Path Planning

Saurabh Himmatlal Mirani (A53319557)
Planning & Learning in Robotics
University of California, San Deigo
smirani@ucsd.edu

I. Introduction

In robotics, we need to solve many shortest path problems. For example, in path planning, to reach to the goal in minimum amount of time or to use minimum fuel, we should find the shortest path to reach the goal quickly.

The path planning problem is one of the most important problems in autonomous robotics. Given the prior information of the environment and knowledge about robot motion capabilities, a path planning problem is then reduced to optimization problem where we want to find the best path under a set of constraints.

The problem of the robot to reach goal in the least possible steps is defined as shortest path problem. This problem doesn't have noise and hence is reduced to Deterministic Shortest Path problem. The rest of the paper is organised as: Section II formulates the problem, Section III desrcibes the technical approach, Section IV has the results and discussions and Section V concludes the paper.

II. Problem Formulation

The problem of motion planning requires to find a path from start to goal. In 3D space, this is a continuous space problem. We formulate the problem below.

Consider a graph with a infinite vertex space V and a weighted edge space $C := (i, j, c_{ij}) \in V \text{x} V \text{x} \mathbb{R} \cup \infty$ where c_{ij} denotes the arc length or cost from vertex i to vertex j.

Objective: find the shortest path from a start node s to an end node τ, where $s, \tau \in V$. This is the deterministic shortest path problem.

Path: a sequence $i_{1:q} := (i_1, i_2, ..., i_q)$ of nodes $i_k \in V$.

All paths from $s \in V$ to $\tau \in V$:

$\mathbb{I}_{s,\tau} := i_{1:q} | i_k \in V, i_1 = s, i_q = \tau$

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

Objective: find a path $i^*_{1:q} = \underset{i_{1:q} \in \mathbb{I}_{s,\tau}}{\arg\min} J^{i_{1:q}}$ that has the smallest length from node $s \in V$ to $\tau \in V$

Assumption: For all $i \in V$ and for all $i_{1:q} \in \mathbb{I}_{i,i}$, $J^{i_{1:q}} \geqslant 0$, i.e., there are no negative cycles in the graph.

For this project, c_{ij} is taken as the Euclidean distance between the two vertices, which is nothing but the l_2 norm in 3D space. For two vertices $a_i, a_j \in V$,

$$c_{ij} = ||a_i - a_j||_2$$

where $a_i = (x_i, y_i, z_i)$, and $a_j = (x_j, y_j, z_j)$

III. Technical Approach

The problem defined in Section II can be solved using many approaches. Two of the approaches have been discussed in the following sections.

A. Collision detection

We need two types of collision detection, first in which given a state $a \in V$, we need to know if a lies in free space or in obstacle. Second in which given states $a_i, a_j \in V$, we need to know if motion from a_i to a_j is collision free or not.

For the first one, we simply check using the coordinates if the point lies in obstacle or free space. We have the bounds of boundary and blocks. If the state lies in free space, True is returned or else False is returned

For the second one, we use Ray-Box Intersection. Given states $a_i, a_j \in V$, it is assumed that a_i is collision free. Now a ray is created from a_i passing through a_j. The obstacle boxes are converted to axis-aligned-bounding-box. This is done using the Pyrr [1] library. If the ray intersects aabb (may intersect at one or more points), the point closest to a_i is checked if it lies between a_i and a_j. If the point lies in between then the motion is not feasible. Else the motion is feasible and True is returned.

B. A* algorithm

The problem defined in Section II can be solved using A* algorithm.

A* is a grid based algorithm. Hence, we divide the continuous space into voxels first. We choose a map resolution (0.1m in this project). Hence we have a environment with voxels which are nothing but cubes of 0.1m x 0.1m x 0.1m. For all $a_i \in V$, $b_i = (x'_i, y'_i, z'_i) \in V'$

$$x'_i = [\frac{x_i - x_{min}}{MAP\ resolution}] \tag{1}$$

$$y'_i = [\frac{y_i - y_{min}}{MAP\ resolution}] \tag{2}$$

$$z'_i = [\frac{z_i - z_{min}}{MAP\ resolution}] \tag{3}$$

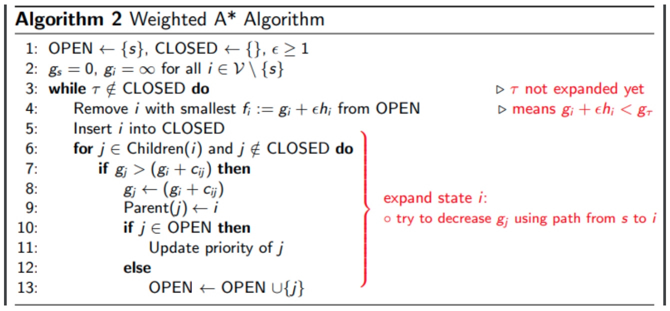
where, [.] is the greatest integer function or ceiling function.

