Maze Traversal Using Q Learning Algorithm

Student ID: 2021313702

Name: Hyunwoo Oh

1 Introduction

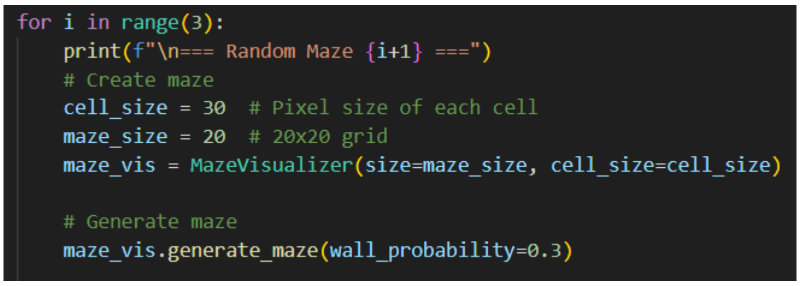
This project utilizes a Q-Learning agent that learns to navigate a randomly generated 20 × 20 maze using Q Table. At every step, the agent pays a unit cost and receives a reward: +100 for reaching the goal, −1 for hitting a wall or leaving the grid, and −0.1 for any legal move. During training, the agent follows an ε-greedy policy with ε that starts at 0.30 and decays by 0.995 each episode so that it explores widely at first and gradually exploits what it has learned. After training, a greedy policy that selects the action with the highest value in the Q-table is executed once to extract the final path, which is visualized with Pygame. By adjusting the parameters including learning rate (α) and discount factor (γ), user can observe how these hyper-parameters affect convergence speed, path length, and total cost. This report explains the code implementation and code flow, presents various results depending on the parameter given to the q_learning() function, analyzes the impact of each parameter, and summarizes the key findings.

2. Code Flow and Implementation

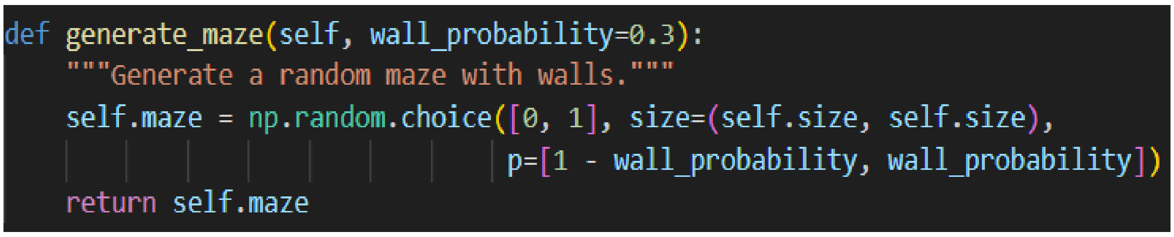
2.1 main() function

1> Generate three Random Maze
: Create a new maze, learn a policy, display the result.

2> Maze creation
: generate_maze() function fills a 20 × 20 NumPy array with (0 = free cell), (1 = wall) with default 30 % walls.

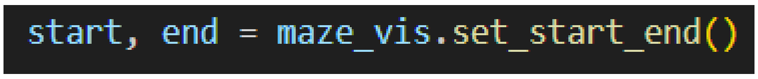


```
for i in range(3):
    print(f"\n=== Random Maze {i+1} ===")
    # Create maze
    cell_size = 30  # Pixel size of each cell
    maze_size = 20  # 20x20 grid
    maze_vis = MazeVisualizer(size=maze_size, cell_size=cell_size)

    # Generate maze
    maze_vis.generate_maze(wall_probability=0.3)
```



```
def generate_maze(self, wall_probability=0.3):
    """Generate a random maze with walls."""
    self.maze = np.random.choice([0, 1], size=(self.size, self.size),
                                  p=[1 - wall_probability, wall_probability])
    return self.maze
```

3> Start/goal placement
: set_start_end() function picks two free cells and guarantees that at least one path exists between them (checked by ensure_path_exists() function).



