Ali Jones | SDM HW 1 | 01/27/23 pg 1

1. Search-based planning (20 pts)

(a) Consider the planning problem shown in Figure 1. Let ‘1’ be the initial state, and let ‘6’ be the goal state.

(i) By hand, use backward value iteration to determine the stationary cost-to-go (i.e, minimum cost between each state and the goal). **Show your work.**

(ii) Do the same but instead use forward value iteration. **Show your work.**

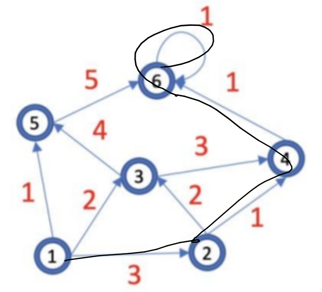


Figure 1: Six-state discrete planning problem.

1. Backward value iteration

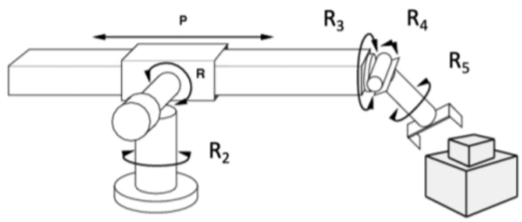
1	2	3	4	5	6	
∞	∞	∞	∞	∞	0	– cost to get to the goal in 0 steps
∞	∞	∞	1	5	1	– cost to get to the goal in 1 step
6	2	4	2	6	2	– in 2 steps
5	3	5	3	7	3	– in 3 steps
6	4	6	4	8	4	– in 4 steps

lowest cost from start → goal

2. Forward value iteration

1	2	3	4	5	6	
0	∞	∞	∞	∞	∞	– cost to get to # from 1 in 0 steps
∞	3	2	∞	1	∞	– in 1 step
∞	∞	5	4	6	6	– in 2 steps
∞	∞	∞	8	9	5	– in 3 steps
∞	∞	∞	∞	∞	6	– in 4 steps

(b) Consider the 6-DOF planning problem below. The goal is to get the end effector from a starting configuration (defined by five rotation angles R and one linear dimension P) to an end configuration (also defined in the 6D space). The 6D space is discretized, and the cost metric is defined as: $C = \Delta R + \Delta R_2 + \Delta R_3 + \Delta R_4 + \Delta R_5 + \Delta P$, where ΔR is the total angular distance travelled in the R degree of freedom, ΔP is the total linear distance travelled, etc.



Which of the following heuristics is admissible for A* search on this problem?
i. Euclidean distance in 6DOF space from start to goal
ii. Euclidean distance in 6DOF space from start to goal times 2
iii. Euclidean distance in 6DOF space from start to goal divided by 2
v. Sum of total 6DOF distance travelled by optimal solution to reach goal state

1. Yes – Euclidian distance relaxes the constraints s.t the joints can all be positioned at any angle.

2. No – heuristics must be lower than the actual cost. No guarantee that's true if multiplied by 2.

3. Yes – you've made an admissible heuristic smaller.

4. Yes – ... but if you know this value, why are you searching?

(c) For each heuristic that you deemed admissible in Part (b), order them from least informed to most informed. **Explain your answers.**

