Introduction to Artificial Intelligence HW#2

Name: LeeJaeJun

Student ID: 2021313030

**Abstract**  
 This project employs Q-learning algorithms to solve a maze displayed on PyMaze with 20x20 grids created by DFS backtracking. Initially, the agent is unaware of the maze layout, but it learns to find the optimal path without crossing walls using an ε-greedy strategy and a decaying exploration rate. Users can adjust the number of episodes, learning rate, discount factor, and exploration rate to influence the learning process. These adjustments facilitate easy observation of variable impacts. The final Q-table guides the agent in choosing the most efficient path through the maze.

**Introduction**

This report discusses the implementation of Q-learning algorithms and analyzes the variations resulting from changes in each variable, such as the learning rate and discount factor. I will examine the impact of these variables on the learning process and outcomes. Through this analysis, I aim to find the optimal parameters for navigating a 20x20 maze.

**Definition**

**Q-learning algorithm**: The Q-learning algorithm is a reinforcement learning technique that iteratively updates a Q-table based on actions taken and rewards received to discover the optimal path in each environment.

**Optimal path**: The optimal path is determined by selecting actions with the highest Q-values at each state until reaching the goal state.

**Cost**: The "cost" typically refers to the cumulative number of steps taken to reach the goal.

**Methodology**

**Maze Generation**

The maze utilized for testing was created through a depth-first search backtracking algorithm applied to a 20x20 grid. Each grid cell represents a possible step for the algorithm. In the `maze.py` file, `random.seed(integer)` is utilized to determine the placement of walls within the maze. Meanwhile, `random.seed(integer)` in `algorithm.py` is responsible for determining the entry and exit points. To guarantee consistency and replicability, a fixed seed value was employed during maze generation, ensuring the following:

|  |  |  |
| --- | --- | --- |
|  | src/maze.py | src/algorithm.py |
| Maze 1 | random.seed(0) | random.seed(0) |
| Maze 2 | random.seed(1) | random.seed(1) |
| Maze 3 | random.seed(2) | random.seed(2) |

**Cost Assignment**

The cost is standardized to 1 regardless of direction.

**Visualization**

The first animation depicts the process of moving through cells one by one, highlighting each cell as it is traversed. The cost increases with each movement. Then, the final path is highlighted with circles, showcasing the final route, which includes the final cost.

**Code Details**

**examples/solve\_Q\_learning.py**

**from** **\_\_future\_\_** **import** absolute\_import

**from** **src.maze\_manager** **import** MazeManager

**if** \_\_name\_\_ == "\_\_main\_\_":

*# Create the manager*

manager = MazeManager()

*# Add a 20x20 maze to the manager*

maze = manager.add\_maze(20, 20)

*# Save mp4 file and png*

*# manager.set\_filename("Q\_learning")*

*# Solve the maze using the Q\_learning algorithm*

manager.solve\_maze(maze.id, "Q\_learning")

*# Show how the maze was solved*

manager.show\_solution\_animation(maze.id)

*# Display the maze with the solution overlaid*

manager.show\_solution(maze.id)

The file runs Q-learning algorithm. It creates a maze according to given random seed and shows the navigation process. The maze generation is set to dfs-bractrack by default. You can uncomment the annotated “manager.set\_filename(“Q\_learning”)” part to save the animation video and path png file.

**src/solver.py**

**import** **random**

**import** **numpy** **as** **np**

maximum\_steps = 1000 *# Maximum number of steps per episode*

exploration\_decay = 0.995 *# Rate of decay for exploration*

minimum\_exploration\_rate = 0.05 *# Minimum exploration rate*

Import random to implement the ε-greedy strategy and choose actions with the same Q-value. Use numpy to efficiently manage arrays.

**Maximum\_steps** is the maximum number of learning steps in one episode, which prevents the algorithm from running indefinitely if it doesn't reach a terminating state.

**Exploration\_decay** is the rate of decay for exploration, balancing exploration and exploitation. **Minimum\_exploration\_rate** is the minimum exploration rate, ensuring that exploration does not go to zero and continues to allow for discovering new strategies.

