Jeffrey Williams • David Neilsen • Austin Bachman • David Hull • Robert Trimble

Path Planning Project Report

University of Nevada, Reno
CS 419: Intro to Aerial Robotics
Professor - Kostas Alexis

Quadrotor Path Planning Algorithm Comparison and Analysis using OMPL

TABLE OF CONTENTS

I. Introduction

A. Motivation and Overview of Project

Every day, more applications for robotic systems arise, and following close behind an ever growing number of diverse systems created to fill these application. Common to nearly all is the necessity for motion. There are many different types of motion planners, but for this project, we will be only considering sampling-based approaches. Since each planner is different, it follows that they would be suited to different applications. Thus, it becomes necessary to develop a system to determine which planning algorithm is best suited to a specific scenario. In the case of this project, our team would like to determine the best planning algorithm for point-to-point navigation for multi rotor aerial robots.

The goal of this project was to program, compare and analyze several essential path planning algorithms. Namely Probabilistic RoadMap (PRM), Expansive Space Trees (EST), Kinodynamic motion Planning Interior-Exterior Cell Exploration (KPIECE1), Rapidly-Exploring Random Tree (RRT) and its variant (RRT*). But before we get into describing each these algorithms let's outline the techniques we used to run and analyze our code. Using OMPL we created several virtual environments, including, abstract, cubicle, home and easy environments. ran each the algorithms in them and logged the results. We then analyzed each algorithm in terms of its time to find solution, correctness of solution, number of graph states and memory use.

B. Path Planning Algorithms

Let's start with PRM, this method uses to step to compute a solution. The first step creates the roadmap finding all collision free nodes and edges, the second searches this roadmap to find a sequence of edges connecting the edges between the starting and end points.

The second path planning algorithm we researched was EST or Expansive Space Trees. This algorithm was designed with goal of visiting less explored areas of the configuration space. It works by incorporating the idea of visibility to the roadmap. Another characteristic that sets it apart from PRM is that it starts trees at both initial and goal points, sampling the close neighbor nodes of both and randomly generating two trees until they intersect. Another key aspect of this algorithm that helps it achieve the goal of exploring the less explored is the use of node density weights allowing to to have a bias toward areas with fewer nodes.

The third algorithm KPIECE operates rather differently than the first two. It requires only a view of the state space unlike PRM and EST which need to sample the space state and distances. This algorithm works by starting in a chain of cells and expanding outward. It does this by keeping track of its motions, and expanding/adding cells each time it finds a collision free cell then recalculating to the next.

Fourth and fifth are RRT and RRT*. In a nutshell this one works by sampling a random node and taking a collision free steps toward it within the maximum step size. It repeats this until it reaches the random node. It is useful in environments with lots of obstacles, however, it is not a complete solution to path planning which is why it is often used in conjunction with other algorithms. RRT* is similar to RRT however it has the main difference of finding the optimal solution. To do this it forms a radius around the randomly sampled node and choose the node which produces the least distance from the starting point to the random node. This means that for each step it is calculating the optimal solution until it reaches the goal.

C. Experimental Setup

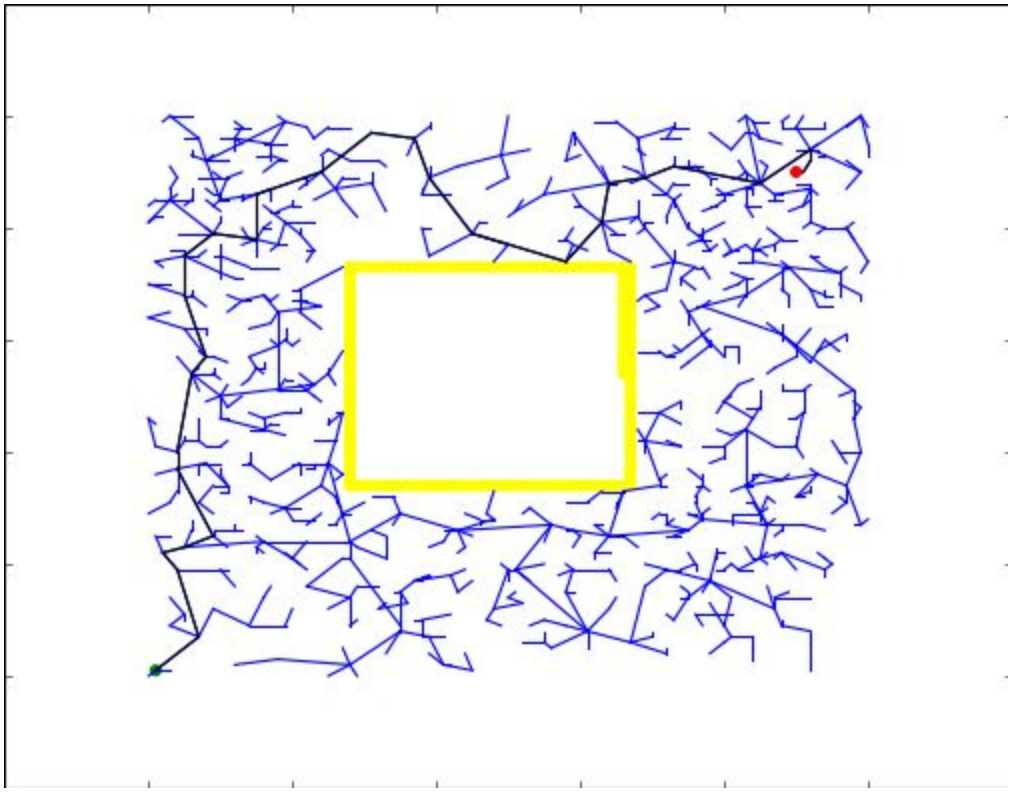
For this experiment, the Open Motion Planning Library (OMPL) will be used. OMPL provides the necessary functionality of implemented planning algorithms, 3D environment and robot models, and tools for benchmarking planners.

To start, a Benchmark object is created. The environment and start and goal positions are then read in from a file and into this object. Then, each planning algorithm that will be tested is added. The Benchmark then runs each algorithm 25 times for a maximum of 10 seconds, and outputs the collected data to a file that may then be processed into a pdf with charts representing this data.

Finally, each planning algorithm is run once more on the problem, and this will output the number of states that were sampled and added to the graph, and the final solution path of states. All of this data is available in the GitHub repository linked at the end of this report.

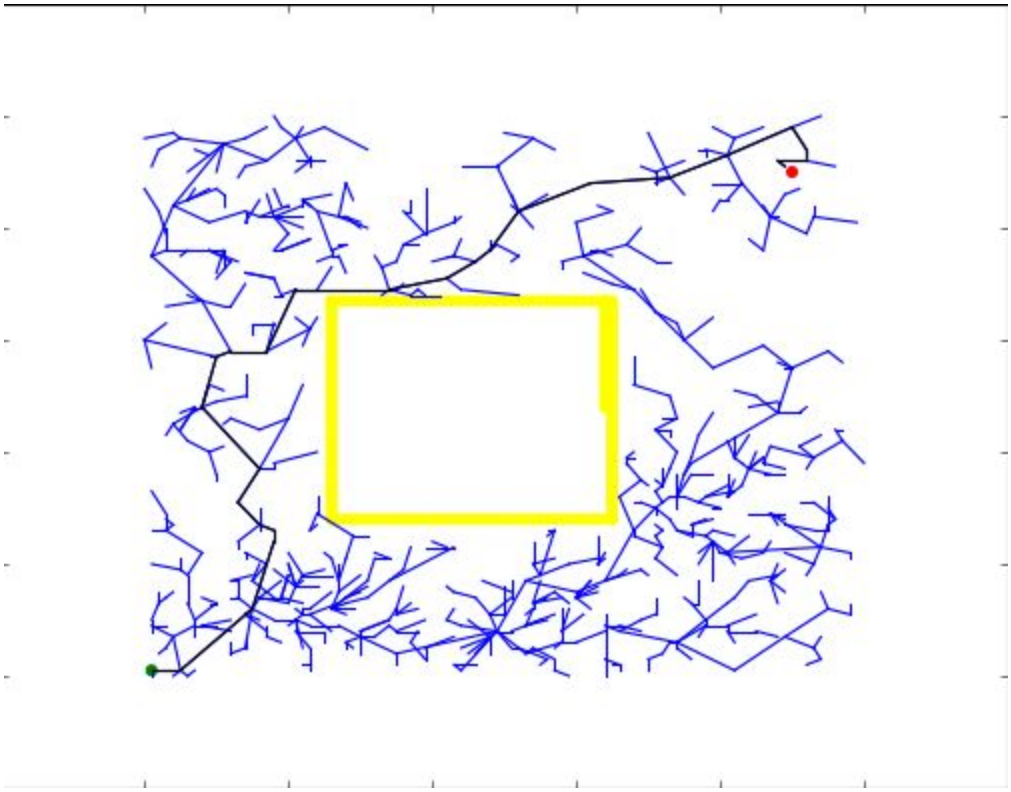
II. RRT and RRT* implementation

Before analyzing the various algorithms of path planning, we decided to recreate one of our own. For this, RRT and RRT* (RRT star) were chosen. Both implementations were written in the Python programming language and can be found in the provided GitHub. Below is an example of RRT with one object in the middle.



RRT works by randomly selecting a point within the known map. It will then create an edge between that point and the closest already established point. The edge (and point) will be discarded, however, if there is an object blocking its direct path, or the point is too far away from any already established points.

RRT* works in a very similar way, except that it has an extra step. Once an edge is created, the new point will check if there are yet additional already established points within a certain radius. If more are found, the algorithm will from what point will create the shortest to the starting point. More importantly though, it will break its current edge if a different one will create a shorter path. Below is an example of RRT* with one object in the middle.

