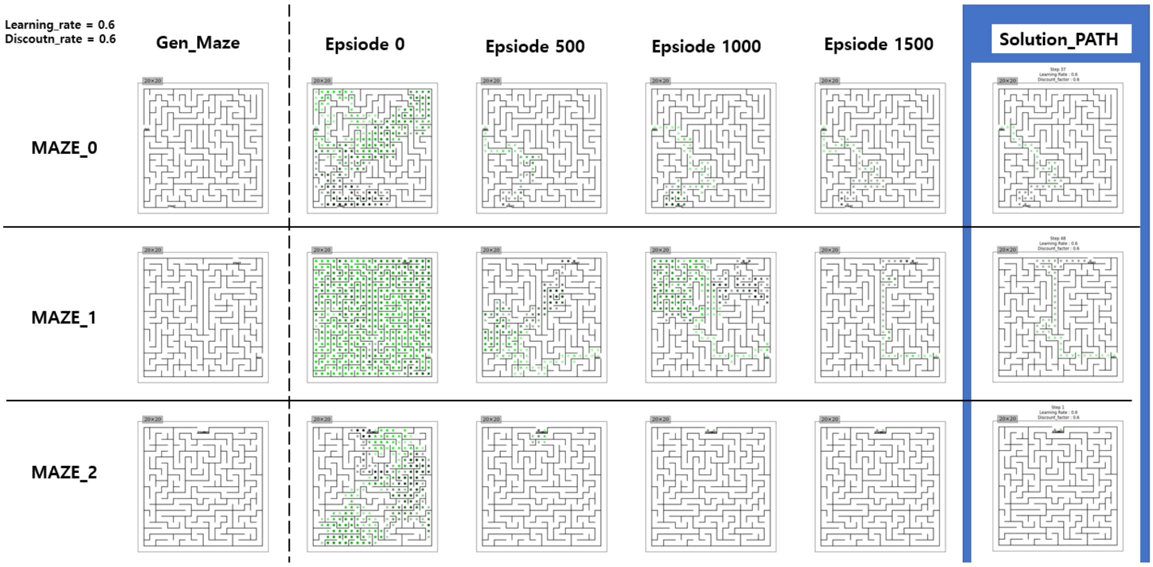
Homework#2: Q-Learning for Maze Traversal

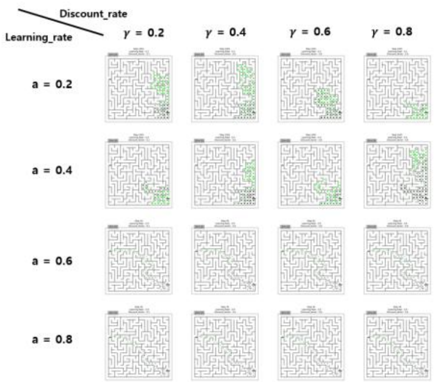
Name : 임용성 (Lim Yong Sung)

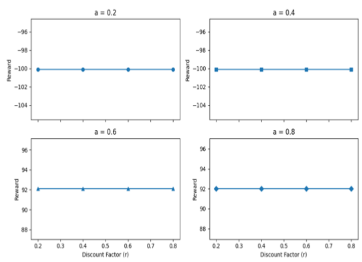
Student ID : 2019310649

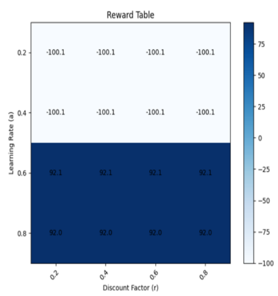
[목차]

<Summary>









Ⅰ. Background

According to the wiki, Q-Learning is a model-free reinforcement learning technique used to find the optimal policy for a given finite Markov decision process (MDP). In an MDP, the agent interacts with the environment and learns the optimal policy by receiving feedback in the form of rewards.

An MDP follows the Markov Property, which means that the current state completely characterizes the state of the world, and the future is only concerned with the present, not the past. It is defined by a tuple (S, A, R, P, r), where:

- S is the set of states in the environment.
- A is the set of actions that the agent can take.
- R is the reward function that provides the immediate reward when an action is taken in a particular state.
- P is the state transition probability function, which defines the probability of transitioning to a new state when an action is taken in a current state.
- r is the discount factor that determines the importance of future rewards.

An MRP (Markov Reward Process) is a Markov chain augmented with rewards. The goal in an MRP is to maximize the return, which is a measure of the cumulative rewards obtained over time. The return is influenced by the discount rate, denoted as γ.

The discount rate serves two purposes. First, it helps avoid infinite returns in cyclic Markov processes by assigning less weight to future rewards as the time horizon extends. This prevents the return from growing unbounded and ensures convergence. Second, the discount rate reflects the degree of uncertainty about future rewards. A smaller discount rate values immediate rewards more and considers future rewards with less weight, indicating a more myopic decision-making approach. Conversely, a larger discount rate values future rewards more and takes a more forward-looking perspective.

The return in an MRP is computed by summing the discounted rewards over time. The formula for the return is:

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Here, the discount rate determines the relative importance of immediate rewards versus future rewards. It allows for trade-offs between immediate gains and long-term benefits, reflecting the time preference of the decision-maker.

By maximizing the return in an MRP, we aim to find policies that achieve the highest cumulative rewards over time, considering both the immediate rewards and the potential future rewards discounted by γ.

The Bellman equation for an MRP expresses the relationship between the value function of the current state and the value function of the next state.

$$\begin{aligned} v(s) &= \mathbb{E}[G_t \mid S_t = s] \\ &= \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \mid S_t = s\right] \\ &= \mathbb{E}[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) \mid S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s] \longrightarrow v(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s') \end{aligned}$$

In an MDP, which is an extension of MRP, actions to be taken by agents are included. The value function of an MDP is expressed as follows:

- **The state-value function** $v_\pi(s)$ of an MDP is the expected return starting from state s, and then following policy π.

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s]$$

- **The action-value function** $q_\pi(s,a)$ is the expected return starting from state s, taking action a, and then following policy π.

