Table 5: Simulation 4 Comparison, 5 Obstacles

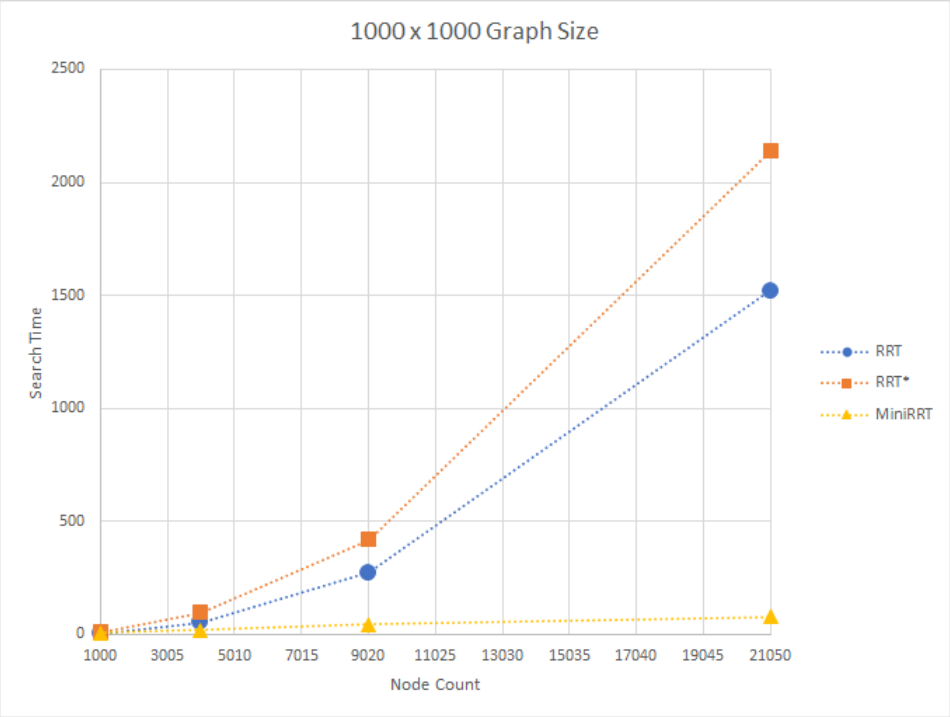

Figure 46: Simulation 4 plot comparison with search time

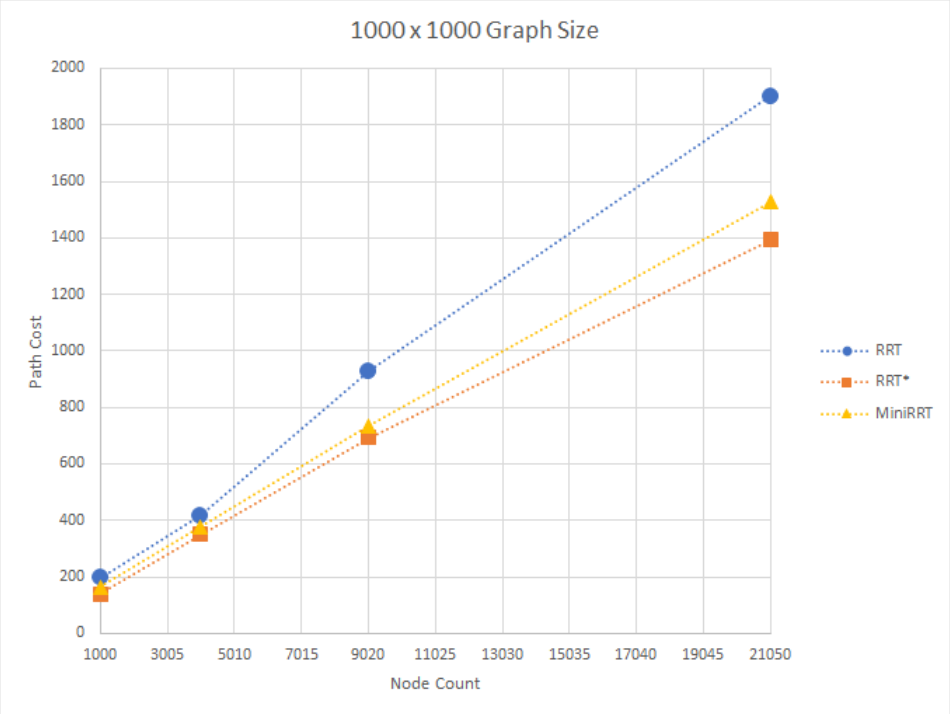

Figure 47: Simulation 4 plot comparison with path cost

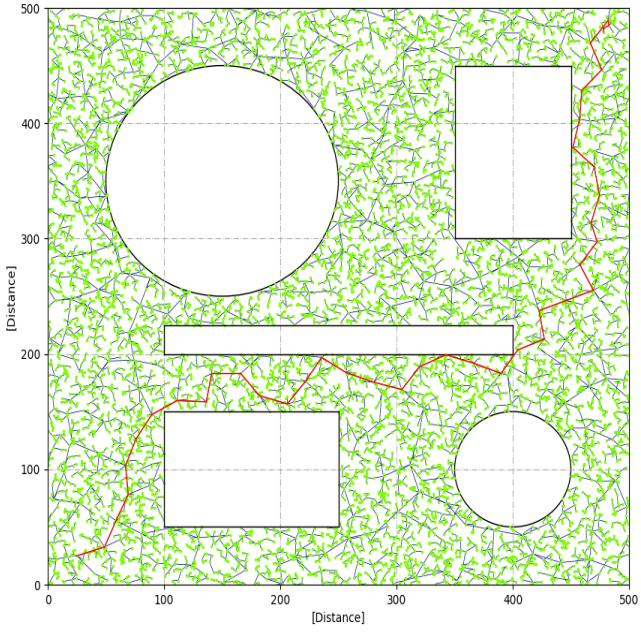

Figure 48: Simulation 4 RRT

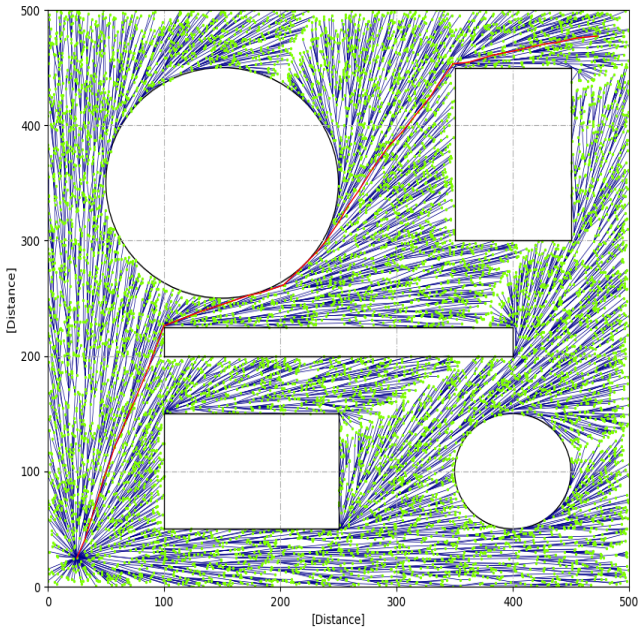

Figure 49: Simulation 4 RRT*

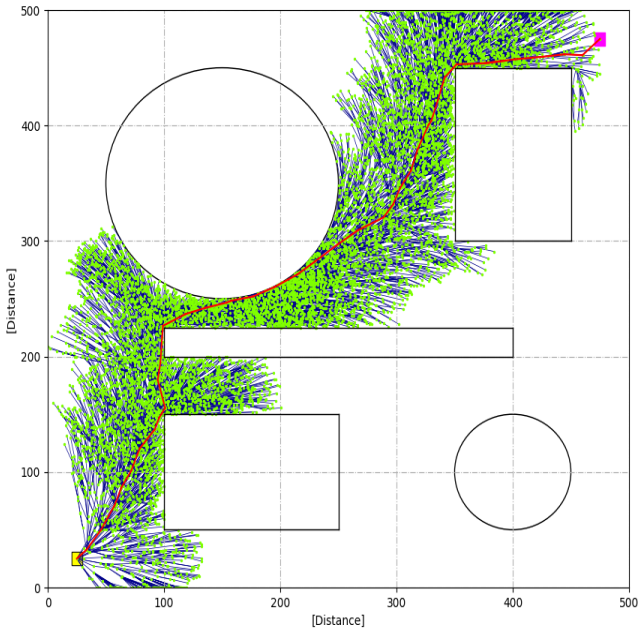

Figure 50: Simulation 4 MiniRRT

The overall trend of each simulation the cost and search time for each graph increased due to the delta parameter staying the same. The delta parameter controls how far apart each node in the search tree is from each other so the tree is able to quickly search the smaller graphs but it is the opposite for the larger graphs. The search time for each algorithm differs however, the MiniRRT search time is significantly less with the larger graphs. This is due to the fact that it only searches locally. The MiniRRT only has to parse through the max nodes for each local search tree, while both RRT and RRT* have to parse through nodes throughout the entire search radius presented in the shrinking ball radius routine this could potentially take up a large area of the graph. However shown in the simulations the MiniRRT algorithm doesn't generate the least cost path nor a fully optimized one. As seen in figure 45 the greediness of the algorithm is shown as it travels along the long obstacle. This is do to the use of the best leaf in the search tree used as the root node for the next tree. Another downside to MiniRRT is that due to the local search methods maps crowded with obstacles has the potential of producing and non-optimal solutions.

5.2 Software-in-the-loop

A few SITL simulations were done with the MiniRRT algorithm were done using a fixed-wing aircraft. The graph size for the simulations was 3000 $\times$ 3000 with the MiniRRT's parameters being seen in table 6. So The first simulation done was a open environment, meaning it had no obstacles in it. Figure 51 shows the simulations on Mission Planner with an overlay of the output of the algorithm's generated paths. The generated paths are the appendages off of the blue line while the path solution is the solid blue line. The red line is the aircraft's position in respect to the algorithm's perspective while the purple line is Mission Planner's perspective on the aircraft's position.

Parameter	Number
n	250
delta	100
r	400
alpha	120°

Table 6: MiniRRT SITL Parameters

Figure 51: Snapshot of the end of the SITL simulation 1 on Mission Planner

The next simulation has a large circular obstacle in the center of the graph. The MiniRRT algorithm generates paths that lead directly to the obstacle, but once it gets close enough to it paths start to diverge to a path around that obstacle. The figure for this simulation can be seen in figures 52.

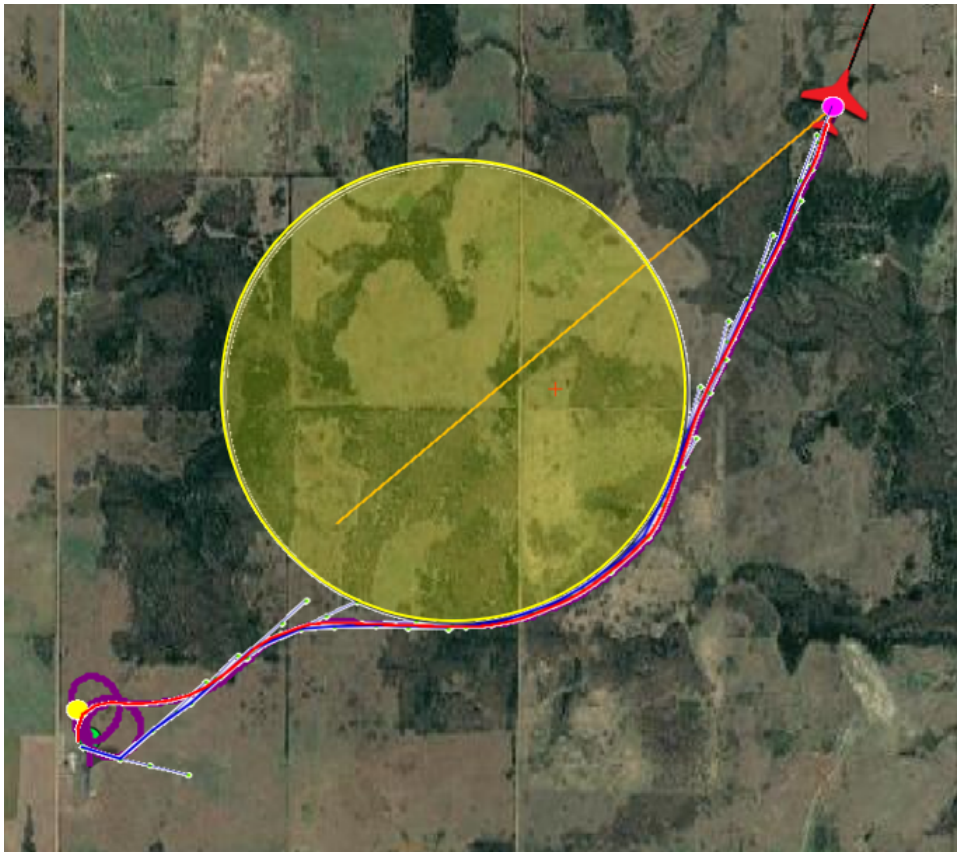

Figure 52: Snapshot of the end of the SITL simulation 2 on Mission Planner

The third and final simulation done uses a multitude of smaller circular obstacles. This was the more interesting simulations as it would produced different solutions around the numerous obstacles. One of the simulation results can be seen in figure 53. The other results can be seen in the Appendix.

Figure 53: Snapshot of the end of the SITL simulation 3 on Mission Planner

In the SITL simulations some simulations done had the fixed-wing aircraft turn around and end up going into the obstacle zones, which is something that is wanted to be prevented. Each simulation started with the fixed-wing in a loiter pattern above the take-off point. The initial tree would be planted at the position of the aircraft when it started. Making the aircraft turn around to hit waypoints generated behind it.

5.3 Flight Tests

5.3.1 Fixed Wing Flights

Due to the PixRacer that the Nano Talon was outfitted with, the first experimental flights ended with the Nano Talon loitering in one spot. The reason that it loitered was due to the PixRacer not being compatible with MAVLink. This resulted in it not being able to actuate

the servos on the aircraft to track towards the waypoints that the algorithm had placed on the graph. With the PixCube this is possible and in similar tests. The interface between the PixCube and the rest of the system was the same as the PixRacer. When the transfer to the PixCube Black happened the pixhawk was not able to receive proper GPS from the GPS chip. This is later swapped out for a GPS puck which allowed for data to be streamed between the Ardupilot and the LattePanda. This setup for the GPS puck can be seen in figure 54. A cutout on top was necessary due to the cable to connect the GPS to the carrier board was too long to be able to fit in the top compartment of the aircraft.

Figure 54: Top View of the Nano Talon

The Nano Talon was taken to the OSU Unmanned Aircraft Flight Station for additional testing. The first flight test was similar to the second SITL simulation in the aspect of a large obstacle being placed between the start point and the goal. However, this time around the graph size was reduced due to fear of having the aircraft be beyond the visual line of sight or BVLOS. The algorithm was able to place waypoints on a trajectory to the goal, however, the sight of the Nano Talon was being lost so the test was aborted. The result of the flight can be seen in figure 55. Where the algorithm generated a path past the forest but was returned manually so that the aircraft was not lost. The path generated path the algorithm can be seen in figure 56. The position data for the aircraft from the algorithm perspective is also shown in red. While the path calculated is in blue. Both lines cross over the obstacle zone for the reason that, the calculated paths use a spline method to smooth the path. If the original tree wraps around an obstacle like it did the spline path ends up slight crossing over the obstacle zone. The red line ends up crossing over the obstacle zone as the aircraft is close enough to the waypoints for it to get counted. To mitigate these issues, a tighter waypoint radius can be defined so that the aircraft doesn't stray too far from the path. For the calculated path, a spline with more waypoints can give the path more definition so that it does not intersect the obstacle zone. The flight data for the tests can be seen in figure 57. The throttle was set to 50% when in guided mode for tracking waypoints.

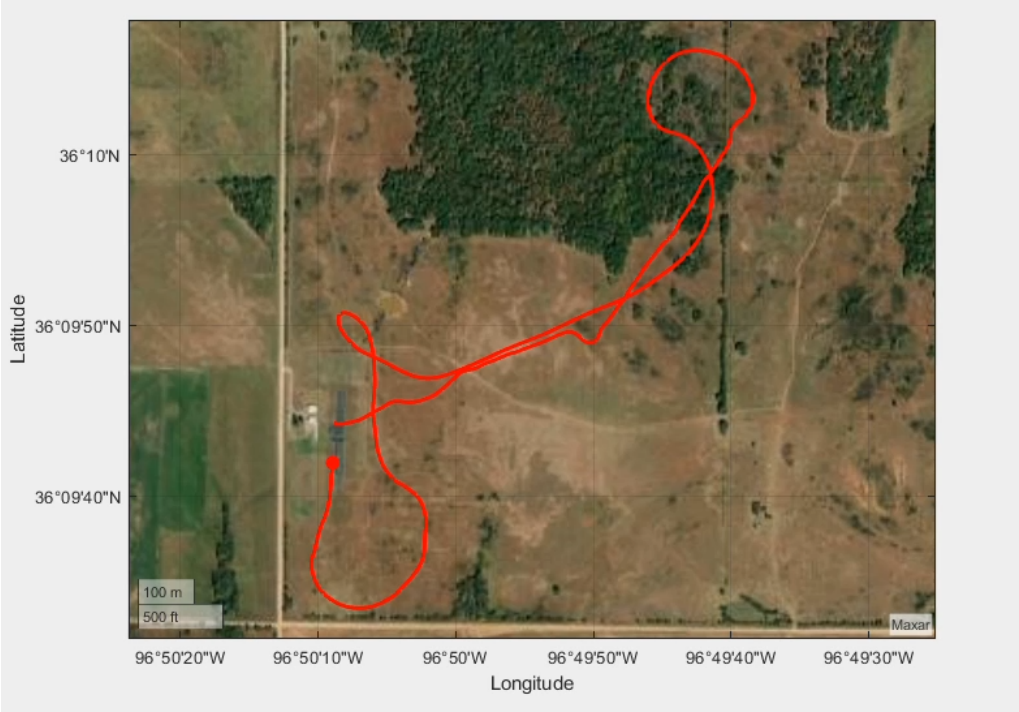

Figure 55: Nano Talon Flight Test 1 result

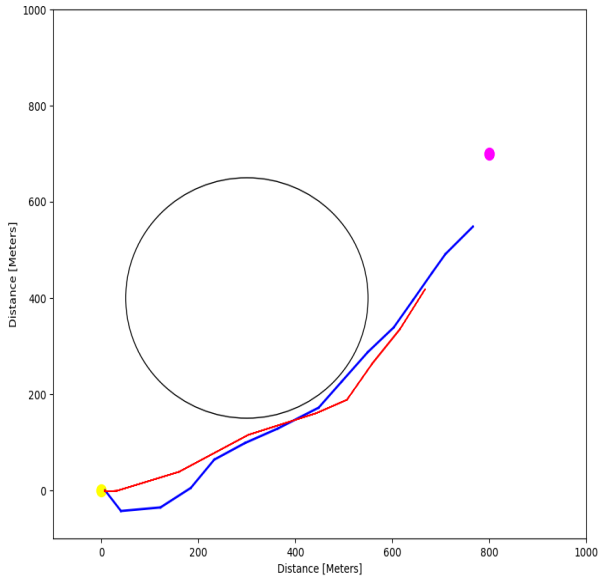

Figure 56: Nano Talon Flight Test 1 algorithm result

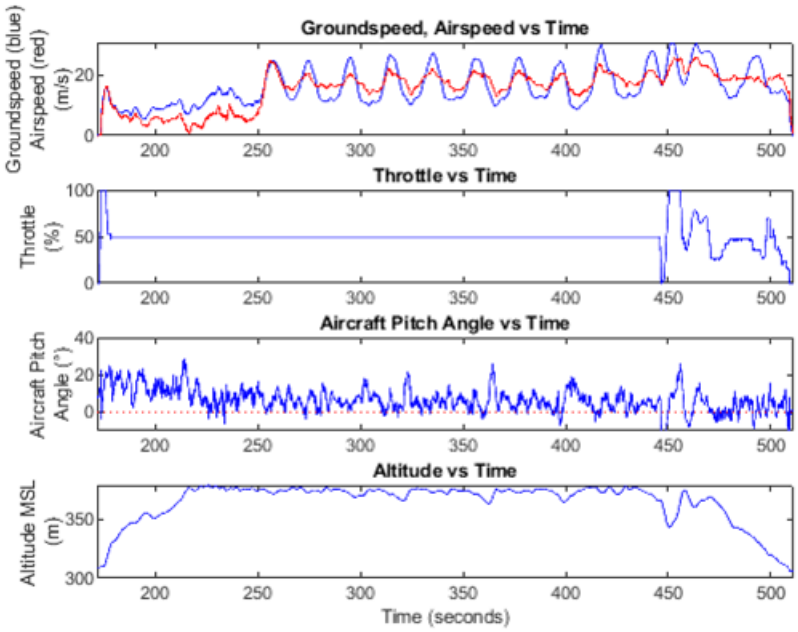

Figure 57: Nano Talon Flight Data for test 1

A second test was done with a reduced scale map. This test was slightly more successful however, the criteria set to where the UAV flipped goal points were not met, due to the UAV not being close enough to the first goal point to switch. The same things that happened with the previous test happened in this one as well. This flight was allowed to be prolonged due to sight of the aircraft was still possible. It loiters in place for roughly two minutes until the pilot took back command of the aircraft to bring it back safely. Figure 58 shows the output of the pixhawk position data in red and the generated path from the algorithm in blue. The flight data for the tests can be seen in figure 59.

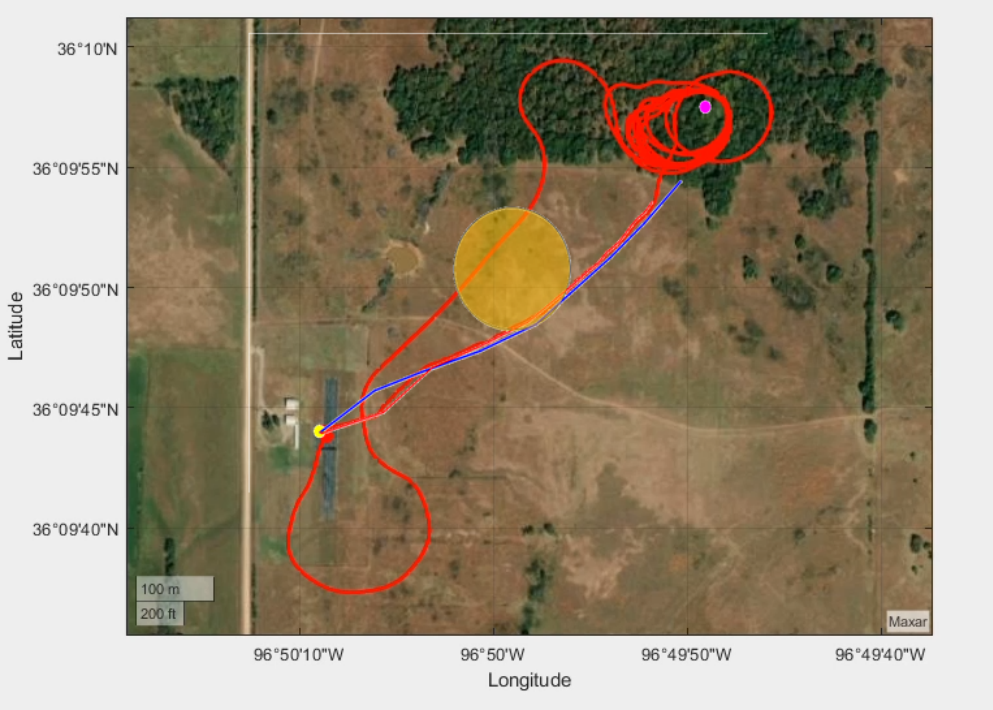

Figure 58: Nano Talon Flight Test 2 result

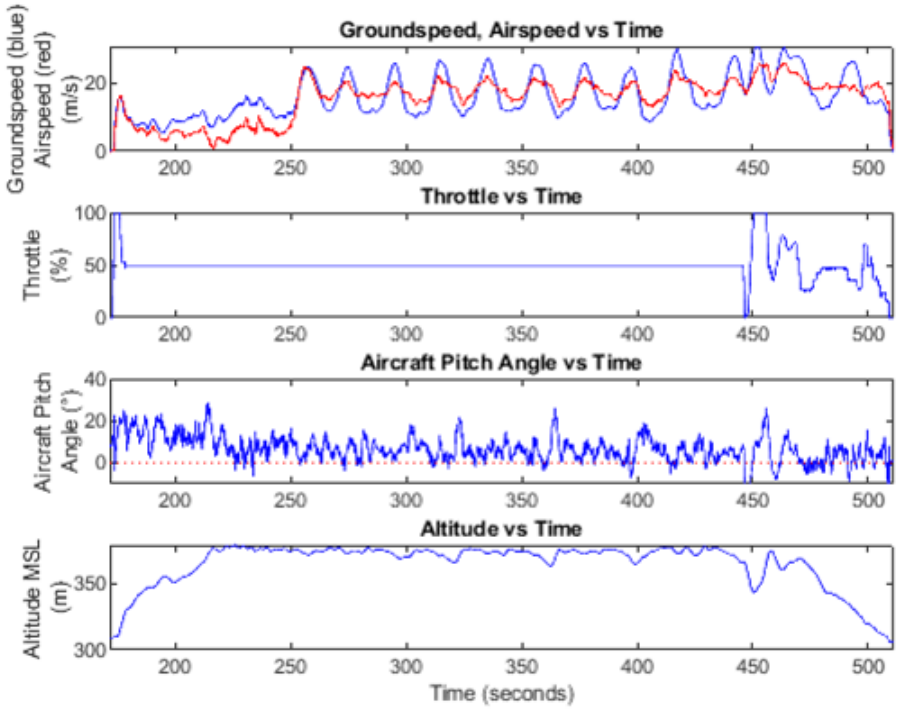

Figure 59: Nano Talon Flight Data for test 2

The nano talon was a good pick to test the algorithm as it was a cheap aircraft that was able to be outfitted with all the proper hardware. Both tests had paths being generated that led the aircraft toward the goal but not quite reaching the goal due to the last path segment not being pushed to the Ardupilot to take it there. The algorithm was set up for the aircraft to find its way back, but due to those issues, it never reached that point.

