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.

```python
# 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.

```python
# 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.

```python
# 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.

```python
# 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, $Q(s,a) = Q(s,a) + \alpha * (R(s,a) + \gamma * \max_{a'} Q(s',a') - Q(s,a))$.

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.

```python
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]
```

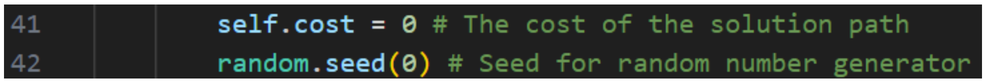
The function `q_learning_path` uses a trained Q-table to navigate a maze from an entry point to an exit. Starting at the maze's entry coordinate, it constructs a path by iteratively selecting and executing the best action at each state, determined by the highest Q-value from the Q-table. The function 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.



```
41          self.cost = 0 # The cost of the solution path
42          random.seed(0) # Seed for random number generator
```

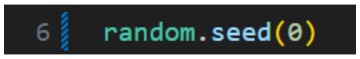
Line 41: The variable self.cost is used to store the cost value returned by the q_learning_path function.

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.

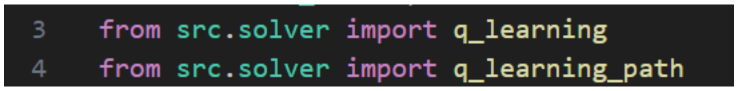
- Remove the sections related to timing information generation and their outputs, as they are not needed.



```
6   random.seed(0)
```

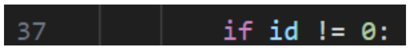
Line 6: determining the entry and exit points.

src/maze_manager.py



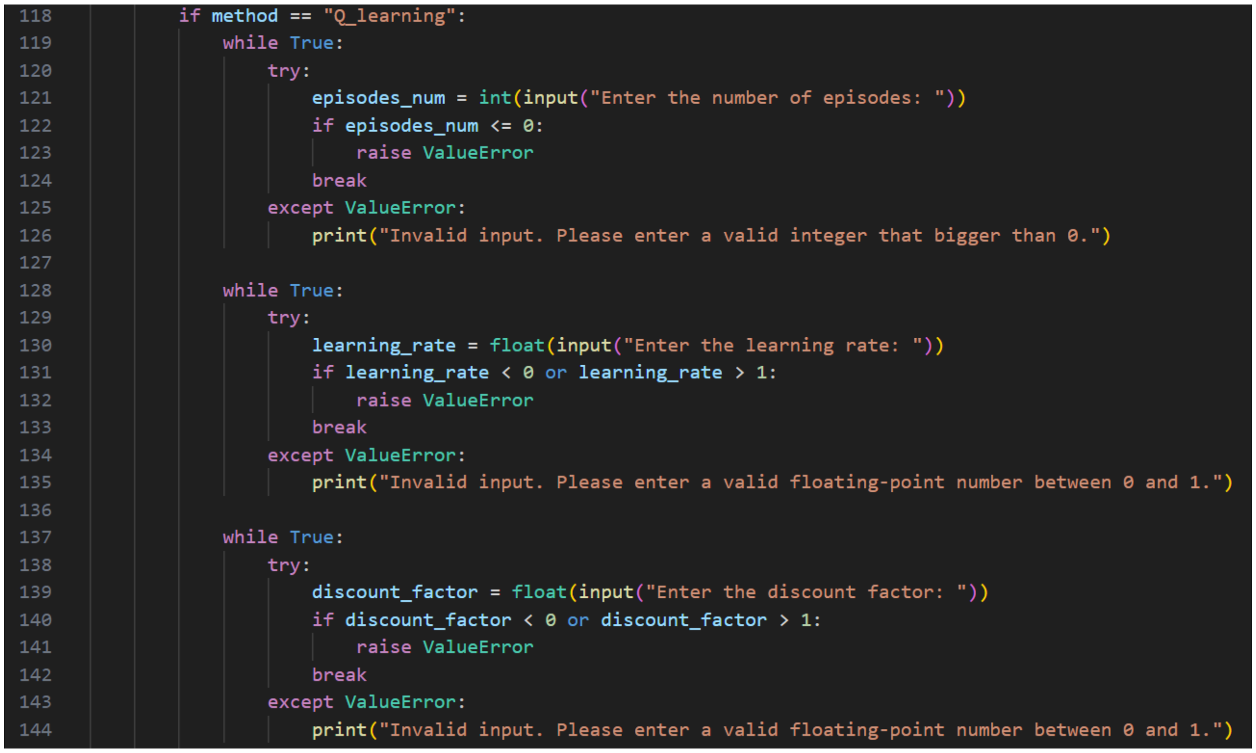
```
3   from src.solver import q_learning
4   from src.solver import q_learning_path
```

Line 3-4: Import the q_learning and q_learning_path functions from src/solver.

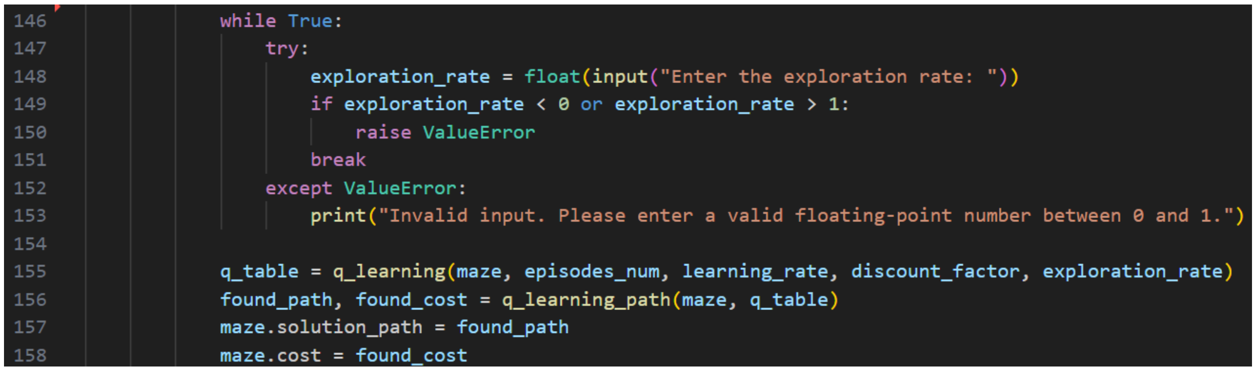


```
37          if id != 0:
```

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



```
118         if method == "Q_learning":
119             while True:
120                 try:
121                     episodes_num = int(input("Enter the number of episodes: "))
122                     if episodes_num <= 0:
123                         raise ValueError
124                     break
125                 except ValueError:
126                     print("Invalid input. Please enter a valid integer that bigger than 0.")
127 
128             while True:
129                 try:
130                     learning_rate = float(input("Enter the learning rate: "))
131                     if learning_rate < 0 or learning_rate > 1:
132                         raise ValueError
133                     break
134                 except ValueError:
135                     print("Invalid input. Please enter a valid floating-point number between 0 and 1.")
136 
137             while True:
138                 try:
139                     discount_factor = float(input("Enter the discount factor: "))
140                     if discount_factor < 0 or discount_factor > 1:
141                         raise ValueError
142                     break
143                 except ValueError:
144                     print("Invalid input. Please enter a valid floating-point number between 0 and 1.")
```



```
146             while True:
147                 try:
148                     exploration_rate = float(input("Enter the exploration rate: "))
149                     if exploration_rate < 0 or exploration_rate > 1:
150                         raise ValueError
151                     break
152                 except ValueError:
153                     print("Invalid input. Please enter a valid floating-point number between 0 and 1.")
154 
155             q_table = q_learning(maze, episodes_num, learning_rate, discount_factor, exploration_rate)
156             found_path, found_cost = q_learning_path(maze, q_table)
157             maze.solution_path = found_path
158             maze.cost = found_cost
```

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.

