HW#2

Name: 이세형

Student Number: 2020312145

1. Implementation:

Initially, the implementation for states, actions, and rewards was as follows. Each coordinate was mapped one-to-one to an array of consecutive numbers to define the state. The code for the 'S2I' function, which maps each coordinate to an index, is as follows.

```
def S2I(state, cnt):  # state -> index
    return state[0] * cnt + state[1]
```

In the current state, the four possible movements—up, down, left, and right—were implemented as actions corresponding to 0, 1, 2, and 3, respectively. Regarding rewards, reaching the goal yields a +100 reward, hitting a wall incurs a penalty, and all other cases incur a -0.1 penalty. The 'go' function was used to determine the next state and reward.

```
def go(state, action, maze):  # return next state, reward, flag
    reward = 0
    next_state = None

    row, col = state

    # 먼저 다음 state 구하기
    if action == 0:  # 상
        next_state = (row + 1, col)
    elif action == 1:  # 하
        next_state = (row - 1, col)
    elif action == 2:  # 좌
        next_state = (row, col - 1)
    else:  # 우
        next_state = (row, col + 1)

    # 다음 state가 유효한 지 확인
    if next_state[0] < 0 or next_state[0] >= maze.num_rows or next_state[1] < 0 or next_state[1] >= maze.num_cols:
        #print(state, " invalid operation! \n")  # for debugging purpose
        reward = -1
        next_state = state

    # wall 존재 유무 판단
    elif maze.grid[state[0]][state[1]].is_walls_between(maze.grid[next_state[0]][next_state[1]]):
        reward = -1
        next_state = state  # 벽이 있어 다음 state로 가지 X
    else:
        reward = -0.1  # 다음 state로 넘어갈 수 있다면,

    # goal에 도달했는지 판단
    if next_state == maze.exit_coor:
        return next_state, 100, True
    else:
        return next_state, reward, False
```

The process of solving a maze using Q-learning can be divided into two major parts: filling the Q table and utilizing it to find the path through the maze. The former is implemented through a function called 'q_learning', while the latter is done through a function called 'q_learning_path'.

