ROB 534: Sequential Decision Making in Robotics, Winter 2025
HW #1: Discrete and Sampling-based Planning
Due: 1/29/25 (midnight)

Questions

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.**

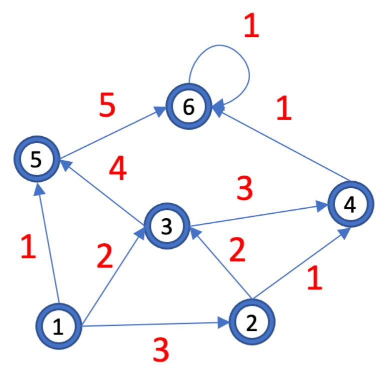
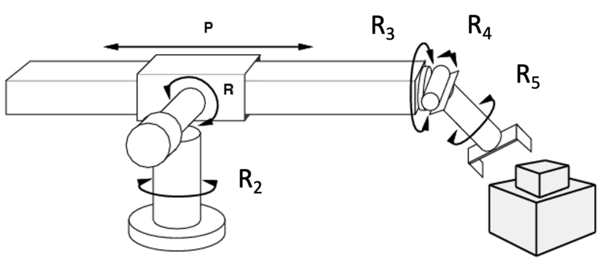


Figure 1: Six-state discrete planning problem.

(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 current node to goal

ii. Euclidean distance in 6DOF space from current node to goal times 2

iii. Euclidean distance in 6DOF space from current node to goal divided by 2

v. Sum of total 6DOF distance travelled by optimal solution to goal state from current node

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

Programming Assignment (Do not start at last minute!) (60 points)

The zip files on canvas contain helper functions and data for this assignment. Download them in addition to this word document. You may use any programming language you wish to complete the homework. Helper functions are provided in the zip file if you choose to use Matlab or Python. The text below is written describing the Matlab code; you should substitute the appropriate functions in Python if you use those functions.

In your Matlab program, nodes should be opaque data structures with operations get_start.m, get_goal.m, and get_neighbors.m. Restrict the number of nodes expanded to 10,000 to avoid very long run times.

To make your job easier, we have provided a Matlab priority queue implementation in the archive. You are free to use this implementation or not, as you see fit. The priority queue has the following functions:
• pq init: Initialize a priority queue.
• pq set: Reset the priority of an element or insert it if it's not already there.
• pq pop: Remove and return the first element. (The first element is the one with the smallest numerical priority value). It breaks ties arbitrarily.
• pq test: Check whether an item is already on the priority queue.

We have also provided two mazes (maze1.pgm and maze2.pgm) along with helper functions to read the mazes, convert from state (x,y) to index, plot paths, find neighbors and other operations. Familiarize yourself with the provided code before getting started.

Step 1 (10 points): Before implementing any algorithms, draw a flowchart of the A* and RRT algorithms. These flowcharts should have text bubbles describing each step with arrows connecting them. Decisions should also be represented as branches with labels (e.g., yes/no). Include the flowcharts in your report.

Step 2 (30 points):

i. **Implement A* with an admissible heuristic. Which heuristic did you use? Provide an image of the path found by your A* implementation on both mazes.** Have the starting node be the top left cell: (x, y) coordinates (1, 1), and have the ending cell be the bottom right cell: (x,y) coordinates (Cols,Rows). Use the total distance traversed by the robot as the cost function.

ii. Allow A* to multiply your admissible heuristic by a constant value epsilon (greedy A*).
Set up a program that does the following:

1. A* runs with a user-provided epsilon
2. Once search is completed it then sets a new_epsilon = epsilon – 0.5*(epsilon-1). If new_epsilon is less than 1.001, it sets new_epsilon = 1.
3. A* runs again with the new epsilon value

4. The loop continues with deflating epsilon until either (a) a running time limit is reached or (b) the search has completed with epsilon = 1.

Use a starting value of epsilon = 10. Record for running time limits 0.05 seconds, 0.25 seconds, and 1 second (on both maps and for each epsilon value where the search completed): number of nodes expanded and path length.

iii. **Implement the RRT algorithm to solve the 2D problem in continuous space on maze1.pgm and maze2.pgm [1].** You may use the knnsearch function in the statistics toolbox for nearest neighbors if using Matlab. Hint: you should sample the goal periodically to get goal directed behavior (do not use fully uniform sampling). You may assume the goal is reached if the x and y values are both within 1 of the goal. You may also assume the robot is modeled as a point. The provided check_hit.m gives an obstacle test. **Provide an image of a path from the RRT algorithm on both maps. Also provide the path length and running time.**

Step 3 (20 points): For this problem, we will expand the state space to 4D to incorporate the dynamics of the agent moving through the world. The agent's state will now consist of (x,y) coordinates as well as (dx,dy) velocities. Instead of controlling its movement directly, the agent now controls its acceleration and deceleration. We have restricted the problem as follows:

• The agent starts in the top left corner (x,y) = (1,1) with zero velocity (dx, dy) = (0, 0).
• The agent must move to the bottom right corner (x,y) = (Rows,Cols) and slow down to zero velocity.
• The agent can accelerate or decelerate by one in either dx or dy (or remain at the same velocity).
• The agent's velocity in both directions must always remain less than or equal to a maximum velocity provided in the variable maxV = 2. Both velocity components must also always remain positive.
• The agent cannot slow down by hitting an obstacle. If it is traveling at a high speed, it must slow down several squares before an obstacle to avoid collision. (You do not want to scratch your car on those pesky obstacles).

For example, if the agent is in state (x,y,dx,dy) = (1,1,2,0), it can choose to speed up dy, slow down dx, or remain at the same speed. It cannot speed up dx because it would break the speed limit, and it cannot slow down dy because it would be moving backwards. If it speeds up dy, it goes to (x, y, dx, dy) = (3,2,2,1). If it slows down dx, it goes to (x,y,dx,dy) = (2,1,1,0). If it remains at the same speed, it goes to (x, y, dx, dy) = (3, 1, 2, 0). For more clarification, see the file get_neighbors_dynamic.m in the archive.

The 4D problem uses the same maps as before, but they are now read in using the read_map_for_dynamics.m function. We have also provided you with the functions test_dynamic_neighbors.m and test_dynamic_path.m to help familiarize you with the format of the maps in 4D space.

For this problem, we have provided you with Matlab code for the functions get goal_dynamic.m, get_start_dynamic.m, and get_neighbors_dynamic.m. These will replace the corresponding functions which you implemented for the 2D problem. The functions dynamic_state_from_index.m and dynamic_index_from_state.m also provide the same functionality as the corresponding 2D functions.

i. **Replace the A* functions that you coded up for the 2D maze problem above and run your search algorithms on the 4D problem to minimize time (not distance). Develop an informed and admissible heuristic for the 4D domain (you only need one). What was your heuristic? Provide an image of your path on both mazes.**

ii. **Apply your deflating heuristic program to the 4D problem. Use a starting value of epsilon = 10. Record for running time limits 0.05 seconds, 0.25 seconds, and 1 second (on both maps and for each epsilon value where the search completed): number of nodes expanded and path length.**

Discussion Questions (20 points): Address these questions (1-2 paragraphs each):
1) Consider a 4D grid problem with constant-time constraints (i.e., limits on time to compute). Informally describe the problem and solution using the constant-time motion planning (CTMP) algorithm [1]. Additionally, discuss potential real-world applications of CTMP beyond those highlighted in the research paper.
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].
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].
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].

References

[1] Islam, F., Salzman, O., Agarwal, A., & Likhachev, M., Provably constant-time planning and replanning for real-time grasping objects off a conveyor belt. The International journal of robotics research, 40(12-14), 1370-1384, 2021.
[2] R. Mashayekhi, M.Y.I. Idris, M.H. Anisi, I. Ahmedy and I. Ali. "Informed RRT*-connect: An asymptotically optimal single-query path planning method." IEEE Access, vol. 8, pp. 19842-19852, 2020.
[3] S. Koenig and M. Likhachev, "D* Lite," Proc. AAAI/IAAI, Vol. 15, 2002.
[4] A. Bry, and N. Roy. "Rapidly-exploring random belief trees for motion planning under uncertainty." Proc. IEEE Conference on Robotics and Automation (ICRA), 2011.

