Nathan Butler
MS Robotics
Oregon State University

Feb. 3, 2025

SEQUENTIAL DECISION MAKING — Assignment 1

1 Search-Based Planning

1.1 Backward/Forward Value Iteration

Backward Value Iteration: Let $V(X)$ denote the value of state X and $c(X, Y)$ define the cost to move from state X to state Y. Values are computed in reverse topological order. Assume that 6 is the goal state, but not a terminal state (so I consider edge (6,6).

- $V(6) = min\{c(6,6)\} = 1$
- $V(5) = min\{c(5,6) + V(6)\} = min\{5+1\} = 6$
- $V(4) = min\{c(4,6) + V(6)\} = min\{1+1\} = 2$
- $V(3) = min\{c(3,5) + V(5), c(3,4) + V(4)\} = min\{4+6, 3+2\} = 5$
- $V(2) = min\{c(2,3) + V(3), c(2,4) + V(4)\} = min\{2+5, 1+2\} = 3$
- $V(1) = min\{c(1,2) + V(2), c(1,3) + V(3), c(1,5) + v(5)\} = min\{3+3, 2+5, 1+6\} = 6$

Forward Value Iteration: Let $V(X)$ denote the value of state X and $c(X, Y)$ define the cost to move from state X to state Y. Values are computed in topological order. We initialize the value of all states but the starting state to infinity.

- $V(1) = 0$
- $V(2) = min\{V(2), V(1) + c(1,2)\} = min\{\infty, 0+3\} = 3$
- $V(3) = min\{V(3), V(1) + c(1,3)\} = min\{\infty, 0+2\} = 2$
- $V(5) = min\{V(5), V(1) + c(1,5)\} = min\{\infty, 0+1\} = 1$
- $V(3) = min\{V(3), V(2) + c(2,3)\} = min\{2, 3+2\} = 2$
- $V(4) = min\{V(4), V(2) + c(2,4)\} = min\{\infty, 3+1\} = 4$
- $V(4) = min\{V(4), V(3) + c(3,4)\} = min\{4, 2+3\} = 4$
- $V(5) = min\{V(5), V(3) + c(3,5)\} = min\{1, 2+4\} = 1$
- $V(6) = min\{V(6), V(4) + c(4,6)\} = min\{\infty, 4+1\} = 5$
- $V(6) = min\{V(6), V(5) + c(5,6)\} = min\{5, 1+5\} = 5$

1.2 Admissible Heuristics of 6-DOF Arm

Admissible heuristics (should be $\leq$ the cost of the optimal solution):

- Euclidean distance in 6DOF space from current node to goal
- Euclidean distance in 6DOF space from current node to goal divided by 2
- Sum of total 6DOF distance traveled by optimal solution to goal state from current node

Multiplying euclidean distance by 2 could lead to an overestimate, so is not admissible. The euclidean distance will be less than the actual distance traveled by each joint, so it is admissible, as is a heuristic that divides it by 2. The optimal solution will also not overestimate, so it is admissible as well.

1.3 Informed Heuristics

Order (most to least informed):

- Sum of total 6DOF distance traveled by optimal solution to goal state from current node
- Euclidean distance in 6DOF space from current node to goal
- Euclidean distance in 6DOF space from current node to goal divided by 2

More informed heuristics more accurately predict the actual cost-to-go value of a state. With this, the actual optimal solution is as informed as possible, followed by euclidean distance. Euclidean distance divided by 2 will greatly underestimate the actual cost-to-go, so it is the least informed of the three.

2 Programming: Step 1

Fig. 1 presents the flowcharts designed for my A* and RRT implementations.