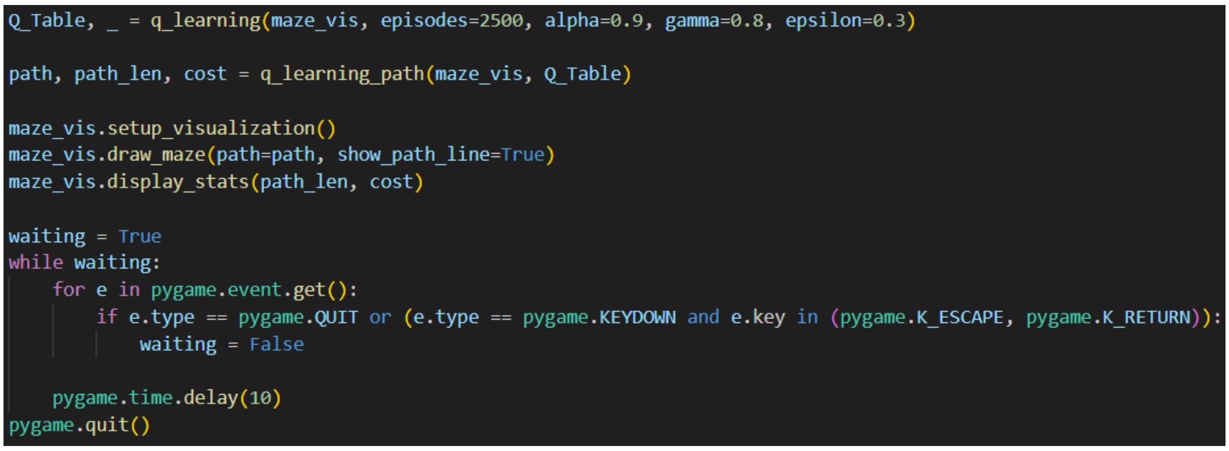
```
start, end = maze_vis.set_start_end()
```

4> Learning Process: The learning process begins by passing the maze object along with key hyperparameters, including the number of episodes, learning rate (α), discount factor (γ), and exploration rate (ε), into the q_learning() function. This function implements the Q-Learning algorithm, where the agent iteratively explores the maze environment, updates the Q-values based on its experiences, and gradually learns the optimal policy for reaching the goal. As a result, the q_learning() function returns a fully trained Q-table, which contains the learned action values for each state in the maze.

Once the Q-table is obtained, it is used in the path extraction step by calling the q_learning_path() function. This function takes the maze object and the trained Q-table as inputs and applies a greedy policy, where the agent selects the action with the highest Q-value at each state to follow the best-known path. The q_learning_path() function then computes the final path from the start to the goal, along with the corresponding path length and total cost. This process demonstrates how the agent transitions from an exploratory phase during training to an exploitation phase using the greedy strategy to navigate the maze effectively.

5> Visualization: The visualization of the maze includes three key functions: setup_visualization(), draw_maze(), and display_stats(). The setup_visualization() function initializes the visualization window using Pygame, setting up the grid, colors, and window size to ensure everything is ready for rendering. The draw_maze() function paints the maze grid on the screen, using different colors to represent various cell states such as walls, the start point, the goal point, and the visited path. It also draws the agent's path as lines or markers. Finally, the display_stats() function writes information like "path length" and "total cost" in a sidebar within the visualization window, providing a summary of the exploration results. Together, these functions enable an interactive and informative visualization of the maze and the agent's traversal.

6> User pause: The window stays open until the user enters 'ESC' or 'Enter', then the next maze begins.



```python
Q_Table, _ = q_learning(maze_vis, episodes=2500, alpha=0.9, gamma=0.8, epsilon=0.3)

path, path_len, cost = q_learning_path(maze_vis, Q_Table)

maze_vis.setup_visualization()
maze_vis.draw_maze(path=path, show_path_line=True)
maze_vis.display_stats(path_len, cost)

waiting = True
while waiting:
    for e in pygame.event.get():
        if e.type == pygame.QUIT or (e.type == pygame.KEYDOWN and e.key in (pygame.K_ESCAPE, pygame.K_RETURN)):
            waiting = False

    pygame.time.delay(10)
pygame.quit()
```

