ECE440 Project 1

Maze Solver using Markov Decision Processes Report

Yankai Zhao(800145233)

Electrical and Computer Engineering, Lehigh University

October 17, 2025

Abstract

This paper applies the Value Iteration algorithm within the Reinforcement Learning framework to solve a maze navigation problem. By modeling the maze as a Markov Decision Process (MDP) and defining its state, action, transition, and reward structures, the agent progressively learns the optimal policy that minimizes path length and successfully reaches the goal. Experimental results show that the algorithm converges efficiently and produces interpretable optimal paths across different maze sizes.

1 Introduction

Reinforcement Learning (RL) provides a mathematical framework for decision-making under uncertainty, where an agent learns through trial and error by interacting with an environment. The maze navigation problem offers a simple yet illustrative context for RL algorithms. This study applies the *Value Iteration* method, a classical dynamic programming approach, to compute the optimal policy in a maze environment without requiring explicit training data.

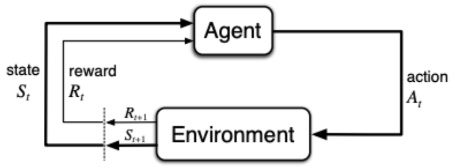

Figure 1: Agent–Environment Interaction

2 Materials & Methods

2.1 Markov Decision Process Formulation

The maze is represented as a deterministic Markov Decision Process (MDP):

$$\mathcal{M} = \langle S, A, P, R, \gamma \rangle \tag{1}$$

where S is the set of accessible grid cells, $A = \{\text{up}, \text{down}, \text{left}, \text{right}\}$, $P(s'|s,a)$ defines the deterministic transition dynamics, $R(s,a,s')$ is the reward function, and γ is the discount factor controlling future reward weighting.

```
class MazeMDP:
    def __init__(self, maze, start, end, gamma=0.995):
        self.maze = maze
        self.start = start
        self.end = end
        self.gamma = gamma
        self.rows, self.cols = maze.shape
        self.actions = [(0,1),(0,-1),(1,0),(-1,0)]
        self.states = [(i,j) for i in range(self.rows)
                         for j in range(self.cols)
                         if maze[i,j]==1]

    def is_valid(self, state):
        i,j = state
        return 0<=i<self.rows and 0<=j<self.cols and\
        self.maze[i,j]==1

    def next_state(self, state, action):
        i,j = state
        di,dj = action
        next_state = (i+di, j+dj)
        return next_state if self.is_valid(next_state) else state

    def reward(self, state, action, next_state):
        if next_state == self.end:
            return 100
        elif next_state == state:
            return -5
        else:
            return -0.1
```

Figure 2: The MazeMDP class definition

2.2 Value Functions and Bellman Equations

The agent evaluates each state through the state-value function:

$$V^\pi(s) = E_\pi\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s\right] \tag{2}$$

and the action-value function:

$$Q^\pi(s,a) = E_\pi\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a\right] \tag{3}$$

Their relationship is given by:

$$V^\pi(s) = \sum_a \pi(a|s) Q^\pi(s,a) \tag{4}$$

The optimal value function satisfies the Bellman optimality equation:

$$V^*(s) = \max_a \left[R(s,a) + \gamma V^*(s')\right] \tag{5}$$

2.3 Value Iteration Algorithm

The Value Iteration procedure updates state values according to:

$$V_{k+1}(s) = \max_a [R(s,a) + \gamma V_k(s')] \tag{6}$$

The process terminates once the following convergence criterion is met:

$$|V_{k+1}(s) - V_k(s)| < \varepsilon \tag{7}$$

After convergence, the optimal policy is obtained by:

$$\pi^*(s) = \arg\max_a [R(s,a) + \gamma V^*(s')] \tag{8}$$

The core update loop of the Value Iteration algorithm is implemented in Python as shown in Figure 3.

```
def value_iteration(mdp, eps=1e-6, max_iter=5000):
    Value = {state: 0 for state in mdp.states}
    for it in range(max_iter):
        delta = 0
        newValue = Value.copy()
        for state in mdp.states:
            if state == mdp.end:
                continue
            q_values = []
            for action in mdp.actions:
                next_state = mdp.next_state(state, action)
                reward = mdp.reward(state, action, next_state)
                q_values.append(reward + mdp.gamma * Value[next_state])
            newValue[state] = max(q_values)
            delta = max(delta, abs(newValue[state] - Value[state]))
        Value = newValue
        if delta < eps:
            print(f"Value iteration converged after {it + 1} iterations")
            break
    return Value
```

Figure 3: Value Iteration algorithm loop

2.4 Reward Function Design

The reward structure is given by:

$$R(s,a,s') = \begin{cases} +100, & s' = s_{\text{goal}} \\ -5, & s' = s \\ -0.1, & \text{otherwise} \end{cases} \tag{9}$$

This formulation promotes reaching the goal quickly and penalizes invalid moves or wall collisions.