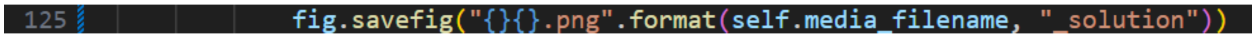
Each parameter must be entered as a valid number within specified ranges. For instance, the number of episodes must be a positive integer, and the learning, discount, and exploration rates must be numbers between 0 and 1. If the user enters an invalid value for any of these parameters, the program raises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

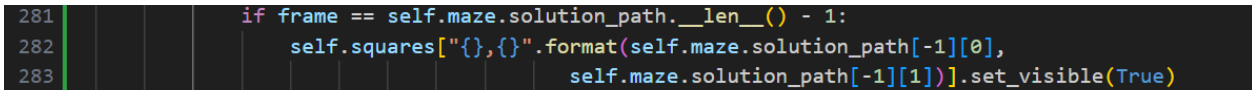
- The log information unnecessary in the assignment. Therefore, I deleted all related outputs.

- 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.



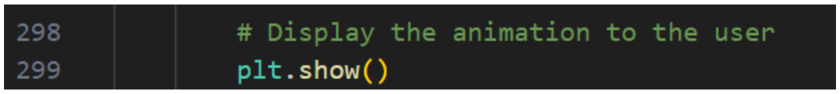
```
125             fig.savefig("{}{}.png".format(self.media_filename, "_solution"))
```

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



```
281             if frame == self.maze.solution_path.__len__() - 1:
282                 self.squares["{},{}".format(self.maze.solution_path[-1][0],
283                                             self.maze.solution_path[-1][1])].set_visible(True)
```

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.



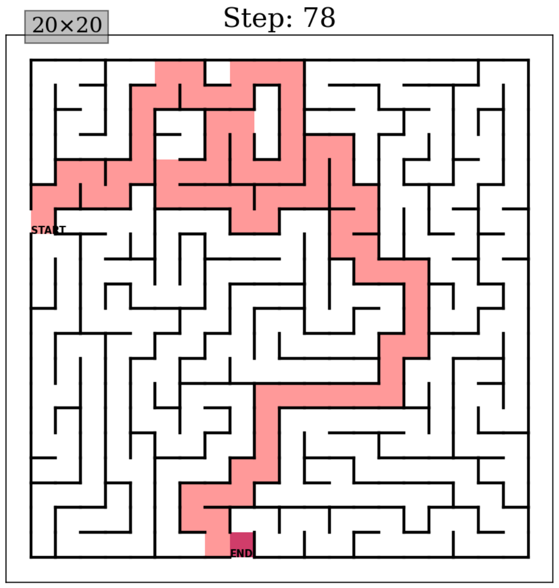
```
298         # Display the animation to the user
299         plt.show()
```

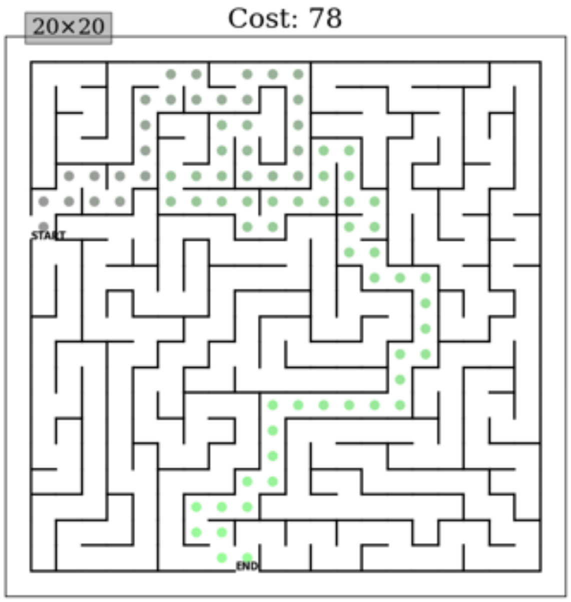
Line 298-299: the original code already painted the path colors as the agent moved in the video. I changed this to record coloring process in the video.

Results and Discussion

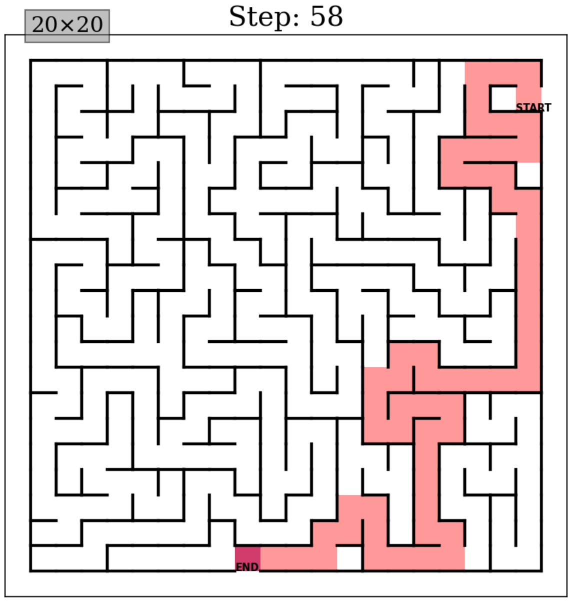
Visualization on the 3 randomly generated maze

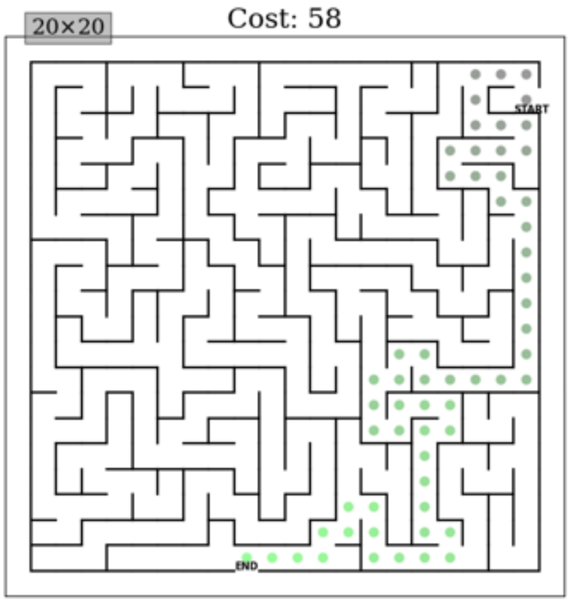
src/maze.py: random.seed(0)
src/algorithm.py: random.seed(0)
episode number= 500, learning rate(α)=0.8, discount factor(γ)=0.9, exploration rate(ε) = 1