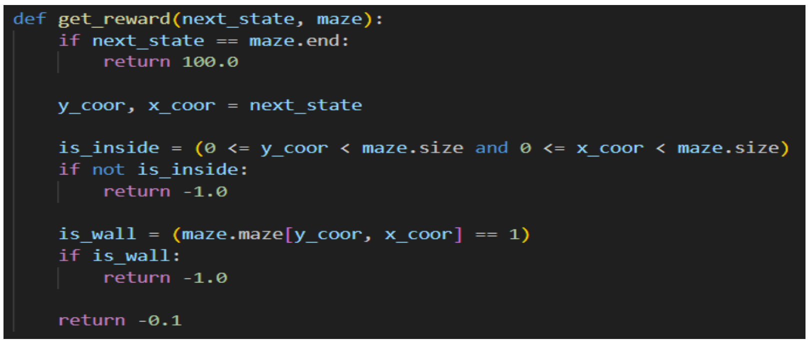
2.2 Environment Helper — MazeVisualizer

- generate_maze(): stochastic maze generator.
- set_start_end(): selects start and end cells, flips them to free space if necessary, and calls ensure_path_exists() function to verify reachability.
- draw_maze(): paints cells, grid lines, legends, an optional orange poly line through the final path, and a red square if the agent is shown live.
- display_stats(): prints path length and total cost in the sidebar.

The class isolates all graphics and maze logistics from learning logic.

2.3 Reward Function — get_reward()

Returns a reward based on the candidate next cell +100 if it is the goal, -1 if it is outside the grid or a wall, -0.1 otherwise.



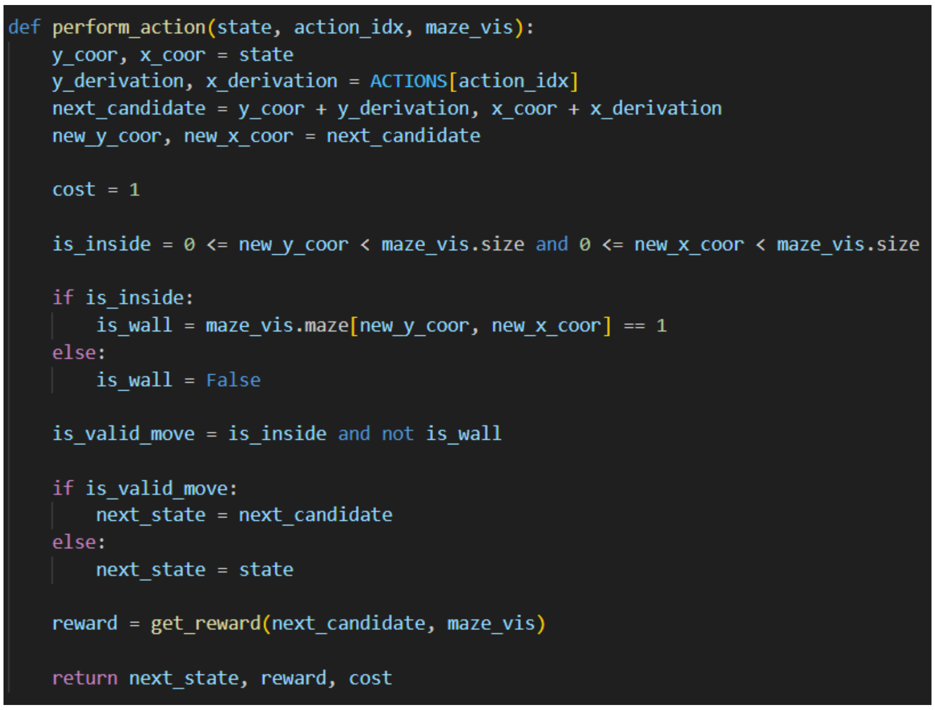
```python
def get_reward(next_state, maze):
    if next_state == maze.end:
        return 100.0

    y_coor, x_coor = next_state

    is_inside = (0 <= y_coor < maze.size and 0 <= x_coor < maze.size)
    if not is_inside:
        return -1.0

    is_wall = (maze.maze[y_coor, x_coor] == 1)
    if is_wall:
        return -1.0

    return -0.1
```

2.4 One Step Transition — perform_action()

Inputs: current state, a discrete action index.

1. Converts the action index to a (y_derivation, x_derivation) move.
2. Determines whether the resulting cell is inside the grid and not a wall.
3. If valid, the agent moves and otherwise it remains in place.
4. Calls get_reward() function to obtain the immediate reward.
5. Always returns cost = 1, satisfying the requirement that a wall collision still ”consumes one turn.”

