Computers and systems engineering

2025

Artificial intelligence course

Dr. Sara Khalil

Maze-Solver-using-Reinforcement-Learning-Q-Learning

الاسم	رقم الجلوس
محمد محمود محمد غانم	6358
كريم وائل طلبه محمد	6349
مصطفى حسن جاب الله عطية	6365
أنطون أنسي عبد الملك	6315

Table of Contents

1. Introduction

This project demonstrates the application of Q-learning, a popular reinforcement learning algorithm, to solve a maze. The agent (an artificial entity) is trained to find the shortest path from a starting point to a goal within a maze. The agent learns through trial and error, receiving rewards or penalties based on its actions, and eventually converges to an optimal path.

Problem Defined

Pathfinding in unknown environments is a classic problem in AI. Many traditional algorithms like ***DFS***, BFS, or A* require full knowledge of the environment. In contrast, **Q-learning** allows an agent to learn optimal strategies through trial-and-error interaction. The challenge lies in training the agent to avoid walls and reach the goal with the least number of moves, using only rewards and penalties.

Objective

Implement a Custom Maze Environment Design a grid-based maze where the agent can interact, move, and learn.

Apply Q-Learning Algorithm Use the Q-learning reinforcement learning algorithm to train the agent to find the shortest path from the start point to the goal.

Develop an Interactive Visualization Create a real-time visual representation of the agent's movement using Pygmy to illustrate the learning result.

Demonstrate Reinforcement Learning Concepts Show how an agent can learn from rewards and penalties, balance exploration and exploitation, and improve its policy over time.

2. AI Algorithms and Tools Used

2.1 Q-learning Algorithm

Q-learning is a model-free reinforcement learning algorithm used to find the optimal action-selection policy for a given environment. It works by learning a **Q-table** that estimates the expected utility (**Q-values**) of taking an action in a particular state, followed by the best future actions.

Key Concepts:

- **State**: The position of the agent in the maze grid.
- **Action**: The movement direction (up, down, left, right).
- **Reward**: The feedback received from the environment:
  - Reaching the goal: High positive reward (e.g., +100)
  - Hitting a wall: Negative reward (e.g., -10)
  - Moving in an empty cell: Small penalty (e.g., -1)

Formula:

Q(state, action) = Q(state, action) + α (reward + γ max(Q(next_state, all actions)) - Q(state, action))

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\left[r + \gamma \max_{a'} Q(s',a')\right]$$

Where:

- α: Learning rate
- γ: Discount factor
- ε: Epsilon (used in **ε-greedy policy** for exploration)

2.2 Tools and Libraries

- **Python**: Programming language used to implement the agent and environment.
- **OpenAI Gym-like Custom Environment**: A custom environment resembling OpenAI Gym structure.
- **Pygame**: Used to create a GUI for visualizing the agent's path in the maze.
- **NumPy**: For numerical operations and Q-table storage.

3. Project Structure

- main.py >> Main execution file
- q_learning.py >> Contains the Q-learning algorithm
- environment.py >> Custom environment with maze logic
- visualization.py >> GUI for maze display and path visualization
- constants.py >> Configuration values and parameters

main.py

This script (main.py) is the **entry point** of a Q-learning Maze Solver project. It allows the user to choose between two maze setup modes:

1. **Randomly generated maze**
2. **Manual wall drawing using GUI**
3. **Exit to exit from program**
4. **Show or hide Training**

Then it trains an agent using Q-learning to solve the maze and finally visualizes the solution.

1. Import Required Modules

```
import pygame
from constants import *
from environment import MazeEnv
from q_learning import train_q_learning
from visualization import run_visualization, setup_environment
```

- pygame: GUI library used for the interface.
- constants: Assumed to store values like screen dimensions, grid sizes, colors (YELLOW, etc.).
- MazeEnv: Custom OpenAI Gym-like environment for the maze.
- train_q_learning: Trains the agent using the Q-learning algorithm.
- run_visualization, setup_environment: Show the training and environment.

2. Global Option Toggle

```
show_training_visualization = True
```

A global flag to show or hide training animation.

3. **draw_button() Function**

```
def draw_button(screen, text, rect, color_normal, color_hover, mouse_pos):
```

  - Draws a clickable button on the screen with hover effect.
  - rect.collidepoint(mouse_pos) checks if the mouse is over the button.
  - It then renders the button with different colors for normal/hover states.

4. **main_menu() Function**

```
def main_menu():
```

Displays a **Pygame window** with:

  - The current maze size.
  - Three buttons: "Random Maze", "Manual Maze", and "Exit".
  - A checkbox to toggle **training visualization**.

- **Detailed flow:**

  a. Initialize Pygame

```
pygame.init()
screen = pygame.display.set_mode((800, 600))
```

  b. Render UI and Buttons

```
draw_button(...)  # Called for each button
```

  c. Draw Checkbox

  Draws a **checkbox** and a label to toggle training visualization.

  If clicked, it toggles the show_training_visualization flag.

  d. Handle Mouse Events

```
if event.type == pygame.MOUSEBUTTONDOWN:
```

If a button is clicked, return 'random', 'manual', or None.