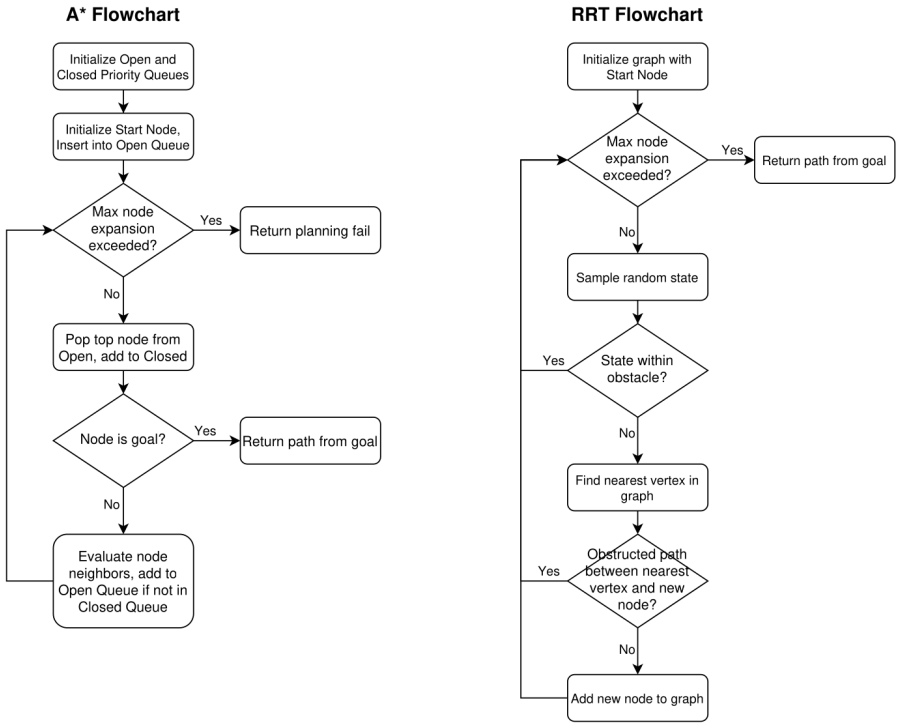

Figure 1: A* and RRT flowcharts

3 Programming: Step 2

3.1 A* with Admissible Heuristic

I implemented A* with a euclidean distance heuristic for estimating cost-to-go from a given state to the goal state. Eq. 3-1 computes this heuristic value h for a given state s composed of i components (such as x, y coordinates).

$$h(s) = \sqrt{\sum_{i \in s} (g_i - s_i)^2} \tag{3-1}$$

Fig. 2 shows the resulting path found on maze 1 with this heuristic, and Fig. 3 shows the final path on maze 2.

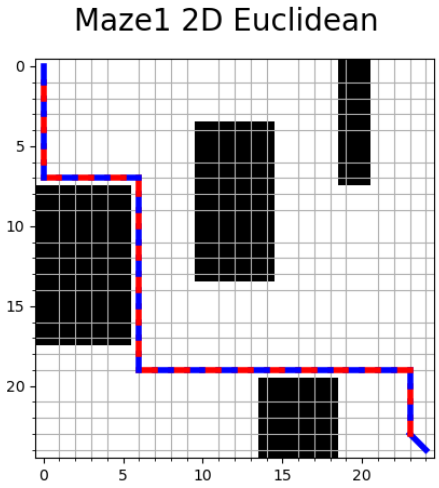

Figure 2: A* example result on maze 1

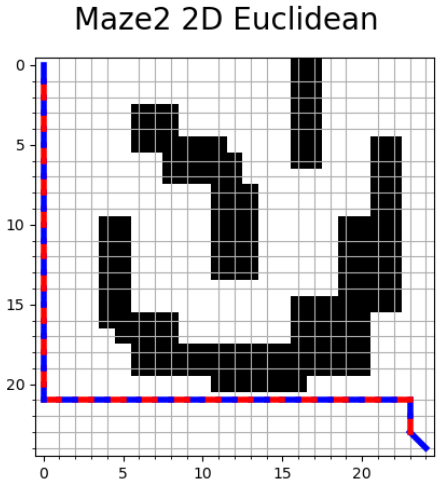

Figure 3: A* example result on maze 2

3.2 Map 1 Decaying Epsilon Results

Tables 1, 2, and 3 display the experimental data for the decaying epsilon tests on maze 1.

