Introduction

Given any 3D environment and start point, This problem asks us to come up two algorithms (one is A^* based, one is RRT based) to tell the robots where to move in order to reach the goal point. These two algorithms needs to satisfy the following three conditions:

1. The algorithms need to produce a move every two seconds.
2. The moves should not lead the robot to collide any obstacles or boundaries.
3. The algorithm will lead the robot to the goal at the end.
   My approach is: for the A^* based algorithm, I decide to use $LRTA^*$. For RRT based algorithm, I decide to let the robot stay untill the RRT graph and path are contructed, and then let it move along the shortest path computed.

Problem Formulation

For $LRTA^*$, the elements are: an array to store all the explored nodes, and an array to store the heuristics of the expolored nodes. For fast run time and keeping the runtime within the 2-seconds time constraints, I decide to keep the expand node N = 1. And also, to ensure fast runtime for each step, I only consider 6 directions, which are left, right, forward, backward, up, and down. Also, I treat the environement as 3D discrete space which origin (0,0,0) is that start point. I also discrete the 3D space with gap ϵ, which also the stepsize for $LRTA^*$. The default ϵ is equal to 0.5. I define my heuristic function to be $h(i) = ||x_i - goal||_2$, which is the Euclidean distance, because the Euclidean distance = 0 when $x_i = goal$, and also is the shortest path distance possible for any point. The Euclidean distance heuristic is consistent.

For RRT based algorithm, the elements are: variables to keep track of the graph construction and path construction, a graph that stores nodes and edges. For RRT construction, I just use the regular RRT with steered distance ϵ, which is also the stepsize. Each node stored in the graph contains three elements: the cooridnate of the node, the cost of the node, and the parent of the node. And since RRT is a tree graph, I compute out the cost and parent during the graph construction.

Technical Approach

My approach:
For $LRTA^*$, there is no need to initialize the whole discrete graph at first. You only need to know your neighbors nodes and the explored nodes. For this algorithms, I define three helper functions: one to check if the node is in free space, one to find the neighbors, one to check if the move is collision free. And the following is a simpiled pseudocode:
Given currpos, goal:

1. find neighbors of currpos.
2. Remove the neighbors that would result collision.
3. Find or calculate the heuristic of the neighbors.
4. Update the heuristic of the neighbors and the currpos.
5. Move to the neighbor with lowest heuristic. (Since the cost are the same, no need to check $c_{ij} + h_j$) And also, always, check if it is possible to reach to the goal from current position. If yes, just return the goal point.

For RRT based algorithm, I need four helper function: one to produce the random point from free space, one to find the nearest node in RRT graph to the random point, one to find a steered point, one to check if collision free. Here is a simpiled pseudocode:

1. Add start node to RRT graph with cost = 0.
2. while (the last_added node cannot reach goal in one step) : Add a steered point to the RRT graph, and set the parent of the steered point to its nearest point. Set the cost of the steered point = the distance from steered point to its parent + the cost of its parent. Then, let the last_added node = newly addded steered point.
3. Add goal node to the RRT. Set goal node's parent = the last_added node.
4. For path, start from the goal node, trace its parent recursively untill it reaches the start point.
5. Move following the path.

Key Ideas that enable your planner to scale to large maps and to compute solutions within the allotted time constraints:
For $LRTA^*$, since I decide to expand only once, i.e., only check the neighbors, the runtime would be very fast. The only problem is that, it can take a little long to check the heuristic of nodes in explored nodes list. But, if you have good hash table, (use dictionary in python), the runtime will be O(1). And since all other operations are also O(1), the total runtime of $LRTA^*$ (N=1) theoretically should be O(1). Thus, no need to worry about the time constraint or large maps.
For RRT, the main idea is to let the robot stay untill the graph and path are computed. The general steps to keep time within: 1. If the graph not finished, keep adding steered nodes until the time is reach. 2.If the graph is finished but path is not, keep add the parents recursively until the time is reached. 3. If path finished, move following the path.