5. main() Function

```
def main():
```

This function controls the program flow.

a. Show Main Menu

```
choice = main_menu()
```

Waits for user input and returns the choice.

b. Based on Choice

```
if choice == 'random':
    env = MazeEnv(mode='random')
    setup_environment(env)
```

- For **random**, generate and show a random maze.
- For **manual**, let the user draw walls using mouse before pressing ENTER.
- If None, user clicked "Exit".

c. Train Q-learning Agent

```
q_table = train_q_learning(env, episodes=200, epsilon_decay=0.99, min_epsilon=0.01, with_visualization=True)
```

- Trains the agent for 200 episodes with epsilon decay.
- Shows animation **if checkbox was checked**.

d. Run Final Visualization

```
run_visualization(env, q_table)
```

Displays the final agent behavior in the trained maze

6. Entry Point

Runs main() only if the script is executed directly

```
if __name__ == "__main__":
    main()
```

q_learning.py

- Implements the core training logic.
- Initializes a Q-table of size (number of states × number of actions).
- Loops through episodes to train the agent using the epsilon-greedy strategy.
- After training, extracts the optimal path using extract path () by following the maximum Q-values from the table.

Initialize Pygame

```
pygame.init()
training_screen = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
pygame.display.set_caption("Q-Learning Training Visualization")
training_clock = pygame.time.Clock()
font = pygame.font.SysFont(None, 24)
```

- Initializes the Pygame window.
- Sets up the window title, dimensions, and font for rendering text.
- A clock (training_clock) is used to control the frame rate during training.

Function: draw_training_state

```
def draw_training_state(env, screen, font, episode, epsilon, total_reward):
```

- **Purpose**: Visually updates the screen to show the maze, agent, goal, and current stats during training.
- It does the following:
  - Clears the screen.
  - Iterates over each cell of the maze.

- If the cell is a wall (env.maze[y][x] == 1), it draws a **trap image**.
- Otherwise, it draws a **white cell.**
- If the cell was visited, it highlights it in **yellow**.

- Draws grid lines over the maze.
- Draws the goal as a **cheese image**.
- Draws the agent (rat) as an image.
- Refreshes the Pygame window (pygame.display.flip()).

Function: `train_q_learning`

```
def train_q_learning(env, episodes=1000, alpha=0.1, gamma=0.95,
                     epsilon=0.2, epsilon_decay=0.995, min_epsilon=0.01):
```

- **Q-learning** algorithm to train the agent in the maze environment.
- Parameters:
  - alpha: Learning rate (how much to update Q-values).
  - gamma: Discount factor (importance of future rewards).
  - epsilon: Initial exploration rate (probability of taking random actions).
  - epsilon_decay: Rate at which epsilon decreases after each episode.
  - min_epsilon: Minimum allowed value for epsilon.

Step-by-step:

1. **Initialize Q-table**: A table of shape [num_states x num_actions], initialized to zeros.
2. Loop through each episode:
   - Reset the environment.

- Until the episode is done:
    - If a quit event is detected, quit Pygame.
    - Choose an action (**explore** or **exploit** using **epsilon-greedy policy**).
    - Execute the action and receive the next state and reward.
    - Update the Q-table using the Q-learning formula
    - Update the current state and reward.
    - Render the updated training state.
- After each episode, decay epsilon.
- Every 10 episodes, print training statistics.

Function: extract_path

```
def extract_path(q_table, env, max_steps=1000):
```

- **Purpose**: Extract the path followed by the agent using the trained Q-table.
- Resets the environment and simulates steps by always choosing the best action (argmax(q_table[state])).
- Stops when it reaches the goal or max_steps is exceeded.
- Returns the path as a list of coordinates.

environment.py

- Defines the Maze Environment class.
- Represents the maze grid, agent location, goal, available actions, and rules for state transitions.

Class: MazeEnv(gym.Env)

This is a custom **reinforcement learning environment**, built by extending **gym.Env**.

__init__() – Constructor

```
def __init__(self, mode='random'):
```

- ➢ mode='random': by default, it generates **a random maze**.
- ➢ If mode=**'manual'**, it creates an **empty grid** and lets the user draw walls.

  Key properties:

  - self.observation_space: one state per cell → GRID_WIDTH * GRID_HEIGHT
  - self.action_space: 4 discrete actions: up, down, left, right
  - self.agent: starts at top-left corner (0, 0)
  - self.goal: bottom-right corner
  - self.maze: 2D grid of 0s (free) and 1s (walls)

Maze Initialization

1. Manual Maze (Empty Grid)

```
def _initialize_empty_maze(self):
    return np.zeros((GRID_HEIGHT, GRID_WIDTH), dtype=int)
```

2.Random Maze

```
def _generate_maze(self):
```

- Uses a simplified **Prim's algorithm** to generate a solvable maze:
  - Start at (0,0) and mark it as free.
  - Track surrounding wall cells and randomly convert walls into paths if they only connect to one other path (to prevent cycles).

- Ensure the maze is **connected** using _is_connected().