Epsilon	Nodes Expanded	Path Length
5.5	129	48.0
3.25	129	48.0
2.125	129	48.0
1.5625	129	48.0
1.28125	464	48.0
1.140625	652	48.0
1.0703125	684	48.0

Table 1: Maze 1 Decaying epsilon results for 0.05s

Epsilon	Nodes Expanded	Path Length
5.5	129	48.0
3.25	129	48.0
2.125	129	48.0
1.5625	129	48.0
1.28125	464	48.0
1.140625	652	48.0
1.0703125	684	48.0
1.03515625	708	48.0
1.017578125	719	48.0
1.0087890625	717	48.0
1.00439453125	723	48.0
1.002197265625	723	48.0
1.0010986328125	727	48.0
1.0	755	48.0

Table 2: Maze 1 Decaying epsilon results for 0.025s

Epsilon	Nodes Expanded	Path Length
5.5	129	48.0
3.25	129	48.0
2.125	129	48.0
1.5625	129	48.0
1.28125	464	48.0
1.140625	652	48.0
1.0703125	684	48.0
1.03515625	708	48.0
1.017578125	719	48.0
1.0087890625	717	48.0
1.00439453125	723	48.0
1.002197265625	723	48.0
1.0010986328125	727	48.0
1.0	755	48.0

Table 3: Maze 1 Decaying epsilon results for 1.0s

3.3 Map 2 Decaying Epsilon Results

Tables 4, 5, and 6 display the experimental data for the decaying epsilon tests on maze 2.

Epsilon	Nodes Expanded	Path Length
5.5	318	70.0
3.25	332	52.0
2.125	296	50.0
1.5625	505	50.0
1.28125	476	48.0

Table 4: Maze 2 Decaying epsilon results for 0.05s

Epsilon	Nodes Expanded	Path Length
5.5	318	70.0
3.25	332	52.0
2.125	296	50.0
1.5625	505	50.0
1.28125	476	48.0
1.140625	579	48.0
1.0703125	608	48.0
1.03515625	649	48.0
1.017578125	653	48.0
1.0087890625	660	48.0
1.00439453125	662	48.0
1.002197265625	663	48.0
1.0010986328125	663	48.0

Table 5: Maze 2 Decaying epsilon results for 0.025s

Epsilon	Nodes Expanded	Path Length
5.5	318	70.0
3.25	332	52.0
2.125	296	50.0
1.5625	505	50.0
1.28125	476	48.0
1.140625	579	48.0
1.0703125	608	48.0
1.03515625	649	48.0
1.017578125	653	48.0
1.0087890625	660	48.0
1.00439453125	662	48.0
1.002197265625	663	48.0
1.0010986328125	663	48.0
1.0	692	48.0

Table 6: Maze 2 Decaying epsilon results for 1.0s

3.4 RRT

Figs. 4 and 5 present example tours solved with RRT for maze 1 and maze 2, respectively. The path length solved for the maze 1 sample was 37.46s, and the running time was 0.76s. The path length solved for the

maze 2 sample was 44.14s, and the running time was 0.46s. These values were found for an RRT running up to 1000 iterations with a 5% biased goal sampling rate.

