Motion Planning

ECE 276B, Project-2, 05-24-2023

Renu Krishna Gutta
PID: A59018210
Department of ECE
UC San Diego
rgutta@ucsd.edu

I. Introduction

The goal of this project is to find a short and safe path to follow in a given 3D space with obstacles. The obstacles here are assumed to be Axis Aligned Bounded Boxes(AABBs). This has many real-life applications such as autonomous vehicles and other path finding and obstacle avoidance problems. The Dynamic Programming approach used in the Project-1 is computationally intensive for real life work-spaces which can be huge. In this project, we employ A* algorithm, which is a search-based label correcting algorithm, and Rapidly-exploring Random Tree (RRT) algorithm which is a sampling based algorithm. Both these algorithms do not require to visit all the nodes to find an optimal path and hence are computationally viable.

II. Problem Statement

We aim to solve a deterministic shortest path problem in a continuous 3-D Euclidean space. The obstacles are Axis-Aligned Bounding Boxes (AABB). We assume the robot is a point object. As the 3-D space is not changing during the robot motion, we can consider it to be the configuration space where we employ path search algorithms.

3-D Environment:
Boundary and obstacles are described by specifying lower left corner $(x_{min}, y_{min}, z_{min})$ and upper right corner $(x_{max}, y_{max}, z_{max})$ for each.
That is, $[x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}]^T \in \mathbb{R}^9$

Deterministic Shortest Path (DSP) problem:
Vertex/Node set: $\mathcal{V} := \mathbb{R}^3$ (3-D Euclidean space)
Edge set: $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$
Edge weights: $c_{ij}, \quad (i,j) \in \mathcal{E}$

$$c_{ij} = \begin{cases} ||i-j|| & \text{if node } j \text{ is in collision-free space} \\ \infty & \text{otherwise} \end{cases}$$

Start Node: $\mathbf{x_s} \in \mathcal{V}$
Goal Node: $\mathbf{x}_\tau \in \mathcal{V}$
Path: $i_{1:q} := (i_1, i_2, \cdots, i_q), \quad i_k \in \mathcal{V}$
Pathlength: $J^{i_{1:q}} = \sum_{k=1}^{q-1} c_{i_k, i_{k+1}}$
Paths from $\mathbf{x_s}$ to $\mathbf{x}_\tau$: $\mathcal{P}_{s,\tau} := \{i_{1:q} | i_k \in \mathcal{V}, i_1 = \mathbf{x_s}, i_q = \mathbf{x}_\tau\}$
Objective:

$$dist(\mathbf{x_s}, \mathbf{x}_\tau) = \min_{i_{1:q} \in \mathcal{P}_{s,\tau}} J^{i_{1:q}}, \quad i^*_{1:q} = \arg \min_{i_{1:q} \in \mathcal{P}_{s,\tau}} J^{i_{1:q}}$$

Assumption: $J^{i_{1:q}} \geq 0, \quad \forall i_{1:q} \in \mathcal{P}_{i,i}, \quad \forall i \in \mathcal{V}$

III. Technical Approach

A. Search-based planning: A* algorithm

Build graph:

- *Node set*: $(\mathcal{V})$
  Divide the given 3-D space into grid blocks(cubes) of a chosen resolution ($res = 0.5$). Assign integer coordinates for each grid block starting with $(0,0,0)$ for the lower left corner of the boundary. That is, there will be integer coordinates assigned to the lower left corner of every grid block. Every grid block is a member of the node set.
- *Edge set*: $(\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V})$
  With the assigned integer coordinates, each grid block is now a unit size cube. Every cube has 26 children that share one of the (6 faces + 8 vertices + 12 edges). We can assign the edges from current grid to its children.
- *Edge weight*: $(c_{ij}, \quad (i,j) \in \mathcal{E})$
  Edge weight/cost is the euclidean distance calculated from the current block i to neighbour block j

Implementation details:
Nodes were stored in a dictionary with the grid coordinates as the key value. I implemented the algorithm in an interleaved graph construction and search manner. That is, I began with start node and looked into each of its children locations. If that location is allowed for the robot to enter, i.e., with finite edge cost, I generated the node, added to the dictionary, and then proceeded with cost update and later steps. The heuristic (h) used here was Euclidean distance between the current node and the goal node.
Infinite cost edge: It is the line segment from one node location to its child location in the 3-D space, which crosses the boundary or collides with at least one obstacle.
Collision checking: The objective is to check if a line segment intersects an AABB, which is a cuboid in the 3-D space whose faces are parallel to the planes formed by the X-Y-Z axes. The basic idea is to apply the line segment - plane