Outputs: next_state, reward, cost.



```python
def perform_action(state, action_idx, maze_vis):
    y_coor, x_coor = state
    y_derivation, x_derivation = ACTIONS[action_idx]
    next_candidate = y_coor + y_derivation, x_coor + x_derivation
    new_y_coor, new_x_coor = next_candidate

    cost = 1

    is_inside = 0 <= new_y_coor < maze_vis.size and 0 <= new_x_coor < maze_vis.size

    if is_inside:
        is_wall = maze_vis.maze[new_y_coor, new_x_coor] == 1
    else:
        is_wall = False

    is_valid_move = is_inside and not is_wall

    if is_valid_move:
        next_state = next_candidate
    else:
        next_state = state

    reward = get_reward(next_candidate, maze_vis)

    return next_state, reward, cost
```

2.5 Learning Loop — q_learning()

Inputs: maze object, number of episode, learning-rate α, discount-factor γ, initial exploration rate ε.

For every episode (maximum square of (maze_vis.size) steps)

1. Action selection
   – With probability ε choose a random action.
   – Otherwise, choose the action with the highest current value in the Q-table (greedy action).

2. Environment transition
   – perform_action() function executes the move and returns the next state, the shaped reward, and a fixed step cost of 1.

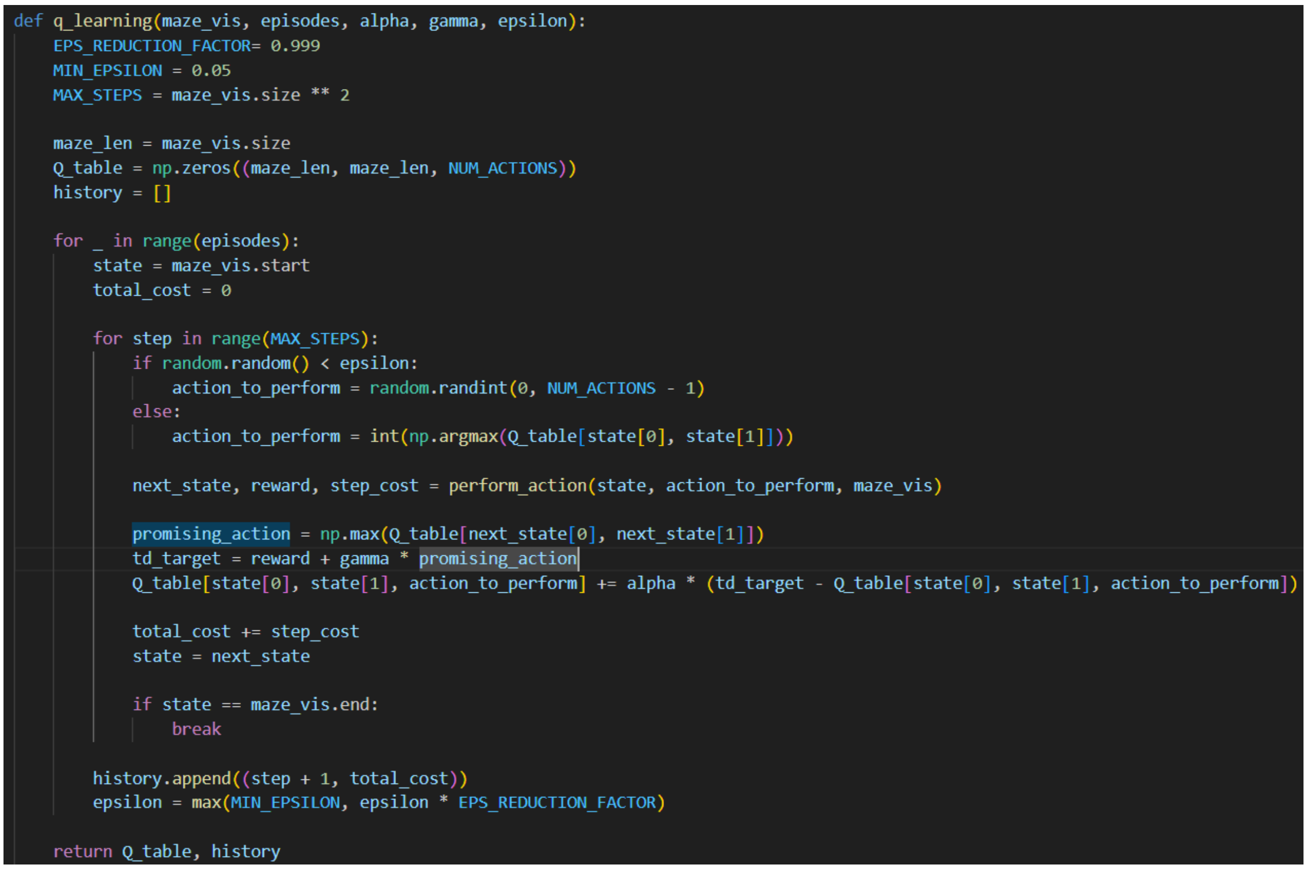
3. TD update

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

4. Episode termination
   – If the agent reaches the goal cell, the episode ends immediately.
   – Otherwise it continues until the maximum steps.

5. Exploration decay
   After each episode, ε ← max(0.05, 0.995ε) to encourage exploitation once learning stabilizes.

The routine accumulates a (length, cost) tuple for every episode and finally returns the fully populated Q-table along with this training history.