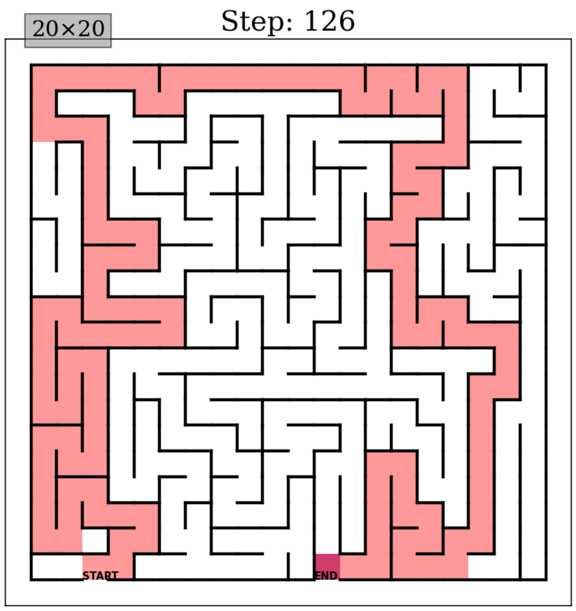


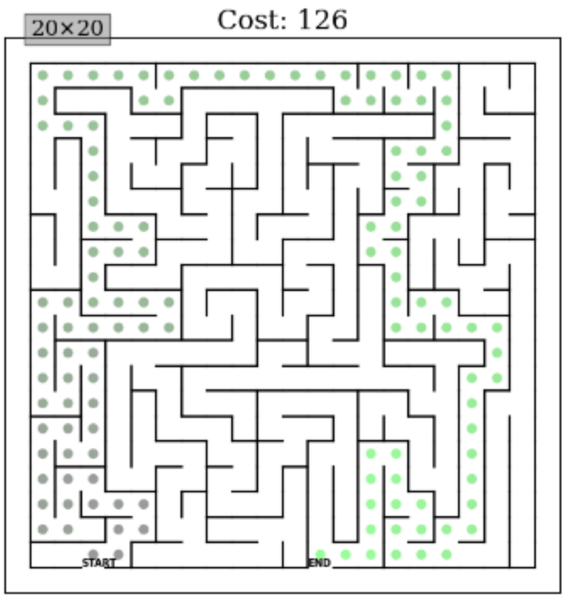
src/maze.py: random.seed(1)
src/algorithm.py: random.seed(1)
episode number= 500, learning rate(α)=0.8, discount factor(γ)=0.9, exploration rate(ε) = 1





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





The Visualization folder contains video and photo files organized into folders for each seed.

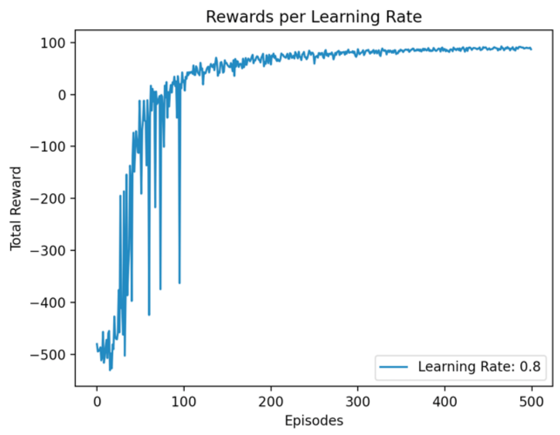
Reasons for using ε-decaying

Epsilon(ε)-decaying is a strategy to balance exploration and exploitation by gradually reducing the probability of choosing a random action over time. Starting with a high epsilon value ensures extensive initial exploration of the environment, allowing the agent to gather diverse information.

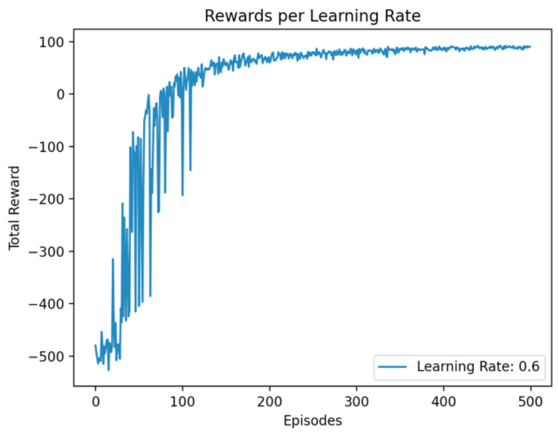
The initial comprehensive exploration, starting with a high exploration rate of 1, allows for thorough exploration of the environment without bias, essential for understanding unknown optimal strategies. As learning progresses, the exploration rate gradually decreases, balancing exploration and exploitation. This shift refines strategies and improves learning efficiency, converging towards an optimal policy while avoiding premature commitment to suboptimal decisions. High initial exploration helps to avoid local optima by ensuring the algorithm explores a variety of actions, aiding in the discovery of more effective long-term strategies as learning advances.

Discussion about the impact of the learning rate(α)

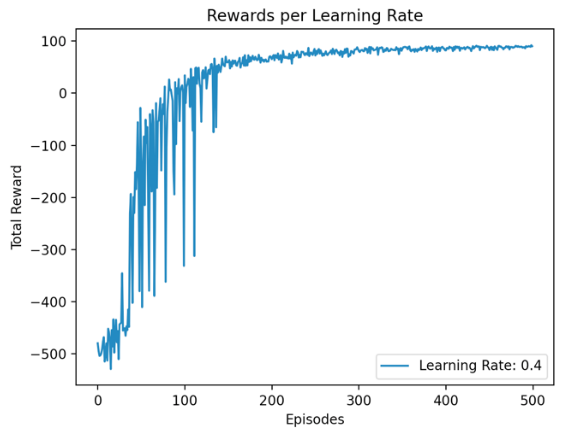
src/maze.py: random.seed(0)
src/algorithm.py: random.seed(0)
episode number= 500, discount factor(γ)=0.9, exploration rate(ε) = 1



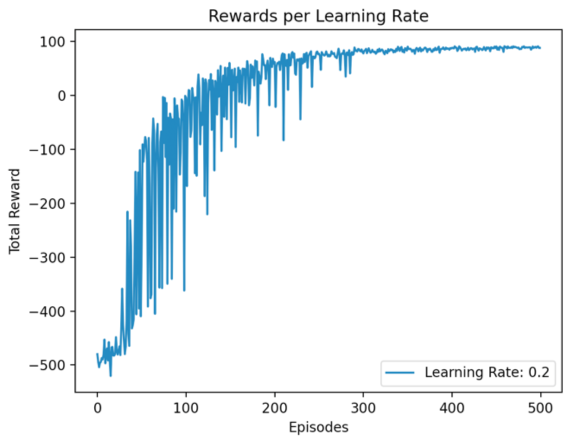
Episode reward graph (learning rate(α)=0.8)



Episode reward graph (learning rate(α)=0.6)