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]$$

To maximize the return in an MDP, Q-Learning aims to find the optimal policy by maximizing the Q-value function Q(s, a), which represents the expected cumulative reward when taking action a in state s. The Bellman Optimality Equation provides the theoretical foundation for Q-Learning and is given by:

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') | s, a\right]$$

Here, Q* represents the optimal Q-value function, R(s, a) is the immediate reward, γ is the discount factor, P(s' | s, a) is the transition probability, and max Q*(s', a') represents the maximum Q-value over all possible actions in the next state.

The Q-Learning algorithm updates the Q-values iteratively based on the observed rewards and transitions. The Q-Learning update equation is:

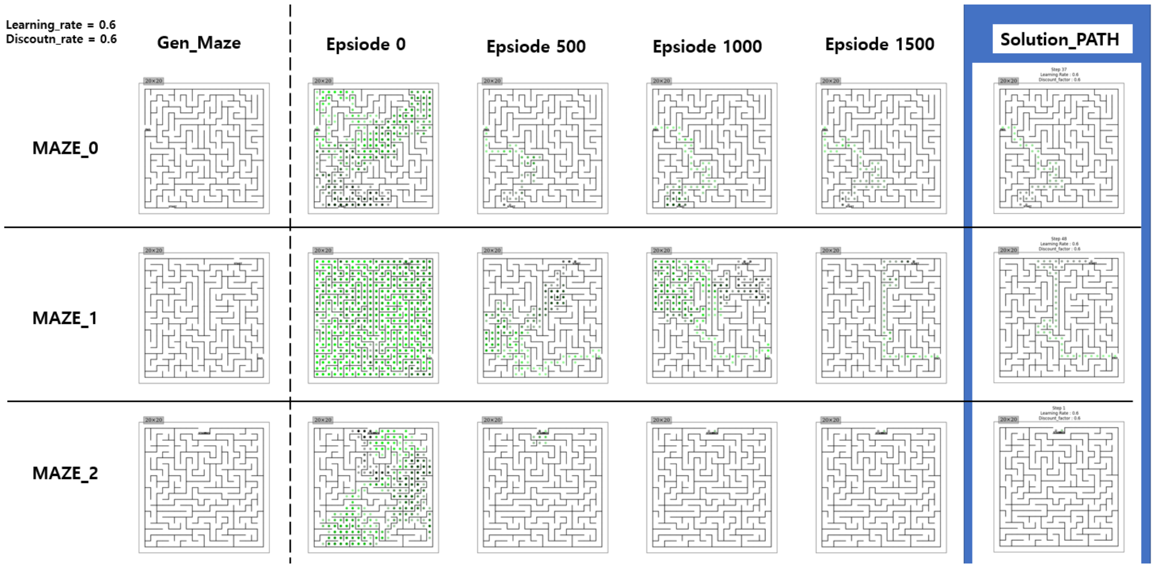
$$Q(s_t,a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} \right)$$

Here, Q(s, a) is the Q-value for state s and action a, α is the learning rate that determines the weight given to new information, R(s, a) is the immediate reward, γ is the discount factor, and max Q(s', a') represents the maximum Q-value over all possible actions in the next state.

By applying Q-Learning iteratively, the agent can learn the optimal Q-values for each state-action pair and ultimately determine the optimal policy for

Ⅱ. Result and Analysis

Let's examine the progression of the Episode by referring to the following image.



Let me analyze the picture above.

First of all, since the Exploration Rate is set high in the early stages of low Epsiode and all Q_Table are similar in initialization state, the Agent updates the Q_Table as it explores most of the space. An episode ends when an agent randomly accidentally arrives at the destination. And as Epsiode progresses, the agent's Exploration Rate gradually decreases, so it tends to rely on Q_Table (dependent on experience). For reference, as I will explain later, in the code I made, if there are multiple same Max values when walking on the street based on Q_Table, it is set to go in a random direction among the directions that are max values. Anyway, as Episode progresses, the agent updates the Q_Table and becomes increasingly dependent on experience.

There is an important point here. In fact, after a long period of code analysis and research, the agent was observed to move back and forth (Ex, the agent moves left and right indefinitely), and as a result of checking the Q_Table in this case, the same Left and Right existed at the same value (for example, -0.166 or -0.125). To understand this reason, let's look at the equation of Q-Learning.

$$Q(s_t,a_t) \leftarrow (1-\alpha)\cdot \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} \right)$$

Notice the inside of the right parenthesis in the above equation.

As explained in Background, Q-Learning gradually updates the Q_table of the final destination's Reward in the form of Backpropagation, and the final destination's Reward affects the way to the final destination, increasing its value. However, if the Discount Rate is very low, it converges to zero of the impact of Reward at the final destination, and most of the values converge to a specific value. The value is the value of zero inside the right bracket in the above equation.

Therefore, if the maze is long from the start position to the end position, the smaller the Discount Rate, the smaller the impact on the final destination, so that most Q_Table around the start position converges to the same value and repeatedly moves back and forth in the space!

As a way to solve this problem, it was possible to solve the problem by significantly increasing the Discount Rate.

Now let's take a closer look at the picture above. As shown in the figure above, it can be seen that there is less tendency to pierce walls well in the beginning, and the closer to the final destination, the greater the tendency to pierce walls.

