Online Unmanned Ground Vehicle Mission Planning using Active Aerial Vehicle Exploration

Anthony J. Wagner

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Masters of Science

in

Mechanical Engineering

May 15, 2019

Blacksburg, Virginia

Keywords: UAV, UGV, Exploration, RRT

Online Unmanned Ground Vehicle Mission Planning using Active Aerial Vehicle Exploration

Anthony J. Wagner

(ABSTRACT)

This work presents a framework for the exploration and path planning for a collaborative UAV and UGV system. The system is composed of a UAV with a stereo system for obstacle detection and a UGV with no sensors for obstacle detection. Two exploration algorithms were developed to guide the exploration of the UAV. Both identify frontiers for exploration with the Dijkstra Frontier method using Dijkstra's Algorithm to identify a frontier with unknown space, and the other uses a bi-directional RRT to identify multiple frontiers for selection. The final algorithm developed was for to give the UGV intermediate plans when an entire plan is not yet found. This improves the overall mission tempo. The algorithm is designed to keep the UGV a safe distance from the unknown frontier to prevent backtracking. All the algorithms were tested in Gazebo using the ROS framework. The Dijkstra Frontier method was also tested on the hardware system. The results show the RRT Explore algorithm to work well for exploring the environment, performing equally or better than the Dijkstra Frontier method. The UGV intermediate plan method showed a decreased traveled distance for the UGV but increases in UGV mission time with more conservative distances from danger. Overall, the framework showed a good exploration of the environment and performs the intended missions.

Online Unmanned Ground Vehicle Mission Planning using Active Aerial Vehicle Exploration

Anthony J. Wagner

(GENERAL AUDIENCE ABSTRACT)

This work presents a framework for the exploration and path planning for a collaborative aerial and ground vehicle robotic system. The system is composed of an aircraft with a camera system for obstacle detection and a ground vehicle with no sensors for obstacle detection. Two exploration algorithms were developed to guide the exploration of the aircraft. Both identify frontiers for exploration with the Dijkstra Frontier method using path planning algorithms to identify a frontier with unknown space (Dijkstra Frontier), and the other uses a sampling based path planning method (RRT Explore) to identify multiple frontiers for selection. The final algorithm developed was for to give the ground vehicle intermediate plans when an entire plan is not yet found. The algorithm is designed to keep the ground vehicle a safe distance from the unknown frontier to prevent backtracking. All the algorithms were tested in a simulation framework using Robot Operating System and one exploration method was tested on the hardware system. The results show the RRT Explore algorithm to work well for exploring the environment, performing equally or better than the Dijkstra Frontier method. The ground vehicle intermediate plan method showed a decreased traveled distance for the ground vehicle but increases in ground vehicle mission time with more conservative distances from danger. Overall, the framework showed a good exploration of the environment and performs the intended missions.

Acknowledgments

I would like to thank all the people who have helped me through the work that went into this thesis. First, I want to thank my advisor Dr. Kochersberger for giving me this opportunity and being a great advisor. I would also like to thank the rest of my committee. I want to thank James Donnelly for providing great help and good company while working on this project, John Peterson for being the go-to guy for robotics and programming related questions, and Haseeb Chaudhry for advice and being great for bouncing ideas off. I would also like to thank Drew Morgan for being a great resource for UAV related questions and running the lab. Lastly, I would like to thank my friends and family for all the support and encouragement they have given to me along the way.

Contents

List of Figures

Chapter 1

Introduction

With powerful computers becoming smaller and lighter, the ability to do more on-board sensing and autonomy with UAVs is quickly improving. This has allowed for stereo imaging and autonomous decision making on-board the aircraft.

Ground vehicles are vehicles that operate on the ground to perform a delivery or sensing mission without a person on-board. Typically the lack of a person is due to some task requirement that involves hazardous environments, heavy weight, size limitations, or long endurance missions. The lack of a person means the vehicle is either teleoperated or autonomous. This presents a large challenge especially with developing autonomy to replace teleoperation. One application of UGVs is reconnaissance, surveillance, and target acquisition (RSTA). This is important to military operators as it allows commanders to have additional sensing capabilities that can go into areas that would endanger human lives. Another application is security. This is similar to RSTA but in known environments, so the hardware and control can be better tuned for the security application [7].

According to a National Research Council on Army UGV development, increasing the levels of autonomy in these vehicles would ”greatly expand the list of military uses.” This same report also stated that autonomous mobility from point A to B is crucial to the ultimate success and acceptance of UGVs in the military [4].

Traditionally, when an autonomous ground vehicle is navigating from point A to point B it uses on-board sensing and a path planning algorithm like D* to navigate. In certain systems, this can be inefficient as most small ground vehicles move slower than air vehicles can. Using a UAV to provide sensing and routing to a ground vehicle takes advantage of the speed and better perspective the vehicle gets from being in the air. Since the UAV is in the air, it can fly above obstacles meaning it has much more freedom in route planning which helps it explore faster. Additionally UAVs also have much higher top speeds than UGVs which gives it another advantage. The advantage of an aerial perspective is especially useful in a post disaster environment such as after a tornado as seen in Figure 1.1. The large amount of debris scattered about means a priori information would possibly not be accurate and thus would not provide a good route for a ground vehicle. Since a UGV traveling over debris is very slow the aerial perspective would mean a larger area could be explored in the same amount of time and a better path could be generated for the ground vehicle.

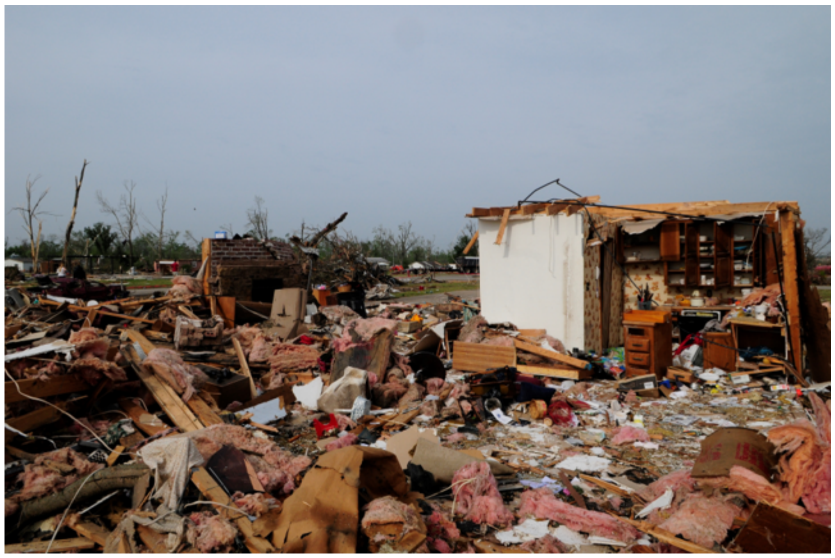


Figure 1.1: A disaster area that a priori information would be out of date due to the large amount of debris [5].

This work focuses on how to integrate an effective and coordinated approach to explore an unknown environment to generate a route for a UGV. This involves the development and bench-marking of two different exploration methods and a method of allowing the UGV to move before a complete route is generated to maintain operation tempo.

Chapter 2

Review of Literature

2.1 Path Planning Algorithms

2.1.1 Dijkstra's Algorithm

Dijkstra's Algorithm is an algorithm that finds the shortest route between two nodes in a discrete environment. The discretized environment is in the formulation of a graph. Figure 2.1 shows a graph for an infinite environment where you can travel up, down, left, and right from each node. The circles represent nodes and the arrows indicate a directed edge connecting the nodes. It is a search based algorithm that is both optimal and complete and is a formulation of dynamic programming.

Any path in the graph will traverse a set of nodes and the edges that connect the nodes. The total cost of a path would be the sum of the cost to traverse that edge. For this to work, all edges must have some non-negative cost associated with the edge. For each state x in the graph there exists some cost-to-come $C(x)$ that represents the optimal cost-to-come $C^*(x)$ from the starting node. To calculate the cost-to-come an incremental search is performed starting with assigning the $C^*(x_{start}) = 0$, since the cost-to-come from the start node to the goal is zero. Then for all of the neighboring nodes to the current node (has an edge from the node to the current node) a temporary cost-to-come $C(x')$ is calculated as: $C(x') = C^*(x) + l(x, u)$ where $l(x, u)$ represents the edge cost associated from coming

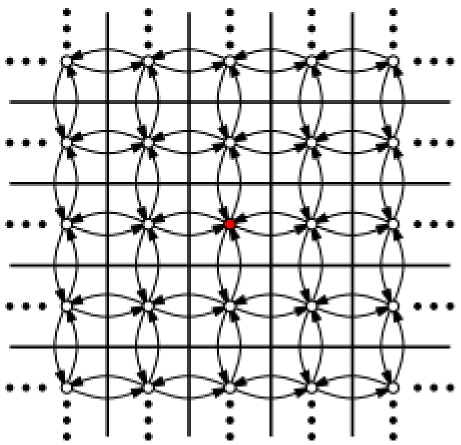


Figure 2.1: State transition graph for an infinite environment with four edges from each node in the graph. Arrows indicate directed edges and circles represent nodes for each grid cell [13].

from node x along edge u. This temporary cost-to-come represents the current best known cost-to-come but is not confirmed to be the optimal cost-to-come. These temporary cost-to-come values are stored in a priority queue called the open list Q. In each iteration of the search, the node with the current lowest temporary cost-to-come is removed from the queue and added to a closed list that stores the optimal cost-to-come for each node. Then the cycle of calculating cost-to-comes for each of its neighbors is repeated. The node with the current lowest cost-to-come in the open list is known to now be optimal because any other path would require going through another state in the open list which has higher cost-to-comes. This cycle is repeated until the termination requirement is met. The termination requirement is that the goal node is removed from Q, meaning $C^*(x_{goal})$ has been found and the optimal path through the environment can be found.

This search method has a time complexity of $O(|V|\log|V| + |E|)$, where |V| is the number of vertices in the graph and |E| is the number of edges in the graph. The main assumption of this is that the open list is implemented as a priority queue [13]. A diagram of how the expansion of nodes added to the closed list and open list progresses is shown in Figure 2.2.

The expansion from the start node (green) can be seen to be circular in the first to time snapshots with edge costs representing the Euclidean distance between the nodes. Overall Dijkstra's algorithm is a simple optimal and complete search algorithm for a discretized environment.

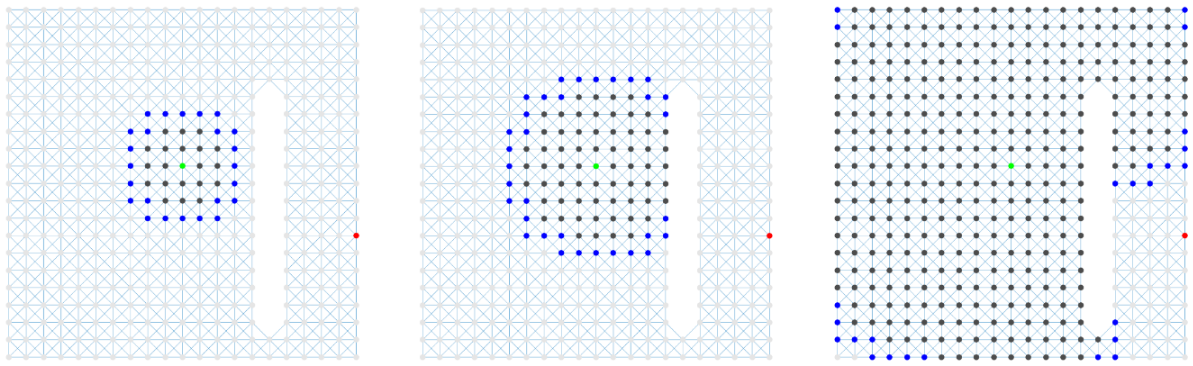


Figure 2.2: Expansion of Dijkstra's Algorithm on a sample environment at three different times in the search. Start is marked as green node and goal is marked as red node.[16].

2.1.2 A-star (A*) Search Algorithm

A* is an improvement to Dijkstra's Algorithm with the addition of a heuristic estimate of the cost-to-go (cost from to get from the current state to the goal). Adding a heuristic directs the search towards the goal and results in a reduction of the total states explored in the search. In order to maintain optimality, the heuristic must be what is called an admissible heuristic. This means that the heuristic cannot overestimate the actual cost-to-go. Normally the Euclidean distance is used as it cannot overestimate the cost since it ignores obstacles and the edge costs cannot be less than the straight line distance. The priority of the nodes in the open list is its current distance value plus the heuristic. Lets say the cost-to-go is denoted as $G^*(x)$ for a given node x and $\hat{G}^*(x)$ as the estimated cost-to-go from the heuristic function. Then the same priority queue Q is sorted by the sum: $C^*(x') + G^*(x')$. Additionally, it should be noted that if a heuristic function of 0 the algorithm becomes Dijkstra's Algorithm [13].

A diagram of how the expansion of nodes added to the closed list and open list progresses for A* is shown in Figure 2.3. This can be compared to Figure 2.2 for the Dijkstra algorithm. It can be seen very clearly that the heuristic directs the search and expands more points towards the goal for a-star. Dijkstra's shows a lot more nodes considered which is why it is not as efficient.

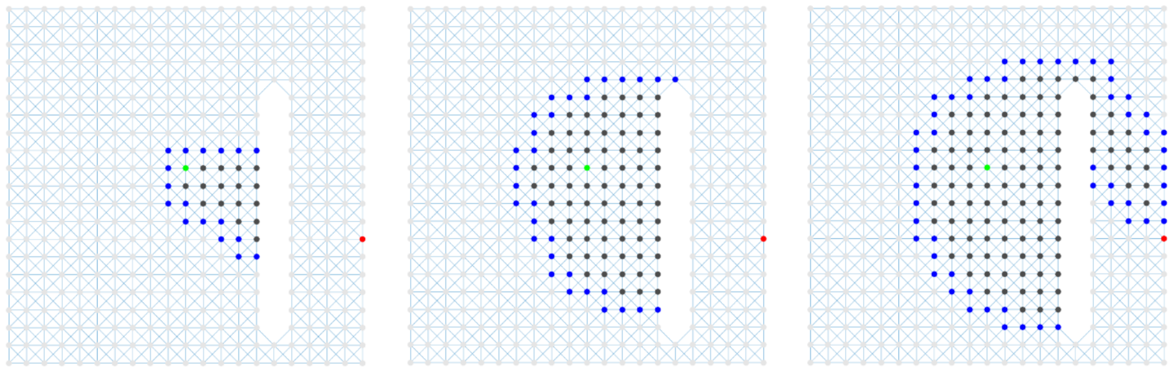


Figure 2.3: Expansion of the A* Algorithm[16].

2.1.3 D-star (D*) and D-star Lite (D* Lite)

D* is similar to A* in that it uses the same method of finding the shortest path with a heuristic but differs in that the cost values are dynamic. This means that the edge costs can change during the planning process. A* can be used in dynamic environments but each time the environment is changed A* must be run from the beginning. The advantage of using D* over A* with re-planning every time the cost values change is that D* results in less computation so it is more efficient and faster. This is advantageous in unknown environments where the map is not known a priori. This works because while the robot is navigating along the path the costs are updated when an obstacle is encountered and D* will update the costs and plan to account for this without having to entirely recompute. [15].

2.1.4 RRTs

Rapidly exploring random tree (RRT) is a sampling based path planning algorithm. While the previous planning algorithms discussed used a discrete graph to plan on the RRT uses random samples from a continuous region. The basis of the algorithm is that a tree is generated by randomly sampling a point in the configuration space and that point is connected to its nearest neighbor. To plan around obstacles a collision checking routine is in place to check that the new connection between the sampled point and its nearest neighbor is collision free before it is added to the tree. An example of an RRT exploring an empty environment is shown in Figure 2.4 for two different numbers of iterations. The first image shows 45 iterations and shows that the RRT has reached the extents of the environment not not every area within. After 2345 iterations the coverage is much denser. This is an important trait of RRTs as even low numbers of samples can completely explore the environment and as more samples are added the density of the tree increases.

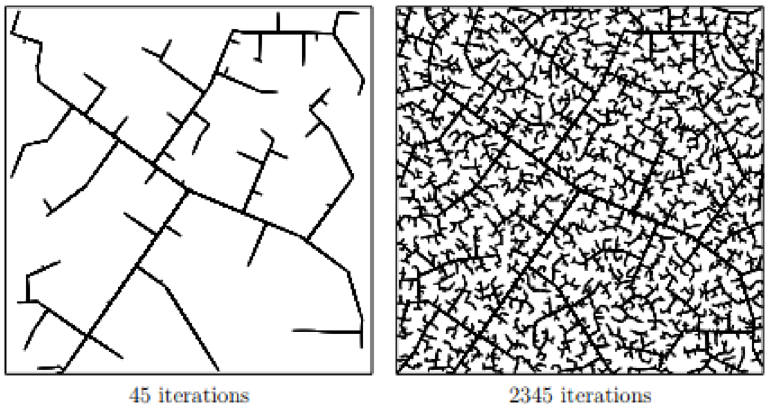


Figure 2.4: The RRT reaches all areas of the environment fairly quickly (as seen in the first image) and then with more iterations a denser coverage if the area is achieved (right image) [13].

The most computationally expensive part of this algorithm is the nearest neighbor search. For low dimensional RRTs such as the 2D used for ground robot,s a KD-tree will be more efficient than a naive search. The creation of a KD-tree with k points in n dimensions can be constructed in $O(nk * log(k))$ time with a query taking $log(k)$ time.

To use RRTs to find a path to the goal there are two main methods of searching. The first is a single tree approach that as the tree is being generated will check if the tree has reached a predefined area around the goal which is considered having found a path. The other method is to use a bidirectional search where two trees are grown from the start and goal respectively. This helps expand the tree in environments with ”bug traps” or other challenging regions that make it difficult for a tree to search in the desired direction [13]. Examples of such environments are shown in Figure 2.5. (a) shows an environment that is hard for a goal biased single directional RRT to explore around since the half circle almost entirely blocks the forward searching direction. This is also a case where biasing the sampling towards the goal can result in worse performance. (b) shows an environment where using a bi-directional RRT is very advantageous as the blue tree has a hard time getting out of the enclosure due to the low probability of sampling a point directly outside the exit. The red tree has less problems getting in due to the funnel shaped entrance. (c) is very similar to (b) but has both the red and blue tree inside the trap. Adding an additional green tree can help speed up the search to get inside the two traps. (d) shows an environment that an RRT is not suited for at all. Getting a tree to explore in or out of those traps would require an immense number of samples that would waste a lot of time and computation.

A basic RRT algorithm is shown in Algorithm 1. The first step in the RRT process is to set the root of the tree q_0, which is normally the configuration (x,y coordinates in 2D) of the robot. Then a cyclic process is repeated for k iterations or until some termination condition is met. The first step in the cyclic process is to sample a point in the environment $\alpha(i)$. This

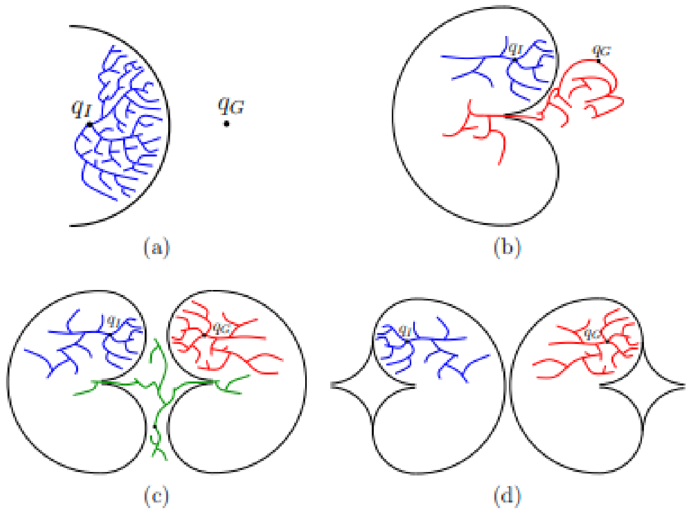


Figure 2.5: Challenging Environments for RRTs to explore due to the sampling based nature of the algorithm [13].

is added as a vertex in the tree. Next, the nearest neighbor to this point is found using a nearest neighbor search q_n. Finally an edge is added to the tree from q_n to $\alpha(i)$. This basic algorithm is shown in Algorithm 1.