ROB 534 SDM HW1 - Chelse VanAtter

Questions:

1. Search-based planning (20 pts)
   a. Consider the planning problem
      i. Use backward value iteration to determine the stationary cost-to-go

Definitions:

J(s) = cost-to-go for state s

C(s,s') = cost to transition from s to s'

Succ(s) = set of successor states for state s

$J(s) = \min_{s' \in Succ(s)} \{C(s,s') + J(s')\}$

Transition Costs:

C(1,2) =3, C(1,3) = 2, C(1,5) = 1,

C(2,3) = 2, C(2,4) = 1,

C(3,4) = 3, C(3,5) = 4,

C(4,6) = 1,

C(5,6) = 5,

C(6,6) = 1.

Initialization:

J(6) = 0, All other states are initialized with J(s) = infinity

Iteration 1:

J(6) = 0

J(5) = min{C(5,6) + J(6)} = min{5 + 0} = 5

J(4) = min{C(4,6) + J(6)} = min{1 + 0} = 1

J(3) = min{C(3,4) + J(4), C(3,5) + J(5)} = min{3 + 1, 4 + 5} = 4

J(2) = min{C(2,3) + J(3), C(2,4) + J(4)} = min{2 + 4, 1 + 1} = 2

J(1) = min{C(1,2) + J(2), C(1,3) + J(3), C(1,5) + J(5)} = min{3 + 2, 2 + 4, 1 + 5} = 5

Iteration 2:

J(5) = 5 (no change)

J(4) = 1J (no change)

J(3) = min{C(3,4) + J(4), C(3,5) + J(5)} = min{3 + 1, 4 + 5} = 4 (no change

J(2) = min{C(2,3) + J(3), C(2,4) + J(4)} = min{2 + 4, 1 + 1} = 2 (no change)

J(1) = min{C(1,2) + J(2), C(1,3) + J(3), C(1,5) + J(5)} = min{3 + 2, 2 + 4, 1 + 5} = 5 (no change)

Final cost-to-go:

J(1) = 5, J(2) = 2, J(3) = 4, J(4) = 1, J(5) = 5, J(6) = 0.

ii. Do the same but with forward value iteration

<u>Initialization:</u>

J(1) = 0

All other states are initialized with J(s) = infinity except J(6) if explicitly updated during the iterations

<u>Iteration 1:</u>

J(1) = 0 (initial state)

Compute costs to successors of state 1:

J(2) = min{C(1,2) + J(1)} = min{3 + 0} = 3

J(3) = min{C(1,3) + J(1)} = min{2 + 0} = 2

J(5) = min{C(1,5) + J(1)} = min{1 + 0} = 1

<u>Iteration 2:</u>

Update successors of state 2:

J(3) = min{J(3), C(2,3) + J(2)} = min{2, 2 + 3} = 2 (no change)

J(4) = min{C(2,4) + J(2)} = min{1 + 3} = 4

Update successors of state 3:

J(4) = min{J(4), C(3,4) + J(3)} = min{4, 3 + 2} = 4 (no change)

J(5) = min{J(5), C(3,5) + J(3)} = min{1, 4 + 2} = 1 (no change)

Update successors of state 5:

J(6) = min{C(5,6) + J(5)} = min{5 + 1} = 6

<u>Iteration 3:</u>

Update successors of state 4:

J(6) = min{J(6), C(4,6) + J(4)} = min{6, 1 + 4} = 5

Update successors of state 6:

J(6) = min{J(6), C(6,6) + J(6)} = min{5, 1 + 5} = 5 (no change)

<u>**Final cost-to-go:**</u>

J(1) = 0, J(2) = 3, J(3) = 2, J(4) = 4, J(5) = 1, J(6) = 5

.

b. Consider the 6-DOF planning problem - Admissibility Heuristic Evaluation:

i. This heuristic is **admissible** because the Euclidean distance is always less than or equal to the actual cost because the actual cost has to consider path constraints, joint limits, and other factors that typically increase travel distance compared to a straight-line path.

   ii. This heuristics is **not admissible** because multiplying the Euclidean distance times 2 is an overestimation of the actual cost in most cases which violates the admissibility condition.
       This heuristic is the **least informed** because it underestimates the true cost significantly.
   iii. This heuristic is **admissible** because dividing the Euclidean distance by 2 results in an underestimation of the actual cost and since it is guaranteed to never overestimate the actual cost, it is admissible.
   iv. This heuristic is **admissible** because it exactly matches the actual cost because it assumes knowledge of the optimal path and since it never overestimates it is admissible.

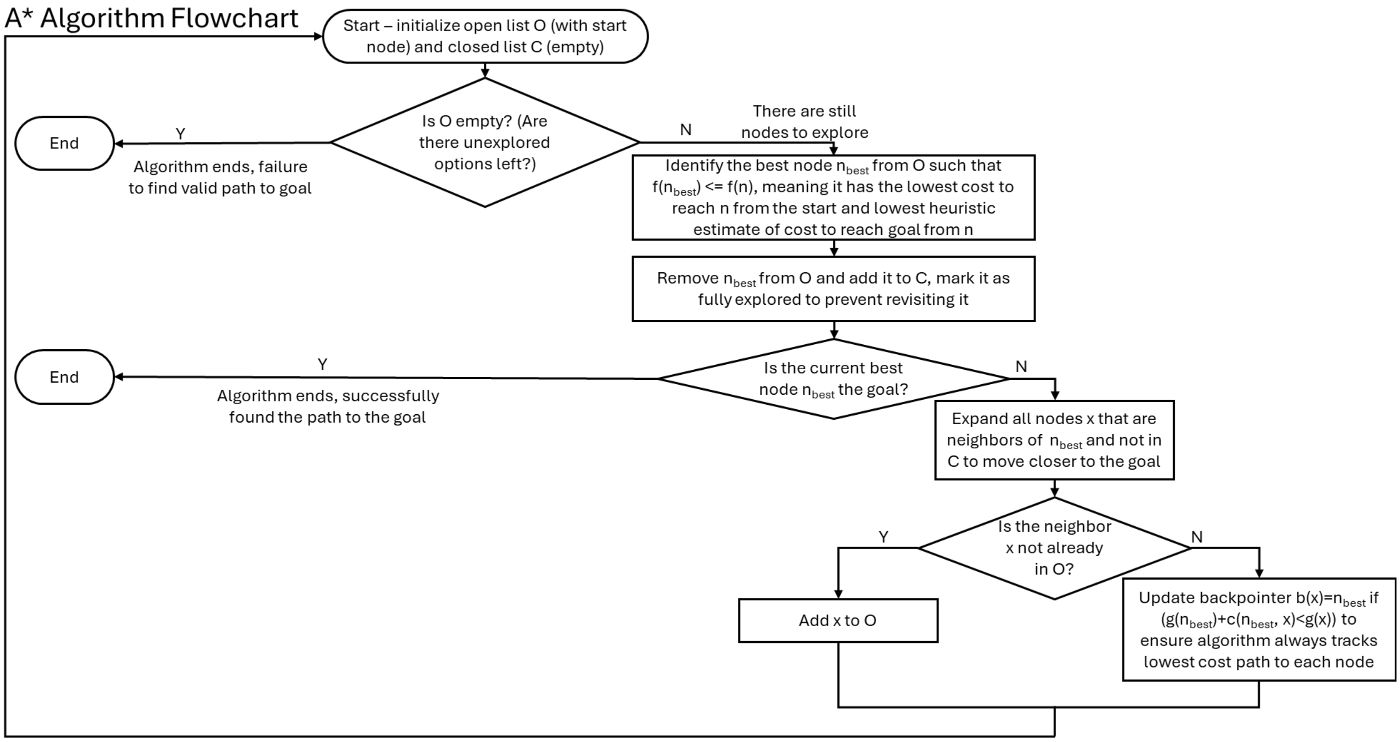
   Summary: heuristics i, iii, iv are admissible and heuristic ii is not admissible

c. Consider the 6-DOF planning problem - Least to Most Informed Heuristics:
   i. This heuristic is **moderately informed**. It is more informed than heuristic iii because it can overestimate the actual cost but it can be equal to the actual cost so the lower bound is accurate since it directly represents the straight line distance to the goal.
   ii. This heuristic is **more informed**. It is more informed than heuristics i and iii because it generally provides values closer to the actual cost since
   iii. This heuristic is the **least informed**. It is the least informed since it significantly underestimates the actual cost and provides the weakest guidance.
   iv. This heuristic is the **most informed** since it matches the true cost-to-go exactly in the optimal case.

