

Introduction to Artificial Intelligence

Fall 2023

Maze Solver Using Reinforcement Learning

Farah Alaa El Din Khattab - 7487
Mariam Khaled Abdel Hakim - 7528

1. Introduction

In the pursuit of efficient maze-solving strategies, this project focuses on leveraging Policy Iteration and Value Iteration, two dynamic programming techniques within the realm of reinforcement learning. These algorithms provide iterative solutions to the maze navigation problem, offering optimal paths while circumventing obstacles. By delving into these iterative methods, we aim to contribute insights that can impact future developments in robotic systems and reinforcement learning, pushing the boundaries of intelligent agent navigation.

2. Assumptions


Here are the assumptions made throughout the code:

1. User Inputs:
 - a. Maze size in terms of N ($N \times N$).
 - b. Discount factor (γ).
 - c. Maze itself.
2. There's no initial state.
3. There's several cells in the maze that has rewards.
4. The cell with the greatest reward is the terminal state.
5. Deterministic environment.
6. Rewards of terminal state is the value in the maze and rewards of empty states equals -1 .
7. Empty states are represented as ".".
8. Barriers are represented as "B".
9. The agent will go to the terminal state but if another cell with reward on its way he will take it and continue to the terminal state.

3. Algorithms Used

3.1. Policy Iteration

Policy Iteration (PI)

1. $i=0$; Initialize $\pi_0(s)$ randomly for all states s
2. While $i \neq 0$ or $|\pi_i - \pi_{i-1}| > 0$  Use a L1 norm: measures if the policy changed for any state
 - Policy **evaluation**: Compute value of π_i
 - $i=i+1$
 - Policy **improvement**:

$$Q^{\pi_i}(s, a) = r(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) V^{\pi_i}(s')$$

$$\pi_{i+1}(s) = \arg \max_a Q^{\pi_i}(s, a)$$

3.2. Value Iteration

Value Iteration (VI)

1. Initialize $V_0(s)=0$ for all states s
2. Set $k=1$
3. Loop until [finite horizon, convergence]
 - For each state s

$$V_{k+1}(s) = \max_a R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V_k(s')$$

- View as Bellman backup on value function

$$V_{k+1} = BV_k$$

$$\pi_{k+1}(s) = \arg \max_a R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V_k(s')$$