Check for Maze Connectivity

```
def _is_connected(self, maze):
```

- Performs **Breadth-First Search (BFS)** from start to goal.
- Returns True if there's a valid path, otherwise the maze is **regenerated.**

toggle_wall()

```
def toggle_wall(self, x, y):
```

- Allows user to manually toggle walls in the grid (used in GUI drawing).
- Prevents toggling start and goal.

reset()

```
def reset(self, seed=None, options=None):
```

- Resets the environment:
  - Places the agent at the start.
  - Clear the visited list.
  - Returns the initial state (cell index).

step()

```
def step(self, action):
```

- Given an action (0=up, 1=down, 2=left, 3=right), it:
  - Moves the agent if the new position is free (0).

- Updates visited cells.
- Calculates reward:
    - Reaching goal: +10, done = True
    - Valid move: -0.1
    - Invalid (hit wall): -0.5

➢ Returns:

- new_state: cell index
- reward
- terminated: goal reached
- truncated: always False here
- Empty info {}
-

visualization.py

- Uses pygmy to render the maze, agent, goal, and optimal path.
- Animates the agent's movement along the learned path step-by-step.
- Highlights visited cells, the goal cell, and the path in different colors.

1. draw_maze(screen, env, path=None)

is function **draws the current state of the maze** on the screen.

Clear the screen:

```
screen.fill(BLACK)
```

Loop over each cell in the maze:

```python
for y in range(GRID_HEIGHT):
    for x in range(GRID_WIDTH):
```

- maze[y][x] == 1: it's a **wall (trap)** → draw trap.png
- Else → draw **white** cell for empty space

Highlight visited cells:

If (y, x) is in env.visited, draw a red border using:

```python
pygame.draw.rect(screen, RED, rect, 1)
```

Draw grid lines (optional for clarity):

```python
pygame.draw.line(...)
```

- **Draw the goal (cheese)**:
    - Use cheese.png scaled to cell size, placed at the goal coordinates.
- **Draw the learned path** (if available):
    - Loop through the path and fill each cell green.
- **Draw the agent (rat)**:
    - Use rat.png and draw it at env.agent.

2. setup_environment(env)

This is a **GUI for manually setting walls** before training.

Create Pygame window with the maze size:

```python
screen = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
```

1. **Handle mouse events**:
    - On **left-click (button 1)** → add wall (maze[y][x] = 1)

- On **right-click (button 3)** → remove wall (maze[y][x] = 0)

2. **Handle keyboard events**:
    - If user presses Enter → exit manual setup
3. **Redraw the maze** continuously during editing.

3. `run_visualization(env, q_table)`

This **animates the agent solving the maze** using the Q-table.

- **Create a Pygame window** for visualization.
- **Extract the optimal path** from the Q-table:

```
path = extract_path(q_table, env)
```

Start from the initial state:

```
env.reset()
```

Step through the path one cell at a time:

- On each frame, update the agent's position.
- Draw the maze including the path so far.

Update screen at fixed speed:

```
clock.tick(10)  # 10 frames per second
```

Exit if user closes the window.

Function	Purpose
draw_maze	Visualizes the maze grid with walls, path, and agent
setup_environment	Allows user to design a custom maze before training
run_visualization	Animates the trained agent solving the maze

constants.py

Stores configuration values like grid size, cell size, window dimensions, and color definitions

1. Grid and Window Settings

```
GRID_WIDTH, GRID_HEIGHT = 6, 6
CELL_SIZE = 70
WINDOW_WIDTH = GRID_WIDTH * CELL_SIZE
WINDOW_HEIGHT = GRID_HEIGHT * CELL_SIZE
```

What This Does:

- Defines a **6x6 maze** grid.
- Each cell is **70 pixels × 70 pixels**.
- Total screen/window size is 6 * 70 = 420 pixels both in width and height.
- So the Pygame window will be **420×420 pixels**.

2. Visualization Frame Rate

```
Frame_RATE = 200  # FPS
```

What This Does:

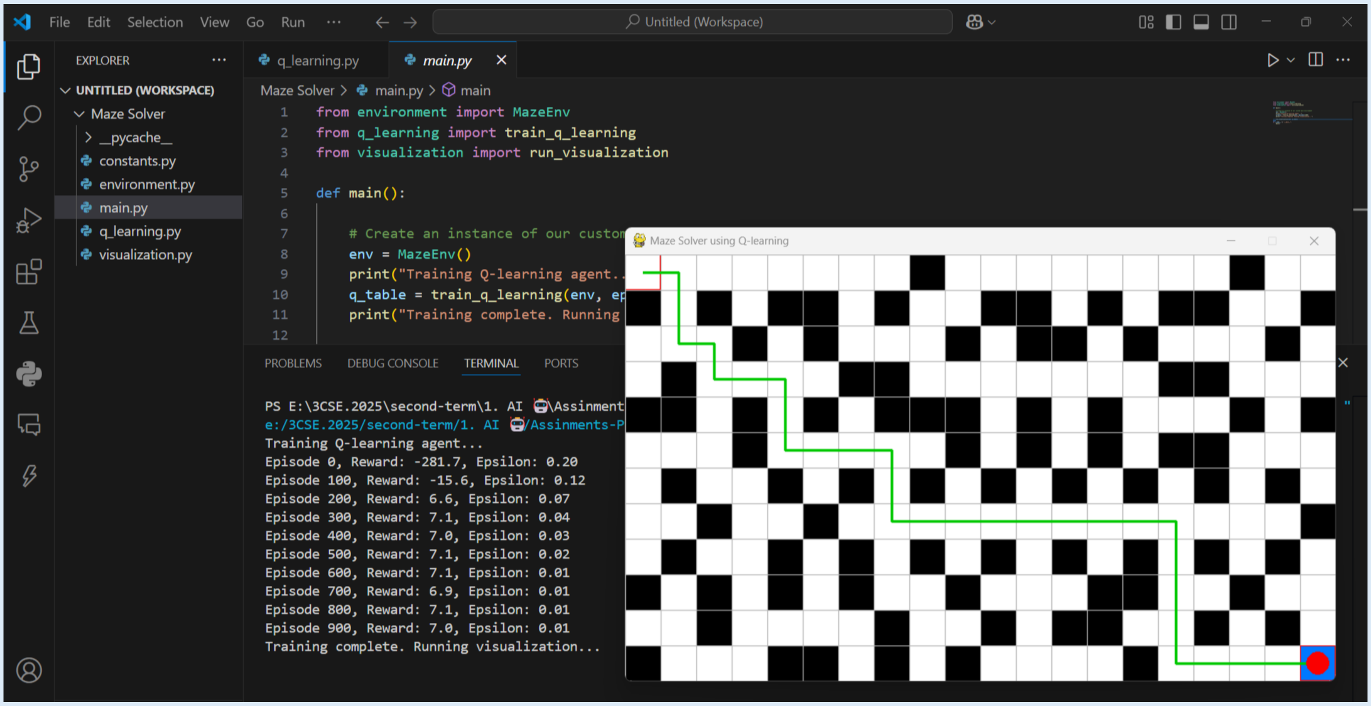
- Controls how fast the Pygame window updates.
- 200 frames per second means **very fast updates**.
- If you're doing visual training, consider lowering it (e.g., 30) to better observe the agent.

3. Color Definitions (RGB Format)

Variable	RGB Value	Meaning
WHITE	(255, 255, 255)	Empty path cells
BLACK	(0, 0, 0)	Background / Wall
RED	(250, 0, 0)	Agent (rat) border or highlight
YELLOW	(255, 255, 200)	Light yellow, for visited cells (unused in code above)
GRAY	(180, 180, 180)	Grid lines between cells
BLUE	(0, 120, 255)	Goal cell (optional, not used — cheese image is used instead)
GREEN	(0, 200, 0)	For drawing the solution path

4.Visualization & Output

4.1 Visualization in start



Training Progress

During training, the agent gradually improved its performance. The output shows:

- **Initial episodes** had very low rewards (e.g., -281.7 at episode 0), indicating poor navigation and collisions with walls.
- As training progressed, the reward increased steadily, showing learning improvement:

Training Q-learning agent...

Episode 0, Reward: -281.7, Epsilon: 0.20

Episode 100, Reward: -15.6, Epsilon: 0.12

Episode 200, Reward: 6.6, Epsilon: 0.07

Episode 300, Reward: 7.1, Epsilon: 0.04

Episode 400, Reward: 7.0, Epsilon: 0.03

Episode 500, Reward: 7.1, Epsilon: 0.02

Episode 600, Reward: 7.1, Epsilon: 0.01

Episode 700, Reward: 6.9, Epsilon: 0.01

Episode 800, Reward: 7.1, Epsilon: 0.01

Episode 900, Reward: 7.0, Epsilon: 0.01

Training **complete**. Running visualization..

- **Epsilon decay** shows the agent relied less on exploration and more on the learned policy over time.