The nodes in the graph are now voxels instead of vertices. The problem is now reduced to finite space vertices problem.

A* requires heuristic. For this project, h_j is taken as the Euclidean distance between the two vertices, which is nothing but the l_2 norm in 3D space. For two vertices $b_i, \tau \in V'$,

$$h_i = ||b_i - \tau||_2$$

where $b_i = (x'_i, y'_i, z'_i)$, and $\tau = (x'_\tau, y'_\tau, z'_\tau)$



```
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                                  ▷ $\tau$ not expanded yet
4:      Remove $i$ with smallest $f_i := g_i + \epsilon h_i$ from OPEN    ▷ means $g_i + \epsilon h_i < g_\tau$
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($j$) ← $i$                                   expand state $i$:
10:             if $j \in$ OPEN then                                ∘ try to decrease $g_j$ using path from $s$ to $i$
11:                 Update priority of $j$
12:             else
13:                 OPEN ← OPEN ∪{$j$}
```

The A* algorithm is a modification to the Label Correction (LC) algorithm in which the requirement for admission to OPEN is strengthened.

The criteria now is:

$$g_i + c_{ij} + h_j < g_\tau$$

where h_j is a positive lower bound on the optimal cost to get from node j to τ , known as a heuristic function.

The more stringent criterion can reduce the number of iterations required by the LC algorithm. The heuristic is constructed depending on special knowledge about the problem. The more accurately h_j estimates the optimal cost from j to τ, the more efficient the A* algorithm becomes. Hence, $\epsilon = 1$ is used.

The shortest path is then obtained from the stored parent nodes.Goal's parent is found, then it's grand-parent and so on until we reach start node. This is the required shortest path.

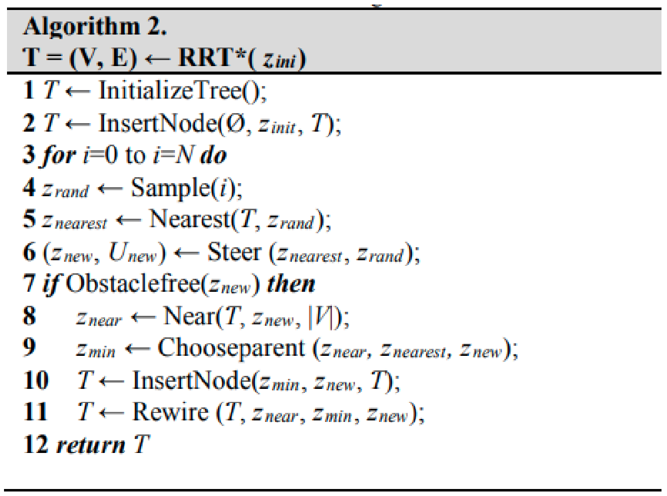
On finite graphs with non-negative edge weights A* is guaranteed to terminate and is complete, i.e. it will always find a solution (a path from start to goal) if one exists.

- Divide the environment into voxels of desired dimension
- The grid environment has values of free space as zero and obstacles as infinity (any number other than zero also works)
- This grid environment will help to identify if queried node is free or not
- Maintain a cost matrix, which has the lowest cost from start. In the beginning this matrix is initialize to infinity, with cost of start node as zero. In pseudo code, this represents the variable g
- Start 26 neighbourhood search from the start voxel
- Add the voxels/nodes which follow the criteria to OPEN list of nodes. For OPEN list, priority queue has been used using pqdict [2] library in python. The keys are the node/vertex cordinates and values are the f function value as in pseudo-code. Priority is given to the key with least value of f which is what is exactly required.
- Keep on propagating till goal is reached. If the OPEN list becomes empty before goal is reached, then there is no path.
- Take the parent list and get the path from start to goal
- Convert these grid based coordinates back to $\mathbb{R}^3$ space using the formula stated in equation (1),(2) and (3)., i.e. convert x'_i to x_i

We do not need to check if the path is collision free i.e. if motion is collision free or not between every two points of the solution path. This is because we are using low map resolution value (0.1) meaning the environment is divided into smaller voxels. If we use higher value, say 0.5, then, it is important to verify if the path is indeed collision free or not. Using low value however has a disadvantage. It exponentially increases the time and space complexity and hence must be chosen considering the trade off.

