ECE276B Project 2: Motion Planning

Mustafa Shaikh
Department of Electrical and Computer Engineering
University of California, San Diego
La Jolla, U.S.A

I. Introduction

As the role of autonomous robotic systems in our society expands, and these systems take on more varied tasks, the problem of autonomous navigation becomes ever more important. Autonomous navigation relates to the task of planning the optimal trajectory for agent to reach a desired goal state, while avoiding obstacles and constraints along the way. This has many uses in areas such as autonomous vehicles, where a car must reach its destination but adhere to traffic stops and avoid pedestrians. There are numerous other applications of this task such as in agriculture, manufacturing and military related applications [4].

There are several approaches to solving the motion planning problem, most of which fit into categories such as search-based methods and sampling-based methods. All approaches rely on an appropriate of representation of the environment. Search-based methods include discretization of the environment using a variety of methods such as cell decomposition or lattice-based methods, in which a graph is constructed for all nodes in the environment. In relatively simple environments with simple agent actions such as in our case, cell decomposition methods can work well, although they can be slower as the resolution of the grid gets finer, and so we also experiment with sampling-based planning methods. These methods do not create a graph using every single possible node, but rather sample certain nodes in the environment to reduce the computational load. These methods can provide sub-optimal solutions at the gain of reduced computation time [5].

In this project, the task is to generate the optimal (in this case shortest) path from the initial point to the goal location while considering obstacles in the environment.

II. Problem Formulation

Motion planning is a deterministic shortest path problem (herein referred to as DSP) [1]. In this class of problems, the objective is to find an optimal path from a given start position to a desired end position. The optimal path is a path such that it has the lowest cost (equivalently, shortest distance to target) among all other possible paths. In general, the cost associated with a path can be a a function of any variable relevant to the domain, such as volume of traffic on a road system.

In order to find such a path, we formulate the possible positions in an environment as a graph G with vertex set V and edges $\epsilon \subseteq VXV$, representing possible connections between locations. Weights between edges represent the cost $C = \{c_{ij} \in \mathbb{R} \cup \{\infty\} | (i,j) \in \epsilon\}$, where c_{ij} represents the cost from vertex i to vertex j. Note it is important to include ∞ in the set that the costs are drawn from, as an infinite cost between nodes i and j represents the idea that node j is not reachable from node i. We define a path between nodes 1 and q as $i_{1:q} = (i_1, i_2, ..., i_q)$ [3].

Formally, the optimal path from start node s to target node τ is given by the path that solves the following optimization problem [3]:

$$dist(s,\tau) = \min_{i_{1:q} \in P_{s,t}} J^{i_{1:q}}, \tag{1}$$

where $P_{s,t}$ is the set of all paths from node s to node τ, and $J^{i_{1:q}} = \sum_{k=1}^{q-1} c_{i_k, i_{k+1}}$ is the length of the path (i.e. the cost of travelling along all edges in the path). A key assumption in this formulation is that there are no negative cycles in the graph, i.e. that there is no sequence of nodes that form a closed loop in which travelling along those nodes repeatedly reduces the cost till $-\infty$. Note that a DSP problem may also be equivalently formulated as a Deterministic Optimal Control problem. This idea is briefly discussed in the next section.

1) Environment: Now that the underlying structure of the DSP framework has been presented, we can discuss the environment in our problem. We work with a 3-D grid in which a start node and desired goal node is given. The environments contain obstacles, which are assumed to be Axis Aligned Bounding Boxes (AABBs) [6]. AABBs are rectangular parallelepipeds that are aligned with the basis vectors of $\mathbb{R}^3$. We assume that these obstacles must be avoided when planning the path from start to goal, i.e. the optimal path may not pass through any obstacle. The assumption that the obstacles are AABBs provides simplifications in collision checking, which will be discussed in the next section. Transferring our graph formulation above, each position in the 3-D environment is considered a node, with edges connecting positions that can be reached from each other. The cost of travelling along an edge between two nodes is simply the Euclidean distance between the two nodes, that is, $c_{ij} = ||p_j - p_i||_2$, where p_i, p_j are the cartesian coordinates of node i and j respectively. Note that the obstacles are also considered nodes, but the cost of travelling from a node to an obstacle node is ∞, that is, $c_{i,o} = \infty$, where i is a given node and o is an obstacle node.

III. Technical Approach

Now that the problem has been formulated as a DSP, we present our approach. The reason we consider this problem as a DSP problem rather than an Optimal Control problem, is that algorithms to solve optimal control problems formulated

as MDPs, such as Dynamic Programming, computes the shortest path from all nodes to the goal. However, most nodes are not on the shortest path from the start node to the goal node. Therefore, a lot of unnecessary computation is performed. Hence, we use a class of algorithms known as Label Correcting (LC) algorithms. In the search-based planning portion, we use a particular case of an LC algorithm called the Weighted A* algorithm, a popular motion planning algorithm of which many variants exist [2]. For the sampling-based planning portion, we use the RRT algorithm. We first discuss collision checking, then we present the search-based planning approach, followed by sampling-based methods.