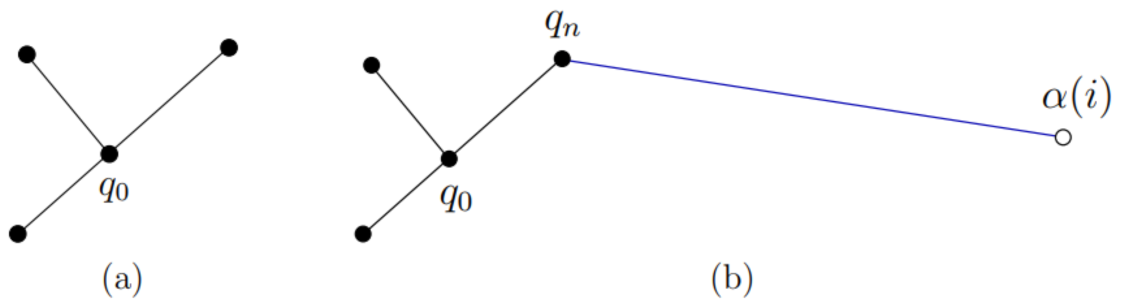


Figure 2.6: (a) shows an existing tree created by the RRT algorithm, (b) shows the sampled point ($\alpha(i)$), its nearest neighbor (q_n), and the edge connecting the two [13].

Algorithm 1 Simple RRT

```
1: G.init($q0$);
2: for $i = 1$ to $k$ do
3:     G.add vertex($\alpha$(i));
4:     $qn \leftarrow$ nearest($S(G)$,$\alpha$(i))
5:     G.add edge($qn$, $\alpha(i)$);
```

2.2 Exploration Methods

Miguel Juliá et al defined autonomous exploration as “The ability of mobile robots to autonomously travel around an unknown environment gathering the necessary information to produce a useful map for navigation.” Exploration has also been loosely used with the problem of path planning [11]. For this thesis, exploration is defined as the exploration algorithms that are being used to obtain information for navigation not just path planning. Within exploration, there are several different areas of interest. Some exploration methods are focused on minimizing exploration time, others focus on developing better maps. Exploration techniques can also depend on the number of vehicles in the system. There has been a lot of work looking at multi-vehicle exploration and leveraging the additional vehicles [3] [6] [12].

2.2.1 Frontier Exploration

One method of exploring unknown environments is the Frontier based method introduced by Brian Yamauchi. He introduced the concept of frontiers as it applies to autonomous exploration. A frontier as he defined it is ”regions on the boundary between open space and unexplored space." Autonomous exploration is used to build a map of a region that can be used for future navigation. Frontiers help address the issue of where to go in an unknown environment to gain as much information as possible. Directing robots to these frontiers pushes back the frontiers and adds information. When using frontiers as the way-points for directing exploration given enough time an entire region will be explored as there will be frontiers until the entire accessible area is explored.

To detect frontiers Yamauchi's proposed method is using "a process similar to edge detection and region extraction in computer vision" to find these boundaries. Once edge detection is run on the map to detect all frontier cells he used region extraction to find threshold-ed region sizes and then grouped cells into regions. An example of this is shown in Figure 2.7.

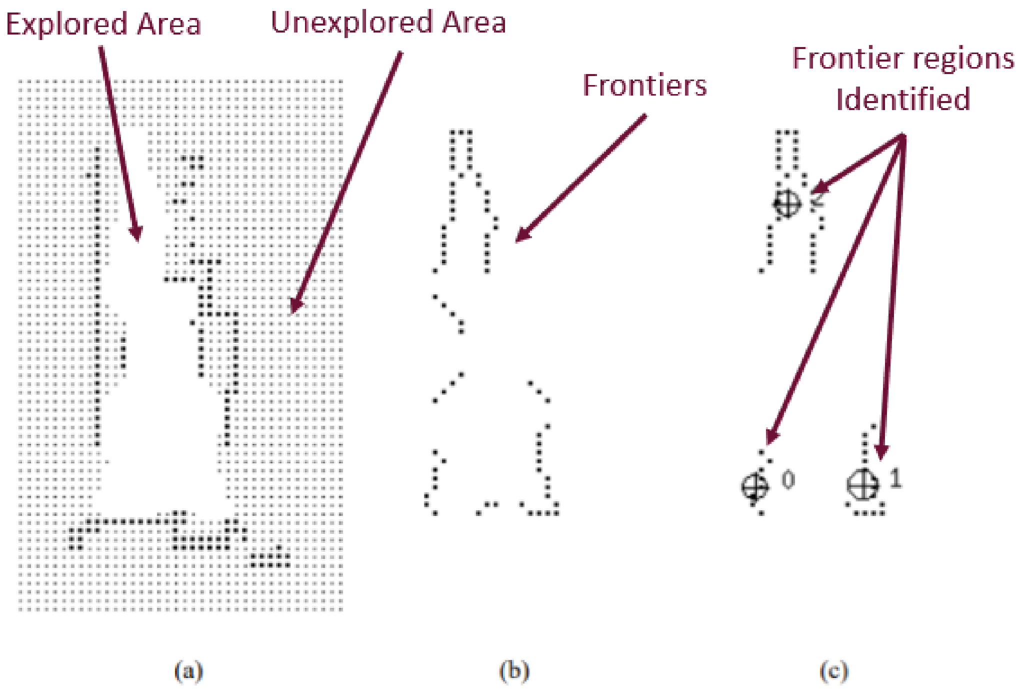


Figure 2.7: Frontier detection: (a) evidence grid, (b) frontier edge segments, (c) frontier regions [17].

As far as an algorithm to select the best frontier to go to Yamauchi proposed sending the robot to the nearest accessible and unvisited frontier. Yamauchi implemented this method on a mobile robot with a laser range finder, sonar sensors, and infrared sensors. This setup was tested in an office space and the results are shown in Figure 2.8.

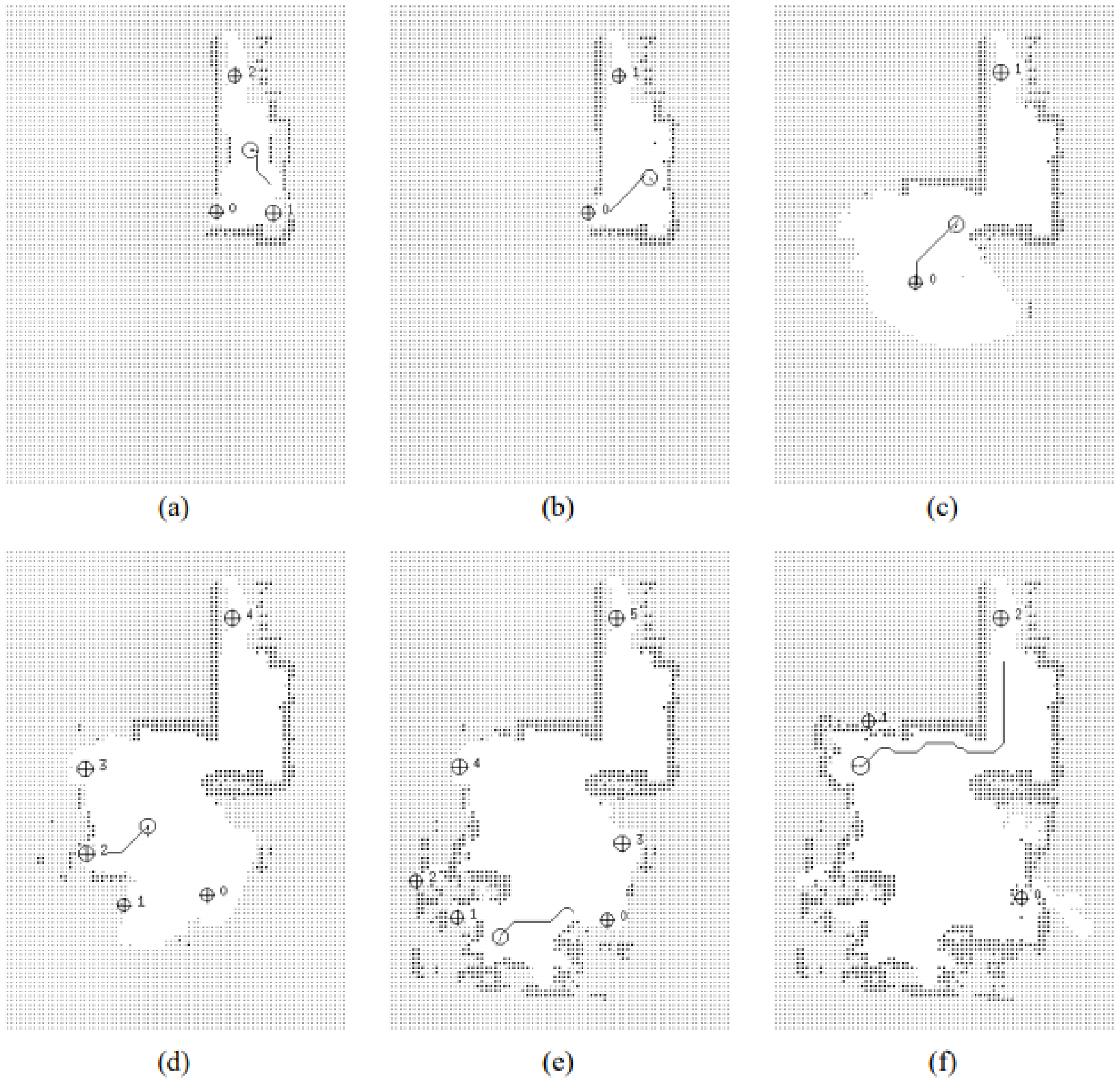


Figure 2.8: Maps Generated by Frontier-based exploration of an office from Yamauchi. a->f Represents time steps of the exploration [17].

Work has been done to expand this method. A method of weighted frontiers was used to help maximize the amount of space explored and help the UAV localize itself. This was important as when using SLAM. SLAM stands for Simultaneous localization and mapping and is a method of using Bayesian techniques to both build a map and localize on a robot. When trying to maximize the robots ability to localize with SLAM keeping a certain number of recognized landmarks is important and thus sending the robot too far into unexplored space would possibly result in loss of localization and mapping accuracy. Using a weighted method improves Yamauchi's method with a better selection of the identified frontiers [1]. Using differently weighted criteria has been used a lot in different frontier selection algorithms. Burgard et al used a weighted cost of reaching a frontier with its expected benefit derived from an expected increase in map entropy [2]. Gonzáles-Baños and Latombe used the same two criteria of distance and benefit but combined them using an exponential function [8].

2.2.2 Other Methods

One exploration method used in 2005 on a UAV was a method based on model predictive control (MPC). Their exploration was to find a path for a large UAV (Yamaha RMAX) through a dense unknown urban environment. An on-board laser scanner was used as the obstacles detection sensor. They used a cost function that minimized penalties for tracking and obstacle avoidance [14].

There has also been some work with collaborative UAV UGV exploration. A UAV would follow a UGV in an indoor environment and aid in situational awareness with its better vantage point. An augmented reality (AR) tag was used for the UAV tracking the UGV and SLAM was used to localize and map the environment for the exploration. An AR tag is a square black and white marker used for measuring the position and pose of the tag using a camera [10].

Chapter 3

System Design

3.1 Overall System Architecture

This collaborative UAV UGV system has been in development by the Unmanned Systems Lab for the past few years. Major focuses have been the custom stereo system, mapping algorithms, and the overall software design. The system has three major components. The UAV called Pinocchio in hardware and Iris in software, the UGV called Anubis, and the operator control station (OCS). The UAV is shown in Figure 3.1 and the UGV is shown in Figure 3.2. The framework used for this system is Robot Operating System (ROS). This framework and collection of tools allow for a system like this to be developed quickly. In addition to the hardware setup, there is also a simulation environment to test code before it is implemented on hardware. The simulation is run using Gazebo which integrates nicely with the ROS system.

All of the major functions run on-board the aircraft. This includes mapping and UGV path planning. The UGV only handles its localization and trajectory following. The OCS handles visualization and user interaction for sending goals to the system.

3.1.1 UAV

The UAV is a Tarot 960 frame capable of a max takeoff weight of 22 lbs. This gives us enough payload to carry all of the computer, networking, and sensors. With our configuration, it gets approximately 18 minutes of flying time running on two 10 Ah 6S lithium polymer batteries. The payload box on the aircraft contains the networking equipment, RTK GPS, and an NVIDIA Jetson TX2 as the main computer. The main sensor on-board used for mapping is a custom 600mm stereo camera. This uses two Point Grey GigE cameras with 16mm lenses. The TX2 allows a CUDA accelerated stereo-matching algorithm [9] allowing the stereo system to run at 3Hz. The exploration algorithm and the UGV planner run on the TX2 aboard the aircraft.

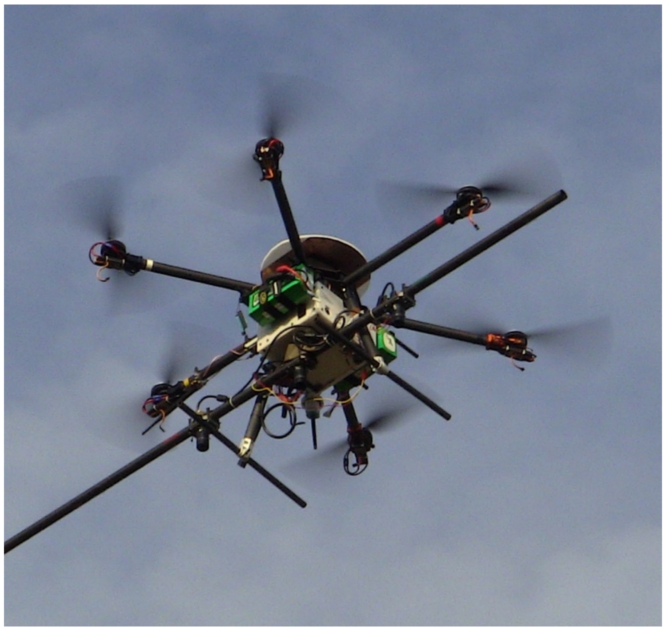


Figure 3.1: UAV used is a Tarot Hexacopter with custom stereo system and payload box

3.1.2 UGV

The UGV is a Clearpath Robotics Jackal. This is an off the shelf robot that comes with ROS integration. This is a simple skid steer robot that can handle fairly gentle off-road terrain. This fact makes it important that the aircraft can scout ahead and prevent the Jackal from going into terrain it cannot handle. The main things added to the Jackal are RTK GPS, communications equipment and a custom version of the ROS navigation package move_base that handles the execution of plans generated by the aircraft.

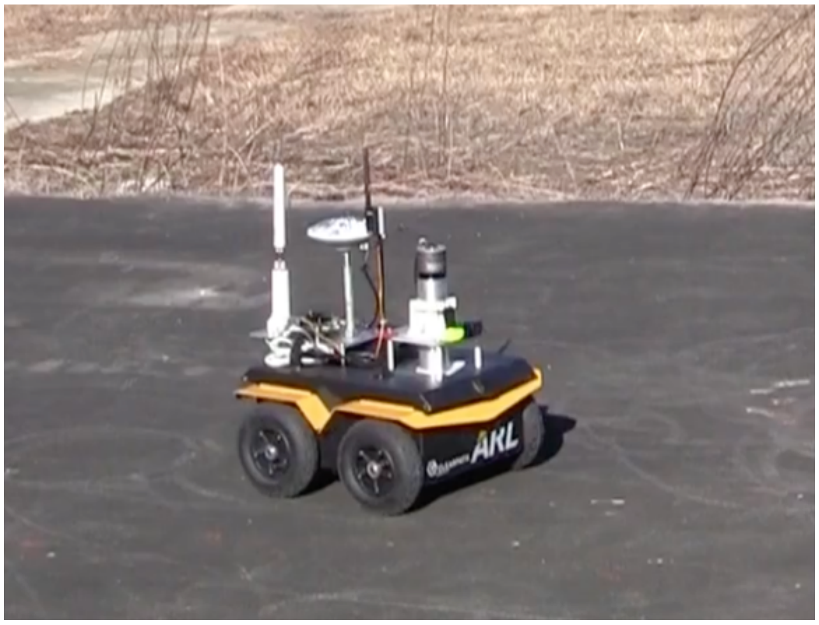


Figure 3.2: UGV is a Clearpath Jackal

3.1.3 OCS

The operator control station is the simplest part of the system. It is a desktop computer with two monitors to allow the user to see what is going on in the system and to send navigation goals. The main user interaction is through the program RVIZ, which is a 3D visualizer for the Robot Operating System (ROS) framework. This allows the user to see the maps being generated by the aircraft, the positions of the vehicles, and click on the map to specify the goal position of the UGV.

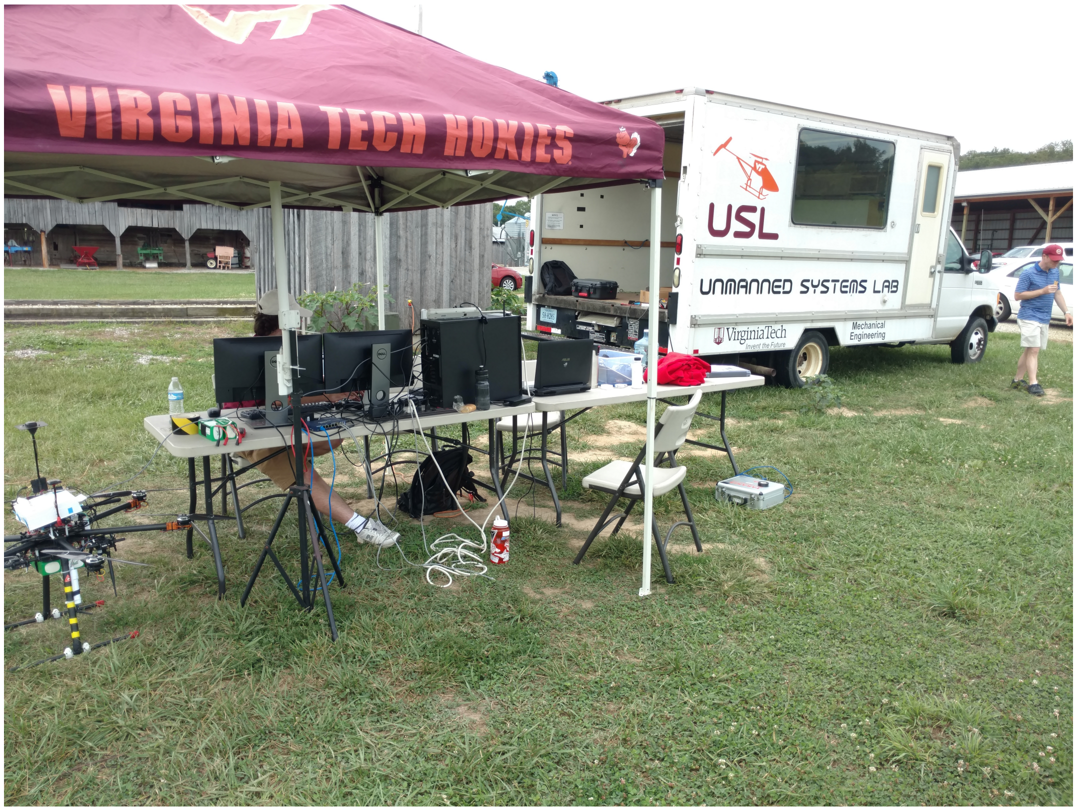
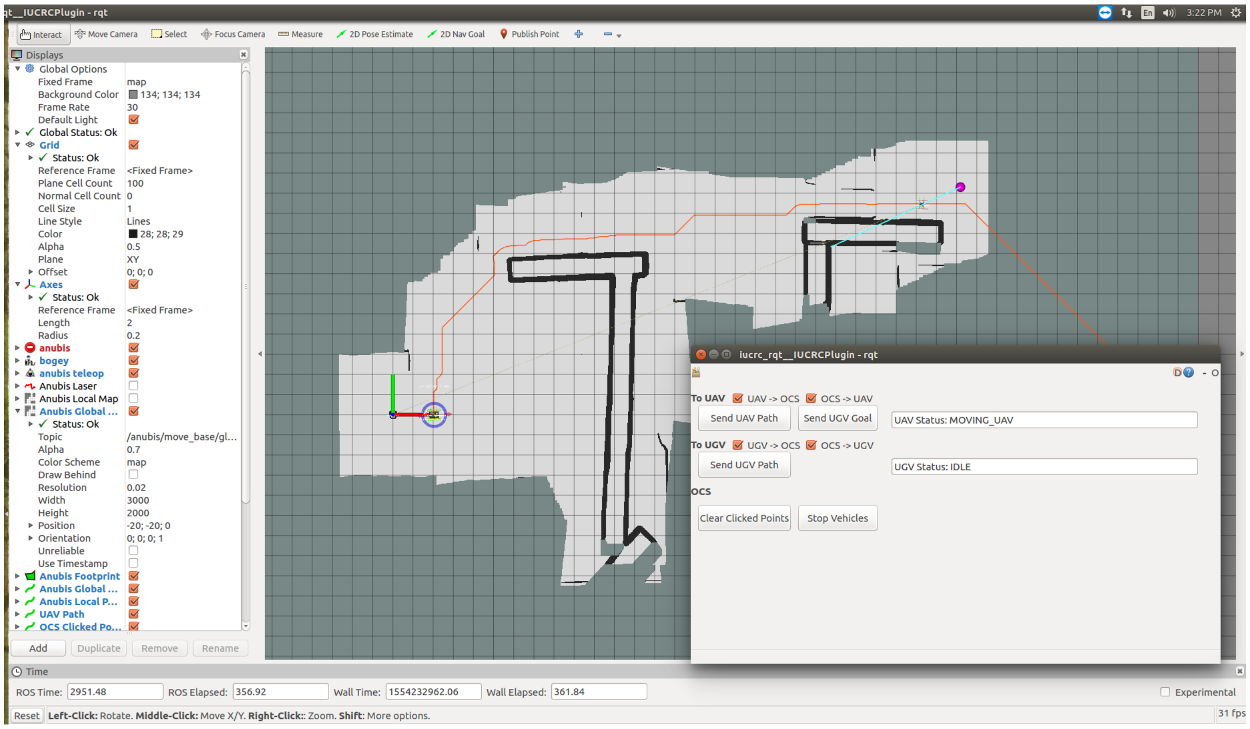