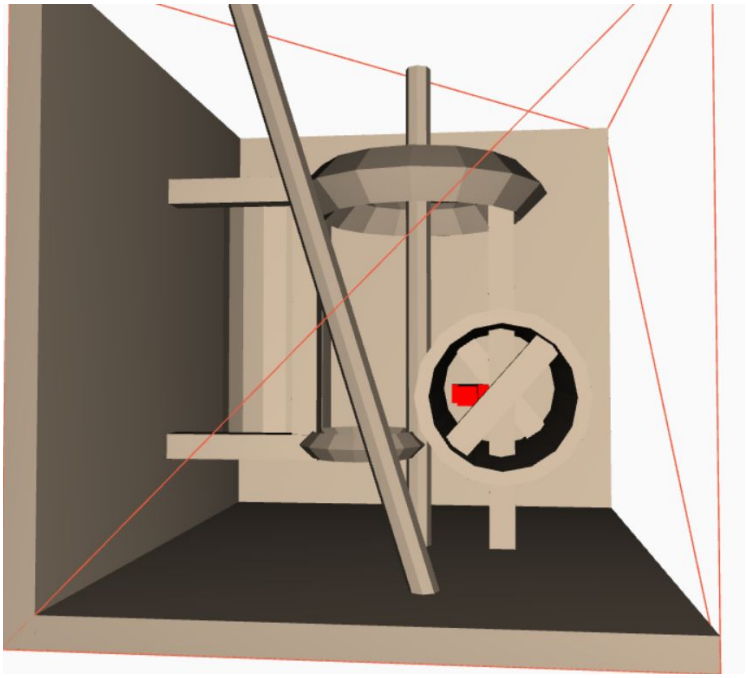


Those are the methods that our own implementation of path planning use. While ours is only in a 2D environment, it can be modified for 3D. Additionally, more objects can be added, the starting and destination locations can be changes, and radius size for RRT* can be adjusted. All of this allows the user to compare the output of RRT and RRT* given different configurations. Overall, our program helped increase our understanding of both RRT and RRT* path planning implementation.

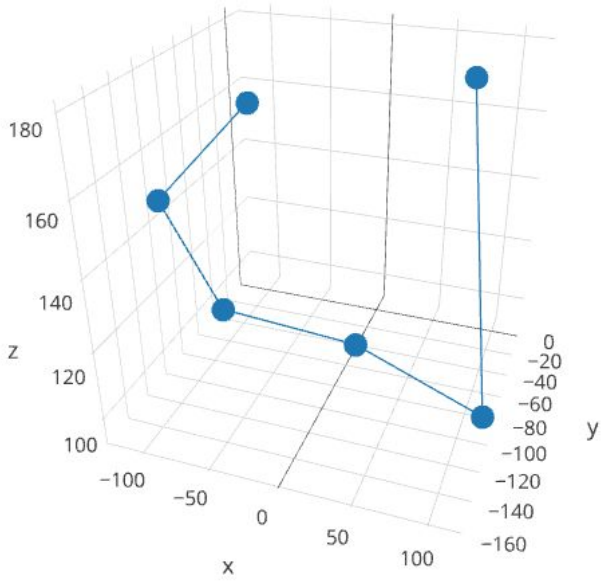
III. Trajectory Analysis

This section will contain for each environment a visualization of the obstacles in the environment, as well as the planned trajectories for each algorithm.

A. Abstract Environment

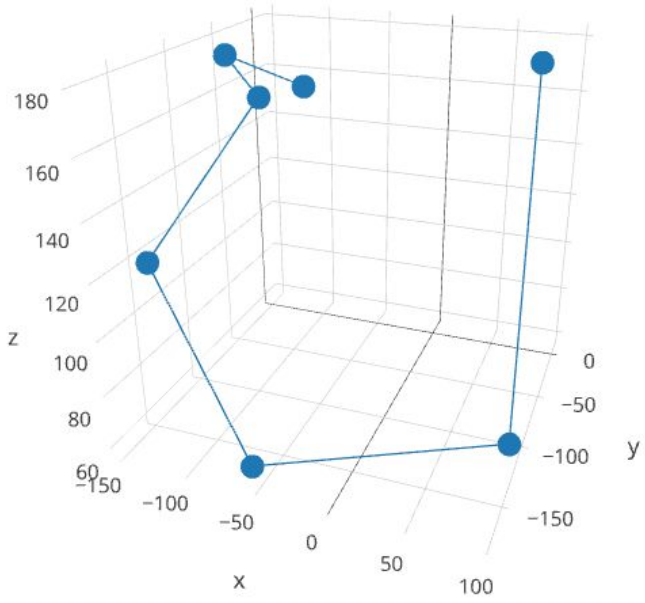


I. RRT*



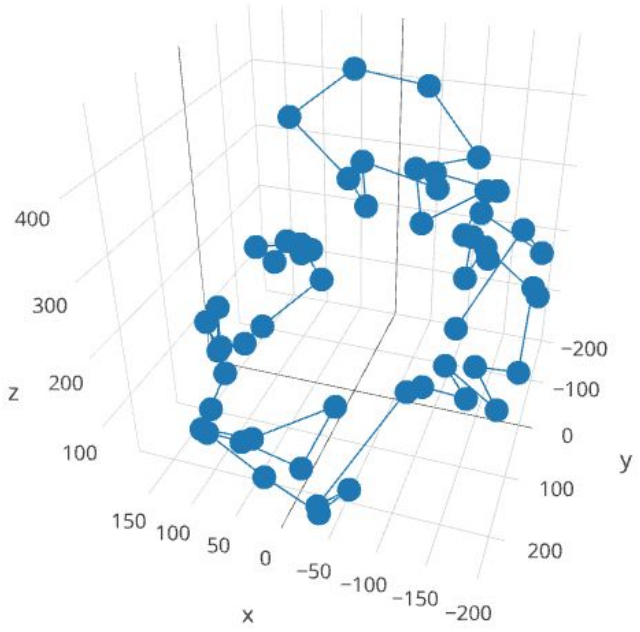
Trajectory for RRT* in the abstract environment.

II. RRT



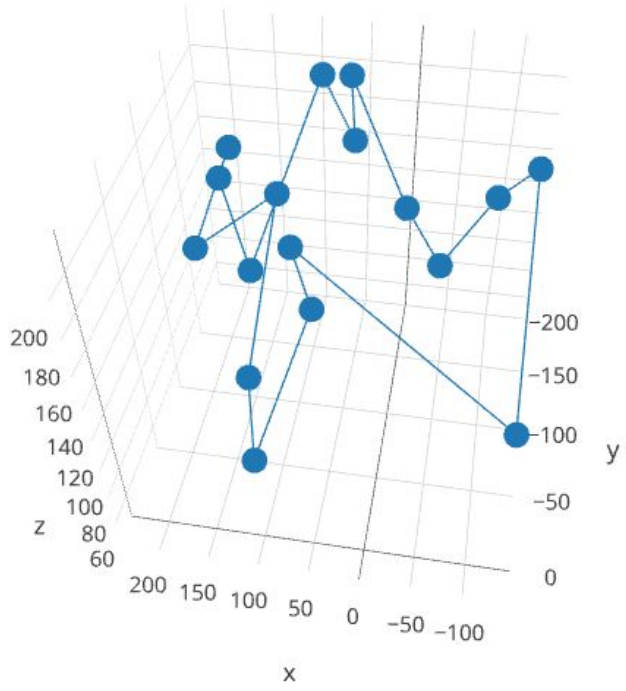
Trajectory for RRT in the abstract environment.

III. KPIECE1



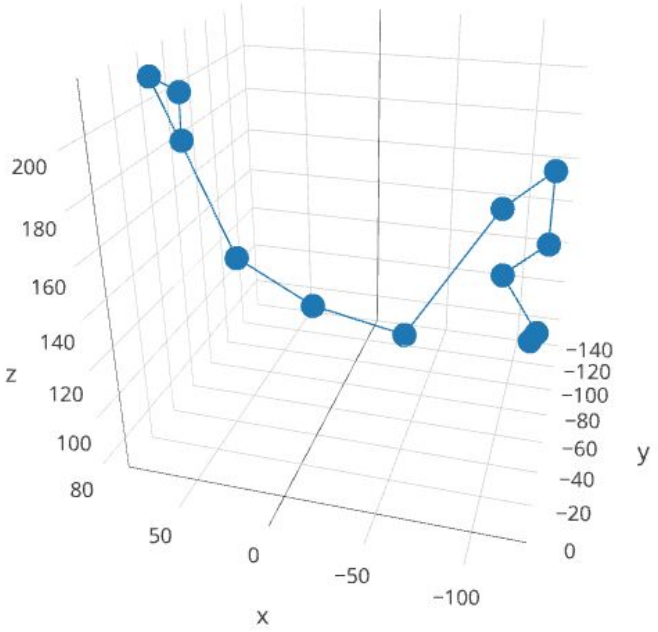
Trajectory for KPIECE1 in the abstract environment.

IV. EST



Trajectory for EST in the abstract environment.

V. PRM

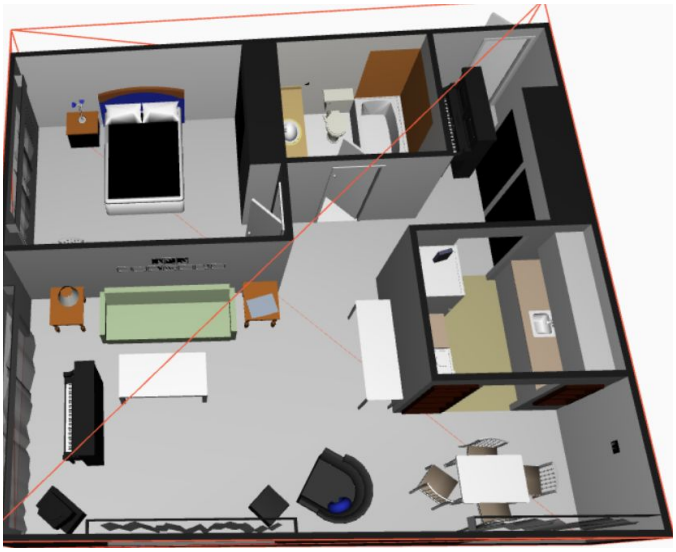


Trajectory for PRM in the abstract environment.

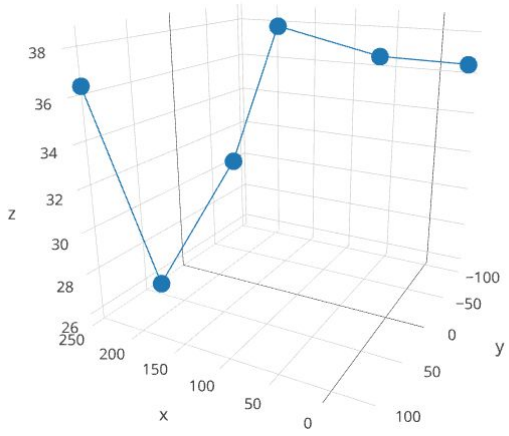
VI. Analysis

It can be seen from the trajectories that in this abstract environment RRT and RRT* used a significantly lower amount of robot states to go from start to finish. PRM uses slightly more states than RRT and RRT*. The highest amount of states was used by KPIECE1. This is most likely due to the randomness of the obstacles in the environment. This randomness is handled well by RRT and RRT* which are random sampling algorithms. On the other hand KPIECE1 and EST are control-based algorithms therefore they use the cell-decomposition method which results in many more states.