A. Discretization

The first step is to discretize the environment. In this project, we discretize the 3-D grid using cell decomposition. We use a resolution of 0.5, i.e. at each cell i, the neighbouring cells can at most have a 0.5 difference in any of the (x, y, z) coordinates. For example, starting at $(0, 0, 0)$, the largest step an agent can take is to $(0.5, 0.5, 0.5)$, but not e.g. to $(0.5, 0.5, 1)$. We also assume that each cell has 26 neighbours, i.e. any adjacent cell in 3-D space.

B. Collision Checking

As mentioned above, the optimal path must avoid all obstacles. To achieve this we use a vectorized line approach. At every iteration, when a path is extended from a node i to a neighbouring node j, we check the following two conditions:

- Endpoints not within obstacle limits: For each obstacle, we check that the (x, y, z) coordinates of both the start and end point of the line connecting node i and j do not fall inside the obstacle limits.
- Any segment of line not within obstacle limits: We obtain the direction vector from node i to node j, and step along this direction starting from node i, until we reach node j. We tune this step parameter, but generally choose is to be 5x smaller than the resolution in order to ensure we capture collisions. At each point along the line, we simply check whether that point is within obstacle limits. This approach allows us to treat line collision checking as point collision checking which is conceptually simple to implement.

Figure 1 illustrates a basic example of collision checking.

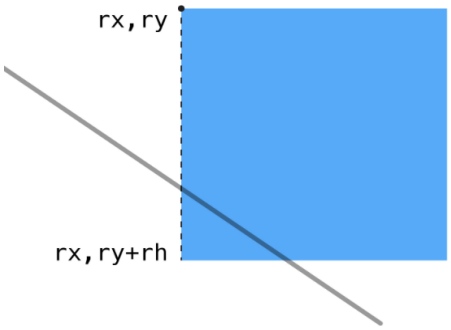

Fig. 1. Visualization of a line intersecting an obstacle; while the endpoints are not within the obstacle, a segment of the line is and therefore the path collides with the obstacle [7]

C. Weighted A*

We now present the weighted A* algorithm, its key components, and a theoretical discussion of its optimality guarantees. Weighted A* is a variant of Dijsktra's algorithm [2] which uses a heuristic function to guide the node expansion towards the goal. It is an example of a search-based planning algorithm which systematically discretizes the environment and constructs a dense graph.

1) Heuristic: The heuristic function must underestimate the distance from any node to the goal. In precise terms, it must be admissible in order to guarantee optimality, i.e. that $0 \leq h_j \leq dist(j, \tau)$, where h_j is the value of the heuristic at node j, and $dist(j, \tau)$ is the true shortest distance from node j to τ [2]. The heuristic serves to narrow down the number of expansions considered; it only considers a node for which the sum of the shortest path to the node from start, plus the estimated path to goal, is less than a previously known shortest path to the goal. The more accurately the heuristic estimates the true shortest distance from a node to the goal, the faster the algorithm converges to the optimal path. In our implementation, we use the commonly used Euclidean distance heuristic. In a grid environment, Euclidean distance is always less than or equal to the distance to the goal.

2) Completeness and Optimality Conditions: A* provides optimality guarantees under certain conditions. If the heuristic is also consistent, i.e. that $h_i \leq c_{ij} + h_j$ for $j \in children(i)$, then A* is guaranteed to find the optimal path if one exists [2]. If the heuristic is ϵ-consistent, i.e. that $h_i \leq c_{ij} + \epsilon h_j$ for $j \in children(i)$, then A* is guaranteed to find an ϵ-suboptimal path with cost $dist(s, \tau) \leq g_{tau} \leq \epsilon dist(s, \tau)$ for $\epsilon \geq 1$. In weighted A*, the value for ϵ is chosen to be greater than 1, and this returns a suboptimal path, but the computation time can be considerably faster as this method biases the node expansions closer to the goal.