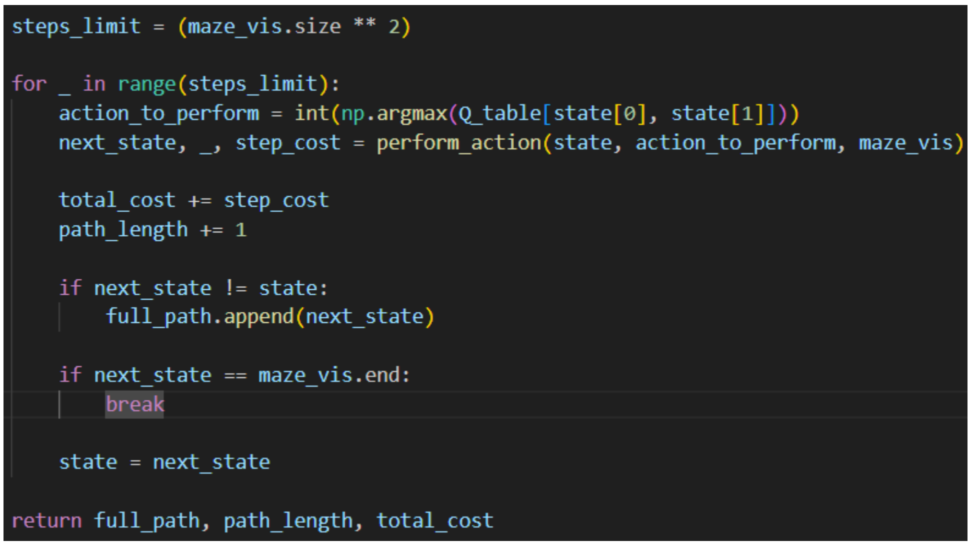
```python
def q_learning(maze_vis, episodes, alpha, gamma, epsilon):
    EPS_REDUCTION_FACTOR= 0.999
    MIN_EPSILON = 0.05
    MAX_STEPS = maze_vis.size ** 2

    maze_len = maze_vis.size
    Q_table = np.zeros((maze_len, maze_len, NUM_ACTIONS))
    history = []

    for _ in range(episodes):
        state = maze_vis.start
        total_cost = 0

        for step in range(MAX_STEPS):
            if random.random() < epsilon:
                action_to_perform = random.randint(0, NUM_ACTIONS - 1)
            else:
                action_to_perform = int(np.argmax(Q_table[state[0], state[1]]))

            next_state, reward, step_cost = perform_action(state, action_to_perform, maze_vis)

            promising_action = np.max(Q_table[next_state[0], next_state[1]])
            td_target = reward + gamma * promising_action
            Q_table[state[0], state[1], action_to_perform] += alpha * (td_target - Q_table[state[0], state[1], action_to_perform])

            total_cost += step_cost
            state = next_state

            if state == maze_vis.end:
                break

        history.append((step + 1, total_cost))
        epsilon = max(MIN_EPSILON, epsilon * EPS_REDUCTION_FACTOR)

    return Q_table, history
```

2.6 Greedy Exploitation — q_learning_path()

Takes the trained Q-table and rebuilds a single path from start to goal:

1. From the current state choose the greedy action.
2. Execute that action through perform_action() function.
3. Increment total cost every time and increment path length even if it encounters the wall.
4. Repeat until the agent reaches the goal or a hard cap equal to the total number of cells (maze_vis.size²) is exceeded.

