Motion Planning with A*, RTAA* and Rapidly Exploring Random Tree Algorithms - A Project Review

Lien-Hsi Lu
Department of Electrical Computer Engineering
University of California, San Diego
llu@eng.ucsd.edu

***Abstract*—This paper is about an implementation of a Motion Planning using A* and RRT algorithm.**

I. Introduction

Living in this fast-pacing era, technology improves everyday. Robot has become more and more important in our lives, for example a large amount of companies has already improve the manufacture method by using robotics arm. In addition, lots of startups developing humanoid robots with sensors are also trying there best to introduce human-like robot to our house to replace maids or cleaner.

Based on the examples above, estimation, decision and control are the foundation stones for the robot to do any action. In this project, we aimed to plan a movement for a robot in different environment. This is any important technique for an Autonomous Vehicle to make decisions while it drives on the road. With the ability of planning and making decisions, the car can prevent collision to anything on the road and drive safely in the city.

To complete this task, we need to implement two popular algorithms of motion planning. As we know there are two big area of motion planning, which are search-based algorithms and sampling-based algorithms. In this project, A* and Rapidly Exploring Random Tree (RRT) will be implemented to solve the planning problems.

II. Problem Formulation

This section including some important parameters and limitations in this problem and the methods to solve this problem.

A. Boundary

The boundary of the environment is given in 3 axis with 9 dimensional vectors which represent the lower left corner, its upper right corner and the color RGB: [X_{min},Y_{min},Z_{min},X_{max},Y_{max},Z_{max},R,G,B].

B. Obstacles

The obstacles are rectangle blocks given in a set of 9 dimensional vectors which represent the lower left corner, its upper right corner and the color RGB: [X_{min},Y_{min},Z_{min},X_{max},Y_{max},Z_{max},R,G,B].

C. Start Point & Goal Point

The start point and the goal point are given inside the boundary with [X,Y,Z].

D. Limitation

There are a few limitation in this problem:

1) Distance: The distance of the robot move every time should be smaller than 1 unit.

2) Collision Free: The robot should be collision-free at any time.

3) Time: The robot should move within 2 seconds or the robot will move randomly which might occur collision or out of boundary.

4) Completeness: The robot should reach the goal at last.

E. Algorithm

1) Search-Based A Algorithm:* A* algorithm is a modification of the Label-Correcting algorithm, with the requirement for admission to OPEN list is strengthened with a heuristic function. The heuristic function is a positive lower bound on the optimal cost to get from node j to the goal.

Algorithm 2 Weighted A* Algorithm

```
1:  OPEN ← {s}, CLOSED ← {}, ε ≥ 1
2:  g_s = 0, g_i = ∞ for all i ∈ V \ {s}
3:  while τ ∉ CLOSED do                          ▷ τ not expanded yet
4:      Remove i with smallest f_i := g_i + εh_i from OPEN   ▷ means g_i + εh_i < g_τ
5:      Insert i into CLOSED
6:      for j ∈ Children(i) and j ∉ CLOSED do
7:          if g_j > (g_i + c_ij) then
8:              g_j ← (g_i + c_ij)                 expand state i:
9:              Parent(j) ← i                      ○ try to decrease g_j using path from s to i
10:             Insert j into OPEN
```

2) Search-Based RTAA Algorithm:* RTAA* is a modification of A* that can plan in real time. We will update the heuristic in the close-list to prevent stuck at local point.

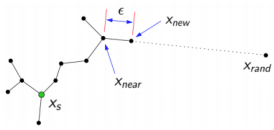
RTAA* with $N \geq 1$ expands

1. Expand N nodes
2. Update h-values of expanded nodes i by $h_i = f_{j^*} - g_i$ where $j^* = \arg\min_{j \in OPEN} g_j + h_j$ (only a single pass through the nodes in CLOSED!)
3. Move on the path to state $j^* = \arg\min_{j \in OPEN} g_j + h_j$

3) Sampling-Based Rapidly Random Exploring Tree: Rapidly Random Exploring Tree is a sampling-based planning algorithm. The tree is constructed from random samples with root x_s. The tree will keeping growing randomly until it contains a path to the goal.