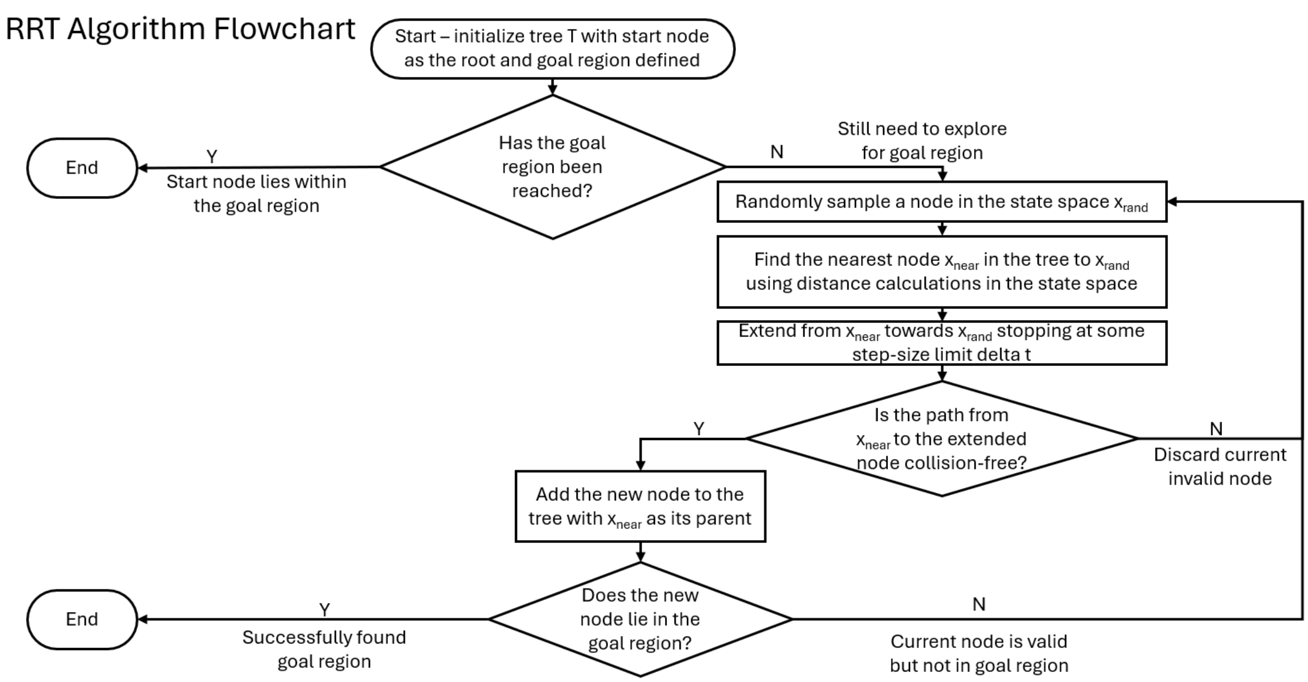
   Summary: leat to most informed: h(iii) < h(i) < h(ii) < h(iv)

Programming Assignment:

Step 1: Flowchart of A*:

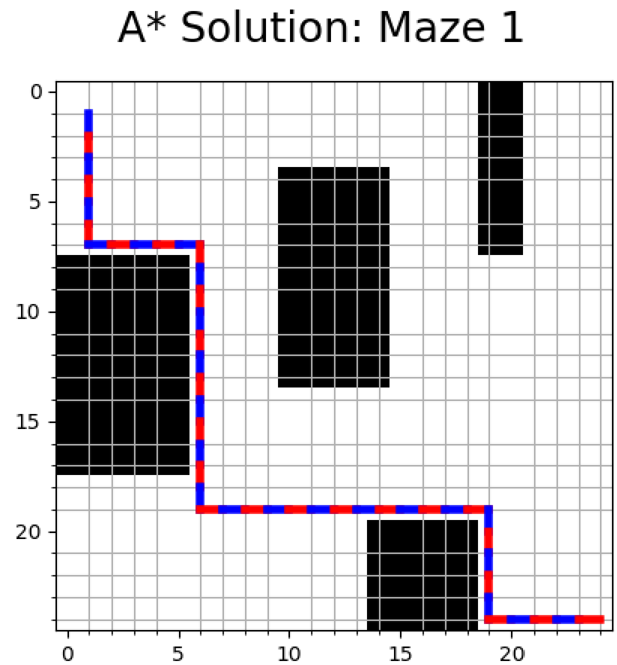


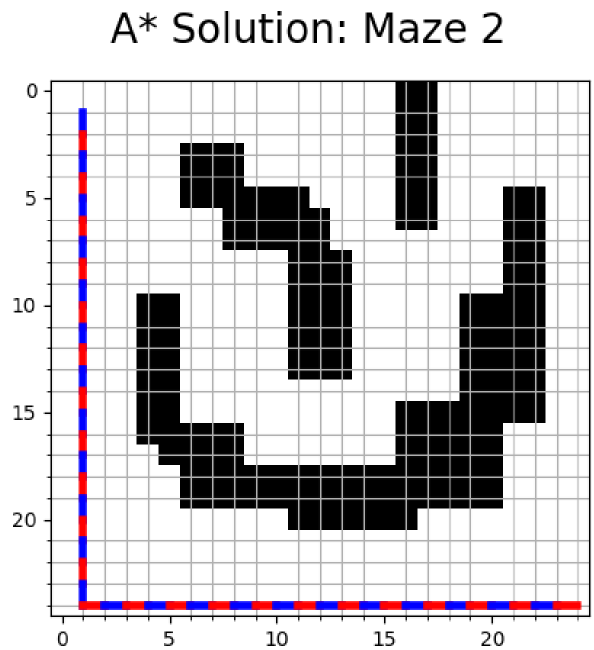
Flowchart of RRT:



Step 2:

i. I used the Manhattan distance heuristic





ii. The number of nodes expanded was the same for all time limits for map 1 and map 2 for all values of epsilon.

