CS 451/551 - Assignment 3

May 19, 2023

1 Introduction

In the third homework, you will be required to implement two reinforcement learning algorithms, namely Q-Learning and SARSA, for developing an intelligent agent that can navigate through Grid-World and find the optimal path as in the first assignment. The Grid-World is a world consisting of four different types of nodes: “Flat”, “Mountain”, “Goal” and “Death”. The agent aims to navigate through the Grid-World and reach the “Goal” node while maximizing the total reward/score. The agent can move in four directions: ‘UP’, ‘DOWN’, ‘LEFT’, and ‘RIGHT’. Additionally, the transition from one node to another has a reward based on the type of node.

You will be provided with a code base and should use Python programming language to implement the algorithms. The algorithms should be able to find the optimal list of actions that lead to the maximum total reward. Additionally, you will be required to compare the performance of the algorithms in various scenarios. You can use the grid generator in the code base to create more scenarios. The performance metrics to be considered include computational time, the number of expansions, the expansion path, and the maximum expanded depth, among others.

The deadline for this homework is **5 June 2023**.

2 Environment&Implementation

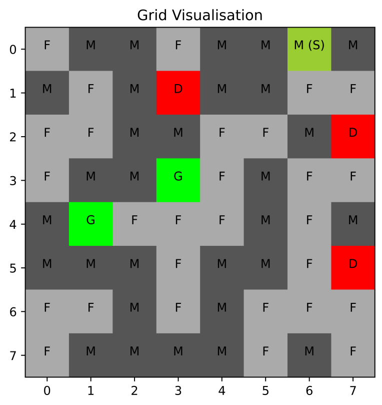
This section describes the Grid-World with transition rewards and the rules in detail. Besides, it explains the provided code base.

2.1 Grid-World

Grid-World is a square map with dimensions $n \times n$ containing a total of n^2 nodes, as illustrated in Figure 1. The agent’s goal is to reach a target node by riding a bike and moving in the directions of ‘UP’, ‘DOWN’, ‘LEFT’, and ‘RIGHT’. The Grid-World consists of four types of nodes, each with its own characteristics:

- **Flat**: It is easy to ride a bike.

Figure 1: Demonstration of Example Grid-World



- **Mountain**: It is hard to ride a bike.
- **Goal**: This node is the target destination for the agent.
- **Death**: If the agent reaches this node, it fails.

One of the nodes in the map is designated as the starting point for the agent. Moving from one node to another node provides a reward based on the type of the nodes, as shown in Table 1. The goal of the tree-search algorithms is to find the best sequence of actions that will maximize the total reward.

From	**To**	**Transition Reward**
Flat	Flat	-1
Flat	Mountain	-3
Mountain	Mountain	-2
Mountain	Flat	-1
(Any)	Goal	100
	Death	-100

Table 1: Predefined Transition Reward between two Node Types

Note: If the agent tries moving out of a Grid-World, it stays on the same node and gets a -1 reward.

2.2 Implementation

In this text, the instructions for a programming assignment are given. The assignment involves implementing Q-Learning and SARSA algorithms in Python for a grid-world environment. The students are expected to complete corresponding methods in “QLearning.py” and “SARSA.py” files in the given code base. The code-base is implemented with Python programming language, and Jupyter is not allowed. Students who do not know how to code with Python can learn from available sources on the internet. The "Environment.py" file contains the necessary information about the grid world and provides moving on it and the transition reward. The "RLAgent.py" file contains an abstract class for Reinforcement Learning (i.e., RL) agents; each RL algorithm must be a sub-class. The "QLearning.py" file contains the implementation of the Q-Learning algorithm, while the "SARSA.py" file contains the implementation of the SARSA algorithm. Finally, some helpful links to learn Python programming language from scratch are shared at the end of the document (Section 6).