Outputs: the ordered list of visited cells, the path length, and the total cost.



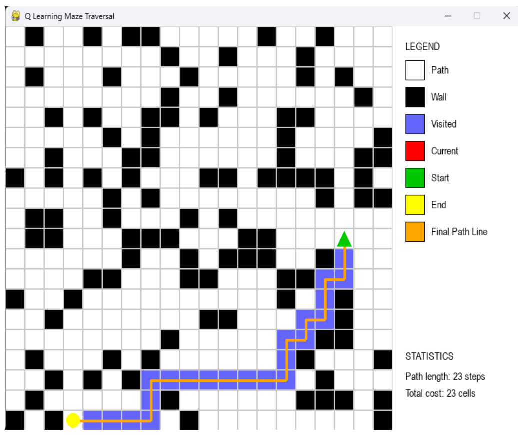
```
steps_limit = (maze_vis.size ** 2)

for _ in range(steps_limit):
    action_to_perform = int(np.argmax(Q_table[state[0], state[1]]))
    next_state, _, step_cost = perform_action(state, action_to_perform, maze_vis)

    total_cost += step_cost
    path_length += 1

    if next_state != state:
        full_path.append(next_state)

    if next_state == maze_vis.end:
        break

    state = next_state

return full_path, path_length, total_cost
```

This structure cleanly separates maze logic, learning, evaluation, and visualization, making it straightforward to swap out reward schemes, exploration schedules, or even the core RL algorithm without altering the other components.

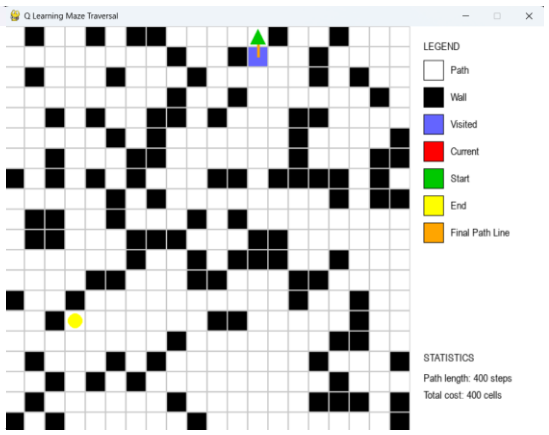
3. Influence of Parameters

# Defualt Setting: alpha=0.9, gamma=0.2, epsilon=0.3, EPS_REDUCTION_FACTOR= 0.999, MIN_EPSILON = 0.05, MAX_STEPS = maze_vis.size ** 2

1> Number of Episodes (# episode: 2500 -> 1000)



STATISTICS
Path length: 23 steps
Total cost: 23 cells



STATISTICS
Path length: 400 steps
Total cost: 400 cells

# Observation

1. State action coverage.
   2 500 episodes give the agent enough exploratory steps to encounter the goal early and propagate the +100 reward through the Q-table. With 1 000 episodes the agent often exhausts the per-episode limit before ever receiving a positive reward, leaving most Q-values near zero.

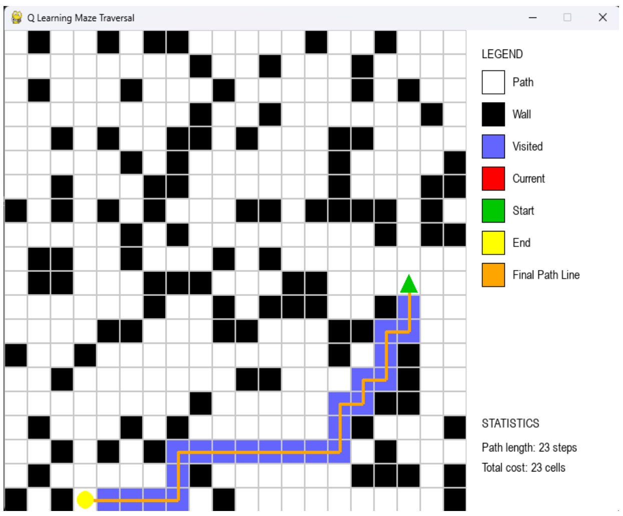
2. ε-decay imbalance.
   Exploration rate ε decays every episode, not per step. Reducing the episode count by 60 % cuts total updates by a similar factor but leaves the ε schedule unchanged, so the agent enters exploitation mode before it has gathered adequate experience.