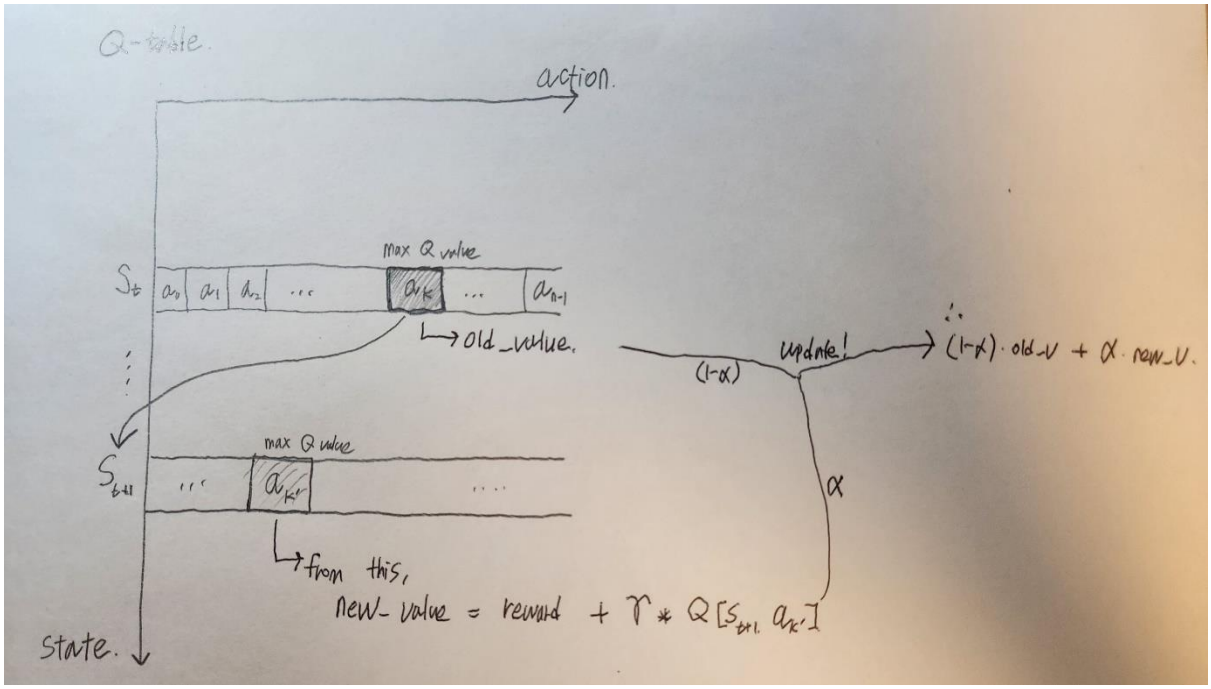
First, the 'q_learning' function is outlined with simple pseudocode and implementation code as follows:

```
func q_learning(maze, episodes, alpha, gamma, epsilon):
        init Q-table (S * A)
        for i from 1 to episodes:  # iterate episodes time!
                if random number < epsilon:
                        pick action randmly
                else:
                        pick action which have max Q-value

                calculate reward, next_state
                old_value <- q_table[current state]
                new_value <- reward + gamma * max(q_table[next_state])

                update q_table

        return q_table
```

```
1 usage
def q_learning(maze, episodes, alpha, gamma, epsilon):  # create Q table
    total_time_start = time.time()  # for experimenting

    S = maze.num_rows * maze.num_cols  # 총 가능한 state의 개수
    A = 4
    """
    action: <integer>
    - 0: 상
    - 1: 하
    - 2: 좌
    - 3: 우

    state: <tuple>
    - (i, j): i행 j열

    q_table: <2D array>
    - [s, a]: Q value
    """
    q_table = np.zeros((S, A))  # init Q table with '0'

    file = open("episodes.txt", "w")  # open file stream
    # episodes번 반복
    for i in range(episodes):
        episode_start = time.time()  # for experimenting
        # init state & flag
        """
        state 정보:
        - s, s_index : 현재 state 정보
        - ns, ns_index : 다음 state 정보

        action 정보:
        - action

        학습 종료 조건 정보:
        - flag
        """
        s = maze.entry_coor
        flag = False

        # print(s, "\n")

        cnt = 0
        while not flag:
            cnt += 1
            # init 작업
            s_index = S2I(s, maze.num_cols)

            #valid_action = get_valid_action1(s, maze)

            # by epsilon greedy policy
            if random.random() < epsilon:
                #action = random.choice(valid_action)
                action = random.choice([0, 1, 2, 3])
            else:
                #q_values = [q_table[s_index, a] for a in valid_action]
                #action = valid_action[np.argmax(q_values)]  # find maximum Q value among current state (meaning row)
```

```
            action = np.argmax(q_table[s_index])

        # (next state, reward, 종료 조건) 구하기
        ns, reward, flag = go(s, action, maze)
        ns_index = S2I(ns, maze.num_cols)

        old_value = q_table[s_index, action]
        new_value = reward + gamma * np.max(q_table[ns_index])
        q_table[s_index, action] = (1 - alpha) * old_value + alpha * new_value  # update Q value

        # for debugging purpose
        #if s == ns:
            #print("Loop!", s, ns, "\n")

        s = ns  # 다음 state로 이동!

    print("successfully reached goal!", "(", i, ") -> ", cnt, "\n")  # for debugging purpose
    file.write(f"{i} episode learning exec time: {time.time() - episode_start}\n")  # for experimenting
    epsilon *= 0.99  # reduce epsilon
    epsilon = max(epsilon, 0.1)  # set under-bound

file.write("Q-learning finished!\n")
file.write(f"Total learning time: {time.time()- total_time_start}\n")  # for experimenting
file.close()  # close file stream

return q_table
```

In the previous assignment, while the A* search and uniform cost search algorithms proceeded with greedy searches to find the optimal path, Q-learning operates in a manner similar to dynamic programming algorithms. (Of course, Q-learning is not a dynamic programming algorithm.) In the 'q_learning' function, the Q-table is filled as follows:



