Motion Planning

1st Marcus Schaller
Department of Electrical and Computer Engineering (ECE 276B)
University of California San Diego
La Jolla, U.S.A
mschalle@ucsd.edu

***Abstract*—Motion planning is crucial in robotics as it is a tool that is used to allow a robot to calculate the ideal path through an environment. In order to implement motion planning one must first select from a variety of methods that can be used to successfully solve such problems. This paper will discuss two common techniques that are often used in motion planning, search-based vs sampling based planning.**

I. Introduction

Motion planning is a very broad subject and therefore implementing an effective motion planning algorithm can be done in a variety of ways. Two popular methods include search based planning and sampling based planning. Search based planning involves systematically searching a graph for a goal and given that such goal exists is guaranteed to return a path to the goal. Sampling based planning generates a sparse graph and as iterations approach infinite the probability of reaching the goal approaches one. Sampling based planning is much better suited for planning a path in higher dimensions than search based. However, since their running time is not entirely dependent on the dimension of the space they are planning in, sampling based may be less effective than search based planning in lower dimension spaces. This paper will discuss the implementation of each of these two motion planning techniques in a variety of environments each with varying difficulties.

II. Problem Formulation

In order to implement a motion planning algorithm one must first formulate the deterministic shortest path problem at hand. The following section will discuss this.

A. Problem Environment

Their are a total of seven different environments used in this project. Each of these environments can be described in one array describing the boundary space, i.e. the total size of the map, and a number of arrays describing axis-aligned bounding boxes (AABB) which function as obstacles in the environment. Additionally a start point and goal point are given. Using this information the world can be generated as seen in figure 1.

B. Deterministic Shortest Path

The method used to navigate to the end position can be described as a deterministic shortest path problem (DSP). A DSP problem can be described by a graph with a finite vertex set V where each vertex is a node $i \in V$ and the distance from node i to node j has cost c_{ij}. The starting node is node s and the final node is node τ. The objective of a DSP problem is to minimize the sum of the edge costs equal to J along a path from s to τ, $J^{i_{1:q}} = \sum_{k=1}^{q-1} c_{i_k, i_{k+1}}$. The optimal path is the one with the minimal cost or length J, $i^*_{1:q} = \underset{i_{1:q} \in \mathcal{P}_{s,\tau}}{\arg\min} J^{i_{i:q}}$. In order to calculate the shortest path the configuration space C is used within which free space, C_{free} and obstacles, C_{obs} are contained. The optimal path J^* should go from s to τ without intersecting C_{obs}.

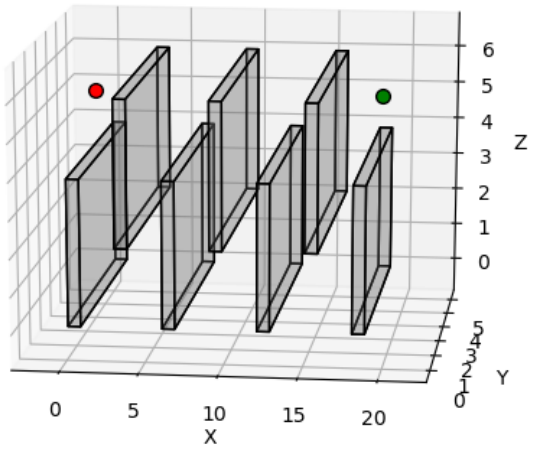

Fig. 1. Flappy Bird Environment

III. Technical Approach

This project contains multiple parts which will be individually discussed within this section.

A. Collision Detection

When efficiently implementing motion planning it is crucial that one's algorithm can accurately determine if a path intersects C_obs. In this project a custom collision detection algorithm was made to do this. The algorithm accepts two connected nodes from a given path and segments the space between the nodes into sub-nodes returning the x, y, and z coordinates of each sub-node. It then checks if the x, y, and z coordinates are between the max and min values of each of the AABB obstacles. If it is this will return true meaning that it does collide with an obstacle. The algorithm also has an error

value which when increased inflates the sizes of the obstacles in the collision detection algorithm. This would allow for larger than point size robots to successfully navigate through the obstacles.

B. Search-Based Algorithm

In this project the Dijkstra algorithm as well as the A^* algorithm was used to find the shortest path.