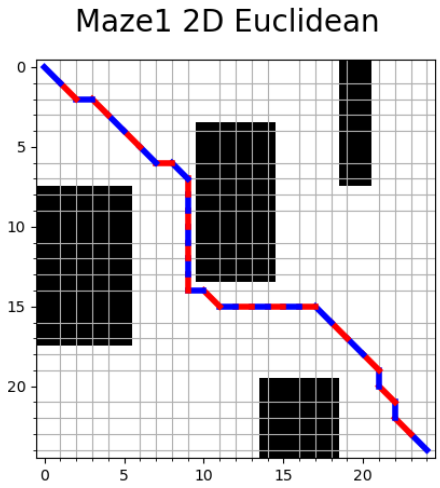

Figure 4: RRT example result on maze 1

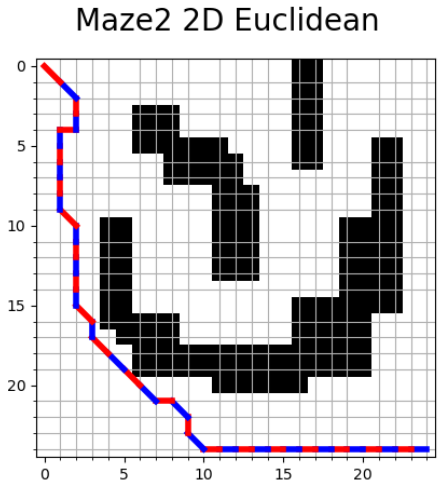

Figure 5: RRT example result on maze 2

4 Programming: Step 3

4.1 4D A* with Informed and Admissible Heuristic

For the time cost version of the problem, I implemented a heuristic that first evaluates the euclidean distance between a state and the goal using Eq. 3-1, then divides that distance by the agent's maximum possible velocity v_{max}. Formally, we have

$$t(s) = h(s)/v_{max}. \tag{4-1}$$

Intuitively, the value represents the time cost that would be incurred if the agent immediately began moving at maximum velocity in a direct path to the goal, and was able to stop immediately once the goal was reached. As the agent must accelerate and decelerate gradually and the euclidean distance is an admissible heuristic, this method is also admissible. Additionally, this heuristic is informed because the optimal solution should see the agent traveling as fast as it can for the majority of its path, while also following the shortest path.

Fig. 6 shows the resulting path found on maze 1, and Fig. 7 shows the final path on maze 2.

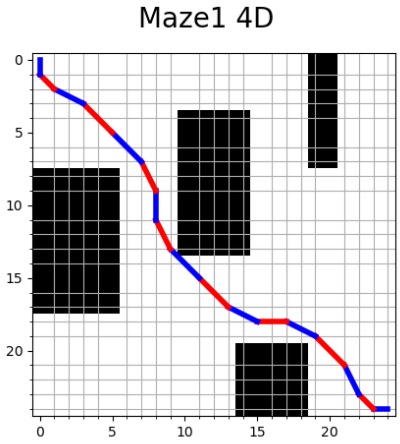

Figure 6: A* example result on 4D maze 1

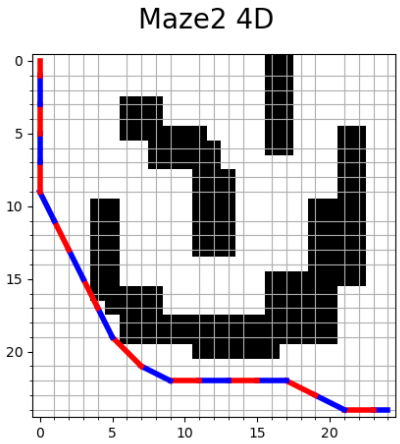

Figure 7: A* example result on 4D maze 2

4.2 4D Map 1 Decaying Epsilon Results

Tables 7, 8, and 9 display the experimental data for the decaying epsilon tests on maze 1.

Epsilon	Nodes Expanded	Path Length
5.5	62	19
3.25	62	19
2.125	62	19
1.5625	89	19
1.28125	224	19
1.140625	564	19
1.0703125	2337	20

Table 7: 4D Maze 1 Decaying epsilon results for 0.05s

Epsilon	Nodes Expanded	Path Length
5.5	62	19
3.25	62	19
2.125	62	19
1.5625	89	19
1.28125	224	19
1.140625	564	19
1.0703125	2337	20
1.03515625	2844	20
1.017578125	567	18
1.0087890625	608	18

Table 8: 4D Maze 1 Decaying epsilon results for 0.025s

Epsilon	Nodes Expanded	Path Length
5.5	62	19
3.25	62	19
2.125	62	19
1.5625	89	19
1.28125	224	19
1.140625	564	19
1.0703125	2337	20
1.03515625	2844	20
1.017578125	567	18
1.0087890625	608	18
1.00439453125	774	18
1.002197265625	776	18
1.0010986328125	803	18
1.0	806	18

Table 9: 4D Maze 1 Decaying epsilon results for 1.0s

4.3 4D Map 2 Decaying Epsilon Results

Tables 10, 11, and 12 display the experimental data for the decaying epsilon tests on maze 2.