B. Apartment Environment

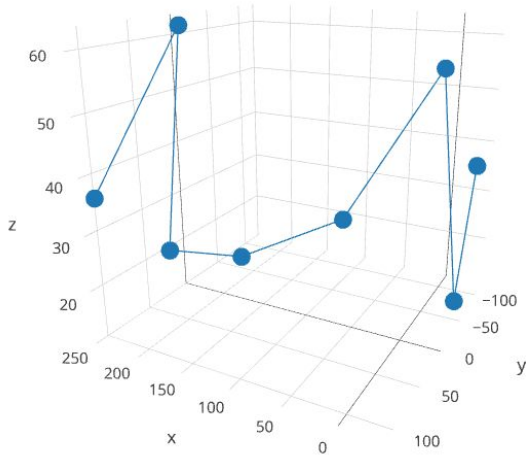


I. RRT*



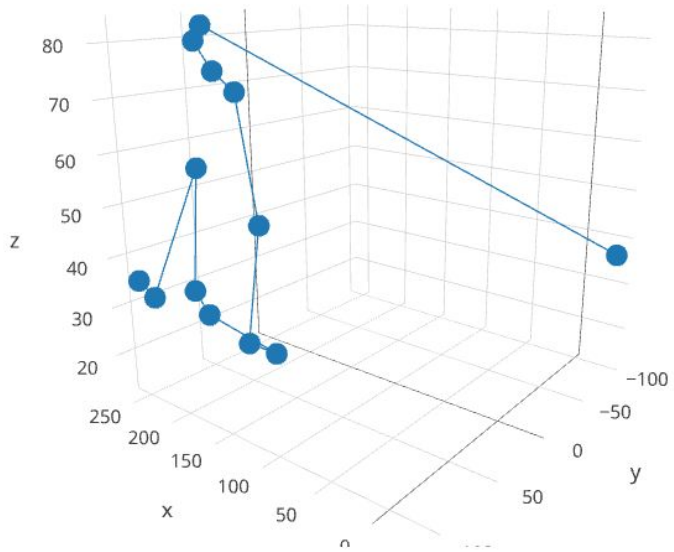
Trajectory for RRT* in the apartment environment.

II. RRT



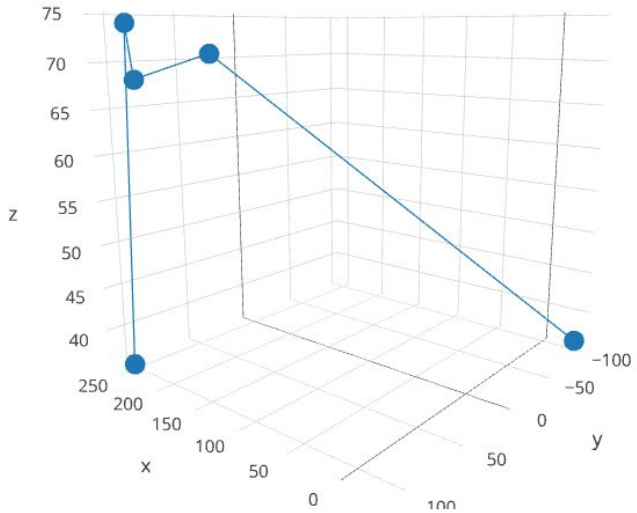
Trajectory for RRT in the apartment environment.

III. KPIECE1



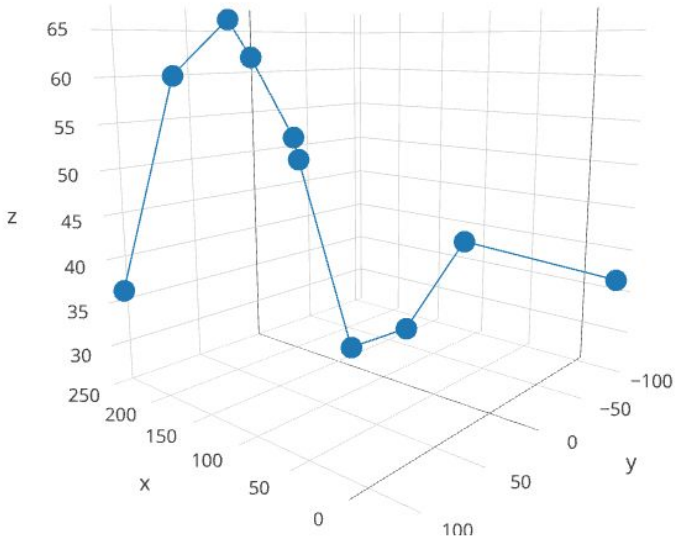
Trajectory for KPIECE1 in the apartment environment.

IV. EST



Trajectory for EST in the apartment environment.

V. PRM

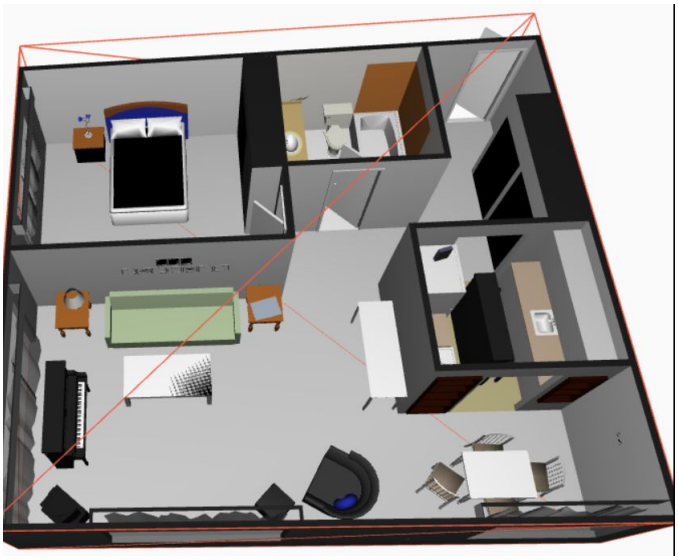


Trajectory for PRM in the apartment environment.

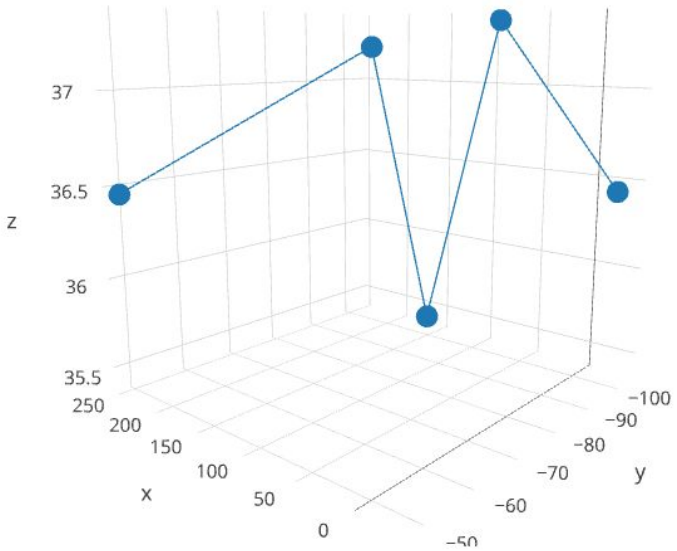
VI. Analysis

For the apartment environment KPIECE again used the most amount of states. RRT* and EST used the lowest amount of states. EST performed well in the apartment complex most likely because of how it works by creating a tree starting from the beginning and the end and moving towards each other. It can be seen in the trajectory that the first jump from the end moving backwards was very significant. RRT struggled slightly most likely due to its random nature.

C. Apartment Hard Environment

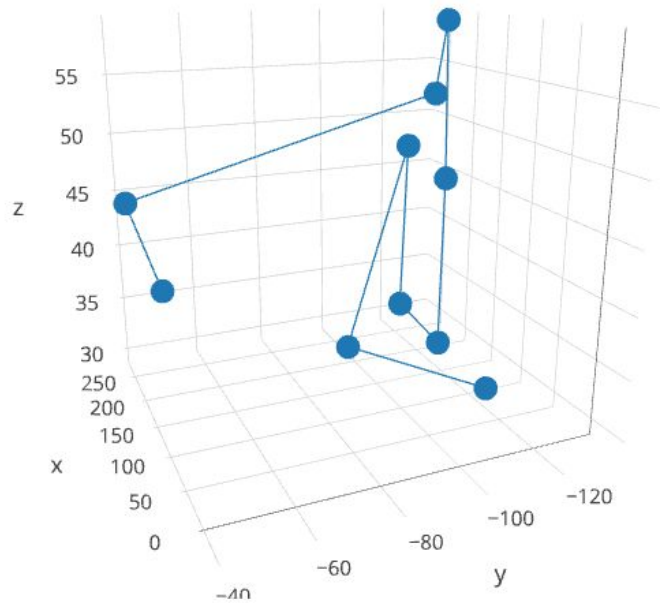


I. RRT*



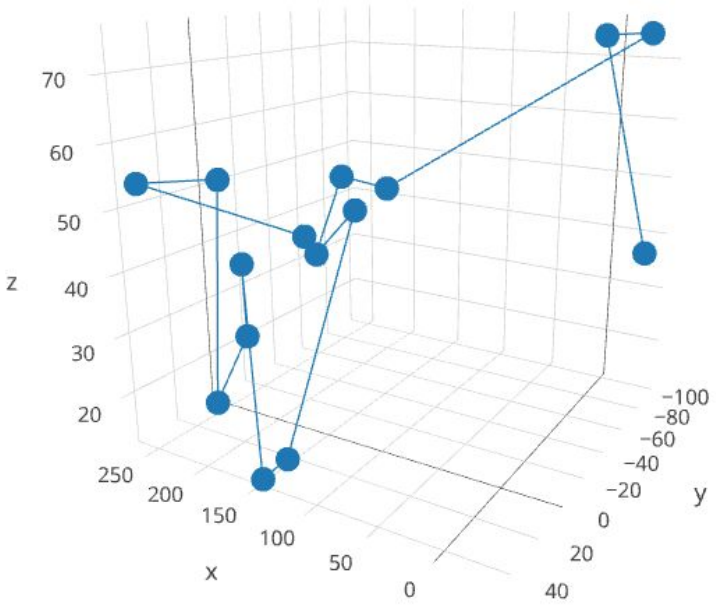
Trajectory for RRT* in the apartment hard environment.

