Maze Simulator and Reinforcement Learning Agent Report

Shivam Chaubey

February 9, 2025

1 Introduction

This report details the implementation of a maze simulator and a reinforcement learning (RL) agent designed to navigate the generated maze. The maze is generated using a randomized depth-first search algorithm. The RL agent employs Q-learning to find a path from a designated start point to an end point within the maze. The primary goal is to develop an agent that can find a reasonably short path through various maze configurations.

2 Code Overview

The Python code is structured into two main components: the `MazeSimulator` class and the `mazeSolverUsingRL` function.

2.1 MazeSimulator Class

The `MazeSimulator` class is responsible for generating and representing the maze.

2.1.1 __init__ Method

This method initializes the maze with a specified width and height, creating a grid filled with walls (represented by 1). It also initializes the start and end points to `None`.

2.1.2 generate_maze Method

This method uses a depth-first search (DFS) algorithm to carve out paths in the maze. It starts from a random cell and recursively visits neighboring cells, marking them as part of the path (represented by 0). The `is_valid_move` helper function ensures that the DFS stays within the maze boundaries and only moves to unvisited cells.

2.1.3 set_start_and_end Method

This method randomly selects two distinct open cells (cells with value 0) in the maze and designates them as the start and end points for the agent.

2.1.4 get_maze_array Method

This method creates a 2D array representation of the maze, where 1 represents walls and 0 represents paths. It also marks the start and end points as 1 for representation purposes.

2.1.5 save_maze_image Method

This method uses Matplotlib to visualize the maze and save it as a PNG image. The start and end points are colored differently for clarity.

2.2 mazeSolverUsingRL Function

This function implements a Q-learning algorithm to train an agent to navigate the maze.

2.2.1 Q-Learning Implementation

The Q-learning algorithm uses a Q-table to store the expected rewards for taking specific actions in different states. The agent iteratively explores the maze, updating the Q-table based on the rewards it receives.

Hyperparameters:

- `alpha` (Learning rate): Determines the extent to which newly acquired information overrides old information.
- `gamma` (Discount factor): Determines the importance of future rewards.
- `epsilon` (Exploration rate): Controls the balance between exploration (trying new actions) and exploitation (using known optimal actions).
- `num_episodes`: The number of training iterations.

Q-Table Initialization: The Q-table is a 3D NumPy array that stores the Q-values for each state-action pair. The dimensions correspond to the height, width, and number of possible actions (up, right, down, left).

Action Mappings: The agent can take four actions: move up, right, down, or left. These actions are represented as coordinate changes: `(-1, 0)`, `(0, 1)`, `(1, 0)`, `(0, -1)`.

Reward Function: The reward function assigns rewards based on the agent's actions:

- Reaching the end point: +100
- Hitting a wall: -10
- Moving to an open cell: -1 (small penalty per step to encourage efficiency)

Q-Learning Loop: The Q-learning loop iterates through a specified number of episodes. In each episode, the agent starts from the start point and continues until it reaches the end point or hits a wall. The agent chooses an action based on an epsilon-greedy policy, which balances exploration and exploitation. The Q-table is updated using the Q-learning update rule.

2.2.2 Path Extraction

After the Q-table is trained, the function extracts the optimal path by starting at the start point and repeatedly choosing the action with the highest Q-value until it reaches the end point. A check is included to prevent the agent from getting stuck in loops.

3 Approach

The approach combines a standard maze generation algorithm (DFS) with a reinforcement learning technique (Q-learning) to solve the maze. This hybrid approach allows for the creation of complex mazes and the training of an agent to efficiently navigate them.

3.1 Maze Generation

The DFS algorithm ensures that the maze is fully connected, meaning there is a path from any cell to any other cell. The randomization aspect ensures that each generated maze is unique.

3.2 Reinforcement Learning

Q-learning is a model-free reinforcement learning algorithm that learns an optimal policy by iteratively updating a Q-table. The Q-table stores the expected rewards for taking specific actions in different states. The epsilon-greedy policy allows the agent to explore the maze initially and then gradually exploit its knowledge to find the shortest path.