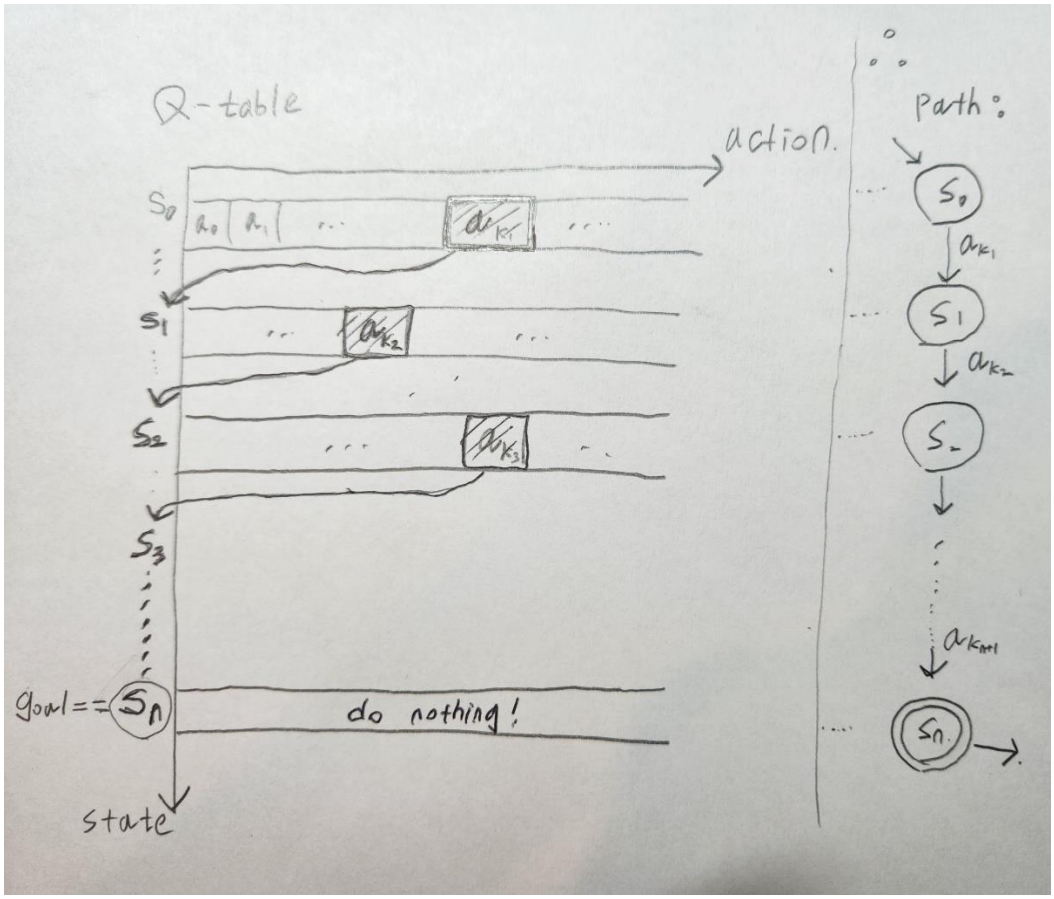
Next, the 'q_learning_path' function is outlined with simple pseudocode and implementation code

as follows:

```
func q_learning_path(maze, q_table):
        while current_state != goal:
                filter valid action (wall X, out of maze X)
                action <- argmax(q_table[current_state])
                calculate next_state
                append path

        return path
```

```python
1 usage
def q_learning_path(maze, q_table):  # find path using Q table
    time_start = time.time()  # for experimenting (time execution)

    path = []
    state = maze.entry_coor  # 시작 위치
    maze.solution_path = []

    with open('path.txt', 'w') as file:  # for experiment
        while state != maze.exit_coor:
            path.append(state)
            file.write(f"{state}\n ")
            maze.solution_path.append((state, False))

            s_index = S2I(state, maze.num_cols)
            valid_actions = get_valid_action1(state, maze)  # filtering 1
            valid_actions = get_valid_action2(state, valid_actions, maze)  # filtering 2
            q_values = [q_table[s_index, a] for a in valid_actions]
            action = valid_actions[np.argmax(q_values)]  # find maximum Q value among current state (meaning row)
            #action = np.argmax(q_table[s_index])  # find optimal action
            state, reward, flag = go(state, action, maze)  # go next state! (cf. "reward", "flag" is useless var)
            print(state, action, reward, flag, "\n")  # for debugging purpose

        path.append(state)
        file.write(f"{state}\n ")
        maze.solution_path.append((state, False))

        file.write("Finish finding path!\n")
        exec_time = time.time() - time_start  # total execution time
        file.write("Total execution time for finding path using Q-table {:.4f}".format(exec_time))

    return path
```

The process of finding a path using the Q-table can be illustrated as follows:



The functions implemented for auxiliary tasks or experiments are as follows:

```
1 usage
def print_q_table(q_table, maze):
    with open('q_table.txt', 'w') as file:
        for i in range(maze.num_rows-1, -1, -1):
            for j in range(maze.num_cols):
                # file output
                q_values = []
                state = (i, j)
                s_index = S2I(state, maze.num_cols)
                for k in range(4):
                    q_values.append(q_table[s_index, k])
                file.write(f"{q_values} ")
            file.write("\n")
```

```
1 usage
def get_valid_action1(state, maze):  # 현재 state에서 가능한 action들만 추출 --- 1
    row, col = state
    valid_action = [0, 1, 2, 3]

    if row == maze.num_rows - 1:
        valid_action.remove(0)  # 상 X
    if col == maze.num_cols - 1:
        valid_action.remove(3)  # 우 X
    if row == 0:
        valid_action.remove(1)  # 하 X
    if col == 0:
        valid_action.remove(2)  # 좌 X

    return valid_action

1 usage
def get_valid_action2(state, actions, maze):  # 현재 state에서 가능한 action들만 추출 --- 2
    row, col = state
    temp = []

    for a in actions:
        if a == 0:  # 상
            next_state = (row + 1, col)
            if maze.grid[state[0]][state[1]].is_walls_between(maze.grid[next_state[0]][next_state[1]]):
                temp.append(a)
        if a == 1:  # 하
            next_state = (row - 1, col)
            if maze.grid[state[0]][state[1]].is_walls_between(maze.grid[next_state[0]][next_state[1]]):
                temp.append(a)
        if a == 2:  # 좌
            next_state = (row, col - 1)
            if maze.grid[state[0]][state[1]].is_walls_between(maze.grid[next_state[0]][next_state[1]]):
                temp.append(a)
        if a == 3:  # 우
            next_state = (row, col + 1)
            if maze.grid[state[0]][state[1]].is_walls_between(maze.grid[next_state[0]][next_state[1]]):
                temp.append(a)

    return [element for element in actions if element not in temp]
```