3) Implementation Details: A key point to note is that the graph is constructed on the fly. This means that the entire set of possible nodes in the environment is not added to the graph at the beginning. Rather, as the algorithm expands nodes, it creates nodes and stores them in the graph. This serves to

Data: Start node s, Goal node τ, Heuristic h
Result: Optimal path from start s to goal τ
OPEN $\leftarrow \{s\}$, CLOSED $\leftarrow \{\}$, Parent $\leftarrow \{\}, \epsilon \geq 1$
$g_s \leftarrow 0$, $g_i \leftarrow \infty \quad \forall i \in V \setminus \{s\}$
while $\tau \notin$ *CLOSED* **do**
  OPEN.pop(i) where $i = \text{argmin}_i(g_i + \epsilon h_i)$
  Insert i into CLOSED
  if *dist*$(i, \tau) \leq$ *GOAL THRESHOLD* **then**
    break
  end
  for $j \in$ *Children(i)* **do**
    if *j not in CLOSED* **then**
      edge cost $= g_i + c_{ij}$
      if $g_j >$ *edge cost* **then**
        $g_j \leftarrow edgecost$
        Parent(j) $\leftarrow i$
        if $j \in$ *OPEN* **then**
          Update priority of j
        end
        else
          OPEN $\leftarrow$ OPEN $\cup\{j\}$
        end
      end
    end
  end
end

Algorithm 1: A* Algorithm

reduce the memory requirements of the implementation. The OPEN list is implemented using a priority queue. This data structure is used for breadth-first searches (A* is one type of a breadth first search algorithm). It is used specifically for when each node in the tree has a value for which we wish to access the minimum value of at any given time. The underlying min heap brings the node with the minimum value (in our case f value), to the root, from where it can be popped at each iteration. The CLOSED list is simply implemented as a set, as there should be no removals or re-adding to the CLOSED list once a node enters it. This holds for admissible, consistent heuristics and if the edge costs are always non-negative, all of which apply to our case. The variable GOAL THRESHOLD above is set to $0.5 * resolution$. This means that once the coordinate of a node is within $0.5 * resolution$ distance of the goal node, we terminate the algorithm. This was required as having a resolution of 0.5 meant that the algorithm could not always exactly reach the goal node as the coordinates of candidate neighbours only changed in increments of 0.5, so if the goal was only 0.1 away from a node, the goal may not be exactly reachable from that node.

D. Goal-biased RRT

RRT is a sampling-based motion planning algorithm that randomly samples a node in the free space in the environment, finds the closest point to in the graph of nodes, and expands the graph in the direction of this randomly sampled point. In Goal-biased RRT, we apply a simple modification in order to achieve faster convergence. Rather than expanding the graph in random directions (as influenced by the uniform random sampling of a node in the environment), every few iterations we sample the goal node as the 'random' node, thereby influencing the algorithm to create paths in the direction of the goal node. Without this modification, RRT can take considerably longer to converge as the random sampling simply may not sample a node near the goal across many iterations. This difference will be shown in the next section.

Data: Start node s, Goal node τ, Step size ϵ
Result: Path from start s to goal τ
while *True* **do**
  if $iterationnum\%10 = 0$ **then**
    $x_{candidate} = \tau$
    if $iterationnum\%200 = 0$ **then**
      $x_{candidate} \leftarrow GetNearest(\tau, G)$
      if *dist*$(x_{candidate}, \tau) \leq \epsilon$ **then**
        break
      end
    end
  end
  else
    $x_{candidate} \leftarrow SampleFree()$
  end
  $x_{nearest} \leftarrow GetNearest(G, x_{candidate})$
  $x_{new} \leftarrow Steer(x_{nearest}, x_{candidate}, \epsilon)$
  if *CollisionFree*$(x_{nearest}, x_{new})$ **then**
    $V \leftarrow V \cup \{x_{new}\}; E \leftarrow E \cup (x_{nearest}, x_{new})$**end**
  end

Algorithm 2: Goal-biased RRT

where SampleFree() picks a uniform random sample from the free cells in the grid, GetNearest() returns the node in the graph closest to the given node, Steer() moves a distance epsilon in the direction of $x_{candidate}$, CollisionFree() returns True if the path from one node to the next does not collide with any obstacles, and dist() returns the Euclidean distance between two nodes.

1) Theoretical Discussion - RRT: The RRT algorithm is extremely popular for a number of reasons. First, it is very simple to implement once a small number of subroutines (see algorithm above) are implemented. Second, it uses a sparse graph that requires little memory and computation; since it is not storing every single neighbour of a node in the graph, but rather is sampling a small subset of them, the memory use is much lower than for search-based planning algorithms. RRT also finds feasible paths quickly, as will be seen in the next section. Sampling based planning methods in general are also probabilistically complete, meaning that that the probability of finding a path if one exists approaches 1 as the number of iterations approaches ∞. However, the major drawback is that RRT does not provide an optimal path. In fact, it has been shown [8] that the probability of an optimal path as the

number of samples approaches ∞ is zero. There are variants of RRT which solve this particular issue, and will be discussed in the closing section.

E. Technical Details

We use a MacBook Pro with 1.4 GHz Quad-Core Intel Core i5, 8 GB 2133 MHz LPDDR3, Intel Iris Plus Graphics 645 1536 MB. We make use of the well known numpy library for data manipulation.