4. Data Structures Used

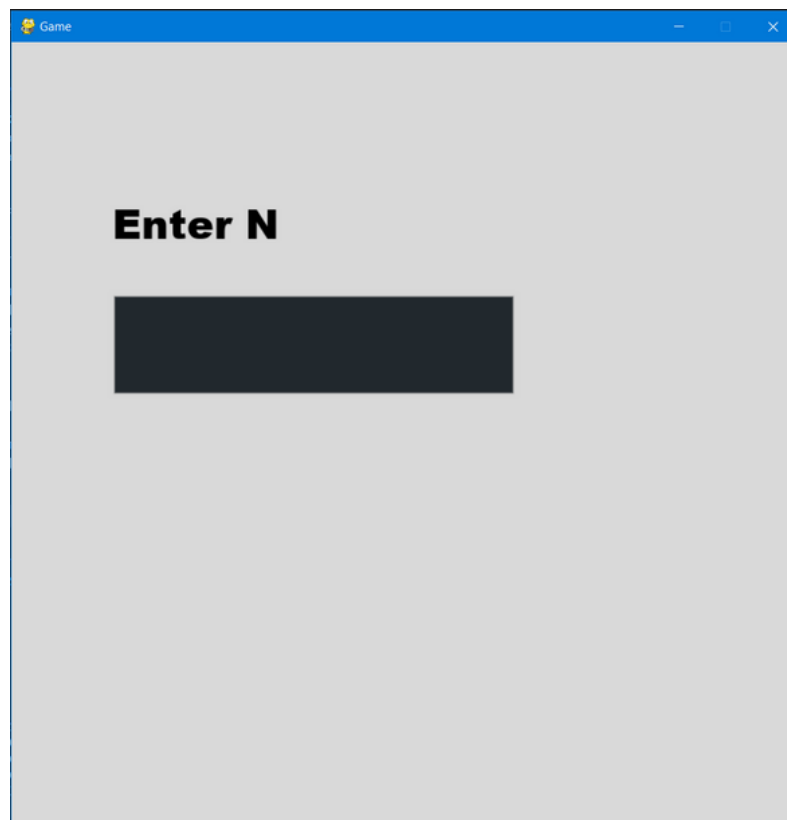
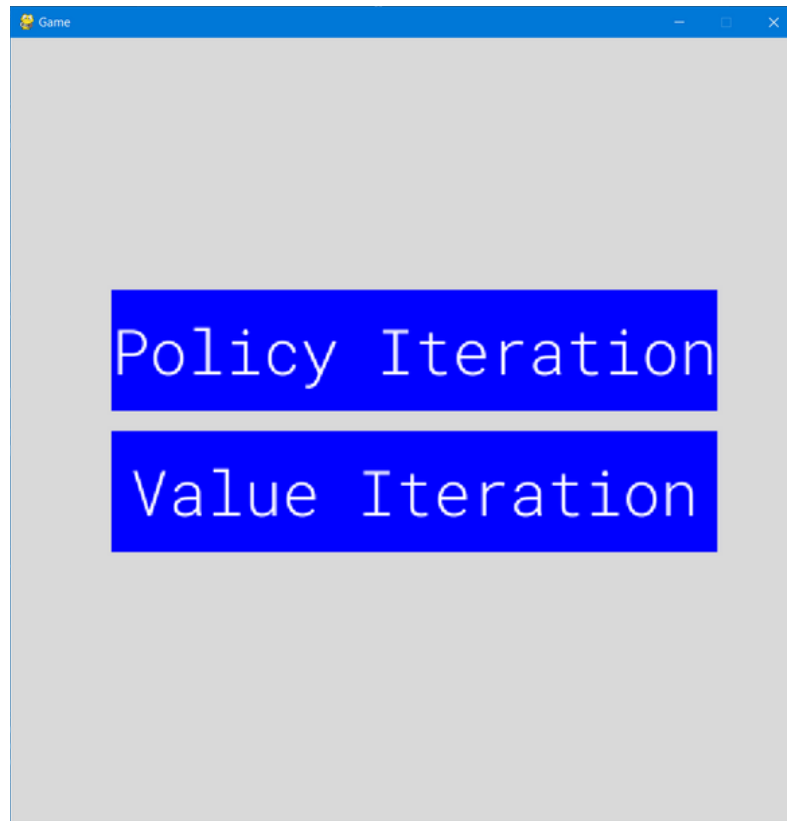
The following part will explain any data structures used within the code:

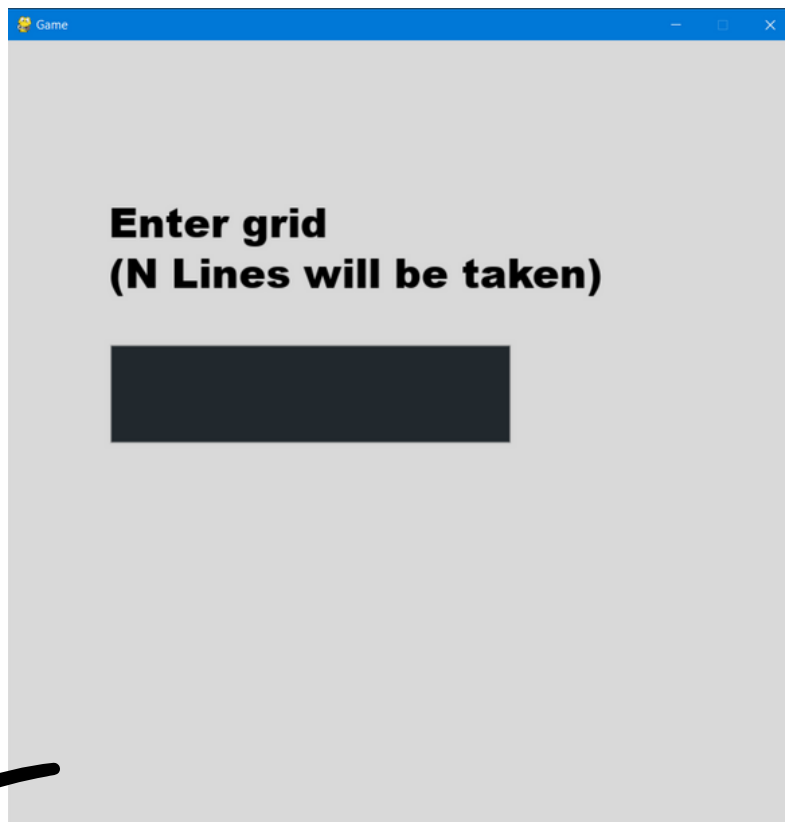
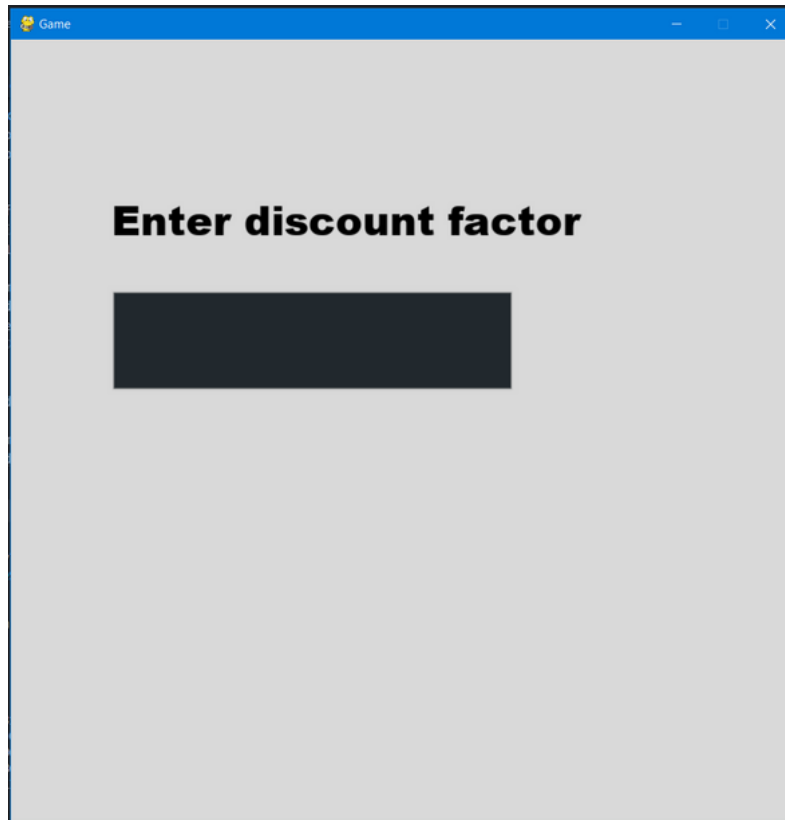
- **The Maze Grid:** The game grid itself is represented as a **2D matrix** of size (N×N)
- **Constants:** All constants in the game are present in a separate "Constants.py" file, these constants include:
 - **Colors:** specified as RGB values as (R, G, B) each in hexadecimal
 - **Fonts:** Font names are declared
 - **Sizes and Dimensions:** Screen, window, and shape dimensions are declared"Constants.py" file also includes some static functions that are used in most files.
- **GUI Objects:** They are all PyGame shapes
 - **Buttons:** A Button Class is created, clicking and hovering mouse events are handled in the class, and each Button is a PyGame rect.
 - **Input:** The input is taken from as a keyboard event in PyGame.
 - **Grid:** The board is drawn using black lines from PyGame.
- **Policy Iteration:**
 - **indx:** A 2D list (N,N) to store the index of each state.
 - **next_states:** A 2D (N*N, 4) list to store the next available actions for each state.
 - **policies:** A list (N*N) to store the current policies for each state.
 - **rewards:** A list (N*N) to store the immediate rewards for each state.
 - **new_value_fns:** A list (N*N) to store the new value functions during policy evaluation.
 - **new_policies:** A list to store the new policies during policy improvement.
 - **value_fns:** A NumPy array (N*N) to store the value functions for each state.
 - **policy_matrix:** A NumPy array to store the optimal policy matrix.
 - **action_value_fns:** A list (4) to store action value functions for available actions of each state during policy improvement.
- **Value Iteration:**
 - **rewards:** A 2D NumPy array to store rewards for each cell in the maze.
 - **actions_str:** A list of string representations for different actions ('>', 'v', '<', '^').
 - **state_values:** A 2D NumPy array to store the values of states in the maze.
 - **state_values_prev:** A copy of state_values from the previous iteration.