- **get_valid_action1:** Actions that result in collision with walls are filtered out from the current state.
- **get_valid_action2:** Among these filtered actions, those that would take the agent out of the maze boundaries are further filtered out.

  ∴ Thus, the agent only considers valid actions.

- **print_q_table:** print q_table information (used for experimenting)

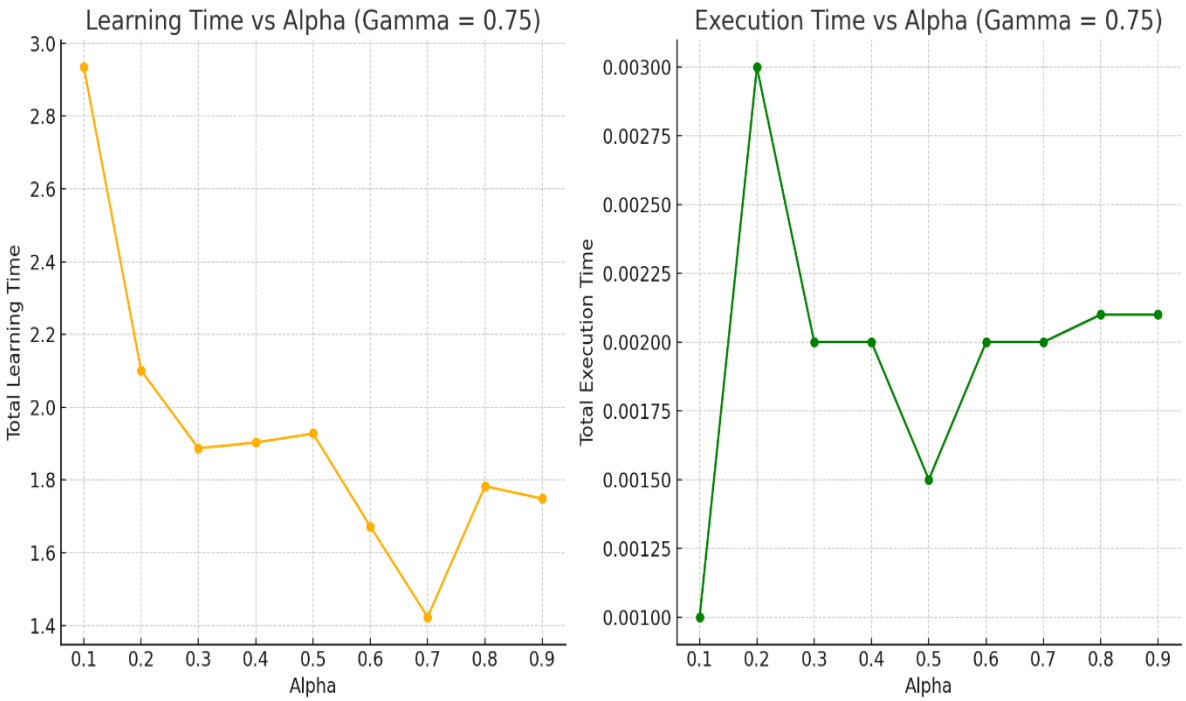
2. Experiment & Analysis:

Experiments were conducted on three mazes to investigate the effects of two parameters (learning rate, discount factor). The number of episodes was set to 1000, and the initial value of epsilon was set at 0.99, which was then multiplied by 0.99 at the end of each episode. (cf. $0.99^{1000} = 0.00004317124$)

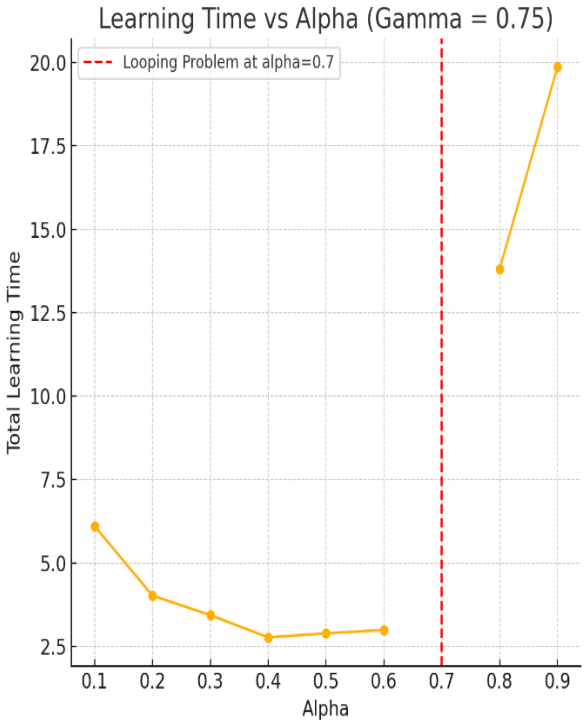
- About learning rate (α):