More informed

3. This is 1. made smaller, so will be less useful / informed

1. This is less informed than 4. because 4 is the optimal path cost

4. This is the optimal path cost

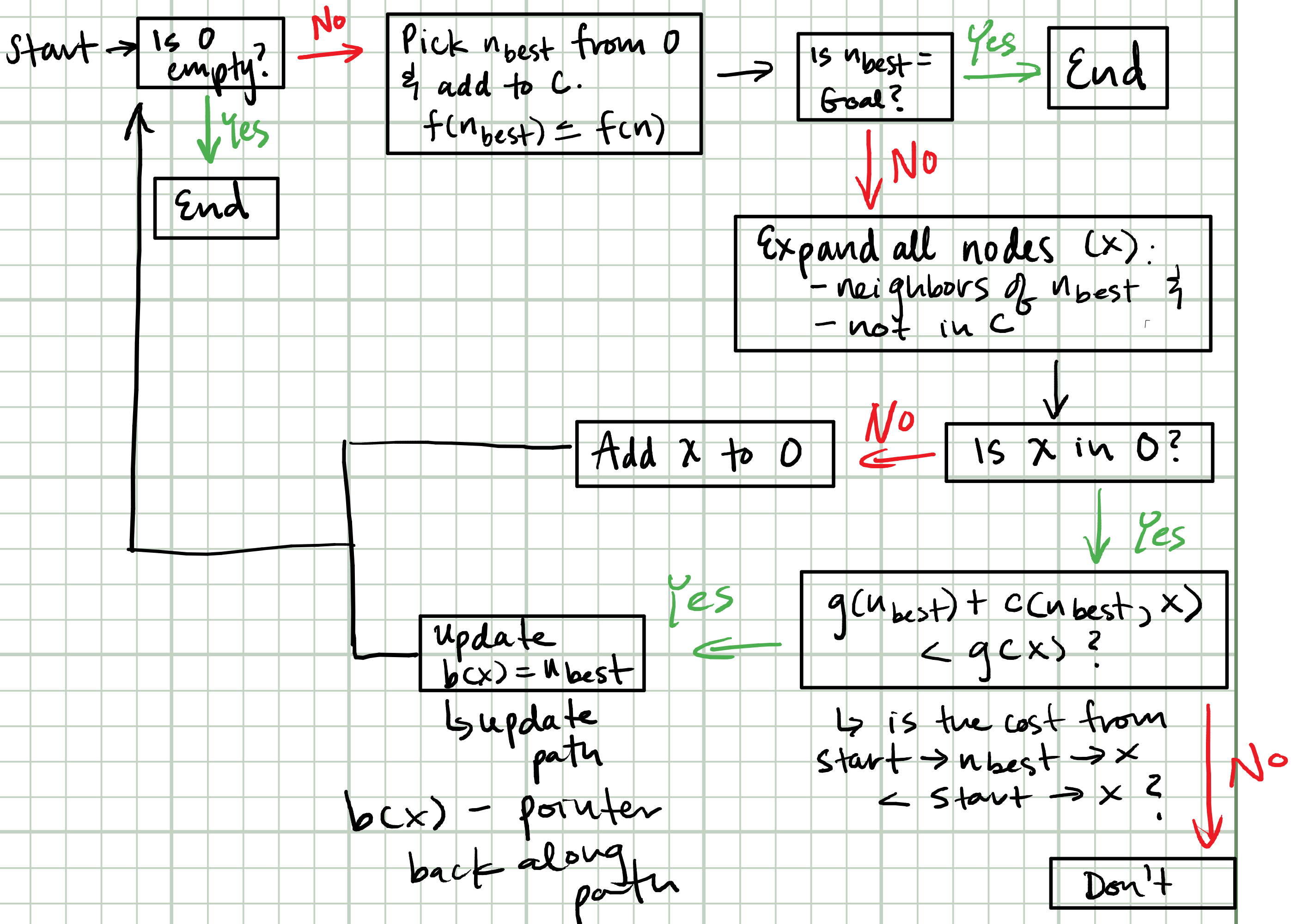
Programming Assignment

1\.