IV. Results

*A. Weighted A**

We now present the results of implementing the A* algorithm to each of the environments. We show plots of the optimal path, some examples of node expansion, and follow up with a discussion on these results as well as the impact of tuning parameters such as the weight and heuristic. Finally, we present performance results of A* on the various environments, including planning time, iterations needed for convergence, and cost of the optimal path.

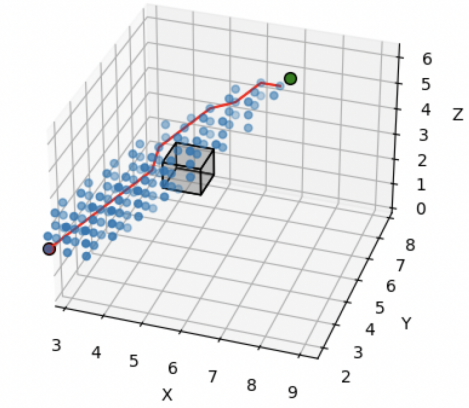

Fig. 2. Single Cube with expanded nodes

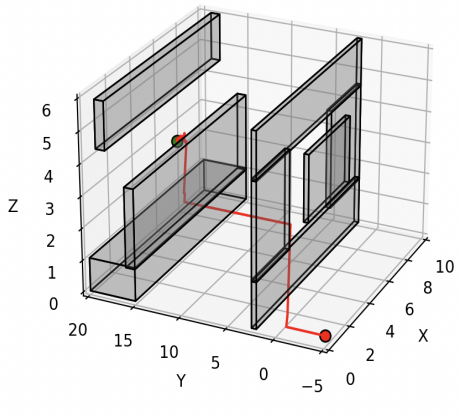

Fig. 3. Window

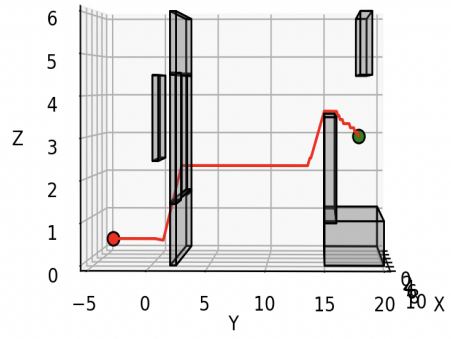

Fig. 4. Another view of the Window environment to demonstrate that the optimal path does not cut through any obstacles

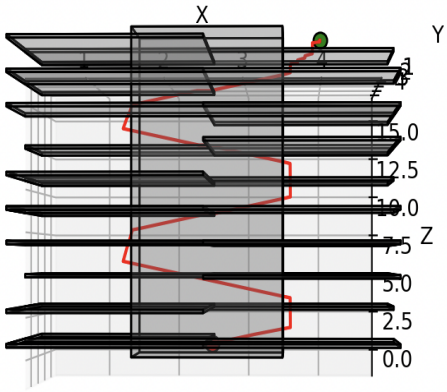

Fig. 5. Tower

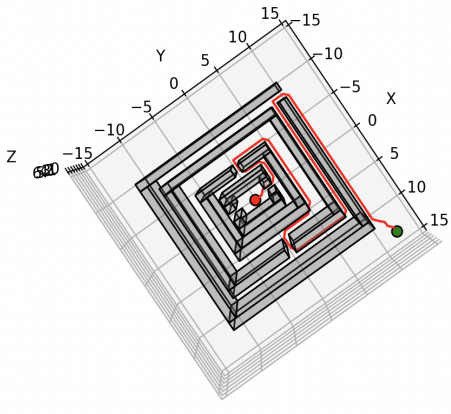

Fig. 6. Maze

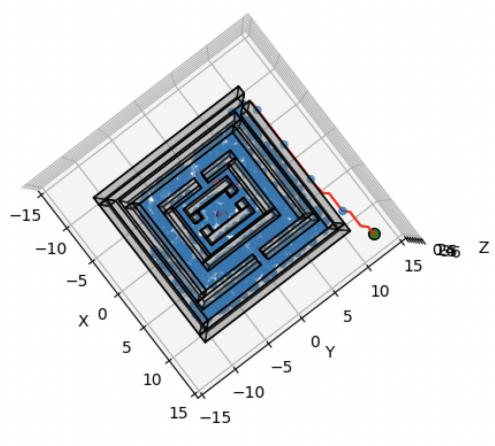

Fig. 7. Maze - plot showing expanded nodes. We see that A* expanded almost all nodes inside the maze area. This is discussed further in the Performance section below