II. RRT



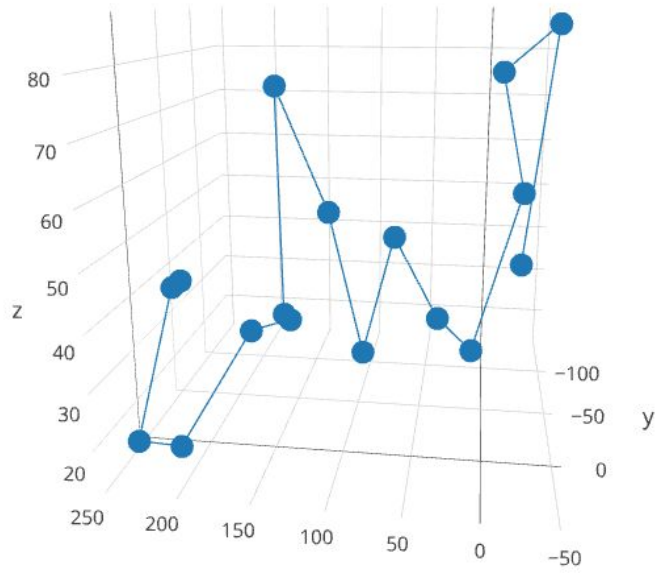
Trajectory for RRT in the apartment hard environment.

III. KPIECE1



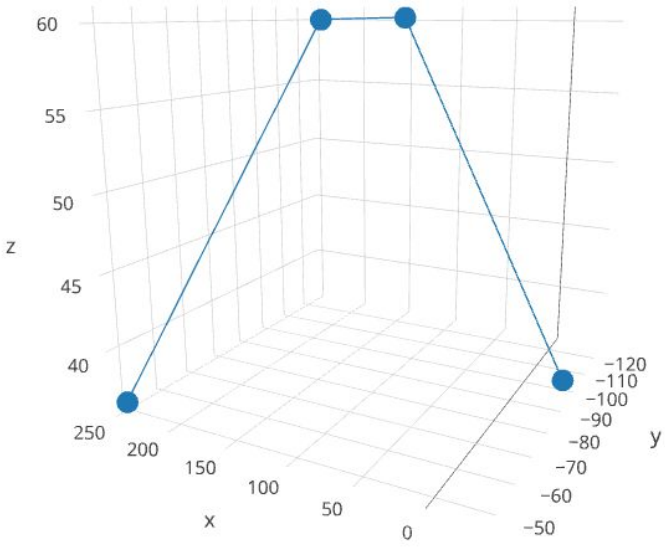
Trajectory for KPIECE1 in the apartment hard environment.

IV. EST



Trajectory for EST in the apartment hard environment.

V. PRM

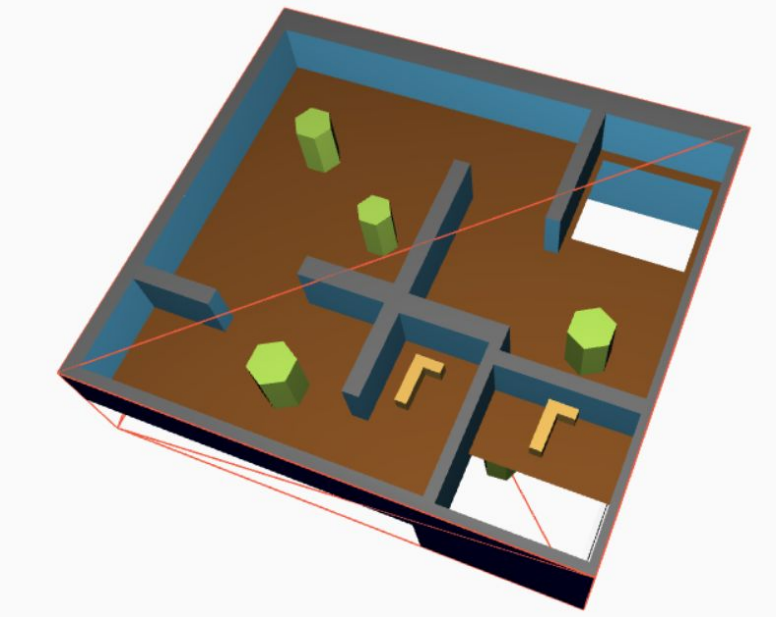


Trajectory for PRM in the apartment hard environment.

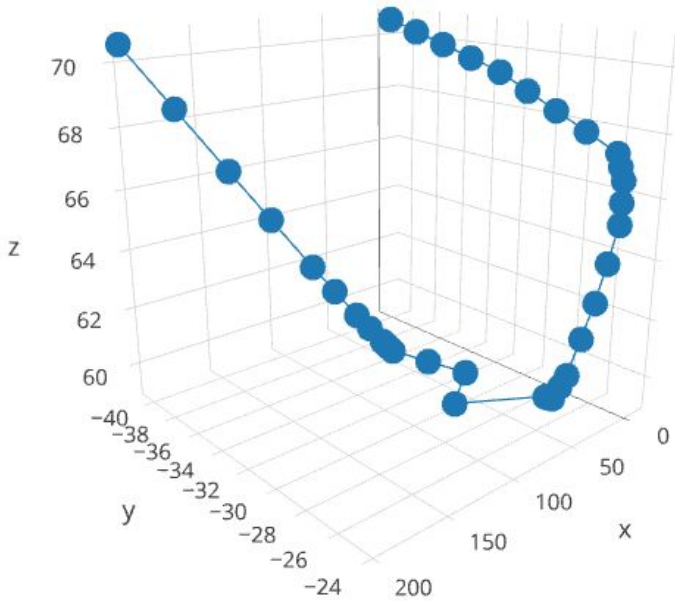
VI. Analysis

For the apartment hard environment PRM and RRT* showed the least number of states. Again we see the control-based algorithms using a significantly higher amount of states than the random sampling algorithms. PRM surprisingly solved this environment with just 4 states.

D. Cubicles Environment

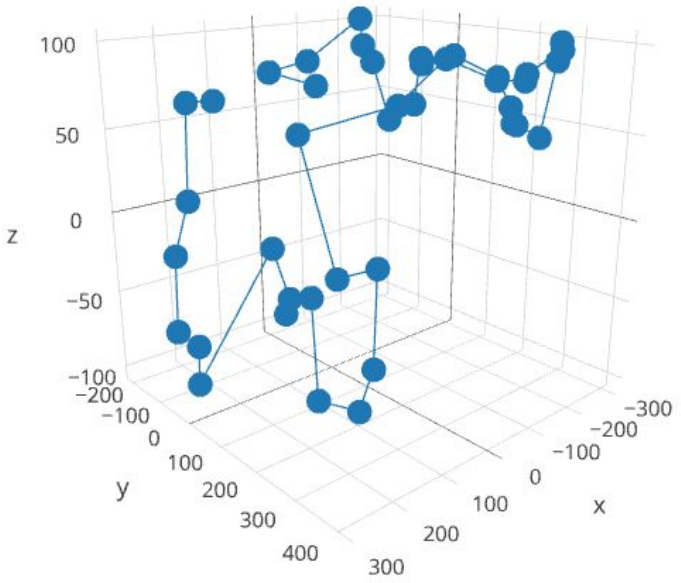


I. RRT*



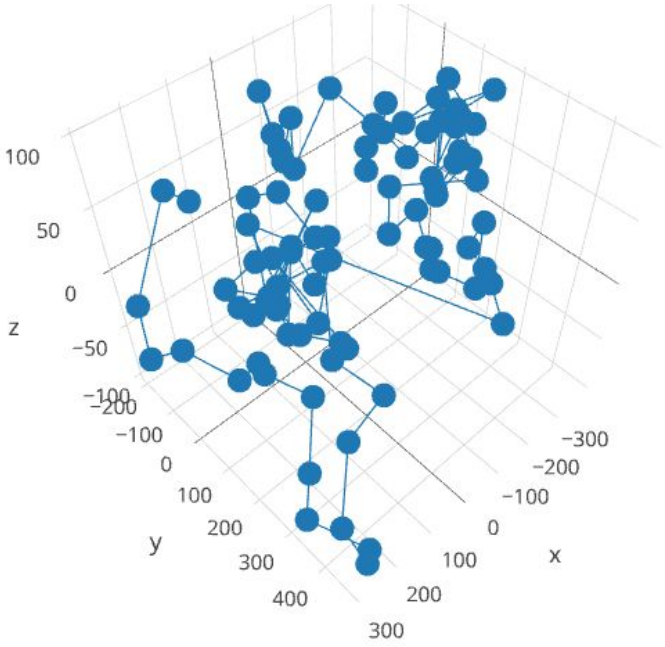
Trajectory for RRT* in the cubicles environment.

II. RRT



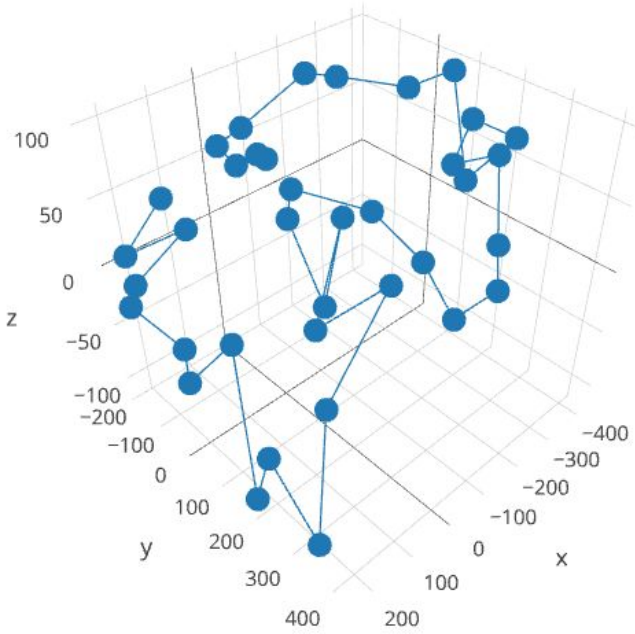
Trajectory for RRT in the cubicles environment.