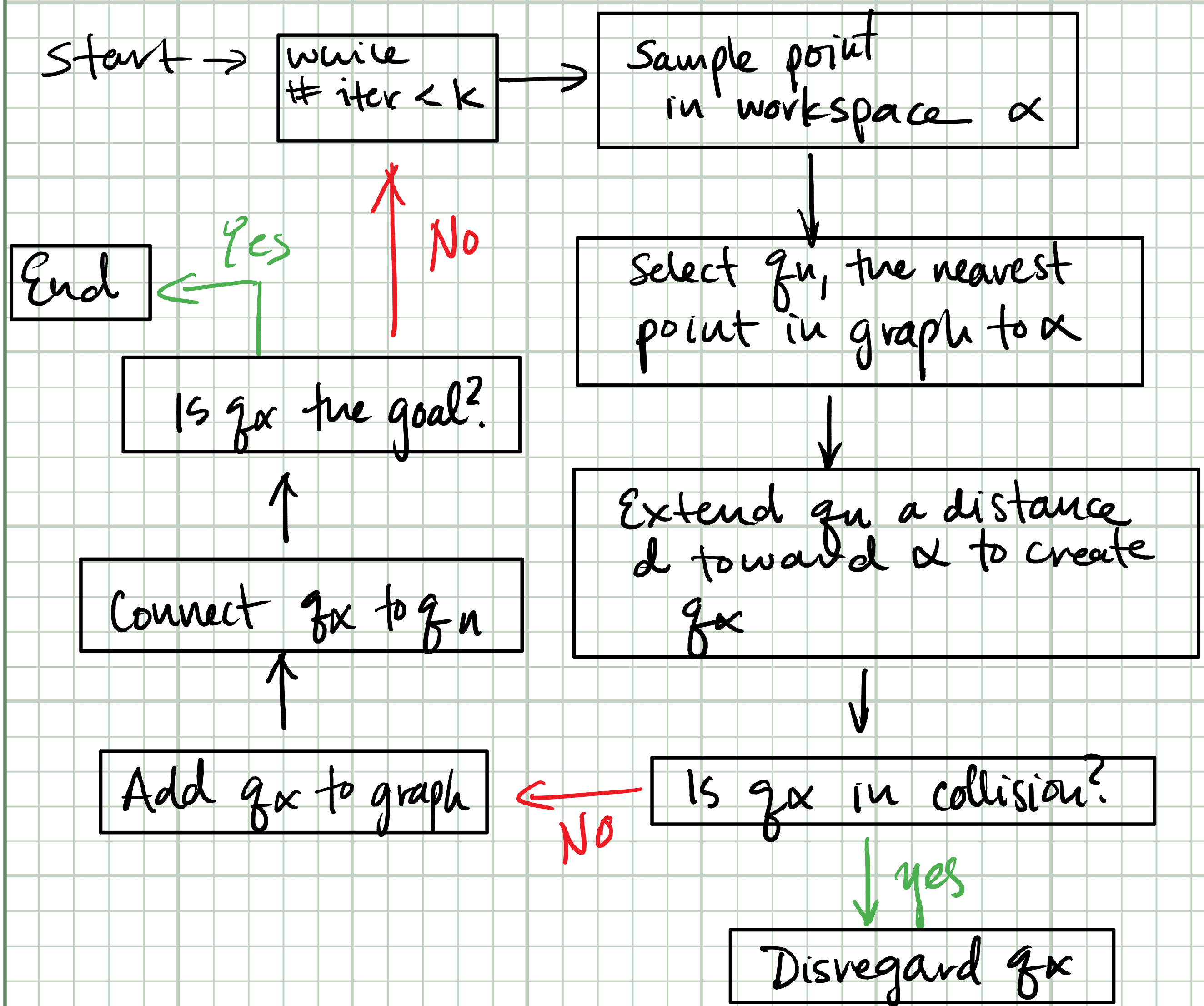
a) A^* flowchart

O – open list, nodes to be expanded

C – closed list, explored nodes

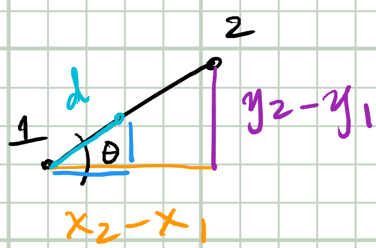


b) RRT Flowchart



$\sin\theta = \Delta y / d$

$\cos\theta = \Delta x / d$



$$\theta = \tan^{-1}\left(\frac{y_2 - y_1}{x_2 - x_1}\right)$$

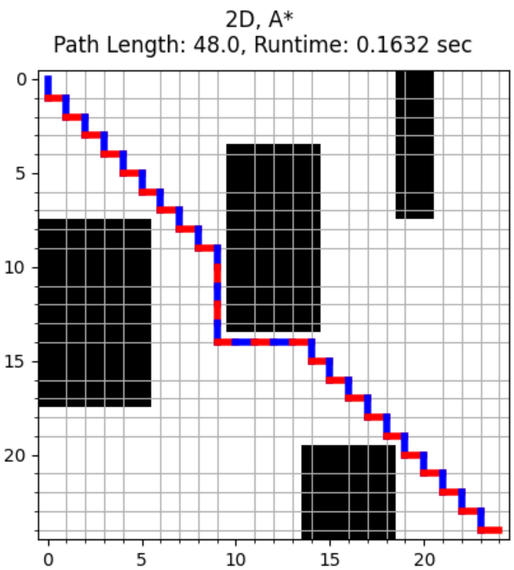
Programming Assignment

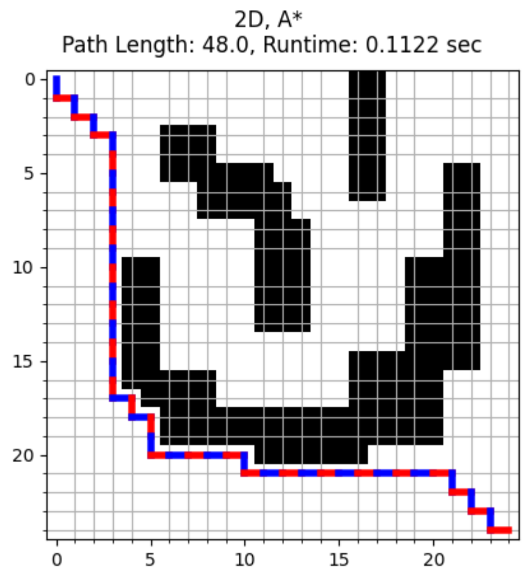
Step 2:

i. Implement 2D A* with an admissible heuristic

   a. The heuristic I used was the <u>Euclidean distance between the current position and the goal</u>. Since there are obstacles and the grid is 4-connected, this distance will always be shorter than the actual optimal path.