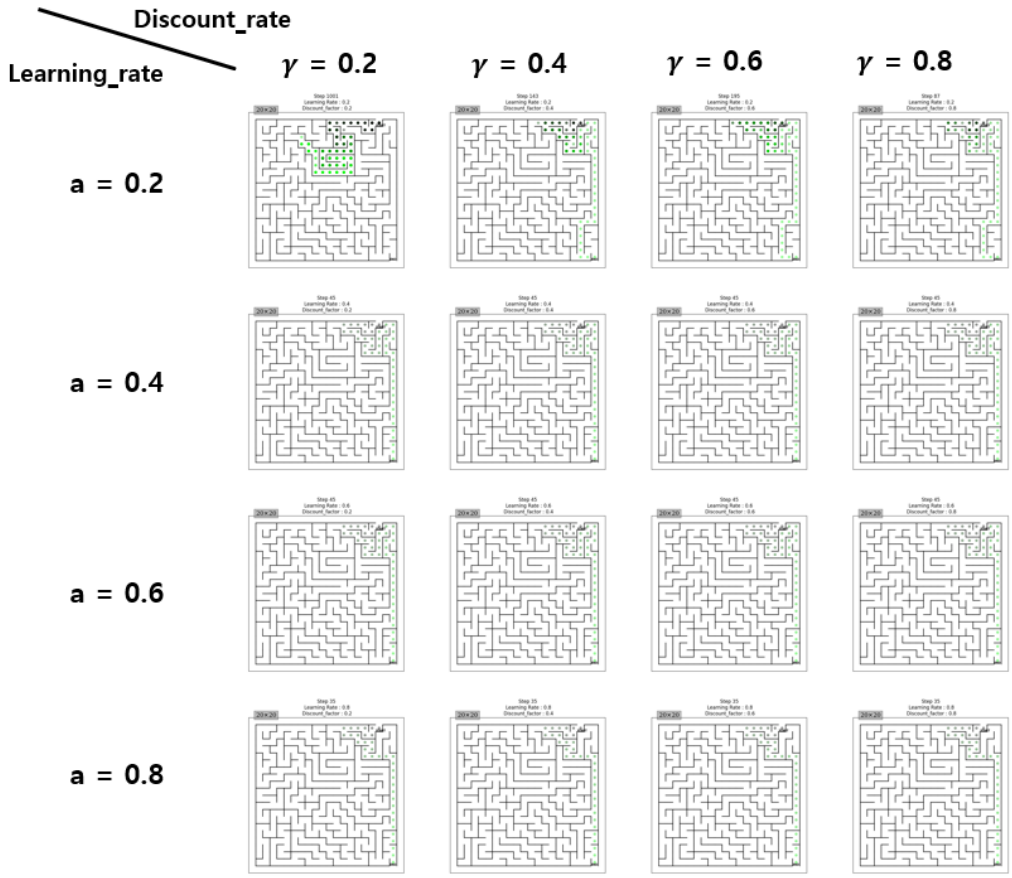
This is due to the fact that when Reward reaches its final destination, +100, -1 when drilling a wall, and -0.1 when just moving, the closer it is to the final destination, the less Discount multiplied, so even if the wall is pierced and reaches the final destination, it has a significant impact on the update of the Q-Table. Therefore, this trend becomes stronger toward the second half, so the tendency is written in the form of a compensation in which the updated value of the Q-Table breaks through a large wall (-1) rather than going back along the road. In the vicinity of the Start position, the effect of the final reward +100 has become very small because the Discount Rate has been multiplied several times, so the Reward (-1) when the wall is drilled is operated larger. The following is part of the Q-Table.

(15, 5)	-1.1498	-0.2500	-1.1500	-0.2500
(15, 6)	-1.1499	-0.2500	-0.2500	-1.1499
(15, 7)	-0.2500	-1.1500	-0.2500	-1.1498
(15, 8)	-0.2500	-1.1500	-0.2500	-1.1500
(15, 9)	-0.2500	-1.1500	-1.1499	-0.2500

As you can see in the figure, there is a lot of -1.15. Also, interestingly, if it's not -1.15, you can see a difference of -0.25, i.e. when it's barely pierced the wall (-1 + 0.1 = 0.9).

For this reason, it can be seen that less wall drilling is performed around the starting position, and the tendency to wall drilling becomes stronger as we reach the final destination.

For a maze, let's observe what happens when you sweep the Learning Rate and Discount Rate.



As shown in the figure above, the smaller the learning_rate(a), the smaller the size of the Q_table is, the smaller the size of the Q_table's update, so even if the final result is reached and Reward (100), it eventually converges to a specific value as Episode is repeated.

So we can see that 0.2 or 0.4 where a is very small does not reach the final destination.

Because the above maze problem is difficult (because the distance is long from the start position to the target position) it is difficult to learn with these small learning rates.(The update size is small, so the final reward cannot affect and eventually converges all around the starting position, resulting in the Agent repeating only the same position) Note that most cases , it will repeat about 5 spaces or less from the starting position, but in my code, it is designed to behave randomly.

Now let's understand the impact of the Discover Rate. I explained the impact of Discount above, and it is the same. While the Learning Rate governs the overall size of updates, the Discount Rate more directly affects the reduction of final rewards. Smaller Discount Rate converges faster, so you become shortsighted, and the larger the Discount Rate, the more far you can see. Therefore, in order not to face the same converging problem, the Discount Rate must be set according to the tendency of the maze to some extent.