5.3.2 Multi-rotor Flights

Multiple tests were done with the Eagle. The eagle that flew the mission can be seen in figure 60. The first flight needed an abort due to the map not having a proper obstacle placement as it got too close to the center pole in the field. The second flight test was a partial success however, the test was halted due to the obstacle zone being too big for the algorithm to plan around. The obstacle zone itself reached the edge of the graph so there was no way around it, so the algorithm got stuck in this cavity. The results of this test can be seen in figure 61. The flight data for the Eagle can be seen in figure 62. The Airspeed measure is not present as the Eagle did not have one opposed to the Nano Talon. The ground speed, however, gives us an indication of when the Eagle hovered in place while the algorithm was calculating a path. When these moments happened the ground speed decreased down to zero. For this test, the Eagle hovered at roughly 180 seconds and 220 seconds then got stuck at around 260 seconds.

Figure 60: The Eagle multi-rotor UAV at the soccer field

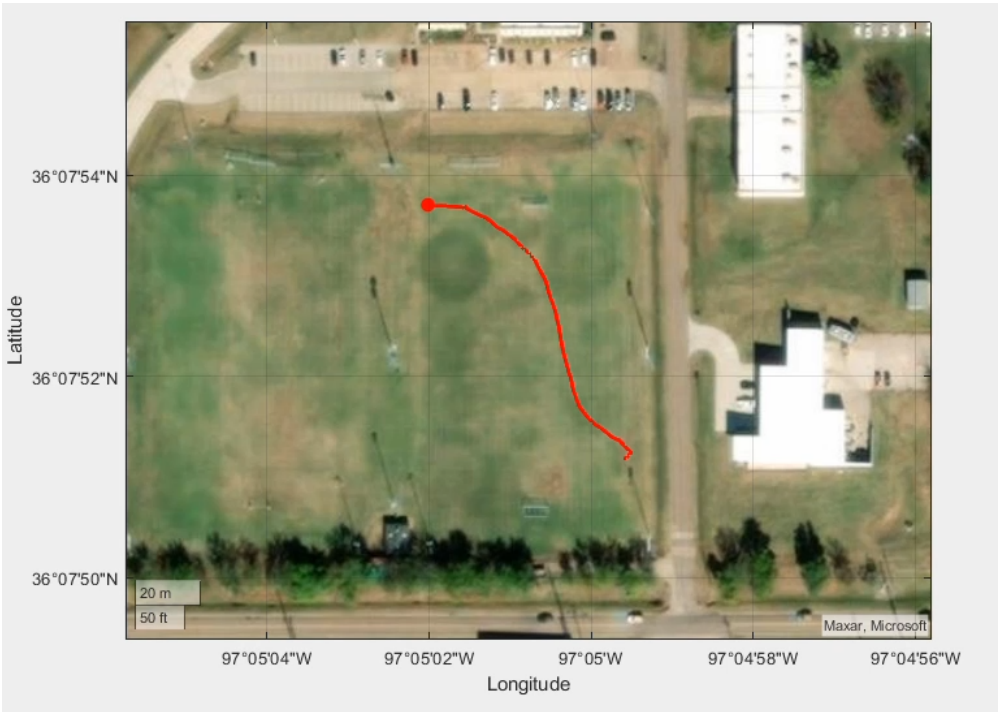

Figure 61: Eagle Flight Test 2 result

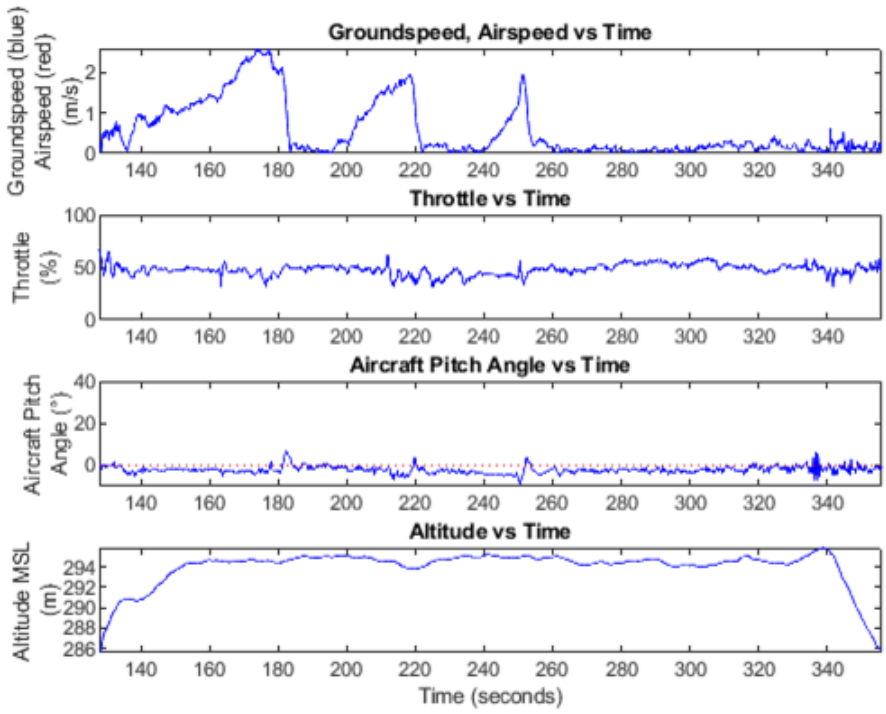

Figure 62: Eagle Flight Data for test 2

The third test was the first fully successful run. The results can be seen in figure 63. From them, it is shown that the Eagle was able to complete its mission without pilot intervention when traveling to the goal point and back. Notice that there are small swirls within the trail line. These were a cause of the Eagle hovering in place waiting for further waypoints to be generated, which was thought to be a tuning issue as this multi-rotor was known for its finicky controller. The flight data for the third test can be seen in figure 64. With noting groundspeed there is an increased frequency of hovering and tracking waypoints as the Eagle plans its path. The fourth flight was a repeat of the third apart from a resize of the obstacle zone however, it produced a slightly different result. This flight was a repeat due to the algorithm not producing the proper logs to plot the generated paths this flight had that same issue. But in this flight, the UAV sat at a location near the goal due to it being the last waypoint pushed to ArduPilot and, it was not getting close enough to the goal to continue the mission. So the pilot in command moved the UAV closer to the goal for

that goal switch to happen, and the algorithm continued to generate a path to the launch location. The result can be seen in figure 65 and flight data can be seen in figure 66. The groundspeed looks similar to the previous test however, at 200 seconds the path was finished however the goal switch criteria were not met yet. At around 280 seconds the goal switch criteria were met when the pilot nudged the Eagle closer to the goal.

Figure 63: Eagle Flight Test 3 result

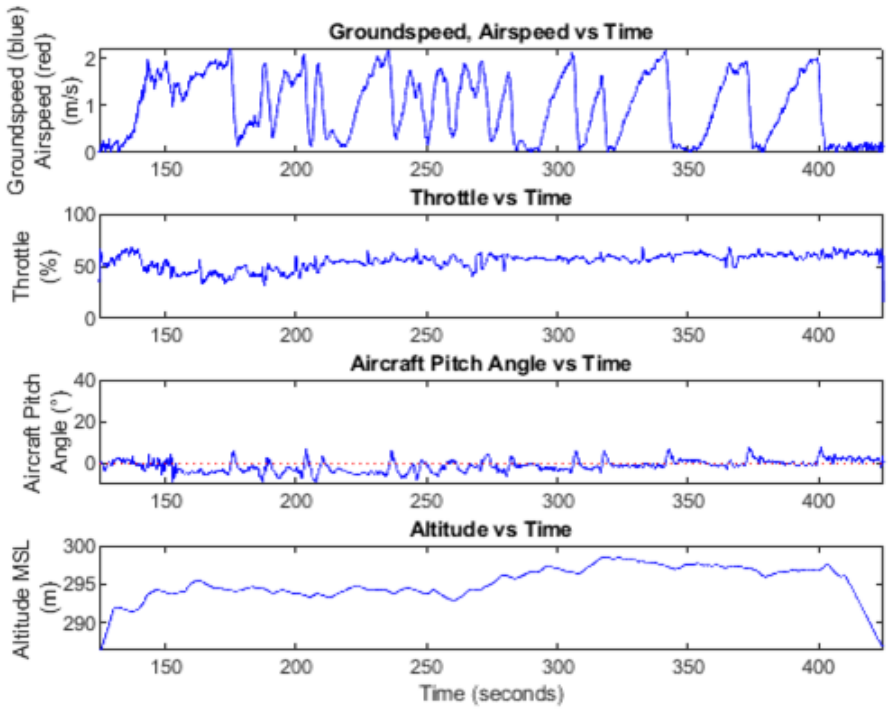

Figure 64: Eagle Flight Data for test 3

Figure 65: Eagle Flight Test 4 result

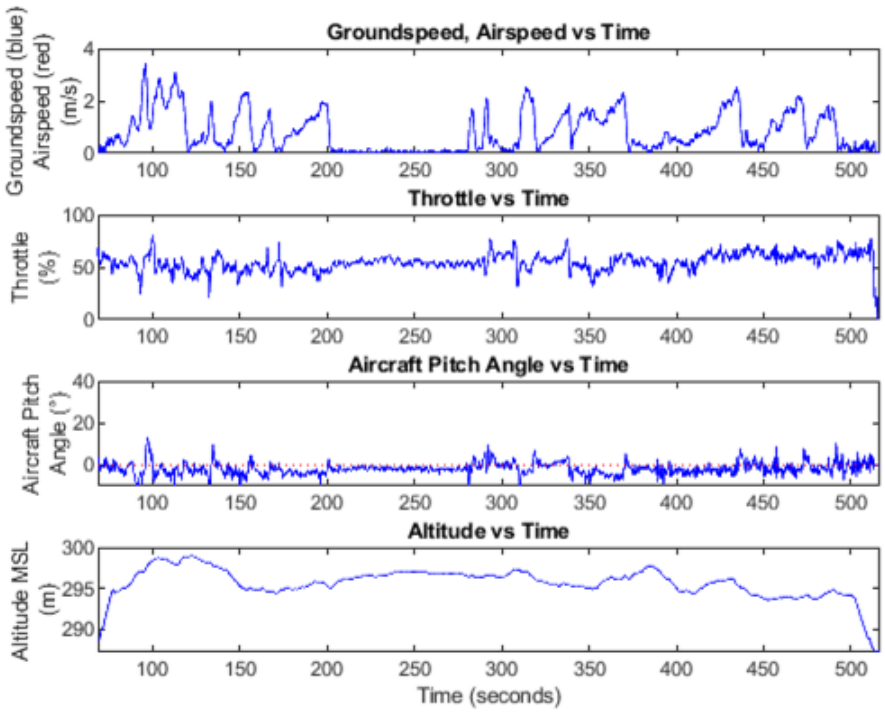

Figure 66: Eagle Flight Data for test 4

In the fifth test, The obstacle was shifted left to be more centered on the light pole in the center of the soccer field as by this time start point was fixed and it was easy to create a map around that. The results of this test can be seen in figure 67. This flight test had a similar issue to the fourth flight test as it was not close enough to the goal to switch it back to the launch point and have the algorithm generate paths there. However, this time around log files were produced from the algorithms side so, the output of that can be seen in figure 67. In the figure, the red line is the Eagle's position data from Pixhawk, the blue line is the generate path by the algorithm, the bright yellow dot is the start point, the purple dots are the goal points, and the gold area is the obstacle. With flight data shown in figure 68 the Eagle was able to get a good amount of the path calculated until 200 seconds. At around 250 seconds the Eagle hovered at the goal position without switch the goal back to home.

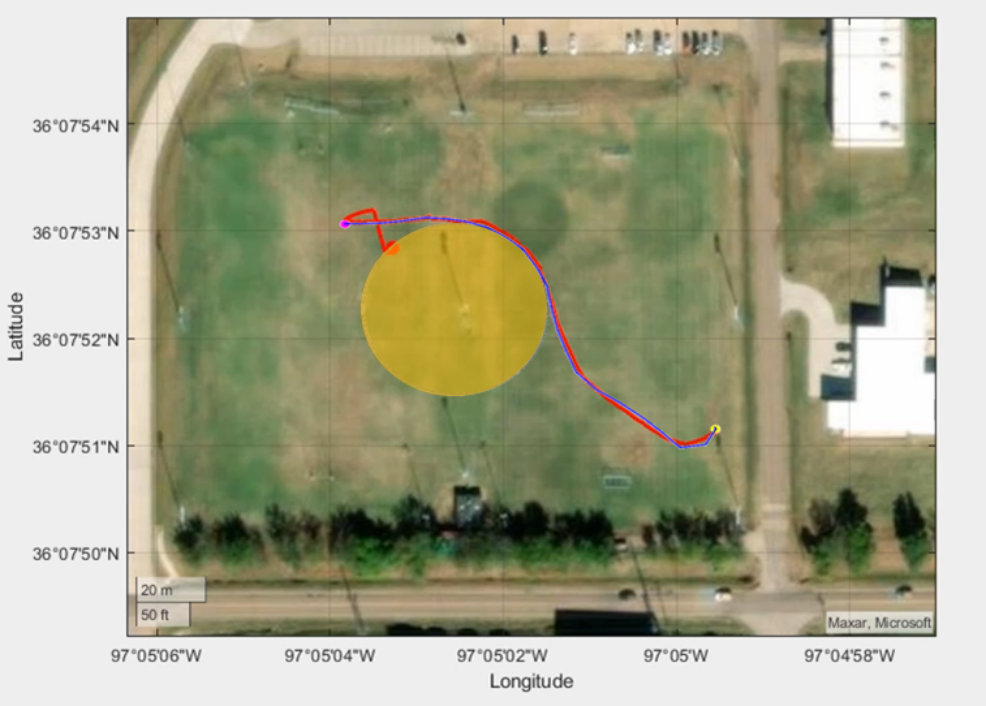

Figure 67: Eagle Flight Test 5 result

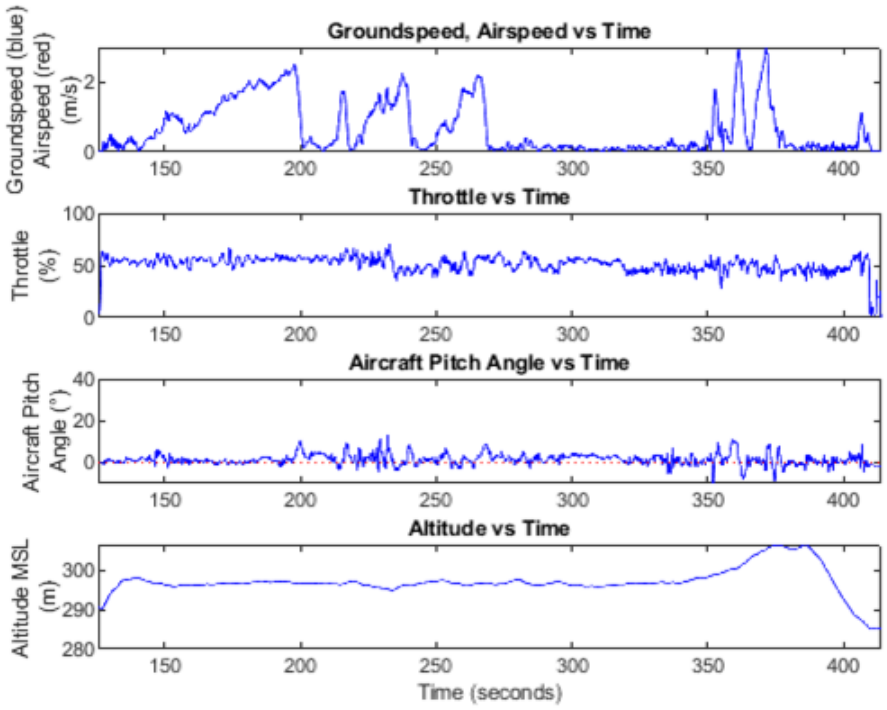

Figure 68: Eagle Flight Data for test 5

The sixth test was done to have a complete mission of the fifth test results however, the algorithm crashed right before a full log output was done. So only half the flight path was able to be plotted. The results can be seen in figures 69. The initial takeoff of this flight test was evident that tuning issues were not the only problem for the Eagle's unsteadiness while hovering. This can be seen in figure 69 with the circle near the takeoff point. The flight data for this test can be seen in figure 70. The third row of the flight data showcases the stability of the Eagle concerning pitch. A quick inspection proved that an additional O-ring was present in the interface of the propeller and the motor. This made the propeller wiggle enough for behavior seen. This part was removed and, another test was done with an additional goal point.

Figure 69: Eagle Flight Test 6 result

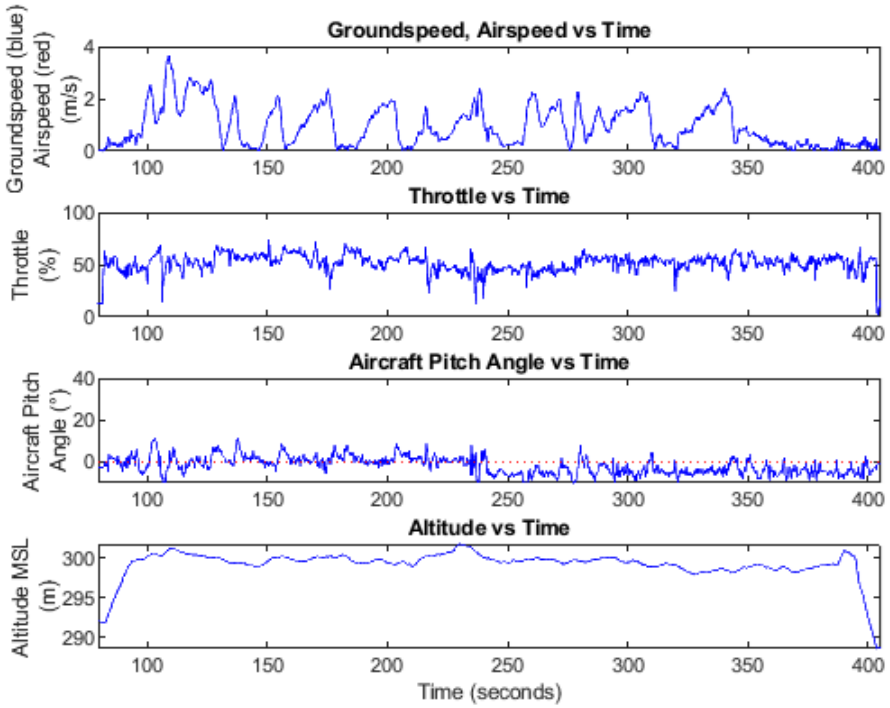

Figure 70: Eagle Flight Data for test 6

The seventh flight test was the last flight of the day as it was getting late into the evening civil twilight was approaching so, due to FAA regulations, more tests could not continue afterward. For this test, the addition of a goal point was placed north of the launch point. The mission for the Eagle was to go to the goal point north then travel east to the second goal point then return home. As seen in figure 71 the eagle did just that all while steering clear of the obstacle zone in the center. The flight data can be seen in figure 72. The stability of the aircraft was better than that of the previous test based off the aircraft pitch angle row, but was still not the best.

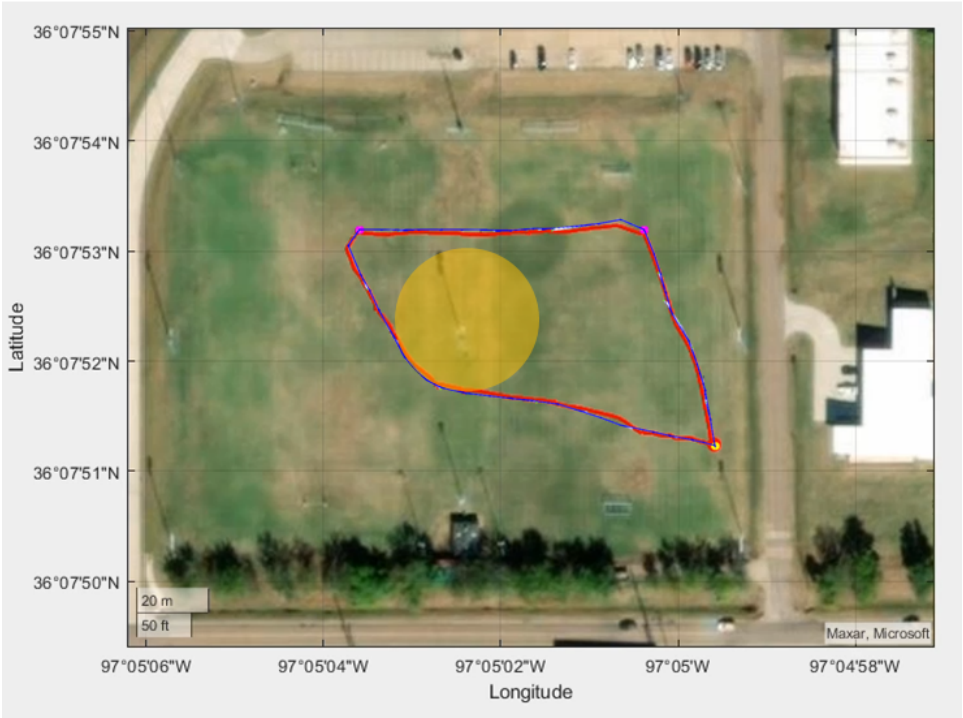

Figure 71: Eagle Flight Test 7 result

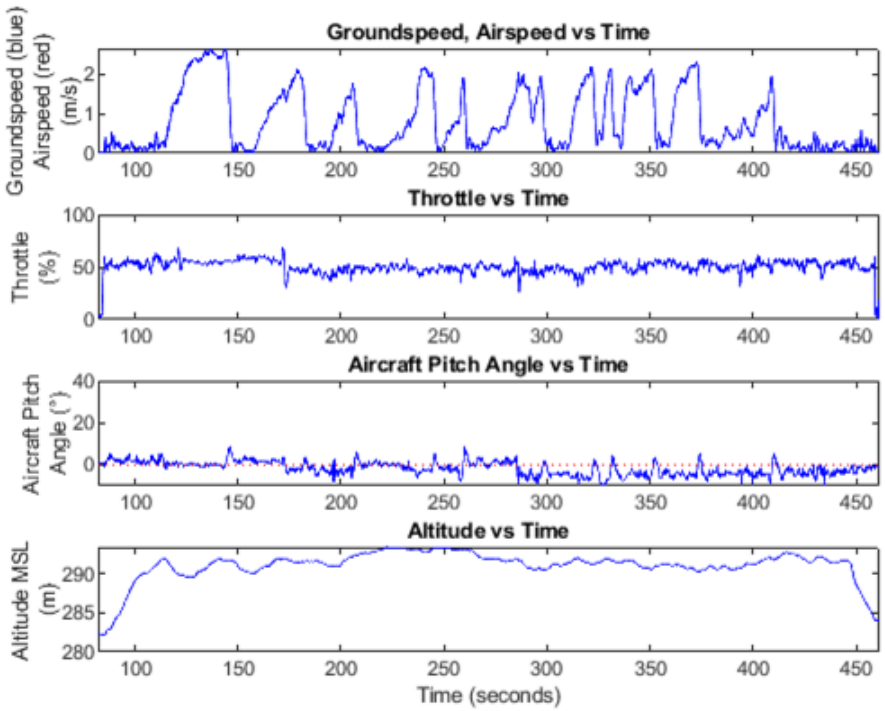

Figure 72: Eagle Flight Data for test 7

More tests were done that included different obstacle shapes and multiple obstacles. However, due to a malfunction, the Eagle was not able to properly fly as it would go out of control from the remote pilot and fly forward. A new Eagle was used for the tests. It utilizes the same hardware and software as the previous Eagle. A few extra results were logged through the algorithm like the entirety of the search trees and a run-time parameter. The first test with the new Eagle was again using the circular obstacle to make sure everything was working properly. Figure 73 shows the flight, and figure 74 shows the MiniRRT's output. In the MiniRRT output, the red line is the overall path sent to the Eagle, the green points and dark blue lines represent the search trees, and the yellow point and purple point are the start and goal locations, respectively. The obstacle outline is in black. Flight data can be seen in figure 75. This time around the frequency of hovering to tracking waypoints is less than previous tests.