Figure 3.3: OCS setup in the field

3.1.4 Simulation Environment

Having a simulation environment for the system is incredibly important. This runs all of the software that runs on hardware with the computer simulating the vehicles and the environment. The backbone of the simulation is the program Gazebo. This handles the simulation of the environment, virtual cameras, and the vehicle physics. Clearpath provides the software to simulate the Jackal in Gazebo. The aircraft simulation is run using the Gazebo software in the loop (SITL) provided by the PX4 firmware developers. The only main difference between the hardware and simulation environment is that the simulation uses



Global Options	
Fixed Frame	map
Background Color	134; 134; 134
Frame Rate	30
Default Light	✓
Global Status: Ok	
Grid	✓
Status: Ok	
Reference Frame	<Fixed Frame>
Plane Cell Count	100
Normal Cell Count	0
Cell Size	1
Line Style	Lines
Color	28; 28; 29
Alpha	0.5
Plane	XY
Offset	0; 0; 0
Axes	✓
Status: Ok	
Reference Frame	<Fixed Frame>
Length	2
Radius	0.2
anubis	✓
bogey	✓
anubis teleop	✓
Anubis Laser	☐
Anubis Local Map	☐
Anubis Global ...	✓
Status: Ok	
Topic	/anubis/move_base/gl...
Alpha	0.7
Color Scheme	map
Draw Behind	☐
Resolution	0.02
Width	3000
Height	2000
Position	-20; -20; 0
Orientation	0; 0; 0; 1
Unreliable	☐
Use Timestamp	☐
Anubis Footprint	✓
Anubis Global ...	✓
Anubis Local P...	✓
UAV Path	✓
OCS Clicked Po...	✓

ROS Time: 2951.48 ROS Elapsed: 356.92 Wall Time: 1554232962.06 Wall Elapsed: 361.84

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click:**: Zoom. **Shift:** More options. 31 fps

☑ UAV -> OCS ☑ OCS -> UAV UAV Status: MOVING_UAV

☑ UGV -> OCS ☑ OCS -> UGV UGV Status: IDLE

Figure 3.4: User interface on the OCS

a depth camera to simulate the stereo system on the real aircraft. Overall the simulation environment is invaluable due to the fact the software can be validated before testing on hardware and that more tests can be done because there are none of the issues of testing in the field such as weather and field availability.

3.2 UAV Exploration Methods

In developing a method to properly explore for the UGV path generation two different methods were developed and tested. The first method developed was based on a simple analysis of what the algorithm needed to do. Fundamentally the problem is to determine which frontier is blocking a valid path. If we treat the unknown space in the environment as high cost but traversable, the path generated by the path planner will result in a frontier

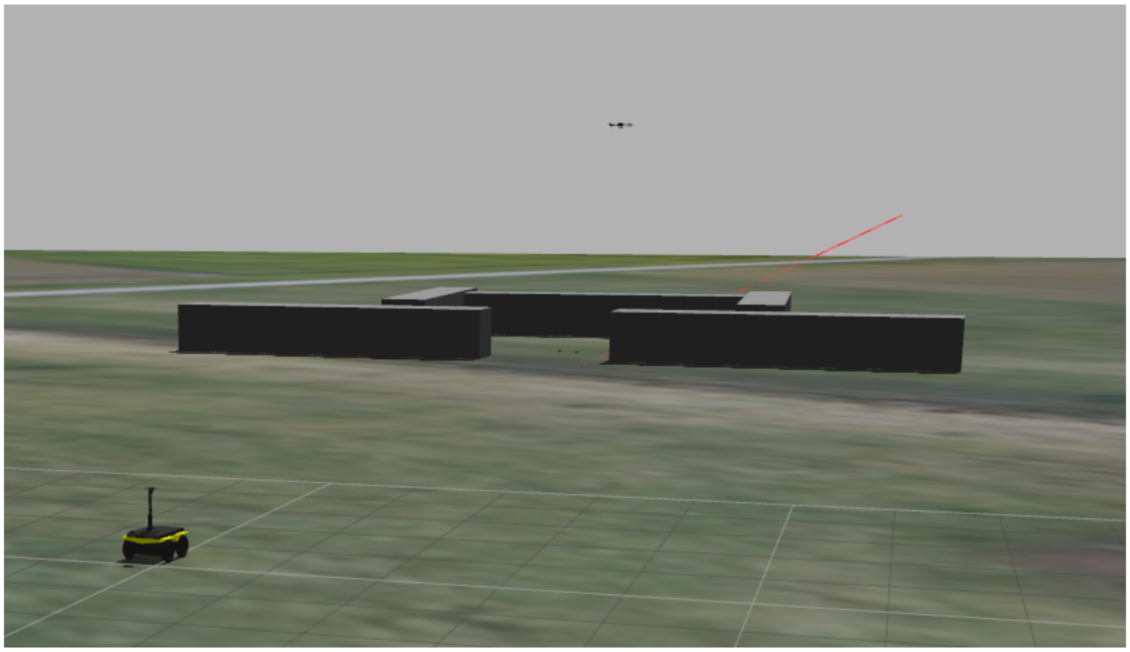


Figure 3.5: Screen-shot of the Simulation Environment in Gazebo

at the intersection of known and unknown space. This method will be called the Dijkstra Frontier method. The second method came from the observed shortcomings of the Dijkstra Frontier method. This method uses a bi-directional RRT to find multiple frontiers of interest impeding the path of the UGV and uses a scoring method to select the best frontier to explore.

3.2.1 Dijkstra Frontier Exploration

The first method explored was the Dijkstra Frontier method and is the much simpler of the two. This method is similar to using D* or re-planning with Dijkstra/A*. The main difference is how it handles where it is planning from and how it deals with the multiple agents. Traditional exploration with a UGV and D* or re-planning would involve updating or re-planning every-time the costs in the map update and would plan from the current location of the ground vehicle. The difference in this system is that the UGV is not doing the exploring but rather the UAV. If we were to just transfer the method of the ground vehicle to the air vehicle we would run into several issues. The first is that the UAV is not bound by the same constraints of obstacles. So if the UAV senses an obstacle in front of it on the ground it can go over it. This has several ramifications. The first is that the UAV could overshoot over an obstacle and continue exploring on the other side of an obstacle and thus would never find a valid path for the UGV. The second is that this means the exploration could benefit from going over obstacles and not trying to avoid them.

Algorithm 2 Dijkstra Frontier

```
1: $validUGVGoal \leftarrow$ generatePlan($obstacleMap$,$allowUnknown = false$)
2: while not $validUGVGoal$ do
3:     $validUGVGoal \leftarrow$ generatePlan($obstacleMap$,$allowUnknown = true$)
4:     for $waypoint$ in $path$ do
5:         $cost \leftarrow cellCostAtWaypoint$
6:         if $cost == unknownCost$ then
7:             $Frontier \leftarrow waypoint$
8:             break
9:     Send UAV to $Frontier$
10:    $validUGVGoal \leftarrow$ generatePlan($obstacleMap$,$allowUnknown = false$)
11: Send UGV to $ugvGoal$
```

The algorithm was designed with these issues taken into consideration. In order to ensure that with enough time and if a valid plan does exist that the UAV will find one, we generate plans from the current UGV location so that the frontiers found will always be impeding the valid path. To find the frontier for exploration, a plan from the current UGV location is made to the goal. The unknown space is treated as high cost but still traversable (Algorithm 2 Line 3). This results in a path going through known space and eventually going into unknown space if a valid plan does not exist yet. A valid plan in this context means that there is a path that goes to the goal without going through unknown space. This is specifically checked by running the planner without allowing unknown space to be traversed and checking if a valid path is returned. To find where the frontier is located, we iterate through the plan generated and extract the cost at each waypoint in the path (Line 5) and check whether the cost at that point is the value that represents known space (-1). Once a point is found in unknown space we extract that point as the UAV waypoint (Line 7). After the UAV reaches its waypoint the planner is run where unknown space is not allowed and if it returns a valid plan we know the exploration is done. If no valid path is returned the cycle is repeated. A visual of the algorithm in action is shown in Figure 3.6. A representation of the overall states of the algorithm is shown in Figure 3.7.

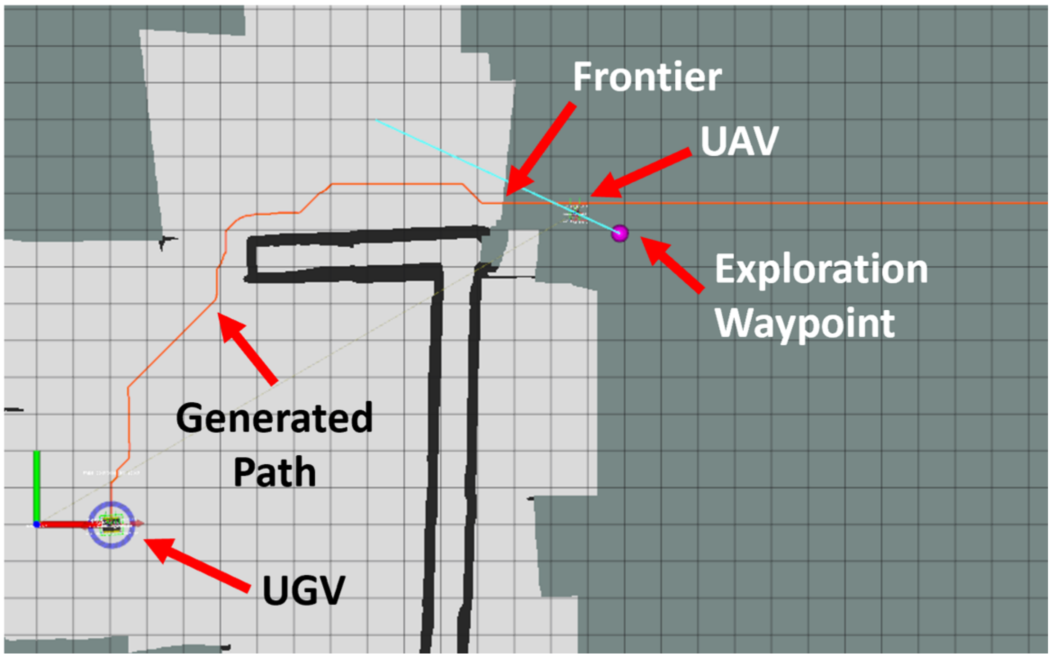


Figure 3.6: Diagram of the Dijkstra Frontier Method in Gazebo

To maximize the amount of information gained the actual point the UAV explores is offset by a fixed amount along the vector of the UAV to the frontier. If the UAV went exactly to the frontier point identified half of the field of view (FOV) of the stereo system would be in known space. This is inefficient in that less information is gathered but would also result in longer overall exploration times due to having to do approximately twice as many exploration cycles.

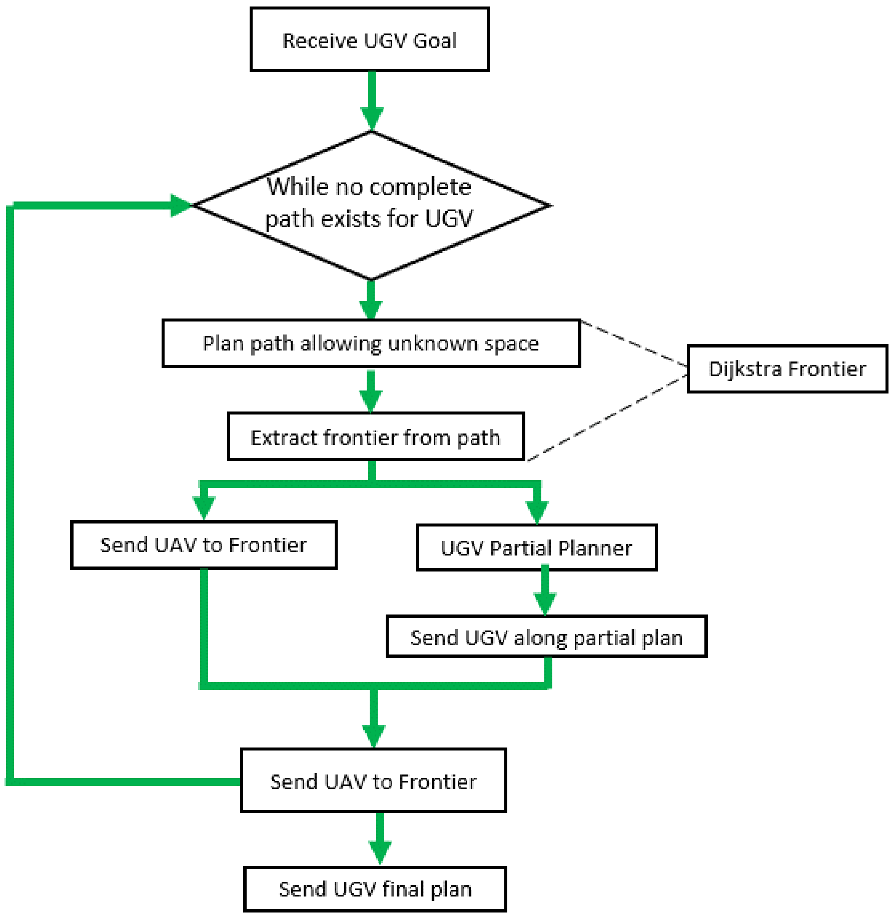

Receive UGV Goal
While no complete path exists for UGV
Plan path allowing unknown space
Extract frontier from path
Dijkstra Frontier
Send UAV to Frontier
UGV Partial Planner
Send UGV along partial plan
Send UAV to Frontier
Send UGV final plan

Figure 3.7: Flow chart of the Dijkstra frontier method and how it fits in the large scope of the system

3.2.2 RRT Exploration

The second method developed is based on some of the observed issues of the Dijkstra Frontier method. This was mainly that sometimes the exploration would oscillate between exploring two sides of an obstacle and make the UAV travel a longer distance. This method attempts to address that by using RRTs to generate multiple path options so that the UAV can select closer frontiers that may be along a sub-optimal path that is closer to the UAV.

RRTs were selected to handle this problem for several reasons. The first reason is that it is a sampling based path planning technique which means that sub-optimal paths can be generated. The problem with Dijkstra frontier method was that Dijkstra and similar method provide an optimal path and have no way of trying to get multiple sub-optimal paths in the environment. Weighted A* will produce sub-optimal paths but not necessarily the multiple options in the environment. The second reason for RRT is that RRT is single query and we are asking to plan once for each exploration cycle as compared to a probabilistic road-map which can be queried multiple times for the same static environment.

To properly find the frontiers, a bi-directional RRT is used to search from the goal and the start point where the RRTs are constrained to unknown and known space respectively. The RRT at the start point starts in known space and thus expands out in the explored space. The RRT at the goal explores outwards in the unknown space. Thus looking at the possible connections of the two RRTs will find the frontiers. Since the known space will only expand the RRT inside the known space is expanded on each iteration cycle and is not regenerated each cycle. The RRT in known space is regenerated each time for simplicity sake as the size environments that this was used for did not present speed issues. This could be improved by using a method to update the tree to remove branches that now intersect with found obstacles and known space.

The next section will detail the specifics of the RRT exploration method as it is implemented. The RRT exploration algorithm is implemented as its own class that is called inside the UAV manager node on the aircraft. It is inside similar logic to the Dijkstra Frontier method as it goes to frontiers until a valid path the UGV goal is found. The call to generate a plan is provided with the following information: ku the number of iterations for adding nodes in the unknown tree, ke the number of iterations for adding nodes to the known tree each cycle, the obstacle map, the UGV goal, and the current UAV position.

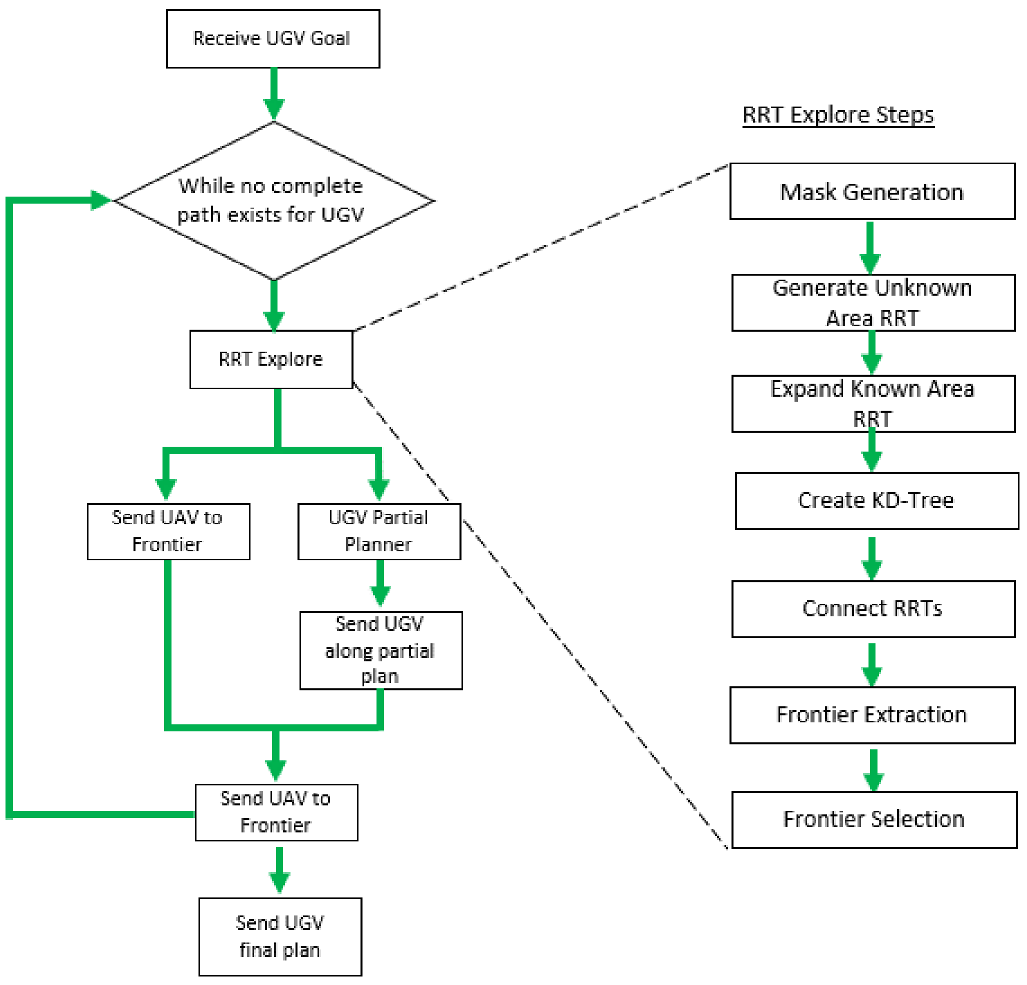


Figure 3.8: Flowchart for how the RRT Explore Algorithm fits in the system and the specific steps of the RRT Explore Algorithm

Mask Generation

The first step in the RRT explore algorithm is to generate some binary masks to help the algorithm later quickly identify certain regions. The three masks generated is known space, unknown space, and obstacles. These are implemented as OpenCV masks so that they can be used with other OpenCV functions. An example of the explored area mask is shown in

Figure 3.9.