Algorithm 1 Search-based planning: Weighted A*

$OPEN \leftarrow \{\mathbf{x_s}\}$, $CLOSED \leftarrow \{\}$, $\epsilon \geq 1$
$g_s = 0$, $g_i = \infty$ for all $i \in \mathcal{V} \backslash \{\mathbf{x_s}\}$
while $\mathbf{x}_\tau \notin CLOSED$ **do**
  Remove i with smallest $f_i = g_i + \epsilon h_i$ from $OPEN$
  Insert i into $CLOSED$
  for $j \in$ Children(i) and $j \notin CLOSED$ **do**
    if $g_j > (g_i + c_{ij})$ **then**
      $g_j \leftarrow (g_i + c_{ij})$
      $Parent(\mathrm{j}) \leftarrow i$
      if $j \in OPEN$ **then**
        Update priority of j
      else
        $OPEN \leftarrow OPEN \cup \{j\}$
      end if
    end if
  end for
end while

intersection test to all the six faces of the AABB. To check line intersection with a plane, we write the parametric line equation with the known two end points and then solve with plane equation to find an intersection. If there is no point on the line segment found intersecting any of the six faces, we conclude it as no collision. Concept and code for this was referred from [1].

(Sub)optimalilty: The weight term ($\epsilon \geq 1$) multipled to the heuristic will give us ϵ-optimal solution. When $\epsilon = 1$ the solution is optimal, otherwise it is sub-optimal.
Completeness: The algorithm terminated when the goal node goes into the CLOSED state. But for ensuring completeness, I added another exit condition of OPEN queue becoming empty. When no path could be found to the goal, the algorithm will exit when the OPEN list is empty and return a null path.
Memory: Since we are generating nodes on the fly, interleaved with the search, we are saving memory by storing only those nodes that will be viewed.

B. Sampling-based planning: RRT algorithm

Build graph-RRT:
In sampling-based planning we generate the graph by random sampling in the free space. We call this random graph as the Probabilistic Roadmap. The algorithm RRT generates a sparse tree where each node has only one child, hence the number of nodes is much less than that in the A* algorithm.

Implementation details:
A Python-based implementation of the RRT and search algorithm was sourced from [2].

- Choosing n: Maximum of random samples to take. This is chosen to be 100×1024 to give sufficient time for the algorithm to reach the goal in various environments.

Algorithm 2 Sample-based planning: RRT - To build probabilistic roadmap (graph)

$\mathcal{V} \leftarrow \{\mathbf{x_s}\}$; $\mathcal{E} \leftarrow \emptyset$
for $i = 1 \cdots n$ **do**
  $\mathbf{x_{rand}} \leftarrow SAMPLEFREE()$
  $\mathbf{x_{nearest}} \leftarrow NEAREST((\mathcal{V}, \mathcal{E}), \mathbf{x_{rand}})$
  $\mathbf{x_{new}} \leftarrow STEER_\epsilon(\mathbf{x_{nearest}}, \mathbf{x_{rand}})$
  if $COLLISIONFREE(\mathbf{x_{nearest}}, \mathbf{x_{new}})$ **then**
    $\mathcal{V} \leftarrow \mathcal{V} \cup \{\mathbf{x_{new}}\}$; $\mathcal{E} \leftarrow \mathcal{E} \cup \{(\mathbf{x_{nearest}}, \mathbf{x_{new}})\}$
  end if
end for
return $G = (\mathcal{V}, \mathcal{E})$

TABLE I: Primitive procedures in RRT

Procedure	Description	Source code
$SAMPLEFREE()$	Returns a random sample from free space	Ref. [2]
$NEAREST((\mathcal{V}, \mathcal{E}), \mathbf{x})$	Returns a vertex $v \in \mathcal{V}$ that is closest to $\mathbf{x}$	Ref. [2]
$STEER_\epsilon(\mathbf{x}, \mathbf{y})$	Returns a point $\mathbf{z}$ from spce, that minimizes $\|\|\mathbf{z} - \mathbf{y}\|\|$ while remaining within ϵ from $\mathbf{x}$	Ref. [2]
$COLLISIONFREE(\mathbf{x}, \mathbf{y})$	Returns $TRUE$ if the line segment between $\mathbf{x}$ and $\mathbf{y}$ lies in the free space and $FALSE$ otherwise	Used the same collision check as in A*, Ref. [1]