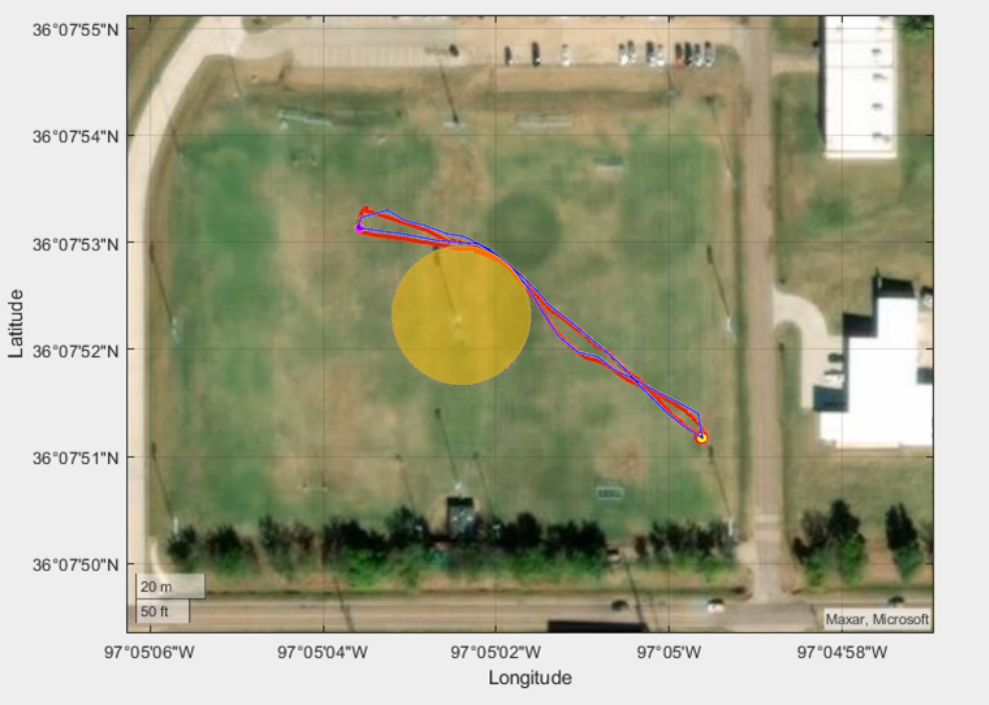

Figure 73: New Eagle Flight Test with a circular obstacle

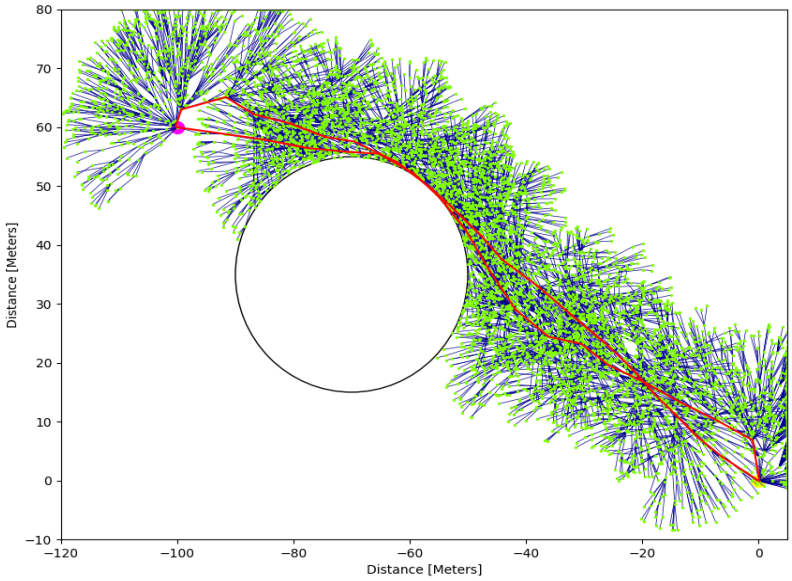

Figure 74: MiniRRT search with a circular obstacle

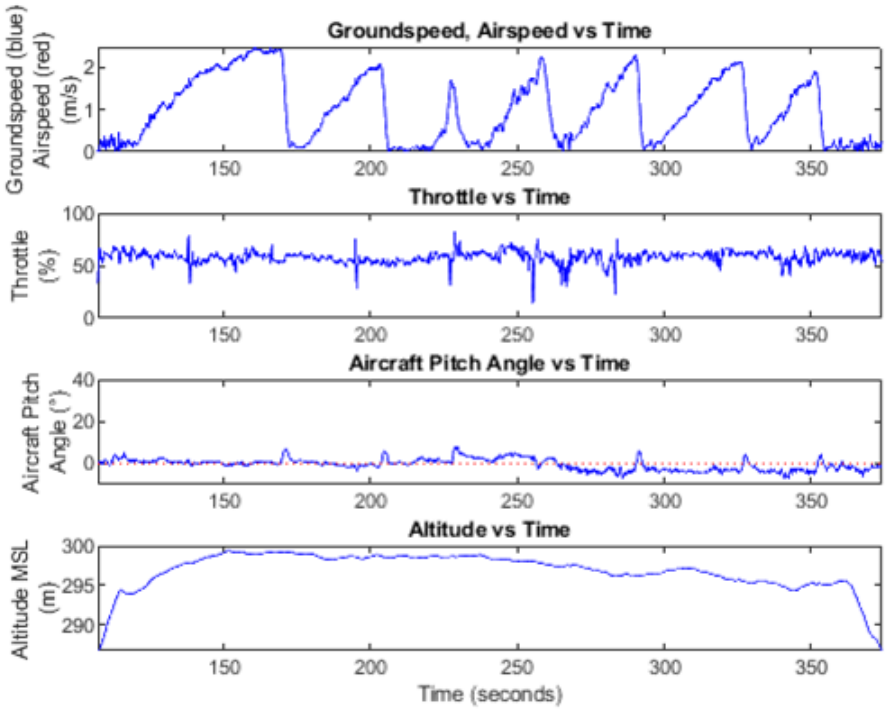

Figure 75: Eagle Flight Data for test with circular obstacle

The next test used a rectangular obstacle that extends further in length than it did width. This test was repeated three times. Where the UAV was able to get around the obstacle in all three of the tests, however on the return back to home. The UAV got stuck in one of the tests on the goal side of the obstacle. This is potentially due to the range at which MiniRRT can search. The result from the Pixhawk can be seen in figure 76, the flight data from the Pixhawk can be seen in figure 77. From figures 78 and 79 you can see the results of one of the tests. From them you can see that the path clips the corners of the obstacle, this is due to the b-spline function creating a smooth path from the tree. And, the search trees themselves don't intersect or are placed inside the obstacle. This tests flight data can be seen in figure 80.

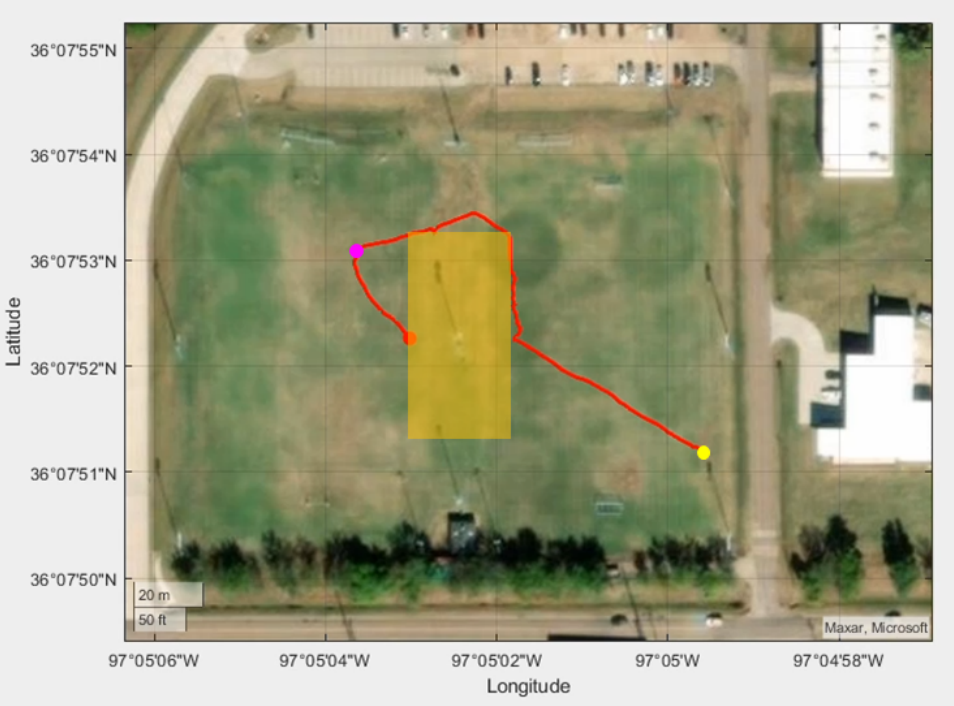

Figure 76: The Eagle getting stuck on the goal side of the rectangular obstacle

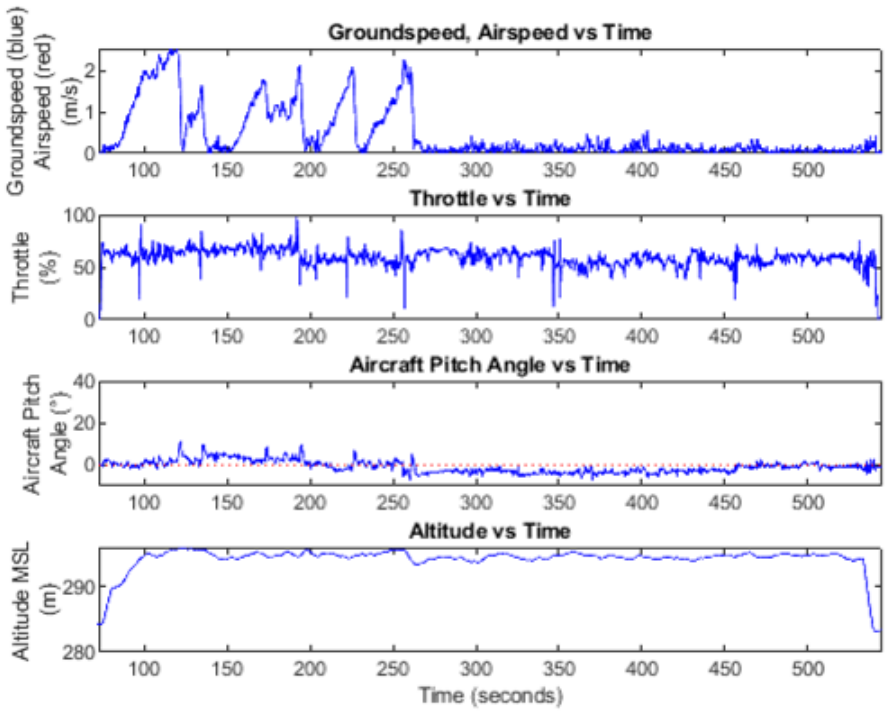

Figure 77: Eagle Flight Data with stuck Eagle and rectangular obstacle

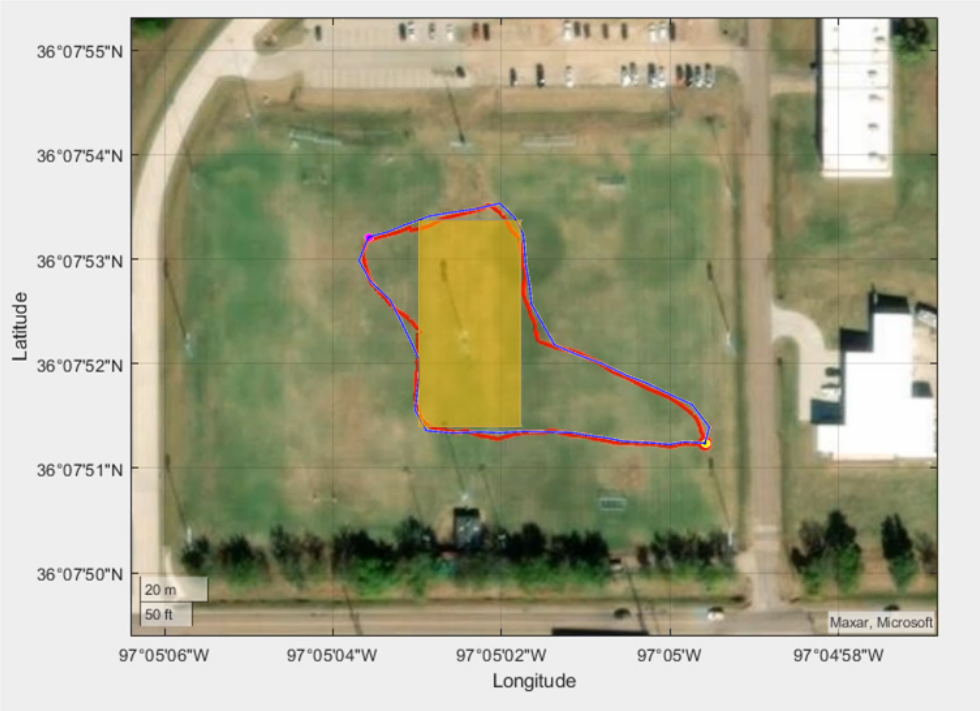

Figure 78: New Eagle Flight Test with a rectangular obstacle

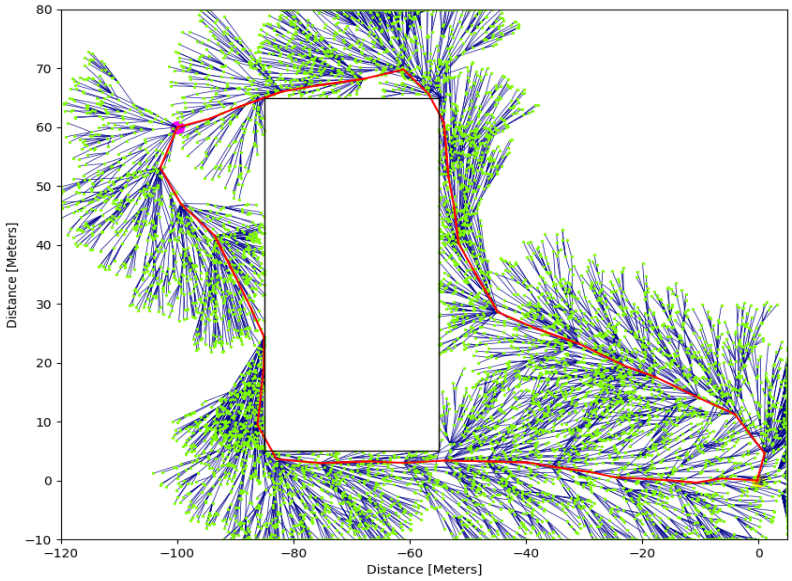

Figure 79: MiniRRT search with a rectangular obstacle

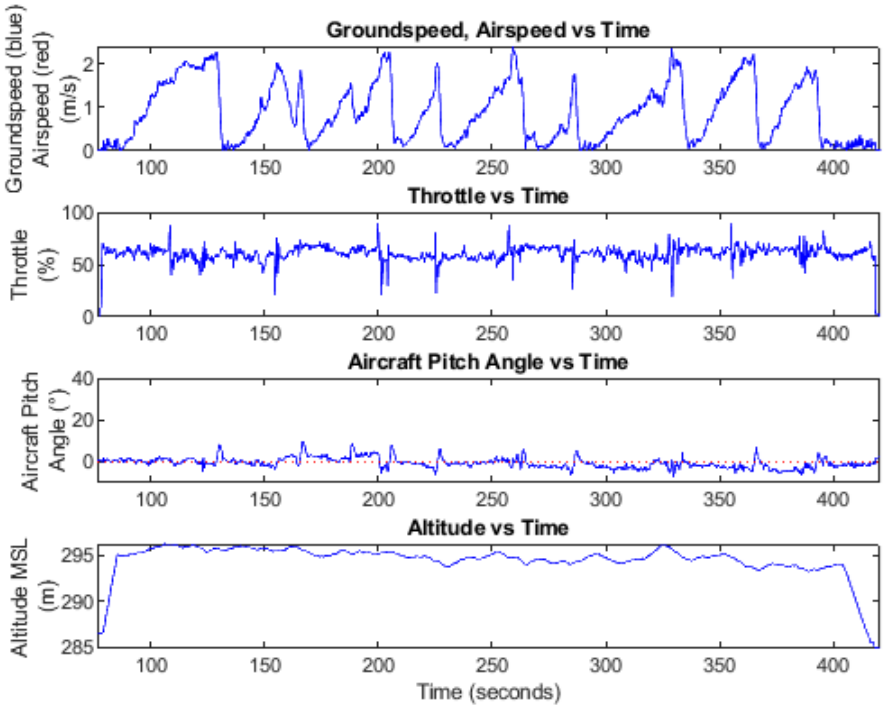

Figure 80: Eagle Flight Data for test with rectangular obstacle

The next obstacle that was used in the map was triangular. This test was repeated four times. On the first two tests, the UAV got stuck on the goal side of the triangle. Figure 81 shows this happening in one of the tests using the UAV's position data and the obstacle from the graph. The flight data shown in figure 82 shows this at around 250 seconds, where the Eagle hovers in place. To fix the UAV from getting stuck the range that the MiniRRT was able to search was increased from 120 degrees to 240 degrees. In the next two tests, the Eagle was able to find a path around the obstacle. The results can be seen in figures 84 and 83. The algorithm still clips the obstacle due to the b-spline function for smoothing the path. The UAV itself also cuts the corner as the UAV was close enough to the waypoints to start tracking the next one. The MiniRRT's output for both the last two tests can be seen in figures 85 and 86. Both flight data sets for each test can be seen in figures 87 and 87.

