Homework2 ECE276B - Project Report

Yihan Hu

2019/05/16

1 Introduction

In this problem, we will implement and compare the performance of the A^* and RRT planning algorithms under strict timing constraints. You are provided with a set of 3-D environments described by a rectangular outer boundary and a set of rectangular obstacle blocks. Besides, the planner should satisfied the following constraint:

$$\begin{array}{l} \bullet \text{ the robot moves a small distance at each time step, i.e., } \|x_{t+1} - x_t\| \leq 1 \\ \bullet \text{ the robot remains collision-free at } x_{t+1} \\ \bullet \text{ the next position } x_{t+1} \text{ is produced within 2 seconds,} \\ \bullet \text{ the robot eventually reaches the goal.} \end{array} \tag{1}$$

Here I proposed several methods that satisfied above constraint. For search based planning, I implement $RRTA*$ and visibility graph methods. For sampling based planning, I implement RRT, Bi-RRT and n-directional RRT methods with path smoothing technique.

In all, search based methods are pretty good for small graph like window and flappy bird, and could return highly optimized path after relatively long run. Somehow, when map becomes larger, search based methods requires too much memory and might be slow. For sampling based methods, though it might not return a optimized path, the computation time is incredibly fast, except for bug trap with narrow path(e.g. maze, monza). I designed n-RRT methods which randomly growing n-RRT simultaneously and connect with each other, which perfectly handle the problem, and the path can be pretty good after smoothing.

2 Problem Formulation

The problem can be formulated as Deterministic Shortest Path (DSP) Problem by discretizing our configuration space. In this case, our search space becomes goal node(s), start node(s) and lots of interval nodes. Here we define:

• Path: an ordered list $Q := (i_1, i_2, \ldots, i_q)$ of nodes $i_k \in \mathcal{V}$

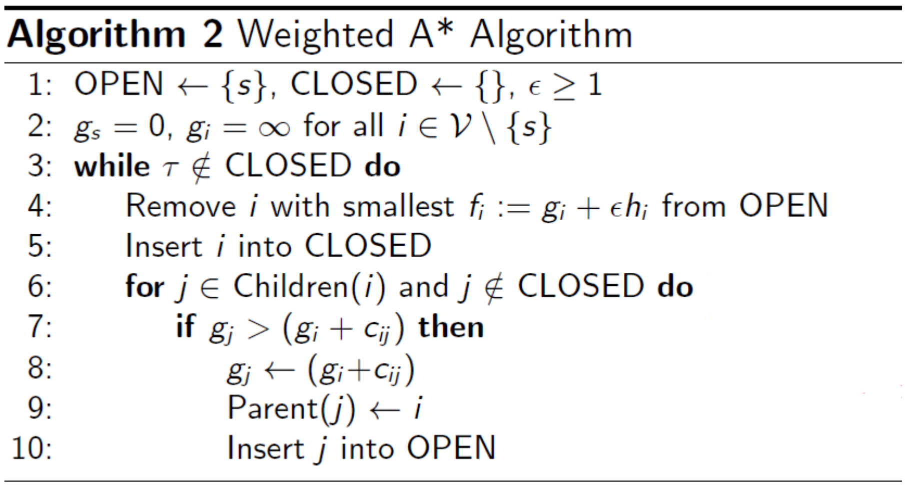
• Set of all paths from $s \in \mathcal{V}$ to $\tau \in \mathcal{V}$: $\mathbb{Q}_{s,\tau}$

• Path Length: sum of the arc lengths over the path: $J^Q = \sum_{t=1}^{q-1} c_{t,t+1}$

• Objective: find a path $Q^* = \underset{Q \in \mathbb{Q}_{s,\tau}}{\arg\min} J^Q$, from node $s \in \mathcal{V}$ to node $\tau \in \mathcal{V}$

•Assumption: For all $i \in \mathcal{V}$ and for all $Q \in \mathbb{Q}_{i,i}$, $J^Q \geq 0$, i.e., there are no negative cycles in the graph and $c_{i,i} = 0$ for all $i \in \mathcal{X}$

Which can be easily convert to Deterministic Finite State (DFS) Problem, by using forward DP to the problem. We can obtain Dijkstra's algorithm. Moreover, when heuristic function is introduced, we can rewrite Dijkstra's algorithm as A^* as follows:



Algorithm 2 Weighted A* Algorithm

```
1:  OPEN ← {s}, CLOSED ← {}, $\epsilon \geq 1$
2:  $g_s = 0$, $g_i = \infty$ for all $i \in \mathcal{V} \setminus \{s\}$
3:  while $\tau \notin$ CLOSED do
4:      Remove $i$ with smallest $f_i := g_i + \epsilon h_i$ from OPEN
5:      Insert $i$ into CLOSED
6:      for $j \in$ Children($i$) and $j \notin$ CLOSED do
7:          if $g_j > (g_i + c_{ij})$ then
8:              $g_j \leftarrow (g_i + c_{ij})$
9:              Parent($j$) ← $i$
10:             Insert $j$ into OPEN
```

Figure 1: A^* algorithm

3 Technical Approach

3.1 Search-based planning

3.1.1 RTAA*

Implementation detail: Since original A^* can be very slow for large maps, so we use incremental search methods, $RTAA^*$. The algorithm only expand certain amount of node at each time step and took several actions after the decision, and the number of actions are bounded by an integer max_num_move_per_expand.

To save memory, I choose dictionary to store my node and heuristic. At each time step, I consider 26 direction around each node(note that it is 26 direction in cube not in sphere). To make sure that our planner return a new position in 2 seconds, I adjust the maximum number of expanded node (i.e N in program). Besides, in case of limited time exceeded, I return a random small movement every 2 second.

Computational Complexity, Completeness and Optimality: The computational complexity depend on the expanded node, ϵ of the heuristic, the complexity of map and the maximum number of step at each search. At each search step, the complexity is $O(Nlog(N)+26N)$. And the number of search step depend on the map, ϵ and maximum number of step. Intuitively, more complex and larger map, smaller ϵ and less step take each round would increasing the search time, which can be observed in later result. Increasing ϵ will reduce search time at the cost of returning a ϵ sub-optimal path. When expand node N are sufficient large, $RTAA^*$ approximates A^*. In this case, set ϵ to 1 will guarantee the optimality.

3.1.2 visibility graph

Implementation detail: Instead taking the whole space into account, we only need to consider the boundaries. For simplicity, I only consider the 8 corners of each block plus start and goal node. In this case, the computation time for A^* search are greatly reduced.

Specifically, $RRTA^*$ considers the whole 3D plane(as meshgrid), while visibility graph only considers hundreds of points. Similarly, we return some random small movement every two second if the search process is not terminated.

Collision Checking: It is worth to mention that I use a special method of collision checking. To avoid the path collide wit blocks, I check the point along the path every small step (0.08 in our case). Also, in visibility graph, we might encounter the case where the optimal path lie between two block containing 0 volume. However, simply set the condition equal to 0 does avoid the case, but it also avoid the optimal path that brush against the boundary with non-zeros volume. So, I check n (usually 8 or more) equally separated directions that vertical to the path. Specifically, I construct one vertical direction with very tiny length (delta = 0.001 in our case), and use quaternion matrix to rotate it. Only if all the tiny checking points lies inside the blocks or outside boundaries, then kick the path out.

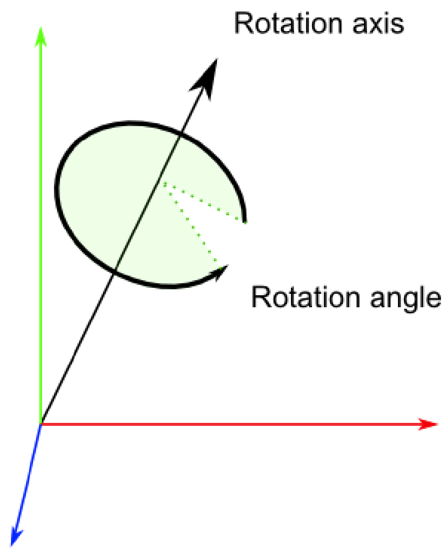


Figure 2: Quaternion Rotation.

Computational Complexity, Completeness and Optimality: Since we need to check collision and volume, adding long path can be extremely slow. We need to point out that actually our methods do not return an optimal path in 3D. For optimal one, we need to consider all the boundaries of the block instead of only corners, but it would be super slow to discretize all the boundaries. Our computational Complexity Would be $O(Nlog(N) + N^2/stepsize)$, where N is the number of blocks in graph. $O(Nlog(N)$ is A^* searching time and $N^2/stepsize)$ is graph building time with collision checking.