- Choosing ϵ for $STEER_\epsilon$: We can call this ϵ as the edge length to traverse from current node to the next valid random node. I gave a list of values $\epsilon = [8, 4, 2, 1]$. The algorithm will repeat these edge lengths sequentially as it samples points randomly. The reason for this is to allow the algorithm to take smaller steps around a narrow passage so that it can find a point to pass through, much faster.
- Collision checking: In the original source code, the collision checking procedure samples the line segment into discrete points at resolution given as input parameter. As a result one have to manually tune the resolution so that the collision checking is fail-proof for all the environments. To avoid this hassle, I replaced with the procedure based on using continuous line and plane equations to check intersections. This was the same one used in A* and was referred from [1].
- Termination - reaching goal: With a certain tunable probability, we occasionally choose the goal $\mathbf{x}_\tau$ as the sample and see if it gets connected to the tree. If yes, we terminate and output the path from $\mathbf{x_s}$ to $\mathbf{x}_\tau$, else we continue with random sampling. Once, the maximum sample limit n is reached, algorithm terminates returning no path.

IV. Results

- The paths generated for both A* and RRT algorithms are shown in Fig. 2 and Fig. 3 in the end.

TABLE II: A* results $\epsilon = 1$

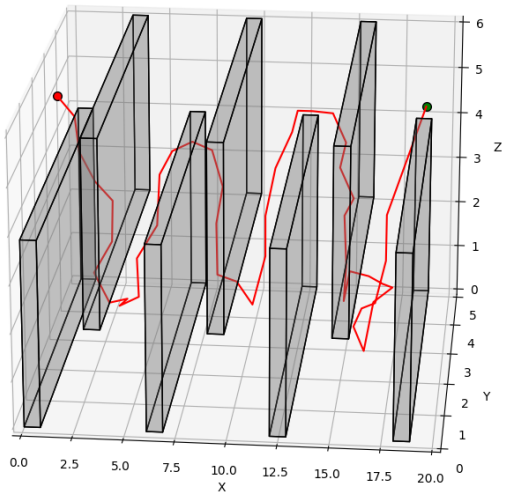
Map environment	Path length	Time taken (s)
Single cube	8	0.05
Maze	79	13.3
Flappy bird	25	2.5
Monza	77	1.43
Window	27	3.39
Tower	32	3.67
Room	12	0.52

TABLE III: RRT results

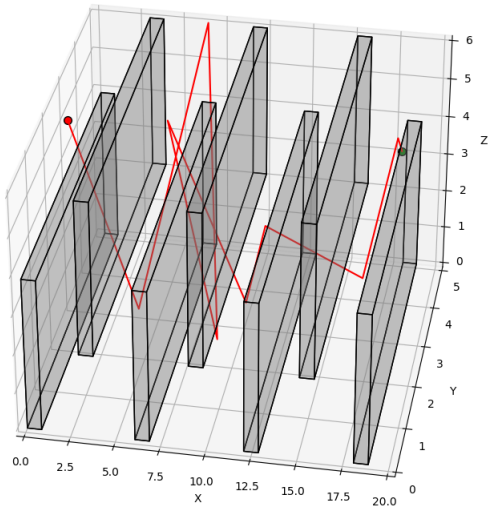
Map environment	Path length	Time taken (s)
Single cube	17	0.005
Maze	126	1.84
Flappy bird	52	0.05
Monza	132	1.99
Window	40	0.01
Tower	56	0.12
Room	24	0.03

- Effect of ϵ in A*: Usually $\epsilon \geq 1$. When $\epsilon > 1$, we call it weigthed A* algorithm. By increasing ϵ we are giving more weightage to the heuristic value of the node to decide whether to send into OPEN queue or not. That is we are emphasizing more on reducing the path's cost found so far. This narrows down the search space and the time taken to reach the goal reduces. For example, for the $maze$ map, $\epsilon = 1$ took 13.3 sec to find a path of length 81, whereas $\epsilon = 5$ took 9.7 sec to find a path of length 79. However, increasing the ϵ too much can render the heuristic to be non-admissible and we may end up with wrong solutions.
- Effect of resolution in A*: When resolution is very low like $res = 1$, for most of the environments, I am not able to find a path to the goal. This is due to loss connectivity to the goal. Due to low resolution, the narrow passages will no longer be collision-free and hence the connectivity loss. $res = 0.5$ is the sweet spot for the A* algorithm to end faster. When $res = 0.1$, the search is taking a very long time to reach the goal. This is due to more nodes being added into the OPEN queue and searching through all these nodes takes a lot of time.