Ⅲ. Code Description

It is a main code that performs TASK 1 to 5 as required in the task. It is important to note that Task 1 to 4 made three Maze and compared each Maze, and Task 5 made another Maze to confirm each tendency as the Learning Rate and Discount Rate were changed.

```python
from __future__ import absolute_import
import os
import sys
import shutil

from matplotlib import colors, pyplot as plt
import numpy as np
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))
from src.maze import Maze
from src.maze_manager import MazeManager
from src.debug_viz import DebugViz

#===========+==========================================================================#
# Task 1 & 2 | Create Function about Q-Learning : q_learning & q_learning_path    #
#===========+==========================================================================#
from src.q_learning_manager import QLearningManager
print("==========================")
print("     Task 1,2 Complete")
print("==========================")

shutil.rmtree('./temp')
shutil.rmtree('./output')

#===========+======================================================================#
# Task 3 | Train 3 randomly generated 20x20 mazes with Visualization          #
#===========+======================================================================#
maze_row, maze_col = 20, 20
manager = MazeManager()
maze_list = []
solver_list = []

for i in range(3):
    maze = manager.add_maze(maze_row, maze_col)
    maze_list.append(maze)
    solver = QLearningManager(maze)
    debugViz = DebugViz()
    debugViz.set_filename("maze{}_generation_20x20".format(i))
    solver_list.append(solver)
    debugViz.show_maze(maze, display_mode=False)

# Define the image file names and paths
manager.combine_images('temp', 'generation', 'maze_output1.png', 'down')

print("==========================")
print("     Task 3 Complete")
print("==========================")
```

```python
#=======+======================================================================================#
# Task 4 | Implement the Q-Learning-based maze traversal algorithm for the randomly generated 20x20 mazes #
#=======+======================================================================================#
solution_path_list = []
solution_cost_list = []
solution_q_table = []

debugViz = DebugViz()
for i in range(3):
    solver_list[i].q_learning(maze_list[i], episodes = 1500, learning_rate = 0.6, discount_factor = 0.6, exploration_rate = 0.95)
    manager.combine_images('temp', 'episode', 'maze{}_episode_solution.png'.format(i), 'right')
    solution_q_table.append(solver_list[i].q_table)
    maze_list[i].solution_path, maze_list[i].solution_cost, maze_list[i].solution_reward = solver_list[i].q_learning_path(maze_list[i], solution_q_table[i])

    debugViz.show_solution(maze_list[i], display_key = False, media_name="maze{}".format(i), learning_rate = 0.6, discount_factor = 0.6)
    solution_path_list.append(maze_list[i].solution_path)
    solution_cost_list.append(maze_list[i].solution_cost)

manager.combine_images('temp', 'solution','maze_q_learning_solution.png', 'down')

manager.combine_images('output', '_episode_','maze_output2.png', 'down')
manager.combine_images('output', '_output','maze_episode_table.png', 'right')


print("==========================")
print("     Task 4 Complete")
print("==========================")
# F. Use Visualization Func(at Codebase). Demonstartes the algorithm's performance from the 3 ramdonly gen. mazes
print("\nShow Solution Animation")
#solver_list[i].show_solution_animation(maze, media_name=f"Q-Learing_maze{i}")
for i in range(3):
    debugViz.show_solution_animation(maze_list[i])
```

```python
#=======+======================================================================================#
# Task 5 | Choose at least two params and change values of them to see the impact of the parameters.      #
#        |          e.g., a=(0.2, 0.4, 0.6, 0.8) and r=(0.2, 0.4, 0.6, 0.8)                              #
#=======+======================================================================================#
shutil.rmtree('./temp')
solution2_path_list  = []
solution2_cost_list  = []
solution2_q_table    = []
solution2_reward     = []
solver_list         = []


a_list = [0.2, 0.4, 0.6, 0.8]
r_list = [0.2, 0.4, 0.6, 0.8]

manager_param = MazeManager()
manager_param.media_name="maze_for_sweep_show"
maze_param = manager_param.add_maze(maze_row, maze_col)
debugViz_param = DebugViz()
manager_param.show_maze(maze_param.id)

i = 0
j = 0
for a in a_list:
    solution2_path_list.append([])
    solution2_cost_list.append([])
    solution2_reward.append([])
    for r in r_list:
        #Make Solver Manager
        solver_param = QLearningManager(maze_param)
        solver_list.append(solver_param)

        #Learn Q Table
        solver_list[i].q_learning(maze_param, episodes = 1500, learning_rate = a, discount_factor = r, exploration_rate = 0.95)
        solution2_q_table.append(solver_list[i].q_table)

        # Solve Maze Using Learned Q_ TABLE
        temp_solution_path, temp_solution_cost, temp_solution_reward = solver_list[i].q_learning_path(maze_param, solution2_q_table[i])
        solution2_path_list[i].append(temp_solution_path)
        solution2_cost_list[i].append(temp_solution_cost)
        solution2_reward[i].append(temp_solution_reward)
        debugViz_param.show_solution(maze_param, display_key = False, media_name="maze_a_{}_r_{}_out".format(a, r),  learning_rate = a, discount_factor = r)
        j+=1

    manager_param.combine_images('temp', '_out', 'maze_a{}_solution.png'.format(a), 'right')
    i+=1

manager_param.combine_images('output', 'maze_a0.', 'maze_a_r_sweep_table.png', 'down')
```

From now on, the Q-Learning Path, which shows the learning code and solution path of Q-Learning, is described as follows.

```
def q_learning(self, maze, episodes, learning_rate, discount_factor, exploration_rate) :
```

```python
def q_learning(self, maze, episodes, learning_rate, discount_factor, exploration_rate) :
    # A. Initialize the Q-Table with zeros and set values of parameters by yourself.
    self.q_table = self._initialize_q_table(maze)
    maze.solution_path = None
    maze.solution_cost = None
    self.solution_reward = None
    print("   Take some minute ... Pleas Wait ...")
    action_list = ["LEFT", "UP", "RIGHT", "DOWN"]
    for episode in range(episodes):
        temp_episode_path = list()
        steps = 0
        epi_cost = 1
        total_reward = 0
        self.before_action = None
        #Predefined Maximum number of iteration
        max_iter = 10000
        #print for debug
        if((episode+1) % (episodes//3) == 0):
            print("===========================================")
            print("Episode : {0:4}, max_iter: {1:5}, a: {2:2.1f} r: {3:2.1f}  e: {4:3.2f}  | Start ...".format(episode+1, max_iter, learning_rate, discount_factor, exploration_rate))
        # B. Set the Initial State (the starting position of the agent).
        current_state = maze.entry_coor
        # G. Repeat Steps c-f until the agent reaches the gaol or a predefined maximum number of iterations is reached.
        while (current_state != maze.exit_coor):
            if(max_iter == 0 and episode > 100):
                print("Episode {} is Ended Becuase of Max_iteration set".format(episode))
                print("==================================================")
                break
            max_iter -= 1
            steps += 1
            temp_episode_path.append((current_state, False))
            # C. Choose an action based on the current state and the Q-Table, using the e-greedy staregy.
            action = self._choose_action(current_state, exploration_rate, self.q_table)   #Return ex) LEFT
            # D. Perform the action and observe the reward and the next state.
            next_state = self.do_action(maze, current_state, action)
            reward = self.get_reward(maze, current_state, next_state)
            action_idx = self.find_word_idx(action, action_list)
            self.q_table[current_state][action_idx] += learning_rate * (reward + discount_factor * max(self.q_table[next_state]) - self.q_table[current_state][action_idx])
            total_reward += reward
            # F. Set the next state as the current state..
            current_state = next_state
            epi_cost += 1
        temp_episode_path.append((current_state, False))
        # Reduce the exploration rate over time.
        exploration_rate = exploration_rate * 0.999
        #------------This is for Debugging-------------------------
        #print log
        self.log_iteration_info(episode, episodes, epi_cost, total_reward, max_iter)
        if (episode == 0 or (episode + 1) % (episodes//3) == 0):
            maze.solution_path = temp_episode_path
            debugViz = DebugViz()
            debugViz.show_solution(maze, episodes = episode ,display_key = False, media_name=f"solve_maze_episode{episode+1}")
            output_folder = os.path.join(os.path.dirname(os.path.abspath(__file__)), '../', 'temp')                    # output 폴더 생성
            if not os.path.exists(output_folder):
                os.makedirs(output_folder)
            filename = os.path.join(output_folder, f"Q_Table_of_Episode{episode+1}.txt") # 파일 생성 및 저장
            debugViz.save_q_table(self.q_table, filename)
    print("=======================================================")
```

The above code is the learning code of Q-Learning implemented by the method required by the task. As can be seen in the figure, the method required in the task was satisfied. In particular, the Q_table update code of Q-Learning is as follows.

```
self.q_table[current_state][action_idx]
+= learning_rate * (reward + discount_factor * max(self.q_table[next_state]) - self.q_table[current_state][action_idx])
```

It can be seen that the requirements of the assignment were met.