At epsilon = 10 for time limits: 0.05 seconds, 0.25 seconds, and 1 second

- Map 1 number of nodes expanded: 128
- Map 1 path length: 57
- Map 2 number of nodes expanded: 115
- Map 2 path length: 47

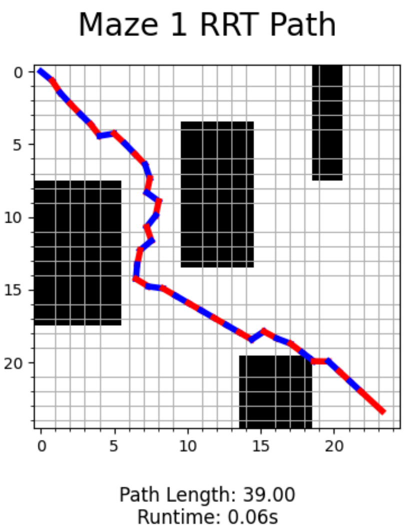
At epsilon = 1 for time limits: 0.05 seconds, 0.25 seconds, and 1 second

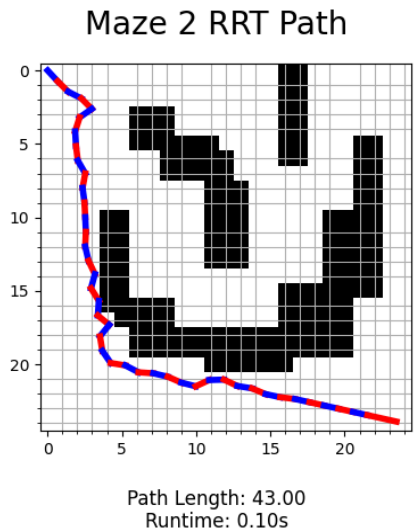
- Map 1 number of nodes expanded: 409
- Map 1 path length: 47
- Map 2 number of nodes expanded: 389
- Map 2 path length: 47

```
Running with time limit 0.05s and initial epsilon 10
Epsilon: 10, Nodes expanded: 128, Path length: 57, Elapsed time: 0.0010
Running with time limit 0.25s and initial epsilon 10
Epsilon: 10, Nodes expanded: 128, Path length: 57, Elapsed time: 0.0010
Running with time limit 1s and initial epsilon 10
Epsilon: 10, Nodes expanded: 128, Path length: 57, Elapsed time: 0.0020
Testing Maze 2 with epsilon decay...
Running with time limit 0.05s and initial epsilon 10
Epsilon: 10, Nodes expanded: 115, Path length: 47, Elapsed time: 0.0000
Running with time limit 0.25s and initial epsilon 10
Epsilon: 10, Nodes expanded: 115, Path length: 47, Elapsed time: 0.0010
Running with time limit 1s and initial epsilon 10
Epsilon: 10, Nodes expanded: 115, Path length: 47, Elapsed time: 0.0000
Testing Maze 1 with epsilon decay...
Running with time limit 0.05s and initial epsilon 1
Epsilon: 1, Nodes expanded: 409, Path length: 47, Elapsed time: 0.0020
Running with time limit 0.25s and initial epsilon 1
Epsilon: 1, Nodes expanded: 409, Path length: 47, Elapsed time: 0.0020
Running with time limit 1s and initial epsilon 1
Epsilon: 1, Nodes expanded: 409, Path length: 47, Elapsed time: 0.0021
Testing Maze 2 with epsilon decay...
Running with time limit 0.05s and initial epsilon 1
Epsilon: 1, Nodes expanded: 389, Path length: 47, Elapsed time: 0.0020
Running with time limit 0.25s and initial epsilon 1
Epsilon: 1, Nodes expanded: 389, Path length: 47, Elapsed time: 0.0020
Running with time limit 1s and initial epsilon 1
Epsilon: 1, Nodes expanded: 389, Path length: 47, Elapsed time: 0.0020
```

This terminal output shows the values above.

iii. RRT algorithm

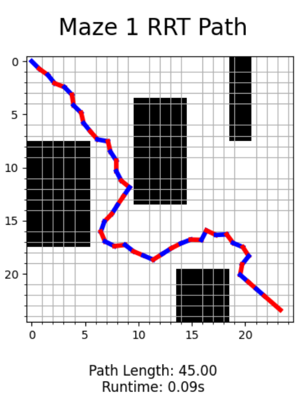


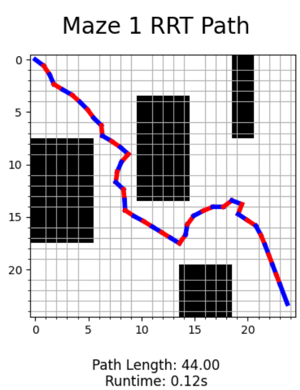


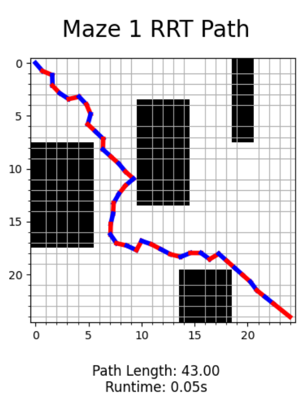
The terminal shows the path length and runtime with more significant figures.

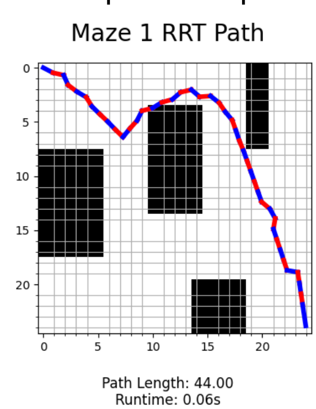
```
PS C:\Users\chels\Downloads\HW1_complete\HW1_complete\hw1_python\hw1_python\provided_code> & c:/Users/chels/
Users/chels/Downloads/HW1_complete/HW1_complete/hw1_python/hw1_python/provided_code/test_maze_step2_iii.py
Running RRT with step size 1 and max iterations 1000 on Maze 1...
Iteration 0/1000, Tree size: 2
RRT Path Found for Maze 1. Path Length: 39.0, Runtime: 0.05655813217163086 seconds.
RRT Path Found for Maze 1. Path Length: 39.0, Runtime: 0.05655813217163086 seconds.
Running RRT with step size 1 and max iterations 1000 on Maze 2...
Iteration 0/1000, Tree size: 2
RRT Path Found for Maze 2. Path Length: 43.0, Runtime: 0.1022193431854248 seconds.
RRT Path Found for Maze 1. Path Length: 43.0, Runtime: 0.1022193431854248 seconds.
PS C:\Users\chels\Downloads\HW1_complete\HW1_complete\hw1_python\hw1_python\provided_code> []
```

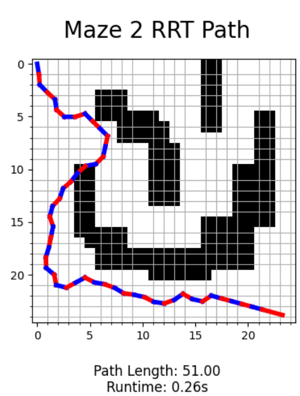
Below are other runs where the program only found a path for either map 1 or map 2