1) Configuration Space Creation: In order to implement a search-based algorithm, the provided boundary and bounding boxes information has to first be translated into a usable configuration space datatype. In order to do this some assumptions had to be made. First, each node is a minimum of 0.5 units from the another node. In other words, the configuration space can be represented as a lattice with each node being 0.5 units away from each other in one dimension, 0.707 in two dimensions, and 0.866 in three dimensions. C_{ij} is equal to each of these values. Second, elements that are in C_{obs} are not included in the final tree map so the final path can not access these nodes. These elements include all nodes outside the boundary and nodes within the bounding boxes. In order to create the map a python dictionary is used. Each node can have a maximum of 26 different neighbors. As the map is being made it samples through each node. The path between the parent node and it's neighboring node is tested with the collision detection algorithm to see if an obstacle prevents nodes from being neighbors. If the path is free it is added to the tree-map dict. An example of a single node can be seen in figure 2.

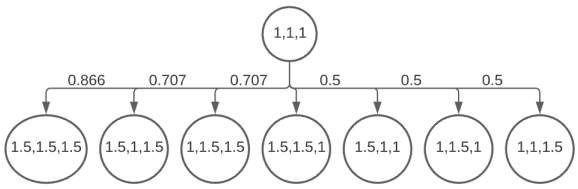

Fig. 2. Configuration Space Tree Map

Once every node is checked the tree map is correct and the shortest path algorithm is ready to be ran.

As this algorithm takes $O(n^3)$ and is not time efficient when ran the tree map is saved to an external file in order to be used in the future. While this is not memory efficient when it works well for a limited deterministic environment like the ones provided in this projects. Should the project include a much larger environment a different method would need to be used to allow for scaling.

2) Shortest Path Algorithms: Two shortest path algorithms were used in this project, Dijkstra and A^*. In order to implement this algorithm, the the Python priority queue dictionary library was used (pqdict). This library is ideal for setting up a priority queue as updating score and removing elements only costs $O(log(n))$ time. Using this library the Dijkstra algorithm was very easy to implement. To start every node in the priority queue is initialized to be cost infinity except the start node which was initialized to 0. After that, the algorithm will begin to remove the lowest cost node and check cost to reach each of the the nodes neighbors. If the cost C_{ij} to reach node j from node i is less than its previous cost c_{kj} the parent for that node will be updated to be node i. These costs are given from the configuration space tree map that was discussed in the previous section. The new node costs are taken into consideration when the priority queue returns the next lowest cost node. The algorithm continues to remove each of the lowest cost nodes from the queue and calculates cost of its neighbors until it reaches the goal node. At that time the algorithm terminates.

The A^* algorithm is nearly identical to the Dijkstra except the heuristic h_j is introduced. This heuristic is equal to the euclidean distance from the a given node to the goal node. Now when the algorithm calculates the cost to reach its node j from node i the cost of node j becomes $c_j = c_{ij} + h_j$ with $h_j = dist(j, \tau)$

Since this implementation was very efficient and didn't require more than a couple seconds to run a ϵ consistent algorithm was used with $\epsilon = 1$. This was selected as it was determined that optimality would be prioritized over speed. The most memory and time intensive environment was the maze environment which required exploring 24276 nodes before reaching the goal state. One interesting thing found was that the A^* algorithm would not always return a shorter path than the Dijkstra algorithm. This will be further discussed in the results section.

C. Sampling based algorithm

Unfortunately, I was unable to successfully install the OMPL library so I elected to use the Python motion planing library instead.

I selected to use the RRT^* algorithm instead as I found that RRT resulted in a very long and chaotic path. Figures 3 and 4 show the difference in path quality between these two algorithms.

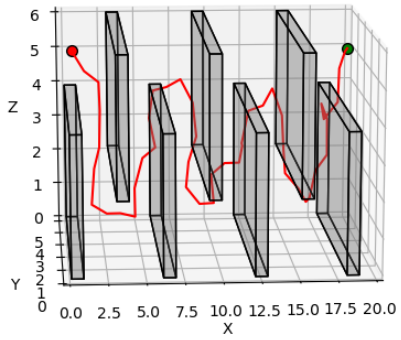

Fig. 3. RRT path

RRT* was slower than the vanilla RRT but it provided a much smoother and shorter path. The RRT algorithm functions by randomly sampling points and if the path to the point is not obstructed by the obstacle that path is added to the tree.