Algorithm 3 RRT

```
V ← {x_s}; E ← ∅
for i = 1 ... n do
    x_rand ← SAMPLEFREE()
    x_nearest ← NEAREST((V, E), x_rand)
    x_new ← STEER(x_nearest, x_rand)
    if COLLISIONFREE(x_nearest, x_new) then
        V ← V ∪ {x_new}; E ← E ∪ {(x_nearest, x_new)}
return G = (V, E)
```



III. Technical Approach

A. RobotPlanner.py

The .py file is an object of Planner Class include some initialization variables, some functions and the planning algorithms.

1) Initialization: The initialization includes the boundary, blocks, a counter for the open-list, open-list, close-list, a dictionary used for storing g and h with the key of tuple points and a g-value parameter equal to 0. Those initializations are mostly used in the A* algorithm. Only the boundary and the blocks are used in the RRT algorithm.

2) A Algorithm:* I implemented A* algorithm function by two supporting functions which compute the transition cost and the value of the heuristic. The transition cost is compute by the distance of a moving direction. The function of heuristic needs 2 parameters which is the current state point and the goal point and return the distance between the 2 points. The function of A* needs 2 parameters which is also the current state point and the goal point as well. First, I initial the moving directions with the length of 0.5 unit for 26 direction in a 3D space. After that, I check if the current point in tuple type is in the dictionary or not. If not, I store the current g value and compute the heuristic of the point with the key of tuple point.

Then I construct a for-loop to go through every neighbor of the point which are the 26 potential moving directions. I first make the counter plus 1. This is because I don't want to confused the heap when popping out new node, I store a tuple of (g+h, g, counter, point) that if (g+h) and g are the same then it will pop out the smaller counter. After that for every iteration of the for-loop I make the new robot point by adding the original point and the direction vector. To check if the new robot point is valid or not I check if the point is in the boundary and is also collision-free. With valid points, because of A* algorithm, we need to check if the new points were already in the close-list or not. If yes, we will not consider the point. If no, I compute the transition cost and the heuristic for the point and check if the point is in the dictionary or not. If no, I directly update the infinity cost (g_j) into the initialized value g (g_i) plus the transition cost (c_{ij}) and store as a tuple (g_j, h_j) and append the tuple ($g_j + h_j$, g_j, $counter$, $[point]$) into the open-list. If yes, I take out the g_j stored in the dictionary and see if the old g_j is larger than the initialized value g (g_i) plus the transition cost (c_{ij}). If yes, I update the old g_j and store it into the dictionary and append the tuple ($g_j + h_j$, g_j, $counter$, $[point]$) into the open-list.

Lastly, I pop out the smallest g_j + ϵ*h$_j$ as setting the $\epsilon = 1$ to be the next robot position. However, I need to check that if this point is too far way to prevent conflicting the limitation. So I used a temp list to store the popped out node that conflicted the limitation and pop out next point in the open list until it satisfy the limitation. After obtained an eligible node, I append the node into the close-list and append nodes in the temp list back to the open-list. I updated the new robot position and the initialized value g and return the new robot position at last.

In conclusion, I implement A* in a real time way, although the robot can reach the goal, there are some problems that more discussion will be in the result section.

3) RTAA Algorithm:* In this algorithm, basically, the algorithm is quite same with A*, but we need to first expand the node, for example 20 in my implementation and update the heuristics. In this case, we choose 26 direction again for each expanded points and add them all to the open list. For every expanding iteration, I will also need to pop out a node from the open list to append into the close list. For the last point in the close list, I will take the ($g_j + \epsilon * h_j$) as f to update all the heuristic in the close list by doing the operation of $f - g$. At this time, I choose the ϵ to be 3 for trusting the heuristic more because of the update.

After updating the heuristics in the expanded nodes in the close list, I will implement A* again which the same as the A* mentioned before. In conclusion, this algorithm solve the problem in the A* case that will truely be a correct real time algorithm rather than modifying the A* algorithm into real time like above.

4) RRT Algorithm: I implemented the RRT algorithm with some functions, including the random point generator, compute distance, get nearest node, get steering point and

a segment collision checking function. In the random point generator, I generate random number with the condition [X_{min},Y_{min},Z_{min},X_{max},Y_{max},Z_{max},R,G,B] that the number of random x will be in the interval of X_{min} and X_{max}, using the same method to generate the random y as well as random z. To get the nearest node from the random generated point, I construct a node list for every node to store the expanded node, so I can use a for-loop to compute every distance between the node and the random generated point. Obtained the index of of the shortest distance node and get the node. Because of the limitation of the robot speed, I need to make a steering point on the line segment between the random generated point and the nearest node that the largest distance is no bigger than 1. I implemented this method by keep updating inner point on the segment by 1/4 and 3/4. Until the distance is not bigger than 1, then do a segment collision check. I use the method of checking the point is inside the block and (the x of end point - the minimum of x) times (the x of start point - minimum of x) smaller than 0 and apply this to y and z axis, if every condition holds, this means there must be a collision between the start and end point. If the steering point and the nearest node is segment collision-free and also the steering point is not collision itself, then I append the steering point into the node list.