3.2 sampling based planning

3.2.1 Rapidly Exploring Random Tree (RRT)

Implementation detail: I first implement a naive version of RRT with probability of pg choosing goal as xrand(i.e SAMPLEFREE). All the node are saved using adjacent list using dictionary. For easy graph, my algorithm will terminate in 2 second, but for complex maps like maze and monza, which have multiple narrow path, it is super slow. Same as before, I return a random movement every 2 second. Another ideal I have tried is with probability of p_{edge} to choose corner as my xrand, since it is more likely to overcome obstacles by exploring around the edges.

Computational Complexity, Completeness and Optimality: The complexity is hard to estimate because of the randomness. For A^* search time, it is $O(Nlog(N))$, since the graph are pre-built, thus can finish in millisecond. Graph building is the most time-consuming process, usually larger pg will reduce the building time because randomness is reduced, though it is not helpful in complex maps. All in all, there is no upper-bound of time-complexity, but it will eventually reach the goal when time approximate infinity. The naive version of RRT returns a highly sub-optimal path, to improve the performance, I implement simple rewiring step in later Bi-RRT and n-RRT algorithm.

3.2.2 Bi-directional RRT (RRT-Connect)

Implementation detail: The experiment shows a dramatically improvement in speed when tree are growing from two side. Besides, I also implement RRT-Connect option, which can relax the ϵ constraint on the tree growth.

Computational Complexity, Completeness and Optimality: The computational complexity are bounded by graph building process. With the help of RRT-connect and Bi-directional algorithm, building time is greatly reduced. Also, our algorithm are more robust to complex maps, and can finish maze and monza though a little bit slow.

3.2.3 n-RRT with rewiring

Implementation detail: To further boosting the building time, I design an n-RRT algorithm. There are n trees, including start node and goal node, growing simultaneously. When n = 2, it degenerates to Bi-RRT algoritm. The roots of trees are randomly sampled from free space. Each round I randomly choose two trees from existing list and start growing. When they are connected, I merged them and delete the redundant tree from existing list. After connecting all the trees (i.e only one tree in exisiting list), I return the tree the do the rewiring step. Because there is search time for A^* is super fast, I connect all the node within the sphere with radius r^*, where r^* is computed using:

$$r^* > 2\left(1+\frac{1}{d}\right)^{1/d}\left(\frac{V_O l\left(C_{\text{free}}\right)}{Vol(\text{ Unit d-ball })}\right)^{1/d}\left(\frac{\log|V|}{|V|}\right)^{(1/d)} \tag{2}$$

Computational Complexity, Completeness and Optimality: Similarly, the search algorithm is A^* and the time complexity is $O(Nlog(N))$, where N is number of node. The is no upper bound for graph building, but it is super fast when n is large(can even finish maze within one second!). The rewiring step is N^2, because we need to consider all the node's near r^* nodes. It is worth to mention that though increasing n can greatly reduce the graph building time, but the path can be highly sub-optimal without rewiring. Because there is only one connection between one tree with another tree, so we must go across the connected point to reach the goal. After rewiring, the path will greatly improved.

4 Result

4.1 Hyperparameters

4.1.1 RTAA*

For our RTAA* algorithm, the optimality of path and computational time are only dependent on number of expanded node(N) , weight of heuristic ϵ and heuristic function. Here we try Manhattan distance and l2-norm (Euclidean distance). It easy to observe that larger ϵ and N would lead to shorter distance and faster computation time. Also, Good choice of heuristic(l2 in this case) does help improving the result.