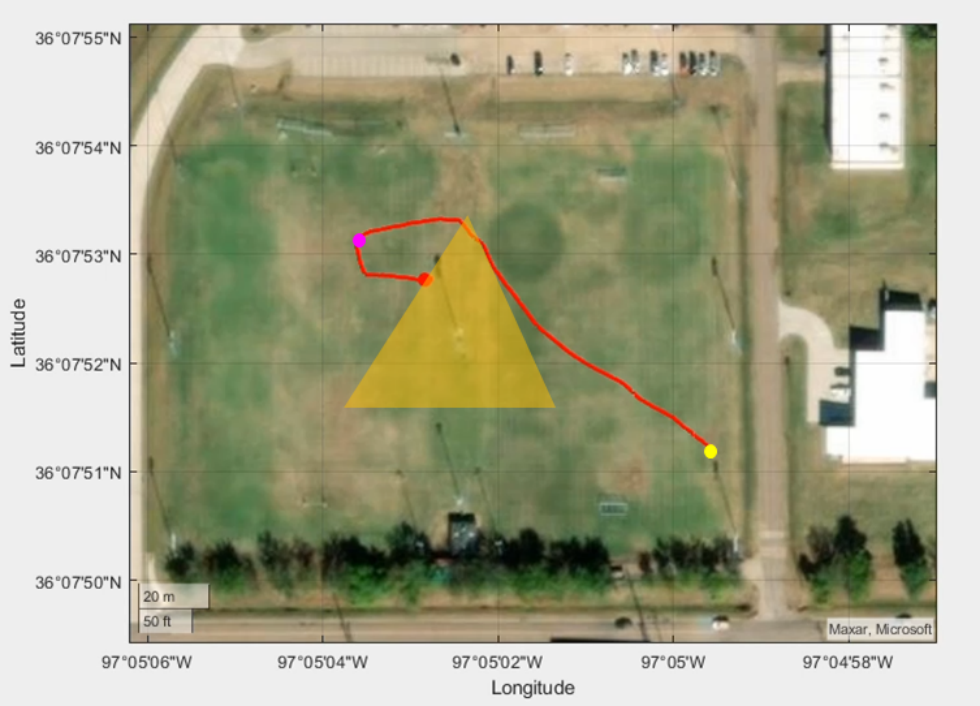

Figure 81: The Eagle getting stuck on the goal side of the triangular obstacle

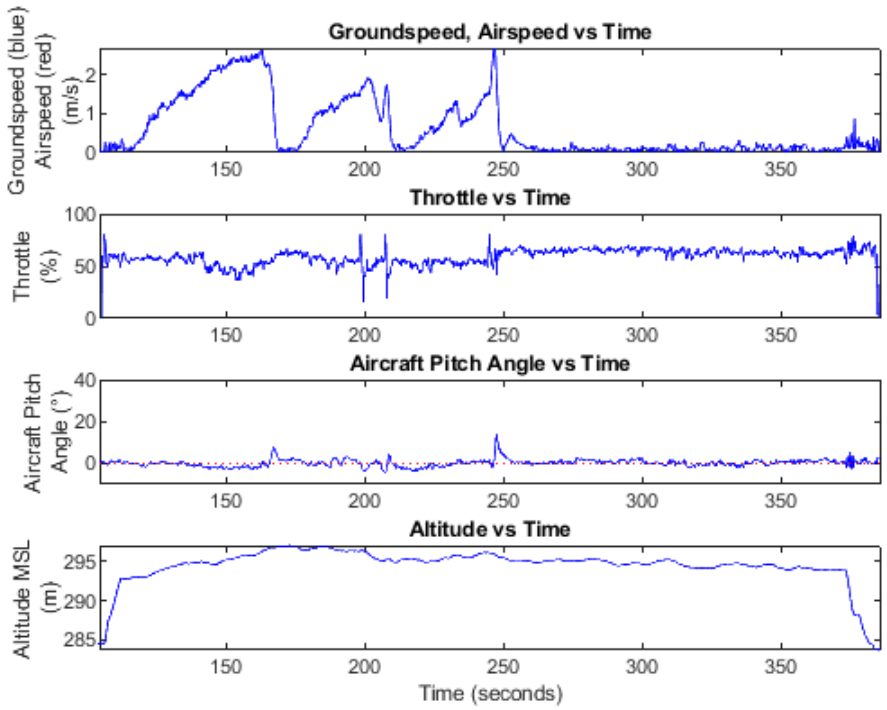

Figure 82: Eagle Flight Data for first test with triangular obstacle

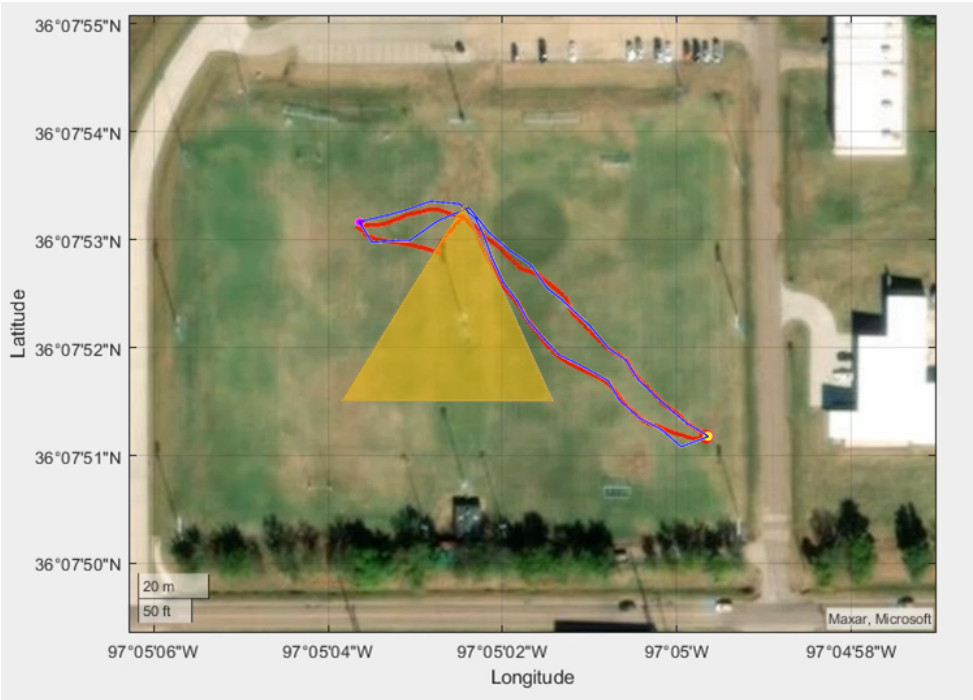

Figure 83: The fourth Eagle Flight test with a triangular obstacle

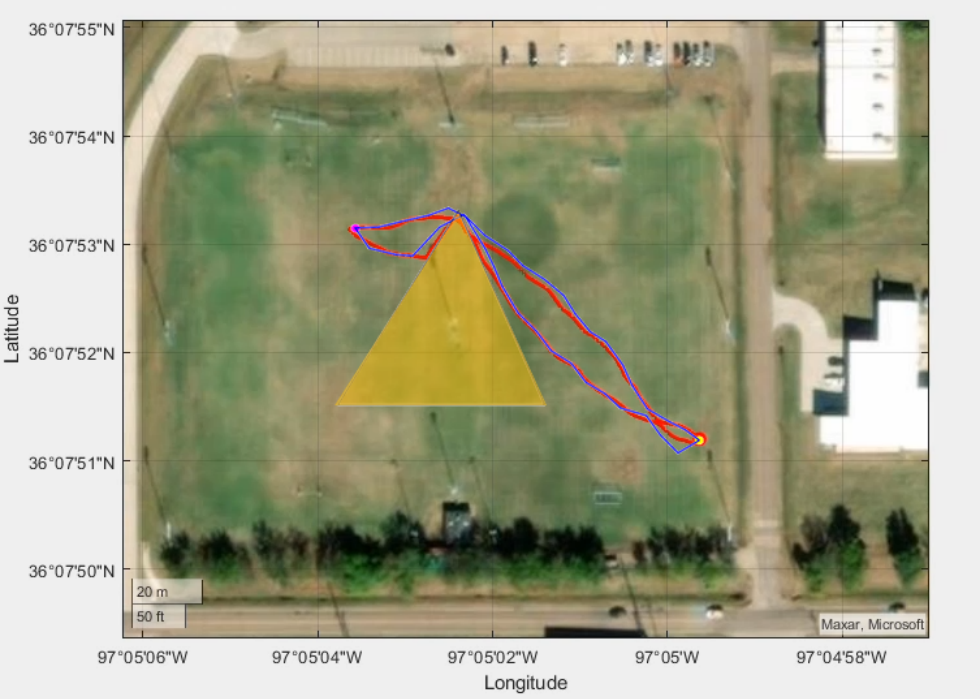

Figure 84: The third Eagle Flight test with a triangular obstacle

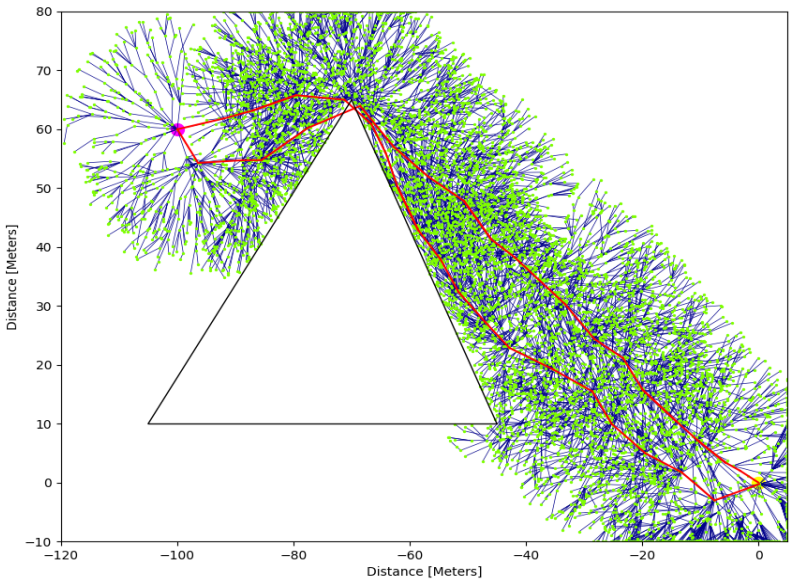

Figure 85: Third flight test MiniRRT search with a triangular obstacle

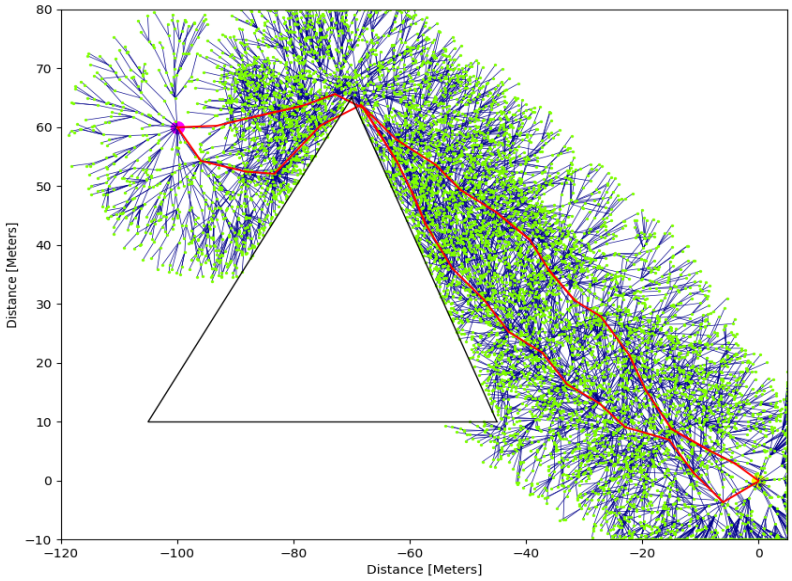

Figure 86: Fourth flight test MiniRRT search with a triangular obstacle

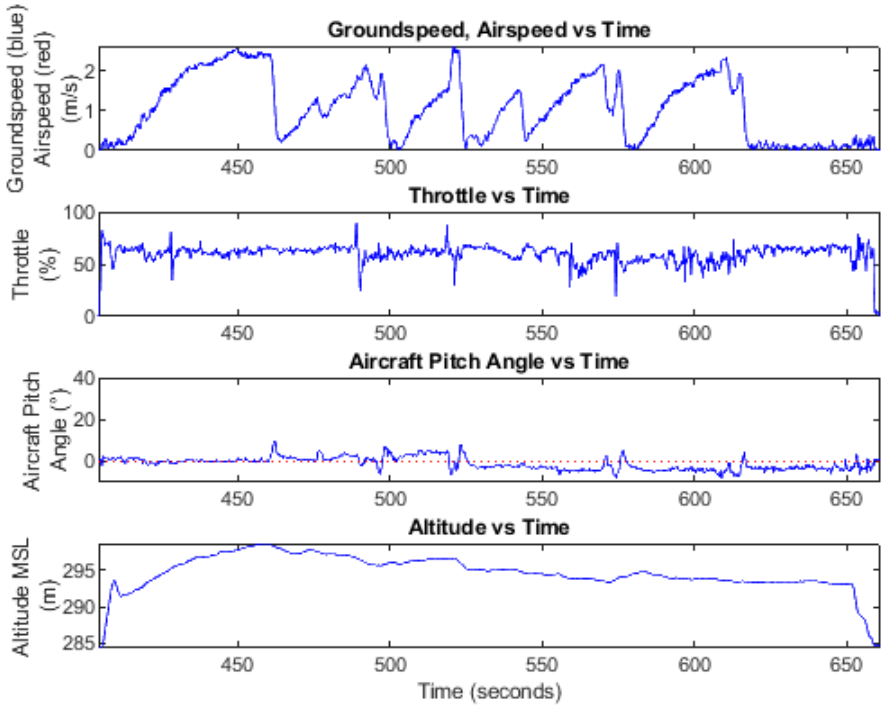

Figure 87: Eagle Flight Data for fourth test with a triangular obstacle

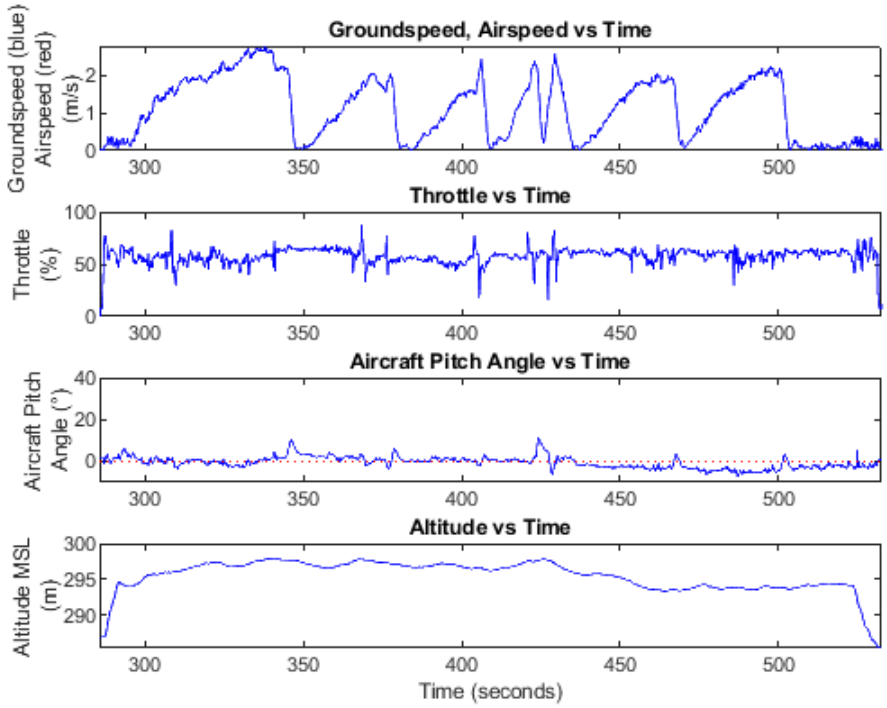

Figure 88: Eagle Flight Data for third test with a triangular obstacle

The next group of tests used multiple obstacles in the graph. The first test in this group used multiple rectangular obstacles that were wider than they were taller. Three were space out vertically at the center of the graph. The rectangle in the middle was double the length of the two outer rectangles. This test was repeated three times and the UAV was able to get to the goal point and back without getting stuck. The range for these tests was reduced back down to 120 degrees. The results for these tests can be seen in figures 89, 90, and 91. The MiniRRT output for these tests can be seen in figures 92, 93, and 94 and the flight data sets can be seen in figures 95, 96, 97. From these, it can be seen that the time for the first test was longer than the other two. However, the frequency of hovering to tracking was higher in the third test. Looking into the MiniRRT plots we can see where each tree is planted and how the obstacles affect the searching process.

Figure 89: The first Eagle Test with multiple rectangular obstacles

Figure 90: The second Eagle Test with multiple rectangular obstacles

Figure 91: The third Eagle Test with multiple rectangular obstacles

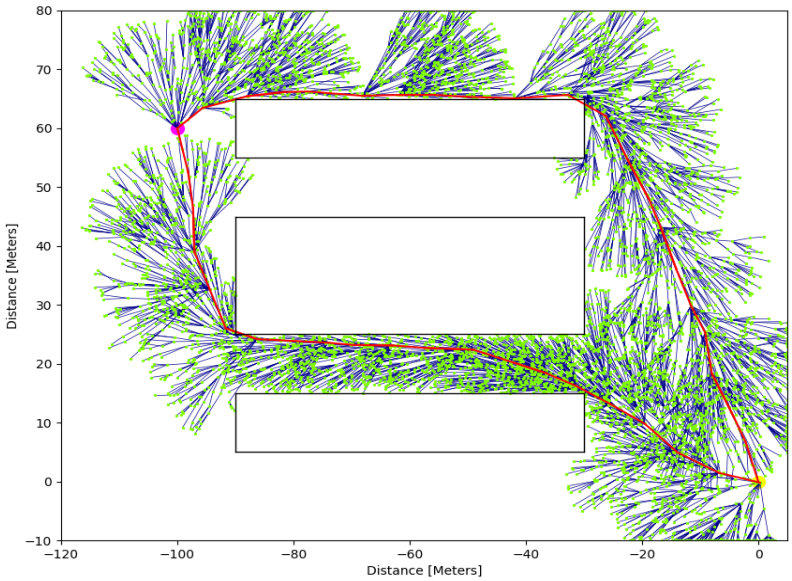

Figure 92: The first test's MiniRRT search with multiple rectangular obstacles

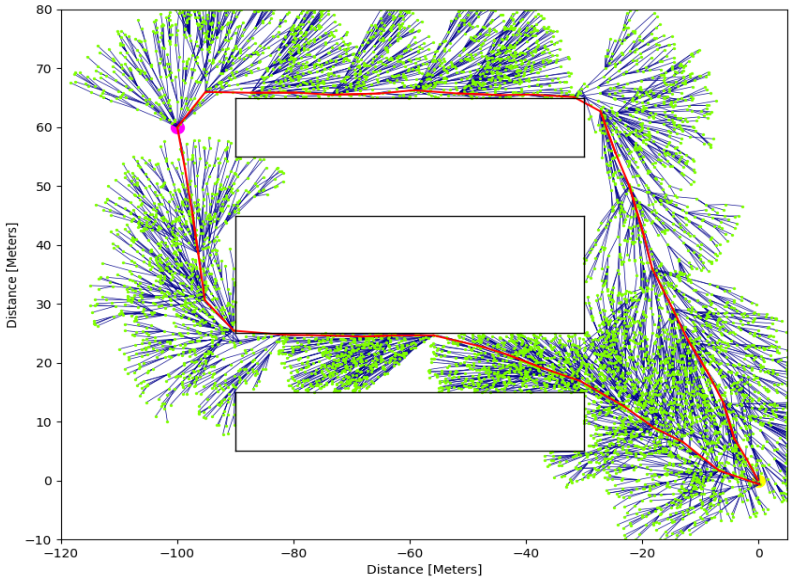

Figure 93: The second test's MiniRRT search with multiple rectangular obstacles

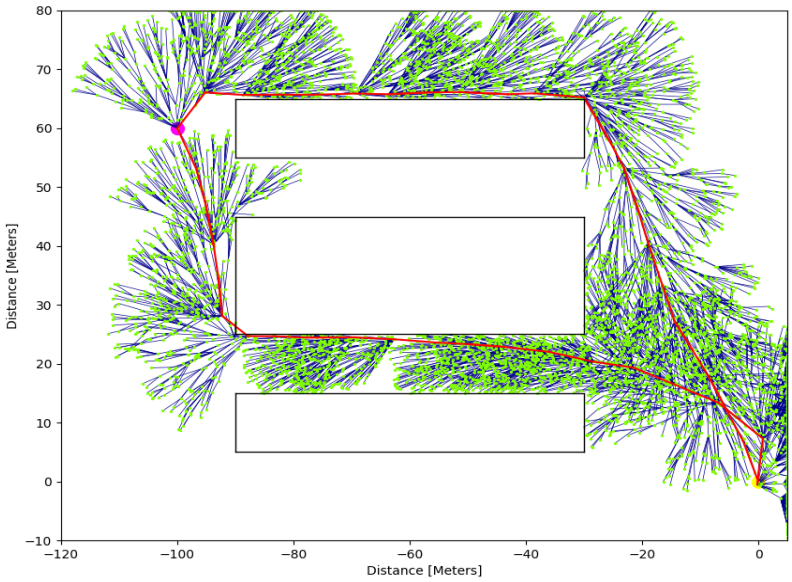

Figure 94: The third test's MiniRRT search with multiple rectangular obstacles

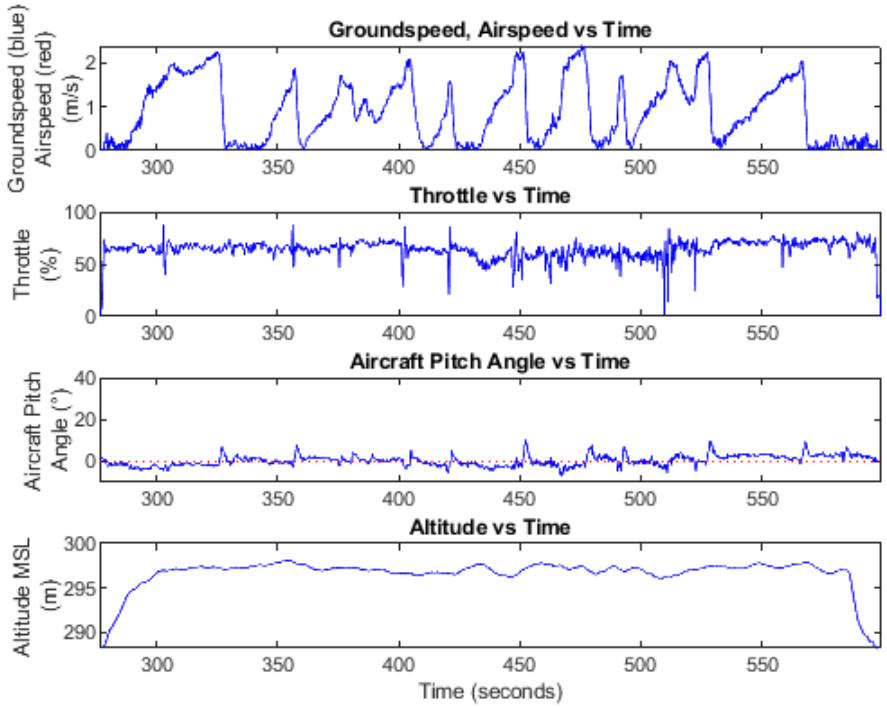

Figure 95: Eagle Flight Data for first test with multiple rectangular obstacles

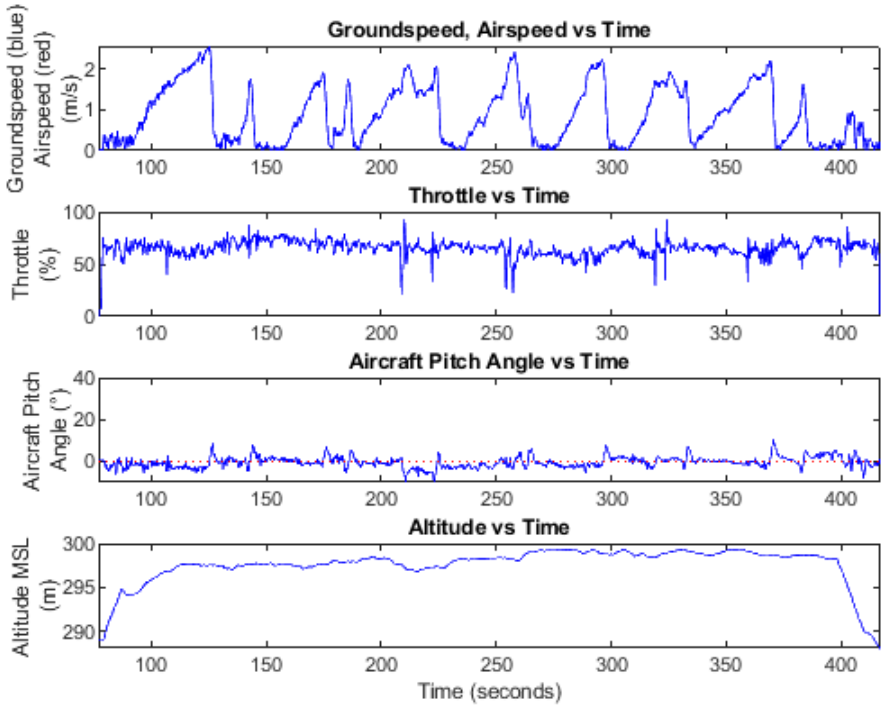

Figure 96: Eagle Flight Data for second test with multiple rectangular obstacles

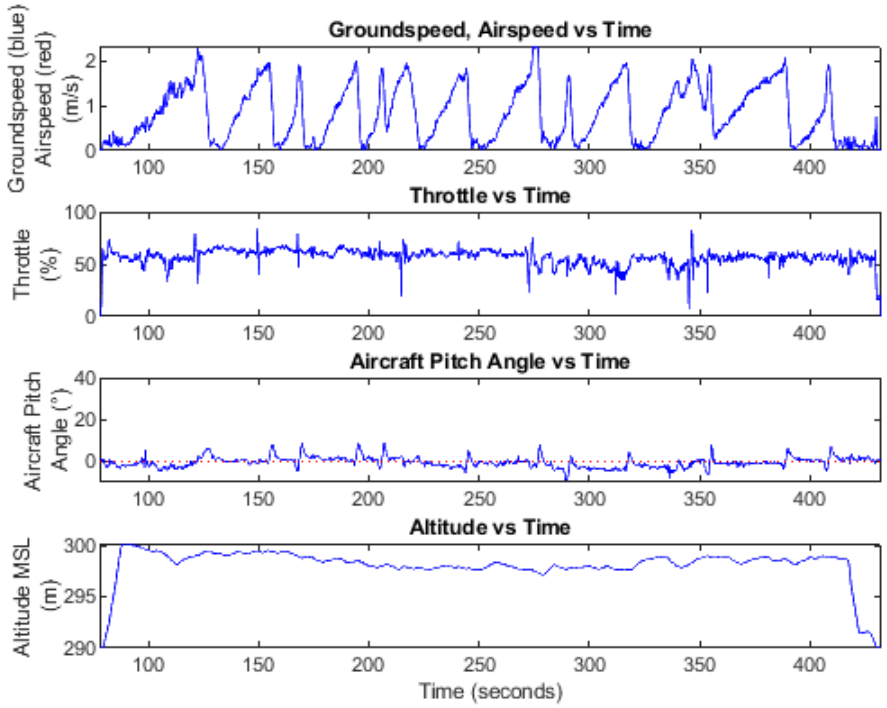

Figure 97: Eagle Flight Data for third test with rectangular obstacles

The next group of tests used multiple triangles in the graph. In one of the tests, the UAV hovered at the goal point without return home. The results can be seen in figure 98. Where in another the UAV was able to calculate a path back home, the result can be seen in figure 99. This test had the worst of cutting corners with the UAV as in figure 99. Adjusting the waypoint radius and the b-spline parameters can fix this issue. The MiniRRT output for each test can be seen in figures 100 and 101 and flight data in figures 102 and 103. In the first test the goal was reached in about 200 seconds, however the Eagle hovered there due to a error in the algorithm.