So after implementing these functions above, I can easily implement RRT algorithm. Beside initializing a node list, I initialized a parent list in a dictionary with the key of point transform into string and the value to be the parent. At first I make the start point parent to be itself. After those initializations, I start to expand my tree in a while-loop with the condition of the distance between the point and the goal is bigger than 1. Once the parent of the goal is in the parent list, I will terminate and connect the steering point and the goal. In the while-loop, I first generate a random point in the space. After that, I get the nearest point in the node list to the random point. Then make a steering point that the distance is smaller than 1. Lastly, I check if the steering point is collision free and the steering point and the nearest point is segment collision point, I append the steering point into the node list and set its parent as the nearest point in the dictionary. When the while-loop is terminated, I add the goal point into the node list and make its parent as the last steering point.

Till this step, this means the tree is fully expanded in a there must be a path between the start point and the goal point. I just extract the path by getting the parent from the dictionary until the parent is the start. Then the planning part is finished.

In conclusion, I implement the RRT off-line with planning the path first and move the robot after planning is finished.

B. main.py

1) A Algorithm:* In this function I create a run test for A* algorithm. In this run test, I set a timing clock for the moving time and every time my planner will give out one new position of the robot. After the new point is generate, I check the distance between the old position and the new position to see if the robot is going too fast. Also check if the point is out of bound and collision. Iterate many times will make the new point very near to the goal. Plot it out every new point on the 3D environment to visualize the result.

2) A Algorithm:* This function will totally be the same as A* algorithm in main.py. But using the RTAA* function in RobotPlanner.py.

3) RRT Algorithm: In the run test of RRT, I first finished the planning part after move the robot after planning. In this case, the moving time is impossible to be longer than the 2 seconds limitation, because the longest time will be the planning part and the moving part is almost 0. So, the iteration is to extract the planned path point to be the new robot position and by doing the same limitation checking. Of course, the visualization of the result will be the same method mentioned above.

C. Computational Complexity

To compare the computational complexity of the two planning algorithms, the complexity of the A* algorithm is mainly depends on the for-loop of visiting the children and the complexity of the RRT algorithm is mainly depends on the while-loop generating new sample points. Although the A* and RTAA* algorithms looks like have the complexity of O(n), but the memory using space of A* is way more bigger than the RRT algorithm. In A* algorithm, I need to construct the space for the close-list, open-list, dictionary and if I want to know the parent, there should be one as well. But in RRT, the space I need is only the node list and the parent dictionary. Moreover, if I implemented differently, I can make the node list and the parent dictionary into a big dictionary, that will save even more space.

To talk about the computational complexity of the RTAA* algorithm, it has an additional while-loop for how many expansion times the for-loop than A*. Also some additional space for visited nodes and open list, so actually the space and memory are way more than A*. However, to obtain a better performance of this problem, I think this is the trade off.

To see the performance time, A* terminates way more faster than the RRT algorithm but that is just because of the random generate point has no strategy. If I generate the random point based on some heuristic reference, I think the algorithm will terminate way more faster. But this doesn't mean the big-O notation are different, actually based on both algorithms themselves, they have actually the same big-O notation. But

RTAA* needs a little more time to terminate than A* because there are additional loops mentioned above.

D. Completeness & Optimality

To discuss the the completeness of both algorithms, in theory all algorithms will give out the complete solution. However, in my real implementation, the A* star algorithm success in some test case but some will go across the wall with neglecting the collision. More discussion will be in the result section. In my implementation of RRT, even if the hardest test case "maze" is passed with certain path which is complete. For RTAA*, some cases is not complete as well, but in theory, the algorithm is also completeness.