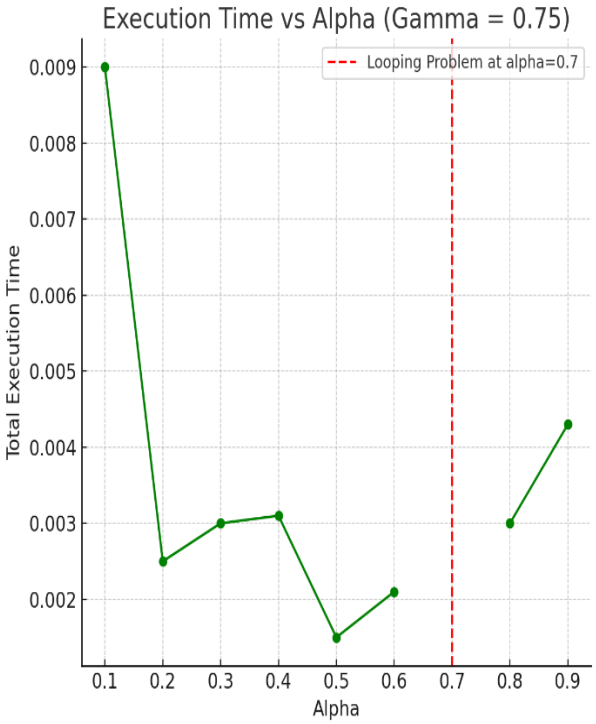
First, the results of the experiment on the learning rate are as follows. (In this case, the discount factor was fixed at 0.75.) The red line indicates that it encountered a looping problem. **Learning time** refers to the time taken to complete the Q-table, and **execution time** refers to the time taken to find the path using the completed Q-table.