*# Function to get the reward for the current state*

**def** get\_reward(current\_state, next\_state, maze):

*# Penalty for hitting a wall*

**If** maze.grid[current\_state[0]][current\_state[1]].is\_walls\_between(

maze.grid[next\_state[0]][next\_state[1]]):

reward = -1

**elif** next\_state == maze.exit\_coor: *# Reward for reaching the exit*

reward = 100

**else**:

reward = -0.1 *# Penalty for moving to a cell*

**return** reward

This reward function assigns rewards or penalties to the current state based on predetermined values. However, I've made a modification to the order of the rewards from function in assignment guide. Originally, the assignment's **if** statement began with **next\_state == maze.exit\_coor**. If this condition was met, it would award a reward of 100. But, if there's a wall between the current state and **exit\_coor**, the algorithm would still grant the 100-point reward. This would encourage jumping over the wall to reach the goal, continuously corrupting the Q-values and failing to create an optimal path. To address this issue, I've rearranged the order of the **if** statements in the function.

*# Function to get the valid actions that do not allow escape from the maze for the current state.*

**def** valid\_action(current\_state, maze):

actions = []

**if** current\_state[0] != 0:

actions.append(0) *# up*

**if** current\_state[0] != maze.num\_rows - 1:

actions.append(1) *# down*

**if** current\_state[1] != 0:

actions.append(2) *# left*

**if** current\_state[1] != maze.num\_cols - 1:

actions.append(3) *# right*

**return** actions

This function determines the valid movement actions from a given state within a maze. It checks the current state's position against the boundaries of the maze and appends permissible actions to the list:

moving up (0) if not on the top row, down (1) if not on the bottom row, left (2) if not on the leftmost column, and right (3) if not on the rightmost column.

The function ultimately returns the list of valid actions that keep the agent within the maze boundaries.

*# Update state based on action taken*

**def** take\_action(current\_state, action):

**if** action == 0: *# up*

**return** (current\_state[0] - 1, current\_state[1])

**elif** action == 1: *# down*

**return** (current\_state[0] + 1, current\_state[1])

**elif** action == 2: *# left*

**return** (current\_state[0], current\_state[1] - 1)

**elif** action == 3: *# right*

**return** (current\_state[0], current\_state[1] + 1)

This function updates the agent's position within a maze based on a specified action. The actions are encoded as integers: 0 for moving up, 1 for moving down, 2 for moving left, and 3 for moving right. Depending on the action passed to the function, it adjusts the current state's row or column index accordingly. The function then returns the new state as a tuple, reflecting the agent's updated position in the maze.

*# Select the valid action with the highest Q-value for the current state*

**def** select\_best\_action(current\_state, q\_table, valid\_actions):

max\_q\_value = -float('inf')

best\_actions = []

**for** action **in** valid\_actions:

q\_value = q\_table[current\_state[0], current\_state[1], action] *# Q(s, a)*

**if** q\_value > max\_q\_value:

max\_q\_value = q\_value

best\_actions = [action]

**elif** q\_value == max\_q\_value:

best\_actions.append(action)

**return** random.choice(best\_actions)

This function selects the optimal action for a given state in a maze based on the Q-values from a Q-table. It iterates over the list of valid actions, retrieving the Q-value for each action and comparing it to the current maximum Q-value.

If a new maximum is found, it updates the list of best actions; if the Q-value is equal to the current maximum, the action is added to the list. Finally, it randomly selects one of the best actions to handle the case where multiple actions have the same maximum Q-value, preventing iterate same action.