Figure 98: First Eagle Flight Test with multiple triangular obstacles

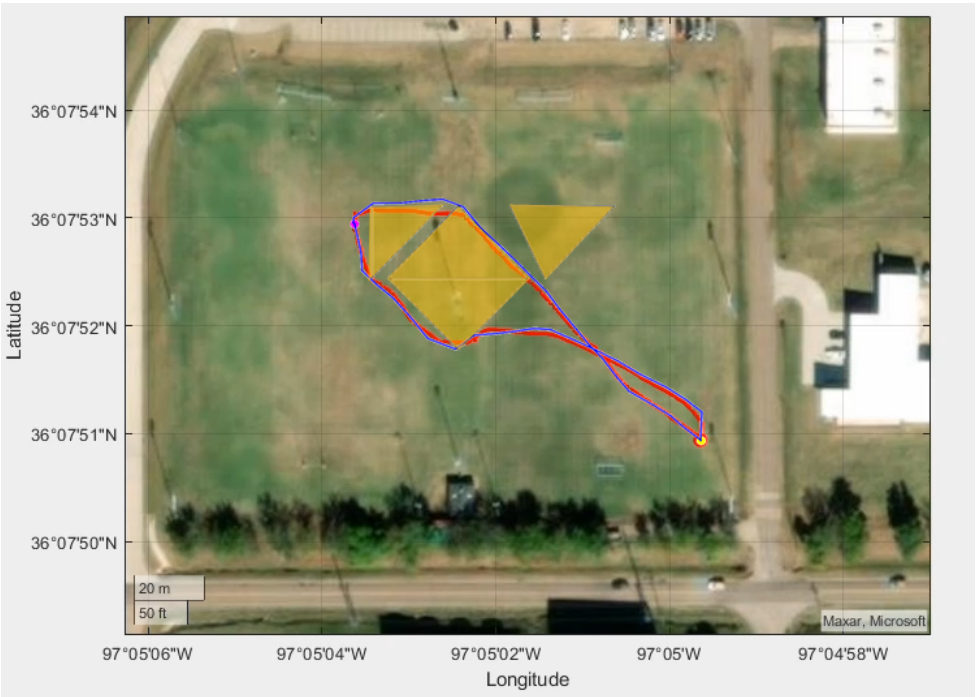

Figure 99: Second Eagle Flight Test with multiple triangular obstacles

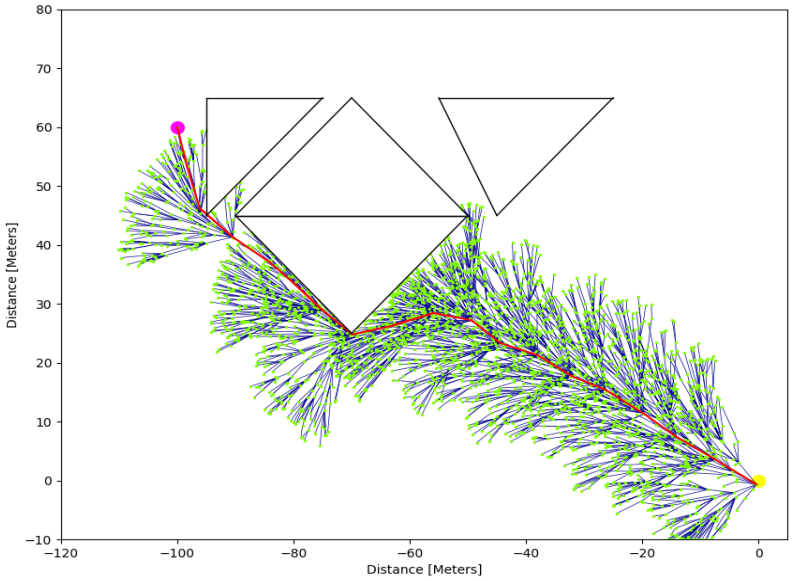

Figure 100: The first test's MiniRRT search with multiple triangular obstacles

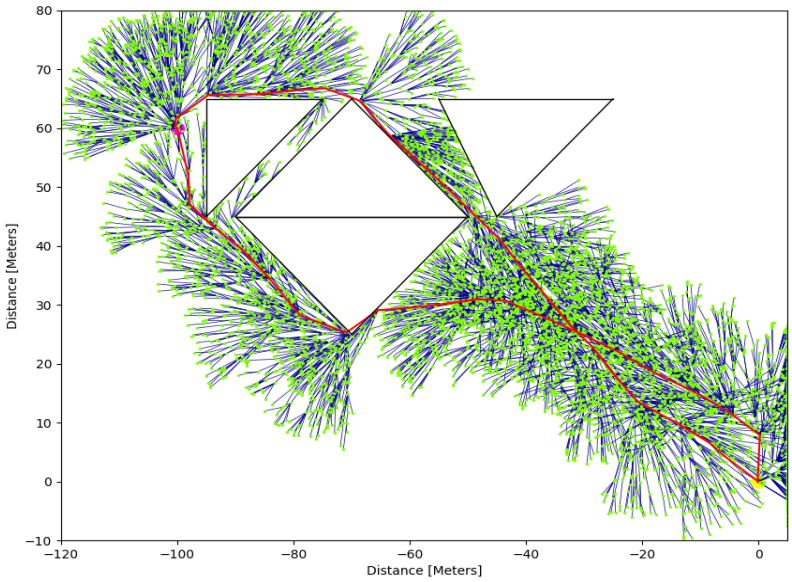

Figure 101: The second test's MiniRRT search with multiple triangular obstacles

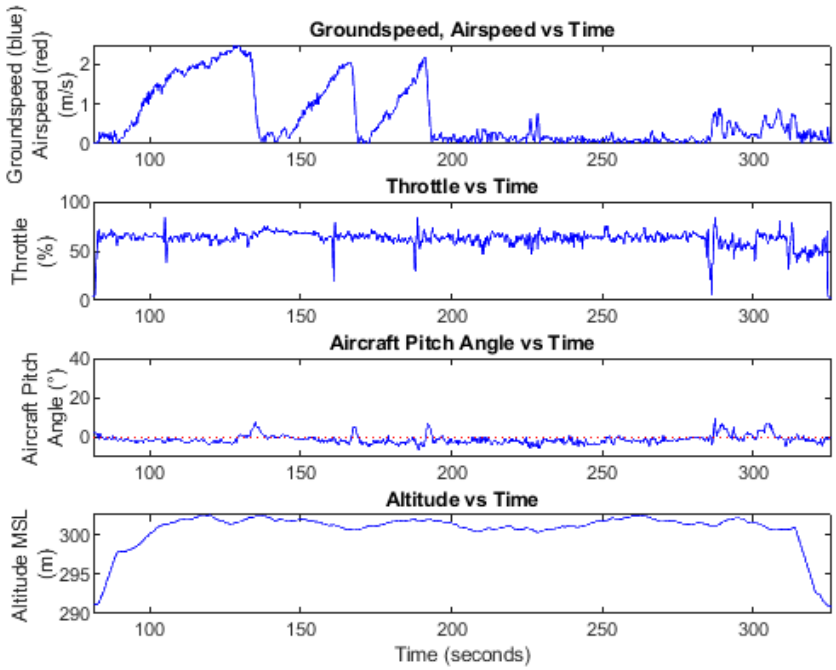

Figure 102: Eagle Flight Data for first test with multiple triangular obstacles

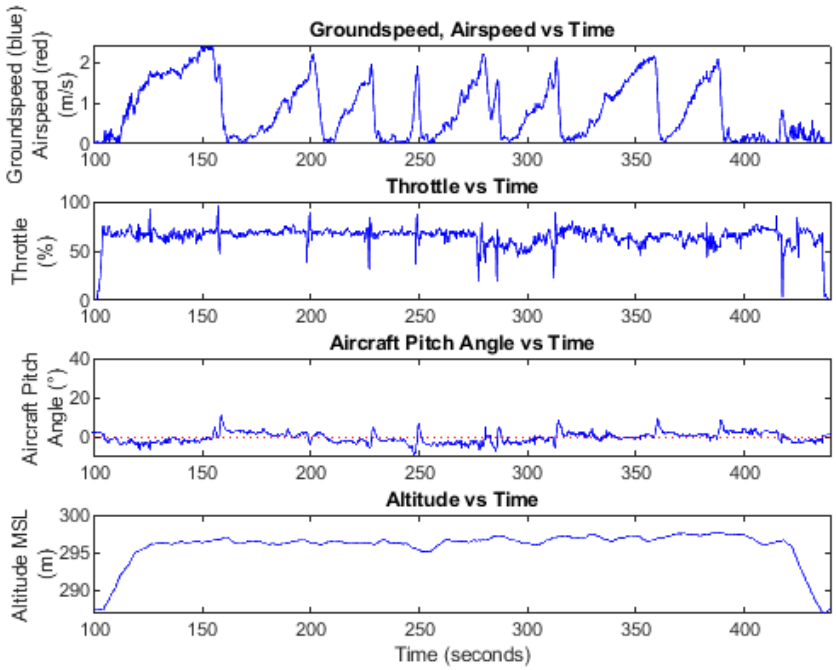

Figure 103: Eagle Flight Data for second test with triangular obstacles

The last tests used multiple circular obstacles. The obstacles were placed in close enough proximity to each other that they overlapped. This created small cavities. Two flights were done in this environment. Both of which had the UAV stuck in a concave section. After the first test, the range and distance of the search area for the MiniRRT were increased to 240 degrees and 40 meters, respectfully. However, this did not help the UAV plan a path around the obstacles. Increasing the distance parameter would have helped more, but this would reduce the smoothness of the path, making it more jagged. Depending on the centerline for the tree, the search area could be taken up by the obstacles reducing the search area. Adjusting the path choosing criteria for the trees can help with this issue. However, this issue brings up the concept of exploration versus exploitation which is not the scope of this thesis. Figures with the Pixhawk's position data output with obstacle placement and flight data can be seen in figures 104 and figure 104. From figure 104 the ground speed increases up to a point where the Eagle hovers in place trying to calculate a path. In the flight data shown in figure 105, the Eagle hovered for quite some time until the tests was aborted.

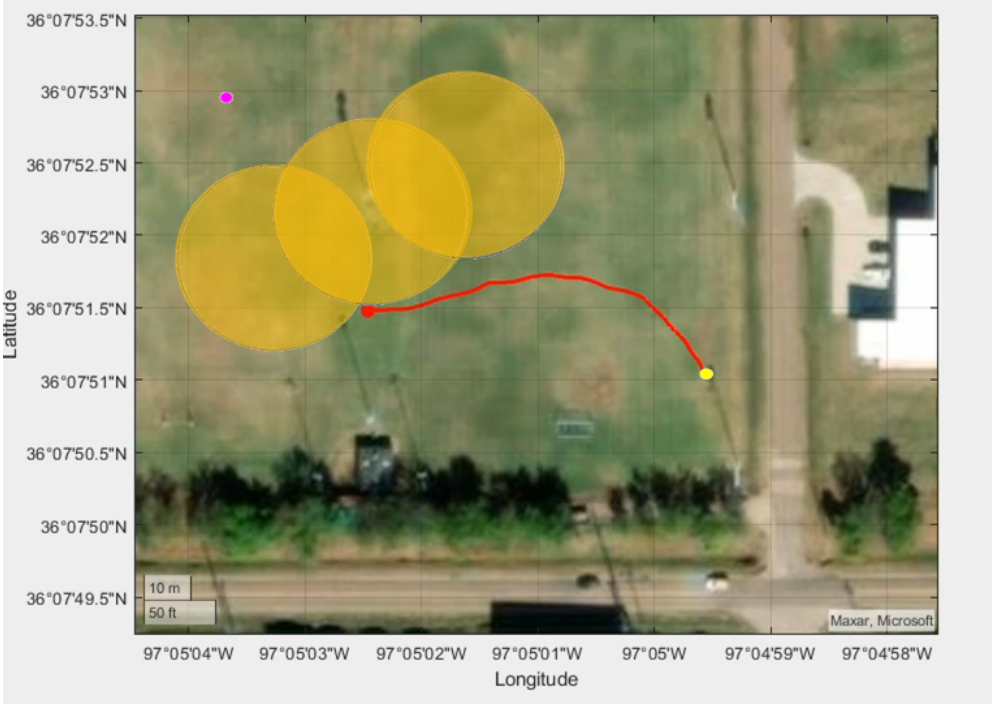

Figure 104: Eagle stuck at position in cavity in second test

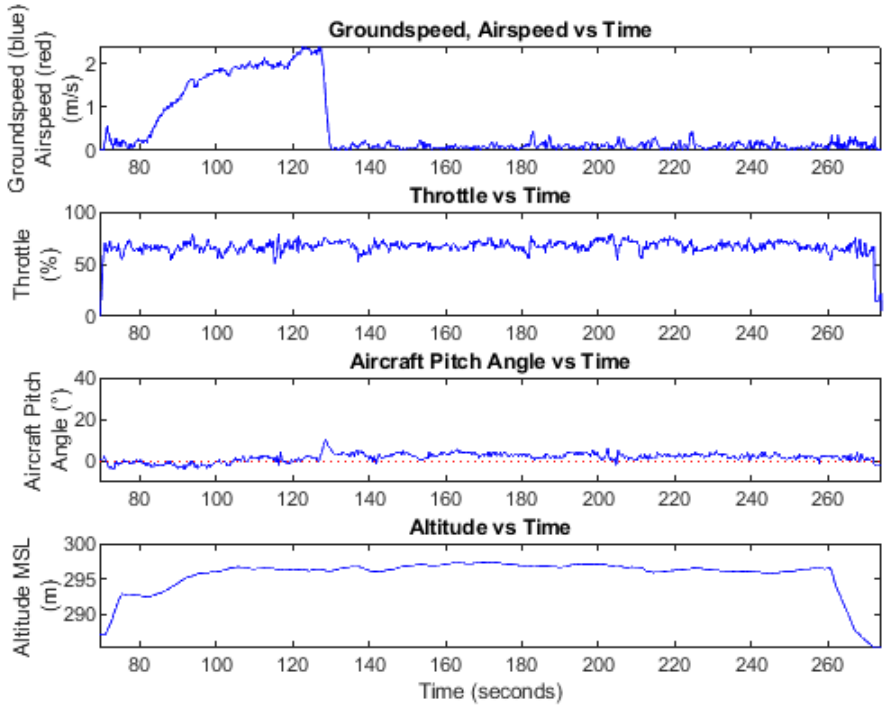

Figure 105: Eagle Flight Data for second test

From the collection of tests done, it is shown that the MiniRRT algorithm calculates paths at different rates, the groundspeed from each flight data set shows the tracking and hovering times. This gives a glimpse at how quickly or slowly a path segment is generated. These tests also show how the MiniRRT algorithm gets around a variety of different obstacles and shows how it gets stuck against obstacle faces due to a low search range. However, getting stuck in cavities between joint obstacles and not being able to back out is a concern. Adjusting parameters can help to alleviate these issues however, new cost criteria would better help, as it would affect how best paths are chosen, and not lead the UAV into dead ends. The tests also show that the parameters of MiniRRT can be catered to fit a fixed-wing aircraft and multi-rotor aircraft. They also show the capability of the MiniRRT algorithm in continuous flight and planning.

CHAPTER VI

CONCLUSIONS

6.1 Outcomes

Through simulations and flight tests it is shown that the path planning algorithm MiniRRT is able to produce paths that reach a specified goal that is within the graph. It does this at a significantly lower time compared to RRT and RRT* due to the fact that it searches locally instead of globally. In addition to doing local searches MiniRRT has the capability of tracking multiple goal points within the graph. While making a feasible path for a UAV to follow. The MiniRRT algorithm is also capable of creating paths that avoid obstacles given proper parameters.

With using the MiniRRT algorithm alongside ArudPilot path nodes that are generated are treated as waypoints. These are converted from the graph's Cartesian system to the World Geodetic System. These waypoints are passed to Ardupilot from the path planning algorithm in small batches for the UAV to track towards. This allows for real-time operation planning a tracking waypoints at the same time. The flight data sets that include ground speed versus time give a sense of the varying frequencies that paths are generated and pushed in time. However, due to the complexity of some search spaces, this can be slowed down as buffering happens with the UAVs flight as it can reach the end of a path segment as one is being calculated.

The MiniRRT algorithm was implemented onto a LattePanda microcomputer as a secondary flight computer. This microcomputer was used as it was able to quickly generate paths efficiently at a low cost with the MiniRRT algorithm. The LattePanda was integrated

into two different UAVs the Nano Talon EVO, which is a fixed-wing aircraft developed by SoniModell. And an Eagle, which is a multi-rotor aircraft developed by the research engineers at USRI. Figure 106 shows the Eagle in the middle of one of the tests.

Figure 106: The Eagle flying in one of the tests.

The MiniRRT algorithm was tested on both of these platforms and was able to generate paths towards multiple goals using a variety of obstacles in the map. The runtimes for the tests done with the Eagle can be seen in table 7. These are the tests that were able to

complete the entire mission with there respective maps. The flight times vary depending on the obstacle orientations.

Obstacle Orientation	Run time (seconds)
Circle	489.99
Rectangle	477.11
Triangle 1	264.29
Triangle 2	265.41
Multiple Rectangles 1	333.04
Multiple Rectangles 2	640.54
Multiple Rectangles 3	356.71
Multiple Triangles	445.68

Table 7: Run times for tests

6.2 Future Work

With the MiniRRT algorithm's cost criteria, it has the possibility of getting stuck on straight edge obstacle faces. Using the parameters of the MiniRRT can reduce the possibility of it getting stuck. However, this can reduce the optimality of the path generated. Using a method to adaptively change the range of the search area can help it get out of the scenarios with having to change these parameters within the algorithm on each flight. A better method of choosing paths can also help with the algorithm getting stuck inside the cavities of obstacles or between joint obstacles.

The MiniRRT algorithm can be extended to work with dynamics obstacles, or with potential fields. Potential fields can be used as cost maps that can change the way the algorithm navigates the graph. These can be used with areas like terrain navigation, or use in weather applications. With dynamics obstacles, this algorithm can extend to multiple vehicle applications, like swarms, or unmanned traffic management (UTM).

The addition of perception via sensors can be useful for obstacle avoidance. Instead of making a map for the UAV it will perceive them and add them to the map themselves. So, the creation of a map with obstacles is not necessary. The defined geo-fence is still necessary, however.

The MiniRRT algorithm can also be easily extended into 3-dimensional space. However, this comes with a longer search time as it adds in a new dimension, increasing the search area and complexity of the graph. Utilizing the LattePanda microcomputer that was used in the tests in this paper, may not be as effective as it was only searching in 2-dimensional space. So, a stronger computer is necessary to be able to do more complex testing.

REFERENCES

[1] Oktay Arslan and Panagiotis Tsiotras, *Use of relaxation methods in sampling-based algorithms for optimal motion planning*, 2013 IEEE International Conference on Robotics and Automation (2013), 2421–2428.

[2] Randal W. Beard and Timothy W. McLain, *Small unmanned aircraft: Theory and practice*, Princeton University Press, 2012.

[3] Dave Carsten, Josephand Ferguson and Anthony Stentz, *3d field d: Improved path planning and replanning in three dimensions*, 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2006, pp. 3381–3386.

[4] Sanjiban Choudhury, Sebastian Scherer, and Sanjiv Singh, *Rrt*-ar: Sampling-based alternate routes planning with applications to autonomous emergency landing of a helicopter*, 2013 IEEE International Conference on Robotics and Automation, 2013, pp. 3947–3952.

[5] J.A. Cobano, D. Alejo, G. Heredia, J.R. Martinez-de Dios, and A. Ollero, *Efficient collision-free trajectory planning for wsn data collection with unmanned aerial vehicles*, IFAC Proceedings Volumes **46** (2013), no. 30, 220–225.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms*, third ed., The MIT Press, 2009.

[7] CubePilot, *Px4 user guided*, https://docs.px4.io/master/en/flight_controller/pixhawk-2.html.

[8] Vaibhav Darbari, Saksham Gupta, and Om Prakash Verma, *Dynamic motion planning for aerial surveillance on a fixed-wing uav*, 2017 International Conference on Unmanned Aircraft Systems (ICUAS), 2017, pp. 488–497.

[9] NetworkX developers, *Networkx documentation*, https://networkx.org.

[10] E. Dijkstra, *A note on two problems in connexion with graphs*, Numerische Mathematik **1** (1959), 269–271.

[11] Mohammad Reza Ranjbar Divkoti and Mostafa Nouri-Baygi, *Rcs: a fast path planning algorithm for unmanned aerial vehicles*, arXiv: Robotics (2019).

[12] Python Software Foundation, *Python*, https://www.python.org/.

[13] Sean Gillies, Aron Bierbaum, and Kai Lautaportti, *Shapely*, https://shapely.readthedocs.io/en/stable/index.html.

[14] C. Goerzen, Z. Kong, and B. Mettler, *A survey of motion planning algorithms from the perspective of autonomous uav guidance*, Journal of Intelligent and Robotic Systems: Theory and Applications **57** (2010), 65–100.

[15] Jonathan L. Gross, Jay Yellen, and Mark Anderson, *Graph theory and its applications*, third ed., Textbook in Mathematics, Taylor & Francis Group, LLC, Boca Raton, FL, 2019.

[16] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael, *A formal basis for the heuristic determination of minimum cost paths*, IEEE Transactions on Systems Science and Cybernetics **4** (1968), no. 2, 100–107.

[17] Open Source Robotics Foundation Inc., *Ros*, https://www.ros.org/.

[18] Scott Howard James and Robert Nicholas Raheb, *Path planning for critical atm/utm areas*, 2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC), 2019.

[19] Sertac Karaman and Emilio Frazzoli, *Sampling-based algorithms for optimal motion planning*, The International Journal of Robotics Research **30** (2011), no. 7, 846–894.

[20] Sven Koenig and Maxim Likhachev, *D* lite*, In Proceedings of the AAAI Conference of Artificial Intelligence (AAAI) (2002), 476–483.

[21] James J. Jr. Kuffner and Steven M. LaValle, *Rrt-connect: An efficient approach to single-query path planning*, Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065), vol. 9, 200, pp. 995–1001.

[22] S.M. LaValle and J.J. Kuffner, *Randomized kinodynamic planning*, Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C), vol. 1, 1999, pp. 473–479 vol.1.

[23] Steven M. LaValle, *Planning algorithms*, Cambridge University Press, New York, NY, 2006.

[24] Dasol Lee and David H. Shim, *Rrt-based path planning for fixed-wing uavs with arrival time and approach direction constraints*, 2014 International Conference on Unmanned Aircraft Systems (ICUAS) (2014), 317–328.

[25] MathWorks, *Motion planning with rrt for fixed-wing uav*, https://www.mathworks.com/help/uav/ug/motion-planning-with-rrt-for-fixed-wing-uav.html.

[26] Omar Mechali, Limei Xu, Mingzhu Wei, Ilyas Benkhaddra, Fan Guo, and Abdelkader Senouci, *A rectified rrt* with efficient obstacles avoidance method for uav in 3d environment*, 2019 IEEE 9th Annual International Conference on CYBER Technology in Automation, Control, and Intelligent Systems (CYBER), 2019, pp. 480–485.

[27] Jiawei Meng, Vijay M. Pawar, Sebastian Kay, and Angran Li, *Uav path planning system based on 3d informed rrt* for dynamic obstacle avoidance*, 2018 IEEE International Conference on Robotics and Biomimetics (ROBIO), 2018, pp. 1653–1658.

[28] B. D. Newton, *Executing rrt paths with the ar . drone quadrotor*, 2012.

[29] Micheal Otte and Emilio Frazzoli, *Rrtx: Asymptotically optimal single-query sampling-based motion planning with quick replanning*, The International Journal of Robotics Research **35** (2016), no. 7, 797–822.

[30] RFDesign, *Rfd900x asynchronous firmware user manual*, http://files.rfdesign.com.au.

[31] RFDesign, *Rfd900x mulipoint firmware user manual*, www.rfdesign.com.au.

[32] RFDesign, *Rfd900x peer-to-peer firmware user manual*, www.rfdesign.com.au.

[33] RFDesign, *Rfdesign*, http://store.rfdesign.com.au/rfd-900x-modem/.

[34] SonicModell, *Sonicmodell mini skyhunter v2*, http://www.sonicmodell.com/product/Sonicmodell-Mini-Skyhunter-V2-1238mm-Wingspan-FPV-EPO-RC-Airplane.html.

[35] SonicModell, *Zohd nano talon evo*, http://www.sonicmodell.com/community/zohd-nano-talon-6.html.

[36] Omar Souissi, Rabie Benatitallah, David Duvivier, AbedlHakim Artiba, Nicolas Belanger, and Pierre Feyzeau, *Path planning: A 2013 survey*, 2013, pp. 1–8.

[37] Anthony Stentz, *Optimal and efficient path planning for unknown and dynamic environments*, International Journal of Robotics and Automation **10** (1993), 89–100.

[38] ArduPilot Dev Team, *Ardupilot*, https://ardupilot.org/.

[39] LattePanda Team, *Lattepanda 2g/32g*, https://www.lattepanda.com/products/1.html.

[40] Naifeng Wen, Lingling Zhao, Xiaohong Su, and Peijun Ma, *Uav online path planning algorithm in a low altitude dangerous environment*, IEEE/CAA Journal of Automatica Sinica **2** (2015), no. 2, 173–185.

[41] Kwangjin Yang, Sangwoo Moon, Seunghoon Yoo, Jaehyeon Kang, Nakju L. Doh, Hong B. Kim, and Sanghyun Joo, *Spline-based rrt path planner for non-holonomic robots*, Journal of Intelligent and Robotic Systems **73** (2014), 763–782.

[42] Christian Zammit and Erik-Jan Van Kampen, *Comparison between a* and rrt algorithms for uav path planning.*

[43] Zhe Zhang, Jian Wu, Jiyang Dai, and Cheng He, *A novel real-time penetration path planning algorithm for stealth uav in 3d complex dynamic environment*, IEEE Access **8** (2020), 122757–122771.