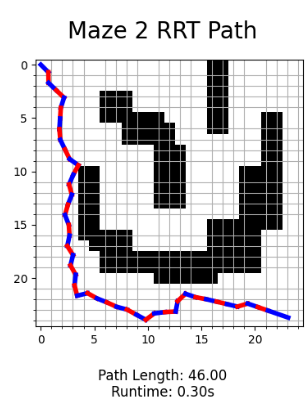


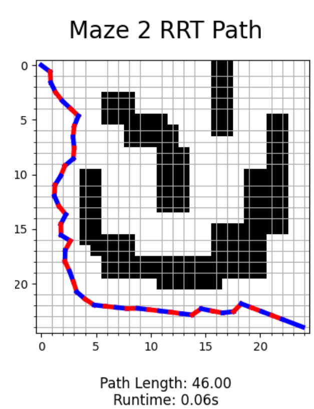


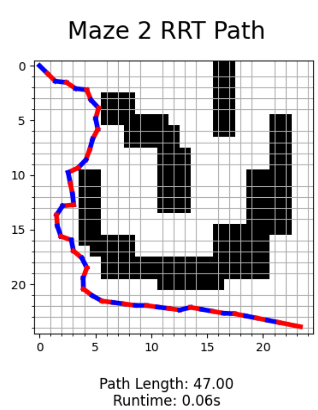






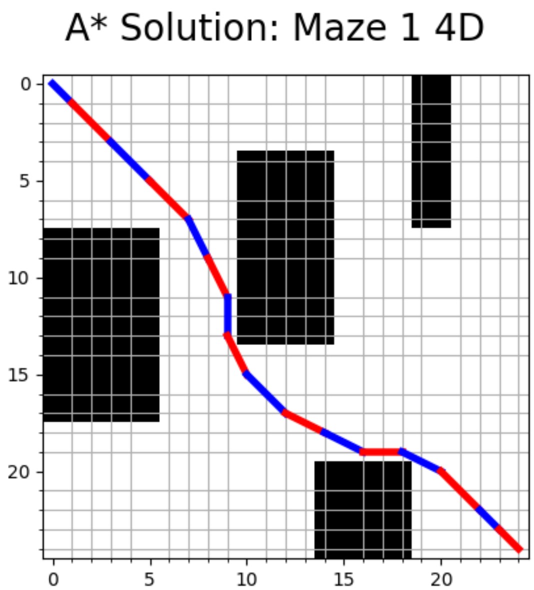


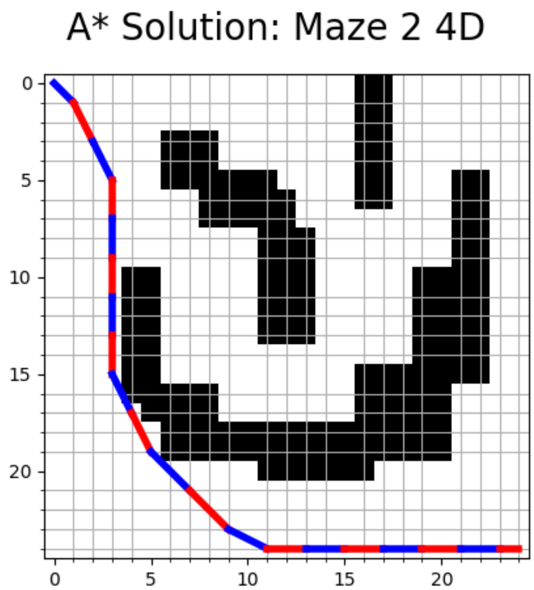




Step 3:

i. I used the Manhattan distance heuristic





ii. The number of nodes expanded was the same for all time limits for map 1 and map 2 for all values of epsilon.

At epsilon = 10 for time limits: 0.05 seconds, 0.25 seconds, and 1 second

- Map 1 number of nodes expanded: 237
- Map 1 path length: 51
- Map 2 number of nodes expanded: 1330
- Map 2 path length: 52

At epsilon = 1 for time limits: 0.05 seconds, 0.25 seconds, and 1 second

- Map 1 number of nodes expanded: 379
- Map 1 path length: 38
- Map 2 number of nodes expanded: 1883
- Map 2 path length: 36

The terminal output shows the values above:

```
Testing Maze 1 ...
Initial epsilon: 10, Nodes expanded: 237, Path length: 51, Elapsed time: 0.0025, Time limit: 0.05s
Initial epsilon: 10, Nodes expanded: 237, Path length: 51, Elapsed time: 0.0020, Time limit: 0.25s
Initial epsilon: 10, Nodes expanded: 237, Path length: 51, Elapsed time: 0.0010, Time limit: 1s
Initial epsilon: 1, Nodes expanded: 379, Path length: 38, Elapsed time: 0.0050, Time limit: 0.05s
Initial epsilon: 1, Nodes expanded: 379, Path length: 38, Elapsed time: 0.0055, Time limit: 0.25s
Initial epsilon: 1, Nodes expanded: 379, Path length: 38, Elapsed time: 0.0050, Time limit: 1s

Testing Maze 2 ...
Time limit 0.05s reached.
Initial epsilon: 10, Nodes expanded: 1330, Path length: 52, Elapsed time: 0.1308, Time limit: 0.05s
Initial epsilon: 10, Nodes expanded: 1330, Path length: 52, Elapsed time: 0.1300, Time limit: 0.25s
Initial epsilon: 10, Nodes expanded: 1330, Path length: 52, Elapsed time: 0.1309, Time limit: 1s
Time limit 0.05s reached.
Initial epsilon: 1, Nodes expanded: 1883, Path length: 36, Elapsed time: 0.0814, Time limit: 0.05s
Initial epsilon: 1, Nodes expanded: 1883, Path length: 36, Elapsed time: 0.0780, Time limit: 0.25s
Initial epsilon: 1, Nodes expanded: 1883, Path length: 36, Elapsed time: 0.0793, Time limit: 1s
```

Python file is attached at the end of this document!

Discussion Questions:

1. Consider a 4D grid problem with constant-time constraints (i.e., limits on time to compute). Informally describe the problem and solution using the constant-time motion planning (CTMP) algorithm [1]. Additionally, discuss potential real-world applications of CTMP beyond those highlighted in the research paper.

   For a 4D grid problem with constant-time constraints, the objective is to find a path for a robot to traverse through a 3D grid with limited time to compute each action. The 4D grid is represented by (x, y, dx, dy) where x and y are spatial coordinates and dx and dy are velocities. The problem is the time required for computing each action and the CTMP algorithm solves this problem by ensuring each action takes a constant amount of time to compute, regardless of how complex the problem is. It uses simplifications to allow for faster, more efficient processing which means that there is some trade off between the level of optimality of the solution to make it faster to provide more real-time feasibility.

   The main real-world applications highlighted by the research paper are for moving objects off conveyor belts in warehouse and manufacturing applications. The CTMP algorithm can have several other real-world applications such as multi-agent systems, video games, simulations, navigation in unpredictable environments which is useful for domains such as agriculture, autonomous vehicles, human-robot-interaction, and military applications. For example, autonomous vehicles have to navigate through complex, unpredictable environments and make real-time decisions and the CTMP algorithm could help make these fast decisions where computation time is critical for safety and effectiveness. The same thing applies for military robots and drones that have to navigate unstructured, dynamic environments and make real-time decisions. Some agricultural robots don't function in real-time since they don't operate with people around, but for better human-robot interaction, safety, and effectiveness using the CTMP algorithm could help make faster decisions. For video games and real-time simulations the CTMP algorithm could help make decisions fast enough that things move fluidly in real-time. For multi-agent systems, fast computation is essential to maintain coordination and efficiency in unpredictable environments especially for mission-critical scenarios such as search and rescue operations.

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].

The RRT algorithm can be slow and sometimes does not find the goal, especially in 4D. The RRT-Connect algorithm offers several advantages and is more efficient. Because it grows two trees simultaneously from the start and goal states, it significantly reduces the search time and makes it easier to find a feasible path in fewer steps. The main advantages of RRT-Connect over RRT are faster convergence, more efficient exploration, lower computation cost, improved path quality, and better handling of large spaces.

The challenges with RRT-Connect are lack of optimality, inefficient use of cost metrics, suboptimal connection strategy, static exploration bias, and lack of guarantee of path quality. The RRT-Connect algorithm terminates as soon as it finds a path that connects the tree growing from the start to the tree growing from the goal which might not be the shortest or most optimal path. RRT*-Connect improves on RRT-Connect and by using cost-based rewiring to refine paths based on a cost function which improves optimality and quality of the path and more efficiently uses cost metrics. RRT*-Connect uses the rewiring to improve paths and results in a increasingly optimal solution as more samples are added.

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].

   Some modifications to the A* algorithm that could help for dynamically exploring the environment are incremental path replanning, heuristic consistency, lazy evaluation, and path cost re-evaluation. Implementing incremental path planning allows the algorithm to dynamically update the path when new information from the environment is received such as obstacles appearing or disappearing as the robot drives towards or past obstacles. Ensuring a consistent heuristic that is admissible and monotonic during updates helps avoid invalid solutions and inefficiency. Lazy evaluation reuses previously generated path information to avoid recalculating paths from scratch as much as possible. Similarly, it could be modified to only recalculate affected nodes rather than the entire grid when the obstacle map changes and the path cost needs to be re-evaluated.