   b. The two figures below show the results for 2D A* on the two mazes with their path lengths.



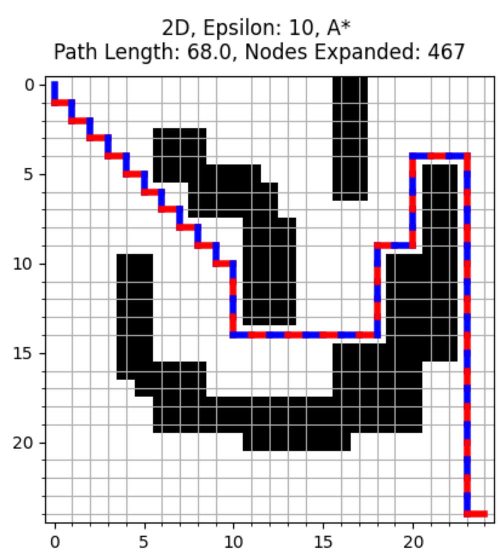


ii. Implement greedy A*

   a. To achieve greedy A*, I multiplied my Euclidean heuristic by a deflating value *epsilon.* The tables below show the results of this deflating-heuristic on both mazes at three different runtime limits.

   b. For Maze 1, I saw no qualifiable or quantifiable difference between the paths generated by the different values of *epsilon.* This was surprising, but might be caused by my choice of heuristic. If the heuristic wasn't informed enough, scaling it by a factor might still be admissible.

   c. For Maze 2, changing the heuristic did change the path length (see figure right), though not the number of nodes expanded.