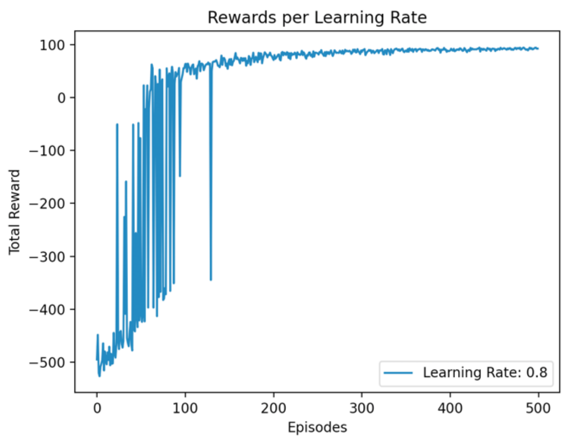


Episode reward graph (learning rate(α)=0.4)

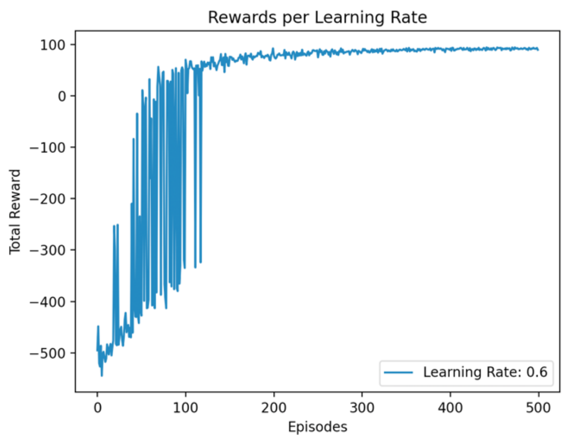


Episode reward graph (learning rate(α)=0.2)

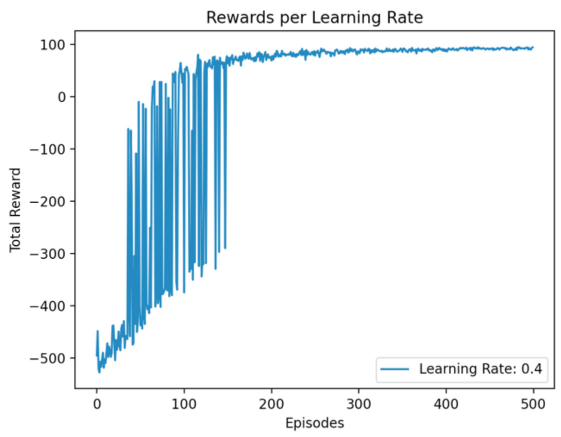
src/maze.py: random.seed(1)
src/algorithm.py: random.seed(1)
episode number= 500, discount factor(γ)=0.9, exploration rate(ε) = 1



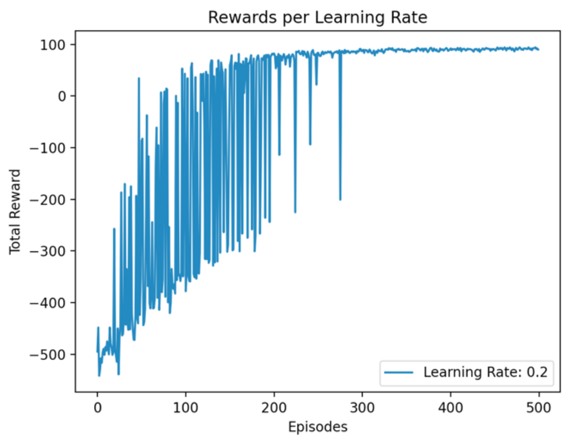
Episode reward graph (learning rate(α)=0.8)



Episode reward graph (learning rate(α)=0.6)

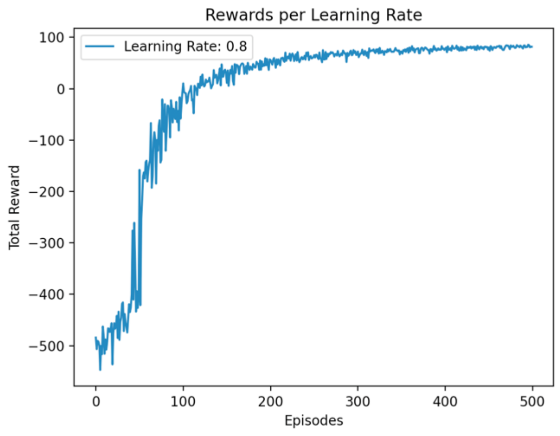


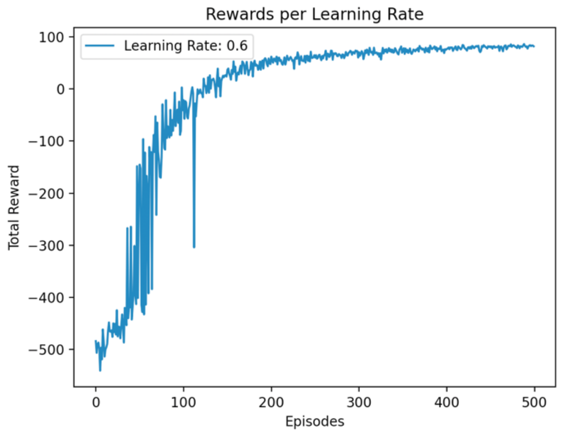
Episode reward graph (learning rate(α)=0.4)



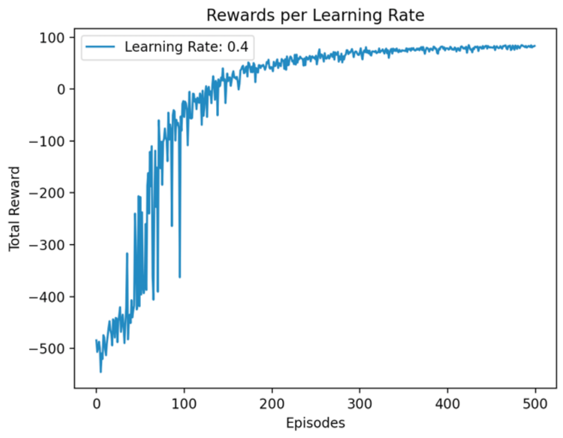
Episode reward graph (learning rate(α)=0.2)

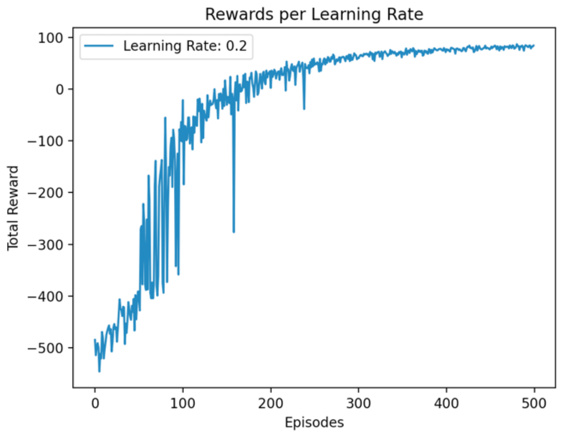
src/maze.py: random.seed(2)
src/algorithm.py: random.seed(2)
episode number= 500, discount factor(γ)=0.9, exploration rate(ε) = 1





Episode reward graph (learning rate(α)=0.8) **Episode reward graph (learning rate(α)=0.6)**





Episode reward graph (learning rate(α)=0.4). **Episode reward graph (learning rate(α)=0.2)**

Above graphs showing the total reward for each episode by changing only the learning rate under the same conditions for each seed. In every case, it arrived at the goal in search of the optimal path.