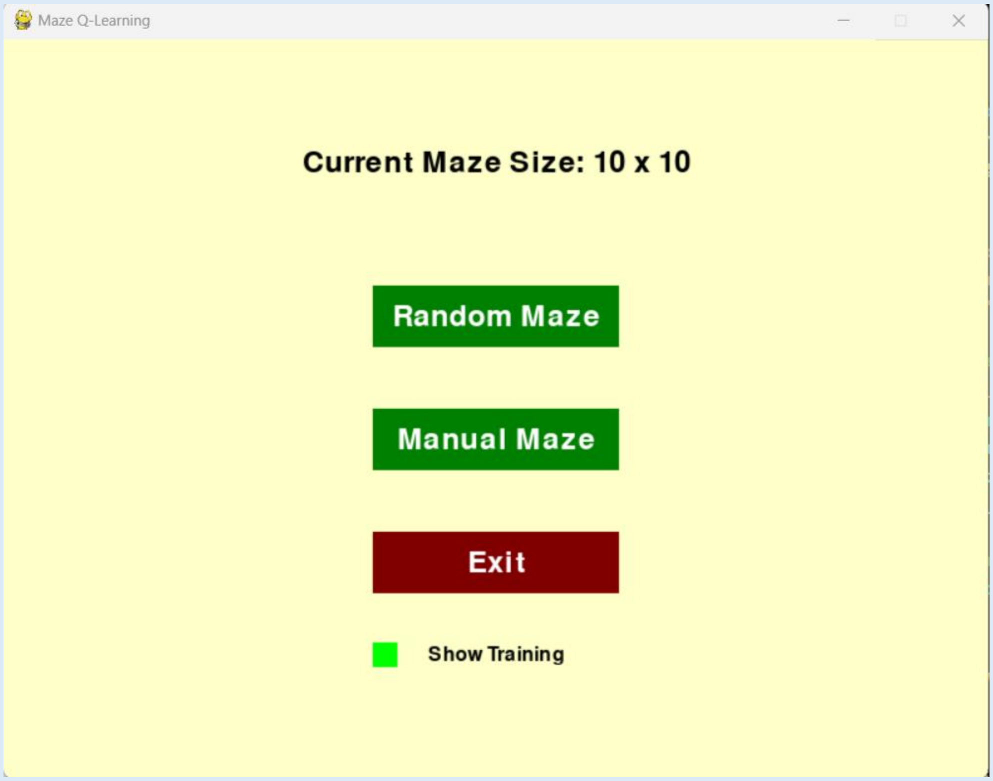
4.2 Final Path Visualization

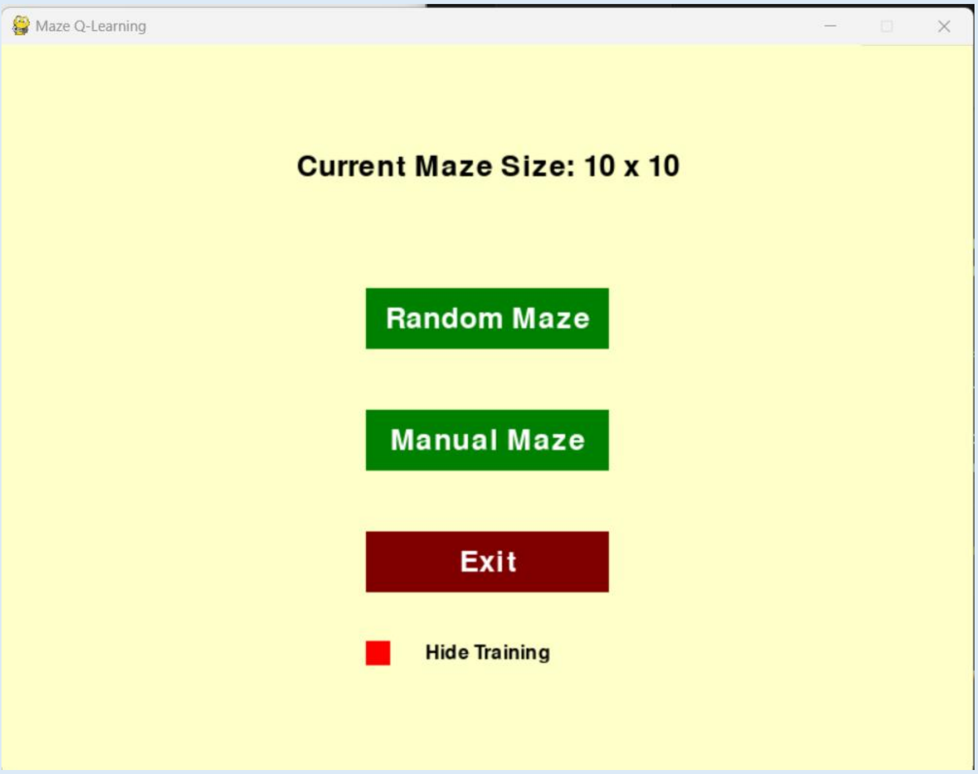
The GUI shows the final path taken by the agent:

- The **green line** indicates the path taken from the **top-left (start)** to the **bottom-right (goal)**.
- **Black cells** represent walls (obstacles), and white cells are walkable.
- The **red circle** marks the goal.
- The final path is optimal and avoids all obstacles.

4.2 Improvement Visualization

When run code >>





Random Maze button:

Generate randomly maze or walls

Manual Maze button:

Generate manually maze or walls

Exit button:

Exit from visualization

Show Training:

To show or hide Training of agent in environment

When selecting Show Training

When press on Random Maze button

Generate randomly maze or walls

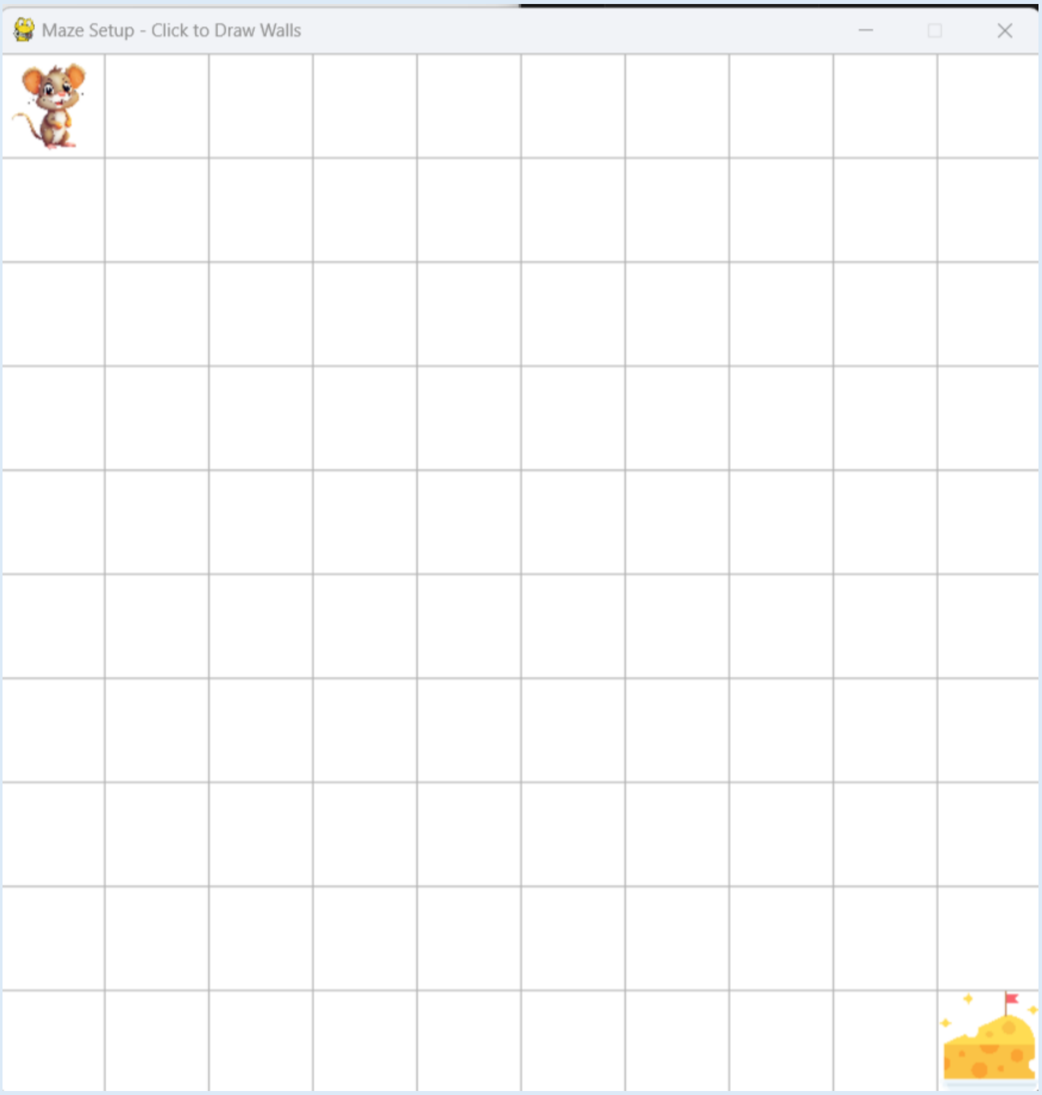
Bold Yellow is visited cell in each Episode in training stage


```
pygame 2.6.1 (SDL 2.28.4, Python 3.13.0)
Hello from the pygame community. https://www.pygame.org/contribute.html
User choice: random

Random maze generated automatically.
Training Q-learning agent...
Visualization enabled.
Episode: 0, Total Reward: -56.9, Epsilon: 0.20
Episode: 1, Total Reward: -177.0, Epsilon: 0.20
Episode: 2, Total Reward: -115.7, Epsilon: 0.19
Episode: 3, Total Reward: -36.9, Epsilon: 0.19
```

Repeat itself to get Episode : 199 , due to Episode : 200 and it started Episode : 0

When select Show Training

When press on Manual Maze button



- ✓ Generate manually maze or walls
- ➢ **Right click in mouse** to add obstacle , **Left click in mouse** to remove obstacle and **Enter in keyboard** to run any training of agent!


