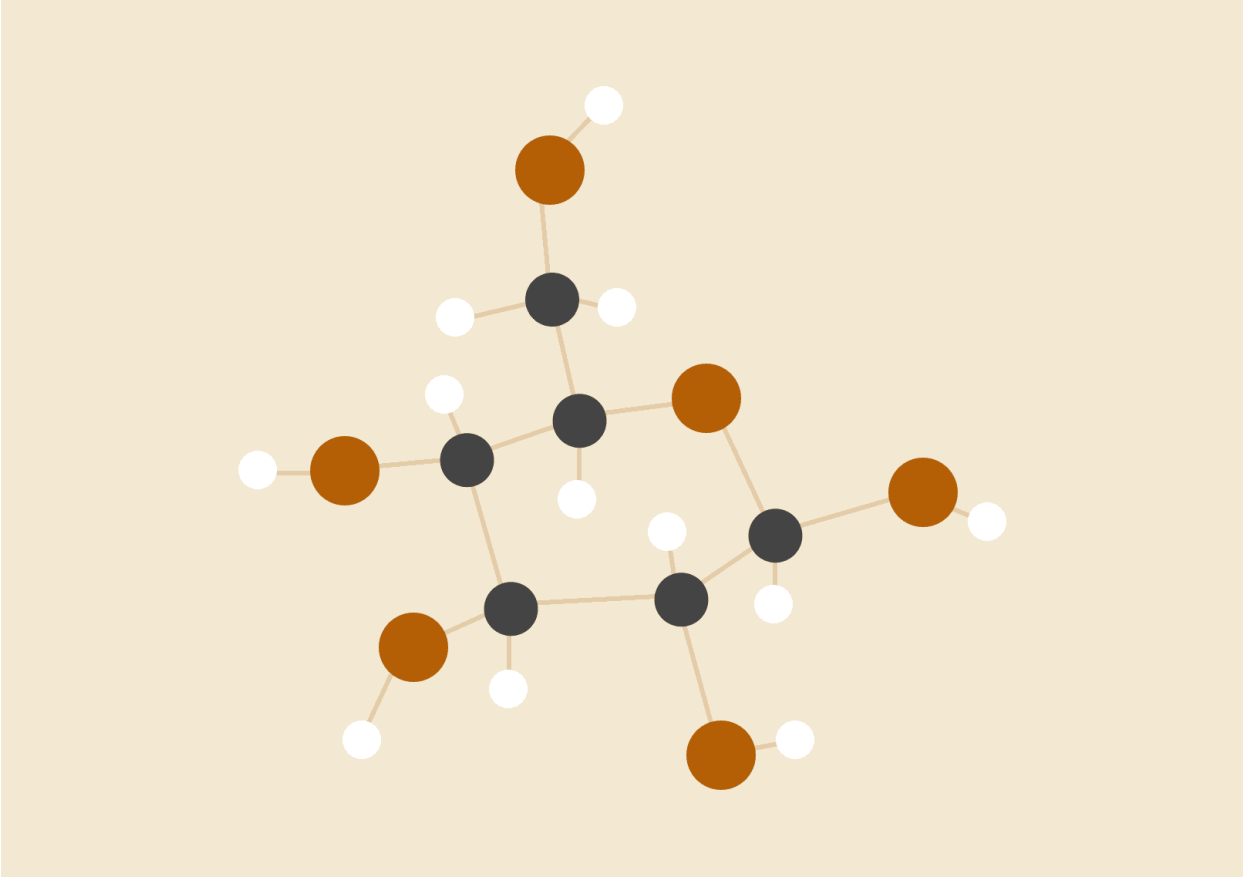
AI Assignment I



Name: Bingqi Xia

Student Id: 22300549

Maze Generator

To generate the maze, I reused the pyamaze package in python. It follows the MIT protocol for the source code. Code repository address: https://github.com/MAN1986/pyamaze.