5. Game GUI

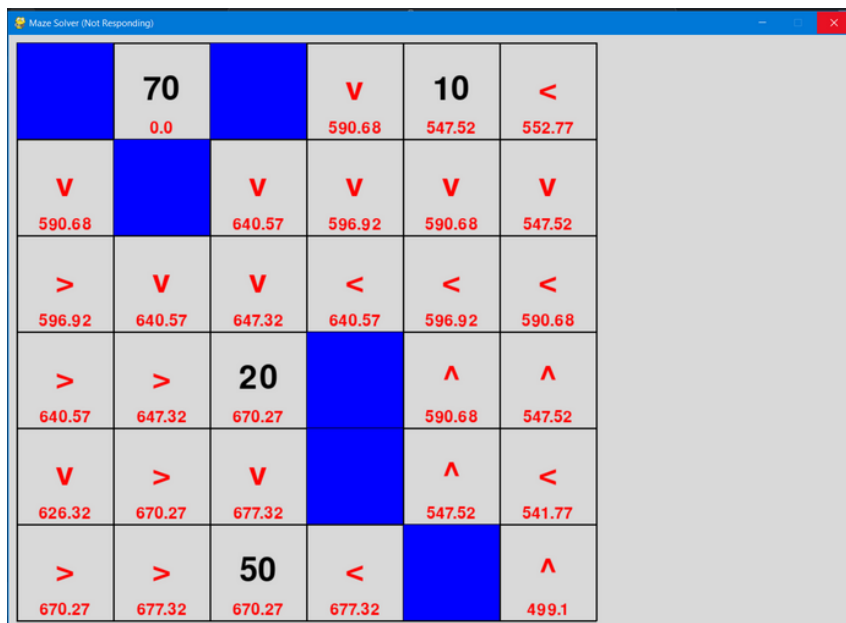
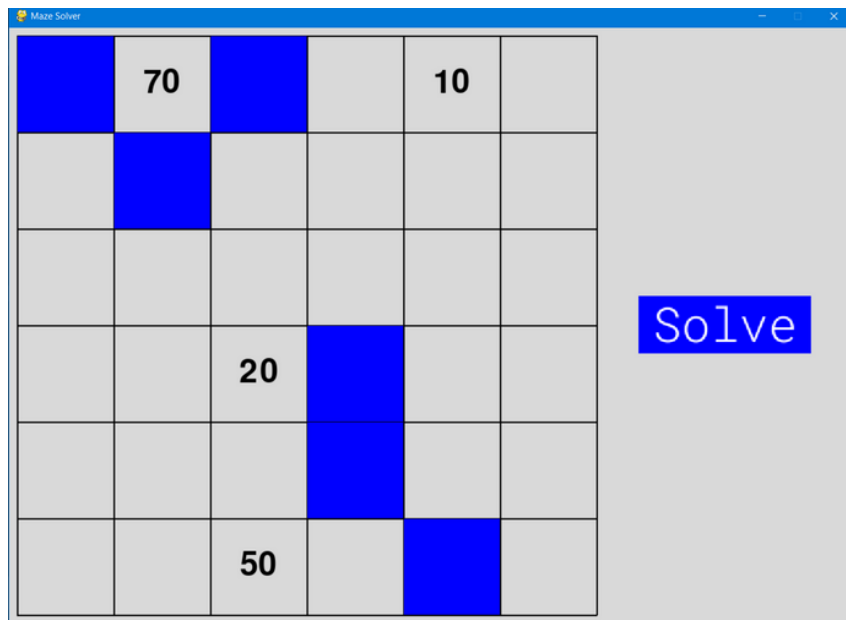
Our game consists of 2 main windows and 3 input windows, main windows:

1. **Main Menu:** where the user chooses the whether the maze to be solved using policy iteration or value iteration.
2. **Game Window:** This is the actual maze where it simulates the solution.