- **Environment.py**: "Environment" class in this file holds the necessary information of the Grid-World, and provides moving on it and the transition reward. It takes the file path where the scenario data exists. The scenario data file is in Pickle format. Some important methods in this class are listed:
  - **reset**(): The agent goes to a predefined starting point/node, then the method returns the state index of the starting point.
  - **to_state**(*position*): This method converts a given position (i.e., row and column indices as a list) to a state index.
  - **to_position**(state): This method converts a state index to a position.
  - **set_current_state**(state): It changes the agent's current position to the given state index.
  - **get_node_type**(position): It returns the node type of the given position as a string. This string is the first (upper) character of the node type name.
  - **get_reward**(*previous_pos*, *next_pos*): This method provides the transition reward from the previous position (*previous_pos*) to the next position (*new_pos*) based on Table 1.
  - **is_done**(*position*): It states whether the given position ends the episode. In other words, it checks if the agent reaches one of the "Goal" and "Death" nodes or not.
  - **move**(*action*): It moves the agent from the current state based on the given action. The action can be "UP" (0), "LEFT" (1), "DOWN" (2) or "RIGHT" (3). Besides, it returns the new state index, the transition reward, and whether the episode is done.
  - **get_goals**(): It finds the state indices of all goals, then returns them as a list.
- **rl_agents/RLAgent.py**: "RLAgent" class in this file is an abstract class. Each RL algorithm must be a sub-class of it.
  - **Constructor**(*env*, *discount_rate*, *action_size*): This constructor takes some necessary parameters and "Environemnt" object.

  - **train**(***kwargs*): This abstract method trains the RL agent with the corresponding RL algorithm.
  - **act**(*state*, *is_training*): This abstract method decides on an action that will take based on the given "state". Note that the action decision can be different depending on if it is called during training or validation.
  - **validate**(): This method plays the provided environment, then returns a list of action decided by trained policy and the total collected reward. Note that this method should be called after training.

- **rl_agents/QLearning.py**: "QLearningAgent" class in this file contains the implementation of the Q-Learning algorithm for this assignment. Also, it is the sub-class of the "RLAgent" class. This means that you must write your whole Q-Learning approach inside of the corresponding methods (i.e., **train** and **act**). Note that you can initiate some parameters/variables in the constructor.

- **rl_agents/SARSA.py**: "SARSAAgent" class in this file contains the implementation of the SARSA algorithm for this assignment. Also, it is the sub-class of the "RLAgent" class. This means that you must write your whole SARSA approach inside of the corresponding methods (i.e., **train** and **act**). Note that you can initiate some parameters/variables in the constructor.

- **Main.py**: This file has a script to run the algorithms and print some results, such as the total scores, elapsed time in terms of microseconds, etc. You can edit this file if you want to add more analyzes or more tree-search algorithms. Although we will test your code with the original script, you can share your own "Main.py" file with us. To run this script, you should enter the name of the grid file, such as "Grid1.pkl" in the console.

- **grid_generator.py**: This file contains a script to generate a new scenario randomly. We provided this file so that you can test your algorithms with more scenarios. When you run this script, it will ask for some parameters in the console.

Briefly, you will implement "Q-Learning" and "SARSA" RL algorithms with Python programming language in the corresponding methods. Do not forget to read the comments in all methods.

3 Reinforcement Learning Algorithms

This section provides essential background information on Reinforcement Learning (RL) and the algorithms that you will be expected to implement for the given problem. RL is a machine learning category where an agent learns to make decisions in an environment by interacting with it. The agent observes the current state and selects an action, then the environment responds with feedback called "reward" (as shown in Equation 1). The primary objective of RL is to maximize the cumulative reward (as illustrated in Equation 2).

$$s_{t+1}, r_t \leftarrow \mathrm{Env}(s_t, a_t) \tag{1}$$

$$\mathbf{maximize}(\sum_{t}^{T} r_t) \tag{2}$$

Reinforcement learning enables an agent to learn how to identify the best actions to take in different states, leading to more successful outcomes in the environment. The Q-value is a critical concept in RL, which indicates how good a particular state-action pair is. By using a Dynamic Programming approach, the Q-value function ($Q(s, a)$) can be defined as shown in Equation 3, where γ is the discount rate.

$$Q(s, a) = r + \gamma \times \mathbf{max}_{a'}(Q(s', a')) \tag{3}$$

$$\pi(s) = \mathbf{argmax}_a(Q(s, a)) \tag{4}$$

The Q-value function represents the expected cumulative discounted reward that the agent can receive by taking a specific action in a particular state and then following its policy thereafter. Note that the policy function can be defined as displayed in Equation 4. The Q-Value function is learned through the Q-learning process, where the Q-value estimates are updated based on the rewards received by the agent in response to its actions. By continually updating the Q-value estimates, the agent can make more informed decisions about which actions to take in different states, ultimately leading to more successful outcomes in the environment.

