




MAZE GAME IN RL

By students:

Ahmed Wael Mohamed 6704

Mostafa Mohamed Salah 6894



Maze Game Report

Maze Generator and MDP Conversion

Introduction

This Python script defines a class **Maze** that generates a maze and converts it into an MDP (Markov Decision Process). The maze is represented as a matrix, where '#' represents walls and ' ' represents empty spaces. The MDPState class is used to create MDP states for each free space in the maze. Additionally, exit points in the maze are randomly selected, and rewards are assigned to these exit points.

Class Structure

The **Maze** class has the following key components:

Constructors

1. **Default Constructor (__init__):** This constructor creates a maze of specified dimensions (**w** and **h**). It also allows specifying the number of exit points (**num_exits**). The maze is generated using a randomized depth-first search algorithm.
2. **Alternative Constructor:** Currently commented out, this constructor allows creating a maze from a predefined grid (**input_grid**). Exit points and their associated rewards can be provided for initialization.

Maze Generation

- The **make_maze** method generates a maze using a randomized depth-first search algorithm (**walk** method).
- The **walk** method is a recursive function that explores the maze randomly and creates passages by removing walls.

MDP Conversion

- The **maze_to_mdp** method converts the maze into an MDP by creating an MDPState object for each free space. Walls are represented as '#' in the MDP.

- Exit points are randomly chosen, and rewards are assigned to them.

MDPState Class

The **MDPState** class represents a state in the gridworld of a Markov Decision Process (MDP). Each instance of this class is associated with a specific space in the grid and has directional attributes pointing to neighboring squares. The class includes the following attributes:

- **up**: Coordinates of the square above the current state.
- **down**: Coordinates of the square below the current state.
- **left**: Coordinates of the square to the left of the current state.
- **right**: Coordinates of the square to the right of the current state.
- **reward**: The immediate reward associated with being in the current state (default value is -1).
- **value**: The current value of the state, often used in reinforcement learning algorithms.

Constructor

The constructor **__init__** initializes the state with the specified coordinates for each direction (**up**, **down**, **left**, **right**). Additionally, it allows setting the initial **reward** and **value** for the state.

String Representation

The **__str__** method provides a string representation of the state by returning its **value**. This is useful for easily inspecting the value of a state when printing or debugging.

Important Notes on the code :

The code doesn't take an input but it generates the maze randomly and generates the exit points randomly based on a given number, also , It calculates the path cost based on a given start point (One start Point).

Now , Let's Continue explaining the code

Policy Iteration for Gridworld Navigation

Introduction

The script focuses on implementing policy iteration for a gridworld navigation problem using a Markov Decision Process (MDP). The script utilizes a **Maze** class to generate a gridworld and convert it into MDP states. The **policy_iteration** function is responsible for finding the optimal policy for navigating the grid. The script also includes a function (**calculate_optimal_path_cost_from_start**) to calculate the cost of the optimal path from a specified starting position.

Policy Iteration

The policy iteration algorithm consists of two main phases: policy evaluation and policy improvement.

Policy Evaluation

The policy evaluation phase involves iteratively updating the value of each state based on the current policy. The Bellman equation for policy evaluation is given by:

$$V(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s)) \cdot V(s')$$

Here,

- $V(s)$ is the value of state s ,
- $R(s)$ is the immediate reward of being in state s ,
- γ is the discount factor,
- $P(s'|s, \pi(s))$ is the transition probability from state s to s' under policy π ,
- $V(s')$ is the value of the next state s' .

Policy Improvement

After policy evaluation, the policy improvement phase updates the policy with greedy actions based on the updated values. The policy improvement equation is given by:

$$\pi'(s) = \underset{a \in A}{\operatorname{argmax}} Q(s, a) \pi'(s) = \underset{a \in A}{\operatorname{argmax}} Q(s, a)$$

Here,

- $\pi'(s)$ is the updated policy for state s ,
- A is the set of possible actions,
- $Q(s, a)$ is the action-value function representing the expected return of taking action a in state s and following the current policy.

The policy iteration process continues iteratively until the policy no longer changes, indicating convergence.

Helper Functions

- The **prettify_policy** function converts the policy matrix into a visually appealing string representation, replacing action names with arrows.
- The **calculate_optimal_path_cost_from_start** function calculates the cost of the optimal path from a specified starting position according to the given policy, considering the rewards associated with each state.