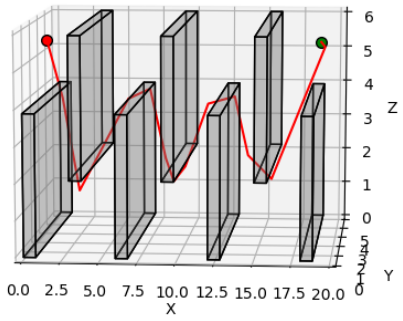

Fig. 4. RRT* path

The RRT* algorithm is slightly different in that given a newly sampled node if a node further in the tree can be connected through it with a shorter path the tree will be re-routed using the new node. This is why the RRT* algorithm provides a much smoother path.

The python motion planning library provides some parameters that had to be tuned. Q is the length of each tree edges after tuning it was found that a value of [1,2] was most optimal. R is the smallest edge to check for obstacle collision. This was set to .09 as the thinnest obstacle was 0.1. Max samples was set to 100000 to allow the complicated maps to be completed. Prc which was the probability of checking for a connection to the goal was set to .01. Finally the rewire count was set to 36 which provides a reasonable number of branches nearby to rewire.

IV. Results

The path and run time information for the A* and RRT* are presented below. Plots for each environment can be found in the appendix.

TABLE I
Dijkstra

Map	Planning_time	Length	Considered Nodes
Single_cube	2.5244	7	17572
Maze	3.4399	79	29231
Flappy_bird	0.4194	25	4972
Monza	0.2809	78	4056
Window	1.1407	27	12406
Tower	0.3164	33	4217
Room	0.1524	12	1806

TABLE II
A*

Map	Planning_time	Length	Considered Nodes
Single_cube	1.284	7	2249
Maze	3.009	81	24276
Flappy_bird	0.563	25	4717
Monza	0.3532	27	3964
Window	1.159	27	11682
Tower	0.38918	33	3966
Room	0.1564	12	1466

TABLE III
RRT*

Map	Planning_time	Length	Considered Nodes
Single_cube	0.19566	8	13-30
Maze	74.46	75	11000 - 12000
Flappy_bird	2.82	27	650-750
Monza	17.87	75	2000-2500
Window	0.915	24	150-200
Tower	9.93	32	950-1200
Room	0.456	12	90-150

The configuration spaces only have to be generated a single time, it is then saved for future use. The time to generate the configuration space for the search based algorithms are seen in table IV

TABLE IV
Configuration Space Generation Time

Map Name	Map Creation time
Single_cube	32.2
Maze	375.5
Flappy_bird	18.7
Monza	6.5
Window	61.8
Tower	38.6
Room	25.7

As it can be seen the search based approaches result in a smoother path but not necessarily a shorter path. Overall the computed path between the two search-based algorithms are consistent except for the maze which resulted in a slightly shorter path. It appears that search-based seemed to outperform sampling-based in complex environments such as the maze in the time domain however not necessarily in the optimal length of the path. One thing to note is the total considered nodes are much greater for the search based algorithm. Between the two search based algorithms the A* considers much fewer nodes. I found that if I scaled down the weighting of the euclidean distance heuristic function than the number of considered nodes would increase, also increasing the run-time. This is clear in open environments such as the single block environment which considered a much larger number of nodes using Dijkstra then it did with A*. Table 5 shows how changing the weighting of the heuristic can vastly change the number of considered nodes.

TABLE V

Weighting	Considered Nodes	Run time
1	2249	1.284
0.75	2465	1.312
0.5	2814	1.433
0.25	3751	1.495
0	17572	2.524

1) Future improvements: The first improvement would be to further rewire the RRT* path to create a much smoother path. This may have been possible if the OMPL library was successfully installed. Next would be to improve the

configuration space generation as it is not feasible to scale the current approach up for large high accuracy environments.

ACKNOWLEDGMENT

Thank you Professor Atanasov as well as Thai. I appreciate all the effort you put into making this project. I learned a lot and had a great time solving this problem.

REFERENCES

[1] N. Atanasov, "Learning in Robotics Lecture 6: Configuration Space" in ECE276B, 15-Apr-2021.
[2] N. Atanasov, "Lecture 7: Search-based Motion Planning" in ECE276B: Planning amp; Learning in Robotics, 20-Apr-2021.
[3] N. Atanasov, "Lecture 9: Sampling-based Motion Planning" in ECE276B: Planning amp; Learning in Robotics, 20-Apr-2021.