Maze 1, greedy A*

Runtime 0.05 sec:
No paths completed regardless of *epsilon*

Runtime 0.25 sec

	Epsilon	Path	Nodes
0	10.000000	48.0	474.0
1	5.500000	48.0	474.0
2	3.250000	48.0	474.0
3	2.125000	48.0	474.0
4	1.562500	48.0	474.0
5	1.281250	48.0	474.0
6	1.140625	48.0	474.0
7	1.070312	48.0	474.0
8	1.035156	48.0	474.0
9	1.017578	48.0	474.0
10	1.008789	48.0	474.0
11	1.004395	48.0	474.0
12	1.002197	48.0	474.0
13	1.001099	48.0	474.0

Runtime 1 sec

	Epsilon	Path	Nodes
0	10.000000	48.0	474.0
1	5.500000	48.0	474.0
2	3.250000	48.0	474.0
3	2.125000	48.0	474.0
4	1.562500	48.0	474.0
5	1.281250	48.0	474.0
6	1.140625	48.0	474.0
7	1.070312	48.0	474.0
8	1.035156	48.0	474.0
9	1.017578	48.0	474.0
10	1.008789	48.0	474.0
11	1.004395	48.0	474.0
12	1.002197	48.0	474.0
13	1.001099	48.0	474.0

Maze 2, greedy A*

Runtime 0.05 sec:
No paths completed regardless of *epsilon*

Runtime 0.25 sec

	Epsilon	Path	Nodes
0	10.000000	68.0	467.0
1	5.500000	68.0	467.0
2	3.250000	56.0	467.0
3	2.125000	56.0	467.0
4	1.562500	56.0	467.0
5	1.281250	48.0	467.0
6	1.140625	48.0	467.0
7	1.070312	48.0	467.0
8	1.035156	48.0	467.0
9	1.017578	48.0	467.0
10	1.008789	48.0	467.0
11	1.004395	48.0	467.0
12	1.002197	48.0	467.0
13	1.001099	48.0	467.0

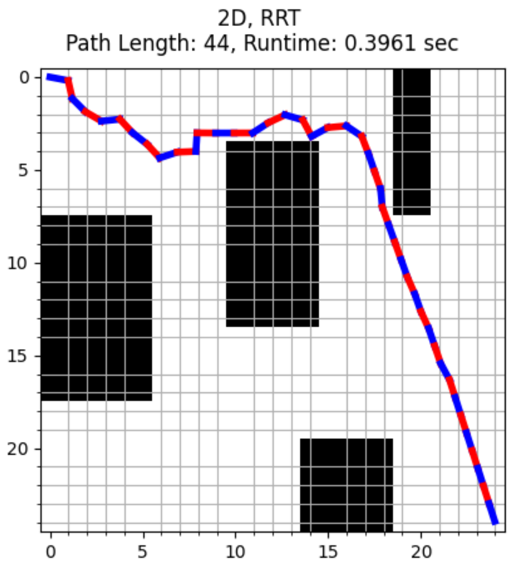
Runtime 1 sec

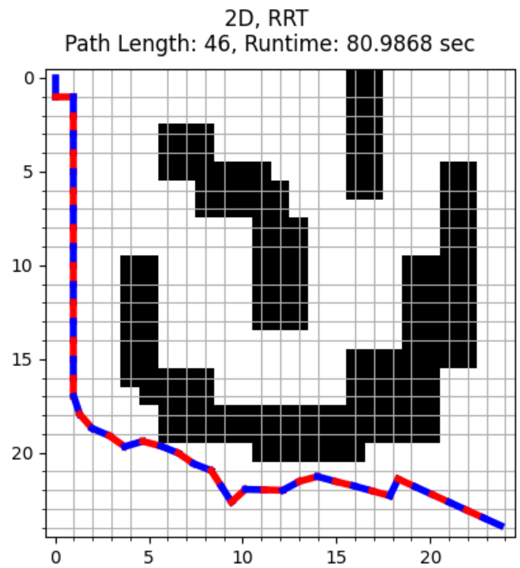
	Epsilon	Path	Nodes
0	10.000000	68.0	467.0
1	5.500000	68.0	467.0
2	3.250000	56.0	467.0
3	2.125000	56.0	467.0
4	1.562500	56.0	467.0
5	1.281250	48.0	467.0
6	1.140625	48.0	467.0
7	1.070312	48.0	467.0
8	1.035156	48.0	467.0
9	1.017578	48.0	467.0
10	1.008789	48.0	467.0
11	1.004395	48.0	467.0
12	1.002197	48.0	467.0
13	1.001099	48.0	467.0

iii. Implement RRT

a. To achieve goal-oriented behavior, I sampled the goal 25% of the time and a random point in the c-space 75% of the time. The two figures below show the behavior of RRT on the two mazes.