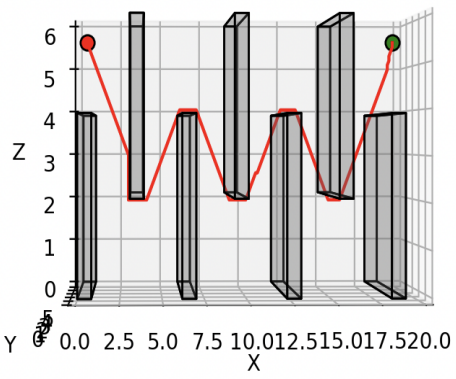

Fig. 8. Flappy Bird

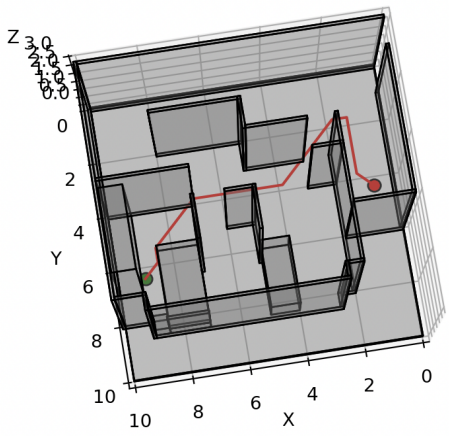

Fig. 9. Room -

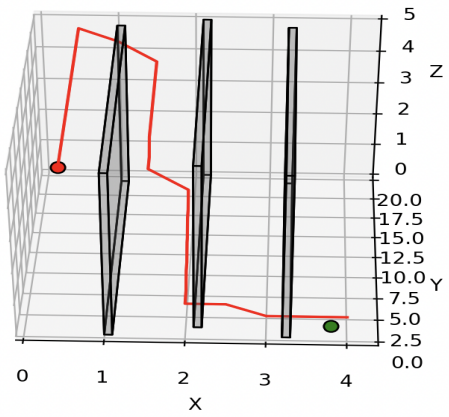

Fig. 10. Monza - an interesting case as the resolution = 0.5 and collision resolution = 0.1 meant that the optimal path cut through the obstacle as the goal was within 0.1 of the candidate node on the other side of the obstacle. We then decreased the resolution to overcome this issue

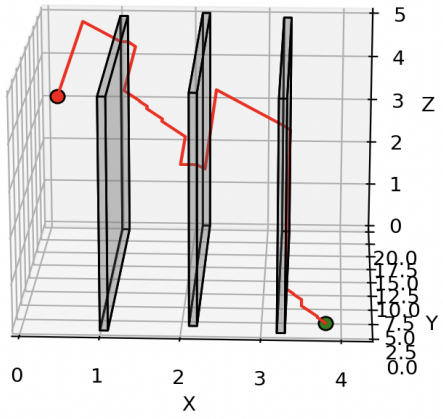

Fig. 11. Monza with resolution = 0.1, collision resolution = 0.01. As we can see, the issue of the path cutting through the obstacle has now been rectified, but at the cost of increased computation time

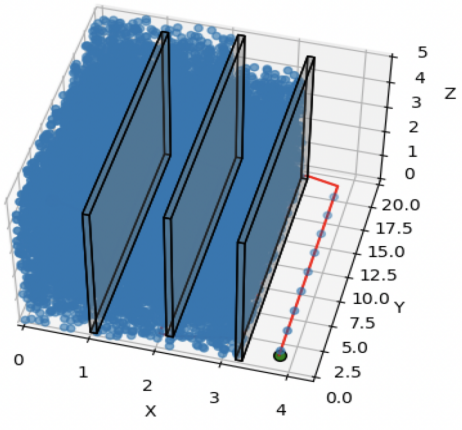

Fig. 12. Monza with resolution = 0.1, collision resolution = 0.01, with expanded nodes. We see that that A* had to expand almost all nodes in the map, and this is also shown in the table below

a) Performance: Table I shows performance measures of the A* algorithm on the various environments. We see that planning in the single cube environment is very quick and requires very few node expansions. The reason for this is that the environment is quite open (i.e. not cluttered with obstacles) and so when the heuristic guides the node expansion in the direction of the goal, those subsequent node expansions are valid in that they do not collide with obstacles. Compare this to maps with lots of obstacles, or tightly constrained spaces such as Maze and Monza, which required the expansion of many nodes. Monza is an especially interesting case, as A* expanded almost all the nodes in the graph. The reason for this is that the heuristics (both Euclidean and Manhattan) guide the node expansion in the direction of the goal i.e. to take a shortcut to the goal, however, this path is blocked by the obstacle. Therefore A* expands all the nodes in each region before moving to a new region, and finally finds the long way around the obstacle. The reason that Monza took much longer to plan is that the resolution used for Monza = 0.1 as opposed to 0.5 for the others, and the collision checking was performed at a resolution of 0.01 instead of 0.1. In general, the performance is quite good and all planning except Monza was complete within 90s with a low number of nodes expanded as a percentage of the volume of the environment.