I have wrapped a method in my MazeSolver base class that generates a random maze of given rows*cols size. And I modified the code of pyamaze so that it supports saving the maze in a specified file in order to use the same maze to test the performance of different search algorithms.

```
def generate_maze(self, rows, cols, fromfile=False, savePath=None):
    self.maze = maze(rows, cols)
    self.title = "%dx%d" % (rows, cols)
    if self.start is None:
        self.start=(self.maze.rows, self.maze.cols)

    if(fromfile and os.path.exists(savePath)):
        self.maze.CreateMaze(theme='light', loadMaze=savePath)
    elif savePath is not None:
        self.maze.CreateMaze(loopPercent=0, theme='light', saveMaze=True, savePath=savePath)
    else:
        self.maze.CreateMaze(loopPercent=0, theme='light', saveMaze=False)
    return self.maze
```

For each search algorithm, I stored the path of the final solution found in a dictionary and animated it by calling the tracePath method.

```
solution_agent = agent(self.maze, footprints=True, color=COLOR.red)
self.maze.tracePath({solution_agent: self.solution}, delay=100)
```

Regarding all the mazes chosen for the experiments, their entrances are at the bottom right corner of the maze, coordinates: (rows, cols), and the goal of the maze is at the top left corner, coordinates: (1, 1).

Algorithms Implementation

BFS

The main idea of the BFS algorithm is to start at a node of the maze and visit all the neighboring nodes of that node, layer by layer. This process is repeated recursively until all nodes are traversed or the goal is found.

I use the Queue in python to implement BFS. Firstly, put the entrance of the maze into a queue and mark it as visited. Then take out the head element of the queue and add all the neighboring nodes that are reachable and not yet visited into the queue. And mark these nodes as visited. Repeat this process until the queue is empty or the goal is found.

(Codes are in the appendix)

DFS

The main idea of the DFS algorithm is to start at a node in the maze and follow a pathway to the end. Then from the node at the end of this pathway backtrack to the previous node to continue following the unvisited path to the end. This process is repeated recursively until all nodes are traversed or the goal is found.

I use stack to implement DFS. Firstly, push the entrance of the maze into a stack and mark it as visited. Then take out the top element of the stack and push the reachable and unvisited neighboring nodes of it into the stack in turn, mark these nodes as visited. Repeat this process until the queue is empty or the goal is found.

(Codes are in the appendix)

A*

Unlike DFS and BFS, the A* algorithm is a heuristic search algorithm that uses a heuristic function to estimate the distance to the target node.

It uses two valuation functions:

a. g(n): the actual cost from the starting node to the current node
b. h(n): the estimated cost from the current node to the target node

Heuristic function: $f(n) = g(n) + h(n)$

For g(n), we can set g(n) at the starting position to 0 and +1 for each move.

For h(n), I have chosen the Manhattan distance between the current node and the target node for two reasons:

a. in a grid-like maze the agent can only move along straight or right-angle turning paths, and the Manhattan distance fits this situation better.
b. the calculation is simpler: to find the Manhattan distance you only need to calculate the sum of the distances between the two nodes in the horizontal and

vertical directions.

During the search, we choose the reachable node with a smaller f(n) to move each time. If two neighboring reachable nodes have the same f(n) we choose the node with a smaller h(n). This is because a smaller h(n) means that the node is closer to the target.

I use PriorityQueue in python to implement A*. Because the priority queue works in such a way that smaller elements have higher priority, we can combine node information and cost in a Tuple(f(node), h(node), node) to store it.

(Codes are in the appendix)

MDP Parameter Selection

The parameters of the MDP include the discount factor gamma, the state space, the action space, the state transfer function, and the reward function.

For the maze problem, the state space is all the cells of the maze and the action space is "N", "S", "E", and "W" is a direction that can be moved. These two parameters are determined.

For the discount factor gamma, which indicates the weight of the reward received at the current moment relative to the future reward, the closer gamma is to 0, the more the agent focuses on the immediate reward, and the closer gamma is to 1, the more the agent focuses on the future reward. For the maze problem, I would prefer the future reward to be weighted more heavily so that the path found by the agent is likely to be optimal overall. Therefore, I set the gamma to 0.9 in my experiments.

The state transfer function describes the probability that the agent will move from one state to the next after performing an action. As in real life, robots are subject to a number of environmental factors and do not always move in the expected direction of movement. To model this randomness, I set the state transfer function to have an 80% probability of moving forward as expected, a 10% probability of moving left, and a 10% probability of moving right.

The reward function describes the reward received for taking a certain action in a state. In the maze problem, I want the agent to avoid hitting the wall and find a shorter path. So my reward function is

a) If the direction of movement is a wall, keep the original state and get a reward of

-1.

b) If the direction of movement is a pathway, move to that node and get a reward of -0.3
c) The goal node has a reward of 1

MDP Value Iteration

The MDP Value Iteration algorithm approximates the optimal value function by iteratively computing the state value function.