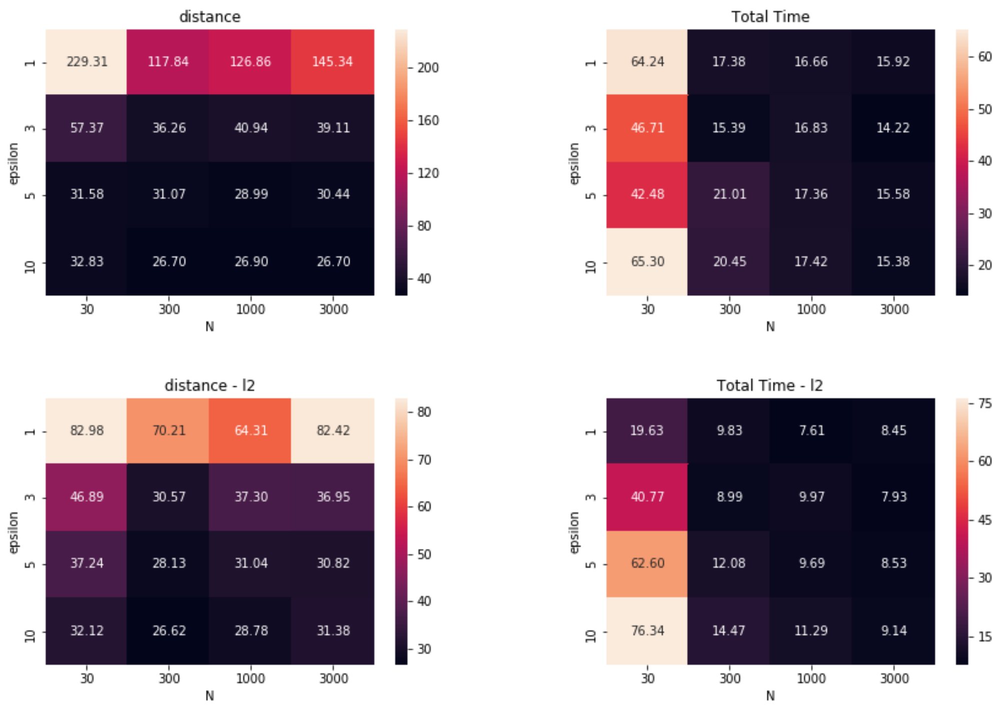


Figure 3: Distance and computational time analysis of RTAA*. Here we take flappy bird as example. The fisrt row use Manhattan distance and second row use Euclidean distance for heuristic.

4.1.2 Visibility Graph

For visibility graph, there is not too much hyper parameters to tune. Since the computational time of this method are bounded by graph building, we set ϵ to 1 for all maps. We can see from table below that there is no significant difference between the two heuristic function.

heuristic	map	distance	total_time
abs	test_single_cube	7.920000	0.223214
abs	test_flappy_bird	25.276950	7.349077
abs	test_monza	72.837239	2.213623
abs	test_window	27.958765	22.779368
abs	test_tower	37.900808	92.208635
abs	test_room	11.975046	96.339494
abs	test_maze	72.744169	155.913754
l2	test_single_cube	7.920000	0.165659
l2	test_flappy_bird	25.276950	7.680370
l2	test_monza	73.024460	1.815240
l2	test_window	27.958765	24.104139
l2	test_tower	37.900808	92.673546
l2	test_room	11.147747	99.112316
l2	test_maze	70.770441	164.668419

4.1.3 RRT

For naive RRT, the only hyper-parameters is probability of goal. We can see from Fig. 4 that total distance increase and total time decrease as probability of goal(pg) increase. It is intuitive that more chances to the goal leads to faster convergence, but more easier to get stuck in obstacles and return a highly suboptimal path.

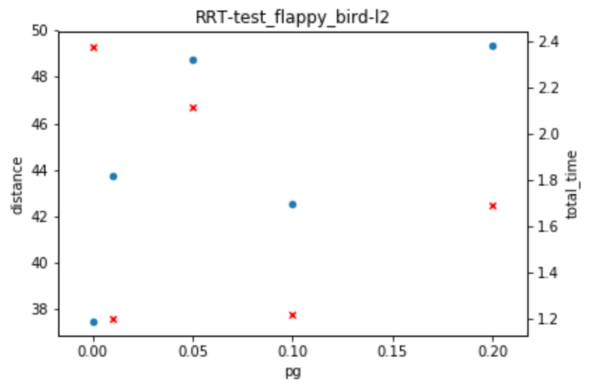


Figure 4: RRT distance and time vs probability of goal(pg) plot

4.1.4 Bi-RRT (RRT-connect)

Here we explore the effectiveness of RRT-connect. From the plot 4.1.4, we can easily observe that RRT-connect largely decrease the computational time and return a more optimal path. Thus, in the following n-RRT algorithm, I will keep using RRT-connect algorithm.

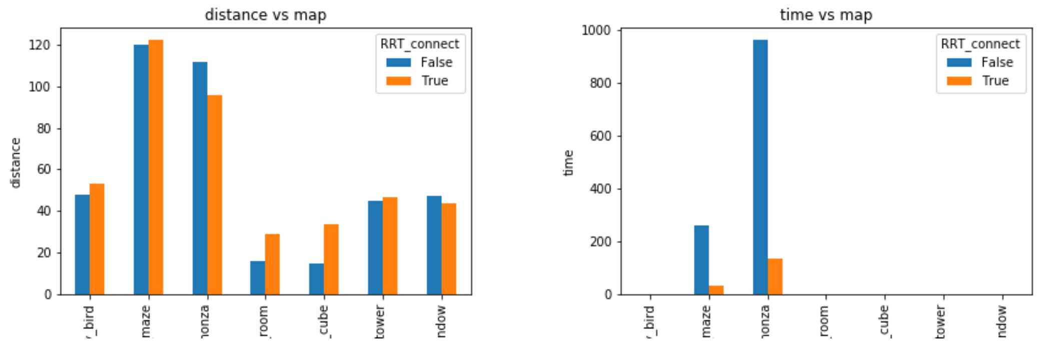


Figure 5: Effectiveness of RRT-connect.