1) Impact of weight (epsilon): Increasing the value of ϵ over 1 provides, in theory, a path that is at most ϵ times worse than the optimal. This comes at the benefit of faster computation. We experimented with the Maze environment using $\epsilon = 2$, and found that the resulting path had a cost within 30% of the cost of the optimal path, but took 60s instead of 85s, and expanded 10,000 nodes instead of 13,000. Therefore, in situations where planning time is important, a higher value of epsilon should be considered as it has shown to provide computational benefits.

2) Impact of heuristic: Selecting the Manhattan heuristic led to a very interesting outcome, namely that planning time drastically reduced and node expansions reduced as well. The cost of the optimal path is very similar, indicating that the paths found are very similar. The most likely reason for this is that the Manhattan heuristic more accurately estimates the true shortest distance between a node and the goal. The more accurate the estimate provided by the heuristic, the fewer node expansions needed and therefore the faster the convergence. Since we are in a grid environment in which motion is limited to lateral, vertical and single-step diagonal movement, it certainly appears that that Manhattan heuristic more closely represents the true distance between two nodes when travelling in this manner. Performance measures are presented in table II.

Figure 13 shows an example of the optimal path found using the Manhattan distance heuristic. The optimal path is very similar to that obtained using the Euclidean distance heuristic.

B. RRT

1) Standard RRT: We first implemented a standard version of RRT without sampling the goal node at every 10th iteration. The figures below show suboptimal paths, with many node expansions, and a long planning time (discussed further in the Performance section below).

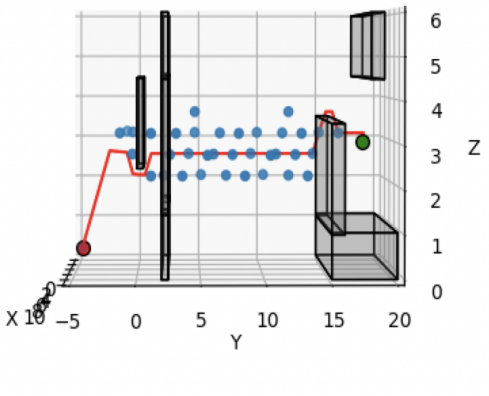

Fig. 13. Window optimal path with expanded nodes, using Manhattan distance heuristic

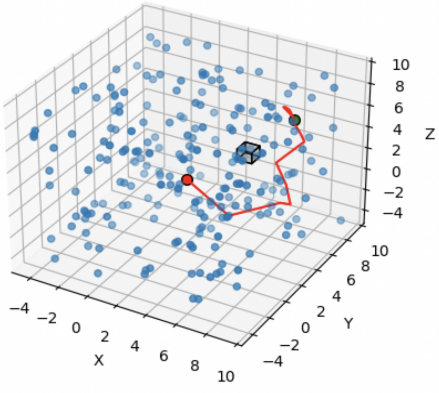

Fig. 14. Single Cube with vanilla RRT (no goal biasing). We see that RRT chooses a highly sub-optimal path with large number of 'node expansions'

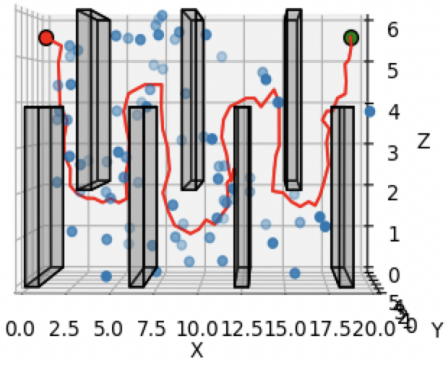

Fig. 15.

Environment	% nodes expanded	Time (s)	Path Cost
Single Cube	0.003%	0.9s	8
Window	0.3%	16s	27
Tower	0.66%	24s	34
Maze	0.23%	85s	80
Flappy Bird	0.63%	11s	26
Room	0.12%	4s	12
Monza	71%	569s	42

TABLE I
PERFORMANCE MEASURES OF A* ALGORITHM WITH EUCLIDEAN DISTANCE HEURISTIC

Environment	% nodes expanded	Time (s)	Path Cost
Single Cube	0.0003%	0.05s	7
Window	0.02%	1.8s	27
Tower	0.57%	21s	34
Maze	0.18%	70s	80
Flappy Bird	0.42%	8s	26
Room	0.05%	1.3s	13
Monza	55%	400s	44

TABLE II
PERFORMANCE MEASURES OF A* ALGORITHM WITH MANHATTAN DISTANCE HEURISTIC

2) Goal-biased RRT: We now present the paths obtained from goal-biased RRT, as well as a table of performance measures, a comparison between standard RRT and goal-biased RRT, as well as between A* and goal-biased RRT. In summary, goal-biased RRT vastly outperforms standard RRT in convergence time and cost of path, and outperforms A* in convergence time but finds paths that are of higher cost than A*.

a) Performance: In general, goal-biased RRT provides good results with fast convergence on most maps. It converges faster and with lower cost than with standard RRT, however, gets stuck in Monza and Maze, both of which are highly constrained environments. Highly constrained environments pose a challenge for sampling-based methods as there may never be a sample chosen in the direction that the path must ultimately take. Even goal-biasing does not help as any path in the direction of the goal in both Maze and Monza runs into obstacles and so will not lead to a valid path. In such environments, it may be more appropriate to choose search-based algorithms for motion planning.