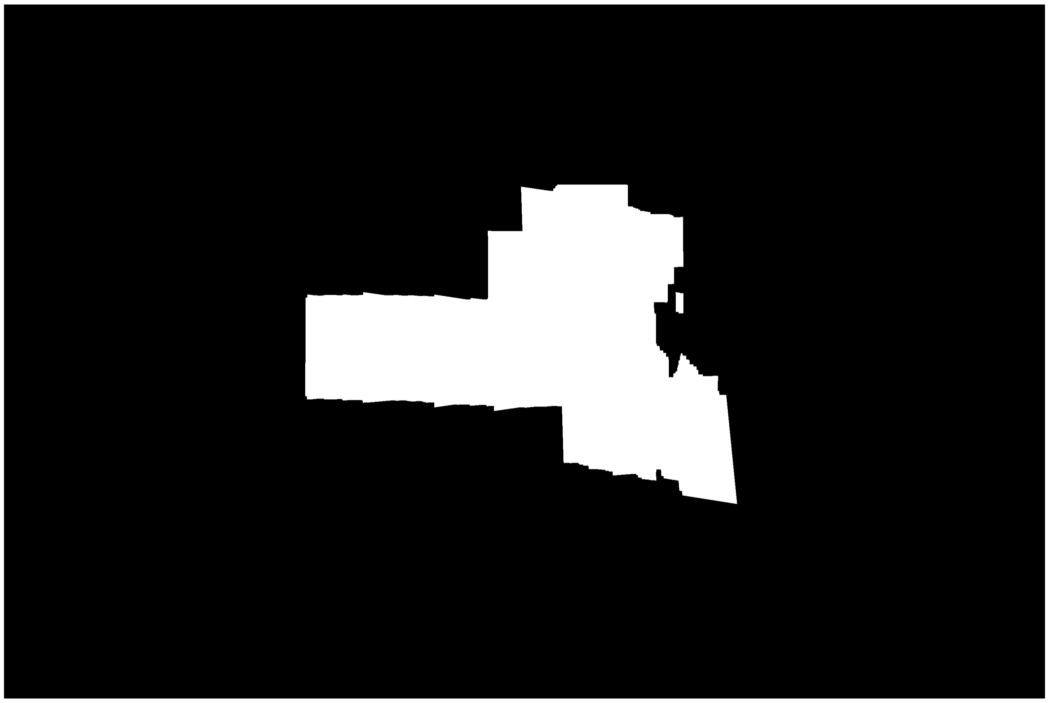


Figure 3.9: Mask of the explored area

RRT Generation

The basics of the RRT implemented are as shown in Algorithm 1. To start a point is selected as the root of the tree to which all branches will eventually lead back to. A random point is sampled inside the region of the obstacle map. The nearest neighbor to this point is found using a simple search since the number of points is relatively low. The RRT implemented uses the fixed length branch method. This means that each branch added is a specified length in the direction of the sampled point from the sampled point's nearest neighbor. This improves how well it can expand in tight spaces. Before the branch is added to the tree it is collision checked with obstacles. This is done using the masks previously generated. Using the OpenCV line iterator the pixels of the branch are iterated through and if any go outside the mask a collision is detected. If no collision is detected the branch is added to the tree. The RRT is implemented as an array of a custom data structure. It contains location information and points to its parent node (nearest neighbor to sampled point). The data structure for nodes in the tree also contains a distance to root of the tree element that is used in the RRT connection step. Each time a node is added its root distance element is created to be the sum of the distance to its parent node and the parent node's root distance.

Unknown Area RRT Generation

The second step after the mask generation is to generate the RRT in the unknown area. Since the unknown area is unknown and shrinking the RRT cannot be regenerated without an algorithm to collision check the existing tree with areas that were updated. Thus for simplicity, sake the RRT in unknown space is entirely regenerated. This has some impact on how fast the algorithm runs but the tree is able to expand quickly in the unknown space since there is more space for it to operate. The unknown tree is generated with the root of the tree at the goal defined for the UGV and expands outwards. The variable ku is used to define how many iterations are used to add to the unknown tree. In the trials done for this research values in the range of 500-2000 were used. This number can be selected based on the size of the environment and performance considerations.

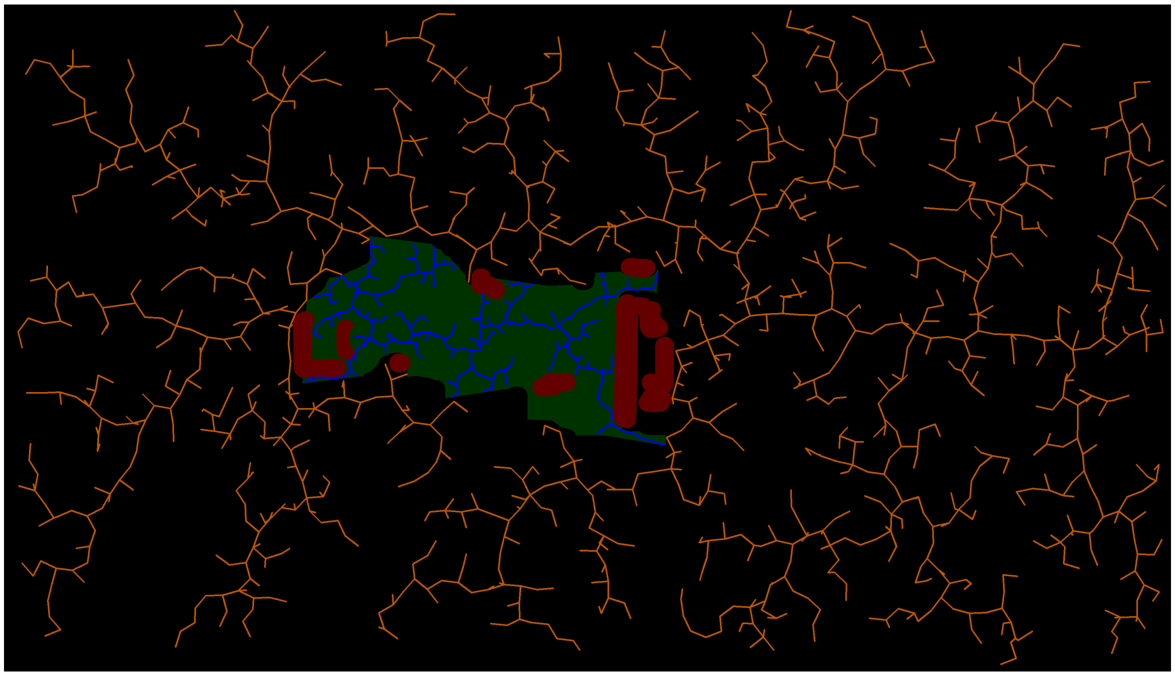


Figure 3.10: RRT (orange) in unknown space (marked black) expanding around the known space (green) and obstacles (red)

Known Area RRT Expansion

The third step is to expand the known tree. This follows the same basic RRT algorithm as the unknown tree but is inside the known space and is expanded instead of regenerated. The known tree is able to be expanded since the cleared space in the known area will not change since this is for a static environment. Thus all existing branches in the tree will remain valid for the lifetime of the exploration. Expanding is also necessary for the RRT to expand properly. Since the known space is smaller and will have obstacles in the way a lot more iterations is required to find a valid path between two points. Thus expanding with each cycle gives a lot of iterations and spreads the computation over the life-cycle of the exploration.

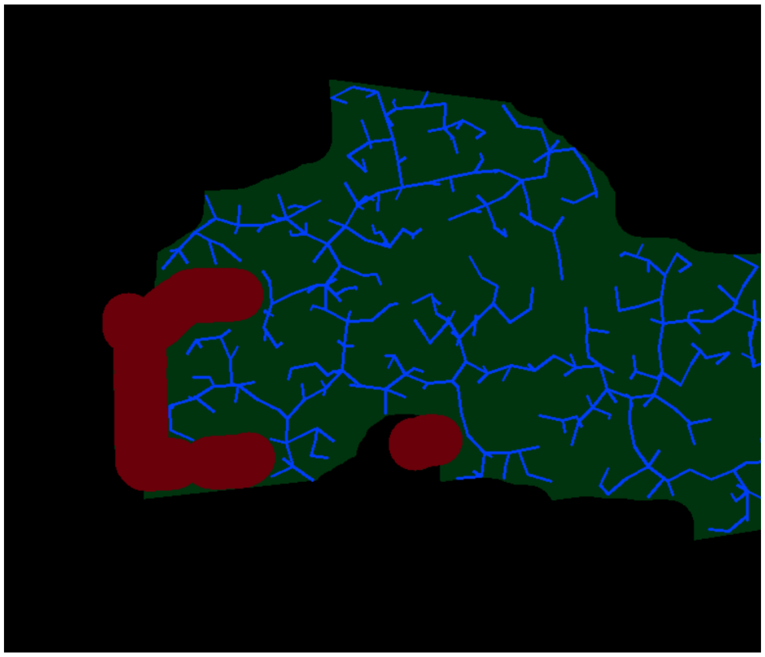


Figure 3.11: RRT in known (marked green) space expanding; obstacles (red)

KD-Tree Generation

The fourth step in the process is to generate a KD-tree for the points in the unknown RRT. This will be used in the connection phase of the algorithm as a large number of nearest neighbor queries are going to be made. The nodes in the KD-tree are the x and y coordinates of the nodes in the unknown tree. An existing implementation of KD-tree was used.

RRT Connection

The fifth step is connecting the two RRTs. Connecting the two trees will find the possible paths in the environment. Due to the large number of nodes in both trees, the number of combinations of nodes produces a large number of possible connections. To reduce this to a more reasonable amount a ”best” connection is generated for each node in the known tree. The first step in this is to identify its nearest neighbors for a node in the unknown tree. Its nearest neighbors are determined on a fixed neighborhood size. Once its nearest neighbors are determined each connection to the neighbor is collision checked. For all the valid connections, a cost function related to these connections is used to determine the best connection. This function is a weighted sum of the distance through the tree from start to goal and the distance between the two neighbors. The distance between the start and goal is calculated using the root distance element in the two nodes. The weighting used in this implementation was a weight of 0.1 for the total distance and 1.0 for the distance between the two nodes. This weighting scheme was determined to find the balance from finding the shortest overall path but also making sure it is a reasonably close connection. This step of connecting is repeated for each node in the known tree to find multiple paths. An example of the connections generated can be seen in Figure 3.13. The connections are represented as yellow lines.

Frontier Extraction

The sixth step is to extract the frontiers from the connections. To get the navigation point for the UAV the point in which the path crosses from known to unknown space needs to be identified. Each connection is stored with both the start and end points. To extract the frontier each point is converted to an OpenCV point and the line iterator is used to iterate through the pixels along the path until unknown space is reached. This point is identified as that connection's frontier. All of the frontiers are stored for the next step. The extracted frontier can be seen in Figure 3.13 where the yellow lines are connections and the pink dot is the extracted frontier.

Frontier Selection

The seventh step is to select the best frontier for the UAV to visit. To select the best frontier to visit a cost function is used. The cost function is the weighted sum of the straight line distance to the goal and the straight line distance to the UAV. These two parameters are used to balance between finding the shortest path for the UGV and minimizing how much the UAV has to travel. Using static weights would work but would not account for how the importance of the two variables change as the exploration frontier gets closer to the goal. When closer to the goal distance to the goal is more important than distance to UAV as going to a point closer for the UAV but that is further from the goal would be inefficient. When the exploration is near the beginning reducing UAV travel distance is more important. To handle this the costs are determined based on how close the frontier is to the goal. The selected weights as a function of frontier distance to goal are shown in Figure 3.12. For the weight functions, a logarithm was chosen for the goal distance weight as it has an increased weight near the goal and fades out as we get away from the goal but not linearly. A linear ramp that saturates was chosen for the UAV distance as we want it to increase as we get away from the goal but should eventually level out. The exact weights were selected based on empirical observations and tuned manually based on watching the performance of the exploration. While the exact values could be tuned more exactly by running a lot of simulations the optimal values would change from environment to environment. Thus these values that worked generally well across the different environments tested were selected. The formulas for the weights are below with the selected values used.

$$weight_{GOALdistance} = -0.1 * log(d_{goal}) + 1$$

$$weight_{UAVdistance} = min((0.027) * d_{goal} + 0.2, 1.0)$$

Once the cost for all of the frontiers is calculated the one with the minimum cost is selected and the waypoint is sent to the navigation handler. This then directs the UAV to the selected frontier via a straight line path.

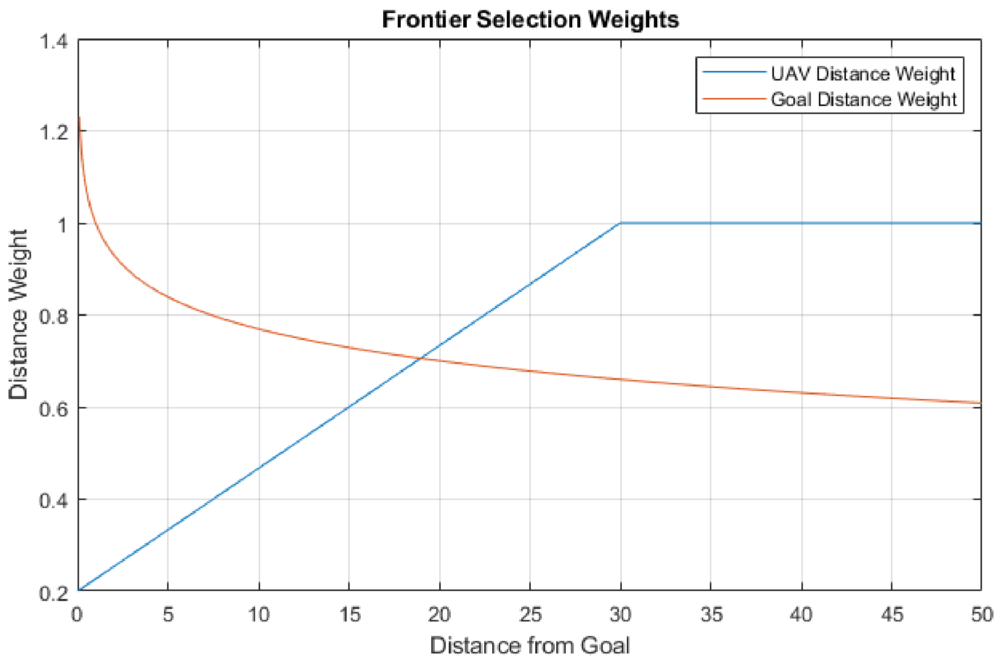


Figure 3.12: Weights for the Frontier Selection Cost Function

Visualization

In order to see what the RRT Explore method is doing a visualization step is done. A map of the environment is made and the different regions (unknown, known, and obstacle) are marked in an image with different colors. In this case unknown space as black, known space as green, and obstacles as red. The two RRTs are drawn as well with blue representing the known RRT and orange representing the unknown RRT. The best connections are also drawn as yellow lines and the frontiers identified are drawn as pink dots. A close up and labeled view of this is shown in Figure 3.13. An example of the visualization with the whole map is shown in Figure 3.14. This visualization is good for seeing how the algorithm is performing in the environment and if the parameters are working well for the environment.

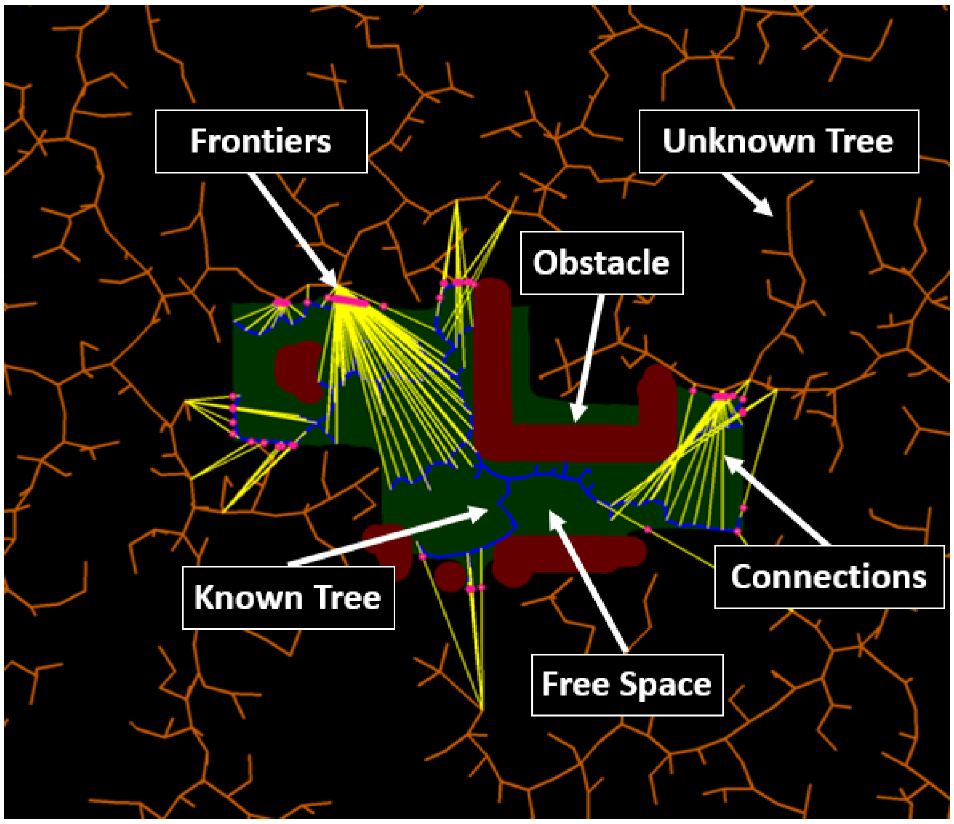


Figure 3.13: RRT Explore Diagram with labels; zoomed in on smaller area in environment

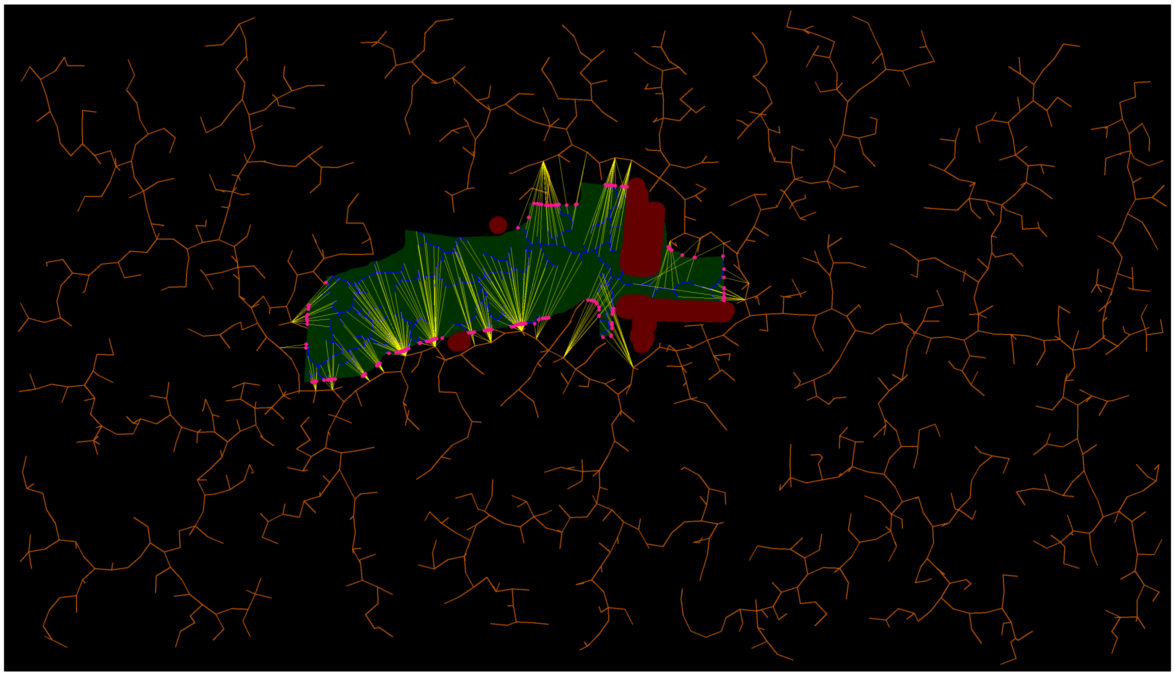


Figure 3.14: Visualization of the RRT Explore Method run on a sample environment, showing the entire map region

3.3 UGV Intermediate Plan Method

Allowing the UGV to move while a complete plan is not found yet allows the overall mission time to be reduced. If the UGV waited at the start until the UAV was done exploring the UGV would follow a valid plan but wastes time that it could have been moving. intermediate plans can be given to the UGV but they risk leading the UGV down a path that is not valid and it will have to backtrack. Thus a balance between directly following the UAV and waiting at the start must be found. Figure 3.15 shows how the UGV can be sent partway to the frontier. The method proposed here uses a safe follow back distance along with looking for "danger areas" along the current path to the frontier.