Computational Complexity:
For $LRTA^*$ with expand N = 1, theorically, the runtime should be O(1), where n is the number of explored nodes. It takes O(1) to find neighbors, O(1) to compute and update the heuristic of the neighbors, and O(1) to find min heuristic of the neighbors, and O(1) to check the heuristic in explored list if use hash table. However,

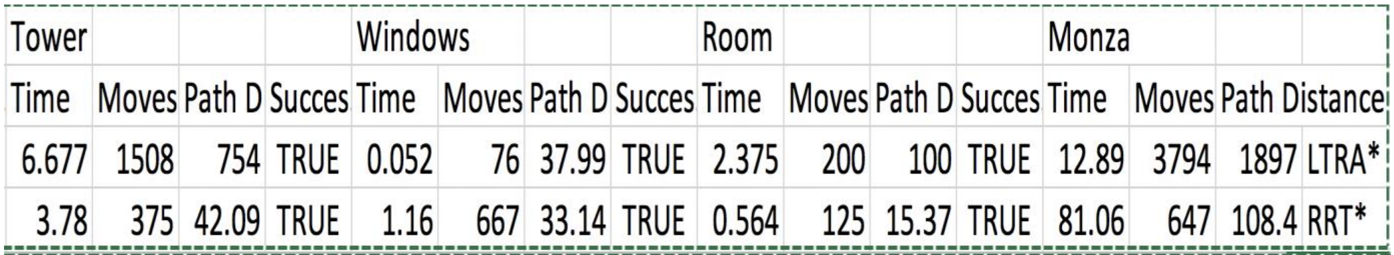
in this case, I just use brute force to search explored list, thus, the runtime would be O(n) <= O(N), where n = number of explored nodes, and N = number of all nodes in the graph.
For RRT based algorithm, the time complexity for find and add a steered node is O(n), where n is the number of nodes in RRT graph. And O(n) < O(N), where N is all possible nodes in discrete 3D graph with ϵ gap.

completeness and optimality:
For $LRTA^*$, it guaranteed to reach the goal, since all the edge is bounded, the graph is finite, all actions are reversible, and the heuristic function (L2) is consistent. $LRTA^*$ (N=1) is just very similar to limited-horizon A* because it makes a move towards the node j in OPEN/Explored with smallest $gj + hj = c_{sj} + h_j$ value. And it might need more steps than LTRA with N>1, however, it is faster for each iteration.
For RRT based, it will find the admissible path, since as the RRT expand the whole graph, you are basically dealing with the whole 3D graph. It is hard to say how fast would RRT find the admissible path due to its randomness, but according to the results, the path RRT found usually is better than LTRA.

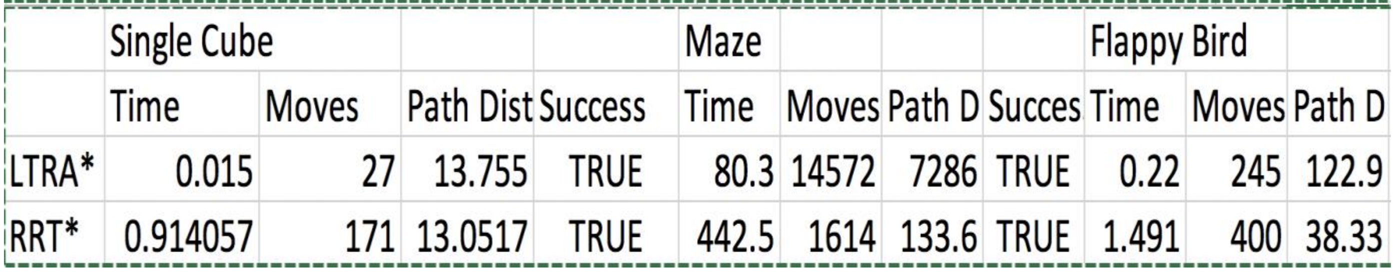
Results

RRT and $LTRA^*$ both runs successfully in all provided maps. There is a table of performance below. Generally, RRT runs slower than LTRA. The reasons why are 1. In RRT, I used graph package which might be slower than numpy. 2. The randomness cause RRT graph explore useless area. However, the path that RRT based algorithm gives much less distance than LTRA, since technically, RRT is not anytime algorithm like LTRA, and in RRT algorithm, you can stay untill the lowest cost value path is computed. One thing to notice is that both RRT based algorithm and LRTA performs worse on the environment maze and monza due to their complex and almost enclosed obstacles.