Epsilon	Nodes Expanded	Path Length
5.5	556	23
3.25	561	23

Table 10: 4D Maze 2 Decaying epsilon results for 0.05s

Epsilon	Nodes Expanded	Path Length
5.5	556	23
3.25	561	23
2.125	564	22
1.5625	635	23
1.28125	1214	21
1.140625	1650	21
1.0703125	2131	21

Table 11: 4D Maze 2 Decaying epsilon results for 0.025s

Epsilon	Nodes Expanded	Path Length
5.5	556	23
3.25	561	23
2.125	564	22
1.5625	635	23
1.28125	1214	21
1.140625	1650	21
1.0703125	2131	21
1.03515625	2367	21
1.017578125	2490	21
1.0087890625	2534	21
1.00439453125	2559	21
1.002197265625	2571	21
1.0010986328125	2576	21
1.0	2582	21

Table 12: 4D Maze 2 Decaying epsilon results for 1.0s

5 Discussion Questions

5.1 1: CTMP

The CTMP algorithm is an effective approach to online planning and replanning problems that require constant-time operation constraints. Such problems include characteristics such as dynamics or uncertainty that limit the effectiveness of a precomputed plan at runtime. Because of these features, any solution approach must have the ability to adapt to new information at runtime. CTMP provides a pipeline for doing so that uses an initial preprocessing stage to generate a set of valid paths followed by a query stage that attempts to use these plans at runtime to reduce computation time. This approach could be applied to the 4D grid problem by initializing a set of feasible paths and then allowing an agent to adapt between these paths at runtime.

The CTMP algorithm could be applied to a variety of real-world scenarios. Any robot operating outside of a laboratory setting is likely to encounter environmental dynamics or uncertainty. For example, a delivery robot may be aware of its start and goal locations relative to a campus environment, but human traffic imposes dynamic constraints that the robot will need to adapt to during deployment. Similarly, an aquatic surface vehicle navigating to waypoints in the ocean is likely to encounter wind and ocean currents that push it off of any initial trajectories that it selects. In both instances, CTMP could be applied to preprocess a set of feasible paths and then adapt online.

5.2 2: RRT-Connect

With the increased dimensionality of the 4D grid problem, RRT-Connect would be an improved approach compared to base RRT. As RRT performs random sampling over the state space, it can struggle to find good solutions in high-dimensional spaces such as the 4D grid. RRT-Connect works to reduce the search space by expanding trees both from the start and goal states using a greedy strategy to extend trees toward each as far as possible at each step. In a sense, this algorithm is more biased towards exploitation than RRT, which specializes in pure state space exploration.

However, RRT-Connect suffers from the same lacking optimality guarantees as RRT, and in the 4D search space may still be inefficient. Unlike RRT*, both RRT approaches do not support path rewiring, thus preventing asymptotic optimality guarantees. Additionally, additional state space guidance may be necessary to enable RRT-Connect to efficiently find near-optimal solutions

5.3 3: D*-Lite

If the robot discovered the environment as it moved towards the goal, it would be necessary to efficiently recompute portions of the original A* path. D*-Lite provides a means for doing so by recomputing the path only within affected areas when new observations are obtained. After computing an initial path with A*, the robot begins to execute it. If the robot discovers an unexpected environmental feature along the path, it updates the relevant edge costs and recomputes only the portion of the path affected. The robot would then resume executing the path until a new disturbance is discovered or the goal is reached.

5.4 4: RRBT

If the robot's position was uncertain, I would update the RRT algorithm to consider a chance-constrained version of the planning problem. To do this, I would need to evaluate the probability of obstacle collisions while constructing the RRT. This could be accomplished by sampling a number of trajectories when considering a new node to add to the graph. From the sampled set of trajectories, I can obtain a collision probability according to the number of unsuccessful paths in my sample. If this probability exceeds my chance constraint, I would not add the new node to the graph. By using this criteria for expanding the RRT, I can ensure that the final solution will succeed within the bounds of the chance constraint.