- Coming to RRT, the search ended much faster due to random sampling and sparse graph generation. But the path lengths are higher than those in A*. The vanilla RRT algorithm prioritizes speed over optimality.
- Effect of edge length (ϵ of $STEER_\epsilon$) in RRT: For large edge lengths, the algorithm will take larger steps. Taking larger steps can either let you reach the goal faster or keep oscillating in a small region due to not being able to pass through narrow passages increasing the running time. So larger edge length is helpful when the space is mostly free of obstacles and the goal is very far from the start.
  Example:
  For edge length = $1 \implies manza$ map took 8 seconds to find a path of length 110.
  For edge length = $8 \implies manza$ map took 5 seconds to find a path of length 181.
  But for edge length = 12, the algorithm timed out and never reached the goal. Hence, sufficiently large edge lengths can get you to the goal faster.
  For smaller edge lengths, though being slower, they give smoother paths to the goal. Whereas for larger edge lengths the paths are not smooth. Comparison shown in Fig. 1
- Alternatives to RRT: Algorithms like RRT* apply A* on the obtained tree. A* rewires the edge connections and gives a more optimal and smoother path. Another algorithm called RRT connect simultaneously builds two trees starting from start and goal nodes. However this is not so feasible in complex or cluttered environments.



(a) Flappy bird, , edge length=8



(b) Flappy bird, edge length=8

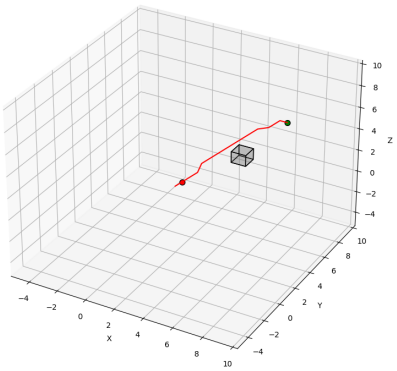
Fig. 1: Comparing different edge lengths in RRT

V. Acknowledgment

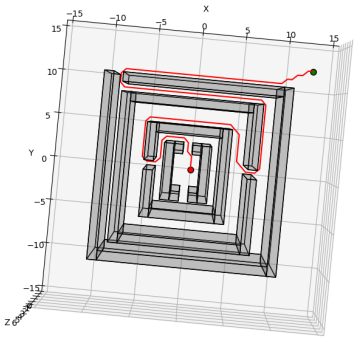
I would like to thank Prof. Nikolay Atanasov and TA – Zhirui Dai for guiding me with this project.

References

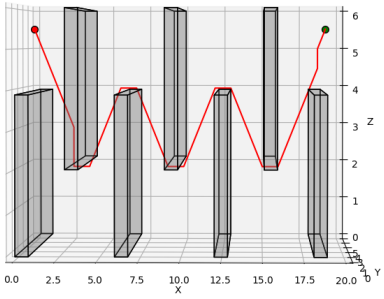
[1] "Intersection Tests in 2D", noonat.github.io/intersect/#aabb-vs-segment.
[2] "Motion Planning RRT Algorithms", github.com/motion-planning/rrt-algorithms.
[3] natanaso.github.io/ece276b/