4.1.5 n-RRT (rewiring)

n-RRT algorithm is till now the most efficient way of solving hard maps(eg. monza, maze). Here we take monza as example, we can see that computational time are greatly reduced when number of trees is less than 10, and slowly increasing after that. Interestingly, the distance is showing the same behavior. It is intuitive, as explained above, there is only one connection between one tree with another tree, so we must go across the connected point to reach the goal, thus increasing tress would increase the total distance. By tuning the number of trees, we can get the optimal number of trees. In this example (monza), the optimal number of trees is 5.

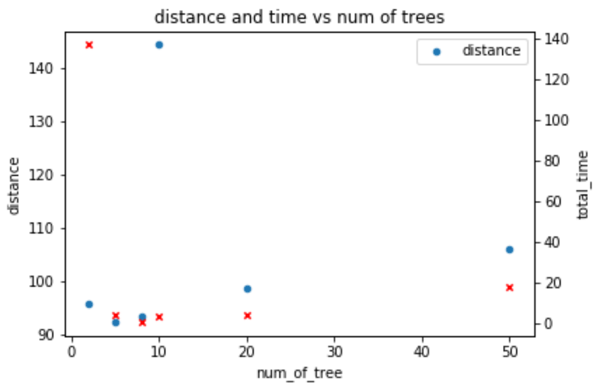


Figure 6: nRRT distance and time vs number of trees plot, example: monza. Where the red marker is time and blue is path length.

To further reduce the path length, I implement a simply rewiring algorithm. We can see from Fig. 4.3 that total distance are greatly reduced after rewiring, but at the cost of tens or even hundreds of rewiring time.

4.2 Comparison between search based and sample based method

From plot below, we can easily observe that for search based method $RTAA*$, it has very high accuracy but usually super slow in computation. When computational time is limited,

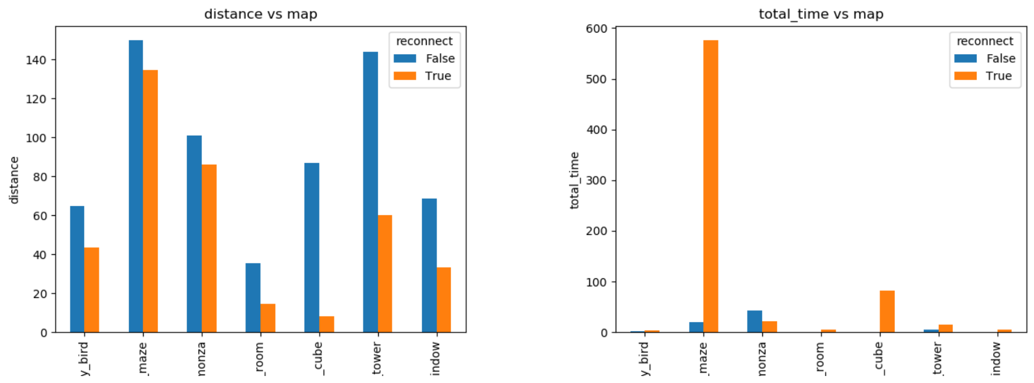


Figure 7: Effectiveness of rewiring.