3. Policy quality.
   In Run A the greedy policy has converged to an almost minimal path (23 legal moves, no collisions).
   In Run B the policy degenerates into a loop. The agent repeatedly selects the same uninformative action, hits the steps_limit = 400.

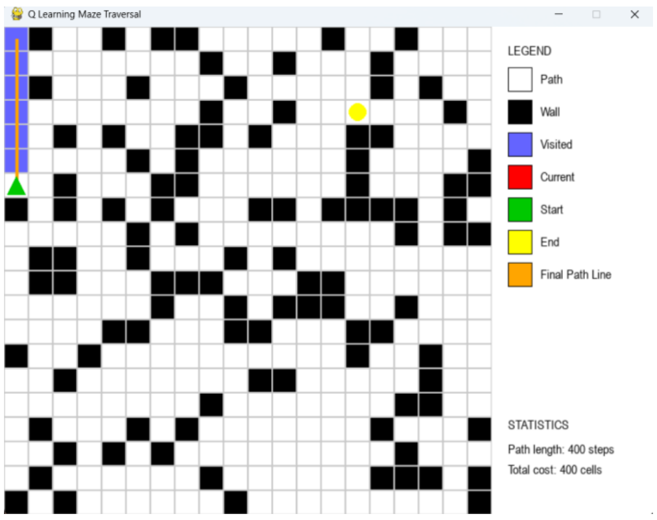
4. Learning-rate interaction.
   Even with a high learning-rate (α = 0.9) the algorithm still needs many bootstrapping iterations, otherwise the TD target cannot propagate far enough through the state space.

For this maze size and wall density, at least ~2 000 episodes are needed to guarantee that most training runs experience the +100 reward early and converge to a usable policy. Episode budgets below that threshold lead to highly variable outcomes and frequent failure to reach the goal.

2> learning rate (α : 0.9 -> 0.2)



Path length: 23 steps
Total cost: 23 cells



Path length: 400 steps
Total cost: 400 cells

# Observation

1. Slower value propagation.
   With α = 0.2 each TD update shifts the Q-value only 20 % toward the target, so the +100 reward encountered travels very slowly back through the state space.

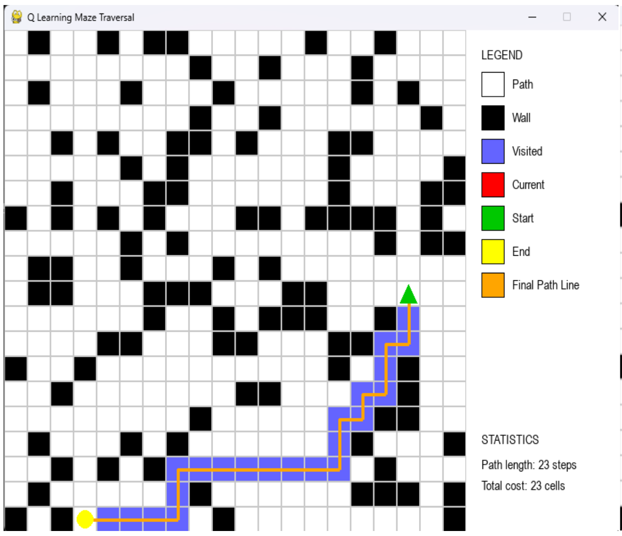
2. Coupling with ε-decay.
   Exploration rate ε decays per episode, independent of α. With small α the table is still almost flat when ε has already fallen close to its 0.05 floor, leaving the agent to exploit essentially uninformative Q-values. Hence,

the repetitive vertical movement happens.

3. Sensitivity to early reward.
   A low α requires many successive updates before a single positive reward can influence upstream states, missing that early experience results in long episodes where every Q entry remains ≈ 0.