III. KPIECE1



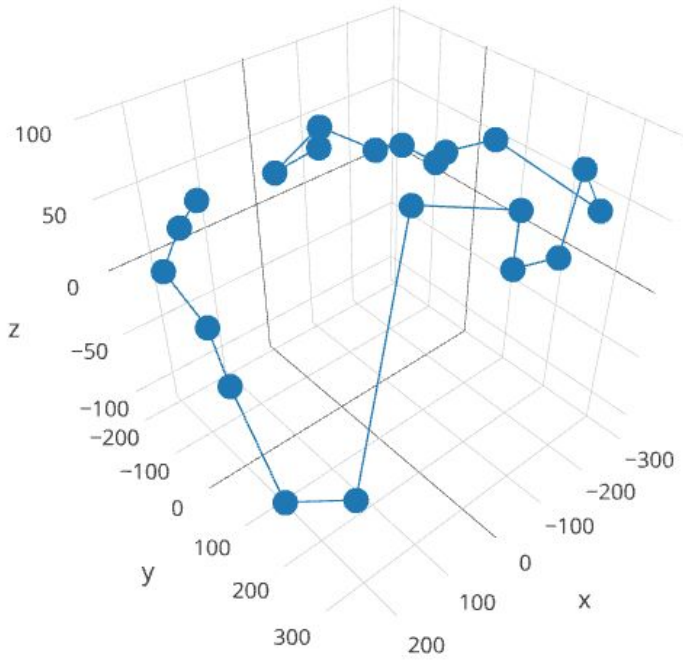
Trajectory for KPIECE1 in the cubicles environment.

IV. EST



Trajectory for EST in the cubicles environment.

V. PRM



Trajectory for PRM in the cubicles environment.

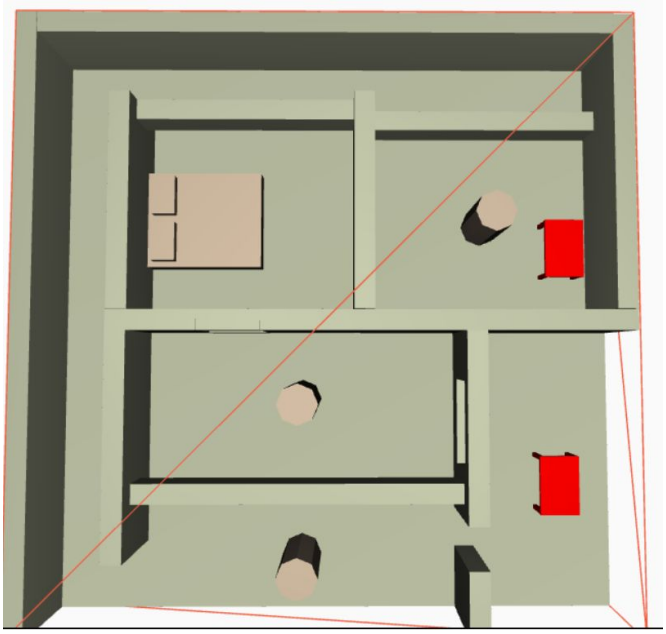
VI. Analysis

The regularity of the obstacles in the cubicles environment caused all algorithms to used an increased amount of states. RRT* found a very elegant horseshoe shaped path while the rest of the algorithms found their own crude solution. KPIECE1 took a significantly large number of states most likely due to its method of mapping the collision areas using vertical lines. The cubicles environment does not facilitate the use of this method as can be seen by the trajectory.

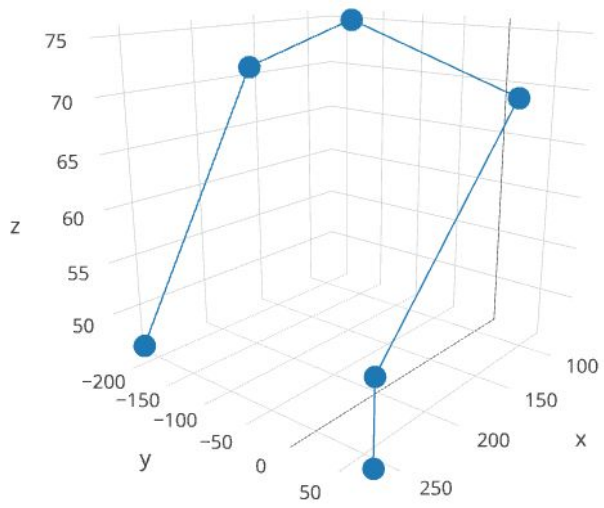
E. Easy Environment

This environment was so simple it took each algorithm no more than 4 states to solve it. For this reason the trajectories have been emitted because they are each painfully similar to one another.

F. Home Environment

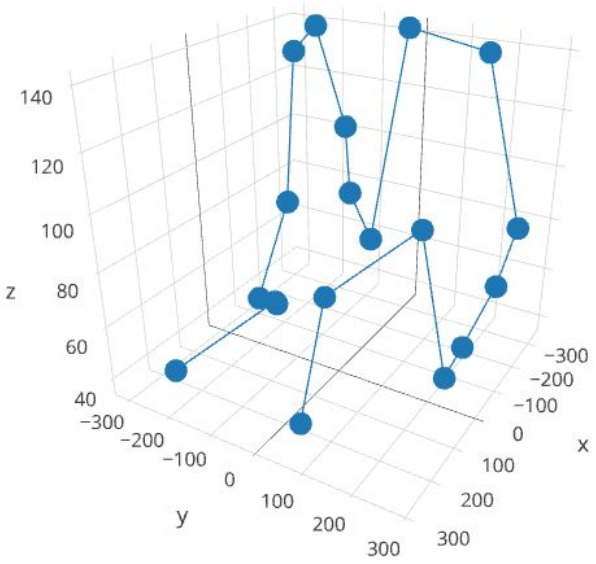


I. RRT*



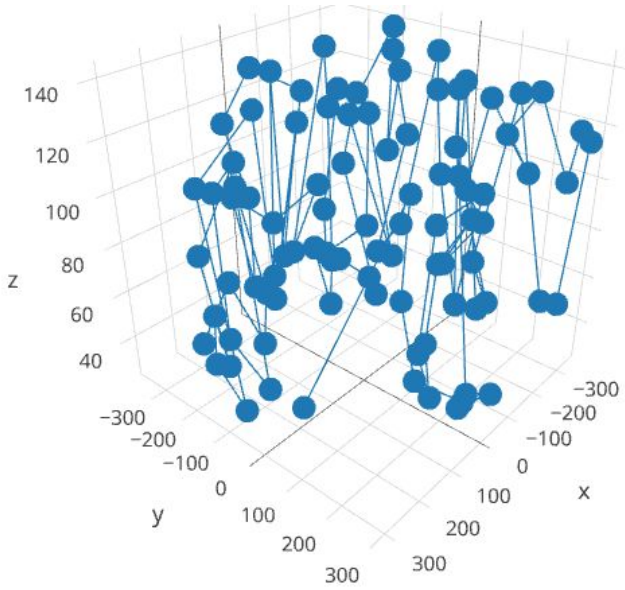
Trajectory for RRT* in the home environment.

II. RRT



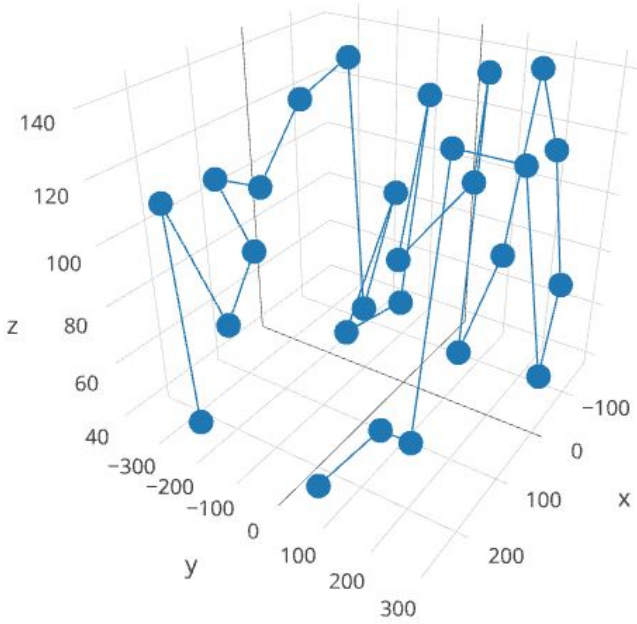
Trajectory for RRT* in the home environment.

III. KPIECE1



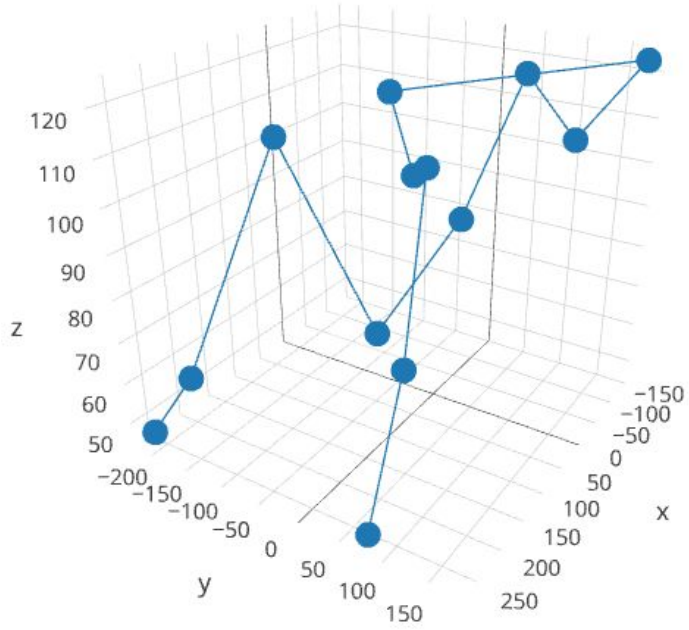
Trajectory for KPIECE1 in the home environment.

IV. EST



Trajectory for EST in the home environment.

V. PRM



Trajectory for PRM in the home environment.

VI. Analysis

For the home environment RRT* had a significantly lower amount of states than the rest of the algorithms. Also, KPIECE1 had a significantly larger amount of states than the rest of the algorithms. PRM and RRT took a similar amount of states that was slightly higher than RRT*. The solution that KPIECE1 found was extremely crude in that its total distance was extremely high compared to the other algorithms.