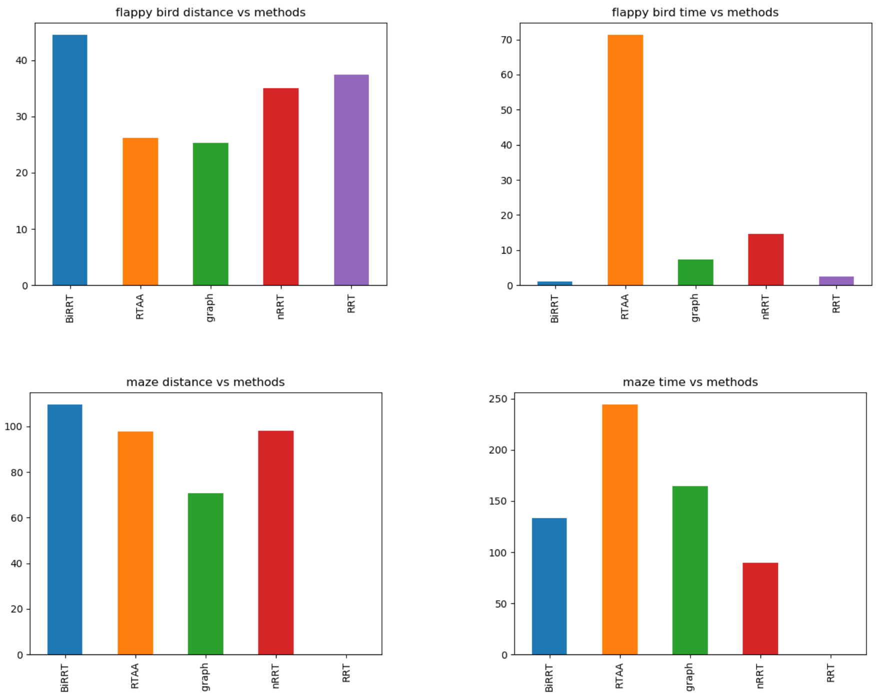
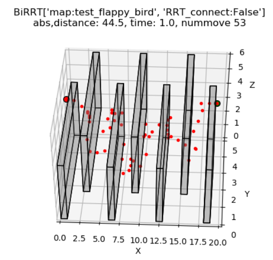


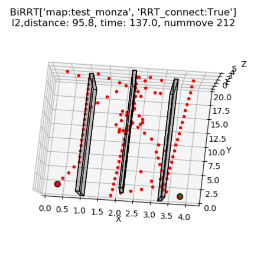
Figure 8: Compare for all the methods. Note that RRT cannot solve maze, so I leave it blank in second row.

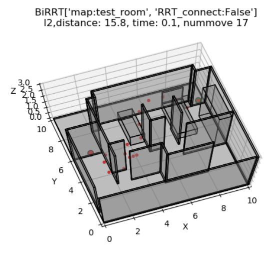
it can only span few nodes and might return a highly optimal path. For another search based methods, visibility graph, is till now the best method, no matter in computational time and optimality. However, the complexity increase quadratically as the number of block increases. For sampling based methods, they are generally return a more sub-optimal path than search based methods, but it is incredible fast. However, it still slow in complex maps with narrow passings. In graph below, we can see that maze is even unsolvable for naive RRT algorithm. After introducing n-RRT algorithm, the computational time is largely reduced, but optimality is even worse. Rewiring is often needed to refine the path at the cost of increasing computaional time.

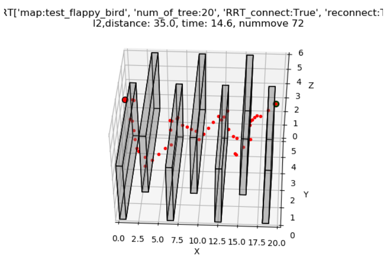
4.3 Best Result

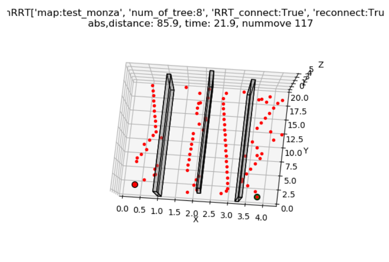
Here we present all the best result for different maps:

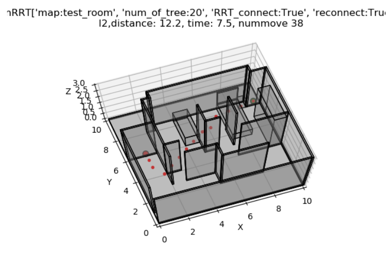


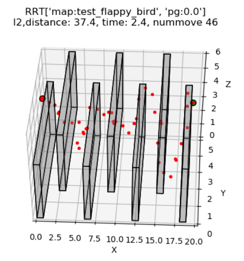


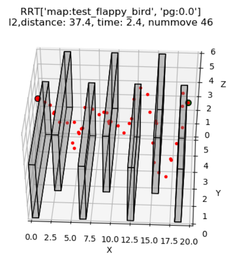


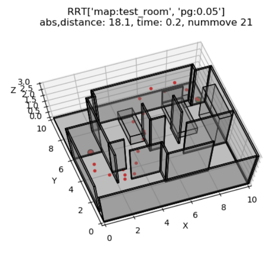


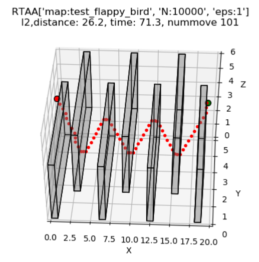


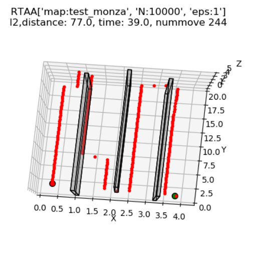


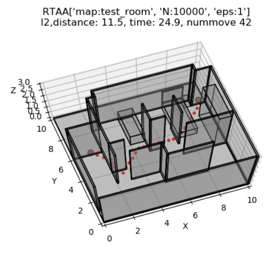


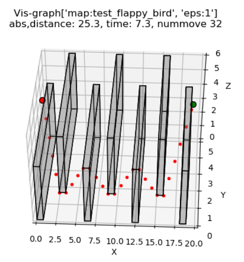


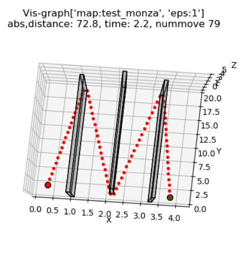


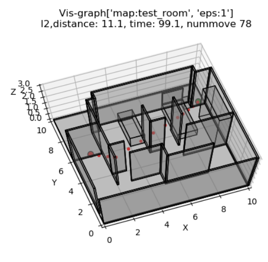




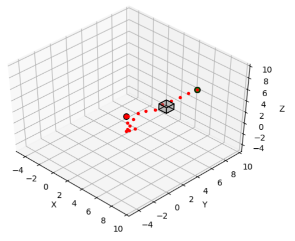




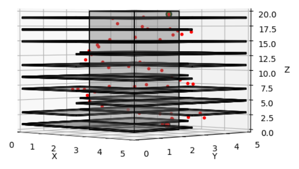




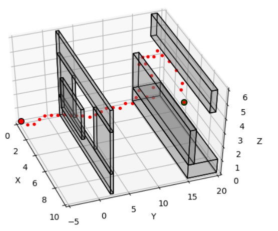
BiRRT['map:test_single_cube', 'RRT_connect:False']
l2,distance: 14.5, time: 0.0, nummove 16



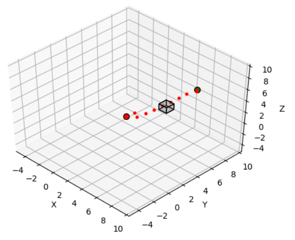
BiRRT['map:test_tower', 'RRT_connect:False']
l2,distance: 44.9, time: 1.8, nummove 56



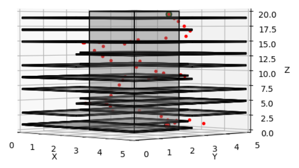
BiRRT['map:test_window', 'RRT_connect:False']
abs,distance: 34.4, time: 0.1, nummove 37



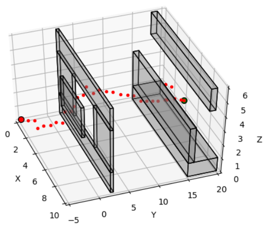
RRT['map:test_single_cube', 'pg:0.1']
abs,distance: 9.1, time: 0.0, nummove 10



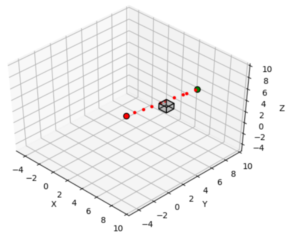
RRT['map:test_tower', 'pg:0.2']
abs,distance: 38.0, time: 3.3, nummove 50



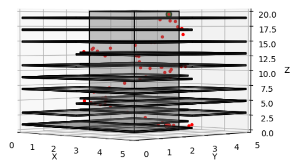
RRT['map:test_window', 'pg:0.1']
l2,distance: 28.8, time: 0.1, nummove 33



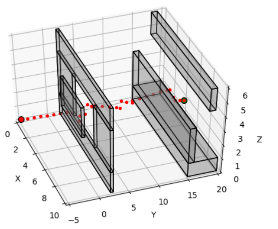
RT['map:test_single_cube', 'num_of_tree:8', 'RRT_connect:True', 'reconnect:T
l2,distance: 7.9, time: 81.4, nummove 67



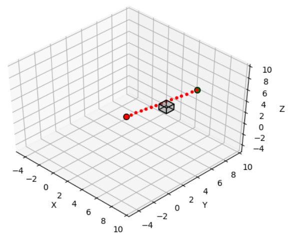
nRRT['map:test_tower', 'num_of_tree:20', 'RRT_connect:True', 'reconnect:Tru
l2,distance: 37.5, time: 73.7, nummove 118