C. *Rapidly-exploring random tree-star (RRT*)*

The RRT* pseudocode [3]:



```
Algorithm 2.
T = (V, E) ← RRT*( $z_{ini}$ )
1 $T$ ← InitializeTree();
2 $T$ ← InsertNode(Ø, $z_{init}$, $T$);
3 for $i$=0 to $i$=$N$ do
4 $z_{rand}$ ← Sample($i$);
5 $z_{nearest}$ ← Nearest($T$, $z_{rand}$);
6 ($z_{new}$, $U_{new}$) ← Steer ($z_{nearest}$, $z_{rand}$);
7 if Obstaclefree($z_{new}$) then
8      $z_{near}$ ← Near($T$, $z_{new}$, $|V|$);
9      $z_{min}$ ← Chooseparent ($z_{near}$, $z_{nearest}$, $z_{new}$);
10     $T$ ← InsertNode($z_{min}$, $z_{new}$, $T$);
11     $T$ ← Rewire ($T$, $z_{near}$, $z_{min}$, $z_{new}$);
12 return $T$
```

OMPL [4] was used to implement RRT* algorithm.

- A 3D vector Space object is initialized.
- Bounds for the environment are added so that the planner know the environment boundaries
- Space Validity checker function is added which returns True if the state lies in free space and False if the state lies in a obstacle
- Motion Validator function is added which returns True if motion is possible from state-1 to state-2. False if there is obstacle in between. It assumes that state-1 is valid.
- Start and Goal states are provided
- Problem definition is initialized with Path Length Optimization Objective.
- RRT* is set as the planner
- A runtime is provided and the algorithm tries to compute the path in given time
- the object has a member funtion hasExactSolution() which is check to verify if goal is reached. If not, planner is given some more time to find path. This takes advantage of the fact that while re-planning the tree is stored in memory and reused. Hence it starts from where it left.

IV. Results and Discussions

A. Quantitative Comparison

Maze

Quantitative properties	A*	RRT*
Path Length (m)	75	71
Number of nodes expanded (vertices)	24533	6184
Planning time (s)	212.3	42.0

Flappy bird

Quantitative properties	A*	RRT*
Path Length (m)	26	26
Number of nodes expanded (vertices)	12173	304
Planning time (s)	51.2	1.0

Single Cube

Quantitative properties	A*	RRT*
Path Length (m)	8	7
Number of nodes expanded (vertices)	4764	12
Planning time (s)	1.3	1.0

Window

Quantitative properties	A*	RRT*
Path Length (m)	26	24
Number of nodes expanded (vertices)	39064	76
Planning time (s)	72.9	1.0

Room

Quantitative properties	A*	RRT*
Path Length (m)	11	10
Number of nodes expanded (vertices)	11709	66
Planning time (s)	13.4	1.0

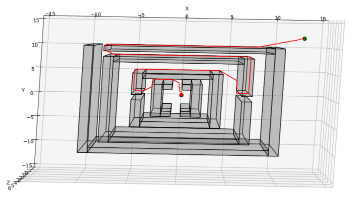
Tower

Quantitative properties	A*	RRT*
Path Length (m)	28	29
Number of nodes expanded (vertices)	7550	218
Planning time (s)	43.0	2.0

Monza

Quantitative properties	A*	RRT*
Path Length (m)	76	73
Number of nodes expanded (vertices)	6311	39725
Planning time	47.7	317.0

B. Solution Paths


(a) A* algorithm

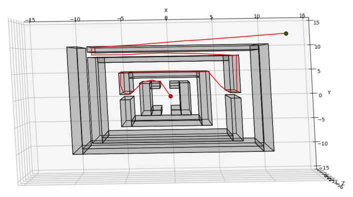
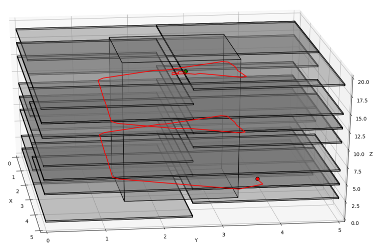