The first part of the intermediate plans for the UGV is to generate a path to the frontier.

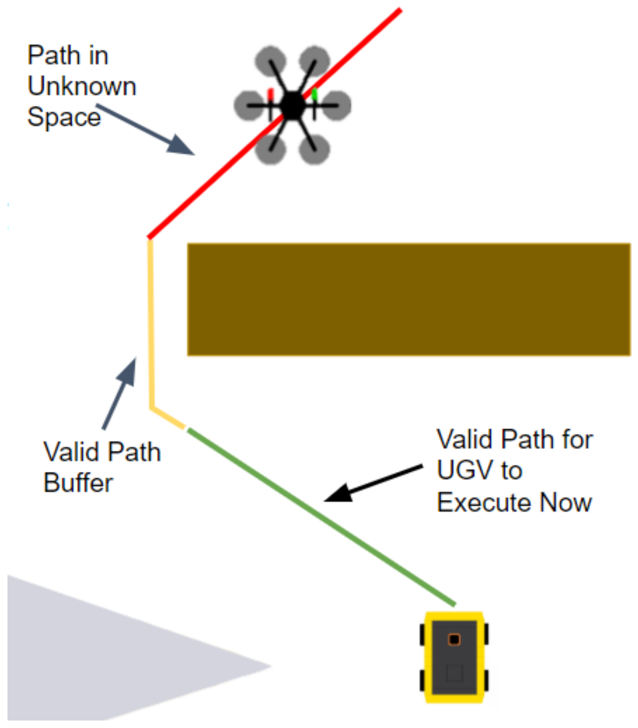


Figure 3.15: Diagram showing how the UGV is held back when an incomplete plan to the goal exists

Since the frontier is right on the edge and sending the UGV there would put it too close to the frontier, a point in towards the known area is used. In the case of the RRT exploration method the node in the known area that makes up the connection for the frontier is used. With a path generated the area near the frontier dangerous areas that could lead to backtracking need to be identified. Danger areas for the UGV would be near where obstacles abut with unknown space. To quickly identify these regions the existing unknown and known area masks are used. The obstacle mask is inflated so it will overlap with areas in the unknown mask and a bit-wise and operation is done to quickly identify the danger regions.

Once the danger areas are identified and stored as a mask the danger along the path needs to be quantified. To do this an area on either side of the path is examined for grid cells marked as dangerous. To measure on either side of the path two points on either side of the path are generated that form a line perpendicular to the path at that point. Using these two points the line iterator from OpenCV is used to iterate through the grid cells perpendicular to the path and accumulate the total danger at that point in the path. This is repeated for all points in the path to generate a danger score along the path. Figure 3.17 shows the measured region on either side of the path.

To prevent the UGV from backtracking it should be held back a certain distance from danger areas. A fixed distance from danger is defined as the follow back distance. Different follow back distances were tested in the results section (4). A sliding window with equivalent length of the follow back distance is applied to the path danger vector starting at the end nearest the frontier. This window is slid back towards the start until the sum of the path danger in the window is less than a threshold. The starting point of this window is then selected as the end waypoint for the UGV. The generated path up to this waypoint is sent for execution by the UGV. A sample path generated is shown in Figure 3.16 without the measured region and with in Figure 3.17.

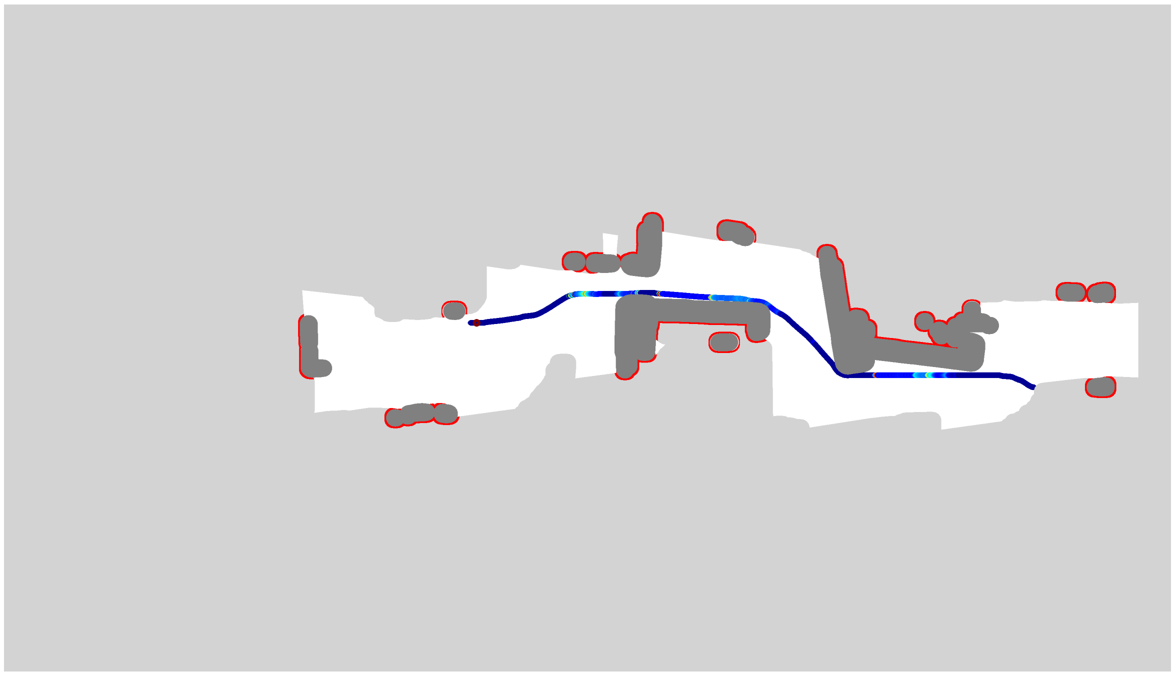


Figure 3.16: UGV Path with Danger Encoded as Color, Danger Areas Marked as Red, Clear explored space marked white, Unexplored area marked light grey, and Obstacles marked as dark grey.

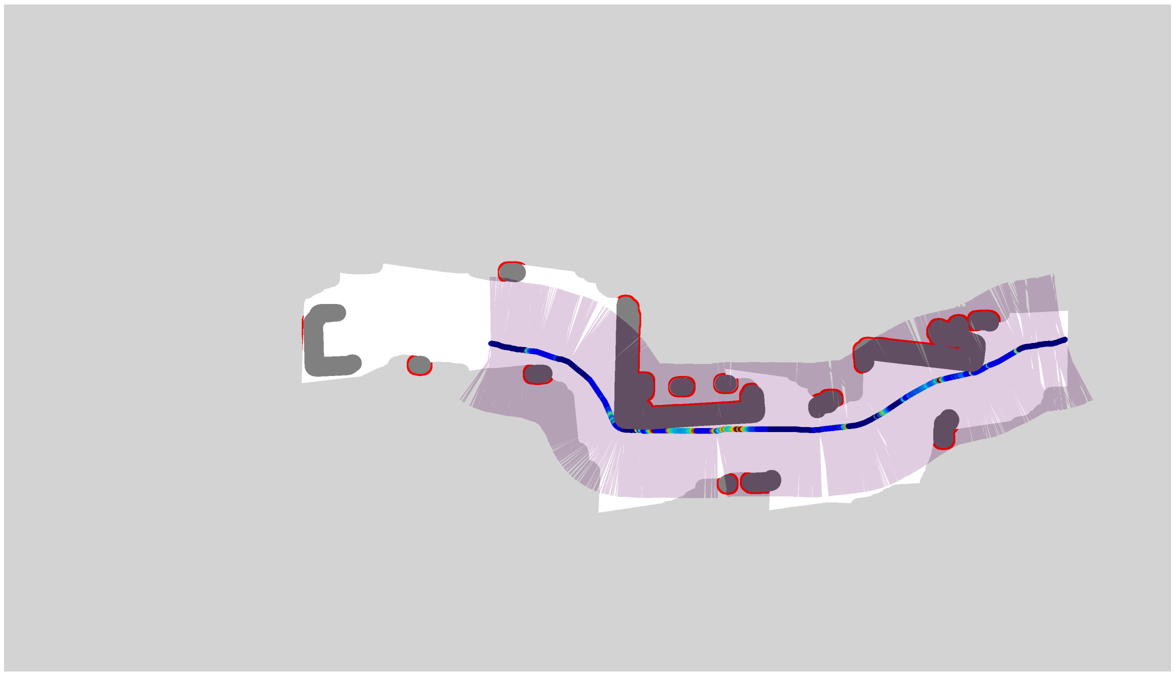


Figure 3.17: UGV Path (blue) with the region around the path measured for danger areas (pink). Increased danger along the path shows up on a colorscale with the maximum being encoded as red.

Chapter 4

Results

4.1 Dijkstra Frontier vs RRT Explore Exploration Distance Comparisons

Both algorithms were tested on different environments setup in Gazebo. Each environment consists of a different configuration of obstacles designed to challenge the algorithms. Diagrams of each environment are shown in Appendix A. The total distance traveled by the UAV finding a path for the UGV was recorded for 50 simulation runs for both the Dijkstra Frontier method and the RRT Explore method. Box and whisker plots are presented for each method to show the difference in the traveled distance. Additionally heatmaps were generated to show where the UAV was over the course of the 50 simulations.

4.1.1 Testing on Demo 1 Environment

The demo 1 environment is designed to simulated several small buildings in the environment. The distance results for demo 1, shown in Figure 4.1, show on average the RRT Explore method had shorter exploration distances than the Dijkstra Frontier method. Both method have shown the best performance to be similar, around 70m, but the upper end of the RRT Explore method is much lower. The heatmaps (Figure 4.2 & 4.3) show the Dijkstra Frontier heads straight for the goal but when an obstacle is encountered searches all around the obstacles. The RRT method does not make a straight line towards the goal since it is a sampling based method. The RRT method does show that it commits to exploring one side without oscillating back and forth between sides of an obstacle. This is what results in shorter exploration distances for the RRT method.

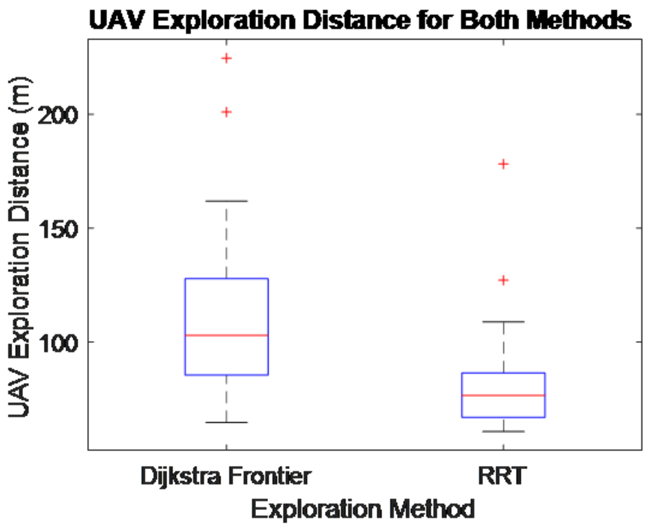


Figure 4.1: RRT exploration method results in less distance traveled by the UAV in the Demo 1 Environment.

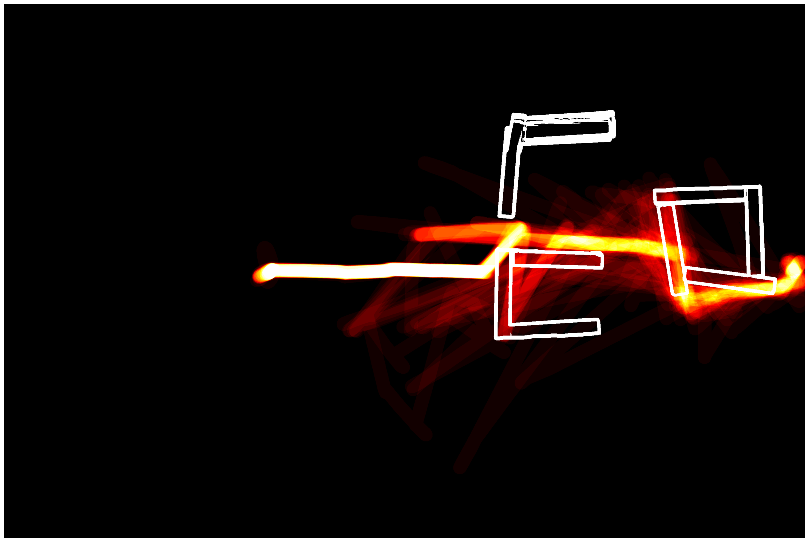


Figure 4.2: Heatmap for Dijkstra Frontier in Demo 1 Environment

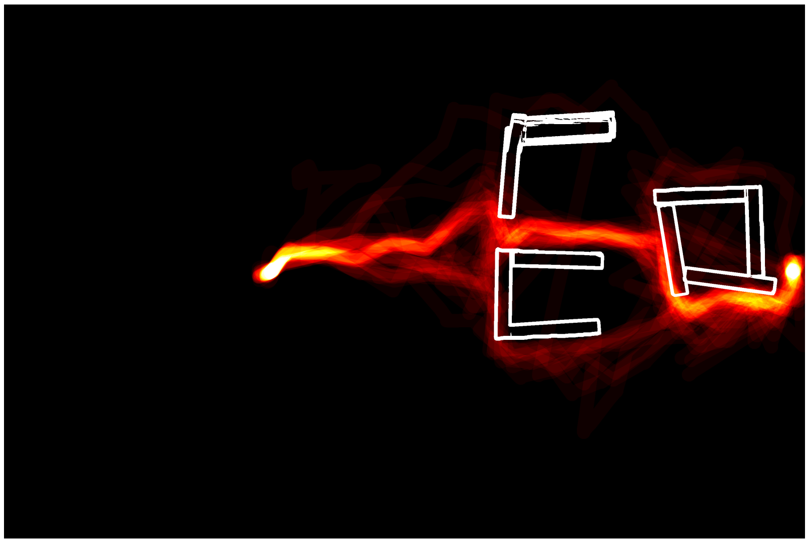


Figure 4.3: Heatmap for RRT Explore in Demo 1 Environment

4.1.2 Testing on Demo 2 Environment

The demo 2 environment is designed to simulate an environment where an obstacle has one side that is much better to explore than the other. This is done by offset rows that make the shortest path right through the center but if the other sides are explored result in longer exploration distances. The distances for this environment show that the Dijkstra Frontier method out performs the RRT Explore method. This is expected as this is one of the worst case scenarios for the RRT Explore method. Since the RRT Explore method is designed to reduce oscillations while exploring it commits to one side more than the Dijkstra frontier method which is always looking for the optimal path. This can be seen in the heatmaps as well (Figure 4.5 & 4.6). The Dijkstra Frontier method shows a pretty direct exploration through the center of the environment and never goes to the long sides of the obstacles. This is in stark contrast to the RRT Explore method which over the course of the 50 simulations explored pretty much every path possible resulting in longer exploration times.

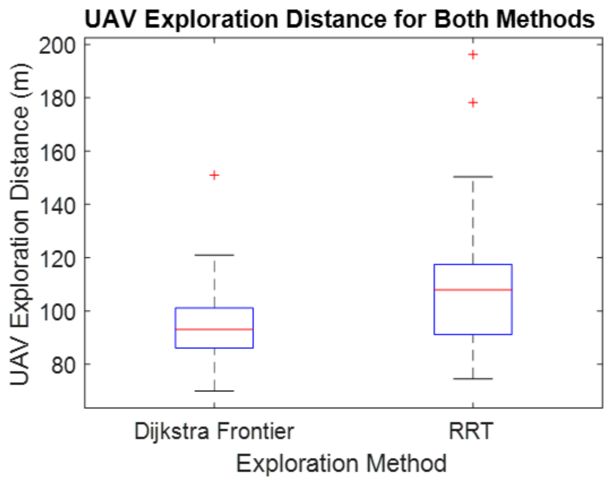


Figure 4.4: RRT exploration method results in a longer distance traveled by the UAV in the Demo 2 Environment.

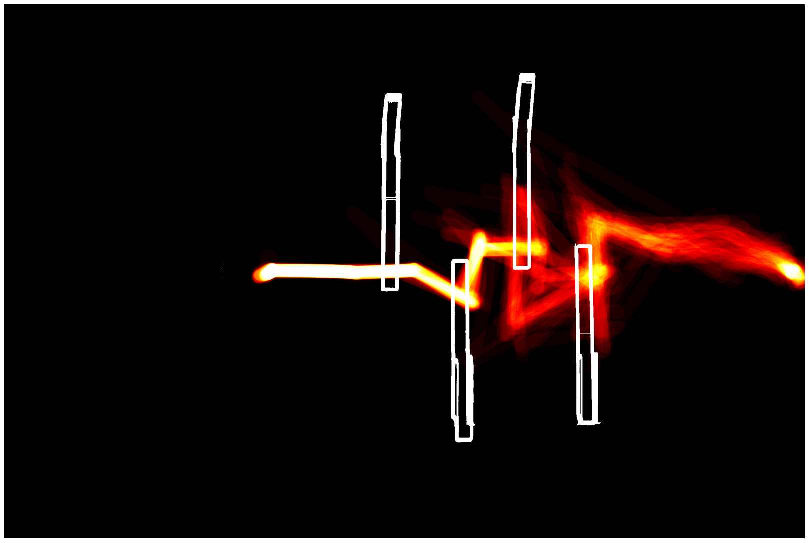


Figure 4.5: Heatmap for Dijkstra Frontier in Demo 2 Environment

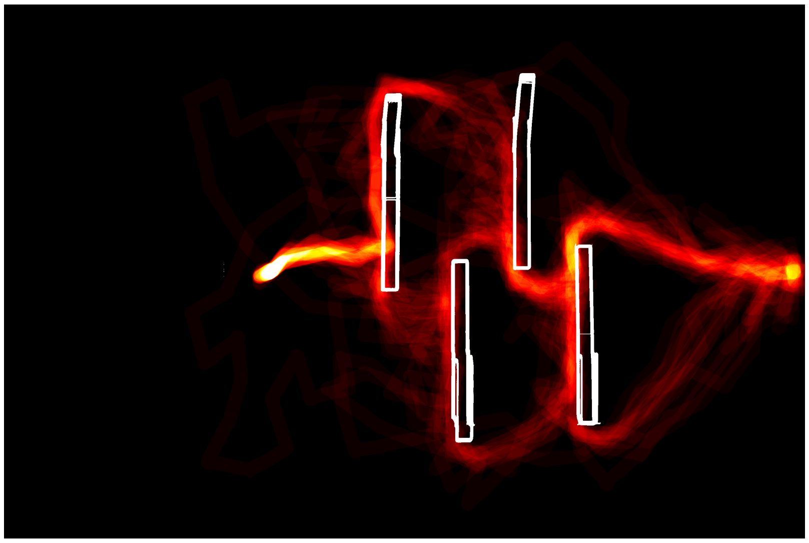


Figure 4.6: Heatmap for RRT Explore in Demo 2 Environment

4.1.3 Testing on Demo 3 Environment

The demo 3 environment is an absolute worst case environment with an extremely long dead end. The result of this test show the largest advantage of all the environments for the RRT Explore method. It averaged over 100m less of exploration distance than the Dijkstra Frontier method. The heatmaps also show this very well (Figure 4.8 & 4.9). The Dijkstra Frontier method shows exploration all over the environment while the RRT Explore method shows a very direct into the dead end and then backtracks out before heading to the goal.