TABLE of Performance

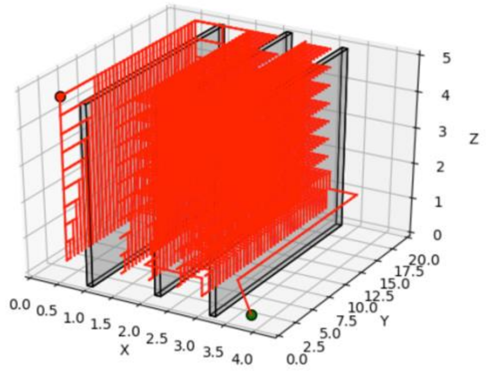


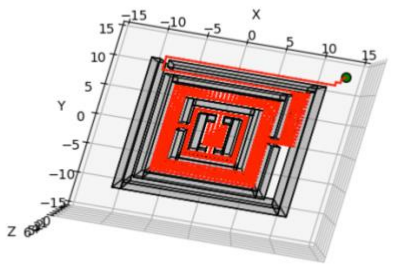
Tower				Windows				Room				Monza			
Time	Moves	Path D	Succes	Time	Moves	Path D	Succes	Time	Moves	Path D	Succes	Time	Moves	Path Distance	
6.677	1508	754	TRUE	0.052	76	37.99	TRUE	2.375	200	100	TRUE	12.89	3794	1897	LTRA*
3.78	375	42.09	TRUE	1.16	667	33.14	TRUE	0.564	125	15.37	TRUE	81.06	647	108.4	RRT*

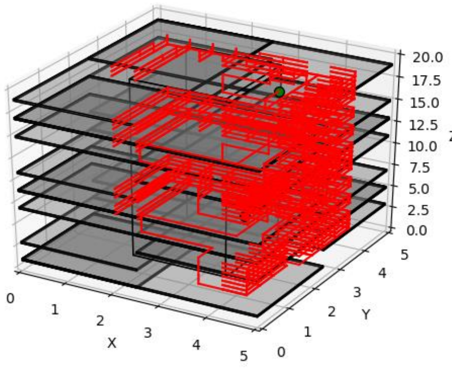


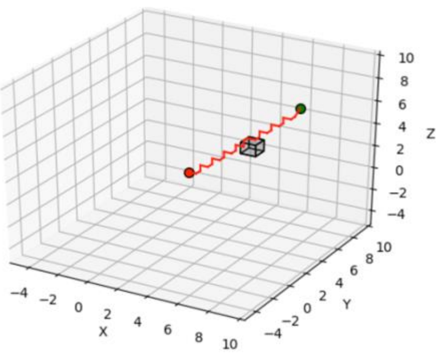
	Single Cube				Maze				Flappy Bird		
	Time	Moves	Path Dist	Success	Time	Moves	Path D	Succes	Time	Moves	Path D
LTRA*	0.015	27	13.755	TRUE	80.3	14572	7286	TRUE	0.22	245	122.9
RRT*	0.914057	171	13.0517	TRUE	442.5	1614	133.6	TRUE	1.491	400	38.33

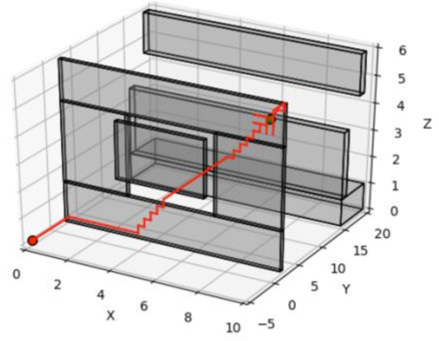
LTRA* PATHS:

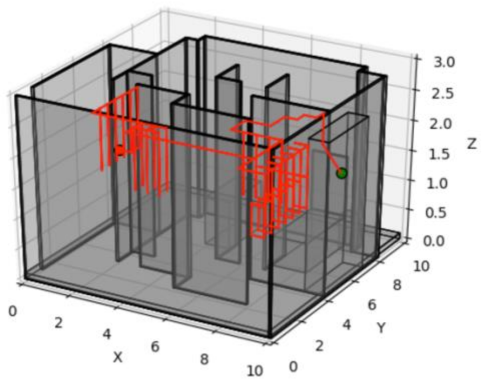


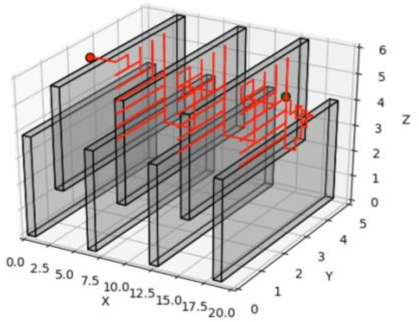




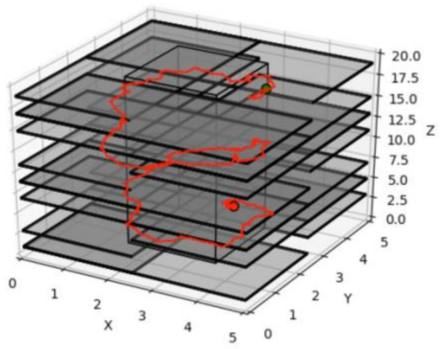



X
Y
Z
0
2
4
6
8
10
−5
0
5
10
15
20
0
1
2
3
4
5
6

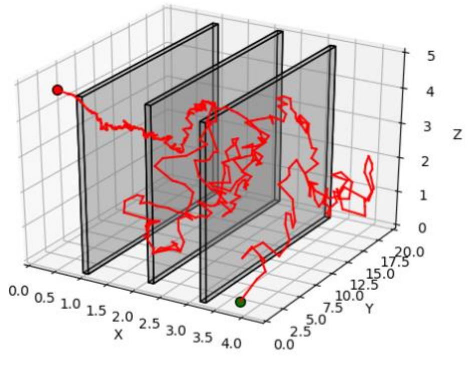

X
Y
Z
0
2
4
6
8
10
0
2
4
6
8
10
0.0
0.5
1.0
1.5
2.0
2.5
3.0


X
Y
Z
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0
1
2
3
4
5
0
1
2
3
4
5
6

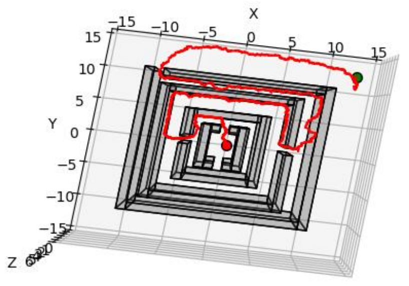
RRT Based Algorithm PATHS:



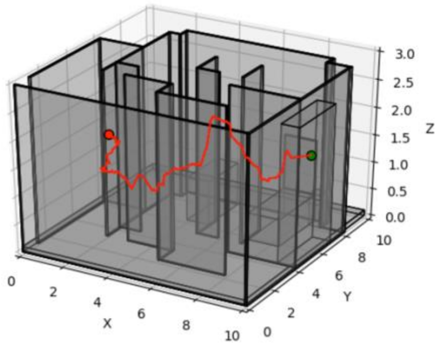
X
0 1 2 3 4 5
Y
0 1 2 3 4 5
Z
0.0 2.5 5.0 7.5 10.0 12.5 15.0 17.5 20.0



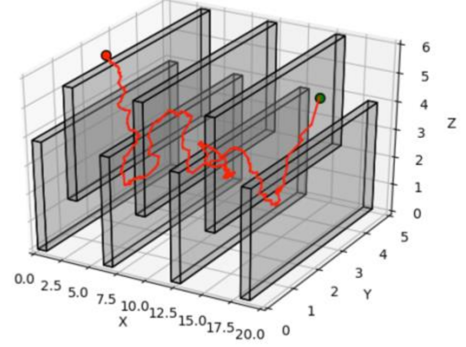
X
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0
Y
0.0 2.5 5.0 7.5 10.0 12.5 15.0 17.5 20.0
Z
0 1 2 3 4 5



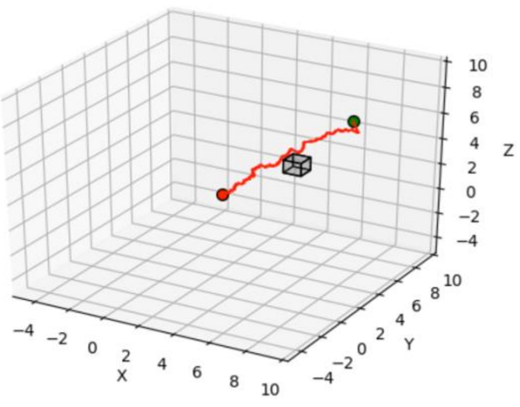
X
−15 −10 −5 0 5 10 15
Y
15 10 5 0 −5 −10 −15
Z



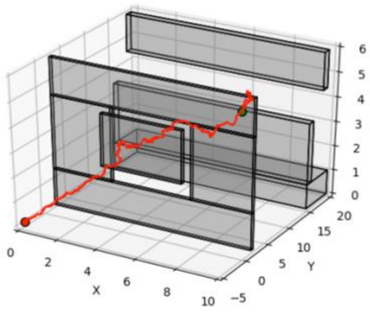
X
0 2 4 6 8 10
Y
0 2 4 6 8 10
Z
0.0 0.5 1.0 1.5 2.0 2.5 3.0



X
0.0 2.5 5.0 7.5 10.0 12.5 15.0 17.5 20.0
Y
0 1 2 3 4 5
Z
0 1 2 3 4 5 6



X
−4 −2 0 2 4 6 8 10
Y
−4 −2 0 2 4 6 8 10
Z
−4 −2 0 2 4 6 8 10



X
0 2 4 6 8 10
Y
−5 0 5 10 15 20
Z
0 1 2 3 4 5 6