Search-Based and Sampling-Based Planning: Project Report

Pengluo Wang
Department of Electrical Computer Engineering
University of California, San Diego
pew067@eng.ucsd.edu

Abstract—This paper presented a brief overview for some basic search-based and sampling-based planning algorithms including A* and RRT algorithm. The project has been uploaded to https://github.com/PenroseWang/Motion-Planning-Basics.

Index Terms—Search-based planning, Sampling-based planning, A*, RRT

I. Introduction

Automated planning and scheduling, sometimes denoted as simply AI planning, is a branch of artificial intelligence that concerns the realization of strategies or action sequences, typically for execution by intelligent agents, autonomous robots and unmanned vehicles [1]. Here in this paper we will focus on motion planning based on deterministic shortest path problem and some basic algorithm to deal with it.

The basic objective of deterministic shortest path problem is to find a path from start node s to end node t with minimum cost in a given graph. This has important practical applications, one of which is to automatically find directions between physical locations given the road map (like Google Maps), since a road map can be considered as a graph with positive weights.

Structure of the report is listed as follows:

- Chapter II briefly states the problem to be solved in mathematical terms.
- Chapter III clarifies the technical approaches.
- Chapter IV presents results of the planning algorithms, and gives a brief analysis given the results.

II. Problem Formulation

A. Deterministic Shortest Path and Motion Planning

Consider a graph with a finite vertex space $\mathcal{V}$ and a weighted edge space $C := \{(i, j, c_{ij}) \in V \times V \times R \cup \{\infty\}\}$ where c_{ij} denotes the arc length or cost from vertex i to vertex j. The graph is shown in figure 1.

The Deterministic Shortest Path (DSP) Problem can be formulated as follows:

- 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 $\tau \in \mathcal{V} : Q_{s,\tau}$.
- Path Length: sum of the arc lengths over the path: $J^Q = \sum_{t=1}^{q-1} c_{t,t+1}$.

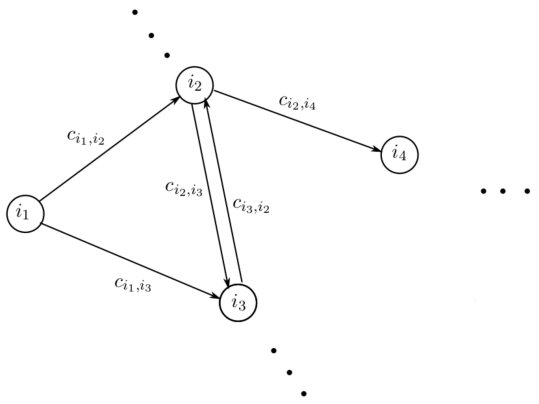


Fig. 1: Deterministic Shortest Path Problem

- Objective: find a path $Q = \text{argmin}_{Q \in Q_{s,\tau}} J^Q$ that has the smallest length from node $s \in \mathcal{V}$ to node $\tau \in \mathcal{V}$.
- Assumption: For all $i \in \mathcal{V}$ and for all $Q \in \mathcal{Q}_{i,i,}, J^Q \leq 0$, i.e., there are no negative cycles in the graph and $c_{i,i} = 0, \forall i \in \mathcal{X}$.

The motion planning problem is to find a feasible path (maybe cost-minimal) from the current configuration of the robot to its goal configuration. This can be solved using DSP algorithm. Two popular algorithms include search-based algorithm (like A*) and sampling-based algorithm (like RRT).

B. A* and RTAA* algorithm

One popular kind of algorithm for solving DSP problem is the label correcting (LC) method, which is a general algorithm for SP problems that does not necessarily visit every node of the graph. This kind of algorithm guarantees to find with the shortest path from node s to τ if there exists at least one finite cost path between the two nodes. A* algorithm is a modification to the LC method with adding prior information into the process, which we call heuristics. The algorithm is shown in table I.