Maze Setup - Click to Draw Walls


Maze Setup - Click to Draw Walls

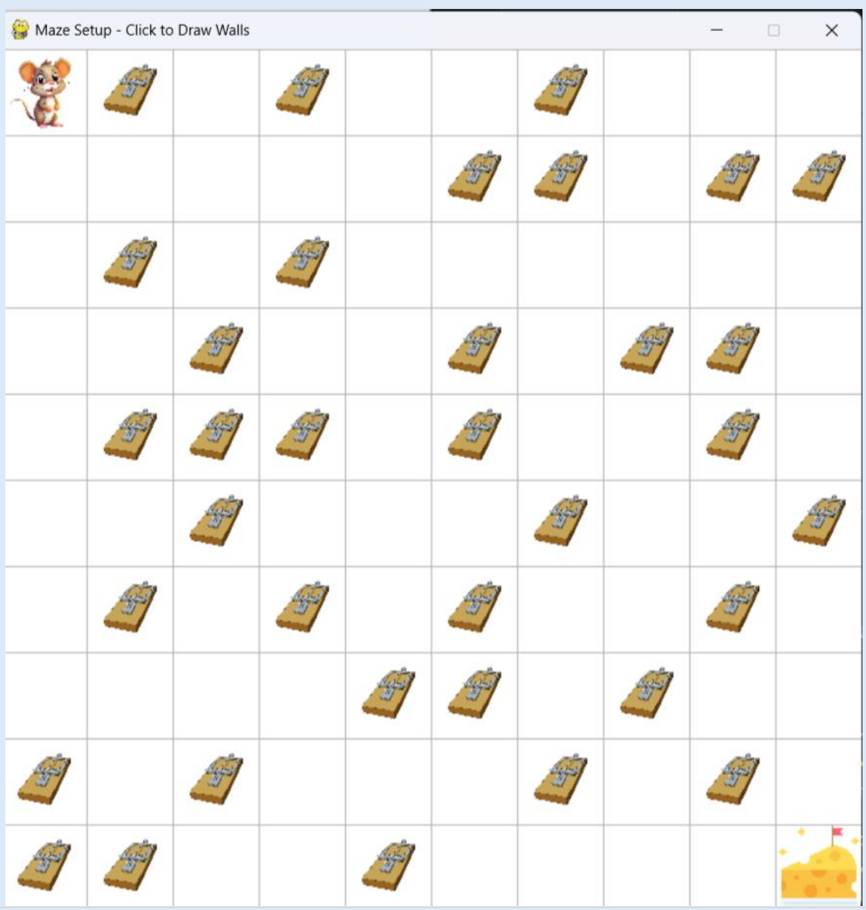
```
Manual maze setup mode activated.
Left-click to add walls, right-click to remove walls.
Press ENTER to start training once done.

Training Q-learning agent...
Visualization enabled.
Episode: 0, Total Reward: -82.7, Epsilon: 0.20
Episode: 1, Total Reward: -27.6, Epsilon: 0.20
Episode: 2, Total Reward: -98.0, Epsilon: 0.19
Episode: 3, Total Reward: -67.9, Epsilon: 0.19
Episode: 4, Total Reward: -14.2, Epsilon: 0.19
```

Repeat to get Episode : 199 , due to Episode : 200 , it started Episode : 0

When select hide Training

When press on Random Maze button

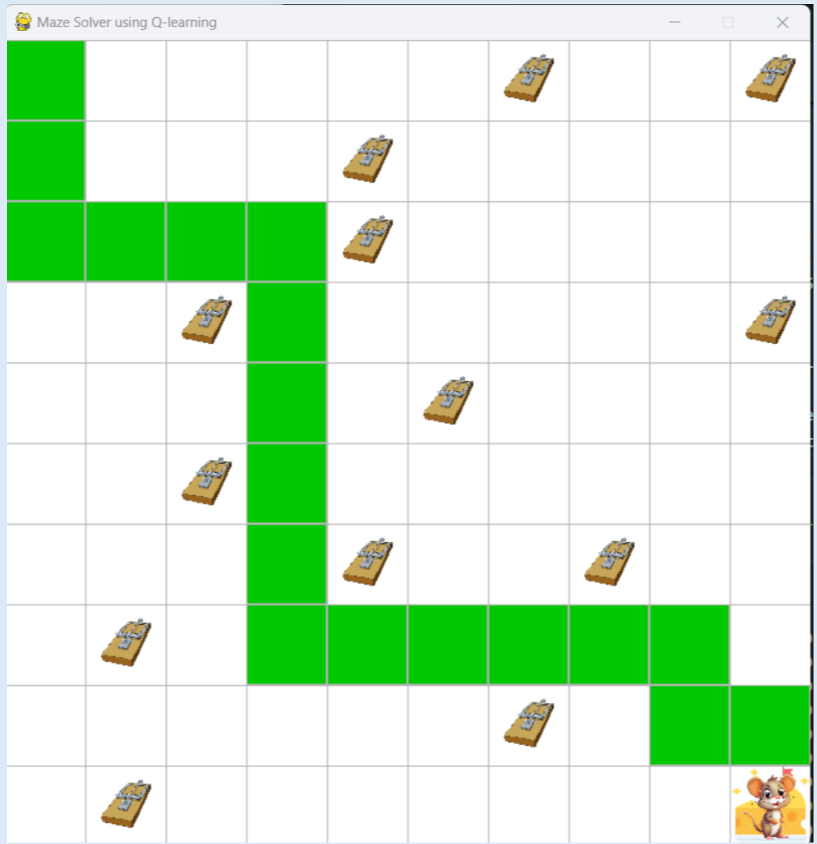
Press on Enter to run




```
Episode: 191, Total Reward: 8.3, Epsilon: 0.03
Episode: 192, Total Reward: 8.3, Epsilon: 0.03
Episode: 193, Total Reward: 7.9, Epsilon: 0.03
Episode: 194, Total Reward: 8.3, Epsilon: 0.03
Episode: 195, Total Reward: 7.8, Epsilon: 0.03
Episode: 196, Total Reward: 8.2, Epsilon: 0.03
Episode: 197, Total Reward: 8.3, Epsilon: 0.03
Episode: 198, Total Reward: 8.3, Epsilon: 0.03
Episode: 199, Total Reward: 8.3, Epsilon: 0.03
Training completed.
Running visualization...
```