   The D*-Lite algorithm is specifically designed to handle these dynamic environments efficiently by using many of these modifications. It uses incremental path planning to minimize redundant calculations by reusing results from previous searches and only updates nodes affected by newly discovered information using a priority queue to maintain and update these nodes. It uses dynamic cost updates for local edge costs when obstacles appear or disappear and efficiently propagates these changes to maintain path optimality. It uses a lazy evaluation strategy to prioritize nodes that are most likely to affect the current most optimal path rather than immediately exploring all possible nodes.

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].

   The main modifications that would need to be made are belief state planning instead of state space planning, uncertainty-aware sampling, propagation of belief nodes, probabilistic collision checks, and cost-to-go under uncertainty. When a robot's position is uncertain, the algorithm would need to be modified to not act in the state space and instead act in the belief space which represents a probability distribution over the possible positions the robot could be in which is estimated by using the robot's internal state estimates and sensor readings. Nodes in the RRT tree would represent the beliefs and to propagate belief nodes and expand the tree it would update the belief using techniques like Bayesian filtering with Kalman or particle filters. The algorithm would also need to do probabilistic collision checks to account for uncertainty in the robot's position and the representation of the environment. The cost function would also need to be modified to not only consider the path length but also the uncertainty at each state so it chooses paths that minimize uncertainty while also being efficient.

   The RRBT algorithm addresses these modifications by operating in the belief space, using feedback controllers for stability, optimizing belief and cost, and re-evaluating belief nodes. The RRBT algorithm samples and expands in the belief space and constructs a tree of beliefs reflecting the uncertainty of the robot's state and the path/environment. It uses feedback controllers to reduce error propagation which ensures stability and reliability of planned paths even with the uncertainty. It reduces the trade off between path efficiency and uncertainty to minimize path length and positional error using an optimization framework that incorporates both cost and belief. When the RRBT algorithm receives new information, it re-evaluates belief nodes and edges to ensure they align with the updated beliefs, which continuously improves the solution quality.

References:

[1] Islam, F., Salzman, O., Agarwal, A., & Likhachev, M., Provably constant-time planning and replanning for real-time grasping objects off a conveyor belt. The International journal of robotics research, 40(12-14), 1370-1384, 2021.

[2] R. Mashayekhi, M.Y.I. Idris, M.H. Anisi, I. Ahmedy and I. Ali. “Informed RRT*-connect: An asymptotically optimal single-query path planning method.” IEEE Access, vol. 8, pp. 19842-19852, 2020.

[3] S. Koenig and M. Likhachev, “D* Lite,” Proc. AAAI/IAAI, Vol. 15, 2002.

[4] A. Bry, and N. Roy. "Rapidly-exploring random belief trees for motion planning under uncertainty." Proc. IEEE Conference on Robotics and Automation (ICRA), 2011.

[5] Geoff/Rakesh, there was a [5] but no reference with it, just letting you know, maybe the numbering is wrong because 2 doesn’t appear in the in-text citations so maybe the in-text citations should be 1, 2, 3, 4 instead of 1, 3, 4, 5