APPENDICES

Title of Appendix

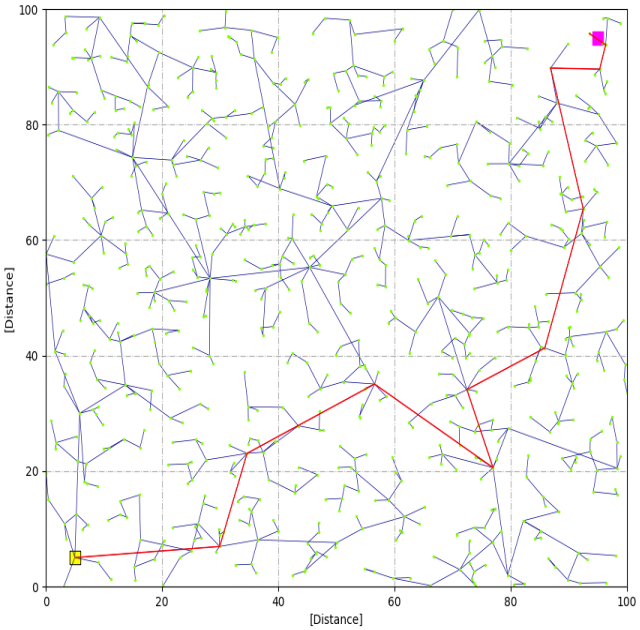

Figure 107: RRT with a 100×100 Graph and no obstacles

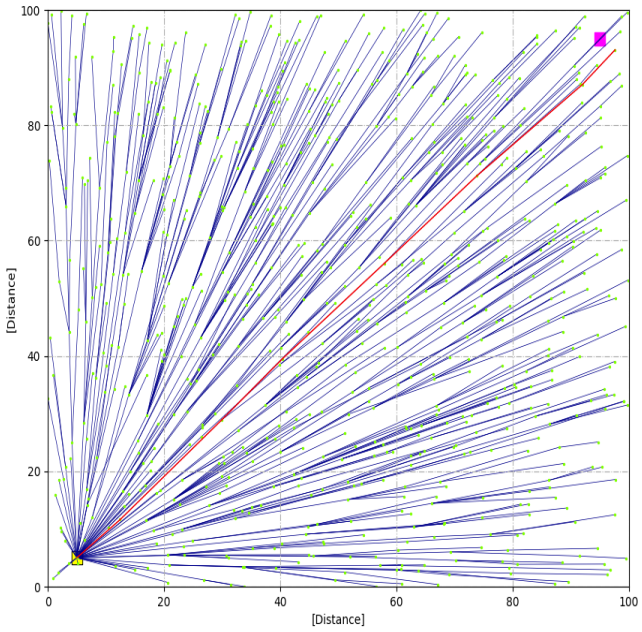

Figure 108: RRT* with a 100×100 Graph and no obstacles

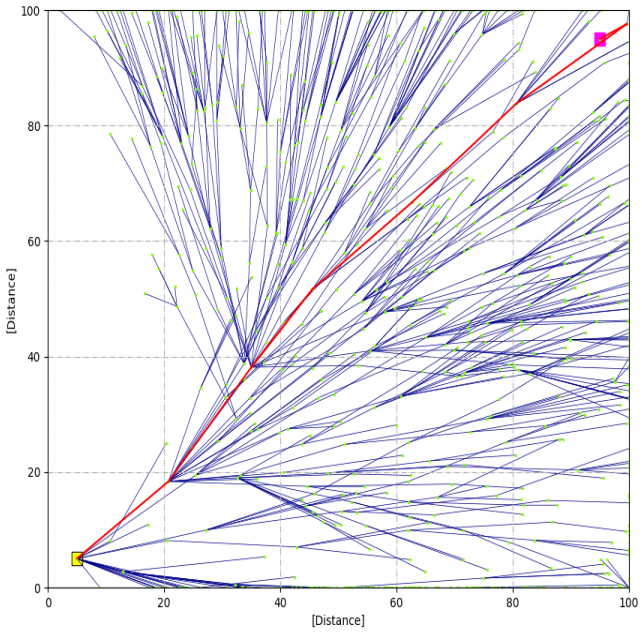

Figure 109: MiniRRT with a 100×100 Graph and no obstacles

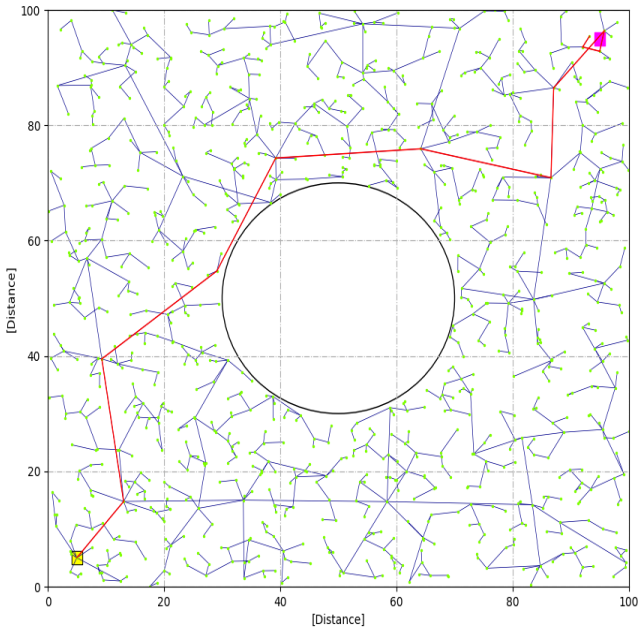

Figure 110: RRT with a 100×100 Graph with one obstacle

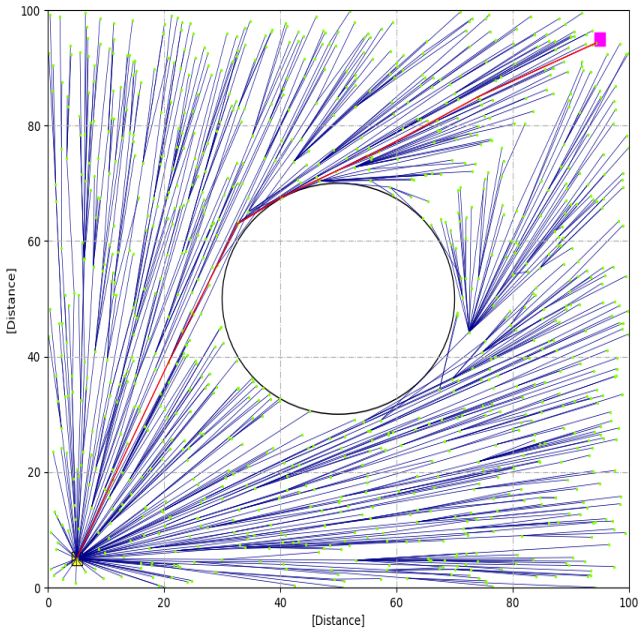

Figure 111: RRT* with a 100×100 Graph and one obstacle

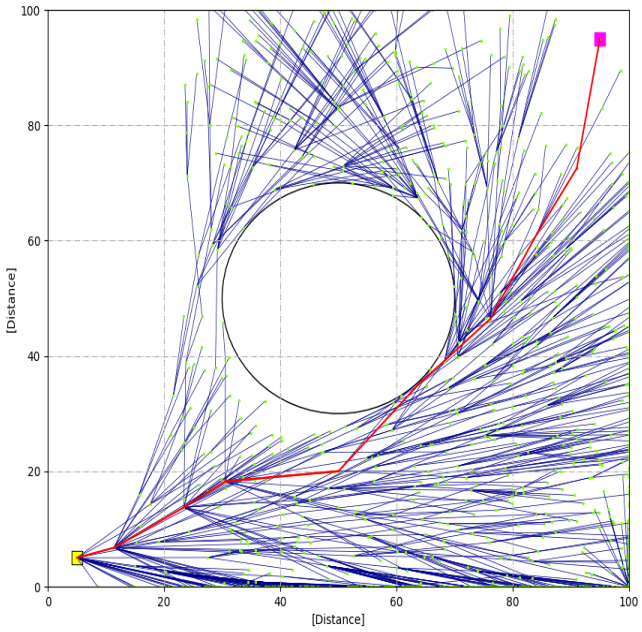

Figure 112: MiniRRT with a 100×100 Graph and one obstacle

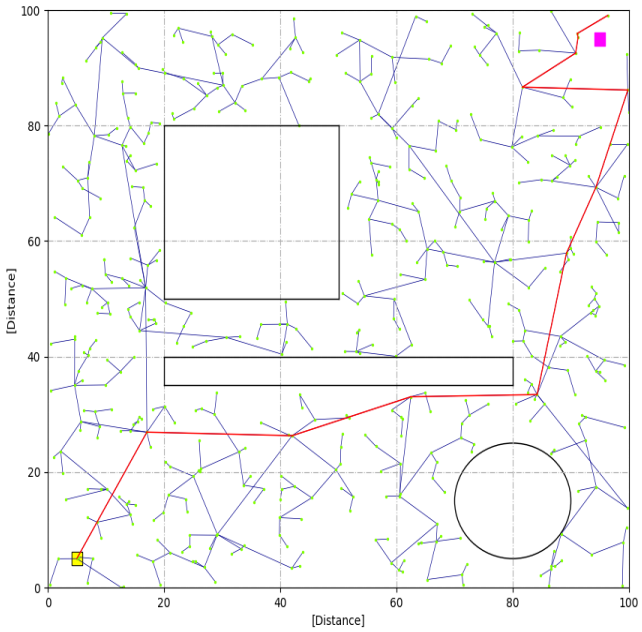

Figure 113: RRT with a 100×100 Graph with three obstacles

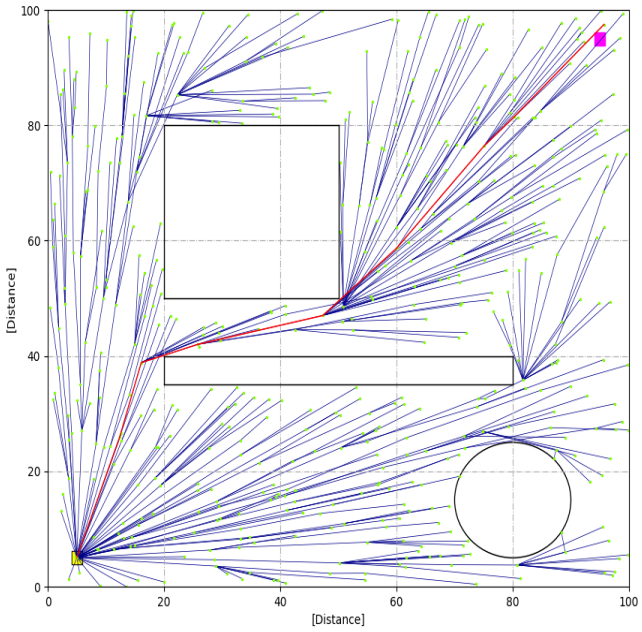

Figure 114: RRT* with a 100×100 Graph with three obstacles

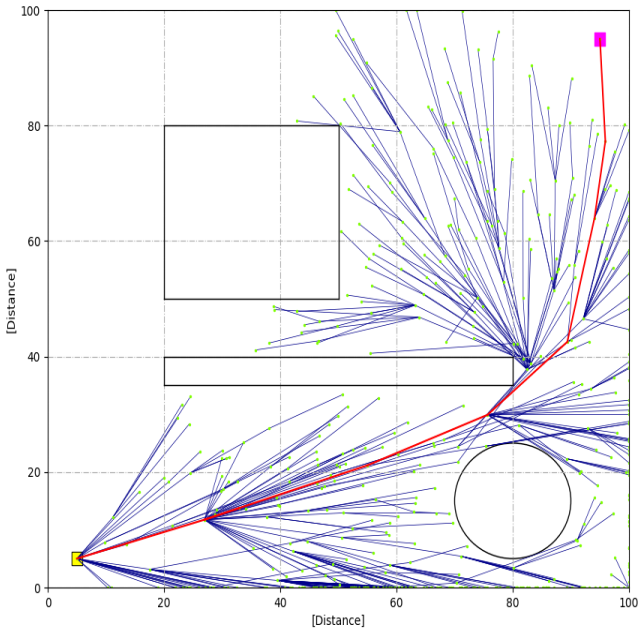

Figure 115: MiniRRT with a 100×100 Graph with three obstacles

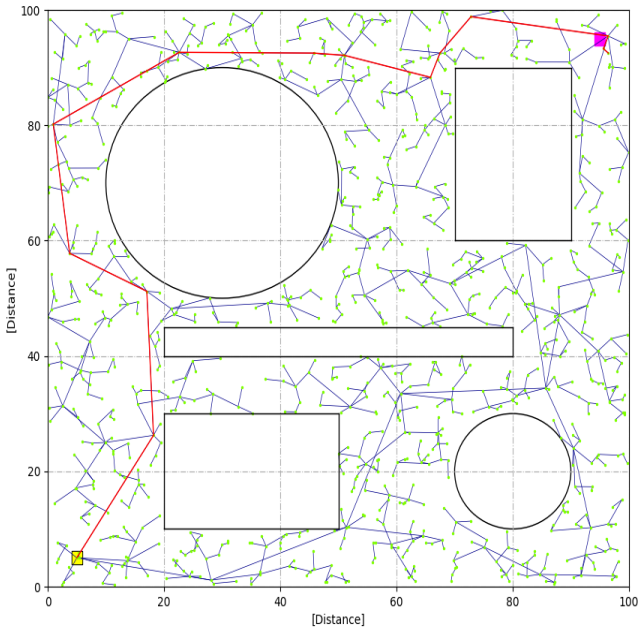

Figure 116: RRT with a 100×100 Graph with five obstacles

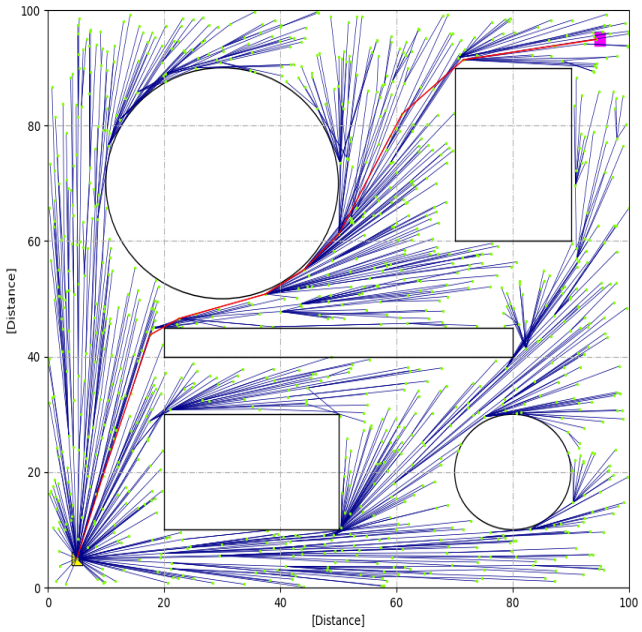

Figure 117: RRT* with a 100×100 Graph with five obstacle

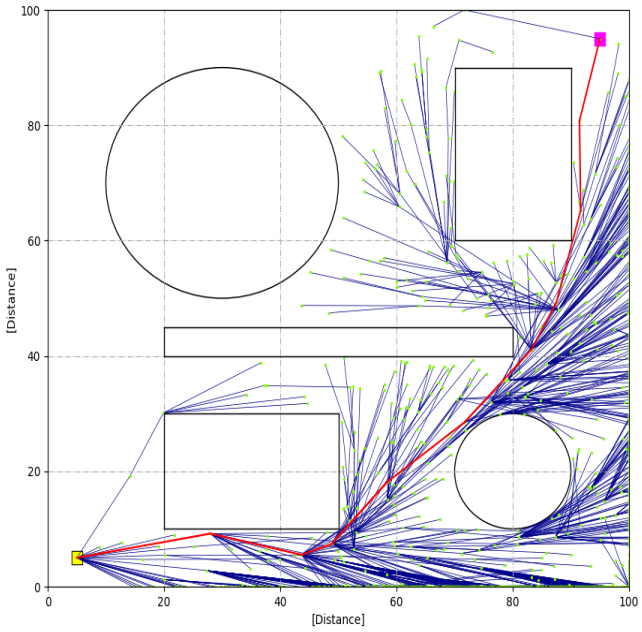

Figure 118: MiniRRT with a 100×100 Graph with five obstacle

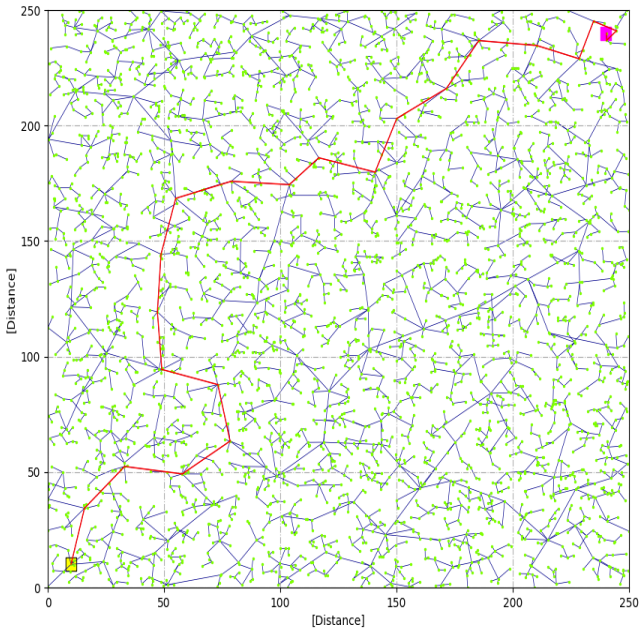

Figure 119: RRT with a 250×250 Graph with no obstacles

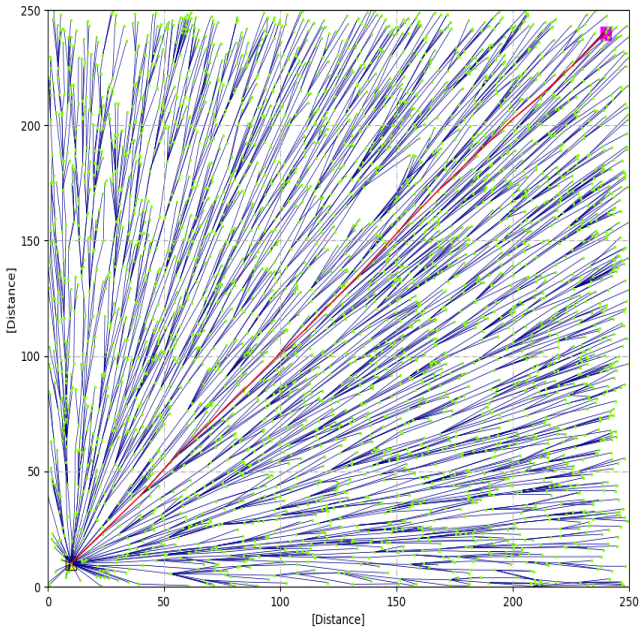

Figure 120: RRT* with a 250×250 Graph with no obstacles

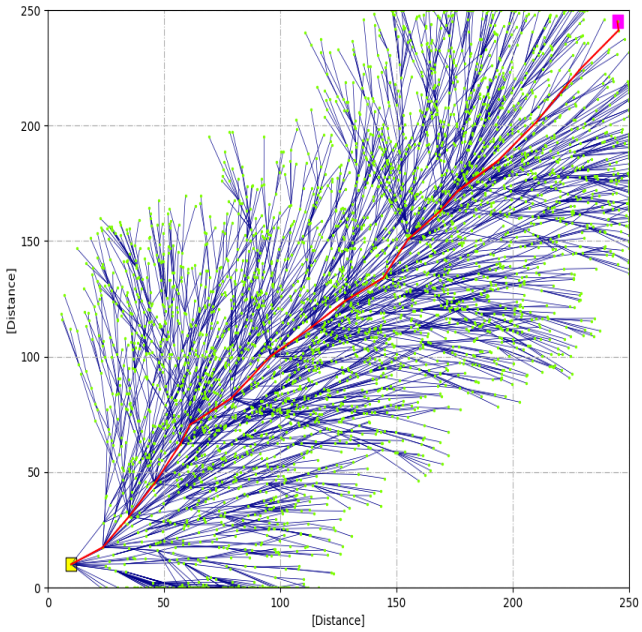

Figure 121: MiniRRT with a 250×250 Graph with no obstacles

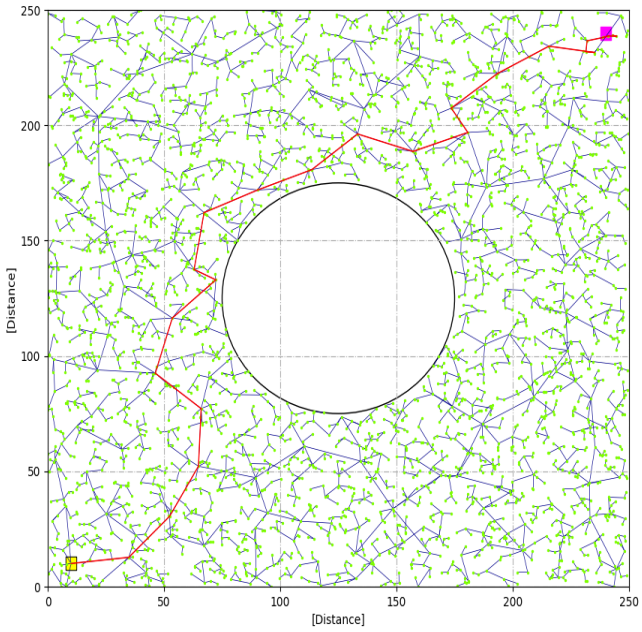

Figure 122: RRT with a 250 × 250 Graph with one obstacles

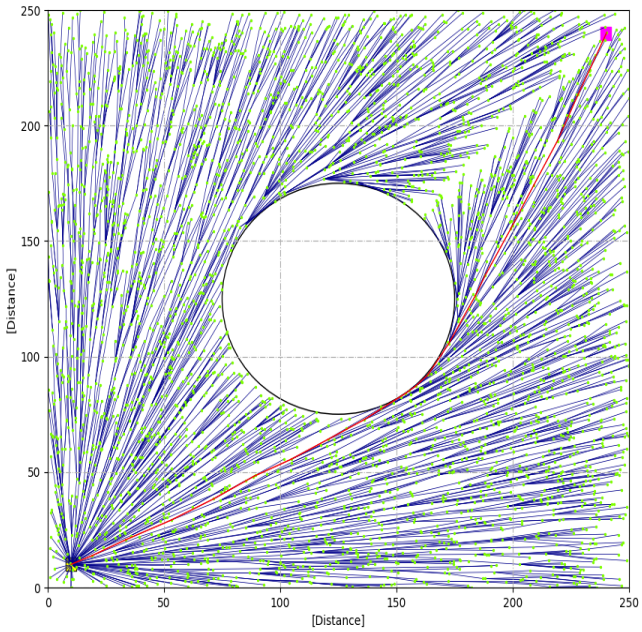

Figure 123: RRT* with a 250 × 250 Graph with one obstacles

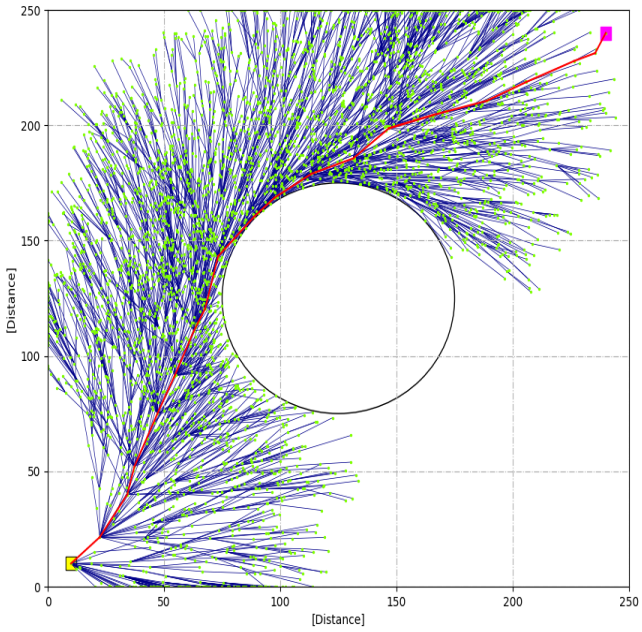

Figure 124: MiniRRT with a 250×250 Graph with one obstacles

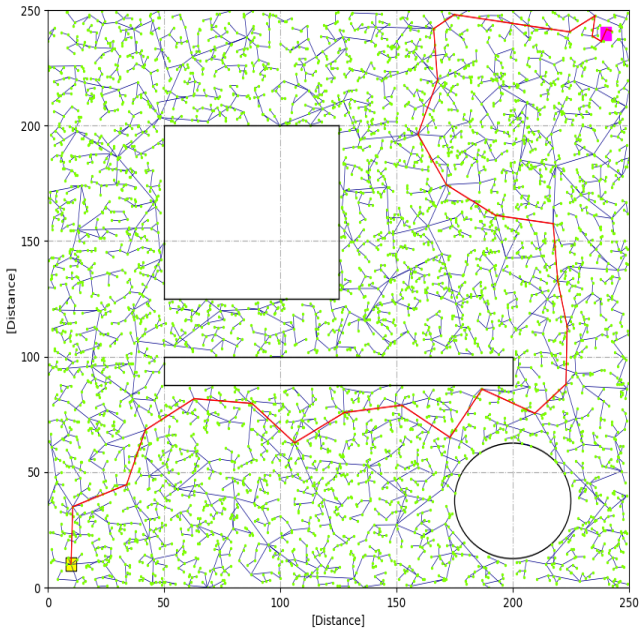

Figure 125: RRT with a 250×250 Graph with three obstacles

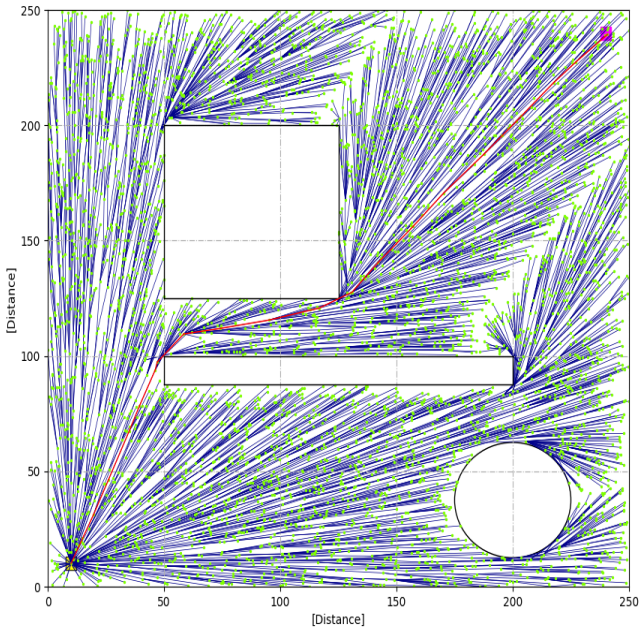

Figure 126: RRT* with a 250×250 Graph with three obstacles

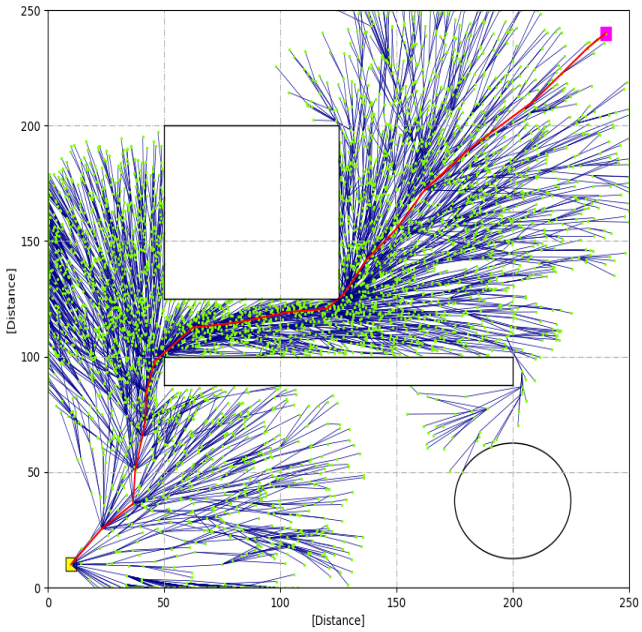

Figure 127: MiniRRT with a 250×250 Graph with three obstacles

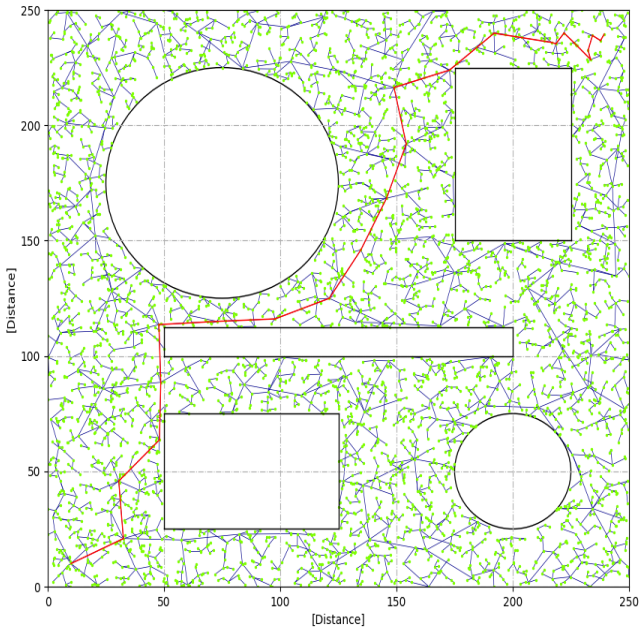

Figure 128: RRT with a 250×250 Graph with five obstacles

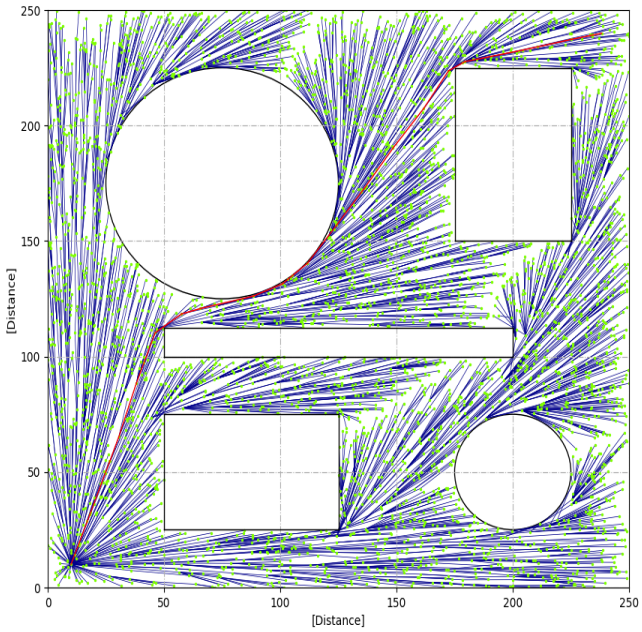

Figure 129: RRT* with a 250×250 Graph with five obstacles

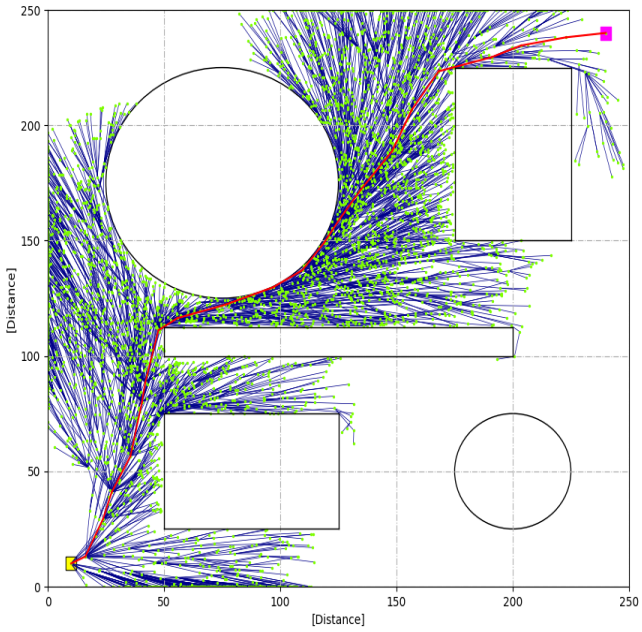

Figure 130: MiniRRT with a 250×250 Graph with five obstacles