2.5 Convergence Analysis

Because the Bellman operator T is a γ-contraction:

$$\|T[V_1] - T[V_2]\|_\infty \le \gamma \|V_1 - V_2\|_\infty \tag{10}$$

Value Iteration therefore converges to a unique fixed point V^*, satisfying:

$$\|V_k - V^*\|_\infty \le \frac{\gamma^k}{1-\gamma} R_{\max} \tag{11}$$

3 Experimental Results

To evaluate the performance of the proposed Value Iteration–based maze solver, two environments with different spatial resolutions were tested: a 10×10 maze and a 20×20 maze. The binary maze images were preprocessed into state matrices, where black cells represented obstacles ($s \notin S$) and white cells represented traversable states ($s \in S$). The agent was initialized at the top-left corner, while the goal was placed near the bottom-right corner.

For the 10×10 maze, the algorithm converged after 538 iterations, resulting in an optimal path of 471 steps from Start $(5,0)$ to End $(99,104)$. In contrast, the 20×20 maze required 1834 iterations to converge, producing a longer but smoother optimal trajectory of 775 steps between Start $(5,0)$ and End $(199,204)$.

The results reveal several key patterns. First, as the maze size increases, the number of reachable states $|S|$ grows quadratically, which directly leads to a proportional increase in computational time. Second, although the convergence rate slows with larger mazes, the Value Iteration algorithm still achieves a stable policy that yields globally consistent paths. Finally, the resulting trajectories, shown as red lines in Figure 5 and Figure 7, demonstrate the effectiveness of deterministic state transitions and the impact of the reward design on exploration efficiency.

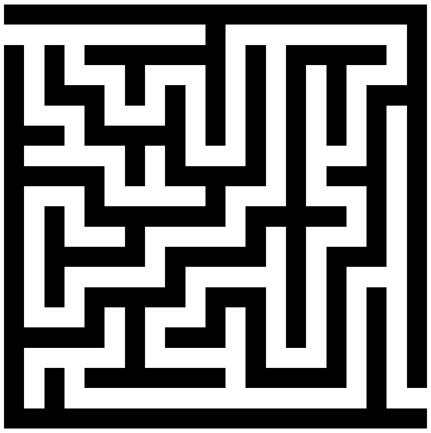
Figure 4: 10*10 size Maze

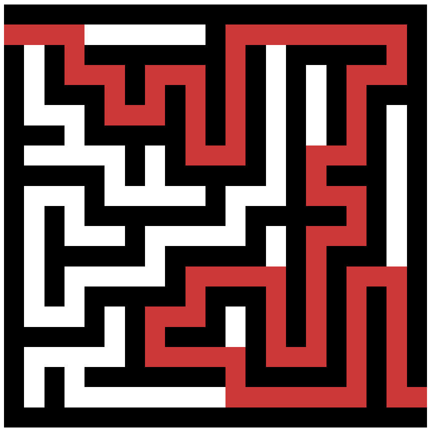
Figure 5: 10*10 Maze result

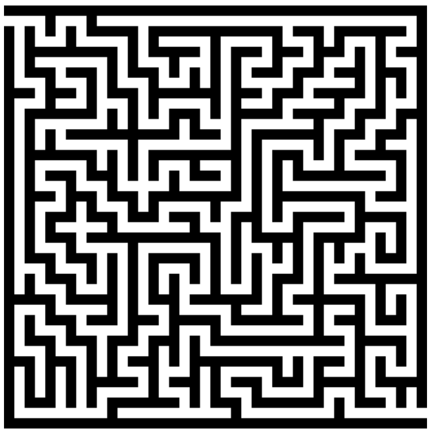

Figure 6: 20*20 size Maze

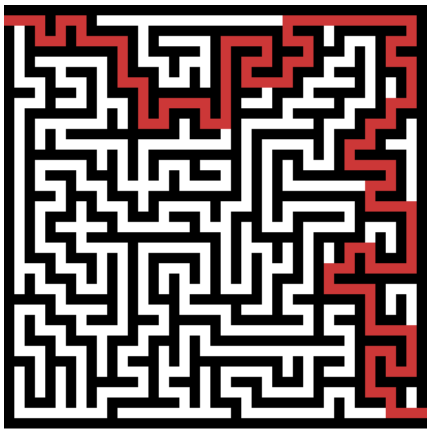

Figure 7: 20*20 Maze result

4 Conclusion

This study demonstrates the effectiveness of the Value Iteration algorithm in solving maze navigation problems. By iteratively applying the Bellman optimality equations, the agent achieved stable convergence to the optimal policy. The approach validates the theoretical basis of dynamic programming in reinforcement learning and serves as a foundation for more advanced methods such as DQN and Actor–Critic.