SDM_HW1_Chelse.py

```
#!/usr/bin/env python3

import abc
import numpy as np
import time
import heapq
import random
from sklearn.neighbors import NearestNeighbors
from matplotlib import pyplot as plt
from matplotlib.ticker import MultipleLocator
from heapq import heappop, heappush

class Maze(abc.ABC):
    """ Base Maze Class """

    def __init__(self, maze_array, start_index=None, goal_index=None):
        """
            maze_array - 2D numpy array with 1s representing free space
                         0s representing occupied space
        """
        self.maze_array = maze_array
        self.cols, self.rows = self.maze_array.shape
        self.start_index = start_index
        self.goal_index = goal_index

    def __repr__(self):
        if isinstance(self, Maze2D):
            output = "2D Maze\n"
        output += str(self.maze_array)
        return output

    @classmethod
    def from_pgm(cls, filename):
        """
            Initializes the Maze from a (8 bit) PGM file
        """
        with open(filename, 'r', encoding='latin1') as infile:
            header = infile.readline()
            width, height, _ = [int(item) for item in header.split()[1:]]
            image = np.fromfile(infile, dtype=np.uint8).reshape((height, width)) / 255

        return cls(image.T)

    def plot_maze(self):
        """ Visualizes the maze """
        self.plot_path([], "Maze")

    def plot_path(self, path, title_name=None, runtime=None, path_length=None):
```

```python
        """
            Plots the provided path on the maze and optionally shows
            runtime and path length in a text box outside the plot area.
        """
        fig = plt.figure(1)
        ax1 = fig.add_subplot(1, 1, 1)

        spacing = 1.0  # Spacing between grid lines
        minor_location = MultipleLocator(spacing)

        # Set minor tick locations.
        ax1.yaxis.set_minor_locator(minor_location)
        ax1.xaxis.set_minor_locator(minor_location)

        # Set grid to use minor tick locations.
        ax1.grid(which='minor')

        colors = ['b', 'r']
        plt.imshow(self.maze_array.T, cmap=plt.get_cmap('bone'))

        if title_name is not None:
            fig.suptitle(title_name, fontsize=20)

        # cast path to numpy array so indexing is nicer
        path = np.array(path)
        for i in range(len(path) - 1):
            cidx = i % 2
            plt.plot([path[i, 0], path[i + 1, 0]], [path[i, 1], path[i + 1, 1]], color=colors[cidx], linewidth=4)

        # If runtime and path length are provided, add them outside the plot
        if runtime is not None and path_length is not None:
            text = f"Path Length: {path_length:.2f}\nRuntime: {runtime:.2f}s"

            # Adjust position and move the text further down
            fig.subplots_adjust(bottom=0.2)  # Make more room for the text box below the graph
            plt.figtext(0.5, 0.02, text, ha="center", fontsize=12)  # Move the text further down

        plt.show()

    def check_occupancy(self, state):
        """ Returns True if there is an obstacle at state """
        return self.maze_array[int(state[0]), int(state[1])] == 0
    def get_goal(self):
        """ Returns the index of the goal """
        return self.goal_index

    def get_start(self):
        """ Returns the index of the start state """
```

```python
        return self.start_index

    def check_hit(self, start, deltas):
        """
            Returns True if there are any occupied states between:
            start[0] to start[0]+dx and start[1] to start[1]+dy
        """
        x, y = start
        dx, dy = deltas

        if (x < 0) or (y < 0) or (x >= self.cols) or (y >= self.rows):
            return True

        if self.maze_array[int(round(start[0])), int(round(start[1]))] == 0:
            return True

        if dx == 0.0 and dy == 0.0:  # no actual movement
            return False

        norm = max(abs(dx), abs(dy))
        dx /= norm
        dy /= norm

        for _ in range(int(norm)):
            x += dx
            y += dy
            if (x < 0) or (y < 0) or (x >= self.cols) or (y >= self.rows):
                return True
            if self.maze_array[int(x), int(y)] == 0:
                return True
        return False

    def check_occupancy(self, state):
        """ Returns True if there is an obstacle at state """
        return self.maze_array[int(state[0]), int(state[1])] == 0


class Maze2D(Maze):
    """ Maze2D Class """

    def __init__(self, maze_array, start_state=None, goal_state=None):
        super().__init__(maze_array, start_state, goal_state)

        if start_state is None:
            start_state = (0, 0)
        self.start_state = start_state
        self.start_index = self.index_from_state(self.start_state)

        if goal_state is None:
            goal_state = (self.cols-1, self.rows-1)
```

```python
        self.goal_state = goal_state
        self.goal_index = self.index_from_state(self.goal_state)

    def index_from_state(self, state):
        """ Gets a unique index for the state """
        return state[0] * self.rows + state[1]

    def state_from_index(self, state_id):
        """ Returns the state at a given index """
        x = int(np.floor(state_id / self.rows))
        y = state_id % self.rows
        return (x, y)

    def get_neighbors(self, state_id):
        """ Returns a List of indices corresponding to neighbors of a given state """
        state = self.state_from_index(state_id)
        deltas = [[0, -1], [0, 1], [-1, 0], [1, 0]]
        neighbors = []
        for delta in deltas:
            if not self.check_hit(state, delta):
                new_state = (state[0] + delta[0], state[1] + delta[1])
                neighbors.append(self.index_from_state(new_state))
        return neighbors

    # Step 2 part i code
    def a_star_step2_i(self):
        """ Perform A* search to find the optimal path from start to goal """
        start = self.start_index
        goal = self.goal_index

        open_set = []
        heappush(open_set, (0, start))  # (f_score, state_index)

        came_from = {}
        g_score = {start: 0}
        f_score = {start: self._heuristic(start)}

        while open_set:
            _, current = heappop(open_set)

            if current == goal:
                return self._reconstruct_path(came_from, current)

            for neighbor in self.get_neighbors(current):
                tentative_g_score = g_score[current] + 1

                if neighbor not in g_score or tentative_g_score < g_score[neighbor]:
                    came_from[neighbor] = current
                    g_score[neighbor] = tentative_g_score
                    f_score[neighbor] = g_score[neighbor] + self._heuristic(neighbor)
```

```python
                    heappush(open_set, (f_score[neighbor], neighbor))

        return []

    def _heuristic(self, state_index):
        """ Calculate Manhattan distance heuristic to goal """
        state = self.state_from_index(state_index)
        goal = self.state_from_index(self.goal_index)
        return abs(goal[0] - state[0]) + abs(goal[1] - state[1])

    def _reconstruct_path(self, came_from, current):
        """ Reconstruct path from start to goal """
        path = [self.state_from_index(current)]
        while current in came_from:
            current = came_from[current]
            path.append(self.state_from_index(current))
        return path[::-1]

    # Step 2 part ii code
    def a_star_step2_ii(self, epsilon=1):
        """ Perform A* search with a greedy heuristic adjusted by epsilon """
        start, goal = self.start_index, self.goal_index

        open_set = [(0, start)]
        came_from, g_score, f_score = {}, {start: 0}, {start: self._greedy_heuristic(start, epsilon)}

        while open_set:
            _, current = heappop(open_set)

            if current == goal:
                return self._reconstruct_path(came_from, current), len(came_from)

            for neighbor in self.get_neighbors(current):
                tentative_g_score = g_score[current] + 1

                if neighbor not in g_score or tentative_g_score < g_score[neighbor]:
                    came_from[neighbor] = current
                    g_score[neighbor] = tentative_g_score
                    f_score[neighbor] = g_score[neighbor] + self._greedy_heuristic(neighbor, epsilon)
                    heappush(open_set, (f_score[neighbor], neighbor))

        return [], 0  # No path found

    def _greedy_heuristic(self, state_index, epsilon):
        """ Calculate the greedy heuristic to the goal """
        state = self.state_from_index(state_index)
        goal = self.state_from_index(self.goal_index)
        return epsilon * (abs(goal[0] - state[0]) + abs(goal[1] - state[1]))
```

```python

    def run_with_epsilon_decay(self, initial_epsilon=10, time_limits=[0.05, 0.25, 1]):
        for time_limit in time_limits:
            epsilon = initial_epsilon
            start_time = time.time()
            print(f"Running with time limit {time_limit}s and initial epsilon {epsilon}")

            while time.time() - start_time < time_limit:
                path, expanded_nodes = self.a_star_step2_ii(epsilon)
                if path:  # If path found
                    break
                # Decay epsilon
                epsilon = max(1, epsilon - 0.5 * (epsilon - 1))

            print(f"Epsilon: {epsilon}, Nodes expanded: {expanded_nodes}, Path length: {len(path)}")

    # Step 2 part iii code
    def rrt(self, max_iter=1000, step_size=1):
            def sample_free():
                while True:
                    if random.random() < 0.2:
                        return np.array(self.goal_state)
                    else:
                        x = random.randint(0, self.cols - 1)
                        y = random.randint(0, self.rows - 1)
                        if self.check_occupancy((x, y)) == 0:
                            return np.array([x, y])

            def nearest_node(tree, sample):
                nbrs = NearestNeighbors(n_neighbors=1, algorithm='ball_tree').fit(tree)
                _, idx = nbrs.kneighbors([sample])
                return tree[idx[0][0]]

            def steer(from_node, to_node, step_size):
                vector = to_node - from_node
                distance = np.linalg.norm(vector)
                if distance <= step_size:
                    return to_node
                return from_node + vector / distance * step_size

            # Initialize tree and time tracking
            tree = np.array([self.start_state])
            parent_map = {}
            start_time = time.time()

            for i in range(max_iter):
                sample = sample_free()
                nearest = nearest_node(tree, sample)
                new_node = steer(nearest, sample, step_size)
```

```
                if self.check_occupancy(nearest):
                    continue  # Skip if new node hits an obstacle

                tree = np.vstack([tree, new_node])
                parent_map[tuple(new_node)] = tuple(nearest)

                if np.linalg.norm(new_node - self.goal_state) <= 1:
                    path = [tuple(new_node)]
                    while tuple(nearest) != tuple(self.start_state):
                        nearest = parent_map[tuple(nearest)]
                        path.append(tuple(nearest))
                    path.reverse()

                    # Calculate path length
                    path_length = sum(np.linalg.norm(np.array(path[i+1]) - np.array(path[i])) for i in range(len(path) - 1))
                    runtime = time.time() - start_time  # Compute runtime
                    return path, path_length, runtime  # Return the correct number of values

                if i % 100 == 0:
                    print(f"Iteration {i}/{max_iter}, Tree size: {len(tree)}")

            # Return an empty path if no valid path is found
            runtime = time.time() - start_time  # Compute runtime after all iterations
            return [], 0, runtime  # Return the correct number of values

# Step 3 code
class Maze4D(Maze):
    """ Maze4D Class """

    def __init__(self, maze_array, start_state=None, goal_state=None, max_vel=2):
        super().__init__(maze_array, start_state, goal_state)

        self.max_vel = max_vel

        if start_state is None:
            start_state = np.array((0, 0, 0, 0))
        self.start_state = start_state
        self.start_index = self.index_from_state(self.start_state)

        if goal_state is None:
            goal_state = np.array((self.cols - 1, self.rows - 1, 0, 0))
        self.goal_state = goal_state
        self.goal_index = self.index_from_state(self.goal_state)

    def index_from_state(self, state):
        """ Gets a unique index for the state """
        velocities = self.max_vel + 1
        return state[3] * self.rows * self.cols * velocities + \
```

```python
               state[2] * self.rows * self.cols + \
               state[0] * self.rows + \
               state[1]

    def state_from_index(self, state_id):
        """ Returns the state at a given index """
        velocities = self.max_vel + 1
        idx = state_id
        dy = int(np.floor(idx / (self.rows * self.cols * velocities)))
        idx -= dy * self.rows * self.cols * velocities
        dx = int(np.floor(idx / (self.rows * self.cols)))
        idx -= dx * self.rows * self.cols
        x = int(np.floor(idx / self.rows))
        y = idx % self.rows
        return (x, y, dx, dy)

    def get_neighbors(self, state_id):
        """ Returns a List of indices corresponding to neighbors of a given state in the 4D maze """
        state = self.state_from_index(state_id)
        neighbors = []

        deltas = [
            [-1, 0], [1, 0],  # horizontal
            [0, -1], [0, 1],  # vertical
            [-1, -1], [1, 1],  # diagonal
            [0, 0]  # no movement
        ]

        for delta in deltas:
            new_dx = state[2] + delta[0]
            new_dy = state[3] + delta[1]

            if new_dx < 0 or new_dx > self.max_vel or new_dy < 0 or new_dy > self.max_vel:
                continue

            if not self.check_hit(state[0:2], [new_dx, new_dy]):
                new_state = (state[0] + new_dx, state[1] + new_dy, new_dx, new_dy)
                neighbors.append(self.index_from_state(new_state))

        return neighbors

    @staticmethod
    def a_star_search_step3_i(maze, start_state, goal_state):
        """ A* search for solving the maze """
        def heuristic(state, goal_state):
            return abs(state[0] - goal_state[0]) + abs(state[1] - goal_state[1])

        def reconstruct_path(came_from, current_state):
            path = [current_state]
```

```
            while current_state in came_from:
                current_state = came_from[current_state]
                path.insert(0, current_state)
            return path

        open_set = []
        heapq.heappush(open_set, (0, tuple(start_state)))
        came_from = {}
        g_score = {tuple(start_state): 0}
        f_score = {tuple(start_state): heuristic(start_state, goal_state)}

        while open_set:
            _, current_state = heapq.heappop(open_set)

            if current_state == tuple(goal_state):
                return reconstruct_path(came_from, current_state)

            for neighbor in maze.get_neighbors(maze.index_from_state(current_state)):
                neighbor_state = maze.state_from_index(neighbor)
                tentative_g_score = g_score[tuple(current_state)] + 1

                if tuple(neighbor_state) not in g_score or tentative_g_score < g_score[tuple(neighbor_state)]:
                    came_from[tuple(neighbor_state)] = current_state
                    g_score[tuple(neighbor_state)] = tentative_g_score
                    f_score[tuple(neighbor_state)] = tentative_g_score + heuristic(neighbor_state, goal_state)
                    heapq.heappush(open_set, (f_score[tuple(neighbor_state)], tuple(neighbor_state)))

        return None  # No path found

    def a_star_step3_ii(self, epsilon=1):
        """
        Perform A* search with a greedy heuristic adjusted by epsilon.
        The cost is based on the heuristic scaled by epsilon for the greedy part.
        """
        start = self.start_index
        goal = self.goal_index

        # Priority queue (min-heap)
        open_set = []
        heappush(open_set, (0, start))  # (f_score, state_index)

        # Dictionaries for tracking costs and paths
        came_from = {}
        g_score = {start: 0}
        f_score = {start: self._greedy_heuristic(start, epsilon)}

        while open_set:
            _, current = heappop(open_set)
```

```python

            # If we reach the goal, reconstruct the path
            if current == goal:
                return self._reconstruct_path(came_from, current), len(came_from)

            # Process neighbors
            for neighbor in self.get_neighbors(current):
                tentative_g_score = g_score[current] + 1  # Assume uniform cost for grid steps

                if neighbor not in g_score or tentative_g_score < g_score[neighbor]:
                    # Update cost to reach neighbor
                    came_from[neighbor] = current
                    g_score[neighbor] = tentative_g_score
                    f_score[neighbor] = g_score[neighbor] + self._greedy_heuristic(neighbor, epsilon)

                    # Push to priority queue
                    heappush(open_set, (f_score[neighbor], neighbor))

        return [], 0  # If no path is found

    def _greedy_heuristic(self, state_index, epsilon):
        """ Calculate the greedy heuristic to the goal, scaled by epsilon """
        state = self.state_from_index(state_index)
        goal = self.state_from_index(self.goal_index)
        # Manhattan distance heuristic for 4D space, can adjust it for more complex calculations
        return epsilon * (abs(goal[0] - state[0]) + abs(goal[1] - state[1]) + abs(goal[2] - state[2]) + abs(goal[3] - state[3]))

    def _reconstruct_path(self, came_from, current):
        """ Reconstruct path from start to goal """
        path = [self.state_from_index(current)]
        while current in came_from:
            current = came_from[current]
            path.append(self.state_from_index(current))
        return path[::-1]  # Reverse the path

    def run_with_epsilon_decay(self, initial_epsilon=10, time_limits=[0.05, 0.25, 1]):
        for time_limit in time_limits:
            epsilon = initial_epsilon
            start_time = time.time()
            #print(f"Running with time limit {time_limit}s and initial epsilon {epsilon}")

            while True:
                #print(f"Running A* with epsilon={epsilon}")
                path, expanded_nodes = self.a_star_step3_ii(epsilon)

                elapsed_time = time.time() - start_time

                # Check for time limit
```

```python
                if elapsed_time >= time_limit:
                    print(f"Time limit {time_limit}s reached.")
                    break

                if path:  # If a path was found
                    #print(f"Found path with {expanded_nodes} nodes expanded.")
                    #print(f"Path length: {len(path)}")
                    break

                # Decay epsilon
                new_epsilon = epsilon - 0.5 * (epsilon - 1)
                if new_epsilon < 1.001:
                    new_epsilon = 1
                epsilon = new_epsilon

            print(f"Initial epsilon: {epsilon}, Nodes expanded: {expanded_nodes}, Path length: {len(path)}, Elapsed time: {elapsed_time:.4f}, Time limit: {time_limit}s")

if __name__ == "__main__":
    mazes = [
        (Maze2D.from_pgm('maze1.pgm'), "Maze 1"),
        (Maze2D.from_pgm('maze2.pgm'), "Maze 2"),
    ]

    # Setup and preprocessing for each maze
    for maze, maze_name in mazes:
        maze.start_state = (1, 1)
        maze.goal_state = (maze.cols - 1, maze.rows - 1)
        maze.start_index = maze.index_from_state(maze.start_state)
        maze.goal_index = maze.index_from_state(maze.goal_state)

    # Step 2 part i: Run A* on each maze
    for maze, maze_name in mazes:
        path = maze.a_star_step2_i()
        print(f"Found path for {maze_name}")
        maze.plot_path(path, f"A* Solution: {maze_name}")

    # Step 2 part ii: Run with epsilon decay on each maze
    for maze, maze_name in mazes:
        print(f"Testing {maze_name} with epsilon decay...")
        maze.run_with_epsilon_decay(initial_epsilon=10, time_limits=[0.05, 0.25, 1])

    # Step 2 part iii: Run RRT on each maze
    for maze, maze_name in mazes:
        print(f"Running RRT with step size 1 and max iterations 1000 on {maze_name}...")
        path, length, runtime = maze.rrt(max_iter=1000, step_size=1)
        if path:
            print(f"RRT Path Found for {maze_name}. Path Length: {length}, Runtime: {runtime} seconds.")
            maze.plot_path(path, f"{maze_name} RRT Path", runtime=runtime, path_length=length)
```

```
        else:
            print(f"RRT failed to find a path for {maze_name}")

    # Step 3
    mazes_4D = [
        (Maze4D.from_pgm('maze1.pgm'), "Maze 1 4D"),
        (Maze4D.from_pgm('maze2.pgm'), "Maze 2 4D")
    ]
    # Setup and preprocessing for each 4D maze
    for maze, maze_name in mazes_4D:
        # Test index_from_state and state_from_index for consistency
        for x in range(maze.cols):
            for y in range(maze.rows):
                for dx in range(3):
                    for dy in range(3):
                        state = (x, y, dx, dy)
                        assert maze.state_from_index(maze.index_from_state(state)) == state, f"Mapping incorrect for state: {state}"

    for maze, maze_name in mazes_4D:
        # Step 3 part i: Solve using A* search
        path = Maze4D.a_star_search_step3_i(maze, maze.start_state, maze.goal_state)
        if path:
            maze.plot_path(path, f'A* Solution: {maze_name}')
        else:
            print("No path found!")

    for maze, maze_name in mazes_4D:
        # Step 3 part ii: Run with epsilon decay on each 4D maze
        print(f"Testing {maze_name} with epsilon decay...")
        maze.run_with_epsilon_decay(initial_epsilon=1, time_limits=[0.05, 0.25, 1])
```