- *random.seed(1):*

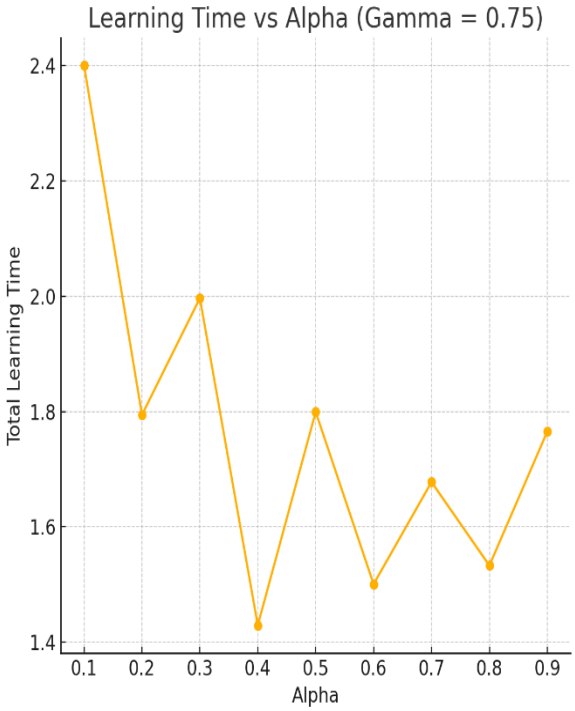


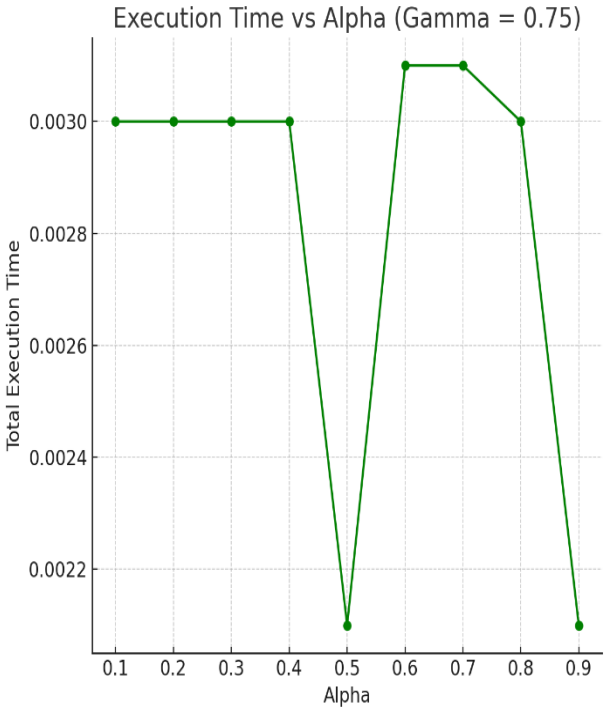
- *random.seed(5):*





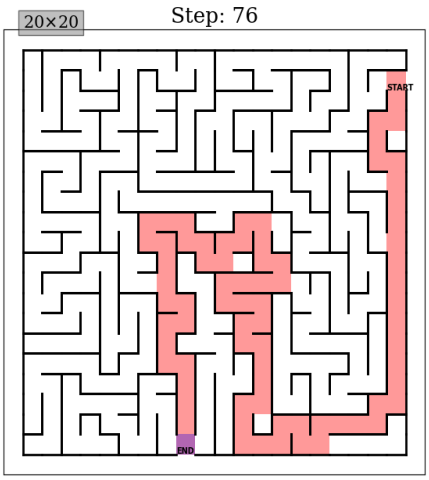
- *random.seed(6):*

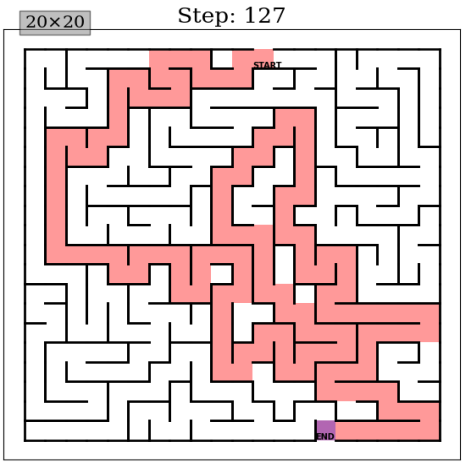


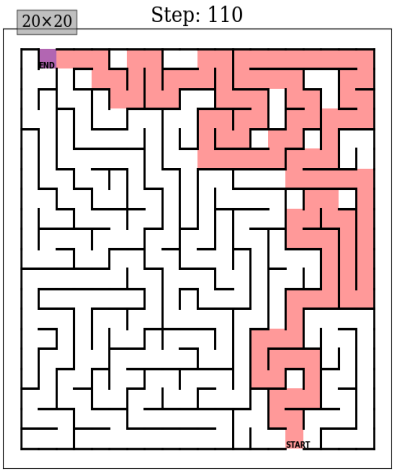


(As long as there is no looping problem when finding the path based on the completed Q-table, there are no significant time differences, so I will focus on analyzing the ***learning time***.)

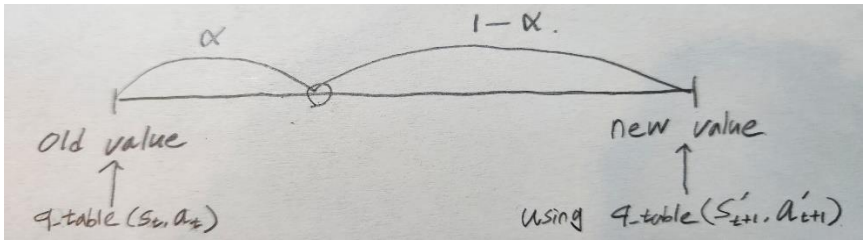
Also, all results that did not encounter a looping problem found the shortest path as follows. (each random seed: 1, 5, 6)







First, the learning rate, or α value, indicates how much new information is incorporated. The following is a diagram illustrating this.



Looking at the experiment results for the three mazes, the learning time generally decreases in the [0.1, 0.6] range and either slightly increases or sharply rises in the [0.7, 0.9] range. (Assuming the correct optimal solution has been found) A shorter learning time indicates that the algorithm has converged correctly and stably with less volatility, while a longer learning time suggests that the learning occurred more slowly with more oscillations. In the [0.1, 0.6] range, the decrease in learning time can be inferred as the increase in α value allowing for better integration of new learning, making the learning process more efficient. When the α value is very low, such as 0.1, it can be inferred that the algorithm is insensitive to changes due to new information, resulting in longer learning times.

Conversely, in the [0.7, 0.9] range, we can see a slight increase or sharp rise, which can be inferred as the α value being excessively high, thereby overwriting important previously learned information and causing more volatile oscillations during learning.

Additionally, in the second maze, a **looping problem** occurs when the α value is 0.7, suggesting that a high α value does not provide sufficient stability for learning, leading to a failure to converge. However, considering that most results do not experience looping problems due to α values, it appears that the looping issue occurred exceptionally due to the interaction between α and gamma values.