However sometimes in really large unknown environments, it is impossible to compute the path all the way up to the goal, thus we introduce another variation of the A* algorithm to solve this kind of situation, which is RTA* algorithm as shown in table II. This algorithm guarantees

OPEN $\leftarrow \{s\}$, CLOSED $\leftarrow \{\}$, $\epsilon \geq 1$
$g_s = 0$, $g_i = \infty, \forall i \in \mathcal{V} \backslash \{s\}$
while $\tau \notin$ CLOSED do
    Remove i with smallest $f_i := g_i + h_i$ from OPEN
    Insert i into CLOSED
    for $j \in$ Children(i) and $j \notin$ CLOSED do
        if $g_i > g_i + c_{ij}$ then
            $g_j \leftarrow g_i + c_{ij}$
            Parent(j) $\leftarrow i$
            Insert j into OPEN

TABLE I: A* algorithm

Expand N nodes using A* algorithm.
Update h-values of expanded nodes i by $h_i = f_{j^*} g_i$ where $j^* = \text{argmin}_{\ j \in OPEN}\ g_j + h_j$ (requires only a single pass through the nodes in CLOSED).
Move on the path to state $j^* = \text{argmin}_{\ j \in OPEN}\ g_j + h_j$.

TABLE II: RTAA* algorithm

$V \leftarrow \{x_s\}; E \leftarrow \varnothing$
for $i = 1, \ldots, n$ do
    $x_{rand} \leftarrow$ SampleFree()
    $x_{nearest} \leftarrow$ Nearest((V, E), x_{rand})
    $x_{new} \leftarrow$ Steer($x_{nearest}, x_{rand}$)
    if CollisionFree($x_{nearest}, x_{new}$) then
        $X_{near} \leftarrow$ Near((V, E), x_{new}, min$\{r^*, \epsilon\}$)
        $V \leftarrow V \bigcup \{x_{new}\}$
        $c_{min} \leftarrow$ Cost($x_{nearest}$) + Cost(Line($x_{nearest}, x_{new}$))
        for $x_{near} \in X_{near}$ do
            if CollisionFree(x_{near}, x_{new}) then
                if Cost($x_{nearest}$) + Cost(Line($x_{nearest}, x_{new}$)) $< c_{min}$ then
                    $x_{min} \leftarrow x_{near}$
                    $c_{min} \leftarrow$ Cost($x_{nearest}$) + Cost(Line($x_{nearest}, x_{new}$))
        $E \leftarrow E \bigcup \{(x_{min}, x_{New})\}$
        for $x_{near} \in X_{near}$ do
            if CollisionFree(x_{new}, x_{near}) then
                if Cost(x_{new}) + Cost(Line(x_{near}, x_{new})) $< x_{near}$ then
                    $x_{parent} \leftarrow$ Parent(x_{near})
                    $E \leftarrow (E \backslash \{(x_{parent}, x_{near})\}) \bigcup \{(x_{near}, x_{new})\}$

TABLE III: RRT* algorithm

that the goal is reached in a finite number of steps given that there exists a feasible path.

C. RRT* algorithm

Sampling-based algorithm is another popular method for solving motion planning problem and also a better choice in high dimensions. It guarantees that the probability of finding one if it exists approaches one as the number of iterations goes to infinity. The process contains two steps: first generates a sparse sample-based roadmap (graph $\mathcal{G}$), which hopefully should be accessible from any point in the free space $\mathcal{C}_{free}$; secondly given a start configuration x_s and goal configuration x_τ , connect them to the roadmap $\mathcal{G}$ using a local planner, then search the augmented roadmap for a shortest path between the two nodes.

Rapidly Exploring Random Tree (RRT) is one of the most popular sampling-based planning techniques and one of its extension RRT* algorithm has been shown in table III [2].

III. Technical Approach

A. RTAA* Algorithm Design

Design of the RTAA* algorithm is somewhat straightforward. However some effort should be made in order to make the program runs faster and functions well in the time constraint.

1) Map representation: The most direct way to represent a road map is to discretize the map into small volumes and create a binary 3D array to save the map inform, possibly using "0" to represent free space $\mathcal{C}_{free}$, and use "1" to represent $\mathcal{C}_{obstacle}$. And similarly the heuristic can also be saved in a 3D array of same size. However this requires the program to know the size of map before hand and is not memory efficient. A better way is to use a hash map to save just the nodes and edges of the graph and another one for the heuristic values. This has the advantage that we don't need to know the size of environment before hand and easy to expand the map if needed. Another advantage is that we can forget or discard the information saved in the hash map easily if it is no longer need.