4 Code Listing

```python
import random
import matplotlib.pyplot as plt
import numpy as np

# Do not change the maze generation

class MazeSimulator:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.maze = [[1 for _ in range(width)] for _ in range(height)]  # 1 = wall, 0 = path
        self.start = None
        self.end = None

    def generate_maze(self):
        def is_valid_move(x, y):
            return 0 <= x < self.height and 0 <= y < self.width and self.maze[x][y] == 1

        def dfs(x, y):
            self.maze[x][y] = 0
            directions = [(0, 2), (2, 0), (0, -2), (-2, 0)]
            random.shuffle(directions)

            for dx, dy in directions:
                nx, ny = x + dx, y + dy
                if is_valid_move(nx, ny):
                    self.maze[x + dx // 2][y + dy // 2] = 0  # Remove the wall
                    dfs(nx, ny)

        start_x, start_y = random.randrange(0, self.height, 2), random.randrange(0, self.width, 2)
        dfs(start_x, start_y)

    def set_start_and_end(self):
        open_cells = [(x, y) for x in range(self.height) for y in range(self.width) if self.maze[x][y] == 0]
        self.start, self.end = random.sample(open_cells, 2)

    def get_maze_array(self):
        maze_array = [[1 if self.maze[x][y] == 0 or (x, y) in [self.start, self.end] else 0 for y in range(self.width)] for x in range(self.height)]
        return maze_array

    def save_maze_image(self, filename="maze.png"):
        maze_copy = [[2 if (x, y) == self.start else 3 if (x, y) == self.end else self.maze[x][y] for y in range(self.width)] for x in range(self.height)]
        plt.figure(figsize=(10, 10))
        plt.imshow(maze_copy, cmap="viridis", origin="upper")
        plt.axis("off")
        plt.savefig(filename)
        plt.close()


    def mazeSolverUsingRL(self):
        """
        Solve the maze using Q-learning.

        The RL agent learns a policy for moving from the start to the end cell.
        Actions: 0 = Up, 1 = Right, 2 = Down, 3 = Left.
        Rewards:
        - +100 for reaching the end.
        - -10 for hitting a wall (invalid move).
        - -1 per step to encourage shorter paths.

        Returns:
            path (list of tuple): The sequence of (row, col) states representing the optimal path.
        """

        # Hyperparameters
```

```
    alpha = 0.1         # Learning rate
    gamma = 0.9         # Discount factor
    epsilon = 0.1       # Exploration rate
    num_episodes = 100000  # Number of training episodes
    max_steps = self.width * self.height * 4  # Maximum steps per episode (safeguard)

    # Initialize the Q-table with zeros.
    # Dimensions: [height][width][number_of_actions]
    q_table = np.zeros((self.height, self.width, 4))

    # Define the four possible actions as (dx, dy) movements.
    actions = [(-1, 0),  # Up
               (0, 1),   # Right
               (1, 0),   # Down
               (0, -1)]  # Left

    def is_valid_state(x, y):
        """Return True if (x, y) is within bounds and is an open cell (path)."""
        return 0 <= x < self.height and 0 <= y < self.width and self.maze[x][y] == 0

    def choose_action(state):
        """Choose an action using an epsilon-greedy strategy."""
        if random.uniform(0, 1) < epsilon:
            return random.choice(range(4))  # Explore: choose a random action
        else:
            return np.argmax(q_table[state[0], state[1]])  # Exploit: choose the best-known action

    def get_reward(state):
        """
        Return the reward for moving into a given state.
        - 100 if the state is the end.
        - -10 if the state is invalid (e.g. a wall or out-of-bounds).
        - -1 for any other (valid) step.
        """
        if state == self.end:
            return 100
        if not is_valid_state(state[0], state[1]):
            return -10
        return -1

    # ---------------------------
    # Q-learning Training Loop
    # ---------------------------
    for episode in range(num_episodes):
        state = self.start
        steps = 0
        while steps < max_steps:
            # Choose an action from the current state
            action = choose_action(state)
            x, y = state
            dx, dy = actions[action]
            new_state = (x + dx, y + dy)
            reward = get_reward(new_state)

            # Determine the maximum Q-value for the new state (if valid)
            if is_valid_state(new_state[0], new_state[1]):
                best_next_q = np.max(q_table[new_state[0], new_state[1]])
            else:
                best_next_q = 0

            # Update the Q-value for the current state and action
            q_table[x, y, action] += alpha * (reward + gamma * best_next_q - q_table[x, y, action])

            # Terminate the episode if:
            # - The agent reaches the goal.
            # - The agent takes an invalid move (hits a wall).
            if new_state == self.end or not is_valid_state(new_state[0], new_state[1]):
                break

```

```
                # Otherwise, move to the new state and continue
                state = new_state
                steps += 1

        # --------------------------------
        # Extract the Optimal Path
        # --------------------------------
        path = [self.start]
        current_state = self.start
        visited = set([self.start])  # To help prevent loops

        # Follow the greedy policy derived from the Q-table until the end is reached.
        while current_state != self.end:
            action = np.argmax(q_table[current_state[0], current_state[1]])
            dx, dy = actions[action]
            new_state = (current_state[0] + dx, current_state[1] + dy)

            # If the new state is invalid or we've already visited it, break out to avoid an infinite loop.
            if not is_valid_state(new_state[0], new_state[1]) or new_state in visited:
                break

            path.append(new_state)
            visited.add(new_state)
            current_state = new_state

        return path


# Game flow
if __name__ == "__main__":
    width, height = 21, 21

    # Initialize simulator
    simulator = MazeSimulator(width, height)
    simulator.generate_maze()
    simulator.set_start_and_end()

    # Generate and display maze
    maze_array = simulator.get_maze_array()
    print("Maze array:")
    for row in maze_array:
        print(row)
    print(f"Start point: {simulator.start}")
    print(f"End point: {simulator.end}")

    # Solve maze using RL
    path = simulator.mazeSolverUsingRL()  # Call the solver
    print(f"\nRL Agent Path ({len(path)} steps):")
    print(path)

    # Save visualization with path
    simulator.save_maze_image("maze.png")
    print("\nMaze image saved as 'maze.png'")
```

Listing 1: Maze Simulator Code

5 Conclusion

The implemented maze simulator and Q-learning agent provide a functional solution for generating and navigating mazes. The agent learns to find paths through the maze, demonstrating the effectiveness of reinforcement learning techniques in solving pathfinding problems. The performance of the agent can be further improved by fine-tuning the hyperparameters and exploring other reinforcement learning algorithms. Future work could also involve visualizing the agent's learning process and comparing the performance of different maze-solving algorithms.