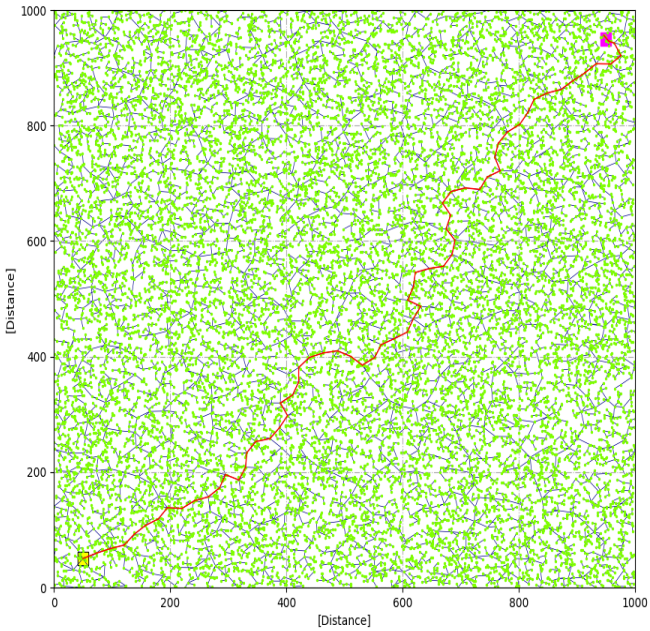

Figure 131: RRT with a 1000×1000 Graph with no obstacles

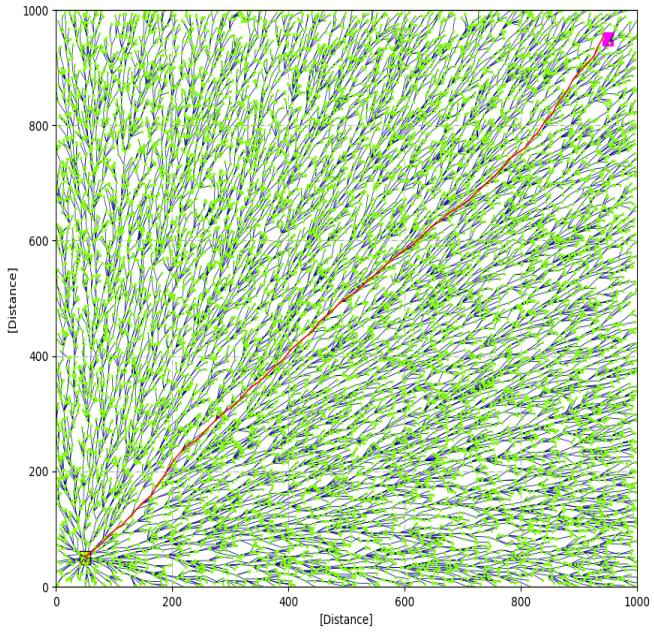

Figure 132: RRT* with a 1000×1000 Graph with no obstacles

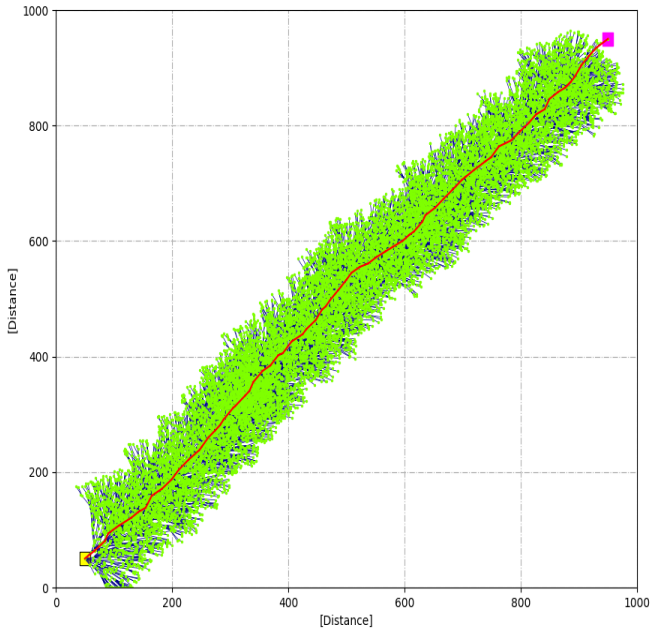

Figure 133: MiniRRT with a 1000×1000 Graph with no obstacles

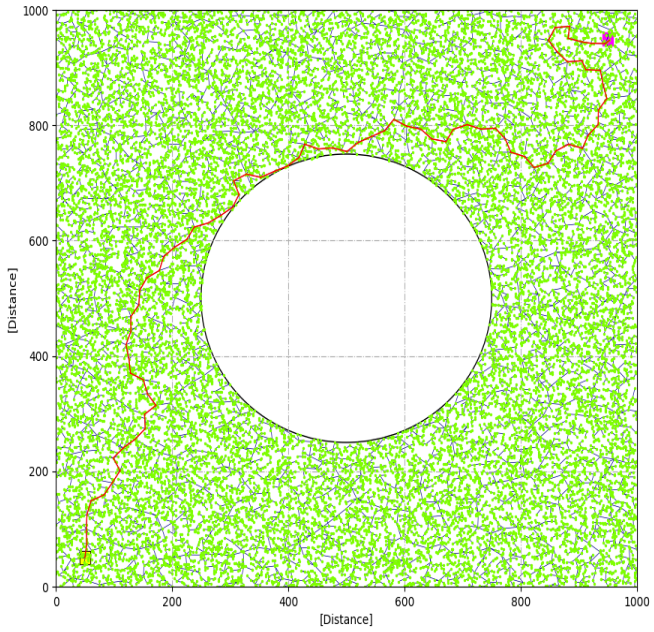

Figure 134: RRT with a 1000×1000 Graph with one obstacles

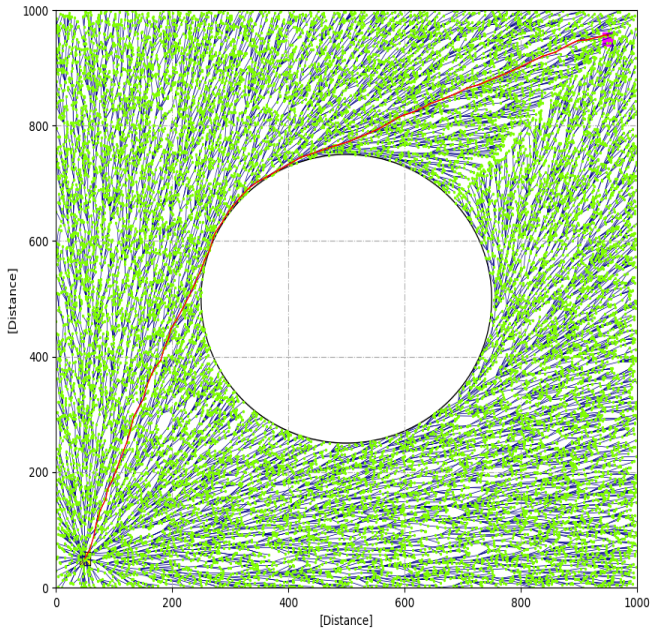

Figure 135: RRT* with a 1000×1000 Graph with one obstacles

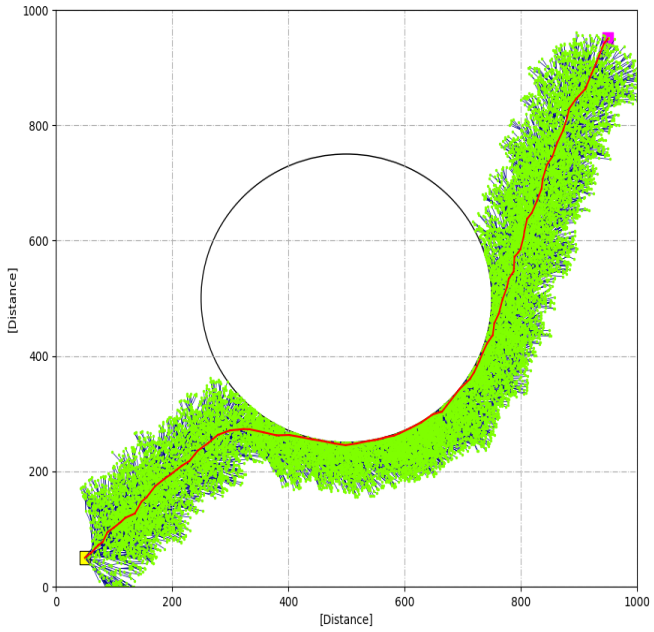

Figure 136: MiniRRT with a 1000×1000 Graph with one obstacles

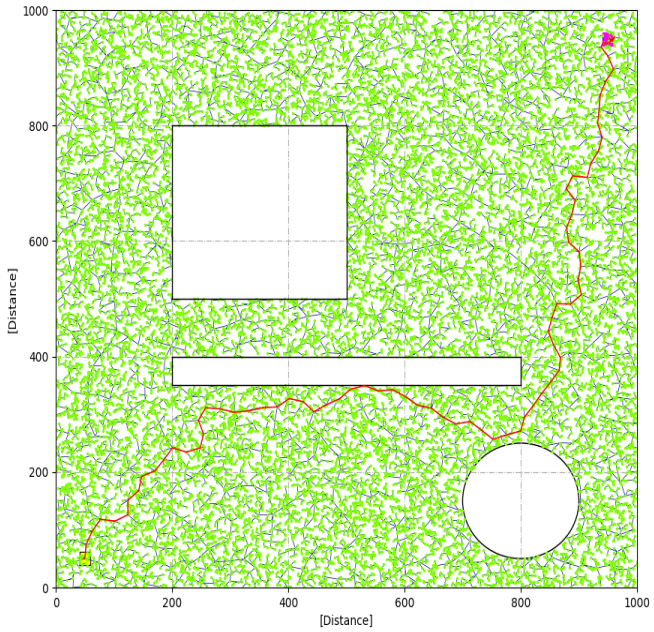

Figure 137: RRT with a 1000×1000 Graph with three obstacles

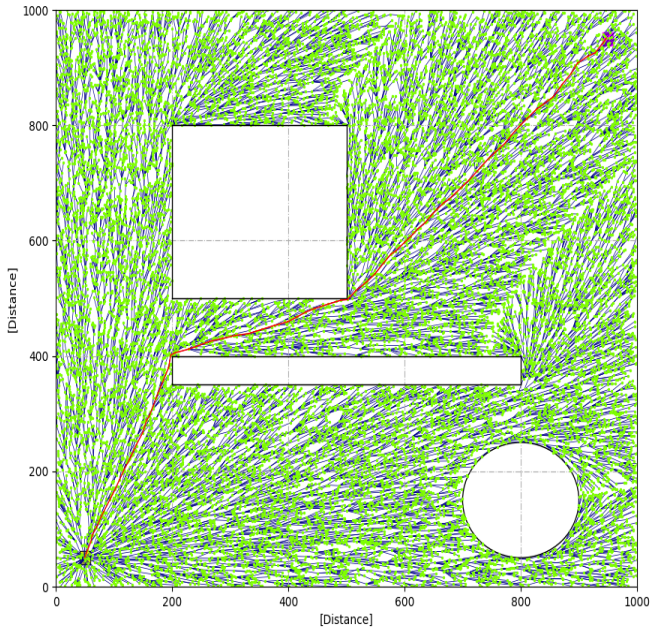

Figure 138: RRT* with a 1000×1000 Graph with three obstacles

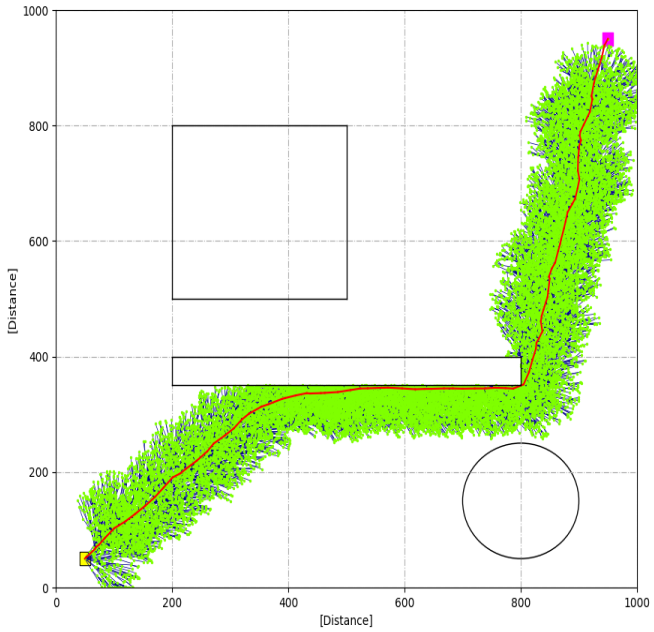

Figure 139: MiniRRT with a 1000×1000 Graph with three obstacles

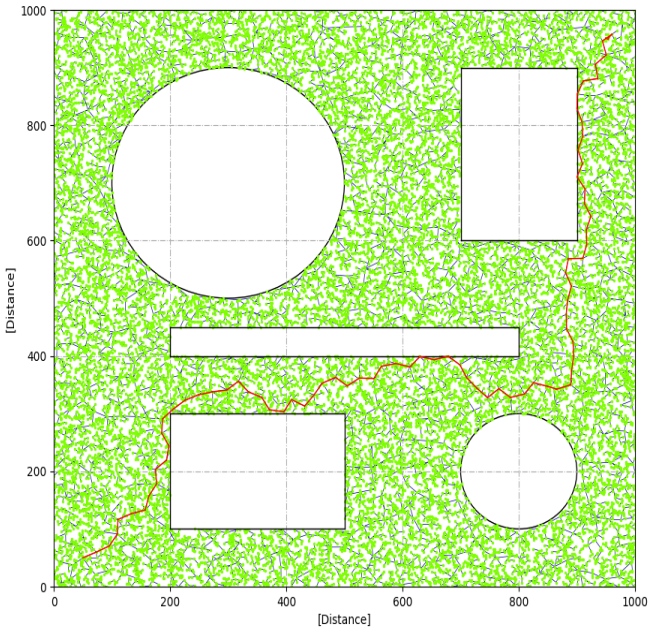

Figure 140: RRT with a 1000×1000 Graph with five obstacles

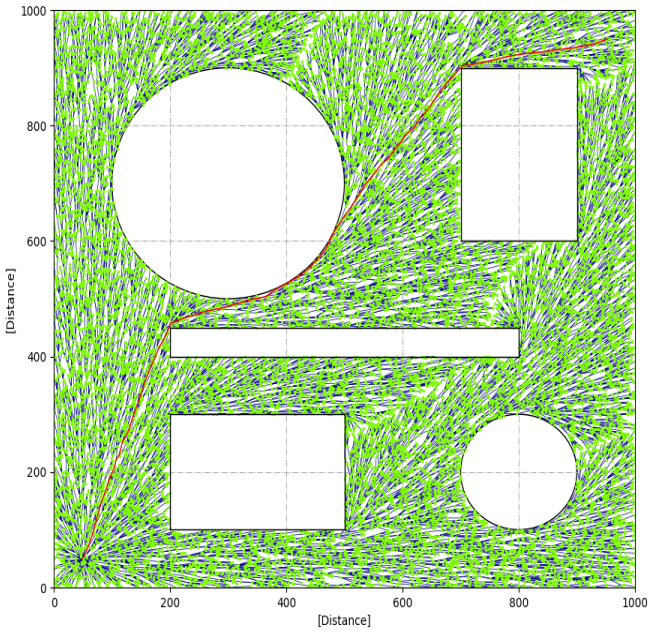

Figure 141: RRT* with a 1000×1000 Graph with five obstacles

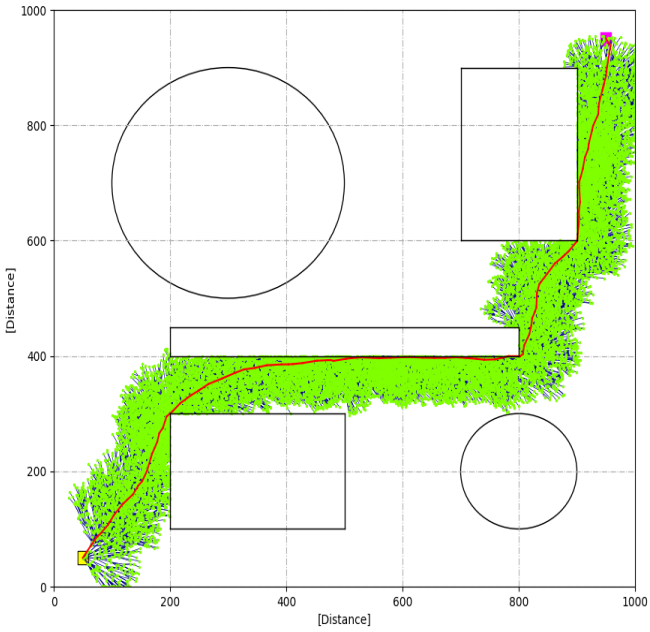

Figure 142: MiniRRT with a 1000×1000 Graph with five obstacles

```
import numpy as np
from .utils import dist, angle, bspline


class BaseRRT:
    """
    Base class for rapidly-exploring random trees.
    """
    def __init__(self, graph, x_init, x_goal, delta, k):
        # initialize variables
        self.graph = graph
        self.x_init = x_init
        self.x_goal = x_goal
        self.delta = delta
        self.k = k
```

Listing 1: BaseRRT Class

```
    def nearest(self, v, r):
        """
        Finds the nearest neighbor to the node 'v' withing a ball radius 'r'.
        """
        # defining the minimum distance to infinity
        min_dist = float('inf')
        # find nodes within the radius 'r'
        nodes = self.near(v, r)
        for node in nodes:
            # check if the distance between the node 'v' and node is less than
            # the minimum distance
            if dist(v, node) <= min_dist:
                min_dist = dist(v, node)
                nearest = node
        # sometimes the k factor is not large enough to find nodes
        try:
            return nearest
        except UnboundLocalError:
            raise UnboundLocalError("Could not find a nearest node within radius.")
```