Compared with standard RRT, goal-biased RRT plans much faster and with lower cost. In fact, with standard RRT, there was large variance in the planning time even for Single Cube which is a basic environment to plan in. The longest planning time obtained for RRT was 120s, and the shortest was 5s, compared with consistently $< 1s$ for goal-biased RRT. This is because in an open environment such as Single Cube, sampling the goal node and extending the tree in its direction leads to a simple straight path from start to end without the need for expanding the tree in other directions. So we can think of goal-biasing as a simple heuristic.

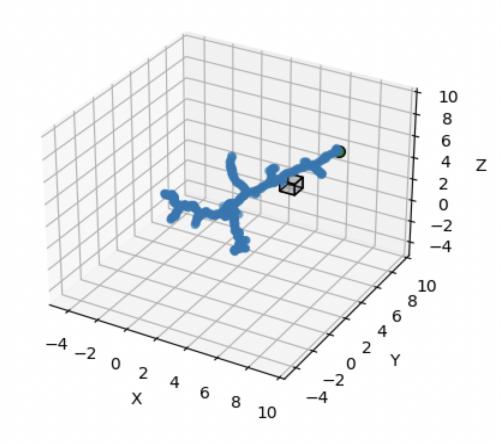

Fig. 16. Single Cube - RRT with goal biasing. Note that the candidate nodes that the tree was expanded along are tight and point in the direction of the goal. There are few nodes considered in other directions as compared with standard RRT.

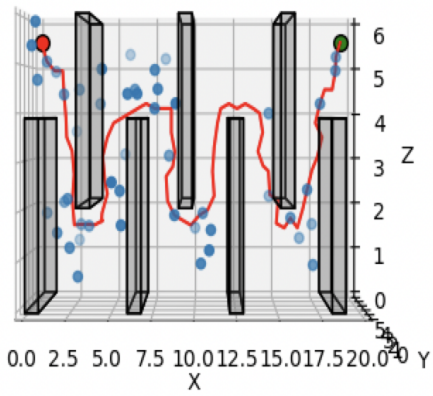

Fig. 17. Flappy bird - RRT with goal biasing.

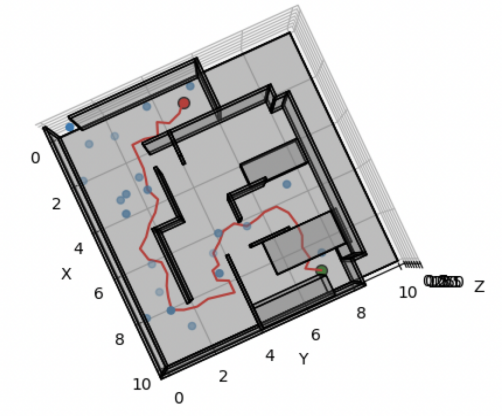

Fig. 18. Room - RRT with goal biasing.

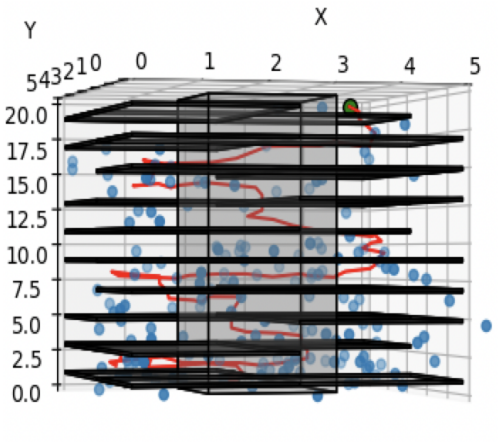

Fig. 19. Tower - RRT with goal biasing

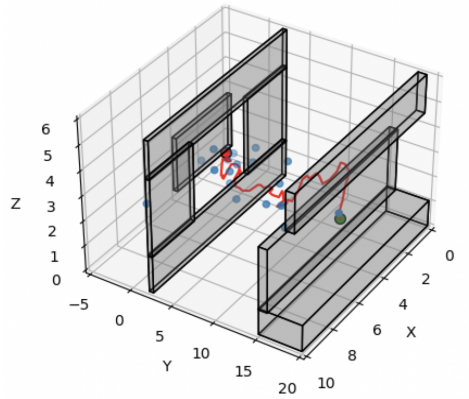

Fig. 20. Window - RRT with goal biasing. It is clear that this path is noisier and less optimal than the one obtained with A*. This is an important feature of sampling based methods compared with search based methods