APPENDIX

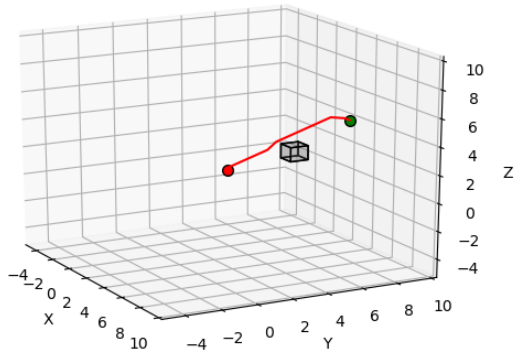

Fig. 5. Single Block A*

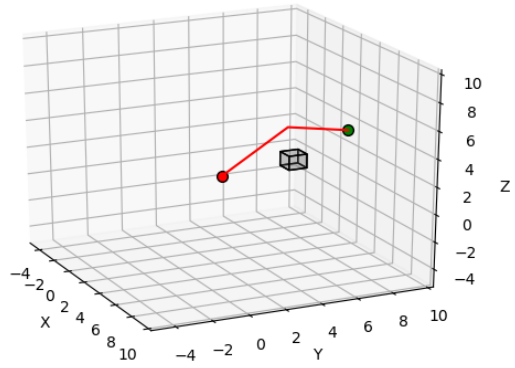

Fig. 6. Single Block RRT*

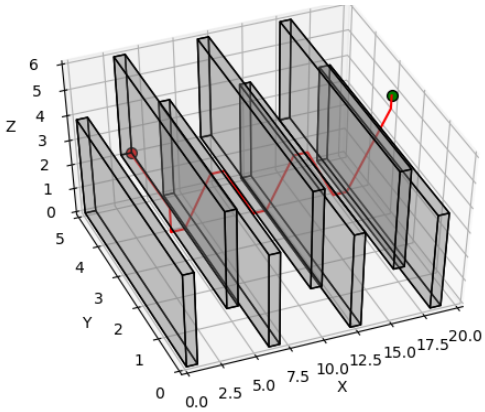

Fig. 7. Flappy Bird A*

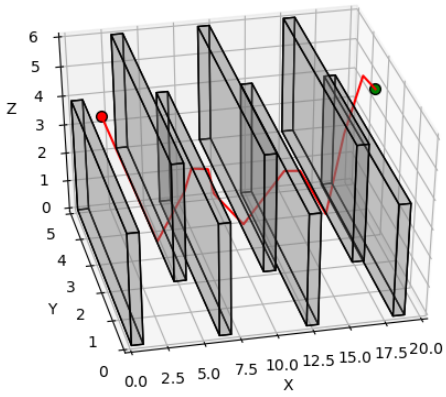

Fig. 8. Flappy Bird RRT*

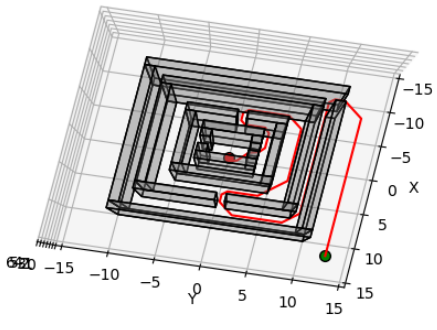

Fig. 9. Maze A*

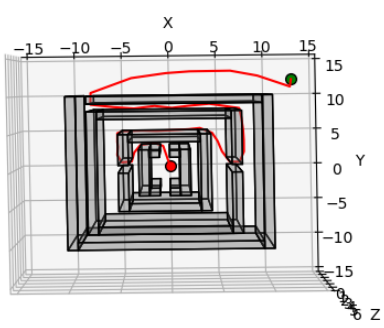

Fig. 10. Maze RRT*

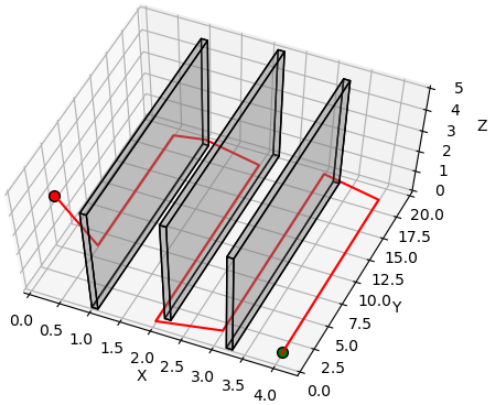