G. Overall Analysis

RRT* always used a relatively low amount of states compared to the other algorithms and also found relatively elegant solutions. RRT used a slightly higher amount of states than RRT* but significantly lower that the control-based algorithms EST and KPIECE1. RRT's solutions were somewhat elegant relative to the control-based algorithms. KPIECE1 always took the most amount of states and had a significantly higher total distance along its path than the other algorithms. EST found an elegant solution in the apartment environment but in other environments performed similar to KPIECE1 with a slightly less number of states. PRM worked very similarly to RRT in that it had slightly more states than RRT* and sometimes found a somewhat elegant solution.

IV. Results

The results following are taken from a series of simulations ranging from simple to complex and reflect the efficiency and effectiveness of the tested algorithms. These algorithms are RRT*, RRT, KPIECE1, EST, and PRM. These were each run in various environments labeled Abstract, Apartment, Apartment-Hard, Cubicles, Easy and Home. Each setup had its own challenges, and each algorithm had its own approach to these setups and its own efficiency.

A. Time To Find a Solution

The time for finding solutions of these path planning algorithms mostly depended on the complexity and size of the area. This also depends on the sampling points for several of the tested algorithms, though not all stop when finding the first solution.

RRT* (RRT-star) and PRM continue to run the entire length of time allotted as they are tailored to find the optimal route within their constraints, and so continue operating. This results in a more efficient path through denser obstacles but leaves simpler results taking considerably longer to calculate. For the given tests, RRT* and PRM both ran 10 seconds, the maximum amount of time given to each simulation.

RRT took the least amount of time in the Abstract, Apartment and Home simulations. A best guess as to why is due to the semi-randomness of the areas which play into the random aspect of the RRT itself; applying one random to another have better results than applying order to random or random to order; RRT took the longest of the three algorithms in the Cubicles experiment, at 4.12 seconds.

KPIECE1 took the longest times in the Apartment and Home simulations, lending to the possibility that the dispersed randomness could affect the algorithm. This conclusion is easier to draw as KPIECE1 took the shortest time in the Cubicles and Easy simulations, a set of orderly designs that lend themselves to the concept of forward, orderly motion. A counter to this idea rests in the Apartment-Hard simulation, where KPIECE took the shortest time (0.22 seconds.)

The last of the tested algorithms is the EST method, which was unable to solve any simulation in the shortest time. This algorithm took the longest to decipher a path in the Abstract scenario, taking 1.64 seconds (over twice the second best, KPIECE1 at 0.63 seconds), and the Apartment- Hard scenario, taking 0.63 seconds (again, over twice the second best, RRT at 0.26 seconds). EST appears to be a strong middle-ground between RRT and KPIECE1 with some exceptions, but still provides a solution in a respectable amount of time compared to the other algorithms tested.

B. Correct Solution %

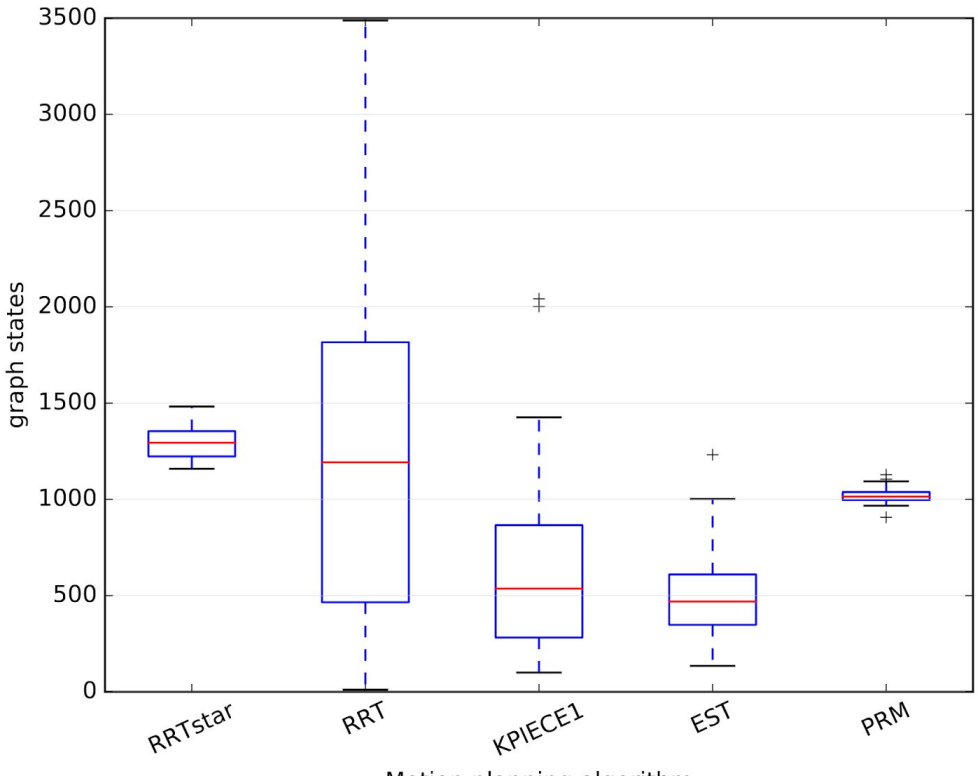
According to the given graphs of each simulation, every algorithm succeeded in acquiring a solution. It is not surprising as all of the tested algorithms are designed specifically for finding paths through areas given certain constraints. As our simulations are not immensely complex and designed for the testing of the algorithms, all five algorithms (RRT*, RRT, KPIECE1, EST, and PRM) were successful 100% in finding correct solutions.

C. Graph States

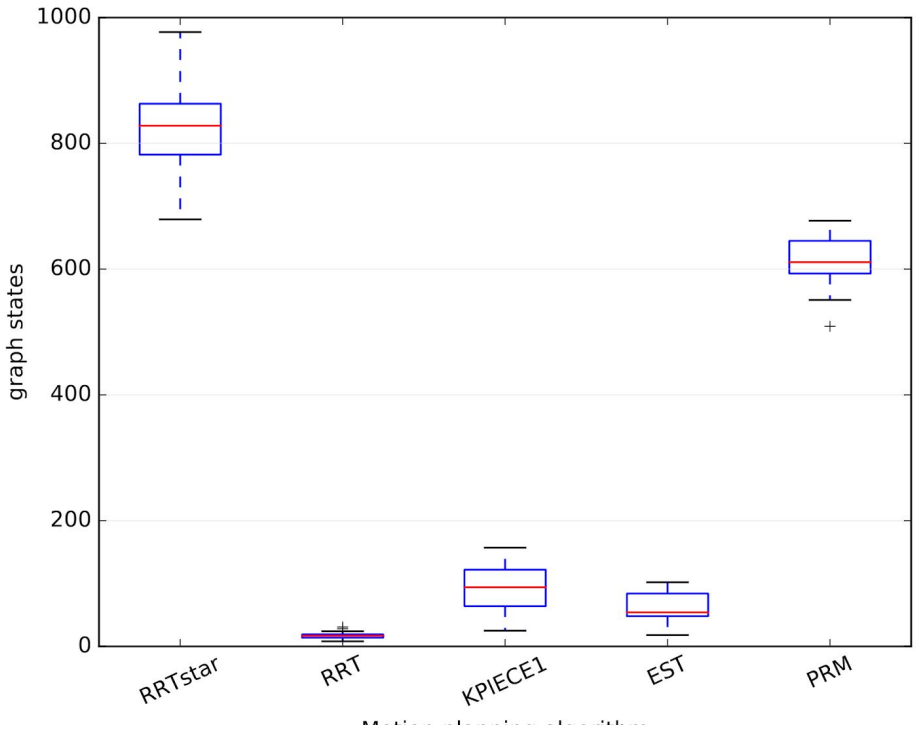
Graph states can be defined as the number of nodes, or locations sampled, by an algorithm and are recorded here to gather a sense of the amount of data processed. As would be expected RRT* and PRM create the highest number of states; this does not include the Cubicles run as RRT creates just under 3x as many states as the PRM algorithm.

Excluding the Cubicles run, RRT KPIECE1 and EST all have lower graph states that complete the simulation. This is most likely due to their design of terminating their operations once a solution has been found and mapped. This is reflected in the previous category of time taken to find a solution and supported by their internal concepts.

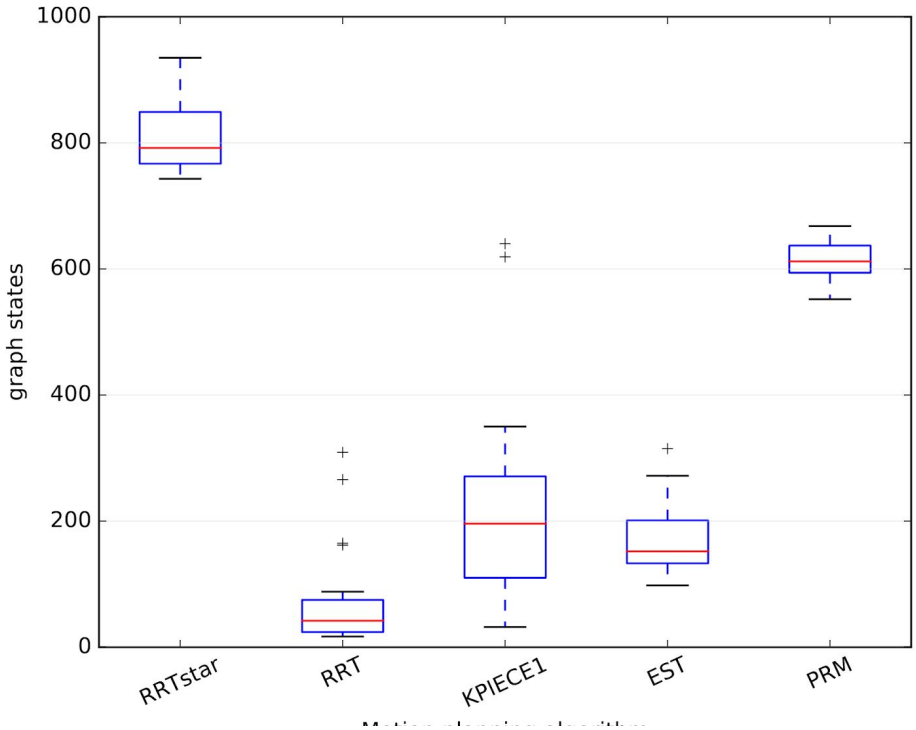
While RRT* and PRM do hold the most graph states of the five algorithms tested, they also have less states than expected for their run times when compared to the others. This most likely stems from the way their data is managed and the operations they take to create more efficient pathways along known obstacles.