6 Code for 2D A*

```
import time
import numpy as np
from maze import Maze, Maze2D
from priority_queue import PriorityQueue

class Node():

    def __init__(self,
                 id,
                 state,
                 parent,
                 epsilon=1.0
                 ):

        self.id = id
        self.state = state
        self.parent = parent
        self.epsilon=epsilon
        self.cost_to_come = 0
        self.cost_to_go = 0
        self.cost = 0

    def compute_cost(self, goal_state):
        if self.parent == None:
            return
        self.cost_to_come = self.parent.cost_to_come + np.linalg.norm(np.asarray(self.state)-np.asarray(

        self.cost_to_go = self._euclidean_heuristic(goal_state)
        self.cost = self.cost_to_come + self.cost_to_go

    def _euclidean_heuristic(self, goal_state):
        """Straight line path from current state to goal state"""
        return self.epsilon*np.linalg.norm(goal_state-np.asarray(self.state))

    def get_path(self):
        path = [self.state]
        node = self.parent
        while node.parent is not None:
            node = node.parent
            path.append(node.state)
        return np.array(path)

    def __str__(self):
        return "Node"+str(self.id)

    def __repr__(self):
        return "Node"+str(self.id)

    def __eq__(self, other):
        if isinstance(other, Node):
            return self.id == other.id
        return False
```

```
    def __hash__(self):
        return hash(self.id)


def a_star_experiments(m: Maze,
                       max_expansion=10000,
                       timeout=1.0,
                       epsilon=10,
                       ) -> np.array:

    start_time = time.time()
    running_time = 0.0
    completion_log = [] #epsilon, node_count, path_length
    last_cycle = False
    while running_time < timeout and not last_cycle:
        if epsilon == 1.0: last_cycle = True

        # === A* SEARCH LOOP ===
        node_ct, path_len, path = solve_a_star(m, max_expansion, epsilon)
        if path_len:
            completion_log.append([epsilon, node_ct, path_len])

        # === EPSILON DECAY AND OTHER BOOKEEPING ===
        epsilon -= 0.5*(epsilon-1)
        if epsilon < 1.001: epsilon = 1.0
        running_time = time.time() - start_time

    return completion_log

def solve_a_star(m: Maze2D,
                 max_expansion=10000,
                 epsilon=10,
                 ):

    # === A* SEARCH LOOP ===

    # Init open and closed node lists
    pq_open = PriorityQueue()
    pq_closed = PriorityQueue()

    goal = np.asarray(m.state_from_index(m.get_goal()))

    # Compute start node heuristic value
    node = Node(m.get_start(),
                m.state_from_index(m.get_start()),
                parent=None,
                epsilon=epsilon
                )
    node.compute_cost(goal)

    # Load start node in open list
    pq_open.insert(node, node.cost)
```

```
    # Planning loop
    count = 0
    # path_done = False
    while count < max_expansion: # and not path_done:

        # Pop top node from open list, put in closed list
        node = pq_open.pop()
        pq_closed.insert(node, 0)

        # If node is goal, create path from parents & return
        if node.id == m.get_goal():
            # path_done = True
            path_length = node.cost_to_come
            return count, path_length, node.get_path()

        # Explore node neighbors (that are not in closed list)
        for id in m.get_neighbors(node.id):

            neighbor = Node(id,
                        m.state_from_index(id),
                        parent=node,
                        epsilon=epsilon
                        )

            if pq_closed.test(neighbor):
                # Don't consider closed nodes
                continue
            else:
                count += 1
                # Compute neighbor heuristic values
                neighbor.compute_cost(goal)
                # Add neighbor to open list
                pq_open.insert(neighbor, neighbor.cost)

    # Return planning timeout
    print("Timeout reached, returning Fail")
    return count, False, False

if __name__ == "__main__":
    max_expansion = 10000
    epsilon=10
    timeout=0.05
    maze_id = 2

    m = Maze2D.from_pgm(f'maze{maze_id}.pgm')

    data = a_star_experiments(m, max_expansion, timeout, epsilon)

    print(data)

    _,_,path = solve_a_star(m, max_expansion, 1)
    m.plot_path(path, f'Maze{maze_id} 2D')
```