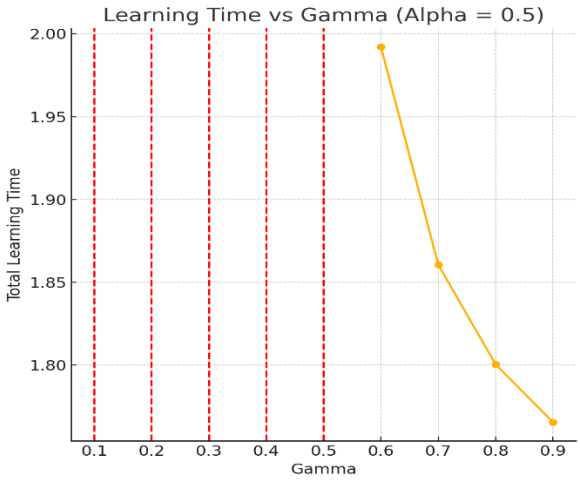
Let's now look at the results for execution time. A shorter execution time indicates that an optimized Q-table has been created for finding the optimal solution. However, overall, there are no significant differences in execution time.

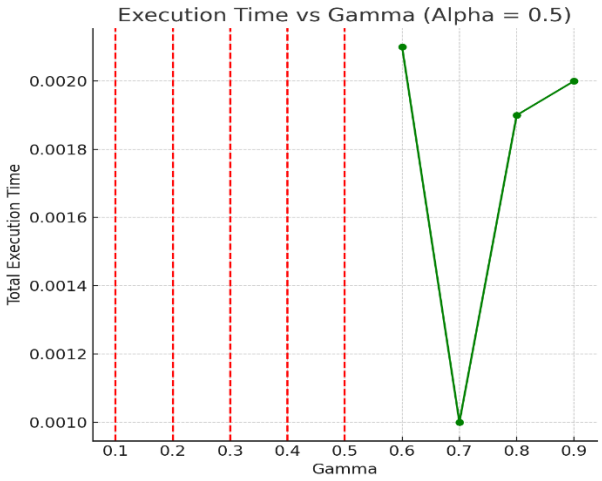
∴ *In summary,* it can be seen that an α value of 0.5 generally provides the most efficient learning and produces an optimized Q-table.

- About discount factor (γ):

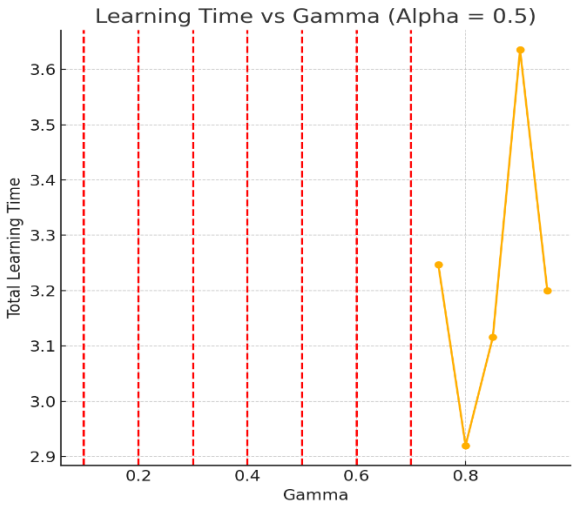
The results of the experiment on the gamma value are as follows. The learning rate was fixed at 0.5 for the experiment.

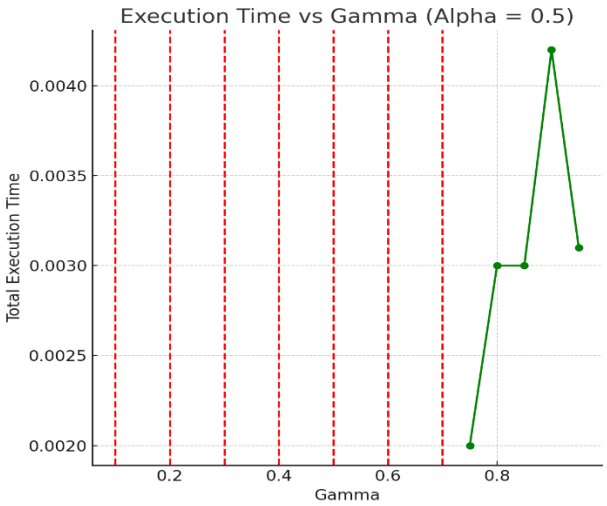
- random.seed(1):