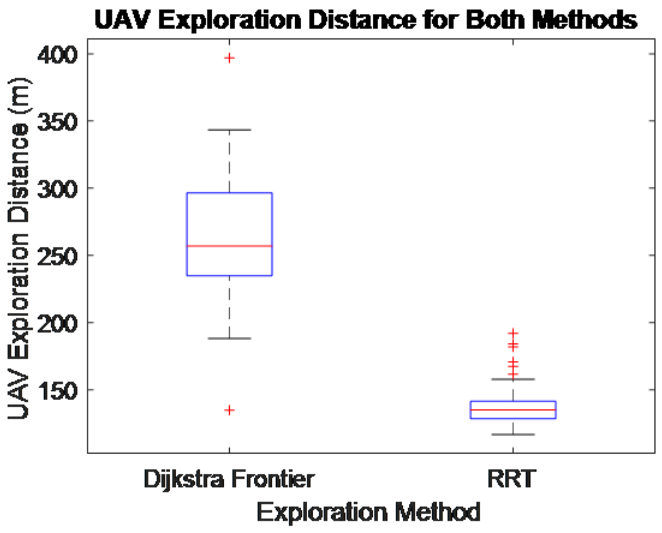


Figure 4.7: RRT exploration method results in less distance traveled by the UAV in the Demo 3 Environment.

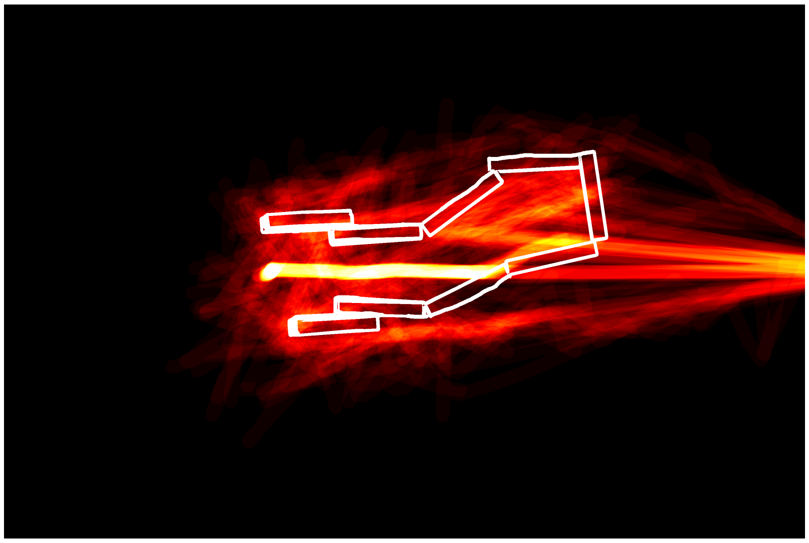


Figure 4.8: Heatmap shows the erratic exploration behaviour of Dijkstra Frontier in this Environment

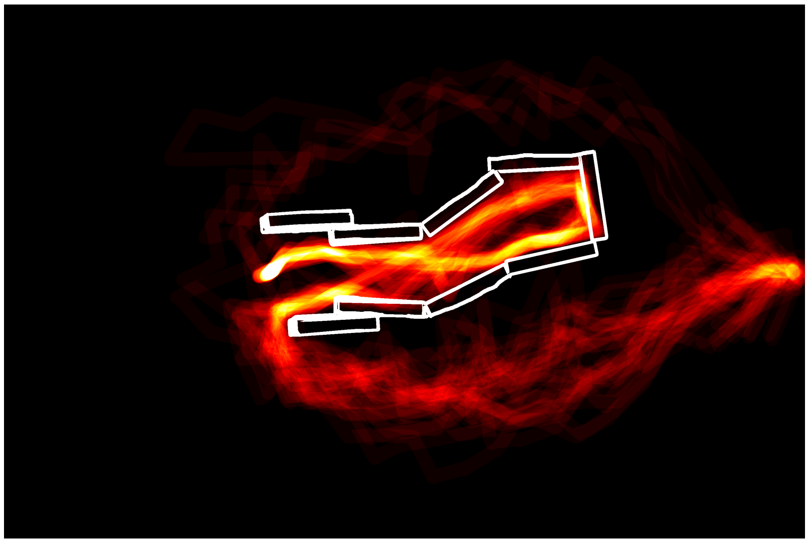


Figure 4.9: RRT exploration method shows better exploration than Dijkstra Frontier

4.1.4 Testing on Demo 4 Environment

The demo 4 environment is fairly simple with a few obstacles right before the goal. The results (Figure 4.10) show that both algorithms perform similarly with the means very close, and the minimum and maximum values similar as well. The interquartile range for the RRT method is slightly larger and higher as the RRT method does search directly ahead as the Dijkstra Frontier method does. The straight line searching is well seen in Figure 4.11. The more erratic searching in free space for the RRT method can be seen in Figure 4.12. This is a result of the random nature of the sampling based path planning.

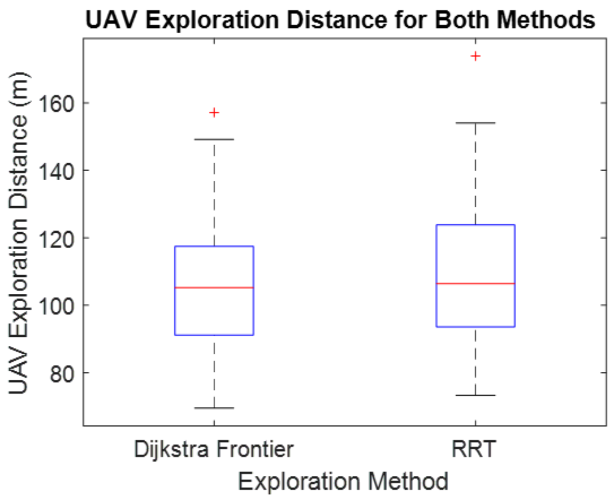


Figure 4.10: RRT exploration method results in similar distances traveled by the UAV in the Demo 4 Environment.

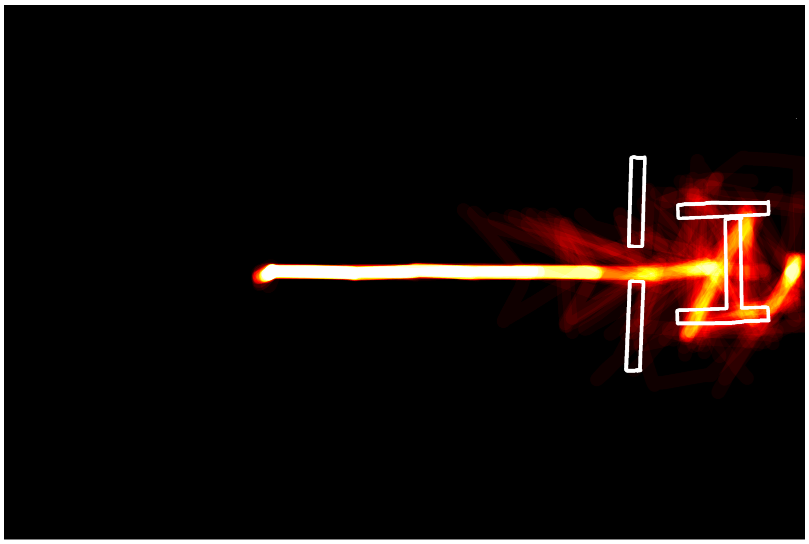


Figure 4.11: Heatmap for Dijkstra Frontier in Demo 4 Environment

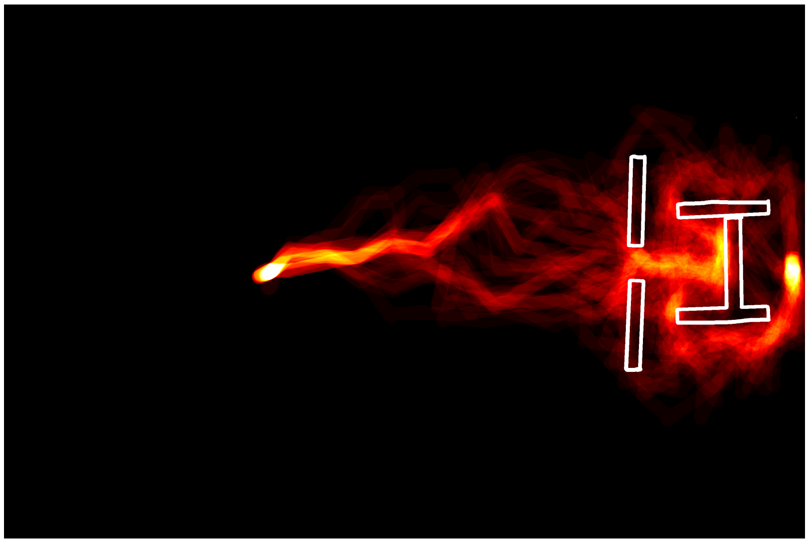


Figure 4.12: Heatmap for RRT Explore in Demo 4 Environment

4.1.5 Testing on Demo 7 Environment

The demo 7 environment was setup to really test the advantage of the RRT Explore method. The Dijkstra Frontier method tends to oscillate between the sides of a large obstacle, so two large obstacles were placed in between the start and the goal. The results (Figure 4.13) show very clearly that the RRT explore method does achieve better exploration with large obstacles in its way. The mean exploration distance was approximately 90 m longer for the Dijkstra Frontier method. This behaviour can also be seen very clearly in the heatmaps (Figure 4.14 & Figure 4.15). The Dijkstra Frontier method shows exploration all over the environment while the RRT method shows fairly direct exploration around the perimeter of the obstacles.

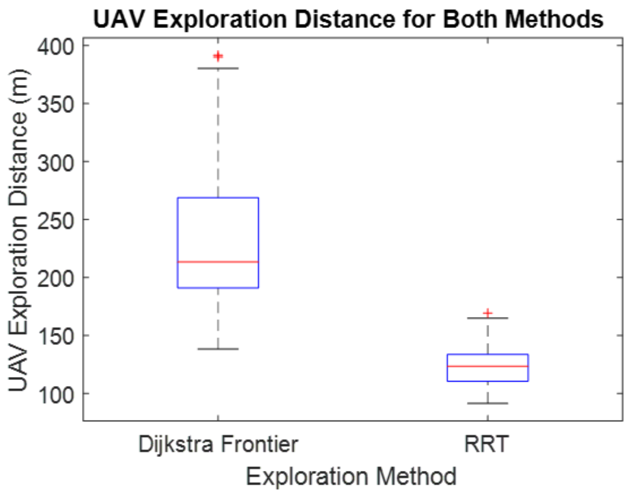


Figure 4.13: RRT exploration method results in less distance traveled by the UAV in the Demo 7 Environment.

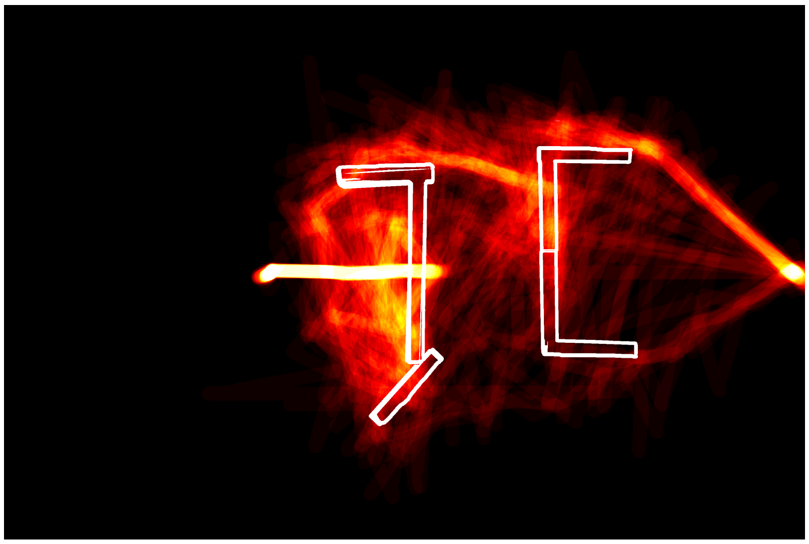


Figure 4.14: Heatmap for Dijkstra Frontier in Demo 7 Environment

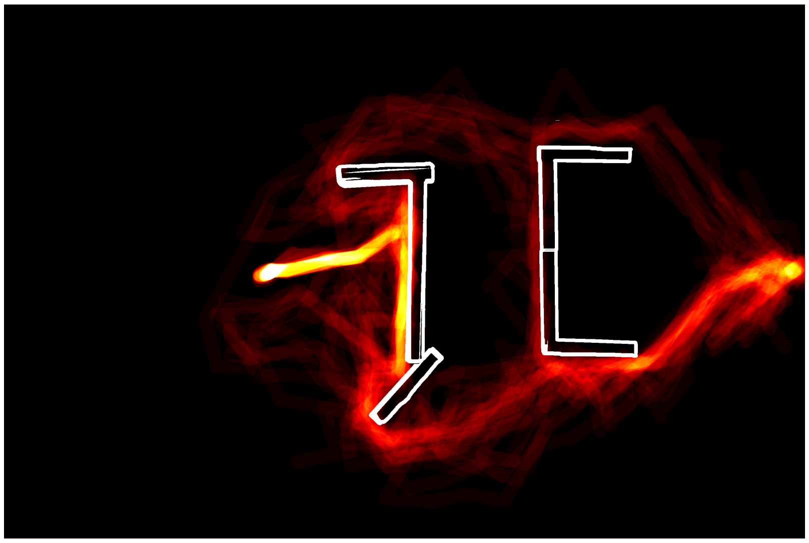


Figure 4.15: Heatmap for RRT Explore in Demo 7 Environment

4.1.6 Testing on Demo 8 Environment

The demo 8 environment is designed to emulate a dense environment with a lot of small buildings in between the start and the goal. The results (Figure 4.16) show that the RRT out preformed the Dijkstra Frontier method by around 40m on average. Upon inspection of the heatmaps (Figure 4.17 and Figure 4.18) the main difference between the two is a tendency for the Dijkstra Frontier method to do more exploration around the obstacles similar to demo 7.

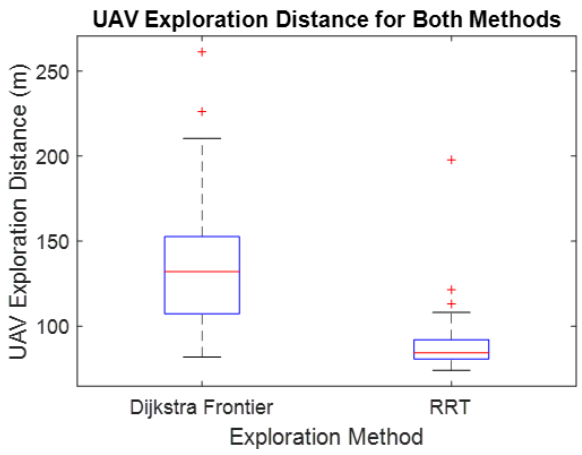


Figure 4.16: RRT exploration method results in less distance traveled by the UAV in the Demo 8 Environment.

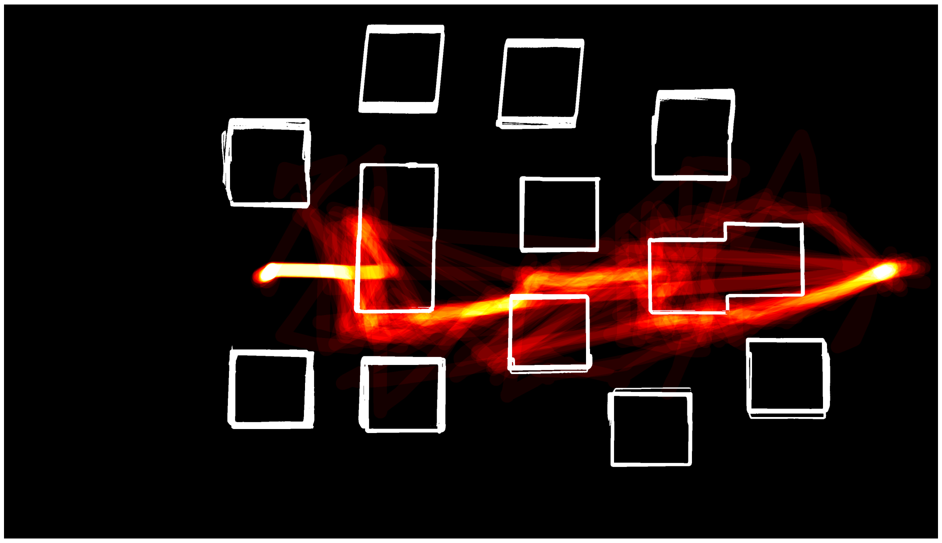


Figure 4.17: Heatmap for Dijkstra Frontier in Demo 8 Environment

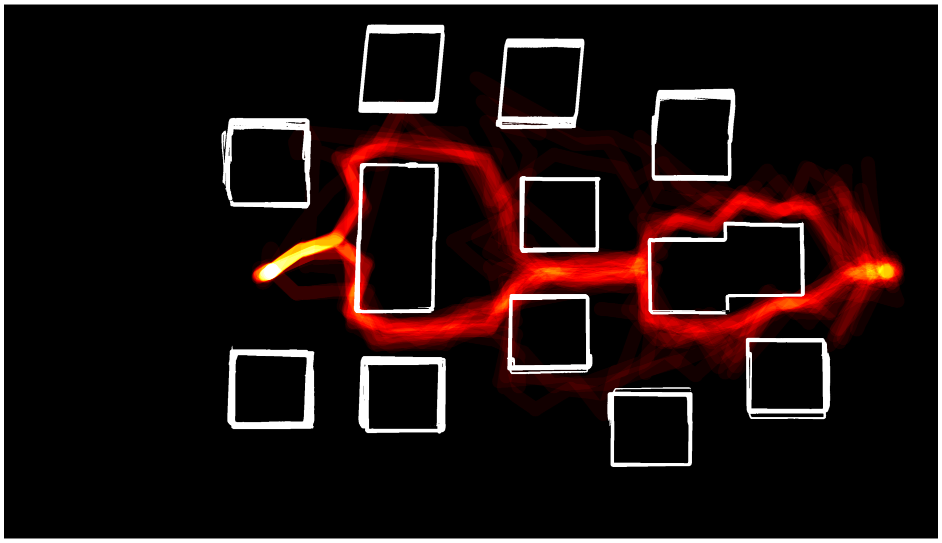


Figure 4.18: Heatmap for RRT Explore in Demo 8 Environment

4.1.7 Computation Requirements

The computational speed performance was measured on two computers for each step in the RRT Explore algorithm. The desktop tested has an AMD 1800X CPU, 32 GB of RAM, and a GTX 980Ti Graphics card. The other computer tested is the computer on-board the aircraft. This is a Nvidia Jetson TX2. Each step described in the algorithm was measured for how long the step took in milliseconds. Additionally a visualization step was measured but this does not need to run for the algorithm to work and the time for the UGV intermediate plan was measured. The computation times were drawn from averaged values running in the same environment. The measured times are presented in Figure 4.19. The results show that the longest step is the intermediate plan generation for the UGV. This is mostly due to having to access a lot of information along the path. This step can be run while the UAV is moving to a waypoint to increase operational speed. The longest step of the RRT Explore algorithm is the unexplored Tree generation. This makes sense as the unexplored tree is regenerated each time unlike the explored tree which takes considerably less time.

This could be sped up using a more clever way to generate the unexplored tree allowing it to be modified instead of entirely regenerated. The connection, mask generation, and KD-Tree generation all take very little time under 70ms for both computers.

On average the TX2 took 1.7 times longer to run the algorithm but with a cycle time of almost under 1000ms this algorithm runs fast enough on the aircraft to finish exploration in a reasonable amount of time and does not contribute much to the overall operation time. This 1000ms is almost 66% from the UGV intermediate plan method which is not needed for the RRT Exploration algorithm.