Value Iteration for Gridworld Navigation

Introduction

The provided Python script focuses on implementing value iteration for solving a gridworld navigation problem using a Markov Decision Process (MDP). Similar to policy iteration, the script utilizes a **Maze** class to generate a gridworld and convert it into MDP states. The **value_iteration** function is responsible for finding the optimal policy for navigating the grid. The script also includes a function (**calculate_optimal_path_cost_from_start**) to calculate the cost of the optimal path from a specified starting position.

Value Iteration

Value iteration is an iterative algorithm used to compute the optimal value function and policy for an MDP. The process involves two main steps: updating the values of states and extracting the optimal policy from the updated values.

Value Update

The script iteratively updates the value of each state using the Bellman equation:

$$V(s) = R(s) + \gamma \max_{a \in A} \sum_{s'} P(s'|s, a) \cdot V(s')$$

Here,

- $V(s)$ is the value of state s ,
- $R(s)$ is the immediate reward of being in state s ,
- γ is the discount factor,
- A is the set of possible actions,
- $P(s'|s, a)$ is the transition probability from state s to s' under action a ,
- The expression $\max_{a \in A} \sum_{s'} P(s'|s, a) \cdot V(s')$ represents the maximum expected future value.

Policy Extraction

Once the values converge, the optimal policy is extracted by selecting actions that maximize the expected return for each state.

Helper Functions

- The **prettify_policy** function converts the policy matrix into a visually appealing string representation, replacing action names with arrows.
- The **calculate_optimal_path_cost_from_start** function calculates the cost of the optimal path from a specified starting position according to the given policy, considering the rewards associated with each state.

Sample Runs and Outputs:

On $w = 3$ and $h = 3$:

_____ Policy Iteration _____

_____ Iterations _____

Iteration : 0

↑↑↑↑↑↑↑

↑↑↑↑↑↑↑

↑↑↑↑↑↑↑

↑↑↑↑↑↑↑

↑↑↑↑↑↑↑

↑↑↑↑↑↑↑

↑↑↑↑↑↑↑

Iteration : 1

#####

#↑↑→↑←#

#↑#####

#↑#↑↑↑#

#↑###↑#

#↑↑↑↑↑#

#####

Iteration : 2

#####

#↑→→↑←#

#↑#####

#↑#↑↑↑#

#↑###↑#

#↑↑↑↑↑#

#####

Iteration : 3

#####

#→→→↑←#

#↑#####

#↑#↑↑↑#

#↑###↑#

#↑↑↑↑↑#

#####

Iteration : 4

#####

#→→→↑←#

#↑#####

#↑#↑↑↑#

#↑###↑#

#↑←↑↑↑#

#####

Iteration : 5

#####

#→→→↑←#

#↑#####

#↑#↑↑↑#

#↑###↑#

#↑←←↑↑#

#####

Iteration : 6

#####

#→→→↑←#

#↑#####

#↑#↑↑↑#

#↑###↑#
#↑←←←↑#

#####

Iteration : 7

#####

#→→→↑←#

#↑#####

#↑#↑↑↑#

#↑###↑#

#↑←←←←#

#####

Iteration : 8

#####

#→→→↑←#

#↑#####

#↑#↑↑↑#

#↑###↓#

#↑←←←←#

#####

Iteration : 9

#####

#→→→↑←#

#↑#####

#↑#↑↑↓#

#↑###↓#

#↑←←←←#

#####

Iteration : 10

#####

#→→→↑←#

#↑#####

#↑#↑→↓#

#↑###↓#

#↑←←←←#

#####

Iteration : 11

#####

#→→→↑←#

#↑#####

#↑#→→↓#

#↑###↓#

#↑←←←←#

#####

_____Path to Goal_____

['right', 'right', 'right', 'up', '#']

_____Exit Points_____

[(1, 4), (1, 4)]

_____Maze_____

#####

#

#####

#

#

#

#####

_____Optimal Policy_____

#####

#→→→↑←#

#↑#####

#↑#→→↓#

#↑###↓#

#↑←←←←#

#####

_____Path Cost_____

Optimal Path Cost from (1, 1): 7 with time = 0.003966808319091797 seconds

_____Value Iteration_____

Iteration : 0

[70.18999999999997, 79.09999999999997, 88.99999999999996, 99.99999999999996,
88.99999999999996, 62.17099999999997, 54.953899999999976, 12.648024530411384,
15.164471700457092, 17.96052411161899, 48.458509999999998, 21.06724901290999,
42.612658999999999, 37.351393099999999, 32.616253789999999, 28.354628410999999,
24.519165569899999]

_____Path to Goal_____

['right', 'right', 'right', 'up', '#']

_____Exit Points_____

[(1, 4), (1, 4)]