For this assignment, you are expected to implement Q-Learning and SARSA algorithms for the given problem. Q-learning and SARSA are both algorithms used in RL to find an optimal policy for an agent to take action in an environment. Both of these algorithms use the concept of Q-values to estimate the value of a state-action pair. Q-learning (Algorithm 1) is an off-policy algorithm that learns the optimal Q-value function by updating the Q-value of the current state-action pair using the maximum Q-value of the next state. The agent selects actions based on the highest Q-value for the current state, which may not necessarily be the action taken by the agent. This makes Q-learning an off-policy algorithm, as the agent updates its Q-values based on the optimal action, not the action that it took.

SARSA (Algorithm 2), on the other hand, is an on-policy algorithm that learns the Q-value function by updating the Q-value of the current state-action pair using the Q-value of the next state-action pair according to the policy the

Algorithm 1 Q-Learning (Off-Policy)

```
1:  procedure Q-LEARNING(Env, $\gamma$, $max_{iter}$, $\alpha$, $\epsilon_{min}$, $\epsilon_{decay}$)
2:      Initiate $\hat{Q}[s, a]$                                        ▷ Zero-matrix
3:      $iter \leftarrow 0$
4:      $\epsilon \leftarrow 1$
5:      while $iter \leq max_{iter}$ do
6:          $s \leftarrow$ Initial state
7:          repeat
8:              $a \leftarrow \epsilon$-greedy for $s$
9:              $s', reward \leftarrow$ Env$(s, a)$
10:             $TD = reward + \gamma \times max_{a'}(\hat{Q}(s', a')) - \hat{Q}(s, a)$
11:             $\hat{Q}(s, a) = \hat{Q}(s, a) + \alpha \times TD$                  ▷ Soft-update
12:             $s \leftarrow s'$
13:             if $\epsilon > \epsilon_{min}$ then                            ▷ Update $\epsilon$
14:                 $\epsilon = \epsilon \times \epsilon_{decay}$
15:             end if
16:             $iter \leftarrow iter + 1$
17:         until Episode is done
18:     end while
19: end procedure
```

agent is following. The agent selects actions based on the policy it is following, which means that the Q-values are updated based on the action the agent actually takes.

The main difference between Q-learning and SARSA is that Q-learning is an off-policy algorithm, whereas SARSA is an on-policy algorithm. This difference affects how the algorithms update their Q-values and choose actions. In general, Q-learning is better suited for environments with high variance, where exploring and maximizing rewards are equally important. On the other hand, SARSA is better suited for environments with low variance, where the agent should focus more on maximizing rewards.

$$a = \begin{cases} \pi(s) & \text{with probability } 1 - \epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases} \tag{5}$$

On the other hand, in both Q-learning and SARSA, the agent's behavior policy is typically an epsilon-greedy approach during training. This means that the agent selects the best action with probability $1 - \epsilon$ (exploitation) and a random action with probability ϵ (exploration), as defined in Equation 5. The epsilon-greedy approach is important because it allows the agent to explore different actions and learn from them. Without exploration, the agent may not encounter all possible states and actions and could miss out on learning optimal policies. However, exploration also comes at a cost since the agent may choose sub-optimal actions that lead to lower rewards. By balancing exploration

