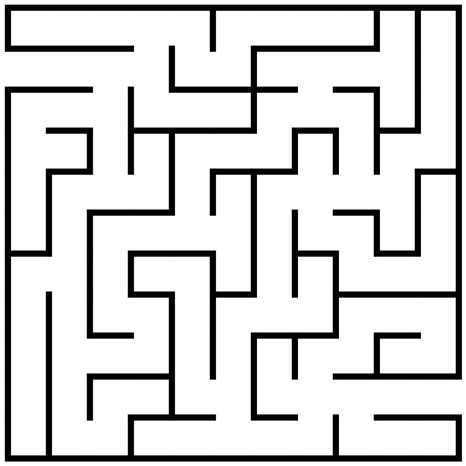
METU EE 449

Computational Intelligence

Getting Out of a Maze with a Reinforcement Learning Agent



Homework 3 - Reinforcement Learning

Due: 23:55, 20/05/2024

Late submissions are welcome, but penalized according to the following policy:

- 1 day late submission: HW will be evaluated out of 70.
- 2 days late submission: HW will be evaluated out of 40.
- 3 or more days late submission: HW will not be evaluated.

You should prepare your homework by yourself alone and you should not share it with other students, otherwise you will be penalized.

Introduction

In this homework, you will model a maze, implement and analyze Temporal Difference Learning and Q Learning to solve it, and provide various plots such as heatmaps and learning curves, concluding in a detailed report that discusses your methods, the evolution of algorithm performance, and a comparative analysis of the learning algorithms used.

The implementations will be in Python language and you will be using NumPy and Matplotlib. Also, some helper codes are provided to you under *HW3* folder in *ODTUClass* course page.

Homework Task and Deliverables

In the scope of this homework, you will explore the application of Reinforcement Learning to solve a maze navigation problem by employing Temporal Difference (TD) Learning and Q Learning. You will begin by modeling the maze, defining its states, actions, transition probabilities, and a strategic reward function. Subsequently, you will implement **TD(0) Learning** to estimate utility value functions and utilize **Q Learning** to directly learn an optimal navigation policy. Throughout the assignment, you are required to provide various plots such as heatmaps of utility values and learning curves to visualize the progression and effectiveness of the algorithms.

This homework is composed of six parts. In the first part, you will answer some basic questions to ensure you have a solid theoretical foundation in reinforcement learning. The second part involves implementing the maze, where you will define states, actions, transition probabilities, and a reward function. In the third part, you will develop the TD(0) algorithm to estimate utility values, followed by the fourth part, where you will implement Q Learning to directly learn optimal policies. The fifth part requires conducting

experimental work to test the algorithms, including data visualization through various plots. Finally, the sixth part involves discussions and analysis of your experimental results, evaluating the effectiveness of the implemented algorithms and considering potential improvements.

You should submit a single report in which your answers to the questions, the required experimental results (performance curve plots, visualizations etc.) and your deductions are presented for each part of the homework. Moreover, you should append your Python codes to the end of the report for each part to generate the results and the visualizations of the experiments. Namely, all the required tasks for a part can be performed by running the related code file. The codes should be well structured and **well commented**. The non-text submissions (e.g. image) or the submissions lacking comments will not be evaluated. Similarly answers/results/conclusions written in code as a comment will not be graded.

The report should be in portable document format (pdf) and named as *hw3_name_surname_eXXXXXX* where *name*, *surname* and *X*s are to be replaced by your name, surname and digits of your user ID, respectively. You do not need to send any code files to your course assistant(s), since everything should be in your single pdf file.

Do not include the codes in *utils.py* to the end of your pdf file.

1 Basic Questions

Compare the following terms in reinforcement learning with their equivalent terms in supervised learning (if any) and provide a definition for each, in your own wording:

- Agent
- Environment
- Reward
- Policy
- Exploration
- Exploitation

2 Implementing the Maze

In this part, you will implement the maze in Figure 1 in Python, using NumPy and basic mathematical operations. Starting point is shown with blue cell, traps with red and the goal point with green. Use 0 for free space, 1 for obstacle, 2 for traps and 3 for goal on your maze in Python. Define states (each cell of the maze), possible actions (move up, down, left, right), and rewards (negative for each state, large positive for the desired state (goal), large negative for undesired final state (trap)).

The environment is stochastic, the probabilities for the results of the chosen actions are as follows:

- Probability of going for the chosen direction: 0.75
- Probability of going opposite of the chosen direction: 0.05
- Probability of going each of perpendicular routes of the chosen direction: 0.10

If the agent hits the wall or goes out of the bounds as the result of the action, it goes back to the current state and the episode continues with the next action.