Press enter after each line



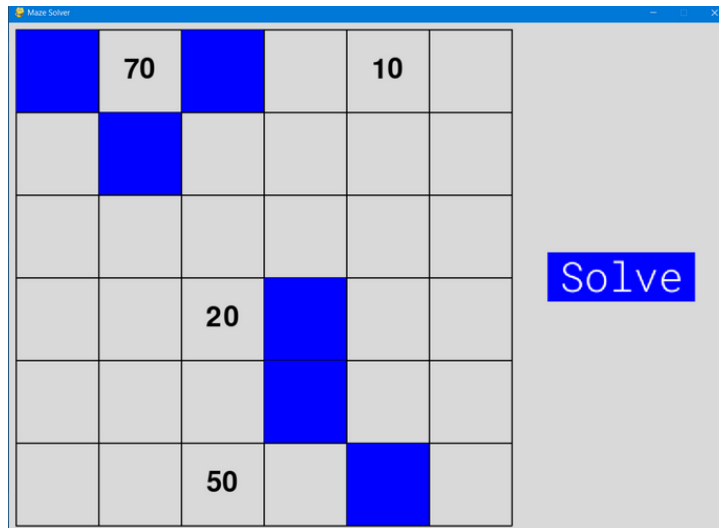
While the agent solving the maze



After the agent solves the maze

6. Sample Runs

SAMPLE RUN #1



Policy Iteration

The screenshot shows the results of Policy Iteration. The grid contains numerical values and symbols (>, <, ^, v) indicating the optimal policy. A 'Solve' button is on the right.

	70		v	10	<
v		v	v	v	v
v	>	v	<	<	<
>	>	20		^	^
>	v	v		^	<
>	>	50	<		^

Total time taken: 0.03690218925476074 seconds

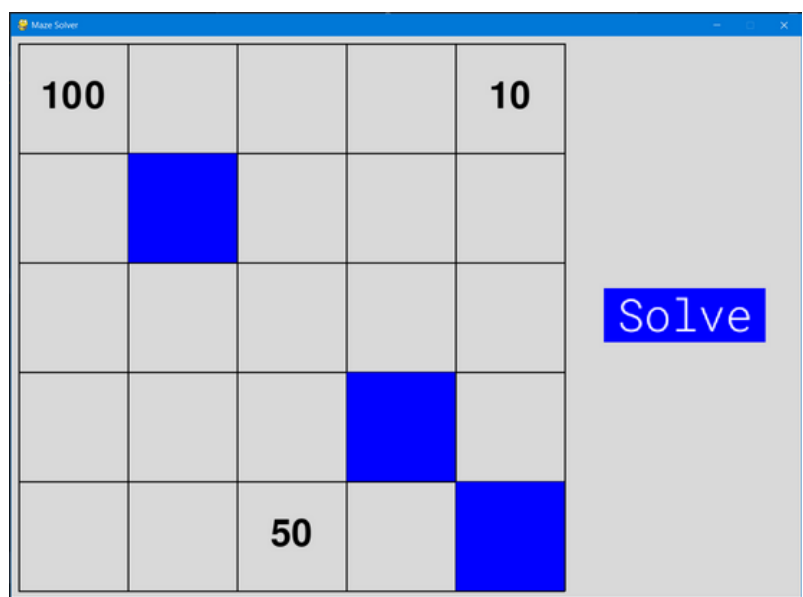
Value Iteration

The screenshot shows the results of Value Iteration. The grid contains numerical values. A 'Solve' button is on the right.