Debugging messages are printed through Print to make it easier for users to understand, and Choose Action, Do Action, and Get Reward functions are called and used. This will be explained later. In the case of the Exploration Rate, it was gradually reduced as the Episode progressed, and through this, as the Episode progressed, it was searched based on experience. Max_Iteration was set to a very large number of 10000, which was set in this way to reach the final destination because of the nature of Q-Learning, the Reward of the final destination is transmitted to the starting position. Therefore, in the case of the initial 100 Episode, the agent continued to move unless the final destination was reached.

```python
    #==========================================================================
    def get_reward(self, maze, current_state, next_state):
        if next_state == maze.exit_coor:
            reward = 100
        elif maze.is_wall(next_state, current_state):
            reward = -1
        else:
            reward = -0.1
        return reward
```

As shown on the left, the compensation function returns 100 when reaching the final destination, -1 when hit by a wall, and -0.1 when just moving.

```python
    def is_wall(self, next_state, current_state):

        #If Agent go Out side Maze
        if(next_state == current_state):
            return True

        next_k, next_i = next_state
        curr_k, curr_i = current_state

        # action == "LEFT":
        if(next_i == curr_i - 1):
            action = "left"
        # action == "UP":
        if(next_k == curr_k + 1):
            action = "bottom"
        # action == "RIGHT":
        if(next_i == curr_i + 1):
            action = "right"
        # action == "DOWN":
        if(next_k == curr_k - 1):
            action = "top"
        # print(current_state)
        # print("action: "+ action)
        # print("is wall: " + str(self.grid[curr_k][curr_i].walls))
        # print(self.grid[curr_k][curr_i].walls[action])
        # input("")

        return self.grid[curr_k][curr_i].walls[action]
```

Earlier, the compensation function used the is_wall function to determine whether it was a wall, and this was determined through the internal walls variable of the grid class. There was a problem in which Top and Bottom were matched in reverse in a given code, and the is_wall function solved the problem by writing it in reverse. In the is_wall function, the action is determined based on the current position and the next position to determine whether it is a wall.

What is important here is that in the Do_action function, the code was designed so that the next position of the agent would return to its current position if the agent acted to go out of the labyrinth, which prevented the agent from exiting the labyrinth. Here, we can see that the is_wall function returns True if the next position and the current position are the same, so when Get Reward, it returns the action of trying to go out of the maze as the Reward of the action of hitting the wall.

```python
    def _initialize_q_table(self, maze):
        q_table = {}
        for row in range(maze.num_rows):
            for col in range(maze.num_cols):
                state = (row, col)
                q_table[state] = [0, 0, 0, 0]  #Actions :[LEFT, RIGHT, UP, DOWN]
        return q_table

    def find_argmax(self, lst):
        if not lst:
            print("No List ERROR at find_argmax")
            return None  # Handle empty list
        max_value = lst[0]
        max_index = 0
        for i, value in enumerate(lst):
            if value > max_value:
                max_value = value
                max_index = i
        return max_index


    def find_word_idx(self, word, list):
        for idx in range(len(list)):
            if(word == list[idx]):
                return idx
        print("Cant find Word IDX ERROR")
        return None

    def _random_argmax_word(self, action_q_list):
        max_value = max(action_q_list)
        max_indices = [i for i, value in enumerate(action_q_list) if value == max_value]
        selected_index = random.choice(max_indices)
        action_list = ["LEFT", "UP", "RIGHT", "DOWN"]
        return action_list[selected_index]
```

The left code is the base code used by default.

The Find_Argmax function finds the index of the Max value in the list.

Find_word_Idx is used when you want to find the index number of the Action, "Left", in a list containing the Action.