Initially, the value function for each state is set to 0. In each iteration, the algorithm computes the value function of the next state by updating the value function of the current state. The process continues until the value function converges.

(Codes are in the appendix)

MDP Policy Iteration

The MDP Policy Iteration algorithm divides the problem into two steps.

a. Policy evaluation: Calculate a value function for each state based on the current policy.
b. Policy improvement: using the greedy algorithm to find the current optimal action based on the current value function and update the current state's policy to the optimal actions.

First set the policy for each state to one of action, then repeat the above two steps until the policy converges.

(Codes are in the appendix)

Performance Analysis

Mazes in Test

To test the performance of the different algorithms in solving the maze problem, I used three different sizes of random mazes: [30*30, 50*50, 70*70].

In addition, I also tried 10*10 and 100*100 mazes. When the maze is too small, the performance of different search algorithms cannot be clearly distinguished. When the maze is too large, it takes too much time for the MDP algorithm, and the animation of the visualization path takes too long. So in the end I chose [30*30, 50*50, 70*70].

In addition, the mazes generated in the test always have at least one path from the start node to the goal.

Metrics

To more fully measure the performance of each algorithm, I evaluate each algorithm using the following four metrics:

a. Time Cost: Measure the execution time of each algorithm, which theoretically is positively related to the time complexity of the algorithm. The shorter the time taken the better the performance.
b. Memory Usage: The memory size used by the algorithm when running, which theoretically is positively related to the space complexity of the algorithm. The smaller the memory footprint, the better the performance.
c. Length of found path: The shorter the path found, the better the performance
d. Optimality: For maze problems, a better performing algorithm should always find the shortest path

Experiments

In order to make the test results more accurate, for each size [30*30, 50*50, 70*70], l randomly generated 5 mazes with different paths and using the average value of the same metric obtained from 5 experiments as comparison data.

The following table shows the result:

Maze Size: 30*30

	BFS	DFS	A*	MDP-Value Iteration	MDP-Policy Iteration
Time Cost (ms)	10.118	0.348	3.501	1849.534	2167.573
Memory Usage (KB)	60.8	3.2	68	103.200	121.600

Length of found path	61	84	61	62	62
Optimality	Yes	No	Yes	No	No
Iteration Number	899	92	212	220	13

Maze size: 50*50

	BFS	DFS	A*	MDP-Value Iteration	MDP-Policy Iteration
Time Cost (ms)	20.352	1.179	8.5	5073.707	7158.996
Memory Usage (KB)	180	27.2	196.8	142.4	180
Length of found path	104	174	104	104	104
Optimality	Yes	No	Yes	Yes	Yes
Iteration Number	2497	275	512	220	14

Maze size: 70*70

	BFS	DFS	A*	MDP-Value Iteration	MDP-Policy Iteration
Time Cost (ms)	50.727	1.063	20.489	10782.570	21451.214
Memory Usage (KB)	193.6	56	156.000	252	255.2

Length of found path	144	277	144	144	144
Optimality	Yes	No	Yes	Yes	Yes
Iteration Number	4896	368	1175	220	15

Results Analysis

From the above experimental results, it can be seen that:

For a maze of the same size, **the average time spent on 5 experiments**,

DFS < A* < BFS < MDP Value Iteration < MDP Policy Iteration.

This is because the running speed of the DFS algorithm is affected by the width and depth of the search tree. In the worst case, the width of the search tree is 4, the depth is n*n. But there may be many solution paths in the generated maze, the actual search depth is very small. Through the number of iterations, we can also see that the iteration number of DFS is much smaller than BFS and A*.

For A* and BFS, although their search space size is the same in theory, since the A* algorithm selects the node which has the smallest manhattan distance with the goal node to move each time, the speed to reach the target is faster, so it takes less time than BFS.

For MDP value iteration, it updates the value function of all states in each iteration until the value function converges. This process takes a lot of computational resources and time, so it takes more time than the DFS, BFS, and A* algorithms.

For MDP policy iteration, it takes the longest time as it not only repeats the computation of MDP value iteration in each iteration but also updates the policy matrix with the computed value function matrix.

Regarding memory spend,

the DFS algorithm has the lowest memory usage because it only needs to maintain information about the state of the nodes on the current path. Its memory usage only

depends on the maximum depth and the current depth.