(b) RRT* algorithm

Fig. 1. Maze environment


(a) A* algorithm

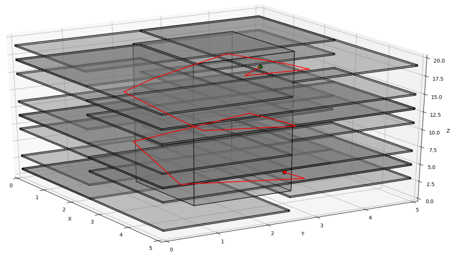
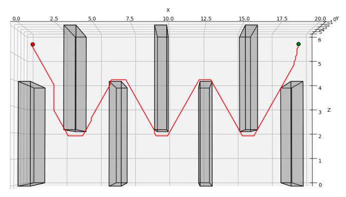

(b) RRT* algorithm

Fig. 2. Tower environment


(a) A* algorithm

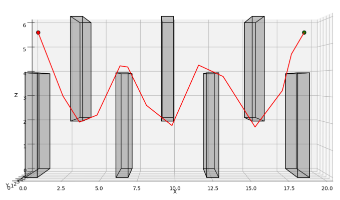
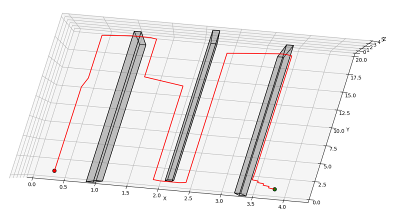
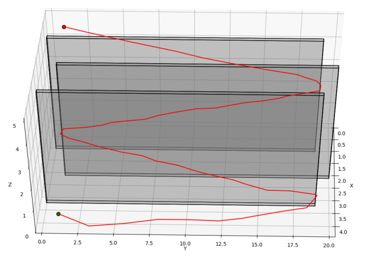