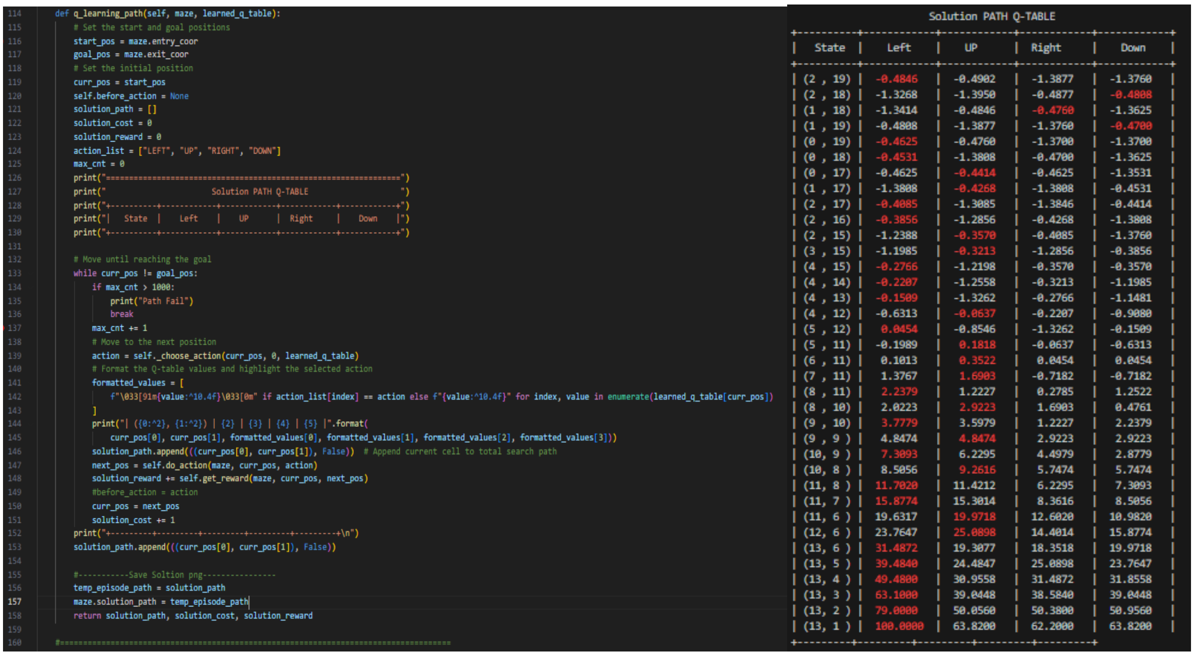
The _random_argmax_word is a function that returns a random action among the same values if the MAX value is the same among the Q values for the action at any position in the Q_Table. Based on this, when converging to the same value, it goes back and forth from side to side or up and down It is designed to avoid moving situations. If you are lucky to move randomly like this and get to where the final reward affects you, you can follow the MAX Q and move to the final position.

```python
    def _choose_action(self, current_coors, exploration_rate, q_table):
        """
        Actions :[LEFT, UP, RIGHT, DOWN]
        """
        # Using e-greedy staregy
        # Cant' go before State

        action_q_list = q_table[current_coors]
        action_list = ["LEFT","UP","RIGHT","DOWN"]
        if random.uniform(0, 1) < exploration_rate:
            random.shuffle(action_list)
            explor_act = action_list[0]
        else:
            explor_act = self._random_argmax_word(action_q_list)

        return explor_act #LEFT, UP, RIGHT, DOWN

    def do_action(self, maze, current_state, action):
        curr_k, curr_i = current_state

        if action == "LEFT":
            next_k = curr_k
            next_i = curr_i - 1
        elif action == "UP":
            next_k = curr_k + 1
            next_i = curr_i
        elif action == "RIGHT":
            next_k = curr_k
            next_i = curr_i + 1
        elif action == "DOWN":
            next_k = curr_k - 1
            next_i = curr_i
        else:
            raise ValueError("Invalid action value: {}".format(action))

        if(next_k < 0 or next_i < 0 or next_k >= maze.num_rows or next_i >= maze.num_cols):
            next_k = curr_k
            next_i = curr_i
            #print("Stupid Act : Can't go Outside Maze")

        next_state = (next_k, next_i)

        return next_state
```

The _choose_action function implements the E-Greedy strategy required in the task. Based on this, the agent experiences a direction other than the direction in which the Q value is maximum, creating a better Q_Table, and finding the optimal movement path while enjoying several places. As you can see from the code, the reciprocating problem was solved by using the above-described _random_argmax_word when it was not in an exploration situation.

The do_action function is a function that, given an action, simply moves the agent in that direction and returns information about the next location.

```
def q_learning_path(self, maze, learned_q_table):
```



```python
def q_learning_path(self, maze, learned_q_table):
    # Set the start and goal positions
    start_pos = maze.entry_coor
    goal_pos = maze.exit_coor
    # Set the initial position
    curr_pos = start_pos
    self.before_action = None
    solution_path = []
    solution_cost = 0
    solution_reward = 0
    action_list = ["LEFT", "UP", "RIGHT", "DOWN"]
    max_cnt = 0
    print("================================================================")
    print("                       Solution PATH Q-TABLE                    ")
    print("+---------+-----------+-----------+-----------+-----------+")
    print("|  State  |    Left   |    UP     |  Right    |    Down   |")
    print("+---------+-----------+-----------+-----------+-----------+")

    # Move until reaching the goal
    while curr_pos != goal_pos:
        if max_cnt > 1000:
            print("Path Fail")
            break
        max_cnt += 1
        # Move to the next position
        action = self._choose_action(curr_pos, 0, learned_q_table)
        # Format the Q-table values and highlight the selected action
        formatted_values = [
            f"\033[91m{value:^10.4f}\033[0m" if action_list[index] == action else f"{value:^10.4f}" for index, value in enumerate(learned_q_table[curr_pos])
        ]
        print("| ({0:^2}, {1:^2}) | {2} | {3} | {4} | {5} |".format(
            curr_pos[0], curr_pos[1], formatted_values[0], formatted_values[1], formatted_values[2], formatted_values[3]))
        solution_path.append(((curr_pos[0], curr_pos[1]), False))  # Append current cell to total search path
        next_pos = self.do_action(maze, curr_pos, action)
        solution_reward += self.get_reward(maze, curr_pos, next_pos)
        #before_action = action
        curr_pos = next_pos
        solution_cost += 1
    print("+---------+---------+---------+---------+---------+\n")
    solution_path.append(((curr_pos[0], curr_pos[1]), False))

    #----------Save Soltion png---------------
    temp_episode_path = solution_path
    maze.solution_path = temp_episode_path
    return solution_path, solution_cost, solution_reward

#==========================================================================
```