Resolution of the map is set to be 0.1, which guarantees that we can reach the goal within an acceptable distance. And the robot can move into 26 directions, which are the neighbors of a volume in 3D space. If the neighbor volumes are connected is determined by the block information. If the line segment formed by the center of two neighbor volumes do not collide with the blocks, then we construct the edge between them and save it into the hash map. This information will be reused if the robot visit the neighborhood again.

2) Time constraint: For each move we have a time constraint of two seconds. Expanding N nodes using A* and find optimal path using A$*$ algorithm after heuristic update are the most time consuming parts in the algorithm design. In order to fully use the time between each move and guarantee that processing will complete in two seconds, the algorithm is designed to have adaptive N and can be stopped if time exceeds limit when finding optimal path and continue to process in the next move. This will greatly improve robustness of the system.

This design can be easily implemented if we have a timer inside the robot. The configuration of the timing is as follows:

- Assign one second for expanding N nodes using A* algorithm. Thus N may vary for each move. After this procedure we have a OPEN list and CLOSED list containing the visited or to be visited nodes in the graph.
- Update heuristics in CLOSED list. The time cost of this step can be neglected.

- Find optimal path using A* algorithm. This step may require much time depending on the number of node in CLOSED list. Thus we allocate another 0.5 second for it to be finished in time during this move. If time exceeds 0.5 second, then we save the data and continue planning during next move. For current move, the robot stays in the same position.

B. RRT* Algorithm Design

Design of RRT* Algorithm mostly follows the steps described in table III and a lot of details should be paid attention to.

1) Map representation: Similar with before I represent the graph using hash map. Since we need to record the minimum cost c_{min} for each node, it's better to use dictionary in Python to save the nodes along with costs. SampleFree() will sample free nodes in the map using the blocks information and Nearest() and Steer() function will determine the location of new node if it's both in boundary and collision free. Here I use goal-biased sampling with probability 0.01 in order to achieve faster sampling procedure.

2) Extend node and rewire tree: Near() function will find the neighbors of new node and find the best connecting node to achieve minimum cost. Then the neighborhood will be rewired to ensure asymptotic optimality [2].

3) Time constraint: The most time consuming part of are constructing tree and planning after tree has been completed. Similar with before, The time for constructing tree is set to be less than 1.5 s. After constructing the tree, motion planning is easy because we have saved the parent for each node. Simply traversing the graph beginning at the goal node and stop at the start should work. This process can also be stopped and continue during next move if time exceeds 1.5 s.

4) Some modification: Some modifications have been made to improve the performance of RRT* algorithm:

- Nearest() function: normally Nearest() function will return the node with nearest distance, whether the node is located in the tree or on the edge of the tree. However the performance is not good enough on "monza" and "maze" map, partially because the space has been separated in to parts that only connected by narrow corridors. In my implementation, Nearest(x_{rand}) function will return the nearest node $x_{nearest}$ which satisfies that line segment of $x_{nearest}$ and x_{rand} is collision free in the map. If there does not exist a nearest node which leads to collision-free path, Nearest() functions as before.
- Path smoothing: RRT* algorithm will not usually give us the optimal path with minimum cost. Path smoothing will significantly improve this situation.

IV. Results and Analysis

This chapter will display some results showing the effectiveness of both algorithms given seven different maps. I show the results both visually and numerically. Some analysis and discussion will also be made accordingly.

A. Test Results

Obviously the choice of heuristic function will influence the performance of RTAA*. Here I compare the results obtained using Euclidean distance and diagonal distance as heuristic, shown in figure 2. Results on map "maze" and "monza" haven't shown here because the implementation failed to find a feasible path in limited time. Results of RRT* are shown in figure 3 and 4. The numerical results are shown in table IV.

B. Analysis

1) Comparison of heuristics: As we can see Euclidean heuristics works better on map "single_cube", "window", and "tower", but diagonal heuristics leads to better performance on map "flappy_bird" and "room". Thus which kind of heuristics to choose should depend on the distribution of the blocks. This is reasonable because heuristics is in fact prior information we have for the unknown map. In map "single_cube", Euclidean heuristic is perfect given that there is no obstacle along the path between start and goal. Thus it gives us the best performance among all algorithms. This shows the power of A* algorithm if we can have a good heuristic function.

2) Comparison of RTAA* and RRT* algorithm: For RTAA* algorithm the robot will move instantly without planning ahead for a long while like RRT* algorithm. However RRT* will performance much better on complicated situations like map "monza" and "maze", given that RTAA* fails to find a path for a long time.