**def** q\_learning(maze, episodes\_num, learning\_rate, discount\_factor, exploration\_rate):

q\_table = np.zeros((maze.num\_rows, maze.num\_cols, 4)) *# Initialize Q-table with zeros*

**for** episode **in** range(episodes\_num): *# Loop over episodes*

current\_state = maze.entry\_coor *# Initialize the current state*

**for** \_ **in** range(maximum\_steps): *# Loop over steps*

*# Apply ε-greedy strategy*

valid\_actions = valid\_action(current\_state, maze)

**if** random.uniform(0, 1) < exploration\_rate:

action = random.choice(valid\_actions)

**else**:

action = select\_best\_action(current\_state, q\_table, valid\_actions)

next\_state = take\_action(current\_state, action)

reward = get\_reward(current\_state, next\_state, maze) *# R(s, a)*

**if** reward == -1: *# Stay in the same place if hit a wall*

next\_state = current\_state

q\_now = q\_table[current\_state[0], current\_state[1], action] *# Q(s, a)*

next\_valid\_actions = valid\_action(next\_state, maze) *# Q(s′,a′)*

max\_next\_q\_value = -float('inf')

**for** next\_action **in** next\_valid\_actions:

next\_q\_value = q\_table[next\_state[0], next\_state[1], next\_action]

**if** next\_q\_value > max\_next\_q\_value:

max\_next\_q\_value = next\_q\_value *# maxQ(s′,a′)*

*# Q-learning formula: Q(s, a) = Q(s, a) + α \* (R(s, a) + γ \* maxQ(s′,a′) - Q(s, a))*

q\_table[current\_state[0], current\_state[1], action] = q\_now + learning\_rate \* (reward + discount\_factor \* max\_next\_q\_value - q\_now)

current\_state = next\_state *# Move to the next state*

**if** next\_state == maze.exit\_coor: *# If the agent reaches the exit, terminate the episode*

**break**

exploration\_rate = max(minimum\_exploration\_rate, exploration\_rate \* exploration\_decay) *# Decay the exploration rate to reduce exploration over time*

**return** q\_table

The function `q\_learning` is designed to train a Q-learning agent within a specified maze environment over multiple episodes. It begins by initializing a Q-table with zero values for each state-action pair in the maze, where each state corresponds to a coordinate in the maze and each action corresponds to a possible movement (up, down, left, right).

For each episode, the process starts from a predefined entry point (`maze.entry\_coor`) of the maze. Within each episode, a maximum number of steps (`maximum\_steps`) is allowed. For each step, the agent selects an action using the ε-greedy strategy: it chooses a random action with probability equal to the current exploration rate, encouraging exploration, and the best action according to the Q-table with the complementary probability, focusing on exploitation.

After selecting an action, the agent moves to the next state and receives a reward based on the transition. If the action results in hitting a wall, the agent remains in the current state. This prevents wrong learning and penalty to wall more and more. The Q-value for the current state-action pair is then updated using the Q-learning formula, .

The loop continues until the agent reaches the maze's exit coordinate or exhausts the allowed steps. The exploration rate is decayed after each episode according to a predefined decay rate and a minimum exploration rate threshold to ensure some level of exploration continues throughout the training process. The function returns the trained Q-table, which ideally represents the optimal policy for navigating the maze.

**def** q\_learning\_path(maze, q\_table):

current\_state = maze.entry\_coor *# Start at the entry point of the maze*

found\_path = [current\_state] *# Initialize the path with the starting position*

**for** \_ **in** range(maximum\_steps):

**if** current\_state == maze.exit\_coor: *# Stop if the exit is reached*

**break**

*# Determine the best valid action to take at the current state*

valid\_actions = valid\_action(current\_state, maze)

best\_action = select\_best\_action(current\_state, q\_table, valid\_actions)

*# Move to the next state based on the best action*

next\_state = take\_action(current\_state, best\_action)

*# Add the new state to the path*

found\_path.append(next\_state)

*# Update the current state to the new state*

current\_state = next\_state

found\_cost = len(found\_path)

**return** [found\_path, found\_cost]

The function `q\_learning\_path` uses a trained Q-table to navigate a maze from an entry point to an exit. Starting at the maze's entry coordinate, it constructs a path by iteratively selecting and executing the best action at each state, determined by the highest Q-value from the Q-table. The function stops if the exit is reached or the maximum number of steps (`maximum\_steps`) is exhausted. It returns the path taken by the agent and the total cost, measured as the number of steps in the path.