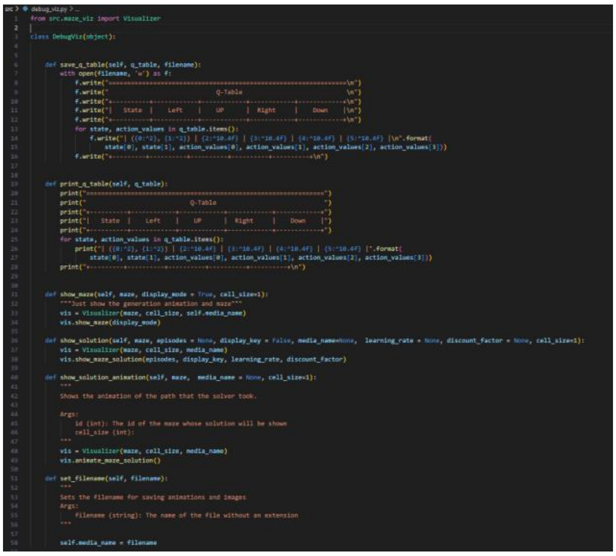
Solution PATH Q-TABLE

State	Left	UP	Right	Down
(2 , 19)	-0.4846	-0.4902	-1.3877	-1.3760
(2 , 18)	-1.3268	-1.3950	-0.4877	-0.4808
(1 , 18)	-1.3414	-0.4846	-0.4760	-1.3625
(1 , 19)	-0.4808	-1.3877	-1.3760	-0.4700
(0 , 19)	-0.4625	-0.4760	-1.3700	-1.3700
(0 , 18)	-0.4531	-1.3808	-0.4700	-1.3625
(0 , 17)	-0.4625	-0.4414	-0.4625	-1.3531
(1 , 17)	-1.3808	-0.4268	-1.3808	-0.4531
(2 , 17)	-0.4085	-1.3085	-1.3846	-0.4414
(2 , 16)	-0.3856	-1.2856	-0.4268	-1.3808
(2 , 15)	-1.2388	-0.3570	-0.4085	-1.3760
(3 , 15)	-1.1985	-0.3213	-1.2856	-0.3856
(4 , 15)	-0.2766	-1.2198	-0.3570	-0.3570
(4 , 14)	-0.2207	-1.2558	-0.3213	-1.1985
(4 , 13)	-0.1509	-1.3262	-0.2766	-1.1481
(4 , 12)	-0.6313	-0.0637	-0.2207	-0.9080
(5 , 12)	0.0454	-0.8546	-1.3262	-0.1509
(5 , 11)	-0.1989	0.1818	-0.0637	-0.6313
(6 , 11)	0.1013	0.3522	0.0454	0.0454
(7 , 11)	1.3767	1.6903	-0.7182	-0.7182
(8 , 11)	2.2379	1.2227	0.2785	1.2522
(8 , 10)	2.0223	2.9223	1.6903	0.4761
(9 , 10)	3.7779	3.5979	1.2227	2.2379
(9 , 9)	4.8474	4.8474	2.9223	2.9223
(10, 9)	7.3093	6.2295	4.4979	2.8779
(10, 8)	8.5056	9.2616	5.7474	5.7474
(11, 8)	11.7020	11.4212	6.2295	7.3093
(11, 7)	15.8774	15.3014	8.3616	8.5056
(11, 6)	19.6317	19.9718	12.6020	10.9820
(12, 6)	23.7647	25.0898	14.4014	15.8774
(13, 6)	31.4872	19.3077	18.3518	19.9718
(13, 5)	39.4840	24.4847	25.0898	23.7647
(13, 4)	49.4800	30.9558	31.4872	31.8558
(13, 3)	63.1000	39.0448	38.5840	39.0448
(13, 2)	79.0000	50.0560	50.3800	50.9560
(13, 1)	100.0000	63.8200	62.2000	63.8200

The above figure is a function to find solution_Path based on the Q_Table required in the task. As you can see in the figure, Print was added for debugging, which is printed as shown in the left picture. Based on this, it is easy to understand what value of the Q_Table, in which direction the Agent moves, and what the next state has become, and debugging is easy.

The important thing is to find the solution path based only on Q_TABLE, so by putting 0 in the Exploration Factor of the _Choose_Action function, the agent always relied on Q_TABLE.

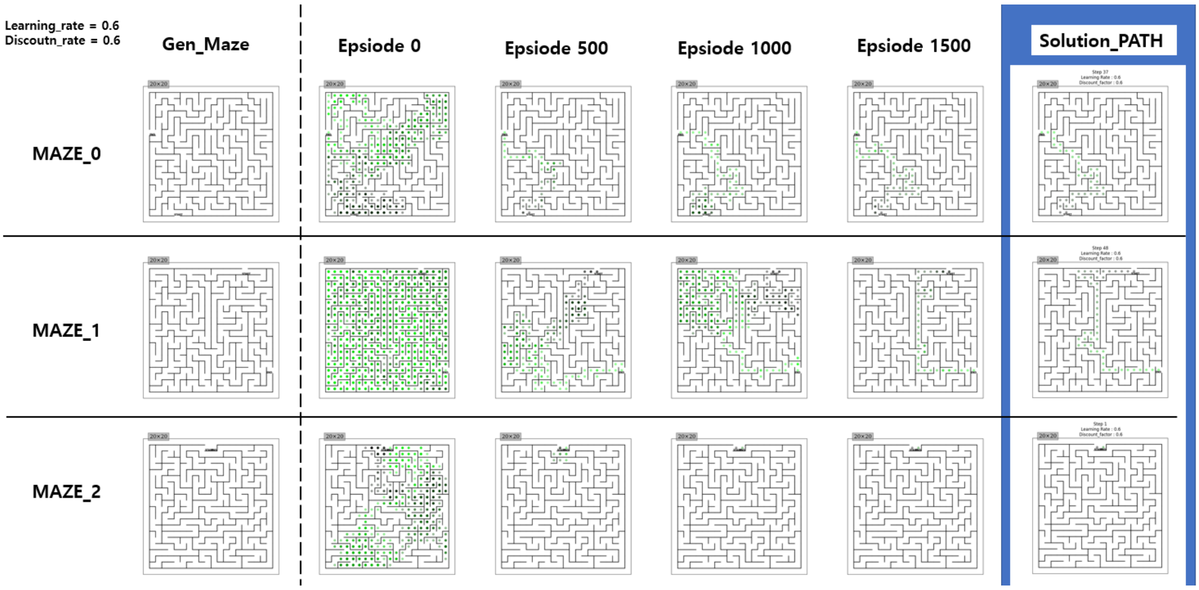
It's not important for the rest of the visualization code, so I'll just attach the code and skip it.