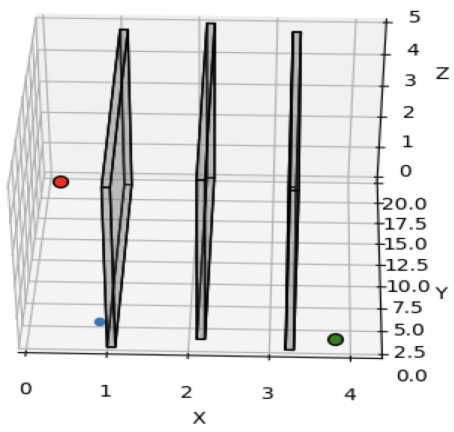

Fig. 21. Monza - RRT with goal biasing. The algorithm gets stuck at the blue dot seen near the start point (red dot). This is the closest point to the goal the algorithm was able to achieve. See the Performance section for a discussion on why this is the case

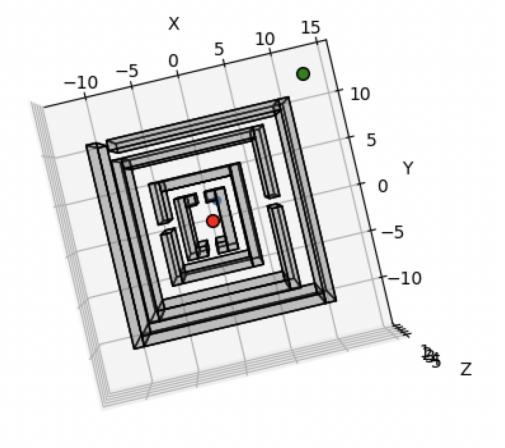

Fig. 22. Maze - RRT with goal biasing. Note that the algorithm gets stuck at the barely visible blue dot right next to the start point (red dot). This means the algorithm was not even able to create a path that gets out of the inner most section of the maze

b) Comparison of A and goal-biased RRT:* Compared with A*, goal-biased RRT provides convergence which can be an order of magnitude faster, although with slightly higher cost. For example, in the Window environment, RRT plans in 0.1 vs. 1.8s for A*, but the cost of the path is 31 for RRT vs. 27 for A*. Similarly, in the Flappy Bird environment, goal-biased RRT plans in an impressive 0.8s vs. 8s for A*. The cost for RRT is 31 vs. 26 for A*. Additionally, RRT generally considers fewer nodes in the path, and this means the implementation is more memory efficient than A*. Depending on the relative importance of finding the shortest path versus planning in the shortest amount of time and saving memory, a designer can choose between A* and RRT.

V. Conclusion

We have implemented a search-based and sampling-based motion planning algorithm in order to find a path from a starting node to a goal node in different environments. We saw that the two methods perform better in different ways and in different environment types. In the future, we will modify the RRT implementation to RRT* to incorporate label correction and there provide suboptimality guarantees. We will also experiment with more sophisticated variants of A* such as Jump Point Search, or RTAA*, that can provide better performance in some cases.

References

[1] N. A. Atanasov, “Markov Decision Processes,” in ECE 276B Lecture 3.
[2] N. A. Atanasov, “Search based motion planning,” in ECE 276B Lecture 7.
[3] N. A. Atanasov, “Deterministic Shortest Path,” in ECE 276B Lecture 5.
[4] “Motion planning,” Wikipedia, 17-Jan-2022. [Online]. Available: https://en.wikipedia.org/wiki/Motion_planning. [Accessed: 01-May-2023]
[5] N. A. Atanasov, “Sampling based motion planning,” in ECE 276B Lecture 9.
[6] 3D Collision Detection, https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection
[7] Collision Detection, https://www.jeffreythompson.org/collision-detection/
[8] S. Karaman, E. Frazzoli, Incremental Sampling based Algorithms for Optimal Motion Planning, Proceedings of Robotics: Science and Systems, 10.15607/RSS.2010.VI.034, 2010,

Environment	% nodes expanded	Time (s)	Path Cost
Single Cube	0.006%	0.2s	8
Window	0.01%	0.1s	27
Tower	0.57%	10s	34
Maze	N/A	N/A	N/A
Flappy Bird	0.17%	0.8s	36
Room	0.2%	0.36s	23
Monza	N/A	N/A	N/A

TABLE III
PERFORMANCE MEASURES OF GOAL-BIASED RRT ALGORITHM ON ALL ENVIRONMENTS