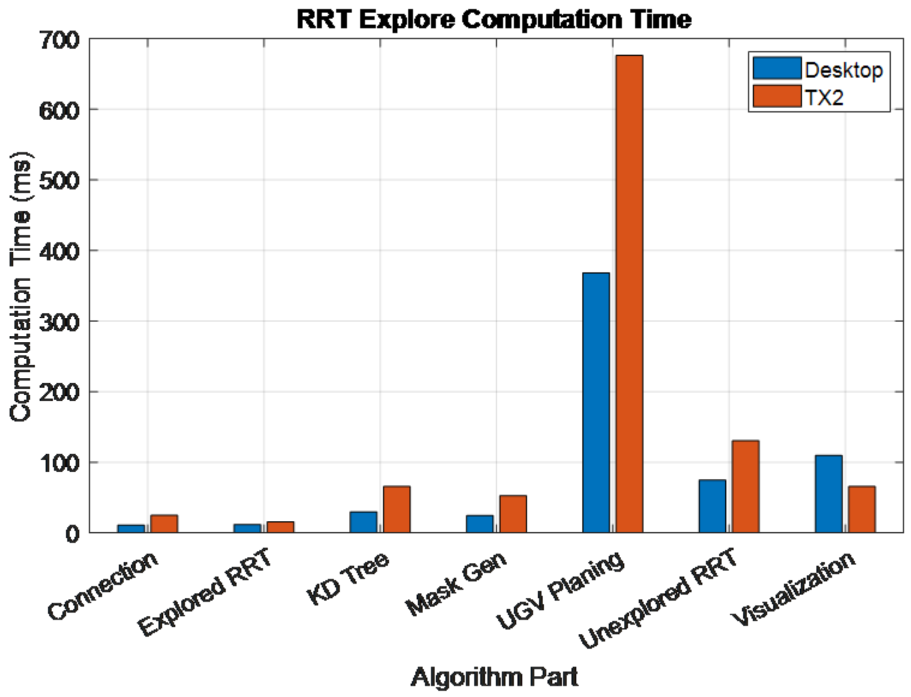


Figure 4.19: Computational time of each step in the RRT Explore Algorithm

4.2 UGV Intermediate Plan Method Results

The next tests were done on the UGV intermediate plan method to validate its performance. Two environments were used. The demo 8 environment was used to simulate a simple environment with obstacles but without dead ends. The demo 9 environment was used to simulate an environment with a dead end. The follow back distance parameter was varied from 0 to 10 meters and 30 simulations were run for each follow back parameter.

The results from the no dead end environment is plotted in Figure 4.20. Both the total distance traveled by the UGV and the total mission time (time from start to UGV reaching goal) were plotted. For this environment the total distance traveled by the UGV slightly decreased as the follow back distance increased which is as expected. The total distance should decrease with increased follow back distance because the UGV holds longer for a valid path which results in a more accurate path which reduces distance. Especially for the really long follow back distances it was observed that sometimes the UGV was held until the entire plan was generated. The total mission time increased with follow back distance. This is as expected as holding the UGV further back increases the time spent. Since the environment has no dead ends larger follow back distances do not benefit mission time since the UGV never has to back track.

The results from the environment with a dead end is plotted in Figure 4.21. The total UGV distance shows a much more dramatic decrease in distance traveled as compared to the no dead end environment. This larger difference is due to the fact that shorter paths are generated once the UAV is done exploring the dead end and the follow back distance holds the UGV further from the end of the dead end. Additionally the extra back-tracking adds distance. The total mission time did not increase as much as with the no dead end environment. This is because the time savings from less back-tracking out of the dead end counteracts the time cost from holding the UGV in place longer.

Overall the proposed method for controlling the UGV when the entire global plan is not yet known shows that holding the UGV back can result in distance traveled savings but not necessarily mission time savings. In environments with no dead ends holding the UGV back only increases mission time. Only when there are possible dead ends should the UGV start to be held back more. It should be noted that this is also most likely a function of the UAV and UGV speed as if the UAV and UGV can move similar speeds than there will be no time savings from holding the UGV back. If the UGV is significantly slower there should be a point when holding the UGV back will result in time savings.

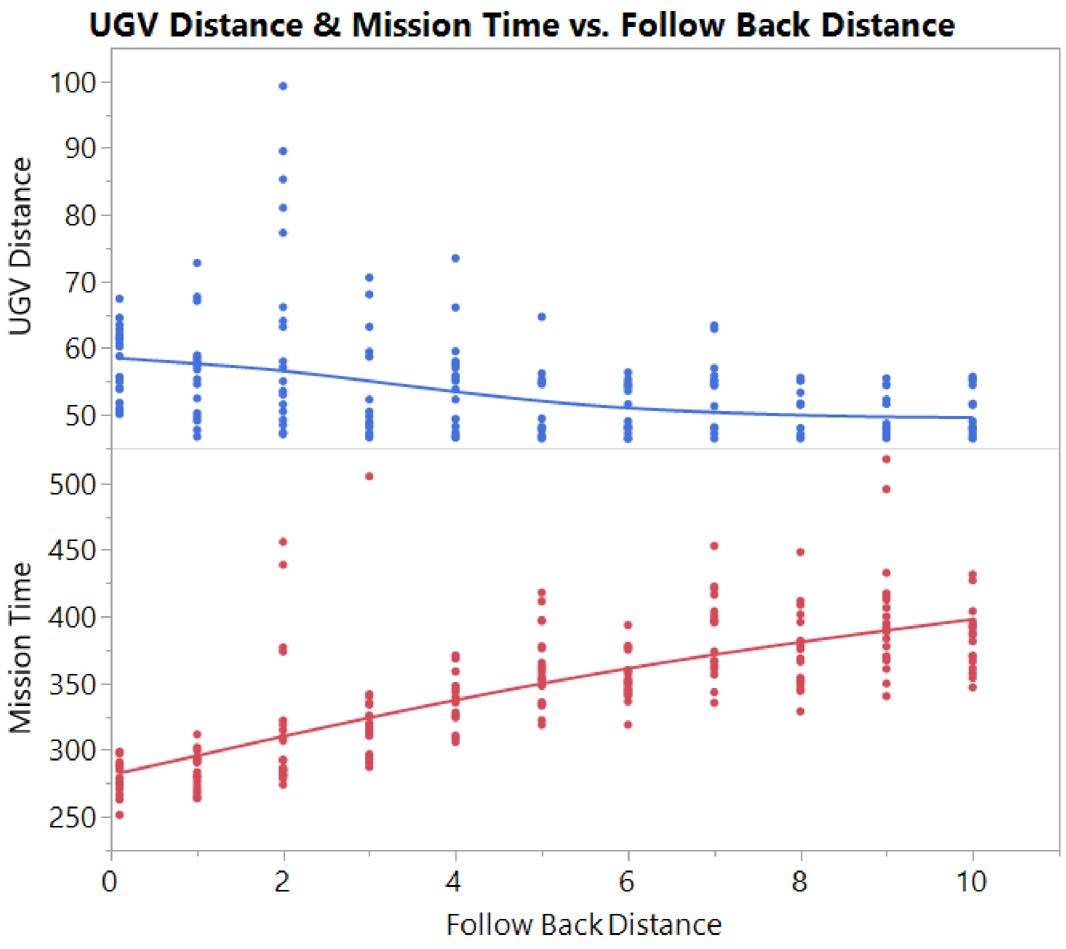


Figure 4.20: Mission time and UGV distance for demo 8 environment without a dead end

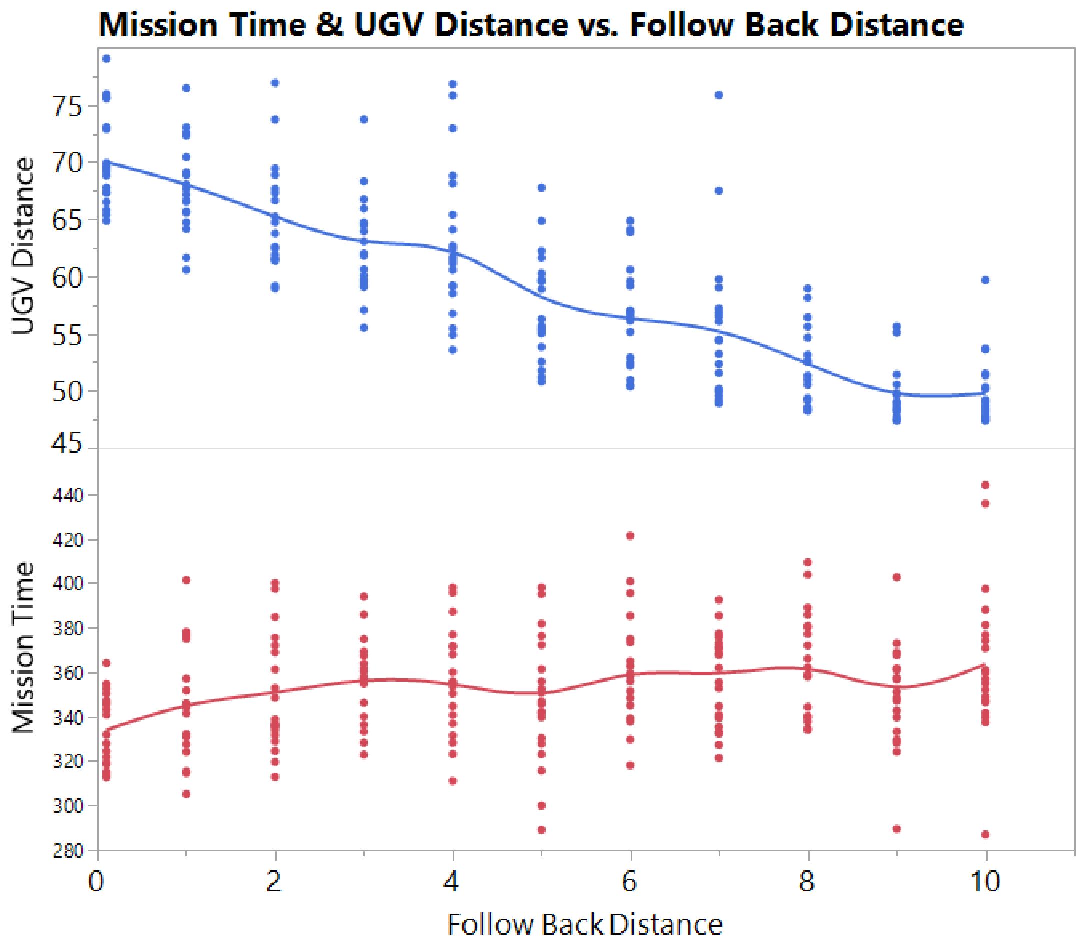


Figure 4.21: Mission time and UGV distance for demo 9 environment with a dead end

4.3 Field Testing

The Dijkstra frontier method was tested in the field as a part of a larger systems test. The test layout is shown in Figure 4.23. To test the system the UGV was placed near a large barn. Once the air vehicle has taken off and is over the ground vehicle the user specified a goal around the corner of the barn using the OCS. The UAV then explored around the building. Two snapshots from the OCS are shown in Figure 4.22 showing the exploration. The result of the experiment was that the UAV was able to successfully explore and find a path for the UGV. Only two issues presented itself during the test and were not related to the exploration algorithm. The first was that the stereo system on the aircraft was slightly out of calibration and was using an older set of cameras which have since been upgraded. Due to this the final UGV plan shows an incursion with the building where the stereo camera failed to see the building. The second was that obstacle inflation in the cost map was not yet properly handled and the UGV wanted to clip the corner of the building. Overall the tests outdoors showed a good and quick exploration of the environment on real hardware in the real world.

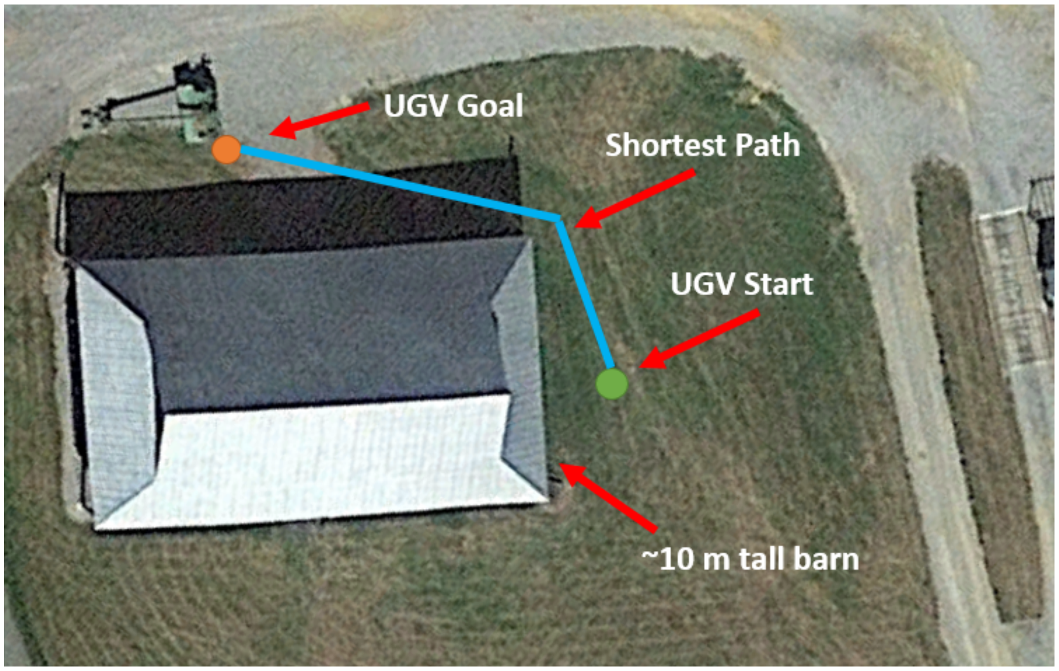


Figure 4.22: Outdoors Demonstration Setup

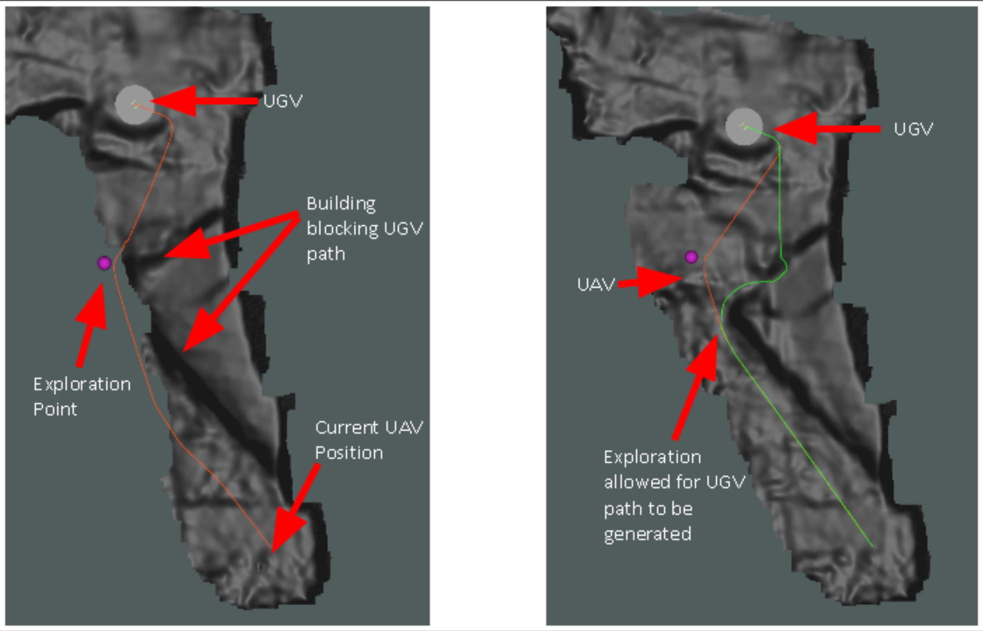

UGV
Building blocking UGV path
Exploration Point
Current UAV Position
UGV
UAV
Exploration allowed for UGV path to be generated

Figure 4.23: Exploration using Dijkstra Frontier Method in the field

Chapter 5

Summary & Conclusions

This work presents a framework and algorithms for the directed exploration of an environment with a UAV to find a path for a ground vehicle in an unknown environment. Two algorithms for exploration were presented. The Dijkstra Frontier method which uses Dijkstra's algorithm to develop a plan which is allowed in unknown space. From this plan, a frontier for exploration is identified. The second method was an RRT based method called RRT Explore. This method uses a bi-directional RRT, where each side of the RRT is confined to known and unknown space. From this, a connection step is done and possible frontiers are found and selected from using a cost function. The RRT Explore method was developed to overcome some of the shortcomings of the Dijkstra Frontier Method which the results showed has been accomplished. In multiple demonstration environments setups the RRT explore method equally performed or outperformed the Dijkstra Frontier method. The Dijkstra Frontier method was shown to work on the real world system. Additionally, the RRT Explore method was shown to execute on the on-board computer fast enough to run this in the real world on the system.

The second part of this work was the navigation framework for controlling the UGV. When a complete plan is not yet known there are multiple options for where to send the UGV. The framework developed used a Dijkstra generated plan to the frontier the UAV is exploring. Along this plan a danger value was calculated for each grid cell the path traversed based upon the proximity to obstacles that touch unknown space. Using this danger value the UGV was sent as far down the path as possible that does not exceed a specified danger threshold along the remainder of the path. This was tested in two environments. An environment with and without a dead end. The results showed that the method reduces the total distance traveled by the UGV in both environments. It also showed that when there is no dead end the further the UGV is held back from danger the longer the mission takes. When there is a dead end the mission time still increases but only slightly. Thus this method can be used and should be tuned by the operators for the type of environment its operating in and whether mission time or UGV distance traveled is more critical.

Only the Dijkstra frontier method was tested on the real system and showed successful operation. The Dijkstra Frontier, RRT Explore, and UGV intermediate plan methods were all validated using Gazebo simulation. Overall this framework works well for generating valid plans for the UGV and getting the UGV to its goal in a timely manner.

Bibliography

[1] Abraham Bachrach, Ruijie He, and Nicholas Roy. Autonomous flight in unknown indoor environments. *International Journal of Micro Air Vehicles*, 1(4):217–228, 2009. doi: 10.1260/175682909790291492.

[2] W. Burgard, M. Moors, C. Stachniss, and F. E. Schneider. Coordinated multi-robot exploration. *IEEE Transactions on Robotics*, 21(3):376–386, June 2005. ISSN 1552-3098. doi: 10.1109/TRO.2004.839232.

[3] K. Cesare, R. Skeele, , , and G. Hollinger. Multi-uav exploration with limited communication and battery. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2230–2235, May 2015. doi: 10.1109/ICRA.2015.7139494.

[4] National Research Council. *Technology Development for Army Unmanned Ground Vehicles*. The National Academies Press, Washington, DC, 2002. ISBN 978-0-309-08620-2. doi: 10.17226/10592. URL https://www.nap.edu/catalog/10592/technology-development-for-army-unmanned-ground-vehicles.

[5] Barry Bahler (FEMA). Debris field in oklahoma, 2008. URL https://commons.wikimedia.org/wiki/File:FEMA_-_35225_-_Debris_field_in_Oklahoma.jpg.

[6] E. Ferranti, N. Trigoni, and M. Levene. Brick mortar: an on-line multi-agent exploration algorithm. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 761–767, April 2007. doi: 10.1109/ROBOT.2007.363078.

[7] Douglas Gage. Ugv history 101: A brief history of unmanned ground vehicle (ugv) development efforts. *Unmanned Systems Magazine*, 13, 02 1970.

[8] Héctor H. González-Baños and Jean-Claude Latombe. Navigation strategies for exploring indoor environments. *The International Journal of Robotics Research*, 21(10-11): 829–848, 2002. doi: 10.1177/0278364902021010834.

[9] Daniel Hernandez-Juarez, Alejandro Chacón, Antonio Espinosa, David Vázquez, Juan Carlos Moure, and Antonio M. López. Embedded real-time stereo estimation via semi-global matching on the GPU. In *International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA*, pages 143–153, 2016. doi: 10.1016/j.procs.2016.05.305. URL http://dx.doi.org/10.1016/j.procs.2016.05.305.

[10] S. Hood, K. Benson, P. Hamod, D. Madison, J. M. O'Kane, and I. Rekleitis. Bird's eye view: Cooperative exploration by ugv and uav. In *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 247–255, June 2017. doi: 10.1109/ICUAS.2017.7991513.

[11] Miguel Juliá, Arturo Gil, and Oscar Reinoso. A comparison of path planning strategies for autonomous exploration and mapping of unknown environments. *Autonomous Robots*, 33(4):427–444, Nov 2012. ISSN 1573-7527. doi: 10.1007/s10514-012-9298-8. URL https://doi.org/10.1007/s10514-012-9298-8.