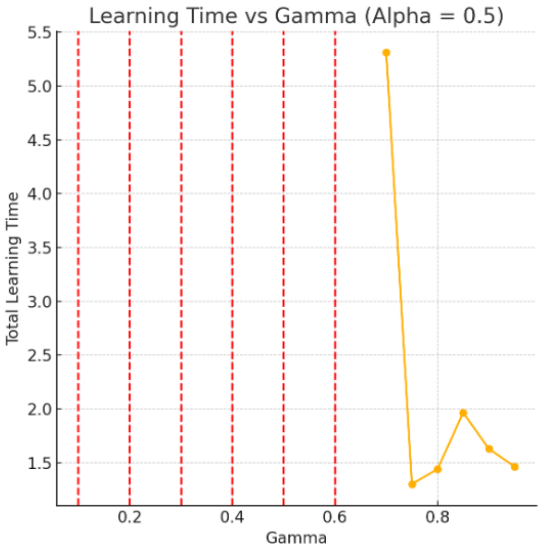


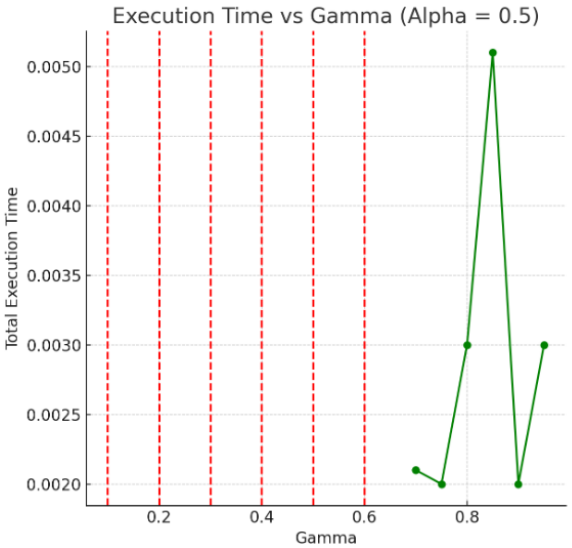
- random.seed(5):





- random.seed(6):





In cases where there were no looping problems, like the alpha experiments, it was successful in finding the optimal solution.

Looking at the results of the experiment with gamma values, unlike the experiments with alpha values, the pattern of increase or decrease in learning time with the increase in gamma values is not distinctly apparent. However, when the gamma value was set to about 0.7 or lower, all cases fell into looping problems, which can be reasoned as follows:

If the gamma value is low, the agent prioritizes immediate rewards, thereby not exploring the path widely in the long term. Consequently, the **reward structure might be built in a local and limited area**, rather than encompassing the entire maze (this can exhibit characteristics similar to a greedy algorithm).

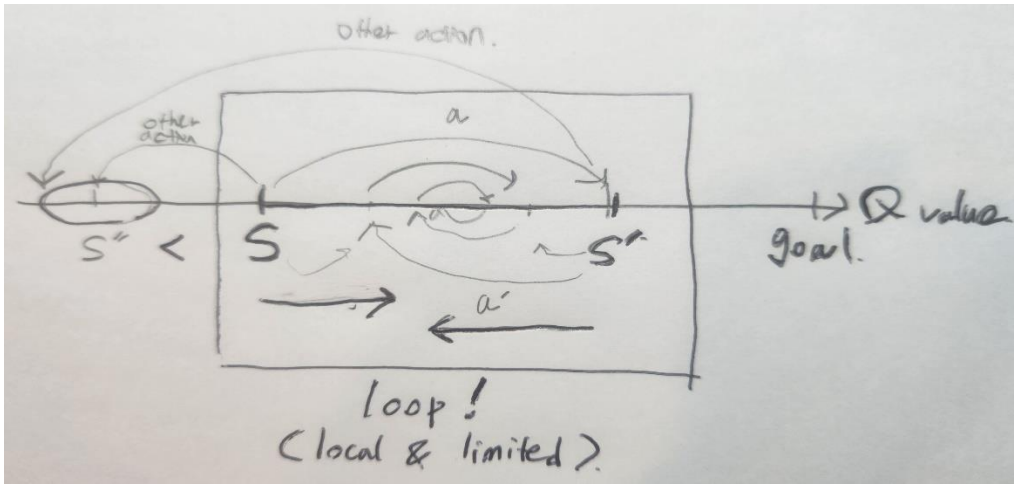
∴ *In summary,* considering both learning time and execution time, it seems that a gamma value of about 0.8 is most efficient for learning appropriately

3. Conclusion:

In summary, in cases where there were no looping problems, the fact that **optimal solutions** were obtained demonstrates the effectiveness of Q-learning. However, the experiment shows that it is sensitive to parameter values, especially when the gamma value is set low, which can be particularly detrimental. It appears that optimal learning occurs around an alpha value of 0.5 and a gamma value of 0.8. Furthermore, as episodes progress, it is possible to make learning more efficient by reducing the alpha value to become less sensitive to new information.

4. More about looping problem:

The diagram for the looping problem situation described earlier would look like this.



Additionally, although not shown in the experiment results, there are occasional observations where a looping problem occurs in the exploration process even after successfully completing the Q-table. This usually happens when the Q-values show very fine differences and the Q-table learning has ended. A diagram illustrating this would look as follows .