Listing 2: Nearest Function

```
    def near(self, pivot, radius):
        """
```

```
        Checks if nearby nodes are within a defined radius.
        """
        pivot_x, pivot_y, pivot_z = pivot
        nodes = []
        for node in self.graph._node:
            x, y, z = node
            # check if node points are within the ball's radius
            if ((x - pivot_x)**2 + (y - pivot_y)**2 + (z - pivot_z)**2) <= radius**2:
                nodes.append((x, y, z))
        return nodes
```

Listing 3: Near Function

```
    def brute_force(self, x, bunch):
        """
        Finds the nearest node to x from bunch.
        """
        # set minimum distance to infinity
        min_dist = float('inf')
        # loop through nodes
        for n in bunch:
            # check if distance between nodes is less than the minimum distance overall
            if dist(x, n) < min_dist:
                # set new minimum distance
                min_dist = dist(x, n)
                # set nearest node
                nearest = n
        try:
            return nearest
        except TypeError:
            raise TypeError('Bunch list is empty.')
```

Listing 4: Brute Force Function

```
    def steer(self, x_nearest, x_rand):
        """
        Guides the edge from x_nearest to x_rand.
        """
        # check if distance of nodes is greater than delta
        if dist(x_nearest, x_rand) > self.delta:
            # saturate and bound the distance between the two nodes within the graph
            return self.bound_point(self.saturate(x_nearest, x_rand, self.delta))
```

```
        else:
            # bound the edge to within graph
            return self.bound_point(x_rand)
```

Listing 5: Steer Function

```
    def bound_point(self, node):
        """
        Bounds a node to within the graph.
        """
        node = np.maximum(node, self.graph.area[:, 0])
        node = np.minimum(node, self.graph.area[:, 1])
        return tuple(node)
```

Listing 6: Bound Point Function

```
    def saturate(self, v, w, delta):
        """
        Shortens the distance between nodes 'v' and 'w'.
        """
        start, end = np.array(v), np.array(w)
        z = end - start
        # calculates the unit vector of v and w
        u = z / np.sqrt((np.sum(z**2)))
        saturated_node = start + u*delta
        return tuple(saturated_node)
```

Listing 7: Saturate Function

```
    def shrinking_ball_radius(self):
        """
        Radius used to find nearest nodes.
        Shrinks as the number of nodes increase.
        """
        # number of graph dimensions
        d = self.graph.dimensions
        # calculates the lebesque measure, https://en.wikipedia.org/wiki/Lebesgue_measure
        leb_meas = (self.graph.area[0][1] - self.graph.area[0][0]) * (self.graph.area[1][1] -
            self.graph.area[1][0])
        # volume of unit ball
        zeta_D = (4.0/3.0) * self.delta**3
        # gamma variable
        gamma = (2**d * (1 + (1/d)) * leb_meas)**self.k
```

```
        # shrinking radius
        r = int(((gamma/zeta_D) * (np.log10(self.graph.num_nodes())/self.graph.num_nodes())))
            **(1/d)
        if r == 0.0:
            r = self.graph.area[0][1] + self.graph.area[1][1]
        return r
```

Listing 8: Shrinking Ball Radius Function

```
    def parent(self, v):
        """
        Returns the parent of node 'v'.
        """
        if list(self.graph.predecessor(v)) == []:
            return None
        return list(self.graph.predecessor(v))[0]
```

Listing 9: Parent Function

```
    def children(self, v):
        """
        Returns the children of node 'v'.
        """
        return list(self.graph.successors(v))
```

Listing 10: Children Function

```
    def is_leaf(self, v):
        """
        Checks if node 'v' is a leaf, ie a node with no children.
        """
        if self.graph.degree(v) == 0:
            return True
        return False
```

Listing 11: Is Leaf Function

```
    def depth(self, v):
        """
        Checks the depth of the node 'child'.
        The number of nodes away from the root.
        """
        # set depth to zero
```

```
        node_depth = 0
        # loop through parents of node 'v' until root is reached
        while self.parent(v) != self.x_init:
            # add to depth
            node_depth += 1
            # set 'v' to parent of 'v'
            v = self.parent(v)
        return node_depth
```

Listing 12: Depth Function

```
    def g(self, v):
        """
        Calculates cost to between node 'v' and goal.
        """
        return dist(v, self.x_goal)
```

Listing 13: G Function

```
    def cost(self, v):
        """
        Calculates the cost from node to node until the root node is reached.
        """
        cost = 0
        # Loop until root node is reached
        while v != self.x_init:
            # add cost of node to overall cost
            cost += self.graph._node[v]
            # go to parent of current node
            v = self.parent(v)
        return cost
```

Listing 14: Cost Function

```
    def compute_trajectory(self):
        """
        Finds the best path to the goal.
        """
        leaves = []
        # grab all nodes in graph
        for n in self.graph._node:
            # checks if node is a leaf
            if self.is_leaf(n):
```

```
                leaves.append(n)
        # find best leaf of the tree that is closest to the goal
        goal = self.best_leaf(leaves)
        # construct a path to the goal
        path = bspline(self.construct_path(self.x_init, goal), n=10)
        return path
```

Listing 15: Compute Trajectory Function

```
    def best_leaf(self, leaves):
        """
        Finds the best leaf in the group of leaves.
        """
        # set the best cost to inifinity
        best = float('inf')
        # if goal in graph return goal
        if self.x_goal in self.graph._node:
            return self.x_goal
        # loop through leaves in graph
        for leaf in leaves:
            # calculate cost between leaf and goal
            cost = self.g(leaf) #+ self.cost(leaf)
            # if cost is less then best update goal node and best cost
            if cost <= best:
                best = cost
                goal = leaf
        return goal
```

Listing 16: Best Leaf Function

```
    def construct_path(self, start, end):
        """
        Constructs a path rootward from start to end.
        """
        # add the end node to the path
        path = [end]
        # mark end node as the child
        child = end
        # if end and start node are the same return the path
        if start == end:
            return path
        # loop through nodes while root is not part of the path
```

```
        while start not in path:
            path.append(self.parent(child))
            child = self.parent(child)
        # reverse the path to go from root to leaf closest to goal
        path.reverse()
        return path
```

Listing 17: Construct Path Function

```
    def connect_to_goal(self, v):
        """
        Attempt to connect goal to tree if node is close enough.
        """
        # check if goal node is already in graph
        if self.x_goal not in self.graph._node:
            # check if distance between node 'v' and goal node is less than delta
            if dist(v, self.x_goal) <= self.delta/2:
                # add goal node to nodes and edges in graph
                self.graph.add_node(self.x_goal, dist(v, self.x_goal))
                self.graph.add_edge(v, self.x_goal)
        else:
            # check if node'v' is closer than parent of the goal node
            if dist(v, self.x_goal) <= dist(self.parent(self.x_goal), self.x_goal):
                # change parent to node 'v'
                self.graph.remove_edge(self.parent(self.x_goal), self.x_goal)
                self.graph.add_edge(v, self.x_goal)
```

Listing 18: Connect To Goal Function

```
class MiniRRT(BaseRRT):
    """
    Class for optimal imcrementail RRT.
    This class uses potential fields, so pgraph is needed.
    """
    def __init__(self, graph, x_init, x_goal, delta, k, path, heading):
        self.alpha = np.radians(120)
        self.range = delta*4
        self.path = path
        self.heading = heading
        super().__init__(graph, x_init, x_goal, delta, k)
```

Listing 19: MiniRRT Class

```
    def sample_free(self):
        """
        Returns a random node within the defined bounds of the graph.
        """
        r = self.range * np.sqrt(np.random.uniform())
        if self.path is None:
            theta = np.random.uniform() * self.alpha + (self.heading - self.alpha/2)  # initial
                tree sample free, bounds to angle
        else:
            # trees after sample free, bounds to angle
            theta = np.random.uniform() * self.alpha + (angle(self.path[0], self.path[1]) - self.
                alpha/2)
        # random tuple
        return tuple((self.x_init[0] + r * np.cos(theta), self.x_init[1] + r * np.sin(theta),
            self.graph.area[2][0]))
```

Listing 20: Sample Free Function

```
    def extend(self, x_new, x_nearest):
        """
        Expands the tree out into the frontier.
        """
        X_near = self.near(x_new, self.delta)
        self.graph.add_node(x_new, dist(x_nearest, x_new))
        self.find_parent(x_new, x_nearest, X_near)
        return X_near
```

Listing 21: Extend Function

```
    def find_parent(self, x_new, x_min, X_near):
        c_min = self.cost(x_min) + dist(x_min, x_new)
        for x_near in X_near:
            if self.graph.collision_free(x_near, x_new) and self.cost(x_near) + dist(x_near,
                x_new) < c_min:
                x_min = x_near
                c_min = self.cost(x_near) + dist(x_near, x_new)
        self.graph.add_edge(x_min, x_new)
        self.graph._node[x_new] = dist(x_min, x_new)
```

Listing 22: Find Parent Function

```
    def rewire_neighbors(self, x_new, X_near):
        """
```

```
        Rewires the tree to nodes that are closer to the new node.
        """
        for x_near in X_near:
            if self.graph.collision_free(x_new, x_near) and self.cost(x_new) + dist(x_new, x_near
                ) < self.cost(x_near):
                x_parent = self.parent(x_near)
                self.graph.remove_edge(x_parent, x_near)
                self.graph.add_edge(x_new, x_near)
                self.graph._node[x_near] = dist(x_new, x_near)
```

Listing 23: Rewire Neighbors Function

```
    def search(self):
        """
        Function to search through the graph.
        """
        r = self.shrinking_ball_radius()
        x_rand = self.sample_free()
        x_nearest = self.nearest(x_rand, r)
        x_new = self.steer(x_nearest, x_rand)
        if self.graph.collision_free(x_nearest, x_new):
            X_near = self.extend(x_new, x_nearest)
        if x_new in self.graph._node.keys():
            self.rewire_neighbors(x_new, X_near)
        self.connect_to_goal(x_new)

        return self.compute_trajectory()
```

Listing 24: Search Function

```
from shapely.geometry import LineString, Point


class Graph:

    def __init__(self, area, obstacles=[]):
        self.graph_attr_dit_factory = dict
        self.node_dict_factory = dict
        self.node_attr_dict_factory = dict
        self.adjlist_outer_dict_factory = dict
        self.adjlist_inner_dict_factory = dict
        self.edge_list_factory = list
        self.edge_attr_dict_factory = dict

        self.graph = self.graph_attr_dit_factory()
        self._node = self.node_attr_dict_factory()
        self._adj = self.adjlist_outer_dict_factory()
        self._pred = self.adjlist_inner_dict_factory()
        self._succ = self._adj
        self._edge = self.edge_list_factory()

        self.nodes = [*self._node]
        self.edges = self._edge

        self.area = area
        self.dimensions = len(area)
        self.obstacles = obstacles
```

Listing 25: Graph Class

```
    def add_node(self, v, cost):
        """
        Adds node 'v' to the graph.
        """
        if v not in self._succ:
            self._succ[v] = self.adjlist_inner_dict_factory()
            self._pred[v] = self.adjlist_inner_dict_factory()
            # Add cost to corresponding node in the node dictionary
            self._node[v] = cost
```

Listing 26: Add Node Function

```
    def remove_node(self, v):
        """
        Removes node 'v' from the graph.
        """
        try:
            nbrs = self._succ[v]
            del self._node[v]
        except NameError:
            raise NameError("The node {v} is not in the graph.".format(v=v))
        for u in nbrs:
            del self._pred[u][v]
        del self._succ[v]
        for u in self._pred[v]:
            del self._succ[u][v]
        del self._pred[v]
```

Listing 27: Remove Node Function

```
    def num_nodes(self):
        """
        Returns the number of nodes in the graph.
        """
        return len(self._node.keys())
```

Listing 28: Number of Nodes Function

```
    def add_edge(self, u, v):
        """
        Adds edge 'u-v' to the graph.
        """
        if u not in self._node:
            self._succ[u] = self.adjlist_inner_dict_factory()
            self._pred[u] = self.adjlist_inner_dict_factory()
        if v not in self._node:
            self._succ[v] = self.adjlist_inner_dict_factory()
            self._pred[v] = self.adjlist_inner_dict_factory()
        if (u, v) not in self._edge:
            self._edge.append((u, v))
        datadict = self._adj[v].get(v, self.edge_attr_dict_factory())
        self._succ[u][v] = datadict
        self._pred[v][u] = datadict
```

Listing 29: Add Edge Function

```
    def remove_edge(self, u, v):
        """
        Removes edge 'u-v' from the graph.
        """
        try:
            del self._succ[u][v]
            del self._pred[v][u]
            self._edge.remove((u, v))
        except NameError:
            raise NameError("The edge {u}--{v} is not in the graph.".format(u=u, v=v))
```

Listing 30: Remove Edge Function

```
    def num_edges(self):
        """
        Returns the number of edges in the graph.
        """
        return len(self._edge)
```

Listing 31: Number of Edges Function

```
    def neighbors(self, v):
        """
        Returns the neighbors of node 'v'.
        """
        try:
            return iter(self._adj[v])
        except KeyError:
            raise KeyError("The node {v} is not in the graph.".format(v=v))
```

Listing 32: Neighbors Function

```
    def successors(self, v):
        """
        Returns the successors of node 'v':
        """
        try:
            return iter(self._adj[v])
        except KeyError:
            raise KeyError("The node {v} is not in the graph.".format(v=v))
```

Listing 33: Successors Function

```
    def predecessor(self, v):
        """
        Returns the predecessor of node 'v'.
        """
        try:
            return iter(self._pred[v])
        except KeyError:
            raise KeyError("The node {v} is not in the graph.".format(v=v))
```

Listing 34: Predecessor Function

```
    def degree(self, v):
        """
        Returns the number of successors of node 'v'.
        """
        nbrs = self._succ[v]
        return len(nbrs) + (v in nbrs)
```

Listing 35: Degree Function

```
    def clear(self):
        """
        Removes all nodes and edges from the graph.
        """
        self._succ.clear()
        self._pred.clear()
        self._node.clear()
        self._edge.clear()
        self.graph.clear()
```

Listing 36: Clear Function

```
    def obstacle_free(self, v):
        """
        Checks if the node 'v' is free of an obstacle.
        """
        node = Point(v)
        # check if node in obstacle
        return not any(node.within(obstacle) for obstacle in self.obstacles)
```

Listing 37: Obstacle Free Function

```
    def collision_free(self, u, v):
        """
        Checks if the edge 'u -- v' is free of collision.
        """
        edge = LineString([u, v])
        # check if edge intersects with obstacle
        return not any(edge.intersects(obstacle) for obstacle in self.obstacles)
```

Listing 38: Collision Free Function

```
import rospy
import numpy as np
import time
import datetime

from mavros_msgs.msg import WaypointList, Waypoint
from mavros_msgs.srv import WaypointPush
from mavros_msgs.msg import HomePosition
from std_msgs.msg import Float64
from sensor_msgs.msg import NavSatFix
from nav_msgs.msg import Odometry

from shapely.geometry.point import Point

from graph import Graph
from algorithms import MiniRRT
from utils import dist


class PathPlanning:
    def __init__(self):
        rospy.loginfo_once('INITIALIZING')
        rospy.init_node('RRTnode')
        rospy.wait_for_service('/control/waypoints')

        # initialize variables
        self.home_pos_data = np.zeros(shape=(3,))
        self.position = np.zeros(shape=(3, 1))
        self.heading = None
        self.home_set = False
        self.gps_status = None
```

Listing 39: Path Planning Class

```
    def home_pos_cb(self, data):
        if (np.round(data.geo.latitude, 2) != np.round(self.home_pos_data[0], 2) or
                np.round(data.geo.longitude, 2) != np.round(self.home_pos_data[1], 2) or
                np.round(data.geo.altitude, 2) != np.round(self.home_pos_data[2], 2)):
            print("CALLBACK\n")
            self.home_pos_data[0] = data.geo.latitude
            self.home_pos_data[1] = data.geo.longitude
            self.home_pos_data[2] = data.geo.altitude
```

```
9
10          rospy.loginfo('Reference set\n')
11          self.home_set = True
```

Listing 40: Home Position Callback Function

```
1   def global_heading(self, data):
2       self.heading = data
```

Listing 41: Global Heading Function

```
1   def global_position(self, data):
2       self.gps_status = data.status.status
3       if self.gps_status >= -1: # needs to be >=0
4           self.position[0] = data.latitude
5           self.position[1] = data.longitude
6           self.position[2] = data.altitude
```

Listing 42: Global Position Function

```
1   def local_position(self, data):
2       X = data.pose.pose.position.y
3       Y = data.pose.pose.position.x
4       Z = -1*data.pose.pose.position.z
```

Listing 43: Local Position Function

```
1   def main(self):
2       """
3       dims : graph's dimensions (numpy array of tuples)
4       obstacles : obstacles to be placed in the graph (list of shapely.Polygons)
5       home : home location (tuple)
6       init_state : start location (tuple)
7       goal_state : goal location (tuple)
8       delta : distance between nodes (int)
9       k : shrinking ball facotr (int)
10      n : number of waypoints to push (int)
11      path : list of waypoints (list of tuples)
12      case : case number to set behavior (int)
13      Units: meters
14      """
15
16      wp_push = rospy.ServiceProxy('/control/waypoints', WaypointPush)
```

```
17
18          # subscriber to get home position of aircraft
19          rospy.Subscriber('/mavros/home_position/home', HomePosition, self.home_pos_cb)
20          # subscriber to get the compass heading of aircraft
21          rospy.Subscriber('/mavros/global_position/compass_hdg', Float64, self.global_heading)
22          # subscriber to get the global positioning of aircraft
23          rospy.Subscriber('/mavros/global_position/global', NavSatFix, self.global_position)
24          # subscriber to get the local positioning of aircraft
25          rospy.Subscriber('/mavros/global_position/local', Odometry, self.local_position)
26
27          rate = rospy.Rate(1)  # send msgs at 1 Hz
28          start_time = rospy.get_time()  # start rospy timer
29          rate.sleep()  # sleep for 1 second
30
31          # Eagle Map
32          #               N-x      E-y       D-z
33          dims = np.array([(-10, 80), (-120, 5), (-10, -10)])  # dimensions of the graph
34          obstacle = [(Point(35, -70).buffer(20))]  # define obstacles in graph
35          goals = [(60, -100, -10), (0, 0, -10)]  # goals of the graph
36          count = 0  # count for goals
37          init = self.pos  # initial positioning
38          goal = goals[count]  # goal position
39          delta = 5  # edge saturation distance
40          k = 2  # shrinking factor
41
42          '''
43          Nano Talon Map
44          dims = np.array([(-500, 2500), (-100, 2900), (-50, -50)])
45          obstacle = [(Point(1300, 1300).buffer(500))]
46          init = self.pos
47          goal = (2300.0, 2600.0, -50.0)
48          delta = 100
49          k = 2
50          '''
51
52          graph = Graph(dims, obstacle)
53          path = None
54
55          trail = []
56          position = []
57          nodes = []
```

```
        edges = []

        start_index = 0

        # logging ---------------------------------
        datestamp = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        filename = '/home/r2/logs/flight_log' + datestamp + '.txt'

        f = open(filename, 'w+')
        f.write('Log StartTime : {}\n'.format(datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S
            ')))
        f.flush()
        # -----------------------------------------

        rospy.loginfo("Calculating Trajectory...")
        t0 = time.time()  # start timer for search algorithm

        while not rospy.is_shutdown():
            if graph.num_nodes() == 0:  # check for node count
                # initializing MiniRRT algorithm
                rrt = MiniRRT(graph, init, goal, delta, k, path, self.heading)
                # add root node to graph
                graph.add_node(init)
            if graph.num_nodes() <= 250:  # check for node count
                # run the search algorithm
                path = rrt.search()
            else:
                A = rrt.brute_force(self.pos, path)  # find the nearest node in the path to the
                    aircraft
                B = path.index(A)  # find the index of the closest waypoint to the aircraft
                init = path[B+1]  # add one to the index

                trail.append(path)  # add the path to the overall trail
                position.append(self.pos)  # add the aircraft position to the list of positions
                nodes.append(list(graph._node))  # add nodes to the list of nodes
                edges.append(list(graph._edge))  # add edges to the list of edges

                rate.sleep()  # sleep for 1 second
                graph.clear()  # clear the graph

                pathNED = np.asarray(path)  # set path to a numpy array data type
```

```
                wp_msg = []  # initilize the waypoint list

                for i in range(len(pathNED)):
                    wp_point = Waypoint()  # call waypoint function
                    wp_point.frame = 1  # 3

                    wp_point.x_lat = pathNED[i][0]  # pathNED[0][i]
                    wp_point.y_long = pathNED[i][1]  # pathNED[1][i]
                    wp_point.z_alt = pathNED[i][2]  # pathNED[2][i]
                    wp_msg += [wp_point]  # add waypoint to list

                resp = wp_push(start_index, wp_msg)  # push waypoints to aircraft
                print(resp, '\n')

                start_index += path.index(rrt.brute_force(self.pos, path))+1  # set waypoint
                    index
                rate.sleep()  # sleep for 1 second

                if goal in path:  # check if goal is in path
                    while dist(self.pos, goal) > delta:  # check if aircraft position is delta
                        away from goal
                        rate.sleep()  # sleep for 1 second
                        position.append(self.pos)  # add aircraft position to positions list
                    count += 1  # add one to the count

                    if count >= len(goals):  # check if count is equal to the number of goal
                        points
                        rospy.loginfo("Final Goal Reached")

                        # logging ----------------------------------
                        f.write('Log EndTime : {}\n'.format(datetime.datetime.now().strftime("%Y
                            -%m-%d %H:%M:%S")))
                        f.write('Run Time : {} secs'.format(round(time.time()-t0,3)))
                        f.write('Paths :  {}\n'.format(trail))
                        f.write('Positions :  {}\n'.format(position))
                        f.write('Nodes :  {}\n'.format(nodes))
                        f.write('Edges :  {}\n\n'.format(edges))
                        f.flush()
                        f.close()
                        # ------------------------------------------
                        break # Break out of while loop
```

```
134
135                     init = goal  # set root node to goal location
136                     graph.clear()  # clear the graph
137                     goal = goals[count]  # set goal to next goal in list
138
139                     rospy.loginfo("Goal Updated")
```

Listing 44: Main Function

```
def dist(p, q):
    """
    Finds the Euclidean distance between two points.
    """
    return math.sqrt(sum([(a - b)**2 for a, b in zip(p, q)]))
```

Listing 45: Distance Function

```
def angle(a, b):
    """
    Finds the angle between the horizon and the points a and b.
    """
    x1, y1, z1 = a
    x2, y2, z2 = b
    return np.arctan2(y2 - y1, x2 - x1)
```

Listing 46: Angle Function

```
def unit_vector(angle):
    """
    Returns the unit vector of angle.
    """
    return np.array([np.cos(angle), np.sin(angle), 0], dtype="float")
```

Listing 47: Unit Vector Function

```
def bspline(cv, n=100, degree=3, periodic=False):  # Used for path smoothing
    """ Calculate n samples on a bspline
        cv :      Array ov control vertices
        n  :      Number of samples to return
        degree:   Curve degree
        periodic: True - Curve is closed
                  False - Curve is open
    """

    # If periodic, extend the point array by count+degree+1
    cv = np.asarray(cv)
    count = len(cv)

    if periodic:
        factor, fraction = divmod(count+degree+1, count)
        cv = np.concatenate((cv,) * factor + (cv[:fraction],))
        count = len(cv)
```

```
        degree = np.clip(degree,1,degree)

    # If opened, prevent degree from exceeding count-1
    else:
        degree = np.clip(degree,1,count-1)

    # Calculate knot vector
    kv = None
    if periodic:
        kv = np.arange(0-degree,count+degree+degree-1)
    else:
        kv = np.clip(np.arange(count+degree+1)-degree,0,count-degree)

    # Calculate query range
    u = np.linspace(periodic,(count-degree),n)

    # Calculate result
    return list(np.array(si.splev(u, (kv,cv.T,degree))).T)
```

Listing 48: B-Spline Function

VITA

Nicholas William Rozell

Candidate for the Degree of

Master of Science

Thesis: IMPLEMENTATION OF A LOCAL PATH PLANNING ALGORITHM FOR UNMANNED AERIAL VEHICLES

Major Field: Mechanical and Aerospace Engineering

Biographical:

Education:

Completed the requirements for the Master of Science in Mechanical and Aerospace Engineering at Oklahoma State University, Stillwater, Oklahoma in July, 2021.

Completed the requirements for the Bachelor of Science in Aerospace Engineering at Undergraduate Oklahoma State University, Stillwater, OK in 2019.

Completed the requirements for the Bachelor of Science in Mechanical Engineering at Undergraduate Oklahoma State University, Stillwater, OK in 2019.

Professional Membership:

American Institute of Aeronautics and Astronautics