The BFS and A* algorithms need to maintain state information for the entire search space and therefore require more memory space than DFS. The difference in the size of memory usage between the two of them is small and is related to the structure of the specific maze.

For MDP Value Iteration, it needs to maintain an n*n matrix of value functions in memory and therefore takes more space than BFS and A*.

MDP Policy Iteration not only maintains the value functions matrix but also maintains the policy matrix and therefore takes up the most memory.

The length of the solution and Optimality:

The BFS and A* algorithms can always find the shortest path, they are optimal. The BFS algorithm always expands along the nearest neighbor node to the node, so the path it finds must be the shortest. For the A* algorithm, in the grid maze scene, since the most suitable Manhattan distance is selected as the heuristic function, it accurately estimates the distance from the target and always selects the node closest to the target. So the shortest path is also found.

The two MDP algorithms have the same performance, and they both find the shortest path in 50*50 mazes and 70*70 mazes. They almost are optimal. The MDP algorithm determines the optimal strategy by calculating the value function of each state. The paths it finds have optimal values, but are not necessarily the shortest.

The path found by the DFS algorithm is always the longest because the maze has many forks, the DFS algorithm always goes to the end along a certain path, and it is easy to miss the shortest path. It’s not optimal.

Conclusion

From the above discussion, it can be seen that DFS has the least time and space cost, but it is not optimal. BFS and A* can find the shortest path, but it requires more memory to store search space information. Due to the need to maintain and update the value function matrix in each iteration, MDP value Iteration takes more time and memory than

the first three algorithms, but it is better than MDP policy Iteration. MDP policy iteration has the worst performance because it maintains the value function matrix and the policy matrix at the same time, and updates them in each iteration. The MDP algorithm always finds the path of the optimal value, but not necessarily the shortest path.

Appendix

I BFS Implementation

```
class BFS(MazeSolver):
    def __init__(self, start=None):
        super().__init__(start)
        self.name = 'BFS'
    def solve(self):
        self.start_time = time.process_time()
        queue = Queue()
        visited_node = set()
        queue.put(self.start)
        visited_node.add(self.start)
        self.iterations = 0
        while not queue.empty():
            current_node = queue.get()
            if current_node == self.maze._goal:
                break
            self.iterations += 1
```

```
            neighbors = self.get_neighbors(current_node)
            for neighbor in neighbors:
                if neighbor not in visited_node:
                    queue.put(neighbor)
                    visited_node.add(neighbor)
                    self.saved_path[neighbor] = current_node
                    self.search_path.append(neighbor)
        self.end_time = time.process_time()
        self.reverse_path()
        return self.solution
```

II DFS Implementation

```
class DFS(MazeSolver):
    def __init__(self, start=None):
        super().__init__(start)
        self.name = 'DFS'
    def solve(self):
        self.iterations = 0
        self.start_time = time.process_time()
        stack = deque()
        visited_node = set()
        stack.append(self.start)
        visited_node.add(self.start)
```

```
        while stack:
            current_node = stack.pop()
            if(current_node == self.maze._goal):
                break

            self.iterations += 1
            self.search_path.append(current_node)
            neighbors = self.get_neighbors(current_node)
            for neighbor in neighbors:
                if neighbor not in visited_node:
                    stack.append(neighbor)
                    visited_node.add(neighbor)
                    self.saved_path[neighbor] = current_node
        self.end_time = time.process_time()
        self.reverse_path()
        return self.solution
```

III A* Implementation

```
class AStar(MazeSolver):
    def __init__(self, start=None):
        super().__init__(start)
        self.name = 'AStar'
```

```
    def manhattan(self, node1, node2):
        x1, y1 = node1
        x2, y2 = node2
        return abs(x1-x2) + abs(y1-y2)
    def solve(self):
        self.start_time = time.process_time()
        self.iterations = 0
        queue = PriorityQueue()
        # use manhattan distance as h(n), for the start position, g(n)=0. f(n) = h(n) + g(n)
        initalH = self.manhattan(self.start, self.maze._goal)
        queue.put((initalH, initalH, self.start))
        # init g(n) and f(n)
        g_score = {row: float("inf") for row in self.maze.grid}
        g_score[self.start] = 0
        f_score = {row: float("inf") for row in self.maze.grid}
        f_score[self.start] = initalH
        while not queue.empty():
            # get current node
            first = queue.get()
            current_node = first[2]
            if(current_node == self.maze._goal):
                break
            self.iterations += 1
            self.search_path.append(current_node)
```

```
            neighbors = self.get_neighbors(current_node)
            for neighbor in neighbors:
                tmp_g_score = g_score[current_node] + 1
                tmp_h = self.manhattan(neighbor, self.maze._goal)
                tmp_f_score = tmp_g_score + tmp_h
                if tmp_f_score < f_score[neighbor]:
                    g_score[neighbor] = tmp_g_score
                    f_score[neighbor] = tmp_f_score
                     queue.put((tmp_f_score, tmp_h, neighbor))
                    self.saved_path[neighbor] = current_node
        self.end_time = time.process_time()
        self.reverse_path()
        return self.solution
```