In reinforcement learning, a reward is direct feedback from the environment, indicating the outcome of the agent's actions. The total reward obtained in each episode (episode reward) serves as a metric to assess how well the agent performed during that episode.

As the learning rate increases from 0.2 to 0.8, there is a noticeable improvement in how quickly the algorithm achieves higher rewards, showing that a higher learning rate can lead to faster initial performance. However, the stability of these rewards, especially as the number of episodes increases, also seems to stabilize more quickly.

For example, with a learning rate of 0.2, the algorithm learns slowly, as evidenced by a gradual increase in rewards and frequent fluctuations. When the learning rate is adjusted to 0.4 and higher, the convergence speed—indicating how quickly the algorithm consistently yields higher rewards—also increases, observable from the sharp rise in rewards in the early episodes.

An optimal balance between fast learning and reward stability appears at a learning rate of 0.6, where the rewards not only improve quickly but also reach a high level with less volatility compared to very high learning rates like 0.8. At a learning rate of 0.8, the algorithm shows rapid initial improvements, suggesting quick adaptation to received rewards. However, this high rate may also make the algorithm more sensitive to noise or variance in the reward signals, potentially leading to uneven performance across episodes.

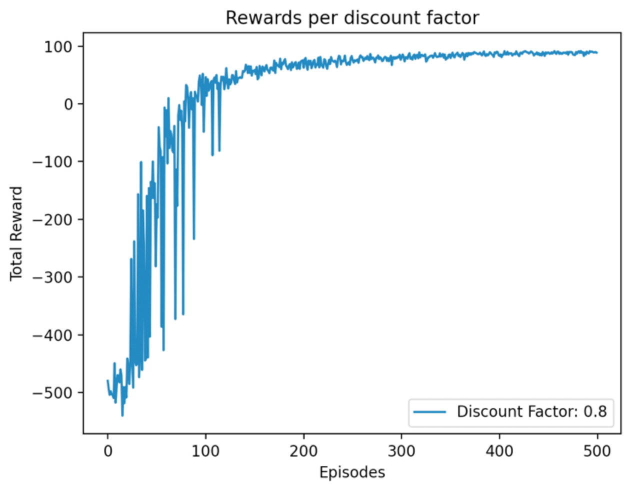
As the learning rates reach the upper limits, such as 0.8, we observe that increasing the learning rate continuously does not always translate into better long-term average rewards. This suggests a potential for overfitting or excessive policy updates that could lead to instability in learning. Thus, choosing the right learning rate is crucial; in your case, a moderately high rate around 0.6 seems to offer the best compromise between rapid learning and stable performance.

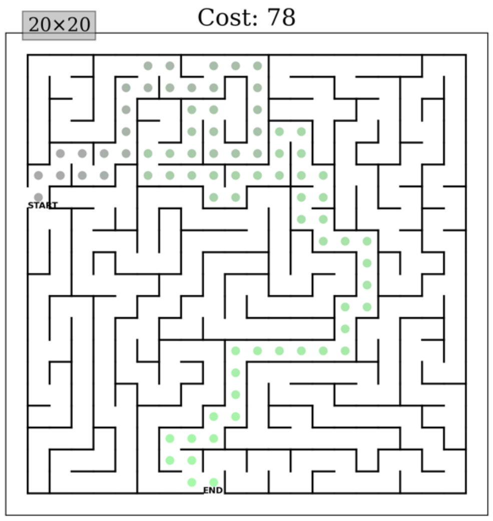
Therefore, a higher learning rate leads to faster updates to the model and faster learning progress. This can be beneficial in the initial stages of learning, but too large of a learning rate increases the risk of divergence. A smaller learning rate helps the model converge more steadily, but it may take longer to converge.

In conclusion, finding the optimal learning rate for a given environment, one that balances model performance and learning speed, is crucial for building a good model.

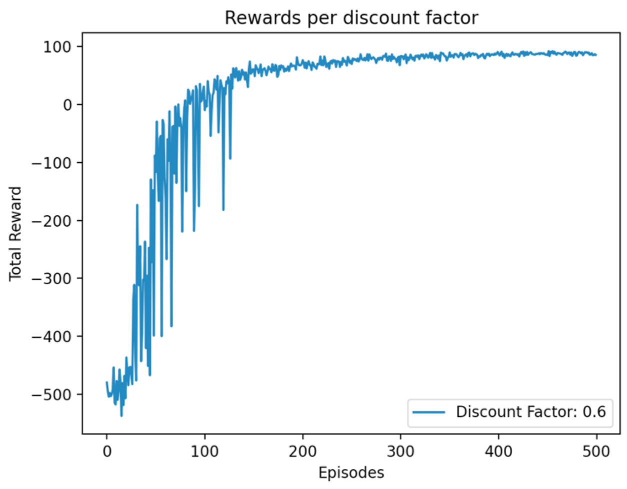
Discussion about the impact of the discount factor(γ)

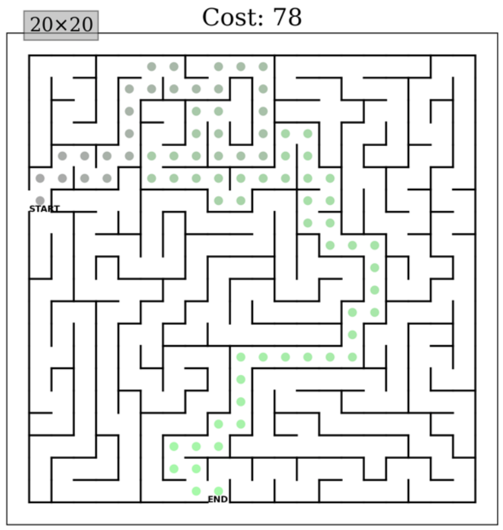
src/maze.py: random.seed(0)
src/algorithm.py: random.seed(0)
episode number= 500, learning rate(α)=0.6, exploration rate(ε) = 1