Algorithm 2 SARSA (On-Policy)

```
1: procedure SARSA(Env, $\gamma$, $max_{iter}$, $\alpha$, $\epsilon_{min}$, $\epsilon_{decay}$)
2:     Initiate $\hat{Q}[s, a]$                                  ▷ Zero-matrix
3:     $iter \leftarrow 0$
4:     $\epsilon \leftarrow 1$
5:     while $iter \leq max_{iter}$ do
6:         $s \leftarrow$ Initial state
7:         $a \leftarrow \epsilon$-greedy for $s$
8:         repeat
9:             $s', reward \leftarrow$ Env$(s, a)$
10:            $a' \leftarrow \epsilon$-greedy for $s'$
11:            $TD = reward + \gamma \times \hat{Q}(s', a') - \hat{Q}(s, a)$
12:            $\hat{Q}(s, a) = \hat{Q}(s, a) + \alpha \times TD$       ▷ Soft-update
13:            $s \leftarrow s'$
14:            $a \leftarrow a'$
15:            if $\epsilon > \epsilon_{min}$ then                      ▷ Update $\epsilon$
16:                $\epsilon = \epsilon \times \epsilon_{decay}$
17:            end if
18:            $iter \leftarrow iter + 1$
19:        until Episode is done
20:    end while
21: end procedure
```

and exploitation through the epsilon-greedy approach, the agent can learn an optimal policy while still exploring the environment. The value of ϵ is typically decreased over time as the agent learns more about the environment. In the beginning, when the agent has little knowledge of the environment, it is essential to explore more to learn optimal policies. As the agent's knowledge grows, the exploitation term becomes more important, and the exploration term can be decreased. This approach is known as annealing epsilon-greedy, where the value of epsilon is annealed or gradually decreased over time.

For this assignment, Q-Learning and SARSA algorithms must be implemented; then, their performance must be compared in terms of computational time and total reward during the validation phase. Additionally, their performance depends on some hyper-parameters (e.g., maximum iteration, γ, ϵ_{min}, ϵ_{decay}, and *alpha*). Thus, you are expected to tune these parameters and analyze the impact on the performance of these methods in your report. Some example graphs that you can prepare for your report are displayed in Section 5.

4 Requirements

The requirements of this homework are listed below:

1. You have to implement Q-Learning and SARSA algorithms to find the

maximum path to reach a Goal" state in a given scenario.

2. Your algorithms must return the list of actions, the total score, and the list of expansions. They should find the optimal solution, the maximum total score from the starting point to a "Goal" state.
3. You must fill in the corresponding methods (i.e., **train** and **act**) in "QLearning.py" and "SARSA.py". Please, do not change the other files (except "Main.py").
4. You must use Python programming language with 3.8 or 3.9 versions. Jupyter is not allowed. You are free to choose any IDE for implementation, but PyCharm is recommended. You can create a student account with your "@ozu.edu.tr" e-mail address for the educational (free) license.
5. No library except *NumPy*, *Matplotlib*, and visualization libraries (e.g., Seaborn & plotly, etc.) is allowed. Using other libraries will be penalized.
6. You must also write a detailed and well-organized report. Also, your report must be clear and in English.
7. The report must cover the implementation details and design and the comparison of the algorithms.
8. You must compare the performance of the algorithms in terms of the computational time, the total score, the output of the algorithms, etc. You are free to extend these analyzes. It would be best to put some graphs and tables in your report for better evaluation, as in Section 5.
9. You must also analyze the effect of the hyper-parameters (e.g., maximum iteration, γ, ϵ_{min}, ϵ_{decay} and α) on the performance of the algorithms.
10. You can also implement and analyze different stopping conditions in the literature.
11. We provide five different scenarios for you. Please, do not forget to test/evaluate your code with these scenarios. With "grid_generator.py", you can create more various scenarios for your evaluation. We have more scenarios we did not share to test your algorithms.
12. You can also implement other RL algorithms for better performance evaluation, such as Double Q-Learning, Double SARSA, Value Iteration, etc. If you implement more RL algorithms, you can also compare them in your report.
13. An example implementation for A*and Uniform Cost Search is provided. You can compare your results with these approaches.
14. Do not put any screenshots of the code or console because we will already examine your code and run it to test.