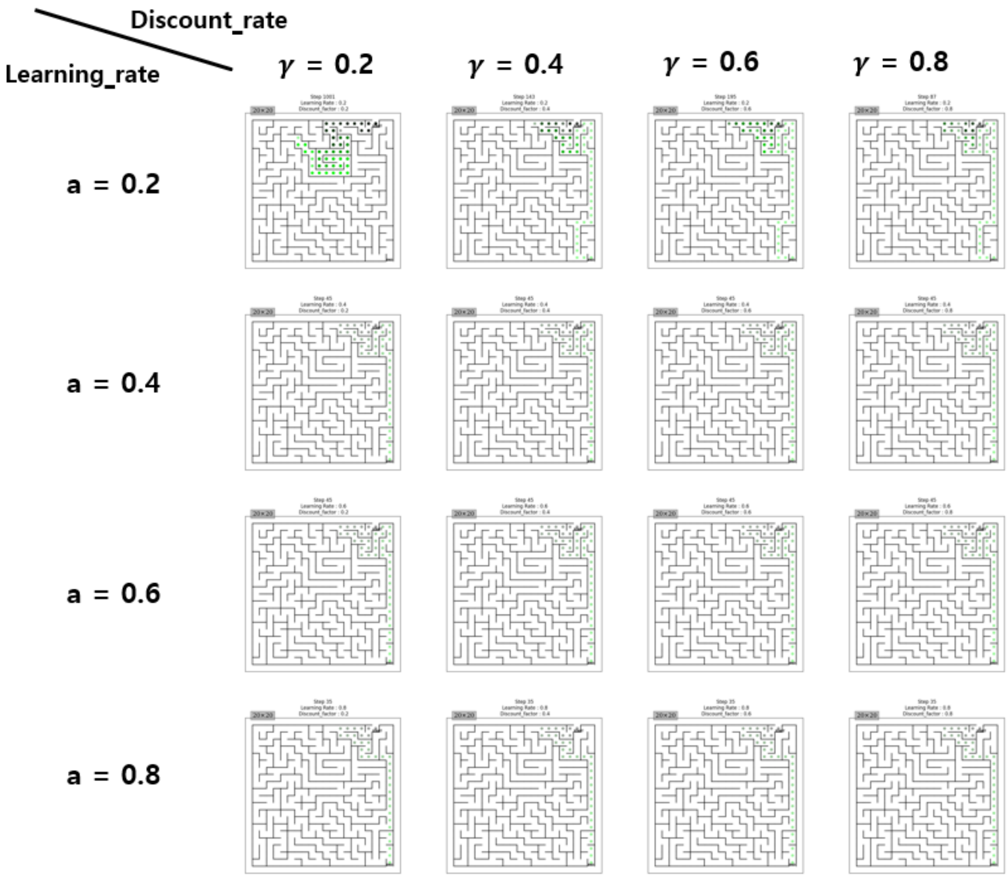
```python
from src.maze_viz import Visualizer

class DebugViz(object):


    def save_q_table(self, q_table, filename):
        with open(filename, 'w') as f:
            f.write("================================================================\n")
            f.write("                           Q-Table                              \n")
            f.write("+---------+-----------+-----------+-----------+-----------+\n")
            f.write("|  State  |    Left   |    UP     |  Right    |    Down   |\n")
            f.write("+---------+-----------+-----------+-----------+-----------+\n")
            for state, action_values in q_table.items():
                f.write("| ({0:^2}, {1:^2}) | {2:^10.4f} | {3:^10.4f} | {4:^10.4f} | {5:^10.4f} |\n".format(
                    state[0], state[1], action_values[0], action_values[1], action_values[2], action_values[3]))
            f.write("+---------+---------+---------+---------+---------+\n")


    def print_q_table(self, q_table):
        print("================================================================")
        print("                           Q-Table                              ")
        print("+---------+-----------+-----------+-----------+-----------+")
        print("|  State  |    Left   |    UP     |  Right    |    Down   |")
        print("+---------+-----------+-----------+-----------+-----------+")
        for state, action_values in q_table.items():
            print("| ({0:^2}, {1:^2}) | {2:^10.4f} | {3:^10.4f} | {4:^10.4f} | {5:^10.4f} |".format(
                state[0], state[1], action_values[0], action_values[1], action_values[2], action_values[3]))
        print("+---------+---------+---------+---------+---------+\n")


    def show_maze(self, maze, display_mode = True, cell_size=1):
        """Just show the generation animation and maze"""
        vis = Visualizer(maze, cell_size, self.media_name)
        vis.show_maze(display_mode)

    def show_solution(self, maze, episodes = None, display_key = False, media_name=None,  learning_rate = None, discount_factor = None, cell_size=1):
        vis = Visualizer(maze, cell_size, media_name)
        vis.show_maze_solution(episodes, display_key, learning_rate, discount_factor)

    def show_solution_animation(self, maze,  media_name = None, cell_size=1):
        """
        Shows the animation of the path that the solver took.

        Args:
            id (int): The id of the maze whose solution will be shown
            cell_size (int):
        """
        vis = Visualizer(maze, cell_size, media_name)
        vis.animate_maze_solution()

    def set_filename(self, filename):
        """
        Sets the filename for saving animations and images
        Args:
            filename (string): The name of the file without an extension
        """

        self.media_name = filename
```

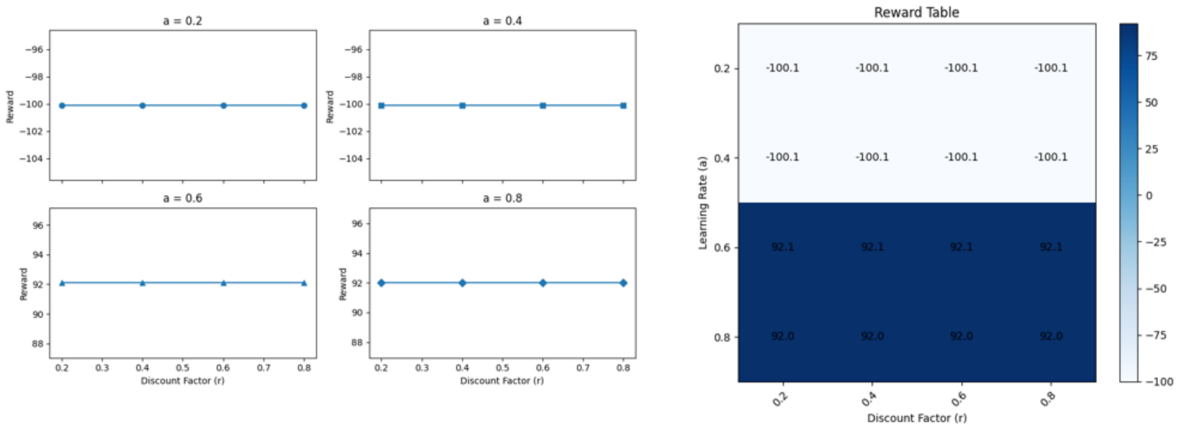
IV. Visualization

TASK 1~4



TASK 5




a = 0.2
a = 0.4
a = 0.6
a = 0.8
Reward
Discount Factor (r)
Reward Table
Learning Rate (a)
-100.1 -100.1 -100.1 -100.1
-100.1 -100.1 -100.1 -100.1
92.1 92.1 92.1 92.1
92.0 92.0 92.0 92.0