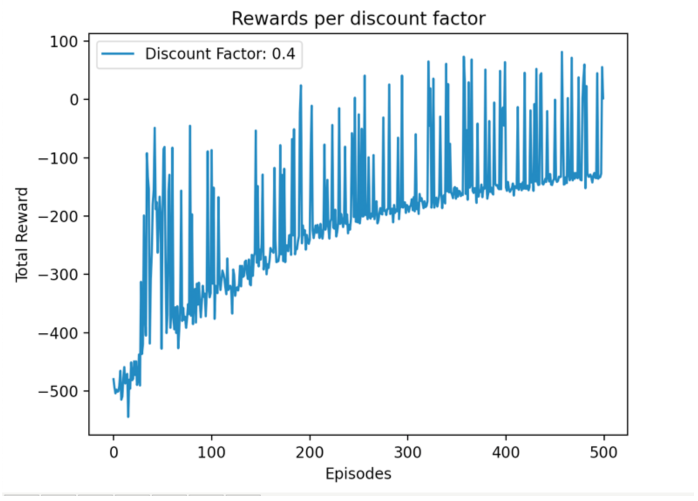


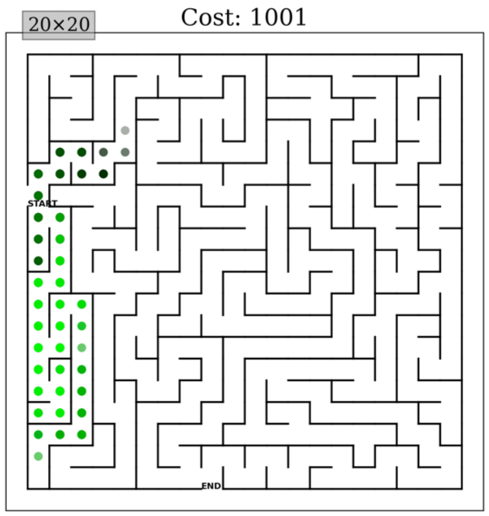
Episode reward graph (discount factor(γ)=0.8) and path result



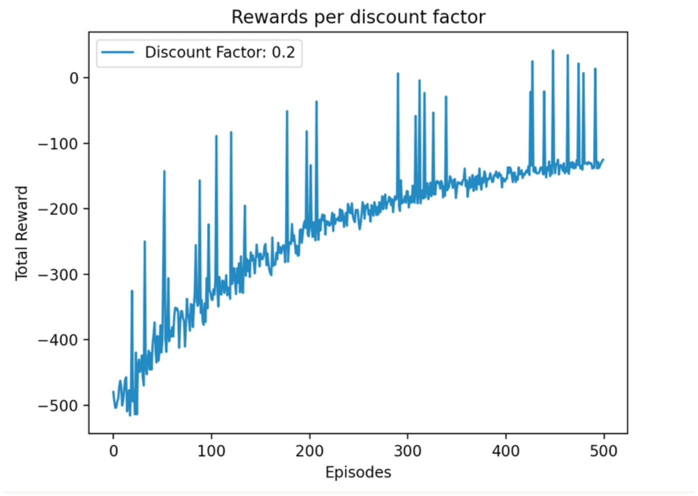


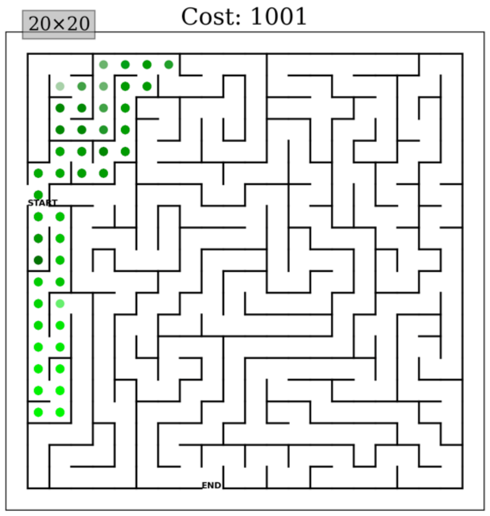
Episode reward graph (discount factor(γ)=0.6) and path result





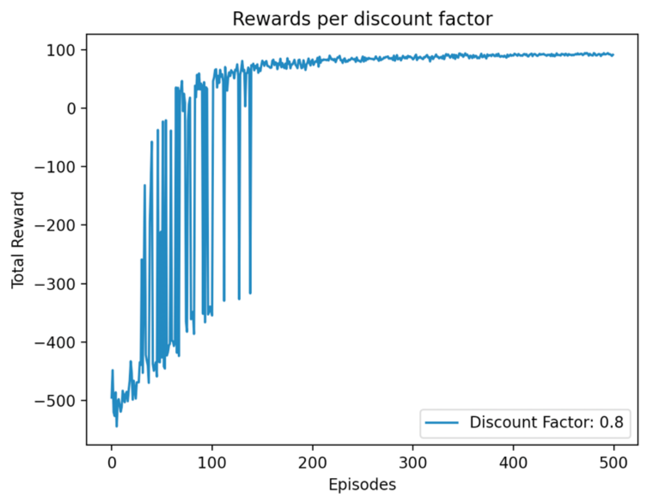
Episode reward graph (discount factor(γ)=0.4) and path result

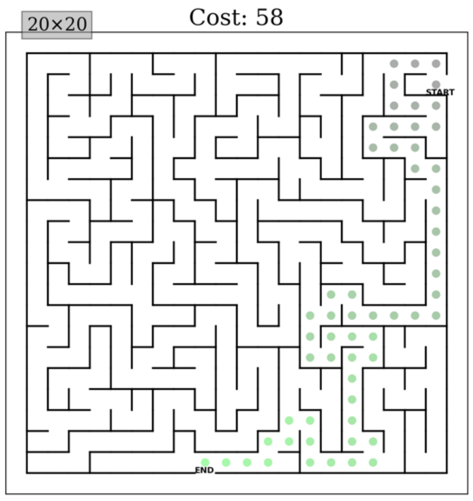




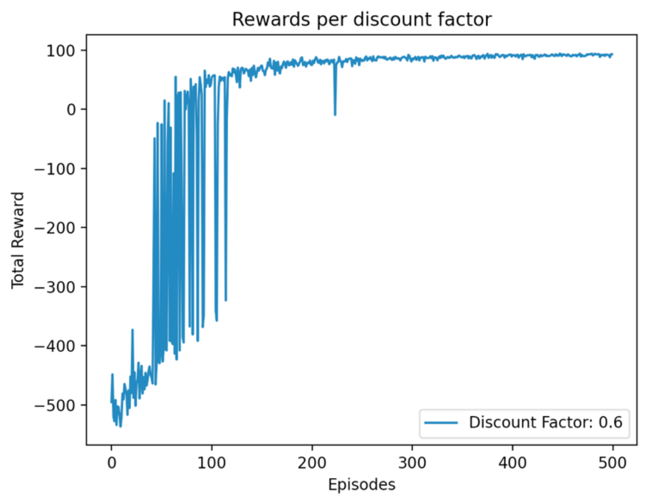
Episode reward graph (discount factor(γ)=0.2) and path result

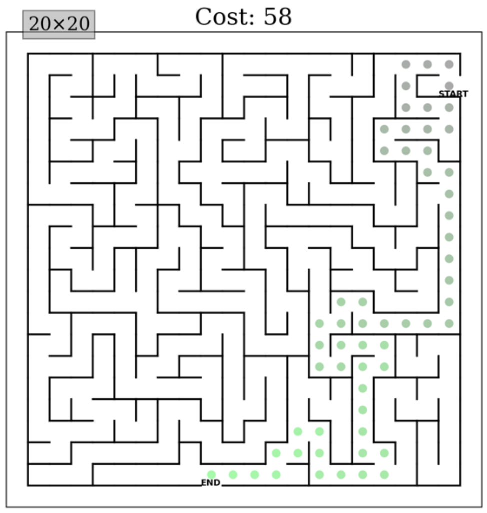
src/maze.py: random.seed(1)
src/algorithm.py: random.seed(1)
episode number= 500, learning rate(α)=0.6, exploration rate(ε) = 1