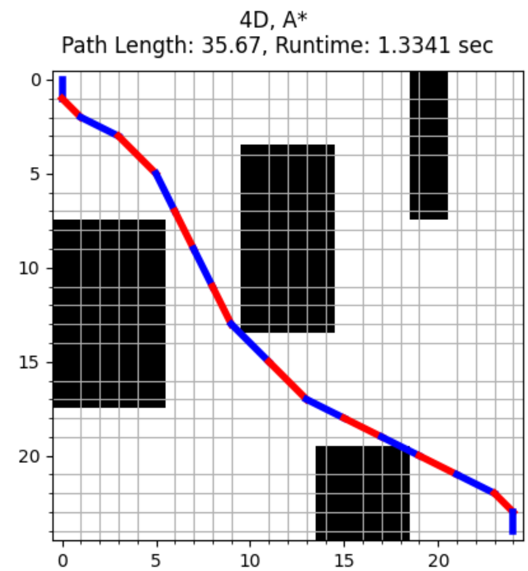
b. The paths generated are clearly not optimal and take longer to compute than A*, but since RRT sampled continuous space, the paths are shorter.

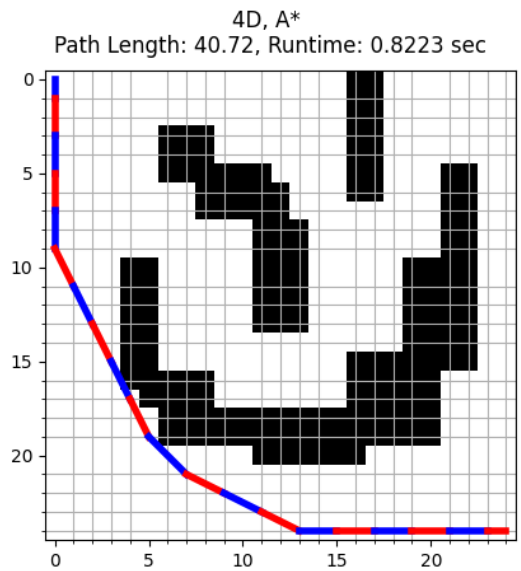




Step 3:

i. Replace the A* 2D maze with the 4D problem to minimize time.
   a. For the higher-dimension case, I made two major changes to the 2D A* algorithm: heuristic and cost-to-come. The paths look smoother than the 2D case, which intuitively makes sense if we're optimizing for time.
   b. The new heuristic I chose was the <u>Euclidean distance between the current position and the goal divided by the maximum velocity the robot could achieve</u>. This resulted in a unit of time that is guaranteed to underestimate the actual time to reach the goal.
   c. The new cost function was the total time the robot was traveling, found by adding each step's (distance / velocity) value to its parent's.





ii. Apply the deflating heuristic (greedy A*) to the 4D problem
   a. The 4D problem exceeded all the running lengths on both mazes. When I let the algorithm exceed the running time, it produces the results below. The path generated and its length both change based on the value of epsilon.

Epsilon	Path	Nodes
10.000000	36.616466	2809.0
5.500000	36.616466	2809.0
3.250000	36.616466	2809.0
2.125000	36.616466	2809.0
1.562500	36.616466	2809.0
1.281250	38.373825	2809.0
1.140625	38.373825	2809.0
1.070312	35.674388	2809.0
1.035156	36.030679	2809.0
1.017578	35.674388	2809.0
1.008789	35.674388	2809.0
1.004395	35.674388	2809.0
1.002197	35.674388	2809.0
1.001099	35.674388	2809.0

Left: Maze 1 (runtime exceeded)