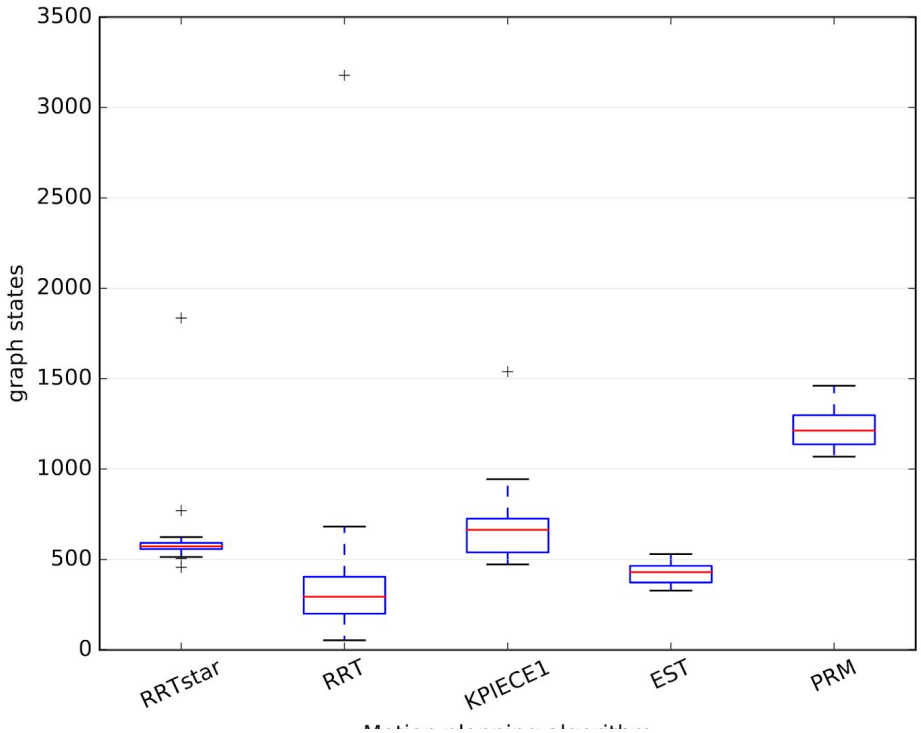
Graph States of the Abstract run.



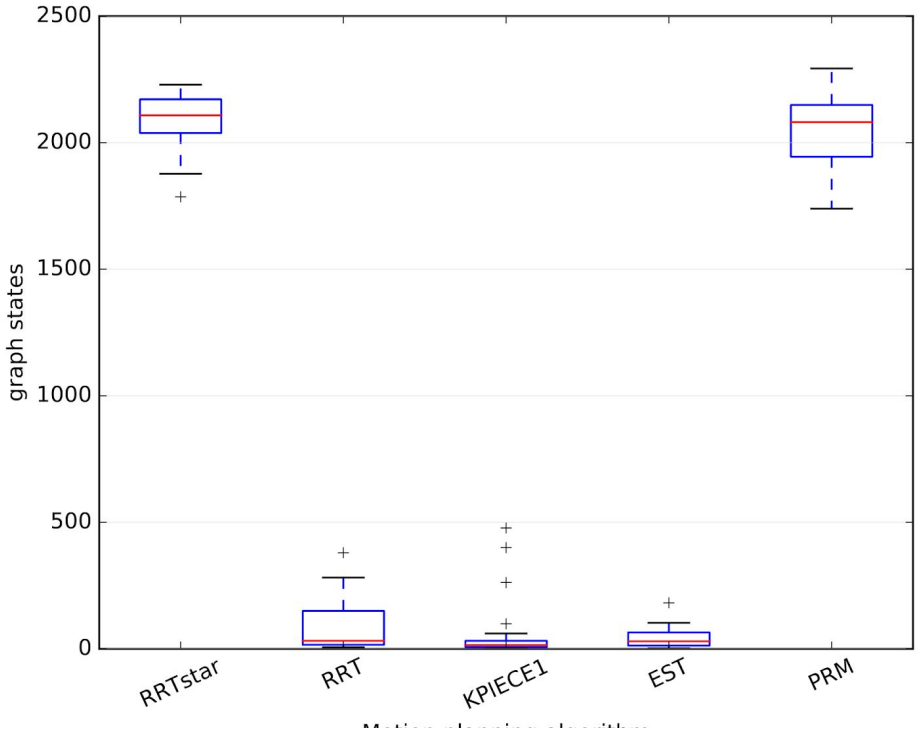
Graph States of the Apartment run.



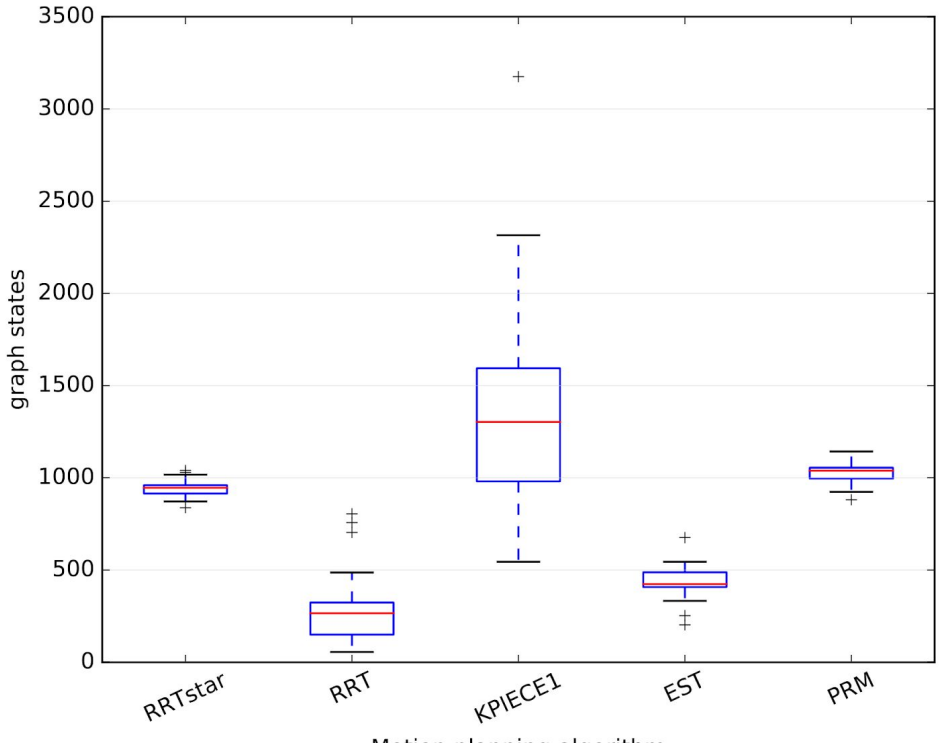
Graph States of the Apartment Hard run.



Graph States of the Cubicle run.



Graph States of Easy run.

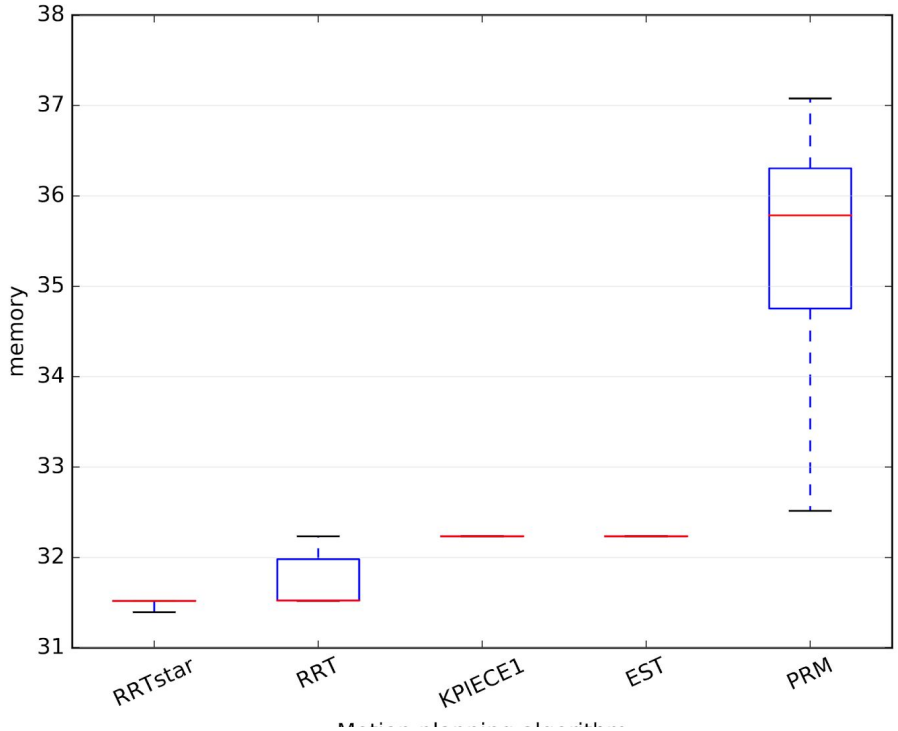


Graph States of the Home run.