[12] N. Kalra, D. Ferguson, and A. Stentz. Hoplites: A market-based framework for planned tight coordination in multirobot teams. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pages 1170–1177, April 2005. doi: 10.1109/ROBOT.2005.1570274.

[13] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, New York, NY, USA, 2006. ISBN 0521862051.

[14] David Shim, Hoam Chung, Hyoun Jin Kim, and Shankar Sastry. *Autonomous Exploration In Unknown Urban Environments For Unmanned Aerial Vehicles*. 2005. doi: 10.2514/6.2005-6478. URL https://arc.aiaa.org/doi/abs/10.2514/6.2005-6478.

[15] Anthony (Tony) Stentz. Optimal and efficient path planning for unknown and dynamic environments. Technical Report CMU-RI-TR-93-20, Carnegie Mellon University, Pittsburgh, PA, August 1993.

[16] Pratap Tokekar. Lecture notes in robot motion planning astar, February 2017.

[17] Brian Yamauchi. A frontier-based approach for autonomous exploration. In *In Proceedings of the IEEE International Symposium on Computational Intelligence, Robotics and Automation*, pages 146–151, 1997.

Appendices

Appendix A

Gazebo Test Environments

The following appendix shows the different demonstration (demo) environments that were setup in Gazebo to test the framework.

A.1 Demo 1

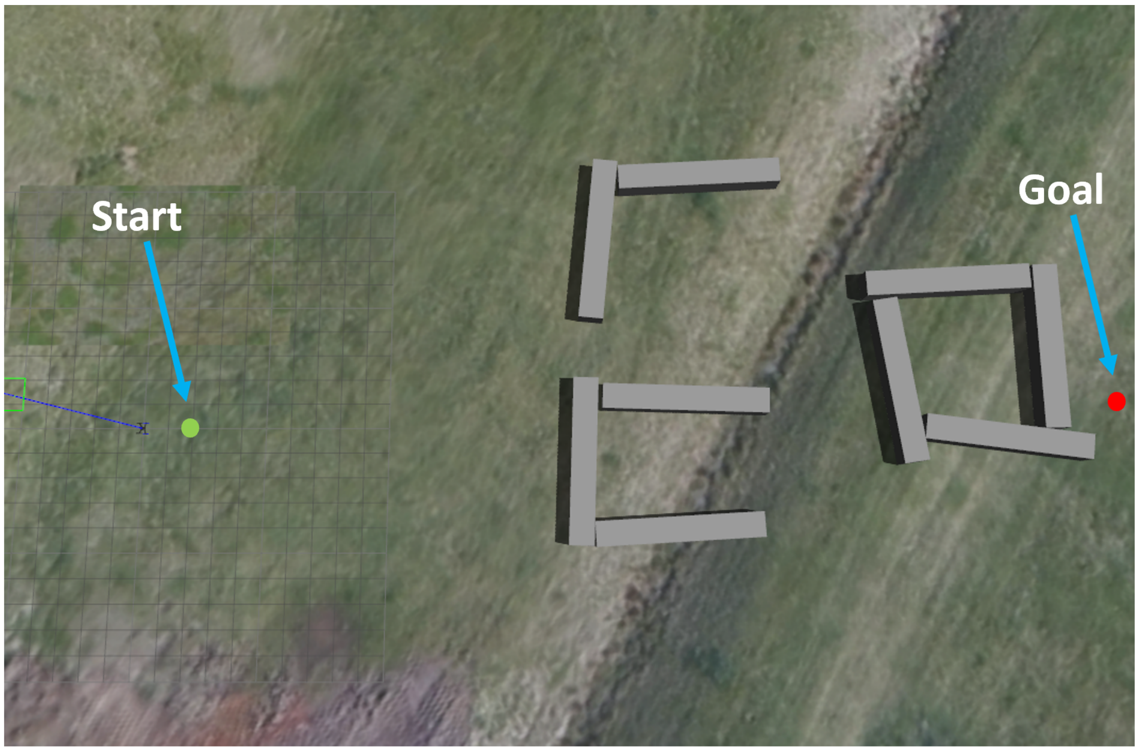


Figure A.1: Demo 1

A.2 Demo 2

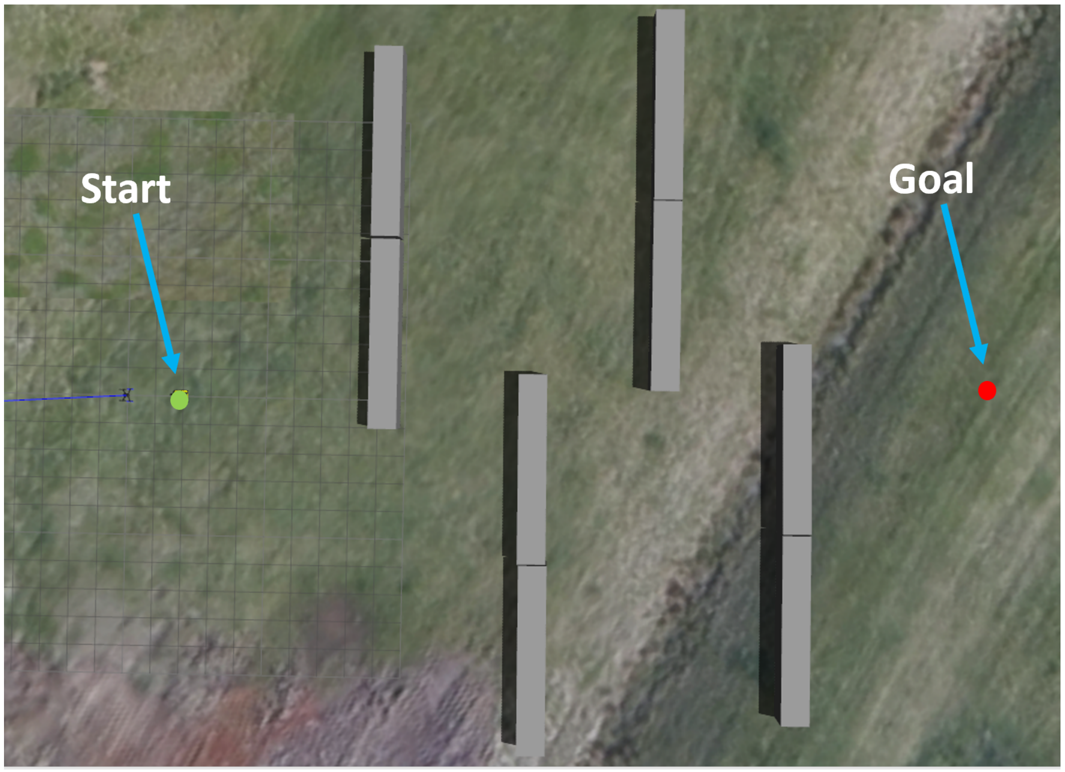


Figure A.2: Demo 2

A.3 Demo 3

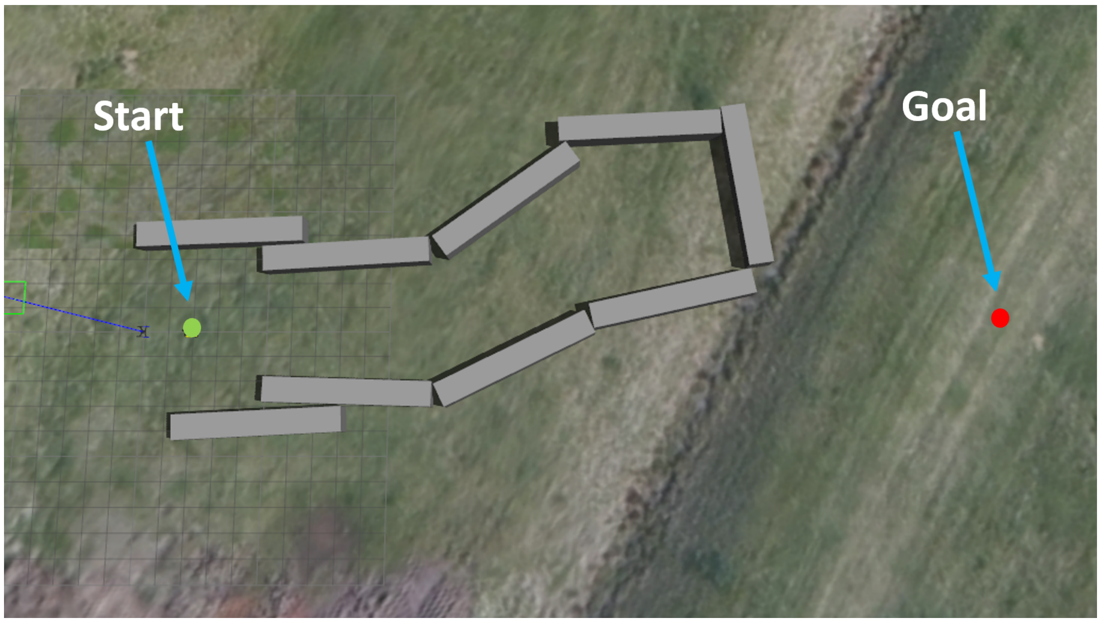


Figure A.3: Demo 3

A.4 Demo 4

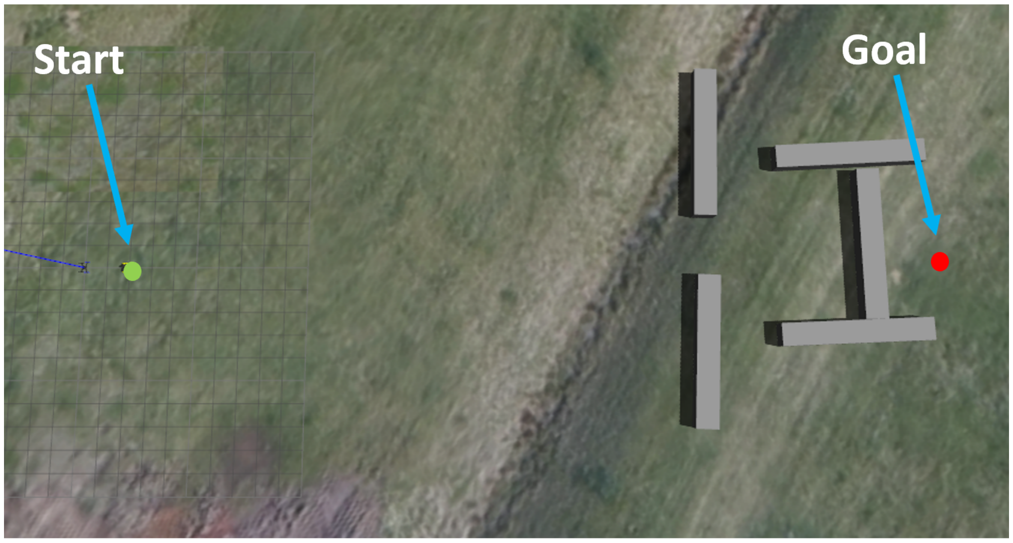


Figure A.4: Demo 4

A.5 Demo 5

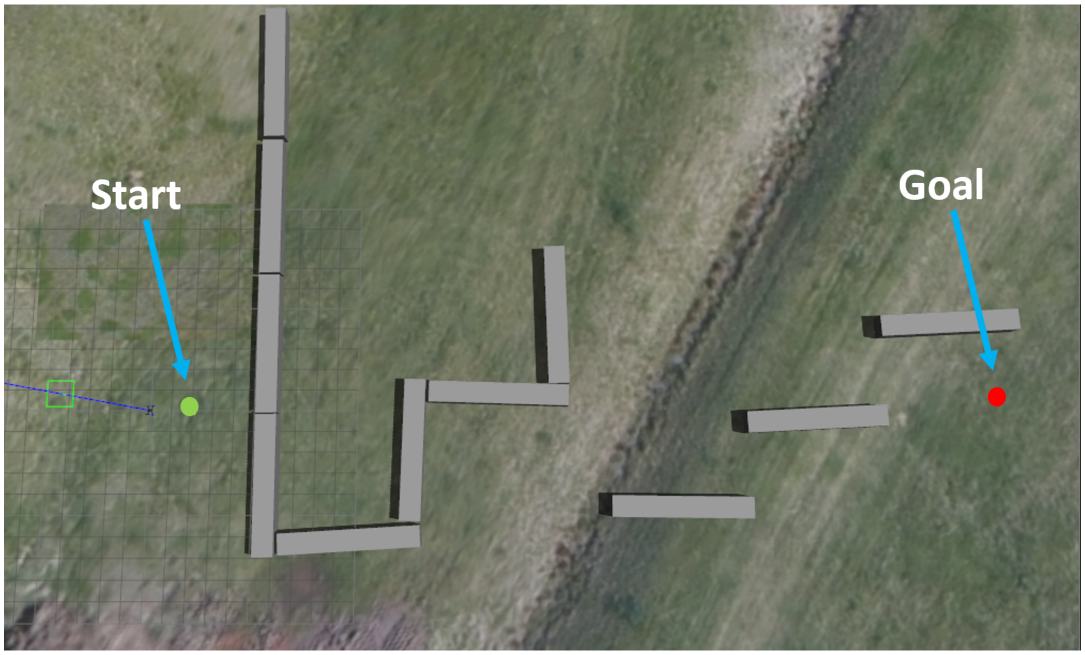


Figure A.5: Demo 5

A.6 Demo 7

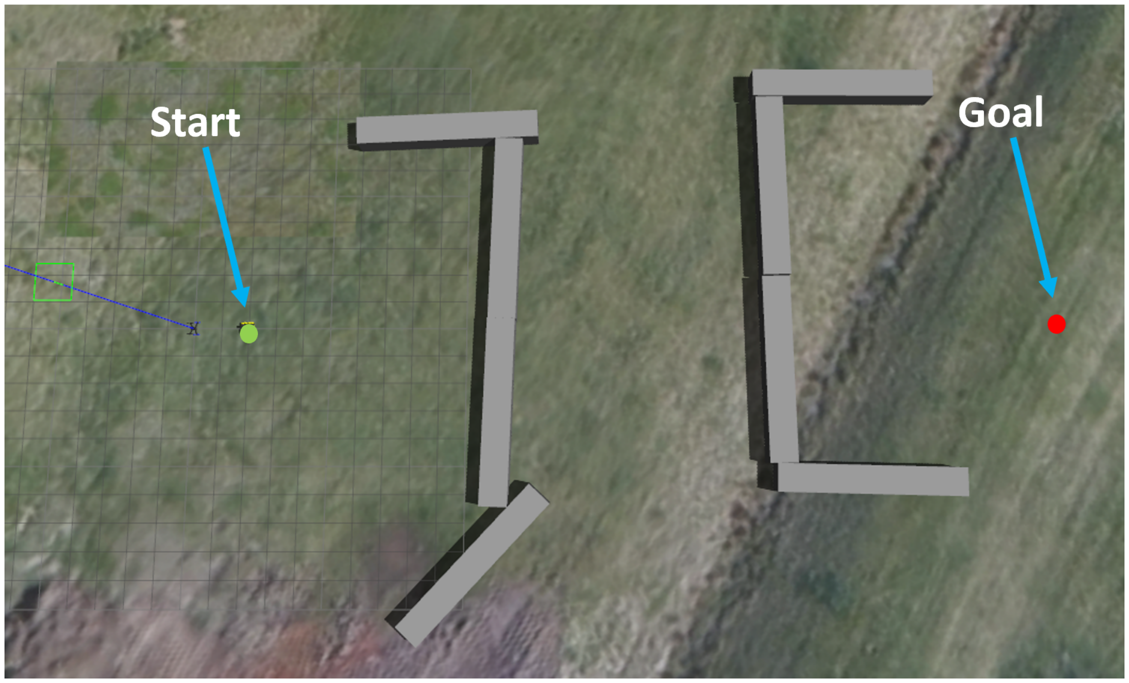


Figure A.6: Demo 7

A.7 Demo 8

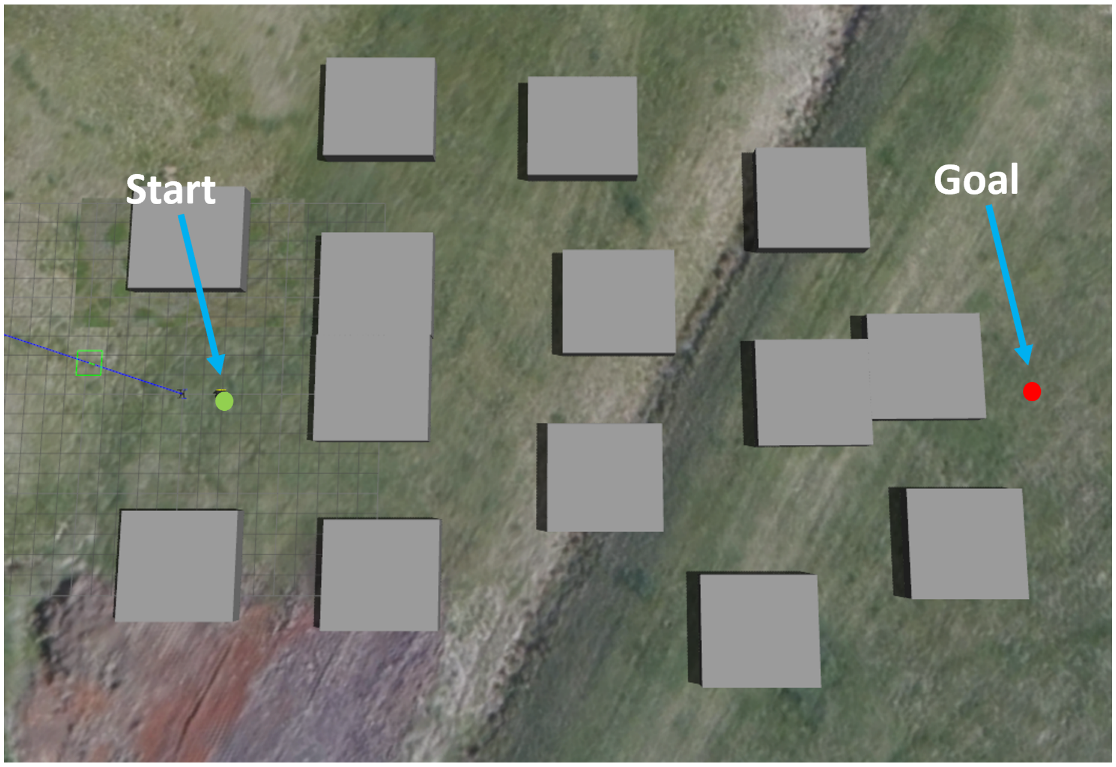


Figure A.7: Demo 8

A.8 Demo 9

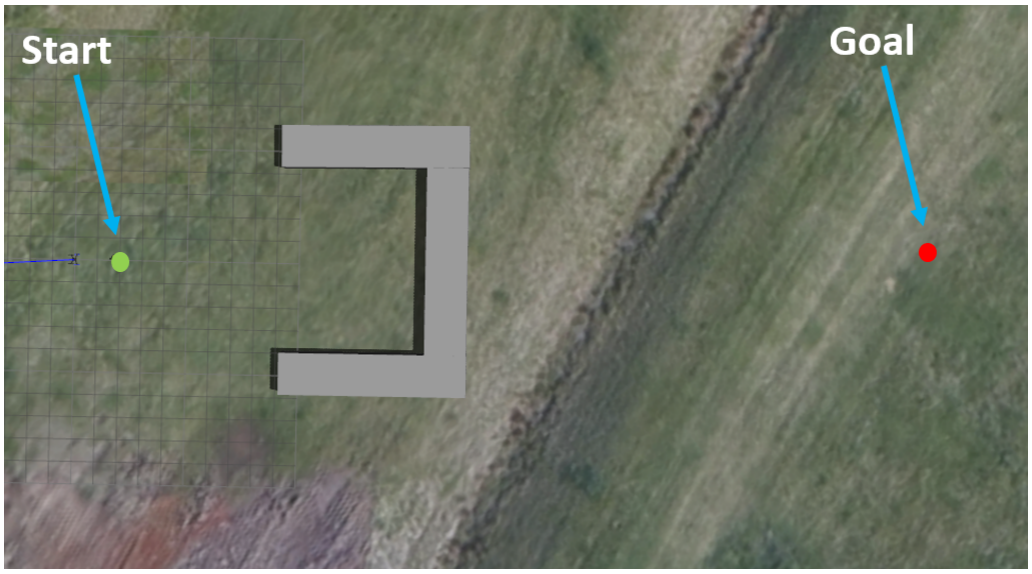


Figure A.8: Demo 9