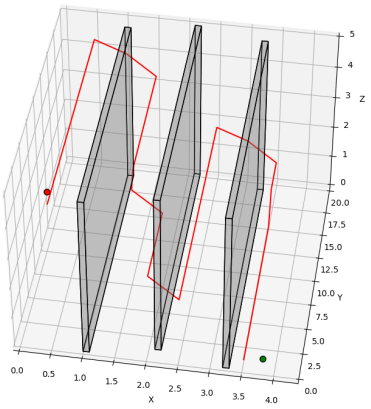
(a) Single cube



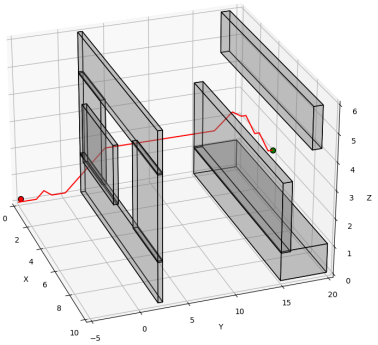
(b) Maze



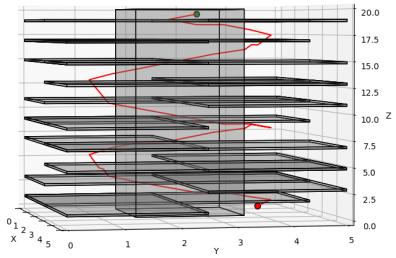
(c) Flappy bird



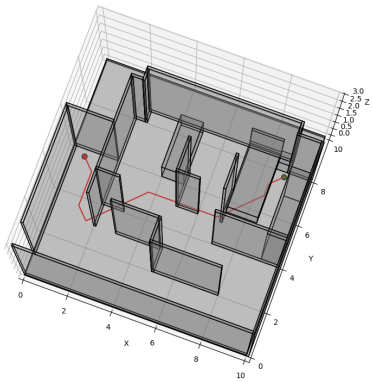
(d) Monza



(e) Window

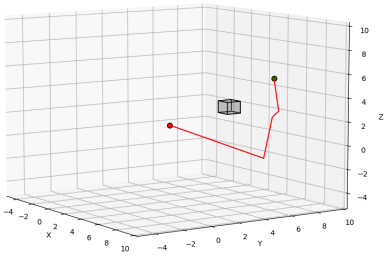


(f) Tower

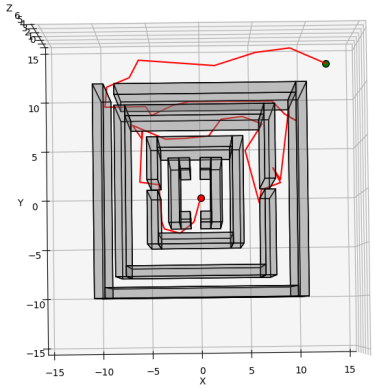


(g) Room

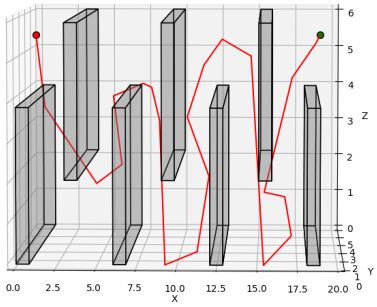
Fig. 2: Paths generated by A* algorithm



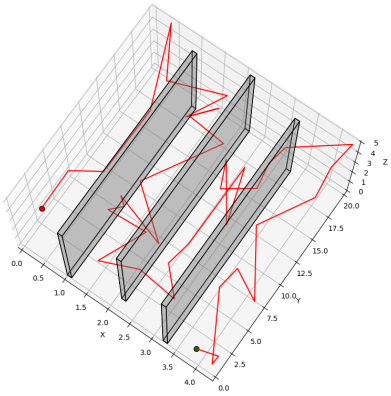
(a) Single cube



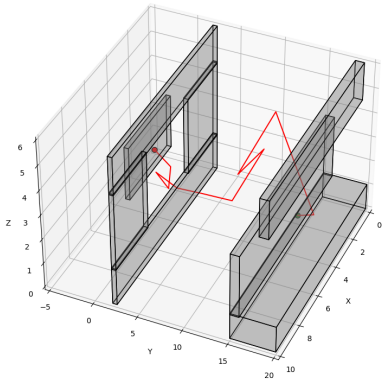
(b) Maze



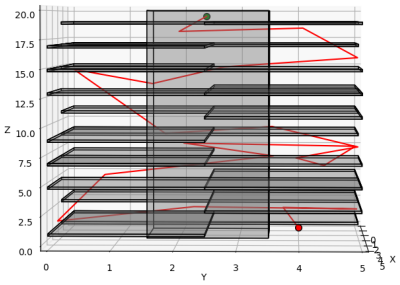
(c) Flappy bird



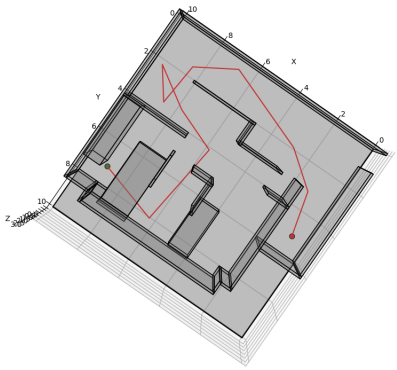
(d) Monza



(e) Window



(f) Tower



(g) Room

Fig. 3: Paths generated by RRT algorithm