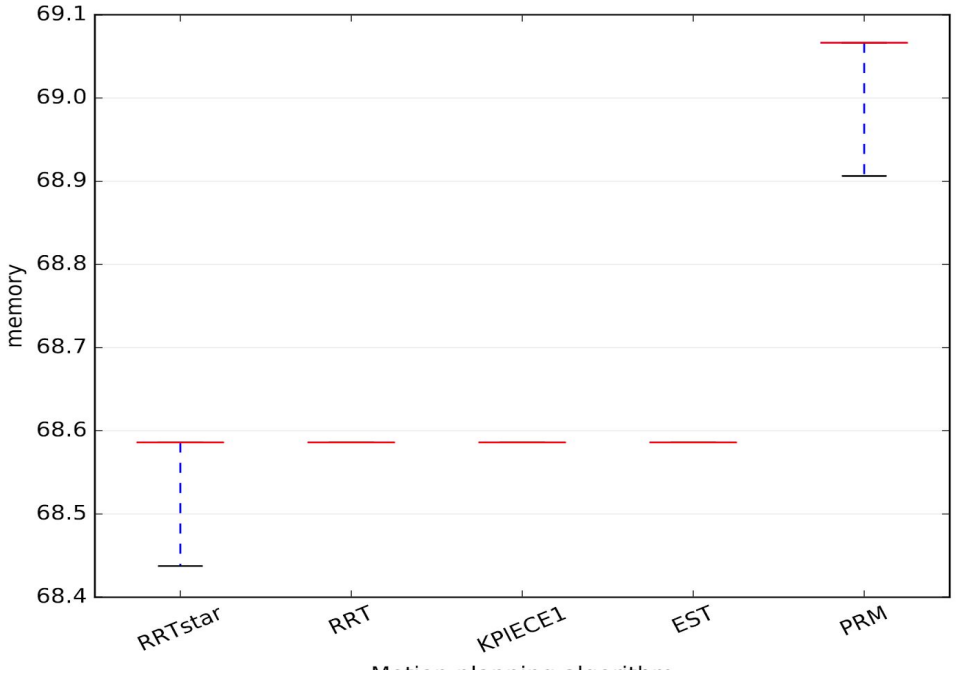
D. Memory

When looking at the memory usage of these algorithms, it's important to remember how they are written to complete their tasks. While a quick solution seems to be the most efficient in memory, it is the one that runs the longest with the same memory requirement that is considered truly efficient. To this point, the final results of the runs with our algorithms have come with an interesting conclusion: RRT*, RRT, KPIECE1 and EST use a very similar, if not same, amount of memory for every given simulation; PRM uses up to 20% more memory for the same runs.

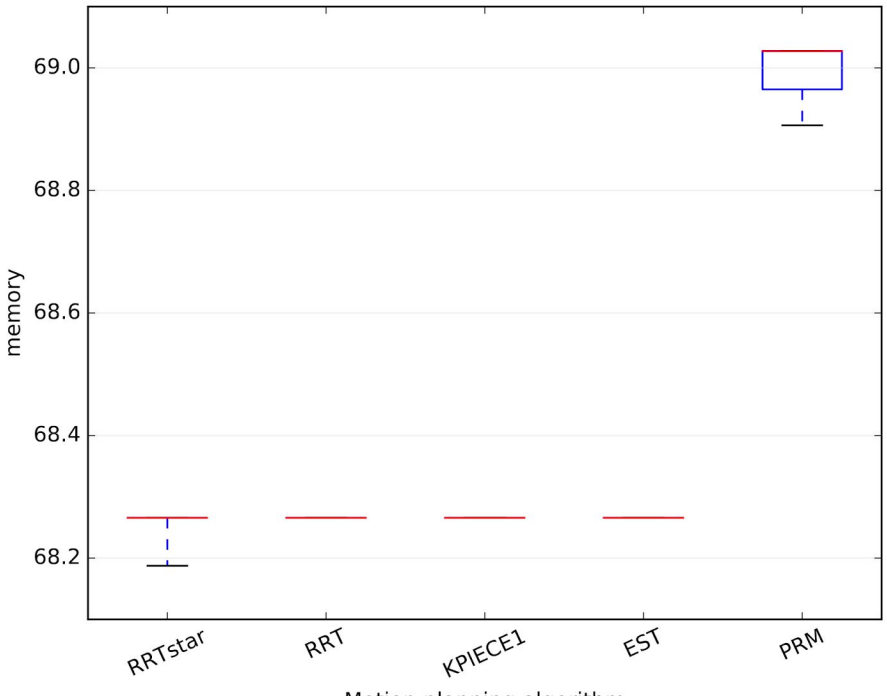
This result isn't entirely unexpected, but it does show that while RRT* may hold the record for most graph states in the given runs it is also extremely efficient in its memory usage. This is an excellent comparison to the PRM which works in a similar way (high graph states, long run time) and is the largest drain of memory out of the given algorithms, including for the easy simulation.



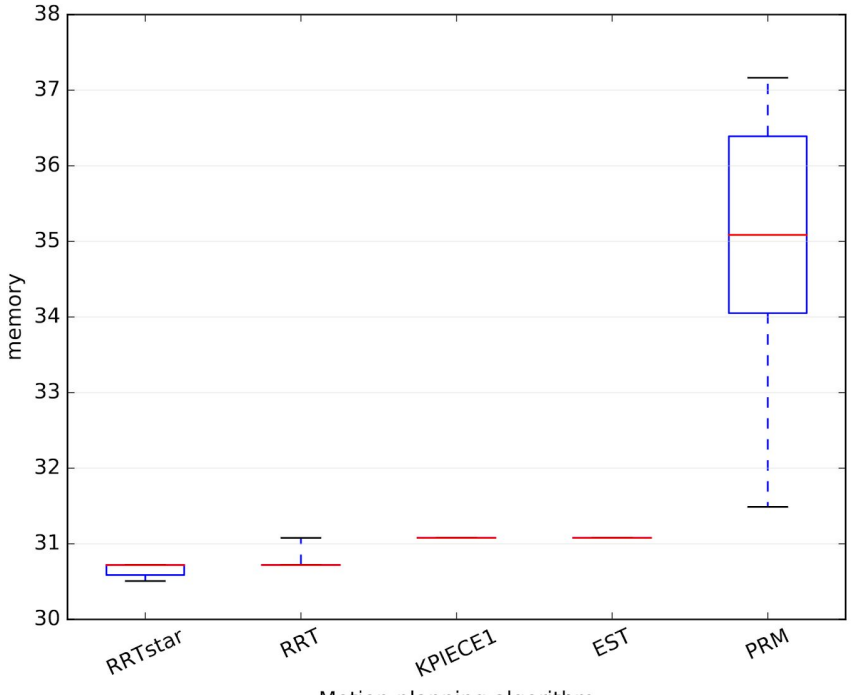
Memory graph for the Abstract run.
Memory is in Megabytes.



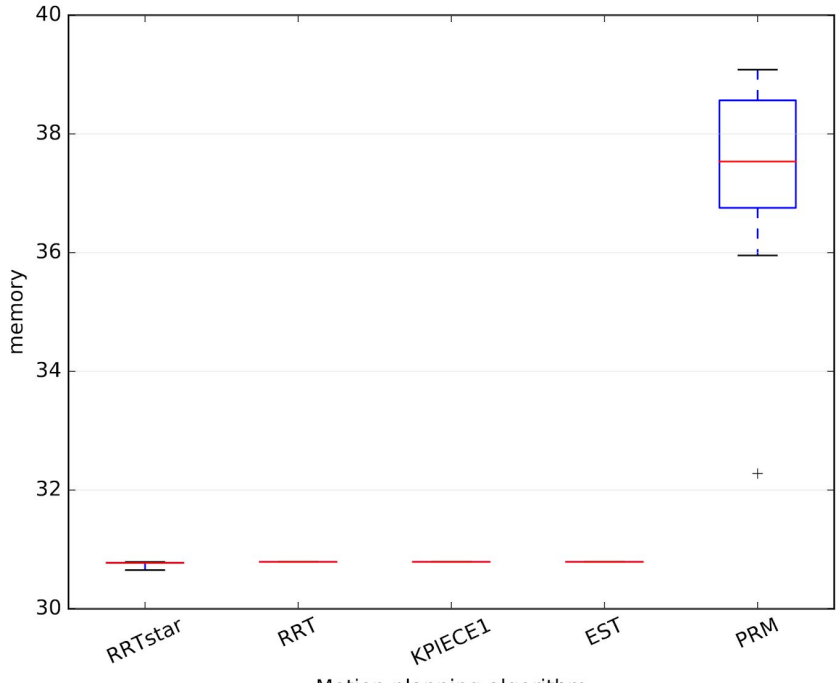
Memory graph for the Apartment run.
Memory is in Megabytes.



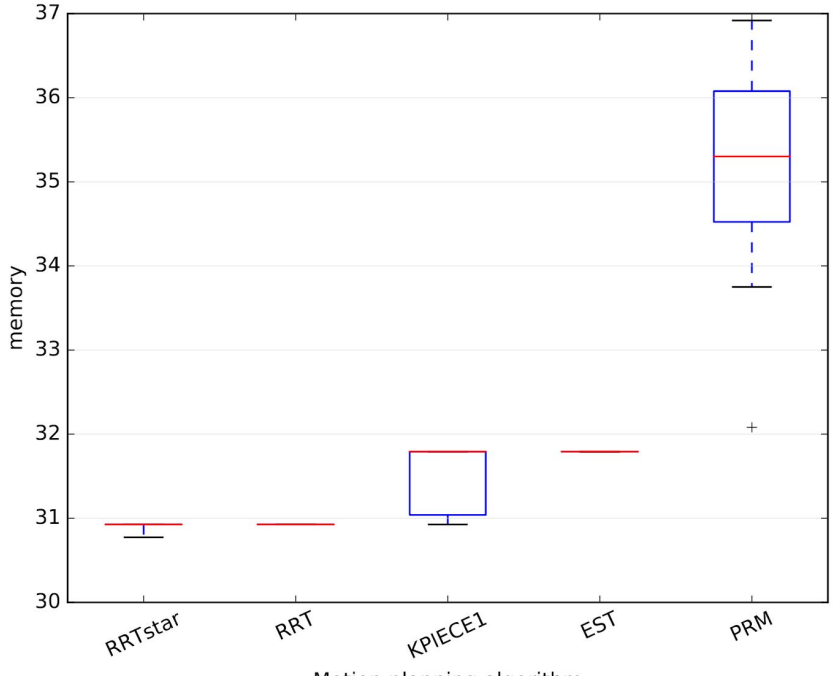
Memory graph for the Apartment Hard run.
Memory is in Megabytes.



Memory graph for the Cubicles run.
Memory is in Megabytes.



Memory graph for the Easy run.
Memory is in Megabytes.



Memory graph for the Home run.
Memory is in Megabytes.

V. Conclusion and Link to GitHub

https://github.com/austinbachman/CS419PathPlanningGroup

This project programmed, compared, and analyzed several essential path planning algorithms through the use of OMPL. Virtual environments were created and the path planning algorithms were run on them. Each algorithm was analyzed by its trajectory, time to find solution, correctness of solution, number of graph states, and memory use. We also included a 2D implementation of RRT and RRT* with Object Avoidance. Explanation on running the code can be found in the readme file in the github.

In conclusion, RRT* was the most efficient in the given time for the application of aerial quadrotor robots. However, more testing would be required to determine if this result would be the same given a shorter time to find the solutions, since RRT* uses the whole time allotted to optimize the solution. Given that the time allotted was only 10 seconds, however, most aerial robotic applications that do not require maximum speed and have modern processors should be able to use this algorithm to its full potential.