7 Code for 2D RRT

```
import time
import numpy as np
from maze import Maze, Maze2D, Maze4D

class Node():

    def __init__(self,
                 id,
                 state,
                 parent=None
                 ):

        self.id = id
        self.state = state
        self.parent = parent


    def get_path(self):

        return []

    def __str__(self):
        return "Node"+str(self.id)

    def __repr__(self):
        return "Node"+str(self.id)

    def __eq__(self, other):
        if isinstance(other, Node):
            return self.id == other.id
        return False

    def __hash__(self):
        return hash(self.id)

def sample_maze(m: Maze, goal_prob) -> tuple:
    if np.random.rand() < goal_prob:
        # sample goal
        return m.state_from_index(m.get_goal())
    else:
        # do random sample
        x_samp = np.random.randint(m.cols)
        y_samp = np.random.randint(m.rows)
        return (x_samp, y_samp)

def get_nearest(sample: Node, node_list: list[Node])-> Node:
    nearest = node_list[0]
    np_sample = np.asarray(sample)
    min_dist = np.linalg.norm(np_sample-np.asarray(nearest.state))
    for node in node_list[1:]:
        dist = np.linalg.norm(np_sample-np.asarray(node.state))
        if dist < min_dist:
```

```
            nearest = node
            min_dist = dist

    return nearest

def step_toward_sample(node: Node, sample_state: tuple, step_size) -> tuple:
    if node.state == sample_state:
        # return false if same point
        return False

    np_sample = np.asarray(sample_state)
    np_orig = np.asarray(node.state)
    vec = np_sample-np_orig
    unit_vec = np.ceil((vec)/np.linalg.norm(vec))
    np_step_state = np.asarray(node.state) + step_size*unit_vec
    # print(f"Vec from {node.state} to {sample_state}: {vec}. Unit vec: {unit_vec}. New state: {tuple(np

    return tuple(np_step_state)

def solve_rrt_2D(m: Maze,
                 max_iters=1000,
                 goal_prob=0.05,
                 step_size=1.0
                 ):

    start_time = time.time()


    # === RRT ALGO START ===
    # Initialize graph
    node_list = [Node(m.get_start(),
                m.state_from_index(m.get_start()),
                parent=None,
                )]

    # Planning loop
    count = 0
    while count < max_iters:
        count += 1

        # Sample random state
        sample_state = sample_maze(m, goal_prob)

        # Check if in obstacle
        if m.check_occupancy(sample_state):
            continue

        # Find nearest vertex
        nearest_node = get_nearest(sample_state, node_list)

        # Step toward sampled point
        new_state = step_toward_sample(nearest_node, sample_state, step_size)

        # Attempt to connect
```

```
        if new_state and not m.check_hit(nearest_node.state,
                            np.asarray(new_state)-np.asarray(nearest_node.state)):
            node_list.append(Node(m.index_from_state(new_state),
                                  new_state,
                                  nearest_node,
                                  ))
    # === RRT END ===

    run_time = time.time() - start_time
    # Extract plan
    for node in node_list:
        if node.id == m.get_goal():
            # Goal node found, attempt to backtrack path
            path = [node.state]
            path_cost = 0
            while node.parent:
                path_cost += np.linalg.norm(np.asarray(node.state)- \
                                            np.asarray(node.parent.state))
                node = node.parent
                path.append(node.state)
            if node.id == m.get_start():
                # Complete path found
                print("Path complete!")
                return path, path_cost, run_time
            else:
                # Incomplete path
                print("Incomplete path")
                return path, path_cost, run_time

    # Goal never reached
    print("Goal state never sampled")
    return False, False, False


if __name__ == "__main__":
    max_iters = 1000
    goal_prob = 0.05
    step=1.0
    maze_id = 1

    m = Maze2D.from_pgm(f'maze{maze_id}.pgm')

    path, cost, run_time = solve_rrt_2D(m, max_iters, goal_prob, step)

    if path:
        print(f"Completed in {run_time}s | Cost: {cost}")
        m.plot_path(path, f'Maze{maze_id} 2D')
    else:
        print("No path found!")
```