15. You will submit your homework as *zip* format. Other formats, such as *rar*, will not be accepted. The *zip* file must contain your report as a PDF file and your implementation as **.py* file format. Do not put the whole project. Instead, you must put “QLearning.py” and “SARSA.py”. Also, you can put “Main.py” if you make any changes. If you implement more RL algorithms for your report, please do not forget to share them with us. Otherwise, we cannot evaluate them. Do not forget to write your name in your Python files.

16. File name format is **NAME_SURNAME_STUDENTID_hw1.zip**.

17. Any compiler or run-time error will be penalized **(-20pt)**.

18. Any plagiarism will also be penalized (e.g., Sharing code, copy code from the Internet, asking someone to do your homework, using any language model such as **ChatGPT**, etc.).

19. **Grading**:

    - Implementation **(40pt)**:
      - Implementation of Q-Learning **(20pt)**: Points will be awarded based on the correctness and efficiency of the implementation.
      - Implementation of SARSA **(20pt)**: Points will be awarded based on the correctness and efficiency of the implementation.
    - Report **(60pt)**:
      - Implementation details (10pt)
      - Analyzes the effect of hyper-parameters (10pt)
      - Performance evaluation and comparison of the algorithms (10pt)
      - Analyzes and visualization of trained Q-Tables (10pt)
      - Graphs and tables (10pt)
      - Overall (10pt)
    - Plagiarism Penalty: Any plagiarism will result in a direct deduction of **-100pt** from this assignment.

20. The assignment must be done individually. While you may discuss the homework with your peers, you are strictly prohibited from sharing any ideas or code. Failure to comply with this rule will result in a penalty for plagiarism.

21. The deadline for this homework is **5 June 2023**. Late submissions will not be allowed according to the course policy.

If you have any questions regarding the homework, we encourage you to seek help during our office hours. Our office hours are held online via Zoom to ensure accessibility for all students, and the corresponding meeting links can be found on LMS. Additionally, if you prefer to communicate via e-mail, you can contact Anil Dogru or Onur Keskin. Please keep in mind that we cannot provide any extra code or ideas before the deadline, as this would be considered a violation of academic integrity. However, we will do our best to assist you with any conceptual or technical questions.

5 Example Graphs

This section provides some example graphs to get some idea of what kind of graphs are expected in your report. Also, it would be best if you extended these graphs with more analysis and comparisons.