Firstly you need to call required libraries and functions. You also need to call functions from `utils.py`.

```
import numpy as np
from matplotlib import pyplot as plt
```

Then, you will define an MazeEnvironment class. You can use the following code directly, but you need to complete missing parts denoted with FILL HERE.

```
class MazeEnvironment:
    def __init__(self):
        # Define the maze layout, rewards, action space (up, down, left, right)

        self.start_pos = (0,0) # Start position of the agent
        self.current_pos = self.start_pos

        self.state_penalty = -1
        self.trap_penalty = -100
        self.goal_reward = 100

        self.actions = 0: (-1, 0), 1: (1, 0), 2: (0, -1), 3: (0, 1)

    def reset(self):
        self.current_pos = #FILL HERE
        return #FILL HERE
    def step(self, action):
        #FILL HERE
```

Make sure the functions of the MazeEnvironment class works properly, then you may go on with the next part.

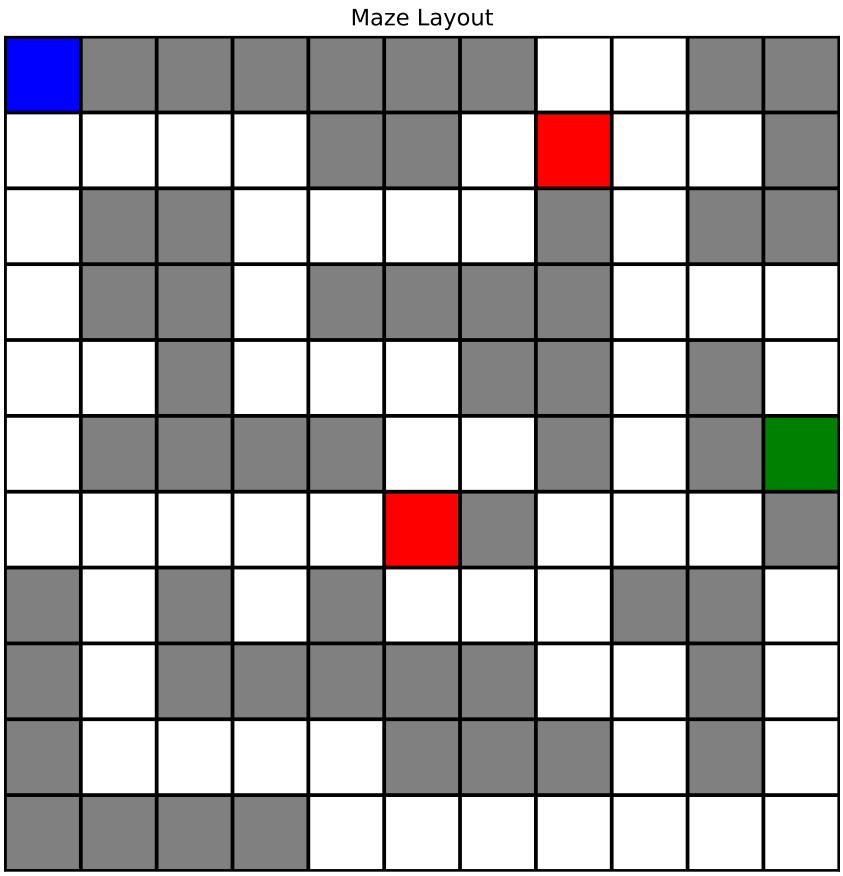


Figure 1: The maze.

3 Temporal Difference (TD) Learning

In this part, **TD(0) algorithm** is implemented to update the utility value estimate after every action taken in the environment. This means adjusting the estimated utility value of the current state based on the reward received and the estimated utility value of the next state. For that, ε-greedy strategy can be used, where the agent randomly chooses an action with probability ε (exploration), or chooses the best known action based on the current utility values (exploitation).

```
class MazeTD0(MazeEnvironment): # Inherited from MazeEnvironment
    def __init__(self, maze, alpha=0.1, gamma=0.95, epsilon=0.2, episodes=10000):
        super().__init__()

        self.maze = maze
        self.alpha = alpha #Learning Rate
        self.gamma = gamma #Discount factor
        self.epsilon = epsilon #Exploration Rate
        self.episodes = episodes
        self.utility = #FILL HERE, Encourage exploration
```

```
    def choose_action(self, state):
        #Explore and Exploit: Choose the best action based on current utility values
        #Discourage invalid moves
        #FILL HERE
    def update_utility_value(self, current_state, reward, new_state):
        current_value = #FILL HERE
        new_value = #FILL HERE

        # TD(0) update formula
        self.utility[current_state] = #FILL HERE

    def run_episodes(self):
        #FILL HERE
        return self.utility
```