8 Code for 4D A*

```
import time
import numpy as np
from maze import Maze, Maze4D
from priority_queue import PriorityQueue

class Node():

    def __init__(self,
                 id,
                 state,
                 parent,
                 epsilon=1.0
                 ):

        self.id = id
        self.state = state
        self.parent = parent
        self.epsilon=epsilon
        self.cost_to_come = 0 # previous steps
        self.cost_to_go = 0 # heuristic
        self.cost = 0 # f = g + h

    def compute_cost(self, goal_state, max_vel=2):
        """
        Minimize time
        """
        if self.parent == None:
            return
        self.cost_to_come = self.parent.cost_to_come + 1

        self.cost_to_go = self._time_heuristic(goal_state, m.max_vel)
        self.cost = self.cost_to_come + self.cost_to_go

    def _time_heuristic(self, goal_state, max_vel):
        """Minimum time to goal from current location; assumes max vel
        for entire straight line path."""
        return self.epsilon*np.linalg.norm(goal_state-np.asarray(self.state))/max_vel

    def get_path(self):
        path = [self.state]
        node = self.parent
        while node.parent is not None:
            node = node.parent
            path.append(node.state)
        return np.array(path)

    def __str__(self):
        return "Node"+str(self.id)

    def __repr__(self):
        return "Node"+str(self.id)
```

```
    def __eq__(self, other):
        if isinstance(other, Node):
            return self.id == other.id
        return False

    def __hash__(self):
        return hash(self.id)

def a_star_experiments(m: Maze,
                       max_expansion=10000,
                       timeout=1.0,
                       epsilon=10,
                       ) -> np.array:

    start_time = time.time()
    running_time = 0.0
    completion_log = [] #epsilon, node_count, path_length
    last_cycle = False
    while running_time < timeout and not last_cycle:
        if epsilon == 1.0: last_cycle = True

        # === A* SEARCH LOOP ===
        node_ct, path_time, path = solve_a_star(m, max_expansion, epsilon)
        if path_time:
            completion_log.append([epsilon, node_ct, path_time])

        # === EPSILON DECAY AND OTHER BOOKEEPING ===
        epsilon -= 0.5*(epsilon-1)
        if epsilon < 1.001: epsilon = 1.0
        running_time = time.time() - start_time

    return completion_log

def solve_a_star(m: Maze4D,
                 max_expansion=10000,
                 epsilon=10,
                 ):

    # === A* SEARCH LOOP ===

    # Init open and closed node lists
    pq_open = PriorityQueue()
    pq_closed = PriorityQueue()

    goal = np.asarray(m.state_from_index(m.get_goal()))

    # Compute start node heuristic value
    node = Node(m.get_start(),
                m.state_from_index(m.get_start()),
                parent=None,
                epsilon=epsilon
                )
    node.compute_cost(goal, m.max_vel)
```

```
    # Load start node in open list
    pq_open.insert(node, node.cost)

    # Planning loop
    count = 0
    # path_done = False
    while count < max_expansion: # and not path_done:

        # Pop top node from open list, put in closed list
        node = pq_open.pop()
        pq_closed.insert(node, 0)

        # If node is goal, create path from parents & return
        if node.id == m.get_goal():
            # path_done = True
            path_time = node.cost_to_come
            return count, path_time, node.get_path()

        # Explore node neighbors (that are not in closed list)
        for id in m.get_neighbors(node.id):

            neighbor = Node(id,
                        m.state_from_index(id),
                        parent=node,
                        epsilon=epsilon
                        )

            if pq_closed.test(neighbor):
                # Don't consider closed nodes
                continue
            else:
                count += 1
                # Compute neighbor cost + heuristic values
                neighbor.compute_cost(goal, m.max_vel)
                # Add neighbor to open list
                pq_open.insert(neighbor, neighbor.cost)

    # Return planning timeout
    print("Timeout reached, returning Fail")
    return count, False, False

if __name__ == "__main__":
    max_expansion = 10000
    epsilon=10
    timeout=0.05
    maze_id = 2

    m = Maze4D.from_pgm(f'maze{maze_id}.pgm')

    data = a_star_experiments(m, max_expansion, timeout, epsilon)

    print(data)

    _,_,path = solve_a_star(m, max_expansion, 1)
```

```
    m.plot_path(path, f'Maze{maze_id} 4D')
```