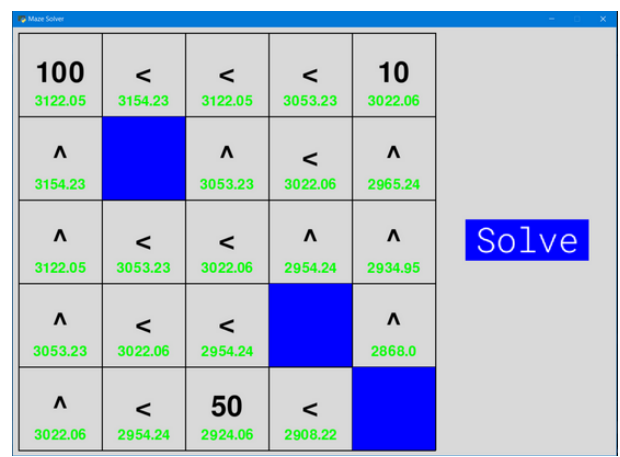
	70		v	10	<
v		v	v	v	v
>	>	v	<	<	<
>	>	20		^	<
>	>	v		^	<
>	>	50	<		^

Total time taken: 0.003988504409790039 seconds

SAMPLE RUN #2



Policy Iteration



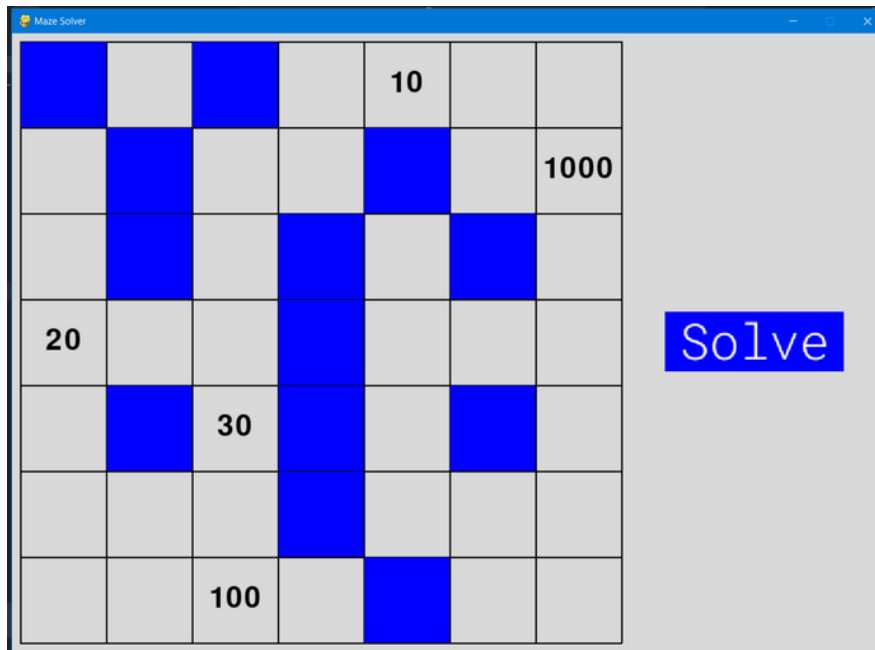
Total time taken: 0.01196599006652832 seconds

Value Iteration

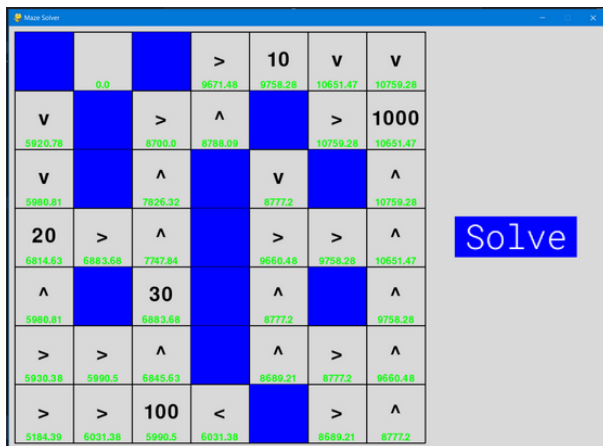


Total time taken: 0.0029914379119873047 seconds

SAMPLE RUN #3



Policy Iteration



Total time taken: 0.0428929328918457 seconds

Value Iteration



Total time taken: 0.006983041763305664 seconds