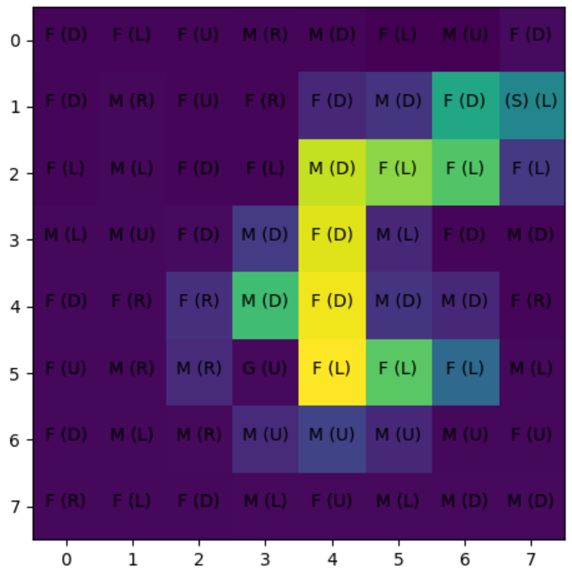


Figure 2: Max. Q-Value for each State with Optimal Action

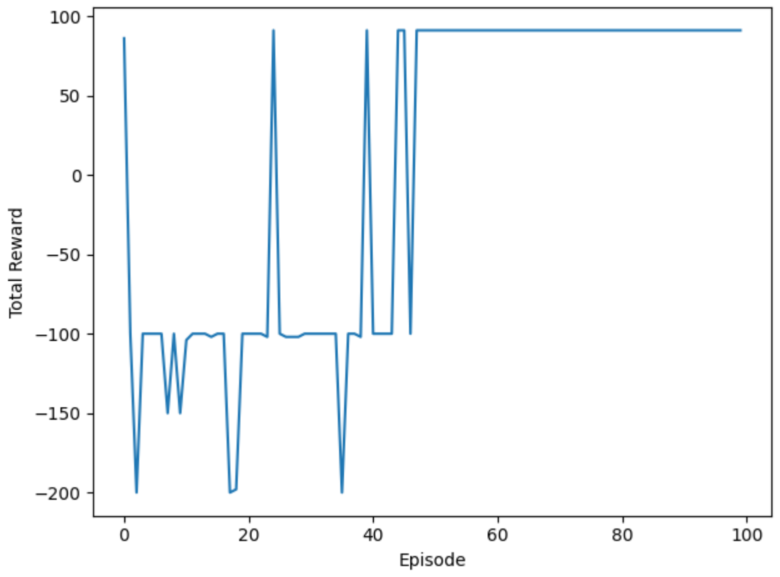


Figure 3: Total Reward Change over Episode

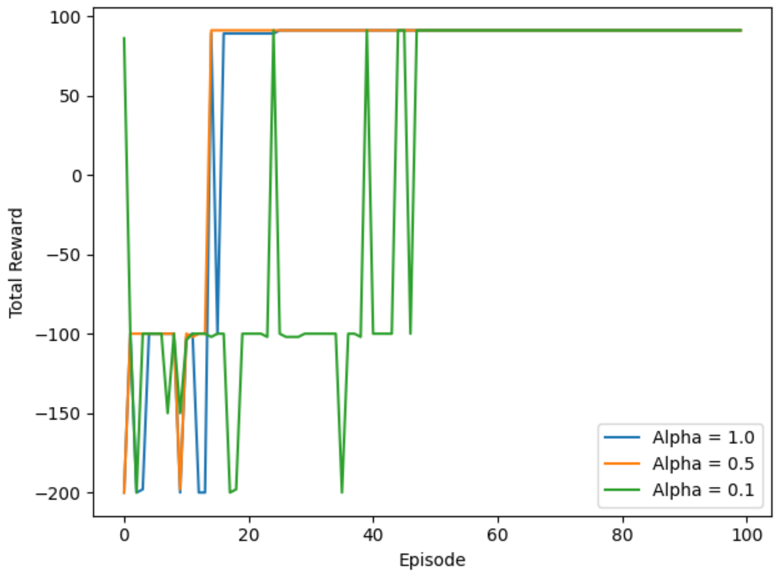


Figure 4: Effect of Alpha Hyper-Parameter on Total Reward

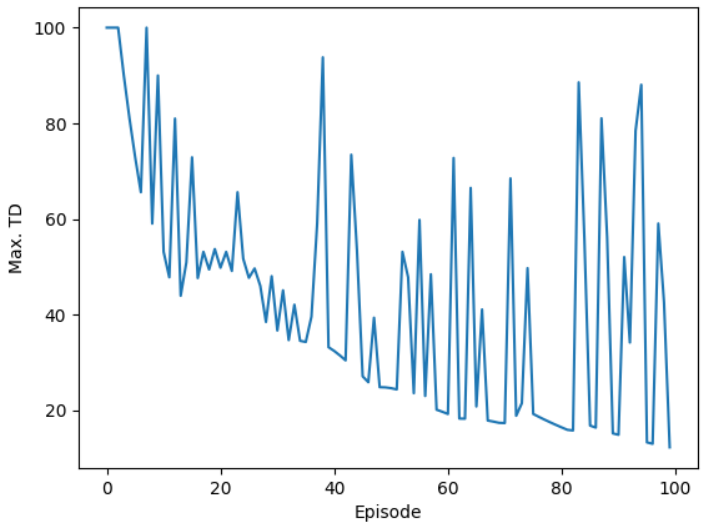


Figure 5: Max. TD on each Episode

6 Helpful Links for Python

You can learn the required Python programming with the helpful links below:

- Python Docs
- Tutorials Point
- Geeks for Geeks
- W3 Schools
- Programiz
- Learn Python