IV MDP Value Iteration

```
def value_iteration(self, gamma, error=1e-10):
        self.name = 'MDP-Value Iteration'
        self.iterations = 0
        if not self.rewards:
            self.setupMdp()
        self.start_time = time.process_time()
        V = {}
        policy = {}
```

```
all_states = self.maze.maze_map
for state in all_states:
    V[state] = 0
    policy[state] = 'N' # initialize policy to any action
while True:
    delta = 0
     self.iterations += 1
    for state in all_states:
        v = V[state]
        max_v = -np.inf
        best_action = 'N' # initialize best action to any action
        for action in self.actions:
            action_v = 0
            curret_reward = self.rewards[state][action][1]
             possible_actions = self.get_possible_action(state, action)
            for pa in possible_actions:
                next_state, _ = self.rewards[state][pa]
                prob = possible_actions[pa]['prob']
                action_v += (prob*V[next_state])
            action_v = action_v * gamma + curret_reward
            if action_v > max_v:
                max_v = action_v
                best_action = action
```

```
                V[state] = max_v
                policy[state] = best_action
                delta = max(delta, abs(v - V[state]))
            if delta < error:
                break
        self.end_time = time.process_time()
        # trace path
        current_state = self.start
        while current_state != self.maze._goal:
            # print(current_state)
            action = policy[current_state]
            next_state, _ = self.rewards[current_state][action]
            self.solution[current_state] = next_state
            current_state = next_state
        # print('Optimal path:', self.solution)
        return V, policy
```

V MDP Policy Iteration

```
 def policy_iteration(self, gamma, error=1e-10):
        self.name = 'MDP-Policy Iteration'
        self.iterations = 0
        if not self.rewards:
            self.setupMdp()
```

```
self.start_time = time.process_time()
V = {}
policy = {}
all_states = self.maze.maze_map
for state in all_states:
    V[state] = 0
    policy[state] = 'N' # initialize policy to any action
while True:
    # policy evaluation step
    while True:
        delta = 0
        for state in all_states:
            v = V[state]
             action = policy[state]
            curret_reward = self.rewards[state][action][1]
            possible_actions = self.get_possible_action(state, action)
            action_v = 0
            for pa in possible_actions:
                next_state, _ = self.rewards[state][pa]
                prob = possible_actions[pa]['prob']
                action_v += (prob*V[next_state])
            action_v = action_v * gamma + curret_reward
            V[state] = action_v
```

```
            delta = max(delta, abs(v - V[state]))
        if delta < error:
            break


    # policy improvement step
    policy_stable = True
    for state in all_states:
        old_action = policy[state]
        max_v = -np.inf
        best_action = 'N' # initialize best action to any action
        for action in self.actions:
            action_v = 0
            curret_reward = self.rewards[state][action][1]
            possible_actions = self.get_possible_action(state, action)
            for pa in possible_actions:
                next_state, _ = self.rewards[state][pa]
                prob = possible_actions[pa]['prob']
                action_v += (prob*V[next_state])
            action_v = action_v * gamma + curret_reward
            if action_v > max_v:
                max_v = action_v
                best_action = action
        policy[state] = best_action
```

```
            if old_action != best_action:
                policy_stable = False
        self.iterations += 1
        if policy_stable:
            break
    self.end_time = time.process_time()
    # trace path
    current_state = self.start
    while current_state != self.maze._goal:
        # print(current_state)
        action = policy[current_state]
        next_state, _ = self.rewards[current_state][action]
        self.solution[current_state] = next_state
        current_state = next_state
    # print('Optimal path:', self.solution)
    return V, policy
```