Then, the environment and the agent should be ready for the learning.

```
# Create an instance of the Maze with TD(0) and run multiple episodes
    maze = #FILL HERE
    maze_td0 = MazeTD0(self, maze, alpha=0.1, gamma=0.95, epsilon=0.2, episodes=10000)
    final_values = maze_td0.run_episodes()
```

4 Q Learning

In this part, you will implement the Q Learning algorithm to directly learn an optimal policy. For this, you will initialize the Q values (state-action pair values) arbitrarily, implement the Q Learning update rule and use an ε-greedy strategy to improve exploration and exploitation balance.

```
class MazeQLearning(MazeEnvironment): # Inherited from MazeEnvironment
    def __init__(self, maze, alpha=0.1, gamma=0.95, epsilon=0.2, episodes=10000):
        super().__init__()

        self.maze = maze
        self.alpha = alpha #Learning Rate
        self.gamma = gamma #Discount factor
        self.epsilon = epsilon #Exploration Rate
        self.episodes = episodes
        self.q_table = #FILL HERE, Initialize Q-table

    def choose_action(self, state):
        #Explore and Exploit
        #FILL HERE

    def update_q_table(self, action, current_state, reward, new_state):
        current_q = #FILL HERE
        max_future_q = #FILL HERE
        new_q = #FILL HERE
        self.q_table[state[0], state[1], action] = #FILL HERE

    def run_episodes(self):
        #FILL HERE
```

Table 1: Parameter configurations to be experimented.

α	0.001	0.01	**0.1**	0.5	1.0
γ	0.10	0.25	0.50	0.75	**0.95**
ε	0	**0.2**	0.5	0.8	1.0

Then, the environment and the agent should be ready for the learning.

```
# Create an instance and train
    maze = #FILL HERE, Use 0 = free space, 1 = obstacle, 2 = goal
    maze_q_learning = MazeQLearning(self, maze, alpha=0.1, gamma=0.95, epsilon=0.2, episodes=10000)
    maze_q_learning.run_episodes()
```

Now, you should be able to access the Q table via `maze_q_learning.q_table`. Using Q table, you may find the utility value with simple mathematical operations and NumPy. Hint: Check the lecture notes.

5 Experimental Work

For both TD(0) Learning and Q Learning, you will experiment on several different hyperparameters. Namely,

- Learning rate (α)
- Discount factor (γ)
- Initial exploration rate (ε)

To see the effect of each hyperparameter, run the algorithm for each value in a row given in Table 1 while using default values for the other hyperparemeters. The default values are bolded. Use <num_episodes>= 10000. You can use the same set of parameters for both TD(0) and Q Learning.

For each hyperparameter, explain its effect and give the value which produces the best result. For each experiment, you need to include:

- Utility value function heatmap for episodes 1, 50, 100, 1000, 5000, 10000 (use the function from `utils.py`)
- Policy for episodes 1, 50, 100, 1000, 5000, 10000 (use the function from `utils.py`)
- Convergence plot. A line graph showing the change in the value function from one iteration to the next over time, to assess convergence. Plotting the sum of absolute differences between the value functions of successive iterations against the number of episodes can illustrate this.

6 Discussions

Answer the following questions:

1. Discuss the rationale behind your transition probabilities and reward function. How do they influence the agent's behavior?
2. In TD(0) Learning, how did the utility value functions evolve over time? You may explain via your heatmap plots at different stages.
3. In TD(0) Learning, did the utility value function converge? If so, how many episodes were required for convergence?
4. How sensitive is the TD(0) Learning process to parameters like learning rate and discount factor? What effects did changes in these parameters have on the learning outcomes?
5. What challenges did you encounter while implementing TD(0) Learning, and how did you address them?

6. Compare the convergence of Q Learning to that of TD(0) Learning. Which one stabilized faster and why might that be?
7. Discuss the impact of exploration strategies (e.g., ε-greedy) on the learning process. How did changes in ε affect performance over time?
8. Compare and contrast TD(0) Learning and Q Learning based on your experimental results. Which method would you prefer for different types of mazes and why?
9. What modifications to the maze design or problem setup could potentially improve the learning process or make it more challenging?
10. How could the algorithms be adjusted or combined with other techniques to enhance their performance or reliability?