For this maze size and reward structure, learning-rates in the 0.6 – 0.9 range are needed to propagate high rewards quickly enough before exploration dies out. Setting α too low dramatically increases convergence time and can prevent learning altogether within a fixed episode budget**.**

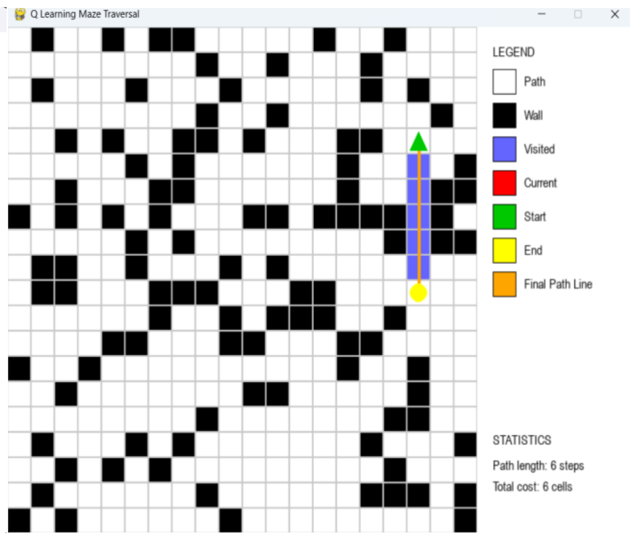
3> Discount Factor (γ: 0.2 -> 0.8)



Q Learning Maze Traversal

LEGEND
⬜ Path
⬛ Wall
🟦 Visited
🟥 Current
🟩 Start
🟨 End
🟧 Final Path Line

STATISTICS
Path length: 23 steps
Total cost: 23 cells



Q Learning Maze Traversal

LEGEND
⬜ Path
⬛ Wall
🟦 Visited
🟥 Current
🟩 Start
🟨 End
🟧 Final Path Line

STATISTICS
Path length: 6 steps
Total cost: 6 cells

# Observation

1. Reward Propagation
   With γ = 0.8 (Run B), the +100 terminal reward is discounted much less and therefore propagates rapidly through the Q-table, creating a strong gradient that pulls the agent toward the goal.
   With γ = 0.2 (Run A) the reward almost vanishes after a few steps. Most distant states still appear worthless, so the agent is guided mainly by the small step penalties.

2. Exploration Pattern
   Because the ε-schedule and the episode budget are identical, both runs visit a comparable number of state–action pairs. The higher γ run converts those visits into usable value information, whereas the lower γ run keeps revisiting uninformative states because the propagated value signal is too weak.

3. Policy Quality
   Run A (γ = 0.2): converges to a longer, zig-zag path with 23 legal moves / cost = 23.
   Run B (γ = 0.8): converges to a nearly optimal straight line with 6 legal moves / cost = 6.
   Higher γ thus yields a policy that is both shorter and cheaper in terms of accumulated penalties.

4. Learning-Rate Interaction
   Even with a high learning rate (α = 0.9), a low γ limits how far the TD target can spread, so many bootstrap updates are effectively wasted.

When γ is raised, those same updates transmit useful information deep into the state space, accelerating convergence without increasing the episode count.

Merely increasing γ from 0.2 to 0.8 dramatically improves path optimality and stability by allowing long-range value propagation, while leaving exploration dynamics unchanged.

4. Conclusion

This report demonstrates how a tabular Q Learning agent can learn to navigate a randomly generated 20 × 20 maze, provided the right combination of parameters is used. The code structure and implementation cleanly separate maze generation, visualization, reward calculation, and the Q-Learning algorithm itself, making it flexible and modular. The core learning process relies on an ε-greedy strategy with decaying exploration, a reward system that incentivizes reaching the goal, and a fixed per-step cost to simulate real-world decision-making trade-offs.

Through detailed analysis, it is clear that the number of episodes, learning rate (α), and discount factor (γ) each play a critical role in shaping learning outcomes:

- A sufficiently large episode budget (at least 2 000–2 500) is essential to ensure the agent explores the environment enough to encounter the +100 terminal reward and propagate it through the state space.
- The learning rate (α) needs to be high (0.8–0.9) to enable rapid value propagation within the limited exploration window; a low α results in slow convergence and near-zero Q-values.
- The discount factor (γ) determines how far the reward signal can travel: increasing γ from 0.2 to 0.8 shifts the policy from a long, inefficient zig-zag path to a nearly optimal, straight-line solution.

This report highlights the importance of balancing hyper-parameters in Q-Learning, as well as the need for sufficient exploration and value propagation to achieve convergence in complex, stochastic environments like a randomly generated maze.