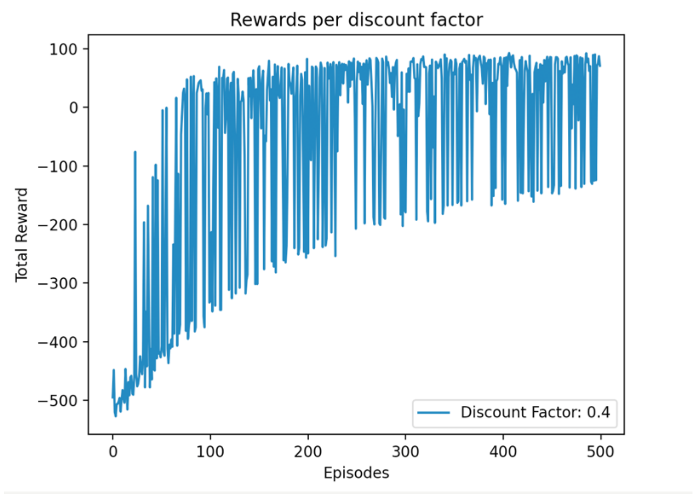


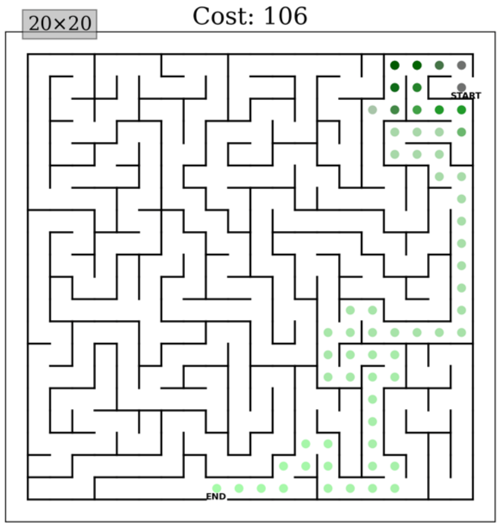
Episode reward graph (discount factor(γ)=0.8) and path result



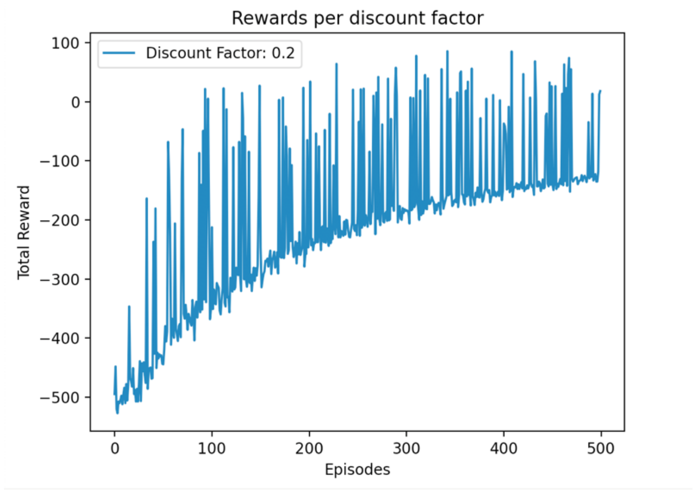


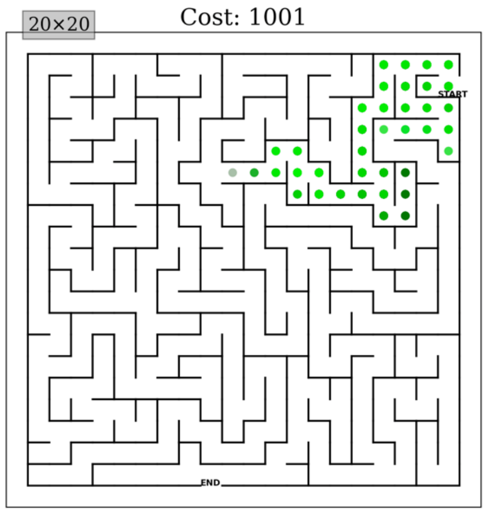
Episode reward graph (discount factor(γ)=0.6) and path result





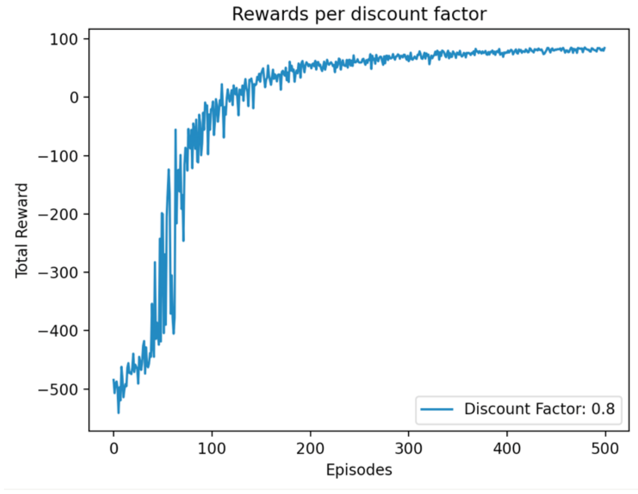
Episode reward graph (discount factor(γ)=0.4) and path result

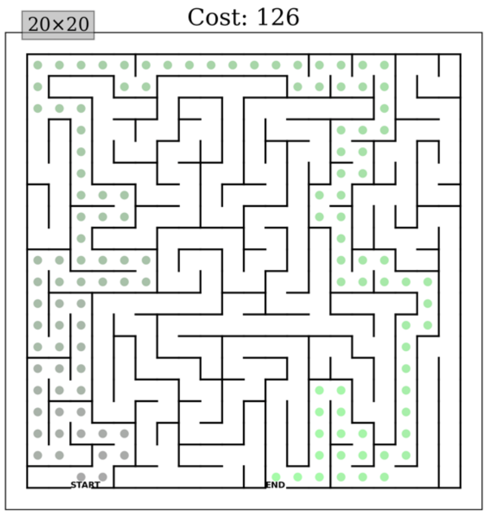




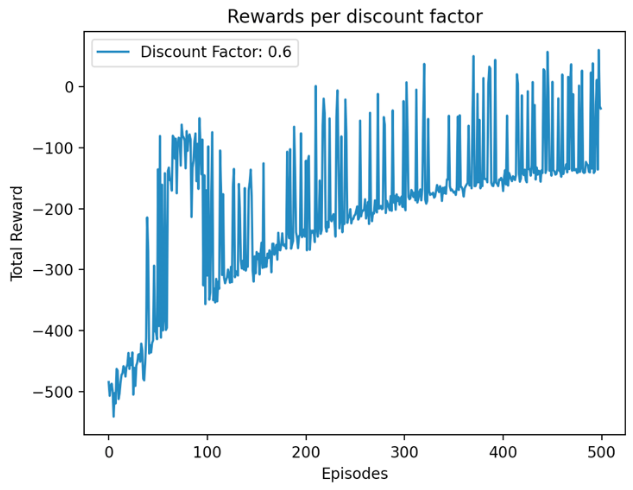
Episode reward graph (discount factor(γ)=0.2) and path result