_____Maze_____

#####

#

#####

#

#

#

#####

_____Optimal Policy_____

#####

#→→→↑←#

#↑#####

#↑#→→↓#

#↑###↓#

#↑←←←←#

#####

_____Path Cost_____

Optimal Path Cost from (1, 1): 7 with time = 0.0 seconds

On w = 4 h =4 :

_____Policy Iteration_____

_____Iterations_____

Iteration : 0

↑↑↑↑↑↑↑↑

↑↑↑↑↑↑↑↑

↑↑↑↑↑↑↑↑

↑↑↑↑↑↑↑↑

↑↑↑↑↑↑↑↑

↑↑↑↑↑↑↑↑

↑↑↑↑↑↑↑↑

↑↑↑↑↑↑↑↑

↑↑↑↑↑↑↑↑

Iteration : 1

#####

#↑#↑↑↑↑↑#

#↑#↓#↑###

#↑→↓#↑#↑#

#↑###↑#↑#

#↑#↑#↑#↓#

#↑#↑#↑#←#
#↑↑↑#↑→↑#

#####

Iteration : 2

#####

#↑#↓↑↑↑↑#

#↑#↓#↑###

#→→↓#↑#↑#

#↑###↑#↓#

#↑#↑#↑#↓#

#↑#↑#↑#←#

#↑↑↑#→→↑#

#####

Iteration : 3

#####

#↑#↓←↑↑↑#

#↓#↓#↑###

#→→↓#↑#↓#

#↑###↑#↓#

#↑#↑#↑#↓#

#↑#↑#↓#←#

#↑←↑#→→↑#

#####

Iteration : 4

#####

#↓#↓←←↑↑#

#↓#↓#↑###

#→→↓#↑#↓#

#↑###↑#↓#

#↑#↑#↓#↓#
#↑#↑#↓#←#
#↑←←#→→↑#

#####

Iteration : 5

#####

#↓#↓←←←↑#
#↓#↓#↑###
#→→↓#↑#↓#
#↑###↓#↓#
#↑#↑#↓#↓#
#↑#↓#↓#←#
#↑←←#→→↑#

#####

Iteration : 6

#####

#↓#↓←←←←#
#↓#↓#↑###
#→→↓#↑#↓#
#↑###↓#↓#
#↑#↓#↓#↓#
#↑#↓#↓#←#
#↑←←#→→↑#

#####

_____ Path to Goal _____

['down', 'down', 'right', 'right', 'down', '#']

_____ Exit Points _____

[(6, 7), (3, 3)]

_____ Maze _____

#####

#

###

#

#

#

#

#

#####

_____Optimal Policy_____

#####

#↓#↓←←←←#

#↓#↓#↑###

#→→↓#↑#↓#

#↑###↓#↓#

#↑#↓#↓#↓#

#↑#↓#↓#←#

#↑←←#→→↑#

#####

_____Path Cost_____

Optimal Path Cost from (1, 1): 6 with time = 0.011967658996582031 seconds

_____Value Iteration_____

Iteration : 0

[62.170999999999997, 79.099999999999997, 70.189999999999997, 62.170999999999997,
54.9538999999999976, 48.458509999999998, 70.189999999999997, 88.999999999999996,
54.9538999999999976, 79.099999999999997, 88.999999999999996, 99.999999999999996,
48.458509999999998, 70.189999999999997, 70.189999999999997, 48.458509999999998,
79.099999999999997, 62.170999999999997, 28.354628410999999, 54.9538999999999976,
88.999999999999996, 54.9538999999999976, 32.616253789999999, 62.170999999999997,

99.99999999999996, 48.458509999999998, 42.612658999999999, 37.351393099999999,
70.189999999999997, 79.099999999999997, 88.99999999999996]

_____Path to Goal_____

['down', 'down', 'right', 'right', 'down', '#']

_____Exit Points_____

[(6, 7), (3, 3)]

_____Maze_____

#####

#

###

#

#

#

#

#

#####

_____Optimal Policy_____

#####

#↓#↓←←←←#

#↓#↓#↑###

#→→↓#↑#↓#

#↑###↓#↓#

#↑#↓#↓#↓#

#↑#↓#↓#←#

#↑←←#→→↑#

#####

_____Path Cost_____

Optimal Path Cost from (1, 1): 6 with time = 0.0 seconds

Outputs on The Ide is conclusive of the steps of both the algorithms and how they work , also each class has a sample run of it solely to be tested alone “Just Run The classes and Change the dimensions to compare the times and working of both algorithms”