Fig. 11. Monza A*

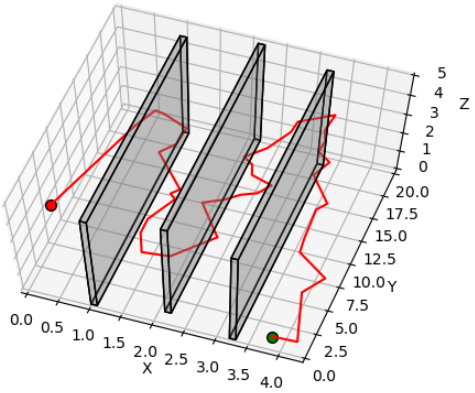

Fig. 12. Monza RRT*

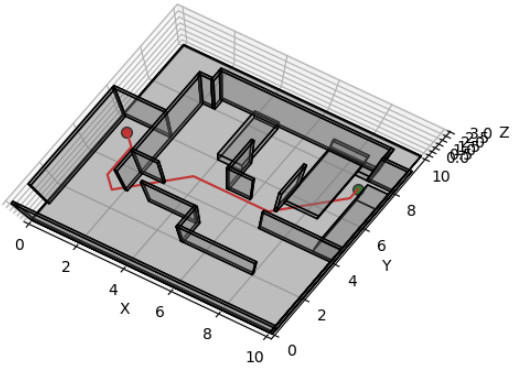

Fig. 13. Room A*

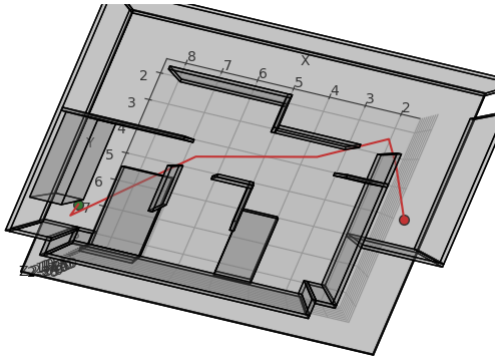

Fig. 14. Room RRT*

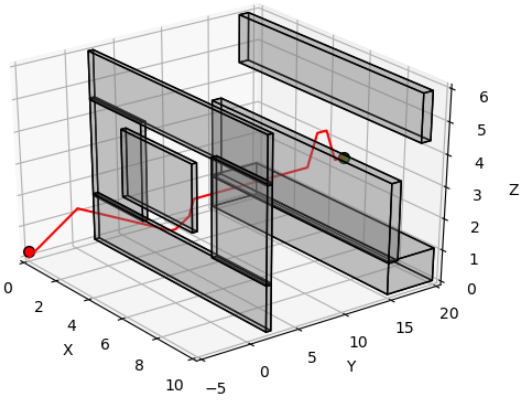

Fig. 17. Window A*

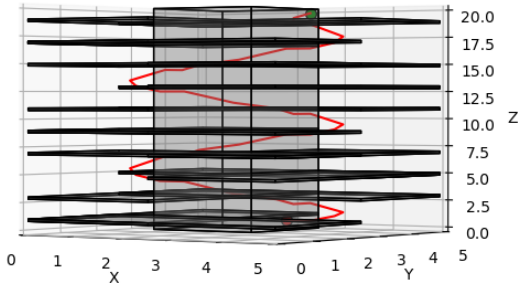

Fig. 15. Tower A*

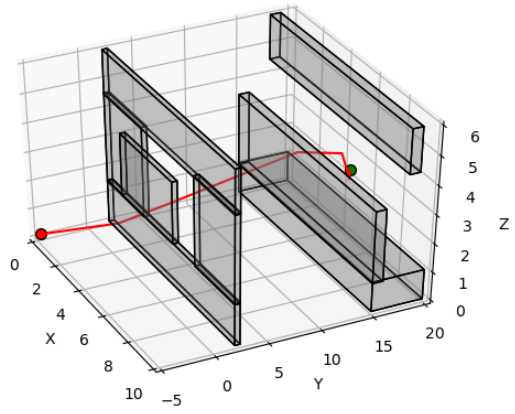

Fig. 18. Window RRT*

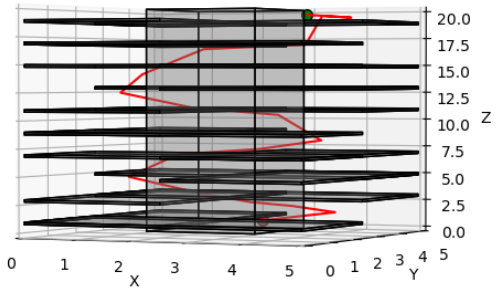

Fig. 16. Tower RRT*