(b) RRT* algorithm

Fig. 3. Flappy bird environment

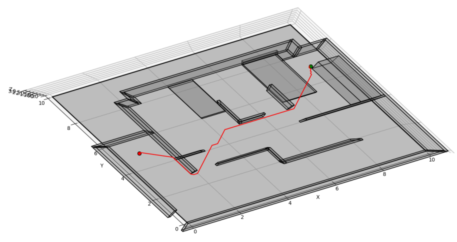


(a) A* algorithm

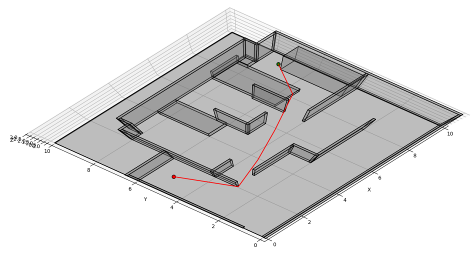


(b) RRT* algorithm

Fig. 4. Monza environment

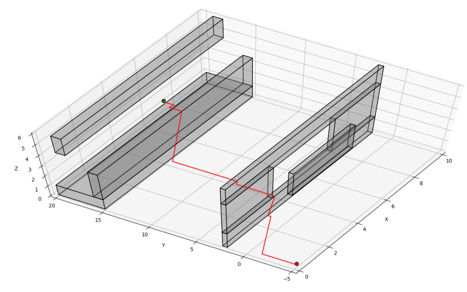


(a) A* algorithm

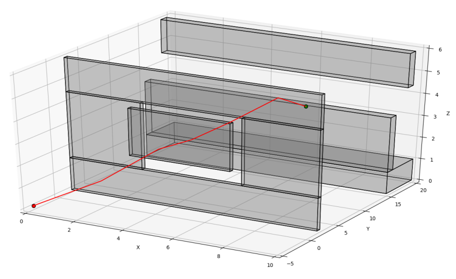


(b) RRT* algorithm

Fig. 5. Room environment

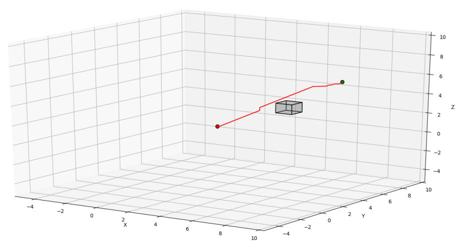


(a) A* algorithm



(b) RRT* algorithm

Fig. 6. Window environment


(a) A* algorithm

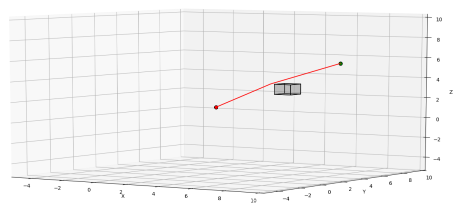

(b) RRT* algorithm

Fig. 7. Single Cube environment

C. Discussion

Quality of Computed paths: It can be seen clearly from the figures that RRT* finds better paths than A* algorithm. This is due the fact that RRT* works on infinite $\mathbb{R}^3$ space whereas A* works on discretized finite $\mathbb{R}^3$ space. If we go on decreasing the size of voxels, that is more vertices in our graph, A* will start to give better paths. For RRT*, if it is allowed to run for a longer time, it gives better path. Ideal path (most optimized) would be given at $t = \infty$.

Numbe of Considered nodes: In almost all cases RRT* uses very very less nodes as compared to A*. This is due to the fact that in A*, nodes need to consecutive voxels, which is not the case in RRT*. In RRT*, random samples are considered and hence it is highly likely that it will require fewer vertices or nodes as compared to A*. The Monza environment however is kind of a worst case scenario for RRT*, where it uses way too many nodes as compared to A* and also too much time as compared to A*. Very thin passages prohibit effiecient random sampling and hence result in using up too may nodes in graph.

Choice of heuristic: Best heuristic is chosen for A*, i.e. true distance (euclidean) between two nodes. Consistent and admissible heuristic is used, and hence performs quite well in all the environments, i.e. the computation time never explodes and is reasonable.

Interesting details: The most interesting thing is poor performance of RRT* than A* in Monza environment. This is an interesting drawback of RRT*, passing through narrow passages which is handled quite nicely in search based algorithms like A*.

Effect of Parameters: The setRange() parameter in OMPL RRT* greatly influences the run-time of the algorithm. It represents the maximum length of a motion to be added in the tree of motions. By trail and error tuning, parameter when set to 1.0 was found to reduce the run-time by over 50%. By increasing the ϵ value in A* algorithm, some environments like window provided results in lesser time while environments like monza took really more time. This may be due to the fact that higher ϵ value results in getting stuck in large local minima. In tower environment, path changed dramatically when $\epsilon = 10$ is used (Refer Fig. 8). The path is still valid though with path length 34 found in 4.7s with 6380 nodes visited. It found the path quickly but is not optimal path. Hence, $\epsilon = 1$ is preferred.

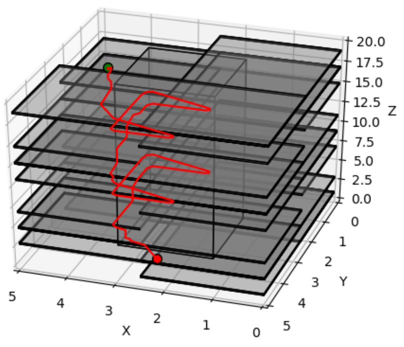


Fig. 8. Tower with $\epsilon = 10$

V. Conclusion and Future Work

Both the algorithms work as expected.

Other variations of RRT like Informed-RRT*, RRTConnect etc can be tried and tested to benchmark which algorithms perform better in which scenarios.

Acknowledgment

I thank Mr. Shiladitya Biswas, Mr. Ayon Biswas for collaborating with me on this project. We held many whiteboard discussion sessions.

References

[1] Pyrr python mathematical library. https://pyrr.readthedocs.io/en/latest/index.html. Accessed: 2020-05-14.

[2] PQDict a priority queue dictionary. https://pypi.org/project/pqdict/. Accessed: 2020-05-14.

[3] Xu Zhang, Felix Lütteke, Christian Ziegler, and Jörg Franke. Self-learning rrt* algorithm for mobile robot motion planning in complex environments. In Emanuele Menegatti, Nathan Michael, Karsten Berns, and Hiroaki Yamaguchi, editors, *Intelligent Autonomous Systems 13*, pages 57–69, Cham, 2016. Springer International Publishing.

[4] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, December 2012. http://ompl.kavrakilab.org.