**Modifications and Enhancements**

I removed the `DepthFirstBacktracker`, `BiDirectional`, and `BreadthFirst` functions from `src/solver`, as well as other files unrelated to the `EXEMPLE` project.

**src/maze.py**

- Remove the import of the time module as it is not used in the code.



**Line 41**: The variable self.cost is used to store the cost value returned by the q\_learning\_path function.

**Line 42**: The random.seed method is utilized to determine the placement of walls within the maze.

**src/algorithm.py**

- Remove the import of the time module and math module as they don’t need to my code.

- Remove the sections related to timing information generation and their outputs, as they are not needed.



**Line 6:** determining the entry and exit points.

**src/maze\_manager.py**

****

**Line 3-4**: Import the q\_learning and q\_learning\_path functions from src/solver.



**Line 37**: In exist code, It was ‘if id is not 0’. But it occurred SyntaxWarning: 'is not' with an 'int' literal. Did you mean '!='? in my environment. Therefore, I changed ‘is not’ to ‘!=’ in the statement.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명

**Line 118-158**: Modified to executes q\_learning function if the selected method is "Q\_learning". Initially, it prompts the user for several parameters needed for the learning process: the number of episodes, learning rate, discount factor, and exploration rate.

Each parameter must be entered as a valid number within specified ranges. For instance, the number of episodes must be a positive integer, and the learning, discount, and exploration rates must be numbers between 0 and 1. If the user enters an invalid value for any of these parameters, the program raises a ValueError and prompts the user to re-enter a correct value.

Once all parameters are inputted, the Q-learning algorithm uses them to generate a learned Q-table. Then, q\_learning\_path function finds the best path and its cost from Q-table. And It store these results in the maze object for future reference.

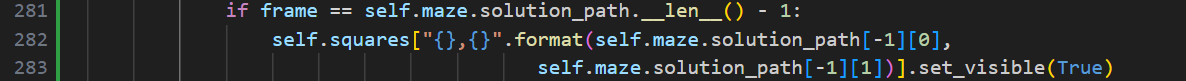
**src/maze\_viz.py**

- The log information unnecessary in the assignment. Therefore, I deleted all related outputs.

- In the original code, the `solution\_path` was used as a three-dimensional array to depict the backtracking process. However, in the animation for the Q-learning algorithm, I determined that backtracking was unnecessary. Therefore, in the `solver.py` functions, this was changed to a two-dimensional array, and I modified all related parts in maze\_vis.py accordingly. Additionally, I removed the parts of the animation that displayed backtracking.



**Line 125**: The `frameon` parameter has been deprecated in recent updates to matplotlib, so I have removed the `frameon=None` part.



**Line 281-283**: In animation video, the final goal part did not turn red and ended with existed code, so I added code to change the color at the end as well.



**Line 298-299**: the original code already painted the path colors as the agent moved in the video. I changed this to record coloring process in the video.