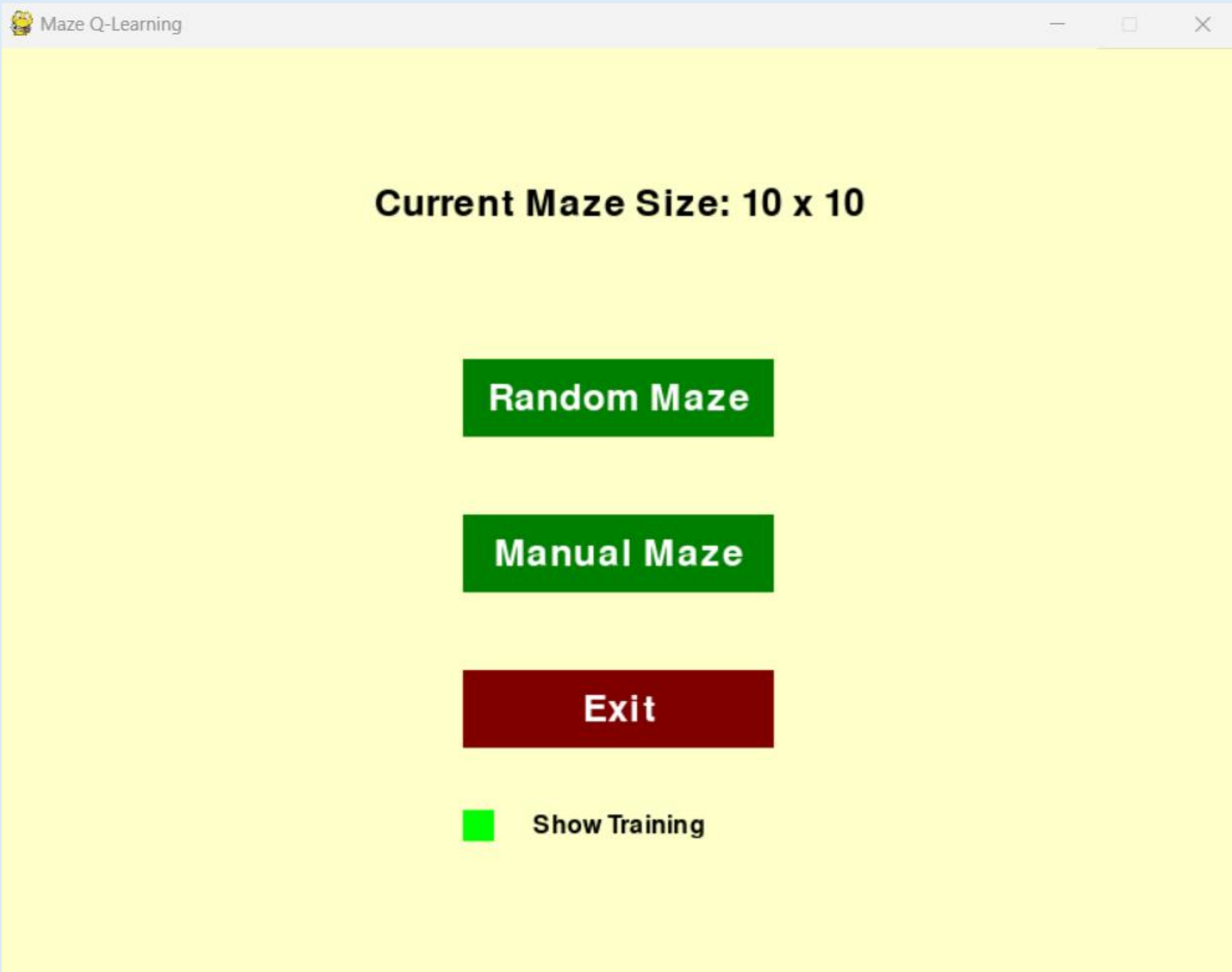
When select hide Training

When press on Manual Maze button




```
Episode: 191, Total Reward: 8.3, Epsilon: 0.03
Episode: 192, Total Reward: 8.1, Epsilon: 0.03
Episode: 193, Total Reward: 8.3, Epsilon: 0.03
Episode: 194, Total Reward: 8.3, Epsilon: 0.03
Episode: 195, Total Reward: 8.3, Epsilon: 0.03
Episode: 196, Total Reward: 7.9, Epsilon: 0.03
Episode: 197, Total Reward: 8.3, Epsilon: 0.03
Episode: 198, Total Reward: 8.3, Epsilon: 0.03
Episode: 199, Total Reward: 8.3, Epsilon: 0.03
Training completed.
Running visualization...
```

To Exit




```
pygame 2.6.1 (SDL 2.28.4, Python 3.13.0)
Hello from the pygame community. https://www.pygame.org/contribute.html
User choice: None
Exiting the program.
PS E:\3CSE.2025\second-term\1. AI 🤖\Assinments-Projects\AI_Project-make-it-realistic1\AI_Project-make-it-realistic>
```

5. Conclusions

- The Q-learning algorithm effectively enables the agent to learn and navigate a complex maze.
- The training phase shows steady improvement in reward, indicating convergence.
- Visualization confirms that the learned path is valid and obstacle-free.
- The project demonstrates the capability of reinforcement learning in solving real-world pathfinding problems.

6. Future Enhancements

- Introduce **dynamic obstacles** for advanced learning.
- Use **deep reinforcement learning (DQN)** for larger and more complex mazes.
- Implement **multi-agent** scenarios for cooperative maze solving.