3) Advantages of Path smoothing: As we can see from table IV, after path smoothing RRT* algorithm leads to best performance for almost all cases. After path smoothing, total number of move will be significantly reduced as the robot will only head to the best direction, reducing noises caused by sampling when constructing road map (tree).

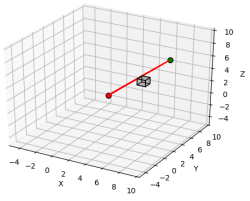
References

[1] Ghallab, Malik; Nau, Dana S.; Traverso, Paolo (2004), Automated Planning: Theory and Practice, Morgan Kaufmann, ISBN 1-55860-856-7.

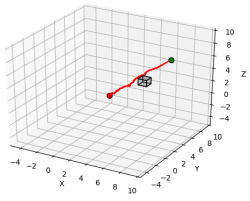
[2] Karaman, S., & Frazzoli, E. 2010, arXiv e-prints , arXiv:1005.0416.

Map	Algorithm	Success	Total moves	Total time (s)	Average/Max time per move (s)	Travel distance
single_cube	RRTA*, Euclidean	True	46	0.53	0.01/0.06	7.97
single_cube	RRTA*, diagonal	True	70	68.01	0.97/1.34	10.03
single_cube	RRT*	True	256	120.67	0.47/1.57	43.95
single_cube	RRT*, smoothed	True	15	1.56	0.10/1.50	8.14
window	RRTA*, Euclidean	True	237	237.62	1.00/1.46	27.57
window	RRTA*, diagonal	True	275	299.45	1.09/1.48	32.62
window	RRT*	True	102	10.01	0.10/1.52	32.00
window	RRT*, smoothed	True	40	9.98	0.25/1.53	24.24
room	RRTA*, Euclidean	True	130	121.39	0.93/1.73	16.47
room	RRTA*, diagonal	True	114	118.47	1.04/1.37	14.83
room	RRT*	True	75	4.22	0.06/1.50	22.28
room	RRT*, smoothed	True	19	1.86	0.10/1.52	10.88
tower	RRTA*, Euclidean	True	419	463.92	1.11/1.73	52.89
tower	RRTA*, diagonal	True	428	546.98	1.07/1.56	53.81
tower	RRT*	True	236	88.24	0.37/1.54	47.35
tower	RRT*, smoothed	True	95	82.91	0.87/1.59	28.73
flappy_bird	RRTA*, Euclidean	True	577	678.96	1.18/1.85	73.04
flappy_bird	RRTA*, diagonal	True	409	474.15	1.16/1.71	51.55
flappy_bird	RRT*	True	256	120.67	0.47/1.57	43.95
flappy_bird	RRT*, smoothed	True	53	24.02	0.45/1.55	26.49
monza	RRTA*, Euclidean	False	-	-	-/-	-
monza	RRTA*, diagonal	False	-	-	-/-	-
monza	RRT*	True	398	129.13	0.32/1.64	88.77
monza	RRT*, smoothed	True	185	133.04	0.72/1.61	74.72
maze	RRTA*, Euclidean	False	-	-	-/-	-
maze	RRTA*, diagonal	False	-	-	-/-	-
maze	RRT*	True	921	985.78	1.07/1.80	104.30
maze	RRT*, smoothed	True	671	898.68	1.34/1.78	73.87

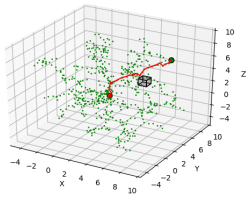
TABLE IV: Test results using different algorithms