Epsilon	Path	Nodes
10.000000	41.251408	2428.0
5.500000	41.251408	2428.0
3.250000	41.251408	2428.0
2.125000	41.073262	2428.0
1.562500	41.073262	2428.0
1.281250	41.073262	2428.0
1.140625	41.073262	2428.0
1.070312	41.073262	2428.0
1.035156	41.073262	2428.0
1.017578	40.716971	2428.0
1.008789	40.716971	2428.0
1.004395	40.716971	2428.0
1.002197	40.716971	2428.0
1.001099	40.716971	2428.0

Left: Maze 2 (runtime exceeded)

Discussion Questions

1. In this exercise, the collisions were easy to compute. Consider a version of the 4D problem where the collision cost dominates the computation time. How would you modify your algorithm to deal with this case? Hint: Look at the Lazy A* paper [1].
    a. If collision-checking dominated the computation time, I would switch to an algorithm that attempted to reduce the amount of necessary collision detection. Two algorithms that do this are Lazy Shortest Path and Lazy Receding Horizon A*. Lazy Shortest Path reduces collision-checking by first solving the path-planning problem and then checking for collisions along the path, rewiring the graph if necessary. Lazy Receding Horizon attempts to reduce rewiring by doing the path planning and expansion in batches.
    b. To modify my algorithm in this situation, I would implement something similar to Lazy Receding Horizon to limit the rewiring step. This could be done by running A* until you've expanded a set number of nodes, and then executing the path generated. After execution, reset the start position and cost, and repeat. This would mean you wouldn't have to check the entire C-space for obstacles, just those in the local region.

2. Consider using RRT-Connect on the 4D problem. What advantages would this have over standard RRT? What challenges would it lead to that would need to be overcome? Hint: Review the RRT*-Connect paper [3].
    a. The RRT-Connect algorithm grows two random trees, one rooted at the start and the other rooted at the goal. The algorithm samples a point in the workspace, extends the nearest tree toward the sampled point, checks for collision, and repeats. The sampling is biased so the trees grow toward each other.
    b. In the 4D problem, RRT-Connect would visit fewer nodes and be faster than RRT. RRT-Connect is also less likely to get trapped in narrow passages. It would likely help the most in Maze 2; it took RRT over a minute to find the goal in Maze 2, which has more channels and passages than Maze 1. Growing two trees, however, would require more compute power and significantly more nearest-neighbor checks. I would want to implement a proper nearest-neighbor search, instead of using `np.linalg.norm` to find the Euclidean distance between each point.

3. What modifications to the A* algorithm would you make if the robot discovered the environment as it went along (i.e., obstacles appeared and disappeared when the robot was near them)? You can assume the world itself is static, but the robot discovers the world as it moves. Do not provide full pseudocode, just a high-level description. Hint: Review the D*-Lite algorithm [4].
    a. In a changing environment, the robot would need to generate a new path each time it encountered an unexpected obstacle. This could be done with A* (replan starting from the current robot state whenever an obstacle is found), but A* would search the entire C-space each time. If the space was changing rapidly, this would get very computationally expensive. A more dynamic algorithm would only re-expand nodes directly affected by the obstacle. The robot would remember nodes it has already visited and expanded, so wouldn't need to revisit the entire space. Nodes that aren't directly affected by the obstacle would keep their original cost values.

4. What modifications to the RRT algorithm would you make if the robot's position were uncertain (partially observable)? Do not provide full pseudocode, just a high-level description. Hint: Review the RRBT algorithm [5].
    a. If the robot's position were uncertain, RRT alone might generate a path that theoretically succeeds but in execution hits an obstacle or doesn't reach the goal. To remedy this, uncertainty should be included in the state information. For each state where the robot's position is uncertain, I would propagate uncertainty through each step by assigning measurement and state uncertainty to each node. Then a Kalman filter could be used to predict state estimates based on the previous values. This creates a region where the robot could be, instead of a point. This region is important in obstacle avoidance.
    b. The RRBT algorithm also had 'measurement regions' where the robot would gain information about its state. These regions would reduce the uncertainty of the robot state, while actions outside of the measurement region would increase the uncertainty.