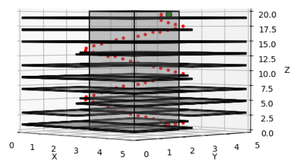
RT['map:test_window', 'num_of_tree:20', 'RRT_connect:True', 'reconnect:Tru
l2,distance: 26.0, time: 35.2, nummove 72



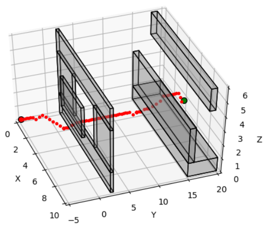
RTAA['map:test_single_cube', 'N:1000', 'eps:1']
abs,distance: 8.1, time: 0.0, nummove 14



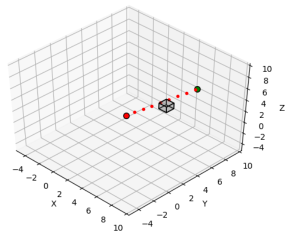
RTAA['map:test_tower', 'N:10000', 'eps:1']
l2,distance: 28.6, time: 172.5, nummove 161



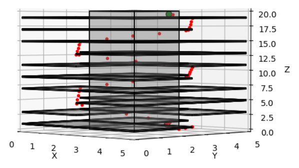
RTAA['map:test_window', 'N:1000', 'eps:1']
l2,distance: 26.4, time: 34.8, nummove 87



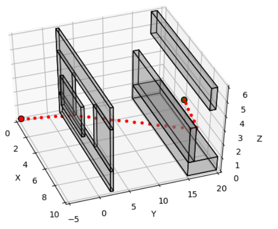
Vis-graph['map:test_single_cube', 'eps:1']
abs,distance: 7.9, time: 0.2, nummove 8



Vis-graph['map:test_tower', 'eps:1']
abs,distance: 37.9, time: 92.2, nummove 108



Vis-graph['map:test_window', 'eps:1']
abs,distance: 28.0, time: 22.8, nummove 46



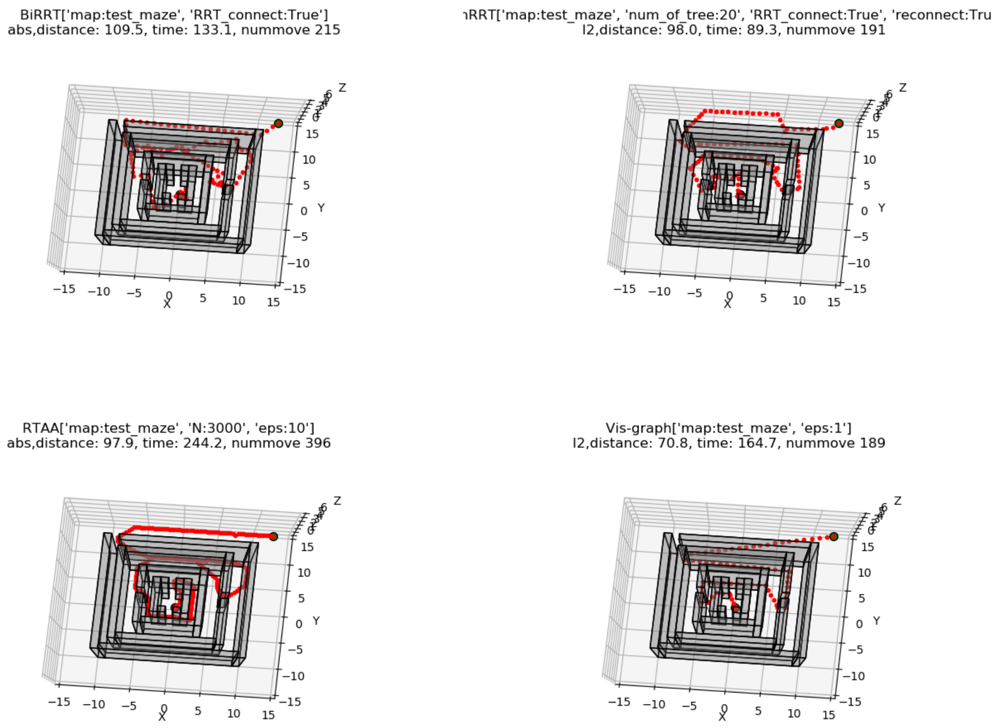


Figure 9: All best result for different methods and figure.