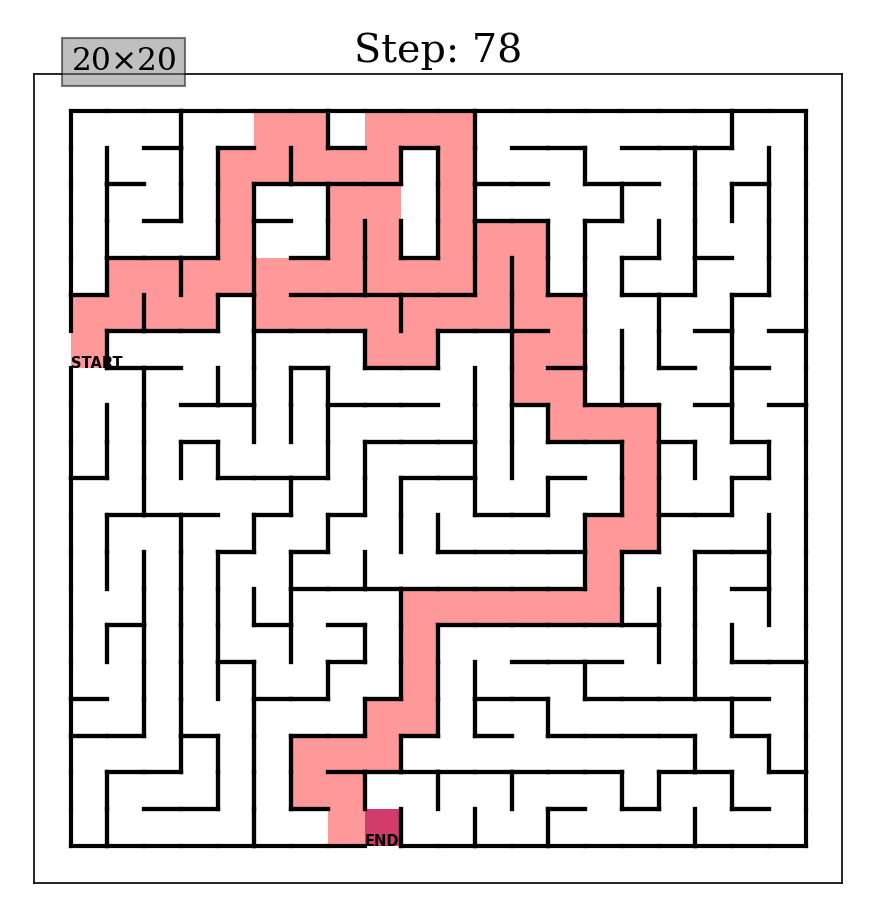
**Results and Discussion**

**Visualization on the 3 randomly generated maze**

**src/maze.py: random.seed(0)**

**src/algorithm.py: random.seed(0)**

**episode number= 500, learning rate(α)=0.8, discount factor(γ)=0.9, exploration rate(ε) = 1**

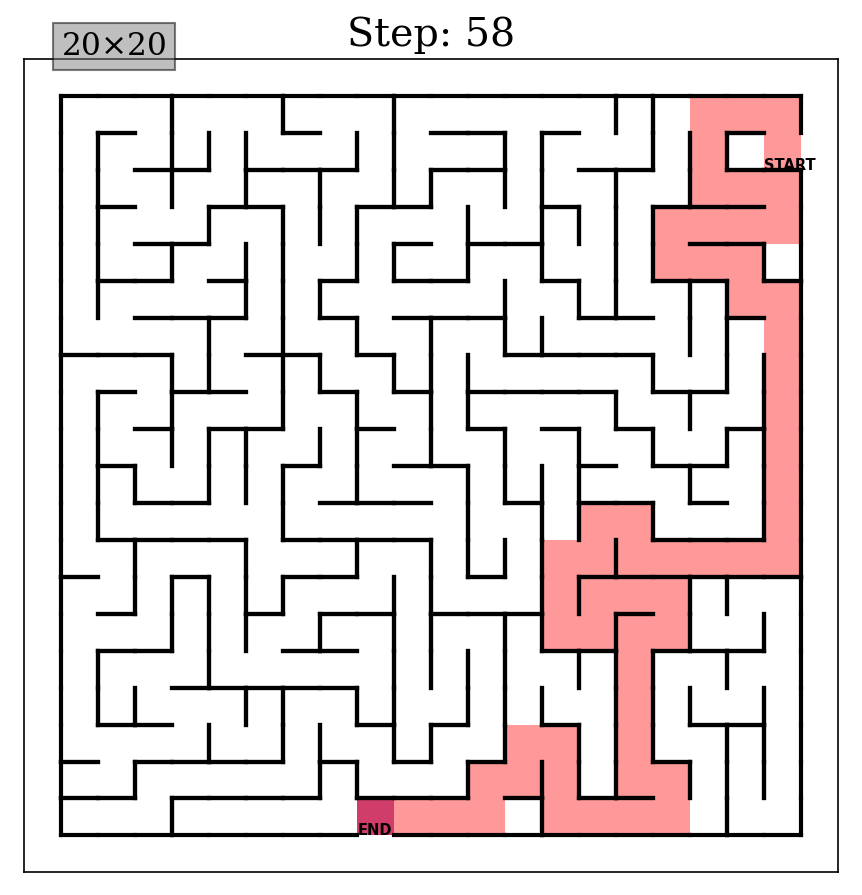
**** 도표, 직사각형, 평면도, 패턴이(가) 표시된 사진

자동 생성된 설명

**src/maze.py: random.seed(1)**

**src/algorithm.py: random.seed(1)**

**episode number= 500, learning rate(α)=0.8, discount factor(γ)=0.9, exploration rate(ε) = 1**

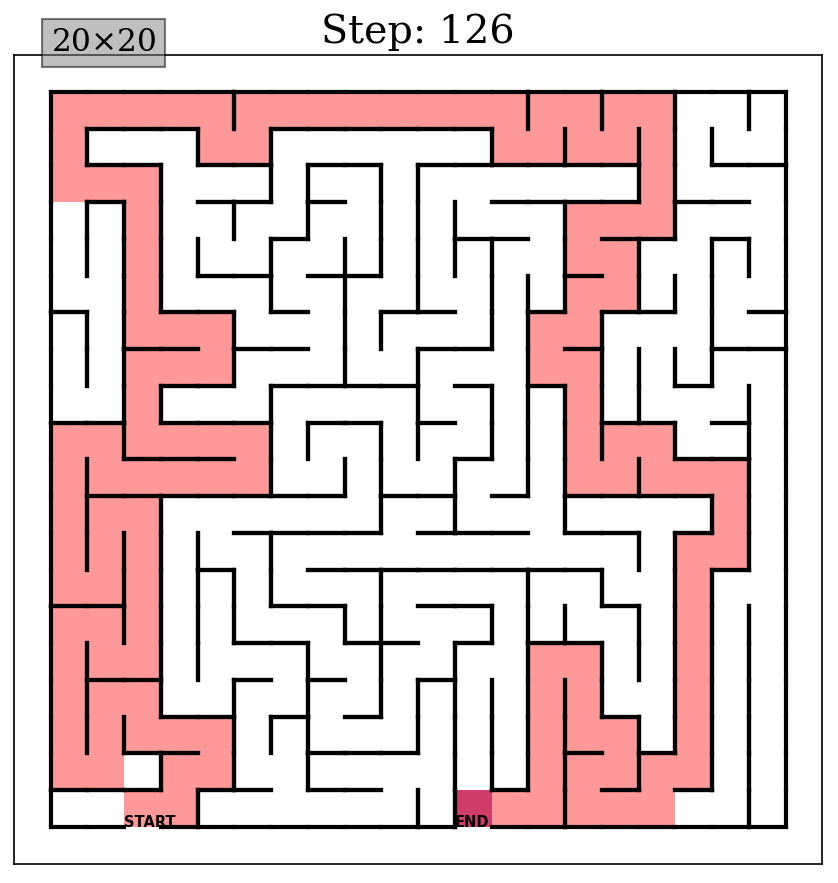
**도표, 패턴, 직사각형, 디자인이(가) 표시된 사진

자동 생성된 설명**

**src/maze.py: random.seed(2)**

**src/algorithm.py: random.seed(2)**

**episode number= 500, learning rate(α)=0.8, discount factor(γ)=0.9, exploration rate(ε) = 1**

**** **도표, 직사각형, 패턴, 디자인이(가) 표시된 사진

자동 생성된 설명**

**각 매개변수 이유? Exploration = 1인거는 점점 줄여가니까. 그리고 첫 에피소드는 다 0이라서 랜덤으로 하는 것과 비슷하다고 생각? 그리고 처음에는 탐색?을 많이하는게 좋다고 생각? 파일들 위치**

**Discussion about the impact of the learning rate(α)**

**Discussion about the impact of the discount factor(γ)**

**Discussion about the optimal episode number on 20x20 maze**

각 시드 3개에 대해서 같은거 5번씩 돌려서 기록하고 그 평균가지고 first hit exit의 episode number 구하기

**Data Appendix**