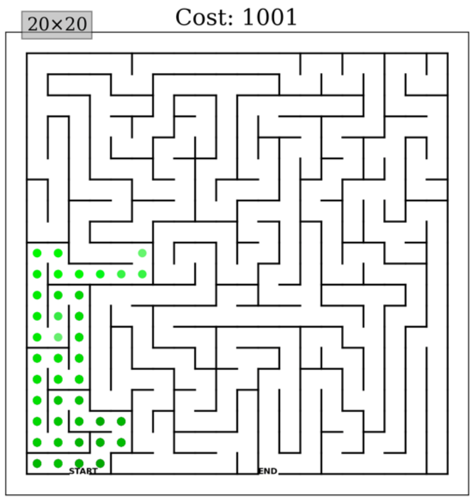
src/maze.py: random.seed(2)
src/algorithm.py: random.seed(2)
episode number= 500, learning rate(α)=0.6, exploration rate(ε) = 1



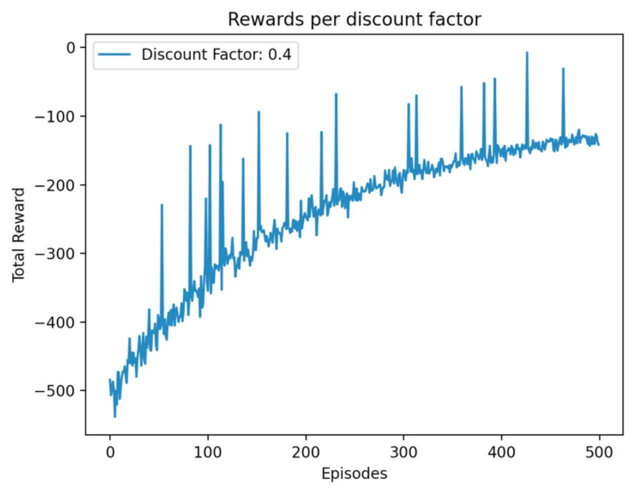


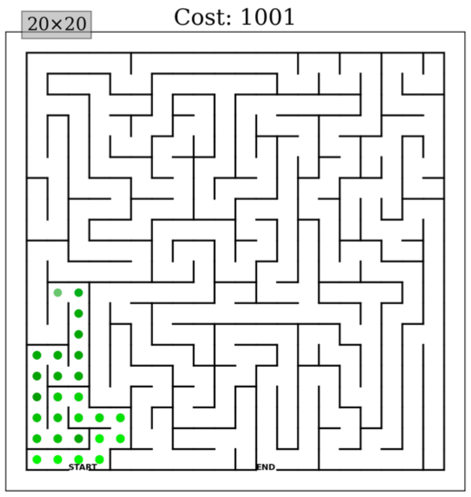
Episode reward graph (discount factor(γ)=0.8) and path result



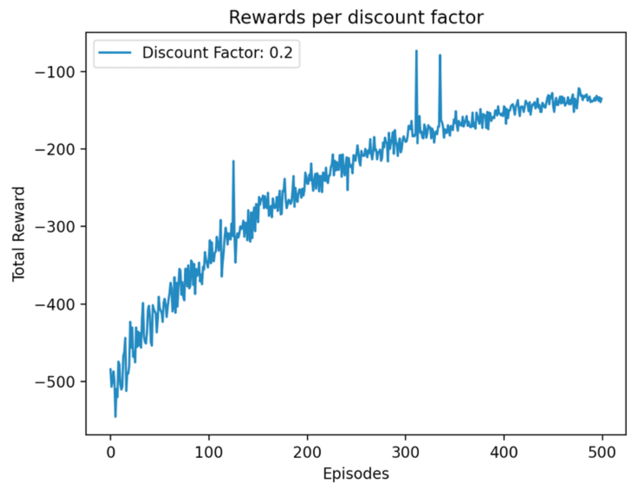


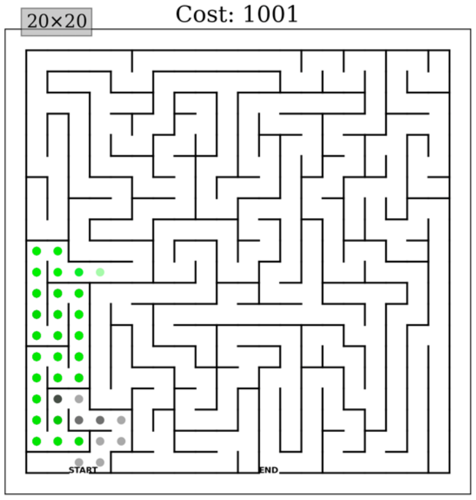
Episode reward graph (discount factor(γ)=0.6) and path result





Episode reward graph (discount factor(γ)=0.4) and path result





Episode reward graph (discount factor(γ)=0.2) and path result

In reinforcement learning, the discount factor is an essential parameter that affects the present value of future rewards. A high discount factor, close to 1, indicates that the model values future rewards as significantly as immediate rewards. Conversely, a discount factor near 0 means the model primarily responds to immediate rewards.

From experiments with different seeds:

- **Seed 0**: With discount factors of 0.8 and 0.6, the agent reached the optimal path, while with 0.4 and 0.2, the agent exhibited oscillations and failed to reach the goal within the maximum steps allowed.

- **Seed 1**: Discount factors of 0.8 and 0.6 again led to the optimal path. A discount factor of 0.4 resulted in the agent reaching the goal, even not at the optimal cost, while at 0.2, it oscillated without reaching the goal.

- **Seed 2**: Only a discount factor of 0.8 reached the optimal path; lower values (0.6, 0.4, and 0.2) failed to reach the goal due to oscillations.

In reinforcement learning, rewards provide direct feedback from the environment, reflecting the outcomes of an agent's actions. Therefore, the total reward obtained in each episode helps assess the agent’s performance.

With a discount factor of 0.8, the learning process is stable, and rewards increase consistently over episodes. The low final path cost indicates that the path to the goal is optimized. A high discount factor helps the agent focus on long-term goals, facilitating the discovery of optimal paths.

At a discount factor of 0.6, the agent initially shows significant volatility in rewards but gradually stabilizes, indicating a balance between short-term and long-term rewards. However, the initial instability suggests that more time is needed to find the optimal path.

With a discount factor of 0.4, significant oscillations in the reward graph indicate frequent exploration of costlier, suboptimal paths. A lower discount factor promotes a myopic learning approach, focusing on immediate rewards, which can obstruct the learning of strategies necessary for long-term success.

At the extremely low discount factor of 0.2, the agent experiences high reward volatility and unstable learning, often failing to find the optimal path. Such a low discount factor excessively prioritizes immediate rewards, preventing the agent from developing effective long-term strategies and leading to inefficient learning outcomes.

In summary, a higher discount factor values long-term rewards, aiding in stable and efficient pathfinding in the learning process. In contrast, a lower discount factor causes the agent to overlook long-term optimal strategies in favor of immediate rewards, which can hinder achieving the final goals and degrade overall problem-solving capabilities.

References

- CS 188 Spring 2024. (2024). *Introduction to Artificial Intelligence*. University of California, Berkeley.

- Russell, S. J., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.