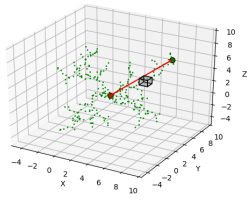
(a) "single_cube" - Euclidean



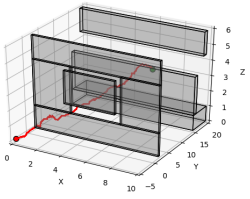
(b) "single_cube" - diagonal



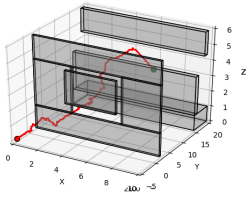
(a) "single_cube" - unsmoothed



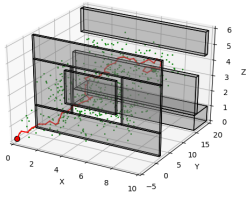
(b) "single_cube" - smoothed



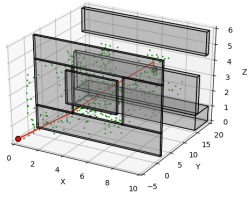
(c) "window" - Euclidean



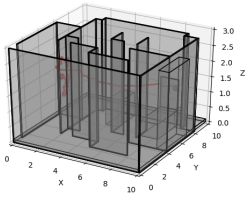
(d) "window" - diagonal



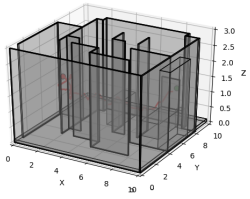
(c) "window" - unsmoothed



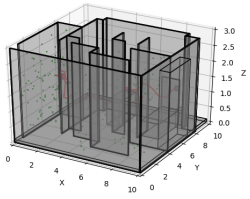
(d) "window" - smoothed



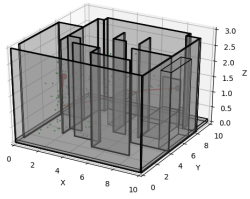
(e) "room" - Euclidean



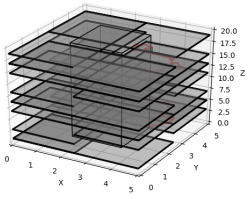
(f) "room" - diagonal



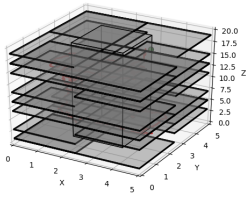
(e) "room" - unsmoothed



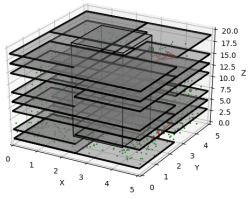
(f) "room" - smoothed



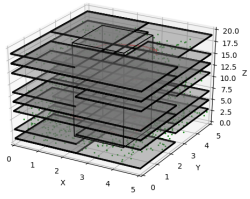
(g) "tower" - Euclidean



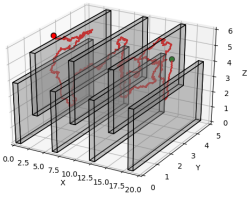
(h) "tower" - diagonal



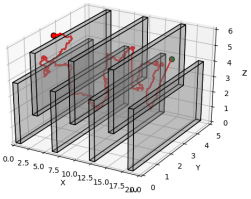
(g) "tower" - unsmoothed



(h) "tower" - smoothed

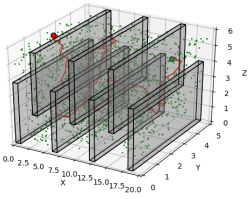


(i) "flappy_bird" - Euclidean

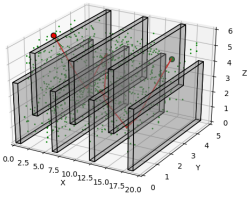


(j) "flappy_bird" - diagonal

Fig. 2: Results on different maps, RTAA*

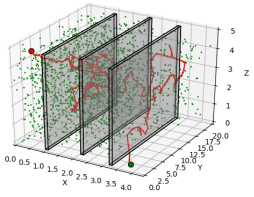


(i) "flappy_bird" - unsmoothed

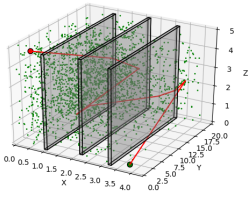


(j) "flappy_bird" - smoothed

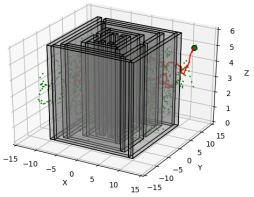
Fig. 3: Results on different maps, RRT*



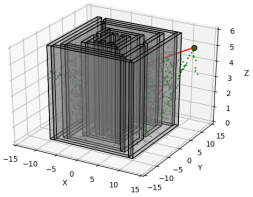
(a) ”monza” - unsmoothed



(b) ”monza” - smoothed



(c) ”maze” - unsmoothed



(d) ”maze” - smoothed

Fig. 4: Results on different maps, RRT*