To discuss the optimality of both algorithms, A* algorithm take the advantages. In the traditional A*, actually we plan the A* algorithm first and then move the robot after that, which is also the same as RRT. But actually A* will generate an optimal path rather than RRT. But in my case, because of making the A* in a real time way, every time the point visits its neighbor it will move directly but it might not be the most optimal step. After implementing RTAA*, the optimality of my algorithm seems not that good in some test cases, but in theory the RTAA* algorithm will come out optimal solution, so there are still some problem needs to be fix. But to imagine move after planning, the A* algorithm will store every nodes g value and heuristic and make a optimal decision at last, so it will be optimal. On the other hand, RRT will only generate a complete path but it will be highly sub-optimal and require post-processing, for example: path smoothing.

IV. Result

In this project, there are 7 test cases in the environment for the search-based and sampling-based planning algorithm. They are: single cube, maze, flappy bird, monza, window, tower and room.

A. A* Algorithm

The result of A* is that the robot can reach the goal in some test case but there are some problem that although I did check the collision in my code, but in the visualization, I still see that the robot is hitting the wall. I think this is because some of the wall is too thing that although the collision test is passed by still might confront a segment collision. By doing a segment collision function myself, the problem still not solved, so I just tried to implement RTAA* algorithm instantly.

This is a on-line algorithm using by modifying A*, however, this is not correct. A* is actually a off-line planning algorithm that the robot will plan an optimal path and then move afterwards. To make A* in real time (on-line), I implemented RTAA* algorithm.

TABLE I
A* Algorithm experiment statistics

Environment	Number of moves
Single Cube	34
Maze	Went out the first hole but got stuck
Flappy Bird	Almost Reach by failed
Monza	Almost Reach by failed
Window	Almost Reach by failed
Tower	Almost Reach by failed
Room	Almost Reach by failed

B. RTAA* Algorithm

The result of RTAA* is better than A* but still there are some problems. Some times the robot seems feasible walking through a path, but at some point the robot got stuck and can't get out. I tried to tune some parameters, for example, number of expanded node, ϵ, the resolution of moving direction, but still the two cases of Maze and Monza cannot pass. There is an interesting point that in some test cases, the robot will stuck at a certain point for a short time, but eventually the robot will update the heuristic and get out of the stucking point, which is the proof of using real time algorithm to solve the stucking problem. The robot will reach the goal at last.

This is a real time algorithm which is also called a on-line planning algorithm.

TABLE II
RTAA* algorithm experiment statistics

Environment	Number of moves
Single Cube	16
Maze	Got stuck
Flappy Bird	311
Monza	Got stuck
Window	57
Tower	186
Room	56

Here are the links to the videos of successful testing cases: Single Cube, Flappy Bird, Window, Tower, Room.

C. RRT Algorithm

The result of my RRT algorithm runs well that I show my result in by testing each test case 5 times and see how many nodes did the tree expand and how many move are there after the path is planned. Of course the move time won't be over 2 seconds because of the paths are already planned, which is a off-line algorithm.

However, because of the run time of Maze and Monza takes long. I only test them for 2 times. The reason that the Maze needs a longer time to plan is because it is a more complicated environment for the robot to plan, so taking longer time is feasible. Although the Monza looks easy to plan, the reason for the Monza needs longer time is because the trespass path is very narrow that the expanded node isn't that easy to expand

there, it takes more time in the three narrow trail. The result are like below:

TABLE III
RRT ALGORITHM EXPERIMENT STATISTICS

Environment	Number of nodes in the Tree	Number of moves
Single Cube	102 / 65 / 245 / 51 / 77	20 / 13 / 29 / 16 / 15
Maze	47959 / 56507	166 / 166
Flappy Bird	593 / 547 / 600 / 837 / 1216	53 / 50 / 48 / 54 / 58
Monza	69178 / 62479	173 / 173
Window	271 / 239 / 171 / 209 / 349	44 / 42 / 41 / 46 / 43
Tower	3462 / 2574 / 4132 / 1557 / 2074	58 / 66 / 68 / 59 / 58
Room	118 / 77 / 117 / 125 / 75	19 / 20 / 22 /22 /19

Here are the links to the videos of successful testing cases: Single Cube, Maze, Flappy Bird, Monza, Window, Tower, Room.

In conclusion, in this project I saw the different planning algorithm results by using the the search-based and sampling-based algorithms. The robot can really plan a path with the algorithms which will be a very significant application in the field of robotics.

REFERENCES

[1] Steven M. LaValle, “Planning Algorithms”